

VAX 9000 Family IBox Technical Description

Order Number EK-KA90I-TD-001

**digital equipment corporation
maynard, massachusetts**

DIGITAL INTERNAL USE ONLY

First Edition, May 1990

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

Restricted Rights: Use, duplication, or disclosure by the U. S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

Copyright © Digital Equipment Corporation 1990

All Rights Reserved.
Printed in U.S.A.

The postpaid Reader's Comment Card included in this document requests the user's critical evaluation to assist in preparing future documentation.

FCC NOTICE: The equipment described in this manual generates, uses, and may emit radio frequency energy. The equipment has been type tested and found to comply with the limits for a Class A computing device pursuant to Subpart J of Part 15 of FCC Rules, which are designed to provide reasonable protection against such radio frequency interference when operated in a commercial environment. Operation of this equipment in a residential area may cause interference, in which case the user at his own expense may be required to take measures to correct the interference.

The following are trademarks of Digital Equipment Corporation:

BI	KDM	RSTS	VAX FORTRAN
CI	KLESI	RSX	VAX MACRO
DEC	MASSBUS	RT	VAXBI
DECmate	MicroVAX	RV20	VAXcluster
DECUS	NI	RV64	VAXELN
DECwriter	PDP	TA	VMS
DHB32	P/OS	TK	VT
DIBOL	Professional	ULTRIX	Work Processor
DRB32	RA	UNIBUS	XMI
EDT	Rainbow	VAX	
KDB50	RD	VAX C	

® IBM is a registered trademark of International Business Machines Corporation.

® Intel is a registered trademark of Intel Corporation.

™ Hubbell is a trademark of Harvey Hubbel, Inc.

® Motorola is a registered trademark of Motorola, Inc.

This document was prepared and published by Educational Services Development and Publishing, Digital Equipment Corporation.

DIGITAL INTERNAL USE ONLY

Contents

About This Manual

xiii

1 General Description

1.1	IBox Introduction	1-1
1.2	Basic Hardware Implementation	1-1
1.2.1	Program Counter Unit	1-1
1.2.2	Virtual Instruction Cache	1-3
1.2.3	Instruction Buffer	1-3
1.2.4	Branch Prediction	1-3
1.2.4.1	Branch Prediction Cache	1-3
1.2.5	Multiple Specifier Decode Unit	1-3
1.2.6	Specifier Handlers	1-4
1.2.6.1	Complex Specifier Unit	1-4
1.2.6.2	Short Literal Unit	1-4
1.2.6.3	Free Pointer Logic	1-4
1.2.7	Read/Write Scoreboards	1-5
1.3	Physical Organization	1-5
1.3.1	VIC MCU	1-6
1.3.2	XBR MCU	1-7
1.3.3	OPU MCU	1-8
1.4	Pipeline Overview	1-9
1.5	IBox Interfaces	1-13
1.5.1	MBox Interface	1-13
1.5.1.1	Instruction Buffer Port	1-13
1.5.1.2	OPU Port	1-13
1.5.2	EBox Interface	1-14
1.5.2.1	IBox-to-EBox Interface	1-14
1.5.2.2	EBox-to-IBox Interface	1-14
1.5.3	VBox Interface	1-14
1.5.4	Service Processor Interface	1-14

2 Program Counter

2.1	Overview	2-1
2.2	Major PCs	2-1
2.2.1	Prefetch PC	2-1
2.2.2	Prefetch PC Data Path	2-3
2.2.2.1	Prefetch PC Control	2-4
2.2.3	Target PC	2-4
2.2.4	Decode PC	2-4
2.2.5	Decode PC Data Path	2-5
2.2.5.1	Delta PC	2-5
2.2.6	Branch PC	2-6
2.2.7	Branch PC Data Path	2-6
2.2.8	Unwind PC	2-7
2.2.9	Unwind PC Data Path	2-7
2.3	Cache Control	2-8
2.3.1	BPC Control	2-8
2.3.2	VIC Control	2-8
2.4	PCU Error Detection	2-9
2.4.1	PCVC Error Logic	2-9
2.4.2	PCBP Error Logic	2-11
2.4.3	PCLO Error Logic	2-11
2.4.4	PCHI Error Logic	2-14
2.5	PCU Inputs	2-14
2.6	PCU Outputs	2-16

3 Instruction Fetch

3.1	VIC	3-1
3.1.1	VIC Hit	3-2
3.1.2	VIC Data Write	3-2
3.1.2.1	VIC Address Selection	3-4
3.1.2.2	VIC Data	3-4
3.1.3	VIC Tag Write	3-5
3.1.3.1	Tag Write	3-6
3.1.3.2	Quadword Valid Write	3-6
3.1.3.3	Block Valid Write	3-6
3.1.3.4	Tag Parity	3-6
3.1.4	VIC Flush	3-7
3.1.5	VIC Data Read	3-7
3.1.5.1	VIC STRAM Bypass	3-8
3.1.6	VIC Parity Coverage	3-8
3.1.7	Disabling VIC Hits	3-8
3.2	Instruction Buffer	3-9
3.2.1	IBEX2	3-10

3.2.2	IBEX	3-11
3.2.2.1	IBEX Valid Count	3-11
3.2.3	Rotator	3-11
3.2.3.1	IBEX2 Rotate Data	3-13
3.2.4	Merger	3-14
3.2.5	Shifter	3-16
3.2.6	IBUF	3-18
3.2.6.1	Simple Decode	3-18
3.2.6.2	Branch Decode	3-19
3.2.6.3	YREG Decode	3-19
3.2.6.4	Short Literal Decode	3-19
3.2.6.5	Register Mode Decode	3-19
3.2.7	Instruction Buffer Parity	3-19
3.3	Instruction Buffer Interface	3-20
3.3.1	Instruction Buffer Requests	3-22
3.3.1.1	Aborting Requests	3-23
3.3.1.2	Page Faults	3-23

4 Instruction Decode

4.1	XBAR	4-1
4.1.1	DRAM	4-4
4.1.2	XRAM	4-5
4.1.3	Simple Decode Logic	4-6
4.1.4	Decode Tree Logic	4-7
4.1.5	Request Logic	4-9
4.1.6	Specifier Count Logic	4-12
4.1.7	Shift Count Logic	4-13
4.1.7.1	FD Shift Opcode	4-13
4.1.8	Fork Logic	4-14
4.1.9	XBAR Displacement Data Path	4-14
4.1.9.1	Displacement	4-14
4.1.9.2	Extended Immediate Mode (X8F) Detection	4-15
4.1.9.3	XREG	4-16
4.1.9.4	YREG	4-16
4.1.10	XBAR Short Literal Data Path	4-16
4.1.10.1	Short Literal Data Select	4-16
4.1.10.2	Short Literal Specifier Number	4-17
4.1.10.3	Short Literal Valid	4-17
4.1.11	XBAR Source and Destination Logic	4-18
4.1.12	XBAR Source 1 Data Path	4-19
4.1.13	XBAR Source 2 Data Path	4-20
4.1.14	XBAR Destination	4-21
4.1.14.1	Destination Valid	4-21
4.1.14.2	Destination Register Valid	4-21

4.1.15	Register Masks	4-22
4.1.15.1	Read Mask	4-22
4.1.15.2	Write Mask	4-23
4.1.15.3	Implied Mask	4-23
4.1.16	Intra-Instruction Read Conflicts	4-23
4.1.17	XBAR Stalls	4-25
4.2	Branch Prediction	4-27
4.2.1	Primary Predictions	4-27
4.2.1.1	Primary Hits	4-28
4.2.1.2	Tag Match Enable	4-29
4.2.2	Demote	4-29
4.2.3	Secondary Predictions	4-30
4.2.4	BPC Correction	4-30
4.2.5	BPC Unwind	4-31
4.3	PCU Microcode	4-31
4.3.1	PCU Microaddress	4-31
4.3.2	PCU Microword	4-34
4.3.3	Writing the BPC	4-40
4.3.3.1	BPC Write Enable	4-40
4.3.3.2	Cache Tag Write	4-41
4.3.3.3	Instruction Length Field Write	4-41
4.3.3.4	Prediction PC Write	4-42
4.3.3.5	BP Displacement Write	4-42
4.3.3.6	BP Prediction Bit	4-42
4.3.3.7	BPC Address Selection	4-45

5 Specifier Decode

5.1	Overview	5-1
5.1.1	Stall Logic	5-3
5.2	Complex Specifier Unit	5-4
5.2.1	OPUA Data Path	5-5
5.2.1.1	AMUX Inputs	5-7
5.2.1.2	BMUX Inputs	5-7
5.2.1.3	Adder	5-8
5.2.2	OPUB Data Path	5-8
5.2.3	Current PC Generation	5-10
5.2.3.1	OPUA Current PC [15:00]	5-11
5.2.3.2	OPUB Current PC [31:16]	5-11
5.2.4	CSU Microcode	5-12
5.2.4.1	CSU Microaddress	5-12
5.2.4.2	CSU Microword	5-13

5.2.5	CSU Stalls	5-15
5.2.5.1	Scoreboard Stalls	5-15
5.2.5.2	Branch Under Branch Stall	5-18
5.2.5.3	AUTOxx Under Branch Stall	5-18
5.2.5.4	OPU Port Grant Wait Stall	5-19
5.3	Short Literal Unit	5-19
5.3.1	Short Literal Processing	5-19
5.3.2	Integer Expansion	5-21
5.3.3	Floating-Point Expansion	5-22
5.3.4	Outputs to the EBox Interface	5-24
5.3.4.1	Order	5-24
5.3.5	Stalls	5-24
5.3.5.1	Source List Full	5-24
5.3.5.2	SLU Stalled	5-24
5.3.5.3	EBox Interface Output Stall	5-24
5.3.6	Parity Coverage and Errors	5-24
5.4	Free Pointer Logic	5-25
5.4.1	Source 1 Pointer	5-25
5.4.2	Source 2 Pointer	5-27
5.4.3	Free Pointer	5-29
5.4.3.1	Free Pointer Initialization	5-30
5.4.4	Destination Pointer	5-30
5.4.4.1	Destination Register	5-30
5.4.4.2	Destination Valid	5-30
5.4.4.3	Destination Memory	5-30
5.5	Operand Control Unit	5-32
5.5.1	Read/Write Masks	5-32
5.5.1.1	Mask Valid	5-33
5.5.1.2	Instruction Done	5-33
5.5.1.3	Correction	5-34
5.5.2	Read/Write Mask Parity	5-35
5.5.3	Flushes	5-35
5.5.4	OCTL Stalls	5-37
5.5.4.1	Scoreboard Stall	5-37
5.5.4.2	Mask Stall	5-37
5.6	OPU Port Interface	5-38
5.7	IBox-to-EBox Interface	5-40
5.8	EBox-to-IBox Interface	5-44
5.9	VBox Interface	5-47

6 IBox Error Descriptions

6.1	IBox Error Registers	6-1
6.2	Fetch Error Register 1	6-1
6.3	Fetch Error Register 2	6-4
6.4	Decode Error Register 1	6-5
6.5	XBAR Decode Error Register	6-6
6.6	Specifier Error Register 1	6-8
6.7	Specifier Error Register 2	6-10

A IBox Input and Output Listing

Index

Figures

1-1	Basic IBox Block Diagram	1-2
1-2	Planar Module Layout	1-5
1-3	VIC MCU Content	1-6
1-4	XBR MCU Content	1-7
1-5	OPU MCU Content	1-8
1-6	IBox Pipeline: No Stalls	1-10
1-7	IBox Pipeline: CSU Stall	1-11
1-8	IBox Pipeline: Branch Taken, Secondary Prediction	1-12
1-9	IBox Pipeline: BP Cache Hit, Prediction Taken	1-12
2-1	PCU Data Path	2-2
2-2	Prefetch PC Data Path	2-3
2-3	Target PC Sources	2-4
2-4	Decode PC Data Path	2-5
2-5	Branch PC Data Path	2-6
2-6	Unwind PC Data Path	2-7
2-7	PCU Cache Control	2-8
2-8	PCVC Error Logic	2-10
2-9	PCBP Error Logic	2-12
2-10	PCLO Error Logic	2-13
2-11	PCHI Error Logic	2-15
3-1	VIC	3-1
3-2	VIC Data Write	3-3
3-3	VIC Address Selection	3-4
3-4	VIC Tag Write	3-5
3-5	VIC Match Logic	3-7
3-6	Instruction Buffer	3-9
3-7	Rotator	3-12
3-8	IBEX2 Rotate Data	3-13
3-9	Simplified Merger	3-14

3-10	Merger	3-15
3-11	IBUF Data	3-17
3-12	Simple Decode	3-18
3-13	Instruction Buffer Interface	3-20
3-14	Instruction Buffer Request	3-22
4-1	XBAR Block Diagram	4-2
4-2	DRAM Logic	4-4
4-3	Simple Decode Logic	4-6
4-4	XBAR Decode Trees	4-7
4-5	R2BW Decode Tree	4-8
4-6	Request Logic	4-10
4-7	Specifier Count Logic	4-12
4-8	Shift Counts	4-13
4-9	XBAR Displacement	4-15
4-10	Short Literal Logic	4-17
4-11	Source and Destination Logic	4-18
4-12	XBAR Source 1 Logic	4-19
4-13	XBAR Source 2 Logic	4-20
4-14	XBAR Destination	4-21
4-15	XBAR Read and Write Masks	4-22
4-16	IRC Detection: Read Mask	4-24
4-17	IRC Detection: IRC Mask	4-25
4-18	BPC Organization	4-27
4-19	BPC Compare: Hit	4-28
4-20	BPC Compare: Demote	4-28
4-21	BP Tag Match Enable	4-29
4-22	Branch Bias Logic	4-30
4-23	PCU Microword Format	4-34
4-24	BPC Write Enable	4-40
4-25	BPC Tag Write	4-41
4-26	BP Instruction Length	4-41
4-27	BP Prediction PC Write	4-42
4-28	BP Displacement Write	4-42
4-29	BP Prediction Bit Write	4-44
4-30	BPC Address Selection	4-45
5-1	OPU Stall Logic	5-3
5-2	CSU Organization	5-4
5-3	OPUA	5-6
5-4	OPUB	5-9
5-5	CSU Current PC (High Slice)	5-10
5-6	CSU Current PC (Low Slice)	5-10
5-7	CSU Microaddress Format	5-12
5-8	CSU Microword Format	5-13
5-9	CSU Microword Select	5-16
5-10	Input and Outputs of the Short Literal Unit	5-19
5-11	Short Literal Unit Block Diagram	5-20

5-12	SLU Integer Expansion	5-21
5-13	SLU Floating Point Literal Format	5-22
5-14	SLU F-Floating Expansion	5-22
5-15	SLU D-Floating Expansion	5-22
5-16	SLU G-Floating Expansion	5-23
5-17	SLU H-Floating Expansion	5-23
5-18	Free Pointer Logic	5-26
5-19	FPL Source 1 Pointer Logic	5-27
5-20	FPL Source 2 Pointer Logic	5-28
5-21	Free Pointer	5-29
5-22	Destination Pointer Logic	5-31
5-23	OCTL Unit	5-32
5-24	OCTL Read/Write Masks	5-33
5-25	OCTL Mask Correction Logic	5-34
5-26	OCTL Flush Logic	5-36
5-27	OCTL Mask Stall Logic	5-37
5-28	OPU Port Interface	5-38
5-29	IBox-to-EBox Interface	5-41
5-30	EBox-to-IBox Interface	5-45
5-31	VBox Interface	5-47
6-1	Fetch Error Register 1	6-2
6-2	Fetch Error Register 2	6-4
6-3	Decode Error Register 1	6-5
6-4	XBAR Decode Error Register	6-6
6-5	Specifier Error Register 1	6-8
6-6	Specifier Error Register 2	6-10

Tables

2-1	PCVC Errors	2-9
2-2	PCBP Errors	2-11
2-3	PCLO Errors	2-11
2-4	PCHI Errors	2-14
3-1	Sample Merge Select	3-15
3-2	Instruction Buffer Interface Signals	3-21
4-1	Case Output	4-6
4-2	XBAR Decode Trees	4-9
4-3	PCU Microaddress Descriptions	4-32
4-4	PCU Microword Field Descriptions	4-35
5-1	CSU Microaddress Descriptions	5-12
5-2	CSU Microword Field Descriptions	5-14
5-3	Modulo 6 Subtraction Logic	5-34
5-4	Register Valid Fields	5-35
5-5	EBox Flush Codes	5-36
5-6	OPU Port Interface Signals	5-39
5-7	IBox-to-EBox Interface Signals	5-42
5-8	EBox-to-IBox Interface Signals	5-46

5-9	VBox Interface Signals	5-47
6-1	Fetch Error Register 1	6-2
6-2	Fetch Error Register 2	6-4
6-3	Decode Error Register 1	6-5
6-4	XBAR Decode Error Register	6-7
6-5	Specifier Error Register 1	6-9
6-6	Specifier Error Register 2	6-11
A-1	IBox-VIC Signals	A-1
A-2	IBox-XBR Signals	A-2
A-3	IBox-OPU Signals	A-4

About This Manual

This manual describes the functions of the IBox in the VAX 9000 family system. It is a reference manual for Customer Services personnel as well as a training resource for Educational Services.

Intended Audience

The content, scope, and level of detail in this manual assumes that the reader:

- Is familiar with the VAX architecture and VMS operating system at the user level
- Has experience maintaining midrange and large VAX systems

Manual Structure

This manual has six chapters, an appendix, a glossary of IBox terms, and an index.

Chapter 1 contains an introduction to the IBox, describing the major features, functions, and physical organization of the unit. Chapter 2 describes the program counter. Chapters 3, 4, and 5 provide a detailed description of the functions of the IBox, with each chapter emphasizing one of the three main pipeline stages of the unit. Chapter 6 is a summary of the errors that can be generated in the IBox.

Appendix A provides a listing of the input and output signals of the three multichip units (MCUs) that comprise the IBox.

General Description

This chapter provides an overview of the VAX 9000 family IBox, including descriptions of the major hardware features, physical characteristics, and interbox interfaces.

1.1 IBox Introduction

The IBox is an independent functional unit that fetches and decodes instructions and their specifiers from the MBox and passes them to the EBox for execution. The EBox is provided with data and control functions so that it can execute several instructions in one cycle.

The IBox provides all required instruction data to the EBox: source, destination, PC, fork address, as well as pointers to the source data and destination. The instruction data is stored in the EBox source list or general-purpose registers (GPRs). In the case of a memory source operand, the IBox generates the operand address and passes it to the MBox requesting a memory read operation. The MBox prefetches the data and then writes it into the EBox source queue.

Using the source pointers provided by the IBox, the EBox accesses its source list or GPRs and executes the instruction. Depending on the destination, the EBox writes the results to a GPR or transfers the results to the MBox to be subsequently written to memory. In effect, the EBox deals only with data (no opcodes or operand specifiers).

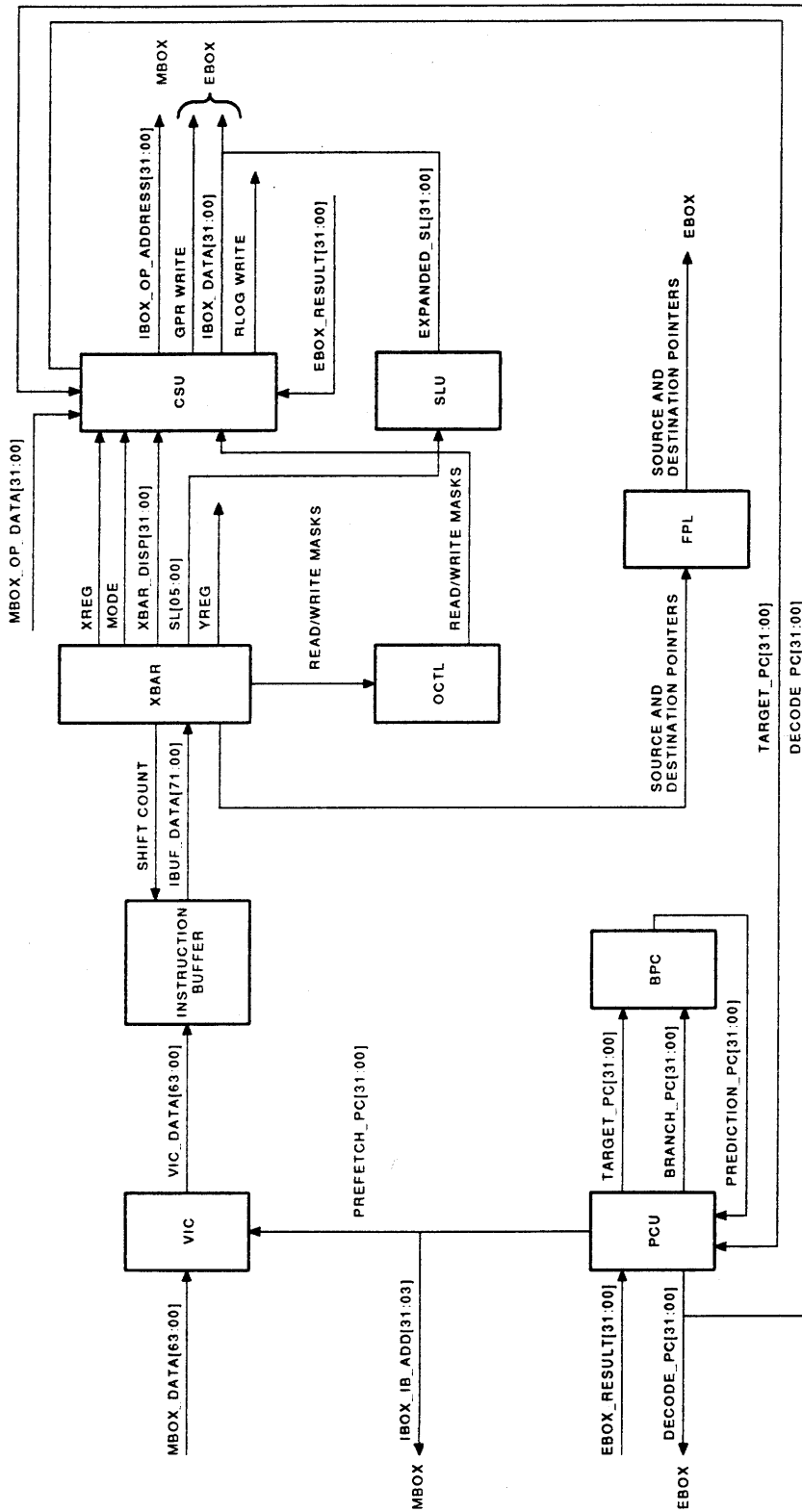
In addition, the IBox can decode and store several instructions ahead of the EBox. However, given the capabilities of the EBox and MBox, it is difficult for the IBox to remain several instructions ahead.

1.2 Basic Hardware Implementation

This section describes the major new hardware implementations incorporated into the IBox. Figure 1-1 shows the basic IBox block diagram.

1.2.1 Program Counter Unit

The program counter unit (PCU) is responsible for directing the I-stream that the IBox will decode, and for generating the PC address used by the EBox and MBox. The PCU provides the control logic for the two IBox caches: virtual instruction cache (VIC) and the branch prediction cache (BPC). In addition, the PCU controls the secondary branch prediction mechanism.



MIL_X0039_58

Figure 1-1 Basic IBox Block Diagram

1.2.2 Virtual Instruction Cache

The virtual instruction cache (VIC) is a virtually addressed, 8 Kbyte, direct-mapped, one-way associative cache that reduces the number of I-stream requests issued to the MBox. By flushing the VIC on every REI, the IBox can ignore writes to memory. Having a virtually addressed instruction cache eliminates the need for address translation and translation logic. Because the VIC has its own MBox request port, it is refilled from the MBox data cache, rather than memory.

1.2.3 Instruction Buffer

The IBox incorporates a 25-byte instruction buffer that latches the I-stream. The instruction buffer is partitioned into a 9-byte instruction buffer (IBUF), an 8-byte extended instruction buffer (IBEX), and an additional 8-byte extended instruction buffer (IBEX2).

The content of the nine IBUF bytes are latched, decoded, and shifted. The remaining 16 bytes of IBEX and IBEX2 contain additional prefetched I-stream from the VIC, which is passed to the IBUF as required.

1.2.4 Branch Prediction

When branch instructions are encountered in the I-stream, the IBox predicts the direction the I-stream will follow (redirects the I-stream by taking a branch or continues decoding sequential I-stream by not taking the branch). When initially encountering a branch, the IBox decides to take or not take the branch by accessing the logic that contains fixed predictions for each branch instruction. Branch predictions are validated by the EBox and the correct predictions are stored in a branch prediction cache that is accessed when the same branch is subsequently encountered.

1.2.4.1 Branch Prediction Cache

To minimize the idle time spent flushing and refilling the pipeline after every branch instruction, the instruction decode stage includes a branch prediction cache (BPC). This 1K virtual cache increases performance by storing information about the branch validity, and the target address. As a branch instruction is being decoded, it is referenced, in parallel, in the BPC and a prediction is made whether to take the instruction. The IBox uses the cached target address to redirect the instruction fetch stage to the new I-stream if the branch is taken. For performance reasons, the BPC is never flushed.

1.2.5 Multiple Specifier Decode Unit

The multiple specifier decode unit (XBAR, or “crossbar”) is implemented as a set of multiplexers that provide the capability of simultaneously decoding up to three operand specifiers. The I-stream is presented to the XBAR nine bytes at a time from the IBUF.

The actual number of specifiers decoded depends on the specifier type. The XBAR can decode up to three specifiers (for example, two simple specifiers and one complex specifier, or three simple specifiers). Simple specifiers are considered register mode or short literal, while all other specifiers are considered complex.

1.2.6 Specifier Handlers

The specifier handlers are in the operand processing unit (OPU). The logic is comprised of three dedicated specifier evaluation units:

- Complex specifier unit
- Short literal unit
- Free pointer logic

In addition, the OPU maintains a set of GPRs. The GPRs are implemented in self-timed register (STREG) files that provide multiported, read/write access capabilities.

The OPU maintains a virtually addressed MBox request port.

1.2.6.1 Complex Specifier Unit

The complex specifier unit (CSU) is responsible for evaluating complex specifiers. The XBAR supplies the CSU with the register, register index, and up to 32 bits of displacement. The CSU calculates branch target addresses, immediate operands, and memory addresses of operands supplied to the EBox from the MBox.

The CSU contains the OPU port interface to the MBox and the interface to the EBox. Operand addresses are sent to the MBox, while the EBox receives the immediate operands directly from the CSU.

The CSU also contains the IBox GPRs. The GPRs are read and written by the CSU, and written by the EBox.

1.2.6.2 Short Literal Unit

The short literal unit (SLU) receives decoded short literal specifiers from the XBAR and expands them for entry into the EBox source list. The SLU can produce a single longword of expansion each cycle. The literal expansion depends on the specifier data type.

1.2.6.3 Free Pointer Logic

The free pointer logic (FPL) manages pointers into the EBox source list. The FPL tracks the available (free) source list addresses and associated pointers. The FPL establishes the correct source 1 and source 2 pointers for operands the EBox will use to execute an instruction. The FPL also generates the correct destination pointer for an instruction result.

1.2.7 Read/Write Scoreboards

The IBox processes specifiers while the EBox is executing previously decoded instructions. The IBox must be prevented from performing an address calculation that depends on the result of a currently executing instruction. This is accomplished by recording the register numbers to be written by the EBox, and by matching those numbers against any register selected for use in an operand specifier. When a match occurs, the CSU stalls and waits for the instruction to be completed before calculating the operand address.

The XBAR generates the read and write masks for conflict checking in the OPU. The masks represent GPR 0 through 14. The masks prevent reading or writing a GPR that is scheduled to be modified or read by the EBox.

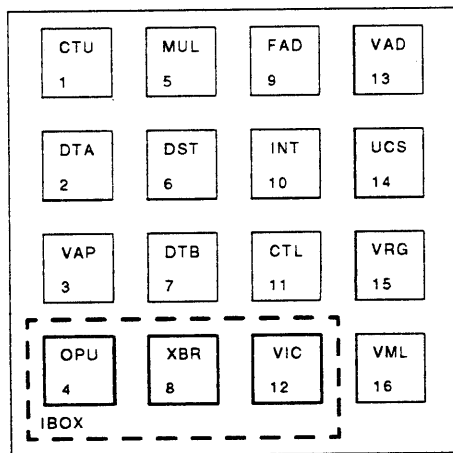
The CSU maintains the read and write register scoreboards for up to six instructions in the pipeline. Both scoreboards contain 15-bit registers, representing GPR 0 through 14.

The read scoreboard tracks the GPRs designated to be read by a previously decoded instruction. These GPRs cannot be written by the OPU for autoincrement and autodecrement until their corresponding instructions are completed by the EBox.

The write scoreboard tracks GPRs designated as destinations by instructions in the pipeline. These GPRs cannot be used by the OPU for address calculations until the instruction is completed by the EBox.

1.3 Physical Organization

The IBox logic is physically contained in three multichip units (MCUs) (Figure 1-2). This section introduces each MCU and its related macrocell arrays (MCAs).



MR_X0040_89

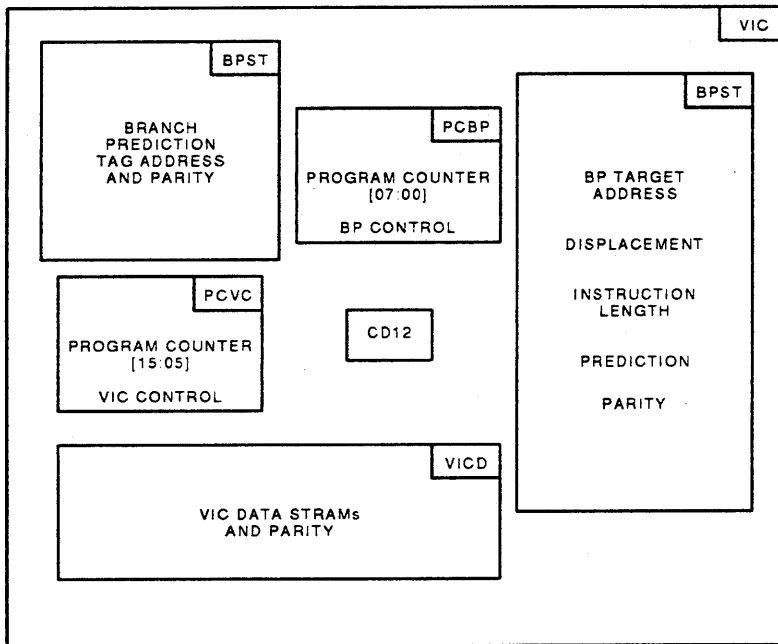
Figure 1-2 Planar Module Layout

1.3.1 VIC MCU

The VIC MCU is comprised of two MCAs and forty-two $1K \times 4$ -bit STRAMs. Two bit-slices of the program counter and the data STRAMs for branch prediction and VIC are resident in this MCU. Figure 1-3 shows the MCU content.

The following list introduces the MCA and STRAM functions of the VIC MCU:

- **PCBP MCA** — The program counter/branch prediction control MCA contains bits [07:00] of the PC and provides branch prediction control.
- **PCVC MCA** — The program counter/VIC control MCA contains bits [15:05] of the PC and provides VIC control.
- **VIC data STRAMs** — These eighteen $1K \times 4$ -bit STRAMs are dedicated to the VIC data and associated byte parity.
- **Branch prediction STRAMs** — These twenty-four $1K \times 4$ -bit STRAMs are dedicated to the branch prediction function. The STRAMs store the branch PC tag, prediction PC, branch instruction length, and prediction bit.



MR_X0041_89

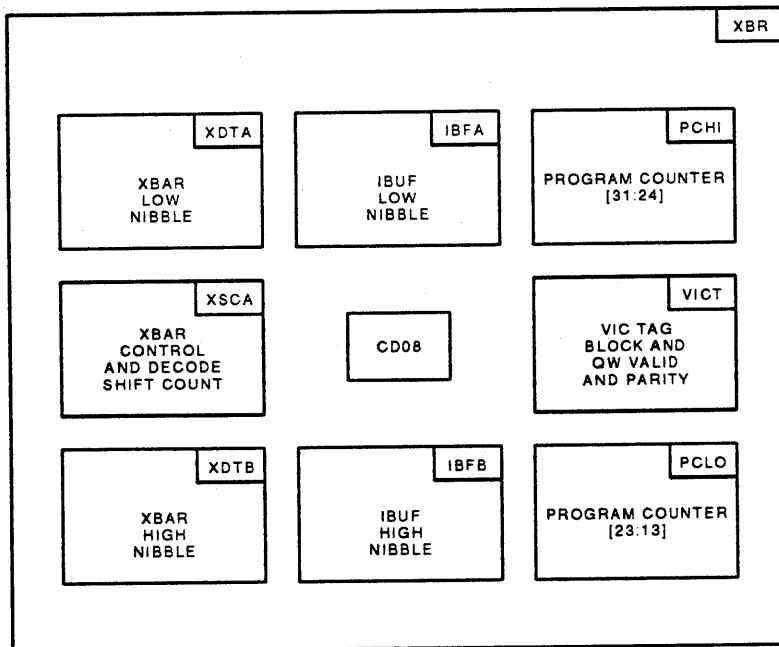
Figure 1-3 VIC MCU Content

1.3.2 XBR MCU

The XBR MCU contains the instruction buffer, XBAR, two of the four PC slices, and tag STRAMs for the VIC. Figure 1-4 shows the MCU content.

The following list introduces the MCA and STRAM functions of the XBR MCU:

- **PCLO MCA** — The program counter low MCA contains bits [23:13] of the PC.
- **PCHI MCA** — The program counter high MCA contains bits [31:24] of the PC.
- **IBFA MCA** — The instruction buffer A MCA contains the low-order nibble of the instruction buffer. IBFA also provides the VIC hit logic.
- **IBFB MCA** — The instruction buffer B MCA contains the high-order nibble of the instruction buffer. Parity checking is performed by combining IBFB parity with a partial parity from IBFA.
- **XDTA MCA** — The XBAR data A MCA contains the low-order nibble of the XBAR. The major MCA outputs are displacements for the OPU.
- **XDTB MCA** — The XBAR data B MCA contains the high-order nibble of the XBAR data path.
- **XSCA MCA** — The XBAR control MCA is the XBAR control unit. It receives I-stream data from the IBUF and performs some simple instruction decoding. The instruction buffer shift control is generated from the number of specifiers decoded and the number of specifiers the instruction contains.
- **VICT STRAMs** — Five of the nine $1K \times 4$ -bit STRAMs contain the 19-bit VIC tags. Two of the STRAMs provide the 4-bit VIC quadword valid fields and associated parity bits and parity for the VIC quadword valid bits. The remaining two provide the VIC block valid field.



MR_X0042_89

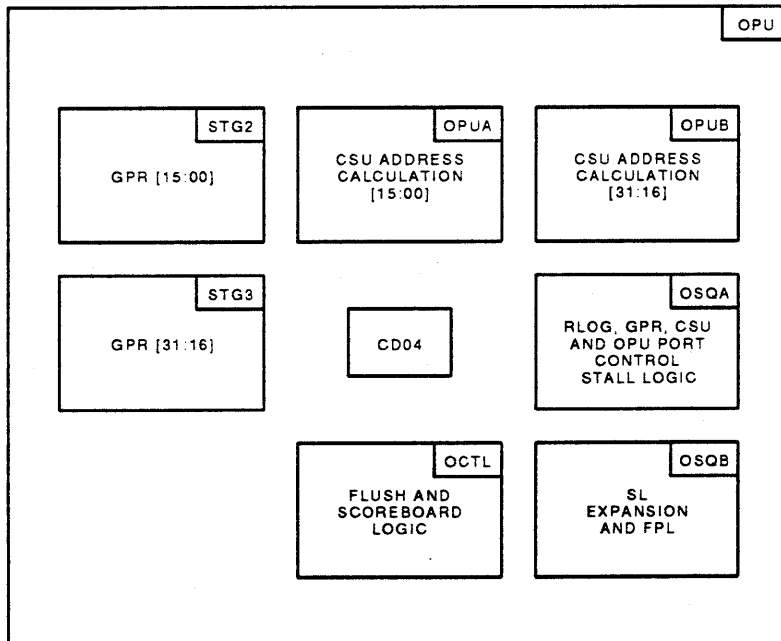
Figure 1-4 XBR MCU Content

1.3.3 OPU MCU

The OPU MCU contains the logic responsible for the specifier decode process. The operand port interface to the MBox also resides in this MCU. The MCU also contains a pair of self-timed registers (STREGs) that provide the IBox GPRs. Figure 1-5 shows the MCU content.

The following list introduces the MCA and STRAM functions of the OPU MCU:

- **OPU_x MCA** — The OPUA and OPUB MCAs provide the data path for the complex specifier unit (CSU). (OPUA provides the low-order word; OPUB provides the high-order word.) The CSU receives up to 32 bits of displacement from the XBAR, operand data from the MBox, and result data from the EBox and directs the outputs to the MBox, EBox, or a loopback to the CSU.
- **OSQA MCA** — This MCA provides control for the GPR STREGs. The OPUA and OPUB multiplexers (AMUX, BMUX) are also provided by this MCA. This MCA also controls the EBox and OPU port interfaces and stall logic.
- **OSQB MCA** — This MCA receives short literal, source, and destination data from the XBAR. Short literal specifiers are expanded into the correct context and passed to the EBox. The source and destination pointers are also passed to the EBox.
- **OCTL MCA** — The operand control MCA maintains the read/write scoreboard and directs flush signals to other appropriate IBox functional units.



MR_X0043_89

Figure 1-5 OPU MCU Content

1.4 Pipeline Overview

The IBox consists of three main pipeline stages that closely relate to the MCU physical structure. Each stage can generate indicators to aid in isolating errors to an MCU FRU. The three pipeline stages are defined as follows:

- **Instruction fetch** — This stage consists of the logic involved in fetching the I-stream before it is latched in the instruction buffer and includes these components:
 - VIC
 - Program counter logic (PCU)
 - IBEX, IBEX2, and part of the IBUF
 - IBUF-to-MBox interface
- **Instruction decode and branch prediction** — This stage consists of the logic involved in decoding the instruction in the IBUF and includes these components:
 - XBAR
 - Branch prediction unit (BPU)
- **Specifier evaluation** — This stage consists of the OPU logic involved in the evaluation of operand specifiers and includes these components:
 - Complex specifier unit (CSU)
 - Branch prediction logic
 - Free pointer logic (FPL)
 - Short literal unit (SLU)
 - Operand control unit (OCTL)
 - OPU-to-MBox interface
 - OPU-to-EBox interface

Because the IBox is pipelined, when a stage cannot complete its operation, previous stages must be stalled. That is, previous stage operations may be suspended.

Figure 1–6 through 1–9 provide examples of basic pipeline flow.

Figure 1-6 shows the IBox pipeline decoding sequential instructions where no stalls are encountered in any of the pipeline stages. The following events occur in each of the machine cycles of this example:

- During the first machine cycle (T0), the IBUF is loaded with the I-stream to be presented to the decode units of the IBox.
- During the second cycle (T1), the logic representing the decode pipeline stage decodes the opcode and the specifier bytes in the IBUF, and passes them to the specifier handlers. The logic representing the fetch stage replenishes the decoded bytes of the IBUF.
- During the third cycle (T2) of this example, the decoded specifiers are processed by the specifier handlers and passed to the EBox. The decode logic and the fetch logic continue to perform their operations simultaneously.

The parallel operations of the three pipeline stages continue until one of the units stalls or until the IBox is flushed to a new I-stream. A stall in one of the units is caused by instruction specifiers that require more than a single cycle to be processed or by conflicts in the instructions (for example, an instruction that contains two sequential complex specifiers and the first one requires more than a single cycle to be processed in the specifier pipeline stage).

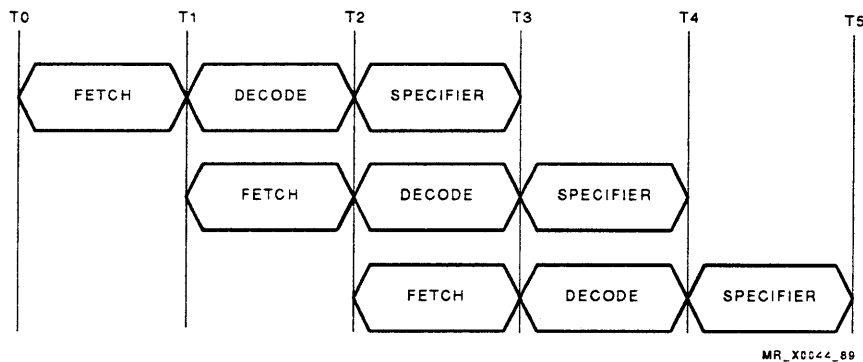


Figure 1-6 IBox Pipeline: No Stalls

Figure 1-7 shows the IBox pipeline when the processing of a specifier requires more than a single cycle and a subsequent specifier requires processing by the same specifier handler. This stall occurs when the CSU is processing an autodecrement-deferred indexing mode specifier and another complex specifier is decoded by the XBAR. The following events occur in this example of the IBox pipeline:

- During the first machine cycle (T0), the IBUF is loaded with the I-stream to be presented to the decode units of the IBox.
- During the second machine cycle (T1), the decode unit decodes an autodecrement-deferred indexing mode specifier and the fetch unit replenishes the decode units with I-stream.
- During the third machine cycle (T2), the CSU begins processing the autodecrement-deferred indexing mode specifier, the decode unit decodes a complex specifier and is replenished by the buffers representing the fetch stage of the IBox pipeline. Because the autodecrement-deferred indexing mode specifier requires multiple cycles to process and a subsequent complex specifier is to be processed by the CSU, the IBox pipeline stalls and cannot resume operations until the first complex specifier is processed. The autodecrement-deferred indexing mode specifier requires a minimum of four cycles to be processed.

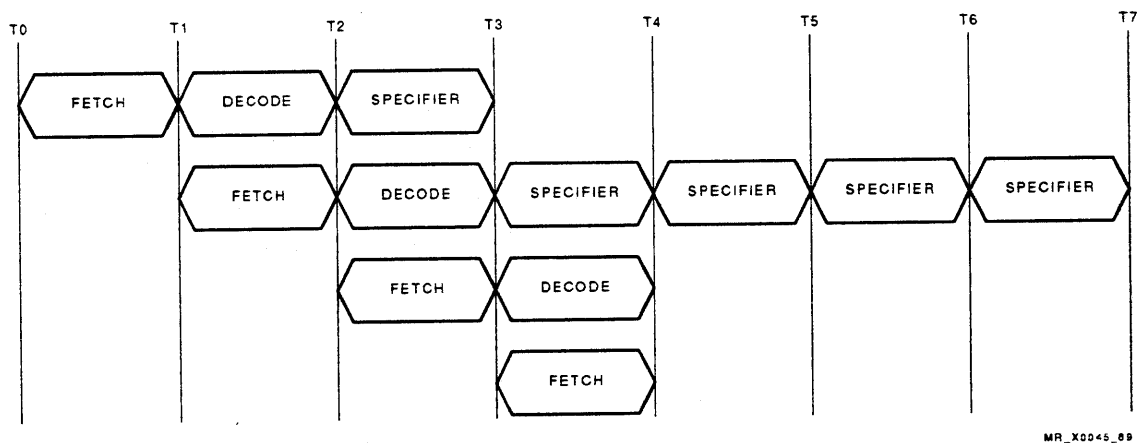


Figure 1-7 IBox Pipeline: CSU Stall

Figure 1-8 shows the pipeline when a branch is taken, but is predicted by the secondary prediction mechanism. The following events occur when a branch is predicted by the secondary prediction mechanism:

- During the second cycle (T1) of this example, a branch is decoded by the XBAR. The BPC is accessed, but no information regarding this branch is valid. The branch displacement is passed to the CSU to calculate the branch target address.
- During the third cycle (T2), the secondary branch prediction logic is accessed and the branch is predicted taken. The branch target PC calculation is completed by the CSU and the PC is passed to the fetch logic the next cycle.
- During the fourth cycle (T3), the fetch logic requests the data at the address supplied by the branch target PC to be loaded into the instruction buffer.

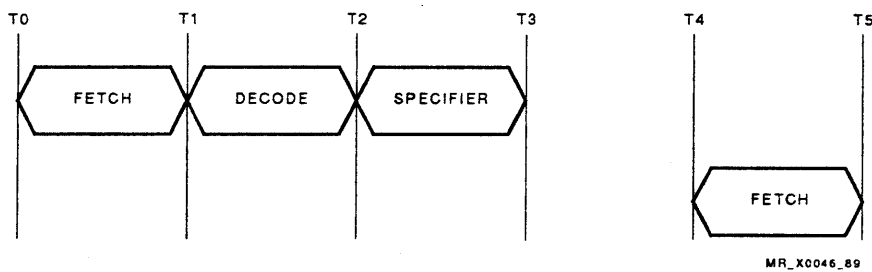


Figure 1-8 IBox Pipeline: Branch Taken, Secondary Prediction

Figure 1-9 shows the IBox pipeline during the execution of a branch instruction. The branch, in this case, is stored in the BPC and the history bit indicates that the branch is to be taken. The BPC lookup requires one extra cycle to predict the branch and load the correct target PC.

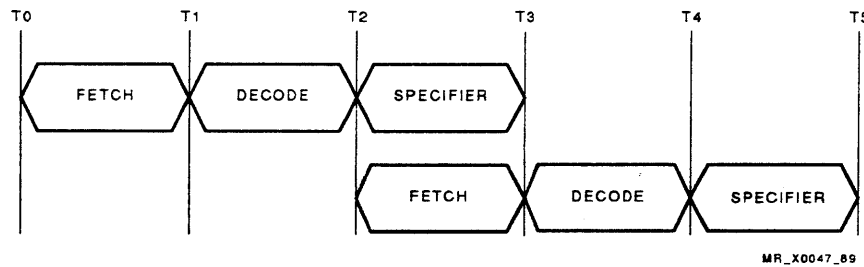


Figure 1-9 IBox Pipeline: BP Cache Hit, Prediction Taken

1.5 IBox Interfaces

The IBox interfaces to each of the other CPU functional units through dedicated interfaces (or ports). This section describes each interface.

1.5.1 MBox Interface

The IBox interfaces to the MBox through two ports in the MBox:

- **Instruction buffer port** — This port requests data from the MBox when a miss is encountered in the VIC.
- **Operand processing unit (OPU) port** — This port requests memory-related source operands from the MBox and passes operand addresses that specify memory destinations to the MBox.

1.5.1.1 Instruction Buffer Port

The instruction buffer port is a read-only port to the MBox. The IBox uses the instruction buffer port to issue requests to the MBox for I-stream data — 64 bits at a time with byte parity. The I-stream is retrieved from the MBox cache. In the case of a cache miss, the request is forwarded to memory through the system control unit (SCU). Typically, a request is for four (aligned) quadwords to fill the VIC. Requests are initiated by the instruction buffer on a VIC miss.

1.5.1.2 OPU Port

The OPU port is a read/write operand access port to the MBox. The port has a 32-bit wide data path with byte parity. Any rotation of the data (justification) is performed by the MBox. The port provides the following functions:

- Operand prefetch from cache or memory on behalf of the EBox and VBox
- Queuing of addresses for operands destined to cache and memory
- Prefetching operands that are address deferred (indirect) from cache or memory

The IBox issues requests on behalf of the EBox and VBox for operands that come from memory. The operands are passed directly from the MBox to the EBox source list.

The IBox sends the destination address to the MBox. In turn, the MBox performs a translation buffer (TB) lookup and stores the physical address in the write queue. The MBox then waits for the result data from the EBox.

For deferred addressing, the MBox returns the address of the operand to the IBox for a successive fetch for the data (operand). The data operand is returned to the EBox.

1.5.2 EBox Interface

The IBox interfaces to the EBox through the queue functional unit port in the EBox. This unit contains a set of FIFO buffers (queues) that accept instruction control information and operands from the IBox.

1.5.2.1 IBox-to-EBox Interface

This interface is used to send operands to the EBox 32 bits at a time with byte parity and their respective pointers. These operands are handled by the OPU within the IBox (sign or zero-extended data, integer and floating short literal operands, immediate mode data). The source and destination pointers are maintained in queues within the EBox, and they allow the EBox to access the correct data.

In addition, control information is passed to the EBox for microcode control, RLOG information, program count, and errors.

1.5.2.2 EBox-to-IBox Interface

This interface is used to transfer:

- Result data
- A starting or flushing PC
- RLOG unwind data
- Control information (branch valid, queue full, keep masks)

This interface transfers 32-bit EBox result data (including byte parity) to the IBox. The result data is generally an operand that has a destination of a GPR. The data, including the byte parity, is written to the IBox GPR set.

In addition to the various control signals, the EBox result data may also be of a controlling function, for example, an address passed to the IBox to initiate instruction fetch.

1.5.3 VBox Interface

The VBox issues requests for operands through the IBox. It sends the address (30 bits) with byte parity and control data. Control data is the reference data size, type of reference (read or write), and whether it is a block read.

1.5.4 Service Processor Interface

The service processor unit (SPU) interfaces through the scan latches throughout the IBox data and control paths, and error logic. The SPU can retrieve and store the state of the IBox scan logic for error reporting and possible recovery. Field Service can then perform analysis on the stored error symptom data for identification of the failing FRU.

The scan paths can implement symptom-directed diagnosis (SDD) as well as test-directed diagnosis (TDD). The scan latches have scan data/clock inputs and the usual system data/clock inputs. The scan inputs can be controlled by the SPU and diagnostics to shift test patterns in and test for the correct pattern that is subsequently retrieved.

This chapter describes the functions of the IBox program counter unit (PCU). It identifies the major PCs that control the IBox and the MCAs that comprise them. This chapter also provides a detailed explanation of the PC parity coverage and the construction of the 32-bit PCs across four MCAs.

2.1 Overview

The PCU directs the flow of the I-stream through the IBox and provides control to the branch prediction cache (BPC) and the virtual instruction cache (VIC). The PCU also directs the EBox and MBox in instruction fetch and instruction execution by supplying a copy of the PC to the EBox and by providing an address to the MBox on a VIC refill operation.

2.2 Major PCs

The IBox generates four major PCs:

- Prefetch PC
- Decode PC
- Branch PC
- Unwind PC

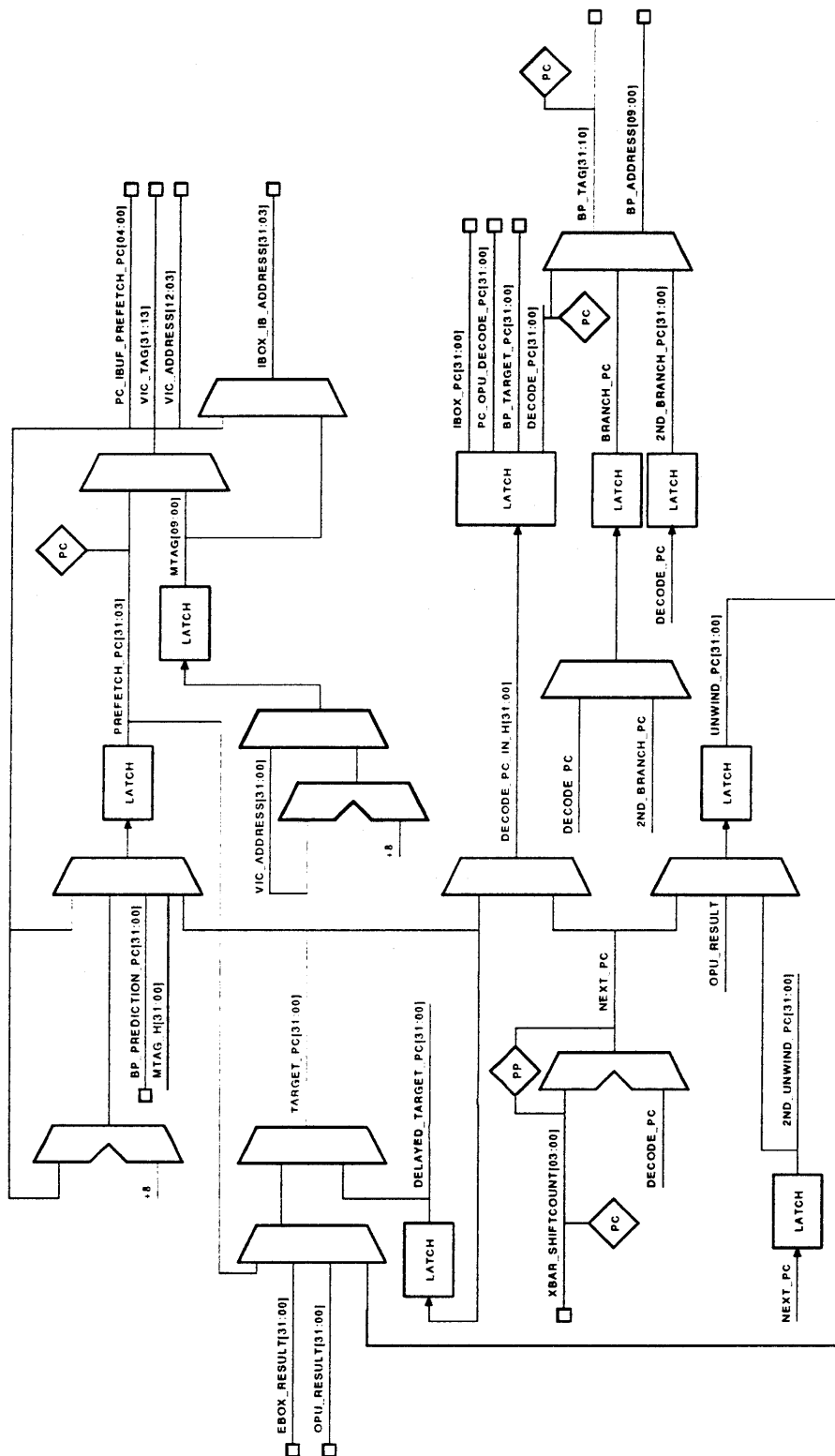
The primary inputs to the PCU are from the EBox, the OPU MCU, and the branch prediction cache (BPC). The EBox input (EBOX_RESULT_H[31:00]) directs the IBox to begin decoding a new I-stream as the result of a flush. The OPU input (OPU_RESULT_H[31:00]) provides the branch target address when a branch in the I-stream is not stored in the BPC. The target address is calculated by the complex specifier unit (CSU) and loaded into the decode PC (DECODE_PC_H[31:00]). When a branch stored in the BPC is decoded, BP_PREDICTION_PC_H[31:00] is loaded into the prefetch PC.

Figure 2–1 is a block diagram of the PCU data path.

2.2.1 Prefetch PC

The prefetch PC is 32 bits wide and is used during the instruction fetch stage of the pipeline. This PC is the address of the next quadword that the instruction buffer receives. This address is used to request data from the VIC. When a cache miss is encountered, this address is the address of a request to the MBox.

Each cycle, the prefetch PC is incremented by a quadword until the buffers of the instruction buffer are completely full. While the instruction buffer is full, the prefetch PC is held. The PC begins incrementing again when IBEX2 is empty.



DIGITAL INTERNAL USE ONLY

When there is a request for I-stream, the prefetch PC is used as the address to the MBox. All requests for I-stream are quadword aligned, so it is not necessary to send bits 0 through 2 of the PC to address the MBox data cache.

On a flush, the EBox provides the new PC to the IBox. The IBox then uses the PC provided to address the VIC and to address the MBox on a VIC miss.

2.2.2 Prefetch PC Data Path

The prefetch PC is loaded from the target PC, BP prediction PC, incremented PC, or MTAG. When a target PC is selected as the prefetch PC, it redirects the IBox from decoding sequential I-stream. The target PC is loaded from the:

- EBox during a flush
- OPU as a calculated branch PC
- Unwind PC when a branch prediction is incorrect

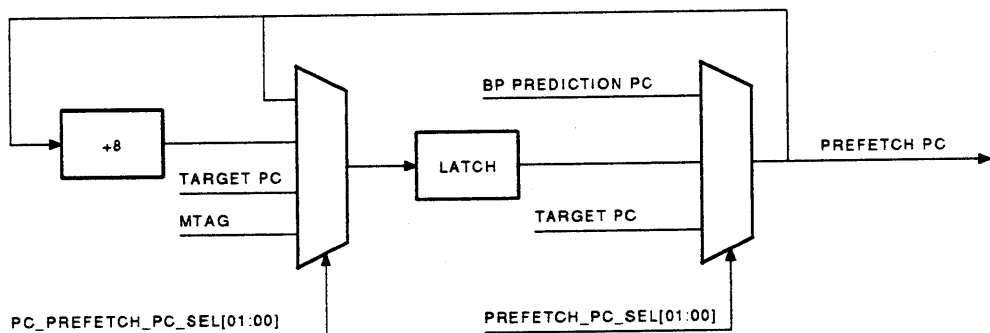
BP_PREDICTION_PC_H[31:00] is selected as the prefetch PC when the branch prediction cache is supplying the PC of the branch being decoded.

MTAG is loaded into the prefetch PC on each cycle of an MBox response. The MTAG is the memory tag of the returning data from the MBox. MTAG is incremented by eight each cycle of the MBox response and, when the response is complete, is loaded into the prefetch PC. The prefetch PC gets loaded with the MTAG because, during the MBox response, the prefetch PC can increment beyond the next quadword it is to receive.

The prefetch PC outputs control the VIC and provide the addresses of requests to the MBox.

Figure 2-2 describes the data path of the prefetch PC. The prefetch PC is selected from:

- **MTAG** — At the completion of a VIC refill, MTAG is loaded into the prefetch PC to ensure that the prefetch PC points to the next quadword loaded into the instruction buffer.
- **Target PC** — A target PC from one of the sources described in Section 2.2.3 can be loaded into the prefetch PC to redirect the I-stream.
- **Incremented PC** — This input increments the prefetch PC by a quadword.



MR_X0052_89

Figure 2-2 Prefetch PC Data Path

2.2.2.1 Prefetch PC Control

The prefetch PC is incremented each cycle until the instruction buffer informs the PCU that IBEX and IBEX2 are full. When these units are full, the prefetch PC is held until IBEX2 again requires an I-stream. This continues until a request is made for an I-stream that is not in the VIC or a new target PC redirects the prefetch PC from sequential decoding.

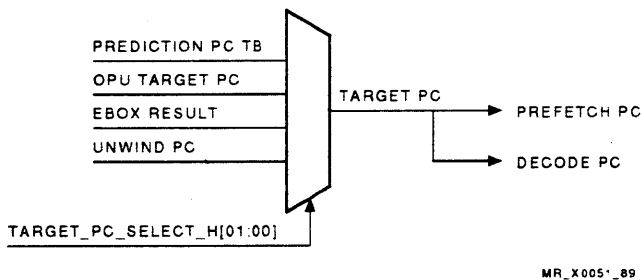
2.2.3 Target PC

The target PC is loaded into both the prefetch PC and the decode PC to redirect the I-stream from decoding sequential instructions. A branch in the I-stream or a flush loads a new target PC into the decode PC and prefetch PC.

Figure 2-3 describes the sources of the target PC.

TARGET_PC_SELECT_H[01:00] is controlled by the PCU microcode and selects an input to redirect the I-stream for each of the following conditions:

- Prediction PC is selected when a branch that is being decoded is stored in the BPC (BP hit).
- OPU target PC provides the target PC when a branch is predicted taken but not stored in the BPC (BP miss).
- EBox result provides the target PC when the EBox directs the IBox to flush to a new I-stream.
- Unwind PC is loaded into the target PC after a bad branch prediction.



MR_X005*_89

Figure 2-3 Target PC Sources

2.2.4 Decode PC

The decode PC is the PC of the instruction whose opcode is in the opcode byte (byte 0) of the instruction buffer. That is, it is the PC of the instruction currently being decoded. The decode PC does not reflect intermediate shift counts that result from partially decoded instructions. The decode PC is used in the instruction specifier stage of the IBox and is passed to the EBox to be placed in the PC history buffer. In the EBox it is called the VAX PC because it is the PC of the instruction to be executed and is used by the EBox when handling exceptions.

This PC is also used by the CSU to evaluate implied specifiers and branch displacements.

The decode PC is loaded from two sources. The next or sequential PC provides the decode PC when instruction decode is following a sequential path (straight line) and when branches are encountered that are predicted not taken. When a branch is predicted taken or a flush is encountered, a new (nonsequential) PC is loaded into the decode PC. This PC is supplied by the OPU or the branch prediction cache on branch predictions or is supplied by the EBox on a flush.

2.2.5 Decode PC Data Path

The decode PC is loaded from the next PC, when decoding sequential I-stream, or is loaded from the target PC. The target PC is loaded into the decode PC at the same time a new target PC is loaded into the prefetch PC. The next PC is generated each cycle by adding the number of bytes the XBAR decodes to the decode PC. When the instruction under decode is completely decoded, this addition yields a new decode PC.

The decode PC is output to the EBox (IBOX_PC_H[31:00]) to be placed in a queue until the instruction is executed. The CSU of the OPU receives a copy of the decode PC (PC_OPU_DECODE_PC_H[31:00]) to identify the instruction that is being decoded. When a branch instruction is being processed, and the branch meets the criteria required to be written in the branch prediction cache, the decode PC is written to that cache. The BP tag is written with DECODE_PC_H[31:10] at the address supplied by DECODE_PC_H[09:00].

Each cycle, the decode PC is added to the number of bytes decoded by the XBAR (XBAR_SHIFTCOUNT_H[03:00]). When all of the specifier bytes of an instruction are decoded (XSCA_SHIFTOPCODE_H asserted), the next sequential PC or next PC is loaded as the decode PC.

Each new decode PC that is loaded asserts IBOX_PC_H[31:00] in the EBox and OPU_DECODE_PC_H[31:00] in the OPU MCU. The EBox copy of the decode PC is stored in a queue for instructions to be executed. The OPU copy of the decode PC is used to calculate specifier PCs and branch displacement.

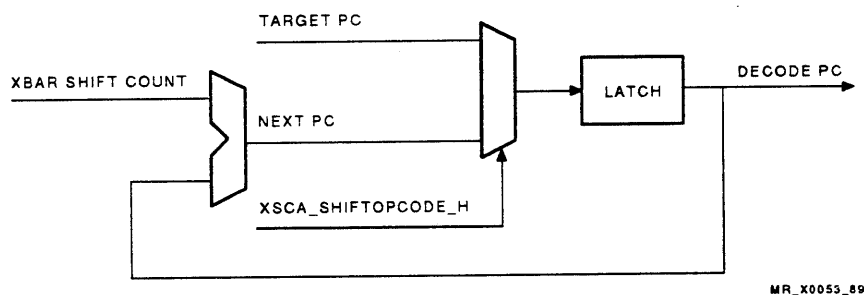


Figure 2-4 Decode PC Data Path

2.2.5.1 Delta PC

The XBAR shift count (XSCA_SHIFTCOUNT_H[03:00]) is accumulated to produce the decode delta. That is, the total number of bytes in an instruction yields the decode delta. The decode delta provides the OPU and the BPC with the instruction length.

2.2.6 Branch PC

The branch PC is the address of the branch instruction that is currently under evaluation. When the IBox encounters a branch, the decode PC is loaded into the branch PC. The branch PC is used to address the branch prediction cache. The branch PC is compared with the branch prediction (BP) tag to produce a hit or miss in the branch prediction cache.

2.2.7 Branch PC Data Path

The branch PC is loaded from either the decode PC or the second branch PC. When a branch instruction is being processed, the address of the branch is saved until the branch is shifted out of the instruction buffer. After the branch is shifted out of the instruction buffer, the branch PC provides the information pertaining to the branch that is written to the branch prediction cache.

Because the IBox can process two branches simultaneously, a second branch PC must be generated for the second branch. The second branch PC (SECOND_BRANCH_PC_H[31:00]) is stored in a scan latch until the first branch is completely processed. The second branch PC is then loaded into the branch PC and used as data to be written to the branch prediction cache.

Figure 2-5 describes the data path of the branch PC. The decode PC is loaded into the branch PC for each branch the IBox encounters. The address is saved until the branch is completely processed. If two branches are being processed, each address is saved.

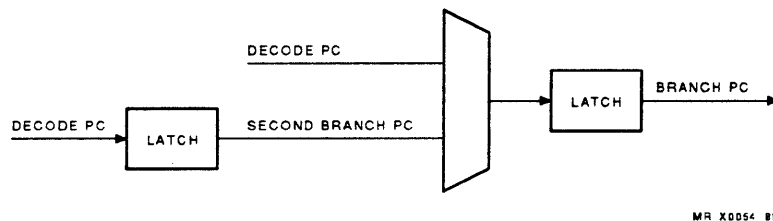


Figure 2-5 Branch PC Data Path

2.2.8 Unwind PC

The unwind PC is the PC of the direction not taken in a branch prediction. When the IBox makes a branch prediction, two PCs relate to the prediction: the branch target PC and the next or sequential PC. When a branch is predicted taken, the branch target PC or the prediction PC is the path that the instruction execution follows. The next PC is the path that is not taken. The IBox saves the path that is not taken (in this case, the next sequential PC) in case the IBox predicted the branch incorrectly. If the EBox informs the IBox that a branch has been incorrectly predicted, the IBox can correct the prediction by loading the unwind PC into the decode and prefetch PCs and start decoding in the correct path.

Because the IBox can process two branches simultaneously, it is necessary to have two unwind PCs. These PCs are called the unwind PC and the second unwind PC. These PCs are both used when a branch is predicted and a second branch is predicted before the first has been validated by the EBox. Once the first branch is validated, the second unwind PC is loaded into the first unwind PC. The original, first unwind PC is discarded.

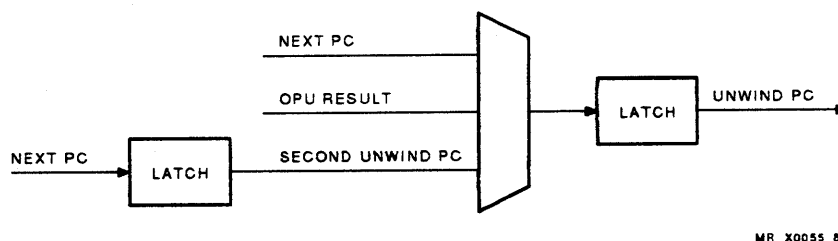
2.2.9 Unwind PC Data Path

When the IBox predicts a branch, the PC of the instruction that will not be executed is saved as the unwind PC. In the event of a wrong prediction, the PCU can return to the PC that was saved and continue with the correct I-stream.

The PCU can maintain information pertaining to two conditional branches, therefore, two unwind PCs are stored. Figure 2-6 describes the data path of the unwind PC. The unwind PC is loaded from one of three sources:

- **Next PC** — This PC is loaded when a branch is predicted taken.
- **OPU result** — This PC is loaded when a branch is predicted not taken.
- **Second unwind PC** — This PC is loaded by latching the next PC and loading it as the unwind PC.

The second unwind PC is loaded into the unwind PC only after the first branch that has been predicted has been validated by the EBox.



MR_X0055_89

Figure 2-6 Unwind PC Data Path

2.3 Cache Control

Control of the branch prediction cache (BPC) and the virtual instruction cache (VIC) is provided by the PCU. The VIC is addressed by the prefetch PC on read and write operations, the VIC tag is compared to the prefetch PC to determine a match when requesting data, and the prefetch PC provides the address for MBox requests on a VIC refill. The decode PC provides both the BP tag and the BP address. Figure 2-7 shows the relationship of the cache control signals to the PCU outputs.

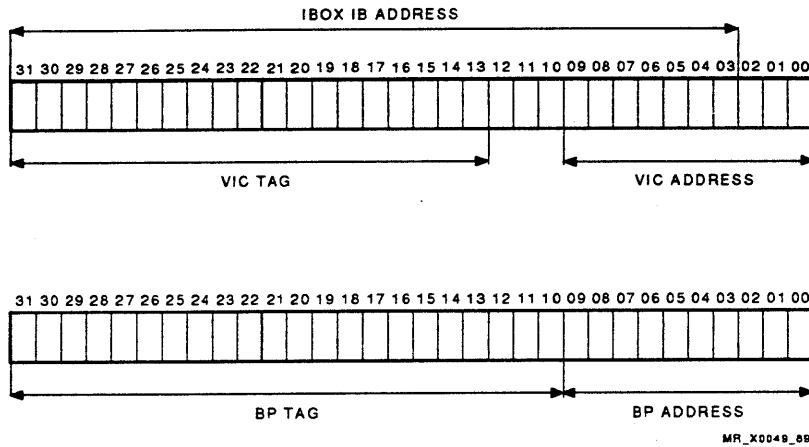


Figure 2-7 PCU Cache Control

2.3.1 BPC Control

The branch PC provides BPC fields. The BP tag is bits 31 through 10 of the branch PC and is written when the branch is first encountered. The tag is written at the cache address that is supplied by decode PC bits 0 through 9. When a branch is encountered, the virtual address of the branch (DECODE_PC_H[31:00]) is compared to the BP tag to determine a hit or miss in the BPC.

2.3.2 VIC Control

The prefetch PC provides control for the VIC. This PC provides the address of a request to the MBox (IBOX_IB_ADDRESS_H[31:03]), the VIC tag (VIC_TAG_H[31:13]), and to address the VIC when replenishing the instruction buffer.

Bits 31 through 13 provide the VIC tag. The tag is written on a VIC refill and subsequently compared with bits 31 through 13 of the prefetch PC when the instruction buffer is requesting VIC data.

Bits 0 through 9 of the prefetch PC provide the cache address on VIC requests and, when requesting data from the MBox data cache bits 31 through 3 of the prefetch PC, provides the requested address (IBOX_IB_ADDRESS_H[31:03]).

2.4 PCU Error Detection

Each of the MCAs that comprise the PCU receives inputs from most of the other functional units in the IBox and from the MBox and EBox. Each PCU MCA contains parity detection logic that detects errors in the internally generated signals and inputs to the PCU. Errors detected in the PCU MCAs result in both fetch and decode errors being asserted in the IBox and EBox.

This section contains block diagrams of the error logic of each PCU MCA and lists the decode and fetch errors that can occur in each MCA.

2.4.1 PCVC Error Logic

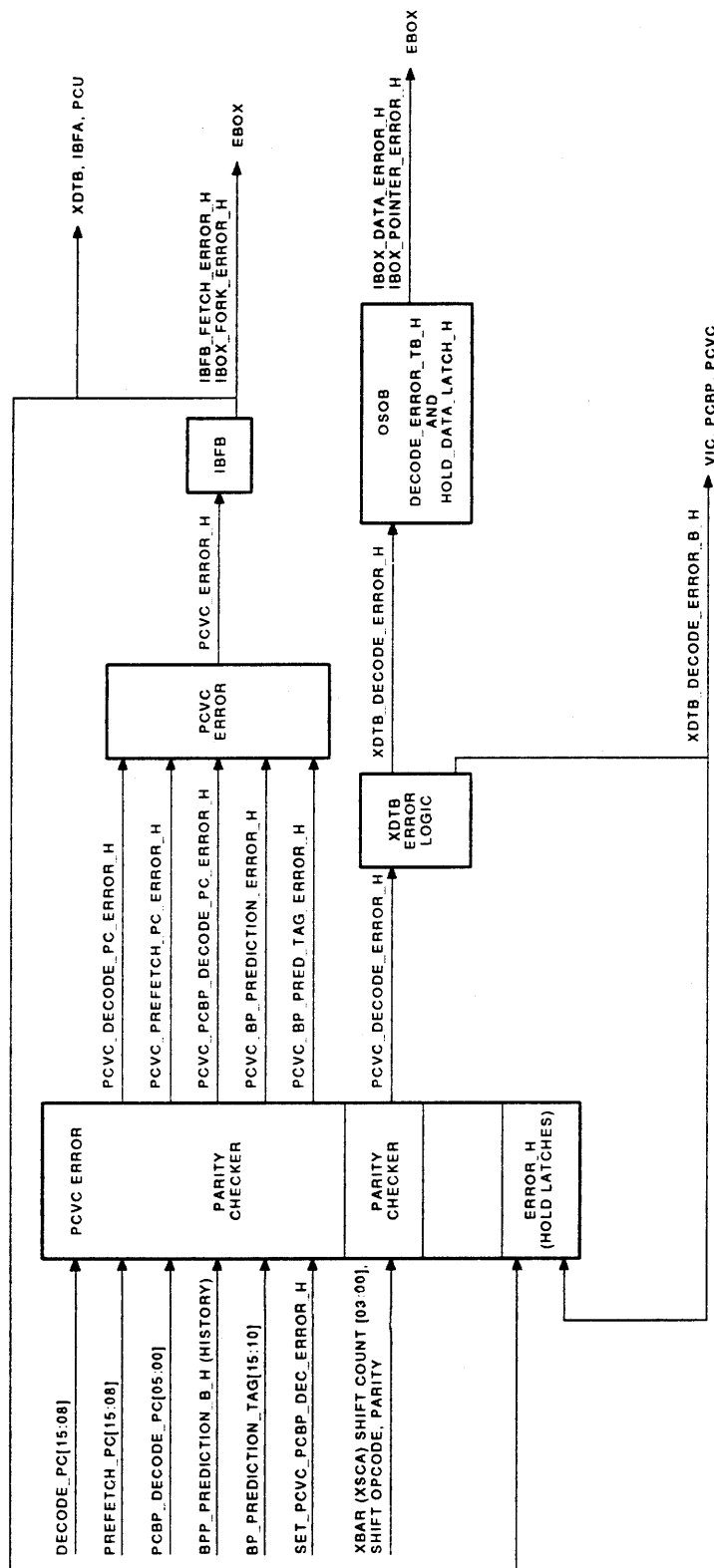
PCVC fetch errors are latched in IBFB as an IBFB_FETCH_ERROR and then asserted in the EBox as IBOX_FORK_ERROR_H. During this process, the scan latches in PCVC receive a hold signal. The hold signal preserves the state of the error scan latches so that the intermediate error can be determined.

PCVC decode errors are latched in XDTB and OSQB as XDTB_DECODE_ERROR_H. OSQB asserts IBOX_POINTER_ERROR_H and DATA_ERROR_H in the EBox.

Figure 2-8 shows a block diagram of the PCVC error logic and Table 2-1 lists the error signals generated in this MCA.

Table 2-1 PCVC Errors

Input Signal	Intermediate Error
Fetch Errors	
DECODE_PC[15:08]	PCVC_DECODE_PC_ERROR
PREFETCH_PC_H[15:08]	PCVC_PREFETCH_PC
PCBP_DECODE_PC[05:00]	PCVC_PCBP_DECODE_PC_ERROR
BP_PREDICTION	PCVC_BP_PREDICTION_ERROR
BP_PREDICTION_TAG_H[15:00]	PCVC_BP_PRED_TAG_ERROR
Decode Errors	
XSCA_XSCA_SHIFTCOUNT_H[03:00]	PCVC_DECODE_ERROR
XSCA_SHIFTOPCODE	PCVC_DECODE_ERROR



MR_X0056_00

Figure 2-8 PCVC Error Logic

2.4.2 PCBP Error Logic

Figure 2-9 shows a block diagram of the PCBP error logic and Table 2-2 lists the error signals generated in this MCA.

Table 2-2 PCBP Errors

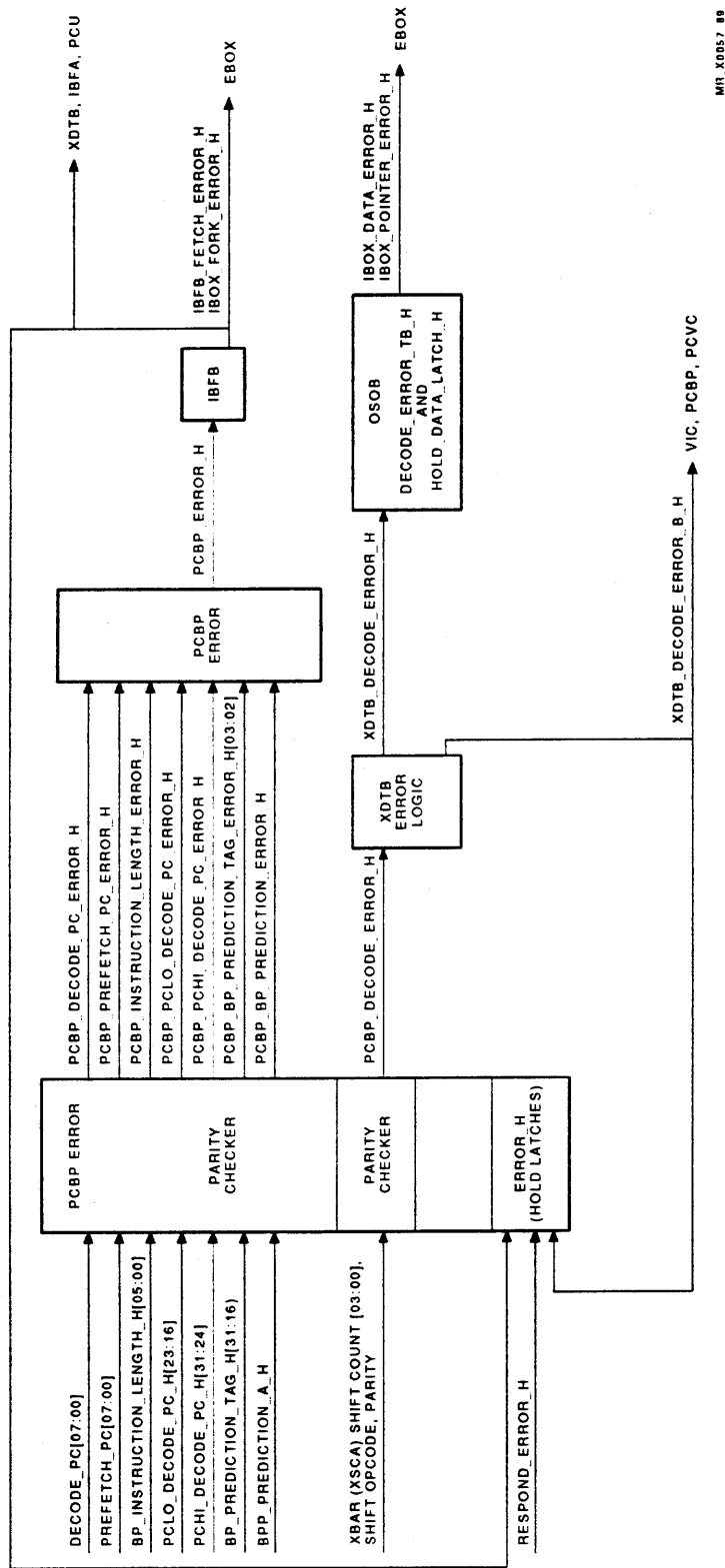
Input Signal	Intermediate Error
Fetch Errors	
DECODE_PC[07:00]	PCBP_DECODE_PC_ERROR
PREFETCH_PC_H[07:00]	PCBP_PREFETCH_PC_ERROR
INSTRUCTION_LENGTH[05:00]	INSTRUCTION_LENGTH_ERROR
DECODE_PC[23:16]	PCBP_PCLO_DECODE_PC_ERROR
DECODE_PC[31:24]	PCBP_PCHI_DECODE_PC_ERROR
BP_PREDICTION	BP_PREDICTION_ERROR
Decode Errors	
XSCA_SHIFTCOUNT_H[03:00]	XSCA
XSCA_SHIFTOPCODE_H	XSCA

2.4.3 PCLO Error Logic

Figure 2-10 shows a block diagram of the PCLO error logic and Table 2-3 lists the error signals generated in this MCA.

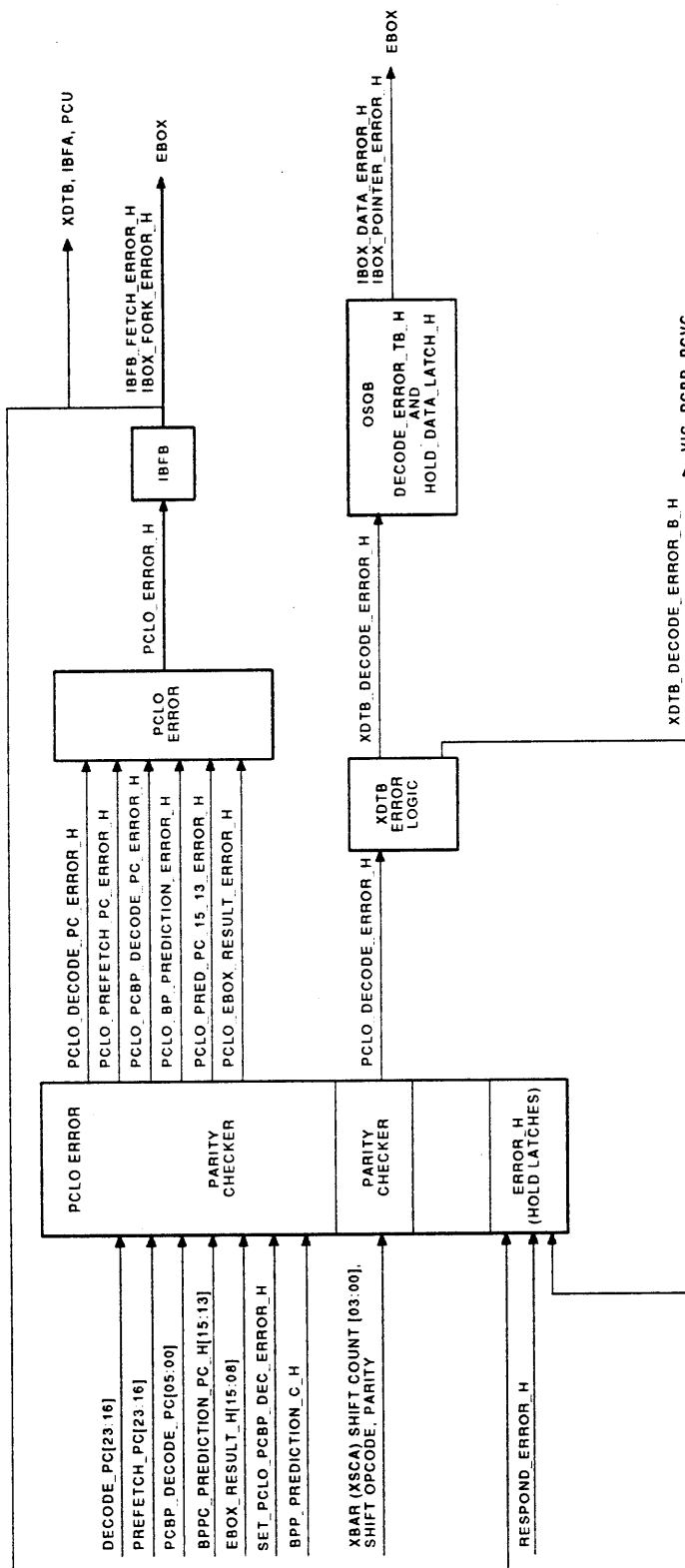
Table 2-3 PCLO Errors

Input Signal	Intermediate Error
Fetch Errors	
DECODE_PC[23:13]	PCLO_DECODE_PC_ERROR
PREFETCH_PC_H[23:13]	PCLO_PREFETCH_PC_ERROR
DECODE_PC[05:00]	PCLO_PCBP_DECODE_PC_ERROR
EBOX_RESULT[15:08]	PCLO_EBOX_RESULT_ERROR
BP_PREDICTION_PC[15:13]	PCLO_PRED_PC_15_13_ERROR
BP_PREDICTION	PCLO_BP_PREDICTION_ERROR
Decode Errors	
XSCA_SHIFTCOUNT_H[03:00]	PCLO_DECODE_ERROR
XSCA_SHIFTOPCODE_H	PCLO_DECODE_ERROR



MR_X0057_89

Figure 2-9 PCBP Error Logic



MR X0058_89

Figure 2-10 PCLO Error Logic

2.4.4 PCHI Error Logic

Figure 2-11 shows a block diagram of the PCHI error logic and Table 2-4 lists the error signals generated in this MCA.

Table 2-4 PCHI Errors

Input Signal	Intermediate Error
Fetch Errors	
DECODE_PC[31:24]	PCHI_DECODE_PC_ERROR
PREFETCH_PC_H[31:24]	PCHI_PREFETCH_PC_ERROR
DECODE_PC[05:00]	PCHI_PCBP_DECODE_PC_ERROR
DISPLACEMENT[11:08, 03:00]	PCHI_XDTA_DISP_ERROR
DISPLACEMENT[15:12, 07:04]	PCHI_XDTB_DISP_ERROR
BPTD_TAG_DISPLACEMENT[15:00]	PCHI_BP_TAG_DISP_ERROR[01:00]
BP_PREDICTION	PCHI_BP_PREDICTION_ERROR
Decode Errors	
XSCA_SHIFTCOUNT_H[03:00]	PCHI_DECODE_ERROR
XSCA_SHIFTOPCODE_H	PCHI_DECODE_ERROR

2.5 PCU Inputs

The MBox inputs to the PCU are in response to an instruction buffer request to the MBox. The two signals the PCU receives are as follows:

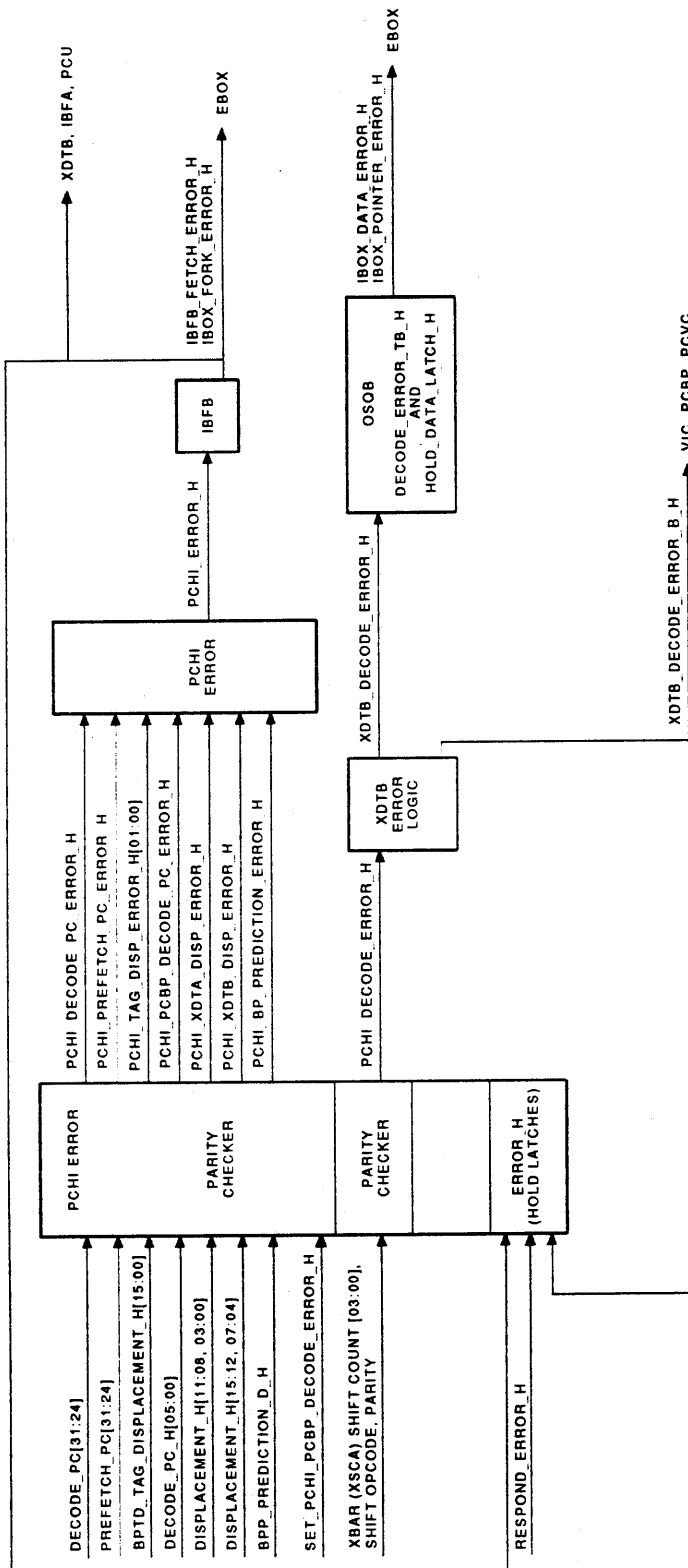
- MBOX_IB_RESPONSE_H directs the control of the MTAG and the prefetch PC when the MBox is returning data for a VIC refill.
- MBOX_IB_PAGEFAULT_H inhibits writing page faulted data to the VIC.

The EBox inputs to the PCU validate predicted branches and, when the IBox is flushed, provides a starting address. The primary EBox inputs to the PCU are as follows:

- EBOX_RESULT_H[31:00] provides the starting address after a flush.
- EBOX_BRANCH_A_H, when asserted, indicates a bad branch prediction.
- EBOX_BRANCH_VALID_A_H validates EBOX_BRANCH_A_H.

The XBAR provides shift counts and error status to the PCU. The XBAR inputs are as follows:

- XSCA_SHIFTCOUNT_H[03:00] is used to increment the decode PC.
- XDTB_DECODE_ERROR_H informs the PCU of an error in the instruction decode pipeline stage of the IBox. The signal sets the PCU in a hold state.



MR.X0055.89

Figure 2-11 PCHI Error Logic

The OPU provides flush signals and the target PC for branch instructions. The OPU inputs are as follows:

- **OCTL_PC_FLUSH** flushes the IBox to a new decode PC, provided by the EBox.
- **OCTL_IBUF_FLUSH** holds the prefetch PC until the IBox can resume decoding an I-stream. The signal is sent when the IBox has to wait for one of the VIC block valids to be flushed.
- **OPU_TARGET_PC_H[31:00]** is provided by the CSU on branch instructions.
- **OCTL_VIC_FLUSH_H** flushes the VIC.

2.6 PCU Outputs

The PCU directs outputs to the MBox, EBox, XBAR, and OPU. The outputs direct the flow and execution of instructions through the functional units that receive them.

The MBox receives a byte parity protected address that directs the prefetching of I-stream. These two outputs are as follows:

- **IBOX_IB_ADDRESS[31:03]** is the prefetch PC. (The lower three bits are always zero.)
- **IBOX_IB_ADDRESS_PARITY[03:00]** is the byte parity for the address that the PCU is sending the MBox.

The EBox receives a copy of the decode PC and information pertaining to branches. The signals the EBox receives are as follows:

- **IBOX_PC[31:00]** is a copy of the decode PC. The EBox, after execution of the instruction to which this PC points, stores it in the PC history buffer, or uses it during exception processing.
- **IBOX_PC_PARITY[03:00]** is the byte parity for the IBox PC.
- **IBOX_CORRECTION_H** is asserted if the IBox can correct a bad branch prediction before it is shifted out of the IBUF.
- **IBOX_PC_VALID_H** is asserted when a new valid opcode is shifted into the opcode byte of the IBUF.
- **IBOX_PREDICTION_H**, asserted, informs the EBox that the IBox has predicted a branch to be taken.

The XBAR receives two signals from the PCU:

- **PCHI_UNWIND_H** is asserted when the EBox detects a bad branch prediction. The signal asserts a restart signal in the three XBAR MCAs.
- **PCHI_DECODE_ERROR_H** is asserted by a parity error on **XSCA_SHIFTCOUNT_C_H** or **XSCA_SHIFTOPCODE_C_H**.

The following PCU outputs provide the OPU with the decode PC and branch information:

- **DECODE_PC_H[31:00]** is the CSU's copy of the decode PC.
- **PCHI_UNWIND_H** is asserted on a bad branch prediction. This signal initiates a flush of the CSU.
- **PCHI_CORRECTION_H** is input to the branch count logic of the CSU.
- **PCVC_VIC_FIP_H** is sent to OCTL when the VIC is being flushed. Asserting this signal directs OCTL to stall the IBox if another flush is received before the first flush is complete. (The VIC flush requires 256 cycles.)

This chapter describes the functional units representing the instruction fetch pipeline stage: the VIC, the instruction buffer, and the MBox interface.

3.1 VIC

The VIC is a direct-mapped, 8-Kbyte cache with a block size of 32 bytes and a fill size of 8 bytes. Physically, the VIC is made up of four groups of STRAMs:

- **Data STRAMs** — Store one quadword of data per location.
- **Tag STRAMs** — Store bits [31:13] of the address of each block in the VIC.
- **Block valid STRAMs** — Store one bit to indicate that the VIC block is valid.
- **Quadword valid STRAMs** — Store four bits to indicate that a corresponding quadword is valid.

VIC data STRAMs are on the VIC MCU and the VIC tag STRAMs are on the XBR MCU. A VIC block consists of four quadwords of VIC data. There are 1024 VIC data locations and 256 VIC tags, VIC block valid bits, and VIC quadword valid bits. There are two separate block valid bits for the VIC. Figure 3–1 is a simplified block diagram of the VIC.

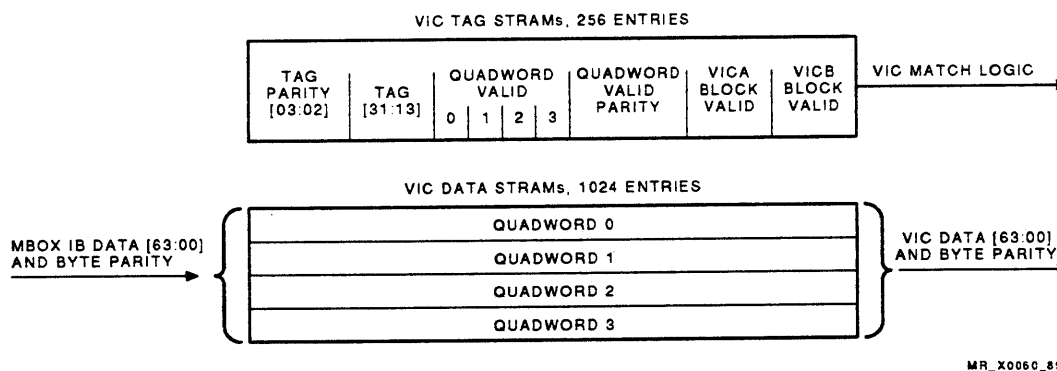


Figure 3–1 VIC

3.1.1 VIC Hit

The instruction buffer determines a VIC hit (VIC_MATCHA_H or VIC_MATCHB_H asserted) if the following conditions are met:

- The VIC block is valid.
- The prefetch PC matches the VIC tag.
- The quadword is valid.

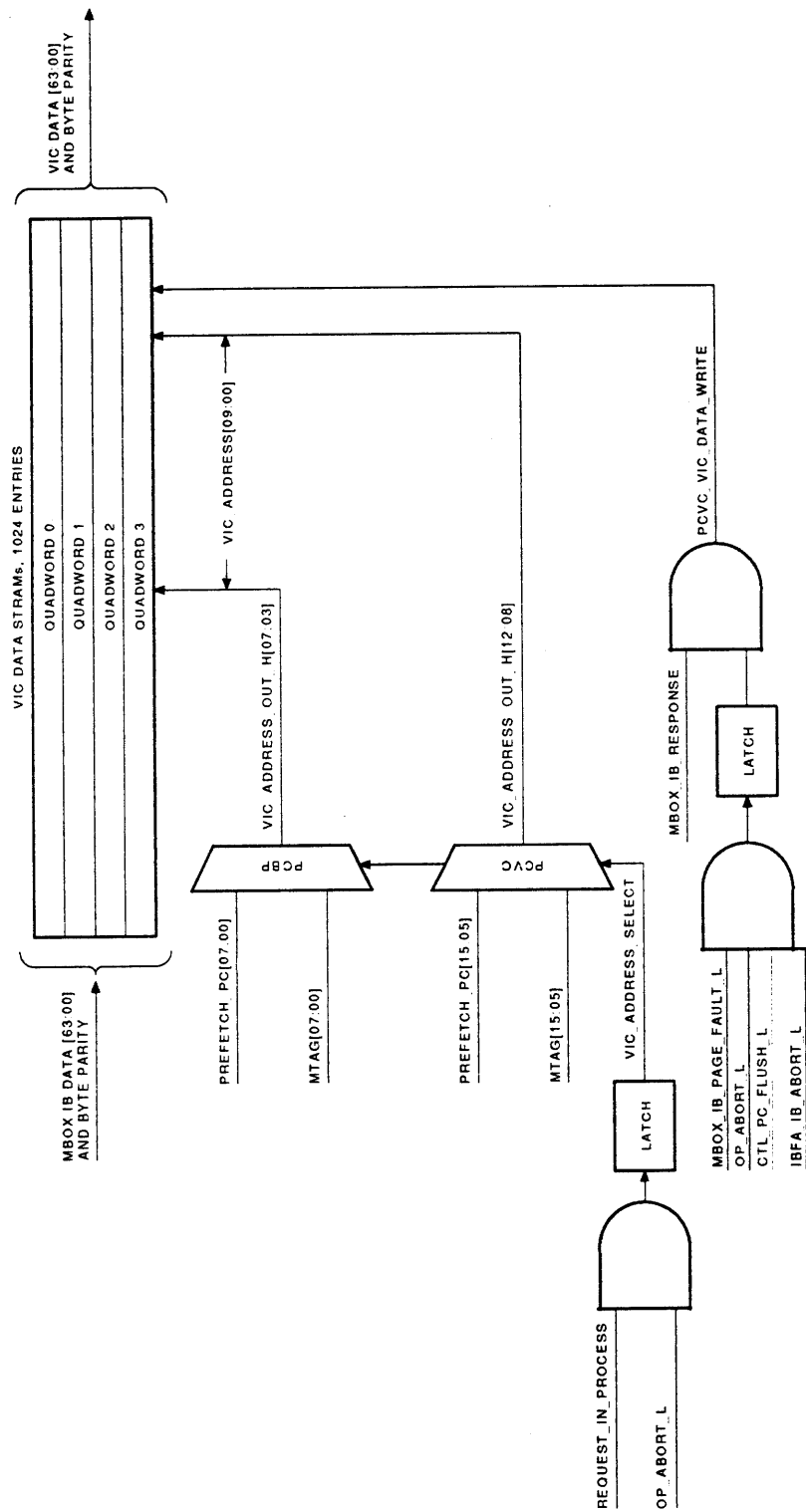
If these conditions are not met, a VIC miss is signaled and an instruction buffer request is made to the MBox.

3.1.2 VIC Data Write

Control of the VIC is provided by the prefetch PC and the instruction buffer. VIC match and instruction buffer request logic are in the instruction buffer, while the requested address is provided by the prefetch PC.

Figure 3-2 shows a block diagram of the VIC data write function. The instruction buffer request initiates the following sequences:

- One cycle after the request is made, the instruction buffer asserts IBUF_REQUEST_IN_PROCESS. This signal selects the memory tag (MTAG) to address the VIC.
- MBOX_IB_RESPONSE is asserted when the MBox responds to the request. Without any stalls present or a page fault on the requested data, this signal provides the write enable for the VIC data.



MR X0001 . 89

Figure 3-2 VIC Data Write

3.1.2.1 VIC Address Selection

The VIC is addressed by the prefetch PC on VIC requests and is addressed by the MTAG when the I-stream is being returned from the MBox. The MTAG is the address that is returned from the MBox in response to an instruction buffer request.

Figure 3-3 shows the generation of the VIC address. The prefetch PC provides the requested address to the MBox and is then held from incrementing until the MBox returns the requested data. When the data is returned (MBOX_IB_RESPONSE_H asserted), the prefetch PC begins incrementing again.

VIC_ADDRESS_SELECT_H is asserted to select MTAG when the MBox returns the requested I-stream.

The prefetch PC does not address the VIC when data is returning from the MBox because the XBAR does not always consume a quadword of I-stream in a single cycle. If the prefetch PC did address the VIC and the XBAR did not consume a quadword of I-stream each cycle of the response, the prefetch PC would be incremented past the last quadword that the instruction buffer receives next. The prefetch PC is loaded with the MTAG at the end of an instruction buffer request so that it points to the next quadword that the instruction buffer requests.

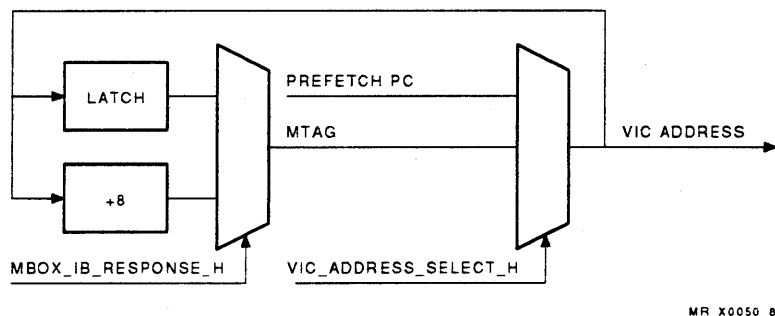


Figure 3-3 VIC Address Selection

3.1.2.2 VIC Data

With the VIC data write signal (PCVC_VIC_DATA_WRITE_H) enabled and the VIC address selected, the MBox returns the I-stream and writes it to the STRAMs. The I-stream arrives one quadword per cycle and is byte parity protected.

Most requests for VIC data are for four quadwords, but requests for less can be issued. If a request is for less than four quadwords, the MBox returns the requested quadword and the remaining quadwords in the block. That is, the request does not wrap around the block. For example, if the requested address is for the third quadword in a block, quadwords 2 and 3 are returned, but quadwords 0 and 1 are not.

3.1.3 VIC Tag Write

As VIC data is being written to the cache, the tags are also written. As the requested I-stream is returned from the MBox, the PCU writes the tag field (bits [31:13] of the MTAG) and validates the block and quadword valid bits for that location in the VIC. Figure 3-4 is a block diagram of the VIC tag write function.

MBOX_IB_RESPONSE_H is asserted in the first cycle of a response from the MBox and enables writing the tag and valid fields to the VIC tag (VICT) STRAMs.

The following sections describe the four write functions associated with the VIC tag field. The four fields to be written are as follows:

- VIC tag
- Quadword valid bits
- Block valid bit
- Tag parity

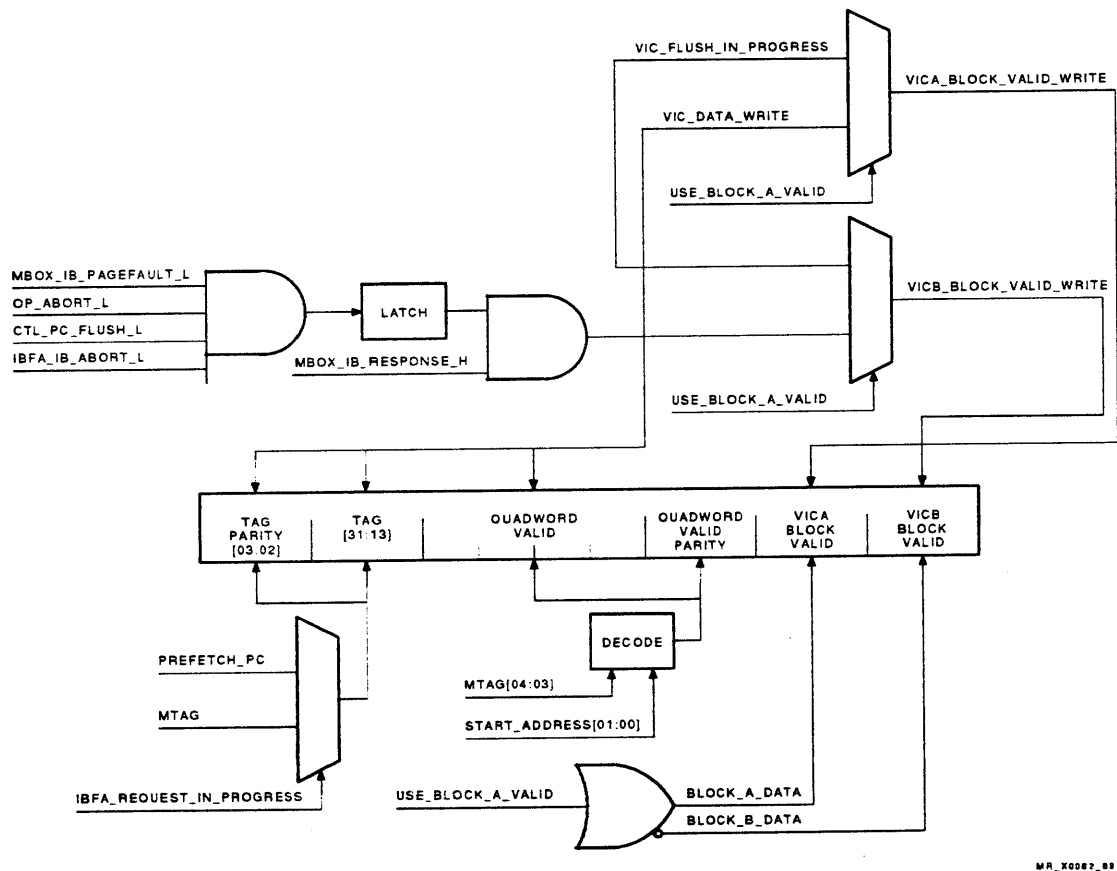


Figure 3-4 VIC Tag Write

3.1.3.1 Tag Write

When the requested I-stream is returned from the MBox, the PCU writes MTAG bits [31:13] to the tag field. The address of the request was originally supplied by the prefetch PC, but when the I-stream is returning, the VIC is addressed by the MTAG.

When the VIC is receiving multiple responses from the MBox, the address of the VIC must be incremented by a quadword in each cycle of the MBox response. The prefetch PC is not incremented by a quadword until a quadword of I-stream is consumed by the XBAR. Because of this, a new address (MTAG) must be used when the I-stream comes from the MBox to the VIC.

To address subsequent returning quadwords, bits [04:03] of the VIC address are incremented each cycle. The VIC address is then loaded into the MTAG and used as the returning quadword address.

3.1.3.2 Quadword Valid Write

As each quadword returns, bits [04:03] of the MTAG are decoded to identify the quadword being written. The bit field is decoded and the single bit valid field is written as the data is written to the data field. Validating this field as the data is being written provides valid data, for the instruction buffer, in the first cycle of the MBox response.

3.1.3.3 Block Valid Write

The VIC contains two sets of the block valid field. The 2-bit field, when validated, indicates that valid data is in the VIC. The purpose of the two copies of the field is to enable a VIC flush in a single cycle.

To validate this field, the correct block valid is selected, then the valid bit is latched into the STRAM. The field is validated when the first quadword, of a request, is returned from the MBox.

A block valid selection and write is initiated by the assertion of MBOX_IB_RESPONSE_H. The selection of the block valid is achieved by inverting USE_BLOCK_A_VALID. This selects the block that was not in use since the last VIC flush.

3.1.3.4 Tag Parity

All VIC tag parity checking is performed in IBFA. When a VIC tag parity error is detected in IBFA, IBFA_VIC_ERROR_H is asserted and passed to the IBFB fetch error circuitry. IBFB asserts IBFB_FETCH_ERROR_H and forwards it to the EBox when a VIC tag parity error is detected.

3.1.4 VIC Flush

The VIC is flushed on every REI and, in most cases, the flush is achieved in a single cycle.

A flush is initiated when OCTL_VIC_FLUSH_H is asserted. This signal is generated in OCTL and is the result of a flush code sent by the EBox. Receipt of this signal inverts the selection of the block valid being used and initiates the flush of the original block valid that was in use. The flush requires 256 cycles.

Each cycle of the block valid flush, VIC_CLEAR_ADDRESS_H[08:00] is incremented. The flush is complete when bit 8 is set (VIC_CLEAR_ADDRESS_H[08:00] = 1FF).

Once a flush is initiated by OCTL, the PCU asserts VIC_FLUSH_IN_PROGRESS_H. This signal remains asserted until the flush is complete and prohibits issuing another flush request until it is negated.

3.1.5 VIC Data Read

When the instruction buffer requires a new I-stream, compare logic on the IBFA MCA checks the VIC tag field for a resident block and valid quadwords then compares the requested address with the VIC address tag field. If there is a match, a VIC hit (VIC_MATCH_H) is signaled and the requested I-stream is loaded into the instruction buffer. Figure 3-5 is a simplified block diagram showing the VIC compare logic.

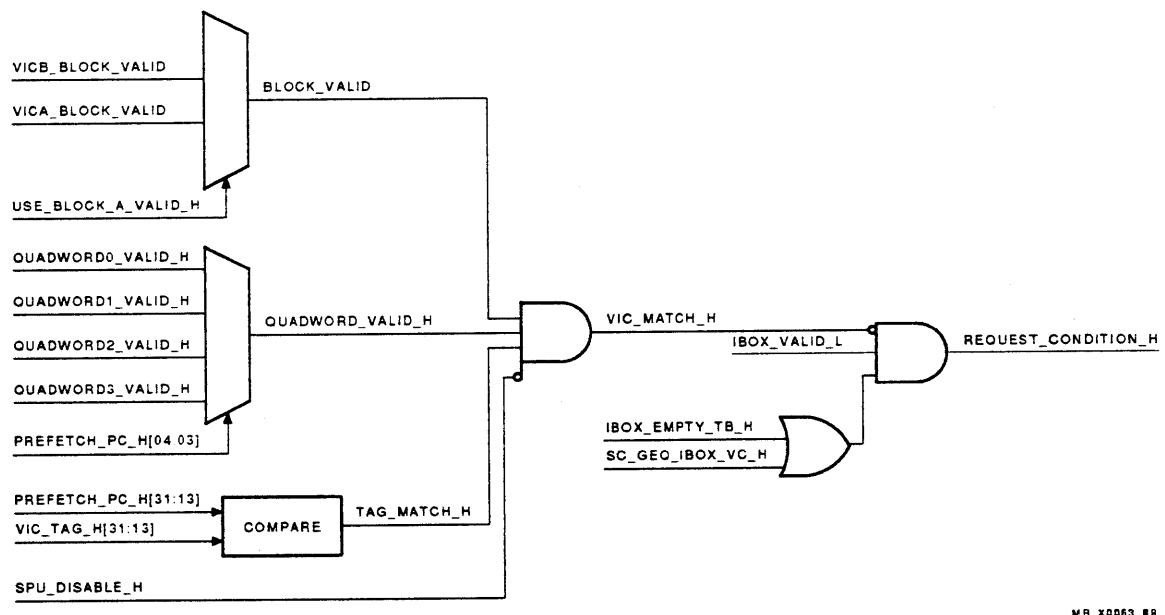


Figure 3-5 VIC Match Logic

The block valid bit is checked to detect a valid VIC block. The prefetch PC (bits [04:03]) selects the quadword valid to be compared. The tag address is compared with prefetch PC [31:13]. A match of the two address fields enables VIC_MATCH_H.

A VIC hit asserts READ_VIC_DATA_H and the requested I-stream is written to the instruction buffer.

3.1.5.1 VIC STRAM Bypass

During a VIC refill, it is possible to write and validate the data and latch the data in the instruction buffer in the same cycle.

When the MBox is returning the first quadword of an I-stream, the block and quadword are marked valid and the tag and data are written to the STRAMs. By placing the data on the output latches of the STRAMs, the instruction buffer is allowed to receive the valid data late in the same cycle.

If the XBAR consumes the instruction buffer data in a single cycle and the MBox is simultaneously returning VIC data, subsequent quadwords are latched in the instruction buffer through the VIC STRAM bypass.

3.1.6 VIC Parity Coverage

The VIC can cause two errors: fetch errors and decode errors. The VIC tag (VICT) fields are parity checked on the IBFA MCA. VIC data (VICD) is parity checked on IBFA and IBFB.

VIC data is byte parity protected and is checked on the output, as the data is latched into the instruction buffer. IBFB_FETCH_ERROR_H is asserted when this error is detected.

Block valid parity, tag parity, and quadword valid parity is checked on IBFA. Errors in these units assert IBFA_VIC_ERROR_H. This signal is latched to IBFB and asserts IBFB_FETCH_ERROR_H.

The SPU can disable the VIC parity error detection logic. Enabling this feature suppresses reporting of VIC parity errors.

3.1.7 Disabling VIC Hits

The SPU can disable the VIC hit logic. Enabling this feature causes every VIC access to result in a VIC miss. The instruction buffer is then forced to initiate an MBox request each time it needs an I-stream. This feature can be used as a troubleshooting aid or as a temporary solution to an excessive error rate in the VIC.

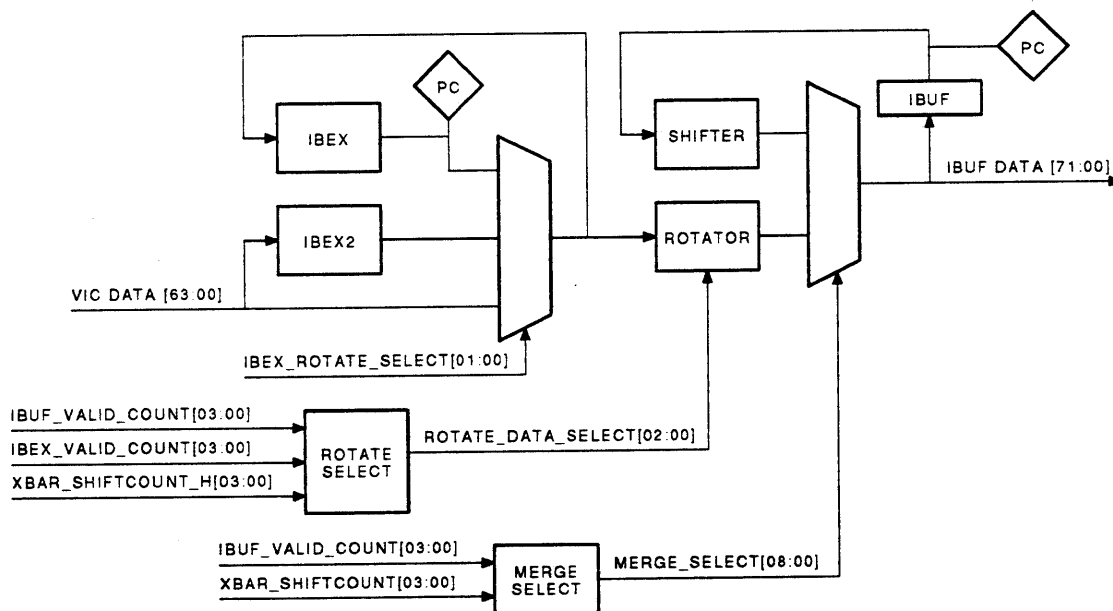
3.2 Instruction Buffer

The instruction buffer logic is physically partitioned across the IBFA and IBFB MCAs on the XBR MCU, with IBFA handling the low nibble and IBFB handling the high nibble. The primary function of the instruction buffer is to present the I-stream to the XBAR for decoding.

The instruction buffer is comprised of six major functional units (Figure 3-6):

IBUF
IBEX
IBEX2
Rotator
Shifter
Merger

Under usual conditions, the instruction buffer is filled in the following manner: The first quadword of VIC data passes through the rotator and merger and is loaded into the IBUF. The second quadword is loaded into IBEX, and IBEX2 receives the third quadword. As the IBUF bytes are decoded, they are shifted out and replenished with bytes from IBEX. The rotator, shifter, and merger align the remaining IBUF bytes and the new IBEX bytes in sequential order in the IBUF. When the IBEX becomes empty, the quadword in IBEX2 is loaded into IBEX and IBEX2 requests more data from the VIC. If no valid data is in the VIC, then a request is passed to the MBox.



MR_X0064_89

Figure 3-6 Instruction Buffer

IBEX and IBEX2 are quadword buffers between the VIC and IBUF. Data can be loaded into IBUF from either of these registers, with valid data in IBEX taken before any valid data from IBEX2. Data taken from either of these sources is rotated by the rotator to provide the correct byte position in IBUF.

As the data in IBUF is consumed, decoded bytes are shifted out and replenished with new valid data from IBEX, IBEX2, or the VIC. The shifter provides this function by shifting the decoded bytes out and shifting remaining bytes down into vacant lower byte positions.

The merger ties the functions of the rotator and shifter together by replenishing IBUF with valid data from both the shifter and rotator.

Control for the instruction buffer depends primarily on three signals:

- `IBUF_VALID_COUNT_H[03:00]` is the number of valid bytes in the 9-byte IBUF.
- `IBEX_VALID_COUNT_H[03:00]` is the number of valid bytes in the 8-byte IBEX.
- `XSCA_SHIFTCOUNT_H` is the number of IBUF bytes the XBAR has decoded in a single cycle.

These three signals direct the flow of data through the rotator and shifter, and they initiate requests to the VIC and MBox for replenishment.

The shift count (`XSCA_SHIFTCOUNT_H[03:00]`) is the number of bytes the XBAR has decoded and directs the shifter to shift decoded bytes out of IBUF.

The IBUF valid count (`IBUF_VAL_H[03:00]`) selects rotate and merger data for replenishing IBUF. This valid count is calculated by subtracting the XBAR shift count from the previous IBUF valid count.

The IBEX valid count (`IBEX_VALID_COUNT_H[02:00]`) is the number of valid bytes in the 8-byte IBEX. This valid count is used by the rotator to select valid bytes to be loaded into IBUF. When no valid bytes are in IBEX (`IBEX_VALID_COUNT_H = 0`), data is loaded into IBUF from IBEX2, the VIC, or a request is made to the MBox.

3.2.1 IBEX2

IBEX2 is an 8-byte buffer between the VIC and IBEX. IBEX2 receives and outputs data eight bytes per cycle. IBEX2 receives input data from the VIC and outputs to either IBEX or IBUF. Which unit receives IBEX2 data depends on the valid counts of IBEX and IBUF. For example, if IBEX is empty (`IBEX_VALID_COUNT_H = 0`) and IBUF is full (`IBUF_VALID_COUNT_H = 9`), then the eight bytes of IBEX2 data is loaded into IBEX. This load is accomplished by selecting IBEX2 data at the rotator multiplexer (`IBEX_ROTATE_SELECT_H[01:00] = 1`). The data is not passed through the rotator because the IBUF valid count indicates that IBUF is full.

Validating IBEX2 requires only a single bit, as the buffer is either full or empty. `IBEX2_VALID_H` is asserted when valid data is in IBEX2. To load data into IBEX2 and assert the IBEX2 valid bit, IBEX must contain valid data (`IBEX_EMPTY_TB_H` is negated) and a VIC match must occur (`VIC_MATCHA_H` or `VIC_MATCHB_H` asserted).

When IBEX2 contains valid data (IBEX2_VALID_H asserted), the prefetch PC cannot be incremented past the next sequential quadword it is addressing. A hold signal (IBUF_HOLD_PREFETCH_PC) is asserted to stop incrementing the PC. This signal is asserted when IBEX2_VALID_H is asserted and IBEX_EMPTY_H is negated. The PC is held because it must continue to point to the next quadword the instruction buffer will receive. When IBEX2 becomes empty, the quadword that the prefetch PC is addressing is loaded into IBEX2 and the PC is incremented.

3.2.2 IBEX

IBEX, unlike IBEX2, can contain valid data in any of its byte locations. This allows IBEX to replenish IBUF with any number of bytes as they are decoded by XBAR. Data that is loaded into IBEX is sent to the rotator of the instruction buffer. IBEX data is passed to the rotator in a low-to-high byte order, providing IBUF with a sequential I-stream.

IBEX receives data from either IBEX2 or the VIC. If IBEX2 is valid, then it replenishes IBEX. If IBEX2 is not valid, then the VIC (if valid) replenishes IBEX.

3.2.2.1 IBEX Valid Count

Because IBEX replenishes IBUF with as many bytes as needed, it can contain any number of valid bytes between eight and zero.

The IBEX valid count (IBEX_VALID_COUNT_H[03:00]) is calculated by subtracting the valid count, of the previous cycle, from the number of bytes loaded into IBUF (MERGE_COUNT_H[03:00]).

When IBEX contains valid data, and the IBUF valid count (IBUF_VC_H[03:00]) is less than nine, IBEX data is selected at the rotator multiplexer. The IBEX data is placed on the input of the rotator. The rotator rotates the bytes needed to fill IBUF (ROTATE_SELECT_H[03:00]) and passes the rotated IBEX bytes to the merger. The merger passes shift data and rotate data to IBUF and produces a merge count (MERGE_COUNT_H[03:00]). The previous IBEX valid count is subtracted from the merge count to produce a new IBEX valid count.

When the IBEX valid count is decremented to zero, IBEX_EMPTY_H is asserted and directs IBEX_ROTATE_SELECT_H[01:00] to select VIC data or IBEX2 data at the IBEX rotator multiplexer. When either of these sources is supplying data to IBUF, and IBUF does not require all eight bytes, the remaining bytes are stored and validated in IBEX.

3.2.3 Rotator

The rotator aligns the bytes from IBEX, IBEX2, or the VIC so that they are correctly placed in IBUF. This logic consists of a bank of multiplexers that can select any of the eight bytes and insert them into IBUF in the correct byte position.

The rotator is controlled by the IBEX valid count, the IBUF valid count, and the XBAR shift count. The rotator receives data from the IBEX rotator multiplexer, which selects IBEX data, IBEX2 data, or VIC data to be input to the rotator. The source of the data at the IBEX rotator multiplexer is selected by IBEX_ROTATE_SELECT_H[01:00]. This select signal is derived from IBEX_EMPTY_H (asserted when IBEX contains no valid data), IBEX2_VALID_H (the single valid bit for IBEX2), and READ_VIC_DATA_H (the read enable signal for the VIC).

IBEX_ROTATE_SELECT_H[01:00] selects valid data from IBEX before selecting IBEX2 or VIC data. If the IBEX data is not valid, IBEX2 data is selected (if valid) before the VIC data. If neither IBEX nor IBEX2 data is valid, VIC data is selected.

Figure 3-7 shows the rotator supplying IBEX data to IBUF to replenish four bytes that have been decoded and shifted out.

- During cycle 1, four bytes, including the opcode, are decoded and shifted out of IBUF. IBEX contains seven valid bytes and IBEX2 is valid.
- During cycle 2, IBUF bytes 4 through 9 are shifted into bytes 0 through 4, and IBEX bytes 1 through 4 are rotated into the empty IBUF bytes. IBUF is full, IBEX contains three valid bytes, and IBEX2 is valid.

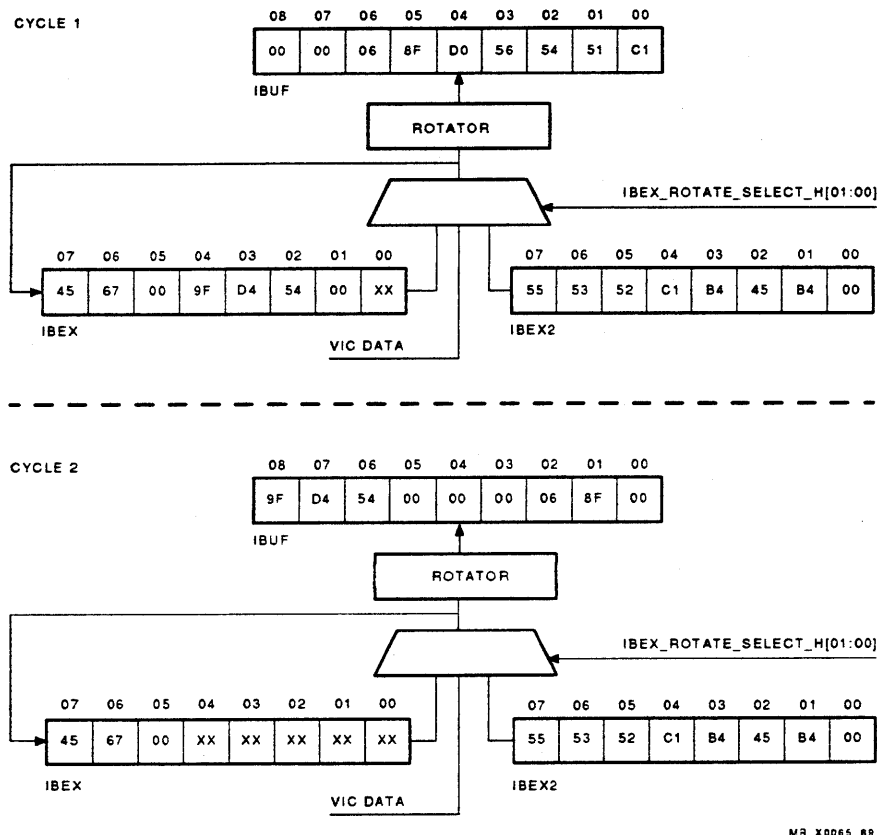


Figure 3-7 Rotator

3.2.3.1 IBEX2 Rotate Data

When IBEX is empty, IBEX2 (if valid) replenishes IBUF with an I-stream. Figure 3-8 shows the loading of IBEX2 data into IBUF. The quadword that IBEX2 contained is partially loaded into both IBUF and IBEX. In Figure 3-8, the initial IBUF valid count is nine, the IBEX valid count is zero, and IBEX2 is valid.

- During cycle 1, seven bytes of IBUF data are decoded and shifted out of IBUF. The new IBUF valid count is two.
- During cycle 2, IBEX2 data is selected at the rotator multiplexer. Seven bytes of IBEX2 data are loaded into IBUF; one byte is loaded into IBEX. The IBEX2 valid bit (IBEX2_VALID_H) is negated, the IBEX valid count (IBEX_VALID_COUNT_H[03:00]) is one, and the IBUF valid count (IBUF_VC_H[03:00]) is nine.

When the IBEX2 valid count is negated, a read request is sent to the VIC and the prefetch PC is incremented (IBFA_HOLD_PREFETCH_PC_H is negated).

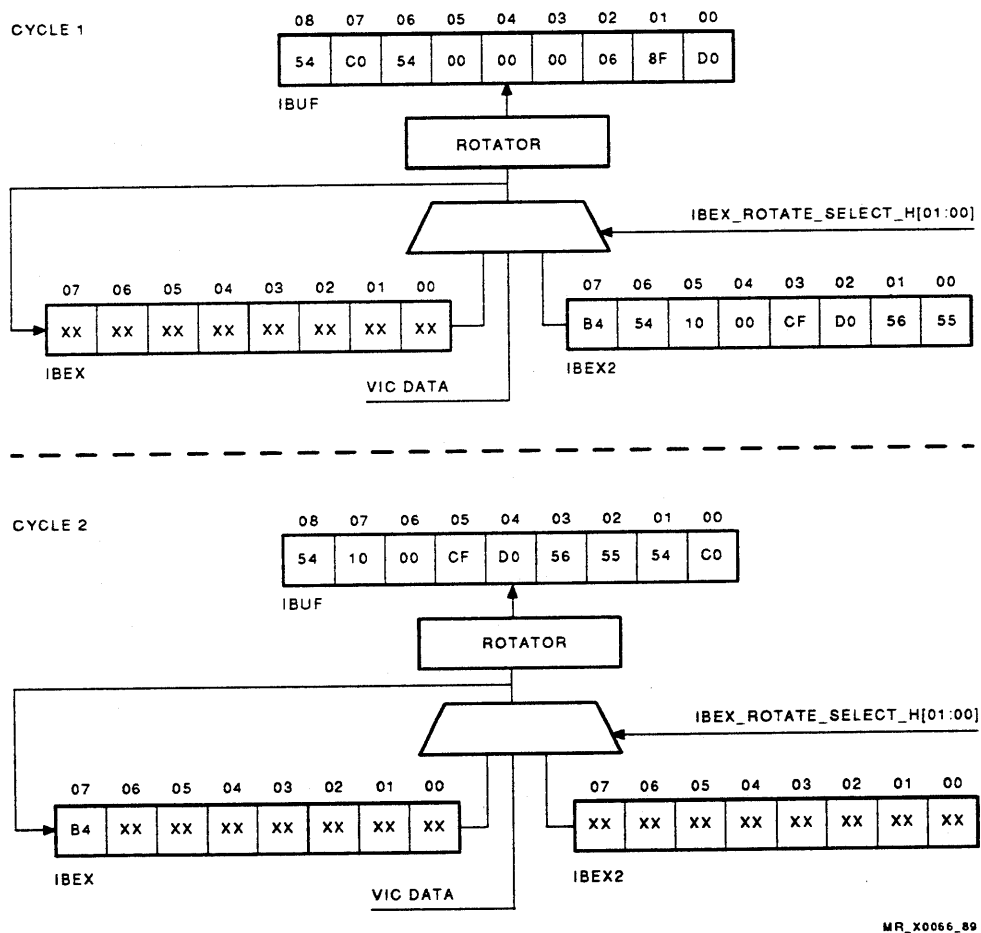


Figure 3-8 IBEX2 Rotate Data

3.2.4 Merger

The merger consists of nine, 8-bit, 2-to-1 multiplexers where each multiplexer replenishes one of the nine bytes of the IBUF. Either rotate data or shift data can be selected as the input, with IBUF as the destination of the output.

Figure 3-9 shows a simplified block diagram of the merger. Each merger multiplexer receives SHIFT_DATA_H and ROTATE_DATA_H for inputs. MERGE_SELECT_H[08:00] selects each byte to be loaded into IBUF.

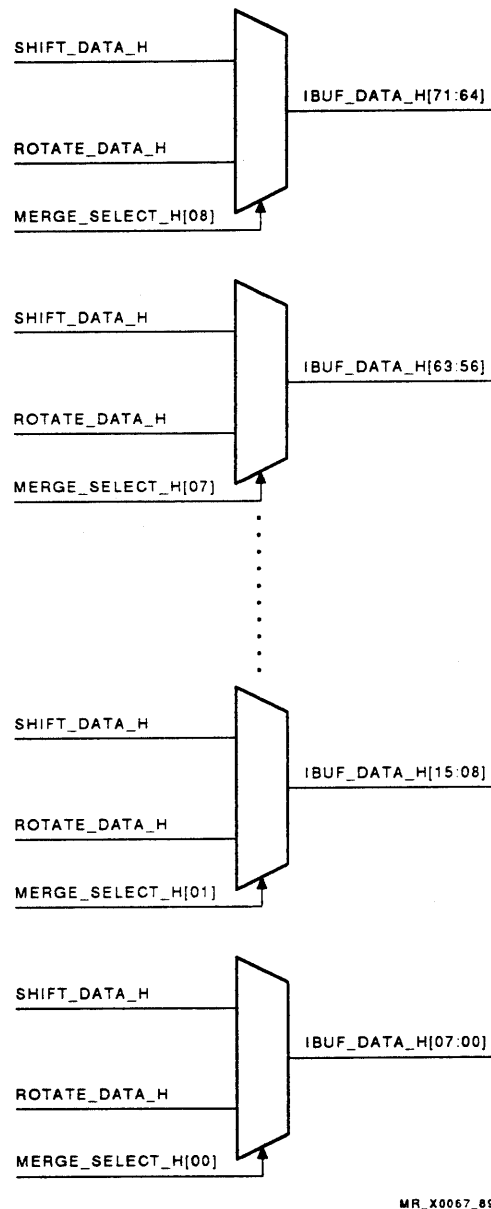
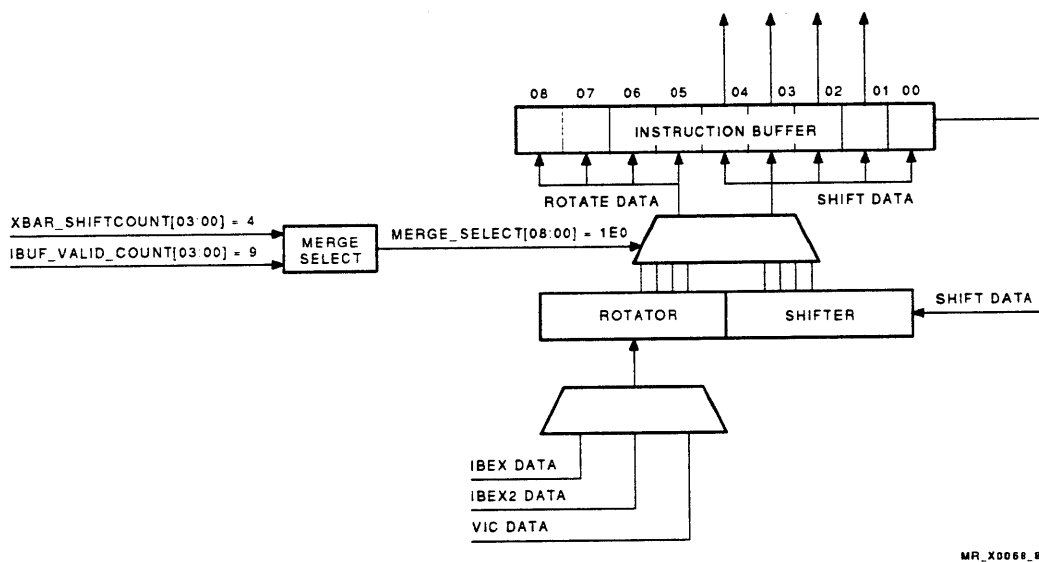


Figure 3-9 Simplified Merger

Merge select is calculated by subtracting the XBAR shift count from the IBUF valid count. The select logic outputs a 9-bit field (MERGE_SELECT[08:00]), with each bit controlling one of the multiplexers of the merger. A logical one, in the field, selects rotate data while a logical zero selects shift data.

Figure 3-10 shows the merger supplying IBUF with data from both the shifter and the rotator. With a XBAR shift count of 4, bytes 1 through 4 are shifted out of IBUF and bytes 0 (opcode) and 5 through 8 are inputs to the merger from the shifter. The merge select logic receives the XBAR shift count and the IBUF valid count, and outputs the 9-bit field that selects shifter bytes 1 through 4 and rotator bytes 5 through 8. The opcode byte, because it is not shifted, is recirculated through the shifter and merger until the instruction is completely decoded.

Table 3-1 is a sample of the merge select output. All outputs are based on an initial IBUF valid count of nine with different shift counts.



MR_X0068_09

Figure 3-10 Merger

Table 3-1 Sample Merge Select

IBUF Valid Count	XBAR Shift Count	MERGE_SELECT_H[08:00]
9	9	1FF
9	8	1FE
9	7	1FC
9	6	1F8
9	5	1F0
9	4	1E0
9	3	1C0
9	2	180
9	1	100
9	0	0

3.2.5 Shifter

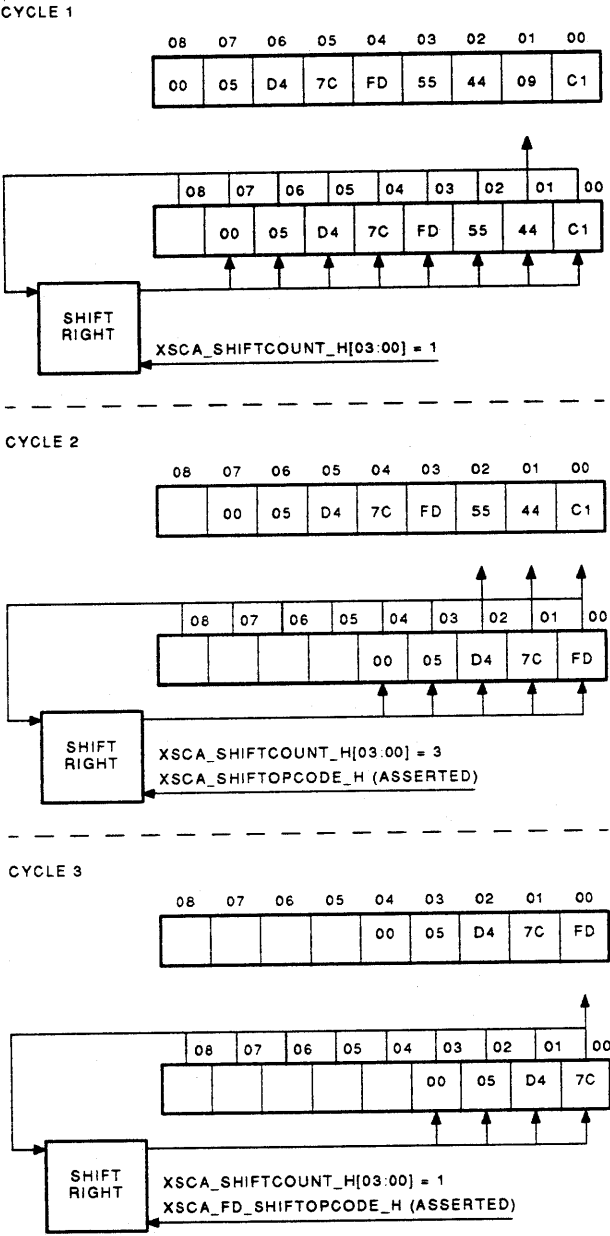
The instruction buffer shifter is responsible for shifting out decoded bytes, holding the opcode byte (byte 0), and realigning the remaining valid bytes in IBUF. XBAR provides three signals that control the shifter:

- **XSCA_SHIFTCOUNT[03:00]** provides the number of IBUF bytes decoded in the last cycle.
- **XSCA_SHIFTOPCODE** shifts the opcode out of byte 0. The signal is received when all of the specifier bytes of an instruction have been decoded.
- **XSCA_FD_SHIFTOPCODE** is asserted when the XBAR detects an FD (extended opcode) in the opcode byte of IBUF. The FD byte is shifted out of byte 0 and the second byte of the opcode is shifted into byte 0.

NOTE

Figure 3-11 provides examples of the state of IBUF after the completion of three cycles. For simplification, no new I-stream is placed in IBUF at the completion of each cycle.

- **Cycle 1** — XBAR decodes one byte and produces a shift count of one. Byte 1 is shifted out of IBUF and the remaining bytes are shifted down to replace it.
- **Cycle 2** — XBAR decodes the two remaining bytes of the instruction, asserts **XSCA_SHIFTOPCODE_H**, and produces a total shift count of three. The opcode and the two remaining bytes of the instruction are shifted out, with the remaining bytes in IBUF again being shifted down to replenish those shifted out.
- **Cycle 3** — XBAR decodes the FD in the opcode byte of IBUF, asserts **XSCA_FD_SHIFTOPCODE_H**, and produces a total shift count of one. The FD in the opcode byte is shifted out and the remaining bytes are shifted down one byte.



MR_X0069_89

Figure 3-11 IBUF Data

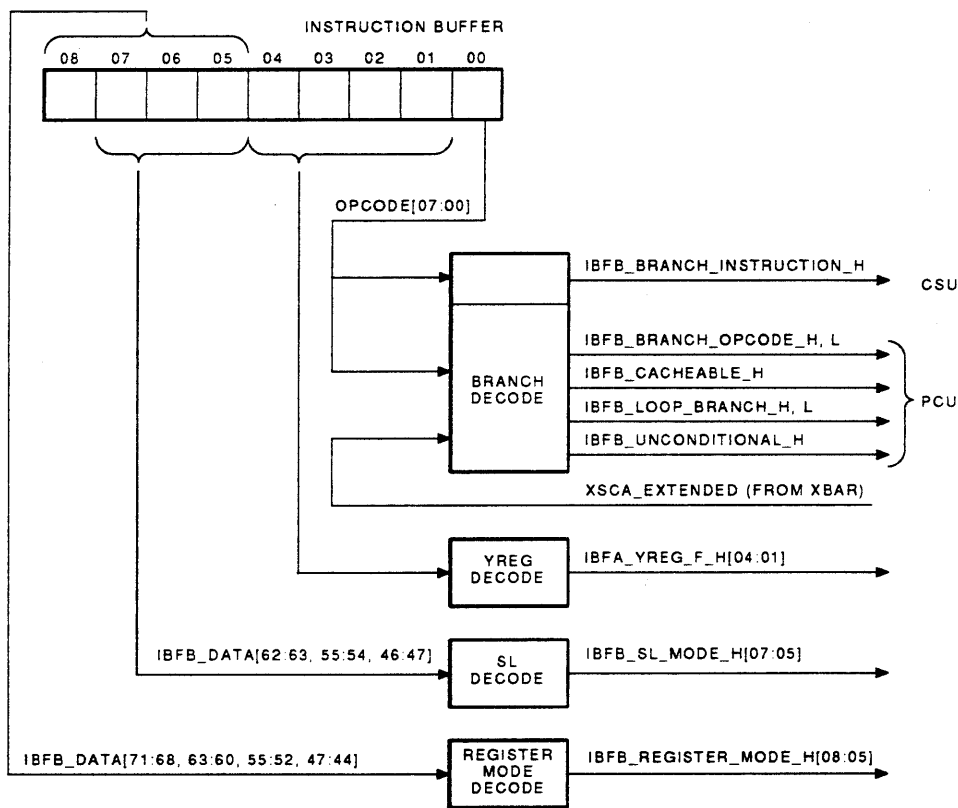
3.2.6 IBUF

IBUF holds nine bytes of I-stream that are passed to XBAR for decoding. Byte 0 always contains the opcode of the instruction that is being decoded. A copy of this opcode is sent to the EBox, OPU, XBAR, and the branch prediction logic.

Specifier bytes are parsed by XBAR and handled by the XBAR decode units while the opcode bytes remain in IBUF until the instruction has been completely decoded.

3.2.6.1 Simple Decode

As the I-stream is presented to XBAR for decoding, logic in the instruction buffer performs a small amount of decode. The decoded I-stream provides branch information for the PCU and CSU. Register, short literal, and YREG information is also decoded and passed to XBAR. Figure 3-12 shows a block diagram of the instruction buffer decode units.



MR_X0070_89

Figure 3-12 Simple Decode

3.2.6.2 Branch Decode

Each time a new opcode is loaded into IBUF, the opcode is decoded to detect a branch instruction. The decoded branch opcode has two destinations, the CSU and the PCU. The CSU receives a single signal (IBFB_BRANCH_INSTRUCTION_H) that indicates a branch instruction is currently being decoded.

The PCU receives more detailed information relevant to the branch. The four signals provide information to aid in directing the PCU and branch prediction cache (BPC) handling. The four signals provided by the branch decode logic are as follows:

- IBFB_BRANCH_OPCODE_H is asserted if the opcode is a branch instruction.
- IBFB_UNCONDITIONAL_H is asserted if the opcode is an unconditional branch.
- IBFB_LOOP_BRANCH_H is asserted for any loop branch instruction, except for an emulated branch. (Loop branches are AOBLEQ, AOBLSS, SOBGEG, SOBGTR, ACBL, ACBW, and ACBB.)
- IBFB_CACHEABLE_H informs the PCU that the branch opcode in IBUF can be cached in the BPC. (Noncacheable branches are RSB, JSB, JMP, CALLG, and CALLS. These instructions do not have normal branch displacement specifiers.)

3.2.6.3 YREG Decode

The low nibble of bytes 1, 2, 3, and 4 are decoded to provide YREG information for XBAR. When all four bits are set, PC addressing is possible.

3.2.6.4 Short Literal Decode

Bytes 5, 6, and 7 are decoded to detect short literal specifiers. If the upper two bits of bytes 5, 6, and 7 are zeros, a short literal specifier is detected. Each bit in the signal corresponds to the byte number in the I-stream of IBUF.

3.2.6.5 Register Mode Decode

Bytes 5 through 8 are decoded to a value of 5. The high nibble of the IBUF I-stream is input to the decoder, producing an output with each bit in the signal corresponding to the byte being decoded (for example, if IBFB_REGISTER_MODE_H[08:05] = 3, then bytes 5 and 6 may contain register specifiers).

3.2.7 Instruction Buffer Parity

Instruction buffer data is byte parity protected and is checked at two locations. The outputs of IBEX and IBUF contain parity detection circuitry.

As data is output from IBEX, a parity check is performed. IBFB receives partial parity from IBFA and performs the check. Detected errors assert IBFB_IBEX_ERROR_H, which asserts IBFB_FETCH_ERROR_H and is forwarded to the EBox as IBOX_FORK_ERROR_H.

The data that is being presented to the XBAR is checked on the output to the instruction buffer. IBFB receives partial parity from IBFA and performs the check. Detected errors assert IBUF_ERROR_H, which is latched to the XBAR as DECODE_ERROR_H. This error is forwarded to the EBox as IBOX_POINTER_ERROR_H.

3.3 Instruction Buffer Interface

The instruction buffer interface is a read-only port to the MBox. Requests are made across this port for the I-stream to be loaded into the VIC. Most requests are for a VIC block (four aligned quadwords).

Figure 3-13 summarizes the instruction buffer interface to the MBox. Table 3-2 describes each signal line.

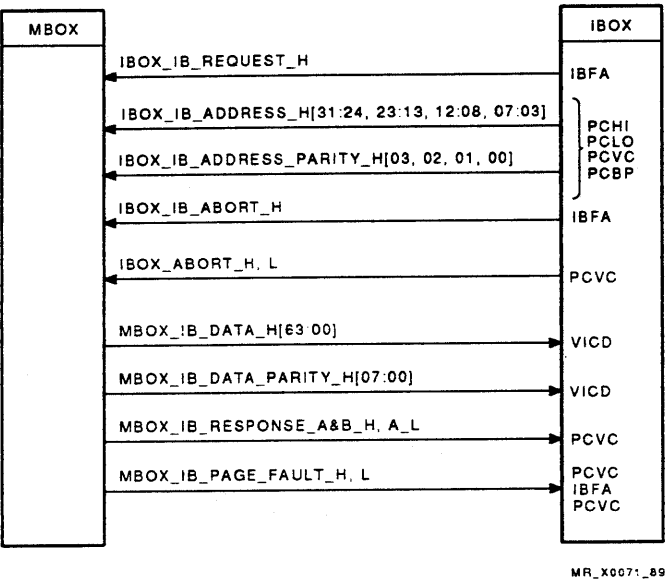


Figure 3-13 Instruction Buffer Interface

Table 3-2 Instruction Buffer Interface Signals

Name	Description
IBOX_IB_REQUEST_H	The request for data from the MBox. Detection of a VIC miss or flush of the VIC asserts this line.
IBOX_IB_ADDRESS_H[31:03]	Address lines for the requested quadword. The address is generated by the PCU and, because all requests are quadword aligned, the lower three bits of the address are assumed to be zero.
IBOX_IB_ADDRESS_PARITY_H[03:00]	Parity protection for the address that the IBox sends.
IBOX_IB_ABORT_H	Asserted to abort a request. The signal aborts a request only if it is in the TB stage of the MBox.
IBOX_ABORT_H	Overriding abort signal for the interface. When asserted, it asserts IBOX_IB_ABORT. An EBox flush or branch prediction unwind asserts this signal. This signal also aborts OPU port transactions.
MBOX_IB_DATA_H[63:00]	Returning quadwords requested by the IBox.
MBOX_IB_DATA_PARITY_H[07:00]	Byte parity for the quadword being returned by the MBox.
MBOX_IB_RESPONSE	Informs the IBox that data will be returned in the next cycle. The signal is negated late in the cycle that the last quadword is being returned.
MBOX_IB_PAGE_FAULT_H, L	Asserted to inform the IBox that the requested data has page faulted in the MBox. The instruction buffer and the XBAR receive this signal and inform the EBox of the page fault if decode of the data is attempted.

3.3.1 Instruction Buffer Requests

Figure 3-14 shows an instruction buffer request that is honored, with three quadwords written to the VIC.

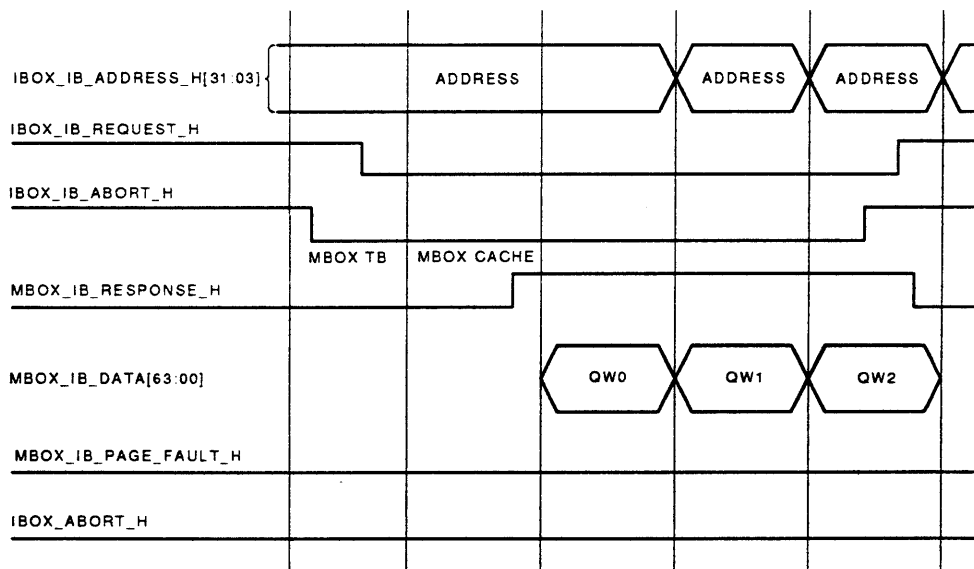
Usually, `IBOX_IB_ADDRESS_H[31:03]`, `IBOX_IB_REQUEST_H`, and `IBOX_IB_ABORT_H` are asserted. In response to these signals, the MBox latches the address, and detects and aborts the request each cycle.

A request is initiated by negating the abort signal (`IBOX_IB_ABORT_H`) early in the cycle after the address has been latched in the MBox. This signal is negated when `REQUEST_CONDITION_H` is asserted by `IBEX_EMPTY_H` asserted, and `IBEX2_VALID_H`, `VIC_MATCHA_H`, and `VIC_MATCHB_H` are all negated. The MBox detects the request early in this same cycle and the IBox lowers the request signal.

The request is detected in the MBox translation buffer (TB) stage. In this stage, the MBox latches the request and address, and arbitrates for the TB. When control of the TB is received, a TB lookup and a validation of the translation is performed. Figure 3-14 shows the request in the TB stage for a single cycle. Figure 3-14 is a best case example because TB arbitration could cause several cycles of delay if the MBox was honoring other requests. The instruction buffer port has the lowest priority of all ports that arbitrate for the TB.

Cache arbitration results in a cache hit or miss. A cache hit asserts `MBOX_IB_RESPONSE_H`, late in the cycle. In the following cycle, the MBox returns the requested quadword (`MBOX_IB_DATA[63:00]`) and subsequent quadwords to the end of the block.

Late in the same cycle that the last quadword is being returned, the MBox response signal is negated. In the same cycle, the abort signal then the request signal are asserted again.



MR_X0072_69

Figure 3-14 Instruction Buffer Request

3.3.1.1 Aborting Requests

Two abort signals are associated with the instruction buffer interface:

IBOX_IB_ABORT_H
IBOX_ABORT_H

IBOX_IB_ABORT_H, when asserted, attempts to abort the request. This signal is not asserted for a request that has entered the cache stage because of the penalties the MBox must pay to clean up. The MBox returns the first quadword and then aborts. The quadword is not written to the VIC because IBOX_IB_ABORT_H, asserted, negates VIC_DATA_WRITE_H.

IBOX_ABORT, when asserted, unconditionally aborts instruction buffer requests. This signal is asserted by an EBox flush (EBOX_FLUSH_H[02:00]) due to an error, interrupt, or branch unwind situation.

3.3.1.2 Page Faults

When a page fault is detected, the MBox asserts MBOX_IB_RESPONSE_H and MBOX_IB_PAGE_FAULT_H in the same cycle. The page fault signal is sent to the instruction buffer and the XBAR. The XBAR notifies the EBox of page faults if decode of the data is attempted. If the VIC is flushed, the data is not accessed and the MBox clears the page fault and associated registers.

This chapter describes the two functional units representing the instruction decode pipeline stage: the XBAR and branch prediction logic.

4.1 XBAR

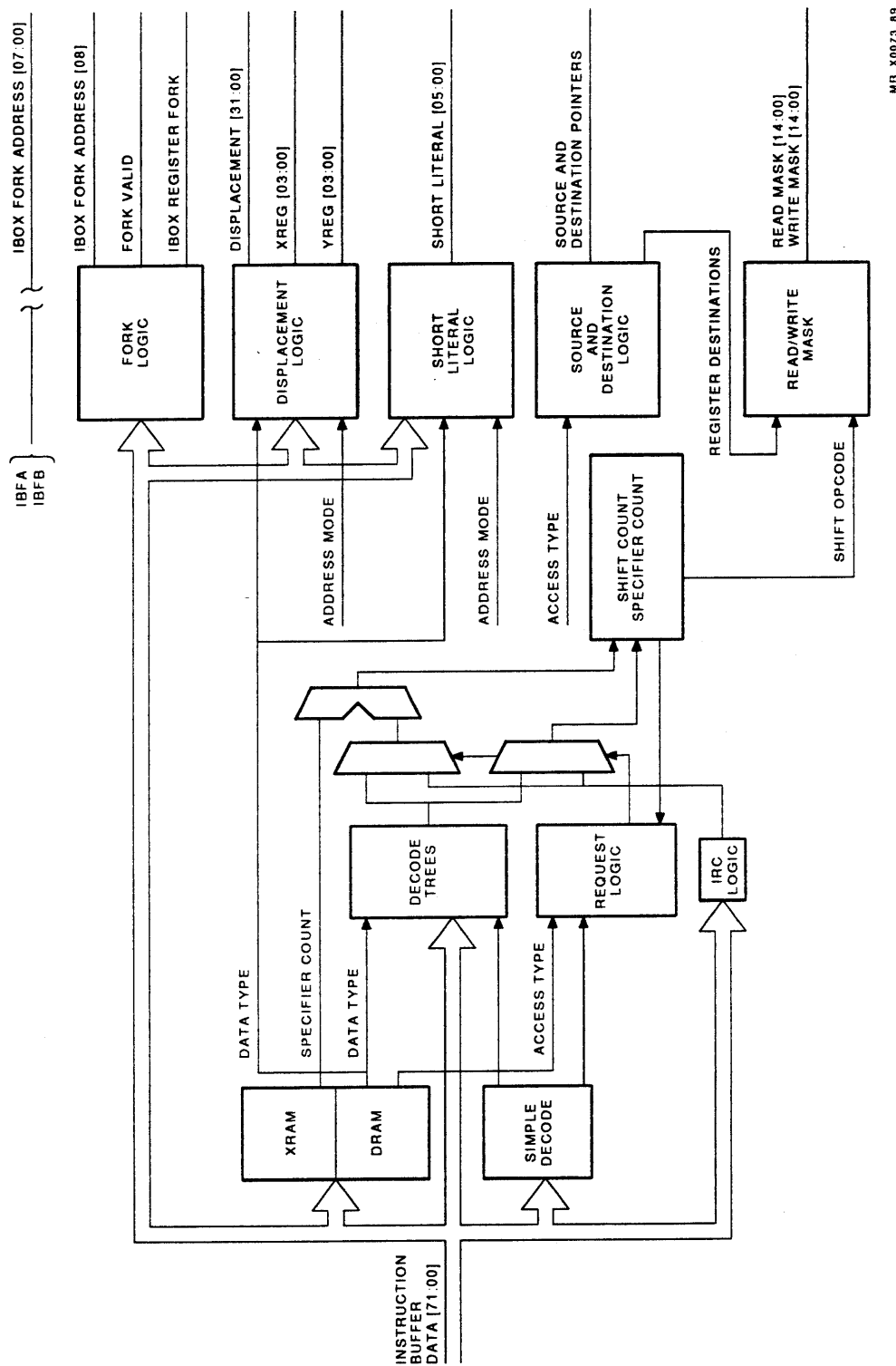
The XBAR decodes the individual macroinstructions and determines the following:

- Number of instruction specifiers
- Destination of each decoded specifier
- Number of specifiers decoded in a single cycle

Each cycle, the XBAR attempts to decode and pass specifier data to the specifier handling units of the IBox: SLU, CSU, and FPL. Because each unit processes a unique specifier type, the number of combinations of specifiers the XBAR can successfully decode each cycle is restricted. The XBAR can decode up to three specifiers in a single cycle. The specifiers may be all register mode, two register mode and one short literal or complex, or one of each (register, complex, or short literal) in any order. Complex specifiers are branch displacements and all specifiers other than short literal mode and register mode specifiers. The XBAR concurrently processes specifiers of a single instruction only.

The XBAR is also responsible for generating read and write register masks for conflict checking by the CSU. In special cases where conflicts occur in a single instruction, the XBAR redirects register specifiers to the CSU for processing.

Figure 4–1 is a block diagram of the XBAR. As shown in Figure 4–1, the XBAR receives 72 bits of instruction buffer data and distributes it throughout the associated MCAs (XSCA, XDTA, and XDTB). The following list introduces the major XBAR functional blocks.



MR_X0073_89

Figure 4-1 XBAR Block Diagram

- **DRAM and XRAM** — The DRAM and XRAM decode the opcode and produce the following:
 - **Specifier count** — The total number of specifiers the instruction contains.
 - **Specifier data type** — The data type of each specifier in the instruction (byte, word, longword, etc.).
 - **Specifier access type** — The access type of each specifier in the instruction (read, write, modify, etc.).

The XRAM specifier count output controls the instruction buffer shifter by providing a running total of specifiers decoded each cycle and by informing the instruction buffer to load a new opcode into the opcode byte (byte 0) of the IBUF when all of the specifiers of an instruction have been decoded by the XBAR.

The specifier attribute outputs (data type and access type) are distributed throughout the XBAR to the individual data path units (displacement, short literal, and source and destination logic units) to validate outputs to the specifier handlers. These outputs are also used to generate a read and write mask and to detect intra-instruction read conflicts (IRC).

- **Simple decode logic** — Simple decode logic decodes the I-stream and provides the addressing mode (register mode, absolute mode, and so on) for the first four bytes of the instruction. The addressing mode outputs are used to validate the data path unit outputs and by the decode tree logic and request logic when generating their shift count and specifier count outputs.
- **Decode tree and request logic** — Each cycle, the decode tree logic and request logic perform a parallel operation that determines the number of specifiers the XBAR will decode and the number of specifier bytes that will be decoded.

The decode trees produce shift count and specifiers decoded outputs based on the decoded addressing modes and data types and by decoding the I-stream. There are 14 pairs of output (one shift count and one number of specifiers decoded) generated each decode cycle.

Request logic decodes addressing modes, specifier access types, and the specifier count. It produces an output that selects the correct decode tree output for the instruction currently being decoded.

For example, for the instruction ADDL2 R0, R1, the request logic determines that there are two specifiers, the access types are read and modify, and there are two register specifiers in the instruction. The request logic selects R2 (R2_SC_H[02:00] shift count and R2_N_H[001:00] specifiers decoded count) from the request logic. The R2 output provides the correct shift count to the instruction buffer and the correct number of specifiers decoded to the specifier count logic.

The decode tree logic has produced 14 outputs for the ADDL2 R0, R1 with only one of the outputs producing the correct specifiers decoded and the correct shift count. The R2 output is produced by logic that produces only an output that is based on an instruction that contains two specifiers. In the same cycle that the R2 output is selected, outputs based on different numbers and structures of specifiers are produced. If the R3BW output were selected, the shift count and number of specifiers decoded would be equal to those based on decoding three specifiers, with the third specifier being a branch word displacement.

The parallel operation of the decode tree logic and the request logic is implemented to reduce the cycle time required to decode the specifiers of an instruction (to determine their addressing modes, access types, and data types) and then calculate the number of bytes that will be decoded and the number of specifiers that can be decoded.

- **Displacement logic** — The displacement logic receives the complex specifiers (other than register and short literal specifiers) and, when a specifier is valid, passes up to 32 bits of displacement to the CSU of the OPU MCU.
- **Short literal logic** — The short literal unit decodes the short literal specifiers and passes six bits of short literal data to the SL specifier handler in the OPU for expansion.
- **Source and destination logic** — The source and destination logic outputs source 1 pointers, source 2 pointers, destination pointers, and a read and write register field to the OPU specifier handlers and to the read/write mask logic. The source and destination pointers define the operand addresses (register number or memory location) of the specifiers. The read/write masks record the reading and writing of registers by the EBox during the execution of an instruction.
- **IRC logic** — The IRC logic detects intra-instruction read conflicts. These conflicts occur when there are read conflicts in the specifiers of an instruction. That is, an instruction specifier directs the EBox to read R0 and a subsequent specifier directs the IBox to autoincrement or autodecrement R0. When IRCs occur, the IBox decodes each specifier separately and does not update the EBox GPRs until instruction execution is completed by the EBox.

4.1.1 DRAM

The XBAR DRAM is implemented as a functional logic block in the XSCA MCA. That is, the DRAM is comprised of logic gates instead of a RAM structure, as with previous VAX systems.

The inputs to the DRAM logic are the instruction opcode, the extended opcode bit, and the specifiers remaining in the instruction currently being decoded. Based on these inputs, the number of specifiers the instruction contains and the specifier attributes (access type and data type of each specifier) are determined. The outputs of the DRAM are passed to the other functional units of the XBAR and are used to validate the specifier handler outputs and also are used in the specifier count and shift count logic. Figure 4-2 shows a block diagram of the DRAM logic.

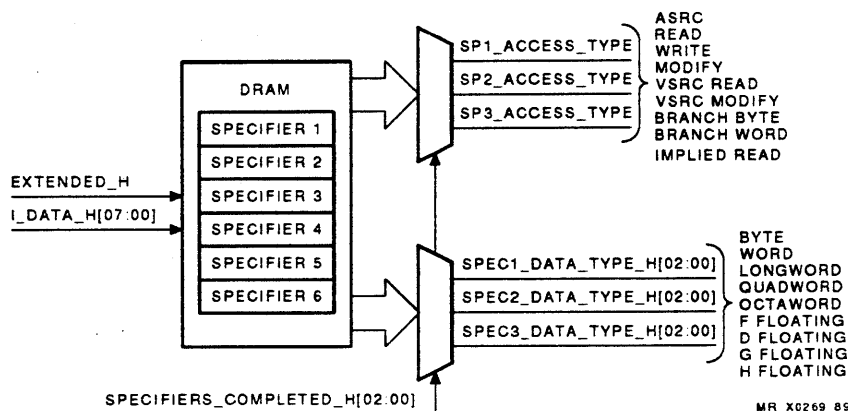


Figure 4-2 DRAM Logic

DRAM data type logic decodes the opcode (I_DATA_H[07:00]) and the extended opcode bit (EXTENDED_H), and produces SP1_DATA_TYPE_H[02:00] through SP6_DATA_TYPE_H[02:00]. Each output describes the data type of the specifier it represents. (For example, SP1_DATA_TYPE_H[02:00] describes the data type of the first specifier of the instruction being decoded.)

Each cycle, SPECIFIERS_COMPLETED_H[02:00] selects up to three of the data type outputs to be used by the XBAR decode units. SPECIFIERS_COMPLETED_H[02:00] is the number of specifiers that have been decoded in the instruction that is currently being decoded.

DRAM access type outputs are also based on the decode of the instruction opcode and the extended opcode bit. The access type of each specifier (for example, SPEC1_READ_H and SPEC1_WRITE_H) is placed on the DRAM output multiplexers and is also selected by SPECIFIERS_COMPLETED_H[02:00].

4.1.2 XRAM

The XRAM is physically structured like the DRAM and is contained on XDTB. The XRAM decodes the instruction opcodes and the extended bit, and produces five fields:

- SPECIFIER_COUNT_H[02:00] is the total number of specifiers the instruction contains.
- IMPLIED_MASK_H is asserted when a character string instruction is detected.
- VSRC_FORK_MODIFY_H is asserted when a variable length bit field instruction is decoded.
- SUSPEND_H is asserted because the instruction that is currently being decoded does not leave predictable results in memory or registers. XBAR continues to decode specifiers until it encounters a complex specifier and then suspends (stops all decoding) and waits for the EBox to restart it (EBOX_UNsuspend_H). Some of the instructions that assert SUSPEND_H are as follows:

ADAWI (add aligned word interlocked)
 ADDP4 (add packed 4-operand)
 EDITPC (edit packed to character string)

- STOP_H is similar to SUSPEND_H. STOP_H is asserted by certain instructions after all of the specifiers of the instruction have been decoded. When STOP_H is asserted, the XBAR does not process any specifiers until the EBox asserts EBOX_UNsuspend_H. Some instruction that assert STOP_H are as follows:

HALT
 CASEx (case byte, word, longword)
 CHMx (change mode)

4.1.3 Simple Decode Logic

The simple decode logic receives the high nibble of the first four bytes of instruction buffer data and `I_YREG_F[04:01]` (used to determine which bytes are relative addressing mode or immediate addressing mode) from the instruction buffer simple decode logic. Outputs defining the addressing modes of bytes 1 through 4 of the I-stream are produced from these inputs. Figure 4-3 shows the inputs and outputs of the simple decode logic.

The simple decode logic also decodes the I-stream to produce `CASE_H[01:00]`. This 2-bit signal defines where in the I-stream the complex specifier is located. `CASE_H[01:00]` is valid when any addressing mode other than short literal or register is detected. Table 4-1 describes the four case outputs.

`CASE_H[01:00]` is input to the decode tree logic to produce shift counts for complex specifiers. Because `CASE_H[01:00]` determines only the location of the complex specifier, it must be decoded with the addressing mode to produce the length of the specifier for the shift count logic.

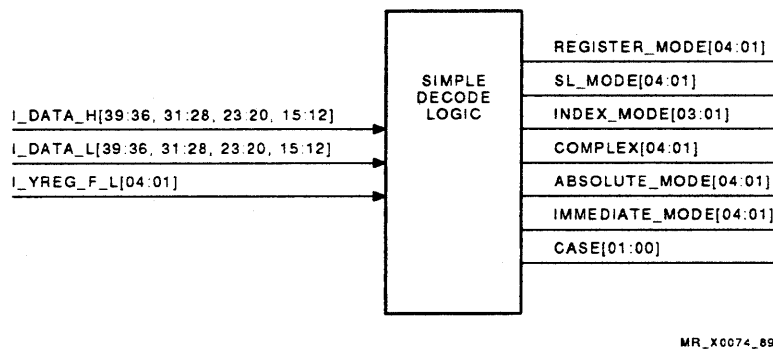


Figure 4-3 Simple Decode Logic

Table 4-1 Case Output

<code>CASE_H[01:00]</code>	Byte Position of Complex Specifier
00	3
01	2
10	4
11	1

4.1.4 Decode Tree Logic

Each cycle, the decode tree logic decodes the I-stream and produces 14 unique outputs. One of the outputs is selected, at the end of a cycle, to produce a shift count and the number of specifiers decoded. Figure 4-4 shows the inputs and outputs of the decode tree logic.

DECODE TREES		
	R1	R1_SC_H[02:00]
		R1_SC_L[02:00]
		R1_N_H[00]
	R1BB	R1BB_SC_H[00]
		R1BB_SC_L[00]
		R1BB_N_H[00]
I_DATA_H[38:37, 31:29, 23:21, 15:13]		
I_DATA_L[39:37, 31:29, 23:21, 15:13]		
I_VALID_H[08:01]	R1BW	R1BW_SC_L[01]
I_VALID_L[08:01]		R1BW_N_H[00]
REGISTER_MODE_H[07:03, 01]	R1I	R1I_N_H[00]
REGISTER_MODE_L[08:01]	R2	R2_SC_H[02:00]
SL_MODE_H[01]		R2_N_H[01:00]
SL_MODE_L[07:01]	R2BB	R2BB_SC_H[01:00]
INDEX_MODE_H[02:01]		R2BB_N_H[01:00]
INDEX_MODE_L[02:01]	R2BW	R2BW_SC_H[01:00]
ABSOLUTE_MODE_H[04:01]		R2BW_N_H[01:00]
ABSOLUTE_MODE_L[03:01]	R2I	R2I_SC_H[00]
IMMEDIATE_MODE_L[03:01]		R2I_N_H[01:00]
CASE_H[01:00]	R2R	R2R_SC_H[02:00]
CASE_L[01:00]		R2R_N_H[01:00]
SP2_DATATYPE_H[02:00]	R3	R3_SC_H[03:00]
SP1_DATATYPE_H[02:00]		R3_N_H[01:00]
IRC_L	R3BB	R3BB_SC_H[01:00]
OPU_STALL_H		R3BB_N_H[01:00]
OPU_STALL_L	R3BW	R3BW_SC_H[02:00]
SL_STALL_H		R3BW_N_H[01:00]
SL_STALL_L	R3XR	R3XR_SC_L[03:00]
		R3XR_N_H[01:00]

MR_X0075_89

Figure 4-4 XBAR Decode Trees

Each of the 14 outputs contains 2 fields. The two fields determine the number of specifiers decoded and a shift count for the XBAR:

- **Rx_SC_H[02:00]** is the shift count. The shift count is the number of IBUF bytes that will be decoded when this tree output is selected.
- **Rx_N_H[01:00]** is the specifier count. This count represents the number of specifiers that is decoded when this tree output is selected.

The decode tree logic receives the following inputs:

- **I-stream** — The high nibble of bytes 1 through 4 (I_DATA_H[38:37, 31:29, 23:21, 15:13]) and their valid bits (I_VALID_H[04:01]). The valid signals are received from the IBUF valid count.
- **Addressing modes** — The addressing mode logic decodes the I-stream and supplies the addressing mode.
- **Case** — Indicates the location of the complex specifier in the I-stream.
- **Data type** — The DRAM decodes the opcode and outputs the specifier data types to the tree logic.
- **IRC** — IRC decode logic passes this signal when an IRC is detected in the I-stream currently being decoded.
- **Stall** — If the CSU or SLU is stalled, the decode tree logic is notified. The stall signal from either of these two units (OPU_STALL_H and SL_BUSY_STALL) influences decode tree output. If the SLU is stalled, the decode tree cannot produce an output that selects data to be sent to the SLU of the OPU.

Figure 4-5 shows a block diagram of the R2BW decode tree. This example of the tree logic is used because it is one of the less complex.

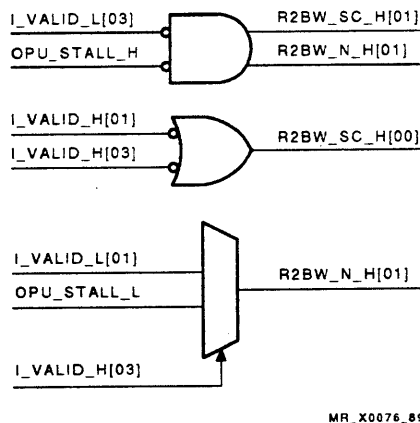


Figure 4-5 R2BW Decode Tree

The outputs of the R2BW tree logic are based on an I-stream that contains two valid specifiers, with the second specifier being a word displacement of a branch instruction.

As shown in Figure 4–5, I_VALID_H[03] (IBUF valid bit for byte 3) selects R2BW_N_H[00] from I_VALID_L[01] or OPU_STALL_L (the stall signal for the CSU). The valid bits for IBUF bytes 1 and 3 of the IBUF are ORed to produce R2BW_SC_H[00], and the valid bit for IBUF byte 3 and the CSU stall signal produce R2BW_SC_H[01] and R2BW_N_H[01].

This tree logic output would be selected if the XBAR is decoding an ACBW R1, R2, R3, displacement word and the first two specifier bytes of the instruction had been decoded in a previous cycle. The output would be valid because the XBAR is decoding two specifiers, with the second specifier being a branch word displacement. The output for this decode cycle of the instruction would be R2BW_SC_H = 2 and R2BW_N_H = 2.

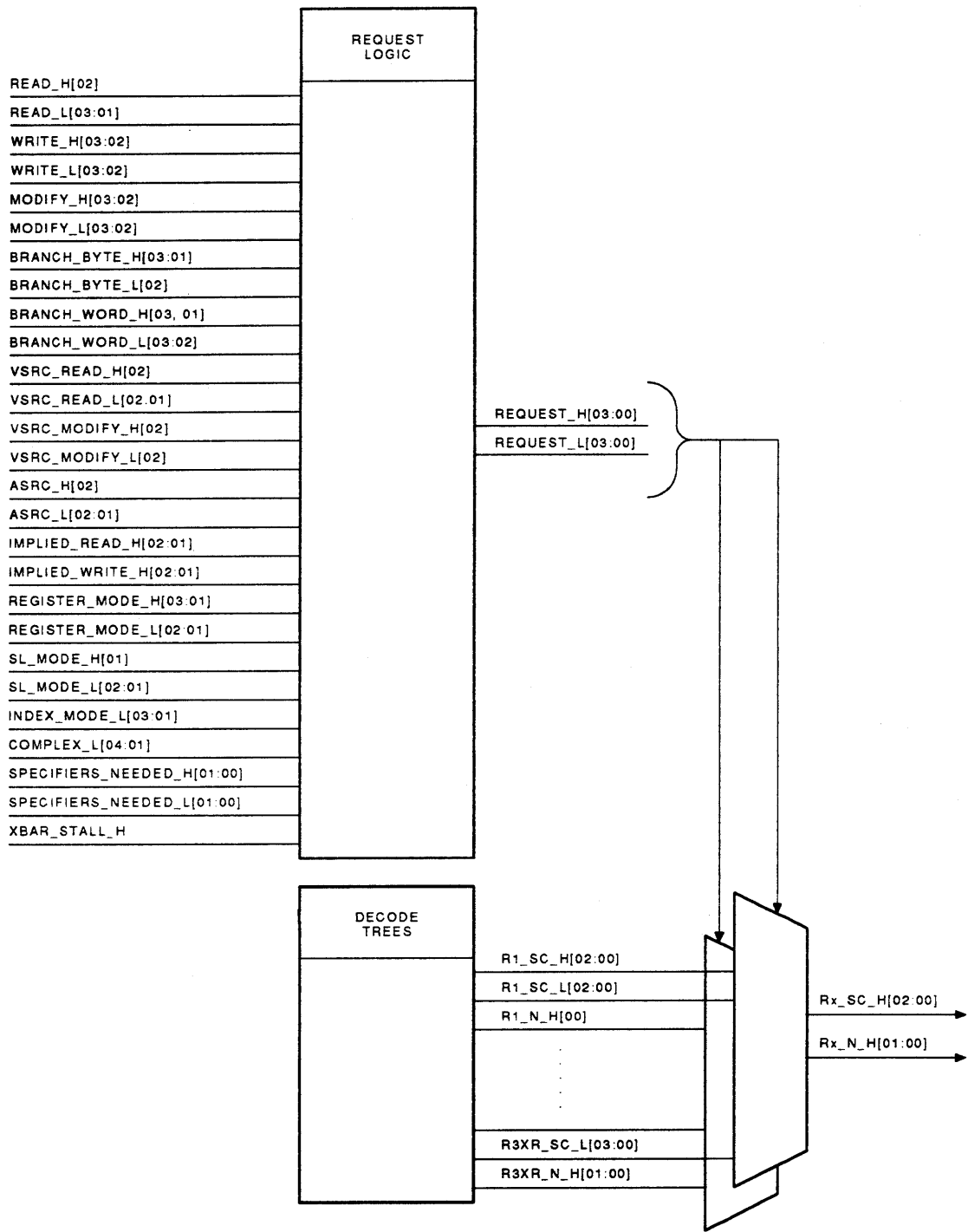
Table 4–2 describes the I-stream addressed by each decode tree.

Table 4–2 XBAR Decode Trees

Name	Function
R0	No specifiers are decoded.
R1	Decodes one specifier.
R1BB	Decodes one branch byte specifier.
R1BW	Decodes one branch word specifier.
R1I	Decodes one implied specifier.
R2	Decodes two specifiers.
R2BB	Decodes two specifiers; the first is a register or short literal specifier and the second is a branch byte displacement.
R2BW	Decodes two specifiers; the first is a register or short literal specifier and the second is a branch word displacement.
R2I	Decodes two specifiers; the second is implied.
R2R	Decodes two specifiers; the second is a register specifier.
R3	Decodes three nonconflicting specifiers.
R3BB	Decodes three specifiers; the third is a branch byte displacement.
R3BW	Decodes three specifiers; the third is a branch word displacement.
R3XR	Decodes three specifiers but not in the same cycle.

4.1.5 Request Logic

The request logic decodes the specifier addressing modes and access types, and it outputs the 4-bit field (REQUEST_H[03:00]) that selects 1 of the 14 decode tree outputs (Rx_SC_H[02:00] and Rx_N_H[01:00]). Figure 4–6 shows the relationship of these units and shows the inputs and outputs of the request logic.



MR_X0077_89

Figure 4-6 Request Logic

The request logic uses SPECIFIERS_NEEDED_H[01:00] to determine the number of specifiers to decode. (For example, if only one specifier is to be decoded, the request logic selects only one of the R1 trees.) The addressing mode logic provides four inputs:

Register mode
Short literal mode
Index mode
Complex

These inputs determine the number of specifiers that can be decoded and passed to the specifier handlers. For example, for ADDL3 R1 #43 R5, the request logic and decode tree logic perform their parallel operations as follows:

- **Request logic** — Receives SPECIFIERS_NEEDED_H[01:00] = 3 (the number of specifiers in the instruction) from the XRAM and specifier count logic. The addressing modes of the three specifiers in the instruction are provided by the simple decode logic. (REGISTER_MODE_H[03:01] = 5, specifiers 1 and 3 are register specifiers and SL_MODE_H[02:01] = 2, specifier 2 is a SL specifier). The access type of the specifiers is also input to the request logic. These inputs, supplied by the DRAM, would be READ_H[03:01] = 011 and WRITE_H[03:02] = 10, signifying read, read, write as the order of access in the instruction.

Specifier data types (SP1_DATATYPE_H[02:00] and SP2_DATATYPE_H[02:00] = 0 longword) are also input to the request logic. This signal is also from the DRAM.

The request logic output (REQUEST_H[03:00]) produces an output based on the above inputs that selects the R3_SC_H[03:00] and R3_N_H[01:00].

- **Tree logic functions** — In parallel with the request logic functions, the decode trees decode the I-stream, the addressing modes, and data types of the specifiers and stall signals from the specifier handlers of the OPU to produce shift counts and specifier decode counts for the instruction.

The tree logic (specifically the R3 tree) produces an output that produces shift counts (R3_SC_H[02:00]) and the number of specifiers decoded (R3_N_H[01:00]) for an instruction that contains three specifiers.

The tree logic produces the R3_N_H[01:00] outputs first and uses them as a basis for generating R3_SC_H[03:00].

To generate R3_N_H[01:00], the logic inputs the valid counts for the instruction buffer data that is being decoded (I_VALID_H[08:00]).

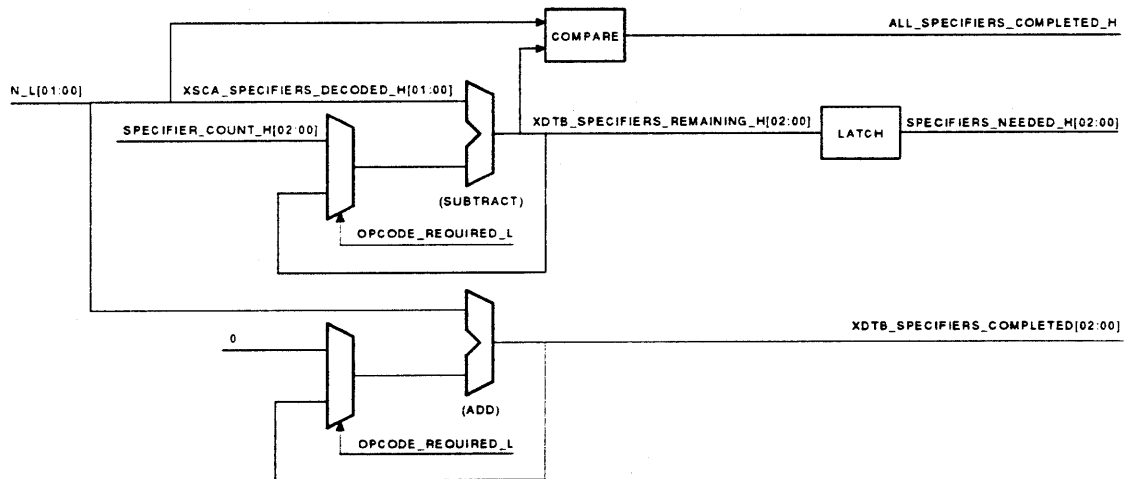
4.1.6 Specifier Count Logic

Specifier counts are produced every cycle to determine the initial number of specifiers an instruction contains (SPECIFIER_COUNT_H[02:00]), the number decoded in a cycle (SPECIFIERS_DECODED_H[02:00]), and the number that remain to be decoded (SPECIFIERS_REMAINING_H[02:00]). Figure 4-7 shows the specifier count logic.

Initially, SPECIFIER_COUNT_H[02:00] is produced by the XRAM and input to the specifier count logic. The XRAM generates the specifier count based on the opcode of the instruction.

The decoded specifier output of the decode tree logic (N_L[01:00]) is subtracted from the specifier count to produce SPECIFIERS_REMAINING_H[02:00].

- SPECIFIERS_NEEDED_H[01:00] is the number of specifiers needed in the current decode cycle. This signal is decoded so that it represents only a maximum of three because the maximum number of specifiers decoded in a single cycle is three. This signal is loaded into an adder and a comparator.
- XSCA_SPECIFIERS_DECODED_H[01:00] is the latched value of N_L[01:00] and is loaded into an adder with XDTB_SPECIFIERS_REMAINING_H[02:00] to produce XSCA_SPECIFIERS_REMAINING_H[02:00].
- The comparator receives N_L[01:00] and XSCA_SPECIFIERS_REMAINING_H[02:00], and outputs XSCA_ALL_SPECIFIERS_COMPLETED_H when they are equal.



MR_X0078_00

Figure 4-7 Specifier Count Logic

4.1.7 Shift Count Logic

Each cycle, the XBAR passes a shift count (XSCA_SHIFTCOUNT_H[03:00]) to the instruction buffer. The shift count directs shifting of decoded bytes out of the instruction buffer and replenishment with a new I-stream.

Three signals provide control to the instruction buffer shifter:

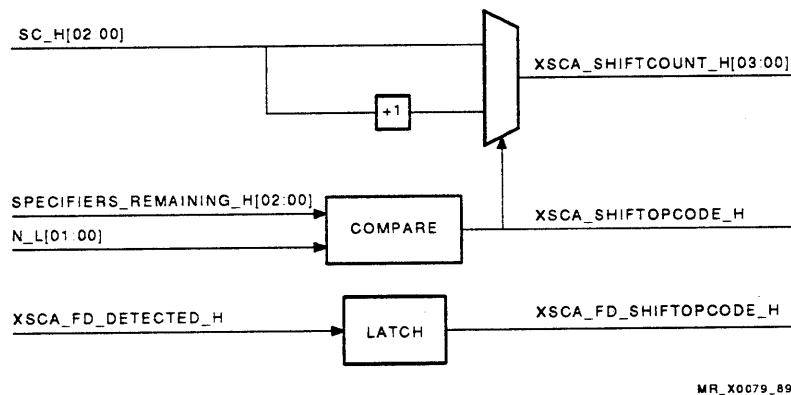
XSCA_SHIFTCOUNT_H[03:00]
 XSCA_FD_SHIFTOPCODE_H
 XSCA_SHIFTOPCODE_H

Figure 4-8 shows the generation of these three signals.

XSCA_SPECIFIER_COUNT_H[03:00] is generated from the decode tree logic SC_H[02:00] and is passed as the value to be loaded into the instruction buffer shifter. When an instruction is completely decoded, XSCA_SHIFTOPCODE_H is asserted and selects the incremented SC_H[02:00] to be sent to the shifter.

4.1.7.1 FD Shift Opcode

XSCA decodes the FD opcodes and asserts XSCA_FD_DETECTED_H. This signal is passed to the shift count logic, which asserts XSCA_FD_SHIFTOPCODE_H and directs the instruction buffer to shift out the FD opcode. Shifting the FD out of the opcode byte results in a shift count of one, as no specifiers are decoded until the FD is shifted.



MR_X0079_89

Figure 4-8 Shift Counts

4.1.8 Fork Logic

The EBox receives a fork address for each instruction that is decoded by the XBAR. The fork address is supplied by the XBAR and the instruction buffer.

The instruction buffer sends two fork signals:

IBOX_FORK_ADDRESS_H[07:00]
IBOX_FORK_ADDRESS_PARITY_H

The fork address is a copy of the opcode.

The XBAR outputs three fork signals:

- IBOX_FORK_VALID_H is the valid signal for the fork address.
- IBOX_FORK_ADDRESS_H[08] is asserted when an FD opcode is decoded.
- IBOX_REGISTER_FORK_H distinguishes between a register and a memory reference (asserted = register) when a VSRC specifier is decoded.

When a VSRC specifier is to be decoded, the fork is not validated until the determination between memory or register reference is made.

4.1.9 XBAR Displacement Data Path

Up to 32 bits of displacement can be passed by the XBAR to the CSU in a single cycle. Figure 4-9 shows the logic that decodes the complex specifiers and passes the related displacement to the CSU.

The XBAR displacement logic outputs four fields to the CSU:

- DISPLACEMENT_H[31:00] is 32 bits of branch displacement or complex specifier data.
- XREG_H[03:00] indicates the GPR of index register for indexed operand specifiers.
- YREG_H[03:00] indicates the GPR of base register for the operand specifier being delivered.
- XDTB_INDEXED_H indicates the specifier under decode is indexed mode.

4.1.9.1 Displacement

DISPLACEMENT_H[31:00] is provided by XDTA and XDTB. The 32-bit field is nibble sliced as follows:

XDTA_DISPLACEMENT_H[27:24, 19:16, 11:08, 03:00]
XDTB_DISPLACEMENT_H[31:28, 23:20, 15:12, 07:04]

The displacement field is selected from the instruction buffer data by decoding:

XSCA_REQUEST_H[03:00]
CASE_H[01:00]
XSCA_X8F_H

Decoding these three fields determines where in the I-stream the complex specifier is located and if the specifier is an extended immediate mode specifier (XSCA_X8F_H).

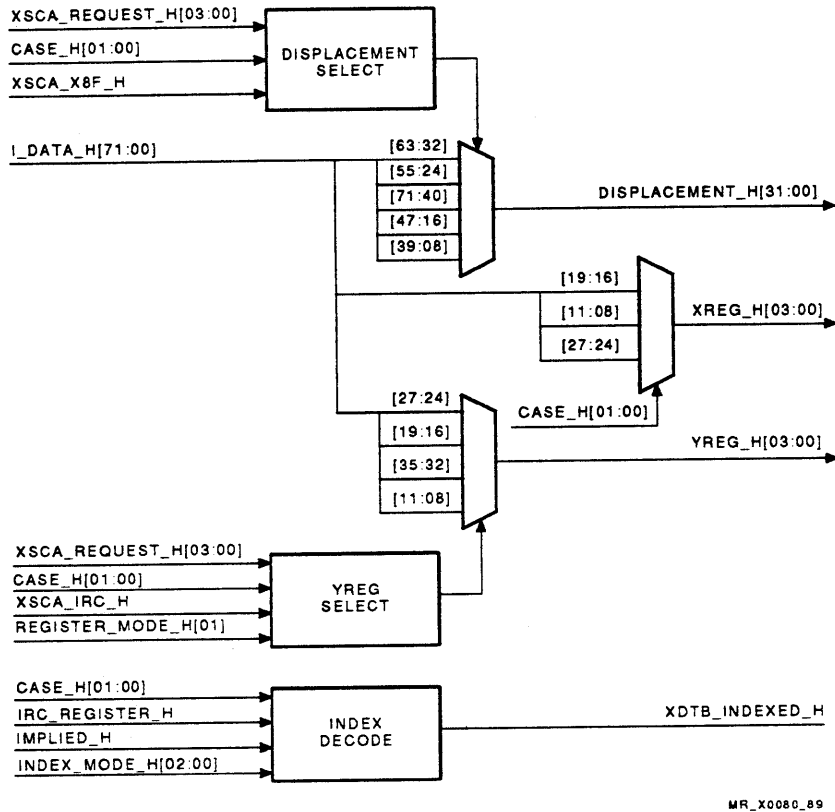


Figure 4-9 XBAR Displacement

4.1.9.2 Extended Immediate Mode (X8F) Detection

Most complex specifiers are decoded by the crossbar and passed to the CSU in a single cycle across the 32-bit XBAR to OPU data path. Extended immediate mode specifiers are of data types longer than 32 bits and require more than a single cycle to be decoded and passed to the OPU. Because of the size of these specifiers, special handling by the XBAR is required to process them.

The special handling of these specifiers involves manipulating the shift counts and specifier counts produced by the XBAR.

XSCA contains logic that detects extended immediate mode specifiers. This logic decodes the specifiers' data types (SP1_DATATYPE_H[02:01] and so on) from the DRAM, IMMEDIATE_MODE_H[04:01], INDEX_MODE_H[02:01], and CASE_H[01:00] (all from the simple decode logic). When extended immediate mode is detected, X8F_H, X8F_INHIBIT_SHIFTOPCODE_H, and X8F_SC_H[02] are asserted.

Asserting these X8F signals forces N_H[01:00] (from the decode trees and signifying the number of specifiers decoded) to equal 1 and also forces SC_H[02:00] (also from the decode trees and signifying the number of bytes to shift out of the instruction buffer) to equal 4. This scenario allows the XBAR to pass the decoded specifier to the OPU in multiple cycles without incorrectly affecting the specifier count logic and without asserting SHIFTOPCODE_H before the specifier is completely decoded.

4.1.9.3 XREG

When an indexed specifier is decoded, XREG_H[03:00] is asserted and sent with XDTB_INDEXED_H to the CSU. XREG_H[03:00] identifies the index register. XDTB_INDEXED_H is generated by decoding the following:

- IRC_REGISTER_MODE is asserted when the IRC is detected and the specifier is register mode.
- IMPLIED_H is asserted by decoding the request logic field.
- CASE_H[01:00] identifies the location of the complex specifier in the I-stream.
- INDEX_MODE_H[02:00], from the addressing mode logic, defines which specifier, if any, is decoded as an index specifier.

4.1.9.4 YREG

YREG_H[03:00] is generated in XDTA and sent to OSQA (CSU) to indicate which GPR the CSU references for an operand address calculation. Byte 1, 2, 3, or 4 of instruction buffer data is selected by CASE_H[01:00] to produce the YREG output.

The YREG generation logic also contains inputs from the request logic, DRAM, and IRC detection logic. When an IRC is detected, this logic supplies the GPR number for specifiers during IRC handling.

4.1.10 XBAR Short Literal Data Path

The XBAR short literal logic outputs a 6-bit short literal specifier, a valid bit, and the number of the short literal specifier to the short literal expansion unit of the specifier evaluation logic. Figure 4-10 shows the organization of the XBAR short literal logic.

4.1.10.1 Short Literal Data Select

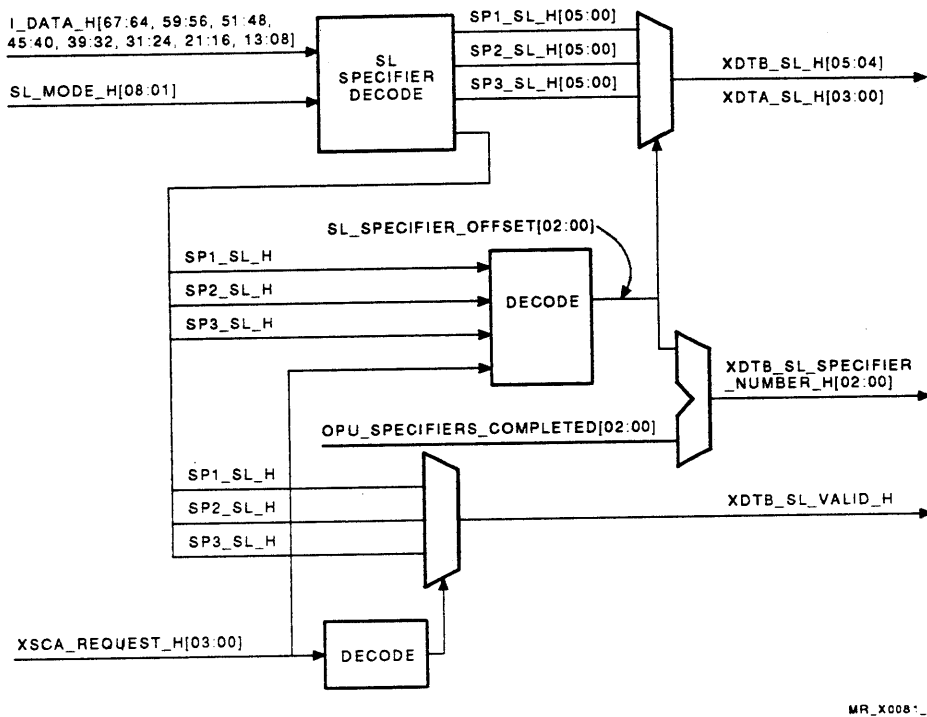
The XBAR SL data path receives the following:

- Instruction buffer data (I_DATA_H[67:64, 59:56, 51:48, 39:32, 31:24, 23:16, 15:08]).
- SL_MODE_H[08:01] is from the XBAR simple decode logic and the instruction buffer simple decode logic. It identifies the bytes that contain short literal specifiers. That is, if bit 1 is asserted, then a short literal specifier is detected in the byte 1 position of the I-stream.
- XSCA_REQUEST_H[03:00] is the output of the request logic.

The specifier decode logic decodes the instruction buffer data and the addressing mode to produce the following:

- SPx_SL_H[05:00] contains the 6-bit short literal data for three specifiers (x = 1, 2, or 3).
- SPx_SL_H corresponds to the three byte positions that the SL data could be in. SP1_SL_H, asserted, denotes specifier 1 is a short literal specifier.

The byte position of the short literal specifier (SPx_SL_H) is decoded with XSCA_REQUEST_H[03:00] to produce the short literal specifier offset (SL_SPECIFIER_OFFSET_H[02:00]). This signal selects the short literal data and outputs the field (XDTA_SL_H[05:04] and XDTB_SL_H[03:00]) to the SL specifier handler.



MR_X008*_89

Figure 4-10 Short Literal Logic

4.1.10.2 Short Literal Specifier Number

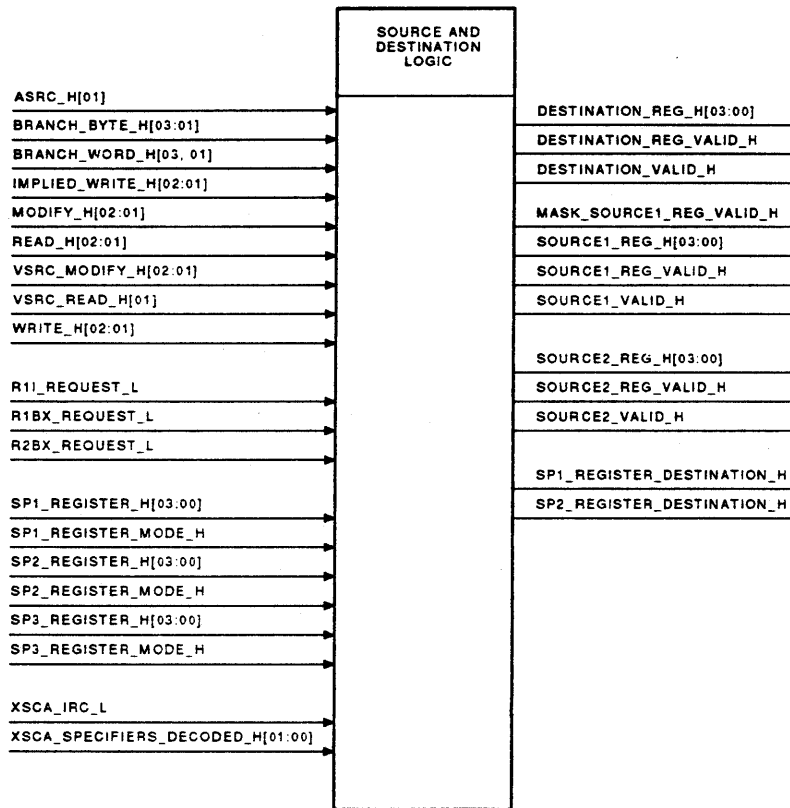
XDTB_SL_SPECIFIER_NUMBER_H[02:00] defines which specifier of the instruction is being passed to the SL specifier handler. This signal is generated by adding the specifier byte position to **OPU_SPECIFIERS_COMPLETED_H[02:00]**.

4.1.10.3 Short Literal Valid

XDTB_SL_VALID_H is sent with the valid short literal data to the SLU of the OPU. This signal is generated by decoding the input from the request logic (**XSCA_REQUEST_H[03:00]**) and selecting the valid byte containing the short literal specifier.

4.1.11 XBAR Source and Destination Logic

Each specifier decoded and passed to the specifier handlers has an associated source and destination field to identify the function of the specifier. The source and destination fields are processed by the specifier handlers and loaded into the EBox pointer queues until the instruction they represent is executed. Figure 4-11 shows the inputs and outputs of the source and destination logic.



MR_X0320_89

Figure 4-11 Source and Destination Logic

4.1.12 XBAR Source 1 Data Path

The source 1 data path logic passes three fields to the FPL and, if the specifier is a register, also passes a register field to the mask logic. Figure 4-12 shows the XBAR source 1 logic.

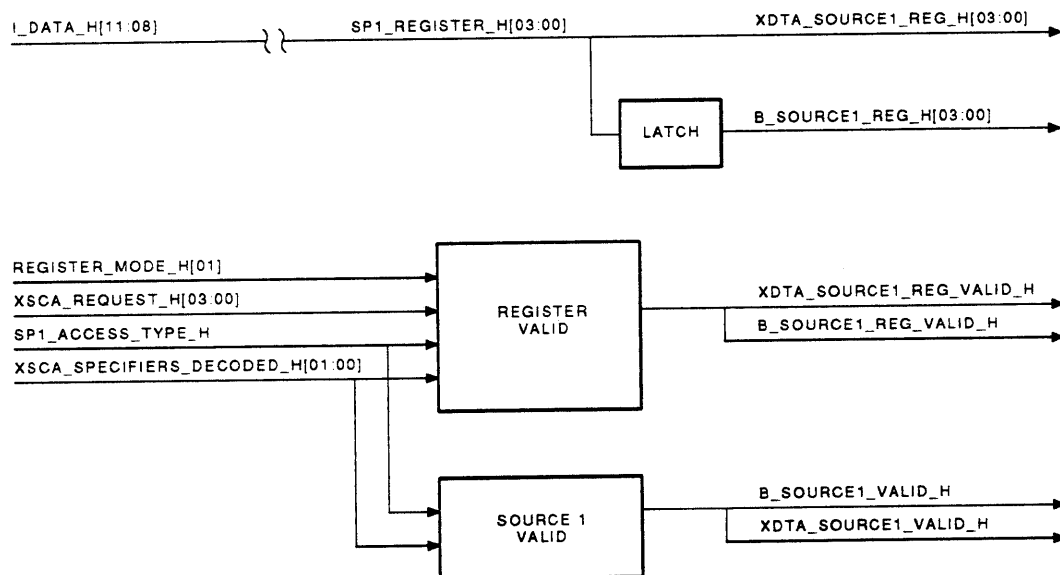
The XBAR source 1 logic passes the low nibble of instruction buffer byte 1 to the FPL as the register field (XDTA_SOURCE1_REG_H[03:00]). The source 1 validation logic receives the following:

REGISTER_MODE_H[01]
 XSCA_REQUEST_H[03:00]
 SPECIFIER1_ACCESS_TYPE_H
 XSCA_SPECIFIERS_DECODED_H[02:00]

These fields are decoded to output the two source 1 valid fields:

- XDTA_SOURCE1_REG_VALID_H is asserted when the present source 1 operand is a register specifier. This field validates XDTA_SOURCE1_REG_H[03:00].
- XDTA_SOURCE1_VALID_H validates the specifier as a source operand.

When SOURCE1_REG_VALID_H is negated and SOURCE1_VALID_H is asserted, the source 1 specifier is a memory operand.



MR_X0082_89

Figure 4-12 XBAR Source 1 Logic

4.1.13 XBAR Source 2 Data Path

Source 2 fields are passed to the operand handlers if two source specifiers are decoded. The source 2 fields are similar to the source 1 fields in structure, but they rely on the source 1 specifier characteristics when they are generated. That is, if source 1 is a complex specifier, then the location in the I-stream that contains source 2 is determined by case, source 1, and source 2 data types and the number of specifiers completed. Figure 4-13 shows the generation the source 2 fields.

The source 2 select logic selects the byte position that contains the specifier. If the specifier is not a register, the byte position is selected, but the field (SP2_REGISTER_H[03:00]) is negated. The inputs to the select logic are as follows:

- CASE_H[01:00] identifies the location in the instruction of the complex specifier.
- XSCA_SP1_DATATYPE_H[02:00] identifies the data type of the source 1 specifier.
- REGISTER_MODE_H[07:01], from register mode logic and instruction buffer simple decode logic, identifies which bytes contain register specifiers.
- INDEX_MODE_H[01], IMMEDIATE_MODE_H[01], and ABSOLUTE_MODE_H[01] identify the source 2 specifier as either of these three addressing modes.
- SP2_ACCESS_TYPE_H provides the access type of the source 2 specifier.
- XSCA_SPECIFIERS_DECODED_H[02:00] identifies the total number of specifiers decoded in the instruction currently being decoded.

The source 2 register field is validated if the specifier is a valid register mode specifier. The specifier 2 access type is input into the valid logic to differentiate between the destination and the source specifiers. A write or modify access negates the validation of a specifier as a source operand.

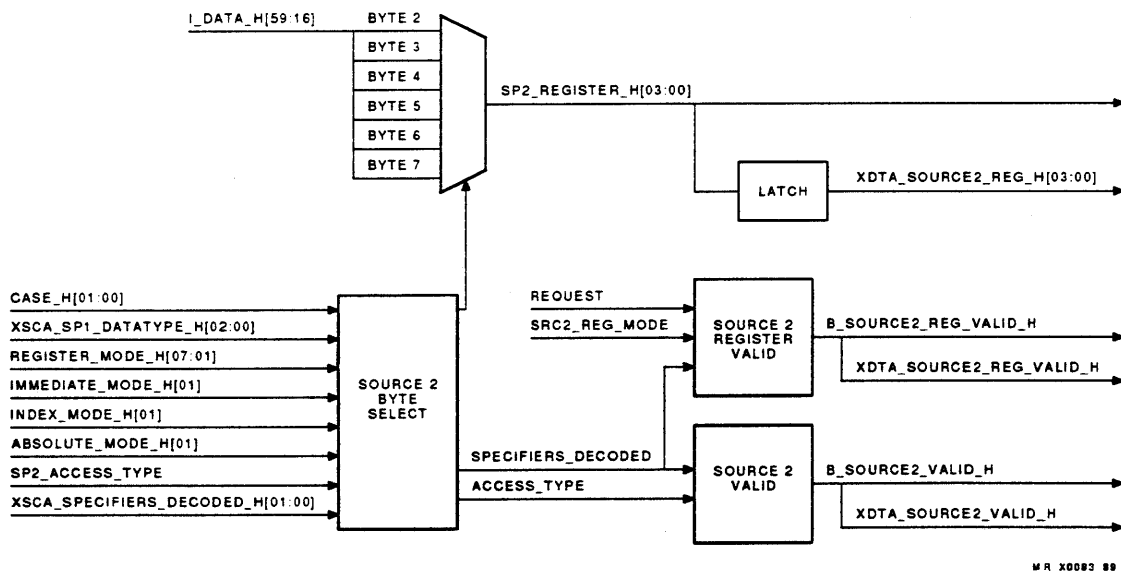


Figure 4-13 XBAR Source 2 Logic

4.1.14 XBAR Destination

The XBAR destination fields are generated for destination specifiers that are passed to the specifier handlers. Figure 4-14 shows a simplified block diagram of the destination logic.

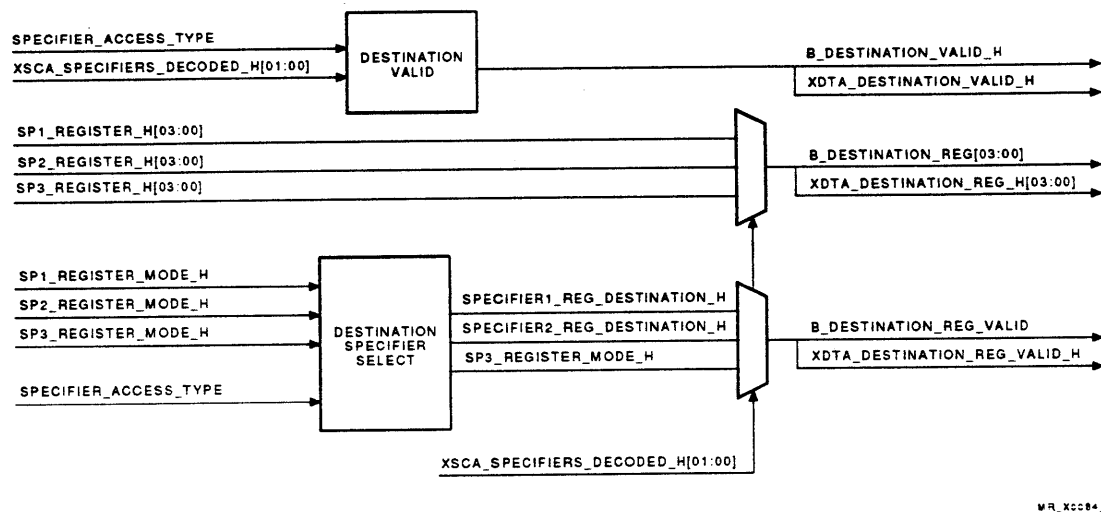


Figure 4-14 XBAR Destination

4.1.14.1 Destination Valid

XDTA_DESTINATION_VALID_H is sent to the FPL of the OPU to validate a destination specifier. This signal is generated from access type and specifiers decoded fields.

The access type of the destination specifier must be write, modify, implied write, or VSRC modify. Any specifier with an access type of read is a source specifier.

To validate the destination specifier, the specifiers decoded must indicate the specifier under decode (destination specifier) is the last specifier of the instruction.

4.1.14.2 Destination Register Valid

Destination registers are validated when the destination specifier is a register. SPx_REGISTER_DESTINATION_H is asserted when the selected destination specifier is a register mode specifier.

4.1.15 Register Masks

The register mask logic receives a register number for any register that is read or written to by the EBox during the execution of an instruction. All of the register accesses in an instruction are accumulated in one of two registers and passed to the OCTL unit of the OPU at the completion of the instruction decode. Figure 4-15 shows the XBAR mask logic.

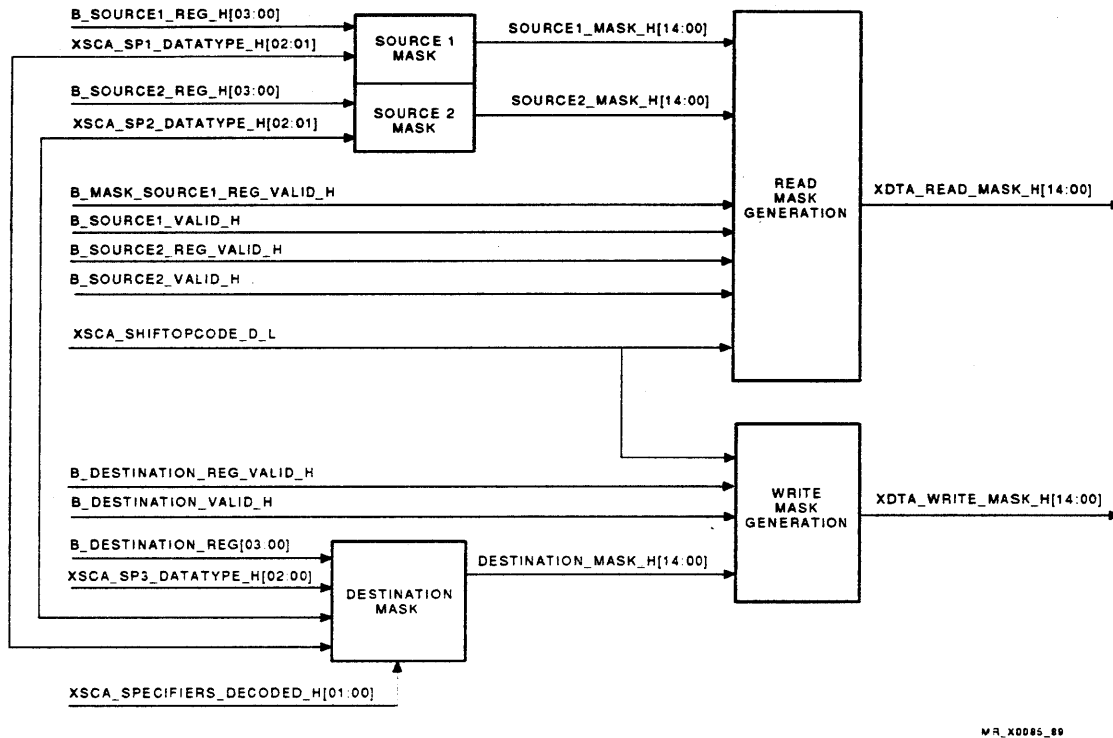


Figure 4-15 XBAR Read and Write Masks

4.1.15.1 Read Mask

A XBAR read mask is generated by decoding each valid source operand that is passed to the FPL. The read mask logic receives the following:

B_SOURCE_x_REG_H[03:00]
 B_SOURCE_x_VALID_H
 B_SOURCE_x_REG_VALID_H
 XSCA_SP_x_DATATYPE_H[02:00]

The source valid and source register fields are decoded to detect valid register specifiers. When a valid register specifier is detected, the register field is stored in the read mask logic.

The data type of the specifier (XSCA_SP_x_DATATYPE_H[02:00]) further defines the read access of the registers by asserting subsequent register numbers in the read mask when the base register data type is quadword or octaword. For example, a quadword read of register 1 would generate XDTA_READ_MASK_H[14:00] = 0006. This mask identifies register 1 and register 2 as being read during the execution of this instruction.

READ_MASK_H[14:00] is passed to OCTL when XSCA_SHIFTOPCODE_H is asserted.

4.1.15.2 Write Mask

The XBAR write mask field contains entries pertaining to registers that are written to during the execution of an instruction. The write mask is generated by decoding the following:

```
DESTINATION_REG_VALID_H
DESTINATION_VALID_H
DESTINATION_REG[03:00]
XSCA_SP3_DATATYPE_H[02:00]
```

The data type and the register field generate DESTINATION_MASK_H[14:00] when the data type is as follows:

```
Write
Modify
Implied write
VSRC modify
```

When both DESTINATION_REG_VALID_H and DESTINATION_VALID_H are asserted, an entry is added to the write mask.

4.1.15.3 Implied Mask

When a character string instruction is detected, the XRAM asserts IMPLIED_MASK_H, which asserts XDTA_WRITE_MASK_H[05:00]. R0 through R5 are asserted in the write mask because they contain the control block that maintains updated addresses and state information during the execution of the instruction.

4.1.16 Intra-Instruction Read Conflicts

Intra-instruction read conflicts (IRCs) occur when a read conflict is in the specifiers of a single instruction. These conflicts occur when a register specifier is to be used as data in the EBox and is subsequently used as the base register for an autoincrement or autodecrement.

An IRC occurs in the instruction ADDL3 R0, (R0)+, R1. This instruction directs the EBox to read R0 and also directs the IBox to use R0 as the base register for an autoincrement. The IBox detects this IRC by monitoring the read and write masks that are generated for each instruction. When the XBAR detects an IRC, it notifies the OPU and passes the autoincrement or autodecrement specifier to the CSU of the OPU and also passes all subsequent register specifiers through the CSU, instead of through only the FPL.

The CSU processes the autoincrement specifier, updates the IBox copy of the GPRs, but it does not update the EBox copy of the GPRs. Subsequent register specifiers are processed by the CSU and passed to the EBox as data and placed in the source list. The data passed to the source list is the content of the IBox copy of the register.

The OPU records the modified register numbers until the instruction has been completed and then writes the registers to the EBox GPRs using the data in the IBox GPRs.

The XBAR does not process any subsequent specifiers until the instruction is complete (in the EBox) and then writes the IBox (modified) GPRs to the EBox GPRs. This process is called a delayed GPR update.

The IRC mask decode logic (shown in Figure 4-16) generates a composite IRC mask (IRC_MASK_H[08:00]). The composite IRC mask identifies the data types of GPR accesses for the IRC detection logic.

IRC_MASK_H[08:00] is generated by decoding the read mask and decoding the I-stream for autoincrement and autodecrement mode specifiers. IRC_YREG_H[03:00] is also input into the IRC mask logic. IRC_YREG_H[03:00] identifies the base register of register specifiers in the instruction.

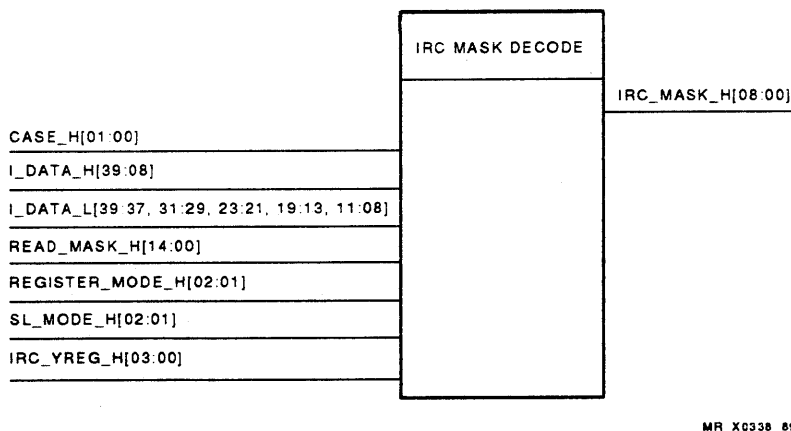
The IRC composite mask, n (number of specifiers decoded), CASE_[01:00], index mode, and the specifier data types are input to the IRC detection logic (Figure 4-17). The R1xx shift count and specifiers decoded count are also input to this logic. The output of the IRC detection logic produces the specifier counts and specifier decoded counts during IRC handling.

When an IRC is detected, three outputs are produced to direct the special handling required for the instruction:

IRC_H
IRC_N_H[00]
IRC_SC_H[02:00]

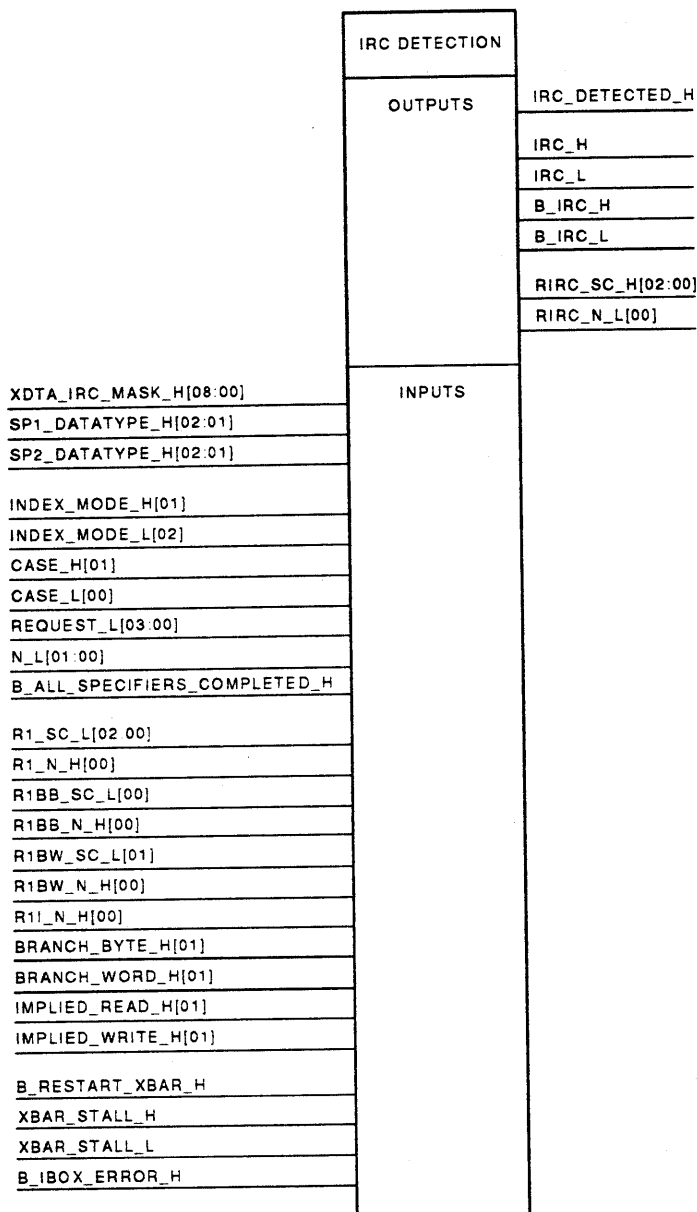
IRC_N_H[00] provides the specifiers decoded count for the IRC and can equal only zero or one. IRC_SC_H[02:00] provides the shift count for the IRC. During an IRC, the shift counts are based only on the decode tree outputs of R1, R1BB, R1BW, and R1I.

The XBAR continues to decode one specifier at a time until the instruction is complete. All decoded specifiers are passed through the CSU. Autoincrements or autodecrements are performed only on the IBox GPRs. The XBAR stalls (IRC_STALL_H) when the instruction is completely decoded and waits for the EBox to complete the instruction. When the EBox completes the instruction, a delayed GPR update is performed and the XBAR resumes decoding (OSQA_DGPR_UPDATE_FINISHED_H negates IRC_STALL_H).



MR_X0338_89

Figure 4-16 IRC Detection: Read Mask



MR_X0339_89

Figure 4-17 IRC Detection: IRC Mask

4.1.17 XBAR Stalls

XSCA detects stalls in the XBAR. Most of the stalls in the XBAR are related to specifiers that require special handling or are related to situations external to the XBAR logic. The external situations are a result of the limitations of the logic units that receive decoded specifiers from the XBAR or because the instruction buffer and/or the VIC are not providing valid I-stream to the XBAR. This section describes the logic that initiates XBAR stalls, and it describes the conditions external to the XBAR that cause the XBAR to stall.

When **XBAR_STALL_H** is asserted, the XBAR forces the decode tree logic and the request logic to produce outputs that generate no shift counts and produce zero for a specifiers decoded output. During a XBAR stall, the request logic produces an output that selects a nonexistent decode tree output (**REQUEST_H[03:00]** = F). Selecting the nonexistent decode tree output generates a shift count of zero (**XSCA_SHIFTCOUNT_H[03:00]** = 0) and generates a number of specifiers decoded count of zero (**N_H[01:00]** = 0). The shift count of zero directs the instruction buffer to hold (not shift) the I-stream it is presenting for decode and the specifiers decoded count of zero results in no valid specifier data being passed to the OPU.

XBAR_STALL_H is asserted when the following conditions are true:

- **IRC_STALL_H** is asserted after the XBAR passes all the specifiers of an instruction that contains an IRC to the OPU.
- **RAF_H** is asserted, signifying a reserved addressing fault. This fault is detected in the XBAR.
- **FPD_FLUSH_H** is asserted when the EBox stalls the execution of an instruction to service an exception or an interrupt.
- **PCHI_IBUF_FLUSH_H** is asserted. This signal is asserted when there is a change in I-stream.
- **I_VALID_L[00]**, when negated, there is no valid opcode in the instruction buffer.
- **OCTL_MASK_STALL_H** is asserted when the OPU read/write register mask logic is full.
- **OSQA_DECODE_STALL_H** is asserted because the OPU cannot receive decoded specifiers because one or more of the following conditions exist:
 - EBox source list or source pointers are full.
 - Read or write conflict stall exists (inter-instruction conflict or scoreboard stall).
 - Autoincrement or autodecrement follows a branch.
 - Branch follows a branch.
 - Two conditional branches are buffered.
 - GPR update is delayed.

The XBAR generates two stall signals, **OPU_STALL_H** and **SL_STALL_H**, that are related to the availability of resources in the OPU. These signals inhibit the XBAR from decoding complex or short literal specifiers when they are asserted.

When **OPU_STALL_H** is asserted, this informs the XBAR that the OPU is processing a complex specifier and has another complex specifier buffered in the OPU stall logic. This stall signal inhibits the XBAR from decoding and passing another complex specifier to the OPU. This signal is input to the decode tree logic and affects the shift count and number of specifiers decoded outputs. When **OPU_STALL_H** is asserted and another complex specifier is decoded, **XBAR_STALL_H** is asserted.

SL_STALL_H is asserted to inform the XBAR that two short literal specifiers are currently in the SLU. This stall inhibits decoding another short literal specifier and asserts **XBAR_STALL_H** when another short literal is encountered.

4.2 Branch Prediction

This section describes the VAX 9000 family branch prediction functions. It describes the three methods used to predict a branch and the microcode that controls them.

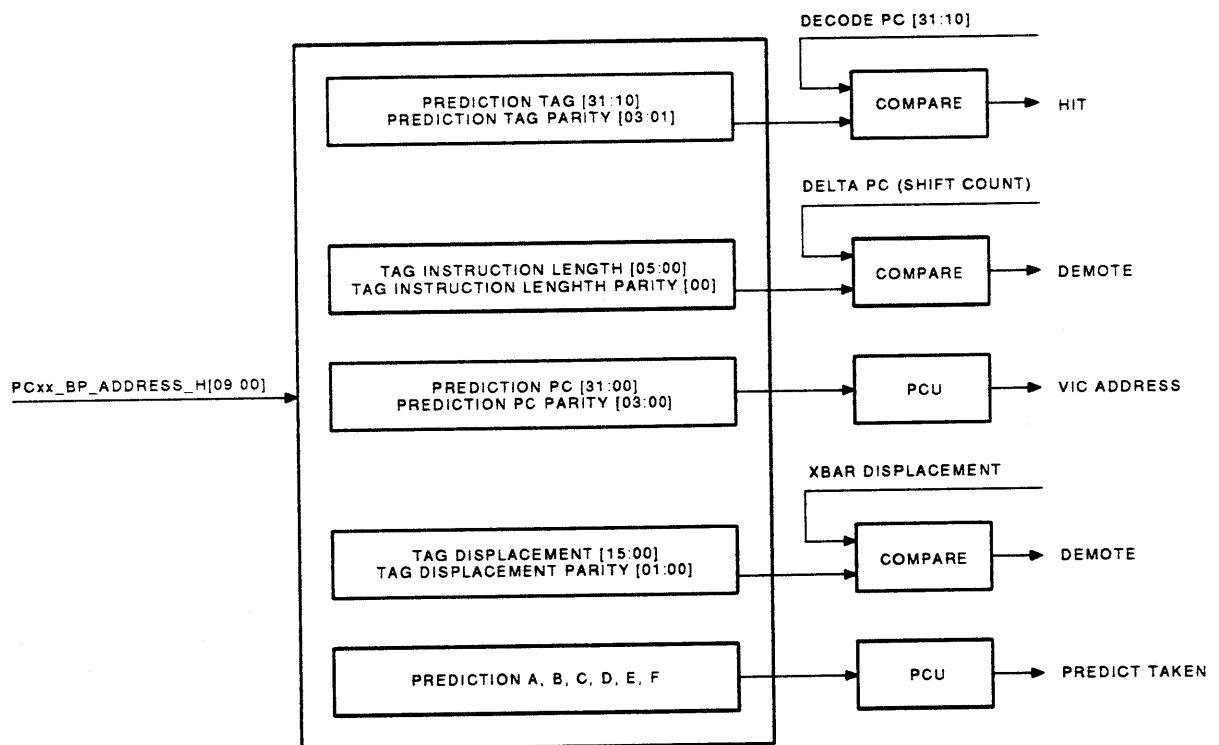
The three methods used to predict branches are as follows:

- Primary mode is used when the branch being decoded is stored in the branch prediction cache (BPC). The fields in the BPC direct the decision of taking or not taking the branch and provide the branch PC.
- Demote is related to primary mode and occurs when the cached displacement or instruction length does not match that of the branch being decoded. Under demote, the cached prediction bit is used and the branch PC is provided by the CSU.
- Secondary mode is used when the branch is not stored in the BPC. The branch may not be cached because it has not been encountered before or because it is not a cacheable branch. In this mode, the branch PC is provided by the CSU.

4.2.1 Primary Predictions

Primary branch predictions use the BPC to direct the flow of I-stream when branches are encountered. These predictions are based on the content of the BPC.

The BPC contains five fields that are written when a cacheable branch is first encountered and read when the branch is subsequently encountered. Figure 4-18 shows the organization of the BPC.



MR_X0340_89

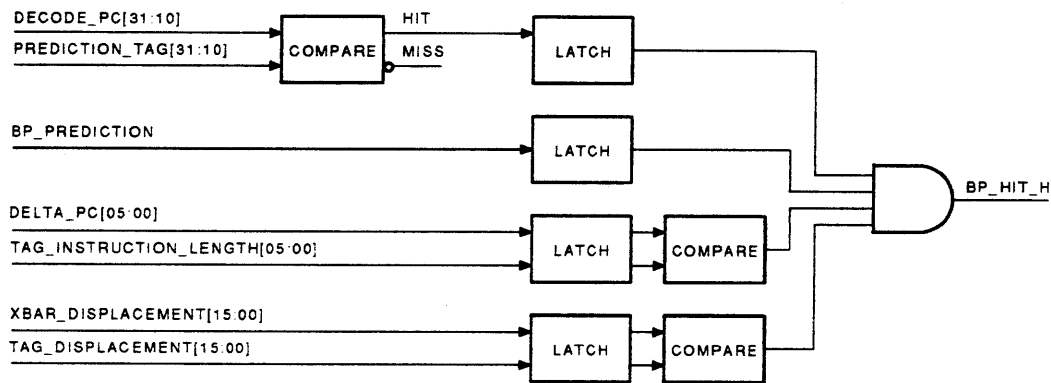
Figure 4-18 BPC Organization

The BPC has 1024 locations and is never flushed. The following fields describe the contents and functions of the branch prediction cache fields:

- **Branch PC tag** — This field contains bits [31:10] of the virtual PC of the branch.
- **Prediction PC** — This 32-bit field is the destination PC of the branch.
- **Branch displacement** — This 16-bit field is the actual displacement of the branch.
- **Branch instruction length** — This 6-bit field contains the actual instruction length of the branch.
- **Prediction bit** — This bit is set when the branch is predicted taken.

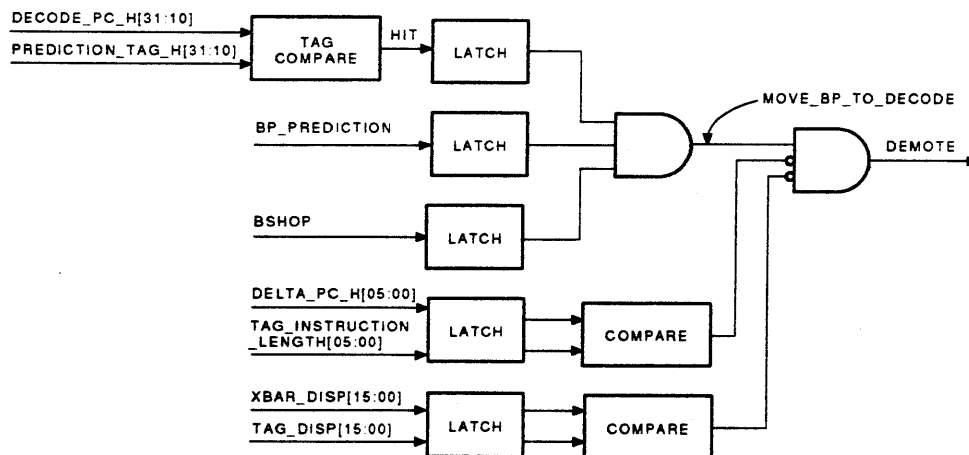
4.2.1.1 Primary Hits

A branch prediction hit occurs when the BP tag field matches the corresponding bits in the decode PC. A hit directs the cached prediction PC to be loaded into the decode PC if the prediction bit is asserted. Figures 4-19 and 4-20 show the compare logic of the BPC and the results of the comparisons.



MR_X0086_89

Figure 4-19 BPC Compare: Hit



MR_X0087_89

Figure 4-20 BPC Compare: Demote

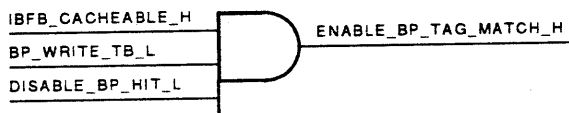
When a branch is encountered, the BPC is accessed to perform a compare. The compare results in one of three outputs: hit, miss, or demote.

Two cycles are required to produce a predicted taken branch from the branch prediction cache.

1. In the first cycle, `DECODE_PC[31:10]` and `PREDICTION_PC[31:10]` are compared and produce a hit or miss. A miss directs the prediction to the secondary mechanism and a hit requires subsequent comparisons to produce a hit or a demote.
2. If a hit is encountered in the PC comparisons, `DELTA_PC[05:00]` and `TAG_INSTRUCTION_LENGTH[05:00]` are compared, producing a hit or demote.
3. The `XBAR_DISPLACEMENT[15:00]` and `TAG_DISPLACEMENT[15:00]` are compared to produce a demote if they do not match.
4. If a hit is encountered, `BP_PREDICTION_H` must be set for the branch to be predicted taken.

4.2.1.2 Tag Match Enable

The IBUF simple decode logic decodes branch instructions and informs the PCU if the branch is cacheable. When `IBFB_CACHEABLE_H` is asserted and the BPC is not being written, a tag match is enabled. Figure 4-21 shows the logic that enables the tag match.



MR_X0088_89

Figure 4-21 BP Tag Match Enable

4.2.2 Demote

A primary prediction hit is demoted if the instruction length or the displacement of the branch do not match those stored in the BPC on a predicted taken branch. This occurs when the virtual PCs of the branch match, but they are not the same branch.

If the displacement and instruction length do not match, the history bit is checked before the branch is demoted. If the branch is predicted not taken (history bit = 0), no demote is necessary and decoding of the sequential I-stream continues. When the prediction bit is set, the branch is demoted, the target PC is provided by the CSU, and the BPC is written. That is, the new branch PC, instruction length, and displacement are written.

If the prediction is incorrect, the BPC is rewritten with the prediction bit cleared.

4.2.3 Secondary Predictions

Secondary predictions are performed by the branch bias logic of the XBAR. A secondary prediction is based on the opcode of the branch instruction. When a secondary prediction directs the IBox to take the branch, the branch target PC is supplied by the CSU.

Secondary mode is used when the branch has not been cached or when the BPC cannot be accessed because it is busy being written. Figure 4-22 shows the inputs and outputs of the branch bias logic.

The branch bias logic (BRAM) is comprised of 22 scan latches and selection logic. The latches are loaded with fixed predictions for branches and accessed when a branch that has not been cached is encountered. The opcode is used to select the prediction that is stored in the branch bias latches.

The fixed predictions for the branches are based on a bias for that instruction. That is, a BEQL tests two conditions for equality. Because equality in mathematical situations is rare, this branch would be predicted not taken.

A branch that is predicted taken in the BRAM logic asserts XDTB_BRAM_BIAS_H. This signal is passed to the branch prediction logic.

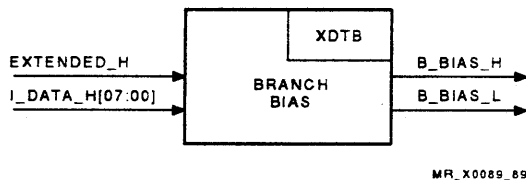


Figure 4-22 Branch Bias Logic

4.2.4 BPC Correction

Correction refers to the actions taken to correct an incorrect branch prediction before the branch instruction is shifted out of the instruction buffer. Branch instructions are predicted when they are shifted into the instruction buffer and are acted upon when they are shifted out of the instruction buffer. If the EBox validates the branch prediction before it is shifted out of the instruction buffer, and the prediction is incorrect, a correction is initiated.

A correction causes the rewriting (inverting) of the BPC prediction bit and the writing of the unwind PC (UNWIND_PC_H[31:00]) to the prediction PC (PREDICTION_PC_H[31:00]). If the branch was predicted to be taken and was incorrect, the IBox continues following sequential I-stream. If the branch was predicted to be not taken and was incorrect, the IBox redirects the I-stream to the PC supplied by the CSU. In both cases, the correct PC (sequential or nonsequential) is loaded from the unwind PC to the prediction PC.

4.2.5 BPC Unwind

Unwind refers to the actions necessary to correct an incorrect branch prediction whose validation arrives after the instruction buffer has shifted the branch instruction out and has started preprocessing the wrong I-stream. A BPC unwind initiates the following events:

- Asserts IBOX_ABORT to abort any instruction buffer requests for new I-stream.
- Flushes the instruction buffer of any I-stream loaded since the branch instruction.
- Invalidates any pointers to the EBox source list that have been entered after the branch prediction was made.
- Invalidates any OCTL GPR read/write masks that pertain to instruction decoded after the bad branch prediction.
- Rewrite (invert) the BPC history bit to reflect the correct prediction for the branch.
- The unwind PC is loaded into the prediction PC so that the IBox can start processing the correct I-stream.

4.3 PCU Microcode

The PCU and BPC are controlled by PCU microcode. This microcode is implemented in hard-coded microcode. That is, the microcode structure is generated in hard logic as opposed to a conventional RAM structure.

The PCU microcode consists of 38 bits that are partitioned into 22 fields. The fields are addressed by an 11-bit address field that contains PCU state information and the control signals to govern the flow of the microcode.

4.3.1 PCU Microaddress

The address for the PCU microcode is 11 bits wide. The address is organized as follows:

UADDRESS[03:00] provides PCU control signals.

UADDRESS[06:04] indicates where to store OPU_TARGET_PC.

UADDRESS[10:07] defines the state of the PCU at the beginning of the current cycle.

Table 4–3 describes the three address fields.

Table 4-3 PCU Microaddress Descriptions

UADDRESS[03:00]		PCU Control
UADDRESS[00]	This bit is set when the CSU has delivered a target PC to the PCU and there was not an unwind, last cycle.	
UADDRESS[01]	This signal indicates whether an IBox branch prediction was correct. The signal is only valid when UADDRESS[02] is asserted.	
UADDRESS[02]	Asserted, this bit validates UADDRESS[01].	
UADDRESS[03]	Asserted when an unconditional branch is in the IBUF.	
UADDRESS[06:04]		OPU Target PC Destination
UADDRESS[04] ¹	This bit directs the target PC to be stored in the unwind PC in the PCU.	
UADDRESS[05] ¹	This bit directs the target PC to be stored in the decode PC.	
UADDRESS[06] ¹	This bit is asserted if the PCU has a branch prediction, predict taken hit. This informs the microcode to ignore the target PC.	
UADDRESS[10:07]	Label	Description
0000	IDLE	The PCU has not encountered a branch and the IBox is executing sequential code.
0001	BSHOP VAL	PCU has received a target PC from the CSU before the branch has been shifted out of the IBUF and before EBox has validated prediction.
0010	VAL UNC TAR VAL TAR	XBAR has decoded a conditional, unconditional, and another conditional branch, in that order. There may have been sequential code between the branches. The PCU is waiting for EBox validation of the first branch, a target PC for the unconditional branch, and EBox validation and target PC for the third branch.
0011	VAL	The PCU is waiting for EBox validation of a branch prediction.
0100	VAL VAL TAR	Two conditional branches have been shifted out of IBUF. The first needs EBox validation and the second needs EBox validation and target PC from the CSU.
0101	BSHOP TAR	This state occurs when the branch is validated before it is shifted out of the IBUF. (EBox is ahead of IBox.)
0110	VAL TAR	XBAR decodes a conditional branch and shifts it out of the IBUF. The validation and target PC follow a few cycles later. (IBox is ahead of EBox.)

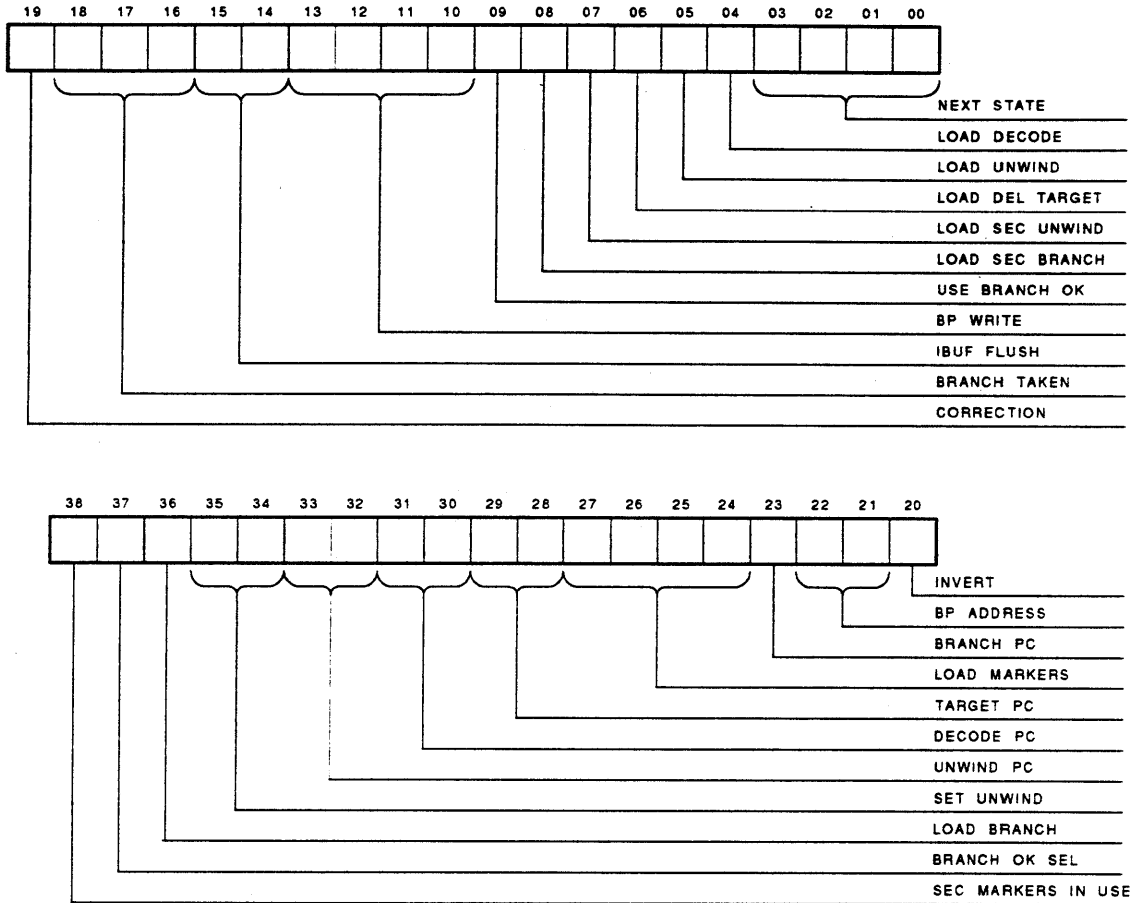
¹These three bits are mutually exclusive. Only one of these bits may be set in a cycle.

Table 4-3 (Cont.) PCU Microaddress Descriptions

UADDRESS[10:07]	Label	Description
0111	VAL TAR VAL TAR	Two conditional branches have been decoded by XBAR with both awaiting validation and target PCs.
1000	TAR VAL TAR	Two branches are being processed. The first has been validated but needs a target PC. The second needs both validation and a target PC.
1001	TAR	A target PC is required from the CSU to process a branch completely.
1010	VAL UNC TAR TAR	The PCU is processing three branches. The first is conditional and needs validation. The second and third are unconditional and need target PCs from the CSU.
1011	TAR TAR	The CSU has not delivered target PCs for two branches.
1100	VAL BSHOP VAL	An outstanding branch is in the PCU and the CSU has provided a target PC before the second branch has been shifted out of the instruction buffer. This state occurs only with JSB or BSB.
1101	VAL UNC TAR	The PCU is waiting for a validation for one branch and a target PC for a subsequent unconditional branch.
1110	VAL TAR TAR	The PCU needs validation and a target PC for a conditional branch and a target PC for a subsequent unconditional branch.
1111	VAL VAL TAR UNC TAR	The IBox has shifted out two conditional and one unconditional branches. The target PC has been delivered for only the first conditional branch and the EBox has not validated either conditional branch.

4.3.2 PCU Microword

The PCU microword is 39 bits wide and is partitioned into 22 fields. Figure 4-23 provides a breakdown of the microword. Table 4-4 describes each microcode field.



MR_X0090_00

Figure 4-23 PCU Microword Format

Table 4-4 PCU Microword Field Descriptions

[03:00]	Next State	Description
0000	IDLE	See Table 4-3 for field descriptions.
0001	BSHOP VAL	
0010	VAL UNC TAR VAL TAR	
0011	VAL	
0100	VAL VAL TAR	
0101	BSHOP TAR	
0110	VAL TAR	
0111	VAL TAR VAL TAR	
1000	TAR VAL TAR	
1001	TAR	
1010	VAL UNC TAR TAR	
1011	TAR TAR	
1100	VAL BSHOP VAL	
1101	VAL UNC TAR	
1110	VAL TAR TAR	
1111	VAL VAL TAR UNC TAR	
[04]	Load Decode	Description
1	—	Bad branch prediction. Load unwind PC into decode PC or PCU waits for target PC from CSU. Load decode is set when destination PC arrives.
[05]	Load Unwind	Description
1	—	Unwind PC is loaded when predict taken branch is shifted out of IBUF. On predict not taken, unwind PC is loaded when CSU provides destination PC.
[06]	Load Delayed Target	Description
1	—	When the CSU delivers a target PC before branch is shifted out, this signal instructs the PCU to latch and hold the target PC.
[07]	Load Second Unwind	Description
1	—	Same as load unwind, except the PC is loaded into second unwind PC (on second branch prediction).

Table 4-4 (Cont.) PCU Microword Field Descriptions

[08]	Load Second Branch	Description
1	–	Asserted when a second branch is shifted out of the IBUF. The PCU must latch and hold the following: - Virtual address - Prediction bit - Cacheable - Loop branch - Instruction length - Branch displacement
[09]	Use Branch OK	Description
1	–	Set when EBox sends validation of branch before shifting the branch out of IBUF. Remains set until branch is shifted.
[13:10]	BP Write	Description
000	FALSE	BPC is not written when this state is chosen.
001	NO HIT	BPC is written if present branch is not stored and cacheable.
010	PT NL	BPC is written in this case if the branch is incorrectly predicted taken and is not a loop branch.
011	CA	Cacheable branch is encountered for the first time and is written to the BPC.
100	NL CA	Not loop, cacheable. Branch predicted incorrectly from BPC is rewritten correctly (prediction bit) if it is not a loop branch.
101	NEXT	This field enables the write of a second branch being processed, to BPC, if the branch is cacheable.
[15:14]	IBUF Flush	Description
00	NOOP	Hold previous values.
01	TRUE	IBUF flush.
10	NPT	Not predict taken.
11	PT	Predict taken.

Table 4-4 (Cont.) PCU Microword Field Descriptions

[18:16]	Branch Taken	Description
00	NOOP	Hold previous value.
01	NPT	Not predict taken.
10	PT	Predict taken.
11	TRUE	Predict taken.
100	NOT BT	Not branch taken.
101	SBT	Second branch taken.
[19]	Correction	Description
1	–	Asserted when the EBox informs the IBox of a bad branch before it has been shifted out of the IBUF.
[20]	Invert	Description
1	–	When the EBox informs the IBox of a bad prediction, this signal directs the PCU to invert its fields.
[22:21]	BP Address	Description
00	–	If the branch is still in the IBUF, the decode PC provides the address.
01	–	If the branch has been shifted out of the IBUF, the stored branch PC provides the address.
10	–	Second branch PC is selected by this value.
[23]	Branch PC	Description
0	Decode PC	The branch PC is loaded into the decode PC.
1	Second branch PC	The branch PC is loaded into the second branch PC (two branches being processed).

Table 4-4 (Cont.) PCU Microword Field Descriptions

[27:24]	Load Markers	Description																																				
0000	NOOP	Load markers with values held last cycle.																																				
0001	TRUE	The marker set depends on the prediction of the branch and if the branch hits in the BPC. The following table provides the value dependent on these two variables. <table><tr><th>BP Hit</th><th>Predict Taken</th><th>Marker Set</th></tr><tr><td>0</td><td>0</td><td>Store in unwind.</td></tr><tr><td>0</td><td>1</td><td>Store in decode.</td></tr><tr><td>1</td><td>0</td><td>Store in unwind.</td></tr><tr><td>1</td><td>1</td><td>Ignore PC.</td></tr></table>	BP Hit	Predict Taken	Marker Set	0	0	Store in unwind.	0	1	Store in decode.	1	0	Store in unwind.	1	1	Ignore PC.																					
BP Hit	Predict Taken	Marker Set																																				
0	0	Store in unwind.																																				
0	1	Store in decode.																																				
1	0	Store in unwind.																																				
1	1	Ignore PC.																																				
0010	FALSE	All markers, first and second, are cleared when all branches are completely processed or when EBox detects a bad branch prediction.																																				
0011	SET SID	This state clears all markers and sets STORE_IN_DECODE.																																				
0100	TAR CASES	This state is set when the validation of a branch arrives before the destination PC. This state depends on three signals: BRANCH_OK, BP_HIT, and PREDICT_TAKEN. <table><tr><th>Branch OK</th><th>BP Hit</th><th>Predict Taken</th><th>Marker Set</th></tr><tr><td>0</td><td>0</td><td>0</td><td>Store in decode.</td></tr><tr><td>0</td><td>0</td><td>1</td><td>Ignore PC.</td></tr><tr><td>0</td><td>1</td><td>0</td><td>Store in decode.</td></tr><tr><td>0</td><td>1</td><td>1</td><td>Ignore PC.</td></tr><tr><td>1</td><td>0</td><td>0</td><td>Ignore PC.</td></tr><tr><td>1</td><td>0</td><td>1</td><td>Store in decode.</td></tr><tr><td>1</td><td>1</td><td>0</td><td>Ignore PC.</td></tr><tr><td>1</td><td>1</td><td>1</td><td>Ignore PC.</td></tr></table>	Branch OK	BP Hit	Predict Taken	Marker Set	0	0	0	Store in decode.	0	0	1	Ignore PC.	0	1	0	Store in decode.	0	1	1	Ignore PC.	1	0	0	Ignore PC.	1	0	1	Store in decode.	1	1	0	Ignore PC.	1	1	1	Ignore PC.
Branch OK	BP Hit	Predict Taken	Marker Set																																			
0	0	0	Store in decode.																																			
0	0	1	Ignore PC.																																			
0	1	0	Store in decode.																																			
0	1	1	Ignore PC.																																			
1	0	0	Ignore PC.																																			
1	0	1	Store in decode.																																			
1	1	0	Ignore PC.																																			
1	1	1	Ignore PC.																																			
0101	FROM SECOND	When processing two branches, at the completion of the first, this field is selected to move the markers stored in the second position into the first position.																																				
0110	SET IG	When the PCU is awaiting a destination PC from the CSU, and the EBox indicates the prediction is wrong, markers are switched from store in decode to ignore PC.																																				
0111	SECOND	When the microcode selects the second set of markers, the TRUE state is used as the input.																																				
1000	PUSH	This field is only used when going to state VAL VAL TAR UNC TAR. The first markers get the content of the second marker and the second marker gets loaded with appropriate markers for the unconditional branch that has just been shifted out of the IBUF.																																				

Table 4-4 (Cont.) PCU Microword Field Descriptions

[29:28]	Target PC Select	Description
00	OPU results	Target PC provided by the SCU.
01	Unwind PC	Unwind PC provides target PC.
10	Delayed target PC	Wait for target PC.
[31:30]	Decode PC Select	Description
00	Target PC	Target PC is selected on predict taken.
01	Next PC	Next PC is selected on predict not taken.
[33:32]	Unwind PC Select	Description
00	Next PC	
01	OPU result	
10	Second unwind PC	
[35:34]	Set Unwind	Description
01	Next	On a bad branch prediction, this signal informs the IBox to unwind from the prediction. This signal is used only if the branch has been shifted out of the IBUF. Next selects next PC on a bad predict taken.
10	OPU result	Load OPU result on bad not taken prediction.
11	Second unwind	Load second unwind PC.
[36]	Load Branch	Description
1	–	This field is the same as load second branch. The same information is stored but pertains to the first branch.
[37]	Branch OK Select	Description
1	–	This field can force the validation signal from the EBox. The signal indicates that the PCU has acknowledged the bad prediction and has corrected it without causing any damage to the integrity of the IBUF.
[38]	Second Markers	Description
0	–	When a demote occurs, demote the first branch.
1	–	When a demote occurs, demote the second branch.

4.3.3 Writing the BPC

The PCU writes the information pertaining to cacheable branches it has encountered. Certain criteria must be met for a branch to be cacheable:

- Branches must have displacements that will not change during the execution of the instruction.
- Branches must have a displacement as part of the I-stream.

A branch that has already been written to cache will not be rewritten unless the prediction is incorrect. The EBox informs the IBox of an incorrect prediction and the prediction bit is inverted.

4.3.3.1 BPC Write Enable

The BP write field of the PCU microcode selects the field that enables the BPC write signal. Figure 4-24 shows the write enable logic.

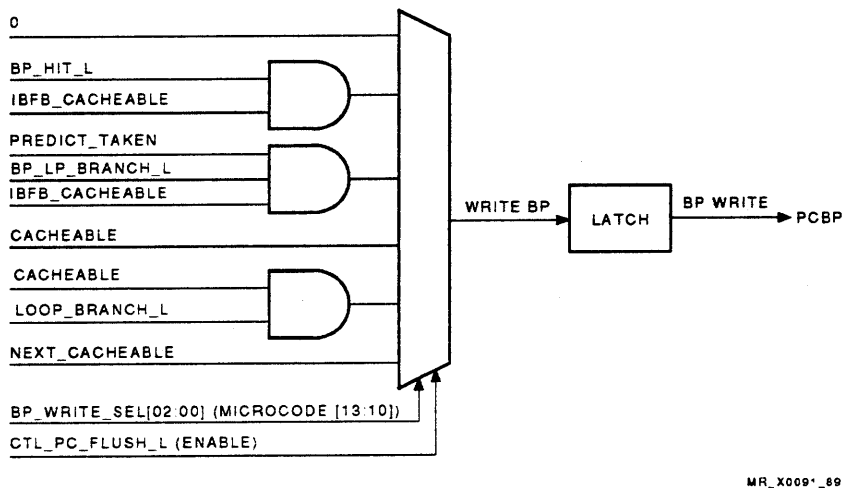


Figure 4-24 BPC Write Enable

4.3.3.2 Cache Tag Write

Figure 4-25 shows the BP tag address write logic. To write the tag, either the branch PC, decode PC, or second branch PC is selected at the BP tag multiplexer. Selection is as follows:

- Decode PC is selected if the branch is still in the instruction buffer. The branch remains in the instruction buffer if the address must be read from or written to.
- Branch PC is selected after the branch has been shifted out of the instruction buffer. The virtual address of the branch is loaded into the branch PC.
- Second branch PC is used when the PCU is processing two branches. This address can be loaded directly into the BPC or into the branch PC when the first branch address is no longer needed.

The selected PC provides bits [31:10] of the virtual address of the branch to be written to the BPC tag field.

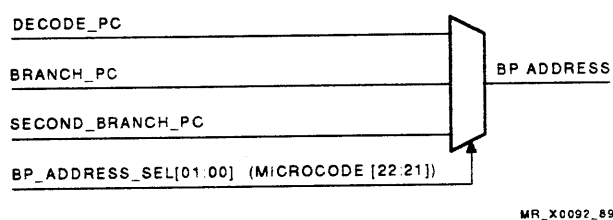


Figure 4-25 BPC Tag Write

4.3.3.3 Instruction Length Field Write

The instruction length field is loaded from the delta PC. This PC is derived from the accumulated shift counts since the last SHIFT_OPCODE_H. This value, when the instruction is shifted out of the instruction buffer, yields the instruction length.

Figure 4-26 shows the logic generating the delta PC. The delta PC represents the current instruction length or, if a second branch is being processed, it is selected as the second branch instruction length.

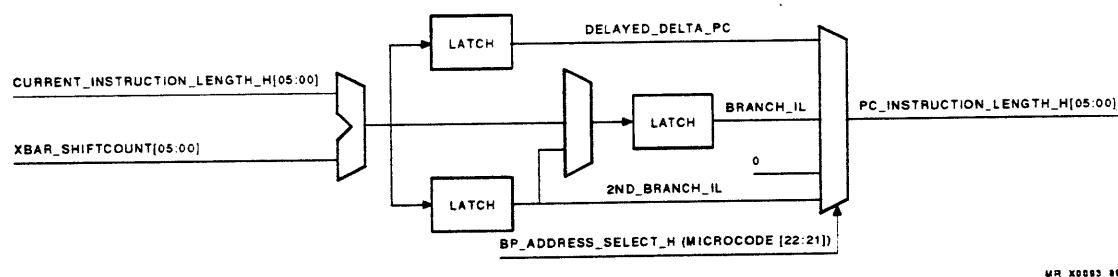


Figure 4-26 BP Instruction Length

4.3.3.4 Prediction PC Write

Figure 4-27 shows the inputs that are written to the BP prediction PC field.

The time when the PCU writes the prediction PC field depends on the prediction of the branch. When a branch is predicted not taken, the field is written when the branch is shifted out of the instruction buffer. The value that is written for a branch predicted not taken is unpredictable. The field that is written is not valuable because the branch is predicted not taken the next time it is encountered.

A branch that is predicted taken must wait for the target PC to arrive from the CSU before it can be written.

When a branch is predicted incorrectly, the BPC must be updated with the correct information. In this case, the unwind PC supplies the correct prediction PC.

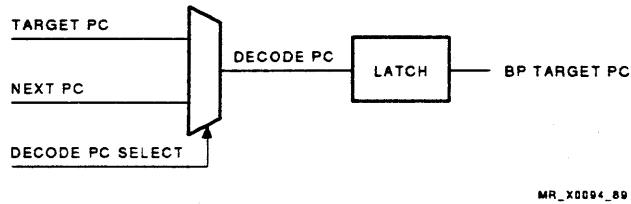


Figure 4-27 BP Prediction PC Write

4.3.3.5 BP Displacement Write

The branch displacement is calculated by the XBAR and, when the branch is cacheable, is written to the BPC. Figure 4-28 shows the logic that selects from two inputs to be written to the BPC.

XBAR_DISPLACEMENT_H[15:00] is written to the BPC or latched as SECOND_BRANCH_DISPLACEMENT_H[15:00] when two branches are being processed.

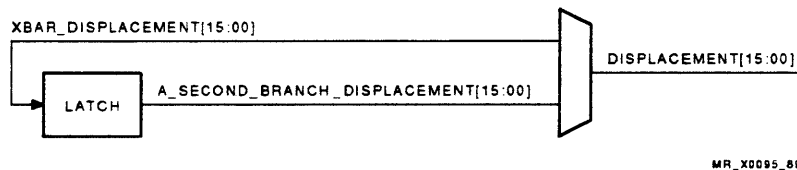


Figure 4-28 BP Displacement Write

4.3.3.6 BP Prediction Bit

The BP prediction bit is the bit that is stored in the BP cache and determines the history of the branch. This bit is the reference to whether the branch was predicted taken or predicted not taken the last time it was encountered.

This section provides a detailed description of the reading and writing of the prediction bit.

4.3.3.6.1 BP Prediction Bit Read

Two inputs supply the BPC prediction bit during a read operation. (refer to Figure 4-29)

- **BPP_PREDICT** - the history bit that is stored in the BPC.
- **XDTB_BRAM_BIAS** - the prediction bit supplied by the BRAM bias logic. This prediction is the fixed prediction that is based on the opcode of the branch instruction.

The selection between these two inputs depends on if the branch prediction tag match occurs in the BPC. When a tag match occurs, the prediction bit from the BPC is used. When there is no tag match in the BPC or the BPC cannot be read because it is currently being written, the branch prediction is supplied by the branch bias logic.

Loop branches and unconditional branches assert **FORCE_OUTPUT** which always selects **PREDICT_TAKEN**.

4.3.3.6.2 BP Prediction Bit Write

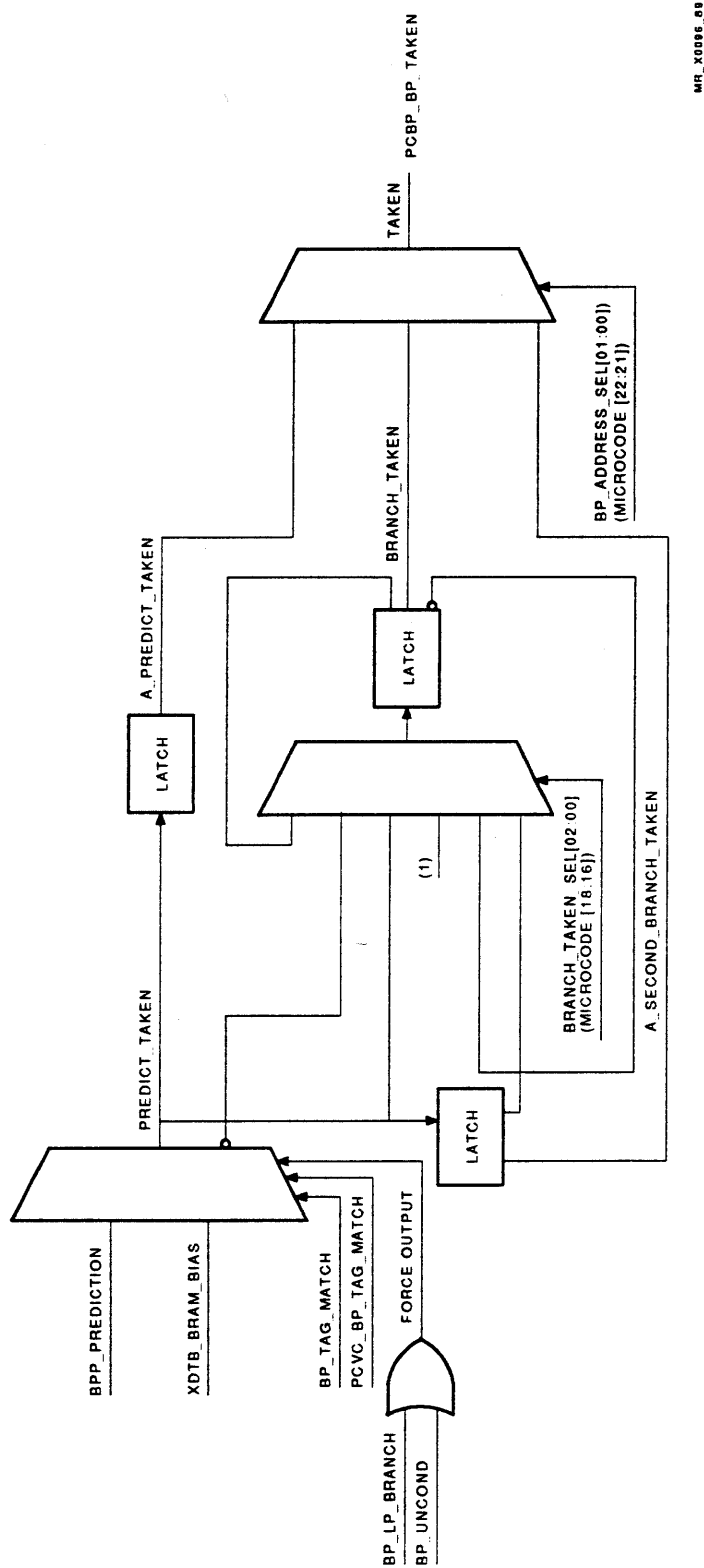
The prediction bit is written to the BPC when the branch is initially encountered and may also require being rewritten (inverted) when a branch prediction is determined to be incorrect. The logic that writes the prediction bit can write a prediction bit for a second branch prediction while the preceding branch is still under evaluation.

In Figure 4-29, **TAKEN** or **PCBP_BP_TAKEN** is the output that is used to access the BPC history bit during read and write operations. For reads, **BP_ADDRESS_SEL_H[01:00]** (PCU microcode bits [22:21]) selects **A_PREDICT_TAKEN** to access the BPC. For writes to the BPC, **BP_ADDRESS_SEL_H[01:00]** selects either **BRANCH_TAKEN** or **A_SECOND_BRANCH_TAKEN**. Either of these fields is selected when writing the BPC on a miss, or for rewriting the BPC on a correction or an unwind.

BRANCH_TAKEN is selected from one of five different fields by **BRANCH_TAKEN_SEL_H[02:00]**. The selection of the source for this field depends on timing and different states or conditions related to the branch instruction. The following list describes the five sources and when they are used:

- The previous output is selected during a noop.
- The inverted output of **PREDICT_TAKEN** is selected for a correction.
- **PREDICT_TAKEN** is selected for the write path when the branch misses in the BPC and it is an unconditional branch.
- On a bad not taken prediction, rewriting the BPC is done by selecting the field that forces the history bit to be asserted (shown as 1 in Figure 4-29).
- An unwind uses the latched and inverted version of **BRANCH_TAKEN** to rewrite the history bit.
- When two branches are being processed, **PREDICT_TAKEN** is latched and then used to rewrite the history bit if it is required (correction or unwind occurs).

BRANCH_TAKEN_SELECT[02:00] controls the selection of the write field and is supplied by the PCU microcode bit [18:16].



MR_X0086_00

Figure 4-29 BP Prediction Bit Write

4.3.3.7 BPC Address Selection

On read and write references, the BPC is addressed by PCBP_BP_ADDRESS[05:00]. When reading the BPC, DECODE_PC[09:00] provides the address (Figure 4-30). When writing the BPC, BRANCH_PC[09:00] is used. BRANCH_PC is also used to address the BPC when rewriting the BPC during a correction or an unwind.

When a second branch is to be written before the first is validated, the decode PC is latched and selected as SECOND_BRANCH_PC[09:00] to address the BPC.

The ten bits of the address are supplied by two of the PCU MCAs:

PCBP provides bits [05:00].

PCVC provides bites [09:06].

The field that selects the BP address (BP_ADDRESS_SEL[01:00]) is provided by the PCU microcode bits [22:21].

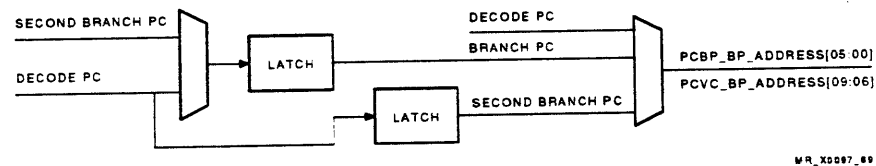


Figure 4-30 BPC Address Selection

5

Specifier Decode

This chapter describes the logic representing the IBox specifier decode pipeline stage. It also describes the three specifier handlers; the interfaces to the MBox, EBox, and VBox; PC generation; and related stalls and flushes.

5.1 Overview

The XBAR decodes macroinstructions and passes individual specifiers to the OPU MCU. Within the OPU, the specifiers are processed by a specifier handler and passed as operand data to the EBox or passed as an operand address to the MBox.

The OPU MCU contains three specifier handlers:

- **Complex specifier unit** — The CSU handles all specifiers other than short literal and register specifiers.
- **Short literal unit** — The SLU expands the short literal and floating short literal specifiers into a format and passes them to the EBox source list.
- **Free pointer logic** — The FPL manages the pointer to the Ebox source list. The free pointer points to the next location in the source list that the IBox can write to. Source and destination pointers are passed through this unit to the EBox source and destination pointer queues.

The following list describes the processing of the instruction ADDL3 R0 #54 R5 in the specifier decode pipeline stage. The example is based on the XBAR having decoded the instruction in a single cycle.

- The FPL receives R0, register valid, and source 1 valid for the first instruction specifier.
- The SLU receives the short literal data (54) and short literal valid bit. The FPL receives source 2 valid and register not valid.
- R5, the register valid bit, and the destination valid bit are sent to the FPL for the destination specifier.
- OCTL receives a 31-bit register mask indicating R0 will be read and R5 will be written.

From these inputs, pointers for each of the specifiers are passed to the EBox in the following manner:

- The source 1 logic passes a 5-bit field to the source pointer queue. The bit field indicates that the operand is a register and contains the register number.
- The short literal specifier (source 2) is expanded by the SLU and passed to the EBox source list. The free pointer provides the address in the source list that will receive the operand. The 5-bit source 2 pointer is placed in the source pointer queue and contains the location in the source list of the short literal operand.
- The free pointer is incremented by 1. This sets the free pointer to the next free location in the source list.
- The destination pointer is passed to the destination pointer queue and contains a 5-bit field indicating that the destination is a register and indicating the register number.
- The register mask is stored in the register mask logic and is not discarded until the instruction is executed by the EBox.

If the destination specifier were a register-deferred specifier, it would be processed by the CSU. When processing ADDL3 R0 #54 (R5), the two source specifiers would process the same as previously described. The destination specifier (R5) would be processed by the CSU. To process this specifier, the CSU sends the contents of R5 (the destination address) to the MBox write queue and the destination pointers to the EBox destination queue. The destination pointer contains a bit signifying a memory destination.

For the instruction CLRW @(R4)+[R5], the CSU performs the following steps:

1. Sends an OPU port request to the MBox to return the contents of the address supplied by R4 and autoincrements R4 by four.
2. Multiplies the contents of the index register (R5) by two and then adds the data returned from the MBox to the contents of R5. The result of the addition produces the address of the operand which is sent as an OPU request with the data being returned to the EBox source list.

5.1.1 Stall Logic

Figure 5-1 shows a block diagram of the stall buffer that each of the specifier handlers has to buffer the inputs from the XBAR. These stall circuits enable the specifier handlers to receive two specifiers from the XBAR before they inform the XBAR that it must stall.

The stall buffer consists of two scan latches, a latch, and a multiplexer. The multiplexer provides selection of input data or stalled data to be processed by the specifier handler. This data is held in a scan latch until the present specifier sequence is completed. The present specifier sequence is represented by current data and can also be stalled data if the sequence requires multiple cycles or if the specifier handler has stalled.

At the start of a specifier sequence, the input data is selected by the following:

OPU_SEQUENCE_START_H in the CSU
 SL_SEQUENCE_START_H in the SLU
 FPL_STALLED_L in the FPL

The input data is selected as current data and passed to the functional units of the specifier handler. Current data is also passed through the latch and scan latch, and placed on the input of the multiplexer. Any subsequent specifier cycles for this operation will continue to select stalled data at the multiplexer until the next sequence start.

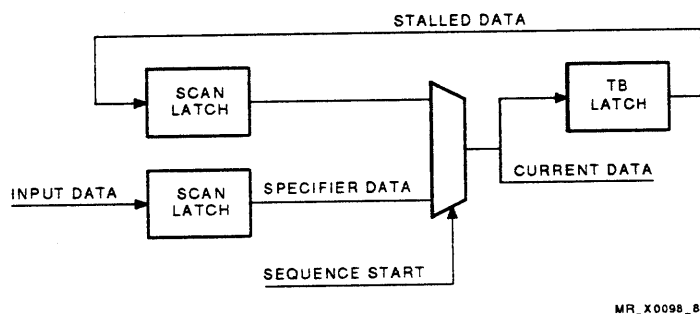


Figure 5-1 OPU Stall Logic

5.2 Complex Specifier Unit

The complex specifier unit (CSU) provides the following functions:

- Handles complex specifiers.
- Calculates operand addresses and branch target addresses.
- Controls the MBox OPU port.
- Provides IBox data control to the EBox.
- Performs GPR and RLOG write operations.

The CSU is contained in the OPU MCU and is divided across five MCAs. The MCAs provide the following functions:

- **OPUA** — Low word of the data path
- **OPUB** — High word of the data path
- **OSQA** — Control unit for RLOG, GPRs, OPUA, OPUB, and the OPU port and EBox interfaces
- **STG2** — Low word of the GPRs
- **STG3** — High word of the GPRs

Figure 5-2 is a basic block diagram showing the organization of the CSU.

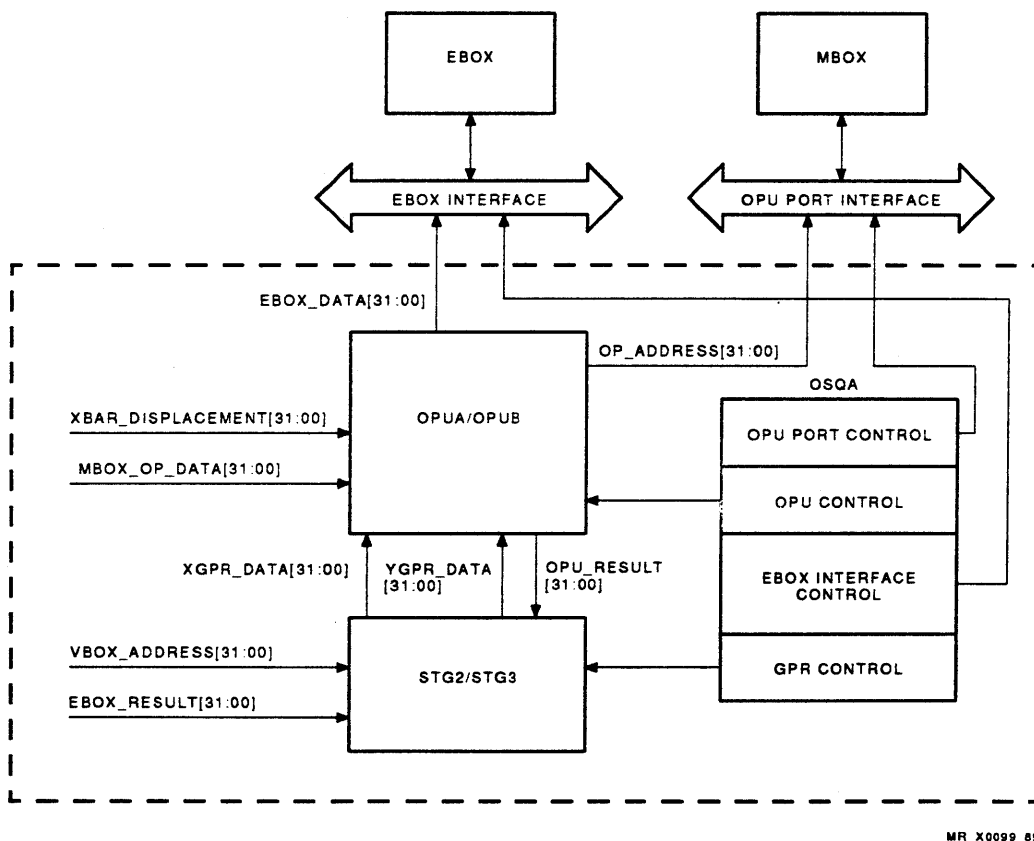


Figure 5-2 CSU Organization

The OPUA and OPUB MCAs contain the adders that calculate the target addresses of operands for the EBox. A context shifter (multiplier) is contained in the OPUA logic. The OPU port and the EBox interfaces are physically contained in these two MCAs and receive control from OSQA and OSQB.

The GPRs are dual port STREGs that can be read from and written to under control provided by OSQA. These GPRs can also receive inputs from the VBox. The VBox sends addresses to the MBox through the OPU.

5.2.1 OPUA Data Path

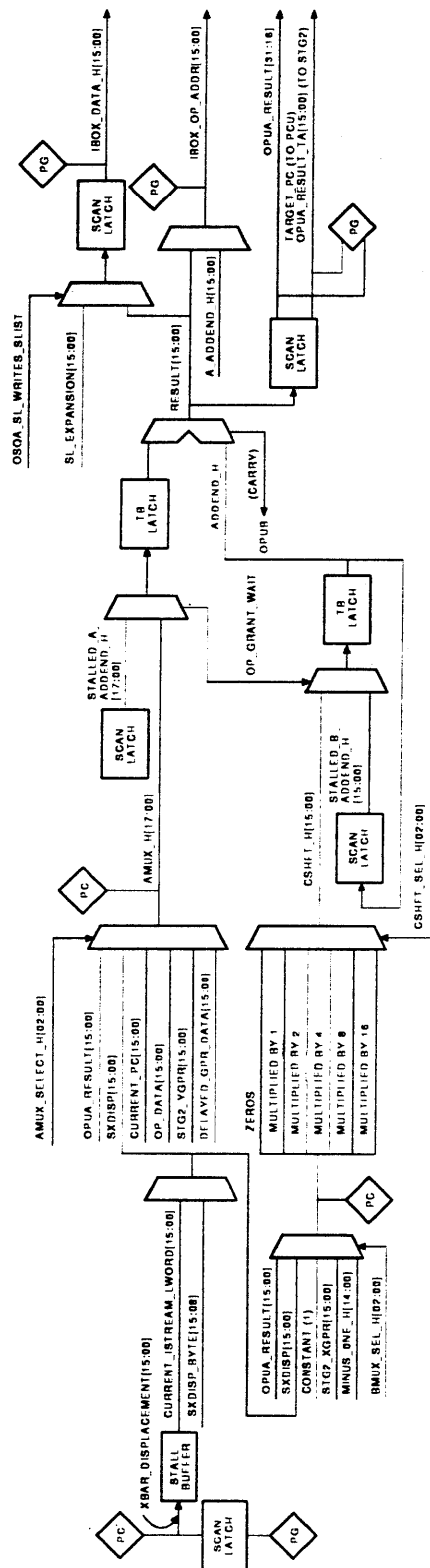
The OPUA MCA controls bits [15:00] of the address and data path of the CSU. OPUA contains an adder that receives inputs that are added together to calculate the operand data or address. Two multiplexers, AMUX and BMUX, supply the data to the adder from a variety of sources. When processing a displacement mode specifier, the AMUX provides the register (YREG) to the adder and the BMUX provides the sign-extended data to the adder. The adder adds the two inputs together and produces the operand address. (For example, OPUA produces the low 16 bits of the address while OPUB produces the high 16 bits.)

OPUA receives displacement data from the XBAR, sign extends the data, and presents it to the adder of the MCA as an operand. OPUA also accesses the GPRs. The CSU microcode selects the relevant operands for the specifier that is currently being evaluated and adds them together. The output of this operation can be used as an address for an OPU port request, passed as data to the EBox, sent to the PCU as a target PC, or placed on the input of the adder for the next cycle operation. Figure 5-3 shows a block diagram of the data and address paths of the OPUA MCA.

The AMUX receives XBAR_DISPLACEMENT_H[15:00], the low 16 bits of the 32 bits of XBAR displacement. This input (XBAR_DISPLACEMENT_H[15:00]) represents the following:

- Byte, word, or longword displacement for addressing modes A through F
 - A Byte displacement
 - B Byte displacement deferred
 - C Word displacement
 - D Word displacement deferred
 - E Longword displacement
 - F Longword displacement deferred
- Immediate data for addressing mode 8F
- Absolute address for addressing mode 9F
- Byte, word, or longword displacement for relative addressing modes (AF through FF)
- Byte or word displacements for branch instructions

XBAR_DISPLACEMENT_H[15:00] is sign extended to produce SXDISP_H[15:00], which is then placed on the input of the two multiplexers (AMUX and BMUX) that provide the data to the adder circuit.



MM_50102_00

Figure 5-3 OPUA

5.2.1.1 AMUX Inputs

The AMUX receives six inputs, one of which is selected by the OPU microcode (AMUX_SELECT_H[02:00]) and passed as one half of the operand to the adder. The inputs to the AMUX are as follows:

- **SXDISP[15:00]** — XBAR displacement that has been sign extended.
- **CURRENT_PC_H[15:00]** — The current updated PC. This PC is generated by adding XBAR_DECODE_PC and DECODE_DELTA and is used for calculating branch target PCs and in PC relative addressing.
- **OPU data** — Data that has been returned from the MBox in response to an OPU port request when processing an indirect addressing mode specifier.
- **STG2_YGPR_H[15:00]** — YGPR is the GPR that is being referenced in the present specifier cycle. YGPR represents any GPR reference except for indexed specifiers.
- **Delayed GPR data** — YGPR data that has not been written to the EBox GPRs. Delayed GPR data is used when processing intra-instruction read conflicts (IRCs).
- **OPUA result** — The result of the last addition (adder output) that is placed back on the input of the AMUX for further processing.

5.2.1.2 BMUX Inputs

The BMUX receives six inputs, one of which is selected by the OPU microcode (BMUX_SELECT_H[02:00]) and passed as the operand to the context shifter and then the adder. The inputs to the BMUX are as follows:

- **OPUA result** — The result of the last addition (adder output) that is placed back on the input of the BMUX for further processing.
- **SXDISP[15:00]** — Displacement data provided by the XBAR.
- **STG2 XGPR** — The input to the BMUX to represent which register is being accessed for an indexed operation.
- **Constant -1** — Selected for autodecrement operations.
- **Constant +1** — Selected for autoincrement operations.

The BMUX output is passed to the context shifter, which multiplies the output by 0, 1, 2, 4, 8, or 16. The multiplication type is selected by the OPU microcode (CSHFT_SEL_H[02:00]) for index, autoincrement, autodecrement, and autoincrement-deferred operations. BMUX inputs are passed unchanged through the CSHFT logic by selecting zeros as the function.

5.2.1.3 Adder

The output of the AMUX multiplexer (AMUX_H[15:00]) is input to a 16-bit adder and added with the output of the context shifter (CSFT_H[15:00]) to produce RESULT_H[15:00]. Any carry produced from the addition is sent to a similar add circuit in OPUB.

The output of the OPUA adder (RESULT_H[15:00]) is passed to one of the following signals to the other functional units of the MBox, EBox, and IBox:

- IBOX_DATA_H[15:00] is passed to the EBox source list as an operand.
- IBOX_OP_ADDRESS_H[15:00] sends result data to the MBox as the address of an operand with data being returned to the IBox or to the EBox source list, or as the destination address of an operand where the EBox is to write the destination data.
- OPUA_RESULT_H[15:00] is selected as the output and sent to the PCU as the target address of a branch instruction. This output (OPUA_RESULT_H[15:00]) can also be selected and written to a GPR (STG2) that was updated (autoincrement or autodecrement operation). OPUA_RESULT_H[15:00] can also be selected as an input to AMUX or BMUX for subsequent add operations.

5.2.2 OPUB Data Path

The OPUB MCA performs functions similar to OPUA. This MCA receives the high-order bytes of XBAR displacement data (XBAR_DISPLACEMENT_H[31:16]) and produces the high-order bytes of the operands or operand addresses and GPR update data.

Figure 5-4 shows a block diagram of the OPUB MCA. OPUA contains an AMUX, a BMUX, and a context shifter. As in OPUA, the output of the AMUX is added to the output of the BMUX and context shifter to produce the operand address or the operand data. OPUB uses two adder circuits. The two outputs they produce are “add with carry” and “add without carry.” (For example, the high-order 16 bits of the operand are added together, producing an output that assumes a carry and a no carry from the low-order addition.) The output of the two adders are placed on a multiplexer and selected by the carry bit from the adder of OPUA (OPUA_ADDER_COUNT_H).

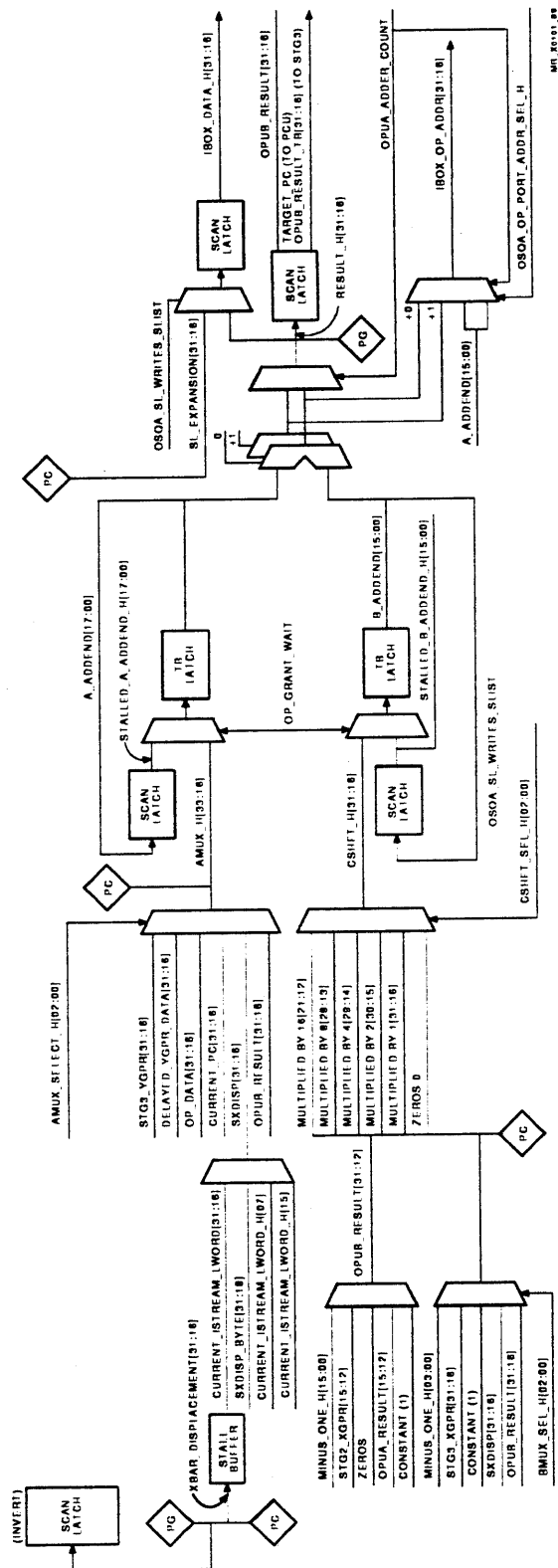


Figure 5-4 OPUB

5.2.3 Current PC Generation

The CSU calculates the current PC for branch instruction target addresses and for PC relative addressing mode specifiers. The current PC generation logic is divided between the OPUA and OPUB MCAs. OPUA generates the low-order word of the current PC while OPUB generates the high-order word. This section provides a detailed description of the current PC logic.

Figures 5-5 and 5-6 are detailed block diagrams describing the current PC generation in the CSU.

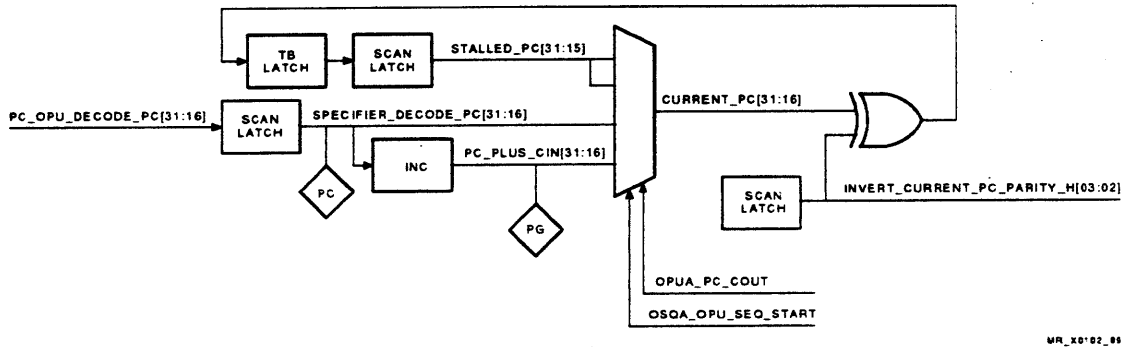


Figure 5-5 CSU Current PC (High Slice)

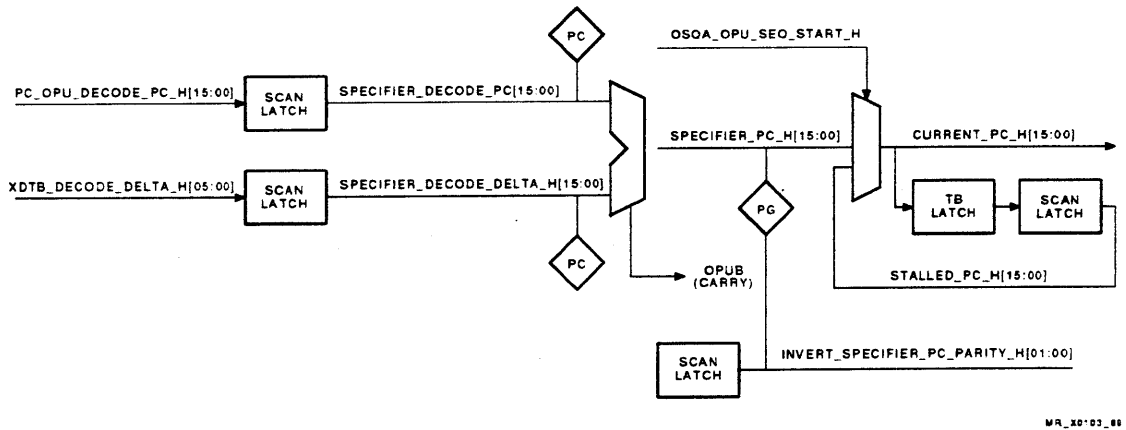


Figure 5–6 CSU Current PC (Low Slice)

5.2.3.1 OPUA Current PC [15:00]

OPUA receives PCxx_OPU_DECODE_PC_H[15:00] from the PCU. These 16 bits are delivered in the following manner:

- PCBP sends bits [07:00].
- PCVC sends bits [12:08].
- PCLO sends bits [15:13].

The XBAR sends XDTB_DECODE_DELTA_H[05:00], which equals the total number of instruction buffer bytes that have been shifted out of the instruction buffer since the last SHIFTOPCODE was asserted.

These two signals are latched and then added (Figure 5-6) together to produce SPECIFIER_PC_H[15:00], which is latched and produces an output of CURRENT_PC_H[15:00]. A carry from the addition results in a carry bit being asserted in the logic that generates the high-order word of the current PC. CURRENT_PC_H[15:00] is an AMUX input in the OPUA data path of the CSU.

This logic also contains a stall buffer that controls selection of a current PC output or a stalled output (STALLED_PC_H[15:00]). The stalled PC is selected when the CSU is in sequence.

5.2.3.2 OPUB Current PC [31:16]

The high-order word for the current PC (CURRENT_PC_H[31:16]) is generated in OPUB and is an input to the AMUX in the OPUB data path of the CSU. Figure 5-5 shows a block diagram of the logic that generates the high-order word of the current PC.

OPUB receives PCxx_OPU_DECODE_PC_H[31:16] from the PCU. These 16 bits are delivered in the following manner:

- PCHI supplies bits [31:24].
- PCLO supplies bits [23:16].

These two PC fields are latched and produce SPECIFIER_DECODE_PC_H[31:16], which is input to a multiplexer for final selection of the current PC. Also input to the multiplexer is an incremented version of SPECIFIER_DECODE_PC_H[31:16] (PC_PLUS_CIN_H[31:16]). The incremented PC is selected when the calculation of the low-order word of the current PC results in a carry.

This logic also contains stall circuitry similar to that in the logic that generates the low-order word of the current PC.

5.2.4 CSU Microcode

The CSU is controlled by microcode logic that is resident on the OSQA MCA. The microcode controls the multiplexers that supply inputs to the adders of the CSU. Control of the OPU port interface, GPR writes, YGPR incrementing, and source list writes is also provided under microcode control.

5.2.4.1 CSU Microaddress

The CSU microaddress is 10 bits wide and contains 4 fields that govern the flow of the microcode (Figure 5-7). Table 5-1 lists the fields of the microaddress.

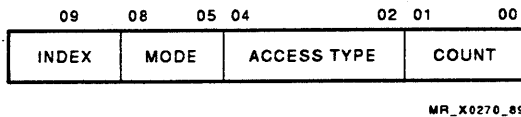


Figure 5-7 CSU Microaddress Format

Table 5-1 CSU Microaddress Descriptions

Index [09]	Indexed Indication	Label
0	Nonindexed specifier	NI
1	Indexed specifier	I

Mode [08:05]	Specifier Mode	Label	Original Mode Values
0000	Autoincrement	AINC	8x
0001	Autoincrement deferred	AINCDEF	9x
0010	Displacement	DISP	Ax, Cx, Ex
0011	Displacement deferred	DISPDEF	Bx, Dx, Fx
0100	Unused	4	0x, 1x, 2x, 3x, 4x
0101	Register	REG	5x ¹
0110	Register deferred	REGDEF	6x
0111	Autodecrement	ADEC	7x
1000	Immediate	IMM	8F
1001	Absolute	ABS	9F
1010	Relative	REL	AF, CF, EF
1011	Relative deferred	RELDEF	BF, DF, FF
1100	Unused	12	0F, 1F, 2F, 3F, 4F
1101	Unused	13	5F
1110	PC deferred	PCDEF	6F
1111	PC autodecrement	PCADEC	7F

Access Type [04:02]	Specifier Access Type	Label
000	Address (ASRC)	A
001	Read	R
010	Write	W

¹This specifier mode is selected when IRC is asserted in the XBAR.

Table 5-1 (Cont.) CSU Microaddress Descriptions

Access Type [04:02]	Specifier Access Type	Label
011	Modify	M
100	Yield source (VSRC)	V
101	Branch displacement	B
110	Implied	I
111	Callx specifier	C

Count [01:00]	Microword in Sequence	Label
00	First microword	0
01	Second microword	1
10	Third microword	2
11	Fourth microword	3

Register Mode Count [01:00]	Register Mode Size	Label
00	Longword data type	0
01	Octaword data type	1
10	Three longwords	2
11	Quadword data type	3

5.2.4.2 CSU Microword

The CSU microword is 19 bits wide and is divided into 12 fields (Figure 5-8). Table 5-2 lists the fields of the microword.

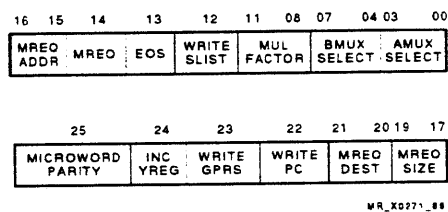


Figure 5-8 CSU Microword Format

Table 5-2 CSU Microword Field Descriptions

Bits	Name	Value	Description
03:00	AMUX select	000	Previous cycle's adder output.
		001	Sign-extended I-stream data.
		010	Current specifier's PC.
		011	OPU port data returned to OPU.
		100	Contents of specifier's base GPR.
		101	Contents of current delayed GPR.
07:04	BMUX select	000	Previous cycle's adder output.
		001	Sign-extended I-stream data.
		010	Constant value of one.
		100	Contents of specifier's index GPR.
		101	Constant value of minus one.
11:08	Multiply factor	000	Multiply BMUX output by zero.
		001	Multiply BMUX output by one.
		010	Multiply BMUX output by two.
		011	Multiply BMUX output by four.
		100	Use specifier's context.
12	Write source list	0	Don't write adder result to source list.
		1	Write adder result to source list.
13	EOS	0	Specifier sequence continues.
		1	End of specifier sequence.
14	MREQ	0	Don't issue an MBox OPU port request.
		1	Issue an MBox OPU port request.
16:15	MREQ address select	0	Use adder result as OPU port address.
		1	Use AMUX output as OPU port address.
19:17	MREQ size	00	Use specifier's context as request size.
		01	Force request size to word.
		10	Force request size to longword.
		11	Force request size to quadword.
21:20	MREQ destination	0	Return data to OPU.
		1	Write data to EBox source list.
22	Write PC	0	Don't send target PC.
		1	Send target PC if appropriate.
23	Write GPRs	0	Don't write GPRs.
		1	Write IBox and EBox GPRs.
24	Increment YREG	0	Don't increment base GPR pointer.
		1	Increment base GPR pointer.
25	Microword parity	0	Parity bit disabled, microword has odd number of bits.
		1	Parity bit enabled, microword has even number of bits.

Figure 5-9 shows signals involved in generating a CSU microword. The following list describes the source of each signal used to generate the CSU microword:

- OSQB_ACCESS_TYPE_H[03:00] is from the OSQB DRAM and defines the access type of the current specifier.
- XDTB_INDEXED_H is from the XBAR and is asserted when the specifier is indexed mode.
- XDTB_MODE_H[03:00] is from the XBAR and provides the addressing mode of the specifier.
- XDTA_XREG_H[03:00] is from the XBAR and provides the base register for any register operands.

A count field is also input to the microaddress generation to control the count of the CSU sequence. The count field is generated by decoding the inputs to the microaddress generation logic.

5.2.5 CSU Stalls

In the CSU, the OSQA MCA contains the logic that monitors conditions that may initiate stalls. The stalls that occur in this unit can be related to handling certain specifiers (read and write conflict stalls) or related to the units that the CSU is supplying data to. This section describes the stalls that occur in the CSU, and it describes how each stall is detected and how each stall is cleared.

5.2.5.1 Scoreboard Stalls

The OCTL MCA contains the scoreboard logic that tracks the reading and writing of GPRs by the EBox during the execution of macroinstructions. Read and write conflicts occur when the IBox is directed to read or write a GPR before the EBox has performed a required operation on the same GPR. When these conflicts occur, the CSU must stall and wait for the EBox to complete the conflicting instruction before it can proceed.

Two types of read and write conflict stalls can be asserted in the CSU. The two types of stalls are caused by the same conflicts but differ only in the timing of the detection of the stall. For example, a read conflict can initiate a stall by asserting a SPECIFIER_READ_CONFLICT_H or a CURRENT_READ_CONFLICT_H. A specifier read conflict occurs when the conflicting data is latched from the XBAR but has not yet been acted on by the CSU. A current read conflict occurs after the CSU begins evaluating the data (OPU_SEQ_START_H is asserted).

This section describes the initiation of read conflict and write conflict stalls, and it describes the clearing of these conditions.

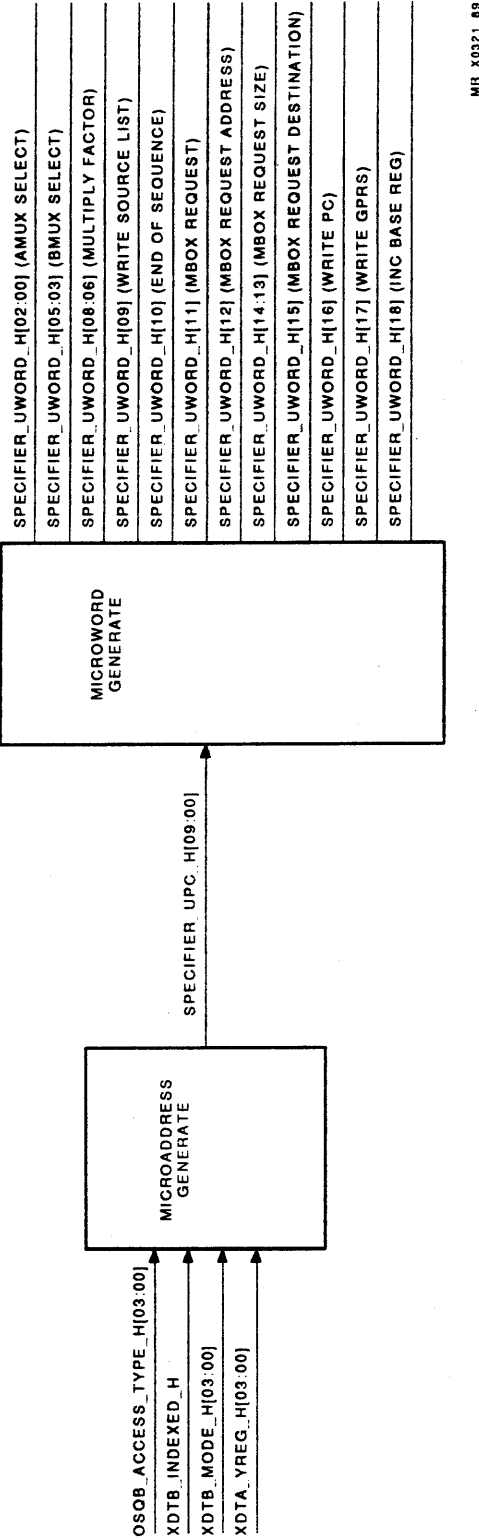


Figure 5-9 CSU Microword Select

5.2.5.1.1 Read Conflict Stall

Read conflicts are detected by OSQA and initiate scoreboard stalls. A read conflict occurs when the CSU is directed to update (autoincrement or autodecrement) a GPR that the EBox is directed to read. For example:

```
MOVL R0, xx (read R0)
ADDL2 yy, (R0)+ (autoincrement R0)
```

The above sequence of instructions causes a read stall because the EBox is directed to read R0 and the IBox is directed to update R0. Depending on how many instructions ahead of the EBox the IBox is, the IBox could possibly update R0 before the EBox reads R0.

Read conflicts are detected by monitoring the read mask (READ_MASK_H[15:00]), the specifier GPR (YREG_H[03:00]), and the occurrence of an autoincrement or autodecrement (AUTOXXX_MODE_H). When the read mask and YREG both point to the same register and an autoincrement or autodecrement is to be performed by the CSU, SPECIFIER_READ_CONFLICT_H and SPECIFIER_READ_STALL_H are asserted in OSQA. These stall signals also assert OPU_STALLED_H and SCOREBOARD_STALL_H.

The stall is cleared by a flush or unwind or when the instruction containing the conflict is executed by the EBox. When a flush or BP unwind occurs, INIT_OPU_AND_SL_H is asserted and any read or write conflicts are negated. When the EBox completes an instruction, EBOX_INSTRUCTION_DONE_H is asserted. This signals directs OCTL to discard the masks related to the completed instruction. If the mask that is discarded is the one that asserted the stall, then the CSU resumes operation.

5.2.5.1.2 Write Conflict Stall

Write conflicts are similar to read conflicts in the CSU. Write conflicts occur when the EBox is directed to write a GPR and the IBox is directed to read the same GPR.

Write conflicts are detected by monitoring the write mask (WRITE_MASK_H[15:00]) and the specifier GPR (YREG_H[03:00]) for register mode addressing. When a match of the write mask and a YREG is detected, a write conflict is asserted.

A write conflict can also occur when processing index mode specifiers. OSQA compares the index register (XREG_H[03:00]) with the write mask to detect this conflict.

Write conflicts assert CURRENT_WRITE_STALL_H or SPECIFIER_WRITE_STALL_H, which asserts OPU_STALLED_H to stall the CSU and asserts SCOREBOARD_STALL_H in the OCTL MCA. These stalls are cleared by an EBox flush, a PCU unwind, or when the EBox has completed the conflicting instruction and directs the OCTL mask logic to discard the write mask associated with it.

5.2.5.2 Branch Under Branch Stall

The CSU stalls when a conditional branch is encountered, if an outstanding branch has not yet been validated by the EBox. OSQA maintains a count of conditional branches that have been processed by the CSU. The unconditional branch count is incremented each time the CSU sends a target PC to the PCU (TARGET_PC_H[31:00], and TARGET_VALID_H) and the current instruction is an unconditional branch (CURRENT_UNCOND_BRANCH_TB_L negated).

The branch under branch stall (CURRENT_BUB_STALL_H) occurs when the following conditions exist:

- BRANCH_CNT_EQL_1_H is asserted.
- SPECIFIER_BRANCH_ACCESS_TYPE_H is asserted. This signal is from the OSQB DRAM access type.
- SPECIFIER_DATA_AVAILABLE_H is asserted. This signal is asserted when XSCA_VALID_H is asserted and XDTA_RAF_L is negated.
- SPECIFIER_UNCOND_BRANCH_L is asserted in the instruction buffer simple decode logic.
- OPU_SEQ_START_H is asserted.

CURRENT_BUB_STALL_H asserts OPU_STALLED_H. The stall is cleared by an EBox flush, PCU unwind, or when the first branch is validated by the EBox (EBOX_BRANCH_VALID_H).

5.2.5.3 AUTOxx Under Branch Stall

The CSU does not process autoincrement and autodecrement specifiers when an outstanding branch prediction is awaiting EBox validation. When an autoincrement, autodecrement, or autoincrement-deferred mode specifier is detected and the branch count is equal to one, the CSU stalls and waits for the branch prediction to be validated before it continues operation.

CURRENT_AUB_STALL_H is asserted to initiate this stall when the following conditions exist:

- SPEC_NOT_BRANCH_OR IMPLIED_H is asserted.
- SPECIFIER_DATA_AVAILABLE_H is asserted.
- SPECIFIER_AUTOXX_MODE_H is asserted.
- BRANCH_COUNT_EQL_1_H is asserted.
- OPU_SEQ_START_H is asserted.

Asserting CURRENT_AUB_STALL_H also asserts OPU_STALLED_H. The stall is cleared by an EBox flush, a PCU unwind, or when the EBox validates the outstanding branch prediction.

During this stall, it is possible for the CSU to initiate an OPU port request to handle the AUTOxx specifier. For an autoincrement specifier, the CSU can read the GPR and then write (autoincrement) the GPR when the stall is negated.

5.2.5.4 OPU Port Grant Wait Stall

When the CSU issues an OPU port request, the MBox must respond to the request before the CSU can continue processing. When the MBox does not respond, the CSU stalls. When an OPU port request is issued, the CSU selects stalled A and B addends as input to the CSU adder by asserting `OSQA_OP_GRANT_WAIT_H`. This holds the port request information until the MBox responds.

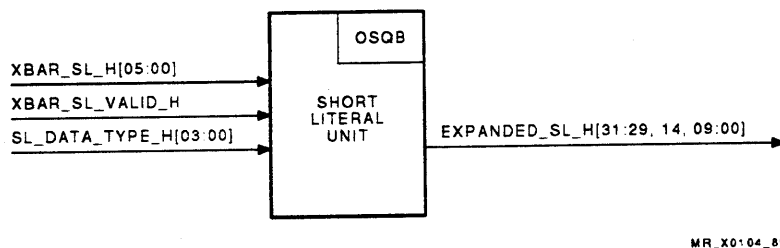
`OP_GRANT_WAIT_H` asserts `OPU_STALLED_H` if another OPU port request is issued. The stall is cleared when the MBox responds to the request by asserting `MBOX_OP_GRANT_TA_L` or if an EBox flush or PCU unwind occurs.

5.3 Short Literal Unit

The short literal unit (SLU) expands the 6-bit integer and floating short literal (SL) operands into a relevant field (32-, 64-, or 128-bit field).

Figure 5-10 shows the inputs and outputs of the SLU. This unit receives `XBAR_SL_H[05:00]` and a copy of the opcode (`OPCODE[08:00]`) from the XBAR. The opcode is decoded to determine the SL data type. The data type and the SL data are input to the SL data formatter, which outputs the expanded SL data.

The SLU can produce a single longword of expansion per cycle. Data formats larger than a longword require successive cycles to produce their output.



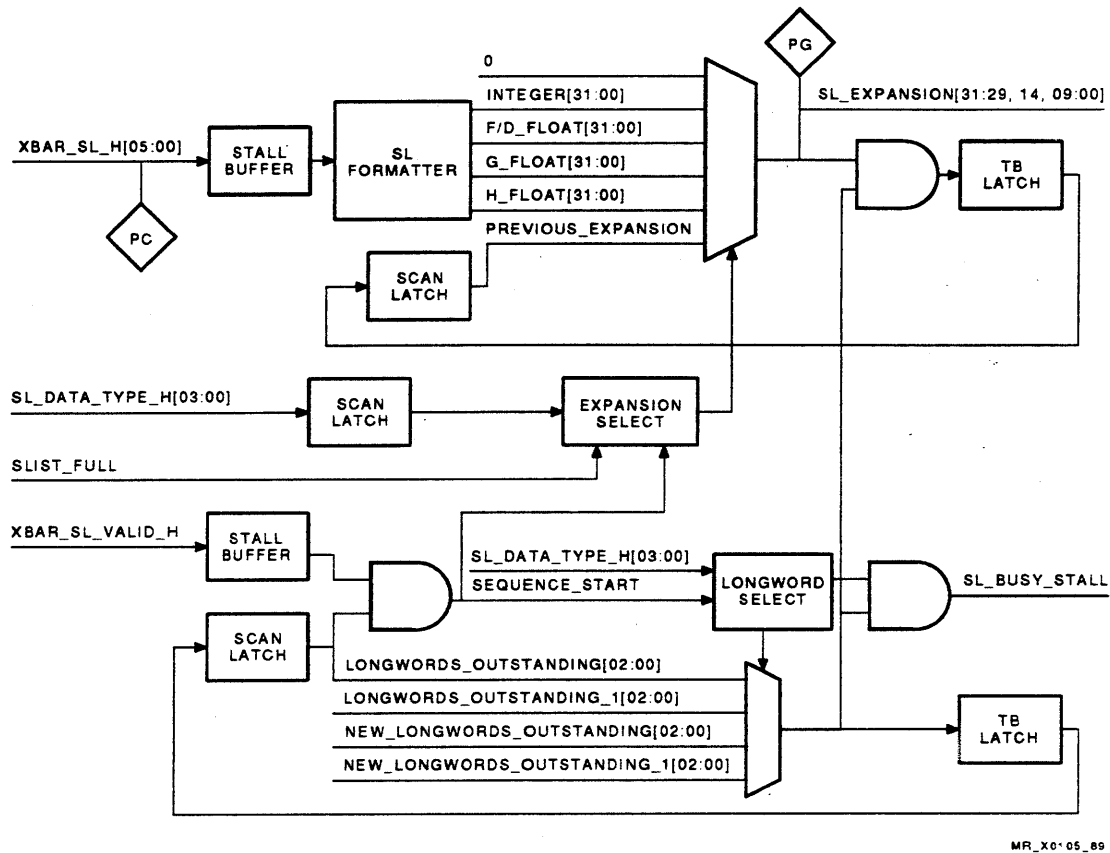
MR_X0104_89

Figure 5-10 Input and Outputs of the Short Literal Unit

5.3.1 Short Literal Processing

Figure 5-11 shows a block diagram of the SLU. The SL formatter provides integer and F-, D-, G-, and H-floating formats to the format select logic. The SL formatter receives the 6-bit short literal data (`XBAR_SL_H[05:00]`) and expands it into four formats (`INTEGER_H[31:00]`, `F/D_FLOAT_H[31:00]`, `G_FLOAT_H[31:00]`, and `H_FLOAT_H[31:00]`).

The expansion select logic selects one of the four SL formatter outputs, `PREVIOUS_EXPANSION_H[31:00]`, or zero. `PREVIOUS_EXPANSION_H[31:00]` is selected when short literal data that was previously expanded was not written to the EBox source list (for example, IBox-to-EBox interface is busy). Zero is selected in cycles other than the first in multiple cycle expansion (for example, D-, G-, and H-floating formats). The data type of the specifier being processed (`SL_DATA_TYPE_H[03:00]`) and a signal monitoring the status of the EBox source list (`SLIST_FULL_H`) are input to expansion select. The data type is generated by decoding instruction opcodes in the OSQB access type and data type logic and provides selection of the expanded short literal data.



MR_X0'05_89

Figure 5-11 Short Literal Unit Block Diagram

Short literal sequences start when `SL_SEQUENCE_START_H` is asserted. Assertion of this signal selects valid short literal data (`XBAR_SL_H[05:00]`) from the input stall buffer to be loaded into the SL formatting logic. `SL_SEQUENCE_START_H` is asserted when the following occurs:

- `XDTB_SL_VALID_H` is asserted, signifying that the XBAR is passing valid short literal data to the SLU.
- `XDTA_RAF_L` is negated, signifying that the XBAR did not detect an RAF.
- `SL_LWORDS_LEFT_TA_EQL_0_H` is asserted, signifying that no short literal specifiers are currently being processed.
- `PREV_SLIST_FULL_L` is negated, signifying that the source list is not full.
- `INIT_OPU_AND_SL_L` is negated, signifying that no flush or PCU unwind is in progress.

At the start of a short literal sequence, SL_LWORDS_H[02:00] is loaded into SL_LWORDS_LEFT_H[02:00]. SL_LWORDS_H[02:00] is originally loaded with the value or count that is derived from the data type of the current SL specifier being evaluated. For byte, word, and longword formats, the count equals one. For quadword formats, the count equals two; for octaword formats, the count equals four.

When CURRENT_SL_LWORDS_H[02:00] equals SL_LWORDS_LEFT_H[02:00], OSQA_SL_WRITES_SLIST_H is asserted to write expanded short literal data to the source list. Each time the write signal is asserted, SL_LWORDS_LEFT_H[02:00] is decremented. When the longword count is decremented to zero, the SL sequence is complete.

5.3.2 Integer Expansion

Integer short literal operands represent values from 0 to 63. The short literal operands are zero extended, according to their data type, and passed to the EBox source list.

Figure 5-12 shows the outputs of the SLU when processing an integer short literal operand. In this example, the zero expansion of 63 occurs in a single cycle for the byte, word, and longword contexts. Quadword outputs occur in a minimum of two cycles. Producing the octaword format requires a minimum of four cycles.

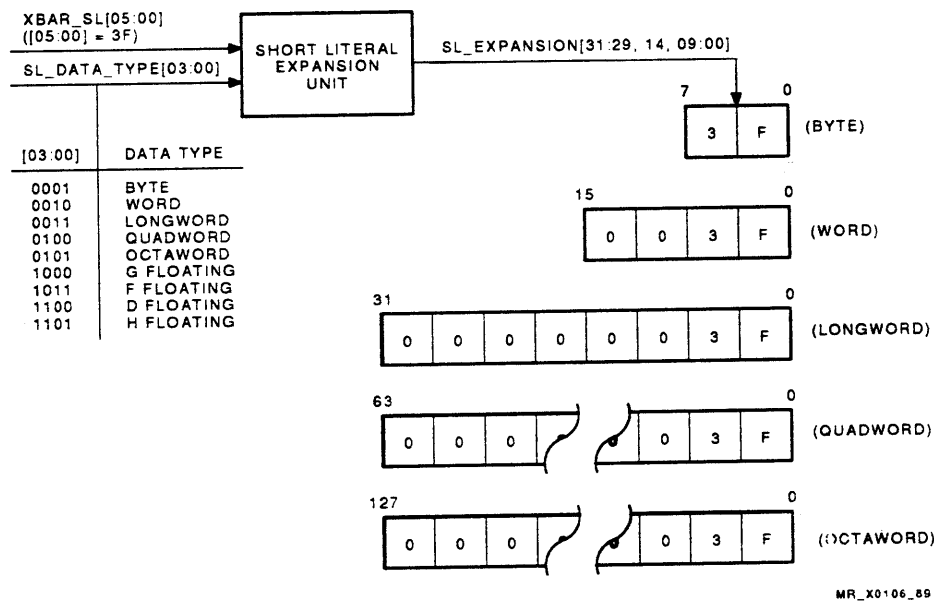


Figure 5-12 SLU Integer Expansion

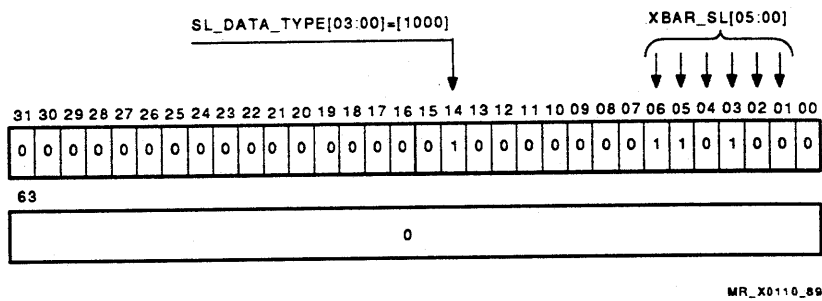


Figure 5-16 SLU G-Floating Expansion

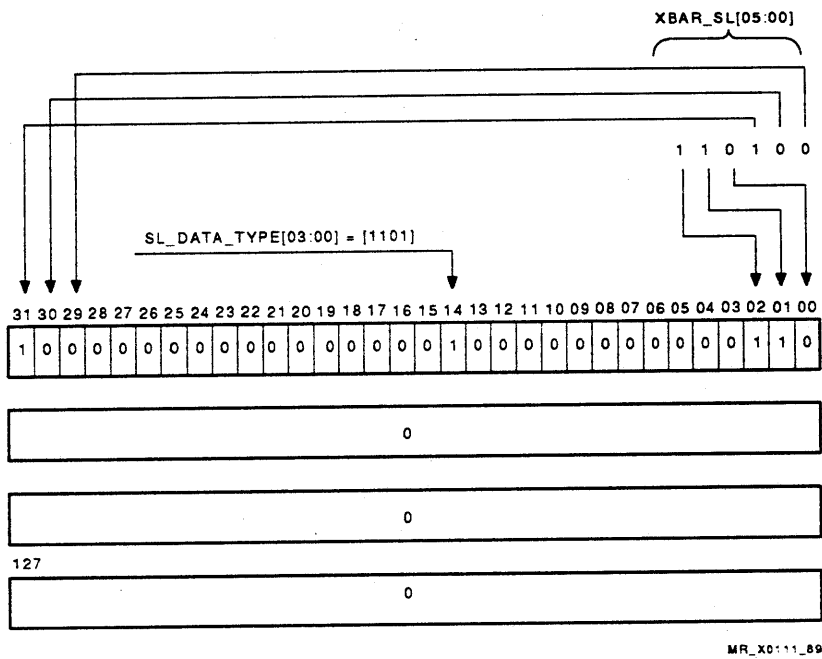


Figure 5-17 SLU H-Floating Expansion

5.3.4 Outputs to the EBox Interface

The 14-bit short literal operand is passed as 32 bits of short literal data to the EBox across the EBox interface. This field is constructed by zeroing out the remaining 18 bits as the data is passed to the EBox source list.

5.3.4.1 Order

When specifiers from the CSU and SLU are both valid, XBAR_SL_ORDER is used to keep the operand entries to the free pointer in order. When asserted, the short literal specifier precedes the complex specifier. This signal is generated by the XBAR.

5.3.5 Stalls

The SLU stalls for three reasons:

- The unit is processing a multicycle SL specifier.
- The EBox interface cannot accept the expanded SL data because the OPU is using the interface.
- The EBox queues are full.

5.3.5.1 Source List Full

The FPL tracks the available free slots in the source list. When the source list is full, OSQB_SLIST_FULL_H is asserted in the OSQB MCA and passed to the OSQA MCA. The receipt of this signal disables any SLU or CSU writes across the EBox interface to the source list.

5.3.5.2 SLU Stalled

The XBAR must be informed of the status of the SLU so that when the SLU is busy processing one specifier, another is not passed to it. OSQA_SL_BUSY_STALL_H is asserted and sent to the XBAR to signify that the SLU is busy processing a specifier. This signal is an output of the comparator, in the SLU, that tracks the number of longwords remaining in a SL expansion.

5.3.5.3 EBox Interface Output Stall

The EBox interface cannot pass both SL and CS data to the EBox simultaneously. If SL data is available to be written to the source list and the EBox interface is busy with CS data, SL_BUSY_STALL_H is asserted until the interface is free to accept the SL data.

5.3.6 Parity Coverage and Errors

Three key signals of the SLU are parity checked before the SL specifier is processed:

```
XDTB_SL_SPECIFIER_NUMBER_H[02:00]
XDTB_ORDER_H
XBAR_SL_H[05:00]
```

5.4 Free Pointer Logic

The free pointer logic (FPL) processes and passes pointers that are used by the EBox to direct the execution of instructions:

- Source 1 pointer
- Source 2 pointer
- Destination pointer
- Free pointer

The EBox maintains queues for the source and destination pointers it receives from the IBox. The source pointers point to entries in the source list or to GPRs. The source list is a queue for storing operands. These operands have either been passed by the IBox or prefetched from memory on behalf of the EBox.

To manage the source list, the IBox generates a free pointer. The free pointer points to the next free location in the source list that the IBox will write to.

The structure of the EBox queues is as follows:

- **Source pointer queue** — A 5-bit wide $\times$ 16-location deep queue. The top bit of the entry specifies if the operand is in the source list or a GPR. The remaining bits contain the address, in the source list, or the GPR number.
- **Destination pointer queue** — A 5-bit wide $\times$ 8-bit deep queue. The top bit in the queue specifies if the destination is memory or a GPR, while the remaining bits contain the GPR number when applicable.
- **Source list** — A 16-entry circular queue for memory, immediate, and short literal operands. Each time an entry is inserted into the source list, the free pointer must be incremented to point to the next free location. If the operand size is a quadword or octaword, the free pointer must be incremented to reflect the size.

Figure 5–18 shows a simplified block diagram of the FPL.

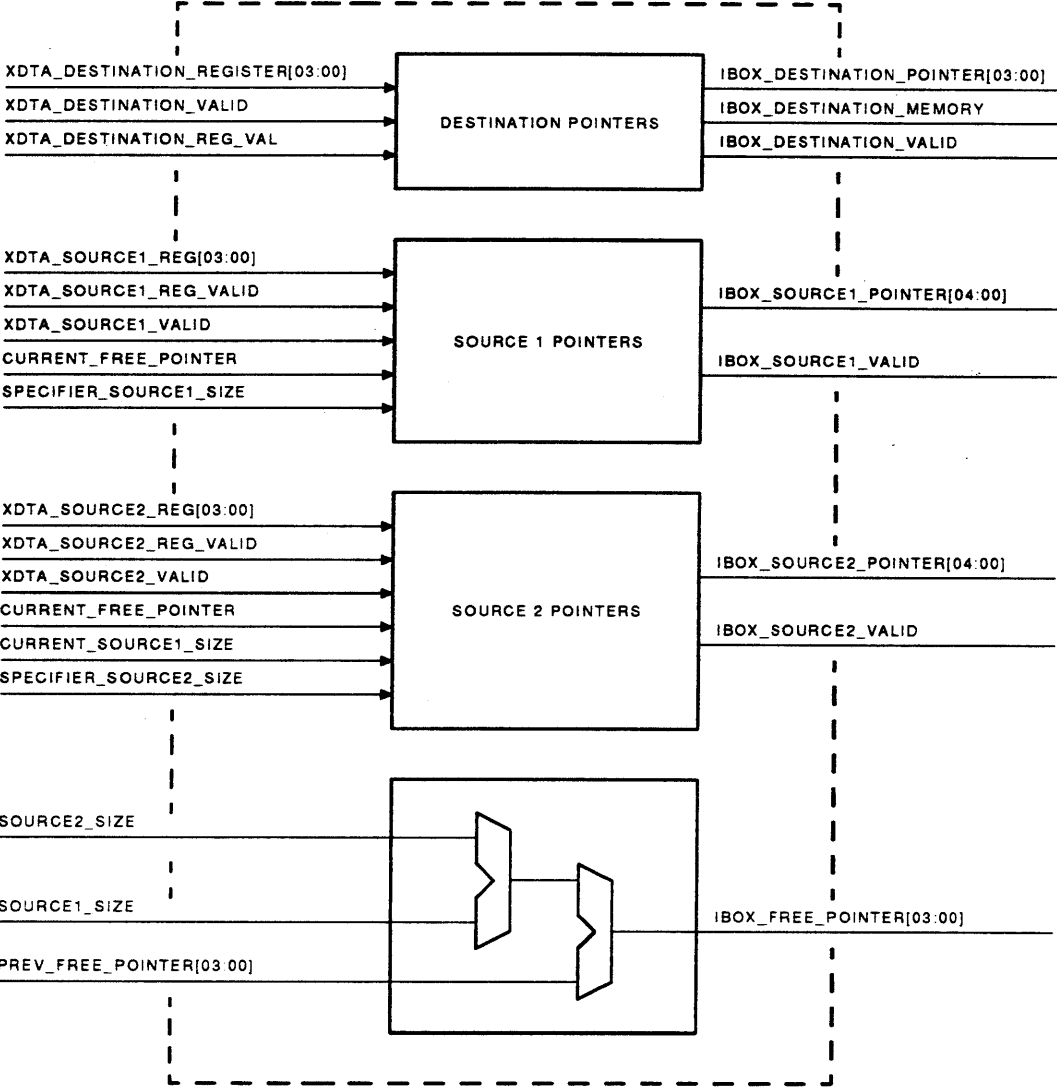
5.4.1 Source 1 Pointer

The source 1 pointer is generated and validated in XDTA and passed to the FPL logic in OSQB. Three signals deliver the source 1 information:

- XDTA_SOURCE_REG_H[03:00] is the register number containing the operand if it is a register operand.
- XDTA_SOURCE_REG_VALID_H is asserted when the operand is a register.
- XDTA_SOURCE_VALID_H is asserted when a valid operand is being passed.

These signals allow the FPL to generate a pointer, validate the pointer, and, if the operand is not a register, allocate the source list entry and update the free pointer.

Figure 5–19 shows a block diagram of the source 1 pointer logic. The source 1 register field (XDTA_SRC1_REG_H[03:00]) is placed on the input of the pointer select multiplexer, if no stalls are present.



MR_X0112_89

Figure 5-18 Free Pointer Logic

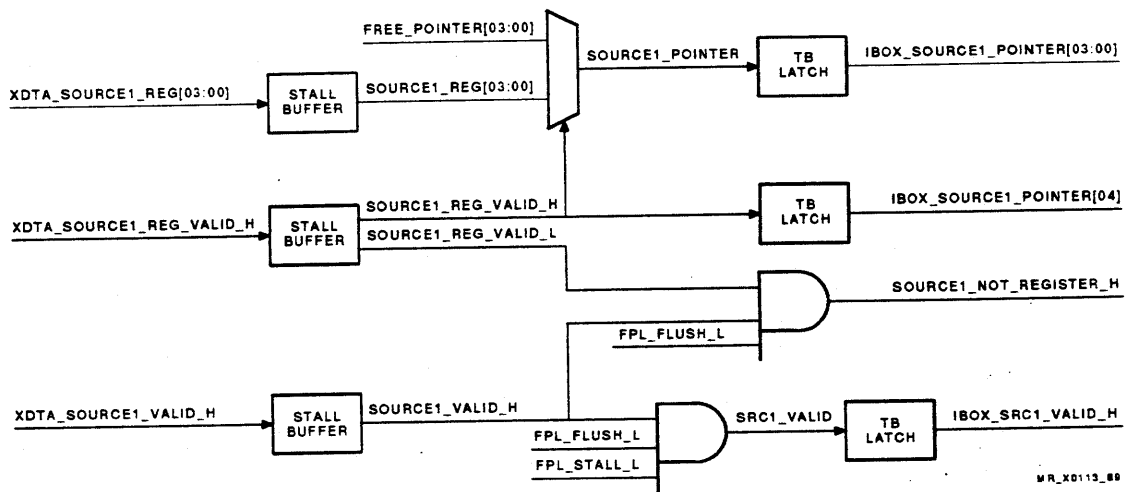


Figure 5-19 FPL Source 1 Pointer Logic

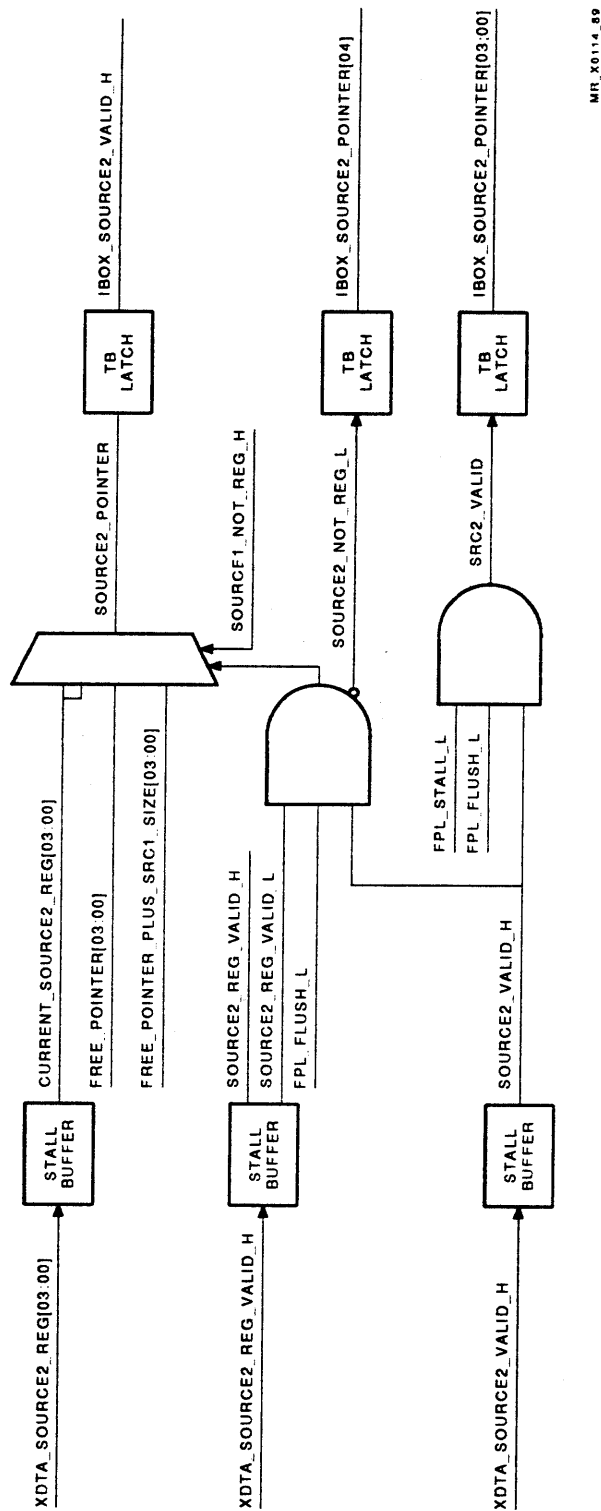
The register valid signal (XDTA_SRC1_REG_VALID) is latched from the XBAR and provides the select at the pointer multiplexer. Register valid asserted denotes a register operand and selects the register field to be entered in the source queue. If negated, the free pointer is selected and an entry is allocated in the source list for the source 1 operand.

Aside from providing a pointer select, the register valid signal is latched as bit 4 of the source 1 pointer and, when negated, generates SRC1_NOT_REGISTER_H. When asserted, SRC1_NOT_REGISTER_H initiates the process of incrementing the free pointer.

To pass a valid source 1 pointer, IBOX_SRC1_VALID_H must be asserted. This signal is passed from the XBAR, and, if no stalls or flushes are present, is passed to the EBox. This signal enables the EBox to enter the source 1 pointer into the source queue.

5.4.2 Source 2 Pointer

The source 2 pointer and valid signals are generated in a similar manner as the source 1 pointer. Figure 5-20 is a detailed block diagram of the source 2 pointer logic.



MR_X0114_09

Figure 5-20 FPL Source 2 Pointer Logic

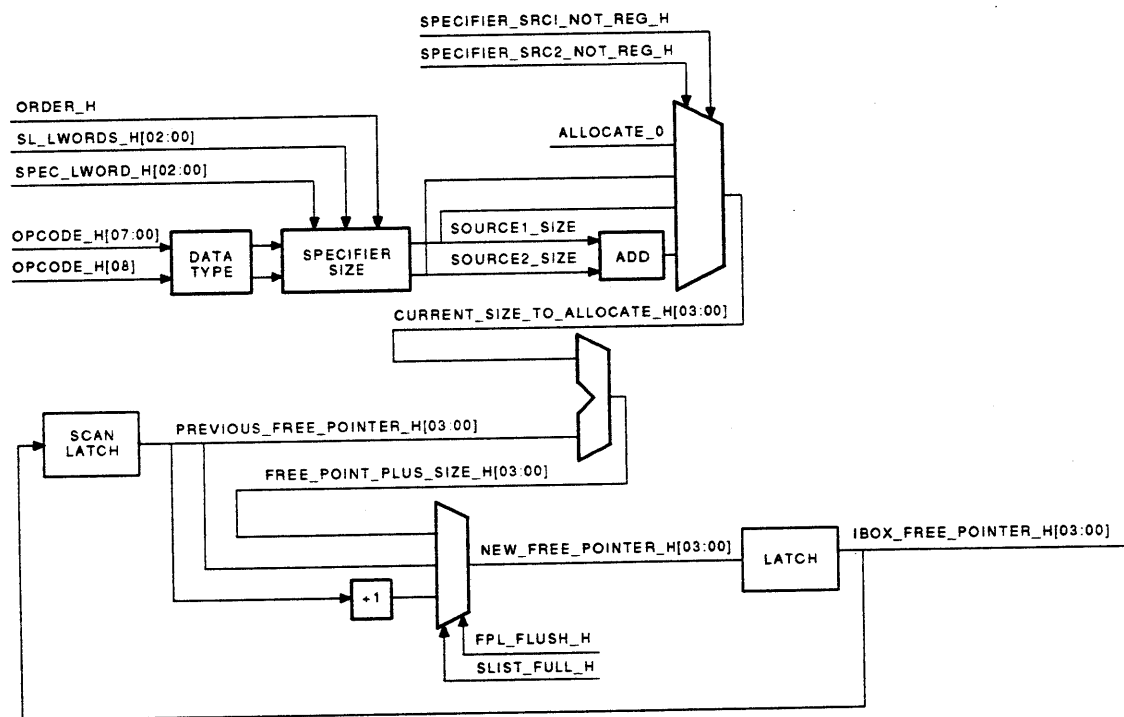
The pointer multiplexer of this logic provides the selection of three sources for the source 2 pointer.

- **CURRENT_SOURCE2_REGISTER[03:00]** — When source 2 is a register operand, the source 2 register field is passed.
- **FREE_POINTER[03:00]** — Selected when source 1 is a valid register and source 2 is valid but not a register operand. This entry is submitted to the source list.
- **FREE_POINTER_PLUS_SOURCE1_SIZE[03:00]** — When both source 1 and source 2 are valid but not registers, the entry for the source list (free pointer) must be allocated after the source list entries for the source 1 operand.

The select for the pointer multiplexer is provided by the source 1 and source 2 not register signals.

5.4.3 Free Pointer

As entries are added to the source list, the free pointer must be incremented to reflect the size of the entries. Figure 5-21 is a block diagram depicting the management of the free pointer.



MR_X0115_00

Figure 5-21 Free Pointer

The opcode of each instruction is decoded in the access/data type logic, with the data type providing the specifier size for each valid memory operand. Register operands have no effect on the free pointer because they are not entered into the source list.

When source 1 and source 2 pointers have both been passed as valid entries into the source list, the free pointer is equal to the size of the two operands (SRC1_PLUS_SRC2_SIZE) plus the previous free pointer.

If either source 1 or source 2 is a register operand, then only the size of the memory operand must be added to the free pointer to produce a new free pointer. When both source 1 and source 2 are register operands, the free pointer is not affected.

5.4.3.1 Free Pointer Initialization

On a flush of the free pointer logic, the EBox copies the free pointer into the last pointer in the source list, and the IBox increments the free pointer (FREE_POINTER_PLUS1[03:00]) so that the source list appears empty.

5.4.4 Destination Pointer

The destination pointer is generated and validated in XDTA and passed to the FPL in OSQB. Three signals deliver the destination pointer information:

- XDTA_DESTINATION_REG_H[03:00] is the register number of the destination operand when it is a register operand.
- XDTA_DESTINATION_REG_VALID_H is asserted when the destination operand is a register.
- XDTA_DESTINATION_VALID_H is asserted when a valid destination operand is being passed.

Figure 5-22 shows a block diagram of the destination pointer logic.

5.4.4.1 Destination Register

The XBAR supplies the destination register field (XDTA_DESTINATION_REG[03:00]) and, when the register valid bit is set (XDTA_DESTINATION_REG_H), the operand is a register and the field is placed in the destination queue as the address of the GPR that receives the destination data.

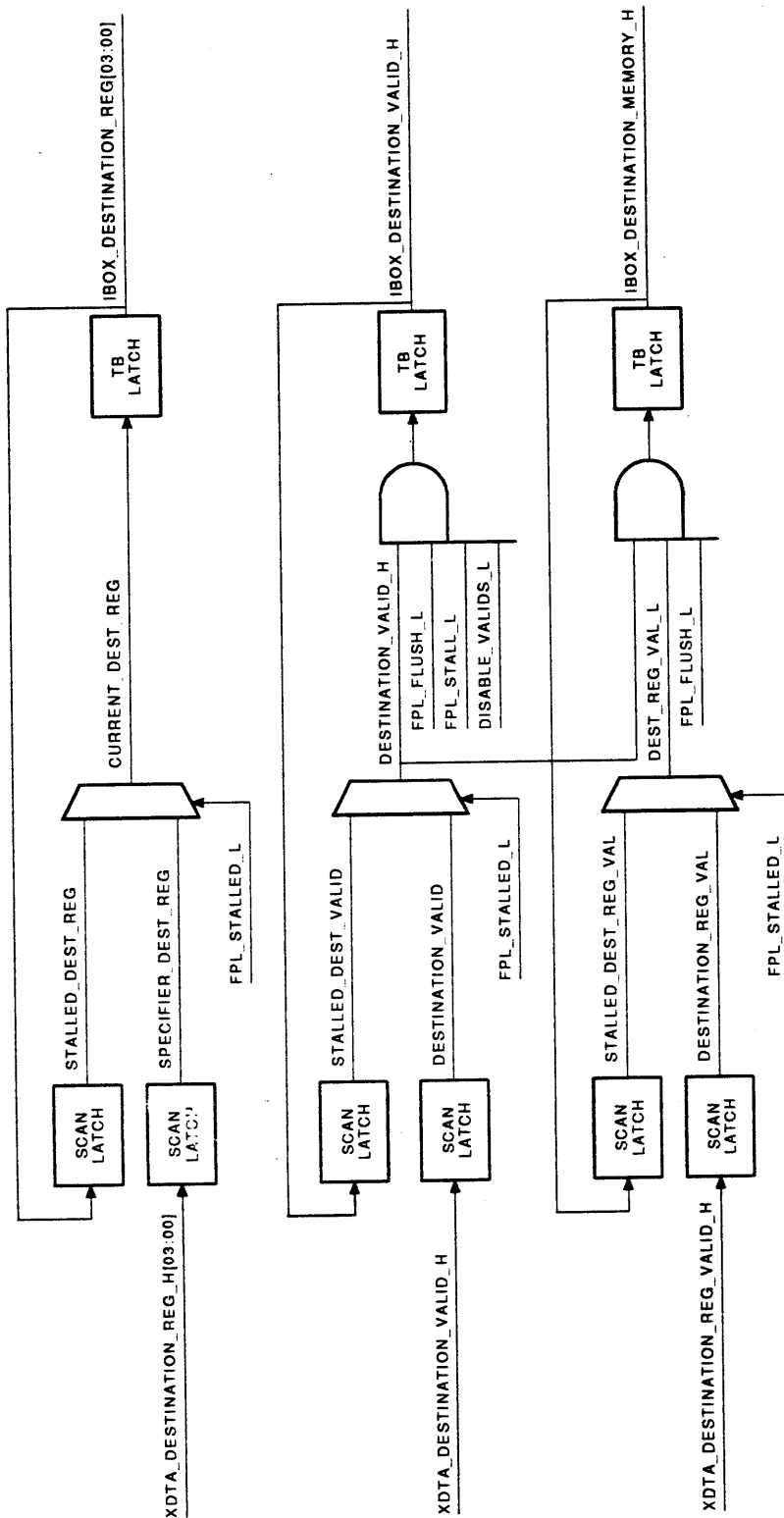
5.4.4.2 Destination Valid

The destination valid signal (XDTA_DESTINATION_VALID) is passed by the XBAR. This signal validates the destination pointer if FPL_FLUSH, EBOX_QUE_FULL, or DISABLE_VALIDS are not asserted.

5.4.4.3 Destination Memory

To differentiate between a register destination and a memory destination, IBOX_DESTINATION_MEMORY_H is used. When asserted, this signal signifies a memory destination is being passed. This signal is asserted when XDTA_DESTINATION_VALID_H is asserted, XDTA_DESTINATION_REGISTER_H is negated, and no flush is present.

All destination pointers are sent to the EBox destination pointer queue.



MR_X0118_58

Figure 5-22 Destination Pointer Logic

5.5 Operand Control Unit

The operand control (OCTL) unit provides control and distribution of stall and flush signals from the EBox and from other IBox functional units. OCTL also stores the read and write register masks that are generated for each instruction. Figure 5-23 is a basic block diagram of OCTL.

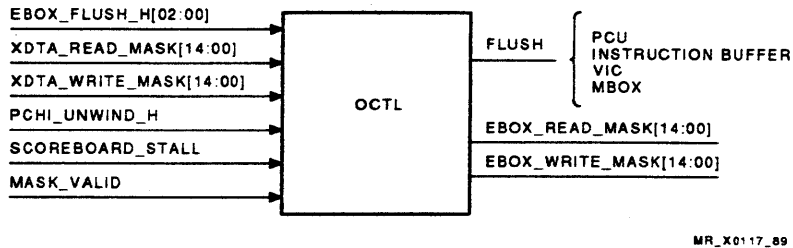


Figure 5-23 OCTL Unit

5.5.1 Read/Write Masks

Up to six read and write masks can be stored in OCTL. Each mask is passed by the XBAR when the instruction it represents has been completely decoded. The masks are generated to prevent the EBox and IBox from using stale data during the execution of instructions. Figure 5-24 is a detailed block diagram of the read and write mask generation, storage, and unwind logic.

Each instruction that is decoded in the XBAR generates a 31-bit mask that is passed to the OCTL unit. This field is passed at the completion of the instruction decode with a valid bit (XDTB_MASK_VALID_H) and is broken down as follows:

- XDTA_MASK_H[30] is the odd parity for the mask field.
- XDTA_MASK_H[29:15] is the write mask field.
- XDTA_MASK_H[14:00] is the read mask field.

If the first instruction the IBox decoded were ADDL2 R0, R1, the mask the XBAR generates would contain bits 0 and 1 set in the read mask and bit 1 set in the write mask. This mask would be selected by REG_INSERT[02:00] to be stored in REG_MASK0.

The following list describes the three basic operations that can be performed with the masks. The signals that initiate the function precede the descriptions.

- XDTB_MASK_VALID_H** — Accept a new mask from the XBAR.
- EBOX_INSTRUCTION_DONE_H** — Discard the oldest mask.
- EBOX_KEEP_MASKS[02:00]** — Unwind the masks on a bad branch prediction.

These three functions affect the following two fields:

- **REGISTER_SIZE_H[02:00]** is the field that indicates the number of the masks that are valid.
- **REGISTER_INSERT_H[02:00]** is the field that points to the position where the next mask should be inserted.

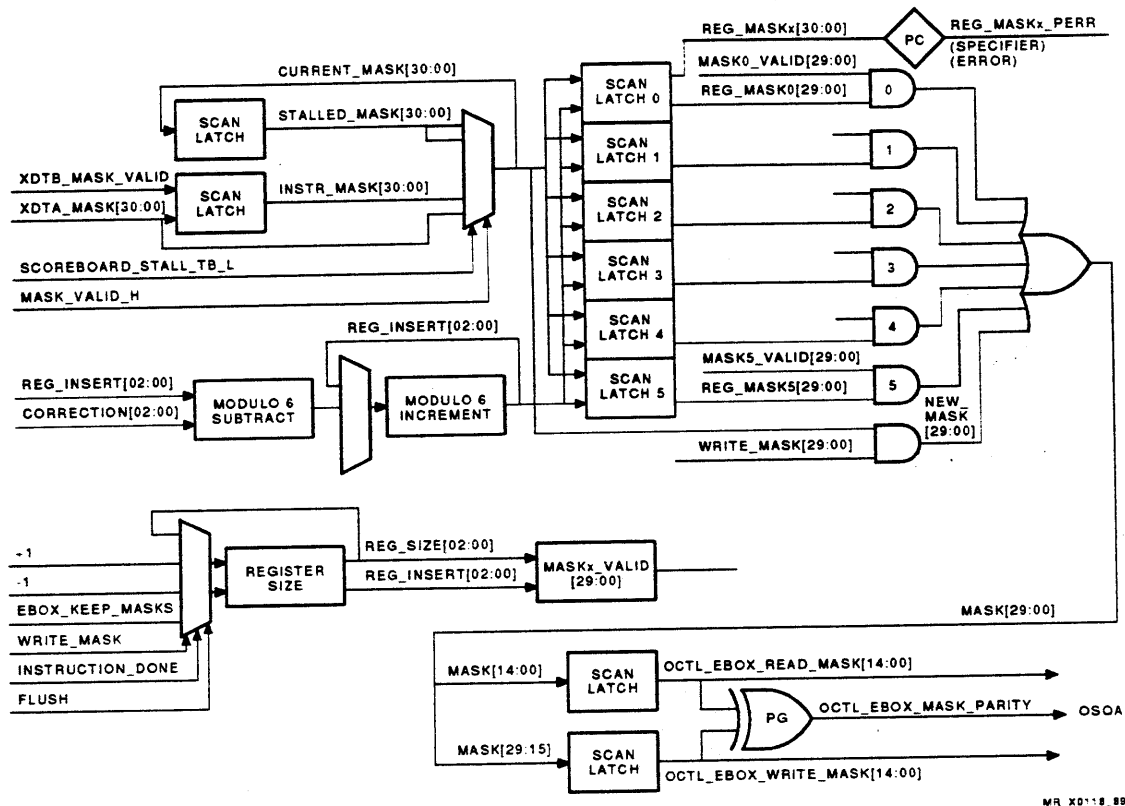


Figure 5-24 OCTL Read/Write Masks

5.5.1.1 Mask Valid

XDTB_MASK_VALID_H, asserted, directs REGISTER_SIZE and REGISTER_INSERT to be incremented. The mask field is then inserted into MASKx. Incrementing is done in modulo 6 (count = 0 through 5).

The six sets of registers that hold the masks are cyclic. That is, if the last mask was MASK5, the next mask to be used would be MASK0.

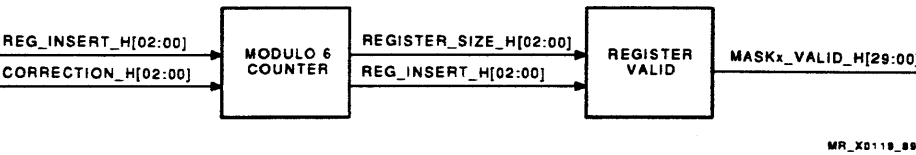
5.5.1.2 Instruction Done

The EBox, at the completion of an instruction, asserts INSTRUCTION_DONE_H. This signal directs the mask logic to discard the mask associated with the instruction that was decoded.

Discarding the mask is accomplished by decrementing (modulo 6) REGISTER_SIZE_H[02:00]. This function directs the oldest mask to be deleted while REGISTER_INSERT_H[02:00] still points to the next valid position for a mask to be written.

5.5.1.3 Correction

In the event of a bad branch prediction, the EBox must direct the IBox masks to be unwound back to the point of the branch. The EBox directs the unwind operation by sending EBOX_KEEP_MASK_L[01:00] to the mask logic. Figure 5-25 shows the logic involved in the correction.



MR_X0118_00

Figure 5-25 OCTL Mask Correction Logic

NEW_REG_INSERT_H[02:00] is generated by subtracting CORRECTION_H[02:00] from REG_INSERT_H[02:00]. The subtraction is modulo 6. Table 5-3 describes the generation of NEW_REG_INSERT_H[02:00] with the possible combinations of REG_INSERT[02:00] and CORRECTION_H[02:00].

Table 5-3 Modulo 6 Subtraction Logic

CORRECTION_ H[02:00]	REG_INSERT[02:00]							
	0	1	2	3	4	5	6	7
0	000	001	010	011	100	101	XXX	XXX
1	101	000	001	010	011	100	XXX	XXX
2	100	101	000	001	010	011	XXX	XXX
3	011	100	101	000	001	010	XXX	XXX
4	010	011	100	101	000	001	XXX	XXX
5	001	010	011	100	101	000	XXX	XXX
6	000	001	010	011	100	101	XXX	XXX
7	XXX	XXX	XXX	XXX	XXX	XXX	XXX	XXX

Legend

0-7 = Input count for CORRECTION_H[02:00] or REG_INSERT[02:00]
 000-101 = NEW_REG_INSERT_H[02:00] binary output
 XXX = Not possible

NEW_REG_INSERT_H[02:00] is decoded with REGISTER_SIZE_H[02:00] to produce the register valid field. Table 5-4 provides the output of the decode logic for the register valid fields.

Table 5-4 Register Valid Fields

NEW_REG_INSERT_H[02:00]	REGISTER_SIZE_H[02:00]						
	000	001	010	011	100	101	110
000	000000	100000	110000	111000	111100	111110	111111
001	000000	000001	100001	110001	111001	111101	111111
010	000000	000010	000011	100011	110011	111011	111111
011	000000	000100	000110	000111	100111	110111	111111
100	000000	001000	001100	001110	001111	101111	111111
101	000000	010000	011000	011100	011110	011111	111111

Legend

000-101 = Input count for REGISTER_SIZE_H[02:00] or NEW_REG_INSERT_H[02:00]
 000000-111111 = REG5 valid through REG0 valid output
 Where a 1 indicates valid and 0 indicates invalid
 For example, 100000 = REG5 valid, 000011 = REG1 valid and REG0 valid

Using Table 5-4, if NEW_REGISTER_INSERT_H[02:00] were 001 and REGISTER_SIZE_H[02:00] were 010, REGISTER_VALID_H would equal 100001. This value represents REG5 and REG0 as valid.

5.5.2 Read/Write Mask Parity

REG_MASK0[30:00] through REG_MASK5_H[30:00] are parity checked at their output of the OCTL MCA. An error detected within one of the masks asserts REG_MASK_PERR_0 through REG_MASK_PERR_5, with 0 through 5 corresponding to the register mask.

REG_MASK_PERRx asserts OCTL_ERROR, which is passed to the OSQA MCA. OSQA, on receipt of this error, asserts SPECIFIER_ERROR_H and forwards it to the EBox.

5.5.3 Flushes

The OCTL unit receives EBOX_FLUSH_H[02:00], decodes the field, and distributes flush signals to the appropriate functional units. Figure 5-26 shows the flush logic in the OCTL unit.

The EBox redirects the flow of the IBox with a 3-bit flush code. Table 5-5 provides the breakdown of these flush fields.

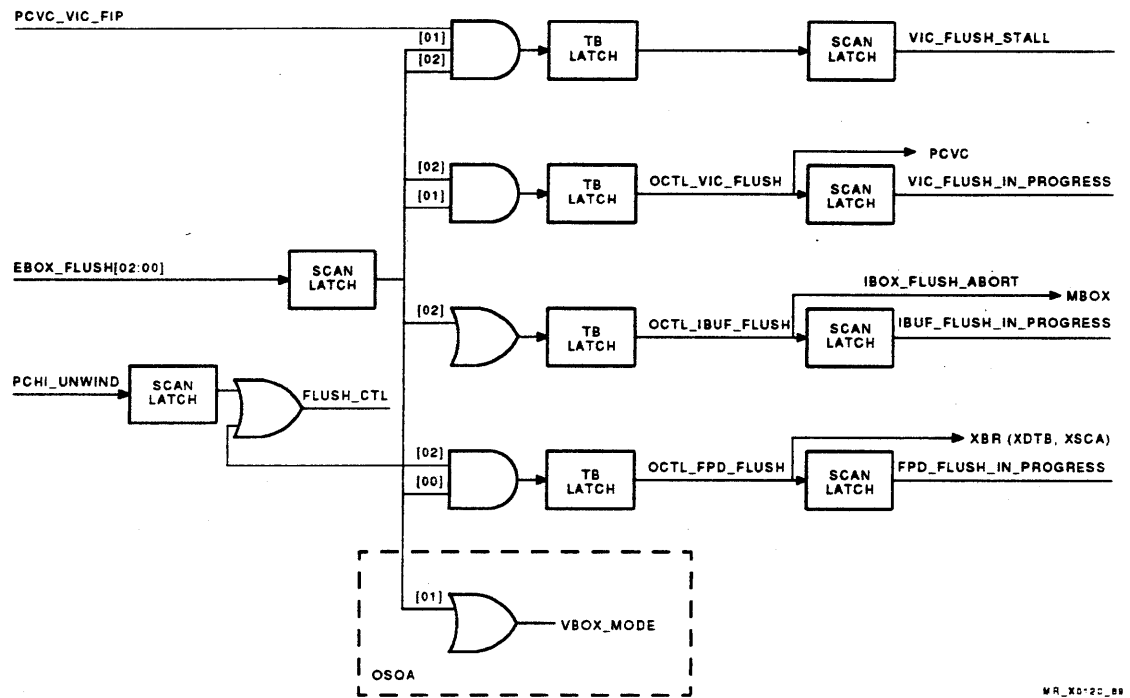


Figure 5-26 OCTL Flush Logic

Table 5-5 EBox Flush Codes

Code	Name	Action
010	VBox mode	Directs the IBox to enter VBox mode.
011	Unsuspend	Directs the XBAR to resume decoding specifiers.
100	IBUF flush	Flushes the instruction buffer.
110	VIC flush	Directs the PCU to flush the VIC. If, during the 256 cycles that it takes the VIC to complete the flush, a subsequent flush is received, the IBox stalls.
111	FPD flush	First part done (FPD) flush is used by the EBox to stall the execution of lengthy instructions. The instruction, under EBox microcode control, is stalled during execution to service interrupts or exceptions. The XBAR receives this signal.
1xx	PC flush	Each flush directed to the IBox functional units is also copied to the PCU. This copy of the flush redirects the PCU to a new prefetch PC.

5.5.4 OCTL Stalls

Two stalls are related to the read/write mask logic: scoreboard stall and mask stall. A scoreboard stall (SCOREBOARD_STALL_H) occurs when instructions containing GPR conflicts are being processed. Scoreboard stalls inhibit updating the EBox GPRs until the instruction containing the conflicts is completely processed. Mask stall (OCTL_MASK_STALL_H) is asserted when the IBox is ahead of the EBox by six instructions. The stall inhibits the IBox from processing more I-stream.

5.5.4.1 Scoreboard Stall

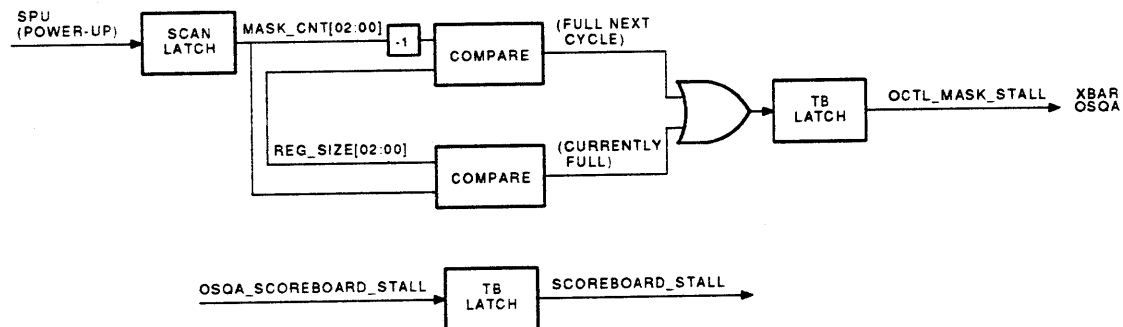
OSQA, in controlling the EBox interface, detects this stall and inform the mask logic. The stall directs the mask to be submitted again for a write to the EBox.

5.5.4.2 Mask Stall

Figure 5-27 is a detailed block diagram describing the mask stall logic.

On power-up, the SPU loads a value of six (MASK_COUNT_H[02:00]) into one comparator and MASK_COUNT_H[02:00] minus one into a second comparator. Each time a new mask is loaded, REG_SIZE_H[02:00] is compared to MASK_COUNT_H[02:00]. The two comparators used in this process output MASK_FULL_NEXT_CYCLE_H when the count reaches five and MASK_CURRENTLY_FULL_H when the count reaches six. OCTL_MASK_STALL_H is asserted to inhibit passing read and write masks to the EBox when the IBox is six instructions ahead of the EBox.

The value that is passed to the mask logic (MASK_COUNT_H[02:00]) is programmable. This value can be set at the console from one to six. Setting the value to one allows the IBox to get one instruction ahead of the EBox.



FR_X0:21_BB

Figure 5-27 OCTL Mask Stall Logic

5.6 OPU Port Interface

The OPU port interface is a read/write interface to the MBox. The interface is 32 bits wide and is byte parity protected. The interface is used for three reasons:

- Prefetching operands, from memory, for the EBox
- Queuing destination operand addresses for data from the EBox
- Fetching operands that are indirectly addressed
- Passing VBox requests to the MBox

Figure 5-28 lists the signals that comprise the OPU port interface and Table 5-6 describes them.

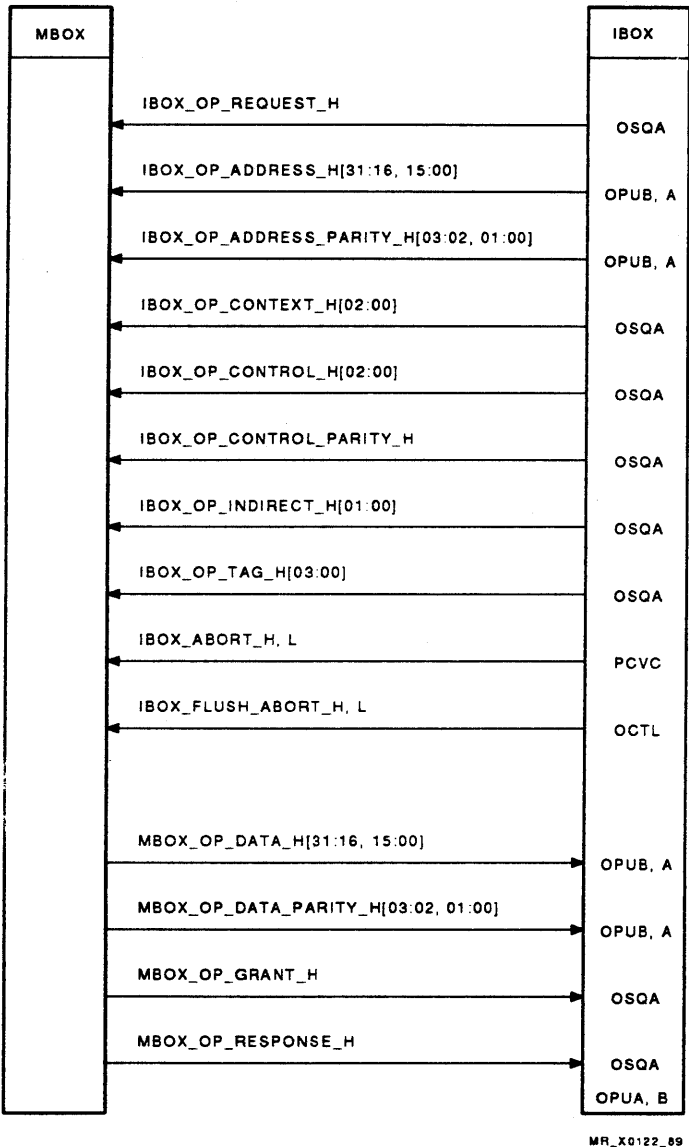


Figure 5-28 OPU Port Interface

Table 5-6 OPU Port Interface Signals

Name	Description
IBOX_OP_REQUEST_H	When asserted, indicates that an OPU port request is being made and that all other OPU port fields are valid.
IBOX_OP_ADDRESS_H[31:00]	These lines are the 32-bit address sent to the MBox.
IBOX_OP_ADDRESS_PARITY_H[03:00]	Byte parity for the OPU address.
IBOX_OP_CONTEXT_H[02:00]	Defines the reference size of the OPU port request.

Field	Reference Size
000	Longword
001	Byte
010	Word
011	Unused
100	Quadword
101	Octaword
110	Unused
111	Block (16 quadwords)

IBOX_OP_CONTROL_H[02:00] Provides the reference type of the transaction.

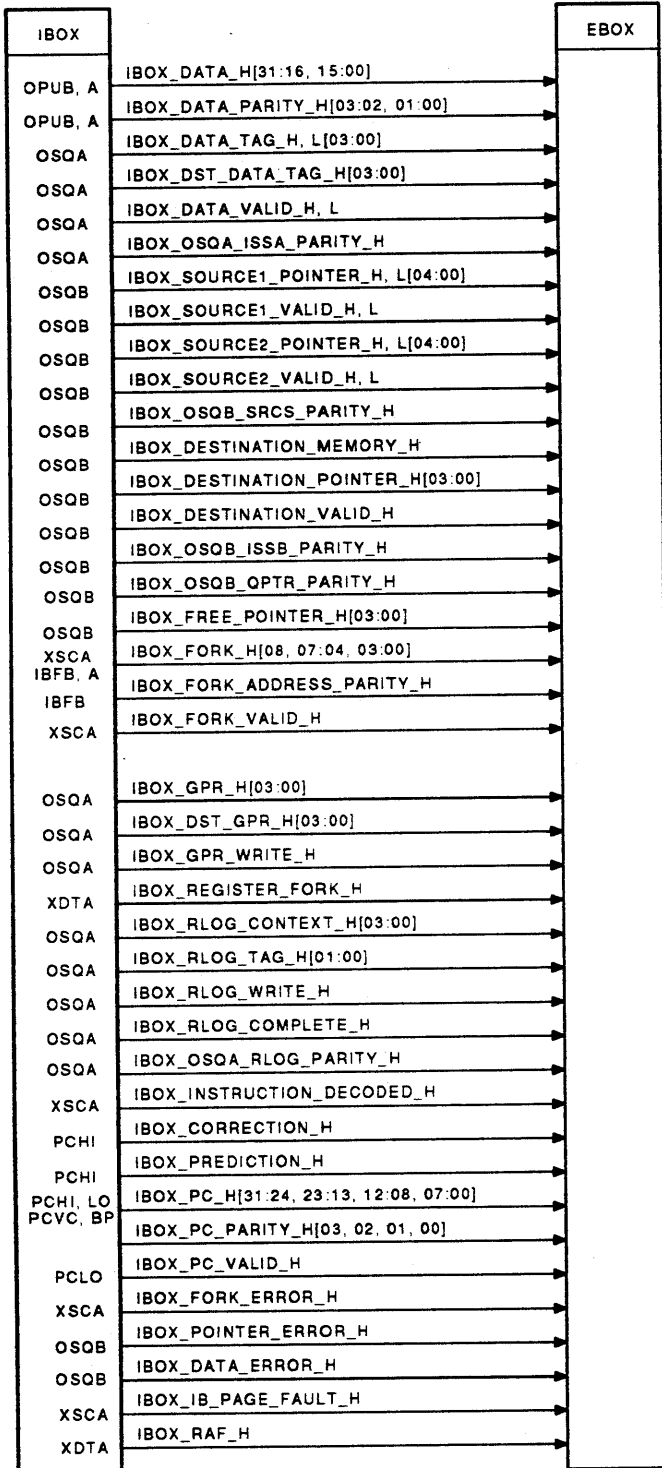
Field	Reference Type
000	Read (lock status)
001	Read with write check (lock status)
010	Read with write check (no conflict check)
011	Write check (lock status)
100	Read (don't lock status)
101	Read with write check (don't lock status)
110	Unused
111	Unused

Table 5-6 (Cont.) OPU Port Interface Signals

Field	Reference Size										
IBOX_OP_INDIRECT_H[01:00]	Indicates the final destination of the MBox reference.										
	<table> <tr> <th>Field</th><th>Destination</th></tr> <tr> <td>00</td><td>Nonwrite reference for the EBox</td></tr> <tr> <td>01</td><td>Write reference for the EBox</td></tr> <tr> <td>10</td><td>Indirect OPU read for a nonwrite specifier</td></tr> <tr> <td>11</td><td>Indirect OPU read for a write specifier</td></tr> </table>	Field	Destination	00	Nonwrite reference for the EBox	01	Write reference for the EBox	10	Indirect OPU read for a nonwrite specifier	11	Indirect OPU read for a write specifier
Field	Destination										
00	Nonwrite reference for the EBox										
01	Write reference for the EBox										
10	Indirect OPU read for a nonwrite specifier										
11	Indirect OPU read for a write specifier										
IBOX_OP_TAG_H[03:00]	Provides the address in the source list that the returning MBox data should be written.										
IBOX_OP_CONTROL_PARITY_H	Odd parity for IBOX_OP_CONTEXT_H[02:00], IBOX_OP_CONTROL_H[02:00], IBOX_OP_INDIRECT_H[01:00], and IBOX_OP_TAG_H[03:00].										

5.7 IBox-to-EBox Interface

The IBox interface to the EBox is used to send the CSU and SLU operand data, control, and error information to the EBox. Figure 5-29 lists the signals that comprise the IBox-to-EBox interface and Table 5-7 describes them.



MR_X0123_89

Figure 5-29 IBox-to-EBox Interface

Table 5-7 IBox-to-EBox Interface Signals

Name	Description
IBOX_DATA_H[31:00]	This bus delivers the short literal or immediate data to the EBox source list. GPR updates, due to autoincrement and autodecrement specifiers, are also delivered to the EBox GRPs with this bus.
IBOX_DATA_PARITY_H[03:00]	Byte parity for IBOX_DATA_H[31:00].
IBOX_DATA_TAG_H, L[03:00]	Address in the source list that the IBOX_DATA is to be written to.
IBOX_DATA_VALID_H	When asserted, indicates that there is short literal or immediate data for the source list.
IBOX_DATA_ERROR_H	Asserted to inform the EBox that a specifier error has been detected.
IBOX_OSQA_ISSA_PARITY_H	Odd parity for IBOX_DATA_TAG_H[03:00] and IBOX_DATA_VALID.
IBOX_SOURCE1_POINTER_H, L[04:00]	Provides the EBox source queue with the address, or the GPR of the source 1 operand. If the operand is SLU, CSU, or MBox data, the field provides the source list address for the operand.
IBOX_SOURCE1_VALID_H, L	Validates the source 1 pointer.
IBOX_SOURCE2_POINTER_H, L[04:00]	The same as the source 1 pointer, except it is the pointer for the source 2 operand.
IBOX_SOURCE2_VALID_H, L	Valid bit for the source 2 pointer.
IBOX_OSQB_SRCS_PARITY_H	Parity bit for source 1 and 2 pointers and valid bits.
IBOX_DESTINATION_MEMORY_H	Asserted to inform the EBox that the destination operand is memory (not register).
IBOX_DESTINATION_POINTER_H[03:00]	Provides the destination operand address. The address contains a GPR reference or write queue address in the MBox.
IBOX_DESTINATION_VALID_H	When asserted, validates all destination signals.
IBOX_OSQB_ISSB_PARITY_H	Parity bit for destination pointers and valid bit.
IBOX_OSQB_QPTR_PARITY_H	Parity bit for source 1, source 2, and destination valid bits and pointers.
IBOX_FREE_POINTER_H[03:00]	Provides the current free pointer value in the source list.
IBOX_FORK_H[08:00]	Copy of the opcode.
IBOX_FORK_PARITY_H	Parity bit for IBOX_FORK_H[08:00].
IBOX_FORK_VALID_H	Asserted to indicate that IBOX_FORK_H[08:00] is valid.
IBOX_GPR_H[03:00]	The address of the EBox GPR that is to be written. This copy of the address is used by the RLOG.
IBOX_DST_GPR_H[03:00]	Contains the address of the EBox GPR that is to be written. This copy of the address is for the STREG.
IBOX_WRITE_H	Valid bit for IBOX_GPR_H and IBOX_DATA_H.
IBOX_REGISTER_FORK_H	Asserted to inform the EBox that a USRC specifier is being decoded.

Table 5-7 (Cont.) IBox-to-EBox Interface Signals

Name	Description																																		
IBOX_RLOG_CONTEXT_H[03:00]	Indicates the context for an RLOG request. The field supplies size and direction of changes.																																		
	<table> <tr> <th>Field</th><th>Change</th></tr> <tr> <td>0000</td><td>No change to GPR</td></tr> <tr> <td>0001</td><td>GPR incremented by one (byte)</td></tr> <tr> <td>0010</td><td>GPR incremented by two (word)</td></tr> <tr> <td>0011</td><td>GPR incremented by four (longword)</td></tr> <tr> <td>0100</td><td>GPR incremented by eight (quadword)</td></tr> <tr> <td>0101</td><td>GPR incremented by sixteen (octaword)</td></tr> <tr> <td>0110</td><td>Unused</td></tr> <tr> <td>0111</td><td>Unused</td></tr> <tr> <td>1000</td><td>Unused</td></tr> <tr> <td>1001</td><td>GPR decremented by one (byte)</td></tr> <tr> <td>1010</td><td>GPR decremented by two (word)</td></tr> <tr> <td>1011</td><td>GPR decremented by four (longword)</td></tr> <tr> <td>1100</td><td>GPR decremented by eight (quadword)</td></tr> <tr> <td>1101</td><td>GPR decremented by sixteen (octaword)</td></tr> <tr> <td>1110</td><td>Unused</td></tr> <tr> <td>1111</td><td>Unused</td></tr> </table>	Field	Change	0000	No change to GPR	0001	GPR incremented by one (byte)	0010	GPR incremented by two (word)	0011	GPR incremented by four (longword)	0100	GPR incremented by eight (quadword)	0101	GPR incremented by sixteen (octaword)	0110	Unused	0111	Unused	1000	Unused	1001	GPR decremented by one (byte)	1010	GPR decremented by two (word)	1011	GPR decremented by four (longword)	1100	GPR decremented by eight (quadword)	1101	GPR decremented by sixteen (octaword)	1110	Unused	1111	Unused
Field	Change																																		
0000	No change to GPR																																		
0001	GPR incremented by one (byte)																																		
0010	GPR incremented by two (word)																																		
0011	GPR incremented by four (longword)																																		
0100	GPR incremented by eight (quadword)																																		
0101	GPR incremented by sixteen (octaword)																																		
0110	Unused																																		
0111	Unused																																		
1000	Unused																																		
1001	GPR decremented by one (byte)																																		
1010	GPR decremented by two (word)																																		
1011	GPR decremented by four (longword)																																		
1100	GPR decremented by eight (quadword)																																		
1101	GPR decremented by sixteen (octaword)																																		
1110	Unused																																		
1111	Unused																																		
	No change is used when the IBox detects an IRC. This records the fact that the GPR was modified, but only in the IBox.																																		
IBOX_RLOG_TAG_H[01:00]	Indicates for which instruction the RLOG information pertains. The field is matched with 3-bit counters that indicate which instruction is being executed in the EBox and IBox.																																		
IBOX_RLOG_WRITE_H	Asserted to indicate that RLOG information is to be written and to validate all other RLOG-related signals.																																		
IBOX_RLOG_COMPLETE_H	Asserted to inform the EBox that the CSU has stopped evaluating specifiers.																																		
IBOX_OSQA_RLOG_PARITY_H	Parity bit for the RLOG-related signals.																																		
IBOX_INSTRUCTION_DECDDED_H	Informs the EBox that the XBAR has completed decoding an instruction.																																		
IBOX_CORRECTION_H	Asserted by the PCU on a bad branch prediction. Inform the EBox that an unwind is not necessary, as the branch has not yet been shifted out of the instruction buffer.																																		
IBOX_PREDICTION_H	Asserted by the PCU to inform the EBox that the branch under decode is predicted taken.																																		
IBOX_PC_H[31:00]	These lines deliver a copy of the decode PC to the EBox.																																		
IBOX_PC_PARITY_H[03:00]	Byte parity for the EBox copy of the decode PC.																																		
IBOX_PC_VALID_H																																			

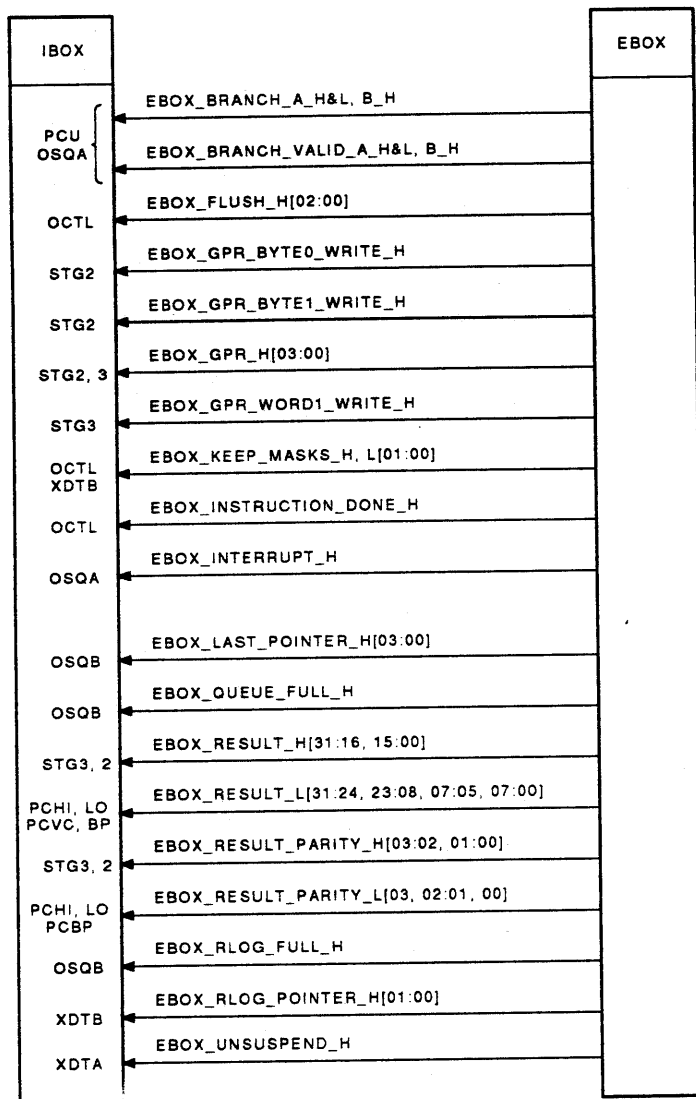
Table 5-7 (Cont.) IBox-to-EBox Interface Signals

Name	Description
IBOX_FORK_ERROR_H	Asserted to inform the EBox that a fork error has been detected.
IBOX_POINTER_ERROR_H	Asserted to inform the EBox of a detected pointer error.
IBOX_IB_PAGE_FAULT_H	Asserted to inform the EBox that the data being decoded page faulted in the MBox. The signal is not sent unless the data is accessed.
IBOX_RAF_H	The XBAR informs the EBox of an RAF by asserting this line.

5.8 EBox-to-IBox Interface

The EBox interface to the IBox is used to write result data to GPRs and to provide a variety of control functions for the IBox. The control signals can include starting or flushing PCs, RLOG unwinds, interrupts, and branch prediction status signals.

Figure 5-30 lists the signals that comprise the EBox-to-IBox interface and Table 5-8 describes them.



MR_X0124_89

Figure 5-30 EBox-to-IBox Interface

Table 5-8 EBox-to-IBox Interface Signals

Name	Description
EBOX_BRANCH_A,B_H	Asserted in the PCU and OSQA when a bad branch prediction is detected.
EBOX_BRANCH_VALID_H	Asserted to inform the IBox that a conditional branch has been retired. The CSU decrements the branch count if the branch was predicted correctly.
EBOX_FLUSH_H[02:00]	This field passes EBox flush signals to the IBox.
EBOX_GPR_BYTE0_WRITE_H	Asserted to indicate that the EBox wishes to write byte 0 of the GPR pointed to by EBOX_GPR_H[03:00].
EBOX_GPR_BYTE1_WRITE_H	Asserted to indicate that the EBox wishes to write byte 1 of the GPR pointed to by EBOX_GPR_H[03:00].
EBOX_GPR_H[03:00]	These lines indicate which GPR the EBox will write to.
EBOX_GPR_WORD1_WRITE_H	Asserted to indicate the EBox wishes to write the high-order word that EBOX_GPR_H[03:00] is addressing.
EBOX_KEEP_MASK_H[01:00]	This field informs the IBox of how many register masks to keep on a bad branch prediction. The field reflects the number of instructions still in progress in the EBox.
EBOX_INSTRUCTION_DONE_H	Asserted when an instruction is complete in the EBox. Directs the register mask logic to delete the oldest mask.
EBOX_INTERRUPT_H	Asserted when the EBox is taking an exception or an interrupt and directs the IBox to stop processing specifiers.
EBOX_LAST_POINTER_H[03:00]	This field provides a pointer to the last used location in the source list.
EBOX_QUEUE_FULL_H	Asserted when the EBox queues are full (except RLOG).
EBOX_RESULT_H[31:00]	These lines deliver the result data to be written to the IBox GPRs and is also the path used to supply a new PC to the IBox.
EBOX_RESULT_PARITY_H[03:00]	Byte parity for the EBox result data.
EBOX_RLOG_FULL_H	Asserted to inform the IBox that the EBox RLOG queue is full.
EBOX_RLOG_POINTER_H[02:00]	This field is sent to XDTB and describes the current RLOG entry.
EBOX_UNSUSPEND_H	This signal is sent to XSCA to negate XBAR_SUSPEND_H.

5.9 VBox Interface

The VBox interfaces to the MBox through the EBox and the IBox. The MBox prefetches the operand requested and the VBox, through the IBox STREGs and OPU port, requests the operands. The operand data is returned to the EBox source list.

Figure 5-31 list the signals that comprise the VBox interface and Table 5-9 describes them.

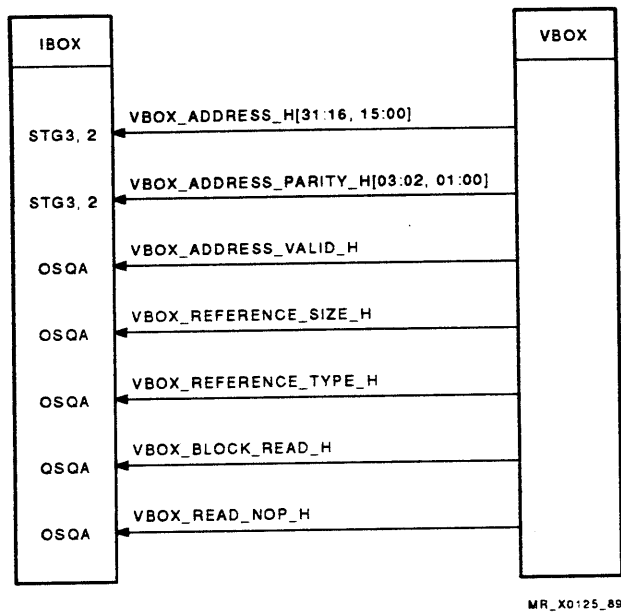


Figure 5-31 VBox Interface

Table 5-9 VBox Interface Signals

Name	Description
VBOX_ADDRESS_H[31:02]	Address lines of the request the VBox is making to the MBox. Bits 0 and 1 are always cleared.
VBOX_ADDRESS_PARITY_H[03:00]	Byte parity for the VBox address.
VBOX_ADDRESS_VALID_H	Asserted to indicate that the VBox is making an MBox request, across the OPU port. The signal validates all other VBox signals received by the IBox.
VBOX_BLOCK_READ_H	Asserted when the VBox request is for a block of data (16 longwords). When asserted, it is assumed that a read request is being made.
VBOX_REFERENCE_SIZE_H	When asserted, indicates that the reference size of the VBox request is a quadword. When negated, the reference is a longword.
VBOX_REFERENCE_TYPE_H	When asserted, indicates that the VBox reference type is write. When negated, the reference type is read.

IBox Error Descriptions

This chapter describes the IBox error registers. Six registers report errors. The tables in this chapter contain a description of each error, including the register bit, the error mnemonic, the error signal that asserts the error bit, and a description of the failing data or control signals that cause the error.

6.1 IBox Error Registers

The six error registers that report IBox errors are as follows:

```
IBOX_FETCH_ERROR_REG1_H[31:00]
IBOX_FETCH_ERROR_REG2_H[31:00]
IBOX_DECODE_ERROR_REG1_H[31:00]
IBOX_XBR_DECODE_ERROR_REGISTER_H[31:00]
IBOX_SPECIFIER_REG1_H[31:00]
IBOX_SPECIFIER_REG2_H[31:00]
```

Each register corresponds to one of the three IBox pipeline stages: fetch, decode, or specifier. Data and control flow mainly from the fetch stage to the decode stage, then to the specifier stage in the IBox. Because of this relationship, if an error is detected in a previous stage, only that error is reported. Once an error is detected in the IBox, the state elements in the IBox are held until the console can scan in new values and restart the IBox.

6.2 Fetch Error Register 1

Two 32-bit registers report errors occurring in the fetch stage of the IBox pipeline. The errors reported are those detected in the following MCAs:

```
PCBP
PCVC
PCLO
PCHI
IBFA
IBFB
```

Figure 6–1 shows the bit field breakdown of fetch error register 1. Table 6–1 describes each error that this register reports.

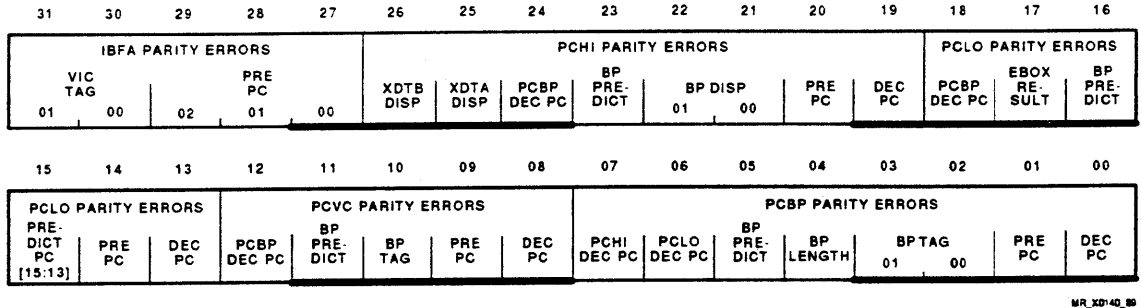


Figure 6-1 Fetch Error Register 1

Table 6-1 Fetch Error Register 1

Bits	Error Name	Description
00	DEC PC (PCBP_DECODE_PC_ERROR_H)	Occurs when a parity error is detected in data being latched in DECODE_PC_TA_H[07:00] in PCBP.
01	PRE PC (PCBP_PREFETCH_PC_ERROR_H)	Occurs when a parity error is detected in the data being latched in PREFETCH_PC_H[07:00] in PCBP.
03:02	BP TAG (PCBP_BP_PRED_TAG_ERROR_H[03:02])	Occurs when a parity error is detected in bits [31:16] of the branch prediction cache tag field. Parity checking is performed in PCBP.
04	BP LENGTH (PCBP_BP_INSTR_LENGTH_ERROR_H)	Occurs when a parity error in the instruction length field is stored in the BP STRAMs on the VIC MCU. Parity checking is performed after the field is passed to PCBP.
05	BP PREDICT (PCBP_BP_PREDICTION_ERROR_H)	Occurs when the two prediction bits from the BP STRAMs are not the same. PCBP receives these bits from the VIC MCU.
06	PCLO DEC PC (PCBP_PCLO_DECODE_PC_ERROR_H)	Occurs when a parity error is detected in PCLO_DECODE_PC_TB_H[23:16]. PCBP receives this field from PCLO and performs the parity check.
07	PCHI DEC PC (PCBP_PCHI_DECODE_PC_ERROR_H)	Occurs when a parity error is detected in PCHI_DECODE_PC_TB_H[31:24]. PCBP receives this field from PCHI and performs the parity check.
08	DEC PC (PCVC_DECODE_PC_ERROR_H)	Occurs when a parity error is detected in the data being latched in DECODE_PC_TA_H[15:08].
09	PRE PC (PCVC_PREFETCH_PC_ERROR_H)	Occurs when a parity error is detected in the data being latched in PREFETCH_PC_H[15:08] in PCVC.
10	BP TAG (PCVC_BP_PRED_TAG_ERROR_H)	Occurs when a parity error is detected on bits [15:10] of the branch prediction tag. The parity check is performed when the STRAM data is passed to PCVC.
11	BP PRED (PCVC_BP_PREDICTION_ERROR_H)	Occurs when the BP prediction bits sent to PCBP are not the same. The bits are stored in the BP STRAMs of the VIC MCU.
12	PCBP DEC PC (PCVC_PCBP_DECODE_PC_ERROR_H)	Occurs when a parity error is detected on data being latched in PCBP_DECODE_PC_TA_H[05:00].

Table 6-1 (Cont.) Fetch Error Register 1

Bits	Error Name	Description
13	DEC PC (PCLO_DECODE_PC_ERROR_H)	Occurs when the data being latched in DECODE_PC_TA_H[23:16] causes a parity error in PCLO.
14	PRE PC (PCLO_PREFETCH_PC_ERROR_H)	Occurs when the data being latched in PREFETCH_PC_H[23:16] asserts a parity error in PCLO.
15	PREDICT PC [15:13] (PCLO_PRED_PC_15_13_ERROR_H)	Occurs when BPPC_PREDICTION_PC_B_H[15:13] from BPST asserts a parity error in PCLO.
16	BP PREDICT (PCLO_BP_PREDICTION_ERROR_H)	Occurs when the two BP prediction bits sent to PCBP are not the same. The bits are stored in the BP STRAMs in the VIC MCU.
17	EBOX RESULT (PCLO_EBOX_RESULT_ERROR_H[01])	Occurs when EBOX_RESULT_H[15:08] from the EBox asserts a parity error in PCLO.
18	PCBP DEC PC (PCLO_PCBP_DECODE_PC_ERROR_H)	Occurs when PCBP_DECODE_PC_TA_H[05:00] or PCBP_DECODE_PC_PARITY_TA_H asserts a parity error in PCBP.
19	DEC PC (PCHI_DECODE_PC_ERROR_H)	Occurs when the data being latched in DECODE_PC_TA_H[31:24] causes a parity error in PCHI.
20	PRE PC (PCHI_PREFETCH_PC_ERROR_H)	Occurs when the data being latched in PREFETCH_PC_H[31:24] asserts a parity error in PCHI.
22:21	BP DISP (PCHI_BP_TAG_DISP_ERROR_H[01:00])	Occurs when BPTD_TAG_DISPLACEMENT_H[15:00] asserts a parity error in PCHI.
23	BP PREDICT (PCHI_BP_PREDICTION_ERROR_H)	If this error occurs, the two prediction bits sent to PCBP from the BP STRAMs are not the same value.
24	PCBP DEC PC (PCHI_PCBP_DECODE_PC_ERROR_H)	Occurs if a parity error is detected when checking PCBP_DECODE_PC_TA_H[05:00] on PCBP.
25	XDTA DISP (PCHI_XDTA_DISP_ERROR_H)	Asserted in PCHI when a parity error is detected in XDTA_DISPLACEMENT_L[11:08, 03:00].
26	XDTB DISP (PCHI_XDTB_DISP_ERROR_H)	Asserted in PCHI when a parity error is detected when checking XDTB_DISPLACEMENT_L[15:12, 07:04] and XDTB_DISPLACEMENT_PARITY_H[01].
29:27	PRE PC (IBFA_PREF_PC_ERROR_H[03:01])	Asserted in IBFA when a parity error is detected in the following signals: PCBP_PREFETCH_PC_H[04:00] from PCBP to IBFA, PCHI_PREFETCH_PC_H[31:24] from PCHI to IBFA, and PCLO_PREFETCH_PC_H[23:16] from PCLO to IBFA.
31:30	VIC TAG (IBFA_VICT_TAG_ERROR_H[02])	Bit 30 of this error is asserted when VICT_TAG_H[23:13] asserts a parity error. Bit 31 is set when VICT_TAG_H[31:24] asserts a parity error. Both VIC tag fields are checked in IBFA when they are passed from the VICT STRAMs to the XBR MCU.

6.3 Fetch Error Register 2

Fetch error register 2 is 11 bits wide and reports errors occurring in IBFA and IBFB. Figure 6-2 shows the bit field breakdown of this register and Table 6-2 describes each error that this register reports.

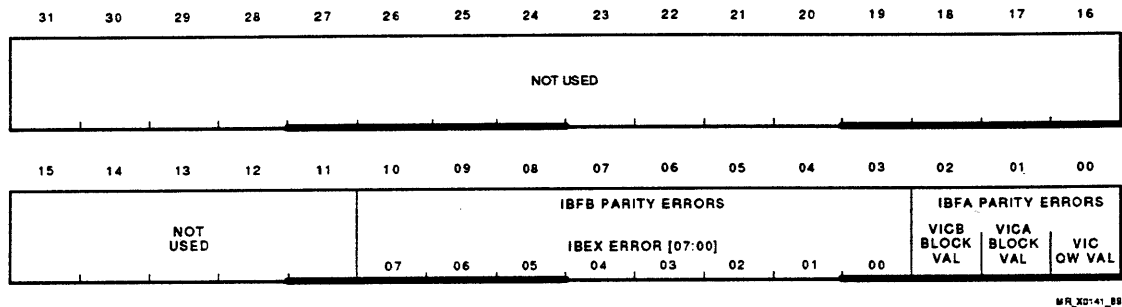


Figure 6-2 Fetch Error Register 2

Table 6-2 Fetch Error Register 2

Bits	Error Name	Description
00	VIC QW VAL (IBFA_VIC_QUADWORD_ERROR_H)	Occurs when a parity check of VICQ_QUADWORD_VALIDS_H[03:00] fails.
01	VICA BLOCK VAL (IBFA_VICA_BLOCK_VALID_ERROR_H)	Occurs when VICA_BLOCK_VALID_H[01:00] is parity checked after being passed from VICT to IBFA on the XBR MCU.
02	VICB BLOCK VAL (IBFA_VICB_BLOCK_VALID_ERROR_H)	Occurs when VICB_BLOCK_VALID_H[01:00] fails parity testing after being passed from VICT to IBFA on the XBR MCU.
10:03	IBEX ERROR (A_IBFB_IBEX_ERROR_H[07:00])	Occurs when IBEX_DATA_H[63:00] fails a parity test that is performed as the data is passed to IBUF. The failing data is held partly in IBFA, partly in IBFB. Each byte of IBEX data has one parity bit. Each byte of IBEX data is split into nibbles. The low nibbles are stored in IBFA and the high nibbles are stored in IBFB.

6.4 Decode Error Register 1

Decode errors are reported in two registers. Decode error register 1 reports errors that occur on instruction buffer data that is passed to XBAR and on control signals that are passed from XBAR to the PCU (shift counts, shift opcode). Figure 6-3 shows the bit field breakdown of this register and Table 6-3 describes each error that this register reports.

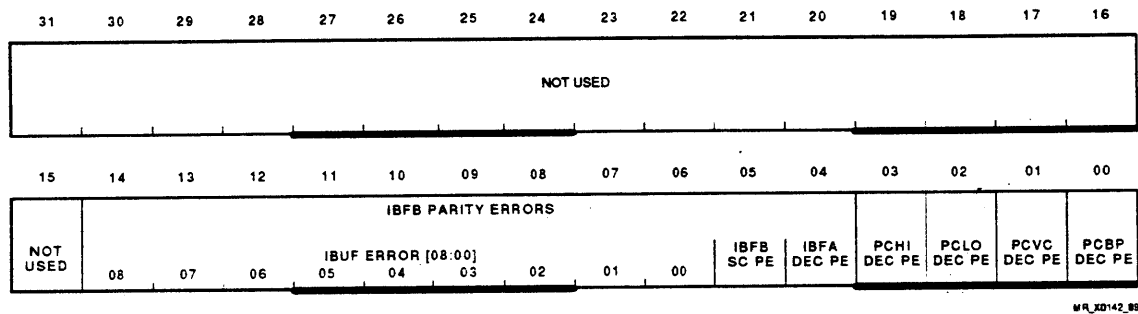


Figure 6-3 Decode Error Register 1

Table 6-3 Decode Error Register 1

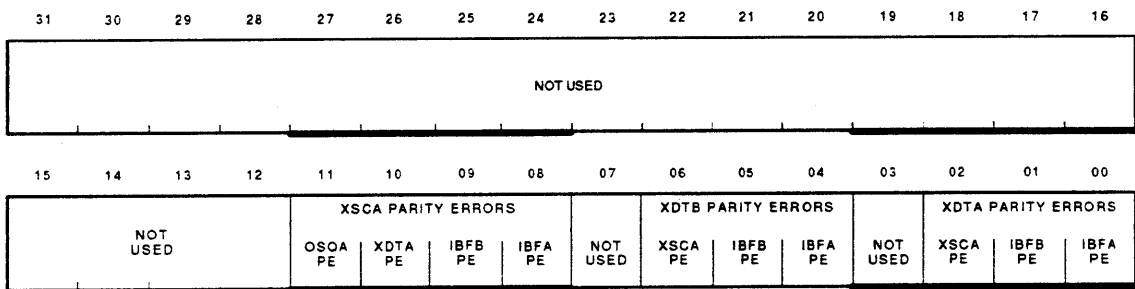
Bits	Error Name	Description
00	PCBP DEC PE (A_PCBP_DECODE_ERROR_H)	Occurs when an error is detected in XSCA_SHIFTCOUNT_C_L[03:00], XSCA_SHIFTOPCODE_C_L, or XSCA_PCBP_B_PARITY_H. Parity checking is performed on PCBP as these signals are passed from XBAR.
01	PCVC DEC PE (A_PCVC_DECODE_ERROR_H)	Occurs when an error is detected in A_XBAR_SHIFTCOUNT_H[03:00], A_XBAR_SHIFTOPCODE_H, and A_XSCA_PCVC_B_PARITY_H as they are passed to PCVC.
02	PCLO DEC PE (A_PCLO_DECODE_ERROR_H)	Occurs when an error is detected in XSCA_SHIFTCOUNT_C_H[03:00], XSCA_SHIFTOPCODE_C_H, and XSCA_PCLO_B_PARITY_H as they are passed to PCLO.
03	PCHI DEC PE (A_PCHI_DECODE_ERROR_H)	Occurs when an error is detected in XSCA_SHIFTCOUNT_B_H[03:00], XSCA_SHIFTOPCODE_B_H, XSCA_DISPLACEMENT_VALID_H, and XSCA_PCHI_B_PARITY_H when they are passed to PCHI.
04	IBFA DEC PE (A_IBFA_DECODE_ERROR_H)	Occurs when an error is detected in XSCA_SHIFTCOUNT_A_H[03:00], XSCA_SHIFTOPCODE_A_H, XSCA_FD_SHIFTOPCODE_H, and XSCA_IBFA_B_PARITY_H as they are passed to IBFA.
05	IBFB DEC PE (A_IBFB_SC_ERROR_H)	Occurs when an error is detected in XSCA_SHIFTCOUNT_A_L[03:00], XSCA_SHIFTOPCODE_A_L, XSCA_FD_SHIFTOPCODE_L, XSCA_EXTENDED_L, and XSCA_IBFB_B_PARITY_H as they are passed to IBFB.

Table 6-3 (Cont.) Decode Error Register 1

Bits	Error Name	Description
14:06	IBUF ERROR (A_IBFB_IBUF_ERROR_H[08:00])	Covers the IBFA and IBFB data paths for IBUF data to XBAR. Detection of a parity error in any of the nine bytes of IBUF data as it is passed to XBAR asserts the respective bit in this signal. (For example, when a parity error is detected in byte 0 of the IBUF data, byte 0 of A_IBFB_IBUF_ERROR_H[08:00] is asserted.)

6.5 XBAR Decode Error Register

The XBAR decode error register reports decode errors that occur, primarily, in the XBAR. Failing data passing between XSCA, XDTA, and XDTB asserts errors that are reported in this register. Control signals with sources external to XBAR are also reported (for example, OSQA_BRANCH_COUNT_H[01:00] and OSQA_SL_BUSY_STALL_H). Figure 6-4 shows the bit field breakdown of this register and Table 6-4 describes each error that this register reports.



WR_X0143_00

Figure 6-4 XBAR Decode Error Register

Table 6-4 XBAR Decode Error Register

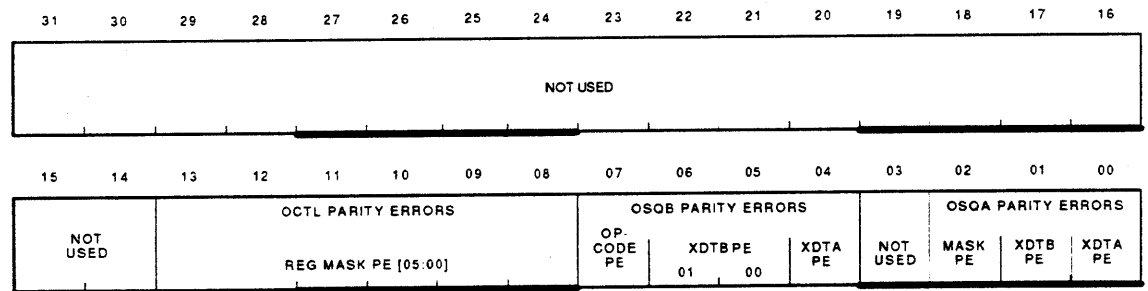
Bits	Error Name	Description
00	IBFA PE (IBFA_XDTA_PARITY_ERROR_H)	Occurs when an error is detected in IBFA_YREG_F_A_L[03:01], IBFA_VALID_A_L[03:01], or IBFA_DATA_A_L[67:64, 59:56, 51:48, 43:40, 35:32, 27:24, 19:16, 11:08] as they are passed from IBFA to XBAR.
01	IBFB PE (IBFB_XDTA_PARITY_ERROR_H)	Occurs when an error is detected in IBFB_DATA_A_L[39:36, 31:28, 23:20, 15:12] or IBFB_REGISTER_MODE_L[08:05] as they are passed from IBFB to XBAR.
02	XSCA PE (XSCA_XDTA_PARITY_ERROR_H)	Occurs when an error is detected on the interconnect between XSCA and XDTA. The signals that can cause this error are as follows: XSCA IMPLIED_MASK_H XSCA_IRC_L XSCA_REQUEST_H[03:00] XSCA_SPECIFIERS_DECODED_H[01:00] XSCA_SP1_ACCESS_H[09:00] XSCA_SP1_DATATYPE_H[02:00] XSCA_SP2_ACCESS_H[06:00] XSCA_SP2_DATATYPE_H[02:00] XSCA_SP3_ACCESS_H[01:00] XSCA_SP3_DATATYPE_H[02:01] XSCA_X8F_H
04	IBFA PE (IBFA_XDTB_PARITY_ERROR_H)	Occurs when an error is detected in IBFA_DATA_A_H[03:00] or IBFA_YREG_F_A_H[04:01] as they are passed from IBFA to XDTB.
05	IBFB PR (IBFB_XDTB_PARITY_ERROR_H)	Occurs when a parity error is detected in IBFB_DATA_A_H[71:68, 63:60, 55:52, 47:44, 39:36, 31:28, 23:20, 15:12, 07:04] after it is passed to XDTB.
06	XSCA PE (XSCA_XDTB_PARITY_ERROR_H)	Occurs when an error is detected on the interconnect between XSCA and XDTB. An error detected in one or more of the following signals can assert this error: XSCA_REQUEST_H[03:00] XSCA_IRC_L XSCA_X8F_H XSCA_SP1_DATATYPE_H[02:00] XSCA_SP2_DATATYPE_H[02:00] XSCA_SP1_DATATYPE_H[02, 00] XSCA_SPECIFIERS_REMAINING_L[02:01] XSCA_SPECIFIERS_DECODED_H[01:00]
08	IBFA PE (IBFA_XSCA_PARITY_ERROR_H)	Occurs when an error is detected in one or more of the following signals: IBFA_DATA_B_H[03:00] IBFA_VALID_H[08:00] IBFA_YREG_F_B_H[04:01] IBFA_IB_PAGE_FAULT_H

Table 6-4 (Cont.) XBAR Decode Error Register

Bits	Error Name	Description
09	IBFB PE (IBFB_XSCA_PARITY_ERROR_H)	Occurs when an error is detected on the interconnect between IBFB and XSCA. The following signals can assert this signal: IBFB_DATA_B_H[39:36, 31:28, 23:20, 15:12, 07:04] IBFB_REGISTER_MODE_H[08:05] IBFB_SL_MODE_H[07:05] IBFB_UNCONDITIONAL_B_H
10	XDTA PE (XDTA_XSCA_PARITY_ERROR_H)	Occurs when an error is detected in XDTA_IRC_MASK_H[08:00] as it is passed from XDTA to XSCA. This error may propagate to a specifier error.
11	OSQA PE (OSQA_XSCA_PARITY_ERROR_H)	Occurs when OSQA_DECODE_STALL_H, OSQA_BRANCH_COUNT_H[01:00], OSQA_SL_BUSY_STALL_H, or OSQA_IN_SEQUENCE_H asserts a parity error in XSCA. This error may propagate to a specifier error.

6.6 Specifier Error Register 1

Specifier error register 1 reports errors that occur when data is being passed from XBAR to OSQA, OSQB, and OCTL. Errors occurring with the control signals passed to the OPU MCU and the data paths for the FPL and SLU are reported through this register. Figure 6-5 shows the bit field breakdown of this register and Table 6-5 describes the errors that this register reports.



WR_X0144_80

Figure 6-5 Specifier Error Register 1

Table 6-5 Specifier Error Register 1

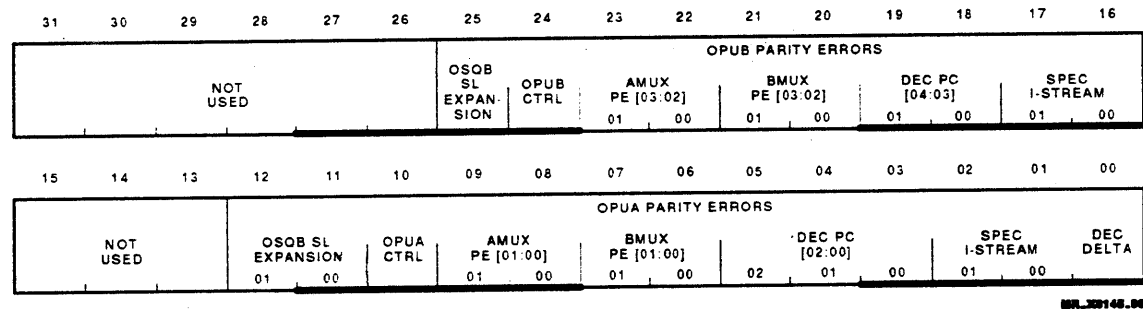
Bits	Error Name	Description
00	XDTA PE (XDTA_OSQA_PERR_TA_H)	Asserted by detection of a parity error in XDTA_XREG_H[03:00], XDTA_OSQA_A_PARITY_H, or XDTA_YREG_H[03:00] after they are passed from XDTA to OSQA.
01	XDTB PE (XDTB_OSQA_PERR_TA_H)	Occurs when a parity error is detected in XDTB_ORDER_H, XDTB_SL_VALID_H, XDTB_OSQA_B_PARITY_H, XDTB_INDEXED_H, XDTB_MODE_H[03:00], or XDTB_RLOG_TAG_H[02:00] in OSQA.
02	MASK PE (OSQA_MASK_PERR_TA_H)	Occurs when a parity error is detected when checking OCTL_EBOX_READ_MASK_H[14:00], OCTL_EBOX_WRITE_MASK_H[14:00], and OCTL_EBOX_MASK_PARITY_H.
04	XDTA PE (XDTA_OSQB_PERR_TA_H)	Asserted by a parity error on the interconnect between XDTA and OSQB. An error in one or more of the following signals asserts this error: XDTA_SL_H[03:00] XDTA_DESTINATION_REG_H[03:00] XDTA_SOURCE1_REG_H[03:00] XDTA_SOURCE2_REG_H[03:00] XDTA_DESTINATION_REG_VALID_H XDTA_DESTINATION_VALID_H XDTA_SOURCE1_REG_VALID_H XDTA_SOURCE1_VALID_H XDTA_SOURCE2_REG_VALID_H XDTA_SOURCE2_VALID_H XDTA_OSQB_B_PARITY_H
05	XDTB PE (XDTB_OSQB_A_PERR_TA_H)	Asserted by a parity error in XDTB_OPU_SPECIFIERS_COMPLETED_H[02:00], XDTB_RSL_H[02:01], and XDTB_OSQB_A_PARITY_H. Parity checking is performed in OSQB when these signals are passed from XDTB.
06	XDTB PE (XDTB_OSQB_B_PERR_TA_H)	Asserted by a parity error in XDTB_SL_SPECIFIER_NUMBER_H[02:00], XDTB_ORDER_H, XDTB_SL_VALID_H, XDTB_SL_H[05:04], and XDTB_OSQB_B_PARITY_H. Parity checking is performed in OSQB as these signals are passed from XDTB.
07	OPCODE PE (OPCODE_PERR_TA_H)	Asserted by a parity error in IBFA_OPCODE_A_L[03:00], IBFB_OPCODE_L[07:04], and IBFB_OPCODE_PARITY_H. Parity checking is performed in OSQB when these signals are received from IBFB and IBFA.
08	REG MASK PE [00] (REG_MASK_PERR_TA_H[00])	Occurs when an error is detected in register read/write mask 0 in OCTL (REG_MASK0_TA_H[30:00]). Each mask (mask 0 through 5) is parity checked in OCTL after being passed from XDTA.
09	REG MASK PE [01] (REG_MASK_PERR_TA_H[01])	Occurs when an error is detected in register read/write mask 1 in OCTL (REG_MASK1_TA_H[30:00]).

Table 6-5 (Cont.) Specifier Error Register 1

Bits	Error Name	Description
10	REG MASK PE [02] (REG_MASK_PERR_TA_H[02])	Occurs when an error is detected in register read/write mask 2 in OCTL (REG_MASK2_TA_H[30:00]).
11	REG MASK PE [03] (REG_MASK_PERR_TA_H[03])	Occurs when an error is detected in register read/write mask 3 in OCTL (REG_MASK3_TA_H[30:00]).
12	REG MASK PE [04] (REG_MASK_PERR_TA_H[04])	Occurs when an error is detected in register read/write mask 4 in OCTL (REG_MASK4_TA_H[30:00]).
13	REG MASK PE [05] (REG_MASK_PERR_TA_H[05])	Occurs when an error is detected in register read/write mask 5 in OCTL (REG_MASK5_TA_H[30:00]).

6.7 Specifier Error Register 2

Specifier error register 2 is 26 bits wide and reports errors detected in the OPUA and OPUB MCAs. Figure 6-6 shows the bit field breakdown of this register and Table 6-6 describes each error that this register reports.



IBL-329148-00

Figure 6-6 Specifier Error Register 2

Table 6-6 Specifier Error Register 2

Bits	Error Name	Description
00	DEC DELTA (SPEC_DECODE_DELTA_PERR_TA_H)	Occurs when a parity check on XDTB_DECODE_DELTA_H[05:00] and XDTB_DECODE_DELTA_PARITY_H fails.
01	SPEC I-STREAM (SPEC_ISTREAM_LWORD_PERR_TA_H[00])	Occurs when a parity error is detected when testing XDTA_DISPLACEMENT_H[11:08], XDTA_DISPLACEMENT_H[03:00], and XDTA_DISPLACEMENT_PARITY_H[00].
02	SPEC I-STREAM (SPEC_ISTREAM_LWORD_PERR_TA_H[01])	Occurs when a parity error is detected when testing XDTB_DISPLACEMENT_H[15:12], XDTB_DISPLACEMENT_H[07:04], and XDTB_DISPLACEMENT_PARITY_H[01].
03	DEC PC [00] (SPECIFIER_DECODE_PC_PERR_TA_H[00])	Occurs when a parity error is detected when testing PCBP_OPU_DECODE_PC_H[07:00] and PCBP_OPU_DECODE_PC_PARITY_H[00].
04	DEC PC [01] (SPECIFIER_DECODE_PC_PERR_TA_H[01])	Occurs when a parity error is detected while testing PCVC_OPU_DECODE_PC_H[12:08] and PCVC_OPU_DECODE_PC_PARITY_H.
05	DEC PC [02] (SPECIFIER_DECODE_PC_PERR_TA_H[02])	Occurs when a parity error is detected while testing PCLO_OPU_DECODE_PC_H[15:13] and PCLO_OPU_DECODE_PC_PARITY_H[01].
06	BMUX PE [00] (BMUX_PERR_TA_H[00])	Asserted by an error detected in the low BMUX data byte in OPUA (BMUX_H[07:00]).
07	BMUX PE [01] (BMUX_PERR_TA_H[01])	Asserted by an error in the high BMUX data byte in OPUA (BMUX_H[15:08]).
08	AMUX PE [00] (AMUX_PERR_TA_H[00])	Asserted by an error in the low AMUX data byte in OPUA (AMUX_H[07:00]).
09	AMUX PE [01] (AMUX_PERR_TA_H[01])	Asserted by an error in the high AMUX data byte in OPUA (AMUX_H[15:08]).
10	OPUA CTRL (OPUA_CONTROL_PERR_TA_H)	Occurs when a parity error is detected in OSQA_AMUX_SEL_H[02:00], OSQA_BMUX_SEL_H[02:00], OSQA_CSHFT_SEL_H[02:00], OSQA_OP_GRANT_WAIT_H, OSQA_OPU_SEQ_START_H, or OSQA_OPUX_CONTROL_PARITY_H. Parity checking is performed in OPUA.
11	SL EXPANSION (SL_EXPANSION_PERR_TA_H[00])	Occurs when an error is detected while testing OSQB_SL_EXPANSION_H[07:00] and OSQB_SL_EXPANSION_PARITY_H[00].
12	SL EXPANSION (SL_EXPANSION_PERR_TA_H[01])	Occurs when an error is detected while testing OSQB_SL_EXPANSION_H[14], OSQB_SL_EXPANSION_H[09:08], and OSQB_SL_EXPANSION_PARITY_H[01].
16	SPEC I-STREAM (SPEC_ISTREAM_LWORD_PERR_TA_H[02])	Occurs when an error is detected while testing XDTA_DISPLACEMENT_H[27:24], XDTA_DISPLACEMENT_H[19:16], and XDTA_DISPLACEMENT_PARITY_H[02].
17	SPEC I-STREAM (SPEC_ISTREAM_LWORD_PERR_TA_H[03])	Occurs when an error is detected while testing XDTB_DISPLACEMENT_H[31:28], XDTB_DISPLACEMENT_H[23:20], and XDTB_DISPLACEMENT_PARITY_H[03].

Table 6-6 (Cont.) Specifier Error Register 2

Bits	Error Name	Description
18	DEC PC [03] (SPECIFIER_DECODE_PC_PERR_TA_H[02])	Occurs when an error is detected while testing PCLO_OPU_DECODE_PC_H[23:16] and PCLO_OPU_DECODE_PC_PARITY_H[02].
19	DEC PC [04] (SPECIFIER_DECODE_PC_PERR_TA_H)	Occurs when an error is detected while testing PCHI_OPU_DECODE_PC_H[31:24] and PCHI_OPU_DECODE_PC_PARITY_H[03].
20	BMUX PE [02] (BMUX_PERR_TA_H[02])	Asserted by an error detected in the low BMUX data byte in OPUB (BMUX_H[23:16]).
21	BMUX PE [03] (BMUX_PERR_TA_H[03])	Asserted by an error detected in the high BMUX data byte in OPUB (BMUX_H[31:24]).
22	AMUX PE [02] (AMUX_PERR_TA_H[02])	Asserted by an error detected in the low AMUX data byte in OPUB (AMUX_H[23:16]).
23	AMUX PE [03] (AMUX_PERR_TA_H[03])	Asserted by an error detected in the high AMUX data byte in OPUB (AMUX_H[31:24]).
24	OPUB CTRL (OPUB_CONTROL_PERR_TA_H)	Occurs when an error is detected in one or more of the following signals: OSQA_AMUX_SEL_L[02:00] OSQA_BMUX_SEL_L[02:00] OSQA_CSHFT_SEL_L[02:00] OSQA_OP_GRANT_WAIT_H OSQA_OPU_SEQ_START_H OSQA_OPUX_CONTROL_PARITY_H
25	OSQB SL EXPANSION (SL_EXPANSION_PERR_TA_H[03])	Occurs when an error is detected while testing OSQB_SL_EXPANSION_H[31:29] and OSQB_SL_EXPANSION_PARITY_H[03].

A

IBox Input and Output Listing

Tables A-1, A-2, and A-3 list the input and output signals of the IBox. The signals are grouped by their MCU and MCA origination or destination and are listed alphabetically. All signals represent communication between the IBox and the VBox, MBox, and EBox.

The box origination of input signals is defined in the prefix of the signal (for example, EBOX_BRANCH_B_H[00] is an input from the EBox).

Table A-1 IBox-VIC Signals

Input	Destination	Origination
EBOX_BRANCH_A_H[00]	VIC-PCBP	USQ-USQC
EBOX_BRANCH_A_H[00]	VIC-PCVC	USQ-USQC
EBOX_BRANCH_VALID_A_H[00]	VIC-PCBP	USQ-USQC
EBOX_BRANCH_VALID_A_H[00]	VIC-PCVC	USQ-USQC
EBOX_RESULT_L[07:00]	VIC-PCBP	MUL-RET0
EBOX_RESULT_L[07:05]	VIC-PCVC	MUL-RET0
EBOX_RESULT_PARITY_L[03:00]	VIC-PCBP	MUL-RET0+RET1
MBOX_IB_DATA_H[03:00]	VIC-CL#4	DTA-DTM0
MBOX_IB_DATA_H[07:04]	VIC-CL#7	DTA-DTM1
MBOX_IB_DATA_H[11:08]	VIC-CL#4	DTA-DTM1
MBOX_IB_DATA_H[15:12]	VIC-CL#4	DTA-DTM1
MBOX_IB_DATA_H[19:16]	VIC-CL#4	DTA-DTM2
MBOX_IB_DATA_H[23:20]	VIC-CL#7	DTA-DTM2
MBOX_IB_DATA_H[27:24]	VIC-CL#4	DTA-DTM3
MBOX_IB_DATA_H[31:28]	VIC-CL#7	DTA-DTM3
MBOX_IB_DATA_H[35:32]	VIC-CL#4	DTA-DTM0
MBOX_IB_DATA_H[39:36]	VIC-CL#7	DTA-DTM0
MBOX_IB_DATA_H[43:40]	VIC-CL#4	DTA-DTM1
MBOX_IB_DATA_H[47:44]	VIC-CL#7	DTA-DTM1
MBOX_IB_DATA_H[51:48]	VIC-CL#4	DTA-DTM2
MBOX_IB_DATA_H[55:52]	VIC-CL#7	DTA-DTM2
MBOX_IB_DATA_H[59:56]	VIC-CL#4	DTA-DTM3
MBOX_IB_DATA_H[63:60]	VIC-CL#7	DTA-DTM3
MBOX_IB_DATA_PARITY_H[01:00]	VIC-CL#7	DTB-DTM0, DTM1
MBOX_IB_DATA_PARITY_H[03:02]	VIC-CL#7	DTA-DTM2, DTM3

Table A-1 (Cont.) IBox-VIC Signals

Input	Destination	Origination
MBOX_IB_DATA_PARITY_H[05:04]	VIC-CL#7	DTB-DTM0, DTM1
MBOX_IB_DATA_PARITY_H[07:06]	VIC-CL#7	DTA-DTM2, DTM3
MBOX_IB_PAGE_FAULT_L[00]	VIC-PCVC	VAP-FALT
MBOX_IB_RESPONSE_H[00]	VIC-PCVC	CTU-CTMV
MBOX_IB_RESPONSE_TA_H[00]	VIC-PCBP	CTU-CTMV
Output	Origination	Destination
IBOX_ABORT_H[00]	VIC-PCVC	MBOX-VAP-VAPO
IBOX_ABORT_L[00]	VIC-PCVC	MBOX-VAP-CCSQ
IBOX_IB_ABORT_H[00]	VIC-PCVC	MBOX-VAP-VAPO
IBOX_IB_ABORT_L[00]	VIC-PCVC	MBOX-VAP-CCSQ
IBOX_IB_ADDRESS_H[05:00]	VIC-PCBP	MBOX-VAP-VAPO
IBOX_IB_ADDRESS_H[12:06]	VIC-PCVC	MBOX-VAP-VAPO
IBOX_IB_ADDRESS_H[31:13]	VIC-PCVC	MBOX-VAP-FXUP
IBOX_IB_ADDRESS_PARITY_H[00]	VIC-PCBP	MBOX-VAP-VAPO
IBOX_IB_ADDRESS_PARITY_H[01]	VIC-PCVC	MBOX-VAP-VAPO
IBOX_PC_H[05:00]	VIC-PCBP	EBOX-CTL-QPCS
IBOX_PC_H[12:06]	VIC-PCVC	EBOX-CTL-QPCS
IBOX_PC_PARITY_H[00]	VIC-PCBP	EBOX-CTL-QPCS
IBOX_PC_PARITY_H[01]	VIC-PCVC	EBOX-CTL-QPCS
IBOX_CORRECTION_H[00]	XBR-PCHI	EBOX-INT-USQC

Table A-2 IBox-XBR Signals

Input	Destination	Origination
EBOX_BRANCH_A_L[00]	XBR-PCHI	INT-USQC
EBOX_BRANCH_A_L[00]	XBR-PCLO	INT-USQC
EBOX_BRANCH_VALID_A_L[00]	XBR-PCHI	INT-USQC
EBOX_BRANCH_VALID_A_L[00]	XBR-PCLO	INT-USQC
EBOX_KEEP_MASKS_L[01:00]	XBR-XDTB	CTL-iSSA
EBOX_RESULT_L[23:08]	XBR-PCLO	MUL-RET0
EBOX_RESULT_L[31:24]	XBR-PCHI	MUL-RET1
EBOX_RESULT_PARITY_L[02:01]	XBR-PCLO	MUL-RET0
EBOX_RESULT_PARITY_L[03]	XBR-PCHI	MUL-RET1
EBOX_RLOG_POINTER_H[02:00]	XBR-XDTB	DST-SRCS
EBOX_UNSPEND_H[00]	XBR-XDTA	INT-USQA
MBOX_IB_PAGE_FAULT_H[00]	XBR-IBFA	VAP-FALT
MBOX_IB_RESPONSE_TA_L[00]	XBR-IBFA	CTU-CTMV
MBOX_IB_RESPONSE_TA_L[00]	XBR-PCHI	CTU-CTMV
MBOX_IB_RESPONSE_TA_L[00]	XBR-PCLO	CTU-CTMV

Table A-2 (Cont.) IBox-XBR Signals

Output	Origination	Destination
IBOX_FORK_ADDRESS_H[03:00]	XBR-IBFA	EBOX-CTL-FRAMX
IBOX_FORK_ADDRESS_H[07:04]	XBR-IBFB	EBOX-CTL-FRAMX
IBOX_FORK_ADDRESS_H[08]	XBR-XSCA	EBOX-CTL-FRAMX
IBOX_FORK_ADDRESS_PARITY_H[00]	XBR-IBFB	EBOX-CTL-QPTR
IBOX_FORK_ERROR_H[00]	XBR-XSCA	EBOX-CTL-ISSE
IBOX_FORK_VALID_H[00]	XBR-XSCA	EBOX-CTL-QTPR
IBOX_IB_ADDRESS_H[21:13]	XBR-PCLO	MBOX-VAP-FXUP
IBOX_IB_ADDRESS_H[31:22]	XBR-PCHI	MBOX-VAP-FXUP
IBOX_IB_ADDRESS_PARITY_H[02]	XBR-PCLO	MBOX-VAP-FXUP
IBOX_IB_ADDRESS_PARITY_H[03]	XBR-PCHI	MBOX-VAP-FXUP
IBOX_IB_PAGE_FAULT_H[00]	XBR-XSCA	EBOX-CTL-ISSA, ISSB, ISSC
IBOX_IB_REQUEST_H[00]	XBR-IBFA	MBOX-VAP-VAPO
IBOX_INSTRUCTION_DECODED_H[00]	XBR-XSCA	EBOX-CTL-ISSA, ISSB, ISSC
IBOX_PC_H[21:13]	XBR-PCLO	EBOX-CTL-QPCS
IBOX_PC_H[31:22]	XBR-PCHI	EBOX-CTL-QPCS
IBOX_PC_PARITY_H[02]	XBR-PCLO	EBOX-CTL-QPCS
IBOX_PC_PARITY_H[03]	XBR-PCHI	EBOX-CTL-QPCS
IBOX_PC_VALID_H[00]	XBR-PCLO	EBOX-CTL-QPCS
IBOX_PREDICTION_H[00]	XBR-PCHI	EBOX-CTL-QPTR
IBOX_RAF_H[00]	XBR-XDTA	EBOX-CTL-ISSA, ISSB, ISSC
IBOX_REGISTER_FORK_H[00]	XBR-XDTA	EBOX-CTL-QTPR

Table A-3 IBox-OPU Signals

Input	Destination	Origination
EBOX_BRANCH_B_H[00]	OPU-OSQA	USQ-USQC
EBOX_BRANCH_VALID_B_H[00]	OPU-OSQA	USQ-USQC
EBOX_FLUSH_H[02:00]	OPU-OCTL, OQSA, OSQB	USQ-USQA
EBOX_GPR_BYTE0_WRITE_H[00]	OPU-STG2	CTL-ISSC
EBOX_GPR_BYTE1_WRITE_H[00]	OPU-STG2	CTL-ISSC
EBOX_GPR_H[03:00]	OPU-STG2	CTL-ISSC
EBOX_GPR_H[03:00]	OPU-STG3	CTL-ISSC
EBOX_GPR_WORD1_WRITE_H[00]	OPU-STG3	CTL-ISSC
EBOX_INSTRUCTION_DONE_H[00]	OPU-OCTL	CTL-ISSA
EBOX_INTERRUPT_H[00]	OPU-OSQA	USQ-USQA
EBOX_KEEP_MASKS_H[01:00]	OPU-OCTL	CTL-ISSA
EBOX_LAST_POINTER_H[03:00]	OPU-OSQB	CTL-ISSA
EBOX_QUEUE_FULL_H[00]	OPU-OSQB	CTL-QPTR
EBOX_RESULT_H[15:00]	OPU-STG2	MUL-RET0+RET1
EBOX_RESULT_H[31:16]	OPU-STG3	MUL-RET0+RET1
EBOX_RESULT_PARITY_H[01:00]	OPU-STG2	MUL-RET0+RET1
EBOX_RESULT_PARITY_H[03:02]	OPU-STG3	MUL-RET0+RET1
EBOX_RLOG_FULL_H[00]	OPU-OSQB	INT-RLOG
MBOX_OP_DATA_H[15:00]	OPU-OPUA	DTA-DTM0, DTM1
MBOX_OP_DATA_H[31:16]	OPU-OPUB	DTA-DTM2, DTM3
MBOX_OP_DATA_PARITY_H[01:00]	OPU-OPUA	DTA-DTM0, DTM1
MBOX_OP_DATA_PARITY_H[03:02]	OPU-OPUB	DTA-DTM2, DTM3
MBOX_OP_GRANT_H[00]	OPU-OSQA	VAP-VAPO
MBOX_OP_RESPONSE_H[00]	OPU-OSQA, OPUA, OPUB	CTU-CTMV
VBOX_ADDRESS_H[15:00]	OPU-STG2	VBOX-VAD-VMKA
VBOX_ADDRESS_H[31:16]	OPU-STG3	VBOX-VAD-VMKA
VBOX_ADDRESS_PARITY_H[01:00]	OPU-STG2	VBOX-VAD-VMKA
VBOX_ADDRESS_PARITY_H[03:02]	OPU-STG3	VBOX-VAD-VMKA
VBOX_ADDRESS_VALID_H[00]	OPU-OSQA, OSQB	VBOX-VAD-VMKB
VBOX_BLOCK_READ_H[00]	OPU-OSQA, OSQB	VBOX-VAD-VMKB
VBOX_READ_NOP_H[00]	OPU-OSQA, OSQB	VBOX-VAD-VMKB
VBOX_REFERENCE_SIZE_H[00]	OPU-OSQA, OSQB	VBOX-UCS-VCTC
VBOX_REFERENCE_TYPE_H[00]	OPU-OSQA, OSQB	VBOX-UCS-VCTA

Table A-3 (Cont.) IBox-OPU Signals

Output	Origination	Destination
IBOX_DATA_H[15:00]	OPU-OPUA	EBOX-DST-STG0
IBOX_DATA_H[31:16]	OPU-OPUB	EBOX-DST-STG1
IBOX_DATA_PARITY_H[01:00]	OPU-OPUA	EBOX-DST-STG0
IBOX_DATA_PARITY_H[03:02]	OPU-OPUB	EBOX-DST-STG1
IBOX_DATA_ERROR_H[00]	OPU-OSQB	EBOX-CTL-ISSE
IBOX_DATA_TAG_H[03:00]	OPU-OSQA	EBOX-CTL-ISSA, ISSB, ISSC, DST-SRCS
IBOX_DATA_VALID_H[00]	OPU-OSQA	EBOX-CTL-ISSA, ISSB, ISSC, DST-SRCS
IBOX_DATA_VALID_L[00]	OPU-OSQA	EBOX-CTL-ISSA, ISSB, ISSC, DST-SRCS
IBOX_DESTINATION_MEMORY_H[00]	OPU-OSQB	EBOX-CTL-ISSA, ISSB, ISSC, QPTR
IBOX_DESTINATION_POINTER_H[03:00]	OPU-OSQB	EBOX-CTL-ISSA, ISSB, ISSC, QPTR
IBOX_DESTINATION_VALID_H[00]	OPU-OSQB	EBOX-CTL-ISSA, ISSB, ISSC, QPTR
IBOX_DST_DATA_TAG_H[03:00]	OPU-OSQA	EBOX-DST-STG0, STG1
IBOX_DST_GPR_H[03:00]	OPU-OSQA	EBOX-DST-STG0, STG1
IBOX_FLUSH_ABORT_H[00]	OPU-OCTL	MBOX-VAP-VAPO
IBOX_FLUSH_ABORT_L[00]	OPU-OCTL	MBOX-VAP-CCSQ
IBOX_FREE_POINTER_H[03:00]	OPU-OSQB	EBOX-ISSA, ISSB, ISSC
IBOX_GPR_H[03:00]	OPU-OSQA	EBOX-DST-SRCS
IBOX_GPR_WRITE_H[00]	OPU-OSQA	EBOX-DST-SRCS
IBOX_OP_ADDRESS_H[31:16]	OPU-OPUB	MBOX-VAP-FXUP
IBOX_OP_ADDRESS_H[15:00]	OPU-OPUA	MBOX-VAP-FXUP
IBOX_OP_ADDRESS_PARITY_H[01:00]	OPU-OPUA	MBOX-VAP-VAPO
IBOX_OP_ADDRESS_PARITY_H[03:02]	OPU-OPUB	MBOX-VAP-FXUP
IBOX_OP_CONTEXT_H[02:00]	OPU-OSQA	MBOX-VAP-VAPO
IBOX_OP_CONTROL_H[02:00]	OPU-OSQA	MBOX-VAP-VAPO
IBOX_OP_CONTROL_PARITY_H[00]	OPU-OSQA	MBOX-VAP-VAPO
IBOX_OP_INDIRECT_H[01:00]	OPU-OSQA	MBOX-VAP-VAPO
IBOX_OP_REQUEST_H[00]	OPU-OSQA	MBOX-VAP-VAPO
IBOX_OP_TAG_H[03:00]	OPU-OSQA	MBOX-VAP-VAPO
IBOX_OSQA_ISSA_PARITY_H[00]	OPU-OSQA	EBOX-CTL-ISSA
IBOX_OSQA_RLOG_PARITY_H[00]	OPU-OSQA	EBOX-INT-RLOG

Table A-3 (Cont.) IBox-OPU Signals

Output	Origination	Destination
IBOX_OSQB_ISSB_PARITY_H[00]	OPU-OSQB	EBOX-CTL-ISSB
IBOX_OSQB_QPTR_PARITY_H[00]	OPU-OSQB	EBOX-CTL-QPTR
IBOX_OSQB_SRCS_PARITY_H[00]	OPU-OSQB	EBOX-DST-SRCS
IBOX_POINTER_ERROR_H[00]	OPU-OSQB	EBOX-CTL-ISSE
IBOX_RLOG_COMPLETE_H[00]	OPU-OSQA	EBOX-DST-SRCS
IBOX_RLOG_CONTEXT_H[03:00]	OPU-OSQA	EBOX-DST-SRCS
IBOX_RLOG_TAG_H[02:00]	OPU-OSQA	EBOX-DST-SRCS
IBOX_RLOG_WRITE_H[00]	OPU-OSQA	EBOX-DST-SRCS
IBOX_SOURCE1_POINTER_H[04:00]	OPU-OSQB	EBOX-CTL-QPTR
IBOX_SOURCE1_POINTER_L[04:00]	OPU-OSQB	EBOX-CTL-SRCS
IBOX_SOURCE1_VALID_H[00]	OPU-OSQB	EBOX-CTL-QPTR
IBOX_SOURCE1_VALID_L[00]	OPU-OSQB	EBOX-CTL-SRCS
IBOX_SOURCE2_POINTER_H[04:00]	OPU-OSQB	EBOX-CTL-QPTR
IBOX_SOURCE2_POINTER_L[04:00]	OPU-OSQB	EBOX-CTL-SRCS
IBOX_SOURCE2_VALID_H[00]	OPU-OSQB	EBOX-CTL-QPTR
IBOX_SOURCE2_VALID_L[00]	OPU-OSQB	EBOX-CTL-SRCS

Index

B

- Branch bias, 4-30
- Branch prediction, 4-27
 - demote, 4-29
 - primary prediction hit, 4-28
 - primary predictions, 4-27
 - secondary predictions, 4-30
- Branch prediction cache
 - match enable, 4-29

C

- Complex specifier unit (CSU), 5-4
 - adder, 5-8
 - AMUX, 5-7
 - BMUX, 5-7
 - CSU microcode control, 5-12
 - current PC generation, 5-10
 - OPUA data path, 5-5
 - OPUB data path, 5-8

E

- EBox-to-IBox interface, 5-44

F

- Free pointer logic (FPL), 5-25
 - destination pointer, 5-30
 - free pointer, 5-29
 - free pointer initialization, 5-30
 - source 1, 5-25
 - source 2, 5-27

I

- IBox error registers, 6-1
 - decode error register 1, 6-5
 - fetch error register 1, 6-1
 - fetch error register 2, 6-4
 - specifier error register 1, 6-8
 - specifier error register 2, 6-10
 - XBAR decode error register, 6-6
- IBox-to-EBox interface, 5-40

- Instruction buffer, 3-9
 - IBEX, 3-11
 - IBEX2, 3-10
 - IBUF, 3-18
 - instruction buffer parity, 3-19
 - merger, 3-14
 - rotator, 3-11
 - shifter, 3-16
- Instruction buffer interface, 3-22
 - aborting requests, 3-23
 - page faults, 3-23
- Instruction decode, 4-1
- Instruction fetch, 3-1 to 3-23
- Intra-instruction read conflict
 - XBAR IRC logic, 4-23
- Intra-instruction read conflicts (IRC), 4-23

M

- MCU, 1-5
 - OPU, 1-8
 - VIC, 1-6
 - XBR, 1-7

O

- Operand control (OCTL), 5-32
 - flush, 5-35
 - read and write register mask parity, 5-35
 - read and write register masks, 5-32
 - stalls, 5-37
- OPU port interface, 5-38

P

- PCU microcode, 4-31
- Pipeline description, 1-9 to 1-12
- Program counter, 2-1
 - branch PC, 2-6
 - branch PC data path, 2-6
 - cache control, 2-8
 - decode PC, 2-4
 - decode PC data path, 2-5
 - PCU errors, 2-9
 - prefetch PC, 2-1

2 Index

Program counter (cont'd.)

- prefetch PC data path, 2-3
- target PC, 2-4
- unwind PC, 2-7
- unwind PC data path, 2-7

S

Short literal specifier handler (SLU), 5-19

- block diagram, 5-19
- floating-point expansion, 5-22
- integer expansion, 5-21
- SLU output, 5-24
- SLU parity protection, 5-24
- stalls, 5-24

Specifier decode, 5-1

V

VBox interface, 5-47

- Virtual instruction cache, 3-1
 - disabling VIC hit, 3-8

VIC flush, 3-7

VIC hit, 3-2

VIC parity, 3-8

VIC read, 3-7

VIC write, 3-2

X

XBAR, 4-1

- decode tree logic, 4-7

DRAM, 4-4

fork logic, 4-14

request logic, 4-9

shift counts, 4-13

simple decode, 4-6

specifier counts, 4-12

XBAR displacement data path, 4-14

XBAR IRC logic, 4-23

XBAR short literal data path, 4-16

XBAR source and destination logic, 4-18

XRAM, 4-5