NVAX Plus CPU Chip
Functional Specification

The NVAX Plus CPU Chip is a high-performance, single-chip implementation of the VAX Architecture for use in low-end and mid-range systems.

Revision/Update Information: This is Revision 0.1 of this specification, the initial external release

Semiconductor Engineering Group
Digital Equipment Corporation, Hudson, Massachusetts

This is copy number

February 1991

The information in this document may be changed without notice and is not a commitment by Digital Equipment Corporation. Digital Equipment Corporation is not responsible for any errors in this document.

This specification does not describe any program or product that is currently available from Digital Equipment Corporation, nor is Digital Equipment Corporation committed to implement this specification in any program or product. Digital Equipment Corporation makes no commitment that this document accurately describes any product it might ever make.

The following are trademarks of Digital Equipment Corporation:

DEC	ULTRIX	VAXstation
DECnet	ULTRIX–32	VMS
DECUS	UNIBUS	VT
MicroVAX	VAX	
MicroVMS	VAXBI	
PDP	VAXcluster	

digital™

Contents

FIGURES

TABLES

Chapter 1

Introduction

The NVAX PLUS CPU is a high-performance, single-chip implementation of the VAX architecture. It is partitioned into multiple sections which cooperate to execute the VAX base instruction group. The CPU chip includes the first levels of the memory subsystem hierarchy in an on-chip virtual instruction cache and an on-chip physical instruction and data cache, as well as the controller for a large second-level cache implemented in static RAMs on the CPU module.

The NVAX Plus chip is an NVAX core with an EVAX external interface. Microcode changes are also required to support the EVAX interlocks and to input from serial ROM at startup. Most of the CBOX-MBOX interface section is reused. The CBOX arbitration logic is redesigned to control the EDAL interface. Cache fills and coherency transactions are controlled by EDAL system logic with only a single CPU request active at a time.

1.1 Scope and Organization of this Specification

This specification describes the operation of the NVAX PLUS chip. It contains an Architecturial Summary, a description of the interface to the chip, an overview of the operation of the instruction pipeline, and extensive detail about the functional operation of the CBOX section of the chip.

The IBOX, EBOX, MBOX, FBOX, and Interrupt sections are taken from the NVAX CPU Functional Specification. These sections retain the high level description of the section, the description of the software visible IPRs, and specify the changes required by NVAX Plus to accommodate the EVAX interface and Vector option. Sections which aid in understanding the interface between the NVAX Plus CBOX and NVAX Core are also retained. For a detailed desription of the IBOX, EBOX, MBOX, FBOX, and Interrupt sections refer to the NVAX CPU Chip Functional Specification.

In addition, the specification contains discussions of error handling, chip initialization, and testability features.

1.2 Related Documents

The following documents are related to or were used in the preparation of this document:

- NVAX CPU Chip Functional Specification
- EV3 and EV4 Specification
- DEC Standard 032 VAX Architecture Standard.

- NVAX CPU Chip Design Methodology.

1.3 Terminology and Conventions

1.3.1 Numbering

All numbers are decimal unless otherwise indicated. Where there is ambiguity, numbers other than decimal are indicated with the name of the base following the number in parentheses, e.g., FF (hex).

1.3.2 UNPREDICTABLE and UNDEFINED

RESULTS specified as UNPREDICTABLE may vary from moment to moment, implementation to implementation, and instruction to instruction within implementations. Software can never depend on results specified as UNPREDICTABLE.

OPERATIONS specified as UNDEFINED may vary from moment to moment, implementation to implementation, and instruction to instruction within implementations. The operation may vary in effect from nothing, to stopping system operation. UNDEFINED operations must not cause the processor to hang., i.e., reach a state from which there is no transition to a normal state in which the machine executes instructions.

Note the distinction between result and operation. Non-privileged software can not invoke UNDEFINED operations.

1.3.3 Ranges and Extents

Ranges are specified by a pair of numbers separated by a ".." and are inclusive, e.g., a range of integers 0..4 includes the integers 0, 1, 2, 3, and 4.

Extents are specified by a pair of numbers in angle brackets separated by a colon and are inclusive, e.g., bits <7:3> specify an extent of bits including bits 7, 6, 5, 4, and 3.

1.3.4 Must be Zero (MBZ)

Fields specified as Must Be Zero (MBZ) must never be filled by software with a non-zero value. If the processor encounters a non-zero value in a field specified as MBZ, a Reserved Operand exception occurs.

1.3.5 Should be Zero (SBZ)

Fields specified as Should Be Zero (SBZ) should be filled by software with a zero value. These fields may be used at some future time. Non-zero values in SBZ fields produce UNPREDICTABLE results.

1.3.6 Register Format Notation

This specification contains a number of figures that show the format of various registers, followed by a description of each field. In general, the fields on the register are labeled with either a name or a mnemonic. The description of each field includes the name or mnemonic, the bit extent, and the type. An example of a register is shown in Figure 1–1. Table 1–1 is an example of the description of the fields in this register.

Figure 1–1: Register Format Example

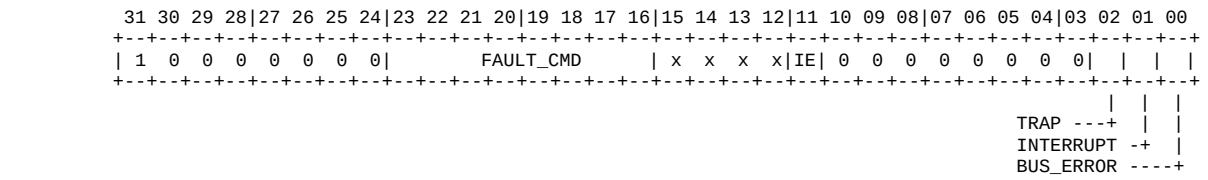


Table 1–1: Register Field Description Example

Name	Bit(s)	Type	Description
BUS_ERROR	0	WC,0	The BUS_ERROR bit is set when a bus error is detected.
INTERRUPT	1	WC,0	The INTERRUPT bit is set when an error that is reported as an interrupt is detected.
TRAP	2	WC,0	The TRAP bit is set when an error that is reported as a trap is detected.
IE	11	RW,0	The IE bit enables error reporting interrupts. When IE is 0, interrupts are disabled. When IE is a 1, interrupts are enabled.
FAULT_CMD	23:16	RO	The FAULT_CMD field latches the command that was in progress when an error is detected.

The "Type" column in the field description includes both the actual type of the field, and an optional initialized value, separated from the type by a comma. The type denotes the functional operation of the field, and may be one of the values shown in Table 1–2. If present, the initialized value indicates that the field is initialized by hardware or microcode to the specified value at powerup. If the initialized value is not present, the field is not initialized at powerup.

Table 1–2: Register Field Type Notation

Notation	Description
RW	A read-write bit or field. The value may be read and written by software, microcode, or hardware.
RO	A read-only bit or field. The value may be read by software, microcode, or hardware. It is written by hardware; software or microcode writes are ignored.
WO	A write-only bit or field. The value may be written by software or microcode. It is read by hardware and reads by software or microcode return an UNPREDICTABLE result.

Table 1–2 (Cont.): Register Field Type Notation

Notation	Description
WZ	A write-only bit or field. The value may be written by software or microcode. It is read by hardware and reads by software or microcode return a 0.
WC	A write-one-to-clear bit. The value may be read by software or microcode. Software or microcode writes of a 1 cause the bit to be cleared by hardware. Software or microcode writes of a 0 do not modify the state of the bit.
RC	A read-to-clear field. The value is written by hardware and remains unchanged until read. The value may be read by software or microcode, at which point, hardware may write a new value into the field.

In addition to named fields in registers, other bits of the register may be labeled with one of the three symbols listed in Table 1–3. These symbols denote the type of the unnamed fields in the register.

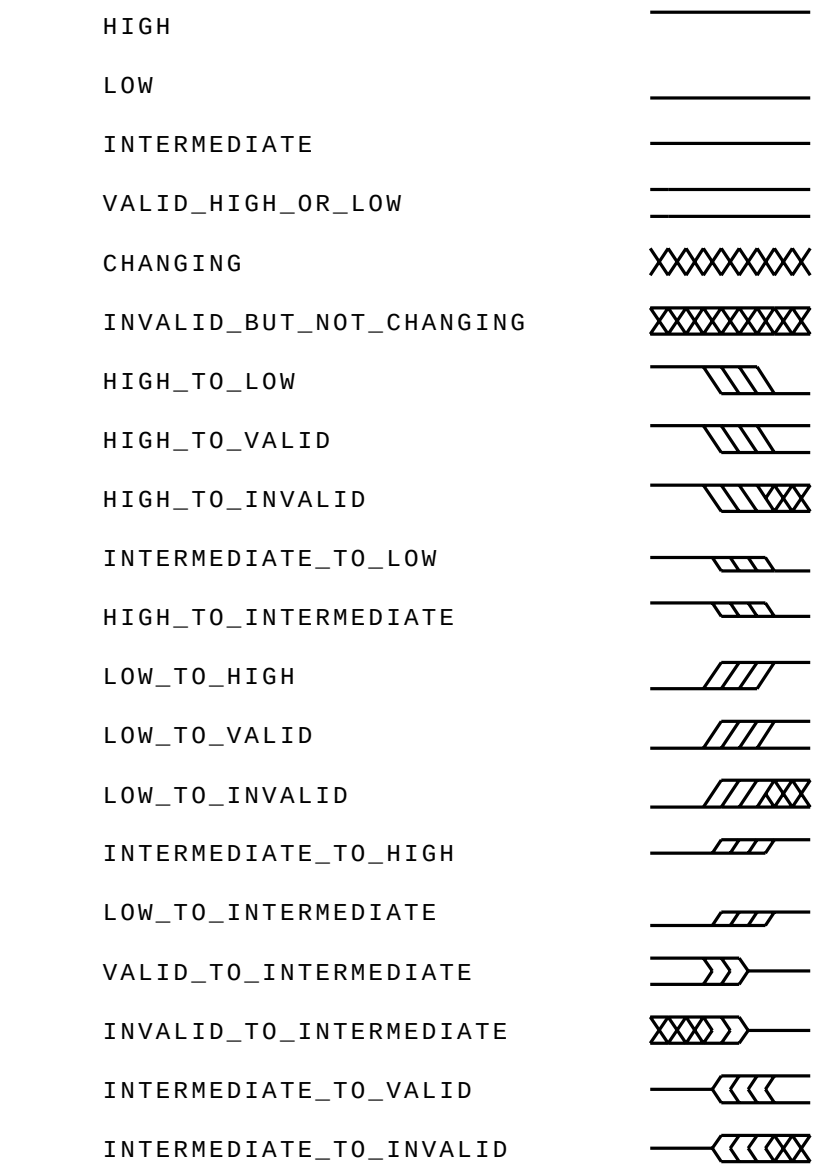
Table 1–3: Register Field Notation

Notation	Description
0	A "0" in a bit position denotes a register bit that is read as a 0 and ignored on write.
1	A "1" in a bit position denotes a register bit that is read as a 1 and ignored on write.
x	An "x" in a bit position denotes a register bit that does not exist in hardware. The value is UNPREDICTABLE when read, and ignored on write.

1.3.7 Timing Diagram Notation

This specification contains a number of timing diagrams that show the timing of various signals, including NDAL signals. The notation used in these timing diagrams is shown in Figure 1–2.

Figure 1–2: Timing Diagram Notation



1.4 Revision History

Table 1–4: Revision History

Who	When	Description of change
Mike Uhler	06-Mar-1989	Release for external review.
Mike Uhler	15-Dec-1989	Update for second-pass release.
Gil Wolrich	15-Nov-1990	NVAX PLUS release for external review.

Chapter 2

Architectural Summary

2.1 Overview

This chapter provides a summary of the VAX architectural features of the NVAX Plus CPU Chip. It is not intended as a complete reference but rather to give an overview of the user-visible features. For a complete description of the architecture, consult the *VAX Architecture Standard* (DEC Standard 032).

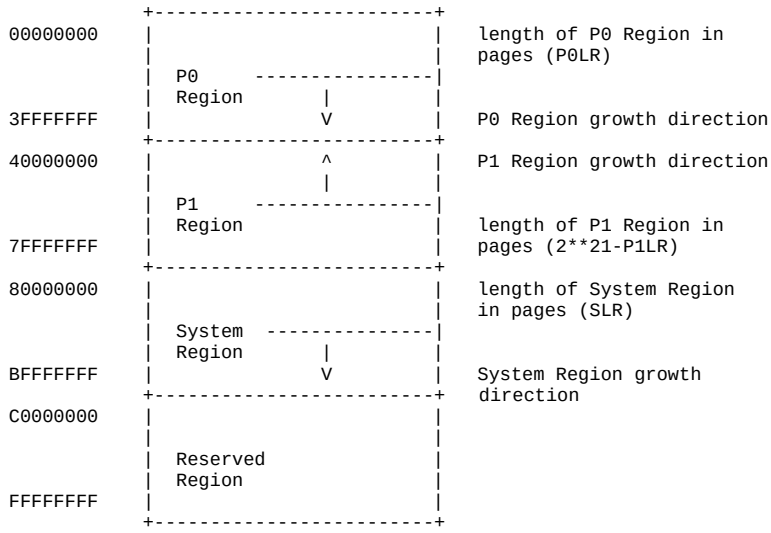
2.2 Visible State

The visible state of the processor consists of memory, both virtual and physical, the general registers, the processor status longword (PSL), and the privileged internal processor registers (IPRs).

2.2.1 Virtual Address Space

The virtual address space is four gigabytes (2**32), separated into three accessable regions (P0, P1, and S0) and one reserved region, as shown in Figure 2–1.

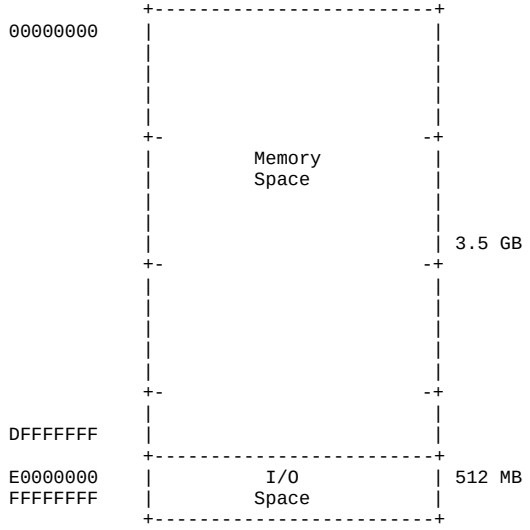
Figure 2–1: Virtual Address Space Layout



2.2.2 Physical Address Space

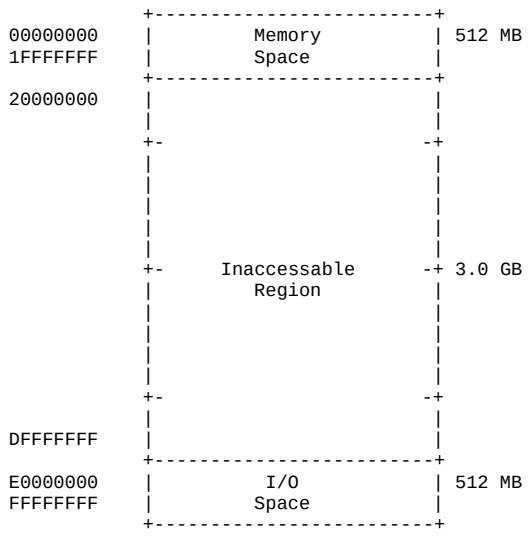
The NVAX Plus CPU naturally generates 32-bit physical addresses. This corresponds to a four gigabyte physical address space as shown in Figure 2–2. Memory space occupies the first seven-eighths (3.5GB) of the physical address space. I/O space occupies the last one-eighth (512MB) of the physical address space and can be distinguished from memory space by the fact that bits <31:29> of the physical address are all ones.

Figure 2–2: 32-bit Physical Address Space Layout



In addition to the natural 32-bit physical address, the CPU may be configured to generate 30-bit physical addresses. In this mode, only 512MB of memory space can be referenced, as shown in Figure 2–3.

Figure 2–3: 30-bit Physical Address Space Layout



The translation from 30-bit addresses to 32-bit addresses is accomplished by sign-extending PA<29> to PA<31:30>. In this mode, the programmer sees a 1GB address space, split evenly between memory and I/O space, which is mapped to the actual 32-bit physical address space as shown in Table 2–1. Unless explicitly stated otherwise, addresses that are given in the remainder

of this specification are the full 32-bit addresses (which, of course, may have been generated from a 30-bit program address via the mapping shown).

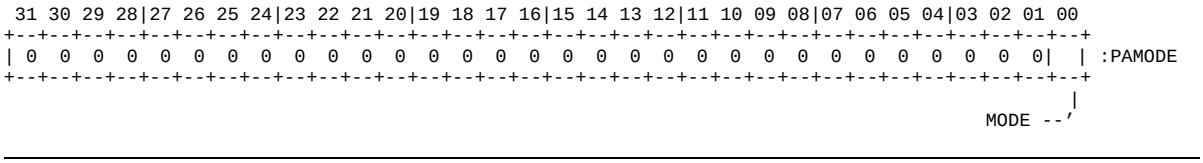
Table 2–1: 30-bit Mapping of Program Addresses to 32-bit Hardware Addresses

Program Address	Hardware Address
00000000..1FFFFFFF	00000000..1FFFFFFF
20000000..3FFFFFFF	E0000000..FFFFFFFF

2.2.2.1 Physical Address Control Registers

During powerup, microcode configures the CPU to generate 30-bit physical addresses. Console firmware may then reconfigure the CPU to generate either 30-bit or 32-bit physical addresses by writing to the MODE bit in the PAMODE and VPAMODE registers, respectively. The PAMODE register is shown in Figure 2–4.

Figure 2–4: PAMODE Register



The VPAMODE register is identical in format to the PAMODE register.

The PAMODE register also determines how PTEs are to be interpreted. In 30-bit mode, PTEs are interpreted in 21-bit PFN format. In 32-bit mode, PTEs are interpreted in 25-bit PFN format (although the two upper bits of the PFN field are ignored). The different PTE formats are described in Section 2.6.4.

2.2.3 Registers

There are 16 32-bit General Purpose Registers (GPRs). The format is shown in Figure 2–5, and the use of each GPR is shown in Table 2–2.

Figure 2–5: General Purpose Registers

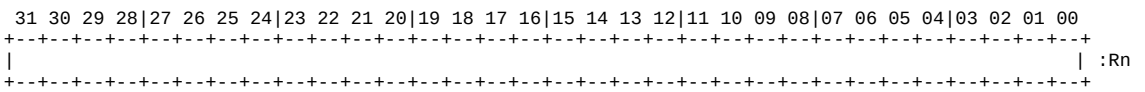


Table 2–2: General Purpose Register Usage

GPR	Synonym	Use
R0–R11		General Purpose
R12	AP	Argument Pointer
R13	FP	Frame Pointer
R14	SP	Stack Pointer
R15	PC	Program Counter

The Processor Status Longword (PSL) is a 32-bit register which contains processor state. The PSL format is shown in Figure 2–6, and the fields of the PSL are shown in Table 2–3.

Figure 2–6: Processor Status Longword Fields

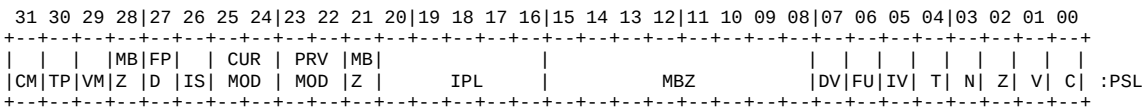


Table 2–3: Processor Status Longword

Name	Bit(s)	Description
CM	31	Compatability Mode
TP	30	Trace Pending
VM	29	Virtual Machine Mode[1]
FPD	27	First Part Done
IS	26	Interrupt Stack
CUR_MOD	25:24	Current Mode
PRV_MOD	23:22	Previous Mode
IPL	20:16	Interrupt Priority Level
DV	7	Decimal Overflow Trap Enable
FU	6	Floating Underflow Fault Enable
IV	5	Integer Overflow Trap Enable
T	4	Trace Trap Enable

[1]MBZ unless virtual machine option is implemented

Table 2–3 (Cont.): Processor Status Longword

Name	Bit(s)	Description
N	3	Negative Condition Code
Z	2	Zero Condition Code
V	1	Overflow Condition Code
C	0	Carry Condition Code

2.3 Data Types

The NVAX Plus CPU supports nine data types: byte, word, longword, quadword, character string, variable length bit field, F_floating, D_floating, and G_floating. These are summarized in Figure 2–7.

Figure 2–7: Data Types

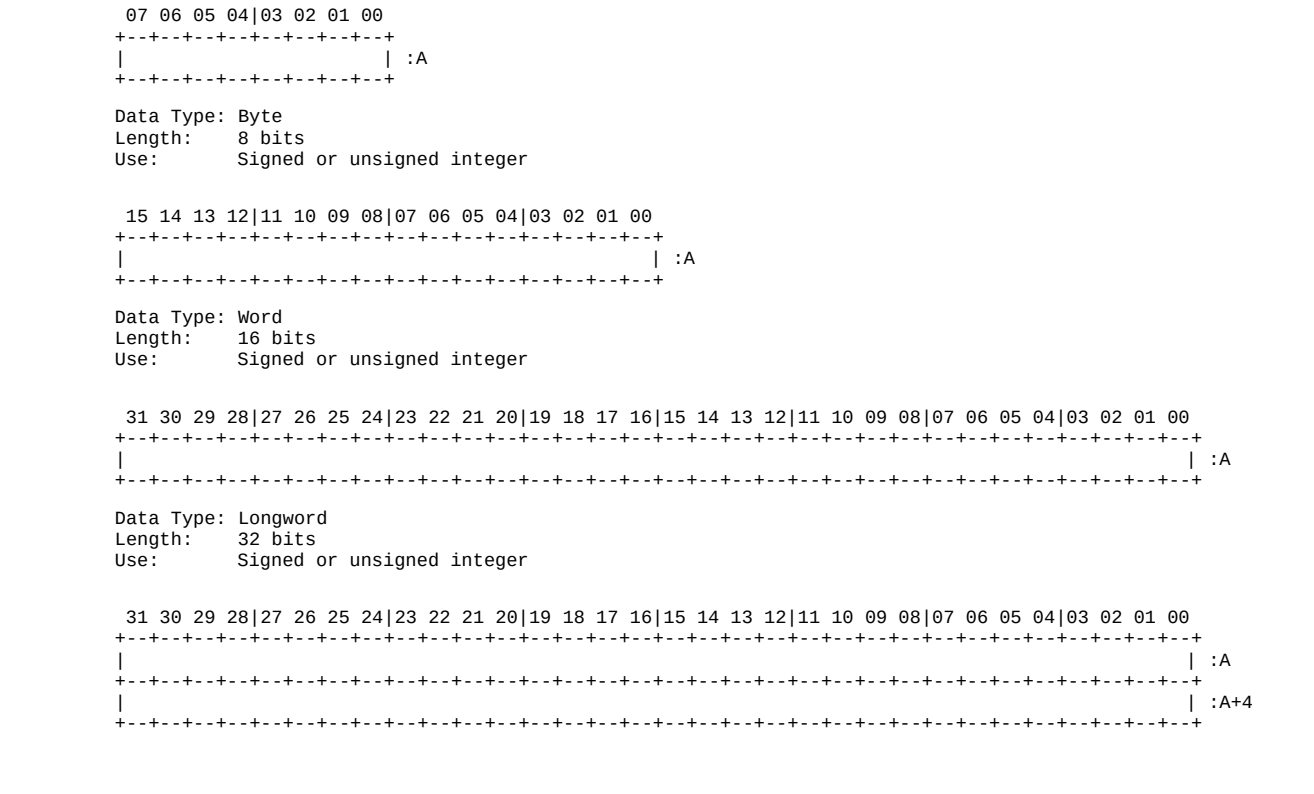


Figure 2–7 Cont'd. on next page

Figure 2–7 (Cont.): Data Types

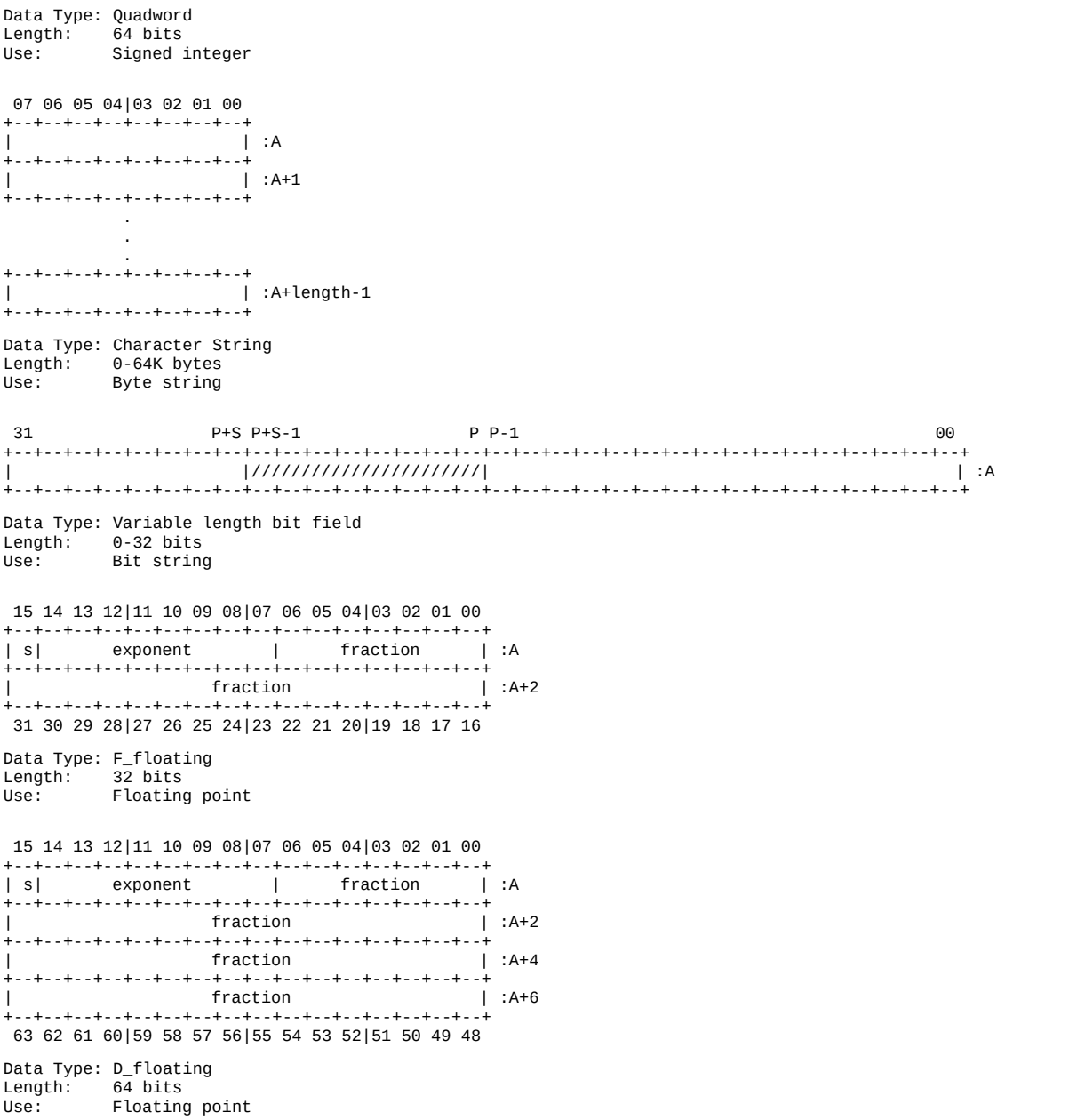
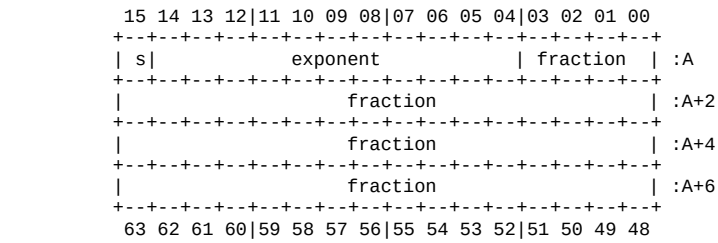


Figure 2–7 Cont'd. on next page

Figure 2–7 (Cont.): Data Types




```
Data Type: G_floating
Length:    64 bits
Use:       Floating point
```

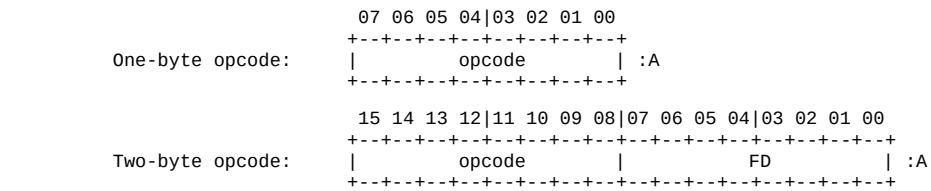
2.4 Instruction Formats and Addressing Modes

VAX instructions consist of a one- or two-byte opcode, followed by zero to six operand specifiers.

2.4.1 Opcode Formats

An opcode may be either one or two contiguous bytes. The two-byte format begins with an FD (hex) byte and is followed by a second opcode byte. The one-byte format is indicated by an opcode byte whose value is anything other than FD (hex). The one- or two-byte opcode format is shown in Figure 2–8.

Figure 2–8: Opcode Formats



2.4.2 Addressing Modes

An operand specifier starts with a specifier byte and may be followed by a specifier extension. Bits <3:0> of the specifier byte contain a GPR number and bits <7:4> of the specifier byte indicate the addressing mode of the specifier. If the register number in the specifier byte does not contain 15, the addressing mode is a general register addressing mode. If the register number in the specifier byte does contain 15, the addressing mode is a PC-relative addressing mode. The

different addressing modes are shown graphically in Figure 2–9. General register addressing modes are listed in Table 2–4 and PC-relative addressing modes are listed in Table 2–5.

Figure 2–9: Addressing Modes

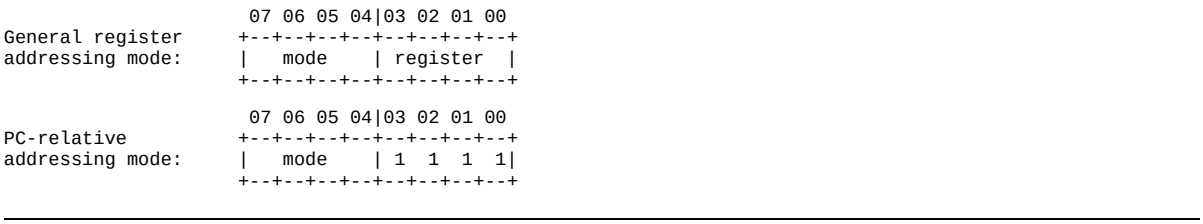


Table 2–4: General Register Addressing Modes

Mode	Name	Assembler	Access r m w a v	PC	SP	Indexable?
0-3	literal	S^#literal	y f f f f	x	x	f
4	index	i[Rx]	y y y y y	u	y	f
5	register	Rn	y y y f y	u	uq	f
6	register deferred	(Rn)	y y y y y	u	y	y
7	autodecrement	-(Rn)	y y y y y	u	y	ux
8	autoincrement	(Rn)+	y y y y y	p	y	ux
9	autoincrement deferred	@(Rn)+	y y y y y	p	y	ux
A	byte displacement	B^d(Rn)	y y y y y	p	y	y
B	byte displacement deferred	@B^d(Rn)	y y y y y	p	y	y
C	word displacement	W^d(Rn)	y y y y y	p	y	y
D	word displacement deferred	@W^d(Rn)	y y y y y	p	y	y
E	longword displacement	L^d(Rn)	y y y y y	p	y	y
F	longword displacement deferred	@L^d(Rn)	y y y y y	p	y	y

Access Types

r = read
m = modify
w = write
a = address
v = variable bit field

Syntax

i = any indexable address mode
d = displacement
Rn = general register, n = 0 to 15
Rx = general register, n = 0 to 14

Results

y = yes, always valid address mode
f = reserved addressing mode fault
x = logically impossible
p = program counter addressing
u = unpredictable
ud = unpredictable for destination of CALLG, CALLS, JMP and JSB
uq = unpredictable for quad, D/G_floating and field if pos+size > 32
ux = unpredictable if index register = base register

Table 2–5: PC-Relative Addressing Modes

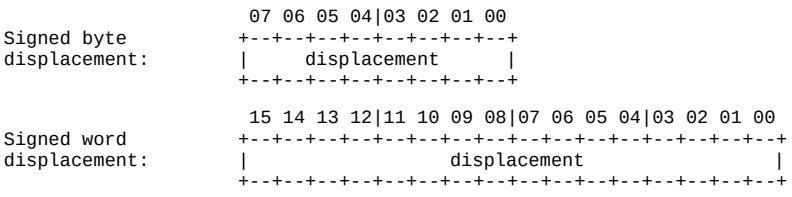
Mode	Name	Assembler	Access r m w a v	PC	SP	Indexable?
8	immediate	I^#constant	y u u y ud			u
9	absolute	@#address	y y y y y			y
A	byte relative	B^address	y y y y y			y
B	byte relative deferred	@B^address	y y y y y			y
C	word relative	W^address	y y y y y			y
D	word relative deferred	@W^address	y y y y y			y
E	longword relative	L^address	y y y y y			y
F	longword relative deferred	@L^address	y y y y y			y

For notation, refer to the key in Table 2–4

2.4.3 Branch Displacements

Branch instructions contain a one- or two-byte signed branch displacement after the final specifier (if any). The branch displacement is shown in Figure 2–10.

Figure 2–10: Branch Displacements



2.5 Instruction Set

The NVAX Plus CPU supports the VAX Base Instruction Group as defined in DEC Standard 032 plus the optional VAX vector instructions and the virtual machine instructions. These instructions are listed in Table 2–6.

Table 2–6: NVAX Instruction Set

Opcode	Instruction	N	Z	V	C	Exceptions
Integer, Arithmetic and Logical Instructions						
58	ADAWI add.rw, sum.mw	*	*	*	*	iov
80	ADDB2 add.rb, sum.mb	*	*	*	*	iov
C0	ADDL2 add.rl, sum.ml	*	*	*	*	iov
A0	ADDW2 add.rw, sum.mw	*	*	*	*	iov
81	ADDB3 add1.rb, add2.rb, sum.wb	*	*	*	*	iov
C1	ADDL3 add1.rl, add2.rl, sum.wl	*	*	*	*	iov
A1	ADDW3 add1.rw, add2.rw, sum.ww	*	*	*	*	iov
D8	ADWC add.rl, sum.ml	*	*	*	*	iov
78	ASHL cnt.rb, src.rl, dst.wl	*	*	*	0	iov
79	ASHQ cnt.rb, src.rq, dst.wq	*	*	*	0	iov
8A	BICB2 mask.rb, dst.mb	*	*	0	–	
CA	BICL2 mask.rl, dst.ml	*	*	0	–	
AA	BICW2 mask.rw, dst.mw	*	*	0	–	
8B	BICB3 mask.rb, src.rb, dst.wb	*	*	0	–	
CB	BICL3 mask.rl, src.rl, dst.wl	*	*	0	–	
AB	BICW3 mask.rw, src.rw, dst.ww	*	*	0	–	
88	BISB2 mask.rb, dst.mb	*	*	0	–	
C8	BISL2 mask.rl, dst.ml	*	*	0	–	
A8	BISW2 mask.rw, dst.mw	*	*	0	–	
89	BISB3 mask.rb, src.rb, dst.wb	*	*	0	–	
C9	BISL3 mask.rl, src.rl, dst.wl	*	*	0	–	
A9	BISW3 mask.rw, src.rw, dst.ww	*	*	0	–	
93	BITB mask.rb, src.rb	*	*	0	–	
D3	BITL mask.rl, src.rl	*	*	0	–	
B3	BITW mask.rw, src.rw	*	*	0	–	

Table 2–6 (Cont.): NVAX Instruction Set

Opcode	Instruction	N	Z	V	C	Exceptions
Integer, Arithmetic and Logical Instructions						
94	CLRB dst.wb	0	1	0	–	
D4	CLRL{=F} dst.wl	0	1	0	–	
7C	CLRQ{=D=G} dst.wq	0	1	0	–	
B4	CLRW dst.ww	0	1	0	–	
91	CMPB src1.rb, src2.rb	*	*	0	*	
D1	CMPL src1.rl, src2.rl	*	*	0	*	
B1	CMPW src1.rw, src2.rw	*	*	0	*	
98	CVTBL src.rb, dst.wl	*	*	0	0	
99	CVTBW src.rb, dst.ww	*	*	0	0	
F6	CVTLB src.rl, dst.wb	*	*	*	0	iov
F7	CVTLW src.rl, dst.ww	*	*	*	0	iov
33	CVTWB src.rw, dst.wb	*	*	*	0	iov
32	CVTWL src.rw, dst.wl	*	*	0	0	
97	DECB dif.mb	*	*	*	*	iov
D7	DECL dif.ml	*	*	*	*	iov
B7	DECW dif.mw	*	*	*	*	iov
86	DIVB2 divr.rb, quo.mb	*	*	*	0	iov, idvz
C6	DIVL2 divr.rl, quo.ml	*	*	*	0	iov, idvz
A6	DIVW2 divr.rw, quo.mw	*	*	*	0	iov, idvz
87	DIVB3 divr.rb, divd.rb, quo.wb	*	*	*	0	iov, idvz
C7	DIVL3 divr.rl, divd.rl, quo.wl	*	*	*	0	iov, idvz
A7	DIVW3 divr.rw, divd.rw, quo.ww	*	*	*	0	iov, idvz
7B	EDIV divr.rl, divd.rq, quo.wl, rem.wl	*	*	*	0	iov, idvz
7A	EMUL mulr.rl, muld.rl, add.rl, prod.wq	*	*	0	0	
96	INCB sum.mb	*	*	*	*	iov
D6	INCL sum.ml	*	*	*	*	iov

Table 2–6 (Cont.): NVAX Instruction Set

Opcode	Instruction	N	Z	V	C	Exceptions
Integer, Arithmetic and Logical Instructions						
B6	INCW sum.mw	*	*	*	*	iov
92	MCOMB src.rb, dst.wb	*	*	0	–	
D2	MCOML src.rl, dst.wl	*	*	0	–	
B2	MCOMW src.rw, dst.ww	*	*	0	–	
8E	MNEGB src.rb, dst.wb	*	*	*	*	iov
CE	MNEGL src.rl, dst.wl	*	*	*	*	iov
AE	MNEGW src.rw, dst.ww	*	*	*	*	iov
90	MOVB src.rb, dst.wb	*	*	0	–	
D0	MOVL src.rl, dst.wl	*	*	0	–	
7D	MOVQ src.rq, dst.wq	*	*	0	–	
B0	MOVW src.rw, dst.ww	*	*	0	–	
9A	MOVZBW src.rb, dst.wb	0	*	0	–	
9B	MOVZBL src.rb, dst.wl	0	*	0	–	
3C	MOVZWL src.rw, dst.wl	0	*	0	–	
84	MULB2 mulr.rb, prod.mb	*	*	*	0	iov
C4	MULL2 mulr.rl, prod.ml	*	*	*	0	iov
A4	MULW2 mulr.rw, prod.mw	*	*	*	0	iov
85	MULB3 mulr.rb, muld.rb, prod.wb	*	*	*	0	iov
C5	MULL3 mulr.rl, muld.rl, prod.wl	*	*	*	0	iov
A5	MULW3 mulr.rw, muld.rw, prod.ww	*	*	*	0	iov
DD	PUSHL src.rl, {-(SP).wl}	*	*	0	–	
9C	ROTL cnt.rb, src.rl, dst.wl	*	*	0	–	
D9	SBWC sub.rl, dif.ml	*	*	*	*	iov
82	SUBB2 sub.rb, dif.mb	*	*	*	*	iov

Table 2–6 (Cont.): NVAX Instruction Set

Opcode	Instruction	N	Z	V	C	Exceptions
Integer, Arithmetic and Logical Instructions						
C2	SUBL2 sub.rl, dif.ml	*	*	*	*	iov
A2	SUBW2 sub.rw, dif.mw	*	*	*	*	iov
83	SUBB3 sub.rb, min.rb, dif.wb	*	*	*	*	iov
C3	SUBL3 sub.rl, min.rl, dif.wl	*	*	*	*	iov
A3	SUBW3 sub.rw, min.rw, dif.ww	*	*	*	*	iov
95	TSTB src.rb	*	*	0	0	
D5	TSTL src.rl	*	*	0	0	
B5	TSTW src.rw	*	*	0	0	
8C	XORB2 mask.rb, dst.mb	*	*	0	–	
CC	XORL2 mask.rl, dst.ml	*	*	0	–	
AC	XORW2 mask.rw, dst.mw	*	*	0	–	
8D	XORB3 mask.rb, src.rb, dst.wb	*	*	0	–	
CD	XORL3 mask.rl, src.rl, dst.wl	*	*	0	–	
AD	XORW3 mask.rw, src.rw, dst.ww	*	*	0	–	
Address Instructions						
9E	MOVAB src.ab, dst.wl	*	*	0	–	
DE	MOVAL{=F} src.al, dst.wl	*	*	0	–	
7E	MOVAQ{=D=G} src.aq, dst.wl	*	*	0	–	
3E	MOVAW src.aw, dst.wl	*	*	0	–	
9F	PUSHAB src.ab, {-(SP).wl}	*	*	0	–	
DF	PUSHAL{=F} src.al, {-(SP).wl}	*	*	0	–	
7F	PUSHAQ{=D=G} src.aq, {-(SP).wl}	*	*	0	–	
3F	PUSHAW src.aw, {-(SP).wl}	*	*	0	–	
Variable-Length Bit Field Instructions						
EC	CMPV pos.rl, size.rb, base.vb, {field.rv}, src.rl	*	*	0	*	rsv
ED	CMPZV pos.rl, size.rb, base.vb, {field.rv}, src.rl	*	*	0	*	rsv

Table 2–6 (Cont.): NVAX Instruction Set

Opcode	Instruction	N	Z	V	C	Exceptions
Variable-Length Bit Field Instructions						
EE	EXTV pos.rl, size.rb, base.vb, {field.rv}, dst.wl	*	*	0	–	rsv
EF	EXTZV pos.rl, size.rb, base.vb, {field.rv}, dst.wl	*	*	0	–	rsv
F0	INSV src.rl, pos.rl, size.rb, base.vb, {field.wv}	–	–	–	–	rsv
EB	FFC startpos.rl, size.rb, base.vb, {field.rv}, find-pos.wl	0	*	0	0	rsv
EA	FFS startpos.rl, size.rb, base.vb, {field.rv}, find-pos.wl	0	*	0	0	rsv
Control Instructions						
9D	ACBB limit.rb, add.rb, index.mb, displ.bw	*	*	*	–	iov
F1	ACBL limit.rl, add.rl, index.ml, displ.bw	*	*	*	–	iov
3D	ACBW limit.rw, add.rw, index.mw, displ.bw	*	*	*	–	iov
F3	AOBLEQ limit.rl, index.ml, displ.bb	*	*	*	–	iov
F2	AOBLSS limit.rl, index.ml, displ.bb	*	*	*	–	iov
1E	BCC{=BGEQU} displ.bb	–	–	–	–	
1F	BCS{=BLSSU} displ.bb	–	–	–	–	
13	BEQL{=BEQLU} displ.bb	–	–	–	–	
18	BGEQ displ.bb	–	–	–	–	
14	BGTR displ.bb	–	–	–	–	
1A	BGTRU displ.bb	–	–	–	–	
15	BLEQ displ.bb	–	–	–	–	
1B	BLEQU displ.bb	–	–	–	–	
19	BLSS displ.bb	–	–	–	–	
12	BNEQ{=BNEQU} displ.bb	–	–	–	–	
1C	BVC displ.bb	–	–	–	–	
1D	BVS displ.bb	–	–	–	–	
E1	BBC pos.rl, base.vb, displ.bb, {field.rv}	–	–	–	–	rsv
E0	BBS pos.rl, base.vb, displ.bb, {field.rv}	–	–	–	–	rsv

Table 2–6 (Cont.): NVAX Instruction Set

Opcode	Instruction	N	Z	V	C	Exceptions
Control Instructions						
E5	BBCC pos.rl, base.vb, displ.bb, {field.mv}	–	–	–	–	rsv
E3	BBCS pos.rl, base.vb, displ.bb, {field.mv}	–	–	–	–	rsv
E4	BBSC pos.rl, base.vb, displ.bb, {field.mv}	–	–	–	–	rsv
E2	BBSS pos.rl, base.vb, displ.bb, {field.mv}	–	–	–	–	rsv
E7	BBCCI pos.rl, base.vb, displ.bb, {field.mv}	–	–	–	–	rsv
E6	BBSSI pos.rl, base.vb, displ.bb, {field.mv}	–	–	–	–	rsv
E9	BLBC src.rl, displ.bb	–	–	–	–	
E8	BLBS src.rl, displ.bb	–	–	–	–	
11	BRB displ.bb	–	–	–	–	
31	BRW displ.bw	–	–	–	–	
10	BSBB displ.bb, {-(SP).wl}	–	–	–	–	
30	BSBW displ.bw, {-(SP).wl}	–	–	–	–	
8F	CASEB selector.rb, base.rb, limit.rb, displ.bw-list	*	*	0	*	
CF	CASEL selector.rl, base.rl, limit.rl, displ.bw-list	*	*	0	*	
AF	CASEW selector.rw, base.rw, limit.rw, displ.bw-list	*	*	0	*	
17	JMP dst.ab	–	–	–	–	
16	JSB dst.ab, {-(SP).wl}	–	–	–	–	
05	RSB {(SP)+.rl}	–	–	–	–	
F4	SOBGEQ index.ml, displ.bb	*	*	*	–	iov
F5	SOBGTR index.ml, displ.bb	*	*	*	–	iov

Table 2–6 (Cont.): NVAX Instruction Set

Opcode	Instruction	N	Z	V	C	Exceptions
Procedure Call Instructions						
FA	CALLG arglist.ab, dst.ab, {-(SP).w*}	0	0	0	0	rsv
FB	CALLS numarg.rl, dst.ab, {-(SP).w*}	0	0	0	0	rsv
04	RET {(SP)+.r*}	*	*	*	*	rsv
Miscellaneous Instructions						
B9	BICPSW mask.rw	*	*	*	*	rsv
B8	BISPSW mask.rw	*	*	*	*	rsv
03	BPT {-(KSP).w*}	0	0	0	0	
00	HALT {-(KSP).w*}	–	–	–	–	prv
0A	INDEX subscript.rl, low.rl, high.rl, size.rl, indexin.rl, indexout.wl	*	*	0	0	sub
DC	MOVPSL dst.wl	–	–	–	–	
01	NOP	–	–	–	–	
BA	POPR mask.rw, {(SP)+.r*}	–	–	–	–	
BB	PUSHR mask.rw, {-(SP).w*}	–	–	–	–	
FC	XFC {unspecified operands}	0	0	0	0	
Queue Instructions						
5C	INSQHI entry.ab, header.aq	0	*	0	*	rsv
5D	INSQTI entry.ab, header.aq	0	*	0	*	rsv
0E	INSQUE entry.ab, pred.ab	*	*	0	*	
5E	REMQHI header.aq, addr.wl	0	*	*	*	rsv
5F	REMQTI header.aq, addr.wl	0	*	*	*	rsv
0F	REMQUE entry.ab, addr.wl	*	*	*	*	

Table 2–6 (Cont.): NVAX Instruction Set

Opcode	Instruction	N	Z	V	C	Exceptions
Operating System Support Instructions						
BD	CHME param.rw, {-(ySP).w*}	0	0	0	0	
BC	CHMK param.rw, {-(ySP).w*}	0	0	0	0	
BE	CHMS param.rw, {-(ySP).w*}	0	0	0	0	
BF	CHMU param.rw, {-(ySP).w*}	0	0	0	0	
06	LDPCTX {PCB.r*, -(KSP).w*}	–	–	–	–	rsv, prv
DB	MFPR procreg.rl, dst.wl	*	*	0	–	rsv, prv
DA	MTPR src.rl, procreg.rl	*	*	0	–	rsv, prv
0C	PROBER mode.rb, len.rw, base.ab	0	*	0	–	
0D	PROBEW mode.rb, len.rw, base.ab	0	*	0	–	
02	REI {(SP)+.r*}	*	*	*	*	rsv
07	SVPCTX {(SP)+.r*, PCB.w*}	–	–	–	–	prv
Character String Instructions						
29	CMPC3 len.rw, src1addr.ab, src2addr.ab	*	*	0	*	
2D	CMPC5 src1len.rw, src1addr.ab, fill.rb,src2len.rw, src2addr.ab	*	*	0	*	
3A	LOCC char.rb, len.rw, addr.ab	0	*	0	0	
28	MOVC3 len.rw, srcaddr.ab, dstaddr.ab, {R0-5.wl}	0	1	0	0	
2C	MOVC5 srclen.rw, srcaddr.ab, fill.rb, dstlen.rw, dstaddr.ab,{R0-5.wl}	*	*	0	*	
2A	SCANC len.rw, addr.ab, tbladdr.ab, mask.rb	0	*	0	0	
3B	SKPC char.rb, len.rw, addr.ab	0	*	0	0	
2B	SPANC len.rw, addr.ab, tbladdr.ab, mask.rb	0	*	0	0	

Table 2–6 (Cont.): NVAX Instruction Set

Opcode	Instruction	N	Z	V	C	Exceptions
Floating Point Instructions						
60	ADDD2 add.rd, sum.md	*	*	0	0	rsv, fov, fuv
40	ADDF2 add.rf, sum.mf	*	*	0	0	rsv, fov, fuv
40FD	ADDG2 add.rg, sum.mg	*	*	0	0	rsv, fov, fuv
61	ADDD3 add1.rd, add2.rd, sum.wd	*	*	0	0	rsv, fov, fuv
41	ADDF3 add1.rf, add2.rf, sum.wf	*	*	0	0	rsv, fov, fuv
41FD	ADDG3 add1.rg, add2.rg, sum.wg	*	*	0	0	rsv, fov, fuv
71	CMPD src1.rd, src2.rd	*	*	0	0	rsv
51	CMPF src1.rf, src2.rf	*	*	0	0	rsv
51FD	CMPG src1.rg, src2.rg	*	*	0	0	rsv
6C	CVTBD src.rb, dst.wd	*	*	0	0	
4C	CVTBF src.rb, dst.wf	*	*	0	0	
4CFD	CVTBG src.rb, dst.wg	*	*	0	0	
68	CVTDB src.rd, dst.wb	*	*	*	0	rsv, iov
76	CVTDF src.rd, dst.wf	*	*	0	0	rsv, fov
6A	CVTDL src.rd, dst.wl	*	*	*	0	rsv, iov
69	CVTDW src.rd, dst.ww	*	*	*	0	rsv, iov
48	CVTFB src.rf, dst.wb	*	*	*	0	rsv, iov
56	CVTFD src.rf, dst.wd	*	*	0	0	rsv
99FD	CVTFG src.rf, dst.wg	*	*	0	0	rsv
4A	CVTFL src.rf, dst.wl	*	*	*	0	rsv, iov
49	CVTFW src.rf, dst.ww	*	*	*	0	rsv, iov
48FD	CVTGB src.rg, dst.wb	*	*	*	0	rsv, iov
33FD	CVTGF src.rg, dst.wf	*	*	0	0	rsv, fov, fuv
4AFD	CVTGL src.rg, dst.wl	*	*	*	0	rsv, iov
49FD	CVTGW src.rg, dst.ww	*	*	*	0	rsv, iov
6E	CVTLD src.rl, dst.wd	*	*	0	0	
4E	CVTLF src.rl, dst.wf	*	*	0	0	
4EFD	CVTLG src.rl, dst.wg	*	*	0	0	
6D	CVTWD src.rw, dst.wd	*	*	0	0	
4D	CVTWF src.rw, dst.wf	*	*	0	0	
4DFD	CVTWG src.rw, dst.wg	*	*	0	0	

Table 2–6 (Cont.): NVAX Instruction Set

Opcode	Instruction	N	Z	V	C	Exceptions
Floating Point Instructions						
6B	CVTRDL src.rd, dst.wl	*	*	*	0	rsv, iov
4B	CVTRFL src.rf, dst.wl	*	*	*	0	rsv, iov
4BFD	CVTRGL src.rg, dst.wl	*	*	*	0	rsv, iov
66	DIVD2 divr.rd, quo.md	*	*	0	0	rsv, fov, fuv, fdvz
46	DIVF2 divr.rf, quo.mf	*	*	0	0	rsv, fov, fuv, fdvz
46FD	DIVG2 divr.rg, quo.mg	*	*	0	0	rsv, fov, fuv, fdvz
67	DIVD3 divr.rd, divd.rd, quo.wd	*	*	0	0	rsv, fov, fuv, fdvz
47	DIVF3 divr.rf, divd.rf, quo.wf	*	*	0	0	rsv, fov, fuv, fdvz
47FD	DIVG3 divr.rg, divd.rg, quo.wg	*	*	0	0	rsv, fov, fuv, fdvz
72	MNEGD src.rd, dst.wd	*	*	0	0	rsv
52	MNEGF src.rf, dst.wf	*	*	0	0	rsv
52FD	MNEGG src.rg, dst.wg	*	*	0	0	rsv
70	MOVD src.rd, dst.wd	*	*	0	-	rsv
50	MOVF src.rf, dst.wf	*	*	0	-	rsv
50FD	MOVG src.rg, dst.wg	*	*	0	-	rsv
64	MULD2 mulr.rd, prod.md	*	*	0	0	rsv, fov, fuv
44	MULF2 mulr.rf, prod.mf	*	*	0	0	rsv, fov, fuv
44FD	MULG2 mulr.rg, prod.mg	*	*	0	0	rsv, fov, fuv
65	MULD3 mulr.rd, muld.rd, prod.wd	*	*	0	0	rsv, fov, fuv
45	MULF3 mulr.rf, muld.rf, prod.wf	*	*	0	0	rsv, fov, fuv
45FD	MULG3 mulr.rg, muld.rg, prod.wg	*	*	0	0	rsv, fov, fuv
62	SUBD2 sub.rd, dif.md	*	*	0	0	rsv, fov, fuv
42	SUBF2 sub.rf, dif.mf	*	*	0	0	rsv, fov, fuv
42FD	SUBG2 sub.rg, dif.mg	*	*	0	0	rsv, fov, fuv

Table 2–6 (Cont.): NVAX Instruction Set

Opcode	Instruction	N	Z	V	C	Exceptions
Floating Point Instructions						
63	SUBD3 sub.rd, min.rd, dif.wd	*	*	0	0	rsv, fov, fuv
43	SUBF3 sub.rf, min.rf, dif.wf	*	*	0	0	rsv, fov, fuv
43FD	SUBG3 sub.rg, min.rg, dif.wg	*	*	0	0	rsv, fov, fuv
73	TSTD src.rd	*	*	0	0	rsv
53	TSTF src.rf	*	*	0	0	rsv
53FD	TSTG src.rg	*	*	0	0	rsv
Microcode-Assisted Emulated Instructions						
20	ADDP4 addlen.rw, addaddr.ab, sumlen.rw, sumaddr.ab	*	*	*	0	rsv, dov
21	ADDP6 add1len.rw, add1addr.ab, add2len.rw, add2addr.ab, sumlen.rw, sumaddr.ab	*	*	*	0	rsv, dov
F8	ASHP cnt.rb, srclen.rw, srcaddr.ab, round.rb, dstlen.rw, dstaddr.ab	*	*	*	0	rsv, dov
35	CMPP3 len.rw, src1addr.ab, src2addr.ab	*	*	0	0	
37	CMPP4 src1len.rw, src1addr.ab, src2len.rw, src2addr.ab	*	*	0	0	
0B	CRC tbl.ab, inicrc.rl, strlen.rw, stream.ab	*	*	0	0	
F9	CVTLP src.rl, dstlen.rw, dstaddr.ab	*	*	*	0	rsv, dov
36	CVTPL srclen.rw, srcaddr.ab, dst.wl	*	*	*	0	rsv, iov
08	CVTPS srclen.rw, srcaddr.ab, dstlen.rw, dstaddr.ab	*	*	*	0	rsv, dov
09	CVTSP srclen.rw, srcaddr.ab, dstlen.rw, dstaddr.ab	*	*	*	0	rsv, dov
24	CVTPT srclen.rw, srcaddr.ab, tbladdr.ab, dstlen.rw, dstaddr.ab	*	*	*	0	rsv, dov
26	CVTTP srclen.rw, srcaddr.ab, tbladdr.ab, dstlen.rw, dstaddr.ab	*	*	*	0	rsv, dov
27	DIVP divrlen.rw, divraddr.ab, divdlen.rw, divdaddr.ab, quolen.rw, quoaddr.ab	*	*	*	0	rsv, dov, ddvz

Table 2–6 (Cont.): NVAX Instruction Set

Opcode	Instruction	N	Z	V	C	Exceptions
Microcode-Assisted Emulated Instructions						
38	EDITPC srclen.rw, srcaddr.ab, pattern.ab, dstaddr.ab	*	*	*	*	rsv, dov
39	MATCHC objlen.rw, objaddr.ab, srclen.rw, srcaddr.ab	0	*	0	0	
34	MOVP len.rw, srcaddr.ab, dstaddr.ab	*	*	0	0	
2E	MOVTC srclen.rw, srcaddr.ab, fill.rb, tbladdr.ab, dstlen.rw, dstaddr.ab	*	*	0	*	
2F	MOVTUC srclen.rw, srcaddr.ab, esc.rb, tbladdr.ab, dstlen.rw, dstaddr.ab	*	*	*	*	
25	MULP mulrlen.rw, mulraddr.ab, muldlen.rw, muldaddr.ab, prodlen.rw, prodaddr.ab	*	*	*	0	rsv, dov
22	SUBP4 sublen.rw, subaddr.ab, diflen.rw, difaddr.ab	*	*	*	0	rsv, dov
23	SUBP6 sublen.rw, subaddr.ab, minlen.rw, minaddr.ab, diflen.rw, difaddr.ab	*	*	*	0	rsv, dov

Table 2–6 (Cont.): NVAX Instruction Set

The notation used for operand specifiers is <name>.<access type><data type>. Implied operands (those locations that are referenced by the instruction but not specified by an operand) are denoted by curly braces {}.

Access Type

a = address operand
b = branch displacement
m = modified operand (both read and written)
r = read only operand
v = if not "Rn", same as a, otherwise R[n+1]'R[n]
w = write only operand

Data Type

b = byte
d = D_floating
f = F_floating
g = G_floating
l = longword
q = quadword
v = field (used only in implied operands)
w = word
* = multiple longwords (used only in implied operands)

Condition Codes Modification

* = conditionally set/cleared
– = not affected
0 = cleared
1 = set

Exceptions

rsv = reserved operand fault
iov = integer overflow trap
idvz = integer divide by zero trap
fov = floating overflow fault
fuv = floating underflow fault
fdvz = floating divide by zero fault
dov = decimal overflow trap
ddvz = decimal divide by zero trap
sub = subscript range trap
prv = privileged instruction fault
vec = vector unit disabled fault

2.6 Memory Management

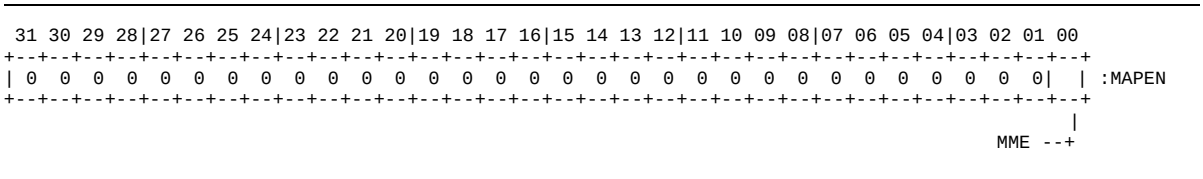
The NVAX Plus CPU Chip supports a four gigabyte (2**32) virtual address space, divided into two sections, system space and process space. Process space is further subdivided into the P0 region and the P1 region.

2.6.1 Memory Management Control Registers

Memory management is controlled by three processor registers: Memory Management Enable (MAPEN), Translation Buffer Invalidate Single (TBIS), and Translation Buffer Invalidate All (TBIA).

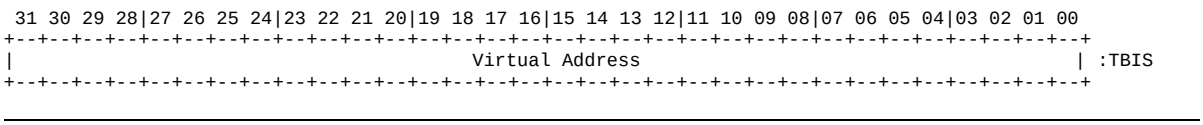
Bit <0> of the MAPEN register enables memory management if written with a 1 and disables memory management if written with a 0. The MAPEN register is shown in Figure 2–11.

Figure 2–11: MAPEN Register



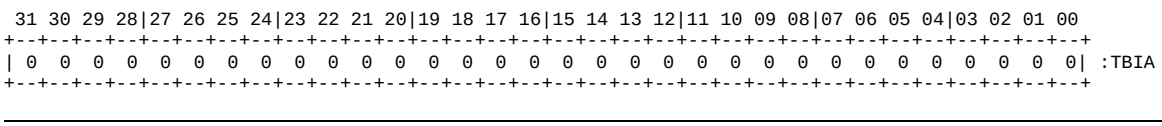
The TBIS register controls translation buffer invalidation. Writing a virtual address into TBIS invalidates any entry which maps that virtual address. The TBIS format is shown in Figure 2–12.

Figure 2–12: TBIS Register



The TBIA register also controls translation buffer invalidation. Writing a zero into TBIA invalidates the entire translation buffer. The TBIA format is shown in Figure 2–13.

Figure 2–13: TBIA Register



2.6.2 System Space Address Translation

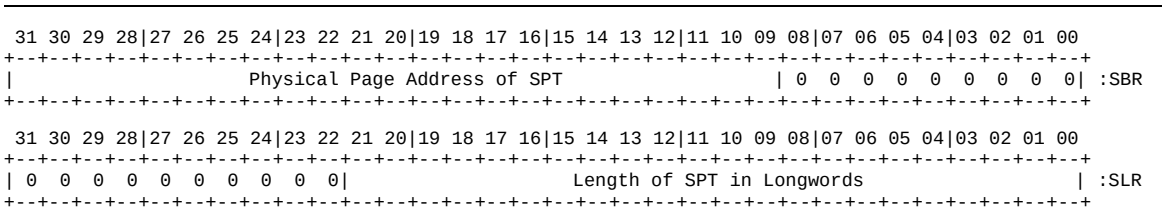
A virtual address with bits <31:30> = 2 is an address in the system virtual address space.

System virtual address space is mapped by the System Page Table (SPT), which is defined by the System Base Register (SBR) and the System Length Register (SLR). The SBR contains the page-aligned physical address of the System Page Table. The SLR contains the size of the SPT in longwords, that is, the number of Page Table Entries. The Page Table Entry addressed by the System Base Register maps the first page of system virtual address space, that is, virtual byte address 80000000 (hex). These registers are shown in Figure 2–14.

NOTE

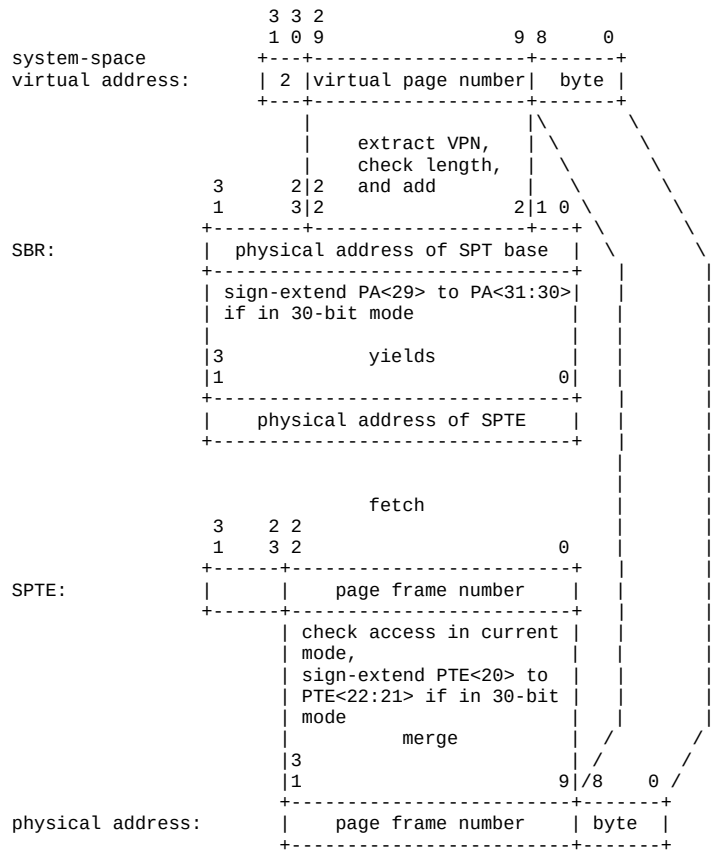
When the CPU is configured to generate 30-bit physical addresses, SBR<31:30> are ignored.

Figure 2–14: System Base and Length Registers



The system space translation algorithm is shown graphically in Figure 2–15.

Figure 2–15: System Space Translation Algorithm



2.6.3 Process Space Address Translation

A virtual address with bit <31> = 0 is an address in the process virtual address space. Process space is divided into two equal sized, separately mapped regions. If virtual address bit <30> = 0, the address is in region P0. If virtual address bit <30> = 1, the address is in region P1.

2.6.3.1 P0 Region Address Translation

The P0 region of the address space is mapped by the P0 Page Table (P0PT), which is defined by the P0 Base Register (P0BR) and the P0 Length Register (P0LR). The P0BR contains the system page-aligned virtual address of the P0 Page Table. The P0LR contains the size of the P0PT in longwords, that is, the number of Page Table Entries. The Page Table Entry addressed by the P0 Base Register maps the first page of the P0 region of the virtual address space, that is, virtual byte address 0. The P0 base and length registers are shown in Figure 2–16.

The P0 space translation algorithm is shown graphically in Figure 2–17.

Figure 2–16: P0 Base and Length Registers

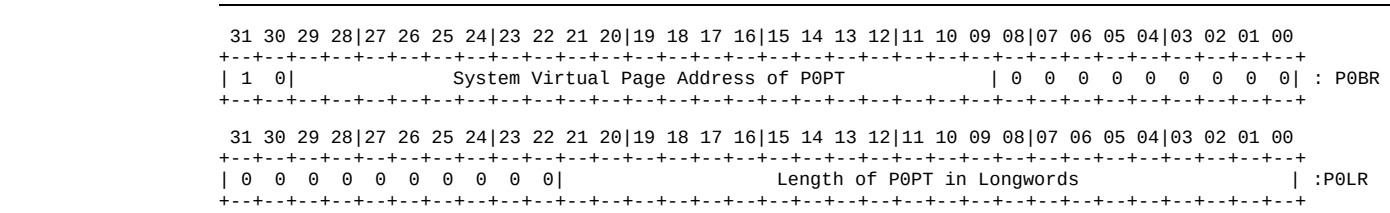
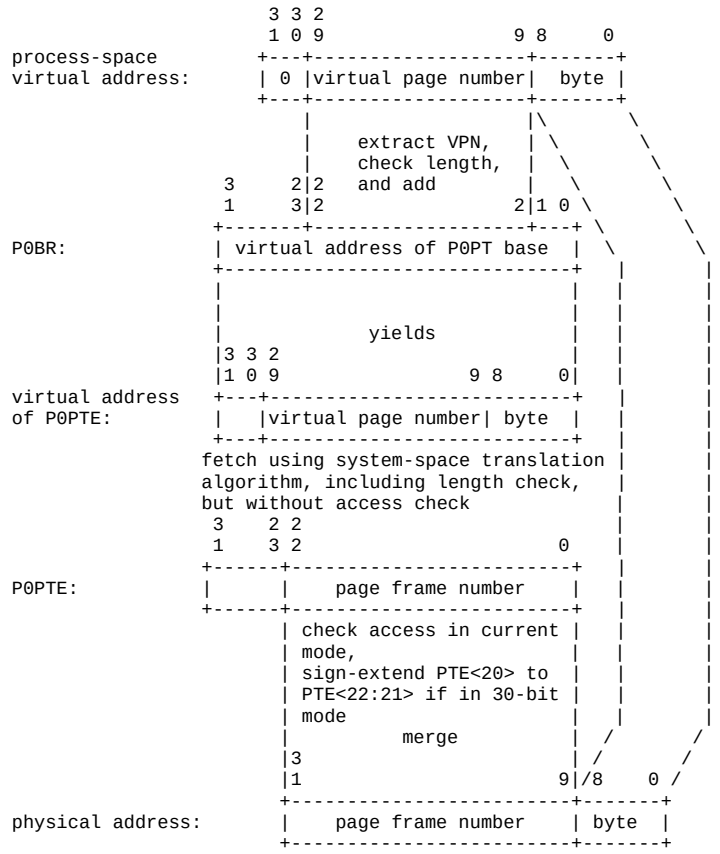


Figure 2–17: P0 Space Translation Algorithm



2.6.3.2 P1 Region Address Translation

The P1 region of the address space is mapped by the P1 Page Table (P1PT), which is defined by the P1 Base Register (P1BR) and the P1 Length Register (P1LR). Because P1 space grows towards smaller addresses, and because a consistent hardware interpretation of the base and length registers is desirable, P1BR and P1LR describe the portion of P1 space that is NOT

accessible. Note that P1LR contains the number of nonexistent PTEs. P1BR contains the page-aligned virtual address of what would be the PTE for the first page of P1, that is, virtual byte address 40000000 (hex). The address in P1BR is not necessarily an address in system space,but all the addresses of PTEs must be in system space.

The P1 space translation algorithm is shown graphically in Figure 2–19.

Figure 2–18: P1 Base and Length Registers

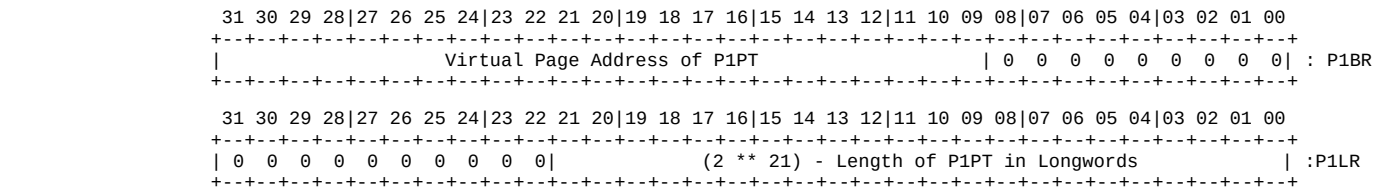
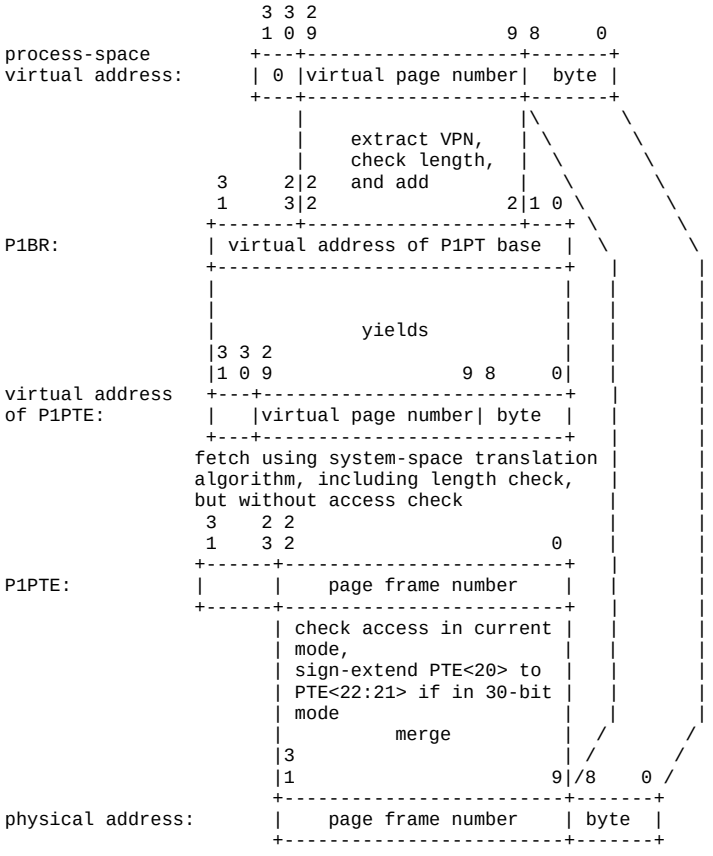


Figure 2–19: P1 Space Translation Algorithm



2.6.4 Page Table Entry

If the CPU is configured to generate 30-bit physical addresses, it interprets PTEs in the 21-bit PFN format shown in Figure 2–20. Conversely, if the CPU is configured to generate 32-bit physical addresses, it interprets PTEs in the 25-bit PFN format shown in Figure 2–21. Note that bits <24:23> of the 25-bit PFN format are ignored by the NVAX Plus CPU chip, which implements only 32-bit physical addresses. The PTE formats shown below are described in DEC Standard 032.

Figure 2–20: PTE Format (21-bit PFN)

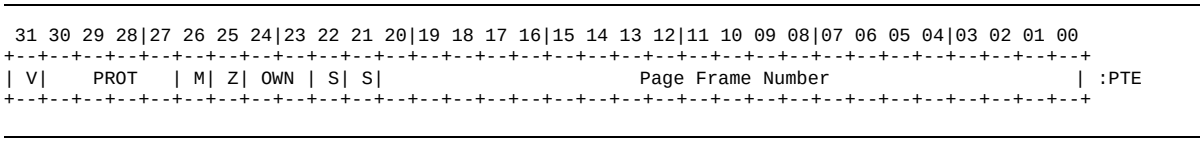


Figure 2–21: PTE Format (25-bit PFN)

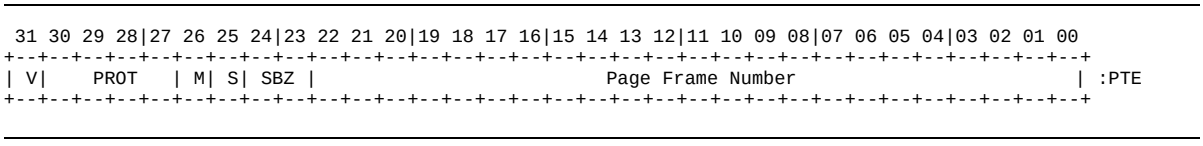


Table 2–7: PTE Protection Code Access Matrix

Code			Current Mode				
Decimal	**Binary**	**Mnemonic**	**K**	**E**	**S**	**U**	**Comment**
0	0000	NA	–	–	–	–	no access
1	0001			unpredictable			reserved
2	0010	KW	RW	–	–	–	
3	0011	KR	R	–	–	–	
4	0100	UW	RW	RW	RW	RW	all access
5	0101	EW	RW	RW	–	–	
6	0110	ERKW	RW	R	–	–	
7	0111	ER	R	R	–	–	
8	1000	SW	RW	RW	RW	–	
9	1001	SREW	RW	RW	R	–	
10	1010	SRKW	RW	R	R	–	
11	1011	SR	R	R	R	–	
12	1100	URSW	RW	RW	RW	R	
13	1101	UREW	RW	RW	R	R	
14	1110	URKW	RW	R	R	R	
15	1111	UR	R	R	R	R	

Access Modes

K = Kernel
E = Executive
S = Supervisor
U = User

Access Types

R = Read
W = Write
– = No access

2.6.5 Translation Buffer

In order to save actual memory references when repeatedly referencing pages, the NVAX Plus CPU Chip uses a translation buffer to remember successful virtual address translations and page status. The translation buffer contains 96 fully associative entries. Both system and process references share these entries.

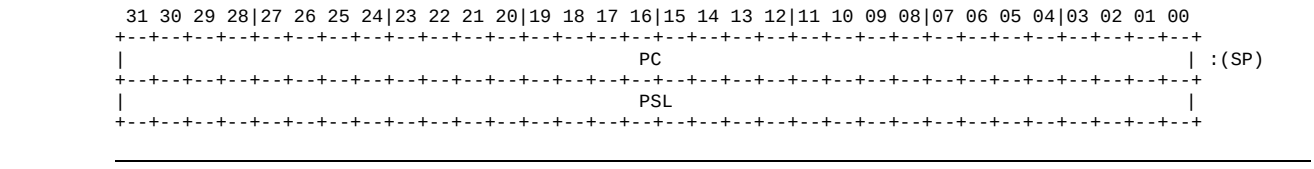
Translation buffer entries are replaced using a not-last-used (NLU) algorithm. This algorithm guarantees that the replacement pointer is not pointing at the last translation buffer entry to be used. This is accomplished by rotating the replacement pointer to the next sequential translation buffer entry if it is pointing to an entry that has just been accessed. Both D-stream and I-stream references can cause the NLU to cycle. When the translation buffer does not contain a reference's virtual address and page status, the machine updates the translation buffer by replacing the entry that is selected by the replacement pointer.

2.7 Exceptions and Interrupts

At certain times during the operation of a system, events within the system require the execution of software routines outside the explicit flow of control of instruction execution. An exception is an event that is relevant primarily to the currently executing process and normally invokes a software routine in the context of the current process. An interrupt is an event which is usually due to some activity outside the current process and invokes a software routine outside the context of the current process.

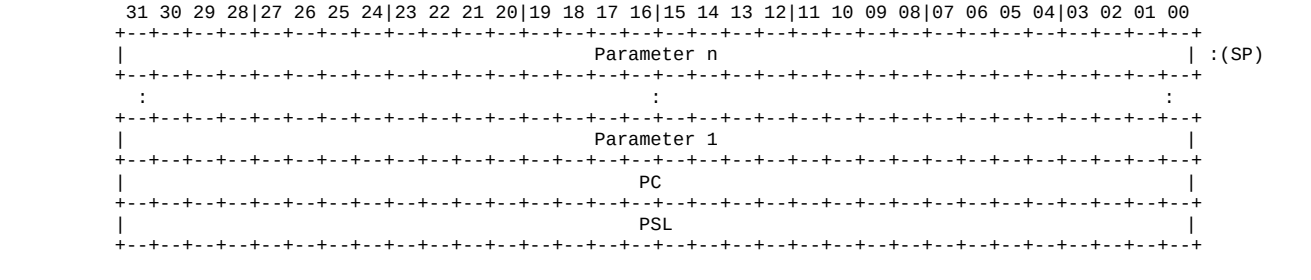
Exceptions and interrupts are reported by constructing a frame on the stack and then dispatching to the service routine through an event-specific vector in the System Control Block (SCB). The minimum stack frame for any interrupt or exception is a PC/PSL pair as shown in Figure 2–22.

Figure 2–22: Minimum Exception Stack Frame



This minimum stack frame is used for all interrupts. Certain exceptions expand the stack frame by pushing additional parameters on the stack above the PC/PSL pair as shown in Figure 2–23.

Figure 2–23: General Exception Stack Frame



What parameters, if any, are pushed on the stack above the PC/PSL pair is a function of the specific exception being reported.

2.7.1 Interrupts

DEC Standard 032 defines 31 interrupt priority levels, a subset of which is implemented by the NVAX Plus CPU. When an interrupt request is generated, the hardware compares the request with the current IPL of the CPU. If the new request is of higher priority an internal request is generated. At the completion of the current instruction (or at selected points during the execution of interruptible instructions), a microcode interrupt handler is invoked to process the request. With hardware assistance, the microcode handler determines the highest priority interrupt, updates

the IPL, pushes a PC/PSL pair on the stack, and dispatches to a macrocode interrupt handler through the appropriate location in the SCB.

Of the 31 interrupt priority levels defined by DEC Standard 032, the NVAX Plus CPU makes use of 23 of them, as shown in Table 2–8.

Table 2–8: Interrupt Priority Levels

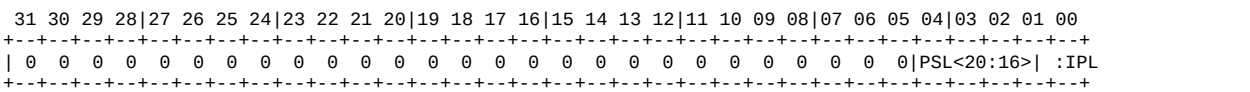
IPL (hex)	IPL (decimal)	Interrupt Condition
1F	31	HALT_H asserted (non maskable)
1E	30	Unused
1D	29	ERR_H asserted (or internal hard error detected)
1C	28	Unused
1B	27	Performance Monitoring Interrupt(internally handled by microcode)
1A	26	Internal soft error detected
18–19	24–25	Unused
17	23	IRQ_H<3> asserted
16	22	IRQ_H<2> or interval timer (IRQ_H<2> takes priority)
15	21	IRQ_H<1> asserted
14	20	IRQ_H<0> asserted
10–13	16-19	Unused
01–0F	01-15	Software interrupt asserted

2.7.1.1 Interrupt Control Registers

The interrupt system is controlled by three processor registers: the Interrupt Priority Level Register (IPL), the Software Interrupt Request Register (SIRR), and the Software Interrupt Summary Register (SISR).

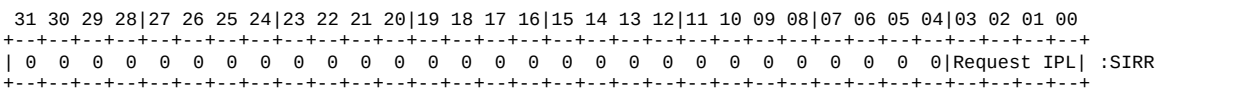
A new interrupt priority level may be loaded into PSL<20:16> by writing the new value to IPL<4:0>. The IPL register is shown in Figure 2–24.

Figure 2–24: Interrupt Priority Level Register



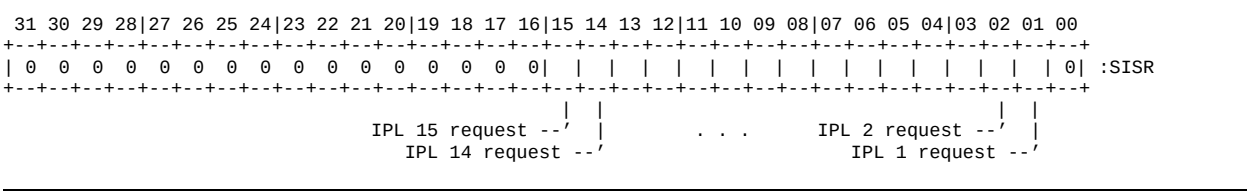
A software interrupt may be requested by writing the desired level to SIRR<3:0>. The SIRR register is shown in Figure 2–25.

Figure 2–25: Software Interrupt Request Registers



The SISR register records pending software interrupt requests at levels 01 through 0F (hex). The SISR register is shown in Figure 2–26.

Figure 2–26: Software Interrupt Summary Register



2.7.2 Exceptions

The VAX architecture recognizes six classes of exceptions. Table 2–9 lists instances of exceptions in each class.

Table 2–9: Exception Classes

Exception Class	Instances
Arithmetic traps/faults	Integer overflow trap Integer divide-by-zero trap Subscript range trap Floating overflow fault Floating divide-by-zero fault Floating underflow fault

Table 2–9 (Cont.): Exception Classes

Exception Class	Instances
Memory management exceptions	Access control violation fault Translation not valid fault M=0 fault
Operand reference exceptions	Reserved addressing mode fault Reserved operand fault or abort
Instruction execution exceptions	Reserved/privileged instruction fault Emulated instruction faults. XFC fault Change-mode trap Breakpoint fault Vector disabled fault
Tracing exceptions	Trace fault
System failure exceptions	Kernel-stack-not-valid abort Interrupt-stack-not-valid halt Console error halt Machine check abort

A trap is an exception that occurs at the end of the instruction that caused the exception. Therefore, the PC saved on the stack is the address of the next instruction that would normally have been executed.

A fault is an exception that occurs during an instruction and that leaves the registers and memory in a consistent state such that elimination of the fault condition and restarting the instruction will give correct results. After the instruction faults, the PC saved on the stack points to the instruction that faulted.

An abort is an exception that occurs during an instruction. An abort leaves the value of registers and memory UNPREDICTABLE such that the instruction cannot necessarily be correctly restarted, completed, simulated, or undone. In most instances, the NVAX Plus microcode attempts to convert an abort into a fault by restoring the state that was present at the start of the instruction which caused the abort.

The following sections describe only those exceptions which are unique to the NVAX Plus CPU, or where DEC Standard 032 is not clear about the implementation.

2.7.2.1 Arithmetic Exceptions

Arithmetic exceptions are detected during the execution of instructions that perform integer or floating point arithmetic manipulations. Whether the exception is reported as a trap or a fault is a function of the specific event. In any case, the exception is reported through SCB vector 34 (hex) with the stack frame shown in Figure 2–27. Table 2–10 lists the exceptions reported by this mechanism.

Figure 2–27: Arithmetic Exception Stack Frame

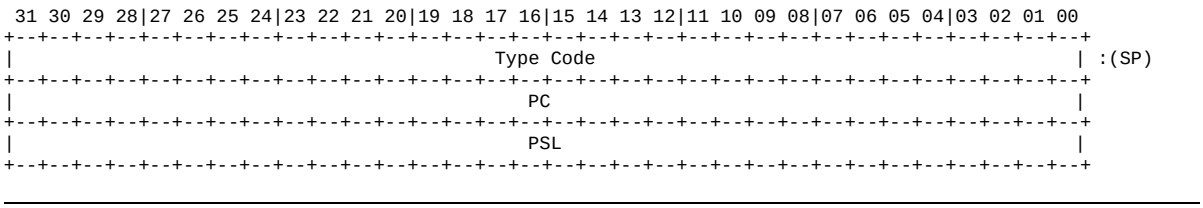


Table 2–10: Arithmetic Exceptions

Type Code			
Decimal	**Hex**	**Type**	**Exception**
1	1	Trap	Integer overflow
2	2	Trap	Integer divide-by-zero
7	7	Trap	Subscript range
8	8	Fault	Floating overflow
9	9	Fault	Floating divide-by-zero
10	A	Fault	Floating underflow

2.7.2.2 Memory Management Exceptions

Memory management exceptions are detected during a memory reference and are always reported as faults. The five memory management exceptions are listed in Table 2–11. All four exceptions push the same frame on the stack, as shown in Figure 2–28. The top longword of the stack frame contains a fault parameter whose bits are described in Table 2–12.

Table 2–11: Memory Management Exceptions

SCB Vector	Exception
20 (hex)	Access control violation
24 (hex)	Translation not valid
3C (hex)	Modify fault

Figure 2–28: Memory Management Exception Stack Frame

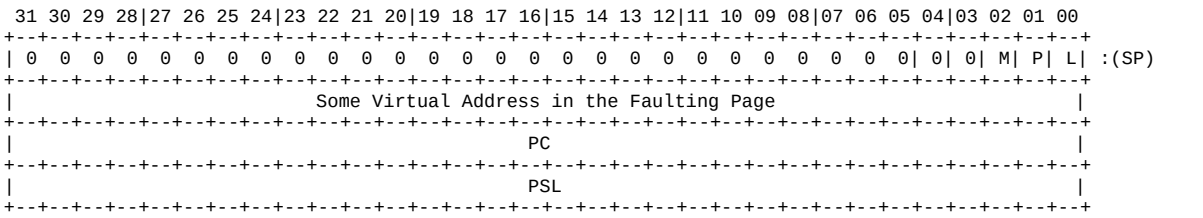


Table 2–12: Memory Management Exception Fault Parameter

Bit	Mnemonic	Meaning
0	L	Length violation
1	P	PTE reference
2	M	Modify or write intent

2.7.2.3 Emulated Instruction Exceptions

The NVAX Plus CPU implements the VAX base instruction group. For certain instructions outside that group, the NVAX Plus microcode provides support for the macrocode emulation of instructions. There are two types of emulation exceptions, depending on whether PSL<FPD> is set at the beginning of the instruction.

If PSL<FPD>=0 at the beginning of the instruction, the exception is reported through SCB vector C8 (hex) as a trap with the stack frame shown in Figure 2–29. The longwords in the stack frame are described in Table 2–13.

Figure 2–29: Instruction Emulation Trap Stack Frame

Figure 2–29 Cont'd. on next page

Figure 2–29 (Cont.): Instruction Emulation Trap Stack Frame

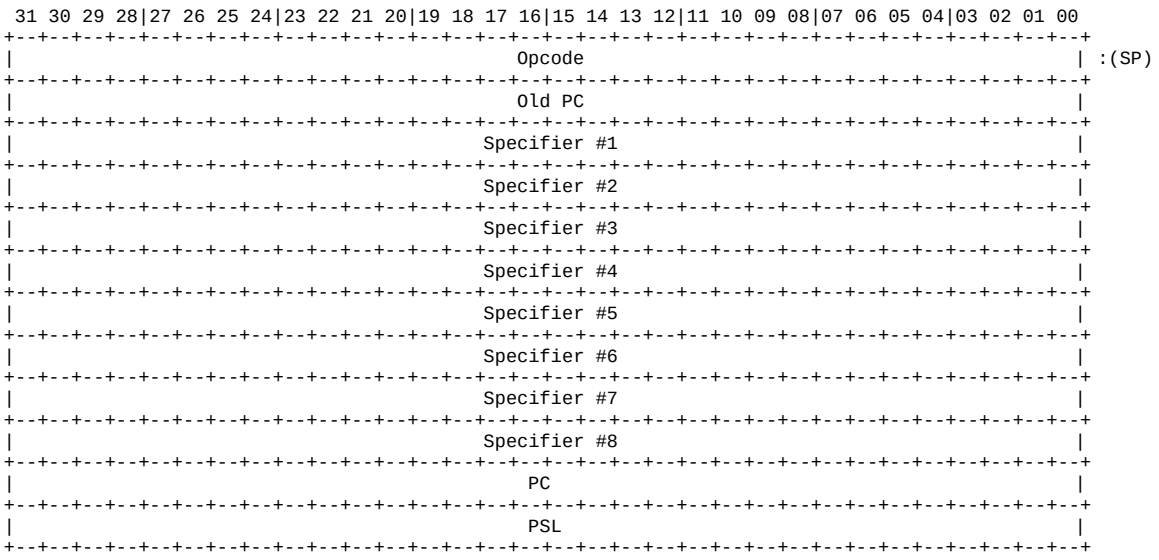
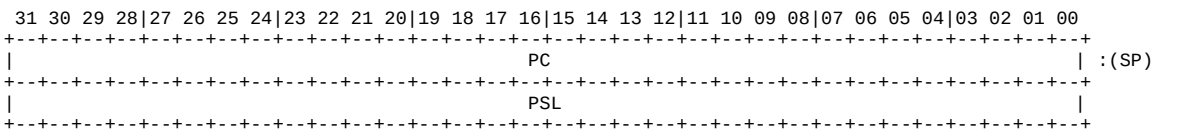


Table 2–13: Instruction Emulation Trap Stack Frame

Location	Use
Opcode	Zero-extended opcode of the emulated instruction
Old PC	PC of the opcode of the emulated instruction
Specifiers	Address of the specified operand for specifiers of access type write (.wx) or address (.ax). Operand value for specifiers of access type read (.rx). For read-type operands whose size is smaller than a longword, the remaining bits are UNPREDICTABLE. For those instructions that don't have 8 specifiers, the remaining specifier longwords contain UNPREDICTABLE values
New PC	PC of the instruction following the emulated instruction
PSL	PSL saved at the time of the trap

If PSL<FPD>=1 at the beginning of the instruction, the exception is reported through SCB vector CC (hex) as a fault with the stack frame shown in Figure 2–30. In this case, PC is that of the opcode of the emulated instruction.

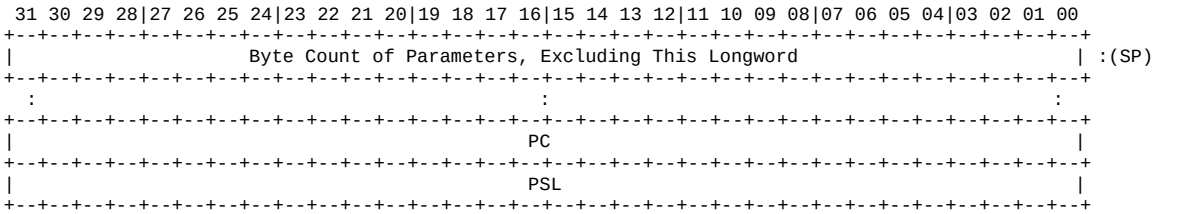
Figure 2–30: Suspended Emulation Fault Stack Frame



2.7.2.4 Machine Check Exceptions

A machine check exception is reported through SCB vector 04 (hex) when the NVAX Plus CPU detects an error condition. The frame pushed on the stack for a machine check indicates the type of error and provides internal state information that may help identify the cause of the error. The generic machine check stack frame is shown in Figure 2–31.

Figure 2–31: Generic Machine Check Stack Frame



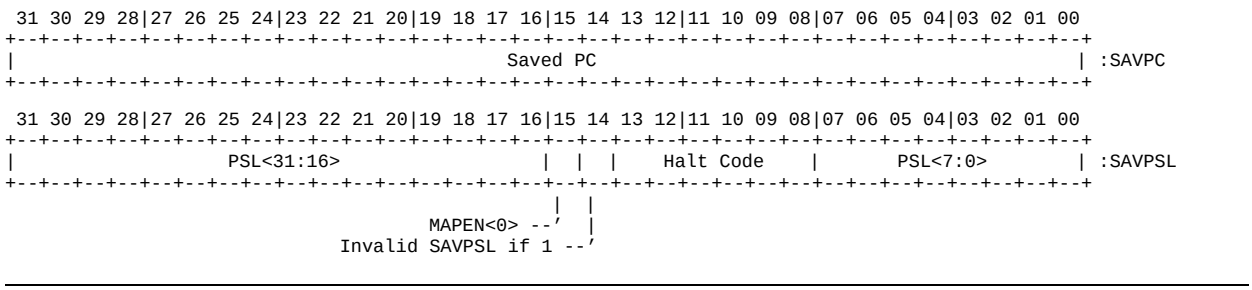
2.7.2.5 Console Halts

In certain microcode flows, the NVAX Plus microcode may detect an inconsistency in internal state, a kernel-mode HALT, or a system reset. In these instances, the microcode initiates a hardware restart sequence which passes control to the console program.

***When a hardware restart sequence is initiated, the NVAX Plus microcode saves the current CPU state, partially initializes the CPU, and passes control to the console program at the physical address contained in the CONSOLE_REG register. ***

During a hardware restart sequence, the stack pointer is saved in the appropriate stack pointer IPR (0 through 4), the current PC is saved in IPR 42 (SAVPC), and the current PSL, halt code, and validity flag are saved in IPR 43 (SAVPSL). The format of SAVPC and SAVPSL are shown in Figure 2–32.

Figure 2–32: Console Saved PC and Saved PSL



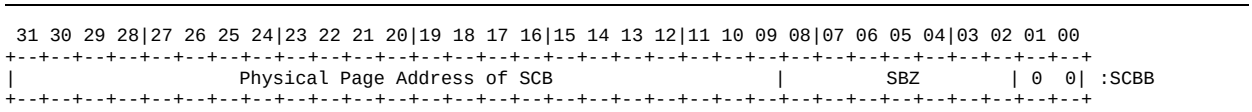
2.8 System Control Block

The System Control Block (SCB) is a page containing the vectors for servicing interrupts and exceptions. The SCB is pointed to by the System Control Block Base Register (SCBB), whose format is shown in Figure 2–33. For best performance, SCBB should contain a page-aligned address. Microcode forces a longword-aligned SCBB by clearing bits <1:0> of the new value before loading the register.

NOTE

When the CPU is configured to generate 30-bit physical addresses, SCBB<31:30> are ignored.

Figure 2–33: System Control Block Base Register



2.8.1 System Control Block Vectors

An SCB vector is an aligned longword in the SCB through which the NVAX Plus microcode dispatches interrupts and exceptions. Each SCB vector has the format shown in Figure 2–34. The fields of the vector are described in Table 2–14.

Figure 2–34: System Control Block Vector

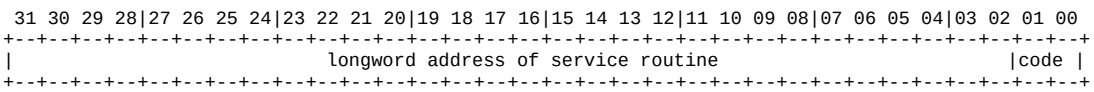


Table 2–14: System Control Block Vector

Bits	Contents
31:2	Virtual address of the service routine for the interrupt or exception. The routine must be longword aligned, as the microcode forces the lower two bits of the address to 00
1:0	Code, interpreted as follows:

Value	Meaning
00	The event is to be serviced on the kernel stack unless the CPU is already on the interrupt stack, in which case the event is serviced on the interrupt stack
01	The event is to be serviced on the interrupt stack. If the event is an exception, the IPL is raised to 1F (hex)
10	Unimplemented, results in a console error halt
11	Unimplemented, results in a console error halt

2.8.2 System Control Block Layout

The System Control Block layout is shown in Table 2–15.

Table 2–15: System Control Block Layout

Vector	Name	Type	Param	Notes
00	unused	–	–	**NVAX passive release**
04	machine check	abort	6	parameters reflect machine state; must be serviced on interrupt stack
08	kernel stack not valid	abort	0	must be serviced on interrupt stack
0C	unused	–	–	**NVAX power fail**
10	reserved/privileged instruction	fault	0	
14	customer reserved instruction	fault	0	XFC instruction
18	reserved operand	fault/abort	0	not always recoverable
1C	reserved addressing mode	fault	0	
20	access control violation/vector alignment fault	fault	2	parameters are virtual address, status code
24	translation not valid	fault	2	parameters are virtual address, status code

Table 2–15 (Cont.): System Control Block Layout

Vector	Name	Type	Param	Notes
28	trace pending	fault	0	
2C	breakpoint instruction	fault	0	
30	unused	–	–	compatibility mode in other VAXes
34	arithmetic trap/fault	trap/fault	1	parameter is type code
38–3C	unused	–	–	–
40	CHMK	trap	1	parameter is sign-extended operand word
44	CHME	trap	1	parameter is sign-extended operand word
48	CHMS	trap	1	parameter is sign-extended operand word
4C	CHMU	trap	1	parameter is sign-extended operand word
50	unused	–	–	–
54	soft error notification	interrupt	0	IPL is 1A (hex)
58	Performance monitoring counter overflow	interrupt	–	See Chapter 18 for details
59–5C	unused	–	–	–
60	hard error notification	interrupt	0	IPL is 1D (hex)
64	unused	–	–	–
68	vector unit disabled	fault	0	vector instructions
6C–80	unused	–	–	**80 was NVAX interprocessor interrupt**
84	software level 1	interrupt	0	
88	software level 2	interrupt	0	ordinarily used for AST delivery
8C	software level 3	interrupt	0	ordinarily used for process scheduling
90–BC	software levels 4–15	interrupt	0	
C0	interval timer	interrupt	0	IPL is 16 (hex)
C4	unused	–	–	–
C8	emulation start	fault	10	same mode exception, FPD=0; parameters are opcode, PC, specifiers
CC	emulation continue	fault	0	same mode exception, FPD=1; no parameters
D0	device vector	interrupt	0	IPL is 14 (hex)
D4	device vector	interrupt	0	IPL is 15 (hex), includes console interrupts

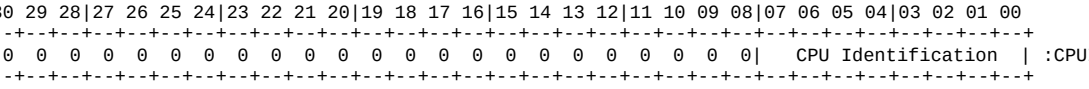
Table 2–15 (Cont.): System Control Block Layout

Vector	Name	Type	Param	Notes
D8	device vector	interrupt	0	IPL is 16 (hex), includes interprocessor interrupts
DC	device vector	interrupt	0	IPL is 17 (hex)
E0–F4	unused	–	–	–
F8–FC	unused	–	–	**F8 was NVAX console receiver-FC was console transmitter -IPL 15**
100–FFFC	unused	–	–	**was NVAX Device interrupt vectors**

2.9 CPU Identification

Software may quickly determine on which CPU it is executing in a multi-processor system by reading the CPUID processor register. The format of this register is shown in Figure 2–35.

Figure 2–35: CPU ID Register



The CPUID processor register is implemented internally as an 8-bit read-write register. The source of the CPU ID information is system-specific, and it is the responsibility of the console firmware at powerup to determine the CPU ID from the system-specific source, and write the CPU ID register to the correct value.

2.10 SYSTEM IDENTIFICATION

The System Identification Register, IPR 62 (SID), is a read-only register implemented per DEC Standard 032 in the NVAX Plus CPU. This 32-bit register is used to identify the processor type and its microcode revision level.

The fields in this register are as follows:

Figure 2–36: System Identification (SID)

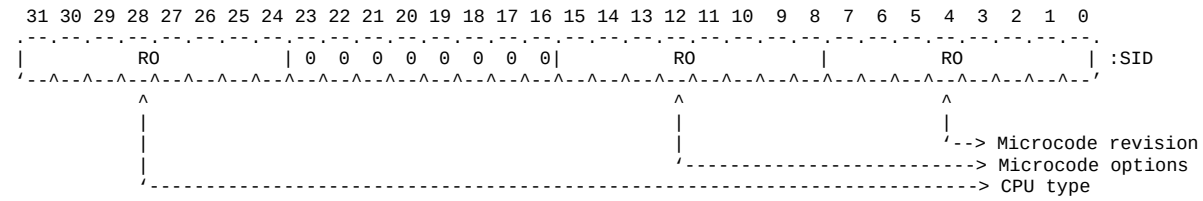
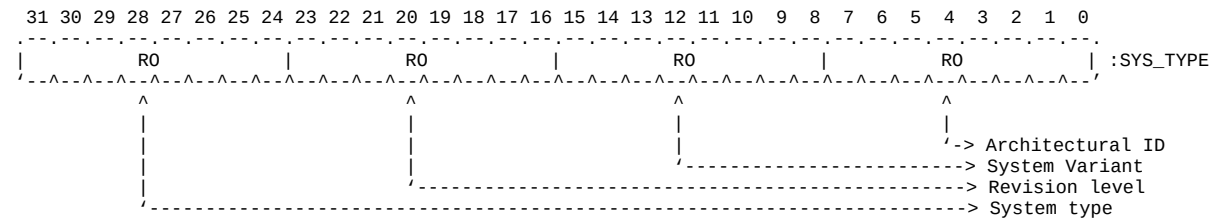


Figure 2–37: System Type (SYS_TYPE)



Microcode revision: This field specifies the microcode revision of the NVAX Plus chip.

Microcode options: This field specifies the microcode options included in this NVAX Plus chip. At present, there are no defined options, so this field is always read as zero.

CPU type: This field specifies the type of CPU used in the system. Because the LNP module uses the NVAX Plus CPU chip, this value is TBD (decimal).

In order to distinguish between different CPU implementations that use the same CPU chip, the LNP, along with all VAX processors which use the NVAX Plus chip, implements a System Type Register (SYS_TYPE). SYS_TYPE resides at the physical address pointed to by the CONSOLE_ REG + 4. This 32-bit read-only register is implemented in the LNP console image. The format of this register is shown in Figure 2–37.

The fields in this register are as follows:

Architectural ID: This field contains licensing bits which distinguish timesharing systems from workstations. Because the LNP module is included in a timesharing system, this field contains 01 (hex).

System Variant: This field distinguishes variants of similar systems. Because this is the first LNP variant, this field contains 01 (hex).

Revision level: This field contains the revision number of the LNP console software. The first LNP console revision will be 01 (hex).

System type: This field indicates the type of system. Because this is a Laser system, this field contains TBD (hex).

SID and SYS_TYPE are accessible only to the CPU on the LNP module. Other devices on the LSB determine the type of node by reading its Laser Device Registers (LDEV).

2.11 Process Structure

A process is a single thread of execution. The context of the current process is contained in the Process Control Block (PCB). The PCB is pointed to by the Process Control Block Base register (PCBB), which is shown in Figure 2–38. The format of the process control block is shown in Figure 2–39. Microcode forces a longword-aligned PCBB by clearing bits <1:0> of the new value before loading the register.

NOTE

When the CPU is configured to generate 30-bit physical addresses, PCBB<31:30> are ignored.

Figure 2–38: Process Control Block Base Register

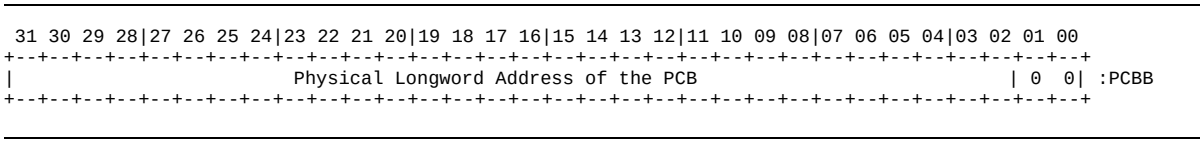
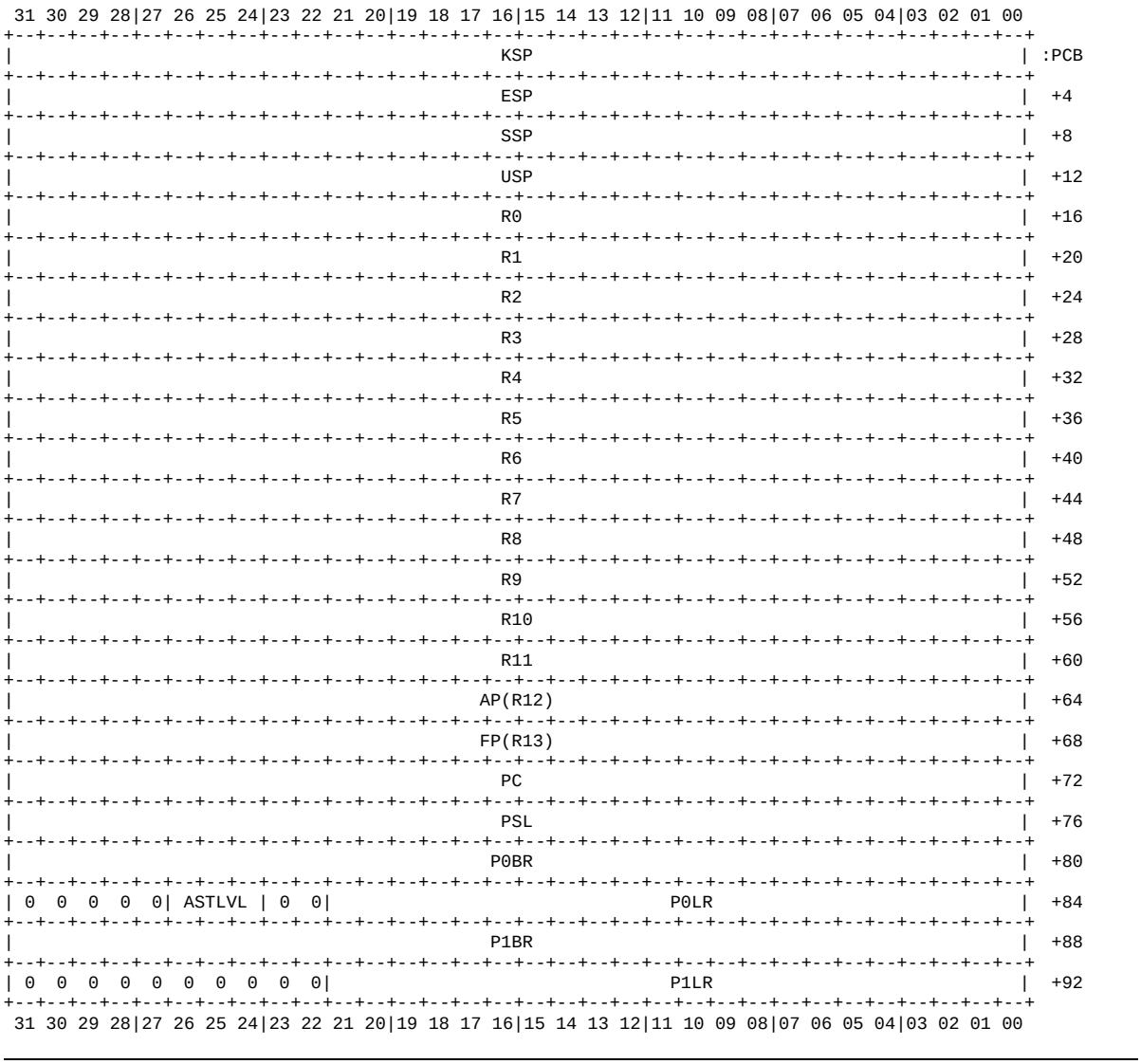


Figure 2–39: Process Control Block



2.12 Mailbox Structure

**For NVAX Plus LASER Bus systems CSRs exist on external I/O busses which are accessed via mailbox structures that exist in main memory. Read requests are posted in mailboxes, and data is returned in memory with status in the following quadword. Mailboxes are allocated and managed by operating system software (successive operations must not overwrite data which is still in use).

The I/O module will service mailbox requests via four mailbox pointer CSRs (LMBPR) located in the I/O modules nodespace. There is one LMBPR for each CPU node. The software sees only one LMBPR address, but the CPU module replaces the least significant two bits of the address (i.e. D<2:1>) with the least significant 2 bits of the node ID (i.e. NIOD<1:0>). If a given LMBPR is in use when it is written to, the I/O module will not acknowledge it, CNF will not be asserted. Processors use the lack of CNF assertion on writes to the LMBPR to indicate a busy status and the write is replayed at a later point in time under software control.

The mailbox pointer CSR has the following format:

Figure 2–40: LMBPR Register

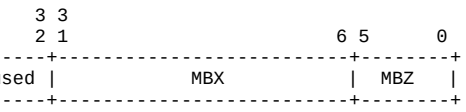


Table 2–16: LMBPR Description

Name	Bit(s)	Type	Description
MBX	26	WO	This field contains the 64-byte-aligned physical address of the mailbox data structure in memory where the I/O module can find information to complete the required operation.

The least significant 6 bits of the mailbox address are always 0, to force 64-byte_alignment. The upper six bits are unused in NVAX Plus systems since NVAX Plus only has a 32 bit wide physical address. The I/O module does however implement these bits. The NVAX Plus chip will allways drive 0's on the upper data lines on I/O space writes such that these bits will be written with 0's.

LMBPR points to a naturally aligned 64 byte data structure in memory that is constructed by software as follows:

Figure 2–41: Mailbox Data Structure

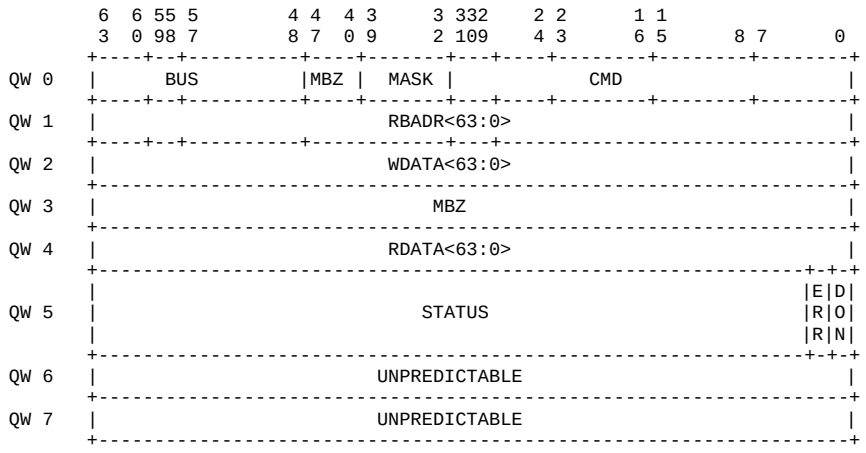


Table 2–17: Mailbox Data Structure Description

Name	Bit(s)	Type	Description
CMD	32	RW	This field contains the command. The I/O module supports read and write commands.
MASK	8	RW	This field contains the byte mask. The I/O module does not use this field.
BUS	24	RW	This field contains the BUS field, which is used to determine which remote bus this command is meant for.
RBADR	64	RW	This field contains the address to be broadcast on the remote bus.
WDATA	64	RW	This field contains the write data to be broadcast on the remote bus.
RDATA	64	RW	This field contains read data returned from the remote bus.
DON	1	RW	This field contains a status bit which is set by the I/O module once a mailbox operation is complete.
ERR	1	RW	This field contains a status bit which indicates that a mailbox operation failed.

For a more complete description of the Laser system mailbox protocol refer to the IOP and LAMB module specifications.

2.12.1 Mailbox Operation

To perform an I/O read or write on one the remote I/O busses software must create a maibox data structure in memory. The command, bus, and address fields must be filled in and the status bits must be cleared. For a write command the write data field must also filled in. At this point the physical address of the maibox data structure must be written to the LMBPR register to initiate the I/O operation. A simple I/O space write, such as with a MOVL, could be used to start the remote I/O operation. However, since writes to LMBPR may be rejected by the I/O module, and no state is preserved across a macro instruction boundry to notify software of this, another method must be used. Microcode implements an IPR register which can used to perform the LMBPR write and return status to software via the condition code bits.

In order for microcode to perform the LMBPR it must know the address of the LMBPR register and the address of the mailbox data structure. Another memory data structure must be created to pass this information to microcode. This structure is called the Mailbox Pointer and consists of 2 longwords.

Figure 2–42: Mailbox Pointer

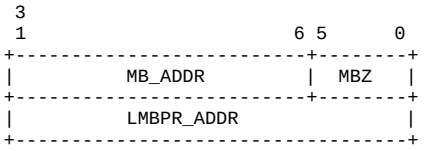


Table 2–18: Mailbox Pointer Description

Name	Bit(s)	Type	Description
MB_ADDR	32	WO	This field contains the physical address of the mailbox data structure. Since the mailbox data structure must be aligned on a 64 byte boundry, bits<5:0> of MB_ADDR must be zero.
LMBPR_ADDR	32	WO	This field contains the virtual address of the LMBPR register.

Once software creates the mailbox data structure and the mailbox pointer structure it may now start the I/O operation. An MTPR to the MAILBOX IPR will initiate the I/O operation. The MAILBOX IPR has the following format:

Figure 2–43: MAILBOX Register

Table 2–19: MAILBOX Register Description

Name	Bit(s)	Type	Description
MBXREG	32	WO	This field contains the address of the mailbox pointer structure.

Microcode will read the MB_ADDR field out of the mailbox pointer structure and then write this value to the LMBPR using the address of the LMBPR provided in the mailbox pointer structure. An EDAL store conditional command is used to perform the write. Microcode will then check a status bit in the CBOX to determine if the write passed or failed. If the write passed, the PSL<Z> bit will be set, otherwise PSL<Z> will be cleared. Software can loop on the MTPR to the MAILBOX Register until the write passes.

After the I/O module has accepted the write to LMBPR it will perform the I/O operation. Software can now poll the status bits in the mailbox data structure until the I/O operation is complete. One the I/O operation is complete the DON bit will be set, if an error occured te ERR bit will also be set. If this was an I/O write operation no further action is needed. If this was an I/O read operation, software can now fetch the returned data from the RDATA field in the mailbox data structure.

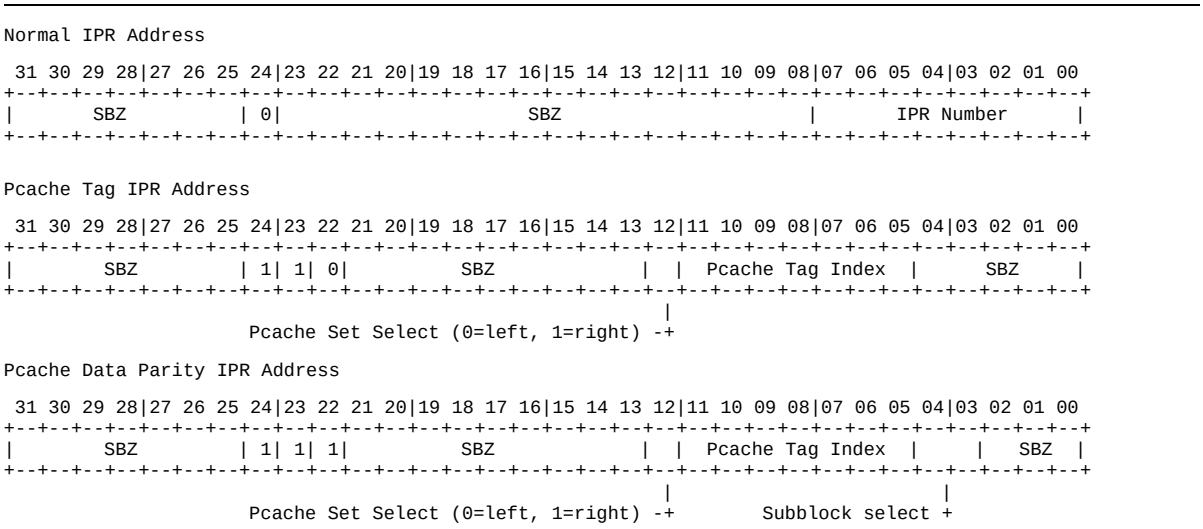
2.13 Processor Registers

The processor registers that are implemented by the NVAX Plus CPU chip are logically divided into three groups, as follows:

- Normal—Those IPRs that address individual registers in the NVAX CPU chip or system environment.
- Pcache tag IPRs—The read-write block of IPRs that allow direct access to the Pcache tags.
- Pcache data parity IPRs—The read-write block of IPRs that allow direct access to the Pcache data parity bits.

Each group of IPRs is distinguished by a particular pattern of bits in the IPR address, as shown in Figure 2–44.

Figure 2–44: IPR Address Space Decoding



The numeric range for each of the four groups is shown in Table 2–20.

Table 2–20: IPR Address Space Decoding

IPR Group	Mnemonic[2]	IPR Address Range (hex)	Contents
Normal		00000000..000000FF[1]	256 individual IPRs.
Pcache Tag	PCTAG	01800000..01801FE0[1]	256 Pcache tag IPRs, 128 for each Pcache set, each separated by 20(hex) from the previous one.
Pcache Data Parity	PCDAP	01C00000..01C01FF8[1]	1024 Pcache data parity IPRs, 512 for each Pcache set, each separated by 8(hex) from the previous one.

[1]Unused fields in the IPR addresses for these groups should be zero. Neither hardware nor microcode detects and faults on an address in which these bits are non-zero. Although non-contiguous address ranges are shown for these groups, the entire IPR address space maps into one of the these groups. If these fields are non-zero, the operation of the CPU is UNDEFINED.

[2]The mnemonic is for the first IPR in the block

NOTE

The address ranges shown above are those used by the programmer. When processing normal IPRs, the microcode shifts the IPR number left by 2 bits for use as an IPR command address. This positions the IPR number to bits <9:2> and modifies the address range as seen by the hardware to 0..3FC, with bits <1:0>=00. No shifting is performed for the other groups of IPR addresses.

Because of the sparse addressing used for IPRs in groups other than the normal group, valid IPR addresses are not separated by one. Rather, valid IPR addresses are separated by either 8 or 20(hex). For example, the IPR address for the first subblock of Pcache data parity is 01C00000 (hex), and the IPR address for the second subblock of Pcache data parity is 01C00008 (hex).

The NVAX Plus chip does not support the Bcache Tag or Bcache Deallocate IPRs. IPR addresses which do not correspond to chip IPRs are NOT converted to I/O space addresses, with IPR reads returning UNPREDICTABLE data, and IPR writes not completed.

The processor registers implemented by the NVAX CPU are are shown in Table 2–21.

NOTE

Many of the processor registers listed in Table 2–21 are used internally by the microcode during normal operation of the CPU, and are not intended to be referenced by software except during test or diagnosis of the system. These registers are flagged with the notation "Testability and diagnostic use only; not for software use in normal operation". References by software to these registers during normal operation can cause UNDEFINED behavior of the CPU.

Table 2–21: Processor Registers

Register Name	Mnemonic	Number (Dec)	Number (Hex)	Type	Cat
Kernel Stack Pointer	KSP	0	0	RW	1-1
Executive Stack Pointer	ESP	1	1	RW	1-1
Supervisor Stack Pointer	SSP	2	2	RW	1-1
User Stack Pointer	USP	3	3	RW	1-1
Interrupt Stack Pointer	ISP	4	4	RW	1-1
Reserved		5	5		
Reserved		6	6		
Reserved		7	7		
P0 Base Register	P0BR	8	8	RW	1-2
P0 Length Register	P0LR	9	9	RW	1-2
P1 Base Register	P1BR	10	A	RW	1-2
P1 Length Register	P1LR	11	B	RW	1-2
System Base Register	SBR	12	C	RW	1-2
System Length Register	SLR	13	D	RW	1-2
CPU Identification[1]	CPUID	14	E	RW	2-1
Reserved		15	F		
Process Control Block Base	PCBB	16	10	RW	1-1
System Control Block Base	SCBB	17	11	RW	1-1
Interrupt Priority Level[1]	IPL	18	12	RW	1-1
AST Level[1]	ASTLVL	19	13	RW	1-1
Software Interrupt Request Register	SIRR	20	14	W	1-1
Software Interrupt Summary Register[1]	SISR	21	15	RW	1-1
Reserved		22	16		
LASER MAILBOX	LMBOX	23	17		
Interval Counter Control/Status[1,2]	ICCS	24	18	RW	1
Next Interval Count	NICR	25	19	W	1
Interval Count	ICR	26	1A	R	1
Time of Year Register	TODR	27	1B	RW	1
Reserved		28	1C		
Reserved		29	1D		
Reserved		30	1E		
Reserved		31	1F		
Reserved		32	20		

[1]Initialized on reset

[2]Subset or full implementation depending on ECR control bit

Table 2–21 (Cont.): Processor Registers

Register Name	Mnemonic	Number (Dec)	Number (Hex)	Type	Cat
Reserved		33	21		
Reserved		34	22		
Reserved		35	23		
Reserved		36	24		
Reserved		37	25		
Machine Check Error Register	MCESR	38	26	W	2-1
Reserved		39	27		
Reserved		40	28		
Reserved		41	29		
Console Saved PC	SAVPC	42	2A	R	2-1
Console Saved PSL	SAVPSL	43	2B	R	2-1
Reserved		44	2C		
Reserved		45	2D		
Reserved		46	2E		
Reserved		47	2F		
Reserved		48	30		
Reserved		49	31		
Reserved		50	32		
Reserved		51	33		
Reserved		52	34		
Reserved		53	35		
Reserved		54	36		
Reserved		55	37		
Memory Management Enable[1]	MAPEN	56	38	RW	1-2
Translation Buffer Invalidate All	TBIA	57	39	W	1-1
Translation Buffer Invalidate Single	TBIS	58	3A	W	1-1
Reserved		59	3B		
Reserved		60	3C		
Performance Monitor Enable[1]	PME	61	3D	RW	2-1
System Identification	SID	62	3E	R	1-1
Translation Buffer Check	TBCHK	63	3F	W	1-1

[1] Initialized on reset

Table 2–21 (Cont.): Processor Registers

Register Name	Mnemonic	Number (Dec)	Number (Hex)	Type	Cat
Reserved		64	40		
Reserved		65	41		
Reserved		66	42		
Reserved		67	43		
Reserved		68	44		
Reserved		69	45		
Reserved		70	46		
Reserved		71	47		
Reserved		72	48		
Reserved		73	49		
Reserved		74	4A		
Reserved		75	4B		
Reserved		76	4C		
Reserved		77	4D		
Reserved		78	4E		
Reserved		79	4F		
Reserved		80	50		
Reserved		81	51		
Reserved		82	52		
Reserved		83	53		
Reserved		84	54		
Reserved		85	55		
Reserved		86	56		
Reserved		87	57		
Reserved		88	58		
Reserved		89	59		
Reserved		90	5A		
Reserved		91	5B		
Reserved		92	5C		
Reserved		93	5D		
Reserved		94	5E		
Reserved		95	5F		

Table 2–21 (Cont.): Processor Registers

Register Name	Mnemonic	Number (Dec)	Number (Hex)	Type	Cat
Reserved		96	60		
Reserved		97	61		
Reserved		98	62		
Reserved		99	63		
Reserved for VM		100	64		
Reserved for VM		101	65		
Reserved for VM		102	66		
Reserved		103	67		
Reserved		104	68		
Reserved		105	69		
Reserved		106	6A		
Reserved		107	6B		
Reserved		108	6C		
Reserved		109	6D		
Reserved		110	6E		
Reserved		111	6F		
Reserved		112	70		
Reserved		113	71		
Reserved		114	72		
Reserved		115	73		
Reserved		116	74		
Reserved		117	75		
Reserved		118	76		
Reserved		119	77		
Reserved for Ebox		120	78		2-4
Reserved for Ebox		121	79		2-4
Interrupt System Status Register[3]	INTSYS	122	7A	RW	2-1
Performance Monitoring Facility Count	PMFCNT	123	7B	RW	2-1
Patchable Control Store Control Register[3]	PCSCR	124	7C	WO	2-1
Ebox Control Register	ECR	125	7D	RW	2-1
Mbox TB Tag Fill[3]	MTBTAG	126	7E	W	2-1
Mbox TB PTE Fill[3]	MTBPTE	127	7F	W	2-1

[3]Testability and diagnostic use only; not for software use in normal operation

Table 2–21 (Cont.): Processor Registers

Register Name	Mnemonic	Number (Dec)	Number (Hex)	Type	Cat
Reserved		128	80		2-4
Reserved		129	81		2-4
Reserved		130	82		2-4
Reserved		131	83		2-4
Reserved		132	84		2-4
Reserved		133	85		2-4
Reserved		134	86		2-4
Reserved		135	87		2-4
Reserved		136	88		2-4
Reserved		137	89		2-4
Reserved		138	8A		2-4
Reserved		139	8B		2-4
Reserved		140	8C		2-4
Reserved		141	8D		2-4
Reserved		142	8E		2-4
Reserved		143	8F		2-4
Reserved		144	90		2-4
Reserved		145	91		2-4
Reserved		146	92		2-4
Reserved		147	93		2-4
Reserved		148	94		2-4
Reserved		149	95		2-4
Reserved		150	96		2-4
Reserved		151	97		2-4
Reserved		152	98		2-4
Reserved		153	99		2-4
Reserved		154	9A		2-4
Reserved		155	9B		2-4
Reserved		156	9C		2-4
Reserved		157	9D		2-4
Reserved		158	9E		2-4
Reserved		159	9F		2-4

Table 2–21 (Cont.): Processor Registers

Register Name	Mnemonic	Number (Dec)	(Hex)	Type	Cat
BIU Control Register	BIU_CTL	160	A0	W	2-3
Reserved for Cbox		161	A1		2-4
Bcache Error Tag	BC_TAG	162	A2	R	2-3
Reserved for Cbox		163	A3		2-4
BIU Status	BIU_STAT	164	A4	W1C	2-3
Reserved for Cbox		165	A5		2-4
BIU Address	BIU_ADDR	166	A6	R	2-3
Reserved for Cbox		167	A7		2-4
FILL Syndrome	FILL_SYN	168	A8	R	2-3
Reserved for Cbox		169	A9		2-4
Fill Address	FILL_ADDR	170	AA	R	2-3
Reserved for Cbox		171	AB		2-4
STxC Pass Fail/CEFSTS	IPR_STR_ COND	172	AC	R	2-3
Reserved for Cbox		173	AD		2-4
Software ECC	BCDECC	172	AE	W	2-3
Reserved for Cbox		175	AF		2-4
Console Halt	CHALT	176	B0	RW	2-3
Reserved for Cbox		177	B1		2-4
Serial I/O	SIO	178	B2	RW	2-3
Reserved for Cbox		179	B3		2-4
Srom_oe/Serial I.E.	SOE-IE	180	B4	W	2-3
Reserved for Cbox		181	B5		2-4
Reserved for Cbox		182	B6		2-4
Reserved for Cbox		183	B7		2-4
Pack to QW	QW_PACK	180	B8	W	2-3
Reserved for Cbox		185	B9		2-4
Reserved for Cbox		186	BA		2-4
Reserved for Cbox		187	BB		2-4
Reserved for Cbox		188	BC		2-4
Reserved for Cbox		189	BD		2-4
Reserved for Cbox		190	BE		2-4
Reserved for Cbox		191	BF		2-4

Table 2–21 (Cont.): Processor Registers

Register Name	Mnemonic	Number (Dec)	Number (Hex)	Type	Cat
Reserved		192	C0		
Reserved		193	C1		
Reserved		194	C2		
Reserved		195	C3		
Reserved		196	C4		
Reserved		197	C5		
Reserved		198	C6		
Reserved		199	C7		
Reserved		200	C8		
Reserved		201	C9		
Reserved		202	CA		
Reserved		203	CB		
Reserved		204	CC		
Reserved		205	CD		
Reserved		206	CE		
Reserved		207	CF		
VIC Memory Address Register	VMAR	208	D0	RW	2-3
VIC Tag Register	VTAG	209	D1	RW	2-3
VIC Data Register	VDATA	210	D2	RW	2-3
Ibox Control and Status Register	ICSR	211	D3	RW	2-3
Ibox Branch Prediction Control Register[3]	BPCR	212	D4	RW	2-3
Reserved for Ibox		213	D5		2-4
Ibox Backup PC[4]	BPC	214	D6	R	2-3
Ibox Backup PC with RLOG Unwind[4]	BPCUNW	215	D7	R	2-3
Reserved for Ibox		216	D8		2-4
Reserved for Ibox		217	D9		2-4
Reserved for Ibox		218	DA		2-4
Reserved for Ibox		219	DB		2-4
Reserved for Ibox		220	DC		2-4
Reserved for Ibox		221	DD		2-4
Reserved for Ibox		222	DE		2-4
Reserved for Ibox		223	DF		2-4

[3]Testability and diagnostic use only; not for software use in normal operation

[4]Chip test use only; not for software use

Table 2–21 (Cont.): Processor Registers

Register Name	Mnemonic	Number (Dec)	(Hex)	Type	Cat
Mbox P0 Base Register[3]	MP0BR	224	E0	RW	2-3
Mbox P0 Length Register[3]	MP0LR	225	E1	RW	2-3
Mbox P1 Base Register[3]	MP1BR	226	E2	RW	2-3
Mbox P1 Length Register[3]	MP1LR	227	E3	RW	2-3
Mbox System Base Register[3]	MSBR	228	E4	RW	2-3
Mbox System Length Register[3]	MSLR	229	E5	RW	2-3
Mbox Memory Management Enable[3]	MMAPEN	230	E6	RW	2-3
Mbox Physical Address Mode	PAMODE	231	E7	RW	2-3
Mbox MME Address	MMEADR	232	E8	R	2-3
Mbox MME PTE Address	MMEPTE	233	E9	R	2-3
Mbox MME Status	MMESTS	234	EA	R	2-3
Reserved for Mbox		235	EB		2-4
Mbox TB Parity Address	TBADR	236	EC	R	2-3
Mbox TB Parity Status	TBSTS	237	ED	RW	2-3
Reserved for Mbox		238	EE		2-4
Reserved for Mbox		239	EF		2-4
Reserved for Mbox		240	F0		2-4
Reserved for Mbox		241	F1		2-4
Mbox Pcache Parity Address	PCADR	242	F2	R	2-3
Reserved for Mbox		243	F3		2-4
Mbox Pcache Status	PCSTS	244	F4	RW	2-3
Reserved for Mbox		245	F5		2-4
Reserved for Mbox		246	F6		2-4
Reserved for Mbox		247	F7		2-4
Mbox Pcache Control	PCCTL	248	F8	RW	2-3
Reserved for Mbox		249	F9		2-4
Reserved for Mbox		250	FA		2-4
Reserved for Mbox		251	FB		2-4
Reserved for Mbox		252	FC		2-4
Reserved for Mbox		253	FD		2-4
Reserved for Mbox		254	FE		2-4
Reserved for Mbox		255	FF		2-4

[3]Testability and diagnostic use only; not for software use in normal operation

Table 2–21 (Cont.): Processor Registers

Register Name	Mnemonic	Number (Dec)	Number (Hex)	Type	Cat
Unimplemented			100–017FFFFF		
See Table 2–20			01800000–FFFFFFFF		2

Type:

R = Read-only register
RW = Read-write register
W = Write-only register
W1C = Write 1 Clear

Cat(egory), *class-subclass*, where:
class is one of:

1 = Implemented as per DEC standard 032
2 = NVAX Plus specific implementation which is unique or different from the DEC standard 032 implementation

subclass is one of:

1 = Processed as appropriate by Ebox microcode
2 = Converted to Mbox IPR number and processed via internal IPR command
3 = Processed by internal IPR command
4 = May be block decoded; reference causes UNDEFINED behavior

2.14 Revision History

Table 2–22: Revision History

Who	When	Description of change
Mike Uhler	06-Mar-1989	Release for external review.
Mike Uhler	15-Dec-1989	Update for second-pass release.
Mike Uhler	20-Jul-1990	Update to reflect implementation.
Mike Callander/Gil Wolrich	15-Nov-1990	NVAX Plus release for external review.

Chapter 3

External Interface

3.1 Overview

NVAX Plus can share system platforms which use EV chips in 128 bit mode. The CPU_CLK runs at a cycle time as fast as 10ns, and SYS_CLK can be set to 2,3,or 4, times the CPU cycle time. NVAX Plus usable in a wide range of systems: workstations, small deskside servers and timesharing machines, and midrange multiprocessor servers and timesharing machines.

3.2 Signals

The following table lists all of the 291 signals on the NVAX_PLUS chip. In the "type" column, an "I" means a pin is an input, an "O" means the pin is an output, and a "B" means the pin is tristate and bidirectional.

Table 3–1: NVAX_PLUS Signals

Signal Name	Count	Type	Function
clkIn_h, _l	2	I	Clock input
testClkIn_h, _l	2	I	Clock input for testing
cpuClkOut_l	1	O	CPU clock output
sysClkOut1_h, _l	2	O	System clock output, delayed
sysClkOut2_h, _l	2	O	System clock output, delayed
osc_test_sel_h	1	I	Select test clock inputs
pp_data_h[6..0]	7	B	Parallel Test Port Data
osc_16m_h	1	I	Interval timer 16MHz oscillator input
dcOk_h	1	I	Power and clocks ok
reset_l	1	I	Reset
sRomOE_l	1	O	Serial ROM output enable
sRomD_h	1	I	Serial ROM data/Rx data

Table 3–1 (Cont.): NVAX_PLUS Signals

Signal Name	Count	Type	Function
sRomClk_h	1	O	Serial ROM clock/Tx data
sRomFast_h	1	I	Serial ROM fast fill
adr_h[33..5]	29	B	Address bus
tagEq_l	1	O	Tag compare output
data_h[127..0]	128	B	Data bus
check_h[27..0]	28	B	Check bit bus
dOE_l	1	I	Data bus output enable
dWSel_h[1..0]	2	I	Data bus write data select
dRAck_h[2..0]	3	I	Data bus read data acknowledge
tagCEOE_h	1	O	tagCtl and tagAdr CE/OE
tagCtlWE_h	1	O	tagCtl WE
tagCtlV_h	1	B	Tag valid
tagCtlS_h	1	B	Tag shared
tagCtlD_h	1	B	Tag dirty
tagCtlP_h	1	B	Tag V/S/D parity
tagAdr_h[33..17]	17	I	Tag address [33..17]
tagAdrP_h	1	I	Tag address parity
tagOk_l	1	I	Tag access from CPU is ok
dataCEOE_h[3..0]	4	O	data CE/OE, longword
dataWE_h[3..0]	4	O	data WE, longword
dataA_h[4]	1	O	data A[4]
dataA_h[3]	1	O	data A[3]
holdReq_h	1	I	Hold request
holdAck_h	1	O	Hold acknowledge
cReq_h[2..0]	3	O	Cycle request
cWMask_h[7..0]	8	O	Cycle write mask
cAck_h[2..0]	3	I	Cycle acknowledge
iAdr_h[12..5]	8	I	Invalidate address
pInvReq_h[1..0]	2	I	Invalidate request, Pcache
pMapWE_h[1..0]	2	O	Backmap WE, Pcache
irq_h[5..0]	6	I	Interrupt requests
vref	1	I	Input reference
tristate_l	1	I	Tristate for testing
cont_l	1	I	Continuity for testing
eclOut_h	1	I	Output mode selection

The following table lists all of the signals that were not on EVAX which are being implemented on the NVAX_PLUS chip. In the "type" column, an "I" means a pin is an input, an "O" means the pin is an output, and a "B" means the pin is tristate and bidirectional.

Table 3–2: New_NVAX_PLUS Signals

Signal Name	Count	Type	Function
osc_test_sel_h	1	I	Select test clock inputs
osc_16m_h	1	I	Interval timer 16MHz oscillator input
pp_data_h[6..0]	7	B	Parallel Test Port Data
pInvReq_h[1..0]	2	I	Invalidate request, Pcache
pMapWE_h[1..0]	2	O	Backmap WE, Pcache

The following table lists all of the signals that were on EVAX which are not being implemented on the NVAX_PLUS chip. In the "type" column, an "I" means a pin is an input, an "O" means the pin is an output, and a "B" means the pin is tristate and bidirectional.

Table 3–3: EVAX Signals

Signal Name	Count	Type	Function
dInvReq_h	1	I	Invalidate request, Dcache
dMapWE_h	1	O	Backmap WE, Dcache
perf_h[3..0]	4	O	Performance monitor outputs
scan_h[3..0]	4	?	Scan

3.2.1 Clocks

External logic supplies NVAX Plus with a differential clock at the desired frequency of the internal phases via the clkIn_h and clkIn_l pins. The NVAX Plus Clock Generator circuit produces the required four single phase clocks, four inverted single phase clocks, and four dual phases clocks required for internal operation.

NVAX Plus divides the input clock by **two** to generate the cpuClkOut_l. The false-to-true transition of cpuClkOut_l is the "CPU clock" used in the timing specification for the tagOk_l signal.

The CPU clock is divided by a programmable value of 4,6,or 8 to generate a system clock, which is supplied to the external interface via the sysClkOut1_h and sysClkOut1_l pins. The system clock is delayed by a programmable number of CPU clock cycles between 0 and 3 to generate a delayed system clock, which is supplied to the external interface via the sysClkOut2_h and sysClkOut2_l pins.

The clock generator runs, generating cpuClkOut_l, and the (correctly timed and positioned) any time an input clock is supplied. In particular, it runs during reset, so that systems can phase-lock the clocks of several chips together before any of them are released from reset.

For a sysClkOut value of 6 times the cpuClkOut, sysClkOut can be programmed as symmetric or asymmetric.

The false-to-true transition of sysClkOut1_h is the "system clock" used as a timing reference throughout this specification.

Almost all transactions on the external interface run synchronously to the CPU clock and phase aligned to the system clock, so the external interface appears to be running synchronously to the system clock (most setup and hold times are referenced to the system clock). The exeception to this are the fast, NVAX Plus controlled tranactions on the external caches and the sample of the tagOk_l input, which are synchronous to the CPU clock, but independent of the system clock.

3.2.2 DC_OK and Reset

NVAX Plus contains a ring oscillator which is switched into service during power up to provide an internal chip clock. The dcOk_h signal switches clock sources between the on-chip ring oscillator and the external clock oscillator. If dcOk_h is false then the on-chip ring oscillator feeds the clock generator, and NVAX Plus is held in reset, independent of the state of the reset_l signal. If dcOk_h is true then the external clock oscillator feeds the clock generator, and vRef input must be valid (so that the real inputs, like reset_l, can be sensed) and NVAX Plus is held in reset by reset_l. The dcOk_h signal is special in that it does not require that vRef be stable to be sensed. It is important to emphasize the importance of driving dcOk_h false until vRef has met its minimum hold time. Because chip testers can apply clocks and power to the chip at the same time, the chip tester can always drive dcOk_h true, but the tester must drive reset_l true for a period longer than the minimum hold time of vRef.

Note if the dcOk_h signal is generated by an RC delay, there is no check that the input clocks are really running. This means that if a board is powered up in manufacturing with a missing, defective, or mis-soldered clock oscillator then NVAX Plus will enter a possibly destructive high-current state. Furthermore, if a clock oscillator fails in stage 1 burn-in then NVAX Plus may also enter this state. The frequency and duration of such events need to be understood by the module designer to decide if this is really a problem.

The reset_l signal forces the CPU into a known state.The reset_l signal is asynchronous, and must be asserted for at least tbd CPU cycles after the assertion of dcOk_h to guarantee that the CPU is reset. This should always be the case, since it also has to be held true for long enough to guarantee that the serial ROM has reset its address counters (which takes about 100ns).

The NVAX Plus CPU chip uses a 3.3V power supply. This 3.3V supply must be stable before any input goes above 4V.

While it is reset, NVAX Plus reads sysClkOut and external bus configuration information off the irq_h pins. External logic should drive the configuration information onto the irq_h pins any time reset_l is true.

The irq_h[2..1] bits encode the value of the divisor used to generate the system clock from the CPU clock.

Table 3–4: System Clock Divisor

irq_h[2]	irq_h[1]	Ratio
F	F	2
F	T	3 symmetric
T	F	3 asymmetric
T	T	4

The irq_h[4..3] bits encode the delay, in CPU clock cycles, from the "system clock" to sysClkOut2.

Table 3–5: System Clock Delay

irq_h[4]	irq_h[3]	Delay
F	F	0
F	T	1
T	F	2
T	T	3

3.2.3 Initialization and Diagnostic Interface

After the reset_l signal is deasserted, but before NVAX Plus executes its first instruction, the Pcache is written with bits out of a serial ROM (such as an AMD Am1736). The serial ROM contains enough VAX code to complete the configuration of the external interface, e.g. setting the timing on the external cache RAMs and diagnose the path between the CPU chip and the real ROM.

Three signals are used to interface to the serial ROM. The sRomOE_l output signal supplies the output enable to the ROM, serving both as an output enable and as a reset (refer to the serial ROM specifications for details). The sRomClk_h output signal supplies the clock to the ROM that causes it to advance to the next bit. The ROM data is read by NVAX Plus via the sRomD_h input signal. The format of the bits in the serial ROM is tbd , however driving sRomD_h false clears the Pcache.

Once the data in the serial ROM has been loaded into the Pcache, sRomD_h can be used for a serial input line, and sRomClk_h can be used as a serial output line.

It is possible to override the loading of the entire Pcache by driving the sRomFast_h signal true when reset is asserted. The Pcache will be loaded with tbd instructions. It is possible that this value will be zero. The intent of this feature is minimize the chip tester time.

3.2.4 Address Bus

The tristate, bidirectional adr_h pins provide a path for addresses to flow between NVAX Plus and the rest of the system. The adr_h pins are connected to the buffers that drive the address pins of the external cache RAMs, and to the transceivers that are located between CPU local address bus and the CPU module address bus.

The address bus is normally driven by NVAX Plus. NVAX Plus stops driving the address bus during reset and during external cache hold. In these states the address bus acts like an input, and the tagEq_l output is the result of an equality compare between adr_h and tagAdr_h. Only bits that are part of the cache tag, as specified by the BC_SIZE field of the BIU_CTL IPR, participate in the compare.

3.2.5 Data Bus

The tristate, bidirectional data_h pins provide a path for data to flow between NVAX Plus and the rest of the system. The data_h pins connect directly to the I/O pins of the external cache data RAMs and to the transceivers that are located between NVAX Plus local data bus and the CPU module data bus.

The tristate, bidirectional check_h pins provide a path for check bits to flow between the CPU and the rest of the system. The check_h pins connect directly to the I/O pins of the external cache data RAMs and to the transceivers that are located between the CPU local check bus and the CPU module check bus.

The data bus is driven by NVAX Plus when it is running a fast write cycle on the external caches, and when some type of write cycle has been presented to the external interface and external logic has enabled the data bus drivers (via dOE_l).

If NVAX Plus is in ECC mode then the check_h pins carry 7 check bits for each longword on the data bus. Bits check_h[6..0] are the check bits for data_h[31..0]. Bits check_h[13..7] are the check bits for data_h[63..32]. Bits check_h[20..14] are the check bits for data_h[95..64]. Bits check_h[17..21] are the check bits for data_h[127..96].

The following ECC code is used. This code is the same one used by the IDT49C460 and AMD29C660 32-bit ECC generator/checker chips.

```
        ddddddddddddddddddddddddddddddd
        33222222222211111111110000000000
        10987654321098765432109876543210
c6  XOR xxxxxxxx               xxxxxxxx
c5  XOR xxxxxxxx        xxxxxxxx
c4  XOR xx      xxxxxx  xx      xxxxxx
c3 XNOR   xxx   xxx   xx  xxx   xxx   xx
c2 XNOR x x  xx x  xx  xx x  xx x  xx  x
c1  XOR    x x x x x xxx   x x x x x xxx
c0  XOR x xx x    x xxx  x  x xxxx x   x
```

By arranging the data and check bits correctly, it is possible to arrange that any number of errors restricted to a 4-bit group can be detected. One such arrangement is as follows:

```
d[00], d[01], d[03], d[25]
d[02], d[04], d[06], c[06]
d[05], d[07], d[12], c[03]
d[08], d[09], d[11], d[14]
d[10], d[13], d[15], d[19]
d[16], d[17], d[22], d[28]
d[18], d[23], d[30], c[05]
d[20], d[27], c[04], c[00]
d[21], d[26], c[02], c[01]
d[24], d[29], d[31]
```

If NVAX Plus is in PARITY mode then 4 of the check_h pins carry EVEN parity for each longword on the data bus, and the rest of the bits are unused. Bit check_h[0] is the parity bit for data_h[31..0]. Bit check_h[7] is the parity bit for data_h[63..32]. Bit check_h[14] is the parity bit for data_h[95..64]. Bit check_h[21] is the parity bit for data_h[127..96].

The ECC bit in the BIU_CTL IPR determines if NVAX Plus is in ECC mode or in PARITY mode.

3.2.6 External Cache Control

The external cache is a direct-mapped, write-back cache. NVAX Plus always views the external cache as having a tag for each 32-byte block (the same as the NVAX Plus Pcache).

The external cache tag RAMs are located between NVAX Plus' local address bus and NVAX Plus' tag inputs. The external cache data RAMs are located between the CPU's local address bus and the CPU's local data bus. NVAX Plus reads the external cache tag RAMs to determine if it can complete a cycle without any module level action, and NVAX Plus reads or writes the external cache data RAMs if, in fact, this is the case.

A cycle requires no module level action if it is a non-LDxL read hit to a valid block, or a non-STxC write hit to a valid but not shared block. All other cycles require module level action. All cycles require module level action if the external cache is disabled (the BC_EN bit in the BIU_CTL IPR is cleared).

All NVAX Plus controlled cycles on the external cache have fixed timing, described in terms of NVAX Plus's internal clock. The actual timing of the cycle is programmable allowing for flexibility in the choice of CPU clock frequencies and cache RAM speeds.

The external cache RAMs can be partitioned into three sections; the tagAdr RAM, the tagCtl RAM, and the data RAM. Sections do not straddle physical RAM chips.

3.2.6.1 The TagAdr RAM

The tagAdr RAM contains the high order address bits associated with the external cache block, along with a parity bit. The contents of the tagAdr RAM is fed to the on-chip address comparator and parity checker via the tagAdr_h and tagAdrP_h inputs.

NVAX Plus verifies that tagAdrP_h is an EVEN parity bit over tagAdr_h when it reads the tagAdr RAM. NVAX Plus asserts c%cbox_hard_error if the parity is wrong and stops the reference.

The number of bits of tagAdr_h that participate in the address compare and the parity check is controlled by the BC_SIZE field in the BIU_CTL IPR. The tagAdr_h signals go all the way down to address bit 17, allowing for a 128Kbyte cache built out of RAMs that are 8K deep.

The chip enable or output enable for the tagAdr RAM is normally driven by a two input NOR gate (such as the 74AS805B). One input of the two input NOR gate is driven by tagCEOE_h, and the other input is driven by external logic. NVAX Plus drives tagCEOE_h false during reset, during external cache hold, and during any external cycle. The OE bit in the BIU_CTL IPR determines if tagCEOE_h has chip enable timing or output enable timing.

3.2.6.2 The TagCtl RAM

The tagCtl RAM contains control bits associated with the external cache block, along with a parity bit. NVAX Plus reads the tagCtl RAM via the three tagCtl signals to determine the state of the block. NVAX Plus writes the tagCtl RAM via the three tagCtl signals to make blocks dirty.

NVAX Plus verifies that tagCtlP_h is an EVEN parity bit over tagCtlV_h, tagCtlS_h, and tagCtlD_h when it reads the tagCtl RAM. NVAX Plus asserts c%cbox_hard_err if the parity is wrong and stops the reference. NVAX Plus computes EVEN parity across the tagCtlV_h, tagCtlS_h, and tagCtlD_h bits, and drives the result onto the tagCtlP_h pin, when it writes the tagCtl RAM.

The following combinations of the tagCtl RAM bits are allowed. Note that the bias toward conditional write-through coherence is really only in name; the tagCtlS_h bit can be viewed simply as a write protect bit.

Table 3–6: Tag Control Encodings

tagCtlV_h	tagCtlS_h	tagCtlD_h	Meaning
F	X	X	Invalid
T	F	F	Valid, private
T	F	T	Valid, private, dirty
T	T	F	Valid, shared
T	T	T	Valid, shared, dirty

NVAX Plus can satisfy a read probe if the tagCtl bits indicate the entry is valid (tagCtlV_h = T). NVAX Plus can satisfy a write probe if the tagCtl bits indicate the entry is valid and not shared (tagCtlV_h = T, tagCtlS_h = F).

The chip enable or output enable for the tagCtl RAM is normally driven by a two input NOR gate (such as the 74AS805B). One input of the two input NOR gate is driven by tagCEOE_h, and the other input is driven by external logic. NVAX Plus drives tagCEOE_h false during reset, during external cache hold, and during any external cycle. The OE bit in the BIU_CTL IPR determines if tagCEOE_h has chip enable timing or output enable timing.

The write enable for the tagCtl RAM is normally driven by a two input NOR gate (such as the 74AS805B). One input of the two input NOR gate is driven by tagCtlWE_h, and the other input is driven by external logic. NVAX Plus drives tagCtlWE_h false during reset, during external cache hold, and during any external cycle.

3.2.6.3 The Data RAM

The data RAM contains the actual cache data, along with any ECC or parity bits.

The most significant bits of the data RAM address are driven, via buffers, from the address bus. The least significant bit of the data RAM address is driven by a two input NOR gate (such as the 74AS805B). One of the inputs of the two input NOR gate is driven by dataA_h[4], and the other input is driven by external logic. NVAX Plus drives dataA_h[4] false during reset, during external cache hold, and during any external cycle.

The chip enables or output enables for the data RAM are driven by a two input NOR gate (such as the 74AS805B). One input of the two input NOR gate is driven by dataCEOE_h[3..0], and the other input is driven by external logic. NVAX Plus drives dataCEOE_h[3..0] false during reset, during external cache hold, and during external cycles. (NVAX Plus sometimes drives dataCEOE_h[3..0] true during external write cycles, to simplify merging old cache data with new write data). The OE bit in the BIU_CTL IPR determines if dataCEOE_h[3..0] has chip enable timing or output enable timing.

The write enables for the data RAM are normally driven by a two input NOR gate (such as the 74AS805B). One input of the two input NOR gate is driven by dataWE_h[3..0], and the other input is driven by external logic. NVAX Plus drives dataWE_h[3..0] false during reset, during external cache hold, and during any external cycle.

3.2.6.4 Backmaps

Some systems may wish to maintain backmaps of the contents of the Pcache to improve the quality of their invalidate filtering. NVAX Plus must maintain the backmaps for external cache read hits, since external cache read hits are controlled totally by NVAX Plus. External logic maintains the backmaps for external cycles (read misses, invalidates, and so on).

The backmaps are only consulted by external logic, so that their format, or, for that matter, their existence, is of no concern to NVAX Plus. All NVAX Plus does is generate backmap write pulses at the right time. Simple systems will not bother to maintain backmaps, will not connect the backmap write pulses to anything, and will generate extra invalidates.

The NVAX Plus Pcache is 8kB and can be configured as either a single set of 256 indexes, or two sets of 128 indexes each. If NVAX Plus is allocating Pcache as two way set associative NVAX Plus drives pMapWE_h[0] or pMapWE_h[1] depending on the Pcache set which is to be allocated whenever it fills the Pcache from the external cache, and systems must assert the corresponding pInvReq_h[1:0] to invalidate an entry in Pcache.

If NVAX Plus is allocating Pcache as direct mapped pMapWE_h[0] is driven and systems assert pInvReq_h[0] to invalidate an entry in Pcache.

The pMapWE_h[1..0] signals assert two cpuClkOut cycles into the second (ast) data read cycle and negate at the end of that cycle.

3.2.6.5 External Cache Access

The external caches are normally controlled by NVAX Plus. Two methods exist for gaining access to the external cache RAMs.

3.2.6.5.1 HoldReq and HoldAck

The simple method for external logic to access the external caches is to assert the holdReq_h signal. NVAX Plus finishes any external cache cycle that is in progress, and then tri-states adr_h, data_h, check_h, tagCtlV_h, tagCtlS_h, tagCtlD_h, and tagCtlP_h, drives tagCEOE_h, tagCtlWE_h, dataCEOE_h, dataWE_h, and dataA_h false, and asserts holdAck_h (the cReq_h and cWMask_h signals are not modified in any way). When external logic is finished with the external caches it negates holdReq_h. NVAX Plus detects the negation of holdReq_h, negates holdAck_h, and re-enables its outputs.

** Systems which use tagOK to obtain access to the cache can assert HoldReq with tagOK deasserted in order to have NVAX Plus tri-state adr_h, data_h, check_h, tagCtlV_h, tagCtlS_h, tagCtlD_h, and tagCtlP_h, drives tagCEOE_h, tagCtlWE_h, dataCEOE_h, dataWE_h, and dataA_h false, and asserts holdAck_h. This allows system which do not use external muxing access to the tag store.**

The holdReq_h signal is synchronous, and external logic must guarantee setup and hold requirements with respect to the system clock. The holdAck_h signal is synchronous to the CPU clock but phase aligned to the system clock, so it can be used as an input to state machines running off the system clock.

The delay from holdReq_h assertion to holdAck_h assertion depends on the programming of the external cache interface, and exactly how the system clock is aligned with a pending external cache cycle. In the best case the external cache is idle or just about to begin a cycle, and holdAck_h asserts at the same system clock edge that samples the holdReq_h assertion. The worst case latency for holdAck_h is three cache access cycles.

A holdReq_h/holdAck_h sequence can happen at any time, even in the middle of an external cycle. All of the acknowledge-like signals (dOE_l, dWSel_h, dRAck_h, cAck_h) work normally, although since NVAX Plus has forced most of its outputs to either tri-state or false, doing anything useful with them is difficult.

3.2.6.5.2 TagOk

The fastest way for external logic to gain access to the external caches is to use the tagOk_l signal. TagOk_l is an NVAX Plus bus interface control signal that allows external logic to stall a CPU cycle on the external cache RAMs at the last possible instant. All tradeoffs surrounding the tagOk_l signal have been made in favor of high-performance systems making tagOk_l next to impossible to use in low-end systems.

The tagOk_l signal is synchronous, external logic must guarantee setup and hold requirements with respect to the CPU clock. This implies very fast logic, since the CPU clock can run at 200 MHz for the binned parts.

Furthermore, the only thing that tagOk_l does is stall a sequencer in the NVAX Plus bus interface unit. NVAX Plus does not tri-state the busses that run between NVAX Plus and the external cache RAMs(unless HoldReq is asserted). External logic must supply the necessary multiplexing functions in the address and data path.

If the tagOk_l signal is true at the falling edge of the CPU_CLK prior to a cache cycle, the external logic is guaranteeing that the tagCtl and tagAdr RAMs were owned by NVAX Plus in the previous cache_speed cycles, that the tagCtl RAMs will be owned by NVAX Plus in the next cache_speed cycles, that the data RAMs were owned by NVAX Plus in the previous cache_speed cycles, and that the data RAMs will be owned by NVAX Plus in the next two cache_speed cycles.

NVAX Plus samples the tagOk_l signal at the very end of the tag read of an external cache cycle. If tagOk_l is true then NVAX Plus knows that no conflict is possible between external logic and its cycle. If tagOk_l is false NVAX Plus stalls. NVAX Plus knows that there is some kind of conflict (it may have already happened, or it may be going to happen before NVAX Plus can finish its cycle). In this case NVAX Plus stalls until tagOk_l is true (at which time all of the above assertions are true, which means, in particular, that any address NVAX Plus has been holding on the address bus all this time has made it through the external cache RAMs), and then it proceeds normally.

3.2.7 External Cycle Control

NVAX Plus requests an external cycle when it determines that the cycle it wants to run requires module level action.

An external cycle begins when NVAX Plus puts a cycle type onto the cReq_h outputs. Some cycles put an address on the adr_h outputs, and additional information (low-order address bits, I/D stream indication, write masks) on the cWMask_h outputs. All of these outputs are synchronous, and NVAX Plus meets setup and hold requirements with respect to the system clock.

The cycle types are as follows.

Table 3–7: Cycle Types

cReq_h[2]	cReq_h[1]	cReq_h[0]	Type
F	F	F	IDLE
F	F	T	not generated-BARRIER
F	T	F	not generated-FETCH
F	T	T	not generated-FETCHM
T	F	F	READ_BLOCK
T	F	T	WRITE_BLOCK
T	T	F	LDxL
T	T	T	STxC

The BARRIER, FETCH and FETCHM cycles are functions generated by EV instructions and are not generated in NVAX Plus systems.

The READ_BLOCK cycle is generated on read misses. External logic reads the addressed block from memory and supplies it, 128 bits at a time, to NVAX Plus via the data bus. External logic may also write the data into the external cache, after writing a victim if necessary.

The WRITE_BLOCK cycle is generated on write misses, and on writes to shared blocks. External logic pulls the write data, 128 bits at a time, from NVAX Plus via the data bus, and writes the valid longwords to memory. External logic may also write the data into the external cache, after writing a victim if necessary.

The LDxL cycle is generated READ_LOCK microinstruction or for writing byte/word data. The cycle works just like a READ_BLOCK, although the external cache has not been probed (so the external logic needs to check for hits), and the address has to be latched into a locked address register.

The STxC cycle is generated by the WRITE_UNLOCK microinstruction and for writes of merged byte/word data. The cycle works just like a WRITE_BLOCK, although the external cache has not been probed (so that external logic needs to check for hits), and the cycle can be acknowledged with a failure status.

On WRITE_BLOCK and STxC cycles the cWMask_h pins supply longword write masks to the external logic, indicating which longwords in the 32-byte block are, in fact, valid. A cWMask_h bit is true if the longword is valid. WRITE_BLOCK commands can have any combination of mask bits set.

NOTE: For NVAX PLus STxC cycles can have any combination of mask bits set, where STxC cycles for EV can only have combinations that correspond to a single longword or quadword.

On READ_BLOCK and LDxL cycles the cWMask_h pins have additional information about the miss overloaded onto them. The cWMask_h[1..0] pins contain miss address bits [4..3] (indicating the address of the quadword that actually missed), which is needed to implement quadword read granularity to I/O devices. The cWMask_h[2] pin is true if the address is not I/O space and will be filled to Pcache. Thus cWMask_h[2] looks like an EV D-stream reference to enable system logic to backmap the NVAX Plus mixed I/D stream Pcache with the D-Map backmap. The cWMask_h[3] pin is false for references that are targeted to bank 0 of the on-chip Pcache, and true for references that are targeted to bank 1 of the on-chip Pcache. The cWMask_h[4] pin is true for I-stream references for use by system logic, i.e. possible I-Stream prefetch to memory.

The cycle holds on the external interface until external logic acknowledges it, by placing an acknowledgment type on the cAck_h pins. The cAck_h inputs are synchronous, and external logic must guarantee setup and hold requirements with respect to the system clock.

The acknowledgment types are as follows.

Table 3–8: Acknowledgment Types

cAck_h[2]	cAck_h[1]	cAck_h[0]	Type
F	F	F	IDLE
F	F	T	HARD_ERROR
F	T	F	SOFT_ERROR
F	T	T	STxC_FAIL
T	F	F	OK

The HARD_ERROR type indicates that the cycle has failed in some catastrophic manner. NVAX Plus latches sufficient state to determine the cause of the error, and machine checks or initiates the hard error interrupt.

The SOFT_ERROR type indicates that a failure occurred during the cycle, but the failure was corrected. NVAX Plus latches sufficient state to determine the cause of the error, and initiates a soft error interrupt.

The STxC_FAIL type indicates that a STxC cycle has failed. It is UNDEFINED what happens if this type is used on anything but an STxC cycle.

The OK type indicates success.

The dRAck_h pins inform NVAX Plus that read data is valid on the data bus, and if ECC checking and correction or parity checking should be attempted. NVAX Plus loads Pcache based on the reference type, i.e. for non I/O space reads. NVAX Plus determines whether data is to be cached and does not use dRAck_h[1]. The dRAck_h inputs are synchronous, and external logic must guarantee setup and hold requirements with respect to the system clock. If dRAck_h is sampled IDLE at a system clock then the data bus is ignored. If dRAck_h is sampled non IDLE at a system clock then the data bus is latched at that system clock, and external logic must guarantee that the data meets setup and hold with respect to the system clock.

The acknowledgment types are as follows.

Table 3–9: Read Data Acknowledgment Types

dRAck_h[2]	dRAck_h[0]	Type
F	F	IDLE
T	F	OK_NCHK
T	T	OK

The first non IDLE sample of dRAck_h tells NVAX Plus to sample data bytes [15..0], and the second non IDLE sample of dRAck_h tells NVAX Plus to sample data bytes [31..16]. Normally external logic will drive the second dRAck_h and the cAck_h during the same system clock. READ_BLOCK and LDxL transactions may be terminated with HARD_ERROR status before all expected dRAck_h cycles are received.

It is UNDEFINED what happens if dRAck_h is asserted in a non-read cycle.

NVAX Plus checks dRAck_h[0] (the bit that determines if the block is ECC/parity checked) during both halves of the 32-byte block. It is legal, but probably not useful, to check only one half of the block.

NVAX Plus does not use dRAck_h[1].

The dOE_l inputs tells NVAX Plus if it should drive the data bus. It is a synchronous input, so external logic must guarantee setup and hold with respect to the system clock. If dOE_l is sampled true at a system clock then NVAX Plus drives the data bus at the system clock if it has a WRITE_BLOCK or STxC request pending (the request may already be on the cReq pins, or it may appear on the cReq pins at the same system clock edge as the data appears). If dOE_l is sampled false at the system clock then NVAX Plus tri-states the data bus on the next system clock cycle. The cycle type is factored into the enable so that systems can leave dOE_l asserted unless it is necessary to write a victim.

The dWSel_h inputs of EV are not needed as NVAX Plus only presents 1 octaword the data bus.

3.2.8 Primary Cache Invalidate

External logic needs to be able to invalidate primary cache blocks to maintain coherence. NVAX Plus provides a mechanism to perform the necessary invalidates, but enforces no policy as to when invalidates are needed. Simple systems may choose to invalidate more or less blindly, and complex systems may choose to implement elaborate invalidate filters.

There are two situations where entries in the on-chip Pcachemay need to be invalidated.

The first situation is the obvious one. Any time an external agent updates a block in memory (for example, an I/O device does a DMA transfer into memory), and that block has been loaded into the external cache, then the external cache block must be either invalidated or updated. If that external cache block has been loaded into a block resident in the Pcache then that Pcache entry must be invalidated.

External logic invalidates an entry in bank 0 of the Pcache by asserting the pInvReq_h[0] signal. NVAX Plus samples pInvReq_h[0] at every system clock. When NVAX Plus detects pInvReq_h[0] asserted, it invalidates the block in bank 0 of the Pcache whose index is on the iAdr_h pins.

External logic invalidates an entry in bank 1 of the Pcache by asserting the pInvReq_h[1] signal. NVAX Plus samples pInvReq_h[1] at every system clock. When NVAX Plus detects pInvReq_h[1] asserted, it invalidates the block in bank 1 of the Pcache whose index is on the iAdr_h pins.

If the Pcache is set to direct map allocation only PinvReq[0] is asserted, iAdr[12] selects the section of Pcache to be invalidated.

It is legal to both pInvReq_h[1..0] in the same cycle.

NVAX Plus can accept an invalidate at every system clock.

The pInvReq_h[1..0] inputs are synchronous, and external logic must guarantee setup and hold with respect to the system clock. The iAdr_h inputs are also synchronous, and external logic must guarantee setup and hold with respect to the system clock in any cycle in which any of pInvReq_h[1..0] are true.

3.2.9 Interrupts

External interrupts are fed to NVAX Plus via the irq_h bus. The 6 interrupts are wired to IRQ<3:0>, halt, and error. The timer interrupt is internal to NVAX Plus. The interrupts are asynchronous, and level sensitive.

3.2.10 Electrical Level Configuration

NVAX Plus drives and receives CMOS levels.

The vRef input supplies a reference voltage to the input sense circuits. If external logic ties this to VSS + 1.4V then all inputs sense TTL levels.

3.2.11 Testing

The tristate_l signal, if asserted, causes NVAX Plus to float all of its pins, with the exception of the clocks.

The cont_l signal, if asserted, causes NVAX Plus to connect all of its pins to VSS, with the exception of the clocks.

3.3 64-Bit Mode

NVAX Plus does not support the EV 64-bit external mode.

3.4 Transactions

3.4.1 Reset

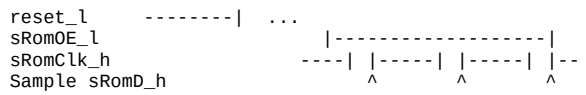
External logic resets NVAX Plus by asserting reset_l. When NVAX Plus detects the assertion of reset_l it terminates all external activity, and places the output signals on the external interface into the following state. Note that all of the control signals have been placed in the state that allows external access to the external cache.

Table 3–10: Reset State

Pin	State
sRomOE_l	F
sRomClk_h	T
adr_h	Z
data_h	Z
check_h	Z
tagCEOE_h	F
tagCtlWE_h	F
tagCtlV_h	Z
tagCtlS_h	Z
tagCtlD_h	Z
tagCtlP_h	Z
dataCEOE_h	F
dataWE_h	F
dataA_h	F
holdAck_h	F
cReq_h	FFF
cWMask_h	FFFFFFFF

After asserting reset_l for long enough to reset the serial ROM (100 ns), external logic negates reset_l.

When NVAX Plus detects reset_l negate, it begins internal initialization. When this initialization is completed NVAX Plus asserts sRomOE_l (enabling the output of the serial ROM onto sRomD_h), and begins clocking bits out of the serial ROM and placing them into the Pcache. The timing is the following (assuming NVAX Plus only read 3 bits from the serial ROM).

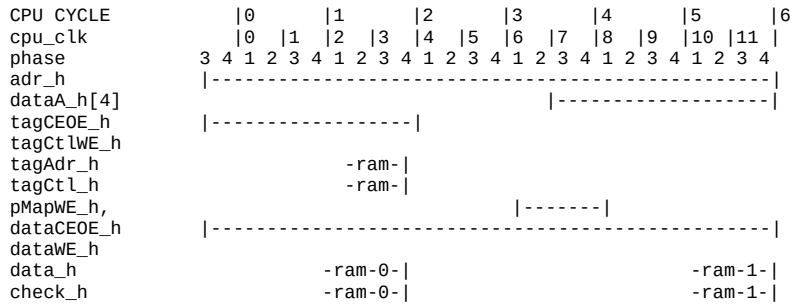


Each half-tick of the sRomClk_h signal is 32 CPU cycles long, which guarantees the 200ns clock high and clock low specifications and the 400ns clock to data specification of the serial ROM with 10ns CPU cycles.

3.4.2 Fast External Cache Read Hit

A fast external cache read consists of a probe read (overlapped with the first data read), followed by the second data read while the probe results are determined. If the probe hits and tagOK_l is asserted the pMapWE_h of the allocated PCache set is driven.

The following diagram assumes that the external cache is using 4X cache_speed timing, chip enable control (OE_H/CE_L = L).



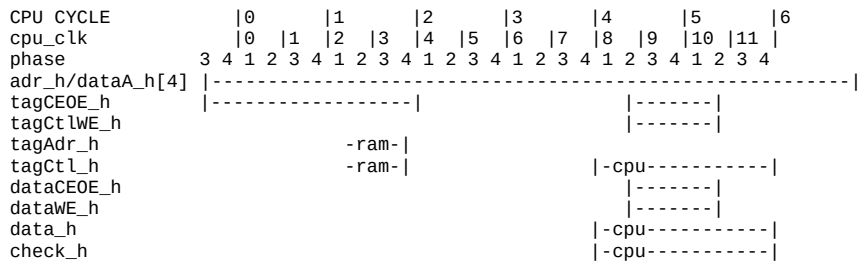
If the probe misses then pMapWE_h does not assert, and the sequence aborts at the end of CPU CYCLE 3.

The address is driven from phase 3 prior to CPU CYCLE 0 and the data is latched at phase 4 of CPU CYCLE 3, providing 9 phases for external access at cache_speed = 4 times the cpu_clk (2CPU CYCLES).

3.4.3 Fast External Cache Write Hit

A fast external cache write consists of a probe read, followed by a compare cycle, and then a single data write.

The following diagram assumes that the external cache is using 2X system clock timing, chip enable control (OE_H/CE_L = L), and a 1 cycle write pulse starting from cpu clock falling edge.

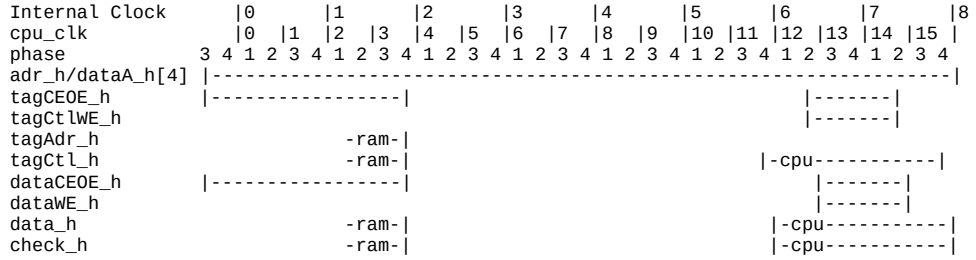


If the probe misses then the cycle aborts at the end of cpu clock cycle 3.

3.4.4 Fast External Cache Byte/Word Write Hit

A fast external cache byte/word write consists of a probe read, followed by a compare cycle, a data merge cycle, and then a single data write.

The following diagram assumes that the external cache is using 2X system clock timing, chip enable control (OE_H/CE_L = L), and a 1 cycle write pulse starting from cpu clock falling edge.

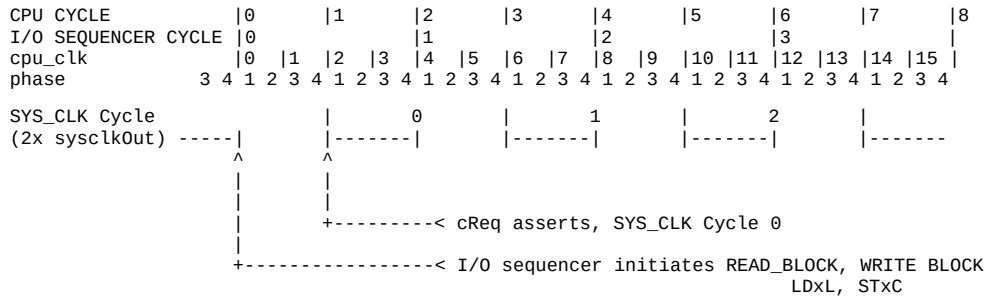


If the probe misses then the cycle aborts at the end of cpu clock cycle 3.

3.4.5 Transfer to SysClk for External tranactions

The remainder of the transactions described in this chapter, READ_BLOCK, WRITE BLOCK, LDxL, and STxC, involve the external system logic, and are described with respect to sysClkOut1. This section describes the delay from the internal cpu cycle which initiates a tranction requiring external system logic, and SYS_CLK cycle 0, where cReq_h is driven with the command request. adr_h and cWMask are valid prior to the start of SYS_CLK cycle 0.

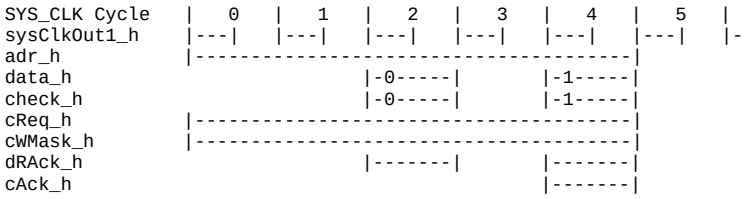
The NVAX Plus I/O sequencer runs once every CACHE_SPEED cycles. If the output of the I/O sequencer initiates a transaction requiring external logic, the cReq_h command is asserted with the next rising edge of sysClkOut1_h. For systems with the CACHE_SPEED and sysClkOut both programmed for 2 CPU cycles, the start of the SYS_CLK cycle is always one CPU cycle after the I/O sequencer initiated the tranaction.



If CACHE_SPEED and sysClkOut are not programmed to the same multiple of cpu_clk, the delay to the rising edge of sysClkOut1_h and the assertion of cReq_h may be a full SYS_CLK cycle.

3.4.6 READ_BLOCK Transaction

A READ_BLOCK transaction appears at the external interface for reads which miss in the Pcache for external cache read misses, either because ithe read really was a miss, or because the external cache has not been enabled.



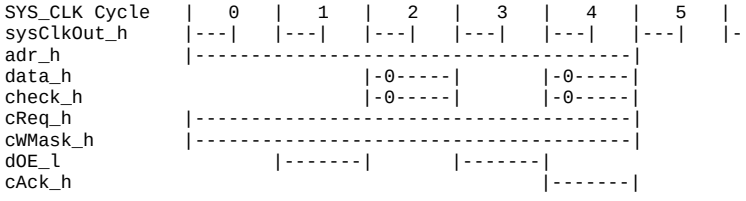
0. The READ_BLOCK cycle begins. NVAX Plus places the address of the block containing the miss on adr_h. NVAX Plus places the quadword-within-block and the I/D indication on cWMask_h. NVAX Plus places a READ_BLOCK command code on cReq_h. The external logic detects the command at the end of this cycle.
1. The external logic obtains the first 16 bytes of data. Although a single stall cycle has been shown here, there could be no stall cycles, or many stall cycles.
2. The external logic has the first 16 bytes of data. It places it on the data_h and check_h busses. It asserts dRAck_h to tell NVAX Plus that the data and check bit busses are valid. NVAX Plus detects dRAck_h at the end of this cycle, and reads in the first 16 bytes of data at the same time.
3. The external logic obtains the second 16 bytes of data. Although a single stall cycle has been shown here, there could be no stall cycles, or many stall cycles.
4. The external logic has the second 16 bytes of data. It places it on the data_h and check_h busses. It asserts dRAck_h to tell NVAX Plus that the data and check bit busses are valid. NVAX Plus detects dRAck_h at the end of this cycle, and reads in the second 16 bytes of data at the same time. In addition, the external logic places an acknowledge code on cAck_h to tell NVAX Plus that the READ_BLOCK cycle is completed. NVAX Plus detects the acknowledge at the end of this cycle, and may change the address
5. Everything is idle. NVAX Plus could start a new external cache cycle at this time.

Note that this picture did not mention the external caches. NVAX Plus drove all of the external cache control signals false when it placed the READ_BLOCK command on the cReq_h outputs. The external logic controls the updating of cache.

NVAX Plus performs ECC checking and correction (or parity checking) on the data supplied to it via the data and check busses if so requested by the acknowledge code. It is not necessary to place data into the external cache to get checking and correction.

3.4.7 Write Block

A WRITE_BLOCK transaction appears at the external interface on external cache write misses (either because it really was a miss, or because the external cache has not been enabled), or on external cache write hits to shared blocks.



0. The WRITE_BLOCK cycle begins. NVAX Plus places the address of the block on adr_h. NVAX Plus places the longword valid masks on cWMask_h. NVAX Plus only write a single octaword

at a time, thus cWMask[7:4] = '0000 if adr_h[4] = '0 or cWMask[3:0] = '0000 if adr_h[4] = '1. The dWsel_h from EV are not needed as NVAX Plus drives the same octaword at the assertion of dOE_l.

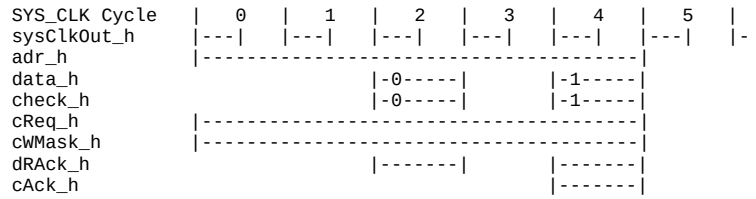
1. NVAX Plus places the WRITE_BLOCK command code on cReq_h. The external logic detects the command at the end of this cycle.
2. The external logic detects the command, and asserts dOE_l to tell NVAX Plus to drive the 16 bytes of data of the block onto the data bus. Although a single stall cycle has been shown here, there could be no stall cycles, or many stall cycles.
3. If the external logic asserts dOE_l a second time to tell NVAX Plus to drive a second 16 bytes of data onto the data bus the same octaword is driven.
4. The external logic places an acknowledge code on cAck_h to tell NVAX Plus that the WRITE_BLOCK cycle is completed. NVAX Plus detects the acknowledge at the end of this cycle, and change the address and command to the next values.
5. Everything is idle.

Note that this picture did not mention the external caches. NVAX Plus drove all of the external cache control signals false when it placed the WRITE_BLOCK command on the cReq_h outputs. The external logic controls the updating of cache.

NVAX Plus performs ECC generation (or parity generation) on data it drives onto the data bus.

3.4.8 LDxL Transaction

An LDxL transaction appears at the external interface as a result of a READ_LOCK micro-instruction or byte/word write which misses in the BCache being executed. The external cache is not probed.



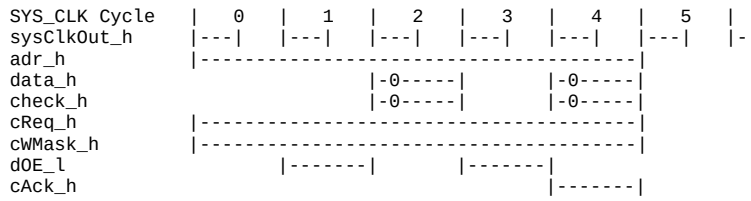
0. The LDxL cycle begins. NVAX Plus places the address of the block containing the data on adr_h. NVAX Plus places the quadword-within-block and the I/D indication on cWMask_h. NVAX Plus places a LDxL command code on cReq_h. The external logic detects the command at the end of this cycle.
1. The external logic obtains the first 16 bytes of data. Although a single stall cycle has been shown here, there could be no stall cycles, or many stall cycles.
2. The external logic has the first 16 bytes of data. It places it on the data_h and check_h busses. It asserts dRAck_h to tell NVAX Plus that the data and check bit busses are valid. NVAX Plus detects dRAck_h at the end of this cycle, and read in the first 16 bytes of data at the same time.
3. The external logic obtains the second 16 bytes of data. Although a single stall cycle has been shown here, there could be no stall cycles, or many stall cycles.

4. The external logic has the second 16 bytes of data. It places it on the data_h and check_h busses. It asserts dRAck_h to tell NVAX Plus that the data and check bit busses are valid. NVAX Plus detects dRAck_h at the end of this cycle, and read in the second 16 bytes of data at the same time. In addition, the external logic places an acknowledge code on cAck_h to tell NVAX Plus that the LDxL cycle is completed. NVAX Plus detects the acknowledge at the end of this cycle, and change the address
5. Everything is idle.

Note that with the exception of the command code output on the cReq pins, the LDxL cycle is the same as a READ_BLOCK cycle.

3.4.9 STxC Transaction

An STxC transaction appears at the external interface as a result of a WRITE_UNLOCK micro_ instruction or byte/word write in which the initial read probe missed in the BCache. The external cache is not probed.



0. The STxC cycle begins. NVAX Plus places the address of the block on adr_h. NVAX Plus places the longword valid masks on cWMask_h. NVAX Plus places an STxC command code on cReq_h. The external logic detects the command at the end of this cycle.
1. The external logic detects the command, and asserts dOE_l to tell NVAX Plus to drive the 16 bytes of the block onto the data bus.
2. NVAX Plus drives the first 16 bytes of write data onto the data_h and check_h busses, and the external logic writes it into the destination. Although a single stall cycle has been shown here, there could be no stall cycles, or many stall cycles.
3. The external logic asserts dOE_l and dWSel_h to tell NVAX Plus to drive the second 16 bytes of data onto the data bus. NVAX continues to drive the same octaword of data. The cWMask_ h output indicates which octaword contains the write data.
4. NVAX Plus drives the second 16 bytes of write data onto the data_h and check_h busses, and the external logic writes it into the destination. Although a single stall cycle has been shown here, there could be no stall cycles, or many stall cycles. In addition, the external logic places an acknowledge code on cAck_h to tell NVAX Plus that the STxC cycle is completed. NVAX Plus detects the acknowledge at the end of this cycle, and change the address and command to the next values.
5. Everything is idle.

Note that with the exception of the code output on the cReq pins, and the fact that external logic has the option of making the cycle fail by using a cAck code of STxC_FAIL, the STxC cycle is the same as the WRITE_BLOCK cycle.

3.4.10 BARRIER Transaction

NVAX Plus does not generate the BARRIER transaction.

3.4.11 FETCH Transaction

NVAX Plus does not generate the FETCH transaction.

3.4.12 FETCHM Transaction

NVAX Plus does not generate the FETCHM transaction.

3.5 Summary of NVAX Plus options

1. SYS_CLK speed, number of CPU cycles, from IRQ lines at reset

```
2X
3X SYMMETRIC (FLAMINGO)
3X ASYMMETRIC (COBRA)
4X    ?
```

2. CACHE_SPEED 2,3, OR 4 CPU CYCLES, from BIU_CTL
3. FLAMINGO-I/O WRITE MAPPING (BIU_CTL[WS_IO])included in CBOX Packer logic specification
4. DIRECT MAP PCACHE, from BIU_CTL

```
allocate Pcache bank based on address<12> instead of allocate bit
1. MBOX
    allocate input to Pcache muxed between allocate bit in
    Miss latches and address<12>
2. CBOX
    invalidate in direct map mode sent to MBOX as;
    invreq<0> if iADR<12> = '0
    invreq<1> if iADR<12> = '1
```

5. QW I/O WRITES, MOVQ and MTPR MAILBOX if not WS_IO

```
force packing to quadword at Packer even if address is I/O space
1. Microcode
   transmit IPR_WR to force quadword pack prior to sending 2 LWs
   to destination for MOVQ instruction
2. CBOX - Packer
   Decode IPR_WR indicating pack next 2 LWs
```

6. QW I/O READS

```
1. a high_LW register is loaded with data<63:32> of I/O read
2. I/O reads with address<2> = '1 (not QW aligned)
   are converted to an IPR_RD of the high_LW register;
   data returns on dat<31:0>
```

7. "PV" mode for LOW END WORKSTATIONS

```
-WRITES GO DIRECTLY TO WRITE_BLOCK
 1. ARB control
    writes dispatch directly to 'SYS_WR' if
  ^Bcache_en or "PV"
  -parity/ecc not needed on "PV" writes
```

```
-OUTPUT BM INSTEAD OF LONG WORD MASK ON cWMask_h
 1. cWMask mux between LWMask and BM<7:0> from Write_Queue
  -3 LINES ALONG WITH MASK
  1. INDICATES QW TO WHICH MASK APPLIES
  2. 2 LINES INDICATES BYTE CONTROL FOR OTHER QW
     from LWMask set at 'ARB pack'

-ON READS COMBINE BYTE PARITY ON CHECK BITS INTO LW PARITY
 1. provide xor tree for 4 check bits for each LW being
    input, for conversion into single LW parity bit
```

8. SW_ECC for diagnostics

3.6 Revision History

Table 3–11: Revision History

Who	When	Description of change
Gil Wolrich	15-Nov-1990	NVAX PLUS release for external review.
Gil Wolrich	15-Jan-1991	Remove Vector references/update.

Chapter 4

Chip Overview

4.1 NVAX Plus CPU Chip Box and Section Overview

The NVAX Plus CPU Chip is a single-chip CMOS-4 macropipelined implementation of the base instruction group, and the optional vector instruction group of the VAX architecture. Included in the chip are:

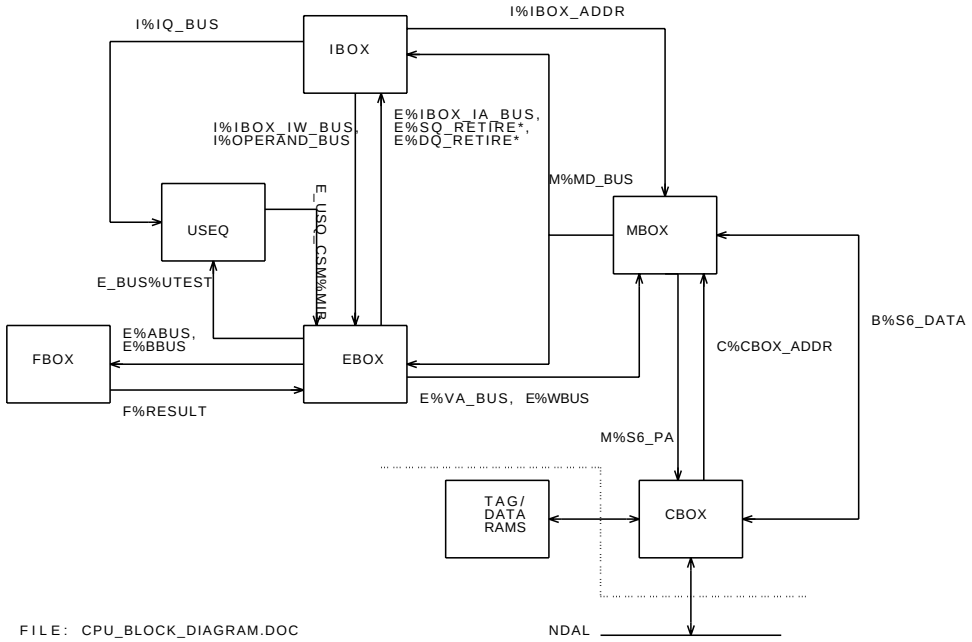
- **CPU:** Instruction fetch and decode, microsequencer, and execution unit
- **Control Store:** **1800**, 61-bit microwords
- **Primary Cache:** 8 KB, 2-way set associative, physically-addressed, write through, mixed instruction and data stream
- **Instruction Cache:** 2 KB, 2-way set associative, virtually addressed, instruction stream only
- **Translation Buffer:** 96 entries, fully associative
- **Floating Point:** 4 stage, pipelined, integrated floating point unit
- **EDAL Interface:** Support for six cache sizes (4MB, 2MB, 1MB, 512KB, 256KB, 128KB), and four RAM speeds.
- **Vector Interface:** Control for an optional Vector Unit

The NVAX chip is designed in CMOS-4 with a typical cycle time of 14 ns, and with the option of running chips at a slower or faster cycle time. The chip can be incorporated into many different system environments, ranging from the desktop to the midrange, and from single processor to multiprocessor systems.

The NVAX is a macropipelined design: it pipelines macroinstruction decode and operand fetch with macroinstruction execution. Pipeline efficiency is increased by queuing up instruction information and operand values for later use by the execution unit. Thus, when the macropipeline is running smoothly, the Ibox (instruction parser/operand fetcher) is running several macroinstructions ahead of the Ebox (execution unit). Outstanding writes to registers or memory locations are kept in a scoreboard to ensure that data is not read before it has been written. See Chapter 5 for a more in-depth discussion of the macropipeline.

This chapter gives an overview of the different sections, or "boxes", that comprise the NVAX Plus CPU. For more information on any of the boxes, please see the appropriate chapters within this specification. Figure 4–1 is a block diagram of the boxes, and the major buses that run between them.

Figure 4–1: NVAX Plus CPU Block Diagram



4.1.1 The Ibox

The Ibox decodes VAX instructions and parses operand specifiers. Instruction control, such as the control store dispatch address, is then placed in the instruction queue for later use by the Microsequencer and Ebox. The Ibox processes the operand specifiers at a rate of one specifier per cycle and, as necessary, initiates specifier memory read operations. All the information needed to access the specifiers is queued in the source queue and destination queue in the Ebox.

The Ibox prefetches instruction stream data into the prefetch queue (PFQ), which can hold 16 bytes. The Ibox has a dedicated instruction-stream-only cache, called the virtual instruction cache (VIC). The VIC is a 2 KB, with a block and fill size of 32 bytes.

The Ibox has both read and write ports to the GPR and MD portions of the Ebox register file which are used to process the operand specifiers. The Ibox maintains a scoreboard to ensure that reads and writes to the register file are always performed in synchronization with the Ebox. The Ibox stops processing instructions and operands upon issuing certain complex instructions (for example, CALL, RET, and character string instructions). This is done to maintain read/write ordering when the Ebox will be altering large amounts of VAX state.

Since the Ibox is often parsing several macroinstructions ahead of the Ebox, the correct value for the PSL condition codes is not known at the time the Ibox executes a conditional branch instruction. Rather than emptying the pipe, the Ibox predicts which direction the branch will take, and passes this information on to the Ebox via the branch queue. The Ebox later signals if there was a misprediction, and the hardware backs out of the path. The branch prediction algorithm utilizes a 512-entry RAM, which caches four bits of branch history per entry.

4.1.2 The Ebox and Microsequencer

The Ebox and Microsequencer work together to perform the actual "work" of the VAX instructions. Together they implement a four stage micropipelined unit, which has the ability to stall and to microtrap. The Ebox and Microsequencer dequeue instruction and operand information provided by the Ibox via the instruction queue, the source queue, and the destination queue. For literal type operands, the source queue contains the actual operand value. In the case of register, memory, and immediate type operands, the source queue holds a pointer to the data in the Ebox register file. The contents of memory operands are provided by the Mbox based on earlier requests from the Ibox. GPR results are written directly back to the register file. Memory results are sent to the Mbox, where the data will be matched with the appropriate specifier address previously sent by the Ibox. At times, the Ebox initiates its own memory reads and writes using **E%VA_BUS** and **E%WBUS**.

The Microsequencer determines the next microword to be fetched from the control store. It then provides this cycle-by-cycle control to the Ebox. The Microsequencer allows for eight-way microbranches, and for microsubroutines to a depth of six.

The Ebox contains a five-port register file, which holds the VAX GPRs, six Memory Data Registers (MDs), six microcode working registers, and ten miscellaneous CPU state registers. It also contains an ALU, a shifter, and the VAX PSL. The Ebox uses the RMUX, controlled by the retire queue, to order the completion of Ebox and Fbox instructions. As the Ebox and the Fbox are distinct hardware resources, there is some amount of execution overlap allowed between the two units.

The Ebox implements specialized hardware features in order to speed the execution of certain VAX instructions: the population counter (CALLx, PUSHR, POPR), and the mask processing unit (CALLx, RET, FFx, PUSHR, POPR). The Ebox also has logic to gather hardware and software interrupt requests, and to notify the Microsequencer of pending interrupts.

4.1.3 The Fbox

The Fbox implements a four staged pipelined execution unit for the floating point and integer multiply instructions. Operands are supplied by the Ebox up to 64 bits per cycle on **E%ABUS** and **E%BBUS**. Results are returned to the Ebox 32 bits per cycle on **F%RESULT**. The Ebox is responsible for storing the Fbox result in memory or the GPRs.

4.1.4 The Mbox

The Mbox receives read requests from the Ibox (both instruction stream and data stream) and from the Ebox (data stream only). It receives write/store requests from the Ebox. Also, the Cbox sends the Mbox fill data and invalidates for the Pcache. The Mbox arbitrates between these requesters, and queues requests which cannot currently be handled. Once a request is started, the Mbox performs address translation and cache lookup in two cycles, assuming there are no misses or other delays. The two-cycle Mbox operation is pipelined.

The Mbox uses the translation buffer (96 fully associative entries) to map virtual to physical addresses. In the case of a TB miss, the memory management hardware in the Mbox will read the page table entry and fill the TB. The Mbox is also responsible for all access checks, TNV checks, M-bit checks, and quadword unaligned data processing.

The Mbox houses the Primary Cache (Pcache). The Pcache is 8KB, writethrough, with a block and fill size of 32 bytes.

The Pcache can be configured at reset to be either direct mapped or 2-way set associative.

The Pcache state is maintained as a subset of the Backup Cache. System logic, possibly using backmaps, is responsible for insuring the Pcache is maintained as a subset of the Backup Cache.

The Mbox ensures that Ibox specifier reads are ordered correctly with respect to Ebox specifier stores. This memory "scoreboarding" is accomplished by using the PA queue, a small list of physical addresses which have a pending Ebox store.

4.1.5 The Cbox

The Cbox initiates access to the second level cache (the Backup Cache, or Bcache), and issues memory requests. Both the tags and data for the Bcache are stored in off-chip RAMs. The size and access time of the Bcache RAMs can be configured as needed by different system environments. The Bcache sizes supported are 4 MB, 2 MB, 1 MB, 512 KB, 256 KB, and 128 KB. System logic is responsible for BCache fills and coherency functions. The Cbox packs sequential writes to the same octaword in order to minimize Bcache write accesses. Multiple write commands are held in the eight-entry WRITE_QUEUE.

4.1.6 Major Internal Buses

This is a list of the major interbox buses:

- **B%S6_DATA**:
  This bidirectional bus between the Cbox and MBox is used to transfer write data to the backup cache, to to transfer fill data to the primary cache.
- **C%CBOX_ADDR**:
  This bus is used to transfer the physical address of a Pcache invalidate from the Cbox to the MBox.
- **E%ABUS**, **E%BBUS**:
  These two 32-bit buses contain the A- and B-port operands for the Ebox, and are also used to transfer operand data to the Fbox.
- **E%IBOX_IA_BUS**:
  This bus is used by the Ibox to read the Ebox Register File in order to perform an operand access. An example is to read a register's contents for a register deferred type specifier.

- **E%DQ_RETIRE***:
  This collection of related buses transfers information from the Ebox to the Ibox when a destination queue entry is retired.
- **E%SQ_RETIRE***:
  This collection of related buses transfers information from the Ebox to the Ibox when a source queue entry is retired.
- **E%VA_BUS**:
  This bus transfers an address from the Ebox to the MBox.
- **E%WBUS**:
  This 32-bit bus transfers write data from the RMUX to the register file and the Mbox.
- **E_USQ_CSM%MIB**:
  This bus carries Control Store data from the Microsequencer to the Ebox.
- **E_BUS%UTEST**:
  This 3-bit bus transfers microbranch conditions from the Ebox to the microsequencer.
- **F%RESULT**:
  This bus is used to transfer results from the Fbox to the Ebox.
- **I%IBOX_ADDR**:
  This bus transmits the virtual address of an Ibox memory reference to the Mbox. The address may be for instruction prefetch or an operand access.
- **I%IQ_BUS**:
  This bus carries instruction information from the Ibox to the Instruction Queue in the Microsequencer.
- **I%IBOX_IW_BUS**:
  This bus is used by the Ibox to write the Ebox Register File for autoincrement/decrement type specifiers and to deliver immediate operands to the Register File.
- **I%OPERAND_BUS**:
  This bus transfers information from the Ibox to the source and destination queues in the Ebox.
- **M%MD_BUS**:
  The bus returns right-justified memory read data from the Mbox to either the Ibox (64 bits) or the Ebox (32 bits).
- **M%S6_PA**:
  This bus transfers the address for a backup cache reference from the MBox to the Cbox.

4.2 Revision History

Table 4–1: Revision History

Who	When	Description of change
Debra Bernstein	06-Mar-1989	Release for external review.
Mike Uhler	18-Dec-1989	Update for second-pass release.
Gil Wolrich	15-Nov-1990	Update for NVAX Plus external release.

Chapter 5

Macroinstruction and Microinstruction Pipelines

5.1 Introduction

This chapter discusses the architecture of the NVAX Plus CPU macroinstruction and microinstruction pipeline. It includes a section of general pipeline fundamentals to set the stage for the specific NVAX Plus CPU implementation of the pipeline. This is followed by an overview of the NVAX Plus CPU pipeline, an examination of macroinstruction execution, and a discussion of stall and exception handling from the viewpoint of the Ebox.

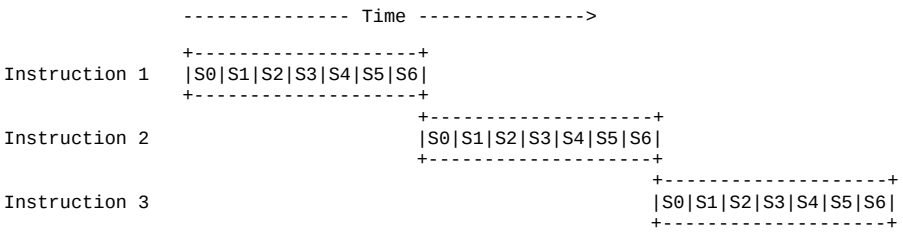
5.2 Pipeline Fundamentals

This section discusses the fundamentals of instruction pipelining in a general manner that is independent of the NVAX Plus CPU implementation. It is intended as a primer for those readers who do not understand the concept and implications of instruction pipelining. Readers familiar with this material are encouraged to skip (or at most skim) this section.

5.2.1 The Concept of a Pipeline

The execution of a VAX macroinstruction involves a sequence of steps which are carried out in order to complete the macroinstruction operation. Among these steps are: instruction fetch, instruction decode, specifier evaluation and operand fetch, instruction execution, and result store. On the simplest machines, these steps are carried out sequentially, with no overlap of the steps, as shown in Figure 5–1.

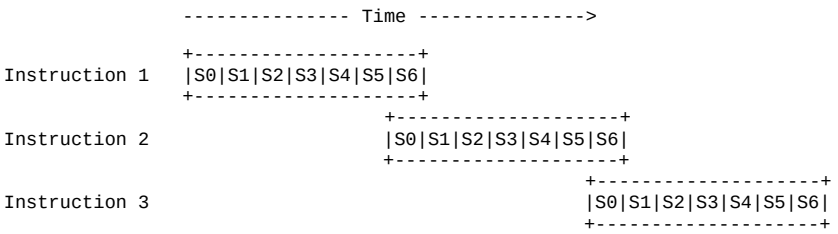
Figure 5–1: Non-Pipelined Instruction Execution



In this diagram, "S0", "S2", ..., "S6" denote particular steps in the execution of an instruction. For this simple scheme, all of the steps for one instruction are performed, and the instruction is completed, before any of the steps for the next instruction are started.

In more complex machines, one or more steps of the execution process are carried out in parallel with other steps. For example, consider Figure 5–2.

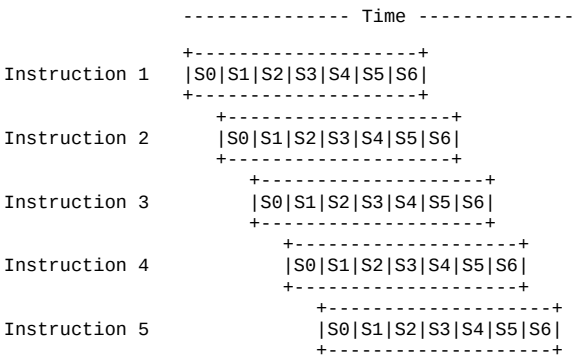
Figure 5–2: Partially-Pipelined Instruction Execution



In this example, step S6 of each instruction is overlapped in time (or executed in parallel) with step S0 of the next instruction. In doing so, the number of instructions executed per unit time (instruction throughput) goes up because an instruction appears to take less time to complete.

In the most complex machines, most (or all) of the steps are executed in parallel as indicated in Figure 5–3.

Figure 5–3: Fully-Pipelined Instruction Execution



In this example every step of instruction execution is performed in parallel with every other step. This means that a new instruction is started as soon as step S0 is completed for the previous instruction. If each step, S0..S6, took the same amount of time, the apparent instruction throughput would be seven times greater than that of Figure 5–1 above, even though each instruction takes the same amount of time to execute in both cases.

Figures 5–2 and 5–3 are examples of the concept of instruction pipelining, in which one or more steps necessary to execute an instruction are performed in parallel with steps for other instructions.

5.2.2 Pipeline Flow

A real-world form of a pipeline is an automobile assembly line. At each station of the assembly line (called segments of the pipeline in our case), a task is performed on the partially completed automobile and the result is passed on to the next station. At the end of the assembly line, the automobile is complete.

In an instruction pipeline, as in an assembly line, each segment is responsible for performing a task and passing the completed result to the next segment. The exact task to be performed in each pipeline segment is a function of the degree of pipelining implemented and the complexity of the instruction set.

One attribute of an automobile assembly line is equally important to an instruction pipeline: smooth and continuous flow. An automobile assembly line works well because the tasks to be performed at each station take about the same amount of time. This keeps the line moving at a constant pace, with no starts and stops which would reduce the number of completed automobiles per unit time.

An analogous situation exists in an instruction pipeline. In order to achieve real efficiency in an instruction pipeline, information must flow smoothly and continuously from the start of the pipeline to the end. If a pipeline segment somewhere in the middle is not able to supply results to the next segment of the pipeline, the entire pipeline after the offending segment must stop, or stall, until the segment can supply a result.

In the general case, a pipeline stall results when a pipeline segment can not supply a result to the next segment, or when it can not accept a new result from a previous segment.

This is a fundamental problem with most instruction pipelines because they occasionally (or not so occasionally) stall. Stalls result in decreased instruction throughput because the smooth flow of the pipeline is broken.

A typical example of a pipeline stall involves memory reads. A simple three-segment pipeline might fetch operands in segment 1, use the operands to compute results in segment 2, and make memory references or store results in segment 3, as shown in Figure 5–4.

Figure 5–4: Simple Three-Segment Pipeline

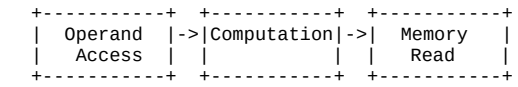
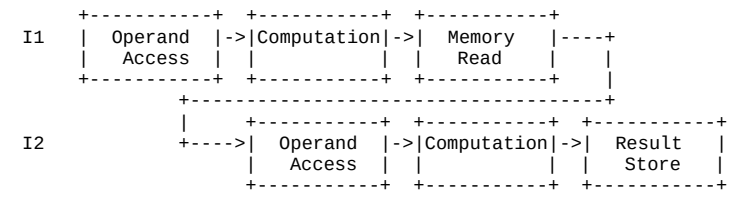


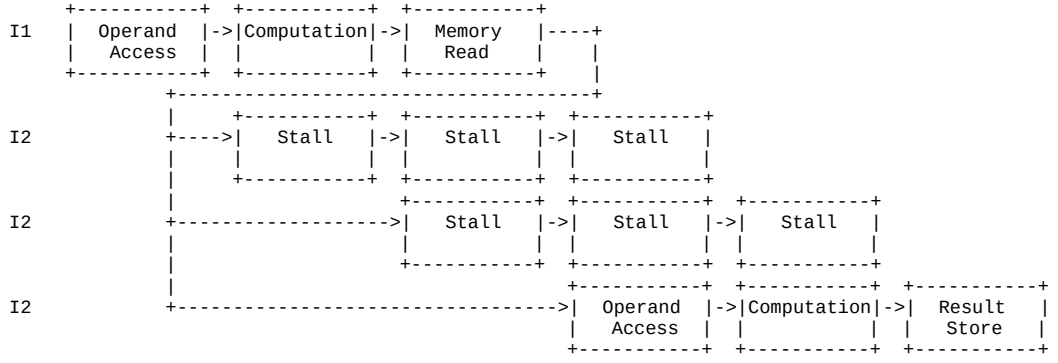
Figure 5–5 illustrates what happens when the pipeline control wants to use the result of the memory read as an operand.

Figure 5–5: Information Flow Against the Pipeline



In this case, the operand access segment of I2 can not supply an operand to the computation segment because the memory read done by I1 has not yet completed. As a result, the pipeline must stall until the memory read has completed. This is shown in Figure 5–6.

Figure 5–6: Stalls Introduced by Backward Pipeline Flow



In this diagram, the memory read data from I1 is not available until the read request passes through segment 3 of the pipeline. But the operand access segment for I2 wants the data immediately. The result is that the operand access segment of I2 has to stall twice waiting for the memory read data to become available. This, in turn, stalls the rest of the pipeline segments after the operand access segment.

This situation is an excellent example of an age-old problem with instruction pipelining. The natural and desired direction of information flow in a pipeline is from left to right in the above diagrams. In this case, information must flow from the output of the memory read segment into the operand access segment. This requires a right-to-left movement of information from a later pipeline segment to an earlier one. In general, any information transfer which goes against the normal flow of the pipeline has the potential for causing pipeline stalls.

5.2.3 Stalls and Exceptions in an Instruction Pipeline

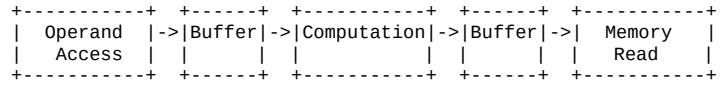
Even the best pipeline design must be prepared to deal with stalls and exceptions created in the pipeline. As mentioned above, a stall is a condition in which a pipeline segment can not accept a new result from a previous segment, or can not send a result to a new segment. An exception occurs when a pipeline segment detects an abnormal condition which must stop, and then drain the pipeline. Examples of exceptions are: memory management faults, reserved operand faults, and arithmetic overflows. One of the inherent costs of a pipelined implementation is the extra logic necessary to deal with stalls and exceptions.

There are two primary considerations concerning stalls: what action to take when one occurs, and how to minimize them in the first place. The design of most instruction pipelines assumes that the pipeline will not stall, and handles the stall condition as a special case, rather than the other way around. This means that each segment of the pipeline performs its function and produces a result each cycle. If a stall occurs just before the end of the cycle, the segment must block global state updates and repeat the same operation during the next cycle. The design of the pipeline control must take this into account and be prepared to handle the condition.

A common stall condition occurs when each pipeline segment has the same average speed, but different peak speeds. For example, a pipeline segment whose task is to perform both memory references and register result stores may take longer to perform memory references than result stores. This can cause earlier segments of the pipeline to stall because the segment can not take new inputs as fast if it is doing a memory reference rather than a result store. A common

technique to minimize this problem is to place buffers between pipeline segments, as shown in Figure 5–7.

Figure 5–7: Buffers Between Pipeline Segments



By placing a buffer of sufficient depth between each segment of the pipeline, segments of differing peak speeds can avoid stalls caused if the next segment is unable to accept a new result. Instead, the result goes into the inter-segment buffer and the next segment removes it from the buffer when it needs it. Unfortunately, adding such buffers means that additional logic must also be added to handle the buffer full/buffer empty conditions.

The performance advantage of an instruction pipeline comes from the parallelism built into the pipeline. If the parallelism is defeated by, for example, a stall, the advantage starts to drop. One problem associated with pipelines is that they can provide "lumpy" performance. That is, two similar programs may experience radically different performance if one causes many more stalls (which defeat the parallelism of the pipeline) than the other.

Pipeline exceptions are different from stalls in that exceptions cause the pipeline to empty or drain. Usually, everything that entered the pipeline before the point of error is allowed to complete. Everything that entered the pipeline after the point of error is prevented from completing. This can add considerable complexity to the pipeline control.

A larger problem occurs when the designer wants exceptions to be recoverable. Consider an exception caused by a memory management fault. On the VAX, this condition can occur because of a TB miss. The correct response to this fault is to read a PTE from memory, refill the TB, and restart the request that caused the fault. This can add considerable complexity to the design.

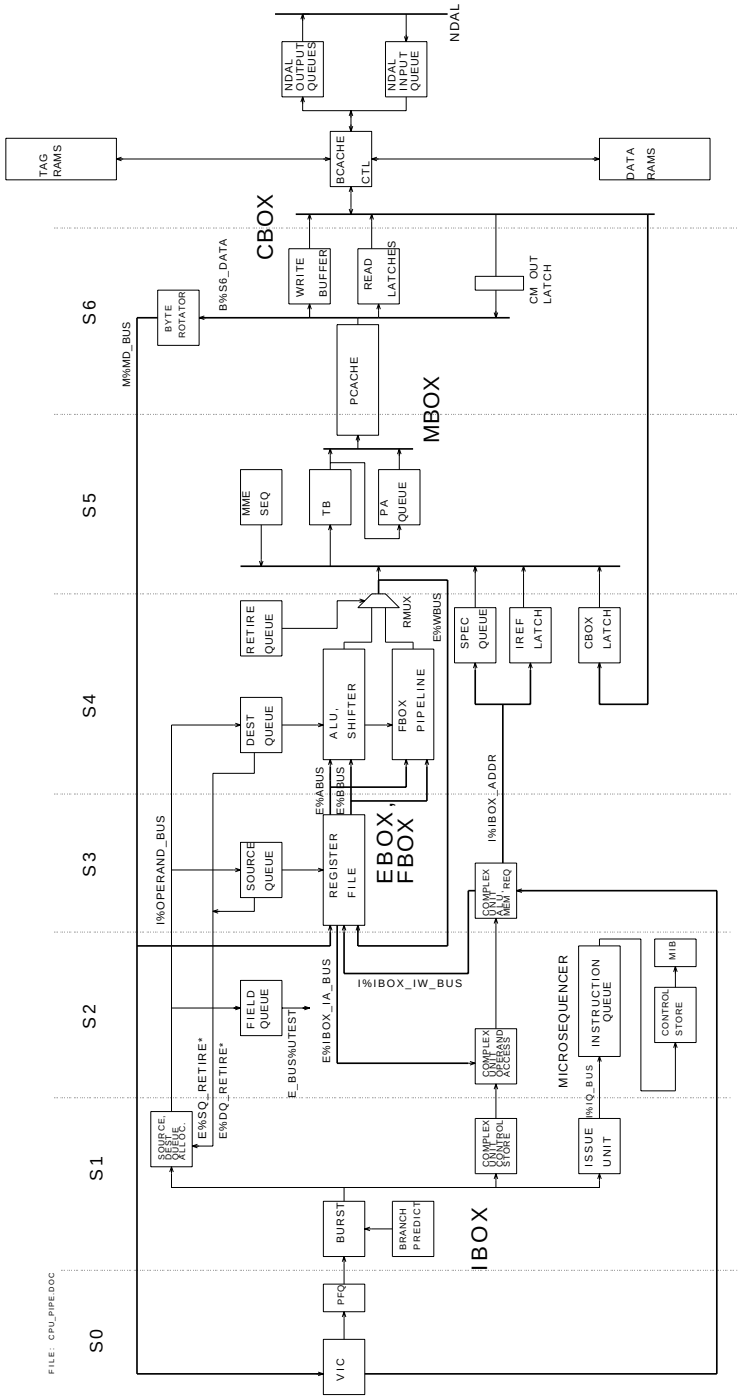
5.3 NVAX Plus CPU Pipeline Overview

The remainder of this chapter discusses the NVAX Plus CPU pipeline, which is shown as a block diagram in Figure 5–8. This is a high-level view of the CPU and abstracts many of the details. For a more detailed view of the pipeline, users are encouraged to refer to the individual box chapters in this specification.

The pipeline is divided into seven segments denoted as "S0" through "S6". In Figure 5–8, the components of each section of the CPU are shown in the segment of the pipeline in which they operate.

The NVAX Plus CPU is fully pipelined and, as such, is most similar to the abstract example shown in Figure 5–3. In addition to the overall macroinstruction pipeline, in which multiple macroinstructions are processed in the various segments of the pipeline, most of the sections also micropipeline operations. That is, if more than one operation is required to process a macroinstruction, the multiple operations are also pipelined within a section.

Figure 5–8: NVAX Plus CPU Pipeline



5.3.1 Normal Macroinstruction Execution

Execution of macroinstructions in the NVAX pipeline is decomposed into many smaller steps which are the distributed responsibility of the various sections of the chip. Because the NVAX Plus CPU implements a macroinstruction pipeline, each section is relatively autonomous, with queues inserted between the sections to normalize the processing rates of each section.

5.3.1.1 The Ibox

The Ibox is responsible for fetching instruction stream data for the next instruction, decomposing the data into opcode and specifiers, and evaluating the specifiers with the goal of prefetching operands to support Ebox execution of the instruction.

The Ibox is distributed across segments S0 through S3 of the pipeline, with most of the work being done in S1. In S0, instruction stream data is fetched from the virtual instruction cache (VIC) using the address contained in the virtual instruction buffer address register (VIBA). The data is written into the prefetch queue (PFQ) and VIBA is incremented to the next location.

In segment S1, the PFQ is read and the burst unit uses internal state and the contents of the IROM to select the next instruction stream component—either an opcode or specifier. This decoding processing is known as *bursting*. Some instruction components take multiple cycles to burst. For example, FD opcodes require two burst cycles: one for the FD byte, and one for the second opcode byte. Similarly, indexed specifiers require at least two burst cycles: one for the index byte, and one or more for the base specifier.

When an opcode is decoded, the information is passed to the issue unit, which consults the IROM for the initial Ebox control store address of the routine which will process the instruction. The issue unit sends the address and other instruction-related information to the instruction queue where it is held until the Ebox reaches the instruction.

When a specifier is decoded, the information is passed to the source and destination queue allocation logic and, potentially, to the complex specifier pipeline. The source and destination queue allocation logic allocates the appropriate number of entries for the specifier in the source and destination queues in the Ebox. These queues contain pointers to operands and results, and are discussed in more detail below.

If the specifier is not a short literal or register specifier, which are collectively known as *simple* specifiers, it is considered to be a *complex* specifier and is processed by the small microcode-controlled complex specifier unit (CSU), which is distributed in segments S1 (control store access), S2 (operand access, including register file read), and S3 (ALU operation, Mbox request, GPR write) of the pipeline. The CSU pipeline computes all specifier memory addresses, and makes the appropriate request to the Mbox for the specifier type. To avoid reading or writing a GPR which is interlocked by a pending Ebox reference, the CSU pipeline includes a register scoreboard which detects data dependencies. The CSU pipeline also provides additional help to the Ebox by supplying operand information that is not an explicit part of the instruction stream. For example, the PC is supplied as an implicit operand for instructions that require it (such as BSBB).

The branch prediction unit (BPU) watches each opcode that is decoded looking for conditional and unconditional branches. For unconditional branches, the BPU calculates the target PC and redirects PC and VIBA to the new path. For conditional branches, the BPU predicts whether the instruction will branch or not based on previous history. If the prediction indicates that the branch will be taken, PC and VIBA are redirected to the new path. The BPU writes the conditional branch prediction flag into the branch queue in the Ebox, to be used by the Ebox in the execution

of the instruction. The BPU maintains enough state to restore the correct instruction PC if the prediction turns out to be incorrect.

5.3.1.2 The Microsequencer

The microsequencer operates in segment S2 of the pipeline and is responsible for supplying to the Ebox the next microinstruction to execute. If a macroinstruction requires the execution of more than one microinstruction, the microsequencer supplies each microinstruction in sequence based on directives included in the previous microinstruction.

At macroinstruction boundaries, the microsequencer removes the next entry from the instruction queue, which includes the initial microinstruction address for the macroinstruction. If the instruction queue is empty, the microsequencer supplies the address of a special no-op microinstruction.

The microsequencer is also responsible for evaluating all exception requests, and for providing a pipeline flush control signal to the Ebox. For certain exceptions and interrupts, the microsequencer injects the address of a special microinstruction handler that is used to respond to the event.

5.3.1.3 The Ebox

The Ebox is responsible for executing all of the non-floating point instructions, for delivery of operands to and receipt of results from the Fbox, and for handling non-instruction events such as interrupts and exceptions. The Ebox is distributed through segments S3 (operand access, including register file read), S4 (ALU and shifter operation, Rmux request), and S5 (Rmux completion, register write, completion of Mbox request) of the pipeline.

For the most part, instruction operands are prefetched by the Ibox, and addressed indirectly through the source queue. The source queue contains the operand itself for short literal specifiers, and a pointer to an entry in the register file for other operand types.

An entry in the field queue is made when a field-type specifier entry is made into the source queue. The field queue provides microbranch conditions that allow the Ebox microcode to determine if a field-type specifier addresses either a GPR or memory. A microbranch on a valid field queue entry retires the entry from the queue.

The register file is divided into four parts: the GPRs, memory data (MD) registers, working registers, and CPU state registers. For register-mode specifiers, the source queue points to the appropriate GPR in the register file. For other non-short literal specifier modes, the source queue points to an MD register. The MD register is either written directly by the Ibox, or by the Mbox as the result of a memory read generated by the Ibox.

The S3 segment of the Ebox pipeline is responsible for selecting the appropriate operands for the Ebox and Fbox execution of instructions. Operands are selected onto **E%ABUS** and **E%BBUS** for use in both the Ebox and Fbox. In most instances, these operands come from the register file, although there are other data path sources of non-instruction operands (such as the PSL).

Ebox computation is done by the ALU and the shifter in the S4 segment of the pipeline on operands supplied by the S3 segment. Control for these units is supplied by the microinstruction which was originally supplied to the S3 segment by the microsequencer, and then subsequently moved forward in the pipeline.

The S4 segment also contains the RMUX, whose responsibility is to select results from either the Ebox or Fbox and perform the appropriate register or memory operation. The RMUX inputs come from the ALU, shifter, and **F%RESULT** at the end of the cycle. The RMUX actually spans the S4/S5 boundary such that its outputs are valid at the beginning of the S5 segment. The RMUX is controlled by the retire queue, which specifies the source (either Ebox or Fbox) of the result to be processed (or retired) next. Non-selected RMUX sources are delayed until the retire queue indicates that they should be processed.

As the source queue points to instruction operands, so the destination queue points to the destination for instruction results. If the result is to be stored in a GPR, the destination queue contains a pointer to the appropriate GPR. If the result is to be stored in memory, the destination queue indicates that a request is to be made to the Mbox, which contains the physical address of the result in the PA queue (which is described below). This information is supplied as a control input to the RMUX logic.

Once the RMUX selects the appropriate source of result information, it either requests Mbox service, or sends the result onto **E%WBUS** to be written back to the register file or to other data path registers in the S5 segment of the pipeline. The interface between the Ebox and Mbox for all memory requests is the EM_LATCH, which contains control information and may contain an address, data, or both, depending on the type of request. In addition to operands and results that are prefetched by the Ibox, the Ebox can also make explicit memory requests to the Mbox to read or write data.

5.3.1.4 The Fbox

The Fbox is responsible for executing all of the floating point instructions in the VAX base instruction group, as well as the longword-length integer multiply instructions.

For each instruction that the Fbox is to execute, it receives from the microsequencer the opcode and other instruction-related information. The Fbox receives operand data from the Ebox on **E%ABUS** and **E%BBUS**.

Execution of instructions is performed in a dedicated Fbox pipeline that appears in segment S4 of Figure 5–8, but is actually a minimum of three cycles in length. Certain instructions, such as integer multiply, may require multiple passes through some segments of the Fbox pipeline. Other instructions, such as divide, are not pipelined at all.

Fbox results and status are returned via **F%RESULT** to the RMUX in the Ebox for retirement. When the instruction is next to retire, the RMUX hardware, as directed by the destination queue, sends the results to either the GPRs for register destinations, or to the Mbox for memory destinations.

5.3.1.5 The Mbox

The Mbox operates in the S5 and S6 segments of the pipeline, and is responsible for all memory references initiated by the other sections of the chip. Mbox requests can come from the Ibox (for VIC fills and for specifier references), the Ebox or Fbox via the RMUX and the EM_LATCH (for instruction result stores and for explicit Ebox memory requests), from the Mbox itself (for translation buffer fills and PTE reads), and from the Cbox (for invalidates and cache fills).

All virtual references are translated to a physical address by the translation buffer (TB), which operates in the S5 segment of the pipeline. For instruction result references generated by the Ibox, the translated address is stored in the physical address queue (PA queue). These addresses are later matched with data from the Ebox or Fbox, when the result is calculated.

For memory references, the physical address from either the TB or the PA queue is used to address the primary cache (Pcache) starting in the S5 segment of the pipeline and continuing into the S6 segment. Read data is available in the middle of the S6 segment, right-justified and returned to the requester on **M%MD_BUS** by the end of the cycle. Writes are also completed by the end of the cycle. Although the Pcache access spans the S5 and S6 segments of the pipeline, a new access can be started each cycle in the absence of a TB or cache miss.

5.3.1.6 The Cbox

The Cbox is responsible for accessing the backup cache (Bcache), and for memory requests. The Cbox receives input from the Mbox in the S6 segment of the pipeline, and usually takes multiple cycles to complete a request. For this reason, the Cbox is not shown in specific pipeline segments.

If a memory read misses in the Pcache, the request is sent to the Cbox for processing. The Cbox first looks for the data in the Bcache and fills the Pcache from the Bcache if the data is present. If the data is not present in the Bcache, the Cbox requests a cache fill from the system. When the system returns the data, it is written to the Pcache (and potentially to the VIC). Although Pcache fills are done by making a request to the Mbox pipeline, data is returned to the original requester as quickly as possible by driving data directly onto **B%S6_DATA**, and from there onto **M%MD_BUS** as soon as the bus is free.

Because the Pcache operates as a write-through cache, all memory writes are passed to the Cbox. To avoid multiple writes to the same Bcache block, the Cbox contains a write buffer in which multiple writes to the same quadwords are packed. If possible two quadwords (an octaword) are assembled together before the Bcache is actually written.

5.3.2 Stalls in the Pipeline

Despite our best attempts at keeping the pipeline flowing smoothly, there are conditions which cause segments of the pipeline to stall. Conceptually, each segment of the pipeline can be considered as a black box which performs three steps every cycle:

1. The task appropriate to the pipeline segment is performed, using control and inputs from the previous pipeline segment. The segment then updates local state (within the segment), but not global state (outside of the segment).
2. Just before the end of the cycle, all segments send stall conditions to the appropriate state sequencer for that segment, which evaluates the conditions and determines which, if any, pipeline segments must stall.
3. If no stall conditions exist for a pipeline segment, the state sequencer allows it to pass results to the next segment and accept results from the previous segment. This is accomplished by updating global state.

This sequence of steps maximizes throughput by allowing each pipeline segment to assume that a stall will not occur (which should be the common case). If a stall does occur at the end of the cycle, global state updates are blocked, and the stalled segment repeats the same task (with potentially different inputs) in the next cycle (and the next, and the next) until the stall condition is removed.

This description is over-simplified in some cases because some global state must be updated by a segment before the stall condition is known. Also, some tasks must be performed by a segment once and only once. These are treated specially on a case-by-case basis in each segment.

Within a particular section of the chip, a stall in one pipeline segment also causes stalls in all upstream segments (those that occur earlier in the pipeline) of the pipeline. Unlike Rigel, stalls in one segment of the pipeline do not cause stalls in downstream segments of the pipeline. For example, a memory data stall in Rigel also caused a stall of the downstream ALU segment. In NVAX Plus, a memory data stall does not stall the ALU segment (a no-op is inserted into the S4 segment when S4 advances to S5).

There are a number of stall conditions in the chip which result in a pipeline stall. Each is discussed briefly below and in much more detail in the appropriate chapter of this specification.

5.3.2.1 S0 Stalls

Stalls that occur in the S0 segment of the pipeline are as follows:

Ibox:

- PFQ full: In normal operation, the VIC is accessed using the address in VIBA, the data is sent to the prefetch queue, and VIBA is incremented. If the PFQ is full, the increment of VIBA is blocked, and the data is re-referenced in the VIC until there is room for it in the PFQ. At that point, prefetch resumes.

5.3.2.2 S1 Stalls

Stalls that occur in the S1 segment of the pipeline are as follows:

Ibox:

- Insufficient PFQ data: The burst unit attempts to decode the next instruction component each cycle. If there are insufficient PFQ bytes valid to decode the entire component, the burst unit stalls until the required bytes are delivered from the VIC.
- Source queue or destination queue full: During specifier decoding, the source and destination queue allocation logic must allocate enough entries in each queue to satisfy the requirements of the specifier being parsed. To guarantee that there will be sufficient resources available, there must be at least 2 free source queue entries and 2 free destination queue entries to complete the burst of the specifier. If there are insufficient free entries in either queue,the burst unit stalls until free entries become available.
- MD file full: When a complex specifier is decoded, the source queue allocation logic must allocate enough memory data registers in the register file to satisfy the requirements of the specifier being parsed. To guarantee that there will be sufficient resources available, there must be at least 2 free memory data registers available to complete the burst of the specifier. If there are insufficient free registers, the burst unit stalls until enough memory data registers becomes available.

- Second conditional branch decoded: The branch prediction unit predicts the path that each conditional branch will take and redirects the instruction stream based on that prediction. It retains sufficient state to restore the alternate path if the prediction was wrong. If a second conditional branch is decoded before the first is resolved by the Ebox, the branch prediction unit has nowhere to store the state, so the burst unit stalls until the Ebox resolves the actual direction of the first branch.
- Instruction queue full: When a new opcode is decoded by the burst unit, the issue unit attempts to add an entry for the instruction to the instruction queue. If there are no free entries in the instruction queue, the burst unit stalls until a free entry becomes available, which occurs when an instruction is retired through the RMUX.
- Complex specifier unit busy: If the burst unit decodes an instruction component that must be processed by the CSU pipeline, it makes a request for service by the CSU through an S1 request latch. If this latch is still valid from a previous request for service (either due to a multi-cycle flow or a CSU stall), the burst unit stalls until the valid bit in the request latch is cleared.
- Immediate data length not available: The length of the specifier extension for immediate specifiers is dependent on the data length of the specifier for that specific instruction. The data length information comes from one of the Ibox instruction PLAs which is accessed based on the opcode of the instruction. If the PLA access is not complete before an immediate specifier is decoded (which would have to be the first specifier of the instruction), the burst unit stalls for one cycle.

5.3.2.3 S2 Stalls

Stalls that occur in the S2 segment of the pipeline are as follows:

Ibox:

- Outstanding Ebox or Fbox GPR write: In order to calculate certain specifier memory addresses, the CSU must read the contents of a GPR from the register file. If there is a pending Ebox or Fbox write to the register, the Ibox GPR scoreboard prevents the GPR read by stalling the S2 segment of the CSU pipeline. The stall continues until the GPR write completes.
- Memory data not valid: For certain operations, the Ibox makes an Mbox request to return data which is used to complete the operation (e.g., the read done for the indirect address of a displacement deferred specifier). The Ibox MD register contains a valid bit which is cleared when a request is made, and set when data returns in response to the request. If the Ibox references the Ibox MD register when the valid bit is off, the S2 segment of the CSU pipeline stalls until the data is returned by the Mbox.

Microsequencer:

- Instruction queue empty: The final microinstruction of a macroinstruction execution flow in the Ebox is indicated when a SEQ.MUX/LAST.CYCLE* microinstruction is decoded by the microsequencer. In response to this event, the Ebox expects to receive the first microinstruction of the next macroinstruction flow based on the initial address in the instruction queue. If the instruction queue is empty, the Microsequencer supplies the instruction queue stall microinstruction in place of the next macroinstruction flow. In effect, this stalls the microsequencer for one cycle.

5.3.2.4 S3 Stalls

Stalls that occur in the S3 segment of the pipeline are as follows:

Ibox:

- Outstanding Ebox GPR read: In order to complete the processing for auto-increment, auto-decrement, and auto-increment deferred specifiers, the CSU must update the GPR with the new value. If there is a pending Ebox read to the register through the source queue, the Ibox scoreboard prevents the GPR write by stalling the S3 segment of the CSU pipeline. The stall continues until the Ebox reads the GPR.
- Specifier queue full: For most complex specifiers, the CSU makes a request for Mbox service for the memory request required by the specifier. If there are no free entries in the specifier queue, the S3 segment of the CSU pipeline stalls until a free entry becomes available.
- RLOG full: Auto-increment, auto-decrement, and auto-increment deferred specifiers require a free RLOG entry in which to log the change to the GPR. If there are no free RLOG entries when such a specifier is decoded, the S3 segment of the CSU pipeline stalls until a free entry becomes available.

Ebox:

- Memory read data not valid: In some instances, the Ebox may make an explicit read request to the Mbox to return data in one of the 6 Ebox working registers in the register file. When the request is made, the valid bit on the register is cleared. When the data is written to the register, the valid bit is set. If the Ebox references the working register when the valid bit is clear, the S3 segment of the Ebox pipeline stalls until the entry becomes valid.
- Field queue not valid: For each macroinstruction that includes a field-type specifier, the microcode microbranches on the first entry in the field queue to determine whether the field specifier addresses a GPR or memory. If the field queue is empty (indicating that the Ibox has not yet parsed the field specifier), the result of the next address calculation repeats the microbranch the next cycle. Although this is not a true stall, the effects are the same in that a microinstruction is repeated until the field queue becomes valid.
- Outstanding Fbox GPR write: Because the Fbox computation pipeline is multiple cycles long, the Ebox may start to process subsequent instructions before the Fbox completes the first. If the Fbox instruction result is destined for a GPR that is referenced by a subsequent Ebox microword, the S3 segment of the Ebox pipeline stalls until the Fbox GPR write occurs.
- Fbox instruction queue full: When an instruction is issued to the Fbox, an entry is added to the Fbox instruction queue. If there are no free entries in the queue, the S3 segment of the Ebox pipeline stalls until a free entry becomes available.

Ebox/Fbox:

- Source queue empty: Most instruction operands are prefetched by the Ibox, which writes a pointer to the operand value into the source queue. The Ebox then references up to two operands per cycle indirectly through the source queue for delivery to the Ebox or Fbox. If either of the source queue entries referenced is not valid, the S3 segment of the Ebox pipeline stalls until the entry becomes valid.

- Memory operand not valid: Memory operands are prefetched by the Ibox, and the data is written by the either the Mbox or Ibox into the memory data registers in the register file. If a referenced source queue entry points to a memory data register which is not valid, the S3 segment of the Ebox pipeline stalls until the entry becomes valid.

5.3.2.5 S4 Stalls

Stalls that occur in the S4 segment of the pipeline are as follows:

Ebox:

- Branch queue empty: When a conditional or unconditional branch is decoded by the Ibox, an entry is added to the branch queue. For conditional branch instructions, the entry indicates the Ibox prediction of the branch direction. The branch queue is referenced by the Ebox to verify that the branch displacement was valid, and to compare the actual branch direction with the prediction. If the branch queue entry has not yet been made by the Ibox, the S4 segment of the Ebox pipeline stalls until the entry is made.
- Fbox GPR operand scoreboard full: The Ebox implements a register scoreboard to prevent the Ebox from reading a GPR to which there is an outstanding write by the Fbox. For each Fbox instruction which will write a GPR result, the Ebox adds an entry to the Fbox GPR scoreboard. If the scoreboard is full when the Ebox attempts to add an entry, the S4 segment of the Ebox pipeline stalls until a free entry becomes available.

Fbox:

- Fbox operand not valid: Instructions are issued to the Fbox when the opcode is removed from the instruction queue by the microsequencer. Operands for the instruction may not arrive until some time later. If the Fbox attempts to start the instruction execution when the operands are not yet valid, the Fbox pipeline stalls until the operands become valid.

Ebox/Fbox:

- Destination queue empty: Destination specifiers for instructions are processed by the Ibox, which writes a pointer to the destination (either GPR or memory) into the destination queue. The destination queue is referenced in two cases: when the Ebox or Fbox store instruction results via the RMUX, and when the Ebox tries to add the destination of Fbox instructions to the Ebox GPR scoreboard. If the destination queue entry is not valid (as would be the case if the Ibox has not completed processing the destination specifier), a stall occurs until the entry becomes valid.
- PA queue empty: For memory destination specifiers, the Ibox sends the virtual address of the destination to the Mbox, which translates it and adds the physical address to the PA queue. If the destination queue indicates that an instruction result is in memory, a store request is made to the Mbox which supplies the data for the result. The Mbox matches the data with the first address in the PA queue and performs the write. If the PA queue is not valid when the Ebox or Fbox has a memory result ready, the RMUX stalls until the entry becomes valid. As a result, the source of the RMUX input (Ebox or Fbox) also stalls.
- EM_LATCH full: All implicit and explicit memory requests made by the Ebox or Fbox pass through the EM_LATCH to the Mbox. If the Mbox is still processing the previous request when a new request is made, the RMUX stalls until the previous request is completed. As a result, the source of the RMUX input (Ebox or Fbox) also stalls.

- RMUX selected to other source: Macroinstructions must be completed in the order in which they appear in the instruction stream. The Ebox retire queue determines whether the next instruction to complete comes from the Ebox or the Fbox. If the next instruction should come from one source and the other makes an RMUX request, the other source stalls until the retire queue indicates that the next instruction should come from that source.

5.3.3 Exception Handling

A pipeline exception occurs when a segment of the pipeline detects an event which requires that the normal flow of the pipeline be stopped in favor of another flow. There are two fundamental types of pipeline exceptions: those that resume the original pipeline flow once the exception is corrected, and those that require the intervention of the operating system. A TB miss on a memory reference is an example of the first type, and an access control violation is an example of the second type. M=0 faults are handled specially, as described below.

Restartable exceptions are handled entirely within the confines of the section that detected the event. Other exceptions must be reported to the Ebox for processing. Because the NVAX Plus CPU is macropipelined, exceptions can be detected by sections of the pipeline long before the instruction which caused the exception is actually executed by the Ebox or Fbox. However, the reporting of the exception is deferred until the instruction is executed by the Ebox or Fbox. At that point, an Ebox handler is invoked to process the event.

Because the Ebox and Fbox are micropipelined, the point at which an exception handler is invoked must be carefully controlled. For example, three macroinstructions may be in execution in segments S3, S4, and S5 of the Ebox pipeline. If an exception is reported for the macroinstruction in the S3 segment, the two macroinstructions that are in the S4 and S5 segments must be allowed to complete before the exception handler is invoked.

To accomplish this, the S4/S5 boundary in the Ebox is defined to be the *commit point* for a microinstruction. Architectural state is not modified before the S5 segment of the pipeline, unless there is some mechanism for restoring the original state if an exception is detected (the Ibox RLOG is an example of such a mechanism). Exception reporting is deferred until the microinstruction to which the event belongs attempts to cross the S4/S5 boundary. At that point, the exception is reported and an exception handler is invoked. By deferring exception reporting to this point, the previous microinstruction (which may belong to the previous macroinstruction) is allowed to complete.

Most exceptions are reported by requesting a *microtrap* from the Microsequencer. When the Microsequencer receives a microtrap request, it causes the Ebox to break all its stalls, aborts the Ebox pipeline (by asserting **E_USQ%PE_ABORT**), and injects the address of a handler for the event into the control store address latch. This starts an Ebox microcode routine which will process the exception as appropriate. Certain other kinds of exceptions are reported by simply injecting the appropriate handler address into the control store at the appropriate point.

The VAX architecture categorizes exceptions into two types: faults and traps. For both types, the microcode handler for the exception causes the Ibox to back out all GPR modifications that are in the RLOG, and retrieves the PC from the PC queue. For faults, the PC returned is the PC of the opcode of the instruction which caused the exception. For traps, the PC returned is the PC of the opcode of the next instruction to execute. The microcode then constructs the appropriate exception frame on the stack, and dispatches to the operating system through the appropriate SCB vector.

There are a number of exceptions detected by the NVAX Plus CPU pipeline, each of which is discussed briefly below, and in much more detail in the appropriate chapter of this specification.

5.3.3.1 Interrupts

The CPU services interrupt requests from various sources between macroinstructions, and at selected points within the string instructions. Interrupt requests are received by the interrupt section and compared with the current IPL in the PSL. If the interrupt request is for an IPL that is higher than the current value in the PSL, a request is posted to the microsequencer. At the next macroinstruction boundary, the microsequencer substitutes the address of the microcode interrupt service routine for the instruction execution flow.

The microcode handler then determines if there is actually an interrupt pending. If there is, it is dispatched to the operating system through the appropriate SCB vector.

5.3.3.2 Integer Arithmetic Exceptions

There are three integer arithmetic exceptions detected by the CPU, all of which are categorized as traps by the VAX architecture. This is significant because the event is not reported until after the commit point of the instruction, which allows that instruction to complete.

Integer Overflow Trap

An integer overflow is detected by the RMUX at the end of the S4 segment of the Ebox pipeline. If PSL<IV> is set and overflow traps are enabled by the microcode, the event is reported in segment S5 of the pipeline via a microtrap request.

Integer Divide-By-Zero Trap

An integer divide-by-zero is detected by the Ebox microcode routine for the instruction. It is reported by explicitly retiring the instruction and then jumping directly to the microcode handler for the event.

Subscript Range Trap

A subscript range trap is detected by the Ebox microcode routine for the INDEX instruction. It is reported by explicitly retiring the instruction and then jumping directly to the microcode handler for the event.

5.3.3.3 Floating Point Arithmetic Exceptions

All floating point arithmetic exceptions are detected by the Fbox pipeline during the execution of the instruction. The event is reported by the RMUX when it selects the Fbox as the source of the next instruction to process. At that point, a microtrap is requested.

5.3.3.4 Memory Management Exceptions

Memory management exceptions are detected by the Mbox when it processes a virtual read or write. This section covers actual memory management exceptions such as access control violation, translation not valid, and M=0 faults. Translation buffer misses are discussed separately in the next section. Because the reporting of memory management exceptions is specific to the operation that caused the exception, each case is discussed separately.

- **I-Stream Faults**

  While the Ibox is decoding instructions, it may access a page which is not accessible due to a memory management exception. This may occur on the opcode, a specifier or specifier extension, or on a branch displacement. Should this occur, the Ibox sets a global MME fault flag and stops. Memory management exceptions detected on intermediate operations during specifier evaluation (such as a read for the indirect address of a displacement deferred specifier) are converted by the Ibox into source or destination faults, as described below.

  If the Ebox reaches the instruction which caused the exception (which may not happen due to, for example, interrupt, exception, or branch), it will reference one of the queues, which does not have a valid entry because the Ibox stopped when the error was detected. The particular queue depends on the instruction component on which the error was detected. If the Ibox global MME flag is set when an empty queue entry is referenced, the error is reported in one of four ways.

  If the Ibox global MME flag is set when the microsequencer references an invalid instruction queue entry, it inserts the instruction queue stall into the pipeline and the Ebox qualifies it with the fault flag. When this flag reaches the S4 segment of the pipeline and is selected by the RMUX, a microtrap is requested.

  If the Ibox global MME flag is set when the Ebox references an invalid source queue entry, a fault flag is injected into either the Ebox or Fbox pipelines, depending on the type of instruction. To avoid a deadlock, S3 stalls do not prevent forward prgress of the flag in the pipeline. When the flag reaches the S4 segment of the pipeline and is selected by the RMUX, a microtrap is requested.

  If the Ibox global MME flag is set when the Ebox microcode microbranches on an invalid field queue entry, a fault flag is injected into the Ebox pipeline. When the flag reaches the S4 segment of the pipeline and is selected by the RMUX, a microtrap is requested.

  If the Ibox global MME flag is set when the Ebox references an invalid branch queue entry, and the RMUX selects the Ebox, a microtrap is requested.

  If the Ibox global MME flag is set when the RMUX references an invalid destination queue entry for a store request, a microtrap is requested.

- **Source Operand Faults**

  If the Mbox detects a memory management exception during the translation for a source specifier, it qualifies the data returned to the MD file with a fault flag which is written into the MD file. When this entry is referenced by the Ebox, a fault flag is injected into the pipeline. To avoid a deadlock, S3 stalls do not prevent forward prgress of the flag in the pipeline. When the flag reaches the S4 segment of the pipeline and is selected by the RMUX, a microtrap is requested.

- **Destination Address Faults**

  If the Mbox detects a memory management exception during the translation for a destination specifier, it sets a fault flag in the PA queue entry for the address. When this entry is referenced by the RMUX, a microtrap is requested,.

- **Faults on Explicit Ebox Memory Requests**

  Explicit Ebox reads and writes are, by definition, performed in the context of the instruction which the Ebox is currently executing. If the Mbox detects a memory management exception that was the result of an explicit Ebox read or write, it requests an immediate microtrap to the memory management fault handler.

- **M=0 faults**

  M=0 faults occur when the Mbox finds the M-bit clear in the PTE which is used to translate write-type references. The event is reported to the Ebox in one of the three ways described above: via the MD file or PA queue fault flags, or via an immediate microtrap for explicit Ebox writes.

  Unlike other memory management exceptions, which are dispatched to the operating system, M=0 faults are completely processed by the Ebox microcode handler. For normal instructions, the handler causes the Ibox to back out all GPR modifications that are in the RLOG and retrieves the PC from the PC queue. For string instructions, any RLOG entries that belong to the string instructions are not processed, and PSL<FPD> is set. Using the PTE address supplied by the Mbox, the Ebox microcode reads the PTE, sets the M-bit, and writes the PTE back to memory. The instruction stream is then restarted at the interrupted instruction (which may result in special FPD handling, as described below).

5.3.3.5 Translation Buffer Miss

Translation buffer misses are handled by the Mbox transparently to the rest of the CPU. When a reference misses in the translation buffer, the Mbox aborts the current reference and invokes the services of the memory management exception sequencer in the Mbox, which fetches the appropriate PTE from memory and loads it into the translation buffer. The original reference is then restarted.

5.3.3.6 Reserved Addressing Mode Faults

Reserved addressing mode faults are detected by the Ibox for certain illegal combinations of specifier addressing modes and registers. When one of these combinations is detected, the Ibox sets a global addressing mode fault flag that indicates that the condition was detected and stops.

If the Ibox global addressing mode fault flag is set when the Ebox references an invalid source queue entry, a fault flag is injected into either the Ebox or Fbox pipelines, depending on the type of instruction. To avoid a deadlock, S3 stalls do not prevent forward prgress of the flag in the pipeline. The fault flag is carried along the Ebox or Fbox pipeline and passed to the RMUX, which reports the event by requesting a microtrap when that source is selected.

If the Ibox global addressing mode fault flag is set when the Ebox microcode microbranches on an invalid field queue entry, a fault flag is injected into the Ebox pipeline. When the flag reaches the S4 segment of the pipeline and is selected by the RMUX, a microtrap is requested.

Similarly, if the Ibox global addressing mode fault flag is set when the RMUX, in response to a request by the Ebox or Fbox, references an invalid destination queue entry, a microtrap is requested.

5.3.3.7 Reserved Operand Faults

Reserved operand faults for floating point operands are detected by the Fbox, and reported in the same manner as the floating point arithmetic exceptions described above.

Other reserved operand faults are detected by Ebox microcode as part of macroinstruction execution flows and are reported by jumping directly to the fault handler.

5.3.3.8 Exceptions Occurring as the Consequence of an Instruction

Opcode-specific exceptions such as reserved instruction faults, breakpoint faults, etc., are dispatched directly to handlers by placing the address of the handler in the instruction PLA for each instruction.

Other instruction-related faults, such as privileged instruction faults, are detected in execution flows by the Ebox microcode and are reported by jumping directly to the fault handler.

For testability, the Fbox may be disabled. If this is the case, integer multiply instructions are executed by the Ebox microcode and floating point instructions are converted into reserved instruction faults for emulation by software. When the first Ebox microinstruction of an Fbox operand flow for a floating point macroinstruction reaches the S4 segment of the pipeline, a microtrap is requested. The handler for this microtrap then jumps directly to the reserved instruction fault handler.

5.3.3.9 Trace Fault

Trace faults are detected by the microsequencer with some help from the Ebox. The microsequencer maintains a duplicate copy of PSL<TP>, which it updates as required to track the state of the PSL copy as it would exist when the instruction is executed by the Ebox. At the end of a macroinstruction, the microsequencer logically ORs its local copy of the TP bit with PSL<TP>. If either is set, the microsequencer substitutes the address of the microcode trace fault handler for the address of the next macroinstruction.

5.3.3.10 Conditional Branch Mispredict

When the Ibox decodes a conditional branch, it predicts the path that the branch will take and places its prediction into the branch queue. When the Ebox reaches the instruction, it evaluates the actual path that the branch took and compares it in the S5 segment of the Ebox pipeline with the Ibox prediction. If the two are different, the Ibox is notified that the branch was mispredicted and a microtrap request is made to abort the Ebox and Fbox pipelines. The Ibox flushes itself, backs out any GPR modifications that are in the RLOG, and redirects the instruction stream to the alternate path. The Ebox microcode handler for this event cleans up certain machine state and waits for the first instruction from the alternate path.

5.3.3.11 First Part Done Handling

During the execution of one of the 8 string instructions that are implemented by the CPU, an exception or an interrupt may be detected. In that event, the Ebox microcode saves all state necessary to resume the instruction in the GPRs, backs up PC to point to the opcode of the string instruction, sets PSL<FPD> in the saved PSL, and dispatches to the handler for the interrupt or exception.

When the interrupt or exception is resolved, the software handler terminates with an REI back to the instruction. When the Ibox decodes an instruction with PSL<FPD> set, it stops parsing the instruction immediately after the opcode. In particular, it does not parse the specifiers. When the microsequencer finds PSL<FPD> set at a macroinstruction boundary, it substitutes the address of a special FPD handler for the instruction execution flow.

The FPD handler determines which instruction is being resumed from the opcode, unpacks the state saved in the GPRs, clears PSL<FPD>, advances PC to the end of the string instruction (by adding the opcode PC to the length of the instruction, which was part of the saved state), and jumps back to the middle of the interrupted instruction.

5.3.3.12 Cache and Memory Hardware Errors

Cache and memory hardware errors are detected by the Mbox or Cbox, depending on the type of error. If the error is recoverable (e.g., a Pcache tag parity error on a write simply disables the Pcache), it is reported via a soft error interrupt request and is dispatched to the operating system.

In some instances, write errors that are not recoverable by hardware are reported via a hard error interrupt request, which results in the invocation of the operating system.

Read errors that are not recoverable by hardware are reported via the assertion of a soft error interrupt, and also in a manner that is similar to that used for memory management exceptions, as described above. In fact, the MD file, PA queue, and the Ibox all contain a hardware error flag in parallel with the memory management fault flag. With the exception of TB parity errors, which cause an immediate microtrap request, the event is reported to the Ebox in exactly the same way as the equivalent memory management exception would be, but the microcode exception handler is different. For example, an unrecoverable error on a specifier read would set the hardware error flag in the MD file. When the flag is referenced, the error flag is injected into the pipeline. When the flag advances to the S4 segment and is selected by the RMUX, it causes a microtrap request which invokes a hardware error handler rather than a memory management handler.

Note that certain other errors are reported in the same way. For example, if the memory management sequencer in the Mbox receives an unrecoverable error trying to read a PTE necessary to translate a destination specifier, it sets the hardware error flag in the PA queue for the entry corresponding to the specifier. This results in a microtrap to the hardware error handler when the entry is referenced. PTE read errors for read references are also reported via the original reference.

5.4 Revision History

Table 5–1: Revision History

Who	When	Description of change
Mike Uhler	06-Mar-1989	Release for external review.
Gil Wolrich	15-Nov-1990	Update for NVAX Plus external release.

Chapter 6

Microinstruction Formats

6.1 Ebox Microcode

The NVAX Plus microword consists of 61 bits divided into two major sections. Bits <60:15> control the Ebox Data Path and are encoded into two formats. Bits <14:0> control the Microsequencer and are also encoded into two formats.

6.1.1 Data Path Control

The Data Path Control Microword specifies all the information needed to control the Ebox Data Path. The two formats, Standard and Special, are selected by bit <60>, the FORMAT bit. In addition, bit <45>, the LIT bit, selects the constant generation format of the microword, which may be either an 8-bit constant or a 10-bit constant, depending on a decode in the MISC field. Pictures of the microword formats are in Figure 6–1 and Figure 6–2. A brief description of each field is given in Table 6–1 and Table 6–2.

Figure 6–1: Ebox Data Path Control, Standard Format

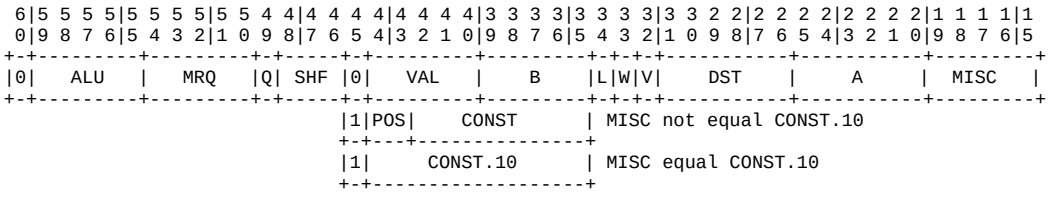


Table 6–1: EBOX Data Path Control Microword Fields, Standard Format

Bit Position	Microword Field	Microword Format	Description
60	FORMAT	—	Microword format–Standard or Special
59:55	ALU	Both	ALU function select

Table 6–1 (Cont.): EBOX Data Path Control Microword Fields, Standard Format

Bit Position	Microword Field	Microword Format	Description
54:50	MRQ	Both	Mbox request select
49	Q	Standard	Q register load control
48:46	SHF	Standard	Shifter function select
45	LIT	Both	ALU/shifter B port control–register or literal
44:40	VAL	Standard[1]	Constant shift amount
39:35	B	Both[1]	ALU/shifter B port select
44:43	POS	Both[2]	Constant position
42:35	CONST	Both[2]	8-bit constant value
44:35	CONST.10	Both[3]	10-bit constant value
34	L	Both	Length control
33	W	Both	Wbus driver control
32	V	Both	VA write enable
31:26	DST	Both	WBUS destination select
25:20	A	Both	ALU/shifter A port select
19:15	MISC	Both	Miscellaneious function select, group 0

[1]NOT Constant generation microword variant

[2]8-Bit Constant generation microword variant, when MISC field not equal CONST.10

[3]10-Bit Constant generation microword variant, when MISC field equal CONST.10

Figure 6–2: Ebox Data Path Control, Special Format

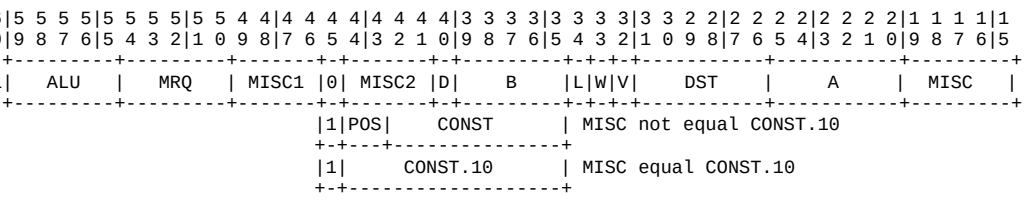


Table 6–2: EBOX Data Path Control Microword Fields, Special Format

Bit Position	Microword Field	Microword Format	Description
60	FORMAT	—	Microword format–Standard or Special

Table 6–2 (Cont.): EBOX Data Path Control Microword Fields, Special Format

Bit Position	Microword Field	Microword Format	Description
59:55	ALU	Both	ALU function select
54:50	MRQ	Both	Mbox request select
49:46	MISC1	Special	Miscellaneous function select, group 1
45	LIT	Both	ALU/shifter B port control–register or literal
44:41	MISC2	Special[1]	Miscellaneous function select, group 2
40	DISABLE.RETIRE	Special[1]	Instruction retire disable
39:35	B	Both[1]	ALU/shifter B port select
44:43	POS	Both[2]	Constant position
42:35	CONST	Both[2]	8-bit constant value
44:35	CONST.10	Both[3]	10-bit constant value
34	L	Both	Length control
33	W	Both	Wbus driver control
32	V	Both	VA write enable
31:26	DST	Both	WBUS destination select
25:20	A	Both	ALU/shifter A port select
19:15	MISC	Both	Miscellaneious function select, group 0

[1]NOT Constant generation microword variant

[2]8-Bit Constant generation microword variant, when MISC field not equal CONST.10

[3]10-Bit Constant generation microword variant, when MISC field equal CONST.10

6.1.2 Microsequencer Control

The Microsequencer Control Microword supplies the information necessary for the Microsequencer to calculate the address of the next microinstruction. The basic computation done by the Microsequencer involves selecting a base address from one of several sources, and then optionally modifying three bits of the base address to get the final next address.

Bit <14>, SEQ.FMT, selects between Jump and Branch formats. Figure 6–3 and Figure 6–4 show the two formats. Table 6–3 and Table 6–4 describe each of the fields.

Figure 6–3: Ebox Microsequencer Control, Jump Format

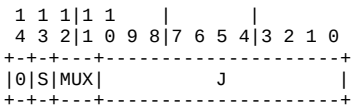


Table 6–3: Ebox Microsequencer Control Microword Fields, Jump Format

Bit Position	Microword Field	Microword Format	Description
14	SEQ.FMT	—	Microsequencer format–Jump or Branch
13	SEQ.CALL	Both	Subroutine call
12:11	SEQ.MUX	Jump	Next address select
10:0	J	Jump	Next address

Figure 6–4: Ebox Microsequencer Control, Branch Format

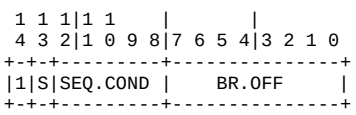


Table 6–4: Ebox Microsequencer Control Microword Fields, Branch Format

Bit Position	Microword Field	Microword Format	Description
14	SEQ.FMT	—	Microsequencer format–Jump or Branch
13	SEQ.CALL	Both	Subroutine call
12:8	SEQ.COND	Branch	Microbranch condition select
7:0	BR.OFF	Branch	Page offset of next address

6.2 Ibox CSU Microcode

The Ibox complex specifier unit is controlled by a 29-bit microword, as shown in Figure 6–5. A brief description of each field is given in Table 6–5.

Figure 6–5: Ibox CSU Format

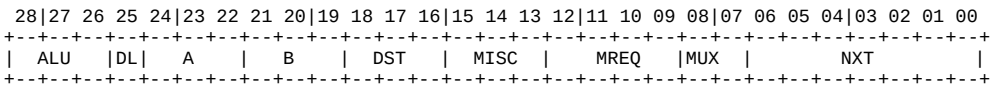


Table 6–5: Ibox CSU Microword Fields

Bit Position	Microword Field	Description
28:26	ALU	ALU function select
25	DL	Data length control
24:22	A	ALU A port select
21:19	B	ALU B port select
18:16	DST	Wbus destination
15:13	MISC	Miscellaneous function select
12:9	MREQ	Mbox request select
8:7	MUX_CNT	Next address mux select
6:0	NXT	Next address

6.3 Revision History

Table 6–6: Revision History

Who	When	Description of change
Debra Bernstein	06-Mar-1989	Release for external review.
Mike Uhler	13-Dec-1989	Update for second-pass release.

Chapter 7

The Ibox

7.1 Overview

The NVAX Plus IBOX chapter includes the overview description, IPR specifications, and description of IBOX testabilty features from the NVAX CPU Chip Specification. For detailed and complete IBOX specification refer to the NVAX CPU Chip Specification.

7.1.1 Introduction

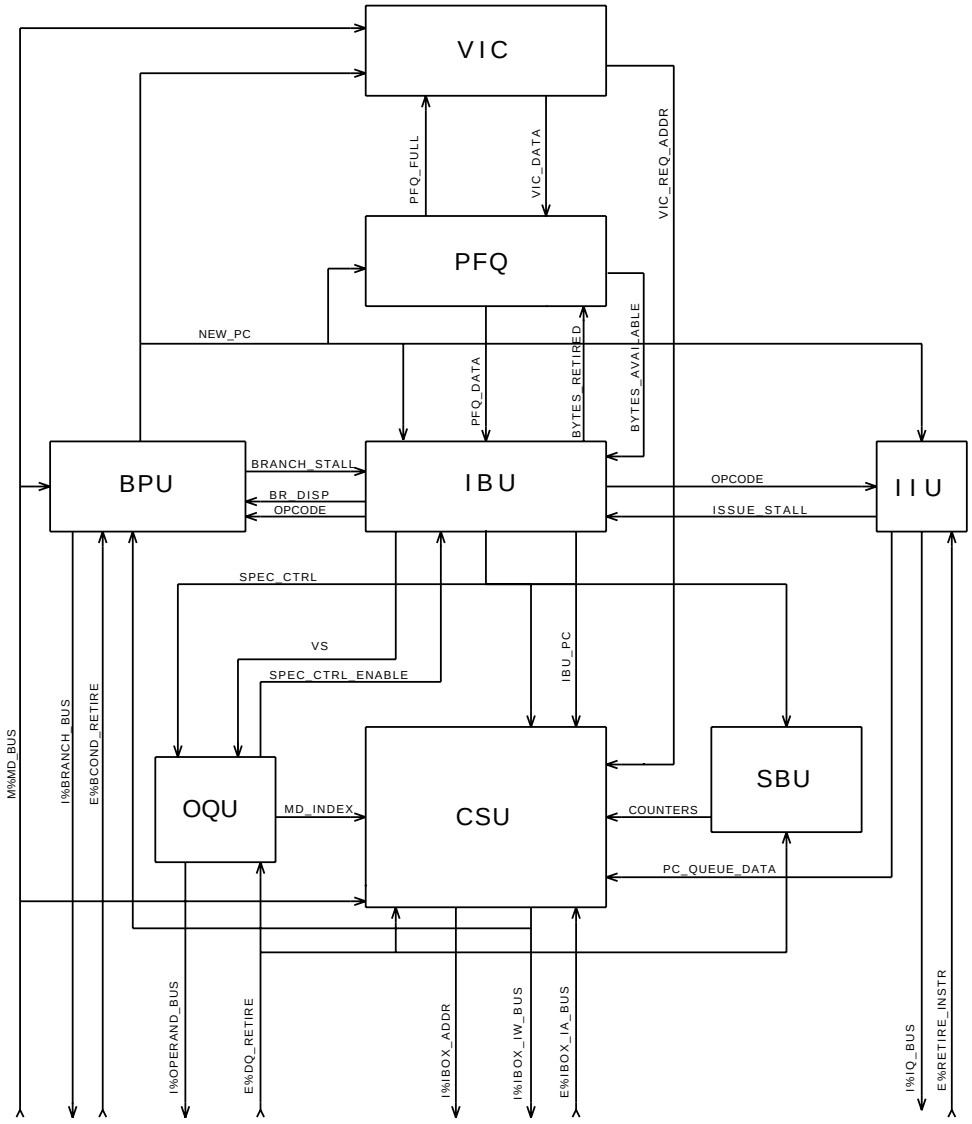
This chapter describes the Ibox section of the NVAX Plus CPU chip. The 4-stage Ibox pipeline (S0..S3) runs semi-autonomously to the rest of the NVAX Plus CPU and supports the following functions:

- **Instruction Stream Prefetching**
  The Ibox attempts to maintain sufficient instruction stream data to decode the next instruction or operand specifier.
- **Instruction Parsing**
  The Ibox identifies the instruction opcodes and operand specifiers, and extracts the information necessary for further processing.
- **Operand Specifier Processing**
  The Ibox processes the operand specifiers, initiates the required memory references, and provides the Ebox with the information necessary to access the instruction's operands.
- **Branch Prediction**
  Upon identification of a branch opcode, the Ibox hardware predicts the direction of the branch (taken vs. not taken). For branch taken predictions, the Ibox redirects the instruction prefetching and parsing logic to the branch destination, where instruction processing resumes.

Figure 7–1 is a top level block diagram of the Ibox showing the major Ibox sub-sections and their inter-connections.

This chapter presents a high-level description of the Ibox functions, then provides details of the Ibox sub-sections which support each function.

Figure 7–1: Ibox Block Diagram



7.1.2 Functional Overview

The Ibox fetches, parses, and processes the instruction stream, attempting to maintain a constant supply of parsed VAX instructions available to the Ebox for execution. The pipelined nature of the NVAX Plus CPU allows for multiple macroinstructions to reside within the CPU at various stages of execution. The Ibox, running semi-autonomously to the Ebox, parses the macroinstructions following the instruction that is currently in Ebox execution. Performance gains are realized when the time required for instruction parsing in the Ibox is hidden during the Ebox execution of

an earlier instruction. The Ibox places the information generated while parsing ahead into Ebox queues.

The Instruction Queue contains instruction specific information which includes the instruction opcode, a floating point instruction flag, and an entry point for the Ebox microcode.

The Source Queue contains information about the source operands for the instructions in the instruction queue. Source queue entries contain either the actual operand (as in a short literal), or a pointer to the location of the operand.

The Destination Queue contains information required for the Ebox to select the location for execution results storage. The two possible locations are the VAX General Purpose Registers (GPRs) and memory.

These queues allow the Ibox to work in parallel with the Ebox. As the Ebox consumes the entries in the queues, the Ibox parses ahead adding more. In the ideal case, the Ibox would stay far enough ahead of the Ebox such that the Ebox would never have to stall because of an empty queue.

The Ibox needs access to memory for instruction and operand data. Instruction and operand data requests are made through a common port to the Mbox. All data for both the Ibox and the Ebox is returned on a shared **M%MD_BUS**

The Ibox port feeds Mbox queues to smooth memory request traffic over time. The Specifier Request Latch holds Ibox requests for operand data. The Instruction Request Latch holds Ibox requests for instruction stream data. These 2 latches allow the Ibox to issue memory requests for both instruction and operand data even though the Mbox may be processing other requests.

The Ibox supports 4 main functions:

1. Instruction Stream Prefetching
2. Instruction Parsing
3. Operand Specifier Processing
4. Branch Prediction

Instruction Stream Prefetching works to provides a steady source of instruction stream data for instruction parsing. While the instruction parsing logic works on one instruction, the instruction prefetching logic fetches several instructions ahead.

The Instruction Parsing logic parses the incoming instruction stream, identifying and pre-processing each of the instruction's components. The instruction opcodes and associated information are passed directly into the Ebox instruction queue. Operand specifier information is passed on to the operand specifier processing logic.

The Operand Specifier Processing logic locates the operands in registers, in memory, or in the Instruction Stream. This logic places operand information in the Ebox source and destination queues, and makes the required operand memory requests.

The Ibox does not have prior knowledge of branch direction for brnaches which rely on Ebox condition codes. The Branch prediction logic makes a prediction on which way the branch will go and forces the Ibox to take that path. This logic saves the program counter of the alternate branch path, so that in the event that Ebox branch execution shows that the prediction was wrong, the Ibox can be redirected to the correct branch direction.

7.2 VIC Control and Error Registers

The VIC contains 4 internal processor registers (IPRs) which provide VIC control and read/write access to the arrays.

MACROCODE RESTRICTION

VIC_ENABLE must be cleared before writing to the VIC IPRs: VMAR, VDATA, or VTAG. **VIC_ENABLE** must be cleared before reading from VIC IPRs: VDATA, VTAG. In functional operation, an REI must preceed the MTPR which enables the VIC.

See Section 7.4 for details of the IPR mechanism.

Figure 7–2: VMAR Register

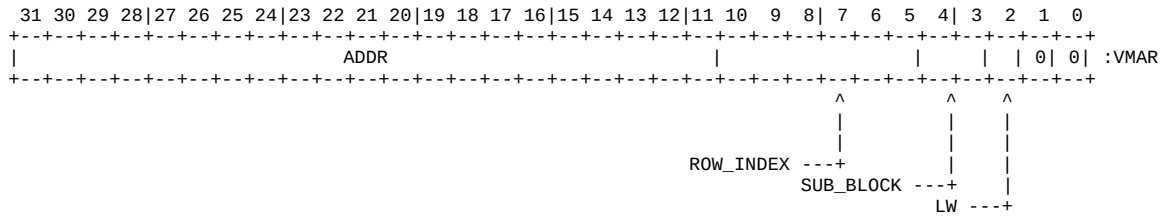


Table 7–1: VMAR Register

Name	Bit(s)	Type	Description
LW	2	WO	Longword select bit. Selects longword of sub-block for access to cache array
SUB_BLOCK	4:3	RW	Sub-block select. Selects data sub-block for access to cache array, also latches **VIBA** on VIC parity errors
ROW_INDEX	10:5	RW	Row select. Row index for read and write access to cache array, also latches **VIBA** on VIC parity errors
ADDR	31:11	RO	Error address field. Latches tag portion of **VIBA** on VIC parity errors

When the VIC is disabled, the VIC Memory Address Register (**VMAR**) may be used as an index for direct IPR access to the cache arrays. **VMAR** supply the cache row index, **VMAR** supply the cache sub-block, and **VMAR** indicates the longword within a quadword address.

VMAR also latches and holds the **VIBA** on VIC array parity errors.

Figure 7–3: VTAG Register

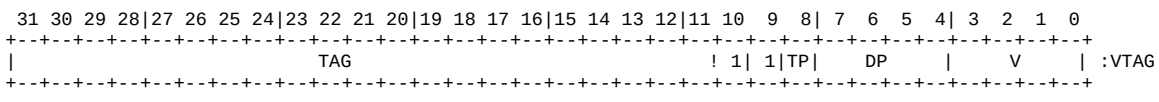


Table 7–2: VTAG Register

Name	Bit(s)	Type	Description
V	3:0	RW	Data valid bits. Supply data valid bits on array read/writes
DP	7:4	RW	Data parity bits. Supply data parity on array read/writes
TP	8	RW	Tag parity bit. Supplies tag parity on tag array read/writes
TAG	31:11	RW	Tag. Supplies tag on tag array read/writes

The **VTAG** IPR provides read and write access to the cache tag array. An IPR write to **VTAG** will write the contents of the **M%MD_BUS** to the tag, parity, and valid bits for the row indexed by **VMAR**. **VTAG** are written to the cache tag. **VTAG** is written to the associated tag parity bit. **VTAG** are used to write the four data parity bits associated with the indexed cache row. Similarly **VTAG** write the four data valid bits associated with the cache row. **DP** and **V** are the data parity and data valid bits, respectively, for the 4 quadwords of data in the same row. **DP** and **V** correspond to the quadword of data addressed when address bits 4:3 = 00, **DP** and **V** correspond to the quadword of data addressed when address bits 4:3 = 01, etc.

Figure 7–4: VDATA Register

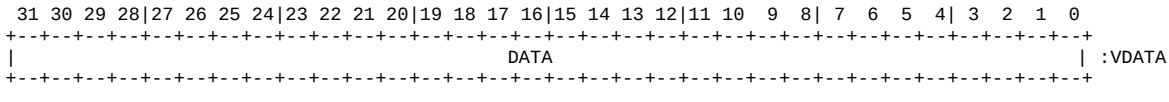


Table 7–3: VDATA Register

Name	Bit(s)	Type	Description
DATA	31:0	RW	Data for data array reads and writes

The **VDATA** IPR provides read and write access to the cache data array. When **VDATA** is written, the cache data array entry indexed by **VMAR** is written with the IPR data. Since the IPR data is a longword, two accesses to **VDATA** are required to read or write a quadword cache sub-block.

Writes to **VDATA** with **VMAR** = 0 simply accumulate the IPR data destined for the low longword of a sub-block in **FILL_DATA**. A subsequent write to **VDATA** with **VMAR** = 1 directs the the IPR data to **FILL_DATA**, and triggers a cache write sequence to the sub-block indexed by **VMAR**.

Reads to **VDATA** with **VMAR** = 0 trigger a cache read sequence to the sub-block indexed by **VMAR**. The low longword of the a sub-block is returned as IPR read data. A read of **VDATA** with **VMAR** = 1 returns the high longword of the sub-block as IPR data.

Figure 7–5: ICSR Register

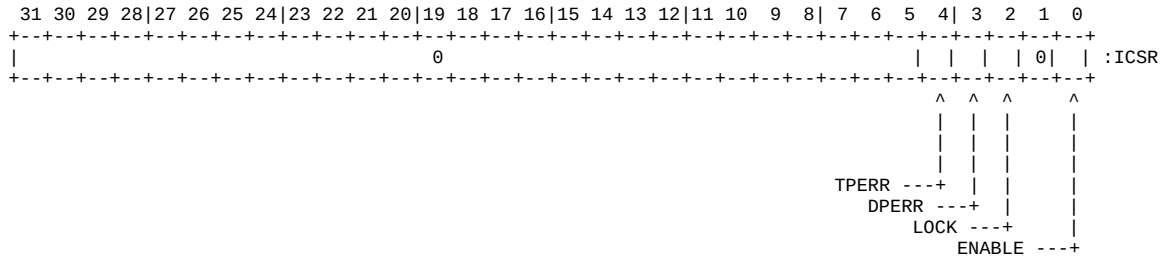


Table 7–4: ICSR Register

Name	Bit(s)	Type	Description
ENABLE	0	RW,0	Enable Bit. When set, allows cache access to the VIC. Initializes to 0 on RESET.
LOCK	2	WC	Lock Bit. When set, validates and prevents further modification of the error status bits in the ICSR and the error address in the VMAR register. When clear, indicates no VIC parity error has been recorded and allows ICSR and VMAR to be updated.
DPERR	3	RO	Data Error Bit. When set, indicates data parity error occurred in data array if Lock Bit also set.
TPERR	4	RO	Tag Error Bit. When set, indicates tag parity error occurred in tag array if Lock Bit also set.

The **ICSR** IPR provides control and status functions for the Ibox. VIC tag and data parity errors are latched in the read-only **ICSR**, respectively. **ICSR** is set when a tag or data parity error occurs and keeps the error status bits and the **VMAR** register from being modified further. Writing a logic one to **ICSR** clears the **LOCK** bit and allows the error status to be updated. When **ICSR** is clear, the values in **ICSR** are meaningless. When **ICSR** is set, a VIC parity error has occurred, and either **ICSR** or **ICSR** will be set indicating that the parity error was either a tag parity error or a data parity error, repectively. **ICSR** cannot be cleared from software. **ICSR** provides IPR control of the VIC enable. It is cleared on **RESET**.

7.3 VIC Performance Monitoring Hardware

Hardware exists in the Ibox VIC to support the NVAX Performance Monitoring Facility. See Chapter 16 for a global description of this facility.

The VIC hardware generates two signals **I%PMUX0** and **I%PMUX1** which are driven to the central performance monitoring hardware residing in the Ebox. These two signals are used to supply VIC hit rate data to the performance monitoring counters.

I%PMUX0 is asserted the cycle when a VIC read reference is first attempted while the prefetch queue is not full. **I%PMUX1** signals the hit status for this event in the same cycle.

The data is captured only on the first read reference that could be used by the PFQ to avoid skewed hit ratios caused by multiple hits or misses to the same reference while the prefetch queue is full or the VIC is waiting for a cache fill.

7.4 Ibox IPR Transactions

The Ebox microcode communicates with the Ibox in part through internal processor registers (IPRs). The IPR reads are handled by CSU microcode. The IPR write control is distributed, however the description is included here for completeness.

Ebox microcode conventions guarantee that the Ibox is idle before initiating Ibox IPR transactions. This is accomplished either by the knowledge that the current Ebox microcode flow takes place in a macroinstruction with an drain Ibox assist or by asserting an explicit **E%STOP_IBOX** command. The only exception involve the issuing of an IPR transaction when the CSU is involved in an RLOG unwind operation. In this case the unwind finishes in the CSU, then the CSU processes the latched IPR command. If the RLOG is empty when the microcode initiates an unwind, 0 will be added to whatever GPR is pointed to by the read pointers.

MICROCODE RESTRICTION

E%IBOX_LOAD_PC and **E%IBOX_IPR_WRITE** must not occur in the same cycle.

7.4.1 IPR Reads

The Ebox signifies an IPR read by asserting the **E%IBOX_IPR_READ** strobe, the **E%IBOX_IPR_NUM**, and the **E%IBOX_IPR_INDEX**. This information is latched in the S1 logic stage, and an IPR request flag is posted. The S1 next address logic responds by creating an IPR dispatch to an IPR microaddress in the utility page of microcode, and by clearing the IPR request flag. All Ibox logic blocks associated with IPR reads examine the **E%IBOX_IPR_NUM**. If the IPR source is within a section, that section prepares to drive the IPR read data onto the **VIC_REQ_ADDR**. The microcode at the common IPR routine reads the **VIC_REQ_ADDR**, passes the value through the ALU, and writes the data to an Ebox working register located at the **E%IBOX_IPR_INDEX** offset in the register array. The **VIC_REQ_ADDR** is used for IPR read data source simply because it is a convenient 32-bit bus that runs through the entire section.

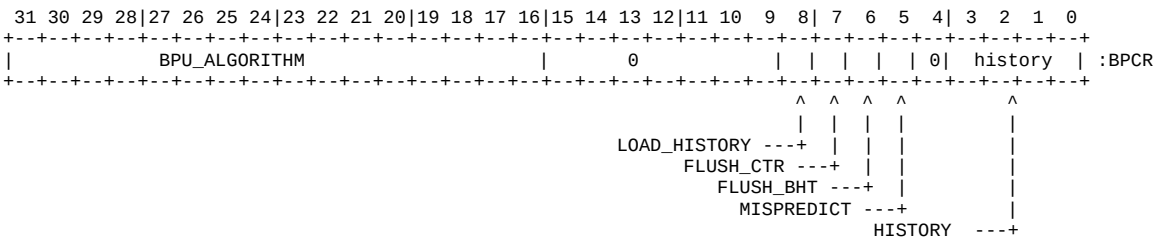
7.4.2 IPR Writes

The Ebox signifies an IPR write by asserting the **E%IBOX_IPR_WRITE** strobe and the **E%IBOX_IPR_NUM**. All Ibox logic blocks associated with IPR writes examine the **E%IBOX_IPR_NUM**. If the IPR destination is within a section, that section prepares to accept the IPR write data from the **M%MD_BUS**. The Mbox drives the **M%MD_BUS** with IPR data and asserts **M%IBOX_IPR_WR** to complete the transaction.

7.5 Branch Prediction IPR Register

The **BPCR** IPR provides control for the BPU and read/write access to the history array. The write-only **BPCR** bit causes a BPU branch history table flush. The flush is identical to the context switch flush, which resets all branch table entries to a neutral value: history bits = 0100. The write-only **BPCR** bit causes the **BRANCH_TABLE_COUNTER** to be cleared. The **BRANCH_TABLE_COUNTER** provides an address into the branch table for IPR read and write accesses. Each IPR read from the **BPCR** or write to the **BPCR** with **BPCR** = 1 increments the counter. This allows IPR branch table reads and writes to step through the branch table array. **BPCR** enables writes to the branch history table. A write to the **BPCR** field with **BPCR** = 1 causes a BPU branch history table write. The history bits for the entry indexed by the counter is written with the IPR data. **BPCR** reads supply the history bits in **BPCR** for the entry indexed by the counter. **BPCR** will return a "1" if the last conditional branch mispredicted. **BPCR** contain the branch prediction algorithm. Any IPR write to the **BPCR** will update the algorithm. An IPR read will return the value of the current algorithm. For example, a "0" in **BPCR** means that the next branch encountered will not be taken if the history is "0000". A "1" in **BPCR** means that the next branch encountered when the prior history is "0101" will be taken.

Figure 7–6: BPCR Register



The microcode will write the following bit pattern as part of the powerup sequence:

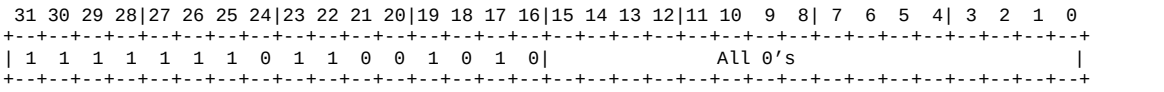


Table 7–5: BPCR Register

Name	Bit(s)	Type	Description
HISTORY	3:0	RW	Branch history table entry history bits.
MISPREDICT	5	RO	Indicates if last conditional branch mispredicted.
FLUSH_BHT	6	WO	Write of a 1 resets all history table entries to a neutral value, hardware clears bit.

Table 7–5 (Cont.): BPCR Register

Name	Bit(s)	Type	Description
FLUSH_CTR	7	WO	Write of a 1 resets BPCR address counter to 0, hardware clears bit.
LOAD_HISTORY	8	WO	Write history array addressed by BPCR address counter.
BPU_ALGORITHM	31:16	RW	Controls direction of branch for given history.

Bits 8,7,6 are defined in Table 7–6 for IPR writes to the BPCR. NOTE: The prediction algorithm will be updated on every IPR write to the BPCR.

Table 7–6: BPCR <8:6>

BIT 8	BIT 7	BIT 6	Write Action
0	0	0	Do nothing, except update algorithm
0	0	1	Flush branch table. History not written
0	1	0	Address counter reset to 0. History not written
0	1	1	Flush branch table, reset address counter, history not written
1	0	0	Write history to table, counter automatically increments
1	0	1	Undefined: Branch table flushed, new history written, counter incremented
1	1	0	Undefined: Write history to old counter value, counter reset to 0
1	1	1	Undefined: Branch table flushed, write history to old counter value, counter reset to 0

7.6 Testability

7.6.1 Overview

Ibox testability is enhanced by architectural features, and connection to the internal scan register and the parallel port.

7.6.2 Internal Scan Register and Data Reducer

Ibox hardware state may be latched and shifted off-chip through the global internal scan register. See Chapter 17 for the implementation details of the internal scan register. State included on the internal scan register for chip debug is TBD.

An Ibox linear feedback shift register (LFSR) is part of the internal scan chain. The register is an observation only structure which can be loaded in parallel or loaded in parallel with feedback, acting like a data reducer. The contents may be shifted out serial through the internal scan register. Table 7–7 lists the signals that are contained in the Ibox LFSR.

Table 7–7: Ibox Scan Chain Fields

Field Name	# bits	Description
STOP_PARSER	2	Stop parser and status flags
SPEC_CTRL	21	spec_ctrl bits <21:13> and <11:0>
E_DL	2	Data length for instruction (DL of last operand)

7.6.3 Parallel Port

The CSU microcode address is routed to the chip parallel port. The microcode address can be monitered on a cycle by cycle basis during chip debug by selecting the Ibox as source to the parallel port. When selected, a buffered version of the control store address, **MUX_H**, appears on **PP_DATA**. See Chapter 17 for the implementation details of the parallel port.

7.6.4 Architectural Features

Internal processor registers are included as architectural features to aid in testability. IPR access to VIC tags and data is available throught the VTAG and VDATA registers. See Section 7.2 for the implementation details of the these registers. IPR access to the branch history table and branch status is available throught the BPCR register. See Section 7.5 for the implementation details of the BPCR.

7.6.5 Metal 3 Nodes

Various Ibox nodes are brought up to minimum size CMOS-4, metal-3 test pads for chip debug. State included on the internal scan register for chip debug is TBD.

7.6.6 Issues

Internal scan register states in the Ibox for chip debug are TBD.

Nodes elevated to metal-3 test pads in the Ibox for chip debug are TBD.

7.7 Performance Monitoring Hardware

7.7.1 Signals

The Ibox provides two signals for performance monitoring: **I%PM_VIC_ACC_H** and **I%PM_VIC_HIT**. These signals enable the Ebox performance monitoring hardware to gather statistics on VIC hits versus VIC accesses.

7.8 Revision History

Table 7–8: Revision History

Who	When	Description of change
Shawn Persels	06-Oct-1988	Initial release.
John F. Brown	19-Dec-1988	Partial Update.
John F. Brown, Paul Gronowski, Jeanne McKinley	06-Mar-1989	Release for external review.
John F. Brown, Ruben Castelino, Mary Field, Paul Gronowski, Jeanne Meyer	12-Jan-1990	Intermediate release.
Gil Wolrich	15-Nov-1990	Retain Overview, IBOX IPRs, and Testability sections for NVAX Plus external release.

Chapter 8

The Ebox

8.1 Chapter Overview

The NVAX Plus EBOX chapter includes the overview description, IPR specifications, and description of EBOX testabilty features from the NVAX CPU Chip Specification. modifications required for Vector Support.

For detailed and complete EBOX specification refer to the NVAX CPU Chip Specification.

8.2 Introduction

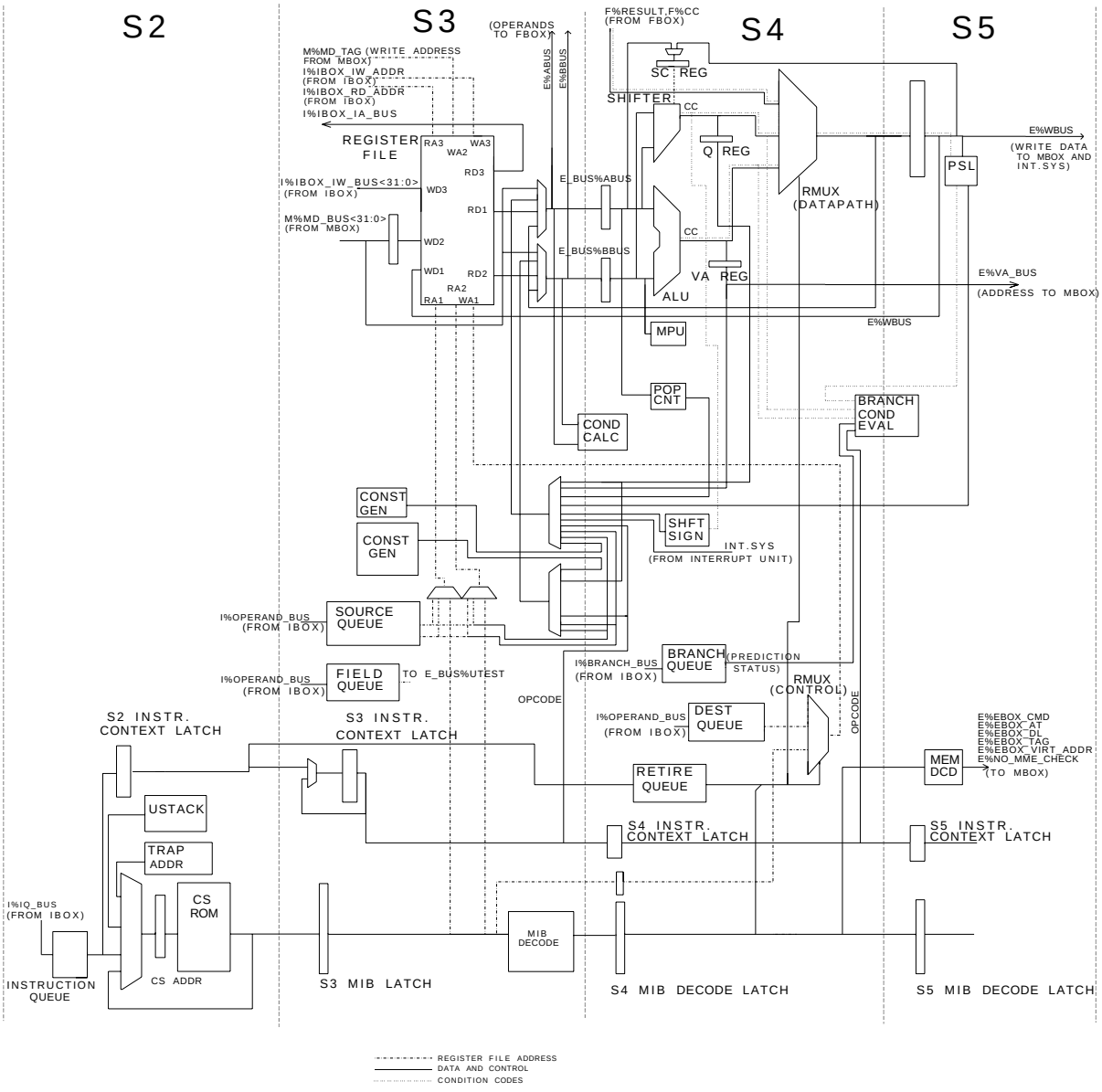
The Ebox is the instruction execution unit in the NVAX CPU chip. It is a 3 stage pipeline (S3..S5) which runs semi-autonomously to the rest of the NVAX Plus chip and supports the following functions:

- **Instruction Execution**
  The Ebox is responsible for carrying out the execution portion of each VAX instruction under control of a microflow whose initial address is provided by the Ibox issue unit.
- **Instruction Coordination**
  The Ebox is a major source of control to coordinate instruction processing in the Ibox, Mbox, and Fbox. It ensures that Ebox and Fbox macroinstructions retire in the proper order, and it provides controls to the Mbox and Ibox which help manage certain inter-macroinstruction dependencies. The Ebox cooperates with the Ibox in handling mispredicted branches.
- **Trap, Fault and Exception Handling**
  The Ebox coordinates trap, fault, and interrupt handling. It delays the condition until all preceding macroinstructions complete properly. It then collects information about the condition and ensures that the correct architectural state is reached.
- **CPU Control**
  Most CPU control is provided by the Ebox. Ebox control functions include CPU initialization, controlling Ibox, Fbox, and Mbox activities, and setting control bits during major CPU state changes (e.g. taking an interrupt or executing a change mode instruction).

The Ebox accomplishes many of the above functions by executing the NVAX Ebox microcode. This chapter views the Ebox as the interpreter of microcode. Describing how microcode functions are used to correctly emulate the VAX architecture or the architectural motivation for Ebox hardware functions is generally outside the scope of this discussion.

Figure 8–1 at the end of this section is a top level block diagram of the Ebox showing all the major Ebox function units, their interconnections, and their place in the pipeline. The pipeline segments are shown in the diagram (S2, S3, S4, and S5). The sections following the diagram describe the function elements depicted and the Ebox pipeline.

Figure 8–1: Ebox Block Diagram



8.3 Ebox Overview

8.3.1 Microword Fields

The Ebox is controlled by the data path control portion of the microword, which is either standard or special format. The other portion of the control word, the microsequencer control portion, controls the microsequencer which determines which microword is fetched in every cycle. The fields of the data path control portion of the microword and their effect within the Ebox are shown in Table 8–1. For more information on microword formats and field widths see Chapter 6.

NOTATION

The notation FIELD/FUNCTION is used throughout this chapter to mean that microword field FIELD specifies FUNCTION.

Table 8–1: Data Path Control Microword Fields

Microword Field	Microword Format	Description
FORMAT	Both	This one-bit field determines whether the microword is in the special format. If it is 1, the MISC1, MISC2, and D fields exist. If it is 0, the Q, SHF, and VAL fields exist instead.
LIT	Both	This one-bit field determines whether the microword is the constant generation variant (format). If it is 1, the POS and CONST fields exist. If it is 0, the VAL and B fields exist instead in standard format, and the MISC2, D, and B fields exist instead in special format.
ALU	Both	Sets the ALU function, including typical ALU operations, and others.
MRQ	Both	Controls initiation of Ebox memory accesses,VECTOR MEMORY ACCESSES, and other Mbox control functions. The Ebox decodes the field and sends the corresponding request to the Mbox.
SHF	Standard	Sets the shifter function. The W and Q fields control how the shifter output is used. Some settings of this field specify a pass operation instead of a shift.
VAL	Standard[1]	Specifies the shift amount (1 to 31) or, if VAL = 0, specifies to shift the amount in the SC register.
A	Both	Specifies the source of **E_BUS%ABUS** for this microword. The A field can select any element in the register file or one of several of Ebox sources. **E_BUS%ABUS** is one of the two sources for the ALU and the shifter.
B	Both[1]	When the source of **E_BUS%BBUS** is a register this field specifies the source of **E_BUS%BBUS**. The B field can select from some of the elements in the register file or from a small number of other Ebox sources. **E_BUS%BBUS** is one of the two sources for the ALU and the shifter.
POS	Both[2]	When the source of **E_BUS%BBUS** is from the constant generator this field specifies which byte the constant value is in. Bytes 0 through 3 may be specified. The other bytes are forced to 0.

[1]Not constant generation microword variant.

[2]Constant generation microword variant.

Table 8–1 (Cont.): Data Path Control Microword Fields

Microword Field	Microword Format	Description
CONST	Both[2]	This field contains the literal byte value which is sourced to one of the bytes of **E_BUS%BBUS** as specified by the POS field. (The other **E_BUS%BBUS** bytes are forced to 0.)
CONST.10[3]	Both[2]	This field contains the literal 10-bit value which is sourced to **E_BUS%BBUS**. (**E_BUS%BBUS** are forced to 0.)
DST	Both	This field specifies the destination of **E%WBUS**. The possible destinations include a subset of the register file and a number of other Ebox destinations.
Q	Standard	Controls whether or not the Q register is loaded with the shifter output for this microword.
W	Both	Selects the driver of **E%WBUS**. Either the ALU or the shifter output is driven on **E%WBUS**.
L	Both	This field controls whether the Ebox operations are done with a data length of longword or the length specified in the DL register. The Ebox operations affected are condition code calculation, size of memory operations, zero extending of **E%WBUS** data, and bytes affected by register file writes.
V	Both	Controls updating of the VA register. Either the VA register is updated with the value from the ALU, or it is not changed from its previous value.
MISC	Both	This field has many uses. Only one use can be selected at a time. This field can control PSL condition code alterations, set the DL register, set or clear state flags, or invoke a box coordination or control function.
MISC1	Special	This field can specify one of a few Ibox or Fbox coordination or control functions, and can be used to set or clear state flags.
MISC2	Special[1]	One Mbox control function and one to add an Fbox destination scoreboard entry.
DISABLE.RETIRE	Special[1]	This field is used to disable retire of macroinstructions and retire queue entries

[1]Not constant generation microword variant.

[2]Constant generation microword variant.

[3]The CONST.10 field is actually the POS field bitwise concatenated with the CONST field, with the POS field in the more significant position. It is simply a way of treating these two microword fields as one. CONST.10 is only used when MISC/CONST.10.BIT is specified.

When a microword field is not present in all formats, it defaults to NOP (no operation) when a microword format without that field occurs. More specifically, standard format microwords effectively specify MISC1/NOP, MISC2/NOP, and DISABLE.RETIRE/NO by default. Special format microwords effectively specify Q/HOLD.Q, SHF/NOP, and VAL/0. When the microword is the constant generation variant of the standard format microword, VAL/0 is effectively specified, and the B field is ignored since this microword variant sources a constant onto **E_BUS%BBUS**. In the constant generation variant of the special format microword, MISC2/NOP and DISABLE.RETIRE/NO are effectively specified, and the B field is ignored because this microword variant also sources a constant onto **E_BUS%BBUS**.

8.3.1.1 Microsequencer Control Fields

In addition to decoding the datapath control portion of the microword, the Ebox decodes a part of the Microsequencer control portion of the microword. Specifically, it detects when the SEQ.FMT and SEQ.MUX fields (see Chapter 9 and Chapter 6) specify LAST.CYCLE or LAST.CYCLE.OVERFLOW. The Ebox fault detection logic and the RMUX control logic use these decodes.

8.3.2 The Register File

The register file contains four kinds of registers: MD (memory data), GPR, Wn (working), and CPUSTATE registers. The MD registers receive data from memory reads initiated by the Ibox, and from direct writes from the Ibox. The Wn registers hold microcode temporary data. They can receive data from memory reads initiated by the Ebox and receive result data from ALU, shifter, or Fbox operations, and from the Ibox in the case of Ibox IPR reads. The GPRs are the VAX architecture general-purpose registers (though R15 is not in the file) and can receive data from Ebox initiated memory reads, from the ALU or shifter, or from the Ibox. The CPUSTATE registers hold semipermanent architectural state (e.g. KSP, SCBB). They can only be written by the Ebox.

8.3.3 ALU and Shifter

Each microword specifies source operands for the ALU or shifter (A, B, POS, and CONST fields), operations for these function units to perform (ALU, SHF, and VAL fields), and a destination (or possibly two destinations if Q or VA is updated) for the result(s) (DST, Q, W, and V fields). Note that in special format microwords no shifter operation can be specified and the Q register can't be altered. In the course of executing the microword, the Ebox will fetch the source operands onto **E_BUS%ABUS** and **E_BUS%BBUS**, carry out the specified ALU and shifter functions, and store the result in the specified locations (if any).

8.3.3.1 Sources of ALU and Shifter Operands

In general the sources of **E_BUS%ABUS** and **E_BUS%BBUS** (the inputs to the ALU and shifter) are either a constant, a register from the register file, an Ebox register (e.g. PSL, Q, or VA), an Ebox source value calculated by a special function unit, a hardware status provided via a special path from outside the Ebox (e.g., interrupt status), or an entry from the source queue. **E_BUS%BBUS** sources are limited to a subset of the register file, certain Ebox registers, and an entry from the source queue. The source queue is introduced in Section 8.3.4.

8.3.3.2 ALU Functions

The ALU is capable of standard operations on byte, word, and longword size operands. It can pass either input to the output and is capable of a number of arithmetic and logical operations on one or two operands, producing condition codes based on data length and operation.

8.3.3.3 Shifter Functions

The shifter does longword and quadword shift operations and certain pass-thru operations, always producing a longword output. The shifter treats the two sources as a single quadword, with **E_BUS%ABUS** as the more significant longword. The longword output is this quadword shifted right 0 to 32 bits and truncated to longword length. The shifter produces condition codes based the longword output data.

8.3.3.4 Destinations of ALU and Shifter Results

The output of the shifter and the output of the ALU can drive **E%WBUS**. The shifter output is also directly connected to the Q register so that the Q register can be loaded with the shifter output regardless of the source of **E%WBUS**. In the same way, the ALU output is directly connected to the VA register. **E%WBUS** data is the input to one of the write ports on the register file and can be used to update any register file entry except an MD register. Certain other Ebox registers (e.g. SC, PSL) can be loaded from **E%WBUS**.

The destination of **E%WBUS** can be specified by the current destination queue entry, when the microword so specifies. The destination queue is introduced in the following section.

8.3.4 Ibox-Ebox Interface

The Ibox-Ebox interface is made up of a number of FIFO queues. The purpose of these queues is to allow the Ibox to fetch and decode new instructions before the Ebox is ready to execute them. The Ibox adds entries as it decodes instructions, and the Ebox removes them from the other end as it executes them. For each opcode, there is a predetermined number of entries added to the various queues by the Ibox. Ebox execution microflows remove exactly the right number of entries from each queue.

The queues which interface the Ibox to the Ebox directly are the source queue, the destination queue, the branch queue, and the field queue. The instruction queue, the PA queue, and the retire queue are introduced here for completeness.

The source queue holds source operand information. Entries are added by the Ibox as it decodes the source type operand specifiers of each instruction. The entry is either a pointer into the register file or the data from a literal mode operand specifier. The Ebox accesses and removes an entry each time a microword specifies a source queue access in either the A or B fields. If the entry is literal data, it is used as an ALU and/or a shifter operand. Otherwise the register file is accessed using the pointer in the entry.

The destination queue holds result destination information. Entries are added by the Ibox as it decodes the destination type operand specifiers of each instruction. A destination queue entry is either a pointer to a GPR in the register file or a flag indicating that the result destination is memory. The Ebox accesses and removes an entry each time a microword specifies a destination queue access in the DST field or the Fbox supplies a result which specifies a destination queue access. If the entry is a pointer to a GPR, the Ebox writes the ALU, shifter, or Fbox data into the register file. Otherwise the data is stored in memory at the address found in the PA queue.

The PA queue is in the Mbox. Each time the Ibox adds an entry indicating a memory destination to the destination queue it also sends the Mbox a virtual address to be translated. When the Mbox has translated the address it puts it in the PA queue. If the current destination queue entry indicates a memory destination, the Ebox sends the result data to the Mbox to be written to the physical address found in the PA queue. The Mbox removes the PA queue entry as it uses it.

The branch queue holds status bits for each branch instruction processed by the Ibox. The Ibox adds an entry to the branch queue each time it finishes processing a conditional or unconditional branch. The Ebox references and removes the current branch queue entry in the execution microflow for the branch. This allows the Ebox to synchronize with the Ibox so that the branch does not finish executing until the Ibox has successfully fetched the branch displacement specifier. It also allows the Ebox to check for an incorrect branch prediction by the Ibox.

Each time the Ibox decodes a branch it calculates the branch address. For unconditional branches it simply begins fetching from the new instruction stream immediately. For conditional branches the Ibox predicts whether the branch will be taken or not. The branch queue entry added by the Ibox indicates the branch prediction. When the Ebox executes an unconditional branch, it references the branch queue simply to ensure that the Ibox was able to fetch the displacement specifier without a fault or error. For conditional branches the Ebox also checks that the branch prediction was correct and initiates a microtrap if it wasn't. If the branch wasn't correct, the Ebox notifies the Ibox, which uses the alternate path PC (which it had kept) to begin fetching along the correct path.

The retire queue holds status for each macroinstruction currently being executed in the Ebox or the Fbox. The status indicates which unit will execute the instruction, the Ebox or the Fbox. The Ebox adds an entry each time the Microsequencer dispatches to a macroinstruction execution microflow. The Ebox references the retire queue when the macroinstruction execution is complete in order to ensure that instructions finish executing in the proper order. A certain amount of concurrent execution in the Fbox and Ebox is possible. The retire queue is used to prevent one box from altering any architecturally visible state before the other box's execution for preceding macroinstructions finishes. The Ebox references and removes a retire queue entry each time an Fbox or Ebox instruction is retired.

The field queue holds a one-bit type status for variable-length bit field base address operands processed in the Ibox. (Note that some operands are treated as variable-length bit field base address operands internally by the NVAX CPU even though the operand is not really the base address of a variable-length bit field. These operands, including the true bit field base address operands, are collectively referred to as field operands.) The field queue entry indicates whether the field operand was register mode. The Ibox adds an entry when it processes operands which it knows by context require an entry. The Ebox retires an entry after it has used the information in a microcode conditional branch. Very different execution microflows are required for some instructions, particularly bit field instructions, depending on whether a particular operand is register mode or specifies a memory address. In the latter case the information sent by the Ibox is a memory address, while in the first case the source and destination queue entries point to the register in the register file.

The instruction queue is part of the Ibox-Microsequencer interface. It holds information derived from the VAX instruction opcode. The Ibox adds an entry as it decodes each instruction. An entry contains the opcode, data length, the microcode dispatch address for execution, and a flag indicating whether the macroinstruction is for the Fbox. The Microsequencer references and removes an entry at the start of execution of each VAX instruction. It uses the dispatch address to fetch the first microword of the macroinstruction execution microflow. At the same time it passes the opcode, data length, and the Fbox execution flag to the Ebox. The Ebox adds an entry to the retire queue at that time. That entry is simply the Fbox execution flag (except if the Fbox is disabled.

8.3.5 Other Registers and States

The Ebox contains several special purpose registers, the SC, VA, and Q registers, and the PSL.

The SC register holds a shift count for use in some shift operations.

The VA register can hold a virtual address or a microcode temporary value. The VA register is directly readable by the Mbox and is the address source for all Ebox initiated memory operations. The VA register is loaded directly from the ALU output.

The PSL is the VAX architecture program status longword register. It is loaded from **E%WBUS** and can be used as a source operand by the ALU or shifter. Its bits are used in many places in the Ebox and elsewhere in the CPU where required by the VAX architecture.

The Q register is loaded from the output of the shifter. It holds shifter results for later use.

8.3.6 Ebox Memory Access

Through the mechanism of the source queue and the destination queue, the Ibox initiates most memory accesses for the Ebox. In certain cases the Ebox must carry out memory accesses on its own. The MRQ field of the microword specifies the Mbox operation. The virtual or physical address is provided from the VA register. If the VA is being updated in this microword, the address is bypassed directly from the output of the ALU. For writes, the data is taken from **E%WBUS**, so it can be the output of the shifter or the ALU. For reads, the DST field of the microword specifies the register file entry which is to receive the data. This register must be a GPR or a working register.

8.3.7 CPU Control Functions

Most control functions are invoked through one of the MISC fields, but some of the MRQ field functions are Mbox control functions or miscellaneous control functions rather than memory access commands. The control functions generally act to reset a function unit (Fbox, Ibox, or Mbox), synchronize Ebox operation with a function unit, or restart semiautonomous operation of the Mbox or Ibox when either of them has stopped for some reason.

8.3.8 Ebox Pipeline

Execution of microwords in the Ebox is pipelined with three pipe stages (S3..S5). These stages are shown in Figure 8–1. In the first stage (S3), the **E_BUS%ABUS** and **E_BUS%BBUS** sources are fetched or prepared. In the second (S4) the ALU and shifter operate on the data. In the third (S5) the result is written into the register file or to some other destination. Stages S3 and S4 can stall for various reasons. Stage S5 cannot stall. Once a particular microword's execution has advanced into S5, it is going to complete. Various stalls occur in S4 in order to ensure that a particular microword's effects do not change any architectually visible state (e.g., GPRs, PSL) before proper completion without memory management faults is guaranteed.

The Microsequencer fetches the microword and delivers it to the Ebox in S3. If the Ebox's S3 stage is stalled, the Microsequencer's S2 activity is stalled as well. See Chapter 9 for more detail.

Even though the operand fetch, function execution, and result store take place in different cycles, the microword specifies the operation as if it all took place in one cycle. The Ebox has bypass paths which allow a microword to use a register as a source even it it is updated by one of the two preceding microwords. For example, if the immediately preceding microword updates W1 in the register file and the current microword specifies W1 as a source to the ALU, the Ebox hardware detects the condition and muxes the data into the staging latch before the ALU at the same time as it forwards the data to the latch which sources **E%WBUS** in stage S5.

Bypass paths are only implemented where performance considerations warrant. Also bypassing isn't the solution to every problem pipelining introduces. For example, after the PSL is updated the microcode allows 2 cycles before a microword specifying SEQ.MUX/LAST.CYCLE or SEQ.MUX/LAST.CYCLE.OVERFLOW because the PSL is not actually updated until S5. The Microsequencer uses the FPD, T, and TP bits in the PSL to determine the proper new microflow

dispatch. It would make the decision based on old PSL information if the microcode didn't allow the 2 cycles.

One place where the effect of pipelining is particularly apparent is in microcode conditional branches. For example, a microcode branch based on **E_BUS%BBUS** data must immediately follow the microword which sources the relevant data onto **E_BUS%BBUS**. Similarly, a microcode branch based on the ALU condition codes must be the second microword after the one which specified the ALU operation. See Chapter 9 for more detail on microcode branches.

8.3.9 Pipeline Stalls

The Ebox pipeline is controlled by the stall and fault logic. This function unit supplies stall signals which are used to gate clocking of control and data latches in each stage. It also controls insertion of effective no-ops into S4 when S3 is stalled and into S5 when S4 is stalled.

The Ebox pipeline stalls in S3 when it is accessing a source operand in the register file or the source queue which is not valid. Many register file entries have a valid bit associated with them. A register file entry is not valid, and its valid bit is not set, if a memory read has been initiated for that entry and hasn't yet completed. A source queue entry is not valid if the Ibox hasn't added that entry yet.

The Ebox stalls in S4 if the current destination queue entry is not valid and the microword in S4 references a destination queue entry. A destination queue entry is not valid if the Ibox hasn't added that entry yet.

The Ebox stalls in S4 if the current destination queue entry is valid but specifies a memory destination for the data and the current PA queue entry is not valid. A PA queue entry is not valid if the Mbox hasn't added that entry yet.

The Ebox stalls in S4 if the microword in S4 requests a memory operation and the Mbox is already working on an Ebox initiated memory operation (that is, the previous request is still in the EM_LATCH).

The Ebox stalls in S4 if the microword in S4 synchronizes with the branch queue and the branch queue entry is not valid. A branch queue entry is not valid if the Ibox hasn't added that entry yet.

The Ebox stalls in S4 if the current retire queue entry specifies that an Fbox instruction must retire before the instruction associated with the microword in S4 and the Ebox is requesting the use of the RMUX to store result data. (The Ebox requests the use of the RMUX if the microword in S4 specifies anything other than NONE in the DST field.)

If the Ebox stalls in S3, the S4 and S5 stages of the pipeline can continue execution. If S4 doesn't stall when S3 does, then an effective no-op is inserted into S4 after the current S4 operation advances into S5. The no-op is necessary so that the stalled S3 microword isn't advanced to S4 and S5 while an S3 stall is in effect.

If the Ebox stalls in S4 then S3 stalls as well. (Microwords can't pass each other in the pipeline.) During S4 stalls, an effective no-op is inserted into S5 after the operation in S5 completes. This is necessary so that the operation in S4 isn't advanced into S5 while an S4 stall is in effect.

In any cycle that the Ibox has not made a microstore dispatch address available to the Microsequencer and a dispatch is needed (i.e., during the last cycle of any microflow), the microsequencer fetches the STALL microword. This microword specifies no Ebox operation and can't cause a stall anywhere in the pipeline (although it does specify SEQ.MUX/LAST.CYCLE). This allows the microwords already in the pipeline to continue even when the Ibox is temporarily unable to supply new instruction execution dispatches. See Chapter 9 for more detail.

A microcode loop which repeatedly accesses the field queue until the current field queue entry becomes valid is also very much like a stall, though the stall logic is not actually involved. This condition is referred to as a field queue stall. In this situation, the Ebox pipeline advances in each cycle (unless the microword in S4 is stalled also). However, the same microword is fetched out of the control store in every cycle. In typical microcode usage of the field queue conditional branch, this microword will not alter any state in S4 or S5.

8.3.10 Microtraps, Exceptions, and Interrupts

The Ebox and Microsequencer together coordinate the handling of exceptions and interrupts. Most interrupts and some exceptions are handled by Microsequencer dispatching to a microcode exception handler routine at the end of the current VAX instruction. These dispatches do not affect the execution of microwords already in the pipeline. Other exceptions cause a microtrap. In a microtrap the Microsequencer signals the Ebox to cause stages S3, S4, and S5 of the Ebox control pipeline to be flushed. It also signals the Ebox to flush the retire queue. (Flushing of the other Ibox-to-Ebox queues, the Fbox pipeline, and the specifier queue in the Mbox is done by microcode, except in the case of a branch misprediction.) At the same time the Microsequencer fetches a new microword from a special dispatch address in the control store based on the particular microtrap condition. This microflow handles any other necessary state flushing. Because a microtrap affects microwords already in the pipeline, the Ebox delays handling most traps until the microword which incurred the fault has reached S4. The microtrap is taken at the time that microword would normally have entered S5. In certain cases, Ebox stalls delay a microtrap until the stall is ended. The purpose of this is to ensure that operations which are part of a preceding VAX instruction are allowed to complete properly.

Most of the microtraps which the Ebox delays until S4 are due to Ibox-initiated memory operations which had an access or translation fault. Faults due to Ibox-initiated reads are detected by the Ebox when it accesses a valid MD register from the register file, and the fault bit associated with that MD is set. Each MD register has a fault bit which is set by the Ibox or the Mbox when a fault occurs in the memory reads necessary to fetch the source data. When the Ebox accesses an MD register with its fault bit set in S3, it carries that fault status down the pipeline into S4.

All faults detected in S3 are piped to S4 before they cause a microtrap. Faults detected in S4 or piped to S4 will cause a microtrap only if the Ebox is next to retire a macroinstruction. Otherwise they are delayed until the Fbox retires an instruction and the retire queue entry indicates the Ebox.

Fault status signals are sent by the Ibox for entries in the instruction queue, source queue, field queue, destination queue, and branch queue. Entries in the PA queue have fault bits. The Ebox detects a fault when it accesses a PA queue entry with its fault bit set or when it finds the instruction queue, source queue, field queue, destination queue, or branch queue empty and one of the fault status signals from the Ibox asserted. In the case of the instruction queue, the fault is detected in S2 and carried into S3 only when there is no S3 stall. In the case of the source queue and field queue, the faults are detected in S3. Instruction queue, source queue, and field queue

related faults are carried down the pipeline until they reach S4, where they cause a microtrap once the Ebox is next to retire a macroinstruction.

Faults encountered in Ebox-initiated memory operations cause the Microsequencer to trap immediately. Ebox memory accesses begin in S5 so these traps cannot affect microwords from preceding VAX instructions. It is up to microcode to make sure that the last Ebox memory access has completed properly before the Microsequencer dispatches to another VAX instruction execution microflow.

Hardware errors are essentially handled in the same way as faults.

8.3.11 Ebox IPRs

The CPUSTATE registers contained in the Register File are used by the microcode to hold elements of architectural state. They are read and written only by the EBOX. There are 10 CPUSTATE registers: KSP, ESP, SSP, USP, ISP, ASTLVL, SCBB, PCBB, SAVEPC, and SAVEPSL. Also the Ebox implements two IPRs. They are IPRs 124-125 (decimal), PCSCR and ECR.

ECR is a possible source of **E_BUS%ABUS**, accessed by specifying ECR in the A field of the microword. ECR and PCSCR are also possible destinations of **E%WBUS**, written by specifying PCSCR or ECR in the DST field of the microword. On writes, the entire register is written, regardless of the current DL value.

8.3.11.1 IPR 124, Patchable Control Store Control Register

The PCSCR is used to load control store patches. Chapter 9 describes the patchable control store function in detail. Figure 8–2 and Table 8–2 show the bit fields and give descriptions.

Figure 8–2: PCS Control Register, PCSCR

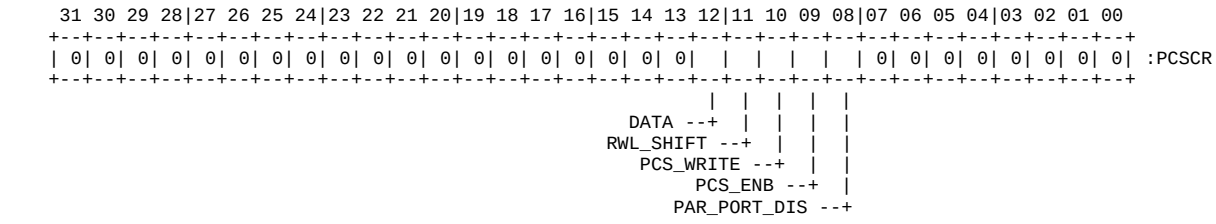


Table 8–2: PCSCR Field Descriptions

Name	Bit(s)	Type	Description
PAR_PORT_DIS	8	WO,0	Writing a 1 disables control by the testability parallel port of the section of the internal scan used in loading the control store CAM (content addressable memory) and RAM. This is necessary when using this register to load the control store CAM and RAM.
PCS_ENB	9	WO,0	Enables the control store CAM and RAM so that patches are fetched and supersede the control store ROM.
PCS_WRITE	10	WO	The event of writing a 1 to this bit causes the PCS scan chain contents to be written into the control store CAM and RAM. The control signal which enables the write returns to the inactive state automatically; there is no need for software to write a 0 to this bit after writing a 1.
RWL_SHIFT	11	WO	The event of writing a 1 to this bit causes the PCS scan chain to shift by one. The control signal which enables the shift returns to the inactive state automatically; there is no need for software to write a 0 to this bit after writing a 1.
DATA	12	W0	This bit holds the data which is shifted into the PCS scan chain when a 1 is written to RWL_SHIFT. By repeatedly setting DATA and writing a 1 to RWL_SHIFT, software can shift any data pattern into the PCS scan chain.

8.3.11.2 IPR 125, Ebox Control Register

The ECR is used to configure certain Ebox functions. Figure 8–3 and Table 8–3 show the bit fields and give descriptions.

Figure 8–3: Ebox Control Register, ECR

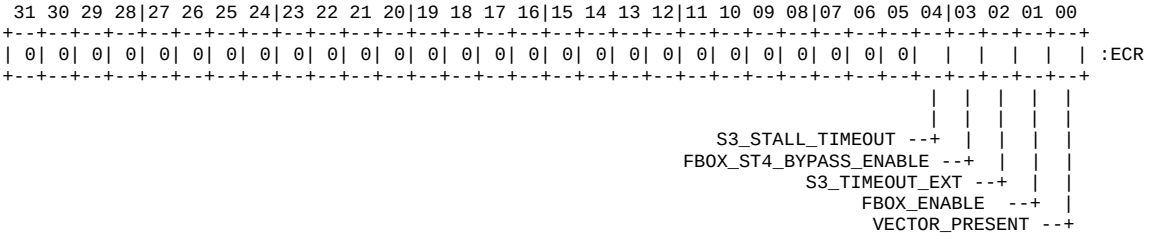


Table 8–3: ECR Field Descriptions

Name	Bit(s)	Type	Description
VECTOR_PRESENT	0	RW,0	This bit is for vector unit support in a future version of this chip.
FBOX_ENABLE	1	RW,0	This bit is set by configuration code to enable the Fbox.
S3_TIMEOUT_EXT	2	RW,0	This bit is set by configuration code to select an external time-base for the S3 stall timeout timer.
FBOX_ST4_BYPASS_ ENABLE	3	RW,0	This bit is set by configuration code to enable Fbox Stage 4 bypass.
S3_STALL_TIMEOUT	4	WC	This bit indicates that an S3 stall timeout occurred. Writing it with 1 clears it.

NOTE

THE SUBSET INTERVAL TIMER FUNCTIONALITY IS REMOVED FROM NVAX Plus.

8.3.12 Initialization

The main mechanism for Ebox initialization is the power-up microtrap, and the MISC/RESET.CPU which occurs in the first microword of this microtrap flow. When this trap occurs, the Microsequencer will assert **E_USQ%PE_ABORT**, aborting the Ebox pipeline as it does for any microtrap. None of the registers in the register file or elsewhere in the Ebox are cleared on initialization, except that IPR bits are cleared where indicated by the bit type (see Section 8.3.11). The state flags are also cleared by reset.

The Ebox asserts **E%STOP_IBOX**, **E%FLUSH_EBOX**, **E%FLUSH_MBOX**, and **E%FLUSH_FBOX** during reset. This is the same effect as MISC/RESET.CPU. See the sections on initialization for each of the boxes for more detail.

8.3.13 Testability

This section describes the testability features in the Ebox.

8.3.13.1 Parallel Port Test Features

The following signals can be observed on the parallel test port.

- **E%S3_STALL**
- **E%S4_STALL**
- **E%RMUX_S4_STALL**
- Ebox retire queue output
- **E_USQ%PE_ABORT**

The following control functions are available on the parallel test port.

- **Force source queue stall**
  Forces a source queue stall in any microword which accesses the source queue regardless of the actual number of entries in the queue.
- **Force destination queue stall**
  Forces a destination queue stall in any microword which accesses the destination queue regardless of the actual number of entries in the queue.
- **Force branch queue stall**
  Forces a branch queue stall in any microword which accesses the branch queue regardless of the actual number of entries in the queue.

8.3.13.2 Observe Scan

A number of signals in the Ebox are readable using the internal scan chain. Most of these are control signals.

This is a list of the signals on the scan chain. They all are connected for observe only.

- **E%WBUS** LFSR.
- The EM bus outputs.
- The significant stall result signals and enough of the precursors to allow determination of which stall is in effect.
- The significant fault results and **E_USQ%PE_ABORT**.
- The bus **E_USQ%UTEST**.

8.3.13.3 E%WBUS<31:0> LFSR

E%WBUS has an LFSR (linear feedback shift register) accumulator. Its output can be scanned out via the observe scan chain. It can be reset to zero by TBS control.

ISSUE

The control to clear **E%WBUS** LFSR will be specified when the testability strategy is settled.

8.3.14 Revision History

Table 8–4: Revision History

Who	When	Description of change
John Edmondson	30-NOV-1988	Initial Release.
John Edmondson	19-DEC-1988	Corrections and Updates.
John Edmondson	06-MAR-1989	Release for external review.
John Edmondson	29-NOV-1989	Updates after external review and modeling complete.
John Edmondson	18-DEC-1989	Further updates, particularly adding real signal names.

Table 8–4 (Cont.): Revision History

Who	When	Description of change
John Edmondson	31-JAN-1990	Updates reflecting minor implementation motivated changes - rev 0.5.
John Edmondson	4-MAY-1990	Updates reflecting minor implementation motivated changes - post rev 0.5.
Gil Wolrich	15-Nov-1990	EBOX chapter for NVAX Plus external release

Chapter 9

The Microsequencer

9.1 Overview

This chapter includes the microsequencer block diagram and descriptions of major hardware components including the Control Store, Patchable Control Store, and Microtest Bus, and the microsequencer testability features. The Microsequencer chapter of the NVAX CPU Chip Functional Specification should be referred to for complete description of the Microsequencer.

The microsequencer is a microprogrammed finite state machine that controls the three Ebox sections of the NVAX Plus pipeline: S3, S4, and S5. The microsequencer itself resides in the S2 section of the pipeline. It accesses microcode contained in an on-chip control ROM, and microcode patches contained in an on-chip SRAM. Each microword is made up of fields that control all three pipeline stages. A complete microword is issued to S3 each cycle, and the appropriate microword decodes are pipelined forward to S4 and S5 under Ebox control.

Each microword contains a microsequencer control field that specifies the next microinstruction in the microflow. This field may specify an explicit address contained in the microword or direct the microsequencer to accept an address from another source. It also allows the microcode to conditionally branch on various NVAX states.

Frequently used microcode can be made into microsubroutines. When a microsubroutine is called, the return address is pushed onto the microstack. Up to six levels of subroutine nesting are possible.

Stalls, which are transparent to the microcoder, occur when an NVAX resource is unavailable, such as when the ALU requires an operand that has not yet been provided by the Mbox. The microsequencer stalls when S3 of the Ebox is stalled.

Microtraps allow the microcoder to deal with abnormal events that require immediate service. For example, a microtrap is requested on a branch mispredict, when the Ebox branch calculation is different from that predicted by the Ibox for a conditional branch instruction. When a microtrap occurs, the microcode control is transferred to a service microroutine.

9.2 Functional Description

9.2.1 Introduction

The NVAX microsequencer consists of several functional units of logic that are explained in the following sections and illustrated in the block diagram, Figure 9–1.

9.2.2 Control Store

The control store is an on-chip ROM which contains the microcode used to execute macroinstructions and microtraps. It is made up of up to 1600 microwords. These are arranged as 200 entries, each entry consisting of 8 microwords. Each microword is 61 bits long, with bits <14:0> being used to control the microsequencer. The remainder of the microword, bits <60:15>, is used by the Ebox to control S3 through S5. The Ebox also receives bits <14,12:11>, enabling it to recognize the last cycle of a microflow and the validity of the microtest bus select lines.

The control store access is performed during Φ_{34} of S2 and Φ_1 of S3 of the NVAX pipeline. The output of the Current Address Latch, **E_USQ_CAL%CAL_H**, is used to address the control store. Bits <10:4,0> are used to select one of the 200 entries. The eight microwords in the selected entry then enter an eight-way multiplexer, where **E_USQ_CAL%CAL_H** select the final control store output. This structure is used because **E_USQ_CAL%CAL_H** are valid later than bits <10:4,0>, since **E_USQ_CAL%CAL_H** must be OR'd with the microtest bus for a BRANCH format microinstruction.

9.2.2.1 Patchable Control Store

The patchable control store is an on-chip SRAM which contains microcode patches. It consists of up to 20 microwords. It operates in parallel with the control store. The microaddress from the **CAL** is the input to its CAM (Content Addressable Memory). If the address hits in the CAM, the output of the patchable control store is selected as the new microword, rather than the output of the regular control store.

The patchable control store and CAM are precharged in Φ_3 and evaluate in Φ_{41}. The **CAL** output, **E_USQ_CAL%CAL_H**, is used in its entirety as the lookup address in the CAM, as opposed to the 1-of-200 selection followed by the 1-of-8 selection used in the ROM control store.

Entries in the Patchable Control Store and its CAM are written under software control from registers in the Ebox. The CAM is disabled during this operation.

9.2.2.2 Microsequencer Control Field of Microcode

The microsequencer control field of the NVAX microword is used to help select the next microword address. The next address source is explicitly coded in the current microword; there is no concept of sequential next address.
The SEQ.FMT field, bit <14> of the microsequencer control field, selects between the following two formats:

Figure 9–1: Microsequencer Block Diagram

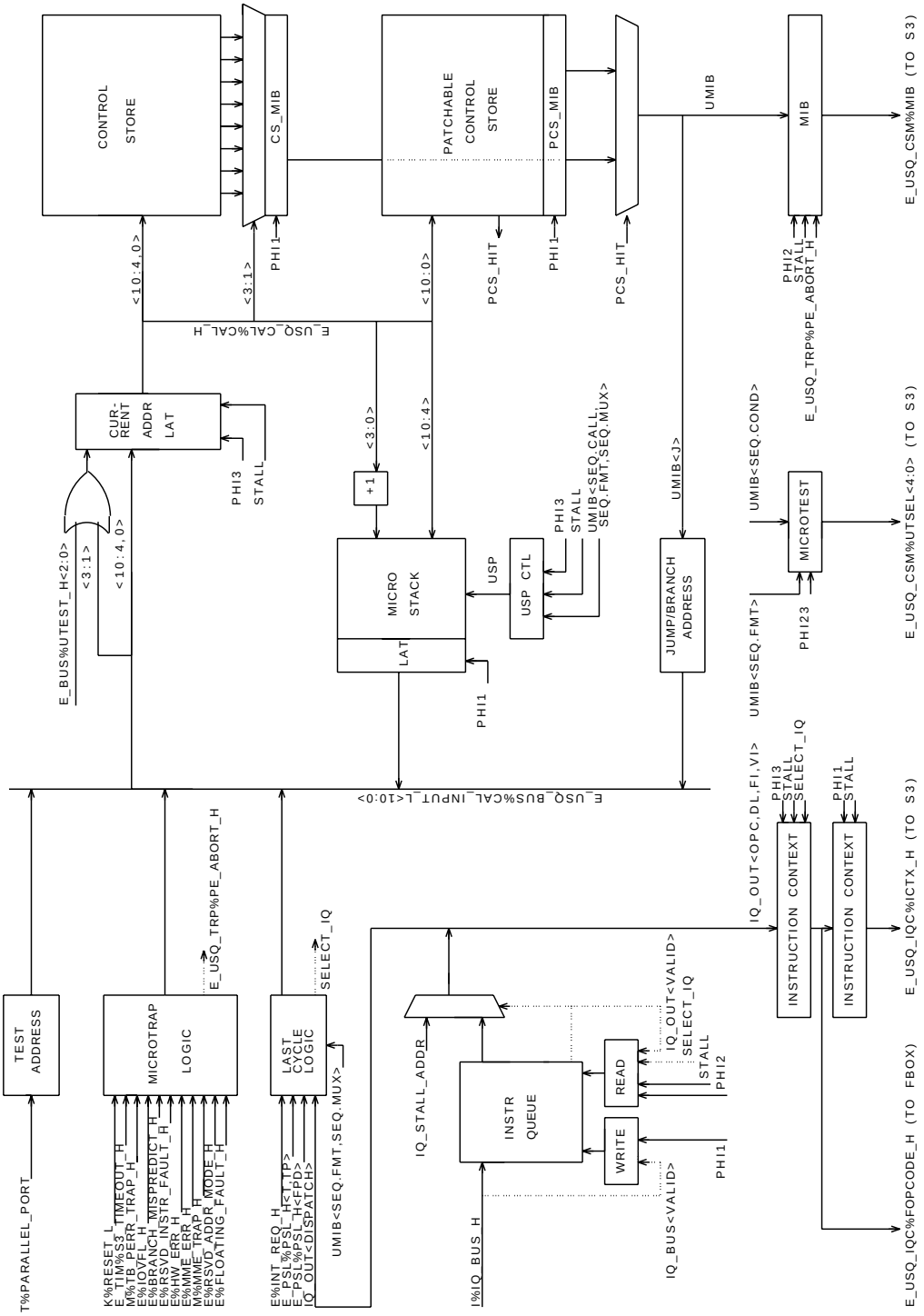


Figure 9–2: Microcode Microsequencer Control Field Formats

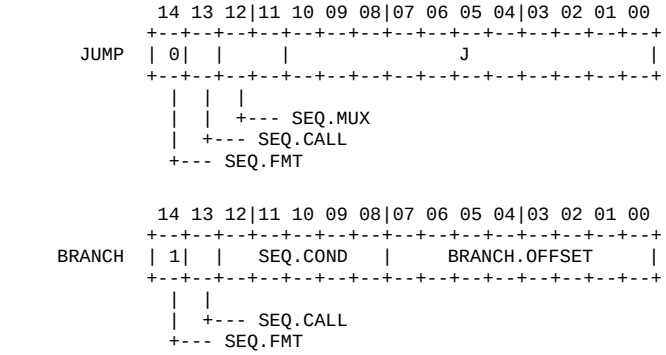


Table 9–1: Jump Format Control Field Definitions

Name	Bit(s)	Description
SEQ.FMT	14	0 for JUMP
SEQ.CALL	13	Controls whether return address is pushed on microstack
SEQ.MUX	12:11	Selects source of next microaddress
J	10:0	JUMP target address

Table 9–2: Branch Format Control Field Definitions

Name	Bit(s)	Description
SEQ.FMT	14	1 for BRANCH
SEQ.CALL	13	Controls whether return address is pushed on microstack
SEQ.COND	12:8	Selects source of Microtest Bus
BRANCH.OFFSET	7:0	Page offset of next microinstruction

9.2.2.3 MIB Latches

The microword output from the Control Store 8-to-1 multiplexer is latched in Φ_1 into the Control Store Microsequencer Microinstruction Buffer (CS_MIB) latch. The microword output from the Patchable Control Store is also latched in Φ_1, into the PCS_MIB latch. The outputs of the CS_MIB and PCS_MIB latches drive a multiplexer, which selects the PCS_MIB output if the **CAL** hit in the Patchable Control Store; otherwise, the multiplexer selects the CS_MIB output.

Bits <14:0> of the multiplexer output (the Microsequencer Microinstruction, **E_USQ_CSM%UMIB_H**) are driven back to the microsequencer; bits <60:14,12:11> are driven to the Microinstruction Buffer (MIB) latch. The MIB latch operates in Φ_2, driving its outputs (**E_USQ%MIB_H**) to S3 of the Ebox. When a microtrap is detected, the contents of this latch are forced to NOP. The MIB latch is stalled on a microsequencer stall.

9.2.3 Next Address Logic

The remainder of the microsequencer is devoted to determining the next control store lookup address. There are five next address sources:

1. JUMP/BRANCH.OFFSET field of Microword
2. Microtrap Logic
3. Last Cycle Logic
4. Microstack
5. Test Address Generator

9.2.3.1 CAL and CAL INPUT BUS

The **CAL**, or Current Address Latch, is a static latch which holds the 11 bit address used to access the control store. It operates in Φ_3, and is stalled on a microsequencer stall. Bits <10:8> are also "stalled" when forming a branch address.

The input to the **CAL** is the **CAL INPUT BUS**. The **CAL INPUT BUS** is a dynamic bus, precharged in Φ_2. The selected next address source drives this bus in Φ_3. Bits <14,12:11> of the microsequencer control field are used in selecting three of the next address sources: **E_USQ_CSM%UMIB_H** (for a BRANCH or JUMP address), the output of the last cycle logic, and the microstack output. The fourth **CAL INPUT BUS** source is the microtrap address; if a microtrap is detected, this input is selected regardless of the value of **E_USQ_CSM%UMIB_H**. The fifth source is a test address, driven from the Test Address Generator. This input has the highest priority. In summary:

Table 9–3: Current Address Selection

TEST ADDR	TRAP DETECTED	SEQ.FMT <14>	SEQ.MUX <12:11>	NEXT ADDRESS SOURCE	REMARKS
0	0	0	00	J	JUMP/CALL microinstructions
0	0	1	XX	Branch Address	BRANCH/CONDITIONAL CALL microinstructions
0	0	0	01	Microstack	RETURN microinstruction
0	0	0	1X	Last Cycle Logic	Start new microflow
0	1	X	XX	Microtrap Logic	Microtrap
1	X	X	XX	Test Address Generator	Test address

9.2.3.1.1 Microtest Bus

The microtest bus allows conditional branches and conditional calls based on Ebox information, such as condition codes. The SEQ.COND field of the BRANCH format is driven on the microtest select lines, **E_USQ%UTSEL_H**, in Φ_{23}. These lines are decoded by all conditional information sources the Ebox, and the selected source drives its information on the microtest bus, **E_BUS%UTEST_H**, in NOT Φ_1. **E_BUS%UTEST_H** must be valid in time to be OR'd with value on the **CAL INPUT BUS** and latched in the **CAL** in Φ_3.

The sources for the microtest bus are as follows:

Table 9–4: Microtest Bus Sources

UTSEL<4:0>	Select	UTEST<2:0>
00	No source	000
01	ALU.NZV[2]	ALU_CC.N,ALU_CC.Z,ALU_CC.V
02	ALU.NZC[2]	ALU_CC.N,ALU_CC.Z,ALU_CC.C
03	B.2-0[1]	EB_BUS<2:0>
04	B.5-3[1]	EB_BUS<5:3>
05	A.7-5[1]	EA_BUS<7:5>
06	A.15-12[1]	EA_BUS<15:14>, EA_BUS<13> OR EA_BUS<12>
07	A31.BQA.BNZ1[1]	EA_BUS<31>, EB_BUS<2:0> = 0, EB_BUS<15:8> NEQ 0
08	MPU.0-6[2]	MPU0_6<2:0>
09	MPU.7-13[2]	MPU7_13<2:0>
0A	STATE.2-0[2]	STATE<2:0>
0B	STATE.5-3[2]	STATE<5:3>
0C	OPCODE.2-0[1]	OPCODE<2:0>
0D	PSL.26-24[3]	PSL<26:24>
0E	PSL.29.23-22[3]	PSL<29>, PSL<23:22>
0F	SHF.NZ[2],INT	SHF_CC.N, SHF_CC.Z, INTERRUPT_REQUEST
10	VECTOR,TEST	ECR<VECTOR_UNIT_PRESENT>[3], TEST DATA, TEST STROBE
11	FBOX	Encoded fault<1:0>, ECR<FBOX.ENABLED> = 0[3]
12	FQ.VR[1]	0, FIELD_QUEUE_NOT_VALID, FIELD_QUEUE_RMODE
13-1F	Not Used	

[1]Data is taken from S3.

[2]Data is taken from S4.

[3]Data is taken from S6.

The microtest select lines are always driven with bits <12:8> of the **CAL** regardless of the microinstruction format. The microtest bus is only OR'd with the **CAL INPUT BUS** if the BRANCH source is selected to drive that bus.

Two of the microtest sources, the Field Queue (FQ) and the Mask Processing Unit (MPU), perform some function based on the value of the microtest select lines. These functions must check SEQ.FMT, **E_USQ%MIB_H**, for validity of the microtest select lines.

The microtest select lines are precharged to a value of zero during Φ_1; no microtest source is selected for this value.

9.2.3.2 Microtrap Logic

Microtraps allow the microcoder to deal with abnormal events that require immediate service. When a microtrap occurs, the microcode control is transferred to a service microroutine. Operations further behind in the pipe than the one which caused the microtrap are aborted.

Microtraps are generated by the Ebox, Mbox, or Ibox. Those Ebox microtrap requests considered faults are asserted in S4 of the microinstruction in which they occurred. Those that are considered traps are asserted in S5 of the microinstruction in which they occurred.

Microtraps have higher priority than all other next address sources except the Test Address Generator. Microtraps are detected in Φ_4. The microtrap signals are OR'd together in Φ_1 to form **E_USQ%PE_ABORT_H**. The trap signals are prioritized and address lookup is done to select the appropriate microtrap handler address, which is driven on the **CAL INPUT BUS** in Φ_3.

9.2.3.3 Last Cycle Logic

The last cycle logic examines several conditions used to determine which new microflow is to be taken when LAST.CYCLE or LAST.CYCLE.OVERFLOW is detected on **E_USQ_CSM%UMIB_H**, no microtraps are detected, and no test address is driven. There are five possible new microflows, listed in order of priority:

1. Interrupt Request Handler
2. Trace Fault Handler
3. First Part Done Handler
4. Instruction Queue Stall
5. The macroinstruction microcode indicated by the top entry in the instruction queue.

The last cycle logic prioritizes these sources and performs address lookup. In addition, the signal **E_USQ_LST%SELECT_IQ_H** is derived. This signal is asserted when an entry is taken from the instruction queue.

Table 9–5: Microaddresses for Last Cycle Interrupts or Exceptions

Priority	Interrupt or Exception	Dispatch Address (Hex)
1	Interrupt request	24
2	Trace fault	28
3	First part done	2C
4	Instruction Queue Stall	30

The priorities in the last cycle logic are assigned using the following dependencies:

1. Interrupts and trace faults must be handled between instructions. (Interrupts may also be serviced at defined points during long instructions such as string instructions; this servicing is handled by microcode.)
2. By definition, an interrupt that is permitted to request service has a higher priority level (IPL) than any exception that occurs in the process to be interrupted, or any instruction to be executed by that process.
3. When tracing is enabled (PSL<TP> is set), a trace fault must be taken before the execution of each instruction.

4. If an instruction begins execution with PSL<FPD> set, the first part done handler must be entered rather than the normal entry point for the instruction.
5. PSL<TP> and PSL<FPD> cannot both be set when an instruction begins execution. In order for PSL<FPD> to be set, the instruction must have been interrupted previously; the interrupt handler always clears PSL<TP> before saving the PSL when interrupting an instruction. (Note that the interrupt handler does not clear PSL<TP> when the interrupt is taken between instructions.)
6. The Instruction Queue Stall microword is executed if an opcode is requested from the Instruction Queue but the queue is empty.

9.2.3.4 Microstack

Frequently used microcode can be made into microsubroutines. When a microsubroutine is called, the return address is pushed onto the microstack. The output of the microstack is driven on the **CAL INPUT BUS** when a RETURN is decoded from the **E_USQ_CSM%UMIB_H**, no microtraps are detected, and no test address is driven.

The microstack is 6 entries deep. It is a circular stack, with the write pointer always one entry ahead of the read pointer. Each entry is an 11-bit control store address. The addresses stored in the microstack incorporate any modification done by the microtest bus.

9.2.4 Stall Logic

The microsequencer is stalled whenever S3 is stalled. The Ebox derives the signal **E_STL%USEQ_STALL_H** which is used to stall the microsequencer. The microsequencer creates delayed versions of this signal as needed to stall various latches. The signals **E_USQ%PE_ABORT_H** (asserted on initiation of a microtrap) and **E_USQ_TST%FORCE_TEST_ADDR_H** (asserted on detection of the Test Address Generator driving a control store microaddress, see Section 9.5) break a microsequencer stall by clearing the delayed versions of **E_STL%USEQ_STALL_H**.

9.3 Initialization

A reset (assertion of **K_E%RESET_L**) causes the microsequencer to initialize in the following state:

- A powerup microtrap is initiated.
- The microstack pointer is reset to zero.
- The instruction queue is flushed and its pointers are reset by **E_MSC%FLUSH_EBOX_H**.

9.4 Microcode Restrictions

1. Every microtrap except Branch Mispredict must contain a RESET.CPU in order to reset the Instruction Queue. (The Ebox is flushed automatically, clearing the queues, on detection of branch mispredict.) RESET.CPU must not be issued within the 3 microwords preceding LAST.CYCLE in order to allow time for the Instruction Queue to be cleared (if RESET.CPU is present in microword N, LAST.CYCLE cannot be present until microword N+4).
2. For correct operation of Trace Fault and First Part Done in the Last Cycle Logic, PSL<T,TP,FPD> must not be changed within the 2 microwords preceeding LAST.CYCLE (if any of these PSL bits are changed in microword N, LAST.CYCLE cannot be present until microword N+3).

3. No Ebox-initiated memory requests can be made in the last cycle of a microflow, other than writes with the translation already known to be valid.
4. No Ebox-initiated memory requests can be outstanding when the microcode references an operand (queue entry or register file location).
5. The instruction queue stall microword must indicate LAST.CYCLE.

9.5 Testability

9.5.1 Test Address

The control store microaddress is both controllable and observable. A microcode address can be driven to the microsequencer from the Test Address Generator. The Test Address Generator is an 11-bit counter which is initialized to a value of zero on assertion of **K_E%RESET_L**. It increments its address counter once on each deassertion of **T%CS_TEST_H**, thus cycling through all possible control store addresses.

This microaddress source takes priority over all others. To ensure immediate control store lookup using this microaddress, assertion of **T%CS_TEST_H** sets an S/R latch whose output is **E_USQ_TST%FORCE_TEST_ADDR_H**. Assertion of this signal breaks any stall on Φ_2, Φ_3, and Φ_4 latches in the microsequencer. This allows the control store to operate, driving the selected microword into the MIB scan chain (see Section 9.5.2). The Ebox stall(s), if any, are unaffected, along with stalls on Φ_1 latches in the microsequencer.

E_USQ_TST%FORCE_TEST_ADDR_H is deasserted when the Test Address Generator has completed generation of all possible addresses.

The microaddress driven from the **CAL** can be be observed on the Parallel Test Port data pins, along with the microsequencer stall signal, under control of the Parallel Test Port command pins. The microsequencer drives to the Parallel Test Port in Φ_2.

Figure 9–3: Parallel Port Output Format

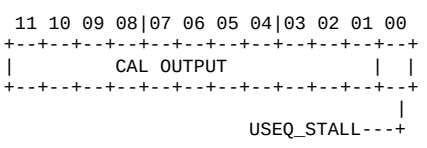


Table 9–6: Parallel Port Output Format Field Definitions

Name	Bit(s)	Description
CAL OUTPUT	11:1	Microaddress driven from **CAL**
USEQ_STALL	0	Microsequencer stall, **E_USQ_STL%VERY_LATE_USQ_STALL_H**

9.5.2 MIB Scan Chain

A 91-bit scan chain is present at the input to the **MIB**, allowing the complete microword to be latched and scanned out of the chip.

In addition, microcode patches are written into the patchable control store via the **MIB** scan chain.

Table 9–7: Contents of MIB Scan Chain

Extent	Description
<90:83>	E_USQ%MIB_H
<82:61>	E_USQ%MIB_H
<60:50>	CAM READ ADDRESS
<49:20>	E_USQ%MIB_H
<19:0>	CAM WRITE ADDRESS

9.6 Revision History

Table 9–8: Revision History

Rev	Who	When	Description of change
0.0	Elizabeth M. Cooper	06-Mar-1989	Release for external review.
0.1	Elizabeth M. Cooper	14-Sep-1989	Post-modelling update.
0.5	Elizabeth M. Cooper	10-Dec-1989	Updates for Rev 0.5 spec release.
0.5A	Elizabeth M. Cooper	5-Jan-1990	Remove vector microtrap and V bit from IQ.
0.5B	Elizabeth M. Cooper	20-Jun-1990	Accumulated updates.
Plus 0.1	Gil Wolrich	15-Nov-1990	Changes for NVAX Plus, retain block diagram and test features.

Chapter 10

The Interrupt Section

10.1 Overview

NVAX Plus inputs six external interrupt signals as IRQ_H<3:0>, HALT_H, and ERR_H. These signals are hardwired and level sensitive. The interrupts are non-vectored with the SCB Vector for each being predetermined. It is the responsibility of the interrupt software to determine the interrupt source and reset the interrupt. An explicit power fail interrupt is not implemented.

Internal interrupts include INT_TIM_H, H_ERR_H, S_ERR_H, SERIAL LINE, PERFORMANCE MONITOR FACILITY, and the architecturally defined Software Interrupt Requests. The full Interval Timer Implementation is present in the NVAX Plus chip, and thus no special considerations for the subset are necessary.

The interrupt section receives interrupt requests from both internal and external sources, and compares the IPL associated with the interrupt request to the current interrupt level in the PSL. If the interrupt request is for an IPL that is higher than the current PSL IPL, the interrupt section signals an interrupt request to the microsequencer which will initiate a microcode interrupt handler at the next macroinstruction boundary.

When an interrupt is serviced by the Ebox microcode, the interrupt section provides an encoded interrupt ID on **E_BUS%ABUS**, which allows the microcode to determine the highest priority interrupt request that is pending. Interrupt requests are cleared in one of two ways, depending on the type of request.

Software interrupt requests are supported via a 15-bit SISR register, which is read and written by the microcode, and which makes requests to the interrupt generation logic.

10.2 Interrupt Summary

Interrupt requests received from external logic are synchronized to internal clocks. In addition, there are several internal sources of interrupt requests which are received by edge-sensitive logic.

10.2.1 External Interrupt Requests Received by Level-Sensitive Logic

Six external interrupt requests are received by level-sensitive logic and synchronized to internal clocks. These signals request general-purpose interrupts at the following IPLs.

- **HALT_H**: The assertion of **HALT_H** causes the CPU to enter the console at IPL 1F (hex) at the next macroinstruction boundary. This interrupt is not gated by the current IPL, and always results in console entry, even if the IPL is already 1F (hex). Note that the implementation of this event is different from a normal interrupt in which a PC/PSL pair are pushed on the interrupt stack. For this event, the current PC, PSL, and halt code are stored in the SAVPC and SAVPSL processor registers.
- **ERR_H**: The assertion of **H_ERR_H** indicates that a error has been detected in the system environment. This results in the dispatch of the interrupt to the operating system at IPL 1D (hex) through SCB vector 60 (hex).
- **IRQ_H**<3:0>: Device interrupts resulting in dispatch of the interrupt to the operating system at IPL 14-17 (hex) through SCB vector D0,D4,D8, or DC (hex).

Interrupt Request	Request IPL (Hex)	Request IPL (Dec)	SCB Vector (Hex)
HALT_H	1F	31	CONSOLE
ERR_H	1D	29	60
IRQ_H<3>	17	23	DC
IRQ_H<2>	16	22	D8
IRQ_H<1>	15	21	D4
IRQ_H<0>	14	20	DO

Each signal must be driven HIGH and remain HIGH to assert the interrupt request. Interrupt routines at the specified SCB acknowledge the interrupt.

10.2.2 Internal Interrupt Requests

The Cbox, Ibox, and Mbox report error conditions by asserting internal interrupt request signals. The H_err signal is ORed with **ERR_H**, while S_err inputs directly. H_err causes an interrupt to SCB 60(HEX), S_err causes an interrupt to SCB 54(HEX).

The performance monitoring facility requests an interrupt at IPL 1B (hex) when the performance counters become half full. This request is serviced entirely by microcode, and cleared by writing to the appropriate bit in the ISR.

The assertion of INT_TIM_H indicates that the interval timer period has expired and ICCS<6> is set. The interrupt is dispatched to the operating system at IPL 16 (hex) through SCB vector C0 (hex).

Architecturally defined software interrupt requests are implemented through an internal register in the interrupt section. Under control of the SISR and SIRR processor registers which are described in Chapter 2, the Ebox microcode sets the appropriate bit in this register, which then results in the dispatch of the interrupt to the operating system at an IPL and through the SCB

vector implied by the interrupt request. The association between the interrupt request, requested IPL, and SCB vector for these requests is shown in the following table.

	Request IPL		SCB Vector
SISR bit	(Hex)	(Dec)	(Hex)
SISR<15>	0F	15	BC
SISR<14>	0E	14	B8
SISR<13>	0D	13	B4
SISR<12>	0C	12	B0
SISR<11>	0B	11	AC
SISR<10>	0A	10	A8
SISR<09>	09	09	A4
SISR<08>	08	08	A0
SISR<07>	07	07	9C
SISR<06>	06	06	98
SISR<05>	05	05	94
SISR<04>	04	04	90
SISR<03>	03	03	8C
SISR<02>	02	02	88
SISR<01>	01	01	84

Ebox microcode explicitly clears the interrupt request when the interrupt is serviced.

10.2.3 Special Considerations for Interval Timer Interrupts

NVAX Plus does not implement the subset Interval Timer and does not require a copy of ICCS<6> at the Interrupt Section.

10.2.4 Priority of Interrupt Requests

When multiple interrupt requests are pending, the interrupt section prioritizes the requests. Table 10–1 shows the relative priority (from highest to lowest) of all interrupt requests. For reference, this table also includes the IPL at which the interrupt is taken, and the SCB vector through which the interrupt is dispatched.

Table 10–1: Relative Interrupt Priority

Interrupt	Request IPL		SCB Vector	
Request	(Hex)	(Dec)	(Hex)	
HALT_H	1F	31	None[1]	Highest priority
ERR_H[2]	1D	29	60	

[1] Direct dispatch to console; PC, PSL placed in SAVPC, SAVPSL processor registers

[2] Includes Cbox, Ibox, and Mbox internally generated requests

Table 10–1 (Cont.): Relative Interrupt Priority

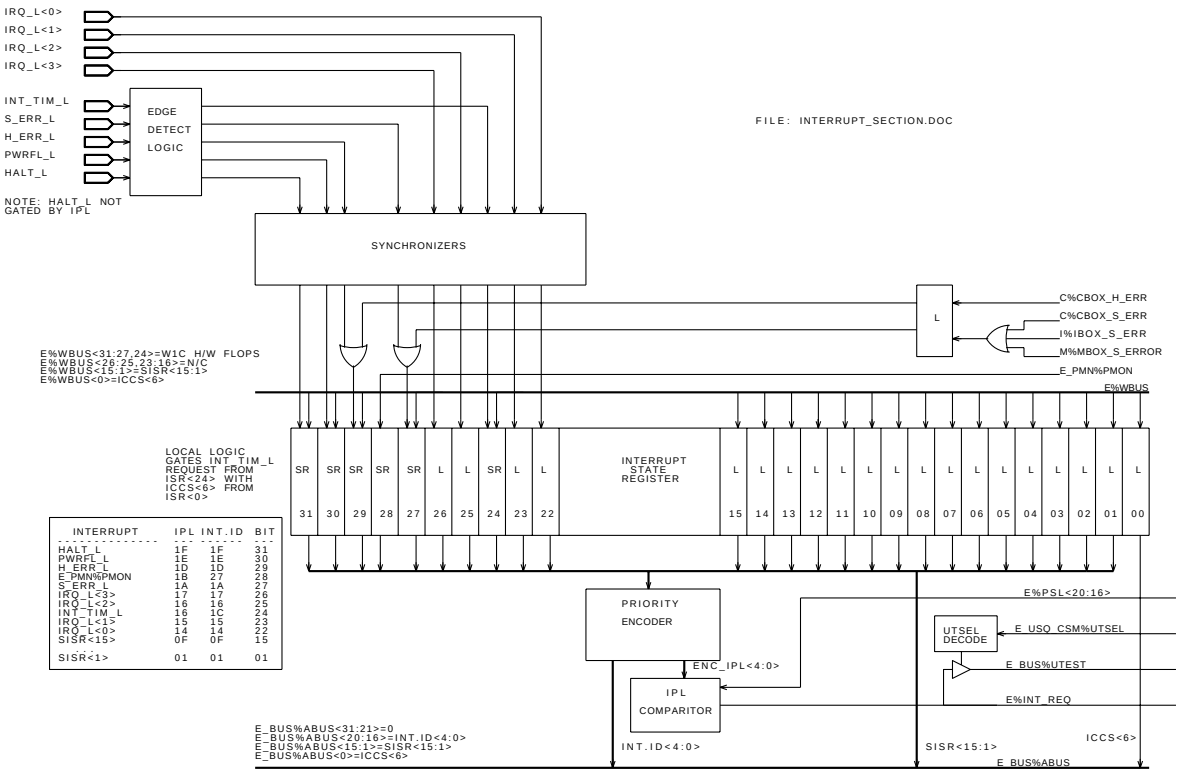
Interrupt Request	Request IPL (Hex)	Request IPL (Dec)	SCB Vector (Hex)	
Performance Monitoring Facility	1B	27	None[3]	
S_ERR_L[2]	1A	26	54	
SERIAL LINE	18	24	??	NEED SCB FOR SERIAL LINE
IRQ_H<3>	17	23	DC	
IRQ_H<2>	16	22	D8	
INT_TIM_L	16	22	C0	
IRQ_H<1>	15	21	D4	
IRQ_H<0>	14	20	D0	
SISR<15>	0F	15	BC	
SISR<14>	0E	14	B8	
SISR<13>	0D	13	B4	
SISR<12>	0C	12	B0	
SISR<11>	0B	11	AC	
SISR<10>	0A	10	A8	
SISR<09>	09	09	A4	
SISR<08>	08	08	A0	
SISR<07>	07	07	9C	
SISR<06>	06	06	98	
SISR<05>	05	05	94	
SISR<04>	04	04	90	
SISR<03>	03	03	8C	
SISR<02>	02	02	88	
SISR<01>	01	01	84	Lowest priority

[2]Includes Cbox, Ibox, and Mbox internally generated requests

[3]Interrupt processed entirely by microcode

The **IRQ_H**<2> request takes priority over the INT_TIM_L request, both of which are at IPL 16 (hex).

Figure 10–1: Interrupt Section Block Diagram



10.3 Interrupt Section Structure

The interrupt section consists of three basic components: the synchroniation logic, the interrupt state register (ISR), and the interrupt generation logic. A block diagram of the interrupt section is shown in Figure 10–1.

10.3.1 Synchronization Logic

The pads for the SIX external interrupt request signals contain synchronizers to allow the use of asynchronous signals for interrupt requests. The synchronized signals are then passed to the ISR.

10.3.2 Interrupt State Register

The interrupt state register is a composite register that implements the 15-bit architecturally defined SISR register, the interrupt latch for the performance monitoring facility interrupt, internal S_err, SERIAL LINE, and the interrupt request latches for the six external interrupts. The ISR contains two kinds of elements: SR flops for the internal interrupt requests, and latches for the external and software request interrupts. The following table lists the types and positions of all elements in the ISR.

ISR bit	State Element	Description
31	L	Interrupt request for **HALT_H** interrupt
29	L	Interrupt request for **ERR_H** and internal C%CBOX_H_ERR from BIU_STAT
28	SR	Interrupt request for performance monitoring facility interrupt
27	SR	Interrupt request for S_ERR_L /internal soft error interrupts
26	L	Interrupt request for **IRQ_H**<3> interrupt
25	L	Interrupt request for **IRQ_H**<2> interrupt
24	SR	Interrupt request for INT_TIM_L interrupt
23	L	Interrupt request for **IRQ_H**<1> interrupt
22	L	Interrupt request for **IRQ_H**<0> interrupt
15:1	L	SISR<15:1> latches and requests for software interrupts

State Element

SR—SR flop
L—Latch

Internal requests from the Cbox, Ibox, and Mbox cause the assertion of one of these signals causes the appropriate request flop to be set in ISR<30,27,24>. These request flops are cleared under Ebox microcode control when written with a 1 from **E%WBUS**.

The performance monitoring faciltiy interrupt request is loaded into the request flop in ISR<28>. The request is cleared by under Ebox microcode control when written with a 1 from **E%WBUS**.

SISR<15:1> is implemented via ISR<15:1>, and is loaded from bits <15:1> of **E%WBUS** under Ebox microcode control. These request latches are cleared under Ebox microcode control when a new value is loaded from **E%WBUS**.

The interval timer request from ISR<24> is not gated with ISR<0> as only a single version of ICCS<6> exits for NVAX Plus. NVAX Plus does not implement ISR<0>. (ISR<31:22,15:1>) go to the interrupt generation logic. ISR<15:1> may also be read onto **E_BUS%ABUS** for return to the Ebox.

10.3.3 Interrupt Generation Logic

The interrupt generation logic priority encodes all interrupt requests from the interrupt state register to determine the highest priority request. The output of the encoder is the request IPL and the interrupt ID of the highest priority request. If any request is pending, the request IPL is compared against **E%PSL**<20:16> from the Ebox. If the request IPL is higher than the PSL IPL, or if the request is for **HALT_H** (**HALT_H** is not gated by the IPL), **E%INT_REQ** is asserted to the microsequencer.

The assertion of **E%INT_REQ** causes the microsequencer to initiate a microcode interrupt handler at the next macroinstruction boundary. The same signal is available on the microtest bus as a microbranch condition, which is checked by the Ebox microcode during long instructions.

Along with the request IPL, the interrupt generation logic provides an encoded interrupt ID that identifies the highest priority interrupt. The interrupt ID is read onto **E_BUS%ABUS** along with ISR<15:1> when microcode references the A/INT.SYS source. For each interrupt, the interrupt ID encoding, request IPL, ISR bit number, method for clearing the interrupt, and SCB vector is shown in Table 10–2.

Table 10–2: Summary of Interrupts

Interrupt Request	Int ID (Hex)	Int ID (Dec)	Request IPL (Hex)	Request IPL (Dec)	ISR Bit (Dec)	Reset Method	SCB Vector (Hex)
HALT_H	1F	31	1F	31	31	Write 1 to ISR bit	Console Halt
SERIAL LINE	18	24	18	24	30	Write 1 to ISR bit	15
ERR_H[1]	1D	29	1D	29	29	Write 1 to ISR bit	60
E_PMN%PMON_L	1B	27	1B	27	28[2]	Write 1 to ISR bit	Handled by microcode
S_ERR_L[1]	1A	26	1A	26	27[2]	Write 1 to ISR bit	54
IRQ_H<3>	17	23	17	23	26	CLEARED BY INTERRUPT HANDLER	DC
IRQ_H<2>	16	22	16	22	25	CLEARED BY INTERRUPT HANDLER	D8
INT_TIM_L	1C[3]	28	16	22	24[2]	Write 1 to ISR bit	C0
IRQ_H<1>	15	21	15	21	23	CLEARED BY INTERRUPT HANDLER	D4
IRQ_H<0>	14	20	14	20	22	CLEARED BY INTERRUPT HANDLER	D0
SISR<15>	0F	15	0F	15	15	Write 0 to ISR bit	BC
SISR<14>	0E	14	0E	14	14	Write 0 to ISR bit	B8
SISR<13>	0D	13	0D	13	13	Write 0 to ISR bit	B4
SISR<12>	0C	12	0C	12	12	Write 0 to ISR bit	B0

[1]Includes Cbox, Ibox, and Mbox internally generated requests

[2]Write-1-to-clear ISR bit is different than IPL and interrupt ID

[3]Interrupt ID is different than IPL

Table 10–2 (Cont.): Summary of Interrupts

Interrupt Request	Int ID (Hex)	Int ID (Dec)	Request IPL (Hex)	Request IPL (Dec)	ISR Bit (Dec)	Reset Method	SCB Vector (Hex)
SISR<11>	0B	11	0B	11	11	Write 0 to ISR bit	AC
SISR<10>	0A	10	0A	10	10	Write 0 to ISR bit	A8
SISR<09>	09	09	09	09	09	Write 0 to ISR bit	A4
SISR<08>	08	08	08	08	08	Write 0 to ISR bit	A0
SISR<07>	07	07	07	07	07	Write 0 to ISR bit	9C
SISR<06>	06	06	06	06	06	Write 0 to ISR bit	98
SISR<05>	05	05	05	05	05	Write 0 to ISR bit	94
SISR<04>	04	04	04	04	04	Write 0 to ISR bit	90
SISR<03>	03	03	03	03	03	Write 0 to ISR bit	8C
SISR<02>	02	02	02	02	02	Write 0 to ISR bit	88
SISR<01>	01	01	01	01	01	Write 0 to ISR bit	84
No Interrupt	00	00	—	—	—	Dismiss interrupt	—

The interrupt ID is the same as the request IPL for all interrupt requests except for the interval timer request.

DESIGN CONSTRAINT

A value of zero for the interrupt ID must be returned if an interrupt is no longer present, or if the highest priority interrupt request is no longer higher than the PSL IPL. Normally, once an interrupt request is made, it remains until it is cleared by the microcode. However, the level-sensitive interrupt requests may be deasserted after the interrupt is dispatched, but before the microcode reads the interrupt ID. Therefore, it is possible that the highest remaining interrupt has a request IPL lower than the current PSL IPL. If zero is not returned for the interrupt ID in this instance, the processor will not function correctly.

10.4 Ebox Microcode Interface

The Ebox microcode interfaces with the interrupt section primarily through reads (via **E_BUS%ABUS**) and writes (via **E%WBUS**) of the ISR accomplished through the A/INT.SYS and DST/INT.SYS decodes. These decodes provide access to the so-called INT.SYS register, which is shown in Figure 10–2. The fields of the register are listed in Table 10–3.

Figure 10–2: INT.SYS Register Format

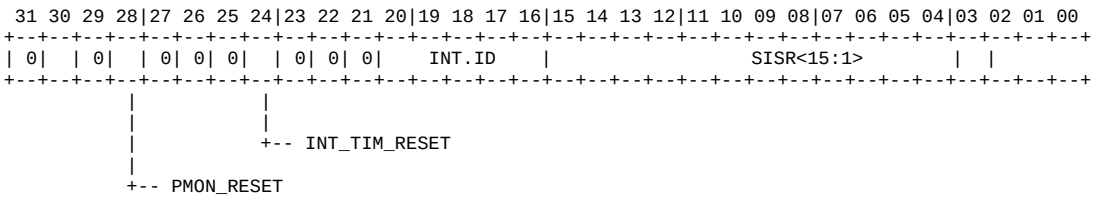


Table 10–3: INT.SYS Register Fields

Name	Bit(s)	Type	Description
SISR	15:1	RW,0	This field contains the 15 architecturally-defined software interrupt request bits. It is set to 0 by microcode at powerup.
INT.ID	20:16	RO	This field contains the encoding of the highest priority interrupt request as listed in Table 10–2. Writes to this field are ignored.
INT_TIM_RESET	24	WC,0	Writing a 1 to this field clears the INT_TIM_L interrupt request. Writing a 0 has no effect on the request. The field is read as a 0 and the interrupt request is cleared by microcode at powerup.
PMON_RESET	28	WC,0	Writing a 1 to this field clears the E_PMN%PMON_L interrupt request. Writing a 0 has no effect on the request. The field is read as a 0 and the interrupt request is cleared by microcode at powerup.

DESIGN CONSTRAINT

When read onto **E_BUS%ABUS**, INT.SYS<30,28,27,24> must be zero. Microcode updates the internal copy of SISR<15:1> by reading the INT.SYS register,modifying the appropriate bits, and writing the updated value back. The write-one-to-clear bits must be read as zero because the microcode does not mask them out before writing them back.

MICROCODE RESTRICTION

The INT.SYS register is not bypassed. A write to INT.SYS in microinstruction *n* must not be followed by a read of INT.SYS sooner than microinstruction *n+4*.

MICROCODE RESTRICTION

Changes to machine state that affect the generation of interrupts (PSL<IPL>, or SISR<15:1>) done by microinstruction *n* must not be followed by a LAST CYCLE microinstruction sooner than microinstruction *n+4* if the change is to be observed by the next macroinstruction.

10.5 Processor Register Interface

Software can interact with the interrupt section hardware and microcode via references to processor registers, as follows:

- SISR, SIRR: References to the architecturally-defined SISR and SIRR processor registers allow access to SISR<15:1>, which are implemented in INT.SYS<15:1>.
- INTSYS: References to the INTSYS processor register allow diagnostic and test software direct access to the INT.SYS register. Reads of the INTSYS processor register return the format shown in Figure 10–2. Writes of the INTSYS processor register are internally masked by microcode such that only the left half write-to-clear bits are written. Other bits remain unchanged. Writes to the INTSYS processor during normal system operation can result in UNDEFINED behavior.

10.6 Interrupt Section Interfaces

10.6.1 Ebox Interface

10.6.1.1 Signals From Ebox

- **E%PSL**<20:16>: IPL field from the current PSL.
- **E%WBUS**: Write data bus, from which SISR<15:1> are loaded, and from which the write-one-to-clear interrupt latches are cleared.
- **E_PMN%PMON_L**: Performance monitoring facility interrupt request.

10.6.1.2 Signals To Ebox

- **E_BUS%ABUS**: A-port operand bus, on which SISR<15:1> and the interrupt ID are returned.

10.6.2 Microsequencer Interface

10.6.2.1 Signals from Microsequencer

- **E_USQ_CSM%UTSEL**: Microtest bus select code.

10.6.2.2 Signals To Microsequencer

- **E%INT_REQ**: Interrupt pending.
- **E_BUS%UTEST**: Microtest bus.

10.6.3 Cbox Interface

10.6.3.1 Signals From Cbox

- **C%CBOX_H_ERR**: Hard error interrupt request.
- **C%CBOX_S_ERR**: Soft error interrupt request.
- **INT_TIM_L**: Interval timer interrupt signal.

10.6.4 Ibox Interface

10.6.4.1 Signals From Ibox

- **I%IBOX_S_ERR**: Soft error interrupt request.

10.6.5 Mbox Interface

10.6.5.1 Signals From Mbox

- **M%MBOX_S_ERROR**: Soft error interrupt request.

10.6.6 Pin Interface

10.6.6.1 Input Pins

- **HALT_H**: Halt interrupt signal
- **ERR_H**: Error interrupt signal
- **IRQ_H**<3:0>: General-purpose interrupt signals

10.7 Revision History

Table 10–4: Revision History

Who	When	Description of change
Mike Uhler	06-Mar-1989	Release for external review.
Mike Uhler	14-Dec-1989	Update for second-pass release.
Ron Preston	09-Jan-1990	Changes to simplify implementation.
Mike Uhler	20-Jul-1990	Update for change to performance monitoring interrupt request and reflect implementation.
Gil Wolrich	15-Nov-1990	NVAX Plus modifications

Chapter 11

The Fbox

11.1 Overview

This chapter provides a high level description of the floating point unit of the NVAX Plus CPU chip. For complete specification of the FBOX refer to the NVAX CPU Chip Functional Specification.

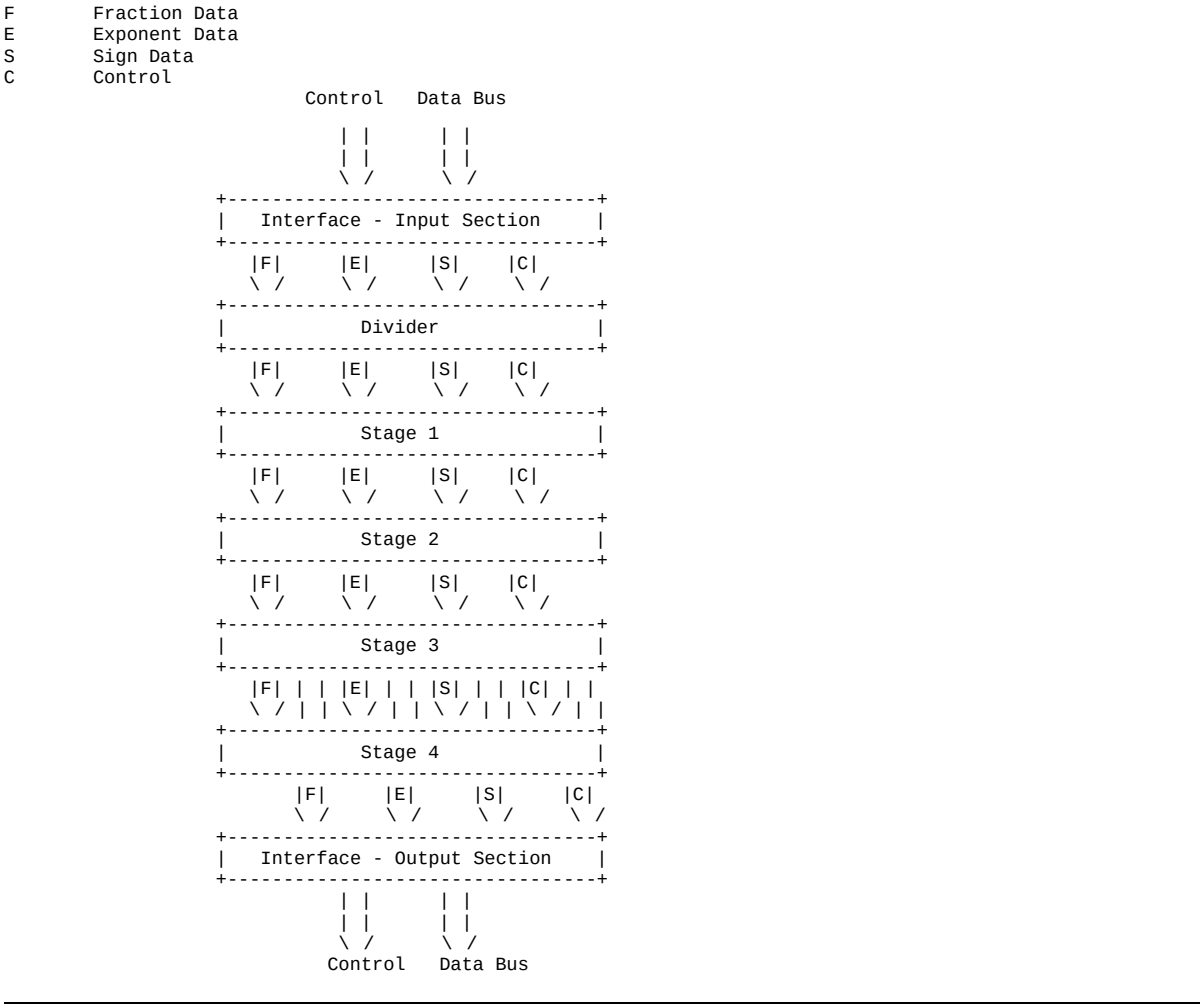
11.2 Introduction

The Fbox is the floating point unit in the NVAX Plus CPU chip. The Fbox is a 4 stage pipelined floating point processor, with an additional stage devoted to assisting division. It interacts with three different segments of the main CPU pipeline, these are the micro-sequencer in S2 and the Ebox in S3 and S4. The Fbox runs semi-autonomously to the rest of the CPU chip and supports the following operations:

- **VAX Floating Point Instructions and Data Types**
  The Fbox provides instruction and data support for VAX floating point instructions. VAX F-, D-, and G-floating point data types are supported.
- **VAX Integer Instructions**
  The Fbox implements longword integer multiply instructions.
- **Pipelined Operation**
  Except for all the divide instructions, DIV{F,D,G}, the Fbox can start a new single precision floating point instruction every cycle and a double precision floating point or an integer multiply instruction every two cycles. The Ebox can supply two 32-bit operands or one 64-bit operand to the Fbox every cycle on two 32 bit input operand buses. The Fbox drives the result operand to the Ebox on a 32-bit result bus.
- **Conditional "Mini-Round" Operation**
  Result latency is conditionally reduced by one cycle for the most frequently used instructions. Stage 3 can perform a "mini-round" operation on the LSB's of the fraction for all ADD, SUB, and MUL floating instructions. If the "mini-round" operation does not fail, then stage 3 drives the result directly to the output, bypassing stage 4 and saving a cycle of latency.
- **Fault and Exception Handling**
  The Ebox coordinates the fault and exception handling with the Fbox. Any fault or exception condition received from the Ebox is retired in the proper order. If the Fbox receives or generates any fault or exception condition, it does not change the flow of instructions in progress within the Fbox pipe.

Figure 11–1 is a top level block diagram of the Fbox showing the six major functional blocks within the Fbox and their interconnections.

Figure 11–1: Fbox block diagram

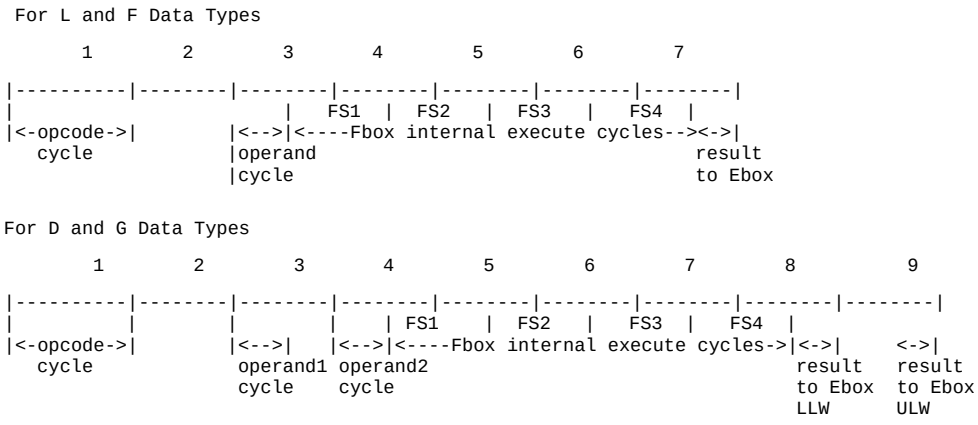


11.3 Fbox Functional Overview

The Fbox is the floating point accelerator for the NVAX CPU. Its instruction repertoire includes all VAX base group floating point instructions. The data types that are supported are F, D, and G. Additional integer instructions that are supported are MULL2, and MULL3.

The number of internal execution cycles and the total number of cycles to complete an instruction within the Fbox is measured as follows in Figure 11–2

Figure 11–2: Fbox Execute Cycle Diagram



The internal execution time for all instructions except MUL{D,G,L} and DIV{F,D,G} is four cycles. The internal execution time of the various Fbox operations is given in the following Table 11–1.

Table 11–1: Fbox Internal Execute Cycles

INSTRUCTION	F	D	G	L
MUL	4	5	5	5
DIV	14	25	24	-
ALL OTHER	4	4	4	4

The total number of cycles taken by the Fbox to complete an instruction is given in Table 11–2. Note that this includes the cycles taken for opcode and operand transfer, in particular, the dead cycle between the opcode and the first operand is counted.

Table 11–2: List of the Fbox Total Execute Cycles

INSTRUCTION	F	D	G	L
MUL	7	10	10	8
DIV	17	30	29	-
ALL OTHER	7	9	9	-

11.3.1 Fbox Interface

This section is responsible for overseeing the protocol with the Ebox. This includes the sequence of receiving the opcode, operands, exceptions, and other control information, and also outputing the result with its accompanying status. The opcode and operands are transferred from the input

interface to stage 1 in all operations except division. The result is conditionally received from either stage 3 or stage 4.

11.3.2 Divider

The divider receives its inputs from the interface and drives its outputs to stage 1. It is used only to assist the divide operation, for which it computes the quotient and the remainder in a redundant format.

11.3.3 Stage 1

Stage 1 receives its inputs from either the interface or the divider section and drives its outputs to stage 2. It is primarily used for determining the difference between the exponents of the two operands, subtracting the fraction fields, performing the recoding of the multiplier and forming three times the multiplicand, and selecting the inputs to the first two rows of the multiplier array.

11.3.4 Stage 2

Stage 2 receives its inputs from stage 1 and drives its outputs to stage 3. Its primary uses are: right shifting (alignment), multiplying the fraction fields of the operands, and zero and leading one detection of the intermediate fraction results.

11.3.5 Stage 3

Stage 3 receives most of its inputs from stage 2 and drives its outputs to stage 4 or, conditionally, to the output. Its primary uses are: left shifting (normalization), and adding the fraction fields for the aligned operands or the redundant multiply array outputs. This stage can also perform a "mini-round" operation on the LSB's of the fraction for ADD, SUB, and MUL floating instructions. If the "mini-round" does not overflow, and if there are no possible exceptions, then stage 3 drives the result directly to the output, bypassing stage 4 and saving a cycle of latency.

11.3.6 Stage 4

Stage 4 receives its inputs from stage 3 and drives its outputs to the interface section. It is used for performing the terminal operations of the instruction such as rounding, exception detection (overflow, underflow, etc.), and determining the condition codes.

11.3.7 Fbox Instruction Set

The instructions listed in Table 11–3 constitute the VAX integer and floating point instructions supported by the Fbox datapath.

Table 11–3: Fbox Floating Point and Integer Instructions

Fbox Opc	Instruction	NZVC	CC MAP	DL	Exceptions
04C	CVTBF src.rb, dst.wf	**00	10	10	
06C	CVTBD src.rb, dst.wd	**00	10	11	
14C	CVTBG src.rb, dst.wg	**00	10	11	
04D	CVTWF src.rw, dst.wf	**00	10	10	
06D	CVTWD src.rw, dst.wd	**00	10	11	
14D	CVTWG src.rw, dst.wg	**00	10	11	
04E	CVTLF src.rl, dst.wf	**00	10	10	
06E	CVTLD src.rl, dst.wd	**00	10	11	
14E	CVTLG src.rl, dst.wg	**00	10	11	
048	CVTFB src.rf, dst.wb	***0	11	00	rsv, iov
049	CVTFW src.rf, dst.ww	***0	11	01	rsv, iov
04A	CVTFL src.rf, dst.wl	***0	11	10	rsv, iov
068	CVTDB src.rd, dst.wb	***0	11	00	rsv, iov
069	CVTDW src.rd, dst.ww	***0	11	01	rsv, iov
06A	CVTDL src.rd, dst.wl	***0	11	10	rsv, iov
148	CVTGB src.rg, dst.wb	***0	11	00	rsv, iov
149	CVTGW src.rg, dst.ww	***0	11	01	rsv, iov
14A	CVTGL src.rg, dst.wl	***0	11	10	rsv, iov
04B	CVTRFL src.rf, dst.wl	***0	11	10	rsv, iov
06B	CVTRDL src.rd, dst.wl	***0	11	10	rsv, iov
14B	CVTRGL src.rg, dst.wl	***0	11	10	rsv, iov
056	CVTFD src.rf, dst.wd	**00	10	11	rsv
199	CVTFG src.rf, dst.wg	**00	10	11	rsv
076	CVTDF src.rd, dst.wf	**00	10	10	rsv, fov
133	CVTGF src.rg, dst.wf	**00	10	10	rsv, fov, fuv
040	ADDF2 add.rf, sum.mf	**00	10	10	rsv, fov, fuv
041	ADDF3 add1.rf, add2.rf, sum.wf	**00	10	10	rsv, fov, fuv
060	ADDD2 add.rd, sum.md	**00	10	11	rsv, fov, fuv
061	ADDD3 add1.rd, add2.rd, sum.wd	**00	10	11	rsv, fov, fuv
140	ADDG2 add.rg, sum.mg	**00	10	11	rsv, fov, fuv
141	ADDG3 add1.rg, add2.rg, sum.wg	**00	10	11	rsv, fov, fuv

Table 11–3 (Cont.): Fbox Floating Point and Integer Instructions

Fbox Opc	Instruction	NZVC	CC MAP	DL	Exceptions
042	SUBF2 sub.rf, dif.mf	**00	10	10	rsv, fov, fuv
043	SUBF3 sub.rf, min.rf, dif.wf	**00	10	10	rsv, fov, fuv
062	SUBD2 sub.rd, dif.md	**00	10	11	rsv, fov, fuv
063	SUBD3 sub.rd, min.rd, dif.wd	**00	10	11	rsv, fov, fuv
142	SUBG2 sub.rg, dif.mg	**00	10	11	rsv, fov, fuv
143	SUBG3 sub.rg, min.rg, dif.wg	**00	10	11	rsv, fov, fuv
0C4	MULL2 mulr.rl, prod.ml	***0	11	10	iov
0C5	MULL3 mulr.rl, muld.rl, prod.wl	***0	11	10	iov
044	MULF2 mulr.rf, prod.mf	**00	10	10	rsv, fov, fuv
045	MULF3 mulr.rf, muld.rf, prod.wf	**00	10	10	rsv, fov, fuv
064	MULD2 mulr.rd, prod.md	**00	10	11	rsv, fov, fuv
065	MULD3 mulr.rd, muld.rd, prod.wd	**00	10	11	rsv, fov, fuv
144	MULG2 mulr.rg, prod.mg	**00	10	11	rsv, fov, fuv
145	MULG3 mulr.rg, muld.rg, prod.wg	**00	10	11	rsv, fov, fuv
046	DIVF2 divr.rf, quo.mf	**00	10	10	rsv, fov, fuv, fdvz
047	DIVF3 divr.rf, divd.rf, quo.wf	**00	10	10	rsv, fov, fuv, fdvz
066	DIVD2 divr.rd, quo.md	**00	10	11	rsv, fov, fuv, fdvz
067	DIVD3 divr.rd, divd.rd, quo.wd	**00	10	11	rsv, fov, fuv, fdvz
146	DIVG2 divr.rg, quo.mg	**00	10	11	rsv, fov, fuv, fdvz
147	DIVG3 divr.rg, divd.rg, quo.wg	**00	10	11	rsv, fov, fuv, fdvz
050	MOVF src.rf, dst.wf	**0-	01	10	rsv
070	MOVD src.rd, dst.wd	**0-	01	11	rsv
150	MOVG src.rg, dst.wg	**0-	01	11	rsv
052	MNEGF src.rf, dst.wf	**00	10	10	rsv
072	MNEGD src.rd, dst.wd	**00	10	11	rsv
152	MNEGG src.rg, dst.wg	**00	10	11	rsv
051	CMPF src1.rf, src2.rf	**00	10	xx	rsv

Table 11–3 (Cont.): Fbox Floating Point and Integer Instructions

Fbox Opc	Instruction	NZVC	CC MAP	DL	Exceptions
071	CMPD src1.rd, src2.rd	**00	10	xx	rsv
151	CMPG src1.rg, src2.rg	**00	10	xx	rsv
053	TSTF src.rf	**00	10	xx	rsv
073	TSTD src.rd	**00	10	xx	rsv
153	TSTG src.rg	**00	10	xx	rsv

CC_MAP: Condition Code Map

00 = No Update
01 = MOV Floating
10 = All Other Floating
11 = Integer

DL: Result Data Length

00 = Byte
01 = Word
10 = Long
11 = Quad

11.3.8 Revision History

Table 11–4: Revision History

Who	When	Description of change
Anil Jain	17-Mar-1989	Initial Release
Anil Jain	18-Dec-1989	Updated to reflect the Fbox implementation
Gil Wolrich	15-Nov-1990	Retain FBOX overview for NVAX Plus Spec

Chapter 12

The Mbox

12.1 INTRODUCTION

This chapter contains the high level description of the NVAX Plus MBOX, and specifies the changes with respect to PCache Invalidates and external map support. It also includes EBOX and CBOX interface descriptions, IPR specifications, and testability features from the NVAX CPU Chip Functional Specification. Refer to NVAX CPU Chip Functional Specification for the detailed decription of the MBOX.

The Mbox performs three primary functions:

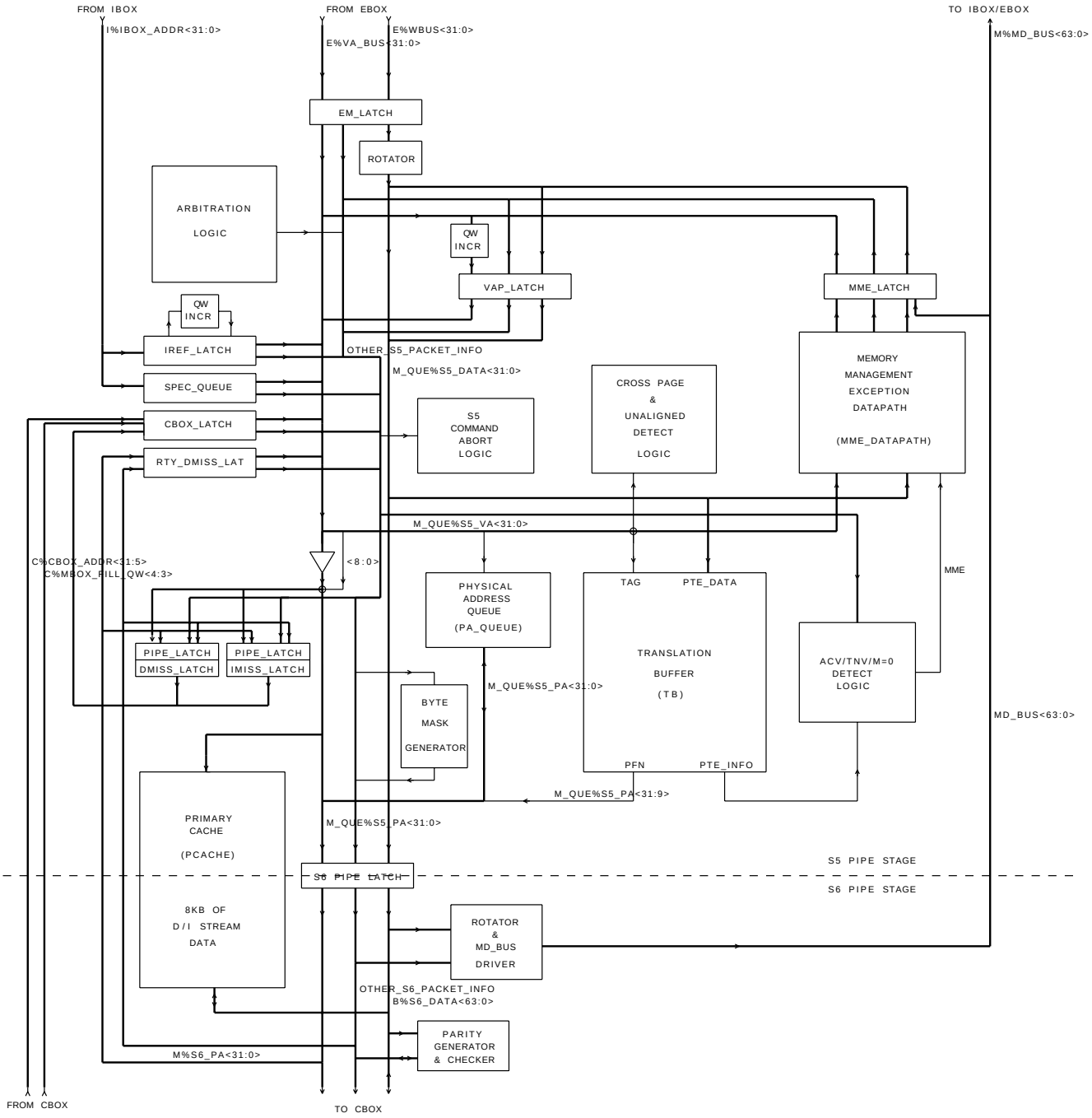
- VAX memory management: The Mbox, in conjunction with the operating system memory management software, is responsible for the allocation and use of physical memory. The Mbox performs the hardware functions necessary to implement VAX memory management. It performs translations of virtual addresses to physical addresses, access violation checks on all memory references, and initiates the invocation of software memory management code when necessary.
- Reference processing: Due to the macropipeline structure of NVAX Plus, and the coupling between NVAX Plus and its memory subsystem, the Mbox can receive memory references from the Ibox, Ebox and Cbox(invalidates) simultaneously. Thus, the Mbox is responsible for prioritizing, sequencing, and processing all references in an efficient and logically correct fashion and for transferring references and their corresponding data to/from the Ibox, Ebox, Pcache, and Cbox.
- Primary Cache Control: The Mbox maintains an 8KB physical address cache of I-stream and D-stream data. This cache, called the Pcache (Primary Cache), exists in order to provide a two cycle pipeline latency for most I-stream and D-stream data requests. It is the fastest D-stream storage medium for NVAX Plus and represents the first level of D-stream memory hierarchy and the second level of I-stream memory hierarchy for the NVAX Plus scalar data. The Mbox is responsible for controlling Pcache operation.

12.2 MBOX STRUCTURE

This section presents a block diagram of the Mbox and defines the function of the basic Mbox components.

The following block diagram illustrates the basic components of the Mbox.

Figure 12–1: Mbox Block Diagram



The Mbox is implemented as a two-stage pipeline located in the fifth and sixth segments of the NVAX Plus macropipeline (S5 and S6). References processed by the Mbox are first executed in S5. Upon successful completion in S5, the reference is transferred into S6. At this point, the reference has either completed or is transferred to the Ibox, Ebox, or Cbox.

During any cycle, the fundamental state of the S5 and S6 stages can be defined by the particular references which currently reside in these two stages. For the purposes of describing the Mbox, all references can be viewed as a packet of information which is transferred on the S5 and S6 buses. The S5 reference packet, and the corresponding S5 buses are defined as:

- ADDRESS: The **M_QUE%S5_VA**<31:0> bus transfers all virtual addresses and some physical addresses into the S5 pipe. The **M_QUE%S5_PA**<31:0> bus transfers some physical addresses into the S5 pipe and transfers all addresses out of the S5 pipe.
- DATA: **M_QUE%S5_DATA**<31:0> transfers data originating from the Ebox, through the S5 pipe.
- COMMAND: **M_QUE%S5_CMD**<4:0> transfers the type of reference through the S5 pipe. This command field is defined in Section 12.3.1.
- TAG: The **M_QUE%S5_TAG**<4:0> transfers the Ebox register file destination address corresponding to the reference through the S5 pipe.
- DEST_BOX: **M_QUE%S5_DEST**<1:0> transfers the reference destination information through the S5 pipe. This field is defined as follows:

M_QUE%S5_DEST	Definition
00:	the reference requests data destined for the Mbox.
01:	the reference requests data destined for the Ibox.
10:	the reference requests data destined for the Ebox.
11:	the reference requests data destined for the Ebox and Ibox.

- AT: The **M_QUE%S5_AT**<1:0> transfers the access type of the reference. This field is defined as follows:

M_QUE%S5_AT	Definition
00:	tb passive query access (See PROBE command)
01:	read access
10:	write access
11:	modify access (read with write check for future write to same addr)

- DL: The **M_QUE%S5_DL**<1:0> transfers the data length of the reference. This field is defined as follows:

M_QUE%S5_DL	Definition
00:	byte
01:	word
10:	longword

M_QUE%S5_DL	Definition
11:	quadword

- BYTE_MASK: The **M_QUE%S5_BM**<7:0> transfers the byte mask information out of the S5 pipe.
- REF_QUAL: The **M_QUE%S5_QUAL**<6:0> transfers information which further qualifies the reference for the purpose of Mbox processing. This field is defined as follows:

M_QUE%S5_QUAL bit	Definition
M_QUE%S5_QUAL<6>	address of reference is currently a virtual address.
M_QUE%S5_QUAL<5>	reference has been tested for cross-page condition.
M_QUE%S5_QUAL<4>	reference is first part of an unaligned reference.
M_QUE%S5_QUAL<3>	reference is second part of an unaligned reference.
M_QUE%S5_QUAL<2>	enable ACV and M=0 checks.
M_QUE%S5_QUAL<1>	reference has or is forced to have a hard error.
M_QUE%S5_QUAL<0>	reference has or is forced to have a memory management fault (ACV/TNV/M=0).

The S6 reference packet, and the corresponding S6 buses are defined as:

- ADDRESS: The **M%S6_PA**<31:0> bus transfers a physical address through the S6 pipe.
- DATA: **B%S6_DATA**<63:0> transfers data through the S6 pipe.
- COMMAND: **M%S6_CMD**<4:0> transfers the type of reference through the S6 pipe. This command field is defined in Section 12.3.1.
- TAG: The **M_QUE%S6_TAG**<4:0> transfers the Ebox register file destination address corresponding to the reference through the S6 pipe.
- DEST_BOX: **M_QUE%S6_DEST**<1:0> transfers the reference destination information through the S6 pipe. This field is defined as follows:

M_QUE%S6_DEST	Definition
00:	the reference requests data destined for the Mbox.
01:	the reference requests data destined for the Ibox.
10:	the reference requests data destined for the Ebox.
11:	the reference requests data destined for the Ebox and Ibox.

- S6_BYTE_MASK: **M%S6_BYTE_MASK**<7:0> transfers the byte mask information through the S6 pipe. The byte mask field is used to indicate which bytes of a longword or quadword write should actually be written to a cache or memory.
- REF_QUAL: **M_QUE%S6_QUAL**<3:0> transfers information which further qualifies the reference for the purpose of Mbox processing. This field is defined as follows:

M_QUE%S6_QUAL bit	Definition
M_QUE%S6_QUAL<3>	reference is first part of an unaligned reference.
M_QUE%S6_QUAL<2>	reference is second part of an unaligned reference.
M_QUE%S6_QUAL<1>	reference has or is forced to have a hard error.
M_QUE%S6_QUAL<0>	reference has or is forced to have a memory management fault (ACV/TNV/M=0).

12.2.1 EM_LATCH

The EM_LATCH latches and stores all commands originating from the Ebox. Each reference is stored until the following two conditions are satisfied: 1) the "complete logical reference" (i.e. the pair of aligned references required if the EM_LATCH reference is unaligned) clear memory management access checks, and 2) the EM_LATCH reference successfully completes in S5.

A 4-way byte barrel shifter is connected to the data portion of the EM_LATCH. This enables the write data to be byte-rotated into longword alignment. The EM_LATCH output can be tristated.

12.2.2 CBOX_LATCH

The CBOX_LATCH stores references originating from the Cbox. These references are I-stream Pcache fills, D-stream Pcache fills, or Pcache hexaword invalidates. Each reference is stored until the reference successfully completes in S5.

Note that no data field is present in this latch even though this latch services cache fill commands.

Cache fill data will be supplied to the Pcache on the **B%S6_DATA** Bus by the Cbox during the appropriate S6 cache fill cycle. The **C%CBOX_ADDR** bus is driven by the Cbox during invalidate commands. During cache fill commands, all but two bits of the **C%CBOX_ADDR** bus are driven by the DMISS_LATCH or IMISS_LATCH. The Cbox will drive **C%MBOX_FILL_QW** during cache fill commands in order to supply the quadword alignment of the fill data within the hexaword block. The CBOX_LATCH output can be tristated.

12.2.3 TB

The TB (translation buffer) is the mechanism by which the Mbox performs quick virtual-to-physical address translations. It is a 96-entry fully associative cache of PTEs (Page Table Entries). Bits 31 through 9 of all S5 virtual addresses act as the TB tag. The replacement algorithm implemented is Not-Last-Used.

12.2.4 DMISS_LATCH and IMISS_LATCH

The DMISS_LATCH stores the currently outstanding D-stream read. That is, a D-stream read, which missed in the Pcache, is stored in the DMISS_LATCH until the corrsponding Pcache block fill operation completes. The DMISS_LATCH also stores IPR_RDs to be processed by the Cbox until the Cbox supplies the data. I-stream reads are handled analogously by the IMISS_LATCH except that IPR_RDs are never handled by the IMISS_LATCH.

These two latches have comparators built in in order to detect the following conditions:

- For NVAX If the hexaword address of an invalidate matches the hexaword address stored in either MISS_LATCH, the corresponding MISS_LATCH sets a bit to indicate that the corresponding fill operation is no longer cacheable in the Pcache. **NVAX Plus invalidates only specify index<12:5>, and the PCache set to be invalidated. If the index and MISS_LATCH allocation bit match an invalidate the the corresponding MISS_LATCH sets a bit to indicate that the corresponding fill operation is no longer cacheable in the Pcache.**
- Address<11:5> addresses a particular Pcache index (corresponding to two Pcache blocks). If address<8:5> of the DMISS_LATCH matches the corresponding bits of the physical address of an S5 I-stream read, the S5 I-stream read is stalled until the entire D-stream fill operation completes. This prevents the possibility of causing a D-stream fill sequence to a given Pcache block from simultaneously happening with an I-stream fill sequence to the same Pcache block.
- By the same argument, address<8:5> of the IMISS_LATCH is compared against S5 D-stream reads to prevent another simultaneous I-stream/D-stream fill sequence to the same Pcache block.
- Address<8:5> of both miss_latches is compared against any S5 memory write operation. This is necessary to prevent the write from interfering with the cache fill sequence.

12.2.5 Pcache

The Pcache is a two-way set associative, read allocate, no-write allocate, write through, physical address cache of I-stream and D-stream data. Some systems may force the Pcache to allocate such that if address[12]=0 set 0 is loaded, and if address[12]=1 set 1 is loaded, using the Pcache as if it were direct mapped so that the Pcache can be backmapped exactly as the EV4 Dcache. The Pcache stores 8192 bytes (8K) of data and 256 tags corresponding to 256 hexaword blocks (1 hexaword = 32 bytes). Each tag is 20 bits wide corresponding to bits <31:12> of the physical address. There are four quadword subblocks per block with a valid bit associated with each subblock. The access size for both Pcache reads and writes is one quadword. Byte parity is maintained for each byte of data (32 bits per block). One bit of parity is maintained for every tag. The Pcache has a one cycle access and a one cycle repetition rate for both reads and writes (note however, that the entire Mbox latency is two cycles due to the two stage Mbox pipeline).

12.3 REFERENCE PROCESSING

This section discusses how references are processed by the Mbox, and how the Mbox functional components interact to carry out reference processing.

12.3.1 REFERENCE DEFINITIONS

The following table describes all types of references processed by the Mbox:

Table 12–1: Reference Definitions

Name	Value (hex)	Reference Source	Description
IREAD	0E	Ibox	Aligned quadword I-stream read

Table 12–1 (Cont.): Reference Definitions

Name	Value (hex)	Reference Source	Description
DREAD	1C	Ibox, Ebox, Mbox	Variable length D-stream read
DREAD_MODIFY	1D	Ibox	Variable length D-stream read with modify intent as a result of Ibox-decoded modify specifiers
DREAD_LOCK	1F	Ebox	Variable length D-stream read with atomic memory lock
WRITE_UNLOCK	1A	Ebox	Variable length write with atomic memory unlock
WRITE	1B	Ebox	Variable length write
DEST_ADDR	1D	Ibox	Supplies address of a write-only destination specifier
STORE	19	Ebox	Supplies write data corresponding to a previously translated destination specifier address.
IPR_WR	06	Ebox	Internal Processor Register Write
IPR_RD	07	Ebox	Internal Processor Register Read
IPR_DATA	04	Mbox	Transfers Mbox IPR data to Ebox
LOAD_PC	05	Ebox	Transfers a PC value to Ibox via M%MD_BUS<31:0>
PROBE	09	Ebox	Mbox returns ACV/TNV/M=0 status of specified address to Ebox.
MME_CHK	08	Ebox, Mbox	Performs ACV/TNV/M=0 check on specified address and invokes the appropriate memory management exception
TB_TAG_FILL	0C	Ebox, Mbox	Writes a TB tag into a TB entry.
TB_PTE_FILL	14	Ebox, Mbox	Writes PTE data into a TB entry.
TBIS	10	Ebox	Invalidates a specific PTE entry in the TB.
TBIA	18	Ebox,Mbox	Invalidates all entries in TB.
TBIP	11	Ebox	Invalidates all PTE entries in TB corresponding to process-space translations.
D_CF	03	Cbox	D-stream quadword Pcache fill
I_CF	02	Cbox	I-stream quadword Pcache fill

Table 12–1 (Cont.): Reference Definitions

Name	Value (hex)	Reference Source	Description
INVAL	01	Cbox	Hexaword invalidate of a Pcache entry
STOP_SPEC_Q	0F	Ibox	Stops processing of specifier references.
NOP	00	Ibox, Ebox, Mbox	No operation

12.3.2 Arbitration Algorithm

Since Cbox references always want to be processed immediately, a validated CBOX_LATCH always causes the Cbox reference to be driven before all other pending references.

A validated RTY_DMISS_LATCH, MME_LATCH, and VAP_LATCH have priority over the EM_ LATCH.

12.4 READS

12.4.1 Generic Read-hit and Read-miss/Cache_fill Sequences

In order to orient the reader as to how memory reads are processed by the Mbox, this section will describe the "vanilla" read sequence. It does not discuss reads which TB_MISS, or otherwise are stalled for a variety of different reasons.

The byte mask generator generates the corresponding byte mask by looking at **M_QUE%S5_VA**<2:0> and **M_QUE%S5_DL**<1:0> and then drives the byte mask onto **M_QUE%S5_BM**<7:0>. Byte mask data is generated on a read operation in order to supply the byte alignment information to the Cbox on an I/O space read.

When a read reference is initiated in the S5 pipe, the address is translated by the TB (assuming the address was virtual) to a physical address during the first half of the S5 cycle. The Pcache initiates a cache lookup sequence using this physical address during the second half of the S5 cycle. This cache access sequence overlaps into the following S6 cycle. During phase four of the S6 cycle, the Pcache determines whether the read reference is present in its array.

If the Pcache determined that the requested data is present, a "cache hit" or "read hit" condition occurs. In this event, the Pcache drives the requested data onto **B%S6_DATA**<63:0>. The signal, **M%CBOX_REF_ENABLE**, is de-asserted to inform the Cbox that it should supply the data from the Pcache.

If the Pcache determined that the requested data is not present, a "cache miss" or "read miss" condition occurs. In this event, the read reference is loaded into the IMISS_LATCH or DMISS_ LATCH (depending on whether the read was I-stream or D-stream) and the Cbox is instructed to continue processing the read by the Mbox assertion of **M%CBOX_REF_ENABLE**. At some point later, the Cbox obtains the requested data. The Cbox will then send four quadwords of data using the I_CF (I-stream cache fill) or D_CF (D-stream cache fill) commands. The four cache fill commands

together are used to fill the entire Pcache block corresponding to the hexaword read address. In the case of D-stream fills, one of the four cache fill command will be qualified with **C%REQ_DQW** indicating that this quadword fill contains the requested D-stream data corresponding to the quadword address of the read. When this fill is encountered, it will be used to supply the requested read data to the Mbox, Ibox and/or Ebox.

If, however, the physical address corresponding to the I_CF or D_CF command falls into I/O space, only one quadword fill is returned and the data is not cached in the Pcache. Only memory data is cached in the Pcache.

Each cache fill command sent to the Mbox is latched in the CBOX_LATCH. Note that neither the entire cache fill address nor the fill data are loaded into the CBOX_LATCH. The address in the IMISS_LATCH or DMISS_LATCH, together with two quadword alignment bits latched in the CBOX_LATCH are used to create the quadword cache fill address when the cache fill command is executed in S5. When the fill operation propagates into S6, the Cbox drives the corresponding cache fill data onto **B%S6_DATA**<63:0> in order for the Pcache to perform the fill.

12.4.1.1 Returning Read Data

Data resulting from a read operation is driven on **B%S6_DATA** by the Pcache (in the cache hit case) or by the Cbox (in the cache miss case). This data is then driven on **M%MD_BUS**<63:0> by the MD_BUS_ROTATOR in right-justified form. The signals **M%VIC_DATA**, **M%IBOX_DATA**, **M%IBOX_IPR_WR**, **M%EBOX_DATA**, **M%MBOX_DATA**, are conditionally asserted with the data to indicate the destination(s) of the data.

In order to return the requested read data to the Ibox and/or Ebox as soon as possible, the Cbox implements a Pcache Data Bypass mechanism. When this mechanism is invoked, the requested read data can be returned one cycle earlier than when the data is driven for the S6 cache fill operation. The bypass mechanism works by having the Mbox inform the Cbox that the next S6 cycle will be idle, and thus the **B%S6_DATA** bus will be available to the Cbox. When the Cbox is informed of the S6 idle cycle, it drives the **B%S6_DATA** bus with the requested read data if read data is currently available (if no read data is available during a bypass cycle, the Cbox drives some indeterminent data and no valid data is bypassed). The read data is then formatted by the MD_BUS_ROTATOR and transferred onto the **M%MD_BUS** to be returned to the Ibox and/or Ebox, qualified by **M%VIC_DATA**, **M%IBOX_DATA**, and/or **M%EBOX_DATA**.

12.4.2 D-stream Read Processing

A DREAD_LOCK command always forces a Pcache read miss sequence regardless of whether the referenced data was actually stored in the Pcache. This is necessary in order that the read propagate out to the Cbox so that the memory lock/unlock protocols can be properly processed.

12.4.3 I/O Space Reads

I/O space reads are defined as reads which address I/O space. Therefore, a read is an I/O read when the physical address bits, addr<31:29>, are set. I/O space reads are treated by the Mbox in exactly the same way as any other read, except for the following differences:

- I/O space data is never cached in the Pcache. Therefore, an I/O space read always generates a read-miss sequence and causes the Cbox to process the reference.

- Unlike, a memory space miss sequence, which returns a hexaword of data via four I_CF or D_CF commands, an I/O space read returns only one piece of data via one I_CF or D_CF command. Thus the Cbox always asserts **C%LAST_FILL** on the first and only I_CF or D_CF I/O space operation. If the I/O space read is D-stream, the returned D_CF data is always less than or equal to a longword in length.
- I/O space D-stream reads are never prefetched ahead of Ebox execution. An I/O space D-stream read issued from the Ibox is only processed when the Ebox is known to be stalling on that particular I/O space read.

NVAX RESTRICTION

I-stream I/O space reads must return a quadword of data. Execution of an I-stream I/O space read which does not return a quadword of data is unpredicatable.

12.5 WRITES

All writes are initiated by the Mbox on behalf of the Ebox. The Ebox microcode is capable of generating write references with data lengths of byte, word, longword, or quadword. With the exception of cross-page checks, the Mbox treats quadword write references as longword write references because the Ebox datapath only supplies a longword of data per cycle. Ebox writes can be unaligned.

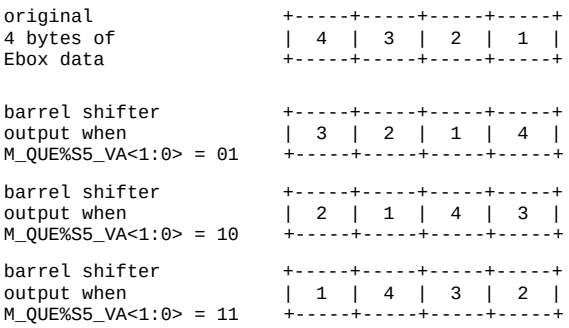
The Mbox performs the following functions during a write reference:

- Memory Management checks: The Mbox checks to be sure the page or pages referenced have the appropriate write access and that the valid virtual address translations are available. (See Section 12.12)
- The supplied data is properly rotated to the memory aligned longword boundary.
- Byte Mask Generation: The Mbox generates the byte mask of the write reference by examining the write address and the data length of the reference.
- Pcache writes: The Pcache is a write-through cache. Therefore, writes are only written into the Pcache if the write address matches a validated Pcache tag entry.

  The one exception to this rule is when the Pcache is configured in force D-stream hit mode. In this mode, the data is always written to the Pcache regardless of whether the tag matches or mismatches.
- All write references which pass memory management checks are transferred to the Cbox via **B%S6_DATA**<63:0>. The Cbox is responsible for processing writes in the Bcache and for controlling the protocols related to the write-back memory subsystem.

When write data is latched in the EM_LATCH, the 4-way byte barrel shifter associated with the EM_LATCH rotates the EM_LATCH data into proper alignment based on the lower two bits of the corresponding address. The diagram below illustrates the barrel shifter function:

Figure 12–2: Barrel Shifter Function



The result of this data rotation is that all bytes of data are now in the correct byte positions relative to memory longword boundaries.

When write data is driven from the EM_LATCH, **M_QUE%S5_DATA**<31:0> is driven by the output of the barrel shifter so that data will always be properly aligned to memory longword addresses.

Note that, while the **M%M_QUE%S5_DATA** bus is a longword wide, the **B%S6_DATA** bus is a quadword wide. **B%S6_DATA** is a quadword wide due to the quadword Pcache access size. The quadword access size facilitates Pcache and VIC fills. However for all writes, at most half of **B%S6_DATA**<63:0> is ever used to write the Pcache since all write commands modify a longword or less of data. When a write reference propagates from S5 to S6, the longword aligned data on **M_QUE%S5_DATA**<31:0> is transferred onto both the upper and lower halves of **B%S6_DATA**<63:0> to guarantee that the data is also quadword aligned to the Pcache and Cbox. The byte mask corresponding to the reference will control which bytes of **B%S6_DATA**<63:0> actually get written into the Pcache or Bcache.

Write references are formed through two distinct mechanisms described below.

12.5.1 Writes to I/O Space

I/O space writes are defined as a write command which addresses I/O space. Therefore, a write is an I/O space write when the physical address bits, addr<31:29>, are set. I/O space writes are treated by the Mbox in exactly the same way as any other write, except for the following differences:

- I/O space data is never cached in the Pcache; therefore, an I/O space write always misses in the Pcache.

12.6 IPR PROCESSING

12.6.1 MBOX IPRs

The Mbox maintains the following internal processor registers:

Table 12–2: Mbox IPRs

Register Name	IPR Address (in hex)
MP0BR (Mbox P0 Base Register)[1]	E0
MP0LR (Mbox P0 Length Register)[1]	E1
MP1BR (Mbox P1 Base Register)[1]	E2
MP1LR (Mbox P1 Length Register)[1]	E3
MSBR (Mbox System Base Register)[1]	E4
MSLR (Mbox System Length Register)[1]	E5
MMAPEN (Map Enable Bit)[1]	E6
PAMODE (Address Mode)	E7
MMEADR (MME Faulting Address Register)[1]	E8
MMEPTE (PTE Address Register)[1]	E9
MMESTS (status of memory management exception)[1]	EA
TBADR (address of reference causing TB parity error)	EC
TBSTS (status of TB parity error)	ED
PCADR (address of reference causing Pcache parity error)	F2
PCSTS (status of Pcache parity error and PTE hard errors)	F4
PCCTL (control state of Pcache operation)	F8
PCTAG	01800000..01801FE0
PCDAP	01C00000..01C01FF8

[1]Testability and diagnostic use only; not for software use in normal operation.

The first thirteen IPRs listed above (memory management IPRs) are stored in the S5 pipe in the register file of the MME_DATAPATH. All other IPRs are stored in the S6 pipe. Note that when an Mbox IPR, other than a Pcache tag, is addressed, the actual IPR address is received on **M_QUE%S5_VA** (the table above is written such that all addresses start at bit<0>).

The following is the format description of each Mbox IPR.

Figure 12–3: MP0BR Register

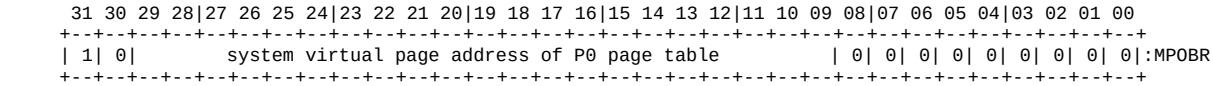


Figure 12–4: MP0LR Register

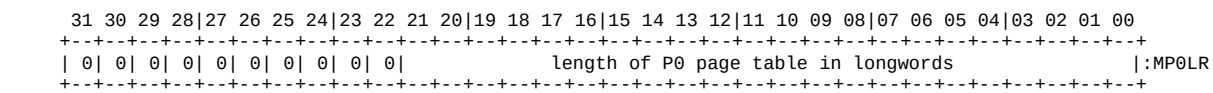


Figure 12–5: MP1BR Register

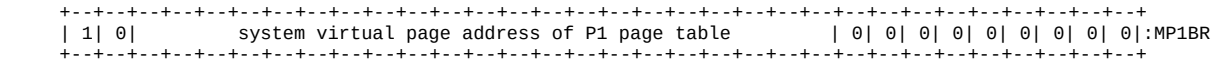


Figure 12–6: MP1LR Register

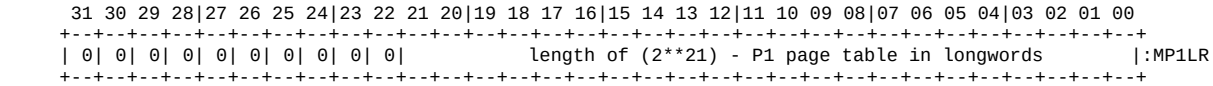


Figure 12–7: MSBR Register

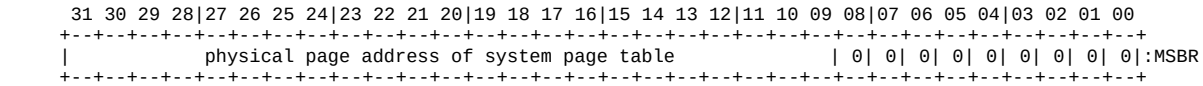


Figure 12–8: MSLR Register

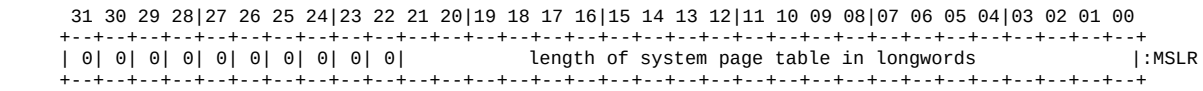


Figure 12–9: MMAPEN Register

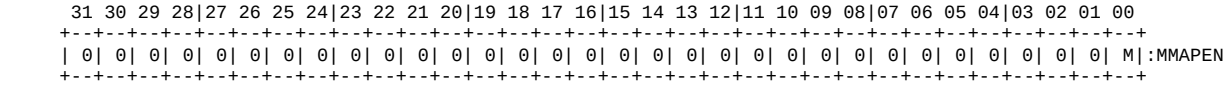


Table 12–3: MMAPEN Definition

Name	Bit(s)	Type	Description
M	0	RW,0	When 0, disables Mbox memory management. When 1, enables Mbox memory management.

Figure 12–10: PAMODE Register

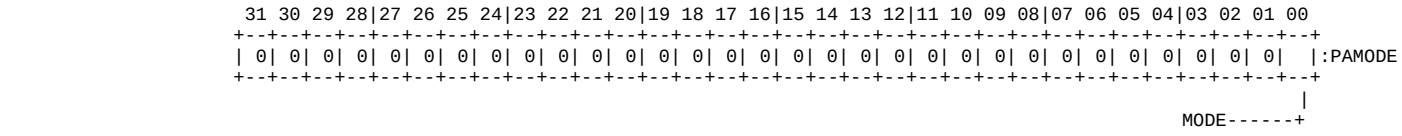


Table 12–4: PAMODE Definition

Name	Bit(s)	Type	Description
MODE	0	RW,0	When 0, maps addresses from a 30-bit physical address space. When 1, maps addresses from a 32-bit physical address space.

Figure 12–11: MMEADR Register

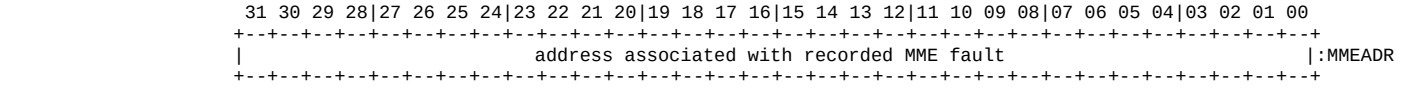


Figure 12–12: MMEPTE Register

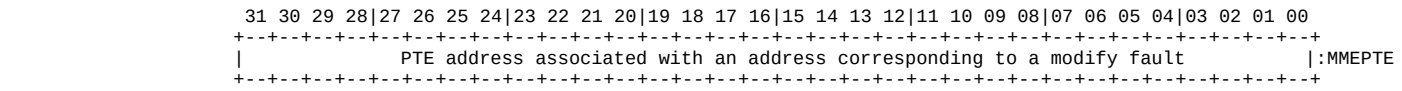


Figure 12–13: MMESTS Register

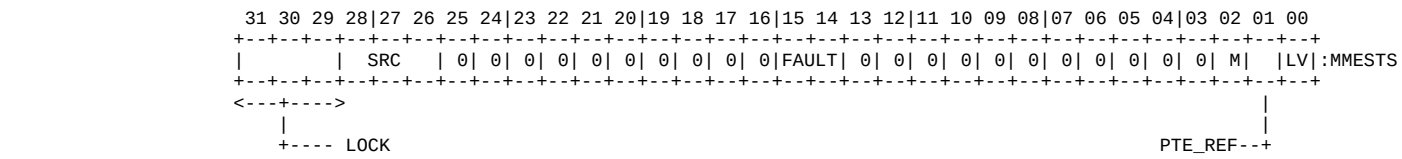


Table 12–5: MMESTS Register Definition

Name	Bit(s)	Type	Description
LV	0	RO,0	Indicates ACV fault occurred due to length violation.
PTE_REF	1	RO	Indicates ACV/TNV fault occurred on PTE reference corresponding to MMEADR.
M	2	RO	Indicates corresponding reference had write or modify intent.
FAULT	15:14	RO	Indicates nature of memory management fault. See Fault bit encodings below
SRC	28:26	RO	Complemented shadow copy of LOCK bits. However, the SRC bits do not get reset when the LOCK bits are cleared.
LOCK	31:29	RO	Indicates the lock status of MMESTS. See LOCK encodings below. This field is cleared on E%FLUSH_MBOX.

Table 12–6: FAULT Encodings

Defined FAULT values (binary)	Definition
01	ACV Fault. This is the highest priority fault in the presence of multiple simultaneous faults.
10	TNV Fault. This is the next highest priority fault.
11	M=0 Fault. This is the lowest priority fault.

Table 12–7: LOCK Encodings

Defined LOCK values (binary)	Definition
000	MMESTS, MMEADR and MMEPTE are unlocked.
001	valid IREAD fault is stored (no other IREAD fault can overwrite MMESTS, MMEADR, or MMEPTE).
011	valid Ibox specifier fault is stored (only an Ebox reference fault can overwrite MMESTS, MMEADR, or MMEPTE).
111	valid Ebox fault is stored (MMESTS, MMEADR, and MMEPTE are completely locked).

Note that the encodings for the SRC bits are the complemented version of the the LOCK bits. Thus, for example, a fully locked SRC encoding is 000.

Figure 12–14: TBADR Register

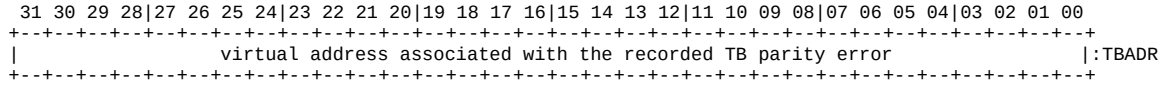


Figure 12–15: TBSTS Register

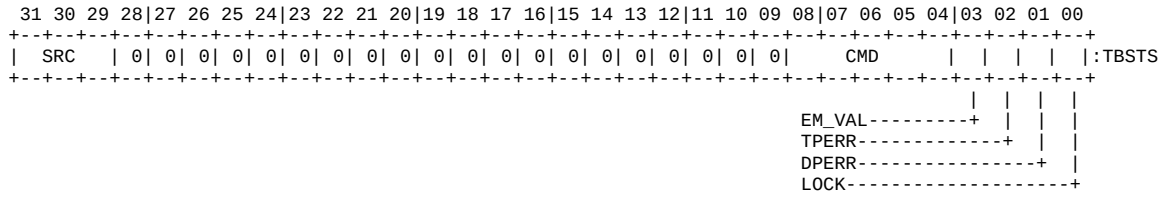


Table 12–8: TBSTS Description

Name	Bit(s)	Type	Description
LOCK	0	WC,0	Lock Bit. When set, validates TBSTS contents and prevents any other field from further modification. When clear, indicates that no TB parity error has been recorded and allows TBSTS and TBADR to be updated.
DPERR	1	RO	Data Error Bit. When set, indicates a TB data parity error.
TPERR	2	RO	Tag Error Bit. When set, indicates a TB tag parity error.
EM_VAL	3	RO	EM_LATCH valid bit. Indicates if EM_LATCH was valid at the time of the error TB parity error detection. This helps the software error handler determine if a write operation may have been lost due to the TB parity error.
CMD	8:4	RO	S5 command corresponding to TB parity error.
SRC	31:29	RO	Indicates the original source of the reference causing TB parity error.

Table 12–9: SRC Encodings

Defined SRC values	Definition
111	valid Mbox reference error is stored
110	valid IREAD error is stored
100	valid Ibox specifier reference error is stored

Table 12–9 (Cont.): SRC Encodings

Defined SRC values	Definition
000	valid Ebox reference error is stored

Figure 12–16: PCADR Register

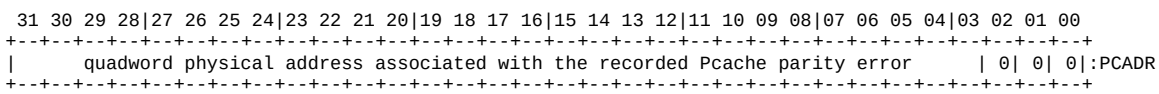


Figure 12–17: PCSTS Register

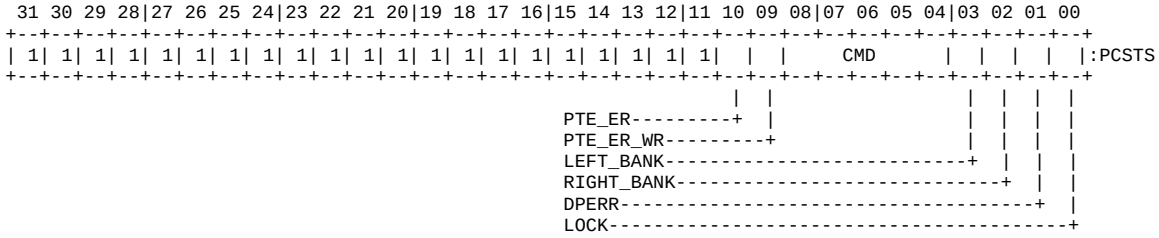


Table 12–10: PCSTS Description

Name	Bit(s)	Type	Description
LOCK	0	WC,0	Lock Bit. When set, validates PCSTS<8:1> contents and prevents modification of these fields. When clear, invalidates PCSTS<8:1> and allows these fields and PCADR to be updated.
DPERR	1	RO	Data Error Bit. When set, indicates a Pcache data parity error.
RIGHT_BANK	2	RO	Right Bank Tag Error Bit. When set, indicates a Pcache tag parity error on the right bank.
LEFT_BANK	3	RO	Left Bank Tag Error Bit. When set, indicates a Pcache tag parity error on the left bank.
CMD	8:4	RO	S6 command corresponding to Pcache parity error.
PTE_ER_WR	9	WC,0	Indicates a hard error on a PTE DREAD which resulted from a TB miss on a WRITE or WRITE_UNLOCK.
PTE_ER	10	WC,0	Indicates a hard error on a PTE DREAD.

Note that the state of PCSTS<31:11> are "don't cares" during an IPR write operation.

Figure 12–18: PCCTL Register

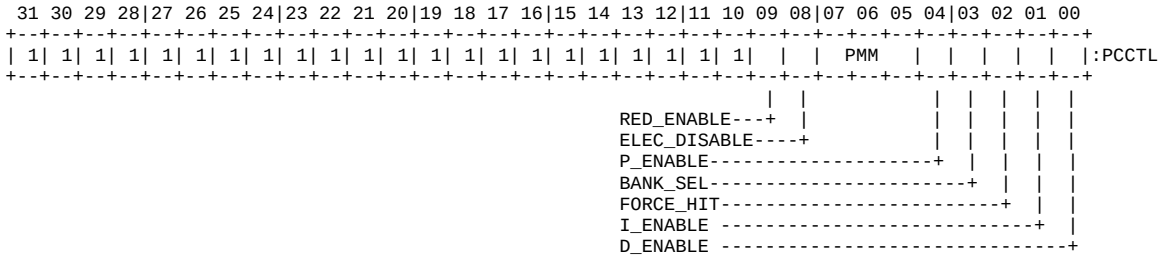


Table 12–11: PCCTL Definition

Name	Bit(s)	Type	Description
D_ENABLE	0	RW,0	When set, enables Pcache for all INVAL operations and for all D-stream read/write/fill operations, qualified by other control bits. When clear, forces a Pcache miss on all Pcache D-stream read/write/fill operations. Note, however, that an ACV/TNV/M=0 condition overrides a desasserted D_ENABLE in that it will force a Pcache hit condition with D_ENABLE=0.
I_ENABLE	1	RW,0	When set, enables Pcache processing of INVAL, IREAD and I_CF commands. When clear, forces a Pcache miss on IREAD operations and prevents state modification due to an I_CF operation. Note, however, that an ACV/TNV/M=0 condition overrides a desasserted I_ENABLE in that it will force a Pcache hit condition with I_ENABLE=0.
FORCE_HIT	2	RW,0	When set, forces a Pcache hit on all reads and writes when Pcache is enabled for I or D-stream operation.
BANK_SEL	3	RW,0	When set with FORCE_HIT=1, selects the "right bank" of the addressed Pcache index. When clear with FORCE_HIT=1, selects the "left bank" of the addressed Pcache index. BANK_SEL is a don't care when FORCE_HIT=0. NOTE: BANK_SEL never affects bank selection during IPR reads and IPR writes to the Pcache tags or Pcache data parity bits; bank selection for these commands is always determined by the specified IPR address.
P_ENABLE	4	RW,0	When set, enables detection of Pcache tag and data parity errors. When deasserted, disables Pcache parity error detection.
PMM	7:5	RW,0	Specifies Mbox performance monitor mode (see Section 12.17). Note that this field does not control or affect the operation of the Pcache in any way. PMM is placed in PCCTL for the convenience of the hardware implementation.

Table 12–11 (Cont.): PCCTL Definition

Name	Bit(s)	Type	Description
ELEC_DISABLE	8	RW,0	When set, the Pcache is disabled electrically to reduce power dissipation. NOTE: This bit should only be set when the Pcache is functionally turned off by the deassertion of both I_ENABLE and D_ENABLE. UNPREDICTABLE operation will result when this bit is set when either I_ENABLE or D_ENABLE is also set. Also note that Pcache tag or parity IPRs will not function properly when this bit is unconditionally set.
RED_ENABLE	9	RO	When set, indicates that one or more Pcache redundancy elements are enabled (see Section 12.11 for more information).

Note that the state of PCCTL<31:10> are "don't cares" during an IPR write operation.

Figure 12–19: PCTAG Register

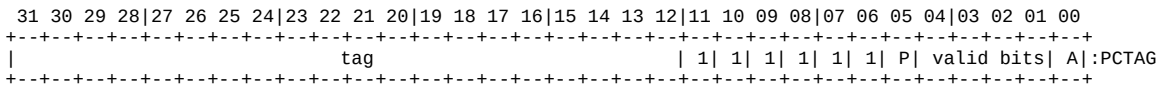


Table 12–12: Pcache Tag IPR Format

Name	Bit(s)	Type	Description
A	0	RW	Allocation Bit corresponding to index of this tag.
valid bits	4:1	RW	Valid Bits corresponding to the four data subblocks. PCTAG<4> corresponds to uppermost quadword in block. PCTAG<1> corresponds to lowermost quadword in block.
P	5	RW	Even Tag Parity
tag	31:12	RW	Tag Data

Note that the state of PCTAG<11:6> are "don't cares" during an IPR write operation.

Figure 12–20: PCDAP Register

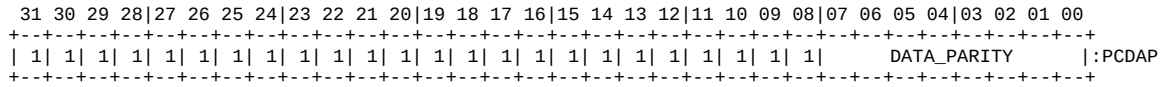


Table 12–13: Pcache Data Parity IPR Format

Name	Bit(s)	Type	Description
DATA_PARITY	7:0	RW	Even byte parity corresponding to addressed quadword of data. Bit n represents parity for byte n of addressed quadword.

Note that the state of PCDAP<31:8> are "don't cares" during an IPR write operation.

12.7 INVALIDATES

The Cbox initiates an invalidate by PASSING iAdr<12:5> and InvReq<1:0> RECEIVED FROM SYSTEM LOGIC qualified by the INVAL command. The INVAL command is latched by the Mbox in the CBOX_LATCH. The set and index specified are unconditional invalidated.

Execution of an INVAL command guarantees that data corresponding to the specified hexaword address will not be valid in the Pcache. THE SYSTEM LOGIC IS RESPONSIBLE FOR PRIMARY CACHE COHERENCY IN NVAX Plus. The block valid bit and the four corresponding subblock valid bits are cleared to guarantee that any subsequent Pcache accesses of this hexaword will miss until this hexaword is re-validated by a subsequent Pcache fill sequence. If a cache fill sequence to the same INDEX AND SET is in progress when the INVAL is executed, a bit in the corresponding MISS_LATCH is set to inhibit any further cache fills from loading data or validating data for this cache block.

Also note that an assertion of **C%CBOX_HARD_ERR** during a cache fill command causes the cache fill operation to be processed as if it were an INVAL operation.

12.7.1 ABORTING REFERENCES

The Mbox abort operation is used to cancel the current S5 operation. When an abort is executed, the S5 state, which would normally be updated due to execution of the current S5 reference, is not updated. The aborted S5 reference is not propagated into S6. Instead, a NOP is introduced into the S6 pipe. In effect, an aborted S5 reference is equivalent to a NOP command being executed in S5.

Note that the abort operation should be viewed as only cancelling the current execution of a reference. In most cases, aborting an operation does not invalidate the existence of the corresponding reference, which will still be stored in one of the reference sources and retried at a later point.

The abort operation is executed when ABORT is asserted. The following changes to Mbox state are inhibited during the cycle in which ABORT is asserted:

- The reference source which drove the aborted command into S5 does not invalidate the corresponding command. Thus, the reference still exists to be retried during a subsequent cycle.

NOTE

There are two exceptions to this rule. The CBOX_LATCH is always invalidated after it drives a command into S5. The EM_LATCH will be invalidated if the Ebox has explicitly requested it to be (via the **E%EM_ABORT** signal).

- Loading the PA_QUEUE with a DEST_ADDR or DREAD_MODIFY command is inhibited. Emptying the PA_QUEUE when a STORE command is driven in S5 is inhibited.
- If the unaligned detection logic detected an unaligned reference during the aborted cycle, the VAP_LATCH is not validated to contain the second portion of the unaligned sequence.

12.8 Conditions for Aborting References

In general, references are aborted for five reasons:

- The reference is aborted to prevent a reference order restriction from occurring.
- The reference is aborted because insufficient hardware resources are available to complete processing of the current command.
- The reference is aborted because a memory management operation must be performed prior to execution of the current reference.
- The reference is aborted in order to avoid a deadlock condition related to unaligned references.
- The reference is aborted due to an external flush condition.

12.9 READ_LOCK/WRITE_UNLOCK

Once a READ_LOCK command has been passed to the Cbox, the Cbox can not process any subsequent I-stream read references, and should not receive any D-stream references besides the IPR read of STxC pass/fail or a retry of the read_lock, until a STxC pass signal is received from the CBOX.

This is accomplished by the arbitration logic by disabling IREF_LATCH selection once a DREAD_LOCK command has successfully been retired from the S5 pipe. Thus, no IREAD TB_MISS can occur between the READ_LOCK and STxC pass, thus avoiding D-Stream references not part of the interlock sequence.

The arbitration logic will re-enable IREF_LATCH selection on either of the following two conditions:

1. The STxC IPR is read and the condition indicates pass. This will cause the Cbox to resume I-stream read processing.
2. **E%FLUSH_MBOX** is asserted by the Ebox due to a hard error. This condition should occur much more infrequently than the above condition because a WRITE_UNLOCK must normally be issued after a READ_LOCK. However, if an error occurred sometime between the READ_LOCK and STxC Pass, a hard error microtrap will result preventing a WRITE_UNLOCK

from being issued. The microtrap will generate **E%FLUSH_MBOX** which re-enables IREF_LATCH selection because no WRITE_UNLOCK will follow.

**Note that the Cbox state, which prevents subsequent I-stream reads from being processed before the WRITE_UNLOCK, will be cleared by an IPR_WRITE during the error handler. **

Note that Ibox processing will have been halted prior to the READ_LOCK/WRITE_UNLOCK sequence. The Ebox microcode will never issue a D-stream read in the middle of a READ_LOCK/WRITE_UNLOCK sequence.

12.10 Pcache Replacement Algorithm

Each line of Pcache contains an allocation bit which is used to indicate which bank (left or right) should be used for the next fill sequence of that index. This results in a "not last used" allocation to the Pcache sets.

When an invalidate clears the valid bits of a particular tag within an index, it only makes sense to set the allocation bit to point to the bank select used during the invalidate regardless of which bank was last allocated. By doing so, we guarantee that the next allocated block within the index will not displace any valid tag because the allocation bit points to the tag that was just invalidated.

For systems that require the Pcache to function as direct mapped, the allocate bit during a fill sequence is ignored, and the fill follows address[12].

12.11 Pcache Redundancy Logic

Due to the extreme density of the Pcache array, the Pcache has a high susceptibility to manufacturing defects. As a result, redundancy logic was designed in order to provide a mechanism which would allow the Pcache to function correctly in the presence of a small number of manufacturing defects. Refer to NVAX CPU Chip Functional Specification for the description of the PCache Redundancy feature.

12.12 MEMORY MANAGEMENT

The Mbox, the Ebox microcode, and the VMS memory management software implement VAX memory management. The Mbox performs the hardware memory management functions necessary to process most references in a quick efficient manner. The operating system software performs all other functions. For a description of the hardware end of VAX memory management, the reader is referred to the Memory Management chapter of the "VAX Architecture Standard" (DEC STD 032). For a complete description of the software end of VAX/VMS memory management, the reader is referred to the Memory Management chapters of "VAX/VMS Internals and Data Structures".

The Mbox is responsible for the following memory management functions:

- Performing virtual-to-physical address translations.
- Maintaining a cache of PTEs to perform the quick translations.
- Performing access mode checks on memory references.
- Performing TNV checks on memory references.

- Performing M=0 checks on memory references.
- Directly or indirectly invoking a software memory management exception handler due to ACV (Access Violation) or TNV (Translation not Valid) or M=0 faults.
- Detecting cross-page conditions and performing the corresponding access mode checks.

12.12.1 ACV/TNV/M=0 Fault Handling:

In order for an ACV, TNV, or M=0 fault to be processed, the following steps must occur:

1. The Mbox must detect the ACV/TNV/M=0 condition.
2. The Ebox microcode must be invoked to start processing the condition.
3. The Ebox microcode must probe Mbox state in order to determine which fault occurred and how it should be processed.
4. The Ebox microcode must service the fault condition directly, or it must invoke an operating system memory management service routine to service the fault.
5. If the memory management fault was not fatal to the process, normal instruction execution resumes by restarting the instruction corresponding to the memory management fault after servicing the fault.

12.12.2 ACV detection:

The protection field of a PTE indicates the authorized access rights for each execution mode. When a reference causes the TB to access a PTE, the protection field of the PTE corresponding to the reference is driven out of the TB. The ACV (Access Violation) detection logic uses the PTE protection field, **M_QUE%S5_AT**<1:0>, and the appropriate CPU execution mode from the Ebox (i.e. user, supervisor, executive, kernel) to detect access violations. If, for example, the protection field indicates a "read-only" access in user mode, the CPU execution mode specifies user mode, and **M_QUE%S5_AT**<1:0> indicates write access, then an ACV condition is flagged since a write reference is not allowed to this page in user mode.

A 2:1 MUX controls the source of the CPU execution mode. The CPU execution mode information is normally taken directly from the current mode field of the PSL (**PSL**<25:24>). On PROBE references, however, the CPU execution mode is driven from **MMGT_MODE**<1:0> in order to check for ACV conditions for an execution mode which the CPU is not currently in.

An ACV condition is also generated when a PTE reference fails to satisfy the page length check corresponding to the virtual space of the reference or when the virtual reference falls into S1 space. A virtual address in S1 space is reported as an ACV length violation.

An ACV check is also performed on the protection field of all PTEs which have just been sent to the Mbox due to an earlier Mbox DREAD issued during the TB_MISS sequence.

ACV protection and length checks are performed on all Ibox and Ebox references and on all MME_CHKs. ACV page length checks are performed on all PTE addresses. However, ACV protection checks are never performed on PTE read references generated by the Mbox.

Note that the ACV protection condition is disabled from occurring during any cycle where the reference is aborted.

When an ACV condition occurs, the MME_SEQ is invoked to execute the ACV/TNV/M=0 sequence. ACV checks only occur on virtual addresses when memory management is enabled and when the reference indicates that memory management checks should be done (i.e. **M_QUE%S5_QUAL** = 1).

12.12.2.1 TNV detection

When the PTE valid bit is clear, it indicates that the corresponding PTE page frame address translation is not valid. This is called a Translation Not Valid Fault (TNV). TNV detection only occurs during the TB_MISS sequence when the Mbox receives PTE data from the Pcache or Cbox such that the PTE valid bit (PTE<31>) is clear. When a TNV fault is detected, the MME_ SEQ interrupts the TB_MISS sequence and invokes the ACV/TNV/M=0 sequence. By doing so, the invalid PTE is never cached in the TB and a memory management fault is recorded (See Section 12.12.2.3 on recording memory management faults).

12.12.2.2 M=0 detection:

When a virtual reference causes the TB to access a PTE, the modify bit of the PTE is read out of the TB. A cleared modify bit indicates that the corresponding page has not been written to. If the valid bit of the PTE is set, and the modify bit is clear and the access type of the S5 reference indicates an intention to modify the page (e.g. write or modify OR VSTR access type), then the Mbox must initiate the proper sequence of events to process this "M=0" condition. The M=0 check is performed when memory management is enabled and a virtual reference hits in the TB.

Note that the M=0 condition is disabled from occurring during any cycle where the reference is aborted.

12.12.2.3 Recording ACV/TNV/M=0 Faults

In order for the microcode to determine the nature of the memory management fault detected by the Mbox, the Mbox must record the necessary fault information. The fault information is recorded in Mbox IPRs which can be read by Ebox microcode. The fault information is stored in three of the registers in the MME register file which are accessible to microcode by IPR reads and writes:

- The MMEADR register stores the virtual address associated with the ACV, TNV or M=0 fault. As per SRM requirements, if the ACV/TNV fault occurred by referencing a PTE during a TB miss sequence, the MMEADR stores the original address and not the PTE address.
- The MMEPTE register stores the virtual or physical address of the Page Table Entry corresponding to a virtual reference upon which an M=0 condition has been detected.
- The MMESTS register stores state which indicates to the microcode the context and type of fault corresponding to the ACV/TNV/M=0 condition. The format of MMESTS is shown below:

Due to the macropipeline design, the MMEADR, MMEPTE and MMESTS registers must be conditionally loaded in a prioritized fashion. These registers are loaded depending on the relative states of their current contents and on the context of the current fault. If the MMESTS register is empty, the current fault state is always loaded. If the MMESTS register contains a valid fault condition, the MMEADR, MMEPTE and MMESTS are only loaded if the current fault is associated with a pipe stage further along in the pipe than the stage corresponding to the stored MMESTS state. This loading priority is necessary because these memory management faults must be reported within the context of the execution of the instruction they are associated with. A fault detected on an Ebox reference is loaded provided that another Ebox reference fault is

not already loaded. Faults detected on Ibox specifier references are only loaded if no Ebox or Ibox specifier reference fault is currently stored. Faults on Ibox I-stream references are only loaded if the MMESTS register is empty. In effect, the MMESTS register captures the first memory management exception that will be associated with Ebox execution. Stated differently, it captures the fault which occurs farthest along in the macropipeline.

The LOCK field of MMESTS specifies the source of the faulting reference currently stored in MMESTS. Thus, the decision to load another faulting reference into MMESTS is made by examining the bits of the LOCK field.

The FAULT field is set in a prioritized manner. That is, an ACV fault takes precedence over a TNV or M=0 fault. A TNV fault takes precedence over an M=0 fault. Therefore, if multiple pending fault conditions are true, only the fault condition with the highest priority is reported in the MMESTS register.

When the Ebox starts the memory management exception microflow, it issues an IPR_RD to the MMESTS to determine the nature of the memory management fault. The MMESTS register is automatically unlocked by resetting the LOCK field when the **E%FLUSH_MBOX** signal is asserted by the Ebox.

12.13 MBOX ERROR HANDLING

Mbox plays a role in the processing of the following types of errors:

- TB tag parity errors.
- TB data parity errors.
- Pcache tag parity errors.
- Pcache data parity errors.
- Errors encountered by the Cbox while processing a memory read, I/O space read, or IPR_RD which were transferred from the Mbox to the Cbox. Note that these errors could originate from the Bcache, or memory subsystem.

All other possible errors are handled without Mbox involvement.

12.13.1 Recording Mbox errors

The Mbox contains four error registers. Two are used to record TB parity errors and the other two are used to record Pcache parity errors.

12.13.1.1 TBSTS and TBADR

When a TB parity error is detected with LOCK=0, TBADR is loaded with the virtual address which caused the TB parity error, and all fields of TBSTS are updated to record the nature of the TB parity error. Note that both the TPERR and DPERR bits can be set at the same time if these two error conditions occurred during the same cycle. When a TB parity error is recorded, the LOCK bit is set to validate the contents of both TBSTS and TBADR registers. When LOCK is set, all bits of both registers are frozen and cannot be changed until the LOCK bit is cleared. Thus, any subsequent error is not recorded if LOCK=1.

When the operating system error handler is invoked, TBSTS and TBADR will be read through an IPR_RD command in order to determine if any TB parity errors were recorded. If the state of the LOCK bit was read to be a zero, then no error has occurred and the remaining state information in these two registers is invalid. If the LOCK bit was found to be set, then the remaining error state of these two registers characterizes the nature of the recorded error.

Once the error handler has read these registers, it re-enables TBSTS to record any new errors by clearing the LOCK bit. Clearing the LOCK bit is accomplished by writing a "1" to LOCK through an IPR_WR operation.

12.13.1.2 PCSTS and PCADR

The PCSTS and PCADR record Pcache tag and data parity errors. The function and operation of these registers is identical to the TBSTS and TBADR registers except that the PCADR stores physical quadword addresses rather than virtual byte addresses, and it also records PTE hard error events. The definitions of these registers are shown in Figure 12–16 and Figure 12–17. Note however, that when **PCSTS** is set, Pcache memory reads, writes and invalidates are disabled.

12.13.2 Mbox Error Processing

12.13.2.1 Processing Cbox errors on Mbox-initiated read-like sequences

The Cbox detects errors that occur in the Bcache, or memory subsystem. When the Cbox detects one of these errors, and it is associated with an Mbox-initiated reference that requires data to be returned (e.g. memory read, I/O space read, or IPR read), the Mbox must transfer the error status of the reference back to the destination corresponding to the reference. The Mbox never records a Cbox-detected error in Mbox error registers because the error is logged in Cbox error registers.

12.13.2.1.1 Cbox-detected ECC errors

The Cbox returns requested data through a I_CF or D_CF command to the Mbox while simultaneously checking the error-correction code to check for a possible Bcache error. If an ECC error is found, the Cbox asserts **C%CBOX_ECC_ERR**. This causes the Mbox to latch a NOP in the CBOX_LATCH rather than the cache fill. As a result, the Mbox does not perform any Pcache state updates resulting from the bad data nor does it assert **M%VIC_DATA**, **M%IBOX_DATA**, **M%EBOX_DATA**, or **M%MBOX_DATA** to indicate the presence of valid data.

C%CBOX_ECC_ERR IS ALSO USED BY THE CBOX LOGIC AS A LATE ABORT FOR FILL DATA DUE TO A MISS OR CACHE TAG COMPARE NOT VALID DUE TO SYSTEM LOGIC OWNING THE CACHE DURING THE READ/PROBE CYCLE.

During subsequent cycles, the Cbox will determine if the ECC error is correctable or not. If it is, the data will be corrected and returned. If the data is not correctable, a Cbox-detected hard error has occurred and will be dealt with as described below.

12.13.2.1.2 Cbox-detected hard errors on requested fill data

If the Cbox has determined that the requested data cannot be returned for some reason, the Cbox drives a cache fill command qualified by **C%CBOX_HARD_ERR**. When this happens, the Mbox performs the following actions:

1. The assertion of **C%CBOX_HARD_ERR** indicates to the Mbox that the cache fill data is invalid. Thus, the Mbox returns the invalid data on the **M%MD_BUS** in the same manner that all data is returned except that the data is further qualified by **M%HARD_ERR**. **M%HARD_ERR** informs the receiver that the data is invalid and that the requested data cannot be returned due to a hard error.
2. Once the Cbox detects a hard error on the requested data, the Cbox immediately terminates the pending fill sequence by the assertion of **C%LAST_FILL**. Thus, no further data corresponding to the same fill sequence will be returned and the Mbox fill sequence corresponding to the error is terminated by invalidating the corresponding MISS_LATCH.
3. An I_CF or D_CF command which is qualified by **C%CBOX_HARD_ERR** is interpreted by the Pcache as an INVAL command. Thus the invalid data is not filled in the Pcache.

12.13.2.1.3 Cbox-detected hard errors on non-requested fill data

The Cbox performs the same actions as described above to indicate the presence of a hard error regardless of whether the data is the requested data or just one of the other three pieces of fill data for the corresponding Pcache block. If the data is non-requested fill data, the Mbox performs the following actions:

1. Once the Cbox detects a hard error on the non-requested data, the Cbox immediately terminates the pending fill sequence by the assertion of **C%LAST_FILL**. Thus, no further data corresponding to the same fill sequence will be returned and the Mbox fill sequence corresponding to the error is terminated by invalidating the corresponding MISS_LATCH.
2. An I_CF or D_CF command which is qualified by **C%CBOX_HARD_ERR** is interpreted by the Pcache as an INVAL command. Thus the invalid fill data is not filled in the Pcache and all previous fills to the same block are invalidated. This is necessary in order to maintain coherency between the Pcache and Bcache because a Bcache data block will only be validated if all the data within the block is error-free.

12.13.2.2 Mbox Error Processing Matrix

The following table summaries all Mbox error handling. A blank entry in the table means that the corresponding error cannot occur for the given reference.

Table 12–14: Mbox Error Handling Matrix

Command	TB tag parity error	TB data parity error	Pcache tag parity error	Pcache data parity error	Cbox hard error
Ibox references					
IREAD	A	A	B	D	F
DREAD	A	A	B	D	F

Table 12–14 (Cont.): Mbox Error Handling Matrix

Command	TB tag parity error	TB data parity error	Pcache tag parity error	Pcache data parity error	Cbox hard error
DREAD_MODIFY	A	A	B	D	F
DEST_ADDR	A	A			
STOP_SPEC_Q					
Ebox references					
DREAD	A	A	B	D	F
DREAD_LOCK	A	A	B		F
STORE			C		
WRITE	A	A	C		
WRITE_UNLOCK	A	A	C		
IPR_RD (to Pcache)					
IPR_RD (non-Mbox)					F
IPR_WR (to Pcache)					
IPR_WR (non-Mbox)					
PROBE	A	A			
MME_CHK	A	A			
TB_TAG_FILL					
TB_PTE_FILL					
TBIS					
TBIP					
TBIA					
LOAD_PC					
Mbox references					
PTE DREAD	A	A	B	D	G
TB_TAG_FILL					
TB_PTE_FILL	A				
IPR_DATA					
MME_CHK	A	A			

Table 12–14 (Cont.): Mbox Error Handling Matrix

Command	TB tag parity error	TB data parity error	Pcache tag parity error	Pcache data parity error	Cbox hard error
Cbox references					
INVAL			E		
D_CF					H
I_CF					H

LEGEND:

A.

- Mbox microtraps Ebox by assertion of **M%TB_PERR_TRAP** during cycle error was detected.
- The faulting reference and all pending Ibox and Ebox references are blown away.
- TBIA command is issued to invalidate entire TB.
- TBSTS and TBADR are updated appropriately.

B.

- A Pcache miss condition is forced to occur on this read reference causing the assertion of **M%CBOX_REF_ENABLE**. This instructs the Cbox to continue processing the read reference.
- **M%MBOX_S_ERR** is asserted to post a soft error interrupt.
- PCSTS and PCADR are updated appropriately (a side effect of this operation turns off the Pcache).

C.

- The Cbox continues to process the write reference, as is done on all write operations regardless of a Pcache parity error.
- **M%MBOX_S_ERR** is asserted to post a soft error interrupt.
- PCSTS and PCADR are updated appropriately (a side effect of this operation turns off the Pcache).

D.

- **M%CBOX_LATE_EN** is asserted to instruct the Cbox to continue processing the reference which caused the Pcache parity error.
- **M%MBOX_S_ERR** is asserted to post a soft error interrupt.
- PCSTS and PCADR are updated appropriately (a side effect of this operation turns off the Pcache).

E.

- The invalidate operation takes place in spite of the tag parity error because the invalidate is only a function of matching all tag bits.
- **M%MBOX_S_ERR** is asserted to post a soft error interrupt.
- PCSTS and PCADR are updated appropriately (a side effect of this operation turns off the Pcache).

F.

- The Cbox indicated a hard error for a non-PTE read or IPR_RD operation by the assertion of **C%CBOX_HARD_ERR** and **C%LAST_FILL**.
- If the hard error corresponded to the data explicitly requested by the Mbox reference, **M%HARD_ERR** qualifies **M%MD_BUS** data indicating to the **M%MD_BUS** receiver that a hard error occurred while accessing the requested data.
- The fill sequence is immediately terminated by the assertion of **C%LAST_FILL**. and the entire Pcache block corresponding to the fill is invalidated.

G.

- The hard error detected by the Cbox on this Mbox-issued PTE DREAD is recorded in PCSTS. The tb miss sequence is immediately terminated.

  IF the error resulted from an Ibox reference, the error is tagged back to the appropriate Ibox reference latch. The error is then signaled via **M%HARD_ERR** when the requested data is returned on **M%MD_BUS**, or is reported through **PA_Q_STATUS** (for DEST_ADDR commands).

  If the original reference came from the Ebox, **M%MME_TRAP** is asserted (in all cases except for PROBE references). This will invoke the memory management fault handler in order to try to report the hard error within the context of the execution of the instruction.
- The fill sequence is immediately terminated by the assertion of **C%LAST_FILL**. and the entire Pcache block corresponding to the fill is invalidated.

H. **C%CBOX_HARD_ERR** was asserted by the Cbox during an I_CF or D_CF command. This is the mechanism by which the Cbox informs the Mbox of a hard error during a read or IPR_RD operation where the Cbox must return data. Thus, see the error responses specified by F and G for the error response within context of the original read operation.

12.14 MBOX INTERFACES

The Mbox passes data and/or control information to four other sections of the NVAX chip. These sections are: 1) Ibox, 2) Ebox, 3) Useq and 4) Cbox. The Cbox interface has additional signals for NVAX Plus and is described in this section. Refer to the NVAX CPU Chip Functional Specification for MBOX interface signal definitions to the IBOX, EBOX, and Useq.

12.14.1 Signals from Cbox

- **C%CBOX_CMD**<1:0>: Command field of Cbox reference sent to Mbox.
- **C%CBOX_ADDR**<12:5>: Invalidate address of Cbox reference sent to Mbox.
- **C%MBOX_FILL_QW**<4:3>: Indicates the aligned quadword within the aligned hexaword.
- **C%REQ_DQW**<>: Qualifies the current D_CF to indicate that this is the requested data.
- **B%S6_DATA**<63:0>: Data of Mbox reference seen by Cbox.
- **C%S6_DP**<7:0>: Even data parity corresponding to **B%S6_DATA**<63:0> during cache fill references.
-
- **C%LAST_FILL**: When asserted, indicates that this is the last fill sent for the current sequence.

- **C%CBOX_HARD_ERR**: When asserted when Cbox is driving data onto the **B%S6_DATA** Bus, it indicates that data on **M%MD_BUS** is associated with a non-recoverable hard error.
- **C%CBOX_ECC_ERR**: Indicates that an ECC error is associated with the Cbox data being returned.
- **C%WR_BUF_BACK_PRES**: Indicates that Cbox cannot accept any more entries in its write buffer.

12.14.2 Signals to Cbox

- **M%S6_SET_NUM_H**: PCACHE ALLOCATION BIT, ALLOWS CBOX TO BROADCAST TO SYSTEM BACKMAPS
- **M%S6_CMD**<4:0>: Command field of Mbox reference seen by Cbox.
- **M%S6_PA**<31:3>: Quadword physical address of Mbox reference seen by Cbox.
- **M%C_S6_PA**<2:0>: Address within addressed quadword of Mbox reference seen by Cbox.
- **B%S6_DATA**<63:0>: Data of Mbox reference seen by Cbox.
- **M%S6_BYTE_MASK**<7:0>: Byte mask field of Mbox reference seen by Cbox.
- **M%CBOX_REF_ENABLE**: Indicates that current S6 read reference packet should be latched and processed by the Cbox. This signal is a don't care on write operations.
- **M%CBOX_LATE_EN**: Asserted at the end of a cycle to indicate that a Pcache parity error was detected. As a result, the Cbox must continue to process this reference regardless of what **M%CBOX_REF_ENABLE** indicated.
- **M%ABORT_CBOX_IRD**: Indicates that any IREAD which the Cbox may be processing should be immediately terminated.
- **M%CBOX_BYPASS_ENABLE**: Indicates that the Cbox may drive **B%S6_DATA**<63:0> during the following cycle in order to attempt a data bypass.

12.15 INITIALIZATION

12.15.1 Initialization by Microcode and Software

It is the responsibility of the power-up microcode to perform an IPR_WRITE operation to clear MAPEN before any virtual memory references are issued to the Mbox from either the Ebox or Ibox. Failure to clear MAPEN could result in UNDEFINED behavior prior to complete memory management state initialization.

PAMODE is also cleared by the power-up microcode via an IPR_WRITE command. If the system configuration requires a 32 bit program-visible physical address space, setting the PAMODE value via an IPR_WRITE must be done under very controlled conditions because writes to the PAMODE processor register affect both physical address generation and interpretation of PTEs. With the possible exception of certain diagnostic code, writes to the PAMODE processor register should not be performed while memory management is enabled. With memory management disabled, writes to the PAMODE processor register should not be performed unless the PC of the MTPR instruction which writes to the register is in one of the following (hex) address ranges:

00000000..1FFFFFFF
E0000000..FFFFFFFF

By restricting PC to one of these address ranges, changes to the PAMODE register do not cause the generated physical address to change in going from 30-bit mode to 32-bit mode, or vice versa.

**The console code should be executing in the specified range in order to write to the PAMODE processor register, and it is expected that this is the place where the PAMODE processor register will be initialized.

In uncontrolled conditions, writes to the PAMODE processor register can cause UNDEFINED results.

12.15.1.1 Pcache Initialization

The Pcache is disabled by the power-up initialization sequence. In order to enable the Pcache, the following sequential actions must be performed:

1. Pcache IPR_WRITE operations must be performed to each Pcache tag to write the tag field to a known state, set the tag parity bit to the corresponding value, and clear the subblock valid bits.
2. An IPR_WRITE to the PCCTL must be done to enable the Pcache in the desired operation mode.

Note that the data array need not be initialized because correct parity will be written into the data array whenever fill data is validated, and data parity is only checked on validated sub-blocks.

12.15.1.2 Memory Management Initialization

Memory management is disabled by MAPEN being cleared by the power-up microcode. Before memory management can be turned on, the following actions must be performed:

- The Ebox must issue a TBIA command to invalidate the TB and reset the NLU pointer to a known state. This is done as part of the microcode processing of an MTPR to MAPEN.
- The Ebox must write the appropriate values into the six memory base and length registers via IPR_WRITE commands.

Once this is done, the Ebox may turn on memory management by setting MAPEN through an IPR_ WRITE command.

12.16 Mbox Testability Features

This section describes what testability features are made use of for Mbox testability, and what Mbox signals are used for each testability function. For a global understanding of NVAX testability, and for a detailed description of each testability strategy and hardware mechanism, the reader is referred to Chapter 17.

12.16.1 Internal Scan Register and Data Reducers

The following Mbox signals exist in the scan chain:

— **S5_PA**<31:0>>
— **S5_TAG**<5:0>
— **S5_DL**<1:0>
— **S5_AT**<1:0>
— **S5_DEST**<1:0>
— **S5_QUAL**<6:0>
— **PA_Q_STATUS**<2:0>
— **M%MME_TRAP**
— IREF_LATCH valid bit
— SPEC_QUEUE valid bits (2)
— EM_LATCH valid bit
— VAP_LATCH valid_bit
— MME_LATCH valid_bit
— RTY_DMISS_LATCH valid_bit
— CBOX_LATCH valid_bit
— **M%CBOX_BYPASS_ENABLE**
— **M%CBOX_REF_ENABLE**
— **M%EM_LAT_FULL**

Note that only **S5_PA**<31:0> contains a data reducer. Implementing a data reducer on this bus should provide coverage for the Mbox S5 pipe as well as coverage for the Ibox, Ebox and Cbox logic which issue references to the Mbox.

12.16.2 Nodes on Parallel Port

The following signals are observable via the Parallel Port:

— **S5_CMD**<4:0>
— Current Reference Source (3 encoded bits). The encodings are as follows:

Reference Source	Encoding
NOP or PA_QUEUE (when cmd = STORE)	000
IREF_LATCH	001
SPEC_QUEUE	010
EM_LATCH (when cmd ^= STORE)	011
VAP_LATCH (when cmd ^= STORE)	100
MME_LATCH	101
RTY_DMISS_LATCH	110

Reference Source	Encoding
CBOX_LATCH	111

— **M%ABORT**

— **M%TB_MISS**

— **M%PCACHE_MISS**

— MME state machine state bits (4 encoded bits). The encodings are as follows:

State Name	Encoding
home	0000
tb_miss_1	0001
tb_miss_2	0010
tb_miss_3	0011
tb_miss_4	0100
tb_miss_5	0101
doub_tb_miss_1	0110
doub_tb_miss_2	0111
doub_tb_miss_3	1000
doub_tb_miss_4	1001
mme_1	1010
mme_2	1011
ipr_rd_1_tb_per_2	1100
xpage_1	1101
tb_per_1	1110
undefined	1111

— MD_BUS Qualifiers (3 encoded bits). The encodings are as follows:

Event	Encoding
undefined	000
Ibox data	001
Ebox data	010
Ibox and Ebox data	011
VIC data	100
Ibox IPR data	101
undefined	110
Mbox data	111

— **M%MME_FAULT**

12.16.3 Architectural features

All MBOX IPRs can be invoked through the use of MTPR or MFPR macroinstructions. See the Architectural Summary Chapter for a list of all Mbox IPR addresses. Note that Mbox IPR addresses referenced through the MxPR instruction are translated by the Ebox microcode into IPR_RD, IPR_WR, TBIS, TBIA, or PROBE operations before being issued to the Mbox.

12.16.3.1 Translation Buffer Testability

The diagnostic user can invalidate the entire TB array by executing an MTPR instruction which addresses the TBIA IPR. This operation will also reset the NLU pointer. The user can invalidate any virtual page address which may cached in the TB by executing a MTPR addressing the TBIS IPR.

The diagnostic user can explicitly query the TB to determine if a given tag is validated and stored in the TB. This is accomplished by addressing the Translation Buffer Check IPR through the MTPR instruction.

Every TB entry can be explicitly filled and validated by the diagnostic user through the use of the TB_TAG_FILL and TB_PTE_FILL commands. The entry on which these two commands operate at any given time is addressed by the NLU pointer. The NLU pointer is a round robin pointer which increments when a TB_PTE_FILL is executed or when a tag match is detected on the entry which the NLU pointer is currently pointing to. The NLU pointer is reset to point to the 0th entry whenever a TBIA command is executed.

12.16.3.2 Pcache Testability

Every bit in the Pcache can be read and written by the user through DREAD, WRITE, IPR_RD and IPR_WR operations. Pcache is accessed by DREADs and WRITEs. All other bits (tag, valid bits and parity bits) are accessed through Mbox IPRs.

The operational mode of the Pcache can be changed to accomodate testing the array. The mode is controlled by the Pcache Control Register (PCCTL) which can be read and written as an Mbox IPR. The PCCTL allows the user to:

1. Enable/disable D-stream and/or I-stream operations to the Pcache.
2. Allow the Pcache to operate in a direct mapped force hit mode.
3. Enable/disable Pcache parity checks.

12.17 Mbox Performance Monitor Hardware

Hardware exists in the Mbox to support the NVAX Performance Monitoring Facility. See Chapter 16 for a global description of this facility.

The Mbox hardware generates two signals, **M%PMUX0** and **M%PMUX1**, which are driven to the central performance monitoring hardware residing in the Ebox. These two signals are used to supply Mbox performance data for the purpose of recording performance statistics. Seven Mbox performance monitoring functions exist. The function to be executed is specified by the PMM field of the PCCTL register.

Table 12–15: Mbox Performance Monitor Modes

PCCTL<7:5>	Performance Monitor Mode
000	TB hit rate for P0/P1 Space I-stream Reads
001	TB hit rate for P0/P1 Space D-stream Reads
010	TB hit rate for S0 Space I-stream Reads
011	TB hit rate for S0 Space D-stream Reads
100	Pcache hit rate for I-stream Reads
101	Pcache hit rate for D-stream Reads
110	illegal mode–Results are UNPREDICTABLE
111	ratio of unaligned virtual reads and virtual writes to total virtual reads and virtual writes

12.18 Revision History

Who	When	Description of change
Bill Wheeler	8-May-1990	Other tweaks
Bill Wheeler	27-Feb-1990	Add perf monitor hardware. Other tweaks
Bill Wheeler	15-Jan-1990	Signal name change
Bill Wheeler	20-Nov-1989	Final Changes prior to review for Rev 1.0 Release
Bill Wheeler	23-Aug-1989	More Updates
Bill Wheeler	31-Jul-1989	Spec Update
Bill Wheeler	06-Mar-1989	For External Release
Bill Wheeler	30-Nov-1988	Initial Release
Gil Wolrich	15-Nov-1990	NVAX Plus External Release

Chapter 13

NVAX Plus CBOX

13.1 Functional Overview

The NVAX Plus and NVAX processors contain common IBOX, EBOX, FBOX, and MBOX internal functionality. The NVAX external interface is to a backup cache and I/O NDAL bus, while the NVAX Plus external interface is a common cache/memory bus used by EV processors. While the MBOX interface section of the CBOX is similar for NVAX and NVAX Plus, the EDAL bus interface sections of NVAX Plus replace the TAG, DATA, and NDAL/BIU sections of the NVAX CBOX.

The NVAX Plus CBOX receives read, and write requests from the MBOX. The CBOX initiates bus cycles and sends fill data to the MBOX. Invalidates are initiated by external logic and sent to the MBOX under CBOX control.

For reads the tag and data stores are read together. If the tag matches and the valid bit is set the associated data is returned to the MBOX. If the read misses a READ_BLOCK request is sent to the system logic. NVAX Plus waits for the system to update the cache and deliver the requested data to a 32 byte Input Buffer.

For writes NVAX Plus requires a probe cycle in which the tag and state bits are read. If the probe indicates a tag match for a valid block which is not shared, then NVAX PLUS writes the data store. If the write probe indicates a miss or the block is shared, NVAX Plus sends a WRITE_BLOCK command to the system logic. The WRITE_BLOCK command has an eight bit longword mask associated with it indicating the longwords which are to be updated. The write data is placed in a 32 byte Output Buffer. The write is completed under external control.

For a NVAX Plus EDAL bus system;

- Only one miss can be issued, the cache can not be used till the miss completes
- The external logic is responsible for writebacks
- The external logic must maintain cache coherence for both backup and primary caches

A Valid, Dirty, and Shared bit are associated with each tag in the external backup cache. The Valid and Shared bits are written by external system logic only. The Dirty bit is written by NVAX Plus on write hits to a non-shared block and indicates the data in cache is no longer the same as main memory. For Writes to Shared blocks NVAX Plus can not write directly into the cache, and must issue a WRITE_BLOCK command to enable the external system logic to broadcast the shared write to all caches in the system.

13.2 CBOX REGISTERS

13.2.1 BIU_ADDR

This read-only register contains the physical address associated with any errors reported in BIU_STAT[7..0]. The BIU_ADDR is locked against further updates, until the error bits of BIU_STAT are cleared.

13.2.2 BIU_STAT

The BIU_STAT is a WRITE-ONE-TO CLEAR W1C IPR. When one of BIU_HERR, BIU_SERR, BC_TPERR or BC_TCPERR is set, BIU_STAT[6..0] are locked against further updates, and the address associated with the error is latched and locked in the BIU_ADDR register. BIU_STAT[7..0] and BIU_ADDR are unlocked when the BIU_STAT[7,3:0] are written with 1's.

When FILL_ECC or BIU_DPERR is set, BIU_STAT[13..8] are locked against further updates, and the address associated with the error is latched and locked in the FILL_ADDR register. BIU_STAT[14..8] and FILL_ADDR are unlocked when BIU_STAT[14,11:8] are written with 1's.

This register is not unlocked or cleared by reset and needs to be explicitly cleared by Microcode.

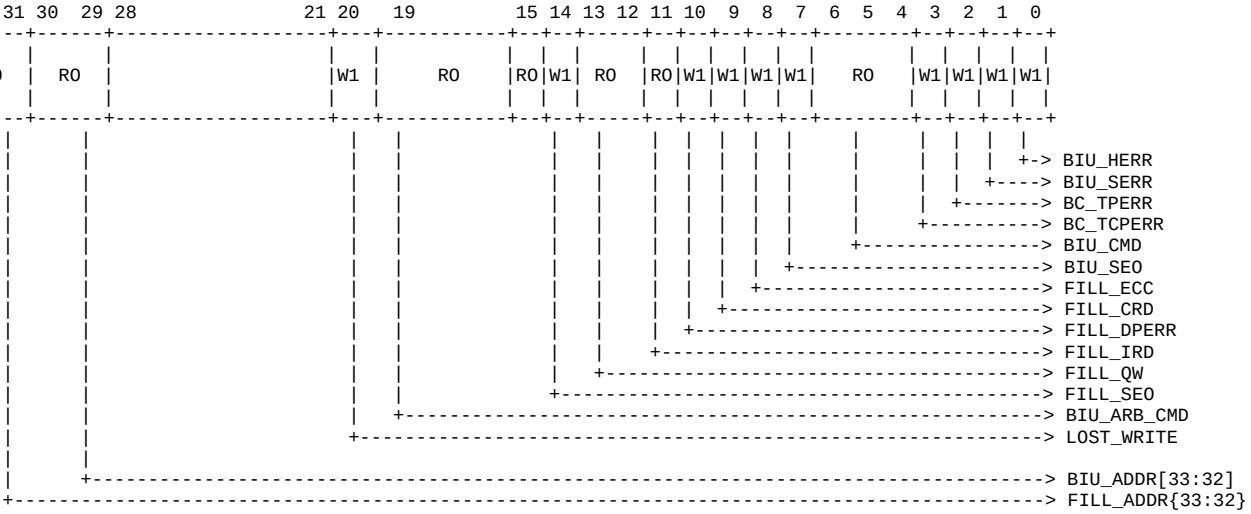


Table 13–1: BIU STAT

Field	Type	Description
BIU_HERR	W1C	This bit, when set, indicates that an external cycle was terminated with the cAck_h pins indicating HARD_ERROR.
BIU_SERR	W1C	This bit, when set, indicates that an external cycle was terminated with the cAck_h pins indicating SOFT_ERROR.
BC_TPERR	W1C	This bit, when set, indicates that a external cache tag probe encountered bad parity in the tag address RAM.

Table 13–1 (Cont.): BIU STAT

Field	Type	Description
BC_TCPERR	W1C	This bit, when set, indicates that a external cache tag probe encountered bad parity in the tag control RAM.
BIU_CMD	RO	This field latches the cycle type on the cReq_h pins when a BIU_HERR, BIU_SERR, BC_TPERR, or BC_TCPERR error occurs, and locks till BIU_CTL[7,3:0] are cleared.
BIU_SEO	W1C	This bit, when set, indicates that an external cycle was terminated with the cAck_h pins indicating HARD_ERROR or that a an external cache tag probe encountered bad parity in the tag address RAM or the tag control RAM while one of BIU_HERR, BIU_SERR, BC_TPERR, or BC_TCPERR was already set.
FILL_ECC	W1C	ECC error. This bit, when set, indicates that primary cache fill data received from outside the CPU chip contained an ECC error.
FILL_CRD	W1C	Corrected read. This bit is only meaningful when FILL_ECC is also set. FILL_CRD is set to indicate that the ECC error was correctable and clear to indicate that the error was not correctable.
BIU_DPERR	W1C	BIU Parity Error. This bit when set, indicates that the BIU received data with a parity error from outside the CPU chip while performing either a Dcache or Icache fill. BIU_PERR is only meaningful when the CPU chip is in parity mode, as opposed to ECC mode.
FILL_IRD	RO	This bit is only meaningful when either FILL_ECC or FILL_DPERR is set. FILL_IRD is set to indicate that the error which caused FILL_ECC or FILL_DPERR to set occurred during an Icache fill and clear to indicate that the error occurred during a Dcache fill and locks till BIU_CTL[14,10:8] are cleared.
FILL_QW	RO	This field is only meaningful when either FILL_ECC or FILL_DPERR is set. FILL_QW identifies the quadword within the hexaword primary cache fill block which caused the error. It can be used together with FILL_ADDR[33..5] to get the complete physical address of the bad quadword. FILL_QW locks till BIU_CTL[14,10:8] are cleared.
FILL_SEO	W1C	This bit, when set, indicates that a primary cache fill operation resulted in either an uncorrectable ECC error or in a parity error while FILL_ECC or FILL_DPERR was already set.
BIU_ARB_CMD	RO	This field latches the ARB_CMD which resulted in the BIU error and locks till BIU_CTL[14,10:8] are cleared.
LOST_WRITE	W1C	An second error, and command is a write.
BIU_ADDR[33:32]	RO	Bits 33,32 of the BIU_ADDR register, should be set only for I/O space address.
FILL_ADDR[33:32]	RO	Bits 33,32 of the FILL_ADDR register, should be set only for I/O space address.

13.2.3 FILL_ADDR

This read-only register contains the physical address associated with any errors reported in BIU_STAT[14..8]. FILL_ADDR is locked against further updates, till BIU_CTL[14,10:8] are cleared.

13.2.4 BIU_CTL

BIU_CTL is cleared by power-up microcode.

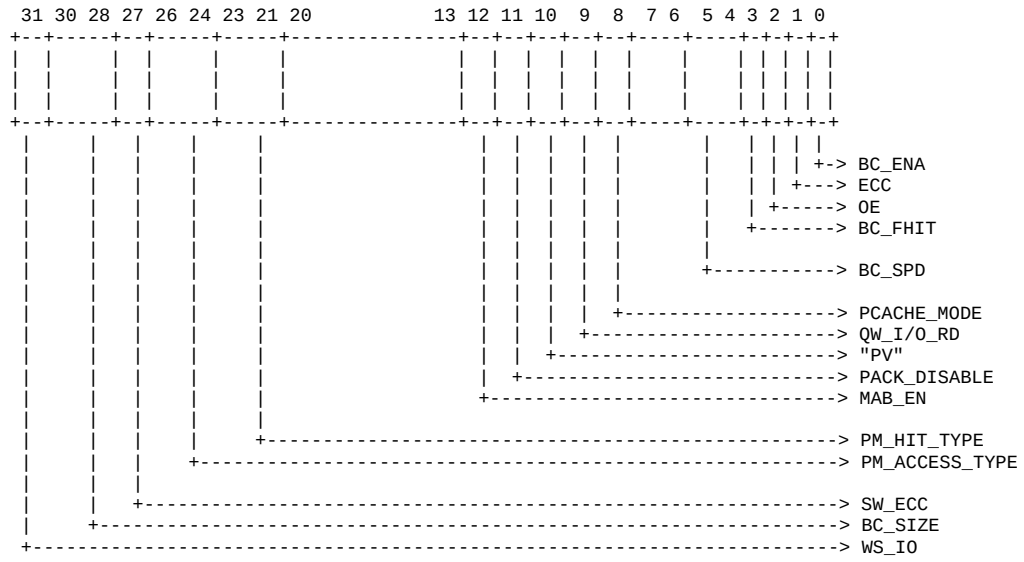


Table 13–2: BIU Control Register

Field	Type	Description
BC_EN	WO	External cache enable. When clear, this bit disables the external cache. When the external cache is disabled the BIU does not probe the external cache tag store for read and write references; it launches a request on cReq_h immediately.
ECC	WO	When this bit is set NVAX Plus generates/expects ECC on the check_h pins. When this bit is clear NVAX Plus generates/expects parity on four of the check_h pins.
OE	WO	When this bit is set NVAX Plus does not assert its chip enable pins during RAM write cycles, thus enabling these pins to be connected to the output enable pins of the cache RAMs.
BC_FHIT	WO	External cache force hit. When this bit is set and BC_EN is also set, all pin bus READ_BLOCK and WRITE_BLOCK transactions are forced to hit in the external cache. Tag and tag control parity are ignored when the BIU operates in this mode. BC_EN takes precedence over BC_FHIT. When BC_EN is clear and BC_FHIT is set no tag probes occur and external requests are directed to the cReq_h pins.

Table 13–2 (Cont.): BIU Control Register

Field	Type	Description
BC_SPD	WO	External cache speed. This field indicates to the BIU the read and write access time of the RAMs used to implement the off-chip external cache, measured in CPU cycles. BCache speeds of 2,3, or 4 times the CPU_clk are available. NVAX Plus replaced BC_RD_SPD and BC_WR_SPD with BC_SPD. NVAX Plus uses the BC_SPD field to program the read and write cache access time. EVAX allows the read and write cache access times to be programmed separately. BC_SPD is not initialized on reset and must be explicitly written before enabling the external cache.
BC_WE_CTL	WO	External cache write enable control. This field is used to control the timing of the write enable and chip enable pins during writes into the data and tag control RAMs. This field will be set to a fixed value for NVAX Plus. This field is programmable on EVAX.
PCACHE_MODE	WO	When this bit is clear the Pcache is allocated as a two way set associative, and when set the Pcache allocates as direct mapped.
QW_IO_RD	WO	When this bit is set IO_SPACE DREADs which are not quadword aligned return data from an internal register which contains bits<63:32> of the previous quadword aligned read.
"PV"	WO	Set for low cost workstations.
PACK_DISABLE	WO	Diagnostic feature to disable write packing, except for QW packing directed by microcode.
MAB_EN	WO,0	Diagnostic feature to allow tagAdr[33:32] to output MAB[7:6] and tagAdr[17,18,19] to output MAB[10:8] depending on Bcache size. This bit is cleared at reset to insure tagAdr[33:32] and tagAdr[17,18,19] are not driven unless enable by software.
PM_HIT_TYPE	WO	Selects Bcache tag compare type for Performance Monitor selection of C%PMUX1.
PM_ACCESS_TYPE	WO	Selects Bcache tag compare type for Performance Monitor selection of C%PMUX0.
SW_ECC	WO	This bit, when set, enables the use of ECC check bits from IPR_BEDECC as given by software for write data. If BIU_CTL[1] = '0, i.e. parity mode if SW_ECC is set BEDECC[0] is the parity bit generated for data[31:0} and BEDECC[7] is the parity bit generated for data[63:32].
BC_SIZE	WO	This field is used to indicate the size of the external cache. BC_SIZE is not initialized on reset and must be explicitly written before enabling the external cache. See Table 13–4 for the encodings.
BC_PA_DIS	WO	This field has been removed on NVAX+.
WS_IO	WO	This bit, when set, indicates that IO-space is mapped for "FLAMINGO" work stations.

The low cost workstation system maintains a write-through cache with byte parity. The cache is not written by NVAX Plus, all writes and byte/word writes issue a WRITE BLOCK to the system. The LW parity generated be NVAX Plus is not used for writes. System logic constructs a byte enable for each of the 16 possible bytes from cWMask<7:0>, dataA<3>, and dataWE_h<1:0>, and generates byte parity. Fast external reads are executed for read hits, with byte parity driven to the check bits.

For low cost workstations BIU_CTL<"PV"> = '1 writes do not probe Bcache. Writes go directly to WRITE_BLOCK, and output byte mask on cWMask<7:0>. dataA<3> identifies the QW to which the cWMask lines apply, and dataWE_h<1:0> output byte mask information for the other QW of data.

```
dataWE   [1]      [0]   byte mask of alternate QW
                            7 6 5 4   3 2 1 0

          0        0        0 0 0 0   0 0 0 0
          0        1        0 0 0 0   1 1 1 1
          1        0        1 1 1 1   0 0 0 0
          1        1        1 1 1 1   1 1 1 1
```

Reads probe the Bcache, byte parity is input as

```
check_h[0]  for data[6:0],    check_h[1]  for data[15:7],    check_h[2]  for data[23:16],   check_h[3]  for data[31:24],
check_h[7]  for data[39:32],  check_h[8]  for data[47:40],   check_h[9]  for data[55:48],   check_h[10] for data[63:56],
check_h[14] for data[71:64],  check_h[15] for data[79:72],   check_h[16] for data[87:80],   check_h[17] for data[95:88],
check_h[21] for data[103:96], check_h[22] for data[111:104], check_h[23] for data[119:112], check_h[24] for data[127:120]

where check_h[3:0]   are xored to produce the LW parity bit for data[31:0],
      check_h[10:7]] are xored to produce the LW parity bit for data[63:32],
      check_h[17:14] are xored to produce the LW parity bit for data[95:64],
      check_h[24:21] are xored to produce the LW parity bit for data[127:96]
```

The dataWE lines are only used for mask information in "PV" mode.

Table 13–3: BC_SPD

BIU_SPD	DRV_CLK/Cache Speed
00	2X cpu cycle
01	3X cpu cycle
10	4X cpu cycle

Table 13–4: BC_SIZE

BC_SIZE	Size
0 0 0	128 Kbytes
0 0 1	256 Kbytes
0 1 0	512 Kbytes
0 1 1	1 Mbytes
1 0 0	2 Mbytes
1 0 1	4 Mbytes
1 1 0	8 Mbytes

BIU_CTL<DISABLE_ERRORS> may added for test and error recovery.

13.2.5 FILL_SYNDROME

The FILL_SYNDROME register is a 14-bit read-only register. If the chip is in ECC mode and an ECC error is recognized during a primary cache fill operation, the syndrome bits associated with the bad quadword are locked in the FILL_SYNDROME register. The FILL_SYNDROME register is locked against further updates, till BIU_CTL[14,10:8] are cleared.

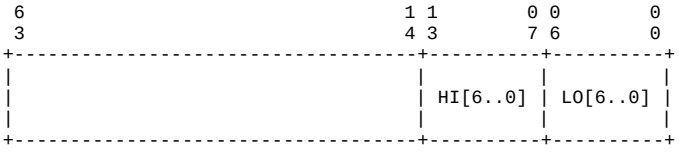


Table 13–6: Fill Syndrome

Field	Description
LO	The LO field latches the ECC syndrome bits for the low longword.
HI	The HI field latches the ECC syndrome bits for the high longword.

13.2.6 BEDECC

The BEDECC register is a 14-bit write-only register. If BIU_CTL[SW_ECC] = '1 the check bits for write data are sourced from BEDECC instead of the normal check bit generation logic.

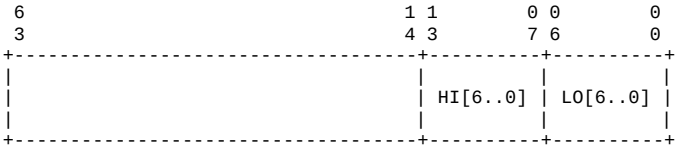
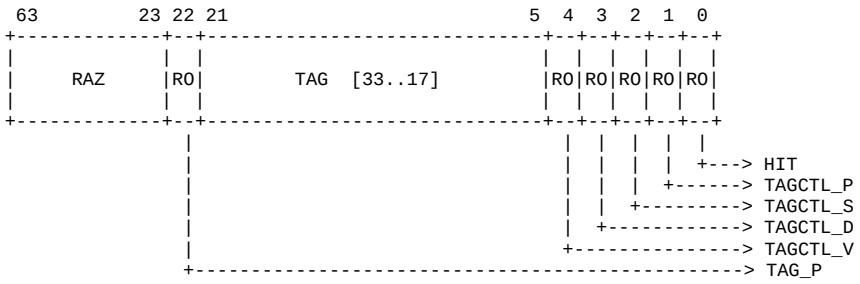


Table 13–6: Fill Syndrome

Field	Description
LO	The LO field for check bits of data[31:0].
HI	The HI field for check bits of data[63:32].

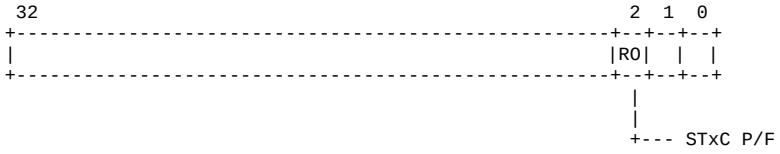
13.2.7 BC_TAG

The BC_TAG is a read-only IPR. Unless locked, the BC_TAG register is loaded with the results of every backup cache tag probe. When a tag or tag control parity error or primary fill data error (parity or ECC) occurs this register is locked against further updates. Software may read this register by using the MFPR instruction. The BC_TAG register is unlocked when the BIU_STAT[7,3:0] are cleared.



13.2.8 STxC_RESULT/CEFSTS

Bit 2 of STxC_RESULT, STxC P/F is read only. **When a write is issued to this IPR address AC(hex) the IREAD latch lockout as a result of a failed READ LOCK is cleared.** Bit 2 is set if the last store conditional failed, an is reset if the last store conditional passed. This register is read by microcode following write_unlocks to determine if the write was successful.



13.2.9 SIO

SIO is a read/write IPR. Bit 0 reads the state of the serial line/SROM INPUT data input pin. Bit 1 is driven to the serial line output/SROM CLOCK output pin.

13.2.10 SOE-IE

SOE-IE is a read/write IPR. Bit 0 drives the SROM_OE pin. Bit 1 receives the SROM_FAST pin which determines the extent to which the SROM is to be read.

13.2.11 QW_PACK

This is a write only ipr used by microcode to inform the WRITE_PACKER to pack the next two LW writes even if the address is in io space or the command is a WRITE_UNLOCK. The IPR_WR takes place during the MOVQ instruction and the MTPR MAILBOX instruction to produce QW writes to IO space. The QW_PACK clears once the QW is loaded into the WRITE_QUEUE.

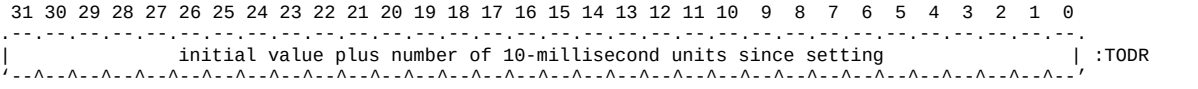
13.2.12 CONSOLE REG

This R/W register contains the start address for the console. It is written by system software, and used to determine the console start address in response to a HALT interrupt.

13.2.13 Time-of-Day Register (TODR)

The Time-of-Day Register forms an unsigned 32-bit binary counter that is driven from a 100Hz oscillator, so that the least significant bit of the clock represents a resolution of 10 milliseconds. The R/W register counts only when it contains a non-zero value.

Figure 13–1: Time of Day Register, TODR



13.2.14 Programmable Interval Clock

The interval clock provides an interrupt at IPL 16 (hex) at programmed intervals. The counter is incremented at 1 microsecond intervals, with at least .01% accuracy. The interval clock consists of three registers in the privileged register space.

1. Interval Count Register (ICR) - The interval count register is a read only register incremented every microsecond. Upon a carry out (overflow) from bit 31, it is automatically loaded from NICR and an interrupt is generated if the interrupt is enabled. That is, the value of ICR on successive microseconds will be FFFFFFFD (hex), FFFFFFFE, FFFFFFFF, <value of NICR>.
2. Next Interval Count Register (NICR) - This reload register is a write only register that holds the value to be loaded into ICR when ICR overflows. The value is retained when ICR is loaded.

3. Interval Clock Control Status Register (ICCS) - The ICCS register contains control and status information for the interval clock.

The interval clock consists of 3 Internal Processor Registers configured as follows:

Figure 13–2: ICCS

13.2.15 Interval Clock Control Register

When bit <0>, the RUN bit, is a 1, the Interval Count Register is incremented once per microsecond. When clear, ICR does not increment automatically. RUN is cleared during reset.

Bits <3:1>, Must be zero.

Writing a 1 to bit <4> (XFR) generates a pulse which causes the Next Interval Count Register to be copied to the Interval Count Register. XFR does not require clearing; Multiple XFRs will produce multiple transfers. XFR is always read as 0.

When RUN is a 0, writing a 1 to bit <5> (SGL) generates a pulse which causes the Interval Count Register to be incremented by 1. When RUN is a 1, or if SGL and XFR are written at the same time, the write to SGL is ignored. SGL does not require clearing; Multiple SGLs will produce multiple increments. SGL is always read as 0.

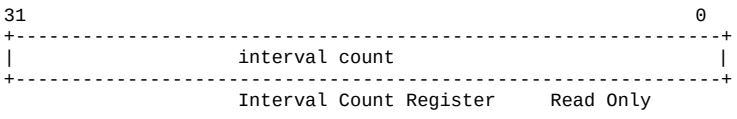
When Bit <6> IE is set, an interrupt request is generated every time ICR overflows (every time Interrupt is set). When clear, no interrupt is requested. Similarly, if Interrupt is already set and the software sets Interrupt Enable, an interrupt is generated. That is, an interrupt is generated whenever the function [Interrupt Enable AND Interrupt] changes from 0 to 1. Interrupt Enable is cleared by reset.

Whenever the Interval Count Register overflows, bit <7> (INT) is set. If IE is set when INT is set, an interrupt is posted.

Whenever the Interval Count Register overflows and INT is already set, ERR (bit <31>) is set. Thus, ERR indicates a missed overflow.

Reset clears ICCS <6> and <0>, and leaves the rest of ICCS unpredictable.

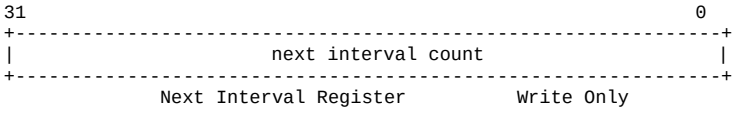
Figure 13–3: ICR



13.2.16 Interval Count Register

This read-only register contains the interval count. When the RUN bit is a zero, writing a 1 to SGL increments the register. When RUN is a 1, the register is incremented once per microsecond. When the counter overflows, the INT bit is set, and an interrupt is posted if IE is set. The register is then loaded from the Next Interval Count Register and continues incrementing. The maximum delay that can be specified is approximately 1.2 hours.

Figure 13–4: NICR



13.2.17 Next Interval Count Register

This contains the value which is loaded into the Interval Count Register after an overflow, or in response to a 1 written to XFR.

The Interval Count Register is cleared by reset.

To use the Interval Clock, load the negative (2's complement) of the desired interval into the Next Interval Count Register. Then, writing 51 (hex) to the ICCS will enable interrupts, load the Next Interval into the Interval Count Register, and set the RUN bit. An interrupt will then occur every "interval count" microseconds. The interrupt routine should write C1 (hex) to the ICCS to clear the interrupt. If Interrupt has not been cleared (the interrupt has not been handled) by the time of the next ICR overflow, Error will be set.

If NICR is written while the clock is running, the clock may lose or add a few ticks. If the interval clock interrupt is enabled, this may cause the loss of an interrupt.

13.3 Cache Organization

Pins for tagAdr_h<31:17> are allocated allowing the cache size to be as small as 128 Kb. The BC_SIZE field of the BIU_CTL register determines which bits of tagAdr_h<22:17> are to be includes in the match comparison.

NVAX Plus cache cycle are 2,3, or 4 times the internal cpu_clk cycle time. ISSUE: SET BY IRQ AT RESET OR IN BIU_CTL.

13.4 Cache_Speed and SYS_CLK

NVAX Plus cache accesses are 2,3, or 4 times the CPU_CLK period.

Transactions requiring system logic intervention are referenced to SYS_CLK which is separately programable, also at 2,3, or 4 times the CPU_CLK period. For systems in which cache_speed and SYS_CLK are both 2 times the CPU Cycle, SYS_CLK lags the cache access by one CPU cycle allowing the fastest transfer of commands to the system.

13.5 DataPath

Table 13–7: Cbox Queues and Major Latches

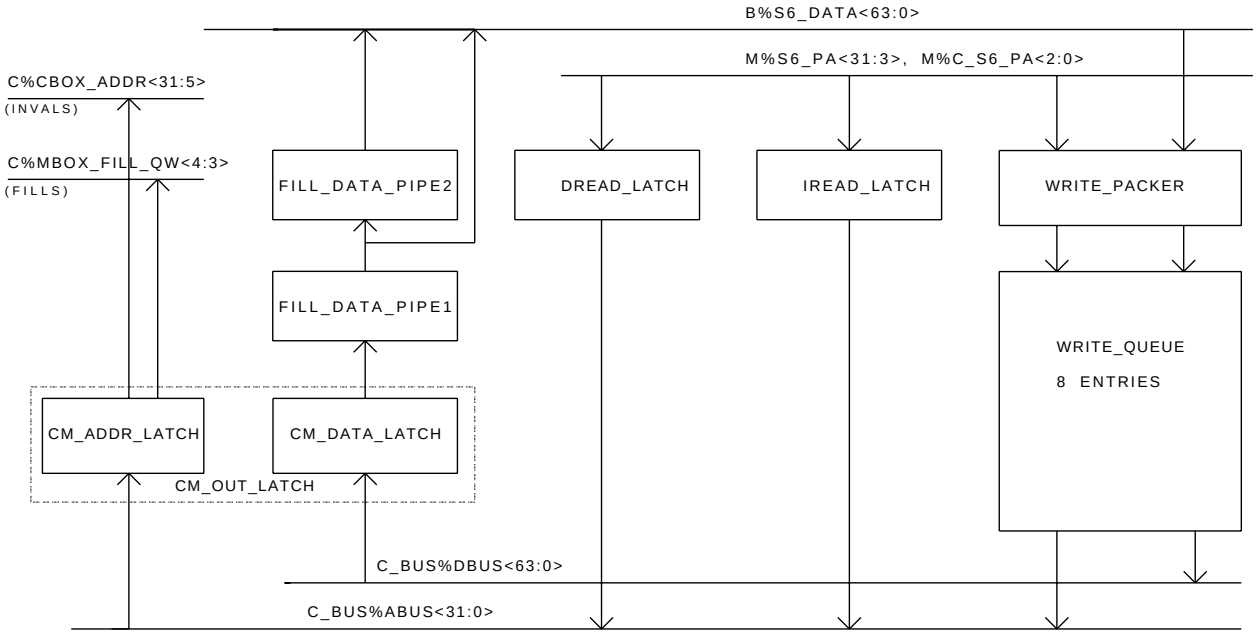
Queue/Latch	Entries	Address/Data	Function
CM_OUT_LATCH	1	Addr<31:3>,data<63:0>	Holds fill data or an invalidate address being sent to the Mbox.
FILL_DATA_PIPEs	2	Data<63:0>	Pipeline data destined for the Mbox.
DREAD_LATCH	1	Addr<33:3>	Holds a data-stream read request from the Mbox.
IREAD_LATCH	1	Addr<33:3>	Holds an instruction-stream read request from the Mbox.
WRITE_PACKER	1	Addr<33:3>,data<63:0>	Compresses sequential memory writes to the same quadword.
WRITE_QUEUE	8	Addr<33:3>,data<63:0>	Queues write requests from the Mbox.
INVADR_LATCH	1	iAddr<12:5>	Holds address for Pcache invalidates.
INPUT_LATCH	2	Data<127:0>	Holds input data from the BD_DATA bus.
OUTPUT_LATCH	1	Data<127:0>	Holds output data to be driven onto the BD_DATA bus.

13.6 Mbox Interface

All NVAX Plus CPU chip transactions for the Cbox arrive through the Cbox-Mbox interface. Reads come from the Mbox to the Cbox through the read latches. Writes arrive through the WRITE_PACKER and the WRITE_QUEUE. All fills returning from the Cbox to the Mbox go through the CM_OUT_LATCH.

A block diagram of the Mbox interface is shown in Figure 13–5.

Figure 13–5: Mbox Interface



When the Mbox has a command for the Cbox, the command appears on M%S6_CMD<4:0>. M%CBOX_REF_ENABLE is asserted for all reads, IPR_RDs, and IPR_WRs. It is not asserted for writes since the Cbox accepts all writes from the Mbox. The Cbox loads the address from M%S6_PA<31:3> into either the IREAD_LATCH, the DREAD_LATCH, or the WRITE_PACKER. If the command is a write, the Cbox loads the data from B%S6_DATA and the byte enable from M%S6_BYTE_MASK into the WRITE_PACKER.

Table 13–8 shows the commands which pass between the Mbox and the Cbox.

Table 13–8: Mbox-Cbox Commands

Command	Description	Cbox datapath element involved
Mbox to Cbox commands driven on M%S6_CMD<4:0>		
IREAD[1]	Instruction stream read	IREAD_LATCH
DREAD[1]	Data stream read	DREAD_LATCH
DREAD_MODIFY[1]	Data stream read with modify intent	DREAD_LATCH
DREAD_LOCK[1]	Interlocked data stream read	DREAD_LATCH
WRITE_UNLOCK	Write which releases lock	WRITE_PACKER, WRITE_QUEUE
WRITE	Normal write	WRITE_PACKER, WRITE_QUEUE
IPR_RD[1]	Read of an internal or external processor register	DREAD_LATCH
IPR_WR[1]	Write of an internal or external processor register	WRITE_PACKER, WRITE_QUEUE
Cbox to Mbox commands driven on C%CBOX_CMD<1:0>		
D_CF	Data stream cache fill	CM_OUT_LATCH
I_CF	Instruction stream cache fill	CM_OUT_LATCH
INVAL	Hexaword invalidate	CM_OUT_LATCH
NOP	No operation.	

[1]Qualified by M%CBOX_REF_ENABLE.

13.6.1 The IREAD_LATCH and the DREAD_LATCH

When the Mbox has a read command for the Cbox, the Cbox loads the address from M%S6_PA<31:3> into either the depending on the command. If M%S6_PA<31:29> = '111 IREAD_LATCH or DREAD_LATCH bits<33:32> are set to '11, else they are set to '00. Only IREADs are loaded into the IREAD_LATCH. The DREAD_LATCH is used for DREAD, DREAD_MODIFY, DREAD_LOCK, and IPR_READ.

The Mbox only has one outstanding IREAD and one outstanding DREAD at a time, so no back-pressure for the latches is needed. When the DREAD_LATCH is valid, the Mbox does not start the next DREAD-type transaction until all fill data from the previous command is returned to the Mbox. When the IREAD_LATCH is valid, the Mbox does not start the next IREAD transaction until either the IREAD has been aborted or all fill data from the IREAD is returned to the Mbox.

Table 13–9 and Table 13–10 show the fields which are contained in the two read latches.

Table 13–9: IREAD_LATCH Fields

Field	Purpose
ADDRESS<31:0>	Physical address of the read request.
CMD<4:0>	Specific command being done (IREAD).

Table 13–10: DREAD_LATCH Fields

Field	Purpose
CMD<4:0>	Specific command being done (DREAD, DREAD_MODIFY, DREAD_LOCK, IPR_READ).
ADDRESS<31:0>	Physical address of the read request.

When the Mbox asserts M%ABORT_CBOX_IRD, the Cbox clears the IREAD_LATCH entry if the reference has not yet started. If the CBOX starts the IREAD sequence before Mbox asserts M%ABORT_CBOX_IRD the sequence is continued but data is not sent to the MBOX.

13.6.2 WRITE_PACKER and WRITE_QUEUE

Writes from the Mbox go through the WRITE_PACKER and into the WRITE_QUEUE. The WRITE_PACKER holds a quadword of data; the WRITE_QUEUE consists of 8 entries, each of which contains a quadword of data. The purpose of the WRITE_PACKER is to accumulate writes to the same quadword which arrive sequentially, so that only one write has to be done into the cache.

A WRITE command with an non I/O space address or a WRITE or WRITE_UNLOCK preceeded by an IPR_WR to the QW_PACK ipr are packed. If the WRITE is to the same octaword as the quadword which is presently being packed, the quadword in the WRITE_PACKER is placed into the WRITE_QUEUE and the SAME_OCTAWORD bit set in the CMD field. The new write reference is loaded into the WRITE_PACKER. If the WRITE is not to the same octaword as the quadword which is presently being packed, the quadword in the WRITE_PACKER is placed into the WRITE_QUEUE and the SAME_OCTAWORD bit not set in the CMD field. The new write reference is loaded into the WRITE_PACKER. Other writes pass immediately from the WRITE_PACKER into the WRITE_QUEUE. The WRITE_PACKER is flushed at the following times:

- When a memory-space WRITE to a different quadword arrives. The new quadword then remains in the write packer until a write packer flush condition is met.
- When a WRITE_UNLOCK arrives. The WRITE_UNLOCK is then passed immediately from the WRITE_PACKER to the WRITE_QUEUE.
- **When an I/O space write arrives. If QW_PACK and not WS_IO the next two longwords are packed into a QW entry. QW_PACK is set by an IPR_WR issued by microcode to inform the WRITE_PACKER to pack the next two LW writes even if the address is in io space or the command is a WRITE_UNLOCK. The IPR_WR takes place during the MOVQ instruction and the MTPR MAILBOX instruction to produce QW writes to IO space. The QW_PACK clears once the QW is loaded into the WRITE_QUEUE. Thus MOVQ to a QW aligned address results in a single QW write, and MB_ADDR is written with a high LW of zeroes.** Otherwise the I/O space write is passed immediately from the WRITE_PACKER to the WRITE_QUEUE.
- When an IPR_WRITE arrives. The IPR_WRITE is then passed immediately from the WRITE_PACKER to the WRITE_QUEUE. IPR_WRITEs to VLDST are not placed in the WRITE_QUEUE.
- If an IREAD or a DREAD arrives to the same hexaword as that of the entry in the WRITE_PACKER.
- Whenever the conditions for flushing the write queue are met.

- If the DISABLE_PACK bit in the CCTL IPR is set. In this case, every write passes directly through the WRITE_PACKER without delay.

THREE-CYCLE LATENCY THROUGH THE WRITE_QUEUE

If the WRITE_QUEUE and the WRITE_PACKER are empty, the latency of any write through them is 3 cycles. The implication of this is that if any reads which flush the WRITE_QUEUE are done alternately with writes, their execution will be greatly slowed. This applies to IPR reads and writes and may be an issue in testing the chip.

Table 13–11 describes the fields in the WRITE_QUEUE.

Table 13–11: WRITE_QUEUE Fields

Field	Purpose
VALID	Indicates that the entry contains valid information.
DWR_CONFLICT	Indicates that this write conflicts with a DREAD, giving the WRITE_QUEUE priority. Check is done using hexaword address.
IWR_CONFLICT	Indicates that this write conflicts with an IREAD, giving the WRITE_QUEUE priority. Check is done using hexaword address.
CMD<4:0>	Specific command being done.
ADDRESS<31:0>	Physical address of the write.
BYTE_EN<7:0>	Byte enable for the write.
DATA<63:0>	Data to be written.

When a quadword of data is moved into the WRITE_QUEUE, it is serviced by the Cbox arbiter as the lowest-priority task, unless special conditions exist.

Servicing writes separately from reads allows reads to take higher priority and gets read data back to the CPU faster. However, a read which follows a write to the same hexaword must not be allowed to complete before the write completes. To prevent this there are conflict bits, DWR_CONFLICT<8:0> and IWR_CONFLICT<8:0>, implemented in the WRITE_QUEUE and WRITE_PACKER, one for each entry. The conflict bits ensure correct ordering between writes and a DREAD or an IREAD to the same hexaword.

When a DREAD arrives, the hexaword address is checked against all entries in the WRITE_QUEUE and WRITE_PACKER. Any entry with a matching hexaword address has its corresponding DWR_CONFLICT bit set. The DWR_CONFLICT bit is also set if the WRITE_QUEUE entry is an IPR_WRITE, a WRITE_UNLOCK, or an I/O space write. If any DWR_CONFLICT bit is set, the WRITE_QUEUE takes priority over DREADs, allowing the writes to complete first.

When an IREAD arrives, the hexaword address is checked against all entries in the WRITE_QUEUE and WRITE_PACKER. Any entry with a matching hexaword address has its corresponding IWR_CONFLICT bit set. The IWR_CONFLICT bit is also set if the WRITE_QUEUE entry is an IPR_WRITE, a WRITE_UNLOCK, or an I/O space write. If any IWR_CONFLICT bit is set, the WRITE_QUEUE takes priority over IREADs, allowing the writes to complete first.

As each write (or Vector Operation) is done, the conflict bits and valid bit of the entry are cleared. When the last WRITE_QUEUE entry which conflicts with a DREAD finishes, there are no more DWR_CONFLICT bits set, and the DREAD takes priority again, even if other WRITE_QUEUE entries arrived after the DREAD. In this way a DREAD which conflicts with previous writes is not done until those writes are done, but once those writes are done, the DREAD proceeds.

The analogous statement is true for an IREAD which has a conflict. If IWR_CONFLICT is set and the IREAD is aborted before the conflicting write queue entry is processed, the WRITE_QUEUE continues to take precedence over the IREAD_LATCH until the conflicting entry is retired.

If both a DREAD and an IREAD have a conflict in the WRITE_QUEUE, writes take priority until one of the reads no longer has a conflict. If the DREAD no longer has a conflict, the DREAD is then done. Then the WRITE_QUEUE continues to have priority over the IREAD_LATCH since the IREAD has a conflict, and when the conflicting writes are done, the IREAD may proceed. If another DREAD arrives in the meantime, it may be allowed to bypass both the writes and the IREAD if it has no conflicts.

This mechanism is used for other cases to enforce read/write ordering. Cases where the WRITE_QUEUE (and the WRITE_PACKER) must be flushed before proceeding are listed below:

1. DREAD_LOCK and WRITE_UNLOCK.
2. All IPR_READs and IPR_WRITEs (includes Clear Write Buffer).
3. All I/O space reads and I/O space writes.
4. Dread or Iread conflict with a write (checked to hexaword granularity, on address bits <31:5>).

When a DREAD_LOCK arrives from the MBOX, DWR_CONFLICT bits for all valid writes in the WRITE_QUEUE and WRITE_PACKER are set so that all writes (WRITE_QUEUE entries) preceding the DREAD_LOCK are done before the DREAD_LOCK is done.

When any IPR_READ arrives, all DWR_CONFLICT bits for valid entries in the WRITE_QUEUE and WRITE_PACKER are set, forcing the writes to complete before the IPR_READ completes. This ensures that IPR reads and writes are executed in order.

When any D-stream I/O space read arrives, all DWR_CONFLICT bits for valid entries in the WRITE_QUEUE and WRITE_PACKER are set, so that previous writes complete first.

When any I-stream I/O space read arrives, all IWR_CONFLICT bits for valid entries in the WRITE_QUEUE and WRITE_PACKER are set, so that previous writes complete first.

Note that when a WRITE_UNLOCK arrives, the WRITE_QUEUE is always empty as it was previously flushed before the READ_LOCK was serviced.

When a new entry for the DREAD_LATCH arrives, it is checked for conflicts with the WRITE_QUEUE. At this time the DWR_CONFLICT bit is set on any WRITE_QUEUE entry which is an I/O space write, an IPR_WRITE, or a WRITE_UNLOCK. Similarly, when a new entry for the IREAD_LATCH arrives, it is checked for conflicts with the WRITE_QUEUE. At this time the IWR_CONFLICT bit is set on any WRITE_QUEUE entry which is an I/O space write, an IPR_WRITE, or a WRITE_UNLOCK.

Thus, all transactions from the Mbox except memory space reads and writes unconditionally force the flushing of the WRITE_QUEUE. Memory space reads cause a flush if they conflict with a previous write.

13.6.3 I/O Space Writes

For WRITE commands with M%S6_PA<31:29> not '111, ADDRESS<33:32> = '00.

For WRITE commands with M%S6_PA<31:29> = '111, ADDRESS<33:32> = '11.

If the QW_PACK ipr is written and the mode is not WS_IO, the next two longwords are packed to the WRITE_QUEUE, otherwise the write is loaded directly.

13.6.3.1 NON-MASKED FLAMINGO I/O Writes

Flamingo workstations require I/O space writes to be mapped to channel addresses. For full LW writes (non-masked) then if the WS_IO bit of BIU_CTL is set with M%S6_PA<31:29> = '111 if either BM<3:0> = '1111 or BM<7:4> = '1111 the operation is a NON-MASKED I/O WRITE

- ADDRESS<31:29> = M%S6_PA<28:26>
- ADDRESS<28> = '0 if either BM<3:0> = '1111 or BM<7:4> = '1111 ; NON-MASKED I/O WRITE
- ADDRESS<27> = '0 for I/O
- ADDRESS<26:5> = '0 | M%S6_PA<25:5>
- ADDRESS<4:3> = M%S6_PA<4:3>
- Write_Queue data<63:0> = S6_DATA<63:0>
- Write_Queue_BM<7:0> = BM<7:0>, sets single LW_MASK bit, longword aligned write

13.6.3.2 MASKED FLAMINGO I/O Writes

If the WS_IO bit of BIU_CTL is set with M%S6_PA<31:29> = '111 if either BM<3:0> not '1111 or BM<7:4> not '1111, a byte or word write to I/O space is required then, the operation is a MASKED I/O WRITE. Note that I/O byte/word writes to the upper LW in FLAMINGO systems (i.e. address not quadword aligned) are UNPREDICTABLE.

- ADDRESS<31:29> = M%S6_PA<28:26>
- ADDRESS<28> = '1 if NOT (BM<3:0> = '1111 or BM<7:4> = '1111) ; MASKED I/O WRITE
- ADDRESS<27> = '0 for I/O
- ADDRESS<26:5> = M%S6_PA<25:5> | '0
- ADDRESS<4:3> = M%S6_PA<4:3>
- Write_Queue data<35:32> = BM<3:0>
- Write_Queue data<31:0> = S6_DATA<31:0>
- Write_Queue_BM<7:0> = '1111 1111, sets pair of LW_MASK bits, from M%S6_PA<4:3>

Thus a QW is written where bit

```
bit 32 is the byte mask for data<7:0>,   bit 33 is the byte mask for data<15:8>,
bit 32 is the byte mask for data<32:16>, bit 35 is the byte mask for data<31:24>
```

13.6.4 CM_OUT_LATCH

The CM_OUT_LATCH holds fill data and invalidate addresses which are destined for the Mbox. The Mbox never backpressures the Cbox (it can always receive a command from the Cbox) so a queue is not needed. The latch has an address portion and a data portion. The fields are shown in Table 13–12.

Table 13–12: CM_OUT_LATCH Fields

Field	Purpose
CMD<1:0>	Specific command being done.
ADDR<12:5>	PCache Index of the invalidate. This field is not used for fills.
InvReq<1:0>	PCache Set of the invalidate. This field is not used for fills.
FILL_QW<4:3>	Quadword alignment of the fill. This field is not used for invalidates.
DATA<63:0>	Fill data.

The CM_OUT_LATCH is loaded with an invalidate when pInvReq<1:0> is set by system logic.

The CM_OUT_LATCH is loaded with fill data when DREAD or IREAD data is obtained by either a Fast External Cache Hit or READ_BLOCK.

The command from the CM_OUT_LATCH is driven on C%CBOX_CMD<1:0>. If the command is an invalidate, the address is driven on C%CBOX_ADDR<11:5>, and no data is driven to the Mbox. If the command is a fill, the quadword alignment is driven on C%MBOX_FILL_QW<4:3>. (The Mbox has the hexaword address during these cycles.) Fill data is piped through the FILL_DATA_PIPEs and driven on B%S6_DATA<63:0>. The Cbox calculates byte parity on the fill data and drives it on B%S6_DP<7:0>.

If an IREAD is in progress in the Cbox and the MBOX asserts M%ABORT_CBOX_IRD, the Cbox prevents any further command, address, or data for that Iread from being driven to the Mbox, as described in Section 13.6.6.

Table 13–13: Cbox-Mbox interface control signals

Field	Purpose
C%CBOX_CMD<1:0>	Specific command being done: either D_CF, I_CF, INVAL, or NOP.
C%REQ_DQW	Indicates that the quadword of fill data being returned was the requested quadword of data: the quadword to which the original address corresponded. It is also asserted if C%CBOX_HARD_ERR is asserted and the requested quadword has not yet been returned; the Mbox then notifies the Ibox and/or Ebox that the requested data has been returned so that the machine does not hang.
C%LAST_FILL	Indicates that this is the last data being sent for the read request.
C%CBOX_HARD_ERR	Indicates that an unrecoverable error is associated with the data. This bit only qualifies fills, not invalidates. When C%CBOX_HARD_ERR is asserted, the Cbox also asserts C%LAST_FILL as no more fills follow. C%CBOX_HARD_ERR may be asserted as the result of an uncorrectable error in the Bcache or as the result of RDE on the NDAL.

Table 13–13 (Cont.): Cbox-Mbox interface control signals

Field	Purpose
C%CBOX_ECC_ERR	Indicates that a correctable backup cache ECC error is associated with the current fill data and the data should be ignored. Valid for fills only, not invalidates. Corrected data will follow.

If an error happens while fill data is being retrieved, the Cbox notifies the Mbox using C%CBOX_HARD_ERR or C%CBOX_ECC_ERR. Table 13–14 shows how both normal cases and error cases are handled by the Mbox.

Table 13–14: Cbox_Mbox commands and actions

C%CBOX_CMD<1:0>	Qualifiers asserted	Mbox Action
NOP		Take no action.
I_CF		Accept fill data for outstanding IREAD.
D_CF		Accept fill data for outstanding DREAD.
I_CF or D_CF	C%CBOX_HARD_ERR, C%LAST_FILL	Perform invalidate, expect no more fills for this read.
I_CF or D_CF	C%CBOX_ECC_ERR	Ignore this fill data, expect fill later.
INVAL		Perform invalidate.
INVAL to outstanding fill		Perform invalidate, expect fill data. Do not validate the data in the Pcache when it returns.

13.6.5 FILL_DATA_PIPE1 and FILL_DATA_PIPE2

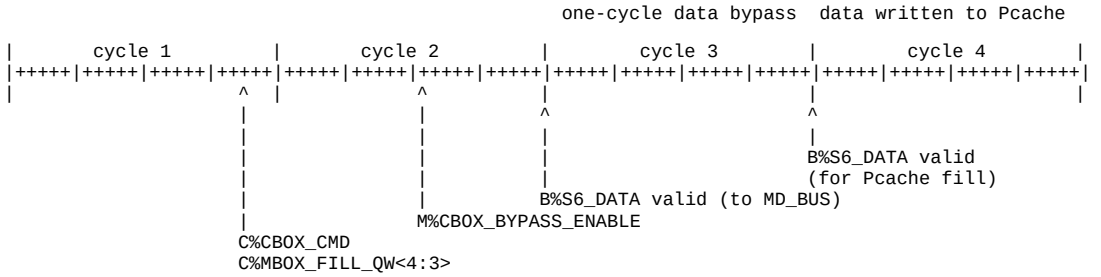
The FILL_DATA_PIPEs are used to pipeline the fill data for two cycles so that the Cbox drives B%S6_DATA coincidentally with the write-enable of the Pcache. If there is a free cycle on B%S6_DATA, the Cbox may bypass the fill data from the FILL_DATA_PIPE1 (to achieve a one-cycle bypass). This allows the Mbox to return data to the Ibox or the Ebox one cycle early. The cache fill to the Pcache is done in the normal cycle, driven from FILL_DATA_PIPE2, even if Ebox or Ibox data was bypassed in an earlier cycle. The timing relationships for one cache fill are shown in Figure 13–6.

In this example, a fill is just arriving in cycle 1, so the Cbox drives C%CBOX_CMD and C%MBOX_FILL_QW<4:3>.

The Mbox drives M%CBOX_BYPASS_ENABLE to the Cbox in cycle 2 to indicate that B%S6_DATA is free during the current cycle. This causes the Cbox to bypass data from FILL_DATA_PIPE1 to B%S6_DATA to achieve a one-cycle bypass.

In cycle 3 the Cbox drives the data from FILL_DATA_PIPE2 to the Pcache for the write. It does this even though the bypass was done previously, because the Pcache is always written in the third cycle after C%CBOX_CMD is driven with the fill command.

Figure 13–6: B%S6_DATA bypass timing



The rules for the Cbox driving data on B%S6_DATA are as follows:

1. IF FILL_DATA_PIPE2 contains valid data, drive B%S6_DATA from FILL_DATA_PIPE2
2. ELSE IF M%CBOX_BYPASS_ENABLE is asserted and FILL_DATA_PIPE1 contains valid data, drive from FILL_DATA_PIPE1 to achieve a one-cycle bypass.

The Mbox keeps enough state to know what the Cbox will be bypassing in any given cycle.

When the Cbox drives B%S6_DATA, it also generates byte parity and drives B%S6_DP with the same timing.

The fields of the FILL_DATA_PIPEs are shown in Table 13–15.

Table 13–15: Fields of FILL_DATA_PIPE1 and FILL_DATA_PIPE2

Field	Purpose
IREAD	Indicates that fill data is for an IREAD.
DATA<63:0>	Fill data.

The IREAD field is necessary in case of an IREAD abort, as described in Section 13.6.6. If M%ABORT_CBOX_IRD is asserted and the data in either FILL_DATA_PIPE1 or FILL_DATA_PIPE2 is for an IREAD, that FILL_DATA_PIPE must be cleared so that data is not driven back to the Mbox.

13.6.6 IREAD Aborts

The Mbox asserts the signal M%ABORT_CBOX_IRD to notify the Cbox to abort any IREAD which it is currently processing. This may happen because of a branch mispredict where the Istream has been prefetching from one branch and has to change over to the other. The Mbox then aborts all outstanding IREADs so that a new IREAD can begin.

When the Cbox receives the abort signal, the read in question may be anywhere in the Cbox read sequence. The exact action taken depends on where the read is, as shown in Table 13–16.

Table 13–16: Cbox Action Upon Receiving M%ABORT_CBOX_IRD

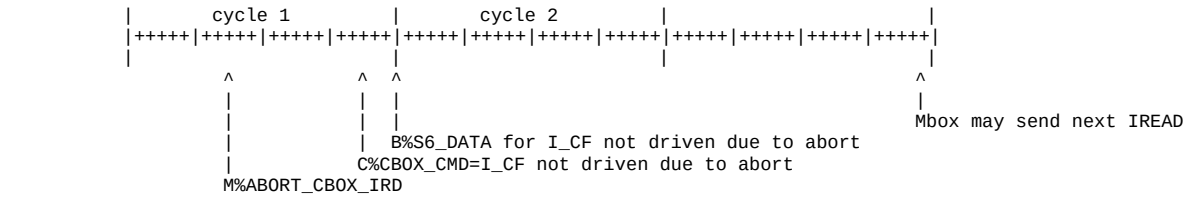
State of the IREAD	Action Taken by the Cbox
No IREAD outstanding	No action taken.
IREAD_LATCH valid but not started	Clear the IREAD_LATCH so the request will not be started.
IREAD in progress	Clear the TO_MBOX bit. When the fill data returns, don't send the data to the Mbox.
IREAD fill data in CM_OUT_LATCH or FILL_DATA_PIPEs	Clear the entry containing IREAD data so that the data is not returned to the Mbox.

Figure 13–7 shows an example of timing for the Cbox abort response. In cycle 1, M%ABORT_CBOX_IRD is asserted during phase 2. The Cbox is ready to drive the I_CF command and B%S6_DATA during phase 4. The assertion of M%ABORT_CBOX_IRD prevents both of those actions.

The next IREAD may appear two cycles after the abort.

If M%ABORT_CBOX_IRD is received after the system backmaps have been instructed to map the reference either by pMapWE for cache hits or by a READ_BLOCK for a miss, the Pcache index to which the IREAD was to be done must be invalidated to avoid the Pcache from maintaining a block which is not backmapped. If IABORT is taken after the ARB sequencer has advanced to 'RDN' (read second octaword), 'SYS_READ' (read block), or 'FILL' (wait for data to be loaded

Figure 13–7: M%ABORT_CBOX_IRD Timing



to Pcache), an invalidate of the location to which the block was to be allocated is driven to the CM_OUT_LATCH.

13.7 Arbiter/Bus Control

The Arbitration/Bus Control Sequencer selects the highest priority command from the DREAD_ LATCH, IREAD_LATCH, or Write Queue.

The following sequences are executed;

1. DREAD
2. READ LOCK
3. IPR READ
4. IREAD
5. WRITE
6. WRITE BYTE/WORD
7. WRITE UNLOCK
8. IPR_WR

13.7.1 Dispatch Controller

The ARB/Bus Control Sequencer controls two satellite machines, the DISPATCH and FILL controllers. The DISPATCH controller selects the next command, controls the WRITE_QUEUE pointers, and drives the required address to the pads. When the Arb Machine is ready to process a new read or write request the DISPATCH controller is enabled. In the first cpu cycle of dispatching a read or write command, the DISPATCH controller determines which command is highest priority and asserts the command code to the ARB Sequencer. The Dispatch commands are,

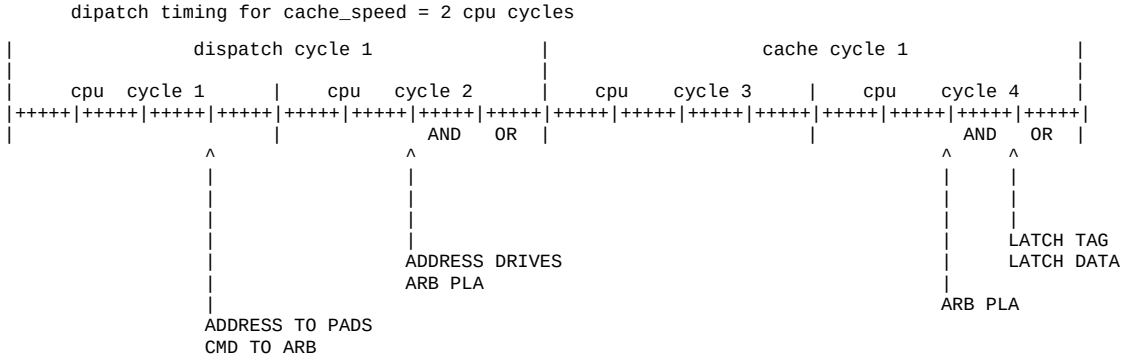
1. DREAD: DREAD_LATCH valid with DREAD CMD not io_space address and no Dread/Write Conflict bits are set
2. DREAD_IO: DREAD_LATCH valid with DREAD CMD io_space address and no Dread/Write Conflict bits are set
3. DREAD_LOCK: DREAD_LATCH valid with READ_LOCK CMD and no Dread/Write Conflict bits are set
4. IPR_READ: DREAD_LATCH valid with IPR_READ CMD and no Dread/Write Conflict bits are set

5. IREAD: the DREAD_LATCH is empty or Dread/Write Conflict bits are set in the Write Queue and IREAD_LATCH valid not io_space address and no Iread/Write Conflict bits are set
6. IREAD_IO: the DREAD_LATCH is empty or Dread/Write Conflict bits are set in the Write Queue and IREAD_LATCH valid, io_space address and no Iread/Write Conflict bits are set
7. WRITE_UNLOCK: the DREAD_LATCH is not valid or Dread/Write Confict, and the IREAD_ LATCH is not valid or Iread/Write Confict, and the Write Queue CMD = Write_Unlock and not io_space address
8. WRITE: the DREAD_LATCH is not valid or Dread/Write Confict, and the IREAD_LATCH is not valid or Iread/Write Confict, and the Write Queue CMD = Write and not io_space address
9. IO_WRITE: the DREAD_LATCH is not valid or Dread/Write Confict, and the IREAD_LATCH is not valid or Iread/Write Confict, and the Write Queue CMD = Write and io_space address
10. WRITE_UNLOCK_IO: the DREAD_LATCH is not valid or Dread/Write Confict, and the IREAD_LATCH is not valid or Iread/Write Confict, and the Write Queue CMD = Write_ Unlock and io_space address
11. IPR_WRITE: the DREAD_LATCH is not valid or Dread/Write Confict, and the IREAD_ LATCH is not valid or Iread/Write Confict, and the Write Queue CMD = IPR_WRITE
12. NOP:the DREAD_LATCH is not valid or Dread/Write Confict, and the IREAD_LATCH is not valid or Iread/Write Confict, and the Write Queue is empty

NOTE: READ_LOCK to I/O space is not implemented.

By the phase 1 of the second cpu cycle of a dispatch request the selected address from either the DREAD latch, IREAD latch, or WRITE QUEUE is driven onto the internal address bus to the pads. By the next phase 3 the selected address starts to be driven externally. The ARB controller changes state once per cache_speed (i.e. 2,3, or 4)cpu cycles, with the ARB 'AND' array enabled at phase 3, and the ARB 'OR' array selecting during phase 4.

Figure 13–8: DISPATCH timing



The DREAD latch or IREAD latch can receive a new request as late as phase 2 of cpu cycle 1 of the dispatch. The Dispatch command and address source are determined in phase 3 and the address is

driven to the pads in phase 4 of cpu cycle 1 allowing 3 phases to drive the address to the pad drivers. The D and I conflict bits for a newly received READ request are not determined until phase 1 of cpu cycle 2. The I and D conflict bits are sent with the dispatch command to the ARB Controller. If the dispatch command is DREAD, DREAD_IO, DREAD_LOCK, or IPR_READ and a D conflict exists, or the dispatch command is IREAD, or IREAD_IO and an I conflict exists the dispatch_in signal is cleared and the ARB state remains 'IDLE' for the next SYS_CLK cycle.

13.7.2 Fill Controller

The FILL controller checks ECC or parity, corrects single bit ECC errors, sets BIU_STAT on errors, moves input data to the CM_OUT_LATCH, merges write data and generates check bits when enabled by the ARB sequencer. The FILL controller is started by FILL_CMDs from the ARB sequence.

1. FILL_IDLE - wait for command
2. FILL_RD_1 - fill first octaword of cache read
3. FILL_RD_2 - fill second octaword of cache read
4. FILL_SYS - fill block from READ_BLOCK or LDxL, or QW if IO_SPACE
5. FILL_BWM_SYS - merge write data with LDxL data from system, generate ECC
6. FILL_EG - generate ECC on write data
7. FILL_BWM_DIR - merge write data with cache read data, generate ECC

The fill rate is limited to one quadword every two cpu cycles.

13.7.3 ARB PLA INPUTS

The following signals are inputs to the ARB PLA "AND ARRAY" and are used in determining the next output and state transition of the ARB Sequencer.

```
dsp_cmd<3:0>            - Dispatch Commands
arb_state<4:0>          - ARB STATE
cack<2:0>               - IDLE, HARD_ERROR, SOFT_ERROR, STxC_FAIL, OK
dispatch_in             - dispatch command present
bcache_en               - BIU_CTL<0> = '1
not bcache_en or "PV"   - BIU_CTL<0> ='0 or BIU_CTL<PV> = '1
hold_in                 - hold_req and dispatch and not (WRITE, WRITE_UNLOCK, or WRITE_IO)
hold_req                - holdReq_h pin is asserted
err_in                  - error detection enabled (err_flag) and an error is detected
stall_req               - tagOK_l and holdReq_h are checked at phase 4
                              (synchronized from last phase 3 of cache probe cycle)
stall_wr                - not tagOK_l or hold request at phase 4 of last cpu cycle of ARB state
ird_abort               - IABORT
same_octaword           - from WRITE_QUEUE, pack QW unless OUT_BUF not empty
byte_word_write         - WRITE_QUEUE BM<7:4 or 3:0> not '1111 or '0000
bwr_chain               - byte/word write in progress
fill_done               - Fill Sequencer operation completed
read_hit                - match, valid, correct tag and ctrl parity
write_hit               - match, valid, not shared, correct tag and ctrl parity
```

13.7.4 ARB PLA OUTPUTS

The ARB PLA outputs next state, enable, and data path control signals.

```
arb_state               - next ARB STATE
dispatch_flag           - enable dispatch_in next access
hold_en                 - enable hold
err_flag                - enable error logic/input
tagok_stall             - block fill done latch
iread_chain_set         - set iread in progress
pcread_chain_set        - set Pcache read in progress
io_chain_set            - set IO in progress
bwr_chain_set           - set bwr in progress
all_chains_clr          - clear all in progress state
FILL_CMD<2:0>           - IDLE, RD_1, RD_2, SYS, BWM_SYS, EG, BWM_DIR
data_write_reg_ld_en    - load OUT_BUF with QW being packed
ipr_rd_en               - return ipr read data
ipr_wr_en               - WRITE_QUEUE data to ipr
rl_retire_en            - clear I or D read latch valid flag
pMapWE_en               - enable map write strobe
lw_mask_calc_en         - set LWMask<7:0> from address<4:3> and WRITE_QUEUE byte mask bits
new_addr4_ld            - toggle dataA<4> at phase 3 of last cpu cycle of next ARB cycle
ce_en                   - assert dataCEOE<3:0> and tagCEOE
tce_en                  - assert tagCEOE
tag_probe_req           - enable tag compare
tce_dis                 - deassert tagCEOE at end of next SYS_CLK cycle
dataceoe_dis            - deassert data chip enables dataCEOE<3:0> at end of next SYS_CLK
in_data_lat_en          - latch cache input at end of next SYS_CLK cycle
wr_arm_en               - causes the dataWE_h<3:0> signals to be "armed"
sys_dp_ctrl_en          - data path control to fill sequencer
creq_lat_en             - latch new CREQ
CREQ<2:0>               - IDLE, READ_BLOCK, WRITE BLOCK, LDxL, STxC
```

13.7.5 IDLE

'IDLE' is the next state upon the completion of all ARB sequences. Dispatch_flag is not asserted when entering 'IDLE', therefore a one SYS_CLK nop cycle exists between ARB requests. The 'IDLE' term enables dispatch_flag allowing the next request to processed. **When the Serial Rom is being read by microcode, the SROM is output enabled (SOE-IE[SROM_OE] = '1), the dispatch_in signal is seen as deasserted by the ARB PLA if the dispatch command is WRITE. This allows microcode to write data to Pcache, with the corresponding write through data going to the Write_Queue. The external WRITE request from the queue is "dropped" while the SROM data is transferred to Pcache.**

13.7.6 DISPATCH

This section describes the dispatch fork; the outputs enabled in response to the dispatch selection, and the next ARB state selection.

1. NOP and not hold_in: 'IDLE'

```
dispatch_flag     - retry dispatch
hold_en           - enable hold
```

2. DREAD and Bcache enabled and not hold_in: 'DRD', start fast external cache read sequence

```
pcread_chain_set  - set Pcache read in progress
FILL_RD_1         - fill of first octaward begins at end of next SYS_CLK cycle
tce_dis           - deassert tag chip enable at end of next SYS_CLK cycle
tag_probe_req     - start tag compare at end of next SYS_CLK cycle
in_data_lat_en    - latch cache input at end of next SYS_CLK cycle
```

3. DREAD and Bcache not enabled and not hold_in: 'SYS_RD', no Bcache direct to system read

```
err_flag          - enable err_in (cack = hard error)
pcread_chain_set  - set Pcache read in progress
FILL_SYS          - fill block when CACK = OK or SOFT
sys_dp_ctrl_en    - data path control to fill sequencer
creq_lat_en       - latch new CREQ
CREQ              - READ_BLOCK
```

4. DREAD_IO and not hold_in: 'SYS_RD', I/O Space direct to system read

```
err_flag          - enable err_in (cack = hard error)
FILL_SYS          - fill target QW (not pcread_chain_set) when CACK = OK or SOFT
sys_dp_ctrl_en    - data path control to fill sequencer
creq_lat_en       - latch new CREQ
CREQ              - READ_BLOCK
io_chain_set      - set IO in progress
```

5. DREAD_LOCK and not hold_in: 'SYS_RD', read_lock, MUST LOCK OUT IREADS TILL STxC pass or IPR_WR

```
err_flag          - enable err_in (cack = hard error)
pcread_chain_set  - set Pcache read in progress
FILL_SYS          - fill block when CACK = OK or SOFT
sys_dp_ctrl_en    - data path control to fill sequencer
creq_lat_en       - latch new CREQ
CREQ              - LDxL
```

6. IREAD and Bcache enabled and not IABORT and not hold_in: 'IRD', start fast external cache read sequence

```
pcread_chain_set  - set read in progress
iread_chain_set   - set iread in progress
FILL_RD_1         - fill of first octaward begins at end of next SYS_CLK cycle
tce_dis           - deassert tag chip enable at end of next SYS_CLK cycle
tag_probe_req     - start tag compare at end of next SYS_CLK cycle
in_data_lat_en    - latch cache input at end of next SYS_CLK cycle
```

7. IREAD and not Bcache enabled and not IABORT and not hold_in: 'SYS_RD', no Bcache direct to system read, set iread

```
err_flag          - enable err_in (cack = hard error)
pcread_chain_set  - set Pcache read in progress
iread_chain_set   - set iread in progress
FILL_SYS          - fill block when CACK = OK or SOFT
sys_dp_ctrl_en    - data path control to fill sequencer
creq_lat_en       - latch new CREQ
CREQ              - READ_BLOCK
```

8. IREAD_IO and not IABORT and not hold_in: 'SYS_RD', I/O Space direct to system read, set iread and IO in progress

```
err_flag          - enable err_in (cack = hard error)
iread_chain_set   - set iread in progress
FILL_SYS          - fill block when CACK = OK or SOFT
sys_dp_ctrl_en    - data path control to fill sequencer
creq_lat_en       - latch new CREQ
CREQ              - READ_BLOCK
io_chain_set      - set IO in progress
```

9. IREAD or IREAD_IO and IABORT and not hold_in: 'IDLE', IABORT before iread starts

```
dispatch_flag     - retry dispatch
hold_en           - enable hold
```

10. IPR_READ and not hold_in: 'IDLE' , ipr_rd_en, rl_retire_en
11. IPR_WRITE and not hold_in: 'IDLE' , ipr_wr_en

12. WRITE and byte_word and not "PV" and Bcache enabled and not hold request: 'BWR_PROBE', start cache read for RMW

```
bwr_chain_set     - set bwr in progress
lw_mask_calc_en   - set LWMask<7:0> from address<4:3> and WRITE_QUEUE byte mask bits
FILL_BWM_DIR      - merge target QW from cache at end of next SYS_CLK cycle
tce_dis           - deassert tag chip enable at end of next SYS_CLK cycle
dataceoe_dis      - deassert data chip enables at end of next SYS_CLK
tag_probe_req     - start tag compare at end of next SYS_CLK cycle
in_data_lat_en    - latch cache input at end of next SYS_CLK cycle
```

13. WRITE and byte_word and not "PV" and Bcache enabled and hold request: 'BWR_STALL', wait for holdreq to deassert

```
bwr_chain_set     - set bwr in progress
hold_en           - enable hold
ce_en             - assert dataCEOE<3:0>
```

14. WRITE and byte_word and not "PV" and not Bcache enabled: 'BWR_SYS_RD', byte_word write, no cache, not "PV"

```
err_flag          - enable err_in (cack = hard error)
FILL_BWM_SYS      - merge target QW when CACK = OK or SOFT
sys_dp_ctrl_en    - data path control to fill sequencer
creq_lat_en       - latch new CREQ
CREQ              - LDxL
bwr_chain_set     - set bwr in progress
lw_mask_calc_en   - set LWMask<7:0> from address<4:3> and WRITE_QUEUE byte mask bits
```

15. WRITE and not byte_word and same_octaword: 'IDLE', enable PACK_WRITE to OUT_BUF

```
hold_en           - enable hold
FILL_EG    - generate ECC on write data
data_write_reg_en - load OUT_BUF with QW being packed
lw_mask_calc_en   - set LWMask<7:0> from address<4:3> and WRITE_QUEUE byte mask bits
```

16. WRITE and not byte_word and not "PV" and not same_octaword and Bcache enabled and not hold request: 'WR_PROBE', start fast external tag read

```
lw_mask_calc_en   - set LWMask<7:0> from address<4:3> and WRITE_QUEUE byte mask bits
FILL_EG    - generate ECC on write data
data_write_reg_en - load OUT_BUF with QW being packed
tce_dis           - deassert tag chip enable at end of next SYS_CLK cycle
tag_probe_req     - start tag compare at end of next SYS_CLK cycle
```

17. WRITE and not byte_word and not "PV" and not same_octaword and Bcache enabled and hold request: 'WR_STALL', wait for holdreq to deassert

```
lw_mask_calc_en   - set LWMask<7:0> from address<4:3> and WRITE_QUEUE byte mask bits
FILL_EG    - generate ECC on write data
data_write_reg_en - load OUT_BUF with QW being packed
hold_en           - enable hold
ce_en             - assert dataCEOE<3:0>
```

18. WRITE and (not byte_word or "PV") and not same_octaword and (not Bcache enabled or "PV"): 'SYS_WR', no cache or "PV", start system write

```
err_flag          - enable err_in (cack = hard error)
sys_dp_ctrl_en    - data path control to fill sequencer
lw_mask_calc_en   - set LWMask<7:0> from address<4:3> and WRITE_QUEUE byte mask bits
FILL_EG    - generate ECC on write data
data_write_reg_en - load OUT_BUF with QW being packed
creq_lat_en       - latch new CREQ
CREQ              - SYS_WR
```

19. IO_WRITE: 'SYS_WR', IO space write direct to WRITE_BLOCK

```
err_flag          - enable err_in (cack = hard error)
sys_dp_ctrl_en    - data path control to fill sequencer
lw_mask_calc_en   - set LWMask<7:0> from address<4:3> and WRITE_QUEUE byte mask bits
FILL_EG    - generate ECC on write data
data_write_reg_en - load OUT_BUF with QW being packed
creq_lat_en       - latch new CREQ
CREQ              - SYS_WR
io_chain_set      - set IO in progress
```

20. WRITE_UNLOCK: 'BWR_SYSMERGE', assume all write_unlocks to byte_word type, get data from IN_BUF

```
lw_mask_calc_en   - set LWMask<7:0> from address<4:3> and WRITE_QUEUE byte mask bits
FILL_BWM_DIR      - merge target QW from cache at end of next SYS_CLK cycle
```

21. WRITE_UNLOCK_IO: 'SYS_WR', IO space write direct to STxC

```
err_flag          - enable err_in (cack = hard error)
sys_dp_ctrl_en    - data path control to fill sequencer
lw_mask_calc_en   - set LWMask<7:0> from address<4:3> and WRITE_QUEUE byte mask bits
FILL_EG    - generate ECC on write data
data_write_reg_en - load OUT_BUF with QW being packed
creq_lat_en       - latch new CREQ
CREQ              - STxC
io_chain_set      - set IO in progress
```

22. hold_in: hold request and hold_en and not dispatch of (WRITE or WRITE_IO or WRITE_ UNLOCK): 'STALL', keep hold_en

13.7.6.1 PACK_WRITE

The Write_packer asserts the same_octaword bit in a Write_queue entry when a new write request is to the alternate QW of the octaword which is presently in the Write_Packer, and the Write_ Packer byte mask bits indicate only full Longwords.

When a write command is received by the ARB Controller from the Write_queue with same_ octaword, it is known the next entry will be to the same octaword, so entry of 1 or 2 LWs is moved to the OUT_BUF, and the write bus cycle is deffered till the next Write command. **If the same_octaword bit is set in Write_Queue and the OUT_BUF is not empty, the write address is returning to the quadword already packed in the OUT_BUF. Since this write may not be to same LW as the previous one, packing at this point can not proceed. The ARB pla for same_octaword is deasserted and the write bus cycle proceeds.**

The quadword of data with ECC check bits (or parity) is moved to OUT_BUF<63:0> if Address<3> = '0, and to OUT_BUF<127:64> if Address<3> = '1. The LW_MASK register is set from the byte mask bits BM<7:0> as

- if address<4:3> = '00 LW_MASK<0> = '1 if BM<3:0> is not '0000
- if address<4:3> = '00 LW_MASK<1> = '1 if BM<7:4> is not '0000
- if address<4:3> = '01 LW_MASK<2> = '1 if BM<3:0> is not '0000
- if address<4:3> = '01 LW_MASK<3> = '1 if BM<7:4> is not '0000
- if address<4:3> = '10 LW_MASK<4> = '1 if BM<3:0> is not '0000
- if address<4:3> = '10 LW_MASK<5> = '1 if BM<7:4> is not '0000
- if address<4:3> = '11 LW_MASK<6> = '1 if BM<3:0> is not '0000
- if address<4:3> = '11 LW_MASK<7> = '1 if BM<7:4> is not '0000

When same_octaword indicates the present WRITE_QUEUE QW is to be packed at the OUT_BUF, the valid longwords are set as

- X0 = '1 if BM<3:0> is not '0000
- X1 = '1 if BM<7:4> is not '0000

and are used to indicate the byte masks for the packed QW in "PV" writes.

13.7.6.2 IPR_READ

The Arb Control State machine executes an IPR_RD if an IPR_RD is in the DREAD_LATCH and no Dread/Write Conflict bits are set (i.e. the Write Queue has emptied).

The IPR address is decoded and the data is driven to the CM_OUT_LATCH and the DREAD_LATCH clears. The next state is 'IDLE', dispatch is not enable.

13.7.6.3 HIGH_LW_TEMP

When a quadword aligned read of I/O space is performed the high LW of data is latched in this register. When a non quadword aligned read to I/O space is dispatched and BIU_CTL<QW_I/O_RD> = '1 then the data from HIGH_LW_TEMP is returned as if an IPR_READ. The bus cycle is not done.

13.7.6.4 DREAD_LOCK

The Arb Control State Machine sequences directly to the 'SYS_RD' state if a DREAD_LOCK is in the DREAD_LATCH and no Dread/Write Conflict bits are set (i.e. the Write Queue has emptied), and tagOK_l and holdReq_h are deasserted.

DREAD_LOCK is issued by microcode for interlock instructions. No further I stream references are tried until the data read via the DREAD_LOCK is modified and successfully writen back to memory using a STxC bus cycle that is CommandACKnowledged OK. After modifying the read_lock data microcode issues a write_unlock which results in a STxC. Microcode then reads the STxC_IPR to see if the data was written successfully. If the STxC indicates fail, the interlock could not be completed, and microcode retries the sequence from the DREAD_LOCK.

If a DREAD_LOck results in a hard error, the error handler executes an IPR_WR to CEFSTS to restart I stream processing.

The DREAD_LOCK dispatch sets a flop inhibiting IREADS until STxC is executed successfully or an IPR_WR (CEFSTS @ AC(hex)) is received at the CBOX.

13.7.6.5 WRITE

A non byte write is the highest priority bus request when,

```
the DREAD_LATCH is not valid or Dread/Write Confict
the IREAD_LATCH is not valid or Iread/Write Confict
the Write Queue CMD = Write
BM<7:4> = '1111 or '0000 or "PV"
BM<3:0> = '1111 or '0000 or "PV"
```

The WRITE_QUEUE address is moved to the pads and the data is latched ECC/parity generate section, and the WRITE_QUEUE head is advanced for a dispatch with CMD = Write. The possible ARB breakouts are,

- 'PACK_WRITE' if SAME_OCTAWORD and the OUT_BUF is empty (LW_MASK<7:0> = '00000000)
- 'WRITE_WAIT' if not SAME_OCTAWORD or the OUT_BUF is not empty and hold_req
- 'WRITE_PROBE' if not SAME_OCTAWORD or the OUT_BUF is not empty and not hold_req and (bcache_en and not "PV")
- 'SYS_WRITE' if not SAME_OCTAWORD or the OUT_BUF is not empty and not hold_req and (bcache_en or "PV")

The Write Queue data with ECC check bits is moved to OUT_BUF<63:0> if Address<3> = '0, and to OUT_BUF<127:64> if Address<3> = '1, and the appropriate LW_MASK bits are set as in the PACK_WRITE dispatch.

13.7.6.6 BWR

If a byte write is the highest priority bus request,

```
the DREAD_LATCH is not valid or Dread/Write Confict
the IREAD_LATCH is not valid or Iread/Write Confict
the Write Queue CMD = Write
not "PV" mode
either BM<7:4> is not (’1111 or ’0000)
 0r BM<3:0> is not (’1111 or ’0000)
```

the 'BWR_PROBE' state is entered if not stall_request else 'BWR_STALL'.

Byte and word writes for "PV" mode go directly to 'SYS_WRITE'.

13.7.6.7 WRITE_UNLOCK

If a Write_Unlock is the highest priority bus request,

```
the DREAD_LATCH is not valid or Dread/Write Confict
the IREAD_LATCH is not valid or Iread/Write Confict
the Write Queue CMD = Write_Unlock
```

the 'SYS_WR' state is entered. cReq_h<2:0> is driven with STxC, and cWMask<7:0> is driven from LW_MASK<7:0> if "PV", else from BM>7:0>. The ARB state remains 'SYS_WR' until cAck is not idle.

```
if cAck is IDLE, ARB state remains ’SYS_WR’
if cAck is HARD_ERROR the error is logged,
                c%cbox_h_err is asserted, microcode is signalled STxC PASS so as not to retry
if cAck is SOFT_ERROR the error is logged,
                c%cbox_s_err is asserted, proceed as OK
if cAck is STxC_FAIL, the STxC IPR bit 2 is set to ’1.
if cAck is OK,the STxC IPR bit 2 is set to ’0.
if cAck is OK or STxC_FAIL, the next state ’IDLE’
```

An IPR read of the STxC register follows the Write_Unlock. Microcode repeats the interlock loop (i.e. read_lock/write_unlock) if the STxC register indicates fail. **An IPR_RD of STxC with bit 2 = '0, renables CBOX IREAD processing and renables the MBOX IREF latch.** If the READ_ LOCK reults an a hard error microtrap, microcode executes an IPR_WR (CEFSTS @ AC(hex)) to renable the CBOX IREAD processing and the MBOX IREF latch.**

13.7.7 DRD

The DREAD address began driving at phase 3 of the second cpu cycle of the Dispatch Cycle. The 'DREAD' state is 2,3, or 4 cpu cycles in duration as programmed from cache_speed. At the phase 4 of the last cpu cycle of 'DRD'

- tagAdr_h<31:17>, tagAdrP_h, tagCtlV_h, tagCtlD_h, tagCtlS_h, and tagCtlP_h are latched
- data_h<127:0> and check_h<27:0> are latched in the INPUT_BUF<dataA_h<4>>.
- tagCEOE is deasserted

The next state is 'RDC'.

```
err_flag         - enable err_in (tag or ctl parity)
new_addr4_ld     - toggle dataA<4> at phase 3 of last cpu cycle of next ARB cycle
pmapwe_en        - assert pmapwe if cache data fills Pcache
```

13.7.8 IRD

The IREAD address began driving at phase 3 of the second cpu cycle of the Dispatch Cycle. The 'IREAD' state is 2,3, or 4 cpu cycles in duration as programmed from cache_speed. At the phase 4 of the last cpu cycle of 'IRD'

- tagAdr_h<31:17>, tagAdrP_h, tagCtlV_h, tagCtlD_h, tagCtlS_h, and tagCtlP_h are latched
- data_h<127:0> and check_h<27:0> are latched in the INPUT_BUF<dataA_h<4>>.
- tagCEOE is deasserted

1. If IABORT, the next state is 'IDLE'.

   ```
   dispatch_flag    - enable dispatch_in next access
   hold_en          - enable hold
   dataceoe_dis     - deassert data chip enables at end of next SYS_CLK
   all_chains_clr   - clear all in progress state
   ```

   If ABORT_CBOX_IRD is asserted the loading of the CM_OUT_LATCH is inhibited so that data is not returned to the MBOX. ABORT_CBOX_IRD inhibits errors from the IREAD.

   IABORT is inhibited when pcread_chain and not iread_chain.

2. If not IABORT, the next state is 'RDC', pla outputs same as 'DRD'.

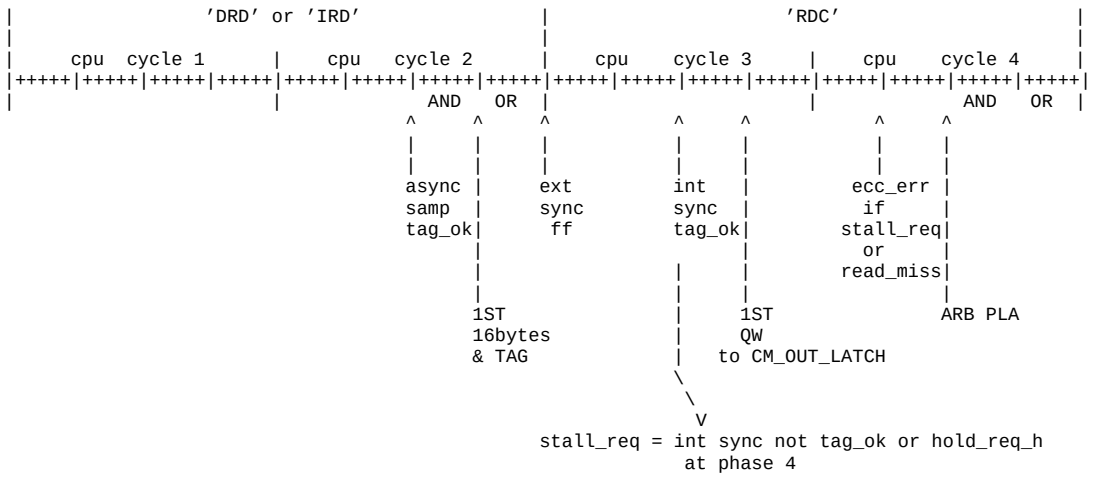
   ```
   err_flag         - enable err_in (tag or ctl parity)
   new_addr4_ld     - toggle dataA<4> at phase 3 of last cpu cycle of next ARB cycle
   pmapwe_en        - assert pmapwe if cache data fills Pcache
   ```

13.7.9 RDC

In the first cpu cycle of 'RDC'

- The target quadword is moved from the data pads to the ECC, ECC check begins at phase 3
- The target quadword is loaded into CM_OUT_LATCH at phase 4 and C_PIPE_%REQ_DQW is set to tag the selected quadword of data.
- Address<31:21/17> is compared to tagAdr_h<31:21/17> as specified by cache_size, tagCtlV_h is checked, and tag and control parity are checked.

Figure 13–9: stall_req timing



- tagOK_l and holdReq_h are checked at phase 4 (synchronized from last phase 3 of cache probe cycle)

read_hit is determined as

```
tagAdr<31:22/17> matches adr_h<31:22/17>
tagCtlV_h is true
tagCtlP_h and tagAdrP_h are correct
or force hit
```

stall request is not tagOK_L or hold request at phase 4 of first cpu cycle of ARB state.

In the second cpu cycle of 'RDC'

- At phase 1 both read hit, and no ECC error are valid
- At phase 2 if not read hit, or ECC error, or stall request, then C%CBOX_ECC_ERR is asserted causing the MBOX to ignore the data in CM_OUT_LATCH
- At phase 2 if read hit and not stall request the proper pMapWE signal is asserted to support system backmaps of Pcache

In the last cpu cycle of 'RDC'

- At phase 3 dataA_h<4> toggles to begin access of second octaword
- At phase 3 the ARB sequencer determines the next state

If cache_speed is 3 or 4 cpu cycles the FILL machine loads the second quadword of the block during cpu cycle 3 of the 'RDC' state if ECC was good for the target QW.

1. If not IABORT and stall request, the next state is 'STALL', wait for stall request to end (returning the cache resource to the NVAX Plus chip)

```
tagok_stall      - block fill done latch
hold_en          - enable hold
all_chains_clr   - clear all in progress state
```

2. If not IABORT and not stall request and read_hit, the next state is 'RDN'.

```
FILL_RD_2        - fill QWs 3 and 4
err_flag         - enable error logic/input
dataceoe_dis     - deassert data chip enables at end of next SYS_CLK
in_data_lat_en   - latch cache input at end of next SYS_CLK cycle
rl_retire_en     - clear I or D read latch valid flag
```

3. If not IABORT and not stall request and not read_hit, the next state is 'SYS_RD'.

```
FILL_SYS         - fill block when dRack
creq_lat_en      - latch new CREQ
CREQ             - READ_LOCK
err_flag         - enable error logic/input
dataceoe_dis     - deassert data chip enables at end of next SYS_CLK
sys_dp_ctrl_en   - data path control to fill sequencer
```

4. If IABORT, the next state is 'IDLE', the IREAD_LATCH valid bit is cleared, need to remove index in Pcache which system backmap replaced!!
5. If tagOk_l and either tagCtlP_h and tagAdrP_h are not correct, the fill is stopped, the error is logged, c%cbox_s_err is asserted, and the ARB state returns to 'IDLE'.

13.7.10 RDN

The address for the second octaword began driving the previous phase 3. For cache_speed = 2 timing the second quadword is moved to the CM_OUT_LATCH during this state. At phase 2 of the first cpu cycle of 'RDN' the pMapWE is deasserted. At phase 4 of the last cpu cycle of 'RDN' the second quadword is latched at the data pads, and the fill sequencer is notified that the second octaword is present.

1. If not IABORT, the next state is 'FILL', enable err_flag.
2. If IABORT, the next state is 'IDLE'. The IREAD_LATCH valid bit is cleared, need to remove index in Pcache which system backmap replaced!!

13.7.11 FILL

The ARB machine stays in FILL until the fill_done signal is received from the FILL sequencer indicating the read is complete, or an error or IABORT is detected.

1. If not fill_done and not error and not IABORT, remain at 'FILL'.

```
err_flag         - enable error logic/input
hold_en          - enable hold
```

2. If fill_done and not error and not IABORT, return to 'IDLE'.

```
dispatch_flag    - enable dispatch_in next access
hold_en          - enable hold
all_chains_clr   - clear all in progress state
```

   The fill is complete, C_PIPE_%LAST_FILL is set by the FILL sequencer to tag the last quadword of data.

   If address<31:29> is '111 "Return_I/O_Data" is driven to the FILL sequencer. The INPUT_BUF quadword addressed by address<4:3> is driven to the ECC check latch. C_PIPE_%REQ_DQW and C_PIPE_%LAST_FILL are set to indicate selected and only return data.
3. If IABORT and not error, the next state is 'IDLE', the IREAD_LATCH valid bit is cleared. If 'FILL' from SYS_READ need to remove index in Pcache which system backmap replaced!!

4. If error, the next state is 'IDLE', and the error is logged.

13.7.12 SYS_RD

The 'SYS_RD' state is entered from

1. DISPATCH for DREAD no Bcache, DREAD_IO, IREAD no Bcache, or IREAD_IO, cReq_h<2:0> is READ_BLOCK.
2. DISPATCH FOR DREAD_LOCK, cReq_h<2:0> is LDxL.
3. 'RDC' for DREAD miss, cReq_h<2:0> is READ_BLOCK.

The cWMask lines are as

- cWMask[1:0} are address[4:3}
- cWMask[2] is '1 if not I/O space, Pcache allocate(EV D-stream)
- cWMask[3] indicates Pcache set being allocated, for systems which support a backmap for each set
- cWMask[4] indicates I-stream

The cReq_h lines become valid with the first sysClkOut1_h rising edge after the first cpu cycle of 'SYS_RD'. The 'SYS_RD' state repeats until cAck_h<2:0> returns error or OK.

1. If CACK_IDLE, remain at 'SYS_RD'.

```
err_flag         - enable error logic/input
sys_dp_ctrl_en   - data path control to fill sequencer
hold_en          - enable hold
```

2. If CACK_OK and not IABORT, the next state is 'FILL'.

```
err_flag         - enable error logic/input
rl_retire_en     - clear I or D read latch valid flag
```

3. If not CACK_IDLE and IABORT, the next state is 'IDLE', need to remove index in Pcache which system backmap replaced!!
4. If error, the next state is 'IDLE', and the error is logged.

13.7.12.1 Read Errors

- bad tagCtlP_h -> c%cbox_s_err; c%cbox_hard_err; (machine check)
- bad tagAdrP_h -> c%cbox_s_err; c%cbox_hard_err; (machine check)
- single bit ECC errors -> c%cbox_s_err
- double bit ECC -> c%cbox_s_err; c%cbox_hard_err; (machine check)
- cAck_h = SOFT_ERROR -> c%cbox_s_err
- cAck_h = HARD_ERROR -> c%cbox_s_err; c%cbox_hard_err; (machine check)

13.7.13 WR_STALL

When a non_byte_word WRITE with the Bcache enabled and not "PV" is dispatched the address, data and mask logic is set, and the entry is removed from the WRITE_QUEUE.

write_stall is not tagOK_l or hold request at phase 4 of last cpu cycle of ARB state.

If write_stall occurs before the non_byte_word write sequence (WR_PROBE/probe, WR_CMP/compare, WR/write) can be completed or during the DISPATCH of the non_byte_word WRITE, the ARB state machine loops in 'WR_STALL' till the write_stall deasserts

```
tagok_stall      - block fill done latch
hold_en          - enable hold
ce_en            - assert dataCEOE<3:0>
```

and then advances to 'WR_PROBE',

```
tag_probe_req    - start tag compare at end of next SYS_CLK cycle
tce_dis          - deassert tag chip enable at end of next SYS_CLK cycle
```

restarting the non_byte_word write sequence with address, data, and mask already at the pins from the DISPATCH.

13.7.14 WR_PROBE

If 'WR_PROBE' is entered from DISPATCH, the address from the Write Queue began driving at phase 3 of the second cpu cycle of the Dispatch Cycle.

The 'WR_PROBE' state is 2,3, or 4 cpu cycles in duration as programmed from cache_speed. At the phase 4 of the last cpu cycle of 'WR_PROBE'

- tagAdr_h<31:17>, tagAdrP_h, tagCtlV_h, tagCtlD_h, tagCtlS_h, and tagCtlP_h are latched
- tagCEOE is deasserted

The next state is

1. If not write_stall, the next state is 'WR_CMP',

   wr_arm_en causes the dataWE_h<3:0> signals are "readied" from LW_MASK<3:0> if address<4> = '0, and from LW_MASK<7:4> if address<4> = '1. tagCtlWE_h is "armed".
2. If write_stall, the next state is 'WR_STALL'.

13.7.15 WR_CMP

Write hit is determined, where write_hit equals

- tagAdr<31:22/17> matches adr_h<31:22/17>
- tagCtlV_h is true
- tagCtlS_h is false
- tagCtlP_h and tagAdrP_h are correct
- or force hit

The next state is

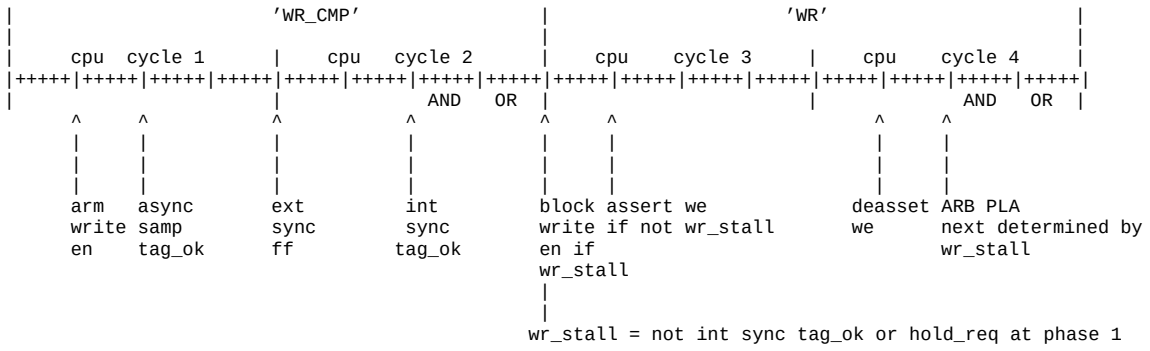
1. If write_hit and not write_stall and not tag_error, the next state is 'WR'.

2. If not write_hit and not write_stall and not tag_error, the next state is 'SYS_WR', and tagCtlWE and dataWE<3:0> are "disabled".

```
err_flag         - enable err_in (cack = hard error)
sys_dp_ctrl_en   - data path control to fill sequencer
creq_lat_en      - latch new CREQ
CREQ             - WRITE BLOCK
```

3. If write_stall, the next state is 'WR_STALL', and tagCtlWE and dataWE<3:0> are "disabled".
4. If not write_stall and tag_error (either tagCtlP_h and tagAdrP_h are not correct), tagCtlWE and dataWE<3:0> are "disabled",the error is logged, c%cbox_s_err is asserted, and the ARB state returns to 'IDLE'.

Figure 13–10: wr_stall timing



13.7.16 WR

data_h<127:0> and check<27:0> are driven onto the EDAL from the OUT_BUF. The tagCtl lines are driven as

- tagCtlD_h is DIRTY
- tagCtlV_h is not changed
- tagCtlS_h is not changed
- tagCtlP_h is toggled if tagCtl_h was previously CLEAN

If write_stall sampled at the previous phase 4 is true tagCtlWE and dataWE<3:0> are "disabled", and the write sequence is retried after the write_stall is completed.

If write_stall sampled at the previous phase 4 is not asserted, tagCtlWE and the selected dataWE<3:0> signals are driven from phase 2 of the first cpu cycle through phase 2 of the last cpu cycle of 'WR', the LW_MASK register is cleared.

1. If not write_stall, the write has completed successfully, the next state is 'IDLE'.

```
dispatch_flag    - enable dispatch_in next access
hold_en          - enable hold
all_chains_clr   - clear all in progress state
```

2. If write_stall, the write enable were blocked, the next state is 'WR_STALL'.

13.7.17 BWR_STALL

When a byte_word WRITE with the Bcache enabled and not "PV" is dispatched the address, data and mask logic is set, and the entry is removed from the WRITE_QUEUE.

write_stall is not tagOK_l or hold request at phase 4 of last cpu cycle of ARB state.

If write_stall occurs before the byte_word write sequence (BWR_PROBE/probe,BWR_CMP/compare, BWR_MERGE/merge, WR/write) can be completed or during the DISPATCH of the byte_word WRITE, the ARB state machine loops in 'BWR_STALL' till the write_stall deasserts

```
tagok_stall      - block fill done latch
hold_en          - enable hold
ce_en            - assert dataCEOE<3:0>
```

and then advances to 'BWR_PROBE',

```
err_flag         - enable error logic/input
FILL_BWM_DIR     - merge target QW from cache at end of next SYS_CLK cycle
in_data_lat_en   - latch cache input at end of next SYS_CLK cycle
tag_probe_req    - start tag compare at end of next SYS_CLK cycle
tce_dis          - deassert tag chip enable at end of next SYS_CLK cycle
```

restarting the byte_word write sequence with address, and mask already at the pins from the DISPATCH, and the WRITE_QUEUE already at the Merge register.

13.7.18 BWR_PROBE

The READ_BYTE/WORD address began driving at phase 3 of the second cpu cycle of the Dispatch Cycle. The 'READ_BYTE/WORD' state is 2,3, or 4 cpu cycles in duration as programmed from cache_speed. At the phase 4 of the last cpu cycle of 'READ_BYTE/WORD'

- tagAdr_h<31:17>, tagAdrP_h, tagCtlV_h, tagCtlD_h, tagCtlS_h, and tagCtlP_h are latched
- data_h<127:0> and check_h<27:0> are latched in the INPUT_BUF<dataA_h<4>>.
- tagCEOE is deasserted

The data from the WRITE_QUEUE is loaded into the MERGE register.

1. If not write_stall, the next state is 'BWR_CMP'.
2. If write_stall, the next state is 'BWR_STALL'.

13.7.19 BWR_CMP

The quadword of data from the INPUT_BUF pointed to address <4:3> is driven to the "ECC/MERGE" logic. ECC is checked, single bit errors are corrected.

- single bit ECC errors -> c%cbox_s_err
- double bit ECC on target quadword aborts "byte/word write"; -> c%cbox_h_err

The data is merged and loaded at the output drivers as in ARB state 'BWR_MERGE'. Write hit is determined. The next state is

1. If write_hit and not write_stall and not (tag_error or fill_error), the next state is 'BWR_MERGE'. wr_arm_en causes the dataWE_h<3:0> signals to be "armed" from LW_MASK<3:0> if address<4> = '0, and from LW_MASK<7:4> if address<4> = '1. wr_arm_en causes tagCtlWE_h to be "armed". If a single bit ECC error is corrected for the read data the error is logged and c%cbox_s_err is set.
2. If not write_hit and not write_stall and not tag_error, the next state is 'BWR_SYS_RD'. cReq_h<2:0> is driven with LDxL.

```
err_flag         - enable err_in (cack = hard error)
FILL_BWM_DIR     - merge target QW from cache at end of next SYS_CLK cycle
sys_dp_ctrl_en   - data path control to fill sequencer
creq_lat_en      - latch new CREQ
CREQ             - LDxL
```

3. If write_stall, the next state is 'BWR_STALL'.
4. If not write_stall and tag_error (either tagCtlP_h and tagAdrP_h are not correct), the error is logged, c%cbox_s_err is asserted, and the ARB state returns to 'IDLE'.
5. If not write_stall and fill_error (uncorrectable ECC), the error is logged, c%cbox_h_err is asserted, and the ARB state returns to 'IDLE'.

13.7.20 BWR_MERGE

The data is merged and loaded at the output drivers.

```
if BM<0>= '1 data<07:00> = Write_Queue<07:00>; if BM<0>= '0 data<07:00> = MERGE_register<07:00>
if BM<1>= '1 data<15:08> = Write_Queue<15:08>; if BM<0>= '0 data<15:08> = MERGE_register<15:08>
if BM<2>= '1 data<23:16> = Write_Queue<23:16>; if BM<0>= '0 data<23:16> = MERGE_register<23:16>
if BM<3>= '1 data<31:24> = Write_Queue<31:24>; if BM<0>= '0 data<31:24> = MERGE_register<31:24>
if BM<4>= '1 data<39:32> = Write_Queue<39:32>; if BM<0>= '0 data<39:32> = MERGE_register<39:32>
if BM<5>= '1 data<47:40> = Write_Queue<47:40>; if BM<0>= '0 data<47:40> = MERGE_register<47:40>
if BM<6>= '1 data<55:48> = Write_Queue<55:48>; if BM<0>= '0 data<55:48> = MERGE_register<55:48>
if BM<7>= '1 data<63:56> = Write_Queue<63:56>; if BM<0>= '0 data<63:56> = MERGE_register<63:56>
```

ECC check bits are generated for data<63:0> which is loaded into the OUT_BUF.

1. If fill_done and not write_stall, the next state is 'BWR_WR'.
2. If not fill_done and not write_stall, the state remains 'BWR_MERGE', dataWE_h<3:0> and tagCtlWE_h are "RE-armed".
3. If write_stall, the next state is 'BWR_STALL'.

13.7.21 BWR

data_h<127:0> and check<27:0> are driven onto the EDAL from the OUT_BUF. The tagCtl lines are driven as

- tagCtlD_h is DIRTY
- tagCtlV_h is not changed
- tagCtlS_h is not changed
- tagCtlP_h is toggled if tagCtl_h was previously CLEAN

If write_stall sampled at the previous phase 4 is true tagCtlWE and dataWE<3:0> are "disabled", and the byte_word write sequence is retried after the write_stall is completed.

If write_stall sampled at the previous phase 4 is not asserted, tagCtlWE and the selected dataWE<3:0> signals are driven from phase 2 of the first cpu cycle through phase 2 of the last cpu cycle of 'BWR', the LW_MASK register is cleared.

1. If not write_stall, the write has completed successfully, the next state is 'IDLE'.
2. If write_stall, the write enable were blocked, the next state is 'BWR_STALL'.

13.7.22 BWR_SYS_RD

The ARB state remains 'BWR_SYS_RD' until the system completes the LDxL command.

1. If CACK = idle, wait in 'BWR_SYS_RD'.

```
err_flag         - enable error logic/input
sys_dp_ctrl_en   - data path control to fill sequencer
hold_en          - enable hold
```

2. If CACK = OK or soft errr the next state is 'BWR_SYS_MERGE', and err_flag is enabled for the ECC check. If soft error the error is logged, c%cbox_s_err is asserted.
3. If CACK = hard error, the next state is 'IDLE', the error is logged in BIU_STAT and BIU_ADDR, the c%cbox_h_err is asserted and the "byte/word write" sequence is aborted.

13.7.23 BWR_SYS_MERGE

The quadword of data from the INPUT_BUF pointed to address <4:3> is driven to the "ECC/MERGE" logic. ECC is checked, single bit errors are corrected.

- single bit ECC errors -> c%cbox_s_err
- double bit ECC on target quadword aborts "byte/word write"; -> c%cbox_h_err

The data is merged and loaded at the output drivers as in ARB state 'BWR_MERGE'. ECC check bits are generated for data<63:0> which is loaded into the OUT_BUF.

1. If not fill_done and not hard_error, the state remains 'BWR_SYS_MERGE', keep err_flag enabled for ECC check.
2. If fill_done and not hard_error, the next state is 'SYS_WR'. If a single bit ECC error is corrected for the read data the error is logged and c%cbox_s_err is set. cReq_h<2:0> is driven with STxC, and cWMask<7:0> is driven from LW_MASK<7:0>. LW_MASK is set from BM<7:0> and address<3:0> as in the 'PACK_WRITE' state. Bits of LW_MASK<7:0> previously set in the 'PACK_WRITE' state remain set. The address buffer is not loaded and remains the same.

```
err_flag         - enable error logic/input
sys_dp_ctrl_en   - data path control to fill sequencer
creq_lat_en      - latch new CREQ
CREQ             - STxC
```

3. If hard_error,the next state is 'IDLE', the error is logged in BIU_STAT and BIU_ADDR, the c%cbox_h_err is asserted and the "byte/word write" sequence is aborted.

13.7.24 SYS_WR

At the first SYS_CLK rising edge on entry to 'SYS_WR' cReq_h<2:0> is driven with

- WRITE_BLOCK if entered from DISPATCH or 'WR_CMP'
- STxC if entered from 'BWR_SYS_MERGE'.

Also at SYS_CLK, cWMask<7:0> is driven from

- LW_MASK<7:0> if not "PV"
- WRITE_QUEUE BM<7:0> if "PV"

If the write is for a "PV" system

- Addr<3> indicates which QW in the OUT_BUF is to be written from the byte mask driven to cWMask<7:0>
- dataWE_h<0> = X0 <- '1 if LW_MASK 0,2,4,6 was set previously at 'PACK_WRITE'
- dataWE_h<1> = X1 <- '1 if LW_MASK 1,3,5,7 was set previously at 'PACK_WRITE'

1. If CACK = idle and not error, wait in 'SYS_WR'.

```
err_flag         - enable error logic/input
sys_dp_ctrl_en   - data path control to fill sequencer
hold_en          - enable hold
```

2. If CACK = OK, or STxC_FAIL and not bwr_chain, the next state is 'IDLE'.

```
dispatch_flag    - enable dispatch_in next access
hold_en          - enable hold
all_chains_clr   - clear all in progress state
```

   If CACK = STxC_FAIL and not bwr_chain, set bit of STxC_RESULT register to indicate write_unlock failure to microcode.

3. If CACK = STxC_FAIL and bwr_chain, the next state is 'BWR_SYS_RD', retry RMW with LDxL.

```
err_flag         - enable err_in (cack = hard error)
FILL_BWM_DIR     - merge target QW from cache at end of next SYS_CLK cycle
sys_dp_ctrl_en   - data path control to fill sequencer
creq_lat_en      - latch new CREQ
CREQ             - LDxL
```

4. If error (CACK not idle, OK, or STxC_FAIL), the next state is 'ERR'. If CACK = soft error, the error is logged, c%cbox_s_err is asserted. If CACK = hard error, the error is logged, c%cbox_h_err is asserted.

13.8 CBOX Error Handling Summary

A summary of the NVAX Plus CBOX error logic is shown in Table 13–17.

Table 13–17: NVAX Plus CBOX Error Handling

Problem	Situation	ERR_CTL	ARB/IPR_CTL	FILL
Tag Parity Error or Tag Control Parity Error	Any READ, DREAD, DREAD_LOCK, IREAD	Assert C%CBOX_S_ERR, Command ARB to go to ERRROR state, Generate C%CBOX_HARD_ERR when ARB send I_CF or D_CF	Send I_CF or D_CF to MBOX and abort. Latch appropriate BIU_STAT bits	Aborts due to MISS
	Any Write, WRITE_UNLOCK, WRITE	Assert C%CBOX_H_ERR, Command ARB to Abort	ARB Aborts. Latch appropriate BIU_STAT bits	Aborts on a BYTE/WORD WRITE, not involved yet otherwise.
Correctable ECC error	Any Read, including I/O read	Assert C%CBOX_S_ERR	Latch appropriate BIU_STAT bits. Wait for Fill to complete.	Assert C%CBOX_ECC_ERR, send corrected data to MBOX.
	BYTE/WORD WRITE, WRITE_UNLOCK, WRITE	Assert C%CBOX_S_ERR	Latch appropriate BIU_STAT bits. Wait for Fill to complete the MERGE.	Continue the MERGE with corrected data.
Uncorrectable ECC error or Parity Error	Any Read, including I/O read	Assert C%CBOX_S_ERR	Latch appropriate BIU_STAT bits. Wait for Fill to complete.	Assert C%CBOX_ECC_ERR, send C%CBOX_HARD_ERR along with I_CF or D_CF.
	BYTE/WORD WRITE, WRITE_UNLOCK, WRITE	Assert C%CBOX_H_ERR	Latch appropriate BIU_STAT bits. Wait for Fill to signal complete.	Abort Merge, restart ARB.
cAck Hard Error	Any READ, DREAD, DREAD_IO, DREAD_LOCK, IREAD, IREAD_IO	Assert C%CBOX_S_ERR, Command ARB to go to ERRROR state, Generate C%CBOX_HARD_ERR when ARB send I_CF or D_CF	Send I_CF or D_CF to MBOX and abort. Latch appropriate BIU_STAT bits	Aborts due to cAck hard error.
	Any Write, WRITE_UNLOCK, WRITE, IO_WR_UNLOCK	Command ARB to Abort, Assert C%CBOX_H_ERR	Latch appropriate BIU_STAT bits. ARB aborts.	Aborts due to cAck hard error.
cAck Soft Error	Any READ, including I/O read	Assert C%CBOX_S_ERR	Latch appropriate BIU_STAT bits. Wait for Fill to complete.	Complete the FILL.

Table 13–17 (Cont.): NVAX Plus CBOX Error Handling

Problem	Situation	ERR_CTL	ARB/IPR_CTL	FILL
	Any WRITE, WRITE_ UNLOCK, WRITE, IO_ WR_UNLOCK	Assert C%CBOX_S_ERR	Latch appropriate BIU_ STAT bits. Wait for Fill to complete the MERGE.	Continue the MERGE with corrected data.

13.9 Invalidates

The external system logic is responsible for keeping the primary cache coherent. If the Pcache is being allocatted as two way associative NVAX Plus asserts pMapWE_h<0> when filling Pcache set 0 and pMapWE_h<1> when filling Pcache set 1 to support systems with backmaps. If the Pcache is being allocatted as direct mapped NVAX Plus asserts pMapWE_h<0> when filling Pcache.

For two way associative operation pInvReq<0> indicates an entry in Pcache set 0 is to be invalidated, while pInvReq<1> indicates an entry in Pcache set 1 is to be invalidated, where iAdr<11:5> determines the index to be invalidated.

In direct map mode pInvReq<0> and iAdr<12:5> indicate the entry to be invalidated. If iAdr<12> = '0 set 0 is invalidated at index = iAdr<11:5>, and if iAdr<12> = '1 set 1 is invalidated at index = iAdr<11:5>.

Systems using two way associative allocation which do not backmap the Pcache issue invalidates to both sets of the Pcache when a block is displaced from the Bcache. The index to be invalidated is driven to iAdr<11:5> and pInvReq<1:0> are both asserted. The MBOX modification for NVAX Plus allows invalidates the address in CM_OUT_LATCH<12:5>, for set a single Pcache set as specified by CM_OUT_LATCH[InvReq]. The CBOX sequences invalidates to set 0 in the first cpu_clk cycle of a system cycle, and to set 1 in the second cpu_clk cycle of a system cycle.

The CBOX sources an invalidate when an IABORT is received and the ARB sequencer has already issued a pMapWE or read to the system which updates the Pcache backmap. Since the present entry in the Pcache may not be removed if an IABORT is detected in ARB states 'RDC', 'RDN', 'SYS_RD', or 'FILL' it is necessary to invalidate the index which was to be allocated, since the backmap no longer contains this address.

Systems which do not backmap that allocate the Pcache as two-way associative and therefore assert both pInnvReq<1:0> can not request invalidates in consecutive sys_clk cycles.

13.10 Revision History

Table 13–18: Revision History

Who	When	Description of change
Gil Wolrich	15-Nov-1990	NVAX PLUS release for external review.
Gil Wolrich	30-Jan-1991	remove vectors features.

Chapter 14

Error Handling

This chapter describes the NVAX Plus error exceptions and interrupts as seen from the macrocoder's point of view. It is organized with respect to the SCB vectors through which the event is dispatched. The SCB layout and SCB vector format are described in the Architecture Summary chapter of the NVAX Plus chip specification.

14.1 Terminology

Term	Meaning
Fill	Any quadword of data returned to the NVAX Plus chip in response to read-type operation. The quadword containing the requested data is a fill.
Dirty	In the Bcache, a bit is stored with each hexaword called the dirty bit. When set this bit indicates that memory does not have the updated data for this block.

14.2 Error handling Introduction and Summary

This chapter discusses all levels of hardware and microcode-detected errors. Errors notification occurs through one of the following events, listed in order of decreasing severity.

- Console error halt—A halt to console mode is caused by one of several errors such as Interrupt Stack Not Valid. For certain halt conditions, the console prompts for a command and waits for operator input. For other halt conditions, the console may attempt a system restart or a system bootstrap as defined by DEC Standard 032. The actual algorithms used are outside of the scope of this document.
- Machine check—A hardware error occurred synchronously with respect to the execution of instructions. Instruction-level recovery and retry may be possible.
- Hard error interrupt—A hardware error occurred asynchronously with respect to the execution of instructions. Usually, data is lost or state is corrupted, and instruction-level recovery may not be possible.
- Soft error interrupt—A hardware error occurred asynchronously with respect to the execution of instructions. The error is not fatal to the execution of instructions, and instruction-level recovery is usually possible.

- Kernel stack not valid—During exception processing, a memory management exception occurred while trying to push information on the kernel stack.

This chapter explains in detail several of the SCB entry points. The purpose is to help the operating system programmer determine exactly what error occurred and to recommend an error recovery method.

The following information is given in this chapter for each SCB entry point:

- What parameters are pushed on the stack.
- What failure codes are defined.
- What additional information exists and should be collected for analysis.
- How to determine what error(s) actually occurred.
- How to restore the state of the machine, and what level of recovery is possible.

Table 14–1 shows the general error categories associated with each of these error notifications.

Table 14–1: Error Summary By Notification Entry Point

Entry Point	SCB Index (hex)	General Error Categories
Console Halt	N/A	Interrupt Stack not valid, kernel-mode halt, double error, illegal SCB vector
Machine Check	04	Memory management, interrupt, microcode detected CPU errors, CPU stall timeout, TB parity errors, VIC tag or data parity errors, Bcache uncorrectable data read errors, CACK_HERR on read
Soft Error Interrupt	54	VIC tag or data parity errors, Pcache tag or data parity errors, Bcache tag parity error on read, Bcache uncorrectable data read errors Bcache correctable data errors
Hard Error Interrupt	60	Bcache uncorrectable data errors on write operations, Bcache tag parity error on writes, CACK_HERR on writes

14.3 Error Handling and Recovery

All errors (except those resulting in console halt) go through SCB vector entry points and are handled by service routines provided by the operating system. A console halt transfers control to the address of the CONSOLE_HALT register. Software driven recovery or retry is not recommended for errors resulting in console halt.

Software error handling (by operating system routines) can be logically divided into the following steps:

- State collection.
- Analysis.

- Recovery.
- Retry.

These steps are discussed in general in the next four sections. After that, details are supplied on analysis, recovery and retry for each error event which results in an exception or interrupt. This information is organized by SCB entry point.

14.3.1 Error State Collection

Before error analysis can begin, all relevant state must be collected. The stack frame provides the PC/PSL pair for all exceptions and interrupts. For machine checks, the stack frame also provides details about the error.

In addition to the stack frame, machine checks and hard and soft error interrupts usually require analysis of other registers. It is strongly recommended that all the state listed below be read and saved in these cases. State is saved prior to analysis so that analysis is not complicated by changes in state in the registers as the analysis progresses, and so that errors incurred during analysis and recovery can be processed with that context.

Ibox
ICSR: Ibox (VIC) control and status register.
VMAR: VIC memory address register.

Ebox
ECR: Ebox control and status register.

Mbox
TBSTS: TB status register.
TBADR: TB address register.
PCSTS: Pcache status register.
PCADR: Pcache address register.

Cbox
BIU_STAT: Bus or Fill error status.
BC_TAG: Contains tag of tag_parity, control_parity, or fill error.
BIU_ADDR: Address associated with cache probe or bus error. (BIU_HERR, BIU_SERR, BC_TPERR, BC_TCPERR)
FILL_ADDR: Address associated with fill error, FILL_ECC or FILL_DPERR. FILL_SYNDROME: Syndrome bits associated with FILL_ADDR.

NOTE

The ERROR interrupt is level sensitive requiring the clearing of the external ERR_H signal if the interrupt source is external to NVAX Plus, and the clearing of the BIU_STAT indication resulting in the internal H_ERR signal to clear the interrupt. The error bits in the BIU_STAT register are W1C, and therfore should be cleared after BIU_STAT is read, so that errors incurred during analysis and recovery can be processed with that context.

For the purposes of the rest of this chapter, it is assumed that each of these states is saved in a variable whose name is constructed by prepending "S_" to the register name. For example, the ICSR would be saved in the variable S_ICSR.

The following example shows allocation of memory storage for the error state.

```
; ERROR STATE COLLECTION DATA STORAGE

                                ;IBOX
S_ICSR:          .LONG   0      ; IBOX VIC CONTROL AND STATUS REGISTER
S_VMAR:          .LONG   0      ; IBOX VIC ERROR ADDRESS REGISTER

                               ;EBOX
S_ECR:           .LONG   0      ; EBOX CONTROL AND STATUS REGISTER

                                ;MBOX
S_TBSTS:         .LONG   0      ; TB STATUS REGISTER
S_TBADR:         .LONG   0      ; TB ERROR ADDRESS REGISTER
S_PCSTS:         .LONG   0      ; PCACHE STATUS REGISTER
S_PCADR:         .LONG   0      ; PCACHE ERROR ADDRESS REGISTER

                               ;CBOX
S_BIU_STAT:  .LONG 0 ; Bus or Fill error status
S_BC_TAG:  .LONG 0 ; Contains tag of tag_parity, control_parity, or fill error
S_BIU_ADDR:  .LONG 0 ; Address associated with BIU_HERR, BIU_SERR, BC_TPERR, BC_TCPERR
S_FILL_ADDR:  .LONG 0 ; Address associated with fill error, FILL_ECC or FILL_DPERR
S_FILL_SYNDROME: .LONG 0 ; Syndrome bits associated with FILL_ADDR
```

The following example shows collection of error state which would normally be done early in the error handling routine. If a second bus or fill error is detected the SEO second error bit is set, but the error address and status are lost.

```
                                                    ;SAVE ALL ERROR STATE UPON ENTRY TO ERROR HANDLING ROUTINE
SAVE_STATE:
                                                    ;CBOX
                MFPR    #PR19$_BIU_STAT,S_BIU_STAT
                MFPR    #PR19$_BIU_ADDR,S_BIU_ADDR
                MFPR    #PR19$_FILL_ADDR,S_FILL_ADDR
                MFPR    #PR19$_FILL_SYNDROME,S_FILL_SYNDROME
                MFPR    #PR19$_BC_TAG,S_BC_TAG

                                                    ;IBOX
                MFPR    #PR19$_ICSR,S_ICSR
                MFPR    #PR19$_VMAR,S_VMAR

                                                    ;EBOX
                MFPR    #PR19$_ECR,S_ECR

                                                    ;MBOX
                MFPR    #PR19$_TBSTS,S_TBSTS
                MFPR    #PR19$_TBADR,S_TBADR
                MFPR    #PR19$_PCSTS,S_PCSTS
                MFPR    #PR19$_PCADR,S_PCADR


                                                    ;SYSTEM ENVIRONMENT
            COLLECTION OF SYSTEM ENVIRONMENT ERROR REGISTERS GOES HERE
```

Additional state collection is recommended while/after flushing the Bcache because certain errors may occur as a result of the flush operation.

For the purposes of the rest of this chapter, it is assumed that each of these states is saved in a variable whose name is constructed by prepending "SS_" to the register name. For example, the BIU_STAT register would be saved in the variable SS_BIU_STAT.

14.3.2 Error Analysis

With the error state obtained during the collection process, the error condition can be analyzed. The purpose is to determine what error event caused the particular notification being handled (to the extent possible), and what other errors may also have occurred. Analysis of machine checks and hard and soft error interrupts should be guided by the parse trees given in the appropriate sections below.

NOTE

Errors detected in or by one of the caches usually result in the cache automatically being disabled. However, to minimize the possibility of nested errors, it is suggested that error analysis and recovery for memory or cache-related errors be performed with the Pcache and Bcache disabled.

In some cases, a notification for a single error occurs in two ways. For example, an uncorrectable error in the Bcache data RAMs will cause a soft error interrupt and may also cause a machine check. **Software should handle cases where a machine check handler clears error bits and then the soft error handler is entered with no error bits set.**

In general an error reporting register can report events which lead to machine check, soft error, or hard error. A given error event can result in machine check and soft error interrupt, or in just one or the other. Events which lead to hard error interrupts generally can not also cause machine check or soft error interrupt. However, if a hard error occurs from a write operation, a subsequent read error can result in a machine check with a SEO bit set.

Multiple simultaneous errors may make useful recovery impossible. However, in cases where no conflict exists in the reporting of the multiple errors (i.e., separate Pcache and Bcache errors), and recovery from each error is possible, then recovery from the set of errors is accomplished by recovering from both of them. For example, recovery from a Pcache tag parity error and FILL correctable data error being reported together is possible by following the recovery procedures for each error in sequence.

The error cause determination parse tree for machine check exception is directed at causes or possible causes of machine checks. It ignores errors which lead to hard or soft error interrupts but not to machine checks. Similarly, the hard error interrupt cause determination ignores errors which lead to machine check or soft error interrupt, and the soft error interrupt cause determination ignores errors which lead to machine check or hard error interrupt.

There is a natural order between machine check, hard error interrupt, and soft error interrupt because the IPL for hard error interrupts is higher than that of soft error interrupts and the IPL in the machine check exception is higher than either of the error interrupts. This hierarchy is important because knowledge of which notification event occurred is used to discriminate between certain error events (e.g., an error on the initial fill quadword for a read-lock is distinguished from a fill error on a subsequent quadword by the fact of machine check notification).

14.3.3 Error Recovery

Recovery from errors consists of clearing any latched error state, repairing damaged state (if necessary and possible), and restoring the system to normal operation. There are special considerations involved in analysis and recovery from cache or memory errors, which are covered in the next sections.

Recovery from multiple error scenarios is possible when there is no conflict in the error registers which report the errors and there is no conflict in the recovery procedures for the errors. However all recovery procedures in this chapter assume that only one error is present. None of the procedures are valid in multiple error scenarios without further analysis.

In some instances, it may be desirable to stop using the hardware which is the source of a large number of errors. For example, if a cache reports a large number of errors, it may be better to disable it. It is suggested that software maintain error counts which should be compared against error thresholds on every error report. If the count (per unit time) exceeds the threshold, the hardware should be disabled.

14.3.3.1 Special Considerations for Cache and Memory Errors

Cache and memory error recovery requires special considerations:

- Cache and memory error recovery should always be done with the Pcache and VIC off.
- Bcache flush should be always be done one block at a time, recapturing the relevant error registers between each block flush.
- Cache coherence requires a specific procedure for re-enabling the caches. See Section 14.3.3.1.1, Cache Coherence in Error Handling.
- Error recovery should be performed starting with the most distant component and working toward the CPU and Ebox. System environment memory errors should be processed first, Bcache tag store and data RAM errors, Pcache errors, TB errors, and, finally, VIC errors.
- Bcache errors are cleared by writing the write-one-to-clear bits in BIU_STAT. See <REFERENCE>(err_spec_wb_recovery\ FULL).
- Pcache tag and data store errors are cleared by writing the write-one-to-clear bits in PCSTS. The suggested way to do this is to write a one to the specific error bit. Pcache flush is necessary after Pcache tag store parity errors. See Section 14.3.3.1.1.1, Cache Enable, Disable, and Flush Procedures.
- TB errors are cleared by writing the write-one-to-clear bits in TBSTS. The suggested way to do this is to write a one to the specific error bit.
- PTE read errors are cleared by writing the PTE error write-one-to-clear bits in PCSTS. The suggested way to do this is to write a one to the specific error bit.
- VIC errors are cleared by writing the write-one-to-clear bits in ICSR. The suggested way to do this is to write a one to the specific error bit. VIC flush and re-enable is necessary after VIC tag store parity errors. See Section 14.3.3.1.1.1, Cache Enable, Disable, and Flush Procedures.

14.3.3.1.1 Cache Coherence in Error Handling

Certain procedures must be followed in order to maintain cache coherence while enabling NVAX caches. Since many errors cause caches to be disabled, and since cache and memory error recovery is normally done with the Pcache and VIC off, the complete cache enable procedure is done as part of recovery from all cache and memory errors.

The VIC (virtual instruction cache) is not automatically kept coherent with memory. It is flushed as a side effect of the REI instruction (as required by the VAX architecture). Normally in error recovery, there is no definite need to flush the VIC. For consistency and for the sake of beginning error retry in a known state, flushing the VIC during error recovery is recommended. However,

in the event of VIC tag parity errors, the complete VIC flush procedure described in the next section must be done.

The TB is not automatically kept coherent with memory. Software uses the TBIS and TBIA functions to maintain coherence, and the LDPCTX instruction clears the process PTEs in the TB. Normally in error recovery, there is no definite need to flush the TB. For consistency and for the sake of beginning error retry in a known state, flushing the TB during error recovery is recommended. When a TB parity error occurs, Mbox hardware flushes the TB by itself (via an internally generated TBIA), but it would be appropriate for software to test the TB after a parity error. This is discussed in Section 14.3.3.1.2.

14.3.3.1.1.1 Cache Enable, Disable, and Flush Procedures

To enable the NVAX Plus caches, the caches are flushed and enabled in a specific order. The ordering is necessary for coherence between the Bcache, Pcache, and memory. For simplicity, one procedure is given for enabling the NVAX Plus caches, even though variations on the procedure may also produce correct results. Disabling the caches can be done in any order, though one procedure is given here.

In error handling, the VIC and Pcache are disabled.

14.3.3.1.1.1.1 Disabling the NVAX Plus Caches for Error Handling

This is the procedure for disabling the NVAX Plus caches:

NOTE

These procedures will be supplied with MACRO coding examples.

- Disable the VIC:

```
TBS (MTPR to ICSR)
```

- Disable the Pcache:

```
TBS (MTPR to PCCTL)
```

- Disable the Bcache:

```
TBS (MTPR to BIU_CTL)
```

14.3.3.1.1.1.2 Enabling the NVAX Caches

The procedure for enabling the NVAX caches after an error is the same as is used to initialize the caches after power-up. This procedure ensures that error retry/restart occurs with the caches in a known state. The procedure is outlined below.

- The caches must all be disabled and the Bcache must be disabled.
- Flush the Bcache
- Enable the Bcache (MTPR to BIU_CTL).
- Flush the Pcache (Loop on MTPR to PCTAG IPRs).
- Enable the Pcache (MTPR to PCCTL).

- Flush the TB:

```
MTPR #0, #PR19$_TBIA
```

- Flush the VIC (Loop on MTPRs to VMAR and VTAG, writing different initial values into the left and right banks).
- Enable the VIC (MTPR to ICSR).

14.3.3.1.1.2 Extracting Data from the Bcache

To extract data from the Bcache, the Bcache is placed in FORCE_HIT mode.

After the Bcache is flushed, set the Bcache in FORCE_HIT mode and extract the data. Note that the code which executes this procedure and its local data must be in IO space. The TB entries (PTEs) which map this code and local data must be fixed in the TB. (This is most easily done by flushing the TB via an MTPR to TBIA and then accessing all the relevant pages in pages in sequence.) Otherwise Bcache FORCE_HIT will interfere with instruction fetch, operand access, and PTE fetches in TB miss sequences.

The following instruction places the Bcache in FORCE_HIT mode:

```
TBS (MTPR to BIU_CTL)
```

With the Bcache in FORCE_HIT mode, a read in memory space of any address whose index portion matches the index of the cache data will return the data (provided there is no uncorrectable data RAM error). This is most easily accomplished by reading from the true address of the data.

NOTE

In FORCE_HIT mode, Bcache data RAM ECC errors are detected. **(unless a BIU_CTL<DISABLE_ERRORS> function similar to NVAX is added)** Software should prepare for an ECC error (BIU_STAT <FILL_ECC>).

14.3.3.1.2 Cache and TB Test Procedures

TBS

OUTLINE OF TO-BE-SPECIFIED TEST PROCEDURES

Testing is generally done using the force hit mode of a cache. The code and data of the test procedure must reside in IO space. Assuming memory management is enabled during this procedure, the needed PTEs must be in the TB before entering force hit mode in the Pcache or Bcache. For the Bcache, testing should be done with errors disabled. **(BIU_CTL<DISABLE_ERRORS> function similar to NVAX is added)** The ECC logic should be tested thoroughly on one location by forcing various check bit patterns and examining the syndrome latched on the read (**FILL_SYNDROME** is loaded on every read in Bcache disable-errors mode). Pcache and VIC parity checking should be tested by writing bad parity into the arrays. TB testing may be accomplished by writing to MTBTAG and MTBPTE (with care to not change any TB entry necessary for the test code and data and not to cause two TB entries to exist for one address). PROBER and PROBEW (setting PSL<PRV_MOD>) are then used to verify the protection bits. Testing the modify bit would be difficult, though approaches exist.

14.3.4 Error Retry

Error retry is a function of the error notification (machine check or error interrupt), error type, and error state. The sections below specify the conditions under which the instruction stream may be restarted.

If retry is to be attempted, the stack must be trimmed of all parameters except the PC/PSL pair. This is necessary only for machine checks, because error interrupts do not provide any additional parameters on the stack. An REI will then restart the instruction stream and retry the error. Some form of software loop control should be provided to limit the possibility of an error loop. Note that pending error interrupts may be taken before the retry occurs, depending on the IPL of the interrupted or machine checked code.

Strictly speaking, an REI from a hard or soft error interrupt handler is not a retry since these interrupts are recognized between macroinstructions. A machine check exception is an instruction abort, and an REI from the handler will cause the failing instruction to be retried (provided retry is indicated by analysis). What these cases all have in common is that the interrupted instruction stream is restarted. This is only done when the result of error analysis and recovery is such that all damaged state has been repaired and there is no reason to suspect that incorrect results will be produced if the image is restarted and another error does not occur.

If complete recovery from one or more errors is not possible (i.e., some state is lost or it is impossible to determine what state is lost), possibly the entire system will have to be crashed, a single process will have to be deleted, or some other action will have to be taken. Software must determine if the error is fatal to the current process, to the processor, or to the entire system, and take the appropriate action.

It is expected that software handles machine checks, soft error interrupts, and hard error interrupts independently. For example, after handling a machine check from which retry is to occur, software does not check for errors which might cause a pending hard or soft error interrupt. The machine check handler is exited via REI (after trimming the machine check information off the stack). If the IPL of the machine checked instruction stream is low enough, any pending hard or soft error interrupt is taken before the retry occurs. However, if the interrupted instruction stream was running at high IPL, then it will continue oblivious of remaining errors.

14.3.4.1 General Multiple Error Handling Philosophy

Multiple errors may be reported at the same time. In some cases the NVAX Plus pipeline will contain multiple operand prefetches to the same memory block. This can cause multiple errors from a single non-transient failure. It could also occur that two separate errors occur at nearly the same time and are thus reported simultaneously.

Multiple error scenarios may be grouped into the following three classes:

1. Multiple distinct errors for which no error report interferes with the analysis of any other (e.g., no lost error bits set).
2. Multiple errors which could have been caused by the NVAX Plus pipeline issuing more than one reference to a given block before the error interrupt or machine check forced a pipeline flush.
3. Multiple errors for which analysis is complicated because the reports interfere with each other.

It is the intent of this chapter to recover from class 1 (above) by simply treating the errors as separate and recovering from each in turn. Retry or restart evaluation is based on the cumulative result of the recovery and repair procedures for each error.

For class 2, specific cases are identified in which lost errors are tolerated. These cases are selected because the NVAX Plus pipeline can easily cause them (given one error), and because sufficient safeguards exist to ensure that correct operation is maintained. <REFERENCE>(err_retry_spec_case) lists these cases.

Class 3 scenarios are generally not considered recoverable. The system is simply crashed in those cases.

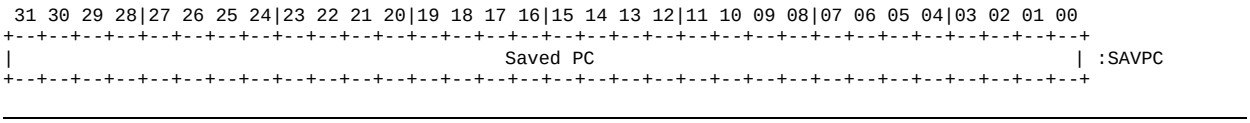
Note that if BIU_STAT<LOST_WRITE_ERR> is clear and BIU_STAT<FILL_SEO> is set with ARB_CMD being a read, then write data has not been lost, the system can be retried after the cache is flushed.

14.4 Console Halt and Halt Interrupt

A console halt is not an exception, but rather a transfer of control by the NVAX Plus microcode directly into console macrocode at the the address of the Console_Halt IPR. Console halts are initiated at powerup, by certain microcode-detected double error conditions, and by the assertion of the external halt interrupt pin, HALT_H.

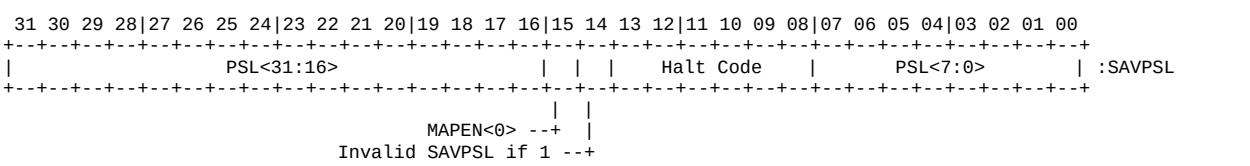
There is no exception stack frame associated with a console halt. Instead, the SAVPC and SAVPSL processor registers provide the necessary information. The format of SAVPC (IPR 42) is shown in Figure 14–1.

Figure 14–1: Console Saved PC



The PSL, halt code, MAPEN<0>, and a validity bit are saved in SAVPSL (IPR 43). The format of SAVPSL is shown in Figure 14–2. The halt codes are shown in Table 14–2.

Figure 14–2: Console Saved PSL



The possible halt codes that may appear in SAVPSL<13:8> are listed in Table 14–2.

Table 14–2: Console Halt Codes

Mnemonic	Code (Hex)	Meaning
ERR_HLTPIN	02	HALT_H pin asserted
ERR_PWRUP	03	Initial power up
ERR_INTSTK	04	Interrupt stack not valid
ERR_DOUBLE	05	Machine check during exception processing
ERR_HLTINS	06	HALT instruction in kernel mode
ERR_ILLVEC	07	Illegal SCB vector (bits <1:0> = 11)
ERR_WCSVEC	08	WCS SCB vector (bits <1:0> = 10)
ERR_CHMFI	0A	CHMx on interrupt stack
ERR_IE0	10	ACV/TNV during machine check processing
ERR_IE1	11	ACV/TNV during kernel-stack-not-valid processing

Table 14–2 (Cont.): Console Halt Codes

Mnemonic	Code (Hex)	Meaning
ERR_IE2	12	machine check during machine check processing
ERR_IE3	13	machine check during kernel-stack-not-valid processing
ERR_IE_PSL_26_24_101	19	PSL<26:24> = 101 during interrupt or exception
ERR_IE_PSL_26_24_110	1A	PSL<26:24> = 110 during interrupt or exception
ERR_IE_PSL_26_24_111	1B	PSL<26:24> = 111 during interrupt or exception
ERR_REI_PSL_26_24_101	1D	PSL<26:24> = 101 during REI
ERR_REI_PSL_26_24_110	1E	PSL<26:24> = 110 during REI
ERR_REI_PSL_26_24_111	1F	PSL<26:24> = 111 during REI
ERR_SELFTEST_FAILED	3F	Microcoded powerup selftest failed

At the time of the halt, the current stack pointer is saved in the appropriate IPR (0 to 4), and SAVPSL<31:16,7:0> are loaded from PSL<31:16,7:0>. SAVPSL<15> is set to MAPEN<0>. SAVPSL<14> is set to 0 if the PSL is valid and to 1 if it is not (SAVPSL<14> is undefined after a halt due to a system reset). SAVPSL<13:8> is set to the console halt code.

To complete the hardware restart sequence and thereby pass control to the console macrocode, the state shown in Table 14–3 is initialized.

Table 14–3: CPU State Initialized on Console Halt

State	Initialized Value
SP	IPR 4 (IS)
PSL	041F0000 (hex)
PC	from CONSOLE_HALT IPR
MAPEN	0
ICCS	0 (after reset, code=3, only)
SISR	0 (after reset, code=3, only)
ASTLVL	4 (after reset, code=3, only)
PAMODE	0 (after reset, code=3, only)
BPCR<31:16>	FECA(hex) (after reset, code=3, only)
CPUID	0 (after reset, code=3, only)
all else	undefined

14.5 Machine Checks

The machine check exception indicates a serious system error. Under certain conditions, the error may be recoverable by restarting the instruction. The recoverability is a function of the machine check code, the VAX Restart bit (VR) in the machine check stack frame, the opcode, the state of PSL<FPD>, the state of certain second-error bits in internal error registers, and most probably, the external error state.

A machine check results from an internally detected consistency error (e.g., the microcode reaches an "impossible" state), or a hardware detected error (e.g., an uncorrectable FILL_ECC error on a data read).

A machine check is technically a macro instruction abort. The NVAX Plus microcode attempts to convert the condition to a fault by unwinding the current instruction, but there is no guarantee that the instruction can be properly restarted. As much diagnostic information as possible is pushed on the stack and provided in other error registers. The rest of the error parsing is then left to the operating system.

When the software machine check handler receives control, it must explicitly acknowledge receipt of the machine check with the following instruction:

```
MTPR    #0,#PR19$_MCESR
```

14.5.1 Machine Check Stack Frame

The machine check stack frame is shown in Figure 14–3. The fields of the stack frame are described in Table 14–4, and the possible machine check codes are listed in Table 14–5. The contents of all fields not explicitly defined in Table 14–4 are UNDEFINED.

Figure 14–3: Machine Check Stack Frame

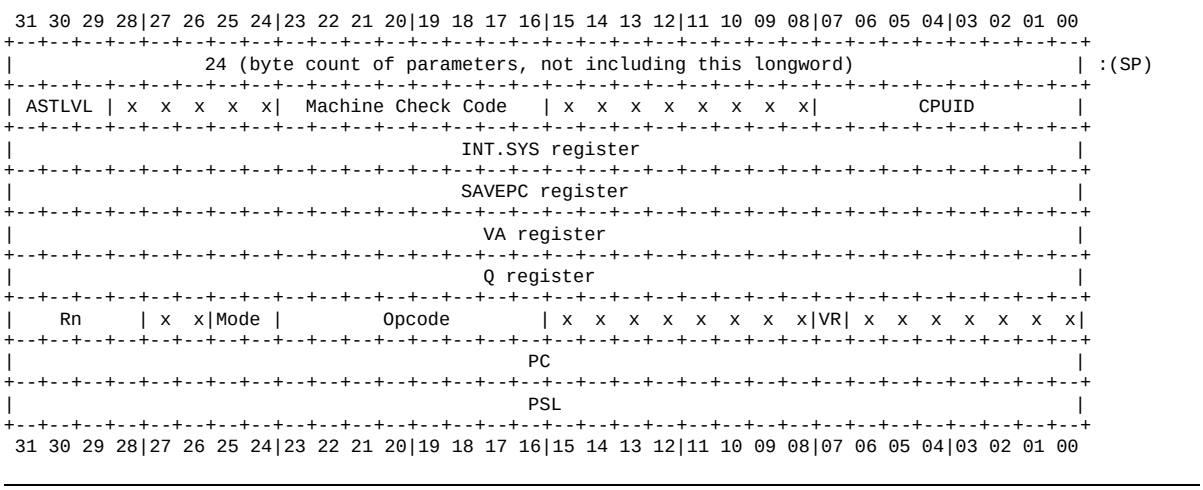


Table 14–4: Machine Check Stack Frame Fields

Longword	Bits	Contents
(SP)+0	31:0	Byte count—This longword contains the size of the stack frame in bytes, not including the PC, PSL, or the byte count longword. Stack frame PC and PSL values should always be referenced using this count as an offset from the stack pointer.
(SP)+4	31:29	ASTLVL—This field contains the current value of the VAX ASTLVL register.
	23:16	Machine check code—This longword contains the reason for the machine check, as listed in Table 14–5.
	7:0	CPUID—This field contains the current value of the VAX CPUID register.
(SP)+8	31:0	INT.SYS register—This longword contains the value of the INT.SYS register and read onto the Abus by the microcode. The fields in this register are described in the Interrupt Section chapter of the NVAX Plus chip specification Chapter 10 of the NVAX Plus chip specification.
(SP)+12	31:0	SAVEPC—This field contains the SAVEPC register which is loaded by microcode with the PC value in certain circumstances. It is used in error handling for PTE read errors with PSL<FPD> set in this stack frame.
(SP)+16	31:0	VA register—This longword contains the contents of the Ebox VA register, which may be loaded from the output of the ALU.
(SP)+20	31:0	Q register—This longword contains the contents of the Ebox Q register, which may be loaded from the output of the shifter.
(SP)+24	31:28	Rn—This field contains the value of the Rn register, which is used to obtain the register number for the CVTPL and EDIV instructions. In general, the value of this field is UNPREDICTABLE.
	25:24	Mode—This field contains a copy of PSL<CUR_MOD>.
	23:16	Opcode—This field contains bits <7:0> of the instruction opcode. The FD bit is not included.
	7	VR—This field contains the VAX Restart bit, which is used to communicate restart information between the microcode and the operating system. If this bit is set, no architectural state has been changed by the instruction which was executing when the error was detected. If this bit is not set, architectural state was modified by the instruction.

Table 14–5: Machine Check Codes

Mnemonic	Code (Hex)	Meaning
MCHK_UNKNOWN_MSTATUS	01	Unknown memory management fault parameter returned by the Mbox (see Section 14.5.2.1)
MCHK_INT.ID_VALUE	02	Illegal interrupt ID value returned in INT.SYS (see Section 14.5.2.2)
MCHK_CANT_GET_HERE	03	Illegal microcode dispatch occurred (see Section 14.5.2.3)
MCHK_MOVC.STATUS	04	Illegal combination of state bits detected during string instruction (see Section 14.5.2.4)
MCHK_ASYNC_ERROR	05	Asynchronous hardware error occurred (see Section 14.5.2.5)
MCHK_SYNC_ERROR	06	Synchronous hardware error occurred (see Section 14.5.2.6)

14.5.2 Events Reported Via Machine Check Exceptions

This section describes all the errors which can cause a machine check exception. A parse tree is given which shows how to determine the cause of a given machine check. After that, there is a description of each error. For each error, the recovery procedure is given. Where appropriate, the conditions for retry are given. See Section 14.3.3 and Section 14.3.4 for more on error recovery and error retry.

Figure 14–4 is a parse tree which should be used to analyze the cause of a machine check exception. The errors shown in the parse tree are described in detail in the sections following the figure. The section is indicated in parenthesis with each error. Note that it is assumed that the state being analyzed is the saved state, as described in Section 14.3.1. Otherwise the state could change during the analysis procedure, leading to possibly incorrect conclusions. (See Section 14.3.2 for general information about error analysis.)

Figure 14–4: Cause Parse Tree for Machine Check Exceptions

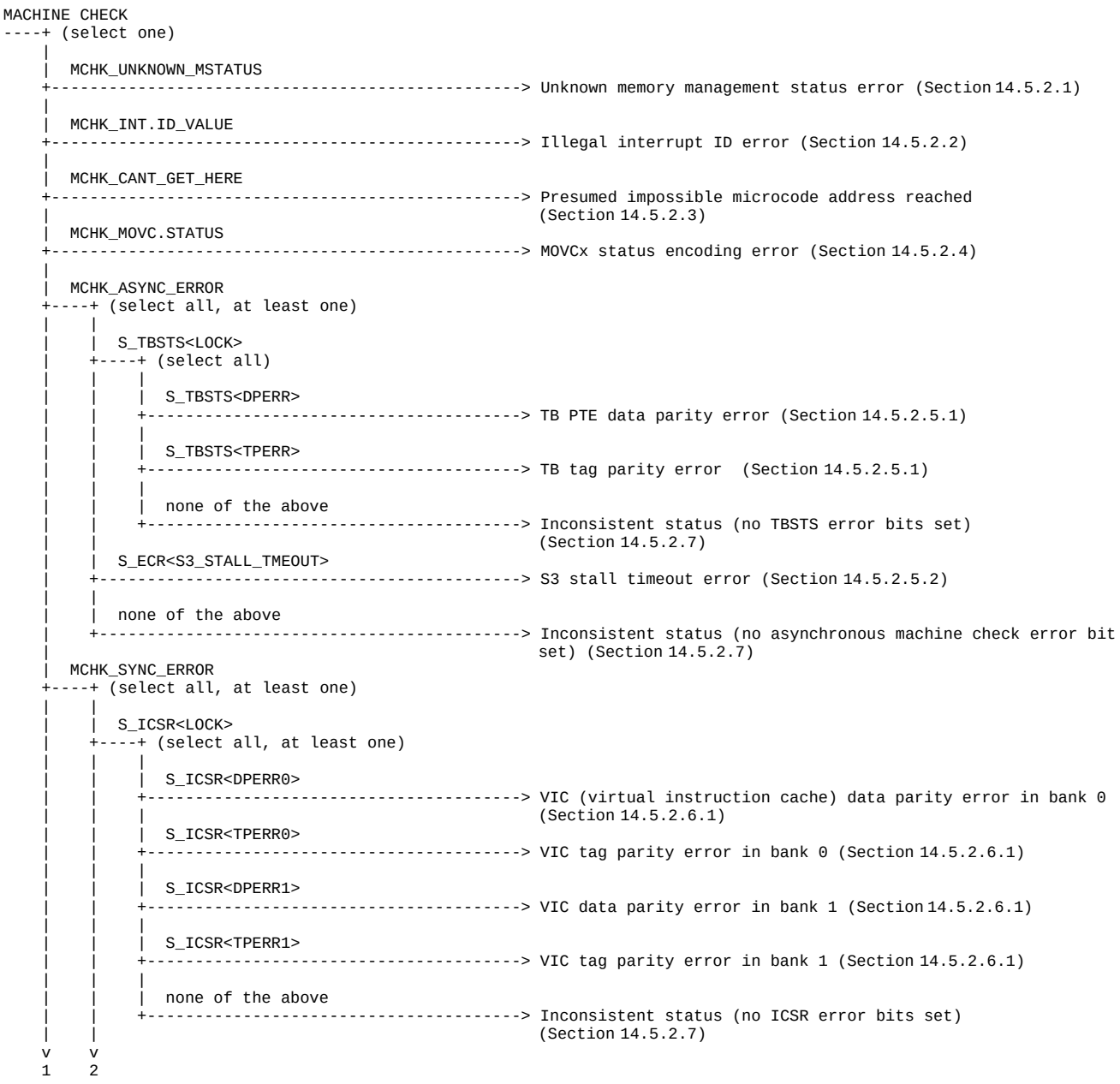


Figure 14–4 Cont'd. on next page

Figure 14–4 (Cont.): Cause Parse Tree for Machine Check Exceptions

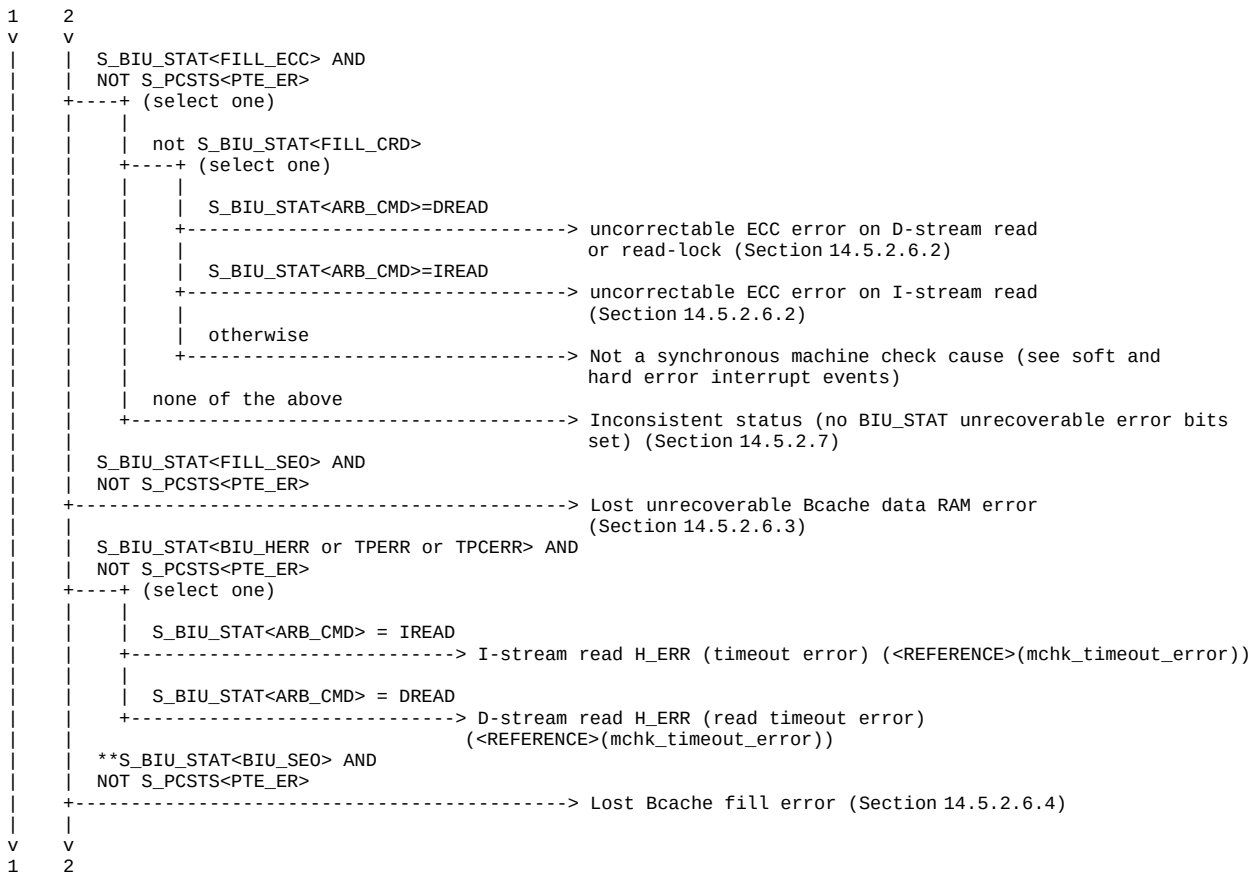
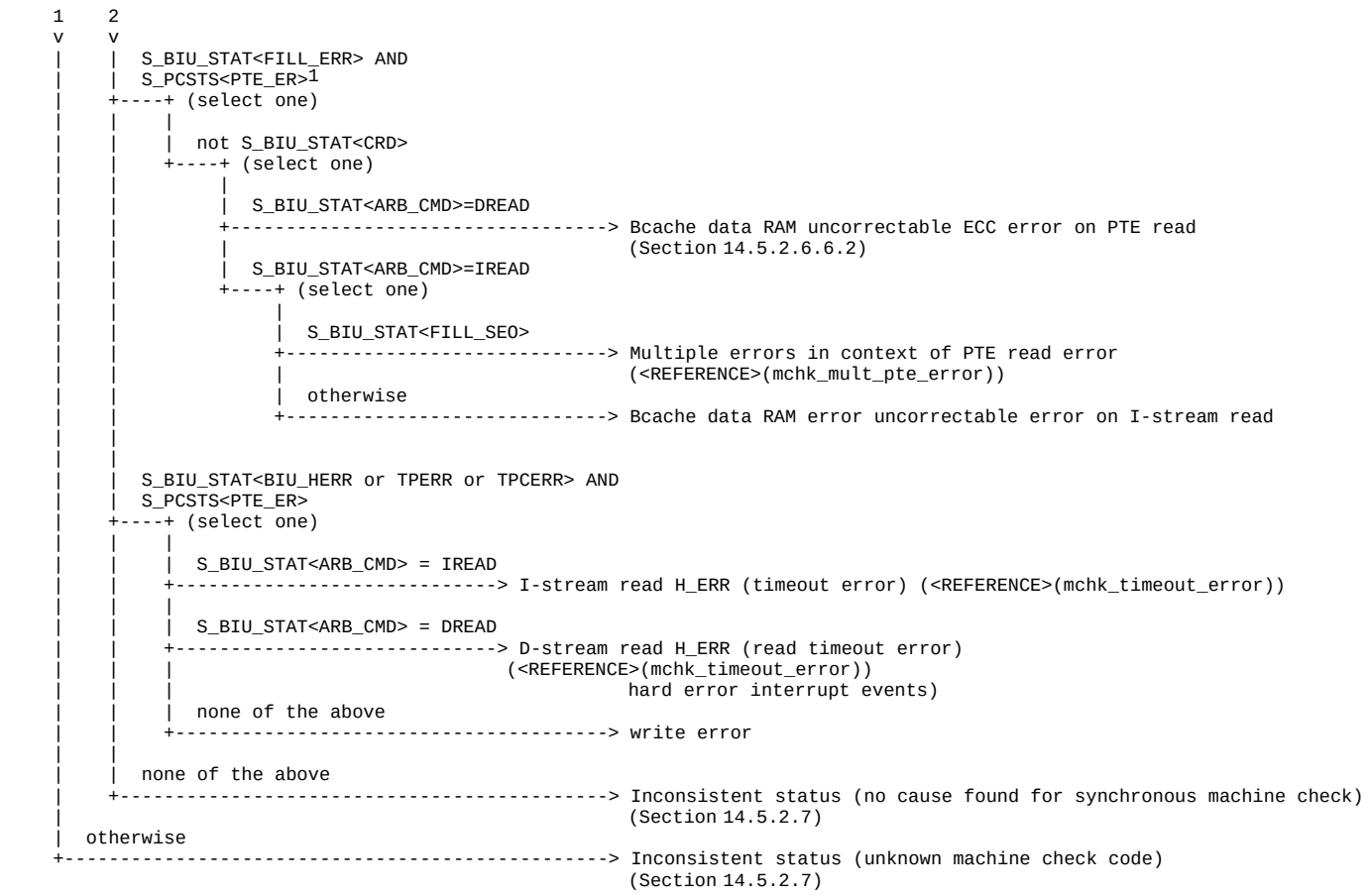


Figure 14–4 Cont'd. on next page

Figure 14–4 (Cont.): Cause Parse Tree for Machine Check Exceptions



Notation:

(select one)	- Exactly one case must be true. If zero or more than one is true, the status is inconsistent.
(select all)	- More than one case may be true.
(select all, at least one)	- All the cases are possible causes of a particular machine check. More than one may be true. At least one must be true or the status is inconsistent. A case is not considered true if it evaluates to "Not a machine check cause".
otherwise	- fall-through case for (select one) if no other case is true.
none of the above	- fall-through case for (select all) or (select all, at least one) if no other case is true.

NOTE

References to VR and PSL<FPD> in the "retry condition" parts of the following descriptions of machine check causes should be understood to refer to the named bit in the machine check stack frame.

1 At least one potential PTE cause must be found or the status is inconsistent (see Section 14.5.2.7). Some of the outcomes indicate a potential synchronous machine check cause which is not a potential PTE read error cause. These errors should be treated separately.

[1] At least one potential PTE cause must be found or the status is inconsistent (see Section 14.5.2.7). Some of the outcomes

indicate a potential synchronous machine check cause which is not a potential PTE read error cause. These errors should be treated separately.

14.5.2.1 MCHK_UNKNOWN_MSTATUS

Description: An unknown memory management status was returned from the Mbox in response to a microcode memory management probe. This is probably due to an internal error in the Mbox, Ebox, or microsequencer.

Recovery procedures: No explicit error recovery is required in response to this error.

Retry condition: This error can only happen in microcode processing of memory management faults for a virtual memory reference. Retry if:

(VR = 1) OR (PSL<FPD> = 1).

14.5.2.2 MCHK_INT.ID_VALUE

Description: An illegal interrupt ID was returned in INT.SYS during interrupt processing in microcode. This is probably due to an internal error in the interrupt hardware, Ebox, or microsequencer.

Recovery procedures: No explicit error recovery is required in response to this error.

Retry condition: This error can only happen in microcode processing of interrupts which occurs between instructions or the middle of interruptable instructions. Retry if:

(VR = 1) OR (PSL<FPD> = 1).

14.5.2.3 MCHK_CANT_GET_HERE

Description: Microcode execution reached a presumably impossible address. This is probably due to a microcode bug or an internal error in the Ebox or microsequencer.

Recovery procedures: No explicit error recovery is required in response to this error.

Retry condition: Retry if:

(VR = 1) OR (PSL<FPD> = 1).

14.5.2.4 MCHK_MOVC.STATUS

Description: During the execution of MOVCx, the two state bits that encode the state of the move (forward, backward, fill) were found set to the fourth (illegal) combination. This is probably due to an internal error in the Ebox or microsequencer.

Recovery procedures: No explicit error recovery is required in response to this error.

Retry condition: Because the state bits encode the operation, the instruction can not be restarted in the middle of the MOVCx. If software can determine that no specifiers have been over-written (MOVCx destroys R0-R5 and memory due to string writes), the instruction may be restarted from the beginning by clearing PSL<FPD>. This should be done only if the source and destination strings do not overlap and if:

(PSL<FPD> = 1).

14.5.2.5 MCHK_ASYNC_ERROR

This machine check code reports serious errors which interrupt the microcode at an arbitrary point. Many internal machine states (e.g., bits in the PSL, the PC or SP) are questionable. Recovery is typically not possible.

14.5.2.5.1 TB Parity Errors

Description: Parity errors in tags and PTE data in the TB cause an asynchronous machine check by directly forcing a microtrap in the microsequencer. The reference being processed by the Mbox may be for and explicit Ebox reference, an operand prefetch or DEST_ADDR reference from the specifier queue, or an instruction prefetch from the IREF latch. Also the reference could be a read generated by the Mbox within a TB miss for a process space virtual address since process page tables are stored in virtual memory (system space).

Description (TB PTE Data Parity Error): A parity error in the PTE data portion of a TB entry which hit had a parity error.

Description (TB Tag Parity Error): A parity error in the tag portion of a TB entry which hit had a parity error.

Recovery procedures: To recover, clear TBSTS<LOCK>.

Retry condition: Since the Ibox is nearly always able to issue instruction prefetches, TB parity errors could occur at practically any time. This makes it impossible to determine what machine state is incorrect. There is no guarantee that all writes with a different PSL<CUR_MOD> completed successfully. Therefore even the stack frame PSL<CUR_MOD> can't be used to determine whether system data is uncorrupted.

So retry is not possible. Crash the system.

NOTE

At this time, a change is being considered in REI (for reasons unrelated to TB parity errors) which might guarantee that the stack frame PSL<CUR_MOD> value is correct for TB parity errors. This would mean that if a given TB parity error occurs in user mode, for example, that writes from higher privilege modes must have completed successfully. In other words, in the event of a TB parity error, it would be known that all pages protected from writes at the stack frame privilege mode were uncorrupted. Software could kill all jobs which had access to the potentially corrupted pages instead of crashing the system. (This might be most feasible for processes incurring TB parity errors in USER mode.)

14.5.2.5.2 Ebox S3 Stall Timeout Error

Description: S3 stall timeout errors occur when the Ebox microcode is stalled waiting for some result or action which will probably never occur. S4 stalls in the Ebox cause S3 stalls and therefore can lead to S3 stall timeout. Additionally, field queue stall and instruction queue stall can cause this timeout. (These last two situations are not Ebox pipeline stalls, but they are similar in effect.) The timeout can occur in any microflow for a number of reasons. Machine state may be corrupted. This timeout is probably due to an internal error in NVAX Plus such that one box is waiting for another to do something which it isn't going to do. An example would be if the Ebox microcode expected one more source specifier than the Ibox delivered. The Ebox will stall until the timeout occurs waiting for the Ibox to deliver one more source operand via the source queue.

S3 timeout errors can be caused by failures of various pipeline control circuits in the Ebox. Also a deadlock within a box or across multiple boxes can cause this error.

Recovery procedures: To recover, clear the S3_STALL_TIMEOUT bit in ECR.

Retry condition: Because this error can occur at any time, it is not possible to determine what machine state is incorrect. Also, this error should never happen and indicates either a serious failure in the chip. So retry is not possible. Crash the system.

14.5.2.6 MCHK_SYNC_ERROR

This machine check code reports errors which occur in memory or IO space instruction fetches or data reads. Except in the case of PTE read errors, core machine state should be consistent since microcode has to explicitly access an operand or instruction in order incur this error. Microcode does not access memory results or dispatch for a new instruction execution with core machine state in an inconsistent state.

PTE read errors on write transactions can cause a microtrap at an arbitrary time, and so core machine state may be inconsistent.

Many of the error events described below for synchronous machine check are possible causes. If more than one is present, there is no way to determine which actually caused the machine check. If exactly one possible cause is discovered, then the machine check may be attributed to that cause. The reason multiple causes may be present is that the NVAX CPU prefetches instructions and data. If the CPU branches or takes an exception before using data it has requested, then the pending machine check is taken as a soft error interrupt (though it might not be recoverable in the final analysis).

If multiple errors occur, recovery and retry may be possible. It is recommended that retry from multiple errors be done only if one error report does not interfere with analysis of, and recovery from, another error.

If two errors are entirely separate, neither interfering with the analysis and recovery of the other, then it is acceptable to retry from these errors provided all the error analyses and recovery procedures result in a retry indication.

In several cases, lost errors are tolerated. See <REFERENCE>(err_retry_spec_case) for a list of these special cases. In each case, the strong tendency to prefetch data exhibited by the NVAX PLUS pipeline makes the particular lost error likely, given that one error of that kind occurred. Also, in each case, if data is lost in the lost error, a hard error interrupt is posted. So these errors are tolerated as long as they do not cause a hard error interrupt.

Errors in opcode or operand specifier fetching are always detected before architecturally visible state within the CPU is modified. This means the VR bit from the machine check stack frame should be 1. This error handling analysis attempts to recover from multiple errors, so the retry condition for each error is made as general as possible. If the machine check handler finds only errors of the kind listed here, then VR should be 1 and it is an inconsistent report if it is not (see Section 14.5.2.7).

- VIC parity errors.
- uncorrectable ECC FILL errors in I-stream reads.
- CACK H_ERR in I-stream reads.

14.5.2.6.1 VIC Parity Errors

Description: A parity error was detected in the VIC tag or data store in the Ibox. VIC parity errors cause a machine check when the Ebox microcode requests dispatch to a new instruction execution microflow or attempts to access an operand within an instruction execution microflow.

VIC Data Parity Errors: A parity error occurred in data bank 0 (DPERR0) or data bank 1 (DPERR1) of the VIC.

VIC Tag Parity Errors: A parity error occurred in tag bank 0 (TPERR0) or tag bank 1 (TPERR1) of the VIC.

In all cases, the quadword virtual address of the error is in VMAR.

Pending Interrupts: A soft error interrupt should be pending.

Recovery procedures: To recover, disable and flush the VIC by re-writing all the tags (using the procedure in Section 14.3.3.1.1.1). Also, clear ICSR<LOCK>.

Retry condition: Retry if:

(VR = 1) OR (PSL<FPD> = 1).

14.5.2.6.2 FILL Uncorrectable ECC Errors

Description (uncorrectable ECC errors): An uncorrectable data error was detected by the Cbox in an I-stream or D-stream read fill. Uncorrectable data errors are the result of a multiple bit error in the data read from the Bcache or supplied by the system on a READ_BLOCK.

Description (all cases): S_FILL_ADDR contains the address of the error, and S_FILL_ SYNDROME contains the syndrome calculated by the ECC logic.

Pending Interrupts: A soft error interrupt should be pending.

Recovery procedures (uncorrectable ECC errors): To recover, clear BIU_STAT<FILL_ ECC>.

Recovery procedures : Flush the Bcache.

Retry condition: If no writeback error occurs in the Bcache flush, retry if:

(VR = 1) OR (PSL<FPD> = 1).

If a writeback error occurs in the Bcache flush, then the data is presumed to be unrecoverable. See <REFERENCE>(err_writeback_bcache_data) for a description of handling an error in a write-back. Given that the address is available (no error in the tag store), software should determine if the error is fatal to one process or the whole system and take appropriate action. Otherwise, crash the system.

14.5.2.6.3 Bcache Lost Data RAM Access Error

Description: A lost Bcache data RAM error may have been a machine check cause. It also might not have been. Lost Bcache data RAM errors which cause machine checks are always read errors, and can be retried unless the aborted instruction has altered essential state. Whether or not it is a machine check cause, the error will have caused either a soft or hard error interrupt. Lost Bcache data RAM errors which can not have caused a machine check are dealt with in the sections on hard and soft error interrupts.

Lost Bcache data RAM errors may be caused by more than one operand prefetch to the same cache block.

Recovery for lost Bcache data RAM errors depends on whether the pending interrupt is a hard or soft error interrupt. The machine check error handling software should defer recovery until the expected hard or soft error interrupt occurs. Once the interrupt is taken, the error recovery and restart instructions found in the hard error interrupt and soft error interrupt sections should be referenced. See <REFERENCE>(hint_lost_fill_error) and <REFERENCE>(sint_lost_ndal_fill_error).

Software should employ some mechanism to record that an interrupt for a lost Bcache data RAM error is pending. This mechanism should allow detection of a case in which an expected interrupt does not occur (once IPL is lowered). If the expected interrupt does not occur when IPL is lowered, then a serious inconsistency exists and the system should be crashed.

Pending Interrupts: A hard or soft error interrupt should be pending, or possibly both.

Recovery procedures: No specific recovery action is required. Note that BIU_STAT<FILL_SEO> is not cleared. It will be cleared by the hard or soft error interrupt handler.

Retry condition: Retry only if:

(VR = 1) OR (PSL<FPD> = 1).

14.5.2.6.4 Lost Bcache Fill Error

Description: Some number of fill errors occurred and were not latched because BIU_STAT already contained a report of an error. There is no guarantee this error could have caused a machine check, though it may be a cause. Lost Bcache fill errors which cause machine checks are always read errors, and can be retried unless the aborted instruction has altered essential state. If it is a machine check cause, the error will have caused a a soft error interrupt. Lost Bcache fill errors which can not have caused a machine check are dealt with in the sections on hard and soft error interrupts.

Lost Bcache fill errors may be caused by more than one operand prefetch to the same cache block.

Recovery for lost Bcache fill errors depends on whether the pending interrupt is a hard or soft error interrupt. The machine check error handling software should defer recovery until the expected hard or soft error interrupt occurs. Once the interrupt is taken, the error recovery and restart instructions found in the hard error interrupt and soft error interrupt sections should be referenced. See <REFERENCE>(hint_lost_fill_error) and <REFERENCE>(sint_lost_ndal_fill_error).

Software should employ some mechanism to record that an interrupt for a lost Bcache fill error is pending. This mechanism should allow detection of a case in which an expected interrupt does not occur (once IPL is lowered). If the expected interrupt does not occur when IPL is lowered, then a serious inconsistency exists and the system should be crashed.

Pending Interrupts: A hard or soft error interrupt should be pending, or possibly both.

Recovery procedures: No specific recovery action is required. Note that BIU_STAT<FILL_SEO> is not cleared. It will be cleared by the hard or soft error interrupt handler.

Retry condition: Retry only if:

(VR = 1) OR (PSL<FPD> = 1).

14.5.2.6.5 CACK_HERR

Description: An I-stream or D-stream read returned CACK_HERR the system environment.

I-stream errors cause a machine check when the Ebox microcode requests dispatch to a new instruction execution microflow or attempts to access an operand within an instruction execution microflow.

D-stream read errors cause a machine check when the Ebox microcode accesses prefetched operand data or when the Mbox returns data tagged with an error indication to the Ebox register file.

The address should not be in IO space. If it is, it is an inconsistent status (see Section 14.5.2.7).

D-stream ownership read errors cause a machine check when the Ebox microcode accesses prefetched operand data.

Pending Interrupts (all cases): A soft error interrupt should be pending.

Recovery procedures (all cases): Clear BIU_STAT<BIU_HERR>.

Retry condition: Retry if:

(VR = 1) OR (PSL<FPD> = 1).

14.5.2.6.6 PTE read errors

The following sections describe error handling for PTE read errors. PTE read errors are read errors which happen in reads issued by the Mbox in handling a TB miss. Handling of these errors is different from handling the same underlying error (BIU_HERR, BC_TPERR, BC_TCPERR, FILL_ECC) when PTE read isn't the cause.

If S_PCSTS<PTE_ER> is set, then a PTE read issued by the Mbox in processing a TB miss had an unrecoverable error. The TB miss sequence was aborted because of the error. The original reference can be any I-stream or D-stream read or write. If the original reference was issued by the Ebox, then the PTE read which incurred the error will have been retried once (because of a special hardware/microcode mechanism for handling PTE read errors on Ebox references).

PTE read errors are difficult to analyze, partly because the read error report in the Cbox does not directly indicate that the failing read was a PTE read. Because of this and because PTE read errors should be rare (a very small percentage of the reads issued by the Mbox are PTE reads), multiple errors which interfere with the analysis of the PTE error are not considered recoverable.

The mechanism for reporting PTE read errors on Ebox references involves the Mbox forcing the Ebox (via a microtrap) into the microcode routine which normally handles memory management faults. This routine probes the address of the original reference, effectively retrying the failing PTE read. Assuming the error is not transient, the probe by microcode will cause a machine check. If the error does not occur on the probe, microcode restarts the current instruction stream. So machine checks caused by PTE read errors can easily occur with the particular PTE read error having occurred twice (with a lost error bit set in the relevant Cbox error register). The analysis here tolerates these particular multiple error reports and allows retry in those cases, provided the remainder of the error analysis indicates retry is appropriate. (Note that there is no way to tell from the information available to the machine check handler whether the original reference was an Ebox or Ibox reference.)

If the reference which incurs the PTE read error is a write, S_PCSTS<PTE_ER_WR> will be set. In this case the original write is lost. No retry is possible partly because the instruction which took the machine check may be subsequent to the one which issued the failing write. Also, PTE read errors on write transactions can cause a machine check at an practically arbitrary time in a microcode flow, and core machine state may not be consistent.

14.5.2.6.6.1 PTE Read Errors in Interruptable Instructions

Another special case associated with PTE read errors exists for interruptable instructions (specifically CMPC3, CMPC5, LOCC, MOVC3, MOVC5, SCANC, SKPC, and SPANC). For these instructions, if the PTE read error occurred for an Ebox reference, the PC in the machine check stack frame points to the instruction following the interrupted instruction. In this case, the SAVEPC element in the machine check stack frame is the PC of the interrupted instruction. However in all other cases, SAVEPC is UNPREDICTABLE. This case is not considered recoverable because analysis of the error information can not unambiguously conclude that this case is present. To tell that this case might be present, the error handler examines the FPD bit in the PSL in the machine check stack frame. If FPD is set in the stack frame (in the case of a PTE read error) then one of the following is true:

- One of the interruptable instructions listed above incurred the PTE read error. In this case, SAVEPC from the machine check stack frame points to the interrupted instruction, and PC in the stack frame points to the next instruction.
- An REI instruction loaded a PSL with FPD set and a certain PC. The Ibox incurred the PTE read error in fetching the opcode pointed to by that PC. In this case, the PC in the stack frame points to the instruction which was the target of the REI and SAVEPC from the stack frame is unpredictable.

It is not possible to determine with certainty which of the two above cases is the cause of a machine check with S_PCSTS<PTE_ER> set and stack frame PSL<FPD> set. Retry is not possible since software can not tell which PC to restart with. However, software may wish to probe the location pointed to by the PC in the stack frame, expecting a possible machine check as a result. If a machine check does occur, that is information indicating that the second case occurred (not totally unambiguously, of course). A very good guess may be made by a person examining the error report if the machine check stack frame and the result of this probe is available in the report.

14.5.2.6.6.2 Uncorrectable ECC FILL Errors and on PTE Reads

Description (uncorrectable ECC errors): A FILL uncorrectable data error was detected by the Cbox in a PTE read. Uncorrectable data errors are the result of a multiple bit error in the data read from the Bcache, of FILL from the system on a READ_BLOCK.

Description (all cases): S_FILL_ADDR contains the cache address of the error, and FILL_ SYNDROME contains the syndrome calculated by the ECC logic. (If the physical address is found to be in IO space, it is an inconsistent status. See Section 14.5.2.7.)

S_BIU_STAT<FILL_SEO> may be set. This error is probably due to the same PTE error occurring more than once. This is an acceptable assumption unless a hard error interrupt occurs after handling this error.

Pending Interrupts: A soft error interrupt should be pending.

Recovery procedures (uncorrectable ECC errors): To recover, clear BIU_STAT<FILL_ECC>.

Recovery procedures (both cases): Flush the Bcache. Clear PCSTS<PTE_ER>.

Retry condition: If no writeback error occurs in the Bcache flush, retry if:

(VR = 1) AND (PSL<FPD> = 0) AND (S_PCSTS<PTE_ER_WR> = 0).

If

(PSL<FPD> = 1) OR (S_PCSTS<PTE_ER_WR> = 1),

crash the system. If a writeback error occurs in the Bcache flush, then the data is presumed to be unrecoverable. See <REFERENCE>(err_writeback_bcache_data) for a description of handling an error in a writeback (software must determine if the error is fatal to one process or the whole system and take appropriate action).

14.5.2.6.6.3 CACK_HERR on PTE Read

Description: A PTE read returned CACK_HERR.

S_BIU_STAT<BIU_SEO> may be set. This error is probably due to the same PTE error occurring more than once. This is an acceptable assumption unless a hard error interrupt occurs after handling this error.

Pending Interrupts: A soft error interrupt should be pending.

Recovery procedures: Clear BIU_STAT<CACK_HERR>. Clear PCSTS<PTE_ER>.

Retry condition: Retry if:

(VR = 1) AND (PSL<FPD> = 0) AND (S_PCSTS<PTE_ER_WR> = 0).

Otherwise, crash the system.

Post Retry Recovery: If the same fill error recurs on retry, then the block is probably "lost". In this case the more general sense of "lost" is implied. Software must determine if the error is fatal to one process or the whole system and take appropriate action.

NOTE

It may be appropriate in this case to first cause each CPU in the system to flush its Bcache, and then retry once more.

14.5.2.7 Inconsistent Status in Machine Check Cause Analysis

Description: A presumed impossible error report was found in the error registers. This could be due to a hardware failure or bug, or to incomplete analysis in this spec.

Pending Interrupts: A hard or soft error interrupt should be pending, or possibly both.

Recovery procedures: No specific recovery action is called for.

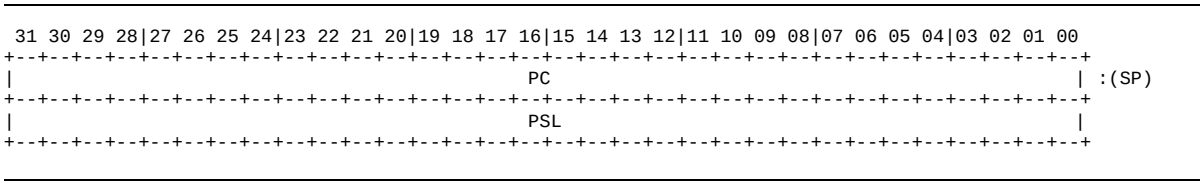
Retry condition: No retry is possible. The integrity of the entire system is questionable. Crash the system.

14.6 Power Fail

Power fail interrupts are requested to report imminent loss of power to the CPU. Power fail interrupts are requested via the ERR_H pin at IPL 1D (hex) and are dispatched to the operating system through SCB vector 60 (hex) as is hard error interrupt.

The stack frame for a power fail interrupt is shown in Figure 14–5.

Figure 14–5: Power Fail Interrupt Stack Frame

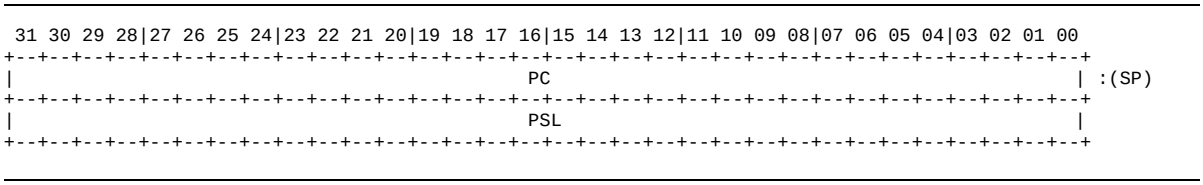


14.7 Hard Error Interrupts

Hard error interrupts are requested to report an error that was detected asynchronously with respect to instruction execution. This results in an interrupt at IPL 1D (hex) to be dispatched through SCB vector 60 (hex). Typically, these error indicate that machine state has been corrupted and that retry is not possible.

The stack frame for a hard error interrupt is shown in Figure 14–6.

Figure 14–6: Hard Error Interrupt Stack Frame



14.7.1 Events Reported Via Hard Error Interrupts

This section describes all the errors which can cause a hard error interrupt.

Figure 14–7: Cause Parse Tree for Hard Error Interrupts

```
HARD ERROR INTERRUPT
----+ (select all, at least one)
    |
    |  (status consistent with hard error interrupt
    |  in system environment error registers)
    +-------------------------------------------------> Hard error interrupt from system environment
    |                                                   (Section 14.7.1.2)
    |
    |  BIU_STAT<lost_write_err>
    +-------------------------------------------------> Uncorrectable ECC error on a write from Mbox
    |                                                   (Section 14.7.1.1)
    |
    |  BIU_STAT<BIU_HERR> and BIU_STAT<BIU_CMD> = WRITE
    +-------------------------------------------------> System failure (timeout) on a write from Mbox
    |                                                   (Section 14.7.1.1)
    |
    |  BIU_STAT<BC_TPERR> and BIU_STAT<BIU_CMD> = WRITE
    +-------------------------------------------------> Bcache tag parity error on a write from Mbox
    |                                                   (Section 14.7.1.1)
    |
    |  BIU_STAT<BC_TCPERR> and BIU_STAT<BIU_CMD> = WRITE
    +-------------------------------------------------> Bcache tag control parity error on a write from Mbox
    |                                                   (Section 14.7.1.1)
    |
    |  BIU_STAT<FILL_ECC> and not BIU_STAT<CRD>
    |          and BIU_STAT<ARB_CMD> = WRITE
    +-------------------------------------------------> Uncorrectable ECC error on a write from Mbox
    |                                                   (Section 14.7.1.1)
    |
    |  otherwise
    +-------------------------------------------------> Inconsistent status (Section 14.7.1.3)

Notation:
    (select all, at least one) - All the cases are possible causes of a hard error interrupt.
                                 More than one may be true. At least one must be true or the status
                                 is inconsistent.
```

14.7.1.1 Uncorrectable Errors During Write or Write-Unlock Processing

Description: In processing a write or write-unlock, the Cbox detected a CACK = HERR from the system, a tag parity error, a control parity error, or an uncorrectable ECC error on the data read which is to be merged Data from the write is lost.

Uncorrectable ECC errors indicate that two or more bits of the stored data quadword have changed and the error correcting code can not correct the data. The write merge sequence is aborted.

Recovery procedures : The data in this block is lost.

Restart condition : If the address of the data is available and no unexpected writeback errors occurred during the Bcache flush, software must determine if the lost data is fatal to one process or the whole system and take the appropriate action.

14.7.1.2 System Environment Hard Error Interrupts

TBS.

14.7.1.3 Inconsistent Status in Hard Error Interrupt Cause Analysis

Description: A presumed impossible error report was found in the error registers. This could be due to a hardware failure or bug.

Recovery procedures: No specific recovery action is called for.

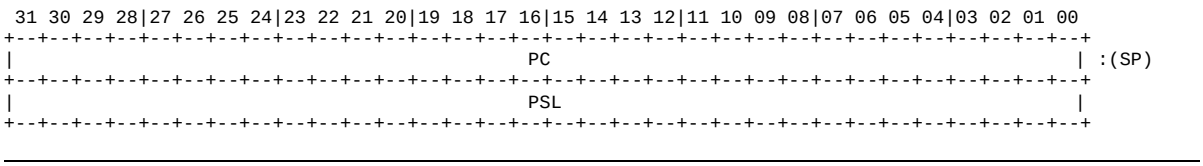
Restart condition: No retry is possible. The integrity of the entire system is questionable. Crash the system.

14.8 Soft Error Interrupts

Soft error interrupts are requested to report errors which were detected, but did not affect instruction execution. This results in an interrupt at IPL 1A (hex) to be dispatched through SCB vector 54 (hex).

The stack frame for a soft error interrupt is shown in Figure 14–8.

Figure 14–8: Soft Error Interrupt Stack Frame



14.8.1 Events Reported Via Soft Error Interrupts

This section describes the errors which can cause a soft error interrupt.

Note that many errors which cause a soft error interrupt may also lead to a machine check exception. For this reason, a soft error interrupt with no apparent cause is not an inconsistent state unless the CPU has executed an instruction while IPL was lower than 1A (hex) since the most recent machine check exception.

When a soft error interrupt is the only notification for any memory read error which could cause a machine check, the error didn't cause a machine check for one of the following reasons.

- The error did not occur on the quadword the Ebox or Ibox requested (Pcache fill error).
- The Ebox took an interrupt before accessing an instruction or operand which was prefetched by the Ibox. (It could be this soft error interrupt.)
- A prefetched instruction or operand belonged to an instruction following a mispredicted branch, so the Ebox never executed the instruction (and it was flushed from the pipeline when the branch mispredict was recognized).
- The Ebox took an exception for a different reason before attempting to use an instruction execution dispatch or access an operand prefetched by the Ibox. (The pipeline was flushed because of the exception.)

Figure 14–9: Cause Parse Tree for Soft Error Interrupts

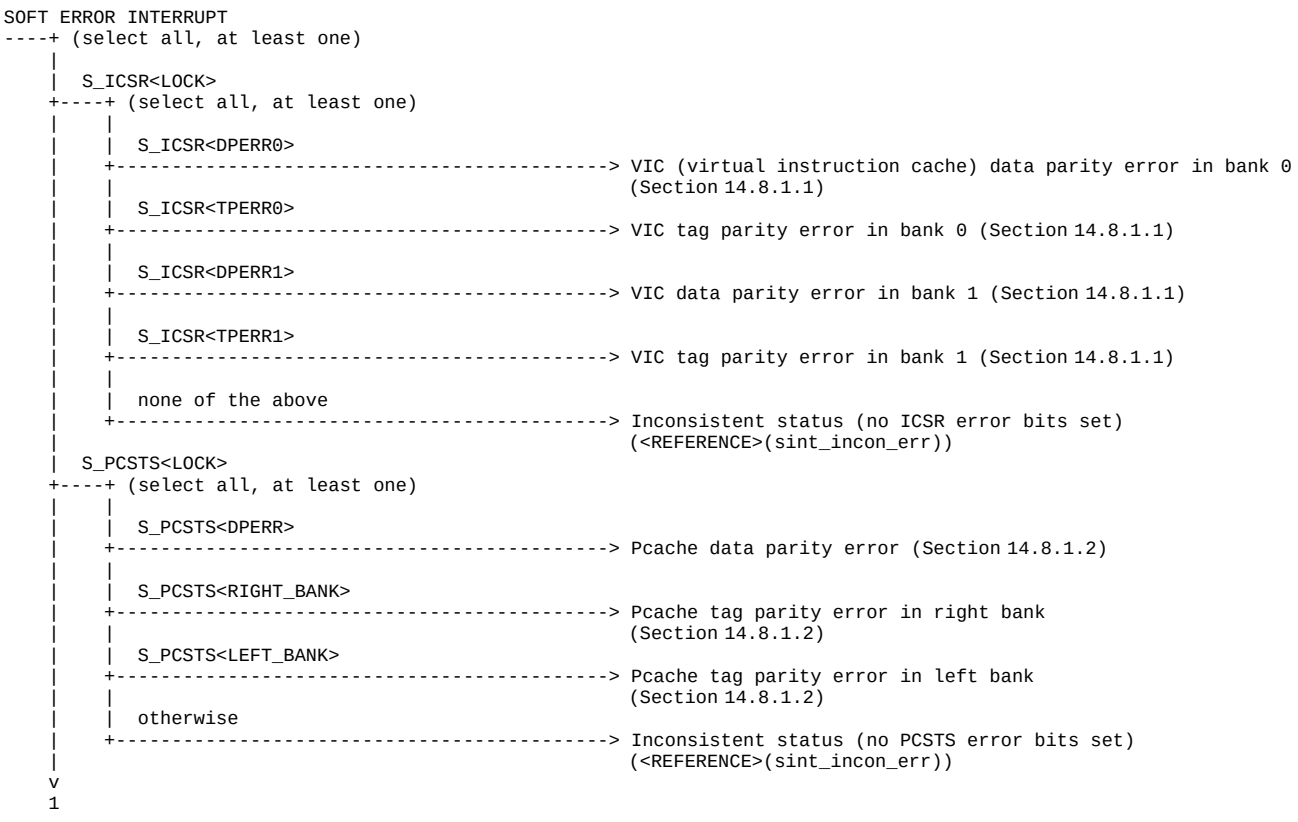
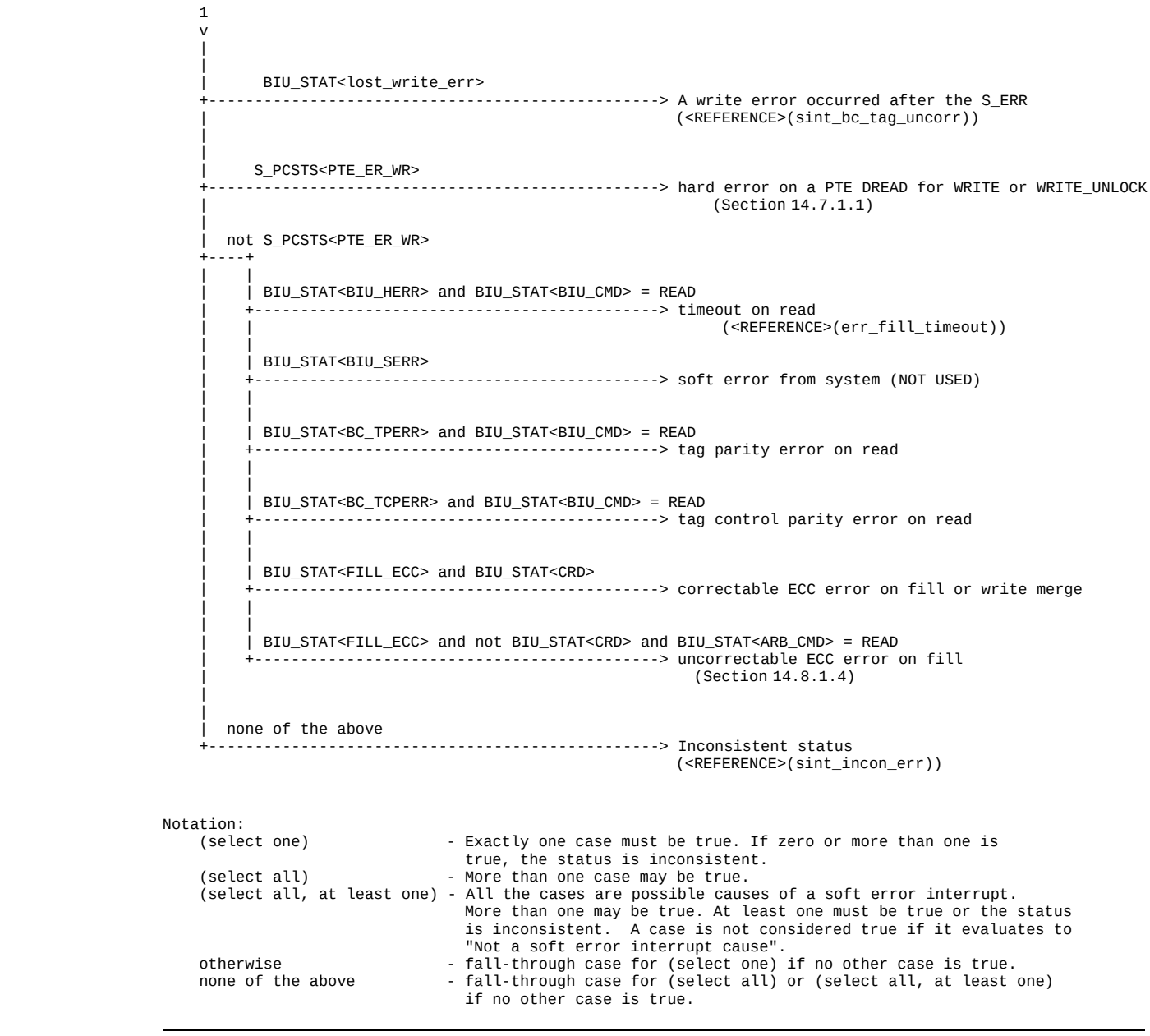


Figure 14–9 Cont'd. on next page

Figure 14–9 (Cont.): Cause Parse Tree for Soft Error Interrupts



14.8.1.1 VIC Parity Errors

Description: A parity error was detected in the VIC tag or data store in the Ibox.

VIC Data Parity Errors: A parity error occurred in data bank 0 (DPERR0) or data bank 1 (DPERR1) of the VIC.

VIC Tag Parity Errors: A parity error occurred in tag bank 0 (TPERR0) or tag bank 1 (TPERR1) of the VIC.

In all cases, the quadword virtual address of the error is in S_VMAR.

Recovery procedures: To recover, disable and flush the VIC by re-writing all the tags (using the procedure in Section 14.3.3.1.1.1). Also, clear ICSR<LOCK>.

14.8.1.2 Pcache Parity Errors

Description: A parity error was detected in the Pcache. Either a tag parity error or a data parity error is reported, though tag parity errors in both the left and right banks may be reported simultaneously. The reference, whether it was a read or write, was passed to the Cbox as if the Pcache had missed. No data is lost. The Pcache is disabled because PCSTS<LOCK> is set.

S_PCADR contains the physical address of operation incurring the error. The address should not be in IO space. If it is, it is an inconsistent status (see <REFERENCE>(sint_incon_err)).

Recovery procedures: Clear PCSTS<LOCK>. Flush the Pcache and initialize the Pcache tag store (see <REFERENCE>(err_flush_bcache_procedure)).

14.8.1.3 Lost Bcache Data RAM Correctable ECC Errors

Description: A correctable error occurred in accessing the Bcache data RAM, but it is lost because of an uncorrectable data RAM error which also occurred. The address and syndrome of the error are not known.

Recovery procedures: Clear BCEDSTS<CORR>.

The Bcache should be flushed (and it would be because of the uncorrectable error in any case). This effectively scrubs the Bcache data RAM location by invalidating it and forcing it to be written back if it is owned.

14.8.1.4 FILL Uncorrectable ECC Errors on I-Stream or D-Stream Reads

Description (uncorrectable ECC error): A Fill uncorrectable ECC error was detected by the Cbox in an I-stream or D-stream read. Uncorrectable data errors are the result of a multiple bit errors in the data read.

Description : S_FILL_ADDRESS contains the address of the error, and S_FILL_SYNDROME contains the syndrome calculated by the ECC logic. (If the physical address is found to be in IO space, it is an inconsistent status. See <REFERENCE>(sint_incon_err)

Recovery procedures: To recover, clear BIU_STAT<FILL_ECC>.

Flush the Bcache. **(BC_TAG CAN BE USED TO DETERMINE IF THE FILL IS FROM BCACHE)** If the data is DIRTY in the Bcache and if the error repeats itself (is not transient), then a writeback error will result from the flush procedure.

Restart Conditions: If a writeback error occurs in the Bcache flush, then the data is presumed to be unrecoverable. Software must determine if the error is fatal to one process or the whole system and take appropriate action.

If the address of the error in the flush is not the same as that of the original error, this is a multiple error case in the data RAMs and is a serious failure. Crash the system.

PTE read errors are difficult to analyze, partly because the read error report in the Cbox does not directly indicate that the failing read was a PTE read. Because of this and because PTE read errors should be rare (a very small percentage of the reads issued by the Mbox are PTE reads), multiple errors which interfere with the analysis of the PTE error are not considered recoverable.

If the reference which incurs the PTE read error is a write, S_PCSTS<PTE_ER_WR> will be set. In this case the original write is lost. No retry is possible partly because the instruction which took the machine check may be subsequent to the one which issued the failing write. Also, PTE read errors on write transactions can cause a machine check at an practically arbitrary time in a microcode flow, and core machine state may not be consistent.

Restart condition: If no writeback error occurs in the Bcache flush, restart if:

(S_PCSTS<PTE_ER_WR> = 0).

If

(S_PCSTS<PTE_ER_WR> = 1),

crash the system.

If a writeback error occurs in the Bcache flush, then the data is presumed to be unrecoverable. (software must determine if the error is fatal to one process or the whole system and take appropriate action). Clear PCSTS<PTE_ER>.

Restart condition: Restart if:

(S_PCSTS<PTE_ER_WR> = 0).

Otherwise, crash the system.

14.8.1.4.1 Multiple Errors Which interfere with Analysis of PTE Read Error

Because PTE read errors lead to several unusual cases, restart is not recommended in the event that other errors cloud the analysis of the PTE read error.

Pending Interrupts: A hard or soft error interrupt should be pending, or possibly both.

Recovery procedures: No specific recovery action is called for.

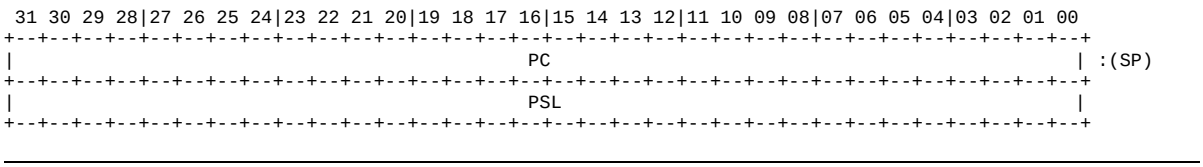
Restart condition: No restart is possible. Crash the system.

14.9 Kernel Stack Not Valid Exception

A Kernel Stack Not Valid Exception occurs when a memory management exception is detected while attempting to push information on the kernel stack during microcode processing of another exception. Note that a console halt with an error code of ERR_INTSTK is taken if a memory management exception is encountered while attempting to push information on the interrupt stack.

The Kernel Stack Not Valid exception is dispatched through SCB vector 08 (hex) with the stack frame shown in Figure 14–10.

Figure 14–10: Kernel Stack Not Valid Stack Frame



14.10 Error Recovery Coding Examples

To be supplied.

14.11 Revision History

Table 14–6: Revision History

Who	When	Description of change
Mike Uhler	19-Dec-1989	Update for second-pass release.
John Edmondson	30-Jun-1990	Update further after internal review and resolution of many issues.
Gil Wolrich	20-Feb-1991	Modify for NVAX Plus.

Chapter 15

Chip Initialization

15.1 Overview

This chapter describes the hardware initialization process for the NVAX Plus chip. The hardware and microcode start the initialization, and then if not SROM_FAST, the 8K bytes of data are read from the Serial Rom and loaded into the Pcache. If SROM_FAST microcode passes control to macrocode at address E0040000.

Much of the job of initialization involves setting the NVAX internal processor registers (IPRs) to a known state, or using NVAX IPRs to perform functions such as cache initialization. See Chapter 2 for a list of the NVAX IPRs. Also, see the individual box chapters for a more in depth definition of many of the IPRs.

15.2 Hardware/Microcode initialization

The NVAX Plus Chip hardware initializes to the following state on powerup or the assertion of chip reset:

1. The VIC, Pcache, and Bcache are disabled.
2. The RLOG is cleared.
3. The Fbox is disabled.
4. The microstack is cleared.
5. The Mbox and Cbox are reset, and all previous operations are flushed.
6. The Fbox is reset.
7. The Ibox is stopped, waiting for a LOAD PC.
8. All instruction and operand queues are flushed.
9. All MD valid bits are cleared, and all Wn valid bits are set.
10. A powerup microtrap is initiated which starts the Ebox at the label IE.POWERUP..

The NVAX Plus Chip microcode at IE.POWERUP then does the following:

1. Hardware interrupt requests are cleared.
2. BIU_STAT is cleared.
3. BIU_CTL is cleared.
4. ICCS is cleared.

5. SISR<15:1> is set to 0.
6. ASTLVL is set to 4.
7. The Mbox PAMODE IPR is set to 30-bit physical address mode.
8. CPUID is set to 0.
9. The BPCR branch history algorithm is reset to the default value.
10. Backup PC is retrieved from the Ibox and saved in SAVPC.
11. PME is cleared. The performance monitoring counters are cleared.
12. The current PSL, halt code, and value of MAPEN are saved in SAVPSL.
13. MAPEN is cleared (memory management is disabled).
14. All state flags are cleared.
15. PSL is loaded with 041F0000.
16. If not SROM FAST load Pcache from the Serial Rom
17. If SROM FAST the PC is loaded with 00000000

The powerup microcode provides a means for loading start-up code from the serial ROM. This microcode could also be used for loading the burn-in and life-test programs. The P-cache is loaded with bit-serial instruction stream data.

```
            o Enable serial ROM this will also tell C-box we are reading
              the serial ROM.
            o Check SROM_FAST bit, if set go to serial ROM fast code.
            o Begin normal serial ROM read and P-cache load, enable P-cache
      loop: o Assert serial line out high for a minimum of 200ns
            o Assert serial line out low for a minimum of 200ns
            o Read data from serial line in and append value onto I-stream data.
            o If I-stream data = 32 bits, then write into P-cache, VA = VA + 4.
            o If every 8th Longword written then write new tag data
              for the next P-cache tag.
            o If I-stream data = 32K bits, then switch P-cache banks.
            o If I-stream data = 64K bits, then go to exit:
            o Go to loop:
      exit: o Write address of power up code to console halt reg.
            o disable SROM, join console code to load PC.

   o PC is loaded with beginning address of SROM code that was loaded into
     the P-cache.
```

NOTE:

```
Currently the serial ROM fast code does nothing except load the
console halt register with what would be the start-up address of
the SROM code and joins the console halt flow to load the value
in that register as the next PC and jump to it.  The P-cache is
disabled.  Currently the address loaded into the PC is zero.
All code related to serial ROM fast is of course subject to change.

On normal serial ROM loading, the P-cache is enabled for I-stream,
D-stream, and parity error detection.  All tags have been initialized
and force hit in not enabled.  Again the console halt register is
loaded with zero, which is the beginning of where the SROM code was
loaded.  This value is used for the start PC.
```

Special Note:

```
Currently none of this is used for the Behavioral Model and the PC
is loaded with a microcoded constant of 800000.
```

15.3 Console initialization

The console macrocode has the job of filling the gap between the initialized state described above and the initial state needed for the operating system. To that end, the console code does the following:

1. Set CPUID to the correct value from the system environment.
2. Set ECR (Ebox Control Register) as follows:
    1. Set FBOX_ENABLE to enable the Fbox.
    2. Set S3_TIMEOUT_EXT as required by the system environment.
    3. Set FBOX_ST4_BYPASS_ENABLE to enable Fbox stage 4 bypass.
    4. Write one to S3_STALL_TIMEOUT to clear any error.
3. Set ICSR (Ibox Control Status Register) as follows:
    1. Clear ENABLE to leave the VIC disabled.
    2. Write one to LOCK to clear any error.
4. Set the PAMODE register MODE bit as required by the system.
5. Set up BIU_CTL (Bcache/System Control) as required by the system.

15.4 Other initialization

Either the console code or the operating system will do the following final initialization steps (code examples are given):

1. Initialize the VIC

```
VIC_MAX_INDEX := 3E0 (hex)
VIC_INDEX_STEP := 20 (hex)
VIC_TAG_INIT := 0

FOR INDEX := 0 TO VIC_MAX_INDEX BY VIC_INDEX_STEP DO
BEGIN
    MTPR INDEX, VMAR
    MTPR VIC_TAG_INIT,VTAG
END;
```

2. Enable the VIC

```
MTPR ENABLE, ICSR
```

3. Initialize the Pcache, Enable the Pcache. The Pcache is initialized by microcode if not SROM FAST.
4. Initialize the Bcache
5. Enable the Bcache, set BIU_CTL[0]

15.5 Revision History

Table 15–1: Revision History

Who	When	Description of change
Debra Bernstein	9-May-1990	Initial edit
Jim Ellis/Gil Wolrich	15-JAN-1991	NVAX Plus release for external review

Chapter 16

Performance Monitoring Facility

16.1 Overview

The NVAX CPU chip contains a facility by which privileged software may obtain performance information about the dynamic behavior of the CPU. The facility is implemented with a combination of hardware and microcode, and controlled by software using privileged instructions.

Two 64-bit performance counters called PMCTR0 and PMCTR1 are maintained in memory for each CPU in the system. The lower 16 bits of each counter are implemented in hardware in the CPU, and at specified points, microcode updates the quadwords in memory with the contents of the hardware counters.

The performance monitoring facility may be configured by privileged software to count a number of events in the system, from which performance analysis data such as cache and TB hit rates, cycles-per-instruction, and stall frequencies may be calculated.

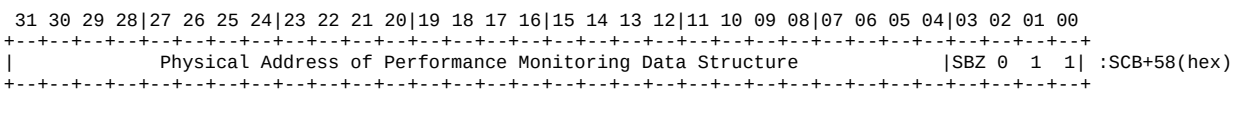
16.2 Software Interface to the Performance Monitoring Facility

The performance monitoring facility makes use of a data structure in memory, and must be configured and enabled via a location in the System Control Block, processor register references, and the LDPCTX instruction.

16.2.1 Memory Data Structure

The two 64-bit performance counters for each CPU are maintained in a data structure in memory. This data structure consists of a pair of quadwords for every CPU in the system. The physical address of the base of the data structure is obtained from offset 58 (hex) in the System Control Block. The format of this location is shown in Figure 16–1.

Figure 16–1: Performance Monitoring Data Structure Base Address



NOTE

An quadword-aligned physical base address is constructed by clearing the lower 3 bits of the longword fetched from offset 58 (hex) in the SCB. Microcode will not update the block in memory unless bits <2:0> of this longword contain 011 (binary). If these bits are found to contain another value, a machine check with code MCHK_PMF_CONFIG is performed to notify software that the performance monitoring facility was incorrectly configured. If is strongly suggested that the physical address be at least octaword aligned, and preferably page aligned.

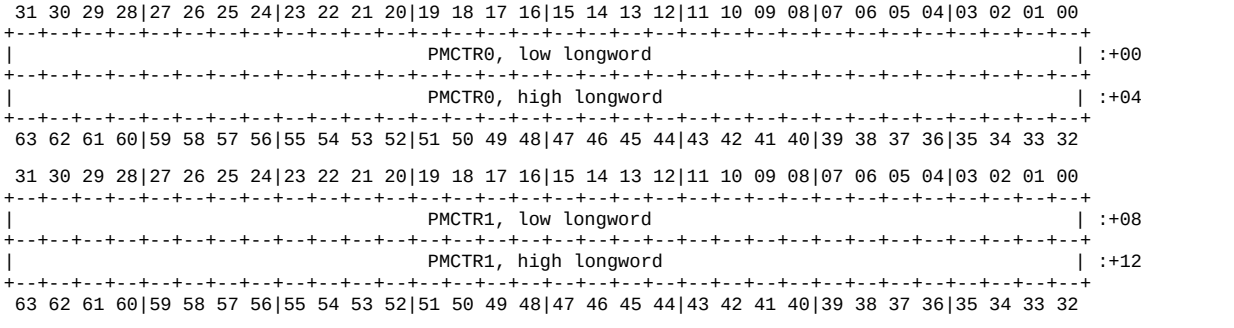
The address of the pair of quadwords for an individual CPU is computed by shifting the CPUID value left 4 bits and adding this value to the base address. This calculation is shown in equation form below (all numbers in these equations are hex).

$$phys\_base\_addr = SCB[58]\ \ AND\ FFFFFFF0;$$

$$phys\_block\_addr = \{\ CPUID\ LSHIFT\ 4\ \} + phys\_base\_addr;$$

The format of the pair of quadwords for each CPU is shown in Figure 16–2.

Figure 16–2: Per-CPU Performance Monitoring Data Structure



16.2.2 Memory Data Structure Updates

When the performance monitoring facility is enabled, the memory data structure is updated from the hardware counters if the one of the counters is more than half full and the current processor IPL is below 1B (hex), if a LDPCTX instruction is executed and the PME bit in the new PCB is off, or if the performance monitoring facility is disabled via a write to the PME processor register. The PME bit is internally implemented as ECR<PMF_ENABLE>, with conversion handled by microcode.

When one of the counters reaches half full, an interrupt at IPL 1B (hex) is requested. This interrupt request is serviced like any other interrupt if the IPL of the processor is below that of the interrupt request IPL. Like any other interrupt, it is serviced between instructions (or in the middle of the interruptable string instructions). Unlike other interrupts, the performance monitoring interrupt is serviced entirely by microcode, with no software interrupt handler required.

When a performance monitoring interrupt occurs, microcode temporarily disables the facility, reads and clears the hardware counters, then updates the memory data structure with the hardware counts. The facility is then re-enabled, the interrupt is dismissed, and the interrupted instruction stream is restarted.

NOTE

Although the performance monitoring facility is disabled during the memory update process, it is re-enabled for the restart of the interrupted instruction stream. Therefore, depending on what events were selected, the facility may count events that are part of the restart process.

At the maximum rate (one increment every 14ns CPU cycle), an interrupt is requested every 459 microseconds.

If a LDPCTX is executed and the PME bit in the new PCB is off, or if the performance monitoring facility is disabled via a write to the PME processor register, the microcode disables the performance monitoring facility, reads and clears the hardware counters, and updates the memory data structure for the CPU with the hardware counts.

NOTE

The hardware counters are not cleared, and the memory data structures are not updated when the performance monitoring facility is disabled via a direct write to ECR<PMF_ENABLE>.

16.2.3 Configuring the Performance Monitoring Facility

Before the performance monitoring facility is enabled, software must select the source of the event to be counted. This is accomplished first by selecting the box that reports the event, and then by selecting the event that is to be counted. The box section is made by writing to the PMF_PMUX field in the ECR processor register, as indicated by Table 16–1.

Table 16–1: Performance Monitoring Facility Box Selection

ECR<PMF_PMUX> (binary)	Source of Information
00	Ibox
01	Ebox
10	Mbox
11	Cbox

The event selection within the box is made by writing to a processor register within the box, as described in subsequent sections, and in the box chapters elsewhere in this specification.

The hardware used to implement the 16-bit counters is constructed such that the PMCTR1 counter increments only if both its selected event, and the PMCTR0 selected event are true simultaneously. As such, PMCTR1 is a strict subset of PMCTR0. As a result, some combinations of event selections will not cause PMCTR1 to be incremented. In some boxes, the event selection is specified in such a way that compatible events are automatically selected. In other boxes, the user must specify compatible events. Where they are required, compatible events are described in the sections below.

16.2.3.1 Ibox Event Selection

The Ibox reports only one event, so if the Ibox is selected, that event is also selected. The Ibox inputs to the PMCTR0 and PMCTR1 hardware counters are shown in Table 16–2

Table 16–2: Ibox Event Selection

PMCTR0 Input	PMCTR1 Input	Description; Use
VIC Access	VIC Hit	VIC hits compared to total VIC accesses; VIC hit ratio.

16.2.3.2 Ebox Event Selection

The Ebox reports several events, as selected by the PMF_EMUX field in the ECR processor register. The Ebox inputs to the PMCTR0 and PMCTR1 counters are shown in Table 16–3.

Table 16–3: Ebox Event Selection

ECR<PMF_EMUX> (binary)	PMCTR0 Input	PMCTR1 Input	Description; Use
000	Cycles	S3 Stall	S3 stalls (source queue, MD, Wn, Fbox scoreboard hit, Fbox input) compared to total cycles; S3 stalls per unit time.
001	Cycles	EM+PA queue Stall	EM latch and PA queue stalls compared to total cycles; EM+PA queue stalls per unit time.
010	Cycles	Instruction Retire	Ebox and Fbox instructions retired compared to total cycles; CPI.
011	Cycles	Total stall	Total Ebox stalls compared to total cycles; Stalls per unit time.
100	Total stall	S3 Stall	S3 stalls compared to total stalls; S3 stalls as a percentage of all stalls.
101	Total stall	EM+PA queue Stall	EM latch and PA queue stalls compared to total stalls; EM and PA queue stalls as a percentage of all stalls.

Table 16–3 (Cont.): Ebox Event Selection

ECR<PMF_EMUX> (binary)	PMCTR0 Input	PMCTR1 Input	Description; Use
111	S5 Microword event	S5 Microword event	Number of times a microinstruction whose MISC field contained INCR.PERF.COUNT reached S5. By using the patchable control store, one may count microcode events by setting the MISC field of selected microwords to this value. If this event is selected, writing to the PMFCNT processor register will increment the counters via the MISC field decode.

16.2.3.3 Mbox Event Selection

The Mbox reports several events, as selected by the PMM field in the PCCTL processor register. The Mbox inputs to the PMCTR0 and PMCTR1 counters are shown in Table 16–4.

Table 16–4: Mbox Event Selection

PCCTL<PMM> (binary)	PMCTR0 Input	PMCTR1 Input	Description; Use
000	P0/P1 I-stream TB access	P0/P1 I-stream TB hit	TB hits for P0 and P1 I-stream references compared to total TB accesses for P0 and P1 I-stream references; P0/P1 I-stream TB hit ratio.
001	P0/P1 D-stream TB access	P0/P1 D-stream TB hit	TB hits for P0 and P1 D-stream references compared to total TB accesses for P0 and P1 I-stream references; P0/P1 D-stream TB hit ratio.
010	S0 I-stream TB access	S0 I-stream TB hit	TB hits for S0 I-stream references compared to total TB accesses for S0 I-stream references; S0 I-stream TB hit ratio.
011	S0 D-stream TB access	S0 D-stream TB hit	TB hits for S0 D-stream references compared to total TB accesses for S0 D-stream references; S0 D-stream TB hit ratio.
100	I-stream Pcache access	I-stream Pcache hit	Pcache hits for I-stream references compared to total Pcache accesses I-stream references; I-stream Pcache hit ratio.
101	D-stream Pcache access	D-stream Pcache hit	Pcache hits for D-stream references compared to total Pcache accesses D-stream references; D-stream Pcache hit ratio.
111	Unaligned reads and writes	Total reads and writes	Unaligned virtual reads and writes compared to total virtual reads and writes; Unaligned references as a percentage of all references.

16.2.3.4 Cbox Event Selection

The Cbox reports several events, as selected by the PM_ACCESS_TYPE and PM_HIT_TYPE fields in the BIU_CTL processor register. The Cbox inputs to the PMCTR0 counter are shown in Table 16–5 and the Cbox inputs to the PMCTR1 counter are shown in Table 16–6.

Table 16–5: Cbox PMCTR0 Event Selection

CCTL<PM_ACCESS_TYPE> (binary)	PMCTR0 Input
000	Bcache access. PMCTR0 increments when the Bcache processes any reference from the CPU.
001	Bcache IREAD access. PMCTR0 increments when the Bcache processes an instruction-stream read request.
010	Bcache DREAD access. PMCTR0 increments when the Bcache processes a data-stream read.
011	Full LW Write access. PMCTR0 increments when the Bcache processes a LW write request.
100	Byte/Word Write access. PMCTR0 increments when the Bcache processes a byte or word write, or write unlock.
101	Any Bcache Write access. PMCTR0 increments when the Bcache processes any write, or write unlock.
110	Pcache Invalidate. PMCTR0 increments when a pInvReq is received.
110	Stall cycles. PMCTR0 increments when hold_req or not tagOk is asserted at SYS_CLK leading edge.

Table 16–6: Cbox PMCTR1 Event Selection

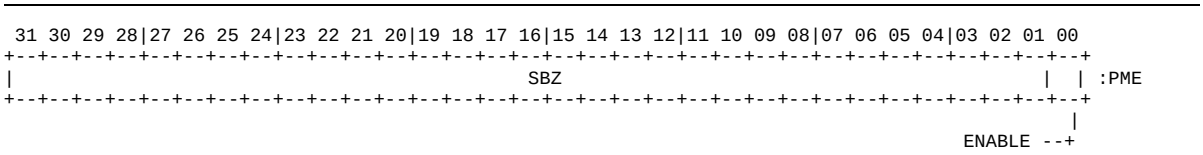
CCTL<PM_HIT_TYPE> (binary)	PMCTR1 Input
000	Bcache hit. PMCTR1 increments when a Bcache access results in any hit.
001	Bcache hit dirty. PMCTR1 increments when a Bcache access results in a dirty hit.
010	Bcache hit clean. PMCTR1 increments when a Bcache access results in a hit and the block is not dirty.
011	Bcache miss dirty. PMCTR1 increments when a Bcache access results in a miss in which both the valid and dirty bits were set.
100	Bcache hit shared. PMCTR1 increments when a Bcache access results in a hit in which both the valid and shared bits were set.
101	Stall Requests. PMCTR1 increments at SYS_CLK leading edge if a new hold_req or not tagOk is asserted.

16.2.4 Enabling and Disabling the Performance Monitoring Facility

The performance monitoring facility is enabled or disabled by setting or clearing the Performance Monitor Enable (PME) bit in the CPU. This bit may be written in one of three ways: with a write to the PME processor register, by loading a new value with a LDPCTX instruction from the PME bit in the new PCB, or by a direct write of the ECR<PMF_ENABLE> bit.

The format of the PME processor register is shown in Figure 16–3.

Figure 16–3: PME Processor Register



If PME<0> is written with a 1, the performance monitoring facility is enabled. If PME<0> is written with a 0, the performance monitoring facility is disabled. Direct writes to ECR<PMF_ENABLE> are similar to writes to PME<0>, with the exception that the hardware counters are not automatically cleared, and the memory counters are not updated on an explicit write to ECR<PMF_ENABLE>.

The CPU PME bit is also loaded by the LDPCTX instruction from PCB+92<31>.

CAUTION

The longword at offset 58 (hex) from the SCB and the correct unique CPUID value for each CPU must be initialized before the performance monitoring facility is enabled. Failure to do so will result in UNDEFINED behavior of the system.

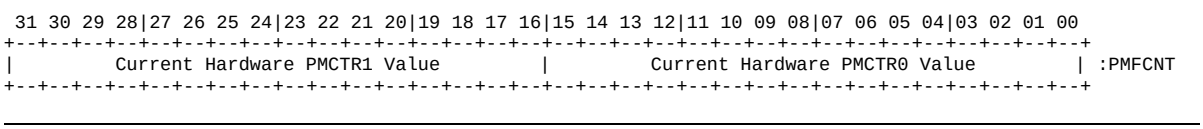
The CPU PME bit is cleared, and the performance monitoring facility is disabled, at powerup.

16.2.5 Reading and Clearing the Performance Monitoring Facility Counts

In normal operation, microcode automatically updates the memory counters by reading the current value of the hardware counters, adding these values to the memory counters, and clearing the hardware counters. This is the preferred mode of operation.

However, there may be some situations in which software wishes to directly read or clear the hardware counters. The current value of the hardware counters may be read from the PMFCNT processor register, whose format is shown in Figure 16–4.

Figure 16–4: PMFCNT Processor Register



The current value of the 16-bit hardware PMCTR1 counter is returned in PMFCNT<31:16> and the current value of the 16-bit hardware PMCTR0 counter is returned in PMFCNT<15:0>.

The two 16-bit hardware counters may be explicitly cleared by software by writing a 1 to ECR<PMF_CLEAR>. If the counters are explicitly cleared, any outstanding interrupt request is also cleared. It is strongly suggested that the hardware counters not be cleared while the performance monitoring facility is enabled.

If the performance model is configured to select the Ebox microword event (ECR<PMF_PMUX>=Ibox, ECR<PMF_EMUX>=S5 microword event, ECR<PMF_ENABLE>=1), a write of any value to the PMFCNT processor register will increment both hardware counters.

NOTE

If the 16-bit hardware counters are explicitly cleared by writing a 1 to ECR<PMF_CLEAR>, any count in these registers is lost and will not be included in the memory counters.

TEST NOTE

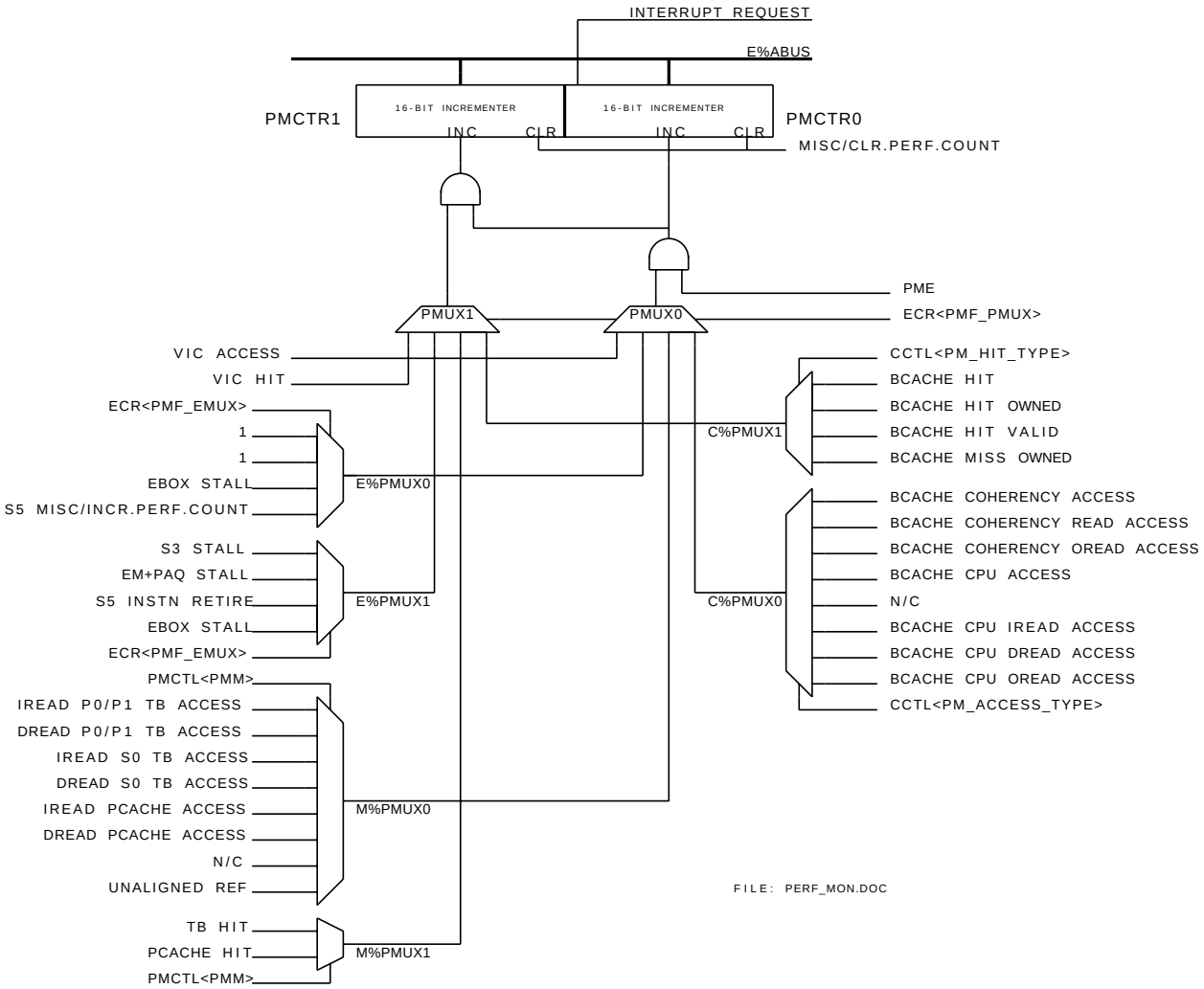
The performance monitoring facility hardware incrementers may be tested by clearing them via ECR<PMF_CLEAR>, selecting the Ebox S5 microword event, and enabling the facility. Each write to the PMFCNT processor register will then increment both hardware counters, and the result may be observed by reading the PMFCNT register. The interrupt request may be tested by incrementing the PMCTR0 hardware counter into bit<15>, which will cause an interrupt to be requested.

16.3 Hardware and Microcode Implementation of the Performance Monitoring Facility

The performance monitoring facility is implemented via both CPU chip hardware and microcode. A block diagram of the performance monitoring hardware is shown in Figure 16–5.

The lower 16 bits of the PMCTR0 and PMCTR1 performance counters are implemented as two 16-bit incrementers in the Ebox. Both incrementers have a common clear line which is driven from MISC/CLR.PERF.COUNT, and each has an increment input. The 32-bit concatenated value from the incrementers can be read onto **E%ABUS**, and the upper bit of PMCTR0 is used to generate **E_PMN%PMON**, the performance monitoring facility interrupt request.

Figure 16–5: Performance Monitoring Hardware Block Diagram



The PMCTR0 and PMCTR1 increment inputs are supplied by PMUX0 and PMUX1, through two AND gates. The PMCTR0 increment is gated by the master performance monitoring facility enable. If the facility is not enabled, PMCTR0 does not increment. The PMCTR1 increment is gated by the PMCTR0 increment, and is therefore a strict subset of PMCTR0.

The top-level selection of events is determined by ECR<PMF_PMUX>, which selects the source to PMUX0 and PMUX1. This selects the source (Ibox, Ebox, Mbox, Cbox) of the increment events. Distributed in the appropriate boxes are second-level muxes which are selected to provide the actual source of the increment events for PMCTR0 and PMCTR1.

16.3.1 Hardware Implementation

The two 16-bit hardare counters are implemented as side-by-side incrementers in the Ebox data-path (this hardware also implements the Wbus LFSR reducer that is described in the testability section of Chapter 8). The increment signals for each of the counters are driven from two 4-to-1 muxes that are selected by ECR<PMF_PMUX>, and which select the appropriate source of inputs to the incrementers.

Logic in the Ibox, Mbox, and Cbox select the appropriate values to drive the two increment signals based on processor register fields in the box. The Ebox increment signals are selected locally and provide the fourth input to the muxes. The PMCTR1 increment signal is forced to be a subset of the PMCTR0 increment signal by ANDing the raw PMCTR1 increment signal with the PMCTR0 increment signal to produce the final PMCTR1 increment signal.

Because the PMCTR1 increment is a strict subset of the PMCTR0 increment, the ultimate source of the two increment signals align them such that they are valid in the same cycle. For example, if the selcted conditions are IREAD PCACHE ACCESS and PCACHE HIT, these two signals are valid in the same cycle, and they refer to the same reference. Therefore the assertion of IREAD PCACHE ACCESS is delayed until the cycle in which PCACHE HIT is valid. In addition to this, the source of the increment signal guarantees that any events that may be retried are only recorded once. For example, a particular Pcache access causes only one increment, even if it is retried multiple times.

When the 16-bit PMCTR0 counter increments into the high-order bit, an interrupt is requested by asserting the **E_PMN%PMON_L** signal to the interrupt section. This signal is sampled by edge-sensitive logic, so the interrupt request is maintained until it is cleared by writing a 1 to the appropriate bit in the INT.SYS register, even if the performance monitoring facility hardware counters are subsequently cleared.

When the 16-bit PMCTR0 incrementer reaches its maximum value, subsequent increments of either incrementer are inhibited. In normal operation, this should not occur, but the counter may overflow if the interrupt request isn't serviced within several hundred microseconds, as would be the case if software spent an extended period of time a high IPL with the performance monitoring facility enabled.

The 32-bit concatenated value of the two 16-bit hardware incrementers can be read onto **E%ABUS** when selected by A/PERF.COUNT. This is the mechanisim by which microcode retrieves the current values of the two incrementers.

16.3.2 Microcode Interaction with the Hardware

There are several points at which the microcode interacts with the performance monitoring facility hardware. At powerup, microcode clears both of the 16-bit hardware incrementers and any potential interrupt request.

MICROCODE RESTRICTION

If the performance monitoring facility hardware incrementers are cleared in cycle 'n' via MISC/CLR.PERF.COUNT, INT.SYS<28> must be written with a 1 no earlier than cycle 'n+3' to guarantee that the interrupt request is cleared. This delay is due to latency introduced between the performance monitoring factility hardware and the interrupt section.

Microcode reads the current value of the hardware incrementers via A/PERF.COUNT as a byproduct of a read of the PMFCNT processor register, and as part of the process of updating the memory counters.

Microcode clears the hardware incrementers via MISC/CLR.PERF.COUNT when ECR<PMF_CLEAR> is written with a 1. Microcode also clears the incrementers after reading and updating the memory counters.

Microcode uses the CPUID processor register value to find the pair of quadwords that contain the performance counter values for this CPU. This value must be correctly initialized by either console firmware or software before the performance monitoring facility is enabled. The operation of the processor is UNDEFINED if CPUID is not correctly initialized.

The memory counters are updated under three circumstances: when a performance monitoring facility interrupt is serviced, when the facility is disabled via a write to the PME processor register, and when the facility is disabled by loading a new value of PME is LDPCTX. The memory updates are done in a common subroutine by disabling the facility by clearing ECR<PMF_ENABLE>, reading the current value of the hardware incrementers and then clearing them, and updating each quadword in memory with the appropriate 16-bit hardware value.

16.4 Revision History

Table 16–7: Revision History

Who	When	Description of change
Mike Uhler	12-Jan-1990	Initial release
Mike Uhler	02-Jul-1990	Update to reflect implementation

Chapter 17

Testability Micro-Architecture

17.1 Chapter Overview

This chapter describes the NVAX PLUS chip Testability Micro-Architecture.

17.2 The Testability Strategy

The NVAX PLUS chip testability strategy addresses the broad issue of providing cost-effective and thorough testing during many life cycle testing phases. The strategy specifically implements test features to support

- chip debug
- high fault coverage test at wafer probe and packaged chip test
- support "reduced probe contact" wafer probe test
- support for effective chip burn-in test

The strategy uses a combination of a variety of testability techniques and approaches that are best suited to address the specific functional elements in the chip. The cost-effective implementation is realized by the appropriate consideration of global issues, by unifying the test objectives, by sharing test resources and by exploiting features inherent in the chip. The strategy also relies on leveraging off the design verification patterns in developing production test patterns to meet the fault coverage goals.

The test features are implemented such that they have no effect on the targeted performance of the chip.

17.3 Test Micro-Architecture Overview

The NVAX Plus Test Micro-Architecture consists of two principal elements: Test Interface Unit and the Testability Features.

Test Interface Unit

The Test Interface Unit (TIU) implements a comprehensive test access strategy for NVAX Plus. It permits an efficient access to testability features implemented on the chip.

The Parallel Test port is used for accessing internal scan registers and test features which benefit from parallel access (for example, microaddress bus).

For NVAX Plus the parallel test port consists of the osc_test_sel_h pin, seven pins which are dedicated to the Parallel Port pp_data[6:0], 5 tagAdr pins (tagAdr_h[33,32,17,18,19]) are used for additional Parallel Port signals when osc_test_sel_h is asserted, and three input pins, dRAck_h[1] and dWSel_h[1:0], which receive the parallel port command when osc_test_sel_h is asserted. If osc_test_sel_h is not asserted the default parallel port mode is observe the MAB (the microcode address bus). The lower 6 bit bits of the MAB and a clock at cpu cycle frequency are driven onto pp_data[6:0] which always output test data. The remaining bits of the MAB are driven to the tagAdr pins not required for the system configuration as determined from BIU_CTL<BC_SIZE> if enabled by BIU_CTL<MAB_EN>.

dRAck_h[1] and dWSel_h[1:0] are unused input in normal operation, and tagAdr_h[33,32] is '00 for all cached addresses. tagAdr_h[17] is included in the tag comparison only if BIU_CTL[BC_SIZE] is '000, the Bcache size is 128 Kbytes, tagAdr_h[18] is included in the tag comparison only if BIU_CTL[BC_SIZE] is '000 or '001, the Bcache size is 128 Kbytes or 256 Kbytes, and tagAdr[19] is included in the comparison only if BIU_CTL[BC_SIZE] is '000 or '001 or '010, the Bcache size less than 1 Mbyte.

BIU_CTL<MAB_EN> is cleared with the reset signal, not by microcode, and must be set by software to enable additional MAB bit to be observed on tagAdr_h[33,32] and tagAdr_h[17,18,19] as determined from BIU_CTL<BC_SIZE>.

Note: **In order to verify the tag compare functionality of tagAdr[19:17] in test mode, dRAck_h[1] must be '0 (parallel port command modes 0-3) to input tagAdr_h[33,32,17,18,19].**

The Test Pads primarily facilitates micro-probing during chip debug. These pads are located at strategic nodes throughout the chip.

NVAX Plus uses the port for the Serial Rom consisting of **SROMD_H**,**SROMCLK_H**, **SROMOE_L**, and **SROMFAST_H** that allows the PCache to be loaded serially at reset for diagnostics. This feature also provides support for convenient self-test operation during the chip burn-in test.

In addition to these test ports, NVAX Plus also uses the normal system port (pins) for test access. This access consists of using the VAX instructions to manipulate a testability feature or to perform the actual tests on the chip's logic.

Table 17–1 summarizes the dedicated test pins for NVAX.

Table 17–1: NVAX Plus Test Pins

Pin Name	Pin Type	Pin Function
OSC_TEST_SEL_H	I	Select test osc and test mode
DRACK_H[1]	I	Parallel Command Port: NVAX PP_CMD_H[2]
DWSEL<1:0>	I	Parallel Command Port: NVAX PP_CMD_H[1:0]
PP_DATA_H<6:0>	B	Parallel Port: NVAX PP_DATA_H<6:0>
TAGADR_H<33:32>	B	NVAX PP_DATA_H<8:7>if enabled
TAGADR_H<19:17>	B	NVAX PP_DATA_H<9:11>if enabled
TRISTATE_L	I	Disables (tri-state) all output drivers

Table 17–1 (Cont.): NVAX Plus Test Pins

Pin Name	Pin Type	Pin Function
CONT_L	I	Continuity for testing
SROMD_H	I	Serial Data In
SROMCLK_H	O	CLK or serial data out
SROMFAST_H	I	Read SROM at reset
SROMOE_L	O	SROM output enable

17.4 Parallel Test Port

This port allows the critical chip nodes to be either controlled or monitored in parallel. The port consists of **16** test pins as follows:

— **OSC_TEST_SEL_H**: selects test mode.

— **PP_DATA_H**<6>: same function as NVAX **PP_DATA_H**<11> in test mode, outputs internal phi_4 on **PP_DATA_H**<6> if not in test mode.

— **PP_DATA_H**<5:0>: same function as NVAX **PP_DATA_H**<5:0> in test mode, outputs **MAB**<5:0> on **PP_DATA_H**<5:0> if not in test mode.

— **TAGADR_H**<33:32>: same function as NVAX **PP_DATA_H**<7:6> in test mode, outputs**MAB**<7:6> if not in test mode and BIU_CTL<MAB_EN> is set.

— **TAGADR_H**<17>: same function as NVAX **PP_DATA_H**<8> in test mode, outputs**MAB**<8> if not in test mode and BIU_CTL<MAB_EN> is set and Bcache size is greater than 128 Kbytes.

— **TAGADR_H**<18>: same function as NVAX **PP_DATA_H**<9> in test mode, outputs**MAB**<9> if not in test mode and BIU_CTL<MAB_EN> is set and Bcache size is greater than 256 Kbytes.

— **TAGADR_H**<19>: same function as NVAX **PP_DATA_H**<10> in test mode, outputs**MAB**<10> if not in test mode and BIU_CTL<MAB_EN> is set and Bcache size is greater than 512 Kbytes.

— **DRACK_H**<1>: same function as **PP_CMD_H**<2> in test mode.

— **DWSEL_H**<1:0>: same function as **PP_CMD_H**<1:0> in test mode.

The following modes of the parallel port can be selected from **DRACK_H**<1>/**DWSEL_H**<1:0> in test mode.

Table 17–2: Parallel Port Operating Modes

Command Pins		Data Pins	
DRACK_H<1>/DWSEL_H<1:0>		PP_DATA_H<6>/TAGADR_H<33:32>/TAGADR_H<17:19>/PP_DATA_H<5:0>	
PP_CMD_H<2:0>	Port Mode	PP_DATA_H<11:0>	Signals controlled/Observed
1 1 1	Observe MAB (Default)	PP_DATA_H<11>	Internal PHI_4
		PP_DATA_H<10:0>	E-Box MAB
1 1 0	Observe M-BOX	PP_DATA_H<11:9>	S5 Reference Source
		PP_DATA_H<8:4>	S5 command
		PP_DATA_H<3>	M%MME_FAULT_H
		PP_DATA_H<2>	S5 Abort
		PP_DATA_H<1>	S5 TB Miss
		PP_DATA_H<0>	S5 PCache Hit
1 0 1	Observer C-Box/M-Box	PP_DATA_H<11:7>	C-box ARB_STATE<4:0>
		PP_DATA_H<6:4>	M-box MD Destination
		PP_DATA_H<3:0>	M-box MME State
1 0 0	Observe I-Box	PP_DATA_H<11>	Internal PHI_4
		PP_DATA_H<10:7>	Undefined
		PP_DATA_H<6:0>	I-MAB
0 1 1	Enable LFSR Mode	PP_DATA_H<11:0>	Undefined
0 1 0	Undefined	PP_DATA_H<11:0>	Undefined
0 0 1	Shift ISRs	PP_DATA_H<11:3>	ISR1 (Control Store data)
		PP_DATA_H<2:0>	ISR2 (Other internal scan data)
0 0 0	Force MAB	PP_DATA_H<11:0>	Undefined

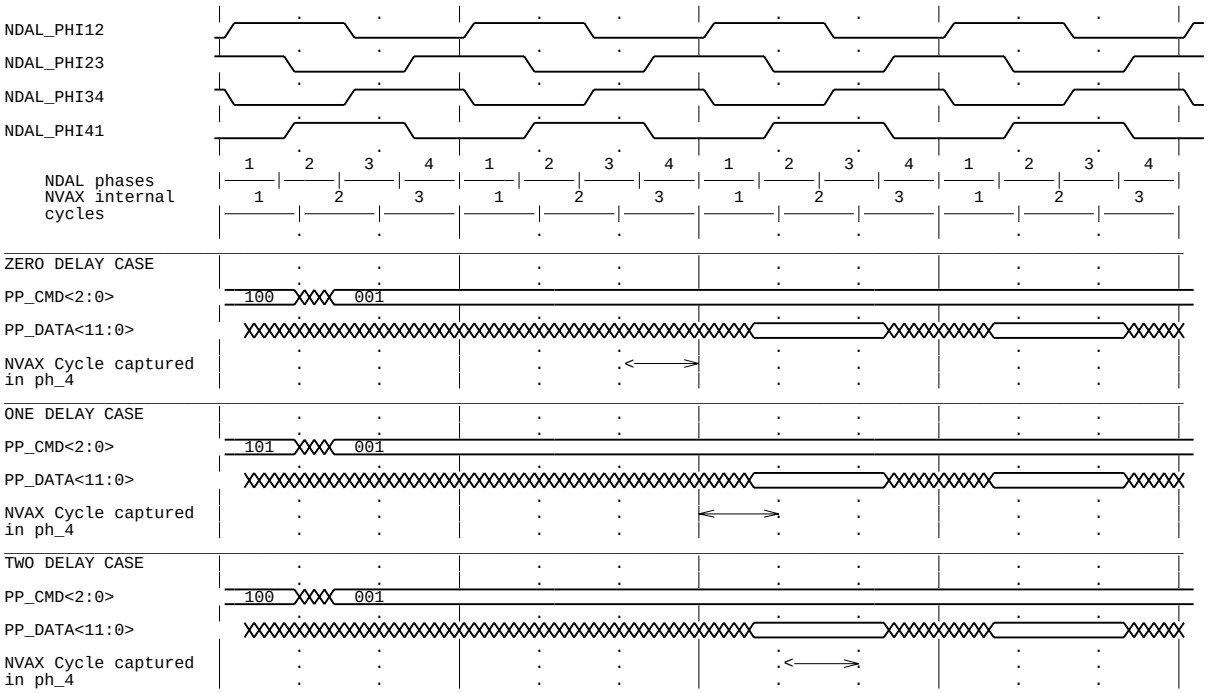
17.4.1 Parallel Port Operation

Internal Scan Register

When shifting, the ISR bits are serial to parallel converted. They change every third cycle on internal phi_2. This gives usable time with respect to sysCLKout1_h_l. The parallel port commands are captured synchronously with respect to sysCLKout1_h_l, at the falling edge. In order to give full flexability in capturing a given internal cycle, a mechanism is provided to delay the capture-and-start-shifting event by 0, 1, or 2 cycles. This delay is determined by the state of the parallel port bits dWSel[1:0] (PP_CMD< 1:0 >) immediately before entering the Shift ISR mode. ('00' corresponds to zero delay, '01' coresponds to 1 cycle delay and '10' correspond to two cycle delays.)

The NVAX design was based on NDAL clocks, which may require NVAX PLUS to set sysCLK-out1 to 3 cpu cycles for proper functioning without design changes.

Figure 17–1: Internal Scan Regiater Operation Timing



See the timing diagrams in Figure 17–1

Note that the initial packets of ISR data contain data from before the load event from the last bit on the chain. After one or two samples, this data is all valid sampled data.

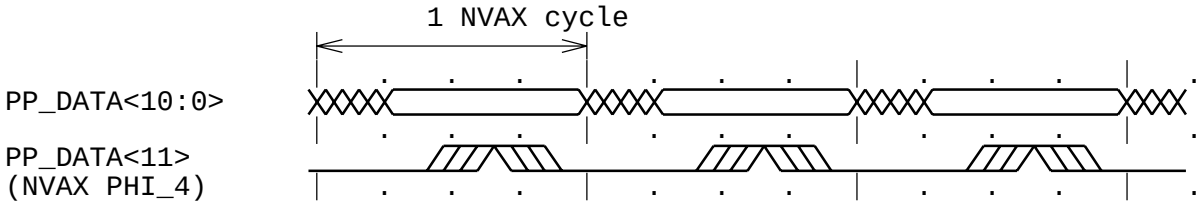
MAB Access

For full speed MAB observation, an internal clock is provided which will allow synchonous capture by a DAS in any debug environment. Figure 17–2 shows the the self-relative timing during Observe MAB mode.

17.5 Test Pads

This port consists of strategic internal nodes brought out to top level of metal in the form of 3x3 micron test pads. These pads will be accessed by probes during chip debug and wafer probe manufacturing tests. The access may primarily provide observability of these nodes, however, controllability may also be provided where appropriate. See the testability sections in box chapters for the list of nodes brought out on the top metal layer.

Figure 17–2: Self Relative Timing in Observe MAB Mode



17.6 System Port

This is simply the normal system I/O of the chip. It is identified as a test access port because of two reason:

- It is used to provide the read/write access to testability features via the VAX architecture's MFPR and MTPR instructions.
- It provides the natural resource for testing the chip via the macro-code based tests.

See the individual box chapters for the list of specific architectural features provided.

It is difficult to achieve high test coverage in the the burn-in and life-test environments due to limited test pattern bandwidth and the difficulty in synchronizing test equipment to the NVAX Plus chip. Using this serial port, burn-in and life-test programs can load the real "test program" into Pcache, where the chip can perform a self-test.

This scheme minimizes test pattern bandwidth, allows for asynchronous transmission of the serial data, provides a means to stimulate multiple chips under test which are running asynchronously, and supplies a means to achieve high test coverage.

17.7 tristate_l

NVAX Plus chip has a dedicated pin **TRISTATE_L**. When asserted low, the CPU chip tri-states output drivers on all output-only and bidirection pins.

The single pin tristate functionality is used only during testing.

17.8 cont_l

NVAX Plus chip has a dedicated pin **CONT_L**. When asserted low, NVAX Plus connects all of its pins to VSS, with the exception of the clocks.

17.9 Revision History

Table 17–3: Revision History

Who	When	Description of change
Gil Wolrich	15-Nov_1990	Release for external review.