EK-KA882-TD-PRE

VAX 8800 System Technical Description

Volume 2

FOR INTERNAL USE ONLY

digital™

EK-KA882-TD-PRE

VAX 8800 System Technical Description

Volume 2

FOR INTERNAL USE ONLY

Prepared by Educational Services
of
Digital Equipment Corporation

Preliminary Edition, July 1986

Copyright Digital Equipment Corporation 1986
All Rights Reserved

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

Printed in U.S.A.

Class A Computing Devices

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

The following are trademarks of Digital Equipment Corporation:

logo	DECwriter	RSX
logo	DIBOL	Scholar
DEC	MASSBUS	ULTRIX
DECmate	PDP	UNIBUS
DECset	P/OS	VAX
DECsystem-10	Professional	VMS
DECSYSTEM-20	Rainbow	VT
DECUS	RSTS	Work Processor

CONTENTS

SECTION 6 INSTRUCTION BOX (IBOX)

CHAPTER 1 INTRODUCTION

1.1 OVERVIEW 1-1
1.1.1 Dual-Processor Configuration 1-1
1.2 LOGIC ELEMENTS 1-2
1.2.1 Physical Implementation. 1-2
1.2.2 Instruction Buffer (IB). 1-5
1.2.2.1 Writing the IB 1-5
1.2.2.2 Reading the IB 1-5
1.2.3 IB Manager 1-6
1.2.3.1 IB Read/Write Control. 1-6
1.2.3.2 Computing Amount of IB Data Consumed 1-6
1.2.4 Decoder Logic. 1-7
1.2.4.1 Decoder RAMs 1-7
1.2.4.2 Special Address Encoder. 1-8
1.2.5 Microsequencer Logic 1-8
1.2.6 Control Store. 1-9
1.2.7 Condition Code and Macrobranch Logic 1-9
1.2.7.1 PSL CC Bits. 1-9
1.2.7.2 CPU State Flags. 1-10
1.2.8 Interrupt and Processor Register Logic 1-10
1.2.8.1 Interrupt Logic. 1-10
1.2.8.2 Processor Register Logic 1-10
1.2.9 File Address Generator 1-10
1.2.10 Gateway Control Logic. 1-11
1.2.10.1 Primary Functions. 1-11
1.3 IBOX BUSES 1-12
1.3.1 Cache Data Bus 1-12
1.3.2 IB Data Bus. 1-12
1.3.3 Cons Bidi Data Bus 1-12
1.4 IBOX RESIDENT INTERNAL PRIVILEGED REGISTERS (IPRs) 1-13
1.4.1 VAX Architecture IPRs. 1-13
1.4.2 VAX 8800-Specific IPRs 1-14
1.4.2.1 NMI Interrupt Control Register (NICTRL). . . 1-14
1.4.2.2 Interrupt Other Processor Register (INOP). . 1-15
1.5 IBOX MICROCODE VISIBLE ONLY REGISTERS. 1-16
1.5.1 Clear Interrupt Other Processor (CIOP) 1-16
1.5.2 IBox Error Register (IBER) 1-16
1.5.2.1 IBER Usage 1-16
1.5.2.2 IBER Bits <7:0>. 1-17
1.5.2.3 IBER Bits <11:08>. 1-17
1.5.3 Clear Error Register (CER) 1-19

CHAPTER 2 MICROCODE OVERVIEW AND PIPELINE CONCEPTS

2.1 CHAPTER SCOPE. 2-1
2.2 VAX 8800 MAIN CONTROL STORE OVERVIEW 2-1
2.2.1 Microcode Size and Allocation. 2-1
2.2.2 Microcode File Structure 2-1
2.2.3 Microcode Assembly 2-2
2.2.3.1 Other Loadable Binary Files. 2-3
2.2.4 Microword Format 2-3
2.2.4.1 Field Naming Convention. 2-3
2.2.4.2 Field Functionality. 2-3
2.2.5 Microcode Definition Files 2-11
2.2.5.1 Field Definition File - DEFIN.MIC. 2-11
2.2.5.2 Macrodefinition File - MACRO.MIC 2-13
2.2.6 Microcode Related Documentation. 2-15
2.3 MICROCODE PIPELINING CONCEPTS. 2-20
2.3.1 Pipelining Rationale 2-20
2.3.2 Pipelined Versus Nonpipelined Machines 2-20
2.3.2.1 Performance Factors. 2-22
2.4 VAX 8800 PIPELINE CHARACTERISTICS. 2-23
2.4.1 CPU Clock Cycle. 2-23
2.4.2 CPU Hardware Design. 2-23
2.4.3 Relationship Between Microcycles and CPU Functions. 2-25
2.4.4 IB Decode Cycle. 2-25
2.4.5 Canonical Time States. 2-27
2.4.5.1 Definition of a Canonical Time State 2-27
2.4.5.2 Overlapping Time States. 2-27
2.4.6 Time State Events. 2-29

CHAPTER 3 IBOX FUNCTIONAL DESCRIPTION

3.1 CHAPTER SCOPE. 3-1
3.2 CONTROL STORE LOGIC. 3-2
3.2.1 Control Store RAM Segments 3-2
3.2.2 Control Store RAM Addressing 3-5
3.2.3 Control Store RAM Data Latches 3-5
3.2.4 Loading the Control Store RAMs 3-6
3.2.4.1 Load Control Store Microaddress. 3-6
3.2.4.2 Write Data to Selected Address 3-7
3.3 MICROSEQUENCING. 3-9
3.3.1 Microsequencer Hardware. 3-9
3.3.1.1 Microbranch Slice (UBRS) MCAs. 3-9
3.3.1.2 Microtrap (UTRP) MCA 3-9
3.3.1.3 Microstack 3-9
3.3.2 Normal Microcode Flow. 3-11
3.3.3 IB Decoder Supplied Microaddress 3-11
3.3.4 Microbranching 3-11
3.3.4.1 Microbranch Conditions 3-13
3.3.4.2 Microbranch Latency 3-19
3.3.5 Microsubroutine Calls and Returns. 3-21
3.3.5.1 Normal Microstack Operation. 3-21
3.3.5.2 Microsubroutine Calls. 3-21

3.3.5.3 Microsubroutine Returns. 3-23
3.3.6 Microtraps 3-23
3.3.6.1 Microtrap Servicing. 3-25
3.3.6.2 Disabling Microtraps 3-30
3.3.7 Console Supplied Microaddresses. 3-30
3.4 MACROINSTRUCTION DECODING. 3-32
3.4.1 Initializing the IB (IB Flush) 3-32
3.4.1.1 Full IB Flush. 3-32
3.4.1.2 IB Flush Logic 3-34
3.4.1.3 Partial IB Flush 3-38
3.4.2 I-Stream Prefetching 3-39
3.4.2.1 General Description. 3-39
3.4.2.2 Refilling Cache. 3-39
3.4.3 Loading the IB 3-40
3.4.3.1 IB Write Control 3-40
3.4.3.2 Cache Monitor Logic. 3-42
3.4.3.3 IB Full Logic. 3-43
3.4.3.4 IB Load Example. 3-43
3.4.4 Reading The IB 3-45
3.4.4.1 Pipeline Timing. 3-45
3.4.4.2 IB Read Ports. 3-46
3.4.4.3 IB Data Aligner. 3-51
3.4.4.4 IB Data Formatter and Data Scrambler 3-60
3.4.4.5 IB Read Example. 3-66
3.4.5 IB Manager Operations. 3-73
3.4.5.1 IB Read Address Logic. 3-73
3.4.5.2 Opcode Watcher Logic 3-76
3.4.5.3 Specifier Size Logic 3-78
3.4.5.4 Checking TEMPINC <2:0> Validity. 3-81
3.4.5.5 Decoder Stall. 3-82
3.4.5.6 Modifying the IB Pointer 3-82
3.4.5.7 IB Read Address Logic Control Signals. . . . 3-85
3.4.5.8 Computing Number of IB Longwords Consumed. . 3-91
3.4.6 Instruction Decoder Operation. 3-93
3.4.6.1 Pipeline Timing Considerations 3-93
3.4.6.2 Operand Specifier Entry Point Addresses. . . 3-96
3.4.6.3 Opcode Entry Point Microaddresses. 3-104
3.4.6.4 Special Microaddresses 3-111
3.4.6.5 IBST MCA Signals Related to Instruction Decoding 3-118
3.4.6.6 Decoder RAM (DRAM) 3-121
3.4.7 Optimized Instructions 3-128
3.4.7.1 Simple Move Instructions 3-128
3.4.7.2 Simple Branch Instructions 3-129
3.5 MACROBRANCH INSTRUCTIONS 3-129
3.5.1 Branch Instruction Basics. 3-129
3.5.2 Branch Instruction Classes 3-130
3.5.3 Unconditional Branches 3-130
3.5.4 Short Conditional Branches 3-135
3.5.5 Long Conditional Branches. 3-141
3.5.6 Condition Code and Macro Branch Logic. 3-147

3.6 SPECIAL REGISTER ADDRESSING. 3-150
3.6.1 RNUM1 and RNUM2 Registers. 3-150
3.6.2 RLOG Register. 3-150
3.6.3 MDNUM Register 3-152
3.7 INTERRUPTS 3-153
3.7.1 Interrupt Requests 3-153
3.7.2 Interrupt Servicing. 3-153
3.8 CONSOLE GATEWAY CONTROL. 3-158
3.8.1 Loading Control Store and Decoder RAMs 3-158
3.8.2 Starting the Micromachine. 3-160
3.8.3 Data Transfer with Console Resident IPRs . . . 3-160
3.8.4 Breakpoint Microtrap 3-160
3.8.5 Console Data Parity Check. 3-161

FIGURES

No. Title Page

1-1 IBox Block Diagram 1-3
1-2 NMI Interrupt Control (NICTRL) Register Bit Map. 1-14
1-3 Interrupt Other Processor (INOP) Register Bit Map. 1-15
1-4 IBox Error Register (IBER) Bit Map 1-18

2-1 Microword Bit Format 2-4
2-2 Sample Microword Field Definiton - I_APORT Field. 2-11
2-3 Basic Time State Diagram - Nonpipelined CPU. . . 2-21
2-4 Basic Time State Diagram - Pipelined CPU 2-21
2-5 Basic CPU Timing 2-24
2-6 Microcycles/CPU Functions. 2-26
2-7 IB Decode Cycle. 2-28
2-8 Canonical Time States. 2-28
2-9 VAX 8800 Pipeline Time State Diagram 2-30

3-1 Control Store Logic Simplified Block Diagram . . . 3-3
3-2 Microaddress Bit Slices for Micromatch Register Loading. 3-7
3-3 Control Store RAM Load Path. 3-8
3-4 Microsequencer Logic 3-10
3-5 Normal INEXT Field Addressing. 3-12
3-6 Microbranch Condition Selection. 3-14
3-7 Microbranch Latency. 3-20
3-8 Microstack Operation 3-22
3-9 Microtrap Servicing. 3-26
3-10 Microtrap Latency 3-27
3-11 Console Supplied Microaddress. 3-31
3-12 Instruction Buffer Logic 3-33
3-13 IB Flush Logic 3-35
3-14 IB Load Logic. 3-41

3-15 I-Stream Data Entering the IB. 3-44
3-16 IB Memory Unit Contents for MOVL Example 3-47
3-17 IB Read Port Example - Part 1. 3-48
3-18 IB Read Port Example - Part 2. 3-49
3-19 IB Data Aligner Muxes. 3-51
3-20 IB Data Aligner Output Example - Part 1. 3-54
3-21 IB Data Aligner Output Example - Part 2. 3-55
3-22 Opcode Mux Sources 3-57
3-23 IB Data Formatter and Data Scrambler Logic . . . 3-61
3-24 IB Read Example. 3-69
3-25 PCNC MCA Block Diagram 3-74
3-26 Simplified IB Read Address Logic 3-83
3-27 Instruction Decode Logic 3-94
3-28 Operand Specifier Entry Address Format 3-96
3-29 Opcode Entry Address Format. 3-104
3-30 Special Microaddress Format. 3-111
3-31 Special Address Encoder Logic. 3-112
3-32 Decoder RAM Read Address Format. 3-121
3-33 Decoder RAM Output Signals 3-123
3-34 Pipeline State for a BRB Instruction 3-131
3-35 Pipeline State for a Successful BEQL Instruction. 3-140
3-36 Pipeline State for a Successful AOBLEQ Instruction. 3-145
3-37 Condition Code and Macro Branch Logic. 3-148
3-38 File Address Slice MCAs. 3-151
3-39 Interrupt Logic Simplified Block Diagram 3-155
3-40 Gateway Control Logic. 3-159

TABLES

No.	Title	Page
1-1	Microcode Features	1-9
1-2	IBox Resident IPRs	1-13
1-3	NICTRL Register Bit Descriptions	1-14
1-4	INOP Register Bit Description.	1-15
1-5	IBER Bit Descriptions.	1-10
2-1	VAX 8800 Microcode Files	2-2
2-2	Microword Field Definitions.	2-6
2-3	Macroexpression Classes.	2-15
2-4	Sample Register Transfer Macros.	2-16
2-5	Sample Cache Command Macros.	2-17
2-6	Sample CREG/IREG Macros.	2-17
2-7	Sample Microbranch Macros.	2-18
2-8	Sample Miscellaneous Macros.	2-19
2-9	Pipelined/Nonpipelined CPU Comparisons	2-22
2-10	Pipeline Time States/CPU Events.	2-31

3-1	Control Store RAM Segment Functionality.	3-2
3-2	Next Microaddress Sources.	3-5
3-3	IBRTYPE/IBRMASK Microword Field Relationship . .	3-15
3-4	Microbranch Conditions	3-16
3-5	Special Microbranch Condition Bit Usage.	3-18
3-6	Microtrap Conditions and Vectors	3-24
3-7	Machine Check Microtrap Conditions	3-25
3-8	IBST and PCNC MCA Outputs After an IB Flush. . .	3-37
3-9	IB Flush Relative State Changes - IBST and PCNC MCAs	3-38
3-10	IB Read Address/IB Read Port Source.	3-46
3-11	Data Aligner Control Signals/Data Selection. . .	3-52
3-12	IB Data Format Control Signals/Functions	3-62
3-13	IB Format Control Signals/Specifier Data Type. .	3-63
3-14	IB Data Formatter and Data Scrambler Output. . .	3-67
3-15	Floating Point Short Literal Formats	3-68
3-16	Specifier Size Logic Control Signals	3-79
3-17	Slow Spec Size <2:0> Values.	3-80
3-18	IB Pointer Source.	3-84
3-19	Operand Specifier Entry Address Bit Descriptions	3-97
3-20	Operand Data Size/Access Type Correlation. . . .	3-98
3-21	Operand Specifier Entry Address Symbolic Labels	3-100
3-22	Opcode Entry Address Bit Descriptions.	3-105
3-23	Special Microaddress Conditions.	3-114
3-24	Special Conditions Serviced During IB Decode Cycles	3-116
3-25	Decoder RAM Output Signal Descriptions	3-124
3-26	Execute Code for a BRB Instruction	3-132
3-27	Microword CTL.BRB.MEM Event Timing	3-134
3-28	IMISC Field Settings for Macrobranch Recipes . .	3-136
3-29	IMISC Field Settings for PSL Condition Code Recipes	3-143
3-30	Execute Code for a AOBLEQ Instruction.	3-146
3-31	IMISC Field Settings for State Flag Control. . .	3-149
3-32	Hardware Interrupt Priority Levels	3-154
3-33	Interrupt ID Codes/IPLs.	3-157

EXAMPLE

No.	Title	Page
2-1	Sample Field Value Assignments - I_APORT	2-12

SECTION 7 EXECUTION BOX LOGIC (EBOX)

CHAPTER 1 INTRODUCTION

1.1 GENERAL. 1-1
1.1.1 EBox Organization. 1-2
1.1.2 EBox Operators 1-4
1.1.2.1 Main ALU 1-4
1.1.2.2 Cache Data Path (CDP) and Bus Watcher/Decoder (BWD). 1-4
1.2 SLICE MODULE (SLC1/SLC0) FUNCTIONS 1-5
1.2.1 Parity Generator/Checker (PAR) 1-5
1.2.2 Register File (RGF). 1-6
1.2.3 Slow Data File (SDF) 1-6
1.2.4 Program Counter (PC) Subsystem 1-7
1.2.5 Cache Data Path (CDP). 1-7
1.2.6 Main Arithmetic Logic Unit (Main ALU). 1-8
1.2.7 Bus Watcher/Decoder (BWD). 1-8
1.3 SHIFTER MODULE (SHR) FUNCTIONS 1-9
1.3.1 Shifter (SHF). 1-9
1.3.1.1 Integer Data 1-9
1.3.1.2 Floating-Point Data. 1-9
1.3.1.3 Decimal String Data. 1-10
1.3.2 Floating-Point (FP) Support. 1-10
1.3.2.1 Priority Encoder (PEN) 1-10
1.3.2.2 Shift ALU (SALU) 1-11
1.3.2.3 Exponent ALU (XALU). 1-11
1.3.3 Multiplier/Divider (MULT). 1-11
1.4 EBOX REGISTERS 1-12
1.4.1 P0LR, P1LR, and SLR Internal Bit Formats . . . 1-13
1.4.2 VAX 8800-Specific Registers. 1-13
1.4.2.1 Machine Check Status Register (MCSTS). . . . 1-13
1.4.2.2 System Identification (SID) Register 1-14
1.4.2.3 Revision Registers (REVR1 and REVR2) 1-15
1.4.2.4 EBox Parity Error Register (EBER). 1-17

CHAPTER 2 FUNCTIONAL DESCRIPTION

2.1 GENERAL. 2-1
2.2 SLICE MODULE (SLC1/SLC0) DESCRIPTION 2-1
2.2.1 Parity Generator/Checker (PAR) 2-1
2.2.1.1 Parity Generator 2-8
2.2.1.2 Parity Checker 2-8
2.2.1.3 EBox Parity Error Register (EBER). 2-11
2.2.1.4 Carry Save Logic 2-12
2.2.2 Register File (RGF). 2-14
2.2.2.1 Floating-Point Shuffle (FPS) 2-14
2.2.2.2 Memory Data (MD) Registers 2-16
2.2.2.3 Traps and Stalls 2-16
2.2.3 Slow Data File (SDF) 2-18
2.2.3.1 Writes 2-19
2.2.3.2 Reads. 2-19
2.2.3.3 Stalls and Traps 2-19

2.2.4 Program Counter (PC) Subsystem 2-22
2.2.4.1 PC VA FA Multiplexer 2-22
2.2.4.2 Virtual Address (VA) File. 2-22
2.2.4.3 Trap Shadow Logic. 2-23
2.2.4.4 Program Counter (PC) 2-24
2.2.5 Cache Data Path (CDP). 2-31
2.2.5.1 Cache Data Buffer (CDBF) 2-31
2.2.5.2 Cache Data Store (CDS) 2-33
2.2.6 Main Arithmetic Logic Unit (Main ALU). 2-41
2.2.6.1 ALU First Half (ALF) 2-41
2.2.6.2 Main ALU Functions 2-44
2.3 SHIFTER MODULE (SHR) DESCRIPTION 2-51
2.3.1 Shifter (SHF). 2-51
2.3.1.1 Shift Count Bus. 2-55
2.3.1.2 General Function Selection 2-55
2.3.1.3 Logical Shift or Rotate. 2-58
2.3.1.4 Arithmetic Shift 2-58
2.3.1.5 Decimal String Conversion. 2-58
2.3.2 Floating-Point (FP) Support. 2-60
2.3.2.1 Priority Encoder (PEN) MCA 2-62
2.3.2.2 Shift ALU (SALU) 2-70
2.3.2.3 Exponent ALU (XALU). 2-78
2.3.3 Multiplier/Divider (MULT). 2-86
2.3.3.1 Data Interface Signals 2-86
2.3.3.2 Carry and Control Signals. 2-86
2.3.3.3 Logical and Arithmetic Functions 2-86
2.3.3.4 Multiplier Operation 2-95
2.3.3.5 Divider Operation. 2-96

FIGURES

No.	Title	Page
1-1	VAX 8800 CPU Kernel Block Diagram.	1-1
1-2	Execution Unit (EBox) Block Diagram.	1-3
1-3	Machine Check Status Register (MCSTS).	1-13
1-4	System Identification (SID) Register	1-14
1-5	Revision Register 1 (REVR1).	1-15
1-6	Revision Register 2 (REVR2).	1-16
2-1	Slice Module (SLC1/SLC0) Block Diagram	2-2
2-2	Parity Generator/Checker (PAR) Block Diagram . . .	2-3
2-3	EBox Parity Error Register (EBER).	2-11
2-4	Register File (RGF) Block Diagram.	2-15
2-5	Slow Data File (SDF) Block Diagram	2-19
2-6	Program Counter (PC) Subsystem Block Diagram . .	2-23
2-7	Cache Data Path (CDP) Block Diagram.	2-32
2-8	Main Arithmetic Logic Unit (Main ALU) Block Diagram.	2-43
2-9	Shifter Module (SHR) Block Diagram	2-52
2-10	Shift MCA (SHFT) Logic and Gating Block Diagram.	2-53

2-11	Shift Control MCA (SHC) Block Diagram.	2-54
2-12	Shift Count Bus Signal and Gating Block Diagram.	2-55
2-13	VAX-11 Floating-Point Formats.	2-61
2-14	Priority Encoder (PEN) Block Diagram	2-64
2-15	INMUX Mapping of the BPORT Input Data.	2-65
2-16	Shift ALU (SALU) Block Diagram	2-73
2-17	Exponent ALU (XALU) Block Diagram.	2-79
2-18	Multiplier/Divider (MULT) Block Diagram.	2-88

TABLES

No.	Title	Page
1-1	Privileged IPRs Maintained by the EBox	1-12
1-2	P0LR, P1LR, and SLR Internal Formats	1-13
1-3	Machine Check Status Register (MCSTS) Bit Descriptions	1-14
1-4	System Identification (SID) Register Bit Field Descriptions	1-15
1-5	Revision Register 1 (REVR1) Bit Field Descriptions	1-16
1-6	Revision Register 2 (REVR2) Bit Field Descriptions	1-17
2-1	Parity Generator/Checker (PAR) Signal Descriptions	2-3
2-2	A-Side Port Control.	2-10
2-3	B-Side Port Control.	2-10
2-4	Keepgoing Conditions for A CD PAR<3,1>	2-10
2-5	E_SHFT<4:0> Control of the EBox Parity Error Register (EBER).	2-12
2-6	E_ALUCI<1:0> Control of the ALU Carry-In Source	2-13
2-7	Register File (RGF) Address Allocation	2-16
2-8	Register File (RGF) Signal Descriptions.	2-17
2-9	Slow Data File (SDF) Signal Descriptions	2-20
2-10	Program Counter (PC) Subsystem Signal Descriptions	2-27
2-11	E_VAWRT and I_APORT<7:6> Control of PC VA FA Multiplexer Input Selection.	2-30
2-12	PC Multiplexer Input Selection PC Multiplexer Select Signals	2-30
2-13	Cache Data Buffer (CDBF) Signal Description. . .	2-34
2-14	Cache Data Store (CDS) Signal Description. . . .	2-38
2-15	CDS Output Multiplexer Control for Each Slice. .	2-40
2-16	ALU First Half (ALF) Signal Descriptions	2-45
2-17	ALU Second Half (ALS) Signal Descriptions. . . .	2-48
2-18	A-Side Select (ASEL) Input Control Signals . . .	2-49
2-19	B-Side Select (BSEL) Input Control Signals . . .	2-49
2-20	Keepgoing/Stall Conditions	2-49
2-21	EALU<5:0> Field Control of the Main ALU.	2-50

2-22 Shift Count Bus Signals and Source 2-55
2-23 ESHFT<4:0> Field Selection of Shifter (SHF) MCA Logic Functions. 2-56
2-24 ESHFTSEL Selection of a Result Output to the BP Bus. 2-57
2-25 EFPFORMAT<1:0> Field Control of Decimal String Data Conversions. 2-59
2-26 EPEFUNC Field Selection of PEN Functions 2-65
2-27 Priority Encoder (PE) Results Passed to the Shift Count Bus 2-67
2-28 Increment Multiplexer Data (INCD) Selection to the Incrementer (INCR). 2-68
2-29 Sticky Bit Logic Input and Test Selection. . . . 2-68
2-30 G<1:0> Guard Bit Input Selection 2-69
2-31 Round Bit R<1:0> Input Selection 2-69
2-32 SALU and XALU Control Signals from the Microcode 2-71
2-33 ESXALUFN Field Control of the SALU Functions . . 2-72
2-34 Resulting Sign of the Fraction 2-74
2-35 SALU Selection of the APORT and BPORT Inputs . . 2-74
2-36 A-Latched Condition Code Inputs to the Branch Multiplexer. 2-76
2-37 Microbranch Condition Code Description 2-77
2-38 ESXALUFN<5:3) Control of the General XALU Functions. 2-80
2-39 XALU Functions with E_SXALUFN<5:3> Equal to 000 2-81
2-40 XALU Functions with ESXALUFN<5:3> Not Equal to 000 2-81
2-41 XALU Condition Code (XALUCC) Tests 2-82
2-42 M1 Inputs Passed to M3 2-83
2-43 M3 Inputs Passed to the Adder B-side 2-84
2-44 M2 Outputs to M6 or the XREG 2-85
2-45 M2 Data Passed to the BP Bus by M6 2-85
2-46 Multiplier/Divider (MULT) Control Signals from the Microcode 2-90
2-47 E_MULDIV Field Control of the MULT Functions . . 2-91
2-48 MULT Logic Signal Port Function Description. . . 2-93

SECTION 8 CACHE BOX LOGIC (CBOX)

CHAPTER 1 INTRODUCTION

1.1 CACHE BOX SYSTEM DESCRIPTION 1-1
1.2 CBOX OPERATION 1-5
1.2.1 CBox Cycles. 1-6
1.2.1.1 Quiescent State. 1-7
1.2.1.2 Read Cycle 1-8
1.2.1.3 Write Cycles 1-8
1.2.1.4 The PIBA 1-9

1.2.1.5 TB Cycles. 1-9
1.2.1.6 Refill Operation 1-10
1.2.1.7 Invalidate Cycle 1-10
1.2.2 CBox Stalls. 1-11

CHAPTER 2 FUNCTIONAL DESCRIPTION

2.1 CBOX SUBSYSTEMS DESCRIPTION. 2-1
2.1.1 Translation Buffer 2-1
2.1.1.1 VA Latch 2-4
2.1.1.2 TB RAM 2-5
2.1.1.3 TB Match MCA 2-8
2.1.1.4 TB RAM Bypass. 2-10
2.1.1.5 PA Latch 2-14
2.2 CBOX SUBSYSTEMS DESCRIPTION. 2-19
2.2.1 Cache. 2-19
2.2.1.1 Cache Data Path Logic. 2-19
2.2.1.2 Cache Tag MCA. 2-25
2.2.1.3 Cache Match MCA. 2-25
2.2.1.4 Cache Control MCA. 2-27
2.2.1.5 MD Number MCA. 2-28
2.3 CBOX SUBSYSTEMS DESCRIPTION. 2-29
2.3.1 NMI Interface. 2-29
2.3.1.1 NMI Address/Data Slices. 2-31
2.3.1.2 NMI Out Control. 2-36
2.3.1.3 NMI In Control 2-42
2.3.1.4 NMI Arbitration/Acknowledgment 2-47
2.3.1.5 CBox NMI Registers 2-52

FIGURES

No. Title Page

1-1 CBox -- Block Diagram. 1-3
1-2 CBox Cycle Timing. 1-5

2-1 Translation Buffer -- Block Diagram. 2-2
2-2 Virtual Address Fields 2-4
2-3 TB -- Write Sequence Diagram 2-6
2-4 TB Match MCA -- Simplified Block Diagram 2-9
2-5 PABH MCA -- Simplified Block Diagram 2-12
2-6 PABL MCA -- Simplified Block Diagram 2-13
2-7 PA Latch -- Logic Diagram. 2-15
2-8 PA Latch Bit Routing -- Refill Cycle 2-17
2-9 PA Latch Bit Routing -- VA Reference 2-18
2-10 Cache -- Block Diagram 2-20
2-11 NMI Interface -- Block Diagram 2-30
2-12 NMI Address/Data Slices MCA -- Simplified Block Diagram. 2-32
2-13 NMI Out Control -- Simplified Block Diagram. . . 2-37
2-14 Control Store Microword Format Diagram 2-41
2-15 NMI In Control -- Block Diagram. 2-42

2-16 NMI Arbitration/Acknowledgment Control -- Simplified Block Diagram 2-49
2-17 Timeout -- Flow Diagram. 2-51
2-18 CBox NMI Registers -- Location Diagram 2-52
2-19 Cache Register -- Bit Format 2-53
2-20 Cache Error Register Byte 2 -- Bit Format. . . . 2-55
2-21 Cache Error Register Byte 1 -- Bit Format. . . . 2-56
2-22 Cache Error Register Byte 0 -- Bit Format. . . . 2-57
2-23 NMI Fault/Status Register Byte 1 -- Bit Format 2-58
2-24 NMI Fault/Status Register Byte 0 -- Bit Format 2-60
2-25 NMI Error Address Register -- Bit Format 2-61
2-26 NMI Silo Byte 2 -- Bit Format. 2-63
2-27 NMI Silo Byte 1 -- Bit Format. 2-65
2-28 NMI Silo Byte 0 -- Bit Format. 2-66
2-29 Cache TAG Initialization Register -- Bit Format 2-67
2-30 Diagnostic ID Register -- Bit Format 2-68
2-33 Diagnostic Control Register -- Bit Format. . . . 2-69

TABLES

No. Title Page

1-1 CBox Cycles. 1-7

2-1 PROTection Field <03:00> Coding and Access Allowed. 2-7
2-2 TB Match MCA Operation Coding. 2-8
2-3 Cache Register - Bit Descriptions. 2-53
2-4 Cache Error Register - Byte 2 Bit Descriptions 2-55
2-5 Cache Error Register - Byte 1 Bit Descriptions 2-56
2-6 Cache Error Register - Byte 0 Bit Descriptions 2-57
2-7 NMI Fault/Status Register Byte 1 - Bit Descriptions 2-58
2-8 NMI Fault/Status Register Byte 0 - Bit Descriptions 2-60
2-9 NMI Error Address Register -- Bit Descriptions 2-61
2-10 NMI Silo Byte 2 -- Bit Descriptions. 2-64
2-11 NMI Silo Byte 1 -- Bit Descriptions. 2-65
2-12 NMI Silo Byte 0 -- Bit Descriptions. 2-66
2-13 Cache TAG Initialization Register -- Bit Descriptions 2-67
2-14 Diagnostic ID Register -- Bit Descriptions . . . 2-68
2-15 Diagnostic Control Register -- Bit Descriptions 2-69

SECTION 1 INTRODUCTION AND SYSTEM OVERVIEW

CHAPTER 1 INTRODUCTION

1.1 MANUAL SCOPE 1-1
1.2 MANUAL ORGANIZATION. 1-1
1.3 RELATED DOCUMENTATION. 1-2
1.4 SYSTEM DESCRIPTION 1-3
1.5 PHYSICAL DESCRIPTION 1-3
1.6 FUNCTIONAL DESCRIPTION 1-7
1.6.1 Console Subsystem. 1-8
1.6.2 Central Processing Unit. 1-10
1.6.2.1 Instruction Box. 1-13
1.6.2.2 Execution Box. 1-16
1.6.2.3 Cache Box. 1-20
1.6.3 Clock Module 1-21
1.6.4 Memory (MBox). 1-23
1.6.4.1 Memory Control Logic 1-24
1.6.4.2 MAR4 Memory Array. 1-25
1.6.5 System Buses 1-25
1.6.5.1 VAX 8800 Memory Interconnect (NMI) 1-25
1.6.5.2 VAX Bus Interconnect (VAXBI) 1-27
1.6.5.3 Visibility Bus (VBus). 1-29
1.6.6 VAX Bus Interconnect and I/O Adapters. 1-30
1.6.7 Power System Complex 1-33
1.6.7.1 876 Power Controller 1-34
1.6.7.2 NBox Port Conditioner. 1-34
1.6.7.3 Module Power Supplies. 1-35
1.6.7.4 Environmental Monitoring Module. 1-35
1.6.7.5 Battery Backup Unit. 1-35

CHAPTER 2 SYSTEM CONTROL

2.1 GENERAL. 2-1
2.2 SYSTEM CONTROL 2-1
2.2.1 Software 2-1
2.2.2 Hardware 2-1
2.3 CONSOLE SOFTWARE COMPONENTS. 2-4
2.3.1 Control Program. 2-4
2.3.1.1 Special Control Program Features 2-4
2.3.1.2 Multiple Command Streams 2-5
2.3.2 File Transfer Program. 2-6
2.3.3 Logical Block Server Program 2-6
2.3.4 Real-Time Interface Driver 2-6
2.4 CONSOLE SUPPORT MICROCODE (CSM). 2-6
2.4.1 Console Support Microcode Structure. 2-6
2.4.2 CSM Data Transfers/Protocol. 2-7
2.4.3 Console Support Microcode Entry Points 2-7

2.5 OPERATIONAL MODES. 2-8
2.5.1 Console Mode 2-8
2.5.2 Program Mode 2-9
2.6 OPERATOR/CONSOLE INTERACTION 2-9
2.6.1 Console State Bits 2-9
2.6.1.1 Command Validity 2-10
2.6.1.2 Saving Console State 2-10
2.6.2 Console Commands 2-10
2.6.2.1 Executing Console Commands 2-12
2.6.3 Console/Operator Display 2-13
2.6.3.1 Local Display During Remote Operation. . . . 2-13
2.6.3.2 Console Command Language Display Prompts . . 2-15
2.6.4 System Logfile 2-15
2.6.4.1 Displaying the Logfile 2-15
2.6.4.2 Logical Terminals/Logfile Integrity. 2-15
2.6.4.3 Saving the Logfile 2-16
2.7 POWER-UP/DOWN SEQUENCING 2-16
2.7.1 Power On 2-16
2.7.2 Power Fail 2-17
2.7.3 Powerdown. 2-17
2.7.4 Warm Restart 2-17

CHAPTER 3 SYSTEM OPERATION

3.1 INTRODUCTION 3-1
3.2 MICROCODE. 3-1
3.2.1 Microcode Characteristics. 3-1
3.2.1.1 Functionality. 3-1
3.2.1.2 Operation. 3-1
3.2.1.3 Structure. 3-1
3.2.2 Microcode Control. 3-2
3.2.2.1 Interrupt and Processor Register (INPR). . . . 3-2
3.2.2.2 Condition Code and Branch (CCBR) 3-3
3.2.2.3 Gateway Control (GWYC) 3-3
3.2.2.4 Microtrap (UTRP) 3-3
3.2.2.5 Microbranch Slices (UBRS). 3-3
3.2.3 Pipelining 3-3
3.2.4 Microtraps 3-4
3.2.5 Micromatch 3-5
3.2.5.1 Stop on Match. 3-5
3.2.5.2 Trap on Match. 3-6
3.3 MEMORY ADDRESSING AND READ/WRITE OPERATIONS. . . . 3-6
3.3.1 Virtual Addresses. 3-6
3.3.1.1 Layout 3-6
3.3.1.2 Format 3-6
3.3.2 Physical Addresses 3-9
3.3.3 Address Translation. 3-10
3.3.3.1 Page Table Entry 3-10
3.3.4 Cache Operation. 3-15
3.3.4.1 Translation Buffer 3-15
3.3.4.2 Cache. 3-16
3.3.4.3 NMI Interface. 3-17

3.3.5 Read/Write Operations. 3-17
3.3.6 Device Address Selection 3-19
3.3.6.1 Transaction Significant Address Bits 3-19
3.4 INTERRUPTS AND EXCEPTIONS. 3-20
3.4.1 Servicing. 3-20
3.4.1.1 NMI Interrupt Enable Register. 3-20
3.4.1.2 Types of Interrupts. 3-20
3.4.1.3 Servicing the Interrupt. 3-20
3.4.2 Priority Levels. 3-20
3.4.3 System Control Block (SCB) Format. 3-21
3.4.3.1 SCB Pagination 3-23
3.4.3.2 Offsetable Devices 3-23
3.4.3.3 VAXBI Node Direct Connected Devices. 3-23
3.4.3.4 SCB Format 3-24
3.4.4 Machine Check Exception. 3-25
3.4.4.1 Types of Exceptions. 3-25

CHAPTER 4 DIAGNOSTIC AND MAINTENANCE AIDS

4.1 INTRODUCTION 4-1
4.2 GENERAL. 4-1
4.3 DIAGNOSTICS. 4-3
4.3.1 Console Selftest 4-3
4.3.2 Microdiagnostics 4-3
4.3.2.1 CSM Commands 4-4
4.3.2.2 Status and Error Information 4-6
4.3.2.3 Micromonitor Error Messages. 4-8
4.3.3 Macrodiagnostics 4-8
4.3.4 Customer Runnable Diagnostics. 4-9
4.3.4.1 Auto-Test Mode 4-10
4.3.4.2 Menu Mode. 4-10
4.3.5 Remote Diagnostics 4-10
4.4 POWER/ENVIRONMENTAL SYSTEM 4-10
4.4.1 Module Placement Verification. 4-10
4.4.1.1 Module Key Test. 4-10
4.4.1.2 Module Placement 4-12
4.4.2 Power Monitoring/Error Reporting 4-12
4.4.2.1 Default Mode Error Reporting 4-13
4.4.2.2 Operational Error Reporting. 4-13
4.4.3 Voltage Margining. 4-14
4.5 MAINTENANCE AIDS 4-16
4.5.1 Machine Check Logout Stack 4-16

FIGURES

No.	Title	Page
1-1	VAX 8800 System Major Component Locations (Rear View)	1-4
1-2	Cardcage Module Layout (Front View)	1-6
1-3	Simplified Block Diagram of the VAX 8800 System	1-8
1-4	Simplified Block Diagram of the Console Subsystem	1-10
1-5	Simplified Block Diagram of Single CPU	1-12
1-6	Simplified Block Diagram of the CPU IBox	1-13
1-7	Simplified Block Diagram of the CPU Execution Box	1-17
1-8	Simplified Block Diagram of the CBox	1-21
1-9	Simplified Block Diagram of the Clock Module	1-22
1-10	Simplified Block Diagram of the MBox	1-24
1-11	Simplified Block Diagram of the VAX 8800 Memory Interconnect	1-26
1-12	Simplified Diagram of VAX Bus Interconnect (Maximum Configuration)	1-28
1-13	I/O Interconnect and Adapters	1-31
1-14	NMI-to-VAXBI Adapter	1-32
1-15	Simplified Block Diagram of the Power System Complex	1-34
2-1	VAX 8800 System Hardware and Software Control Components	2-2
2-2	Console Operational Modes	2-8
2-3	Local/Remote Display Character Flow	2-14
3-1	Simplified Pipelining	3-4
3-2	Virtual Address Space Layout	3-7
3-3	Virtual Address Format	3-8
3-4	Physical Address Space Layout	3-9
3-5	MAP Enable Register Bit Configuration	3-10
3-6	Page Table Entry Bit Configuration	3-10
3-7	System Space Virtual-to-Physical Address Translation	3-12
3-8	Process Space (P0 Region) Virtual-to-Physical Address Translation	3-13
3-9	Process Space (P1 Region) Virtual-to-Physical Address Translation	3-14
3-10	CBox Functional Components	3-15
3-11	NMI Address Selection	3-18
3-12	NMI Address Bit Significance	3-19
3-13	NBI I/O Adapter SCB Vector Offset Format	3-24

4-1	Bottom-Up Testing.	4-2
4-2	Module Keying Test Simplified Block Diagram.	4-11
4-3	Module Key Test Connections.	4-12
4-4	Margin Enable and Margin Hi Lo Registers	4-15
4-5	Machine Check Logout Stack	4-17
4-6	CBox Error Register.	4-18
4-7	IBox Error Register.	4-18
4-8	EBox Error Register.	4-19
4-9	NMI Interrupt Control Register	4-19
4-10	NMI Fault Summary.	4-20
4-11	NMI Silo Data.	4-21
4-12	NMI Error Address Register	4-21
4-13	Cache ON Register.	4-22
4-14	Machine Check Status	4-22
4-15	Revision 1/2 Registers	4-22

TABLES

No.	Title	Page
1-1	Technical Description Manual Organization.	1-2
1-2	Related Documentation.	1-3
1-3	VAX 8800 System Physical Characteristics	1-5
1-4	Cabinet Module Identification.	1-7
1-5	Power Supply Identification.	1-7
1-6	VAX 8800 Processor Functional Units/Data Bus Descriptions	1-11
1-7	System Clocks.	1-23
1-8	MCL Command Operations	1-24
1-9	NMI Function Descriptions.	1-27
1-10	VAXBI Function Descriptions.	1-29
1-11	Optional VAX Bus Interconnect Adapters	1-30
1-12	NBIA Registers	1-33
1-13	NBIB Registers	1-33
1-14	NBox Modules	1-35
2-1	Hardware and Software Component Description.	2-3
2-2	Major Sections of CSM Code	2-7
2-3	Console Support Microcode Entry Points	2-8
2-4	Bit Examples	2-9
2-5	Console Command Overview	2-11
2-6	Console Command Language Prompts	2-15
2-7	Module Power Supply Turn-On Sequence	2-17
3-1	VAX 8800 System Microtraps	3-5
3-2	Page Table Entry Bit Description	3-11
3-4	Translation Buffer Field Description	3-15
3-4	Hardware Interrupt Priority Level Assignments.	3-21
3-5	System Control Block Page 0 (000--1FF)	3-22
3-6	Machine Check Exception Examples	2-25

4-1 TEMP Register Addresses for Use with Diagnostic CSM 4-5
4-2 Microcode Error Register Addresses 4-6
4-3 Macrodiagnostic Tests. 4-9

EXAMPLES

No. Title Page

4-1 Sample Microdiagnostic Display Output. 4-7
4-2 Sample Microdiagnostic Error Display 4-7
4-3 EMM Warning Message. 4-14

SECTION 2 SYSTEM BUS SUMMARY

CHAPTER 1 MEMORY INTERCONNECT (NMI)

1.1 INTRODUCTION 1-1
1.2 BASIC FUNCTIONS 1-4
1.3 NMI SIGNALS AND TIMING 1-5
1.4 NMI ADDRESS SPACE. 1-18
1.5 READ/WRITE TRANSACTIONS. 1-25
1.6 INTERLOCKED OPERATIONS 1-32
1.7 BUS ARBITRATION. 1-33
1.8 INTERRUPTS 1-39
1.8.1 NMI Interrupt Priority Levels. 1-39
1.8.2 Device Interrupts. 1-40
1.8.3 NMI Faults 1-41
1.9 NMI ERRORS 1-42

CHAPTER 2 VAX BUS INTERCONNECT (VAXBI)

2.1 INTRODUCTION 2-1
2.2 BASIC FUNCTIONS. 2-4
2.3 VAXBI SIGNALS AND TIMING 2-5
2.4 VAXBI ADDRESS SPACE. 2-11
2.4.1 Memory Address Space 2-11
2.4.2 I/O Address Space. 2-11
2.4.3 Address Selection. 2-12
2.5 BASIC VAXBI TRANSACTION FORMAT 2-18
2.5.1 Command/Address Cycle. 2-18
2.5.2 Embedded Arbitration Cycle 2-18
2.5.3 Data Cycles. 2-19
2.5.4 Bus Parity 2-19
2.6 READ/WRITE TRANSACTIONS. 2-20
2.6.1 Write Data Cycles. 2-20
2.6.2 Read Data Cycles 2-20
2.6.3 Nonexistent Addresses. 2-23

2.6.4	Stalls	2-23
2.6.5	Retries	2-24
2.7	BROADCAST TRANSACTIONS	2-24
2.8	INVALIDATE TRANSACTIONS	2-26
2.9	INTERRUPT OPERATION (INTR, IDENT, AND IPINTR TRANSACTIONS	2-28
2.9.1	Interrupt (INTR) Transactions	2-29
2.9.2	Identify (IDENT) Transactions	2-31
2.9.3	Interprocessor Interrupt (IPINTR) Transaction	2-34
2.10	STOP TRANSACTIONS	2-36
2.11	BUS ARBITRATION AND CONTROL	2-38
2.11.1	Bus Requests	2-38
2.11.2	Arbitration Modes	2-39
2.11.3	Arbitration Control	2-39
2.11.4	Extending a Transaction	2-41
2.11.5	Special Mode Functions	2-41
2.12	VAXBI ERRORS	2-42
2.12.1	Parity Checking	2-42
2.12.2	Transmit Check Error Detection	2-43
2.12.3	Protocol Checking	2-44

CHAPTER 3 VISIBILITY BUS (VBUS)

3.1	INTRODUCTION	3-1
3.2	BASIC FUNCTIONS	3-1
3.3	VBUS SIGNALS	3-1
3.4	VBUS REGISTERS	3-4
3.5	MODULE VBUS CHANNEL CIRCUITRY	3-7
3.5.1	Minimum Configuration	3-7
3.5.2	Expanded Configuration	3-7
3.6	VBUS ADDRESS/DATA SUMMARY	3-10
3.7	VBUS CONSOLE COMMANDS	3-10

FIGURES

No.	Title	Page
1-1	Memory Interconnect (NMI)	1-2
1-2	Basic NMI Timing	1-5
1-3	NMI Signals	1-6
1-4	NMI Address Space	1-20
1-5	NMI Address Bits	1-23
1-6	NMI Address Selection	1-24
1-7	NMI Write Transaction	1-26
1-8	NMI Read Transaction	1-27
1-9	NMI Write Transaction Types	1-29

1-10	NMI Read Transaction Types	1-30
1-11	Basic NMI Arbitration Line Timing	1-33
1-12	NMI Arbitrator Operation	1-36
1-13	Detailed NMI Arbitration Line Timing (Typical)	1-37
1-14	MEMORY BUSY Timing	1-38
1-15	Fault Signal Timing	1-42
2-1	VAX Bus Interconnect (VAXBI)	2-2
2-2	VAXBI Signals	2-6
2-3	Basic VAXBI Timing	2-10
2-4	VAXBI Address Space	2-13
2-5	VAXBI Node Register Space	2-14
2-6	VAXBI Required Registers	2-15
2-7	BIIC-Specific Device Registers	2-16
2-8	VAXBI Read/Write Address Bits	2-17
2-9	Basic VAXBI Transaction Format	2-18
2-10	VAXBI Write Transaction (Octaword Length)	2-21
2-11	VAXBI Read Transaction (Octaword Length)	2-22
2-12	VAXBI Broadcast (BDCST) Transaction (Octaword Length)	2-25
2-13	VAXBI Invalidate (INVAL) Transaction	2-27
2-14	VAXBI Interrupt (INTR) Transaction	2-30
2-15	VAXBI Identify (IDENT) Transaction	2-33
2-16	VAXBI Interprocessor Interrupt (IPINTR) Transaction	2-35
2-17	VAXBI STOP Transaction	2-37
2-18	Bus Arbitration Request Lines	2-38
2-19	Arbitration State Diagram	2-40
2-20	VAXBI Arbitration (Example)	2-42
3-1	Visibility Bus (VBus) and VBus Control (on CLK Module)	3-2
3-2	VBus Control Register	3-5
3-3	VBus Access Register	3-6
3-4	VBus Channel in CPU Module (Minimum Configuration)	3-8
3-5	VBus Channel in CPU Module (Expanded Configuration)	3-9

TABLES

No.	Title	Page
1-1	Glossary of NMI Terms	1-3
1-2	NMI Signal Descriptions	1-10
1-3	I/O Registers in NBI and Memory Controller	1-21
1-4	NMI Interrupt Priority Levels (IPLs)	1-39
1-5	NMI Errors	1-43

2-1 Glossary of VAXBI Terms. 2-3
2-2 VAXBI Signal Descriptions. 2-7

3-1 VBUS Signal Descriptions 3-3
3-2 VBus Control Register Bit Descriptions 3-5
3-3 VBus Directory (Excerpt) 3-11

SECTION 3 CONSOLE SUBSYSTEM

CHAPTER 1 INTRODUCTION

1.1 GENERAL. 1-1
1.2 RELATED DOCUMENTATION AND REFERENCES 1-1
1.3 FUNCTION AND PURPOSE 1-2
1.4 SUBSYSTEM COMPONENTS 1-2
1.5 CONSOLE/VAX 8800 INTERACTION 1-4
1.5.1 Power-Up Mode. 1-4
1.5.2 Console I/O. 1-4
1.5.3 VAX 8800 State Description 1-6
1.5.3.1 Power Off. 1-6
1.5.3.2 Clock Stopped/WCS Invalid. 1-6
1.5.3.3 Clock Running/WCS Invalid. 1-6
1.5.3.4 Clock Running/WCS Valid. 1-6
1.5.4 Console State Description. 1-7
1.6 CONSOLE SOFTWARE COMPONENTS. 1-9
1.6.1 Control Program. 1-9
1.6.2 Logical Block Server Program 1-9
1.6.3 Real-Time Interface Driver 1-9
1.7 CONSOLE/VAX 8800 POWER SEQUENCE. 1-10
1.7.1 Powerup (Refer to Figure 1-3). 1-10
1.7.2 EMM/Console Initialize (Refer to Figure 1-4). 1-12
1.7.3 Restart/Boot/Halt (Refer to Figure 1-5). . . . 1-17
1.7.4 Power Fail (Refer to Figure 1-6) 1-19
1.7.5 Powerdown (Refer to Figure 1-7). 1-21

CHAPTER 2 FUNCTION DESCRIPTION

2.1 GENERAL. 2-1
2.2 REAL-TIME INTERFACE (RTI). 2-1
2.2.1 Programmable Peripheral Interface (PPI). 2-3
2.2.1.1 Port A 2-3
2.2.1.2 Port B 2-3
2.2.1.3 Port C 2-6
2.2.1.4 PPI Control. 2-8
2.2.2 Serial Line Port 2-10
2.2.2.1 ECPI Registers 2-10
2.2.2.2 Data Transfer and Status Registers 2-14

2.3 CONSOLE INTERFACE. 2-16
2.3.1 Buffer Translate and Synchronize 2-16
2.3.2 Console Address Decode 2-16
2.3.3 Console Sequencer (CSEQ MCA) 2-17
2.3.4 Terminal Register/Interval Clock (TRIC) MCA. . 2-17
2.3.4.1 Program Mode 2-19
2.3.4.2 Console Mode 2-20
2.3.5 Data Output Mux. 2-20
2.3.6 Control Registers. 2-21
2.3.7 Visibility Bus Control 2-22
2.3.8 Console Interrupt Generation 2-23
2.3.9 Power Status 2-23
2.4 CONSOLE/VAX 8800 INTERACTION 2-24
2.4.1 Initialization 2-24
2.4.1.1 Turn ON and Monitor System Power/Reset EMM. 2-24
2.4.1.2 Console Poweron. 2-27
2.4.1.3 Load and Run Console Power-Up Software . . . 2-28
2.4.1.4 Sequenced Power Application. 2-32
2.4.1.5 Initialize Hardware. 2-36
2.4.1.6 Test and Checkout. 2-44
2.4.1.7 Load RAMs and DRAMs. 2-46
2.4.2 VAX 8800 CPU Control 2-63
2.4.2.1 Console Sequencer. 2-64
2.4.2.2 Control Registers. 2-68
2.4.3 Data Transfers 2-70
2.5 CONSOLE/VAX 8800 CLOCKS AND TIMING 2-73
2.5.1 One-MHz Clock. 2-73
2.5.2 Interval Clock 2-75
2.5.3 CPU Timeouts 2-78
2.5.4 Visibility Bus 2-79

CHAPTER 3 DETAILED DESCRIPTION

3.1 GENERAL. 3-1
3.2 TERMINAL REGISTER INTERVAL CLOCK (TRIC). 3-1
3.3 CONSOLE SEQUENCER MCA (CSEQ) 3-9
3.3.1 Console Strobe Sequencer 3-9
3.3.2 Read Acknowledge 3-9
3.3.3 Console Write Sequencer. 3-9
3.3.4 Control Store Load Sequencer 3-9
3.4 CONSOLE/VAX 8800 REGISTER SUMMARY. 3-17
3.4.1 Console Registers (Refer to Figure 3-7). 3-17
3.4.2 VAX 8800 CPU Registers (Refer to Figure 3-8) . . 3-24
3.5 CONSOLE CABLING. 3-28

FIGURES

No.	Title	Page
1-1	Simplified Block Diagram of the Console Subsystem.	1-3
1-2	Modes of Operation	1-5
1-3	Power-Up Sequence.	1-11
1-4	EMM/Console Initialize	1-13
1-5	Restart/Boot/Halt.	1-18
1-6	Power-Fail Sequence.	1-20
1-7	Powerdown Procedure.	1-22
2-1	Console Subsystem Functional Block Diagram	2-2
2-2	PPI Port A Format.	2-3
2-3	PPI Port B Format.	2-3
2-4	PPI Port C Format.	2-6
2-5	PPI Control Register Format.	2-9
2-6	ECPI Mode 1 Register	2-10
2-7	ECPI Mode 2 Register	2-12
2-8	ECPI Command Register.	2-13
2-9	Serial Line Port Data and Status Registers	2-14
2-10	RXDB, TXDB, and DBCS	2-18
2-11	Control Registers.	2-21
2-12	VBus Control and Access Registers.	2-22
2-13	System Power-On Sequence	2-25
2-14	Environmental Monitoring Module Reset Sequence	2-26
2-15	Console Power-On Events.	2-27
2-16	Serial Line Port Data Transfer Registers	2-28
2-17	Load/Run Console Power-Up Software Events.	2-30
2-18	Sequenced Power Application Events	2-32
2-19	Console Interconnect Loopback Testing Through Ports A, B, and C.	2-37
2-20	Console Interconnect Loopback Testing Through Ports B and C.	2-37
2-21	Interface Data Path Loopback Test of Unbuffered Data	2-38
2-22	Interface Data Path Loopback Test of Buffered Data Through RXDB and TXDB	2-39
2-23	Console Sequencer Enable Logic	2-40
2-24	Control Register Initialization.	2-41
2-25	Hardware Initialization Events	2-42
2-26	Test and Checkout Events	2-44
2-27	VBus Parity Bits	2-49
2-28	DRAM Address	2-51
2-29	MNI Control Store Address.	2-53
2-30	RAM Loading Simplified Block Diagram	2-56
2-31	RAM/DRAM Loading Events.	2-57
2-32	Console/Interface Timing Signals	2-64
2-33	Write Sequence	2-65
2-34	Read Sequence (Setup).	2-66
2-35	Read Sequence (Data Out)	2-67

2-36 Simplified Diagram of Control Register 0 2-68
2-37 Simplified Diagram of Control Register 1 2-69
2-38 Simplified Diagram of Control Register 2 2-69
2-39 Data Transfer Control Interface-to-VAX 8800 CPU. 2-71
2-40 Data Transfer VAX 8800 CPU-to-Console Interface. 2-72
2-41 1 MHz Clock Generation 2-74
2-42 Interval Clock Registers Bit Configuration . . . 2-75
2-43 Interval Clock Simplified Block Diagram. 2-77
2-44 Simplified CPU Timeout 2-78
2-45 Clock Status and Timeout Register (CST). 2-78
2-46 VBus Control/Data Signals. 2-79
2-47 VBus Logic Simplified Block Diagram. 2-80
3-1 TRIC MCA Block Diagram 3-2
3-2 TRIC MCA Pin Layout. 3-3
3-3 TRIC MCA Body Drawing. 3-4
3-4 CSEQ MCA Block Diagram 3-10
3-5 CSEQ MCA Pin Layout. 3-11
3-6 CSEQ MCA Body Drawing. 3-12
3-7 Console Registers. 3-17
3-8 VAX 8800 CPU Registers 3-24
3-9 Console Subsystem Cabling Diagram. 3-29

TABLES

No. Title Page

2-1 PPI Port B Bit Description 2-4
2-2 PPI Port C bit Description 2-6
2-3 PPI Control Register Bit Description 2-9
2-4 ECPI Mode 1 Register Bit Description 2-11
2-5 ECPI Mode 2 Register Bit Description 2-12
2-6 ECPI Command Register Bit Description. 2-13
2-7 Serial Line Port Data and Status Registers Bit Description. 2-15
2-8 Serial Line Port Data Transfer Registers Bit Description. 2-29
2-9 Key Initialization Signal/Functions. 2-62
2-10 ICCS Bit Configuration 2-76

3-1 TRIC MCA Pin Assignments 3-5
3-2 TRIC MCA Signal Descriptions 3-6
3-3 CSEQ MCA Pin Assignments 3-13
3-4 CSEQ MCA Signal Descriptions 3-14
3-5 Console Cable List 3-28

SECTION 4 POWER SYSTEM COMPLEX

CHAPTER 1 GENERAL DESCRIPTION

1.1 INTRODUCTION 1-1
1.2 SYSTEM COMPONENTS. 1-4
1.2.1 876A Power Controller. 1-5
1.2.2 NBox Power Converter 1-5
1.2.3 Modular Power System (MPS) 1-6
1.2.4 Environmental Monitoring Module. 1-6
1.2.5 Cooling System 1-7
1.2.6 Battery Backup Unit H7231-M. 1-7
1.3 MECHANICAL CONFIGURATION 1-8
1.3.1 876A Power Controller. 1-11
1.3.2 NBox Port Conditioner. 1-11
1.3.3 MPS Modules (Regulators) and Cage. 1-11
1.3.4 Battery Backup Unit. 1-12
1.3.5 Air Flow System. 1-12
1.4 POWER DISTRIBUTION 1-13
1.4.1 AC Power 1-13
1.4.2 DC Power 1-15
1.4.3 Controls and Breakers. 1-17
1.4.3.1 Controls 1-17
1.4.3.2 Circuit Breakers 1-17
1.5 AC POWER SPECIFICATIONS. 1-19
1.5.1 Electrical Requirements. 1-19
1.5.1.1 AC Power Sources 1-19
1.6 FAULT AND STATUS INDICATORS. 1-21
1.6.1 876A Power Controller. 1-21
1.6.2 NBox 1-21
1.6.2.1 H7170 Built-In Test Equipment. 1-23
1.6.2.2 ILM Built-In Test Equipment. 1-23
1.6.3 Modular Power Supply Regulators. 1-24
1.6.4 Environmental Monitoring Module. 1-25
1.6.5 System Console Device. 1-26

CHAPTER 2 FUNCTIONAL DESCRIPTION

2.1 INTRODUCTION 2-1
2.2 POWER SYSTEM BLOCK DIAGRAM 2-1
2.3 SIMPLIFIED OPERATION 2-3
2.3.1 876A Power Controller. 2-3
2.3.2 NBox 2-4
2.3.2.1 H7170 Power Converter. 2-4
2.3.2.2 Control Start-up Power Module (CSP). 2-5
2.3.2.3 Interface Logic Module (ILM) 2-5
2.3.2.4 New Box Translator Module (NBT). 2-5
2.3.3 Modular Power System (MPS) 2-6
2.3.4 Battery Backup Unit (BBU Model H7231-M). 2-8
2.3.4.1 BBU Control. 2-9
2.3.5 Environmental Monitoring Module (EMM). 2-11

2.3.6 Power System Monitoring. 2-13
2.3.6.1 Key Monitoring 2-14
2.3.6.2 Regulator Control. 2-14
2.3.6.3 BBU Control. 2-14
2.3.6.4 Air Flow Status. 2-14
2.3.6.5 AC/DC LO Signals 2-15
2.3.6.6 Regulator Overtemp. and CPU Cabinet Temperature (Thermistor Volts) 2-15
2.4 POWER SEQUENCES. 2-16
2.4.1 Circuit Breakers 2-16
2.4.2 Summary of Power-Up Sequence 2-18
2.5 SYSTEM POWER-UP FLOWCHART (FIGURE 2-5) 2-24
2.6 CONSOLE POWER-DOWN FLOWCHART (FIGURE 2-6). . . . 2-29
2.7 POWER-DOWN/POWER INTERRUPT WITH BBU FLOWCHART (FIGURE 2-7) 2-31
2.8 POWERUP FROM BBU FLOWCHART (FIGURE 2-8). 2-35

CHAPTER 3 DETAILED DESCRIPTION

3.1 INTRODUCTION 3-1
3.2 BLOCK DIAGRAM OF THE VAX 8800 POWER SYSTEM 3-1
3.3 876A POWER CONTROLLER. 3-3
3.4 NBOX POWER CONVERTER ASSEMBLY. 3-6
3.4.1 NBox Modules 3-8
3.4.1.1 H7170. 3-8
3.4.2 CSP. 3-9
3.4.2.1 ILM. 3-9
3.4.2.2 NBT Module 3-14
3.4.3 Modular Power System -- MPS. 3-16
3.4.3.1 H7186 +5.0-Volt Regulator. 3-18
3.4.3.2 Side Panel 3-20
3.4.3.3 H7186 Main Board 3-21
3.4.4 H7187 -2.0-Volt Regulator. 3-22
3.4.5 H7180 -5.2-Volt Regulator. 3-22
3.4.5.1 H7180 Side Panel 3-24
3.4.5.2 H7180 Main PC Board. 3-24
3.4.6 H7189 BIP Regulator. 3-26
3.4.6.1 H7189 Functional Description 3-29
3.4.7 MPS Regulator BITE Indicators. 3-33
3.4.8 Buses and Backplanes 3-35
3.4.9 MPS Backplane. 3-35
3.4.9.1 300-Vdc Buses. 3-40
3.4.10 Environmental Monitoring Module. 3-41
3.4.10.1 8085A Microprocessor System. 3-43
3.4.10.2 Electric Key Monitor 3-44
3.4.10.3 Regulator Control Circuits 3-45
3.4.10.4 Regulator On/Off Control Circuits. 3-46
3.4.10.5 Regulator Margin Control Circuits. 3-46
3.4.11 Status Registers 3-48
3.4.11.1 AC/DC LO Circuits. 3-49
3.4.11.2 Total-Off Control and Indicator Circuits . . 3-52

3.4.11.3 Temperature Sensing Circuits 3-53
3.4.11.4 Voltage Measuring Circuit. 3-55
3.4.11.5 EMM/Console Voltage Tests. 3-58
3.4.11.6 Battery Backup Unit (BBU) Control. 3-58
3.4.12 Battery Backup Unit (H7231-M). 3-59
3.4.13 Air Flow Sensing Circuit 3-63
3.5 COOLING SUBSYSTEM. 3-65

FIGURES

No. Title Page

1-1 VAX 8800 System Physical Layout (Front View) . . . 1-2
1-2 VAX 8800 Power System Block Diagram (60 Hz). . . . 1-3
1-3 VAX 8800 Power System Block Diagram (50 Hz). . . . 1-4
1-4 VAX 8800 CPU Cabinet - Front View. 1-9
1-5 VAX 8800 CPU Cabinet - Rear View 1-10
1-6 876A Rear View Showing Receptacles 1-14
1-7 DC Power Section Block Diagram 1-16
1-8 VAX 8800 Power System Circuit Location Diagram. 1-18
1-9 876A Power Controller Front Panel. 1-21
1-10 NBox Front-Panel Indicators. 1-22
1-11 Indicators for the Modular Power Regulators. . . 1-24
1-12 EMM Front-Panel Indicators 1-25

2-1 VAX 8800 Power System Block Diagram. 2-2
2-2 MPS Backplane Configuration (Rear View). 2-7
2-3 Battery Backup Subsystem Functional Block Diagram. 2-10
2-4 EMM Functional Block Diagram 2-12
2-5 System Power-Up Flowchart. 2-20
2-6 System Power-Down Flowchart. 2-28
2-7 Powerdown/Power Interrupt with BBU Flowchart . . 2-30
2-8 AC Powerup from BBU Operation Flowchart. 2-34

3-1 Power System Block Diagram 3-2
3-2 876A Front and Rear Panels 3-4
3-3 876A Power Controller Block Diagram. 3-5
3-4 NBox System Interconnect 3-7
3-5 ILM PC Board Signals 3-10
3-6 NBT PC Board Signals 3-15
3-7 MPS Regulator Configuration. 3-17
3-8 H7186 Block Diagram. 3-19
3-9 H7180 Block Diagram. 3-23
3-10 H7189 Block Diagram. 3-27
3-11 BITE Indicators. 3-34
3-12 Organization of the Power System 3-37
3-13 MPS I Backplane. 3-38
3-14 MPS II Backplane 3-39
3-15 EMM Block Diagram. 3-42
3-16 Voltage Margining Circuit. 3-47

3-17 AC/DC LO Timing Diagram. 3-50
3-18 AC/DC LO Circuit 3-51
3-19 Temperature Sensing Circuit. 3-54
3-20 Voltage Measuring Circuit. 3-56
3-21 Voltage Measuring Technique. 3-57
3-22 BBU Block Diagram. 3-62
3-23 Air Flow Sensing Circuit 3-64
3-24 Air Flow Path. 3-66

TABLES

No. Title Page

1-1 Power System Components. 1-4
1-2 NBox Modules 1-5
1-3 876A Power Distribution. 1-15
1-4 VAX 8800 Circuit Breakers. 1-17
1-5 H7170 Status Indicators. 1-23
1-6 MPS Regulator Indicators 1-24
1-7 EMM Magnetic Status Indicator Codes. 1-26

2-1 876A Power Distribution. 2-4
2-2 Modules Using CSP Bias Voltages. 2-5
2-3 Voltage Regulators 2-7
2-4 System Circuit Breakers. 2-17

3-1 876A AC Power Distribution 3-3
3-2 NBox Modules 3-8
3-3 CSP Voltages 3-9
3-4 VAX 8800 MPS Regulators. 3-18
3-5 H7186 Side-Panel Components and Interconnects. . 3-20
3-6 H7186 Main Board Circuits and Interconnects. . . 3-21
3-7 H7180 Side Panel Components and Interconnects. . 3-24
3-8 H7180 Main PCB Circuits and Interconnects. . . . 3-24
3-9 H7189 Module Functions 3-28
3-10 H7189 Outputs. 3-31
3-11 H7189 Module I Circuits and Interconnects. . . . 3-32
3-12 H7189 Module II Circuits and Interconnects . . . 3-32
3-13 MPS Regulator Connectors 3-36
3-14 300-V Buses, Power Sources, and Loads. 3-40
3-15 Battery Backup Interface Signals 3-61

SECTION 5 CLOCK MODULE

CHAPTER 1 INTRODUCTION

1.1 BASIC OPERATION. 1-1
1.2 BASIC COMPONENTS AND TIMING. 1-3
1.3 CLOCK CONTROL (BY CONSOLE) 1-5
1.4 CLOCK STALLS 1-7
1.5 CLOCK STATUS 1-7

CHAPTER 2 FUNCTIONAL DESCRIPTION

2.1	DETAILED BLOCK DIAGRAM	2-1
2.1.1	Oscillator	2-1
2.1.2	Phase Generator	2-2
2.1.3	Clock Control Logic	2-2
2.1.4	Clock Distribution Circuits	2-2
2.2	CLOCK GENERATOR INITIALIZATION	2-7
2.3	SYSTEM CLOCK PERIOD CONTROL	2-8
2.3.1	Phase-Locked Loop Operation	2-8
2.3.2	Changing Clock Period	2-11
2.4	SYSTEM CLOCK START/STOP/BURST CONTROL	2-12
2.4.1	Starting the Clocks	2-13
2.4.2	Stopping the Clocks Unconditionally	2-13
2.4.3	Stopping the Clocks on Micromatch/Scope Sync Generation	2-16
2.4.4	Bursting the Clocks	2-17
2.4.5	Single-Stepping the Clocks	2-17
2.4.6	Single-Stepping the B CLK	2-18
2.5	SLOW CLOCK GENERATION AND CONTROL	2-19
2.6	CLOCK CONSOLE COMMANDS	2-20

FIGURES

No.	Title	Page
1-1	Clock Generator (and Console Interface) on Clock Module	1-2
1-2	System Clock Timing Diagram	1-4
1-3	Clock Control Register	1-8
1-4	Burst Count Register	1-9
1-5	Clock Period Register	1-9
1-6	Clock/Timeout Status Register	1-10
2-1	Clock Generator (Detailed Block Diagram)	2-3
2-2	Simplified Clock Frequency Control Circuitry	2-9
2-3	Simplified Clock Start/Stop/Burst Control Logic	2-14
2-4	Start/Stop/Burst Control Timing Diagram	2-15
2-5	Micromatch and Scope Sync Timing Diagram	2-16
2-6	Single-Stepping B CLK Timing Diagram	2-18
2-7	Slow Clock Timing Diagram	2-19

TABLES

No.	Title	Page
1-1	System Clocks.	1-1
1-2	Clock Control Register Bit Descriptions.	1-8
1-3	Clock/Timeout Status Register Bit Descriptions	1-10
2-1	Clock Generator Inputs	2-4
2-2	Clock Generator Outputs.	2-5
2-3	Clock Console Commands	2-20

SECTION 9 MEMORY SYSTEM (MBOX)

CHAPTER 1 INTRODUCTION

1.1 MANUAL SCOPE 1-1
1.2 WRITING PHILOSOPHY 1-1
1.3 MBOX FUNCTIONS 1-2
1.4 MBOX OVERVIEW. 1-4
1.4.1 NMI Signals Used by the MBox 1-6
1.4.2 MBox Operations. 1-6
1.4.3 Octaword Sort-of-Write 1-6
1.4.4 Command Bus Cycles 1-6
1.4.4.1 Write Longword Bus Cycles (Table 1-3). 1-9
1.4.4.2 Write Quadword Bus Cycles (Table 1-5). . . . 1-10
1.4.4.3 Write Octaword Bus Cycles (Table 1-6). . . . 1-11
1.4.4.4 Read Longword Bus Cycles (Table 1-7) 1-11
1.4.4.5 Read Octaword Bus Cycles (Table 1-8) 1-13
1.4.4.6 Read Hexword Bus Cycles (Table 1-9). 1-14
1.4.5 Memory Controller (MCL). 1-14
1.4.5.1 Command/Address Sequence 1-16
1.4.5.2 Normal Write 1-20
1.4.5.3 Read 1-22
1.4.5.4 Masked Write 1-25
1.4.6 Four-Megabyte Memory Array Board (MAR4). . . . 1-29
1.4.6.1 MAR4 Select and Command/Address. 1-30
1.4.6.2 Write Operation. 1-32
1.4.6.3 Read Operation 1-33

CHAPTER 2 MEMORY CONTROLLER (MCL)

2.1 DFA OVERVIEW (Figure 2-1). 2-1
2.1.1 Command/Address Cycle. 2-1
2.1.2 Write Data Cycle(s). 2-5
2.1.3 First Read Data Cycle. 2-7
2.1.4 "Next" Read Data Cycles. 2-8
2.1.5 NMI Interrupt. 2-9
2.1.6 NMI Memory Busy. 2-9
2.1.7 Single-Bit Error Correction During a Masked Write 2-10
2.1.8 CSR Reads. 2-10
2.2 DATA/ADDRESS (DAD) MCAS. 2-10
2.2.1 DAD Ports. 2-13
2.2.2 NMI Writes to Memory 2-13
2.2.3 NMI Reads from Memory. 2-13
2.2.4 Masked Writes Requiring Single-Bit Error Correction 2-14
2.2.5 Error-Free Masked Writes 2-15
2.2.6 Decode RAM Addressing. 2-15
2.2.6.1 Initially Loading the Decode RAM 2-15
2.2.6.2 Reading CSR1 2-15
2.2.7 CSR Reads to the NMI 2-16
2.3 FUNCTION (FUNK) MCA. 2-17
2.3.1 Function Field Parity. 2-20

2.3.2 Function Decoder 2-20
2.3.3 NEW CMD EARLY/NEW CMD LATE 2-21
2.3.4 Read Lock Function 2-22
2.3.5 Write Unlock Function. 2-22
2.3.6 Lock-Timeout Counter 2-23
2.3.7 Block Command. 2-23
2.3.8 Write Sequence Fault 2-24
2.3.8.1 Write Longword to Memory 2-24
2.3.8.2 Write Longword to CSR. 2-26
2.3.8.3 Write Quadword 2-26
2.3.8.4 Write Octaword 2-27
2.3.9 NMI Faults 2-28
2.3.10 NMI Confirmation 2-30
2.3.11 NMI DEAD 2-31
2.3.12 CSRs (Figure 2-4). 2-32
2.3.13 Read/Return and Read/Continue (Figure 2-5) . . 2-32
2.3.13.1 MCL Immediately Gets the NMI 2-33
2.3.13.2 MCL Waits for the NMI. 2-34
2.3.14 MRM Hold Command (Figure 2-6). 2-35
2.3.15 Force One Cycle (Figure 2-6) 2-35
2.4 ARBITRATION/ID (ARID) MCA. 2-35
2.4.1 NMI Data Parity (Figure 2-7) 2-35
2.4.1.1 Parity In. 2-34
2.4.1.2 Parity Out 2-37
2.4.2 NMI Function/ID Parity (Figure 2-7). 2-38
2.4.2.1 Parity In. 2-38
2.4.2.2 Parity Out 2-38
2.4.3 Fault Detect (Figure 2-8). 2-38
2.4.4 NMI ID/Mask (Figure 2-9) 2-40
2.4.4.1 ID/Mask In 2-40
2.4.4.2 ID Out 2-42
2.4.5 Arbitration/Hold Logic 2-43
2.4.5.1 Memory Gets the Bus Right Away - Longword Read. 2-43
2.4.5.2 Memory Gets the Bus Right Away - Octaword Read (Figure 2-11). 2-45
2.4.5.3 Memory Gets the Bus Right Away - Longword Read Back-to-Back with Another Read Function. 2-45
2.4.5.4 Memory Gets the Bus Right Away - Hexword Read 2-46
2.4.5.5 Memory Does Not Get the Bus Right Away - Longword Read. 2-46
2.4.5.6 Memory Does not Get the Bus Right Away - Two Longword Reads Back-to-Back or an Octaword Read. 2-46
2.4.6 Interrupts (Figure 2-12) 2-47
2.4.7 CSRs (Figure 2-13) 2-49
2.4.7.1 CSR0 2-49
2.4.7.2 CSR3 2-49
2.4.8 Memory Busy (Figure 2-14). 2-49
2.4.9 Clocks and Clock Control (Figure 2-15) 2-50
2.5 MDP OVERVIEW (Figure 2-16) 2-54
2.5.1 MDB Address In 2-54

2.5.2 MDB Address Out. 2-54
2.5.3 MDB Data In. 2-54
2.5.4 MDB Data Out 2-56
2.5.5 Write-Enable and Bad-Data Bits 2-56
2.5.6 Data Parity. 2-57
2.5.7 Data Read Operation. 2-57
2.5.8 Masked Write Operation 2-58
2.5.9 CSR Reads. 2-59
2.6 MEMORY DATA BUFFER (MDB) (Figure 2-17) 2-59
2.6.1 Address Read Port. 2-61
2.6.2 Data Read Port 2-62
2.6.3 W Write Port 2-62
2.6.4 C Write Port 2-63
2.6.5 CSR Logic. 2-64
2.7 MEMORY DATA BUFFER CONTROL (MDBC) MCA. 2-66
2.7.1 Loading Data into the MDB. 2-67
2.7.1.1 Normal Octaword Write With X and Y Buffers Empty. 2-67
2.7.1.2 Normal Octaword Write With Data Already in the X or Y Buffer 2-74
2.7.1.3 Masked Write With No Errors. 2-76
2.7.1.4 Masked Write With a Correctable Error. . . . 2-78
2.7.1.5 Masked Write With an Uncorrectable Error . . 2-79
2.7.2 Unloading Data From the MDB. 2-80
2.7.2.1 General. 2-80
2.7.2.2 Detailed 2-82
2.7.3 Y Out Select Logic 2-83
2.7.3.1 General. 2-83
2.7.3.2 Detailed 2-84
2.7.4 Internal Error, Write Decode RAM, and Clocks 2-87
2.8 DATA CHECK (DCHK) MCA. 2-87
2.8.1 Syndrome Generate. 2-89
2.8.2 Error Check. 2-89
2.8.3 Error Status 2-92
2.8.4 Serializer and CSR2. 2-95
2.8.5 Diagnostic Mode (Figure 2-28). 2-95
2.8.6 Reset and Clocks (Figure 2-28) 2-98
2.9 MRM OVERVIEW (Figure 2-33) 2-99
2.9.1 NAB Board Select 2-104
2.9.2 NAB Command Field and Parity 2-105
2.9.3 MDB Address Selection. 2-105
2.9.4 Internal Error 2-106
2.9.5 MDB Write-Data Load. 2-106
2.9.6 Octaword Writes. 2-106
2.9.7 Read-Data Cycle(s) 2-106
2.9.8 Masked Writes. 2-107
2.9.9 CSR READS. 2-108
2.10 MEMORY SEQUENCE CONTROL (MSC) MCA) 2-108
2.10.1 MSC Buffer Control (Figure 2-35) 2-109
2.10.1.1 Buffer Control Operation 2-109
2.10.1.2 AMRM BUSY REQ. 2-112
2.10.1.3 A CMD PROC START 2-113

2.10.2 BNUM Probe Buffer and Error Logic (Figure 2-36). 2-113
2.10.2.1 Probe Logic. 2-113
2.10.2.2 Error Logic. 2-116
2.10.3 Command/Address/Size Buffer (Figure 2-37). . . 2-116
2.10.3.1 Command Channel. 2-116
2.10.3.2 Address/Size Channel 2-118
2.10.4 Size Logic (Figure 2-38) 2-118
2.10.5 Starting Address Logic (Figure 2-40) 2-120
2.10.5.1 Initial Starting Address 2-120
2.10.5.2 Address Incrementation 2-124
2.10.6 Mask Address/Size Buffer (Figure 2-41) 2-124
2.10.7 Write Command Logic (Figure 2-42). 2-126
2.10.7.1 Write Machine. 2-126
2.10.7.2 Write Command Bits 2-128
2.10.8 Masked-Write Logic (Figure 2-43) 2-129
2.10.9 Command Done Logic (Figure 2-44) 2-131
2.10.9.1 BMSC PRE CMD DONE. 2-131
2.10.9.2 BMSC PRE MASK DONE 2-133
2.10.10 Command Parity 2-133
2.11 MEMORY SEQUENCE CONTROL 1 (MSC1) MCA 2-133
2.11.1 Mask Store (Figure 2-45) 2-134
2.11.2 Select Out Buffer Control (Figure 2-46). . . . 2-137
2.11.3 Read Buffer Control (Figure 2-47). 2-139
2.11.4 MDB Address I/O Select Logic (Figure 2-48) . . 2-139
2.11.4.1 MDB Address In Select Bits 2-139
2.11.4.2 MDB Address Out Select Bits. 2-142
2.11.5 Error Address Pointer. 2-143
2.11.6 BMRM INVERT ADDR4. 2-143
2.12 MEMORY ARRAY SEQUENCE CONTROL (MASC) MCA (Figure 2-49). 2-143
2.12.1 Command/Address Parity 2-143
2.12.2 Force Parity Error 2-145
2.12.3 Board Number (BMAS BNUM<2:0>). 2-145
2.12.4 Send No Command. 2-145
2.12.5 MASC Empty 2-146
2.12.6 Board Select (BMAS BD SEL<2:0>). 2-146
2.12.7 Command Accept (BMAS CMD ACPT) and Board Valid (BMAS BD VALID). 2-146
2.13 READ CONTROL SEQUENCER (RCS) MCA 2-147
2.13.1 Power Control (Figure 2-50). 2-147
2.13.2 CSR Control (Figure 2-51). 2-149
2.13.2.1 BMRM EN SERIAL RD. 2-149
2.13.2.2 BMRM SERIAL RD<2:0>. 2-149
2.13.2.3 AMRM CSR WRITE 2-149
2.13.2.4 ARCS FORCE CMD ACPT. 2-151
2.13.2.5 AMRM MPR DATA SEL. 2-151
2.13.2.6 BMRM FAKE CMD ACPT 2-151
2.13.3 Read Command Bits (Figure 2-52). 2-151
2.13.3.1 AMRM READ CMD<0> 2-151
2.13.3.2 BRCS READ CMD<1> 2-151
2.13.4 Board Select/Enable (Figure 2-52). 2-151
2.13.4.1 Board Select 2-151
2.13.4.2 Board Select Enable. 2-153

2.13.5 Read Data In Signals (Figure 2-52) 2-153
2.13.5.1 AMRM DRIVE NEW DATA. 2-153
2.13.5.2 BMRM NAB GATE. 2-153
2.13.6 RCS Full/Empty Status (Figure 2-52). 2-154
2.13.6.1 ARCS FULL. 2-154
2.13.6.2 BRCS EMPTY 2-154
2.14 BATTERY BACKUP UNIT (BBU). 2-154
2.14.1 Loss of Power. 2-156
2.14.2 Return of Power. 2-156

CHAPTER 3 FOUR MEGABYTE MEMORY ARRAY BOARD (MAR4)

3.1 VAX 8800 ARRAY BUS (NAB) 3-1
3.1.1 Signal Clocks. 3-1
3.1.2 Longword Write Timing. 3-1
3.1.3 Longword Read Timing 3-8
3.1.4 Octaword Read Timing 3-8
3.2 MAR4 OVERVIEW. 3-10
3.2.1 Write Operation (Figures 3-4 and 3-5). 3-12
3.2.2 Read Operation (Figures 3-4 and 3-6) 3-19
3.3 MAR4 DETAILED DESCRIPTIONS 3-22
3.3.1 Clock Logic. 3-23
3.3.2 4MBARRAY Banks 3-26
3.3.2.1 Array Bank Components. 3-26
3.3.2.2 Data Flow. 3-29
3.3.2.3 Array Command Sequencing 3-29
3.3.2.4 Array Refresh Sequencing 3-34
3.3.2.5 Battery Mode 3-36
3.3.2.6 Cold Start 3-38
3.3.2.7 Array Bank Differences 3-38
3.3.3 Input Parser 3-38
3.3.3.1 Command/Address Parity Check 3-39
3.3.3.2 Write Inhibit. 3-39
3.3.3.3 Data Ready Done. 3-39
3.3.3.4 MAR4 Board Selection 3-41
3.3.3.5 Generation of Array Bank Signals 3-41
3.3.3.6 Array Not Busy 3-42
3.3.3.7 Battery Mode 3-42
3.3.4 ECC/DPARITY. 3-42
3.3.4.1 ECC Check Bits 3-42
3.3.4.2 Write Inhibit. 3-42
3.3.4.3 Write Data Parity Check. 3-44
3.3.4.4 INT BAD DATA 3-44
3.3.5 Data Output Control. 3-44
3.3.5.1 MAR4 Read Enable 3-44
3.3.5.2 Bank Select. 3-46
3.3.5.3 MAR4 Data Transfer Enable. 3-47
3.3.5.4 Control of Read Data Transfer. 3-47
3.3.5.5 DRDY SNC CLK 3-48
3.3.5.6 Battery Mode 3-48
3.3.6 Refresh. 3-48
3.3.6.1 Normal Mode. 3-49
3.3.6.2 Battery Mode 3-53

FIGURES

No.	Title	Page
1-1	MBox Simplified Block Diagram.	1-3
1-2	MBox Read/Write Simplified Block Diagram	1-5
1-3	MBox Block Diagram	1-17
1-4	Command/Address Flow Diagram	1-18
1-5	Write Data Cycly Flow Diagram.	1-21
1-6	Read Data Cycle Flow Diagram	1-23
1-7	Masked Write Data Cycle Flow Diagram	1-27
1-8	MAR4 Read/Write Flow Diagram	1-31
1-9	MAR4 Command Fields.	1-32
2-1	DFA Block Diagram.	2-3
2-2	DAD Block Diagram.	2-11
2-3	FUNK Function and Control Logic.	2-18
2-4	FUNK CSRs.	2-25
2-5	Read/Return and Read/Continue Logic.	2-29
2-6	Clock and Command Control Logic.	2-36
2-7	Parity Generation and Checking	2-39
2-8	Fault Detect Logic	2-40
2-9	ID/Mask Logic.	2-41
2-10	Arbitration/Hold Logic	2-44
2-11	NMI Arbitration/Hold Timing.	2-45
2-12	Interrupt Logic.	2-48
2-13	CSR Logic.	2-51
2-14	Memory Busy Logic.	2-52
2-15	Clock, Reset, and Unjam Logic.	2-53
2-16	Memory Data Path (MDP) Block Diagram	2-55
2-17	Memory Data Buffer (MDB) Block Diagram	2-60
2-18	CSR Logic.	2-65
2-19	MDBC -- MDB Data-In Selection.	2-69
2-20	Input Load Command Detect Logic.	2-70
2-21	Full Logic	2-71
2-22	X and Y Bit Storage.	2-72
2-23	MDBC -- MDB Feedback Selection	2-77
2-24	Double-Bit Error Logic	2-80
2-25	MDBC -- MDB Data-Out Selection	2-81
2-26	Y Out Select Flow Diagram.	2-85
2-27	MDBC -- Internal Error, Write Decode RAM, and Clocks	2-88
2-28	DCHK Block Diagram	2-90
2-29	Error Check Block Diagram.	2-91
2-30	Error Status Block Diagram	2-93
2-31	Serializer Block Diagram	2-96
2-32	CSR2 Bit Map	2-97
2-33	MRM Block Diagram.	2-100
2-34	MSC Block Diagram.	2-110
2-35	Buffer Control	2-111
2-36	BNUM Probe Buffer and Error Logic.	2-114
2-37	Command/Address/Size Buffer.	2-117
2-38	Size Logic	2-119
2-39	Hex State Machine Flow Diagram	2-121

2-40 Starting Address Logic 2-122
2-41 Mask Address/Size Buffer 2-125
2-42 Write Command Logic. 2-127
2-43 Mask Write Logic 2-130
2-44 Command Done Logic 2-132
2-45 Mask Store Block Diagram 2-135
2-46 Select-Out Buffer Control Block Diagram. 2-138
2-47 Read Buffer Control Block Diagram. 2-140
2-48 MDB Address I/O Select Block Diagram 2-141
2-49 MASC Block Diagram 2-144
2-50 Power Control. 2-148
2-51 CSR Control. 2-150
2-52 Array Read Control 2-152
2-53 MCL BBU Block Diagram. 2-155
2-54 Power Down Flow Diagram. 2-157
2-55 Power Up Flow Diagram. 2-158

3-1 Longword Write Timing Diagram. 3-5
3-2 Longword Read Timing Diagram 3-6
3-3 Octaword Read Timing Diagram 3-7
3-4 MAR4 Block Diagram 3-11
3-5 Write Flow Diagram 3-13
3-6 Read Flow Diagram. 3-15
3-7 Clock Logic Block Diagram. 3-24
3-8 Clock Timing Diagram 3-25
3-9 4MBARRAY Bank Block Diagram (Bank 0 Shown) . . . 3-27
3-10 Array Command Flow Diagram 3-30
3-11 Array Refresh Flow Diagram 3-35
3-12 Input Parser Block Diagram 3-40
3-13 ECC/DPARITY Block Diagram. 3-43
3-14 Data Output Control Block Diagram. 3-45
3-15 Refresh Time Periods 3-50
3-16 Refresh Block Diagram. 3-51
3-17 Refresh Flow Diagram -- Normal Mode. 3-52
3-18 Refresh Flow Diagram -- Battery Mode 3-55
3-19 Initiation of Battery Mode Refreshes 3-57
3-20 Termination of Battery Mode Refreshes. 3-58

A-1 Flow-Diagram Symbols A-1

TABLES

No. Title Page

1-1 NMI Signals Used by the MBox 1-7
1-2 MBox Command Functions 1-8
1-3 Write Longword Bus Cycles. 1-9
1-4 NMI Confirmation Codes 1-10
1-5 Write Quadword* Bus Cycles 1-10
1-6 Write Octaword Bus Cycles. 1-11
1-7 Read Longword Bus Cycles 1-12
1-8 Read Octaword Bus Cycles 1-13
1-9 Read Hexword Bus Cycles. 1-15

2-1 Command Code 2-2
2-2 Size Code. 2-2
2-3 Function Codes 2-20
2-4 NMI Confirmation Codes 2-30
2-5 Read Function Codes. 2-34
2-6 Read Command Code. 2-59
2-7 Write Commands 2-82
2-8 ENABLE ECC Truth Table 2-99
2-9 Size Code. 2-118
2-10 Initial Address Truth Table. 2-123
2-11 Write State Code 2-128
2-12 Write Command Code 2-128

3-1 NAB Signals. 3-2
3-2 Clock Distribution 3-26
3-3 Read Bank Shift Register Modes 2-46

SECTION 10 NBI (NMI TO VAXBI ADAPTER)

CHAPTER 1 INTRODUCTION

1.1 GENERAL INFORMATION. 1-1
1.2 PHYSICAL DESCRIPTION AND CIRCUIT TECHNOLOGY. . . . 1-3
1.3 BASIC BLOCK DIAGRAM. 1-3
1.4 BASIC OPERATION. 1-7
1.4.1 CPU Read/Write Data Transfers. 1-7
1.4.2 DMA Read/Write Data Transfers. 1-10
1.4.3 Interrupt Operation. 1-13
1.5 NBI REGISTERS. 1-17
1.5.1 NBIA Registers 1-17
1.5.1.1 Control/Status Registers (CSR0 and CSR1) . . 1-18
1.5.1.2 Vector Registers (BR4VR through BR7VR) . . . 1-24
1.5.2 NBIB (BIIC) Registers. 1-25
1.5.2.1 Device Register (DTYPE). 1-26
1.5.2.2 VAXBI Control/Status Register (BICSR). . . . 1-27
1.5.2.3 Bus Error Register (BER) 1-30
1.5.2.4 Error Interrupt Control Register (EINTRCSR) 1-34
1.5.2.5 INTR Destination Register (INTRDES). 1-36
1.5.2.6 IPINTR Mask Register (IPINTRMSK) 1-37
1.5.2.7 IPINTR/STOP Destination Register (FIPSDES). 1-38
1.5.2.8 IPINTR Source Register (IPINTRSRC) 1-39
1.5.2.9 Starting and Ending Address Registers (SADR and EADR). 1-40
1.5.2.10 BCI Control and Status Register (BCICSR) . . 1-42
1.5.2.11 Write Status Register (WSTAT). 1-45
1.5.2.12 Force IPINTR/STOP Command Register (FIPSCMD). 1-46
1.5.2.13 User Interrupt Control Register (UINTRCSR) 1-47
1.5.2.14 General Purpose Registers (GPR <3:0>). . . . 1-49

CHAPTER 2 INTERFACE DESCRIPTIONS

2.1 NMI 2-1
2.1.1 NMI Signals. 2-4
2.1.2 Basic Timing 2-4
2.1.3 NMI Address Space. 2-11
2.1.4 NMI Read/Write Transactions. 2-11
2.1.5 NMI Arbitration/Memory Busy. 2-16
2.1.6 NMI Interrupts 2-20
2.1.7 NMI Errors 2-23
2.2 DATA BUS (BETWEEN NBIA AND NBIB) 2-24
2.3 VAXBI. 2-29
2.3.1 VAXBI Signals. 2-29
2.3.2 Basic Timing 2-34
2.3.3 VAXBI Address Space. 2-35
2.3.4 VAXBI Read/Write Transactions. 2-40
2.3.5 Interrupt Operation (INTR, IDENT, and IPINTR Transactions). 2-44
2.3.5.1 Interrupt (INTR) Transactions. 2-45
2.3.5.2 Identify (IDENT) Transactions. 2-47
2.3.5.3 Interprocessor Interrupt (IPINTR) Transactions 2-50
2.3.6 STOP Transactions. 2-52
2.3.7 Invalidate (INVAL) Transactions. 2-54
2.3.8 Bus Arbitration. 2-56
2.3.8.1 Bus Requests 2-56
2.3.8.2 Arbitration Modes. 2-57
2.3.8.3 Arbitration Control. 2-57
2.3.8.4 Extending a Transaction. 2-59
2.3.8.5 Special Mode Functions 2-59
2.3.9 VAXBI Errors 2-60
2.3.9.1 Parity Checking. 2-61
2.3.9.2 Transmit Check Error Detection 2-61
2.3.9.3 Protocol Checking. 2-62

CHAPTER 3 FUNCTIONAL DESCRIPTION

3.1 INTRODUCTION 3-1
3.1.1 NBIA Block Diagram 3-1
3.1.1.1 NMI Data Buffer. 3-2
3.1.1.2 NPAR MCA 3-4
3.1.1.3 NBIM MCA 3-4
3.1.1.4 NBFD MCA 3-4
3.1.1.5 NBAP MCA 3-4
3.1.1.6 NBCT MCA 3-5
3.1.1.7 DSEQ MCA 3-5
3.1.1.8 DC022 Transaction Buffer 3-5
3.1.1.9 Data (Bus) Buffers 3-5
3.1.1.10 Data Bus (and Transaction Buffer) Controls . . 3-5
3.1.2 NBIB Block Diagram 3-6
3.1.2.1 Data Bus Data Buffer 3-6
3.1.2.2 BCI Data Buffer. 3-6
3.1.2.3 Parity and Translation Logic 3-6

3.1.2.4 Data Buffer Read/Write Control 3-8
3.1.2.5 Length and Interrupt Control Logic 3-8
3.1.2.6 Master and Slave Port Sequencers 3-8
3.1.2.7 BIIC 3-8
3.1.2.8 VAXBI Clock Driver/Receiver. 3-9
3.2 INITIALIZATION/SELFTEST. 3-12
3.2.1 Basic NBI Initialization 3-12
3.2.2 BIIC Initialization/Selftest 3-14
3.2.3 Powerup. 3-14
3.2.4 NBI INIT/UNJAM 3-16
3.2.5 RESET (By Connected VAXBI Device). 3-18
3.3 CPU READ/WRITE OPERATIONS. 3-20
3.3.1 NMI Address Decoding and Translation 3-22
3.3.2 Local Read/Write Operations. 3-24
3.3.2.1 Command/Address Cycle. 3-25
3.3.2.2 Write Data Cycle 3-28
3.3.2.3 Return Data Cycle. 3-30
3.3.2.4 Parity Generation and Checking 3-32
3.3.3 VAXBI Read/Write (and IDENT) Operations. . . . 3-33
3.3.3.1 Command/Address Transfer 3-36
3.3.3.2 Write Data Transfer. 3-43
3.3.3.3 Return Read Data Transfer. 3-49
3.3.3.4 Parity Generation and Checking 3-55
3.3.4 Write Sequence Faults. 3-58
3.3.5 NMI BUS ACCESS TIMEOUTS. 3-58
3.3.6 VAXBI Errors 3-58
3.4 DMA READ/WRITE OPERATIONS. 3-60
3.4.1 Command/Address Transfer 3-66
3.4.1.1 Command/Address to BCI Data Buffer and Data Bus Buffer. 3-66
3.4.1.2 Command/Address to NBIA's Transaction Buffer 3-70
3.4.1.3 Command/Address to NMI (NMI Command/Address Cycle) 3-70
3.4.2 Write Data Transfer. 3-72
3.4.2.1 Write Data to BCI Data Buffer and Data Bus Buffer. 3-73
3.4.2.2 End of VAXBI Transaction (and Retries) . . . 3-76
3.4.2.3 Write Data to NBIA's Transaction Buffer. . . 3-76
3.4.2.4 Write Data to NMI (NMI Write Data Cycle or Cycles) 3-77
3.4.2.5 NMI Write Transaction Retries (NO ACCESS/MEMORY BUSY/NOACK). 3-78
3.4.2.6 DMA Errors 3-78
3.4.3 Return Read Data Transfer. 3-79
3.4.3.1 Return Read Data to Transaction Buffer . . . 3-79
3.4.3.2 NMI Read Transaction Retries (MEMORY BUSY). 3-83
3.4.3.3 Return Read Data to NBIB 3-83
3.4.3.4 Return Read Data to BCI Data Buffer and BIIC (VAXBI READ DATA CYCLE). 3-83
3.4.3.5 End of VAXBI Transaction 3-84
3.4.3.6 DMA Errors (VAXBI Transaction Retries) . . . 3-85
3.4.4 Parity Generation and Checking 3-86

3.4.4.1	Command/Address and Write Data/Mask Parity	3-86
3.4.4.2	Return Read Data/Status Parity	3-88
3.4.5	Timeouts.	3-90
3.4.6	Read Sequence Faults	3-90
3.5	INTERRUPT (INTR AND IPINTR) OPERATIONS	3-91
3.5.1	Decoding Interrupt Requests.	3-92
3.6	MISCELLANEOUS OPERATIONS	3-96
3.6.1	BIIC Register Read/Write Operations (by Other VAXBI Nodes)	3-96
3.6.2	VAXBI Stop Transactions.	3-96
3.6.3	VAXBI INVAL and BROADCAST Transactions	3-96
3.7	DIAGNOSTIC DATA TRANSFERS.	3-97
3.7.1	BIIC Loopback Requests	3-97
3.7.2	NBIA Wraparound.	3-98
3.7.3	CPU Read/Write to Memory (Flip Address Bits <29> and <22>).	3-99

FIGURES

No.	Title	Page
1-1	NBI Configuration.	1-2
1-2	NBI Basic Block Diagram	1-4
1-3	DC022 Transaction Buffer Organization	1-6
1-4	CPU Read/Write Data Transfer	1-8
1-5	DMA Read/Write Data Transfers.	1-11
1-6	INTR/IPINTR Operation	1-14
1-7	Interrupt Vector Format	1-15
1-8	SCB Format (Example)	1-16
1-9	Control/Status Register 0 (CSR0)	1-18
1-10	Control/Status Register 0 (CSR1)	1-22
1-11	Vector Registers (BR4VR through BR7VR)	1-24
1-12	Device Register (DTYPE).	1-26
1-13	VAXBI Control/Status Register (BICSR).	1-27
1-14	Bus Error Register (BER)	1-33
1-15	Error Interrupt Control Register (EINTRCSR).	1-34
1-16	INTR Destination Register (INTRDES).	1-36
1-17	IPINTR Mask Register (IPINTRMSK)	1-37
1-18	IPINTR/STOP Destination Register (FIPSDES)	1-38
1-19	IPINTR Source Register (IPINTRSRC)	1-39
1-20	Starting Address Register (SADR)	1-40
1-21	Ending Address Register (SADR)	1-41
1-22	VAXBI Control/Status Register (BICSR).	1-42
1-23	Write Status Register (WSTAT).	1-45
1-24	Force IPINTR/STOP Command Register (FIPSCMD)	1-46
1-25	User Interrupt Control Register (UINTRCSR)	1-47
1-26	General-Purpose Registers (GPR <3:0>).	1-49
2-1	NBIA and NBIB Input/Output Signals	2-2
2-2	Basic NMI Timing	2-4
2-3	NMI Address Space	2-12

2-4	NMI Write Transaction	2-13
2-5	NMI Read Transaction	2-14
2-6	Basic NMI Arbitration Line Timing	2-16
2-7	NMI Arbitration Line Timing (Typical)	2-18
2-8	MEMORY BUSY Timing	2-19
2-9	Fault Signal Timing	2-22
2-10	Basic VAXBI Timing	2-34
2-11	VAXBI Address Space	2-37
2-12	VAXBI Node Register Space	2-38
2-13	VAXBI-Required Registers	2-38
2-14	BIIC-Specific Device Registers	2-39
2-15	VAXBI Write Transaction (Octaword Length)	2-42
2-16	VAXBI Read Transaction (Octaword Length)	2-43
2-17	VAXBI Interrupt (INTR) Transaction	2-46
2-18	VAXBI Identify (IDENT) Transaction	2-49
2-19	VAXBI Interprocessor Interrupt (IPINTR) Transaction	2-51
2-20	VAXBI STOP Transaction	2-53
2-21	VAXBI Invalidate (INVAL) Transaction	2-55
2-22	Bus Arbitration Request Lines	2-56
2-23	Arbitration State Diagram	2-58
2-24	VAXBI Arbitration (Example)	2-60
3-1	NBIA Detailed Block Diagram	3-3
3-2	NBIB Detailed Block Diagram	3-7
3-3	NBI Powerup	3-15
3-4	Reset by VAXBI Node	3-17
3-5	UNJAM/Programmed NBI INIT	3-19
3-6	NMI Address Decoding and Translation	3-23
3-7	Local Read/Write Command/Address Cycle	3-27
3-8	Local Write Data Cycle	3-29
3-9	Local Read Data Cycle	3-31
3-10	Basic Information Flow Between NMI and VAXBI	3-34
3-11	NMI to VAXBI Command/Address Transfer	3-37
3-12	NMI to VAXBI Write Data Transfer	3-44
3-13	VAXBI to NMI Return Read Data Transfer	3-50
3-14	Aligned and Unaligned Quadword Read Data Ordering	3-62
3-15	Basic Information Flow Between VAXBI and NMI During DMA Read/Write Operations	3-64
3-16	VAXBI to NMI Command/Address Transfer	3-67
3-17	VAXBI to NMI Write Data Transfer	3-74
3-18	NMI to VAXBI Return Read Data Transfer	3-80
3-19	INTR/IPINTR Operations	3-93
3-20	FLIP 29/22 Diagnostic Data Transfers	3-101

TABLES

No.	Title	Page
1-1	NBIA Registers	1-17
1-2	Control/Status Register 0 (CSR0) Bit Descriptions	1-19
1-3	Control/Status Register 1 (CSR1) Bit Descriptions	1-22
1-4	NBIB (BIIC) Registers.	1-25
1-5	Device Register (DTYPE) Bit Descriptions	1-26
1-6	VAXBI Control/Status Register (BICSR) Bit Descriptions	1-28
1-7	Bus Error Register Bit Descriptions.	1-30
1-8	Error Interrupt Control Register (EINTRCSR) Bit Descriptions	1-35
1-9	INTR Destination Register (INTRDES) Bit Descriptions	1-36
1-10	IPINTR Mask Register (IPINTRMSK) Bit Descriptions	1-37
1-11	IPINTR/STOP Destination Register (FIPSDES)	1-38
1-12	IPINTR Source Register (IPINTRSRC) Bit Descriptions	1-39
1-13	Starting Address Register (SADR) Bit Descriptions	1-40
1-14	Ending Address Register (EADR) Bit Descriptions	1-41
1-15	BCI Control/Status Register (BCICSR) Bit Descriptions	1-42
1-16	Write Status Register (WSTAT) Bit Descriptions	1-45
1-17	Force IPINTR/STOP Command Register (FIPSCMD)	1-46
1-18	User Interrupt Control Register (UINTRCSR) Bit Descriptions	1-48
2-1	NMI Signals Connecting to NBIA	2-5
2-2	Data Bus Signals	2-25
2-3	VAXBI Signals.	2-30
3-1	BCI Signals.	3-9
3-2	NBI Initialization	3-13
3-3	CPU Read/Write Summary	3-21
3-4	DMA Read/Write Summary	3-61

SECTION 6
INSTRUCTION BOX (IBOX)

CHAPTER 1
INTRODUCTION

1.1 OVERVIEW

The Instruction unit (IBox) contains the microcode that controls the entire CPU (except for some Cache operations). The major functions of the IBox are as follows:

- Buffer prefetched VAX instruction stream (I-stream) data supplied by the Cache unit (CBox)
- Decode macroinstructions and control their execution
- Monitor and service microtraps, for example, interrupts, and exceptions
- Supply I-stream embedded data (for example, literals, immediate mode data, etc.) to the Execution unit (EBox)
- Provide an interface path between the Clock (CLK) module and the CPU

The IBox also maintains four internal privileged registers (IPRs) and part of the processor status longword (PSL).

1.1.1 Dual-Processor Configuration

Each processor in the VAX 8800 dual-processor configuration has its own IBox. The IBoxs operate independently of each other and have their own interface to the Clock (CLK) module.

1.2 LOGIC ELEMENTS

The IBox resides on three modules: the Decoder (DEC), Sequencer (SEQ) and the Writeable Control Store (WCS). Refer to Figure 1-1. The logic elements contained on each module are as follows:

Module	Logic Element
DEC	Instruction Buffer (IB) IB Manager Decoder Gateway Control Bus Watcher
SEQ	Microsequencer Part of the control store logic Condition Code and Macro Branch Logic Interrupt and Processor Register Logic File Address Generator
WCS	Rest of the Control Store

NOTE

The Bus Watcher logic is functionally part of the EBox and is described in the EBox section of this manual.

1.2.1 Physical Implementation

Most IBox logic is implemented by Macrocell Arrays (MCAs) which are high density, ECL chips that form the basis for most CPU logic. The MCA mnemonics are indicated in parentheses in Figure 1-1.

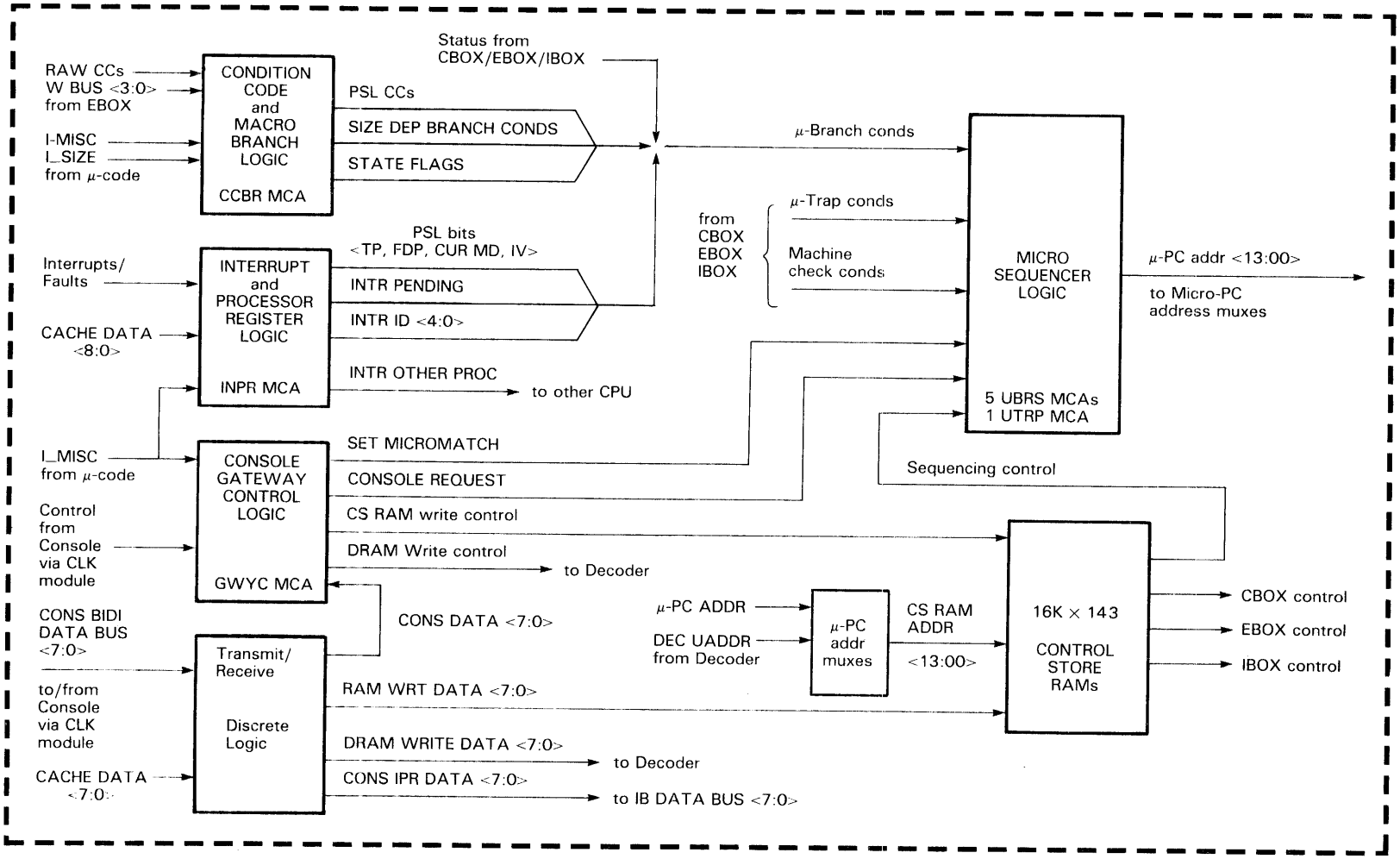


Figure 1-1 IBox Block Diagram (Sheet 1 of 2)

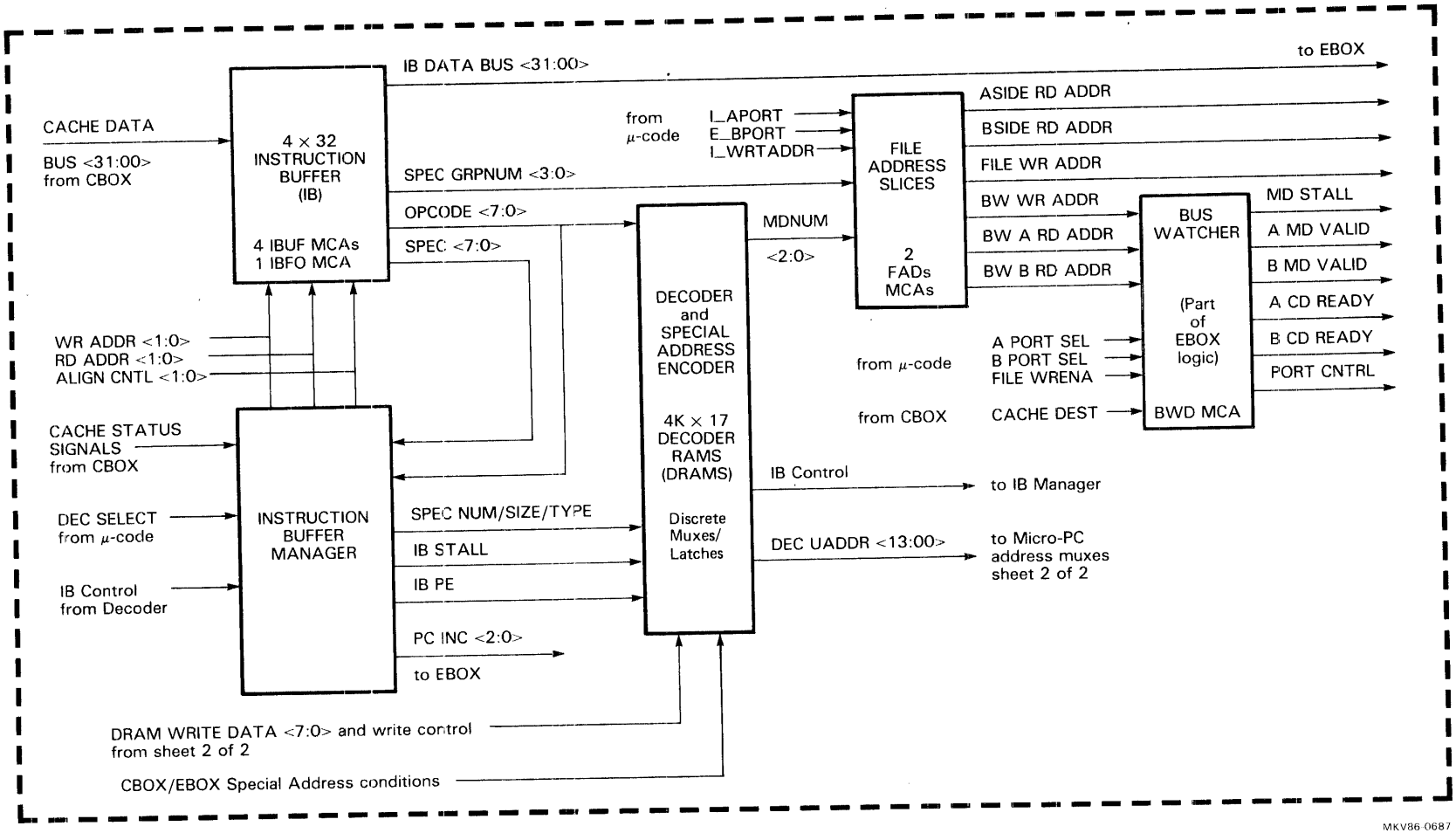


Figure 1-1 IBox Block Diagram (Sheet 2 of 2)

1.2.2 Instruction Buffer (IB)

The instruction buffer is a 4-longword (16-byte) memory. It stores prefetched VAX I-stream data supplied by the CBox and outputs the following data relative to the current macroinstruction:

- Op code byte to the IB manager and to the decoder
- Current operand specifier byte to the IB manager
- Specifier GPR number to the file address generator
- Specifier extension bytes (immediate data, literals, etc.) to the EBox on the IB data bus

The op code and current specifier are output simultaneously, specifier extension bytes (if any) are sign extended and output later.

1.2.2.1 Writing the IB

The IB is treated as a 4-longword memory when written.

Prefetched I-stream data enters the IB one longword at a time from the CACHE DATA BUS. The data are loaded in the IB location specified by the write address, IB WR ADDR <1:0>. The IB manager increments this value by one each time a longword enters the IB to point to the next location to receive data.

1.2.2.2 Reading the IB

The IB is treated as a 16-byte memory when read.

The starting IB byte location is specified by a combination of the read address, IB RD ADDR <1:0>, and the alignment control, IB ALIGN CNTL <1:0>. The read address points to the appropriate longword; the alignment control points to the proper byte in the longword. The IB manager updates these pointers each time a specifier is processed to reflect the new I-stream positioning.

The op code byte is read directly from the IB in the decode cycle for the first specifier. It is then stored in a "Cycle Op Code" register, which becomes the source of the op code for subsequent cycles (second to sixth specifiers).

1.2.3 IB Manager

The IB manager controls the IB read/write operations and computes the amount of IB data "consumed" during each IB decode cycle. It also indicates the current specifier's type (literal, register mode, etc.) and position (first through sixth) in the instruction to the Decoder.

1.2.3.1 IB Read/Write Control - The IB manager supplies the read, write, and the alignment control inputs to the IB. See Sections 1.2.2.1 and 1.2.2.2.

1.2.3.2 Computing Amount of IB Data Consumed - The amount of data consumed during the first cycle of an instruction includes the op code, the first specifier, and up to four specifier extension bytes. In subsequent cycles (second to sixth specifiers), it includes the specifier and up to four extension bytes.

Example: Instruction - MOVL #^X12345678, B^04(R0)

Cycle	IB Data Consumed
First	Six bytes - op code, first specifier, four extension bytes
Second	Two bytes - second specifier, one extension byte

The IB manager indicates the amount of IB data consumed to the EBox as a PC increment value, PC INC <2:0>. The EBox uses this value to update the VAX PC.

In the first IB decode cycle, the PC INC <2:0> value ranges from 0 to 6 and is based on the op code byte and the current specifier type bits. Thereafter, it ranges from 0 to 5 and is based on either the current specifier type bits alone or in combination with "predicted" size bits from the decoder.

NOTES

1. The predicted size bits are only used if a specifier in the second to sixth position is a branch displacement or immediate mode data. The bits are output in the current decode cycle, but indicate the size of the next specifier to be processed.

2. PC INC <2:0> can never be equal to 5 in the first decode cycle; the VAX architecture does not support 3-byte extensions.

1.2.4 Decoder Logic

The decoder logic consists of a 4K word by 17-bit writeable RAM (DRAM) and a special address encoder, which is composed of discrete muxes and priority encoders.

1.2.4.1 Decoder RAMs - The DRAMs are addressed by the current specifier number, the op code byte, and by a "2-byte" signal (2-byte op codes). The specifier number and the 2-byte signal are supplied by the IB manager, the op code byte is supplied by the IB.

Major functions

1. Supply the microsequencer with part of the entry point address for op code and specifier microroutines
2. Assist the IB Manager in controlling the IB
3. Indicate which EBox memory data register (MDR) is to receive data from memory for those specifiers that request data

Entry Address Generation

The DRAMs supply the low 5 bits of the entry-point address for every specifier microroutine. If more than one microword is required to service the specifier, the microsequencer takes control and generates the additional addresses for the specifier routine. When the specifier routine exits, control is returned to the decoder. The next specifier is then processed in the same manner.

Once all specifiers are processed, the DRAMs supply the low 5 bits of the entry address for the routine that performs the actual work of the instruction (the execute code) and inform the IB Manager that a new instruction is to be executed.

IB Control

The DRAMs work in conjunction with the IB manager to logically "shift" the next specifier out of the IB. They also indicate the data size of the specifier where applicable.

MDR Addressing

The MDRs reside in the EBox register file (RGF) and serve as scratchpad registers for all data requested from memory. The DRAMs supply a MDNUM to select the appropriate MDR during each specifier decode cycle. (The MDRs are described in the EBox section of this manual.)

1.2.4.2 Special Address Encoder - The special address encoder monitors certain "special" CPU conditions that may affect instruction execution.

When a special condition is present, the special address encoder:

1. Generates the entry point address for a routine to service the condition

2. Outputs the "special" entry address to the microsequencer in place of the specifier or op code address

If the condition is not critical, such as a TB miss while accessing the I-stream, the special condition microroutine returns control to the decoder after it services the condition.

If the condition is critical, such as an IB parity error, the special routine indicates the error in the IBox error register and passes control to a machine check microroutine (see Section 1.5.2).

Depending on the severity of the condition, the machine check routine either invokes a macrolevel service routine to record the error in the system error log or reports the error on the VAX console and halt the CPU.

1.2.5 Microsequencer Logic

The microsequencer is responsible for determining which of several sources is to supply the address of the next microword to be executed to the microPC address latches of the control store RAMs:

- Current microword
- Decoder entry-point microaddress
- EBox or CBox microtrap vector
- Machine check microtrap vector
- Trapped microPC from a microPC silo
- Microsubroutine return address from a microstack
- Console supplied microaddress

All address sources, except for the decoder, are multiplexed by the microsequencer logic. The address from the decoder and the one from the microsequencer are multiplexed by discrete logic. The selected address, which is 14 bits wide, is stored in microPC address latches and presented to the control store RAMs.

1.2.6 Control Store

The CPU control store microcode is 16K words deep by 143 bits wide and resides in 16K by 1 bit writeable RAMs. The microcode is loaded into the control store RAMs from the console Winchester disk during system initialization. The major microcode features are listed in Table 1-1.

Table 1-1 Microcode Features

Feature	Description
Horizontal in nature	Microword bits are grouped into fields. Each field directly feeds and controls a specific CPU logic element. (Some fields have vertical functionality in that they control more than one element.)
Pipelined operation	More than one microword is active at any given time. Allows the CPU to perform several operations simultaneously.
Segmented structure	Control store RAMs are divided into three physical segments: CS0 - SEQ module, 48 bits wide CS1 - WCS module, 48 bits wide CS2 - WCS module, 47 bits wide Each segment has its own parity bit (odd parity) and one or more spare bits.

Approximately 14K words of microcode control the CPU, 1K are available for user written code, and 1K are reserved to DIGITAL.

1.2.7 Condition Code and Macrobranch Logic

The CCBR MCA maintains the PSL condition code bits (N, Z, C, and V) and 7 CPU state flags.

1.2.7.1 PSL CC Bits - The CCBR MCA receives "raw" condition codes that result from various EBox operations (for example, main ALU functions) and generates a group of size-dependent microbranch conditions based on the raw CCs and the size of the data being processed. The size-dependent conditions can then be tested by microbranch logic in the microsequencer.

The raw CCs can also be compared to the current PSL CC bits to affect a macrobranch instruction or be stored as the new PSL CC bits.

1.2.7.2 CPU State Flags - The 7 CPU state flags are microprogramming aids that provide firmware writers with another means of controlling microcode flow. The flags can be set (one at a time) or cleared (individually or as a group) in one microroutine and then tested as microbranch conditions in a later routine. The flags are explicitly controlled by the microcode and are cleared in the first microword of every macroinstruction.

1.2.8 Interrupt and Processor Register Logic

The interrupt and processor register logic are both contained in the INPR MCA.

1.2.8.1 Interrupt Logic - The interrupt section of the INPR MCA maintains the hardware image of the IPL (interrupt priority level) field of the PSL. It monitors hardware interrupts, encodes the level of the highest pending request, and compares it to the current IPL. If the encoded level is greater than the current IPL, the interrupt logic outputs an interrupt identification tag (INTR ID <4:0>) and request that microcode take an interrupt branch by asserting an interrupt pending line (INTR PEND).

1.2.8.2 Processor Register Logic - The processor register logic maintains the hardware images of four VAX internal privileged registers. These registers, which are described in Section 1.4, control or supply data to the:

- Interrupt logic
- Microsequencer logic
- Memory management logic (in the CBox)

The INPR MCA also maintains copies of PSL bits <30,27,25:24,5:4>. (The entire PSL is kept in an EBox slow data file register.)

1.2.9 File Address Generator

The file address generator consists of 2 file address slice (FADS) MCAs. This logic performs the following:

- Supplies most of the address inputs to the EBox register file (RGF) and slow data file (SDF) registers
- Stores GPR numbers referenced by operand specifiers
- Records changes made to GPRs during autoincrement and autodecrement operations
- Allows fast access to operands requiring more than one GPR (quad and octaword operands)

1.2.10 Gateway Control Logic

The gateway control logic (GWYC MCA) links the CPU to the VAX console by providing a data path to the console interface logic resident on the clock (CLK) module.

1.2.10.1 Primary Functions -

1. Decode console commands

2. Control the CPU/CLK module data path

3. Control data transfers between the CPU and CLK module registers:

   - Console transmit/receive data buffers
   - Console control and status registers
   - CPU interval count registers

   These registers reside in the console interface logic of the CLK module.

4. Control loading of the following CPU elements:

   - CPU control store RAMs
   - CPU micromatch register
   - CPU decoder RAMs
   - Cache control sequencer RAMs

1.3 IBOX BUSES

Three major buses interface the IBox to the rest of the CPU:

1. Cache Data Bus
2. IB Data Bus
3. Cons Bidi Data Bus

1.3.1 Cache Data Bus

The cache data bus is 36 bits wide (32 data, 4 parity). It supplies the IBox with I-stream data for the IB (Section 1.2.2) and with data for the IBox and CLK module registers.

Register data is written one byte at a time from the low byte of the cache data bus. Microcode ensures that the bus contains the correct data for high-order bytes of the selected register.

1.3.2 IB Data Bus

The IB data bus is 36 bits wide (32 data, 4 parity). It supplies the EBox with I-stream embedded data from the IB (Section 1.2.2) and with data read from the IBox and CLK module registers.

Register data are output one byte at a time (least significant byte first) to the low byte of the IB data bus. The bytes are assembled into a longword in the EBox.

1.3.3 Cons Bidi Data Bus

The cons bidi data bus is an 8-bit, bidirectional bus that links the CLK module to the CPU. It allows the CPU to access registers resident on the CLK module and, through the console interface logic of the CLK module, to communicate with the console subsystem.

1.4 IBOX RESIDENT INTERNAL PRIVILEGED REGISTERS (IPRs)

Microcode implements the hardware images of two VAX architecture and two VAX 8800-specific IPRs in the IBox (IPR numbers are in hex). Refer to Table 1-2.

Table 1-2 IBox Resident IPRs

Name	Mnemonic	Number
VAX Architecture IPRs		
Interrupt Priority Level	IPL	12
Performance Monitor Enable	PME	3D
VAX 8800 Specific IPRs		
NMI Interrupt Control	NICTRL	80
Interrupt Other Processor	INOP	81

The IPRs reside in the INPR MCA of the SEQ module. They are written from the low order byte of the CACHE DATA BUS after the bus passes through the DEC module. The INPR MCA reports bad parity to the DEC module if it detects a parity error on the bus.

1.4.1 VAX Architecture IPRs

The IPL and PME registers are read/write to software but write-only to IBox hardware. Microcode maintains the software images for the IPRs in the EBox slow data file (SDF).

When a MTPR instruction writes the IPL or PME, the data is sent to both the INPR MCA and to the SDF. When a MFPR instruction reads an IPR, the data is obtained from the SDF copy.

The INPR MCA receives IPL data as bits <4:0> from the cache data bus. However, the bits are stored as PSL <20:16> in the SDF. Microcode shifts the bits to the proper position when writing the software image to the SDF.

PME bit 0 is sent to the INPR MCA as CACHE DATA BUS bit 1. Microcode shifts the bit to the proper position on the bus. The PME bit is available on the backplane for external monitoring.

1.4.2 VAX 8800-Specific IPRs

The two VAX 8800-specific IPRs both deal with the interrupt mechanism of the CPU.

Both registers are written from the low byte of the cache data bus and appear as 8-bit registers to hardware. The 32-bit software formats are shown in the following text.

1.4.2.1 NMI Interrupt Control Register (NICTRL) - The NICTRL register controls the CPUs response to interrupts requested by the two NBIAs and by NMI memory. The register is write-only and, as such, has no SDF software image.

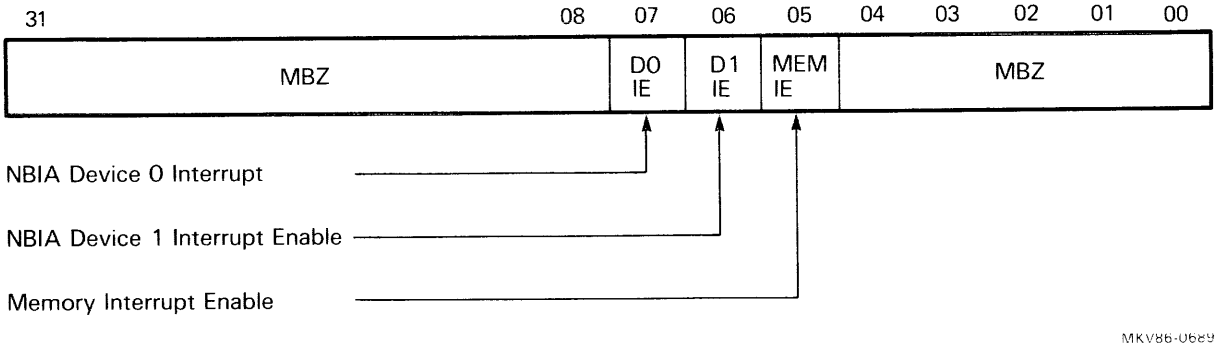


Figure 1-2 NMI Interrupt Control (NICTRL) Register Bit Map

Table 1-3 NICTRL Register Bit Descriptions

Bit	Mnemonic	Description
<7>	D0IE	When set, enables the CPU to respond to interrupts from NBIA Device 0. Cleared by CPU init.
<6>	D1IE	Same as above, but for NBIA 1.
<5>	MIE	Same as above, but for Main Memory.

1.4.2.2 Interrupt Other Processor Register (INOP) - The INOP register controls whether an interrupt is requested of the other processor in a dual-CPU system. This register is also write only and has no SDF image.

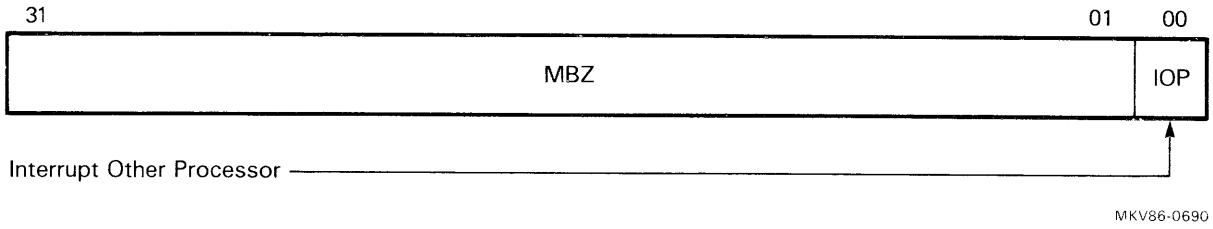


Figure 1-3 Interrupt Other Processor (INOP) Register Bit Map

Table 1-4 INOP Register Bit Description

Bit	Mnemonic	Description
<0>	IOP	When set, causes an interrupt in the other processor of a dual-CPU configuration.

The INOP register exists as a latch in the INPR MCA. The latch is set when microcode addresses the register and is automatically cleared one CPU clock cycle later.

1.5 IBOX MICROCODE VISIBLE ONLY REGISTERS

The IBox maintains three hardware registers that are only accessible by the microcode:

1. Clear interrupt other processor (CIOP)
2. IBOX error register (IBER)
3. Clear error register (CER)

1.5.1 Clear Interrupt Other Processor (CIOP)

This register clears the interrupt requested by the other processor of a dual-processor system. The register only exists as a signal in the INPR MCA that is asserted when microcode addresses the register and is negated one CPU clock cycle later.

1.5.2 IBox Error Register (IBER)

The IBER is a 12-bit register that records errors detected by IBox hardware and by microcode. The register is maintained by microcode in an EBox SDF register. (Refer to Figure 1-4 and Table 1-5.)

1.5.2.1 IBER Usage - The IBER is stored along with other similar error registers from the CBox (CBER) and EBox (EBER) in a SDF data structure known as a machine check error bank.

When a CPU error occurs, the error registers, and other relevant data (virtual PC, current PSL, etc.), are written to the stack and to the MC error bank. If the error is recoverable, the system software will obtain the data from the stack and record it in the system error log. If the error is not recoverable, the VAX console will obtain the data from the MC error bank and report it to the console operator. (Machine checks are discussed in Chapter 3.)

1.5.2.2 IBER Bits <7:0> - These bits reside in discrete latches on the DEC module and report parity errors detected by the DEC and SEQ modules. The latches only store the first error received. They are then "locked" by hardware to prevent a second error from being reported until the first one is serviced. Thus, if a second error occurs before microcode services the first one, the new error indication is lost. The latches are cleared by writing the clear error register (CER).

Bits <7:6,4,0> all indicate parity errors while accessing processor registers. The mnemonics in Table 1-5 indicate the direction of transfer and the module detecting the error. For example, if bit 7 is set, data was being transferred from the DEC module to the SEQ module and the error was detected by the SEQ module. This means that the DEC module received data from the cache data bus ok but dropped (or picked) a bit when it routed the data to the SEQ module.

1.5.2.3 IBER Bits <11:08> - These bits do not exist in latches but only in the SDF image of the IBER. They are written by the microroutines that service the special conditions mentioned in the special address encoder discussion (see Section 1.2.4.2).

Bit 11 is reported by the CBox but is considered an IBox problem since it is related to the IB.

Bit 10 indicates that either the microsequencer, the decoder, or some microroutine generated the address of a microword that should never be accessed. All such microwords contain code to pass control to a routine that will set bit 10. Since this means there is a hardware or a microcode bug in the addressing mechanism, the error causes a fatal machine check.

Bits 8 and 9 only indicate that an IB parity error was detected; there is no logic to determine which longword is at fault. If a double IB parity error occurs, only IB PE UW is reported (bit 09).

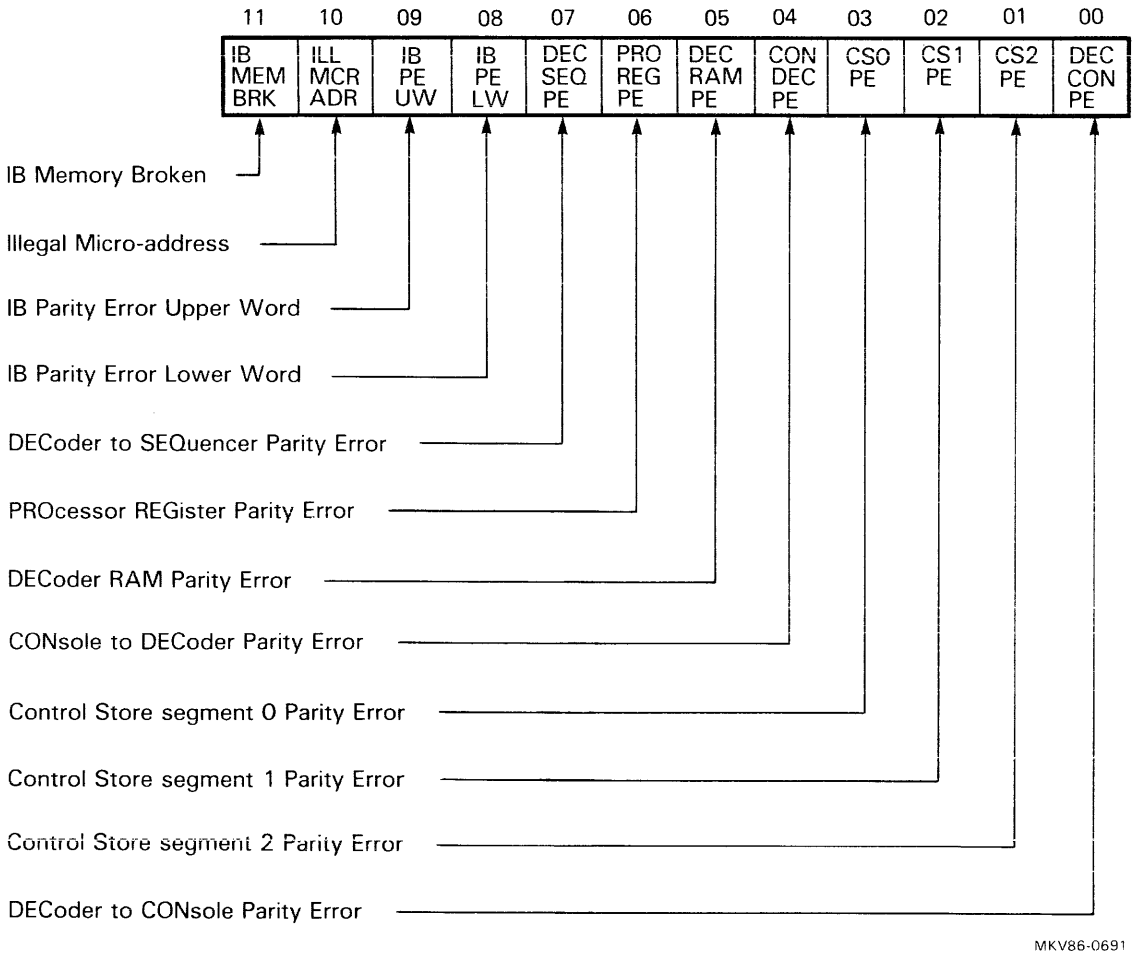


Figure 1-4 IBox Error Register (IBER) Bit Map

Table 1-5 IBER Bit Descriptions

Bit	Mnemonic	Description
<11>	IB MEM BRK	Error detected by CBox (TB, NMI, Cache, etc.) while prefetching I-stream data for IB.
<10>	ILL MCR ADR	Microsequencer, decoder, or microcode itself generated an illegal microaddress.
<09>	IB PE UW	IB longword location received bad parity from upper word of the cache data bus.
<08>	IB PE LW	Same as bit 09 except for low word of bus.
<07>	DEC SEQ PE	SEQ module detected bad parity on data from DEC module while writing a processor register.
<06>	PRO REG PE	DEC module detected bad parity on cache data bus while writing a processor register.
<05>	DEC RAM PE	DEC module detected bad parity on the decoder RAMs.
<04>	CON DEC PE	DEC module detected bad parity on cons bidi data bus while reading a CLK module processor register.
<03>	CS0 PE	SEQ module detected bad parity from the CS0 RAMs.
<02>	CS1 PE	WCS module detected bad parity from the CS1 RAMs.
<01>	CS2 PE	WCS module detected bad parity from the CS2 RAMs.
<00>	DEC CON PE	CLK module detected bad parity on cons bidi data bus while writing a processor register.

1.5.3 Clear Error Register (CER)

The CER is a write only register that exists as a latch in the INPR MCA. The latch is set when microcode addresses the register and is cleared one CPU clock cycle later.

Writing a one to the CER clears the latches that store IBER bits <7:0>. Since IBER bits <11:08> only reside in the SDF, they are not affected by the CER and must be cleared by microcode.

CHAPTER 2
MICROCODE OVERVIEW AND PIPELINE CONCEPTS

2.1 CHAPTER SCOPE

This chapter describes the general structure and organization of the VAX 8800 microcode and presents the concept of microcode pipelining. The following topics are covered in this chapter:

- Microcode file structure and assembly
- Microword format and bit field definitions
- Microcode pipelining concepts
- Characteristics of the VAX 8800 pipeline

Since the VAX 8800 processor is a pipelined machine, understanding how the microcode controls the hardware and the concept of pipelining are essential to understanding the operation of the CPU.

2.2 VAX 8800 MAIN CONTROL STORE OVERVIEW

The main control store microcode controls all CPU kernel operations except for certain CBox functions. The CBox has its own microcode that interprets commands from the main control store and controls the requested CBox functions. Refer to the CBox section of this manual for information on the CBox microcode.

2.2.1 Microcode Size and Allocation

The main control store microcode is 16K words deep by 143 bits wide and resides in a set of 16K by 1 bit writeable RAMs. The microcode is loaded into the control store RAMs from the console's Winchester disk during system initialization.

Approximately 14K words of microcode are dedicated to controlling CPU kernel operations, 1K are available for user-written code, and 1K are reserved for DIGITAL.

2.2.2 Microcode File Structure

The VAX 8800 microcode consists of a large set of microroutines that are logically grouped by function into separate files. For example, all routines that handle integer and logical macroinstructions reside in one file while all memory management routines reside in another. Table 2-1 lists the microcode files and the microroutines contained in each file.

2.2.3 Microcode Assembly

The microcode is initially written in the MICRO2 assembler language as a set of source code files. The source code files are then assembled by MICRO2 into two ASCII output files:

1. UCODE.ULD - Microcode object (data) file
2. UCODE.MCR - Microcode listing file

The UCODE.ULD file is further processed by a MICRO2 support utility that produces a loadable file called UCODE.BIN. This file contains the binary data that is loaded into the main control store RAMs.

The UCODE.MCR file contains the text from the original source files and the hexadecimal equivalent of the machine code generated by the MICRO2 assembler. The UCODE.MCR file is available on microfiche.

Table 2-1 VAX 8800 Microcode Files

File Name	Microroutines For
CHARSTR.MIC	Character string and CRC instructions
CONTROL.MIC	PC control instructions
CSM.MIC	Console support microcode
CSX.MIC	CSM overlay and user WCS area
DECIMAL.MIC	Decimal string instructions
EDITPC.MIC	Edit instructions
FLOAT.MIC	Floating-point instructions
IANDE.MIC	Interrupt and exception routines
INTLOG.MIC	Integer and logical instructions
LDSV.MIC	Load/Save process context instructions
MM.MIC	Memory management routines
MULDIV.MIC	Integer multiply and divide instructions
MXPR.MIC	Move to/from privileged register instructions
PCALL.MIC	Procedure call/return instructions
QUEUE.MIC	Queue instructions
VIELD.MIC	Variable length bit field instructions

NOTE

There are two other files associated with the main control store: DEFIN.MIC and MACRO.MIC. These files contain definitions required by MICRO2 to generate the UCODE.BIN file. Refer to Section 2.2.5.

2.2.3.1 Other Loadable Binary Files - In addition to UCODE.BIN, there are three other binary files that are also loaded during system initialization:

File Name	Destination
CCODE.BIN	CBox - NMI microsequencer RAMs
DRAM.BIN	IBox - Decoder RAMs
SDFDEF.BIN	EBox - Slow data file RAMs

These files are generated in a manner similar to that used to create the UCODE.BIN file.

2.2.4 Microword Format

The VAX 8800 microword is divided into several fields. Each microword field is assigned a unique symbolic name indicative of the function the field controls.

Figure 2-1 shows the microword bit format, the bits supplied by each control store RAM segment (see Chapter 3), and the symbolic name of each field. Table 2-2 briefly describes the function of each field.

2.2.4.1 Field Naming Convention - The first letter of a field name indicates which major CPU kernel unit (CBox, EBox, IBox) contains the logic element(s) the field controls. The rest of the name is a mnemonic for the function. For example, the "I" portion of the I_NEXT field name indicates that the logic element resides in the IBox. The "NEXT" portion means that the field deals with generating the address of the next microword to be executed.

2.2.4.2 Field Functionality - Note in Figure 2-1 that most microword bits are assigned one field name while some are assigned two or more.

Microword bits with one field name assignment are said to have horizontal functionality in that they control only one CPU function. Bits with multiple names are said to have vertical functionality in that they control several functions. The function of bits with more than one field name depends on the setting of other fields.

Certain microword fields are only valid when used in combination with other fields. For example, the E_MULDIV field, which specifies the function performed by the EBox multiplier/divider unit, is only valid if the E_MULDEN field enables the multiplier. Otherwise, E_MULDIV is considered to be part of the larger E_SHFTCNT field.

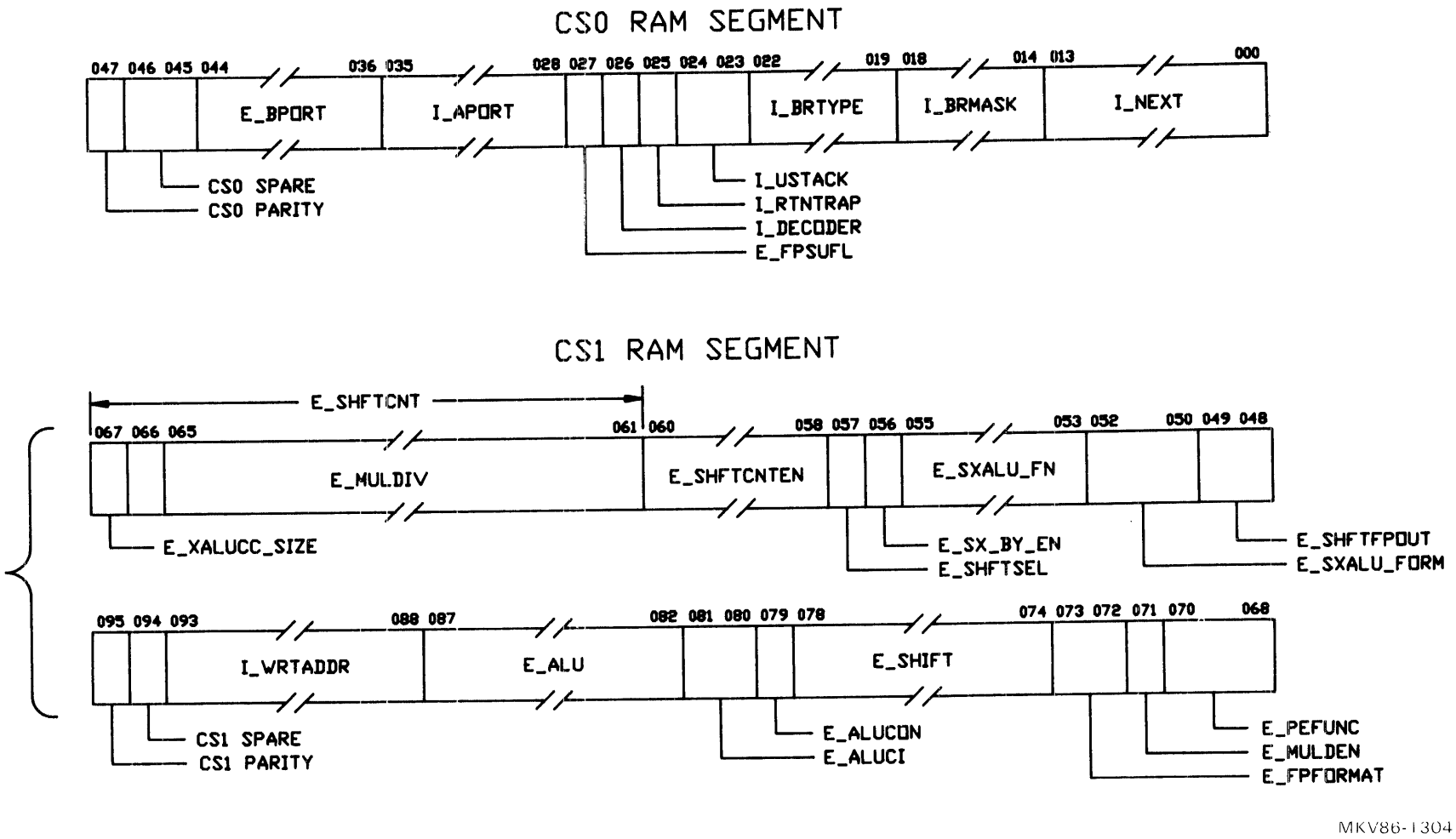


Figure 2-1 Microword Bit Format (Sheet 1 of 2)

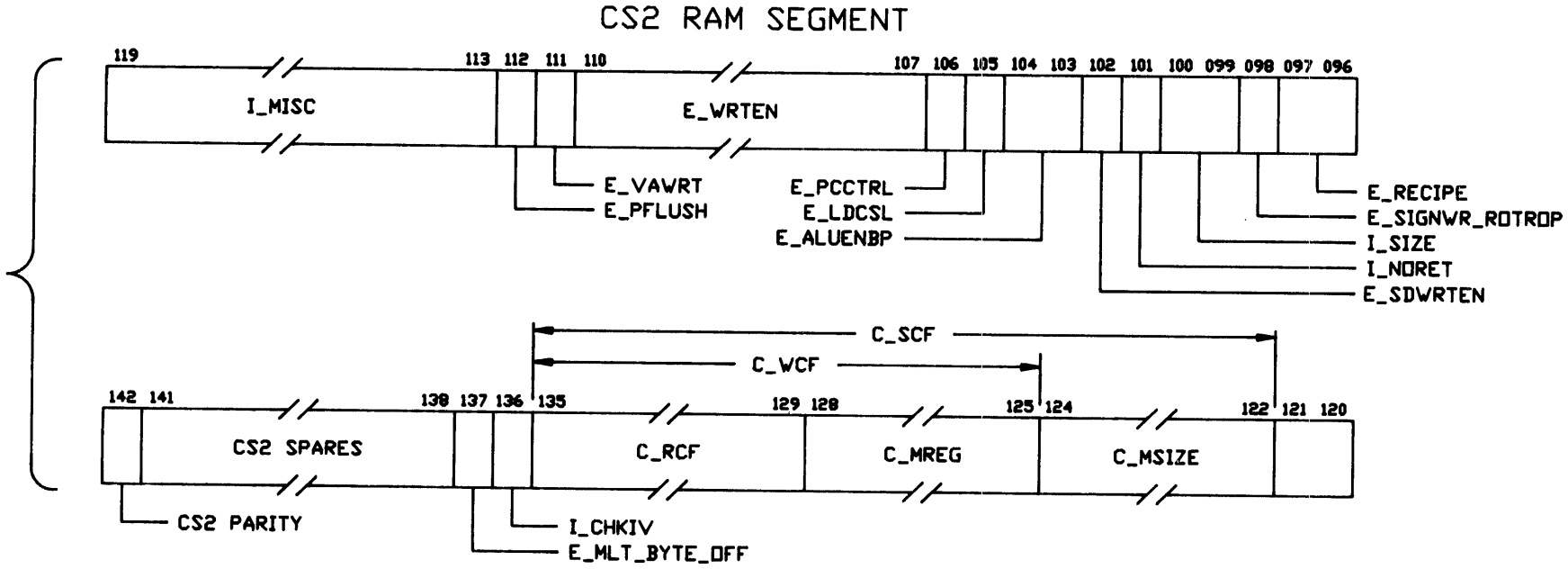


Figure 2-1 Microword Bit Format (Sheet 2 of 2)

Table 2-2 Microword Field Definitions

Bit(s)	Field Name	Description
013:000	I_NEXT	Contains the base address of the next microword to be executed.
018:014 022:019	I_BRMASK I_BRTYPE	These fields combine to control multiway conditional microbranching. I_BRMASK - Specifies which I_NEXT <4:0> bits (one or more) can be modified to affect a microbranch. I_BRTYPE - Specifies which of 16 microbranch condition groups to branch on.
024:023	I_USTACK	Controls microstack operation for subroutine calls/returns and returns from microtraps.
025	I_RTNTRAP	Releases the microPC silo on returning from a microtrap routine.
026	I_DECODER	Selects the decoder logic as the source of the next microaddress.
027	E_FPSUFL	Enables the floating-point "shuffle" function.
035:028	I_APORT	Specifies the source for the EBox A port mux: • Register file • Slow data file • PC or VA register • IB data bus Also controls special register accessing during operand specifier processing (RNUM1, RNUM2, and RLOG registers in the FADS MCAs).
044:036	E_BPORT	Specifies the source for the EBox B Port mux: • Register file • Slow data file • IB data bus
046:045	CS0 SPARES	CS0 RAM segment spare bits.
047	CS0 PARITY	CS0 RAM segment parity bit (odd parity).

Table 2-2 Microword Field Definitions (Cont)

Bit(s)	Field Name	Description
049:048	E_SHFTFPOUT	Specifies the format for data output from the EBox shifter logic.
052:050	E_SXALU_ FORM	Specifies the source and format of data input to, or the format of data output from, the EBox shift and exponent ALUs (SALU, XALU). Interpretation of this field depends on the function encoded in E_SXALU_FN field.
055:053	E_SXALU_FN	Function code for the SALU and XALU.
56	E_SX_BY_EN	Enables the output from the SALU and XALU to be written to the EBox bypass bus. E_SX_BY_EN is only valid if E_SXALU_FN is encoded with a bypass function.
57	E_SHFTSEL	Selects source of data input to the shifter or specifies which longword of a 64-bit result the shifter is to output. Interpretation of this field depends on the function encoded in E_SHFT field.
060:058	E_SHFTCNTEN	Selects the source of the shift count for the shifter's shift count bus.
067:061	E_SHFTCNT	Specifies a direct shift count to the shifter. The field is treated as an absolute (unsigned) value in the range 0 to 63.
065:061	E_MULDIV	Function code for the multiplier/divider unit if the unit is enabled by the E_MULDEN field.
067	E_XALUCC_ SIZE	Specifies the floating-point exponent format (F/D or G type data) required by the XALU to generate correct FP over/underflow condition codes for microbranching.
070:068	E_PEFUNC	Priority encoder function.

Table 2-2 Microword Field Definitions (Cont)

Bit(s)	Field Name	Description
071	E_MULDEN	Multiplier/Divider unit enable.
073:072	E_FPFORMAT	Defines format of FP data input to shifter and priority encoder. Or, specifies type of BCD conversion to be performed by shifter.
078:074	E_SHFT	Shifter function code.
079	E_ALUCON	Selects hardwired constant of 4 as input to main ALU B port mux.
081:080	E_ALUCI	Selects source of carry bit input to main ALU.
087:082	E_ALU	Main ALU function code.
093:088	I_WRTADDR	Specifies address of register file location to be written.
094	CS1 SPARE	CS1 RAM segment spare bit.
095	CS1 PARITY	CS1 RAM segment parity bit (odd parity).
097:096	E_RECIPE	Selects the SALU CC bits that result from FP operations as microbranch recipes.
098	E_SIGNWR_ ROPTRAP	This bit performs three basic functions: 1. Controls loading of the SALU sign latch. 2. Enables FP reserved operand trap checking (if selected by E_SXALU_FORM). 3. Helps control FP exponent subtraction by the SALU.
100:099	I_SIZE	Indicates data size (byte, word, longword) for size-dependent microbranch conditions.

Table 2-2 Microword Field Definitions (Cont)

Bit(s)	Field Name	Description
101	I_NONRET	Sets the NORETRY flag, which is cleared by the IBox hardware during the first microword of every macroinstruction. The NORETRY flag is tested by machine check microcode to determine if a macroinstruction can be restarted following a machine check.
102	E_SDWRTEN	Enables writing the slow data file register addressed by the E_BPORT field.
104:103	E_PCCTRL	VAX PC function code (increment, load PC).
105	E_LDCSL	Loads main ALU's carry save latch. (Saves the carry bit for later use.)
106	E_ALUENBP	Enables the main ALU to drive the bypass bus.
110:107	E_WRTEN	Selects bytes on the WBus to be written to the register file Location addressed by the I_WRTADDR field or by hardware. (There is one E_WRTEN bit per data byte.)
111	E_VAWRT	Enables writing contents of VA Bus to EBoxs' VA register and to CBoxs' VA latch.
112	I_PFLUSH	Specifies that IBox hardware is to perform a partial flush of the IB. Issued prior to returning from microtrap service routines.
119:113	I_MISC	Miscellaneous control field for writing: • PSL CC bits • CPU state flags • Certain VAX IPRs Also used to select the PSL CC bits to be tested by conditional macrobranches.

Table 2-2 Microword Field Definitions (Cont)

Bit(s)	Field Name	Description
121:120	I_MDNUM	Specifies that IBox hardware is to supply the address of the EBox MD register to receive memory read data. The I_MDNUM field overrides the C_MREG field.
124:122	C_MSIZE	Specifies the cache data size (byte, word, longword, quadword, octaword).
128:125	C_MREG	Specifies the MD register to receive memory read data (overridden by I_MDNUM).
135:129	C_RCF	CBox read functions (memory and registers).
135:125	C_WCF	Memory write functions.
135:122	C_SCF	Miscellaneous CBox functions (special memory reads and writes, TB checks and writes, CBox register writes).
136	I_CHKIV	Enables IBox hardware to generate an integer overflow microtrap if PSL <V> and <IV> bits are both set.
137	E_MLT_BYTE_OFF	Controls writing least significant byte from Mul/Div unit to bypass bus.
141:138	CS2 SPARES	CS2 RAM segment spare bits.
142	CS2 PARITY	CS2 RAM segment parity bit (odd parity).

2.2.5 Microcode Definition Files

MICRO2 supports all VAX-based systems and requires certain processor-specific information before it can assemble data to be loaded in the control store RAMs. The microprogrammers supply this information in two source code files:

File Name	Information supplied
DEFIN.MIC	Definitions of the microword fields and the data considered valid for each field.
MACRO.MIC	Definitions of macroexpressions that allow the microprogrammers to specify several microword field settings with one symbolic statement.

2.2.5.1 Field Definition File - DEFIN.MIC - Microprogrammers supply microword field definitions in DEFIN.MIC by specifying the field name, the microword bits spanned by the field, and the default setting for the field. For example, the definition for the I_APORT field appears in DEFIN.MIC as follows:

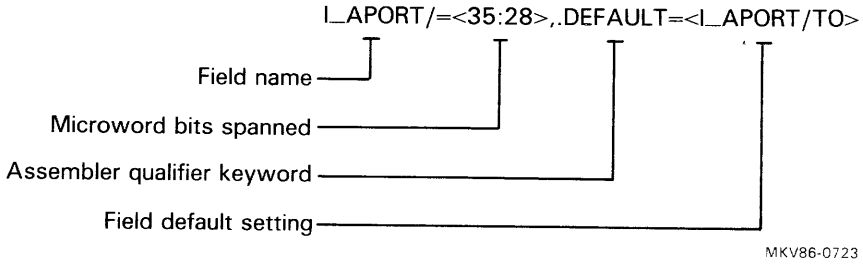


Figure 2-2 Sample Microword Field Definiton - I_APORT Field

The above definition instructs MICRO2 to assign symbolic name I_APORT to microword bits <35:28> and to use the symbolic value of T0 as the default for the field (symbolic values are discussed in the following paragraphs). MICRO2 automatically encodes the default value in the field if a value is not explicitly specified by the microprogrammers.

Symbolic Field Value Names

Each microword field definition is immediately followed by a series of statements that equate the valid hexadecimal values for the field to symbols the microprogrammers use to represent the values.

The following is a partial listing of the symbolic values associated with the I_APORT field.

```
I_APORT/=<35:28>,.DEFAULT=<I_APORT/T0>

   R0=80       ; General Purpose Register 0
   R1=81       ; General Purpose Register 1
   R2=82       ; General Purpose Register 2
   R3=83       ; General Purpose Register 3
   R4=84       ; General Purpose Register 4
   R5=85       ; General Purpose Register 5
   R6=86       ; General Purpose Register 6
   R7=87       ; General Purpose Register 7
   R8=88       ; General Purpose Register 8
   R9=89       ; General Purpose Register 9
   R10=8A      ; General Purpose Register 10
   R11=8B      ; General Purpose Register 11
   R12=8C      ; General Purpose Register 12
   R13=8D      ; General Purpose Register 13
   SP=8E       ; GPR  - STACK POINTER
```

Example 2-1 Sample Field Value Assignments - I_APORT

MICRO2 interprets each field value definition by equating the symbol on the left of the equal sign to the hexadecimal value on the right. The semicolon character separates the code from the comment text.

Assigning Values to Fields

Once MICRO2 equates hexadecimal values to the field value symbols, the microprogrammers can then specify field value settings symbolically. For example, the following statement encodes a value of 80 (hex) in the I_APORT field:

```
I_APORT/R0
```

Microprogrammers use symbols to represent field values to reduce the amount of coding that would otherwise be required if a hardware or microcode update is made. (The microprogrammers need only modify the DEFIN.MIC file, not all microcode source files.)

2.2.5.2 **Macrodefinition File - MACRO.MIC** - MICRO2 supports the use of macroexpressions, which are single-line, functionally descriptive statements that represent the settings for several microword fields.

By using macroexpressions and the default values for each microword field, the microprogrammers can fully define a microword with only a few statements (reducing the amount of coding required).

Macroexpressions and individual microword field definitions can be used in the same microword.

Macroexpression Definition Format

Macrodefinitions consist of a macroname followed by the microword fields the expression represents. The microword fields are enclosed in quotes and are separated by commas. For example, the macro that instructs the CBox to write a longword of data is defined as follows:

```
WRITE LONG          "C_WCF/WRITE.V.CHECK, C_MSIZE/LONG"
```

When MICRO2 encounters the WRITE LONG macroexpression in a microcode source file, it:

1. Searches MACRO.MIC for the definition of the macro
2. Searches DEFIN.MIC for definitions of the CWCF and the CMSIZE fields
3. Encodes the hex values for the symbols:
    a. WRITE.V.CHECK in the CWCF field
    b. LONG in the CMSIZE field

All other microword fields will be encoded with their default values, values from other macros, or values explicitly supplied by the microprogrammer (individual field value symbols).

Macroexpression Parameters

Microprogrammers can directly specify settings for microword fields by supplying the symbolic values for the fields as parameters in a macroexpression. The only restriction is that the parameters must be valid symbolic value names for microword fields.

MICRO2 recognizes the square bracket ([]) and the "at" (@) characters in macroexpression definitions as parameter indicators. For example, the macrodefinition:

```
READ LONG []            "C_RCF/READ.V.CHECK, C_MSIZE/LONG,
                         C_MREG/@1"
```

informs MICRO2 that the programmers will supply the symbolic value for the C_MREG field each time the macro is used by enclosing the symbol in brackets. The @ character in the macrodefinition associates the C_MREG field with the parameter.

The following is an example of a READ LONG macro that instructs the CBox to read a longword of data and to store the data in memory data register 1 in the EBox.

```
READ LONG [MD1]
```

When MICRO2 encounters the READ LONG [MD1] macro it:

1. Searches MACRO.MIC for the macrodefinition.
2. Searches DEFIN.MIC for the CRCF, CMSIZE, and CMREG field definitions.
3. Encodes the hex value for the symbol:
   a. READ.V.CHECK in the CRCF field
   b. LONG in the CMSIZE field
4. Relates the [MD1] parameter in the macro to the field marked by the @ character - CMREG in this case.
5. Encodes the hex value for the MD1 symbol in the CMREG field.

Macroexpressions can have several parameters. The decimal integer following the @ character in the definition indicates the position of the parameter in the macro. If several parameters are used, the left-most parameter will be designated "@1", the one to the right of that "@2", and so on.

Macroexpression Classes

Macroexpressions are grouped by function in the MACRO.MIC file. For example, all macros that deal with data transfer functions are in one group while all macros that deal with microbranching functions are in another group. The macroexpression classes are shown in Table 2-3.

Table 2-3 Macroexpression Classes

Macrogroup	Controls
Register Transfer	Data transfers through EBox data path. Macros further grouped by ALU, shifter, multiplier, and floating-point (SALU, XALU, PE) functions.
Cache commands	Memory read/write functions and special CBox operations.
CREG/IREG	Data transfers to/from CBox, IBox, and console resident registers.
Microbranch	Microbranch functions.
Miscellaneous	Miscellaneous functions: • Set/clear PSL cc bits • Set/clear CPU state flags • Instruction decoder calls • Subroutine calls/returns • Trap returns • Others

Tables 2-4 through 2-8 list examples of the various macroclasses.

2.2.6 Microcode Related Documentation

For a more detailed description of the VAX 8800 microcode structure, refer to the VAX 8800 Microcode Interpretation Guide (EK-KA88E-UG).

Table 2-4 Sample Register Transfer Macros

Macroexpression	CPU Function
ALU<C> <- A[] + B[]	Set ALU carry bit based on sum of A and B port inputs to ALU. Result is not stored in a register.
F[] <- A[] + B[]	Store sum of A and B port inputs in a fast data RAM register (EBox).
F[] <- A[].SL.[]	Store left shifted A port input in a fast data RAM register. Shift amount is given in the E_SHFTCNT field.
F[]<15-0> <- B[]	Store bits <15:0> from B port into a fast data RAM register.
PC & VA <- A[] FLUSH IB	Load PC and VA registers (EBox) with start address for new I-stream data; initialize the instruction buffer.
S[] <- A[].OR.B[]	Store logical OR of A and B port inputs in slow data RAM register (EBox).
SHFT <- A[]	Load EBox shift count latches from A port input of main ALU.
SHFT & F[] <- B[]	Load shift count latches and slow data RAM register from B port.
VA & F[] & S[] <- A[]	Load VA register, a fast data RAM register, and a slow data RAM register from A port.
VA & WBUS <- A[] + B[]	Load VA register with sum of A and B ports; also output result to WBus.
WBUS <- A[] + B[]	Output sum of A and B ports to WBus.

Table 2-5 Sample Cache Command Macros

Macroexpression	CPU Function
CHECK READ ACCESS	Probe TB for read access.
CHECK WRITE ACCESS	Probe TB for write access.
READ LONG []	Read longword and store the data in an EBox memory data register (MDR). MDR address is given by C_MREG field.
READ LONG MDNUM	As above, except MDR address is given by IBox hardware (selected by I_MDNUM field).
WRITE LONG	Write a longword to memory.
WRITE.UNLOCK LONG	Write a longword and unlock memory.

Table 2-6 Sample CREG/IREG Macros

Macroexpression	CPU Function
CLEAR IBCER	Clear IBox and CBox error registers.
READ CONSOLE STATUS	Read console's transmit and receive control and status registers. (Registers reside on the CLK module.)
READ CREG []	Read a CBox resident register. The register address is given by the C_MREG field.
READ IBER1	Read the IBox error register.
READ RXDB DATA	Read the console's receive data buffer.
WRITE INOP	Write the interrupt other processor register (resident in the IBox).
WRITE IREG []	Write an IBox resident register. Register address is given by the I_MISC field.

Table 2-7 Sample Microbranch Macros

Macroexpression	CPU Function - take microbranch based on:
(ACCESS ALLOWED)	N/A. Example of a pseudo macro. *
? ACCESS ALLOWED ?	Status returned by CBox after a TB access probe check.
? ALU<C> ?	Main ALU <C> bit.
? ALU<C> + WBUS<N> ?	Main ALU <C> bit and WBUS <N> bit.
? FLAG0 ?	CPU state flag 0.
? INT_PEND ?	Interrupt pending flag.
? PSL<C> ?	PSL <C> bit.
? WBUS<3-0> ?	WBUS bits <3:0>.

* Pseudo macros are programming aids microprogrammers use to debug code associated with dynamic microbranch conditions. While the pseudo macros appear in the microcode listing file, they do not generate actual code.

The programmer includes a pseudo macro in a microword if the word is to set up a dynamic microbranch condition to be tested by a later microword. A MICRO2 support program checks all such microword pairs to ensure that the word that tests the condition is at least three CPU cycles removed from the one that set up the condition (microcode pipeline requirement). If not, the support program reports an error when the microcode is assembled.

Note that pseudo macros and true microbranch macros have the same basic format. The only difference is that the pseudo macros are enclosed in parentheses instead of question marks.

Table 2-8 Sample Miscellaneous Macros

Macroexpression	CPU Function
CALL []	Subroutine call. Push address of current microword on microstack, get start address of called routine from I_NEXT field.
CHECK IV	Check for integer overflow trap.
CHECK ROP A.FD	Check for floating-point reserved operand trap; data type is F/D-float.
CLEAR FLAG0	Clear CPU state flag 0.
CLEAR ALL FLAGS	Clear all CPU state flags.
CLEAR TRAP	Clear microtrap. Release trap silos, do not restart trapped microwords (see entry for EXIT TRAP macro.)
END INSTRUCTION	End current macroinstruction, request start decode of next instruction. Next microword address supplied by instruction decoder. decoder.
EXIT TRAP	Return from microtrap. Release trap silos, silos, restart trapped microwords (see entry for CLEAR TRAP macro).
FORCE MDVALID	Mark all EBox memory data registers valid.
GOTO []	Jump to new microroutine. I_NEXT field has starting address of new routine.
GOTO DECODER	Get address of next microword to be executed from the instruction decoder.
NOP	Do nothing.
RETURN []	Return from subroutine. Logically OR address popped from microstack with I_NEXT field.
SETCC []	Set PSL CC bits. Recipe in I_MISC field.
SET FLAG0	Set CPU state flag 0.

2.3 MICROCODE PIPELINING CONCEPTS

VAX 8800 CPU operations are based on a design technique commonly known as microcode pipelining. This section presents the microcode pipeline concepts common to most pipelined machines. This section discusses the characteristics of the VAX 8800 pipeline.

2.3.1 Pipelining Rationale

The major advantage to microcode pipelining is that it enhances the operational speed of a CPU by allowing more than one microword to be active at any given time.

In a pipelined machine, the microcode can control several CPU logic elements simultaneously. For example, the hardware may be instructed to read new data, perform a computation on previously read data, and store the result of a previous computation all at the same time. This more efficient use of the hardware yields a substantial increase in performance over a nonpipelined machine.

2.3.2 Pipelined Versus Nonpipelined Machines

Each microword of a microcode-controlled CPU must, in some manner, perform three basic operations:

1. Read data from memory or from a register
2. Modify the data (add, shift, etc.) if required
3. Write the result to memory or to a register

These operations are generally considered to be performed during what are known as a "microcycles" with each microcycle being one or more CPU clock cycle in duration.

Figure 2-3 illustrates the timing relationship between microwords and microcycles of a nonpipelined CPU. Figure 2-4 does the same for a pipelined CPU. Both figures assume that it takes three microcycles to execute one microword (ignoring microaddress look-up time, data access time, traps, stalls, etc.).

Table 2-8 lists the major differences between the two machine types.

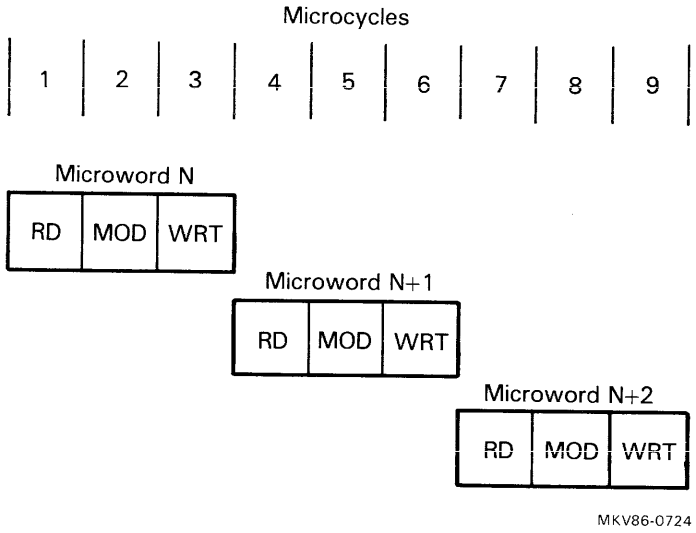


Figure 2-3 Basic Time State Diagram - NonPipelined CPU

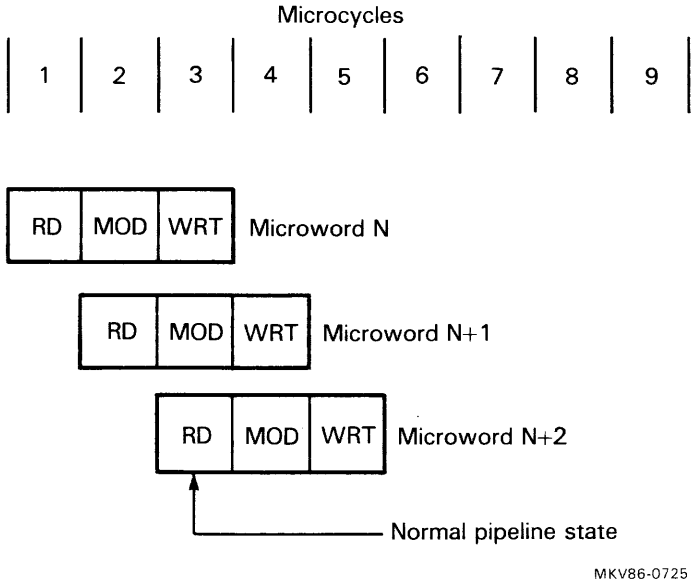


Figure 2-4 Basic Time State Diagram - Pipelined CPU

Table 2-9 Pipelined/Nonpipelined CPU Comparisons

Nonpipelined CPU	Pipelined CPU
New microword started every three microcycles.	New microword started every microcycle.
Serial operation - one operation per cycle.	Parallel operation - three operations per cycle.
Majority of hardware idle in each cycle.	Majority of hardware active in each cycle.

2.3.2.1 Performance Factors - Although it takes the same number of microcycles to execute any given microword in either CPU type, a substantial increase in performance is gained in a pipelined CPU since the operations are "overlapped." For example, with one microword executed every three microcycles, a nonpipelined CPU would take nine cycles to execute a three-microword routine. A pipelined CPU would take only five, a performance factor ratio of 1.8 to 1. As the number of microwords in a routine increases, the performance factor ratio also increases, to a limit of 3 to 1 in the case of a three-stage pipeline.

2.4 VAX 8800 PIPELINE CHARACTERISTICS

Microcode pipelining in the VAX 8800 processor is possible because of the precise timing supplied by the clock subsystem and the latch-based design of the hardware. The primary timing signals, A CLK and B CLK, ensure that data is propagated through the hardware in the correct sequence and at the proper time.

2.4.1 CPU Clock Cycle

The VAX 8800 CPU clock cycle is considered to be the time period from leading edge to leading edge of either the A CLK or the B CLK signal. IBox clock cycles start on a B CLK; CBox and EBox clock cycles start on an A CLK. (Refer to Figure 2-5.)

The nominal clock rate is 45 nanoseconds. The rate can be modified by issuing the SET CLOCK command from the console. (Refer to the VAX 8800 Console User's Guide.)

2.4.2 CPU Hardware Design

The CPU hardware design is based on coupling high-speed, combinational logic elements together with data latches. The logic elements perform the required CPU functions (under microcode control), the data latches serve as buffers between the elements. The data latches are strobed by either the A CLK or the B CLK timing signal.

The hardware design requires data to be transferred between latches strobed by opposite clock phases. For example, data previously stored in an "A" latch can only be transferred to a "B" latch.

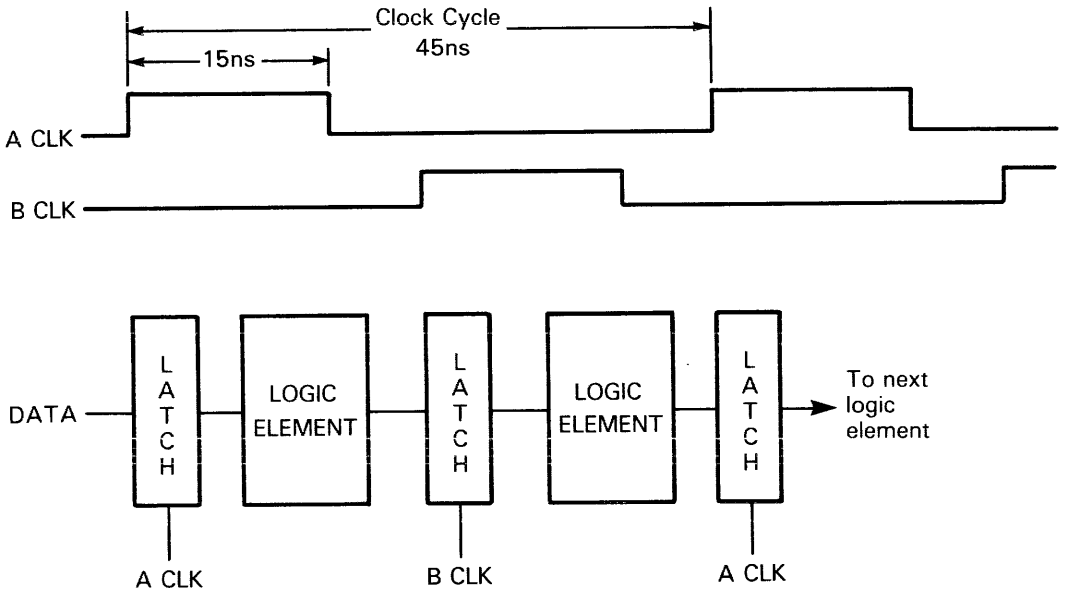
Refer to the lower portion of Figure 2-5. During a single-CPU clock cycle, data can be:

1. Obtained from an A latch
2. Processed by a logic element
3. Clocked into a B latch
4. Obtained from the B latch
5. Processed by another logic element

Once processed by the second logic element, the data is then input to another A latch and made available for the next clock cycle.

NOTE

Clock cycles will be referred to as microcycles in the remaining text.



Note: Cbox and Ebox clock cycles start on an A CLK, Ibox cycles start on a B CLK.

MKV86-0726

Figure 2-5 Basic CPU Timing

2.4.3 Relationship Between Microcycles and CPU Functions

VAX 8800 microwords can take from three to five microcycles to execute (ignoring stalls, traps, etc.). Three of these cycles correspond to the basic functions the CPU must perform (read, modify, and write). The other two cycles are optional and are associated with decoding macroinstructions and performing certain cache operations.

Figure 2-6 and the following table are concerned with the basic read, modify, and write microcycles of a microword.

Cycle	CPU Function
Read	Select the source of data input to the EBox data path (memory, GPR, microcode temporary register, etc.)
Modify	Perform the specified operation on the data (arithmetic, logical, etc.)
Write	Store the result in the specified destination (memory, GPR, microcode temporary register, etc.)

2.4.4 IB Decode Cycle

In addition to the read/modify/write cycles of a microword, there is an optional cycle that is associated with decoding I-stream data from the instruction buffer.

The IB decode cycle precedes the three basic microword cycles in time, but is only used during macroinstruction execution. Otherwise, it is effectively treated as a "no-op" cycle by the hardware.

NOTE

Figure 2-7 is for illustration purposes only. Refer to Figure 2-9 for a more precise representation.

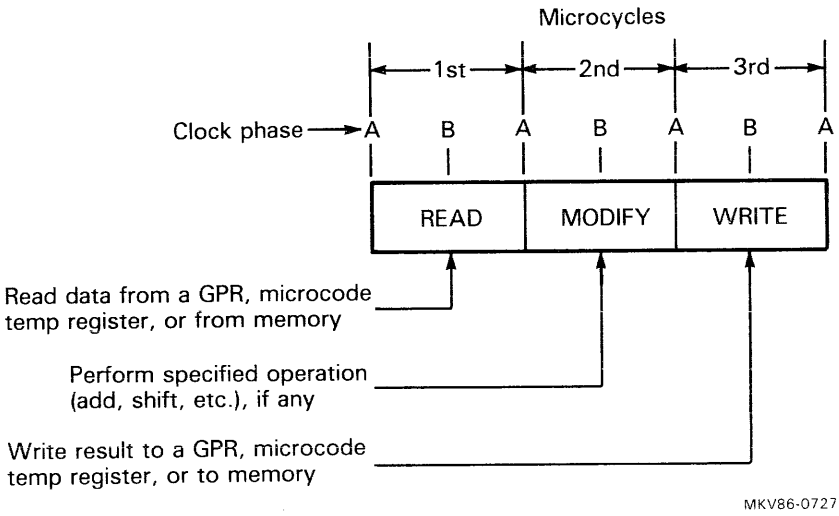


Figure 2-6 Microcycles/CPU Functions

2.4.5 Canonical Time States

Since the microcycles of the microwords in the pipeline overlap, the time at which a CPU event occurs in a given microword is referred as its canonical time state.

NOTE

The term "canonical" with reference to the VAX 8800 CPU refers to the set of rules used by the hardware design engineers.

2.4.5.1 Definition Of A Canonical Time State - A canonical time state is defined as the time period from the leading edge of one clock phase to the leading edge of the next opposite phase. That is, the time periods from A CLK to B CLK and from B CLK to A CLK are both considered canonical time states.

Note in Figure 2-8 that microcycles are divided into two time states: even numbered time states start on an A CLK pulse, odd numbered time states start on a B CLK pulse.

2.4.5.2 Overlapping Time States - Figure 2-8 also shows that each microword has its own canonical time state periods with each time state being one cycle removed from the corresponding time state of the next microword. For example, T10 time of the first (top) microword in Figure 2-8 is also T8 of the second microword, T6 of the third, and so forth.

With several microwords in process at a time, to avoid confusion, the hardware designers always refer to the timing of CPU events relative to the microword just entering the pipeline.

NOTE

Canonical T0 time is shown for reference only. T0 is considered to be T2 of the previous microword.

It is assumed that the address of the first microword shown in Figure 2-8 was loaded into the microPC address latches by the console. The address of each subsequent microword is generated during canonical T4 time of the previous microword (see Section 2.4.6).

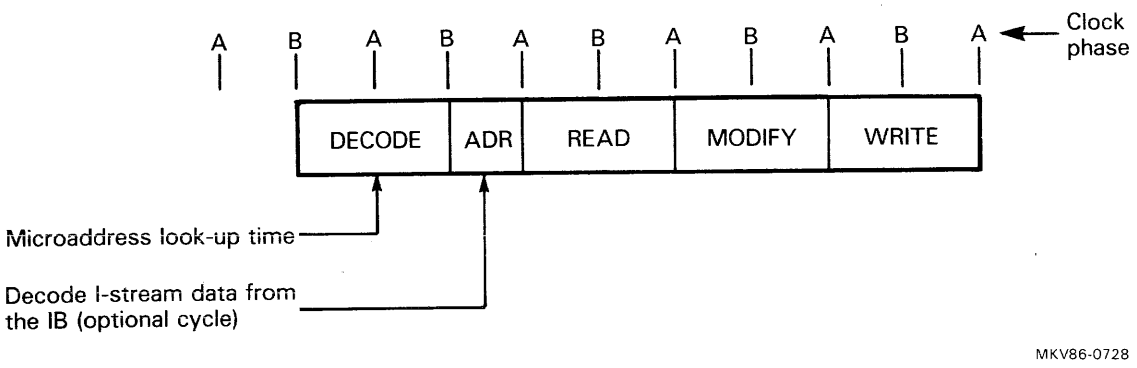


Figure 2-7 IB Decode Cycle

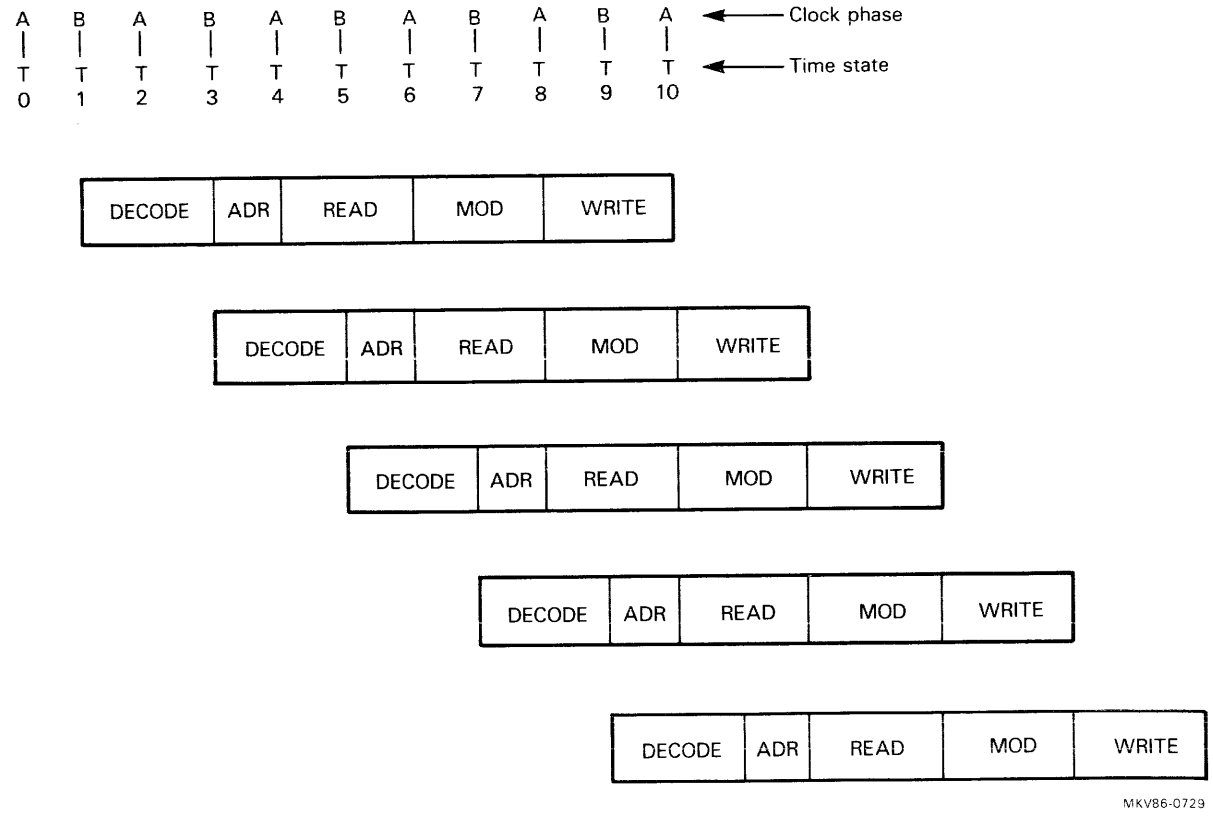


Figure 2-8 Canonical Time States

2.4.6 Time State Events

A microword is considered active in the pipeline from canonical T3 to T10 except when the CPU is executing a macroinstruction. In this case, canonical T0 to T2 are included.

Figure 2-9 illustrates the CPU event timing for several microwords in the pipeline. Table 2-9 briefly describes the events that occur in each time state. The time states given in the table are relative to the top microword shown in the figure. It is assumed that the CPU is in the process of executing a macroinstruction.

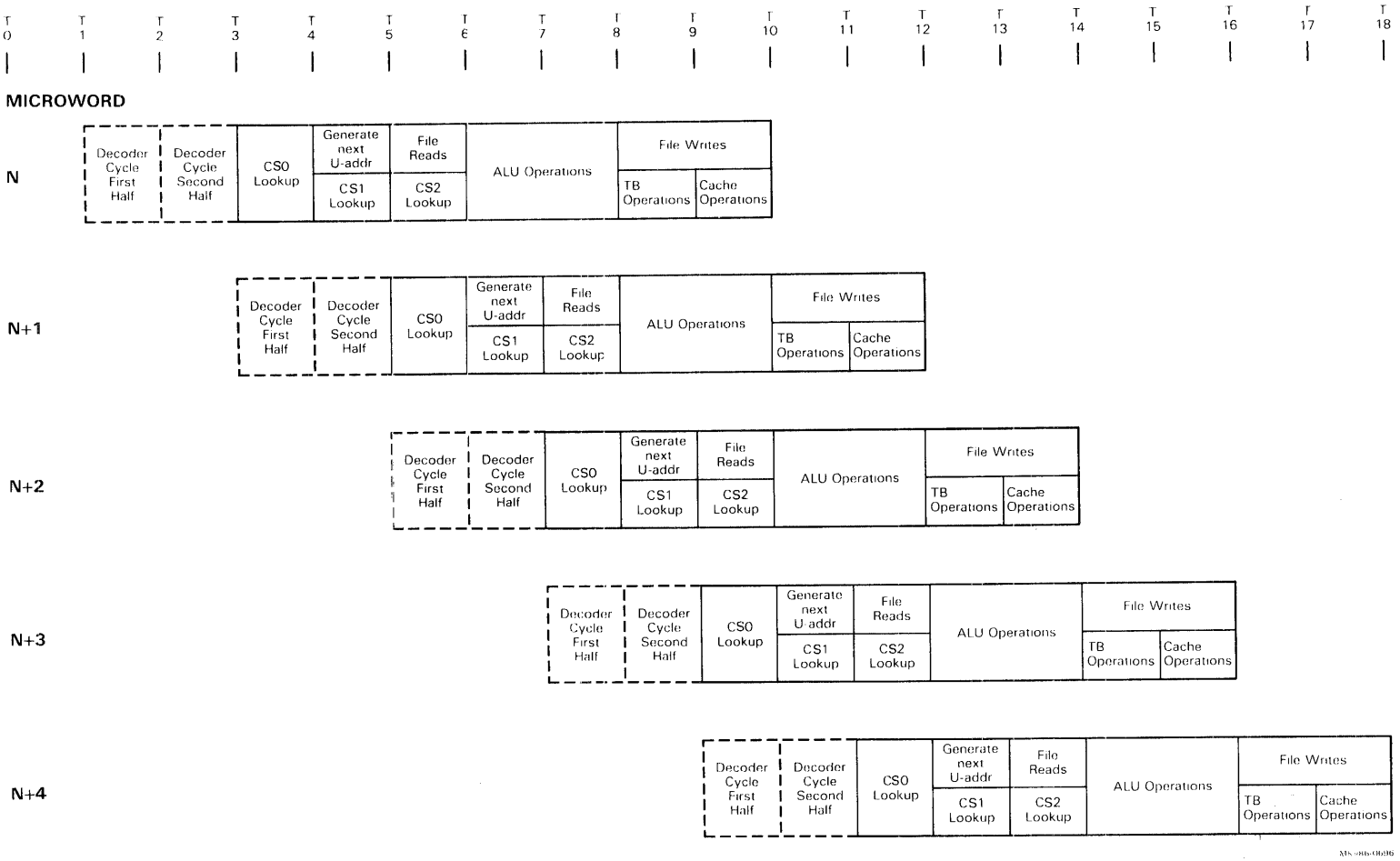


Figure 2-9 VAX 8800 Pipeline Time State Diagram

Table 2-10 Pipeline Time States/CPU Events

Time State	CPU Events
T0 to T1	IB checks if it can accept a new longword of I-stream data. If so, IB decodes its write address in preparation for the longword. (Note: canonical T0 time is generally considered to be T2 time of the previous microword.)
T1 to T2	IB outputs op code and next specifier (if any) of the macroinstruction currently being processed to the decoder. New longword of I-stream data enters IB if the IB write operation is enabled.
T2 to T3	Decoder generates entry-point address for micro-routine required to service the specifier. If there are no more specifiers, the IB decoder generates the entry-point address for the op code routine.
T3 to T4	IB decoder generated microaddress latched in the microPC address latches for the CS0 RAMs (first of three control store RAM segments). CS0 RAMs start outputting microword bits for the read cycle of the microword.
T4 to T5	Current microaddress passed to microPC address latches for CS1 RAMs (second CS RAM segment). CS1 RAMs start outputting microword bits for the modify cycle of the microword. CS0 RAM microword bits latched and routed to the appropriate EBox and IBox logic. Microsequencer examines microword bits <26:00> from CS0 segment to determine address of next microword to be executed.
T5 to T6	Current microaddress passed to microPC address latches for CS2 RAMs (third CS RAM segment). CS2 RAMs start outputting microword bits for the write cycle of the microword.

Table 2-10 Pipeline Time States/CPU Events (Cont)

Time State	Event
T5 to T6 (cont'd)	CS1 RAM microword bits latched and routed to the appropriate EBox logic. CS0 RAM microword bits select source of data to be input to EBox data path. Microaddress of next microword to be executed is latched in microPC address latches of CS0 RAMs; new microword started.
T6 to T7	CS0 segment completed. CS1 segment controls the first half cycle of the EBox data path operation (main ALU, multiplier/ divider, shifter, etc.). CS2 RAM microword bits latched and routed to the appropriate CBox, EBox, and IBox logic elements.
T7 to T8	CS1 segment completed. CS2 segment controls second half cycle of EBox data path operation. CBox receives and decodes cache command (if any) supplied by CS2 segment.
T8 to T9	CBox performs any virtual to physical address translation required. (Note: This operation is controlled by the CBox, not by microcode.)
T9 to T10	CS2 segment controls EBox register writes. CBox performs cache read/write functions (if any).
T10 to T11	CBox outputs data to EBox if memory read request was made. (CBox generates a stall condition if the required data is not in cache and must be retrieved from main memory.) Pending microtrap conditions (if any) signaled to IBox microtrap logic.

CHAPTER 3
IBOX FUNCTIONAL DESCRIPTION

3.1 CHAPTER SCOPE

This chapter describes the operation of the IBox hardware to the block diagram level. It includes functional block diagrams of each major IBox logic section and discusses how the hardware interacts with the various microword fields where applicable.

Certain sections of this chapter, such as the one that discusses the macroinstruction decode process, include more detailed block diagrams to better illustrate the operation of the hardware.

To understand the interaction between the hardware and the microcode, it is recommended that the following VAX 8800 reference material be available to the reader:

- Microcode Listings
- Microcode Interpretation Guide, EK-KA88E-UG
- Machine Check Interpretation Guide, EK-KA88H-UG

3.2 CONTROL STORE LOGIC

The control store logic resides on the SEQ and the WCS modules. It consists of the 16K by 1 bit writeable RAMs, which contain the main CPU microcode, and the necessary RAM address and data latches.

Figure 3-1 is a simplified block diagram of the control store logic.

3.2.1 Control Store RAM Segments

To implement the microcode pipeline effect, the control store RAMs are physically divided into three segments known as CS0, CS1, and CS2. The CS0 segment resides on the SEQ module, the CS1 and CS2 segments both reside on the WCS module.

Each CS RAM segment corresponds to a different cycle of the microword. CS0 supplies the microword bits for the read cycle, CS1 for the modify cycle, and CS2 for the write cycle. The following table indicates the primary CPU functions controlled by each segment.

Table 3-1 Control Store RAM Segment Functionality

Segment	U-Word Bits	CPU Functions
CS0	<047:000>	Next micro-PC address formation. Microstack operations. Register file reads.
CS1	<095:048>	Data manipulation operations. Register file write set-up.
CS2	<142:096>	Register file writes. Cache and TB operations. Condition code setting. Miscellaneous operations.

Refer to Chapter 2 for descriptions of the microword bits supplied by each RAM segment.

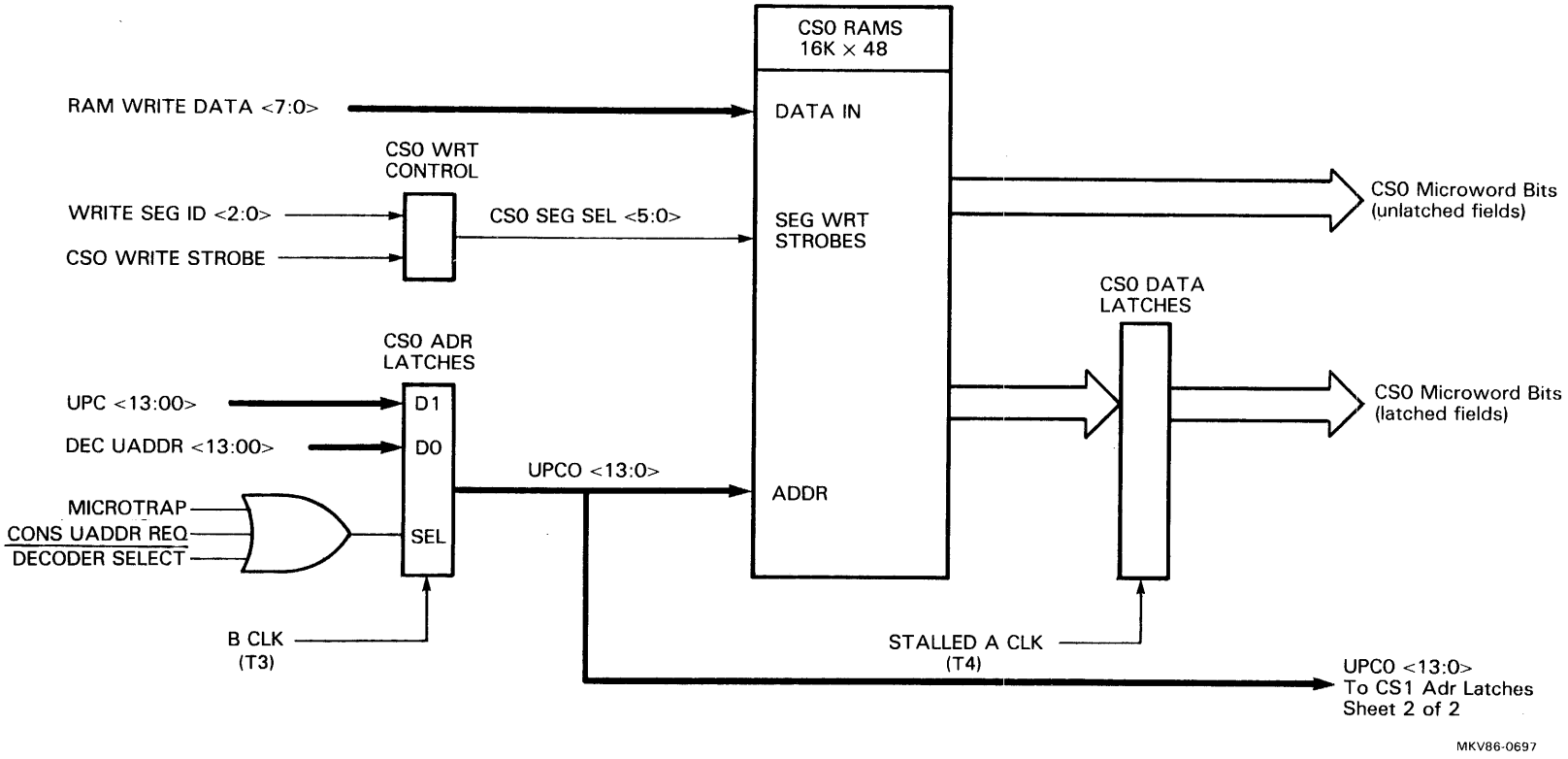


Figure 3-1 Control Logic Simplified Block Diagram (Sheet 1 of 2)

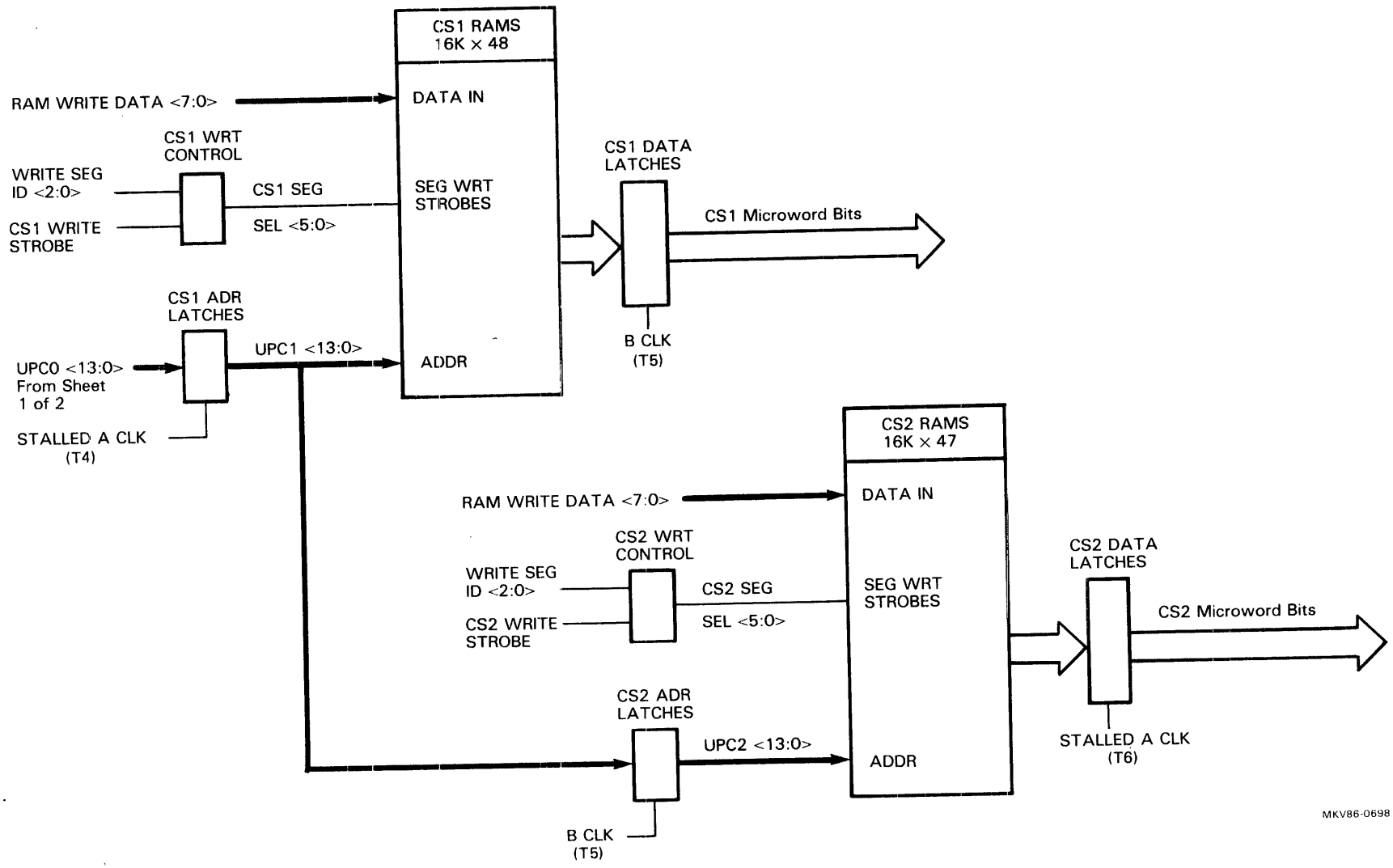


Figure 3-1 Control Logic Simplified Block Diagram (Sheet 2 of 2)

3.2.2 Control Store RAM Addressing

Refer to Figure 3-1. The pipeline effect is a direct result of the way in which the address of the next microword to be executed is propagated through the address latches of the CS RAM segments.

The address of the next microword to be executed is derived from one of two sources:

Table 3-2 Next Microaddress Sources

Source Logic	Signals	Comment
IB Decoder	DEC UADDR <13:0>	Entry point microaddresses for microroutines that: o Process operand specifiers o Execute macroinstructions o Service special conditions
Microsequencer	UPC <13:0>	All other microaddresses.

The state of the DECODER SELECT input to the CS0 RAM address latches is the primary factor in determining whether the IB decoder or the microsequencer is to supply the next microaddress. If the signal is asserted, the decoder is selected. Otherwise, the microsequencer is selected.

The selected microaddress is loaded in the CS0 RAM address latches at canonical T3 of the microword (CS0 lookup cycle). The address is then propagated to the CS1 address latches at T4 (CS1 lookup) and to the CS2 address latches at T5 (CS2 lookup).

3.2.3 Control Store RAM Data Latches

The microword bits output by each CS RAM segment are clocked into data latches one canonical Tn time after the segment is read. For example, the CS0 microword bits are read at T3 time and latched at T4. Once stored in the data latches, the microword bits are then routed to the appropriate CPU logic elements.

Note that microword bits <25:00> from the CS0 segment are not latched in the CS0 data latches. These bits all deal with the generation of the next microaddress and are fed directly to, and latched in, the microsequencer logic. This ensures that the microsequencer will be able to compute the next microaddress during canonical T4 and have it ready for the CS0 address latches by canonical T5 (T2 and T3 relative to the next microword).

3.2.4 Loading The Control Store RAMs

The microcode resides on the console's Winchester disk and is loaded into the CS RAMs during system initialization. Figure 3-3 shows the CS RAM load path.

NOTE

> The CS RAM load process is covered in the console section of this document. Only the highlights of the process are presented here.

The RAMs are loaded a byte at a time from the 8-bit bidirectional Cons Bidi Data bus which links the CLK module to the DEC module. The console interface logic on the CLK module controls the load process by passing commands, addresses, and data over the Cons Bidi Data bus to the gateway control (GWYC) MCA on the DEC module.

There are three basic steps involved with loading the CS RAMs:

1. Write address of the microword to be loaded in the micromatch register of the UBRS MCAs.
2. Write data to the selected microword a byte at a time over the Cons Bidi Data bus.
3. Verify parity of each microword loaded.

3.2.4.1 Load Control Store Microaddress - Since microaddresses are 14 bits wide and the Cons Bidi Data bus is only 8 bits wide, the microaddresses are loaded in the micromatch register in slices as shown in Figure 3-2.

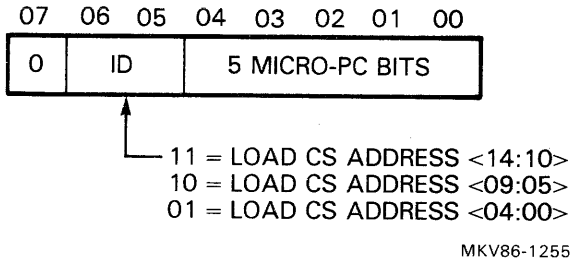


Figure 3-2 Microaddress Bit Slices For Micromatch Register Loading

The ID field points to the appropriate microaddress bit slice in the micromatch register. The address may be sent in any order and need only be set up once for each microword. The address remains in the micromatch register and is clocked through all CS RAM address latches. Note that bit 5 of the top address slice, which would become address bit 14, is not used.

3.2.4.2 Write Data To Selected Address - Data for each microaddress is loaded into the CS RAM segments in the following order:

1. CS0 RAMs
2. CS1 RAMs
3. CS2 RAMs

The CS RAM segments are divided into 6 banks of 8 RAM chips each. The CS0, CS1, and CS2 WRITE STROBE signals select the segment to load, the WRT SEG ID <2:0> signals specify the bank within the segment. The GWYC MCA supplies the control signals.

The CS RAMs are loaded a byte at a time, most significant byte first, from the Cons Bidi Data bus. Data is loaded to the selected CS RAM location with 18 consecutive writes to the Cons Bidi Data bus:

First	Second	Third
CS0	CS1	CS2
<047:040>	<095:088>	<142:136> (7 bits)
<039:032>	<087:080>	<135:128>
<031:024>	<079:072>	<127:120>
<023:016>	<071:064>	<119:112>
<015:008>	<063:056>	<111:104>
<007:000>	<055:048>	<103:096>

Note that if a RAM data segment is less than 8 bits wide, the most significant bits are "don't cares".

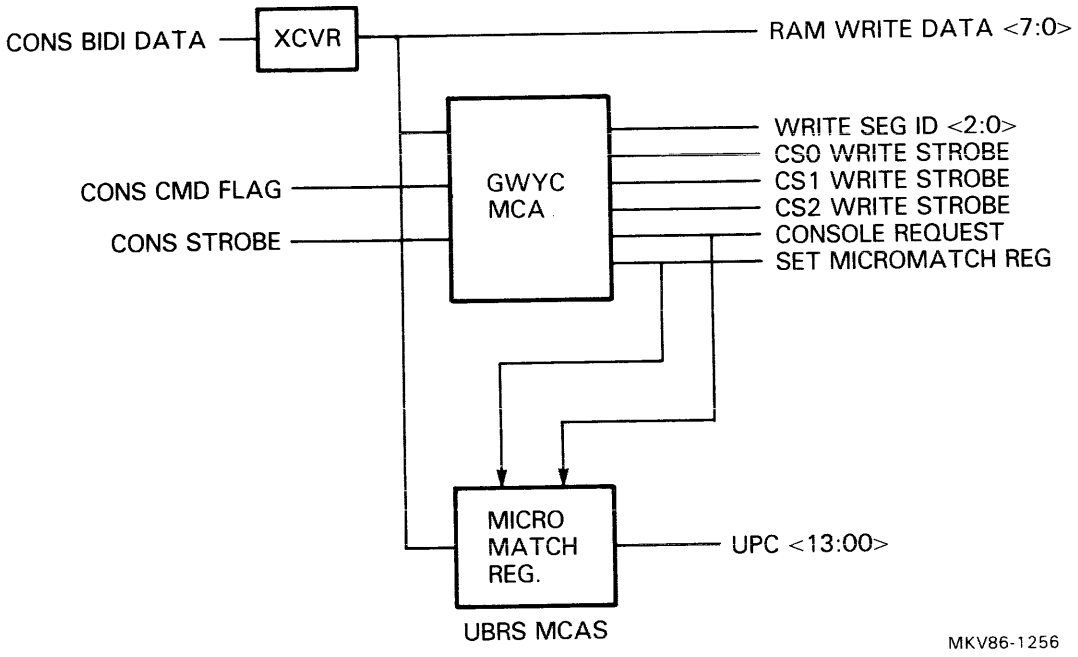


Figure 3-3 Control Store RAM Load Path

3.3 MICROSEQUENCING

Each microword contains several fields that either contribute to the formation of the next microaddress or inform the microsequencer when to select one of several alternate address sources. This section overviews the microsequencer hardware and describes the various microsequencing methods.

3.3.1 Microsequencer Hardware

Refer to Figure 3-4. The microsequencer consists of 5 identical microbranch slice (UBRS) MCAs, a microtrap (UTRP) MCA, and a 16 word by 15 bit microstack.

3.3.1.1 Microbranch Slice (UBRS) MCAs - The UBRS MCAs are responsible for determining the address of the next microword to be executed and for supplying the address to the CS0 RAM address latches. The next microaddress may be derived from any of the following sources:

- Current microword
- IB Decoder
- Microstack
- Microtrap logic
- VAX Console

The UBRS MCAs multiplex all address sources except the one from the IB decoder. The address from the decoder and the one from the UBRS MCAs are multiplexed by the CS0 RAM address latches.

Each UBRS MCA handles a 3 bit slice of the microaddress, a total of 15 address bits. The low 14 bits become the UPC <13:00> lines fed to the CS0 RAM address latches, the 15th bit is unused.

3.3.1.2 Microtrap (UTRP) MCA - The UTRP MCA receives and prioritizes all CPU microtrap conditions and supplies the UBRS MCAs with the microvector of the highest priority condition present. It also generates the address and control signals for the microstack, and notifies the rest of the CPU hardware when a microtrap has occurred.

3.3.1.3 Microstack - The microstack supports the subroutine call and return functions of the microcode. Subroutines may be nested to a depth of 15 calls, after which the microstack wraps back. The microstack operates on a last-in/first-out basis which means that the last address pushed onto the stack is the first one popped off.

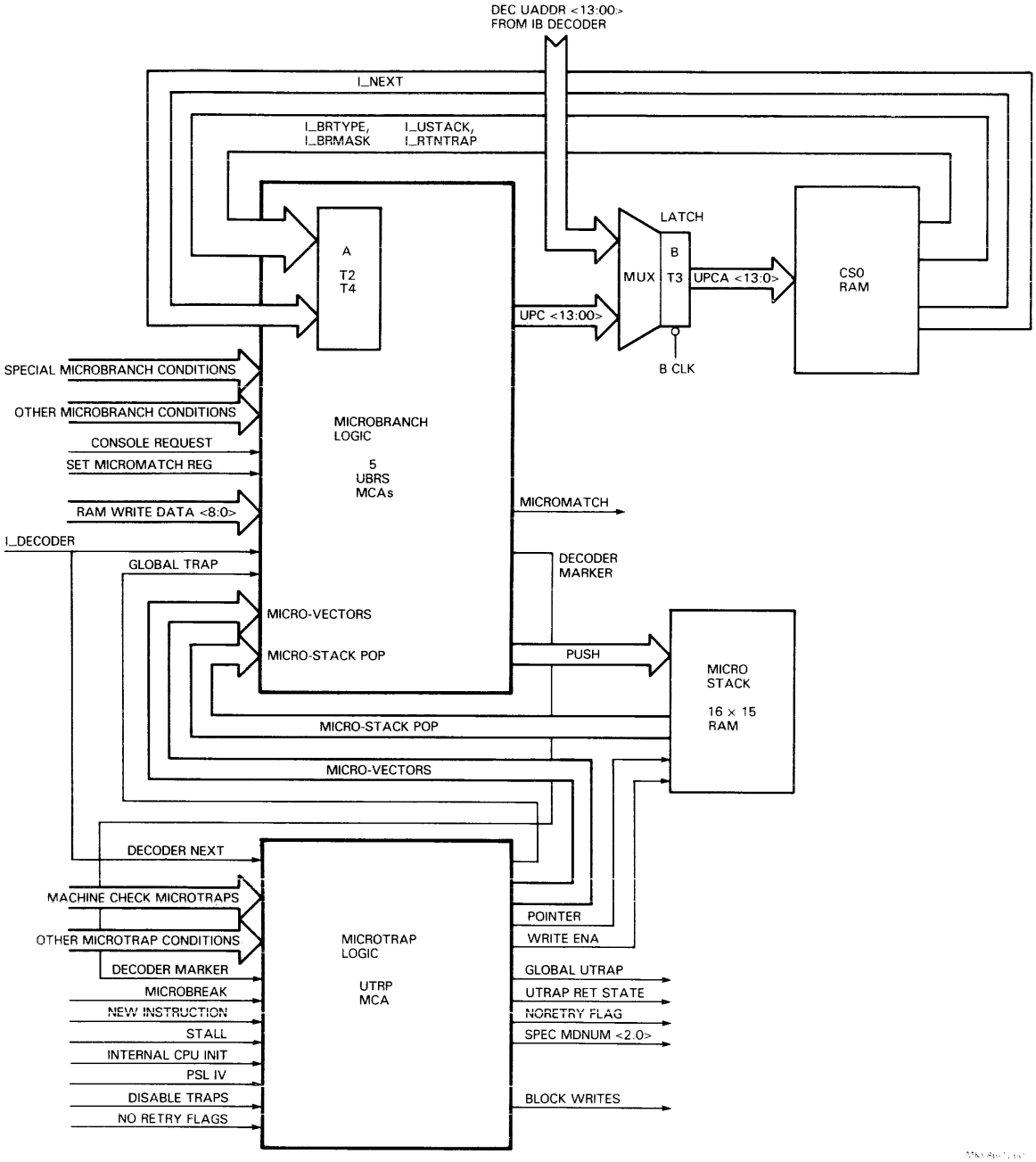


Figure 3-4 Microsequencer Logic

3.3.2 Normal Microcode Flow

The microsequencer normally selects the I_NEXT field of the current microword as the next microaddress unless instructed to do otherwise (modify the address based on a microbranch condition, pop an address from the microstack, select a microtrap vector, etc.).

Refer to Figure 3-5. The I_NEXT field is read from the CS0 RAM segment at canonical T3 of the current microword and is latched in the UBRS MCAs at T4. During the T4 to T5 time frame, the UBRS MCAs check the other microword fields and various control signals to determine if the I_NEXT field is to become the next microaddress. If so, the UBRS MCAs will output the I_NEXT field as UPC <13:00> to the CS0 RAM address latches prior to T5 time (T3 of the next microword).

3.3.3 IB Decoder Supplied Microaddress

The IB decoder is responsible for supplying the entry point (first) microaddress for each microroutine that processes an operand specifier of the current macroinstruction. If more than than one microword is required to service a specifier, the microsequencer will generate the additional addresses for the specifier routine. Once all specifiers are processed, the decoder will supply the entry point address for the execute code of the instruction.

Note that the CS0 RAM address latches will only select the IB decoder supplied address if the I_DECODER bit of the current microword is set. Otherwise, they will select the microsequencer supplied address.

The instruction decode process is covered later in this chapter.

3.3.4 Microbranching

Refer to Figure 3-6 and Table 3-3.

The I_BRTYPE and I_BRMASK microword fields, in combination, allow the UBRS MCAs to test up to five CPU hardware conditions at a time and to modify the I_NEXT field based on the result of the test. The state (one or zero) of the tested conditions are ORed with the low order 5 bits of the I_NEXT field, allowing the UBRS MCAs to generate up to 32 possible target microaddresses.

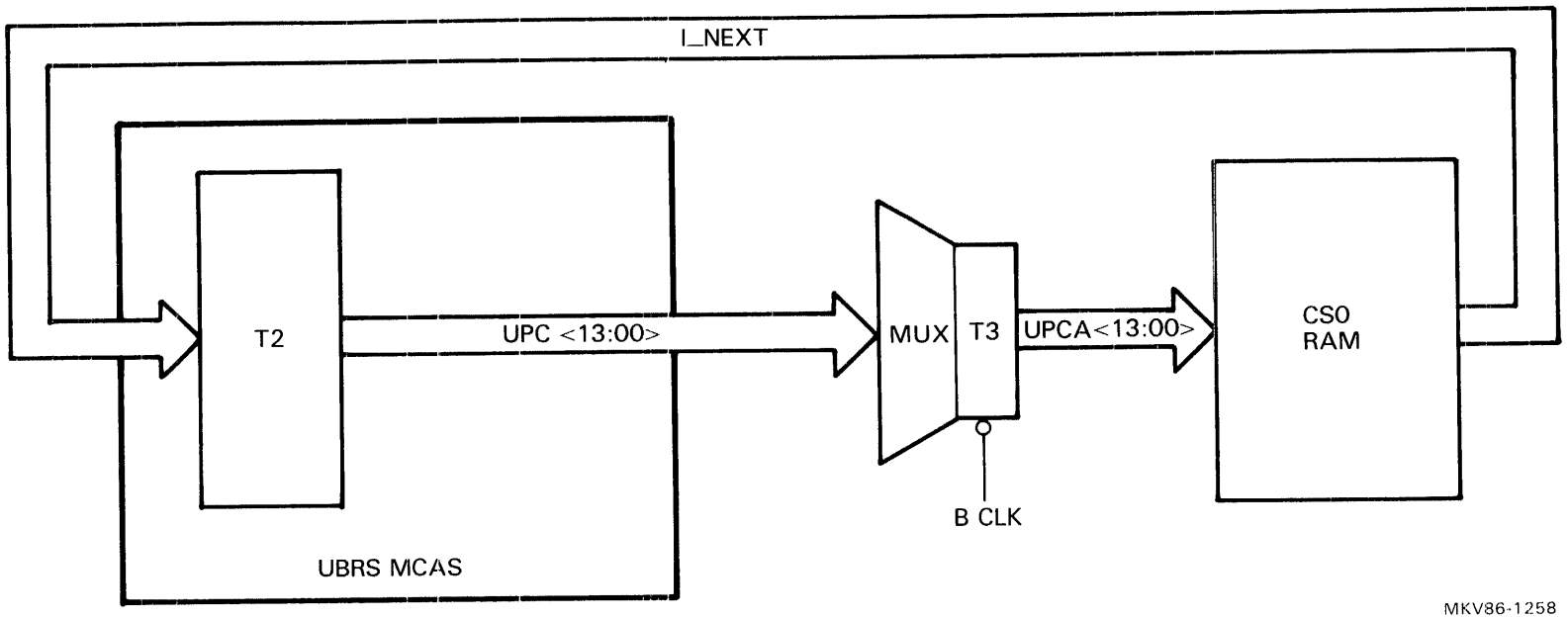


Figure 3-5 Normal I_NEXT Field Addressing

Microbranch conditions are divided into 16 groups of 5 conditions each (some groups have less than 5 conditions). Each UBRS MCA monitors up to 16 conditions, one per group, and is responsible for modifying one of the 5 branch-sensitive I_NEXT field bits. For example, UBRS MCA slice 0 handles I_NEXT bit <0>, slice 1 handles I_NEXT bit <1>, and so on.

Each UBRS MCA receives all four I_BRTYPE microword field bits and one masking bit from the I_BRMASK field. The I_BRTYPE field specifies the microbranch condition group to be selected by all five UBRS MCAs. The masking bit determines if the condition tested by a given UBRS MCA is to affect the branch-sensitive I_NEXT bit handled by the MCA.

Note that to implement a microbranch, the microprogrammers ensure that the appropriate branch-sensitive bits of the I_NEXT field are zeros. For example, to implement a full 32-way microbranch, all 5 low order bits of the field must be zeros.

3.3.4.1 Microbranch Conditions - Microbranch conditions are generated by all CPU logic units, including the IBox itself. Table 3-4 lists the various microbranch conditions and the branch-sensitive I_NEXT bit associated with each condition.

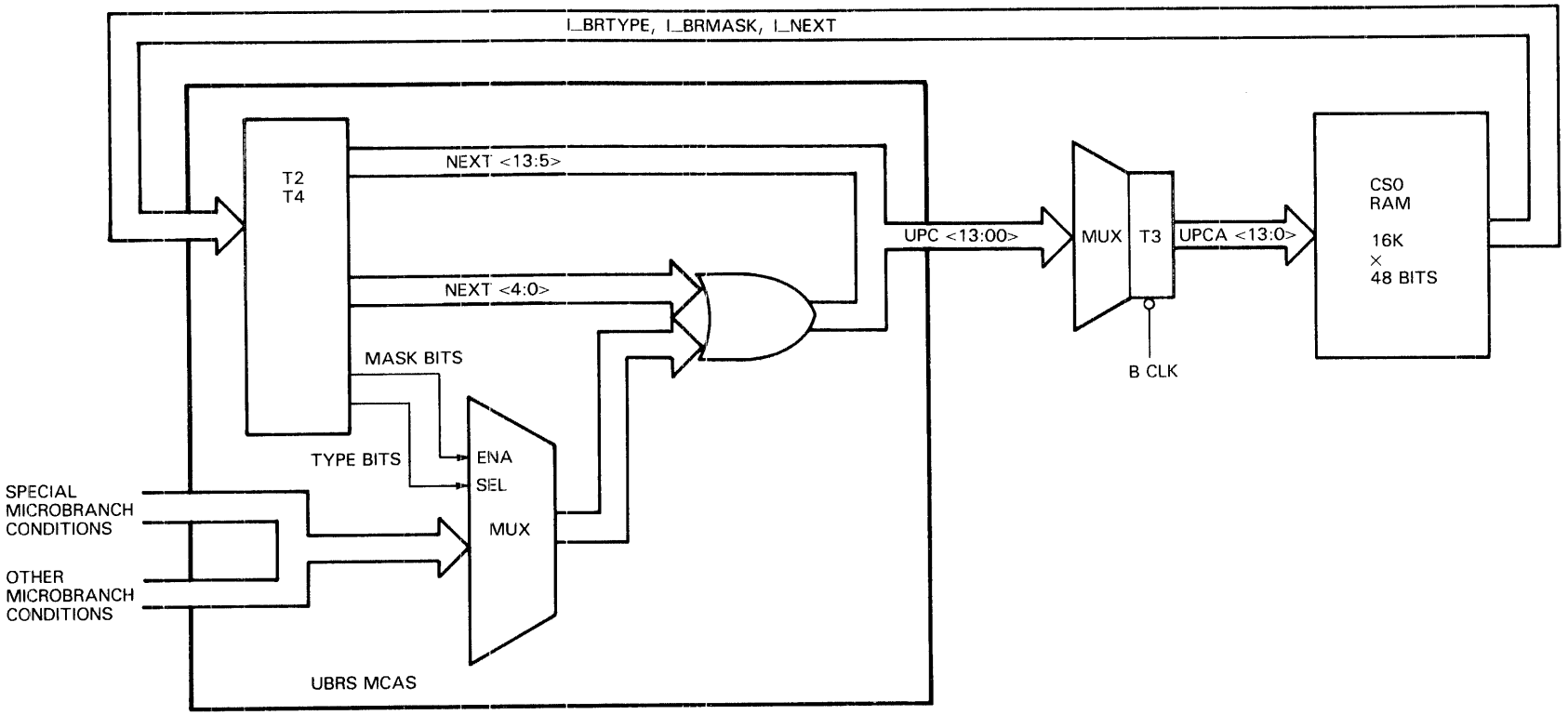
Size Dependent Conditions

Some EBox generated microbranch conditions are intermediate condition code bits that are based on the size of the data being processed. The EBox outputs these microbranch conditions as either WBUS <N,Z,C,V> or as ALU <N,Z,C,V>.

To ensure that the EBox data path generates the correct size dependent microbranch conditions, the data size must be specified in the I_SIZE field of the same microword that produces the conditions. The table below indicates the values for the I_SIZE field.

I_SIZE <1:0>	Data Size
0 0	Unused
0 1	Byte
1 0	Word
1 1	Longword

Size dependent microbranch conditions are denoted by the asterisk (*) character in Table 3-4.



NOTE: EACH UBRS MCA HANDLES ONE BIT OF THE FIVE BRANCH-SENSITIVE I_NEXT FIELD BITS. THE I_BRMASK FIELD SELECTS WHICH UBRS MCAS ARE TO BE ENABLED.

MKV86-1218

Figure 3-6 Microbranch Condition Selection

Table 3-3 I_BRTYPE/I_BRMASK Microword Field Relationship

Specifies the UBRS MCA(s) to be enabled (one or more).

Specifies the microbranch condition group under test (1 of 16).

I_BRMASK <4:0>	I_BRTYPE <3:0>	Microbranch Operation	Code (Hex)
1 1 1 1 1	N/A	N/A, use I_NEXT field	1FX
1 1 1 1 0		Enable UBRS MCA # 0	1EX
1 1 1 0 1	ANY	1	1DX
1 1 0 1 1	COMBI-	2	1BX
1 0 1 1 1	NATION	3	17X
0 1 1 1 1		Enable UBRS MCA # 4	0FX
	0 0 0 0	Select branch group 0	XX0
	0 0 0 1	1	XX1
	0 0 1 0	2	XX2
	0 0 1 1	3	XX3
	0 1 0 0	4	XX4
	0 1 0 1	5	XX5
ANY	0 1 1 0	6	XX6
COMBI-	0 1 1 1	7	XX7
NATION	1 0 0 0	8	XX8
	1 0 0 1	9	XX9
	1 0 1 0	A	XXA
	1 0 1 1	B	XXB
	1 1 0 0	C	XXC
	1 1 0 1	D	XXD
	1 1 1 0	E	XXE
	1 1 1 1	Select branch group F	XXF

XX denotes any combination (except 1FX)

Table 3-4 Microbranch Conditions

Key: @ - dynamic condition
* - size dependent condition

Microbranch condition group - selected by I_BRTYPE field

I_NEXT bits to be modified - selected by I_BRMASK field

	<4>	<3>	<2>	<1>	<0>
0	STATE FLAG <5>	STATE FLAG <1>	STATE FLAG <0>	SALU CC<5> @	XALU CC @
1	TB STATUS <1> @	TB STATUS <0> @	* ALU <C> @		
2	* WBUS <Z> @	WBUS <3> @	WBUS <2> @	WBUS <1> @	WBUS <0> @
3	SALU CC<0> @	SALU CC<1> @	SALU CC<2> @	SALU CC<3> @	* WBUS <N> @
4	PSL <N>	PSL <Z>	PSL <V>	PSL <C>	PSL <TP>
5	INTR PENDING		SALU CC<2> @	XALU CC @	* WBUS <Z> @
6	PSL <FPD>	PSL <CUR1>	PSL <CUR0>	HALT PENDING	ILLEGAL OP
7	WRITE CHK (C_SCF<7>)	WRITE (C_SCF<13>)	MD NO <2>	MD NO <1>	MD NO <0>

Table 3-4 Microbranch Conditions (Cont)

Key: @ - dynamic condition
* - size dependent condition

Microbranch condition group - selected by I_BRTYPE field

I_NEXT bits to be modified - selected by I_BRMASK field

	<4>	<3>	<2>	<1>	<0>
8	CACHE CMD<4> (C_SCF<7>)	CACHE CMD<3> (C_SCF<10>)	CACHE CMD<2> (C_MSIZE<2>)	CACHE CMD<1> (C_MSIZE<1>)	CACHE CMD<0> (C_MSIZE<0>)
9	STATE FLAG <5> @	PE CC @	* ALU <C> @	* WBUS <N> @	* WBUS <Z> @
A	* ALU <V> @	STATE FLAG <4>	AC LOW	DIGIT VALID @	VALID <3> @
B	WBUS <31> @	WBUS <30> @	WBUS <29> @	WBUS <28> @	WBUS <27> @
C				NORETRY FLAG @	* WBUS <Z> @
D	SALU CC <2> @	SALU CC <3> @	SALU CC <4> @	SALU CC <5> @	STATE FLAG <5>
E	INTR ID <4>	INTR ID <3>	INTR ID <2>	INTR ID <1>	INTR ID <0>
F	STATE FLAG <6>	STATE FLAG <3>	STATE FLAG <2>	VA REG<31>	VA REG<30>

Special Microbranch Conditions

Microbranch conditions in groups 7 and 8 (I_BRTYPE field equal to 7 or 8) are used exclusively by microtrap service routines that handle memory management related microtraps (TB miss, TB ACV, etc.). This includes memory management microcode and certain sections of the interrupt and exception microcode.

All special microbranch conditions, except for the low three conditions in group 7, are latched in the UBRS MCAs when a microtrap is detected and are saved in the MCAs for the duration of the trap service routine.

The low three group 7 conditions are copies of TRAP MD <2:0> bits from the CBox. The TRAP MD <2:0> bits are latched and saved in the CBox on a microtrap (refer to the CBox section of this document). The top two group 7 and all group 8 conditions are copies of certain bits from the C_SCF field of the trap causing microword.

NOTE

> The C_SCF field is read from the CS2 RAMs at canonical T6 of the trap causing microword, but the UBRS MCAs do not check microbranch conditions until T10 (a pipeline restriction). The C_SCF bits are propagated through additional latches in the UBRS MCAs to accommodate for the timing discrepancy.

The following table briefly describes how microtrap service routines use the special microbranch condition bits.

Table 3-5 Special Microbranch Condition Bit Usage

Group	Bits	Used by Microtrap Service Routine to:
7	<2:0>	Restore the pointer to the EBox memory data register (MDR) that was to receive memory read data prior to the microtrap.
7	<4:3>	Determine if the command originally issued to the CBox by the trap causing microword was a write or write check. The microtrap handler must know the command type to call the proper subroutine to fix the fault.
8	<4:0>	Perform a 32-way microbranch based on the original command issued to CBox on exiting from the trap (requeues the cache command).

State Flags

The IBox contains 7 CPU state flags which provide the microprogrammers with more flexibility in controlling microcode flow. The flags reside in the CCBR MCA and can be set (individually) or cleared (individually or as a group) as specified by the I_MISC field. The state flags are written at canonical T9.

The IBox hardware clears all state flags during the first microword of every macroinstruction. The flags can then be set (or cleared) by one microroutine and tested as microbranch conditions, after a minimum 3 cycle branch latency period, by a later routine.

Note that since the state flags are hardware cleared during the first microword of a macroinstruction, the I_MISC field of this microword may not be encoded to set a state flag (microcode restriction).

Noretry Flag

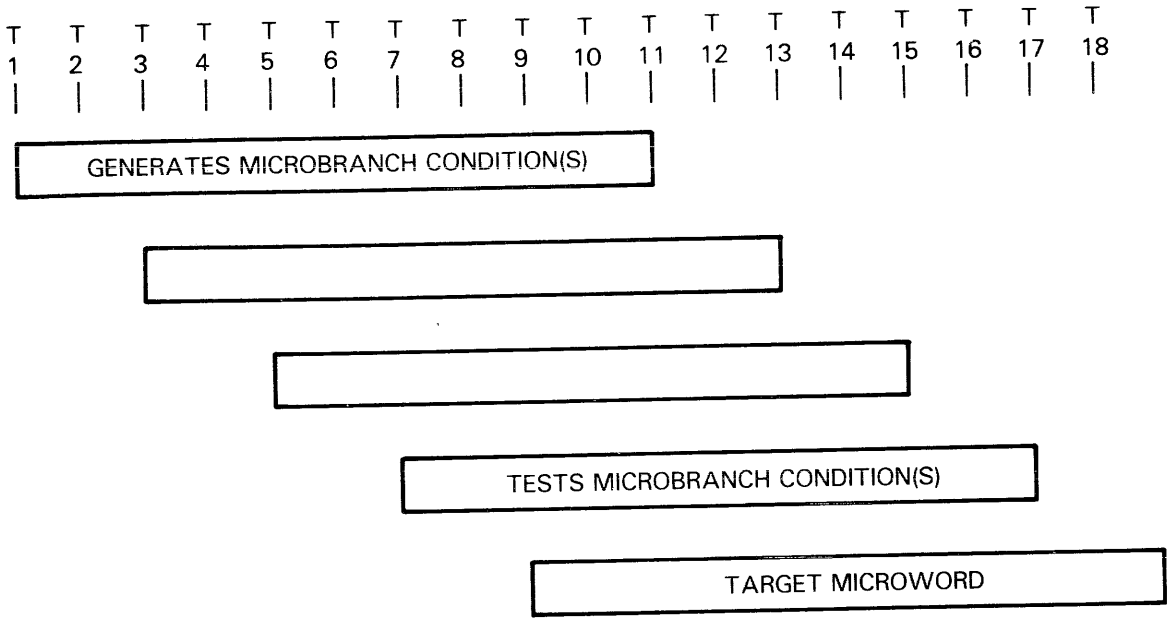
Microprogrammers use the special NORETRY flag to inform machine check interrupt service routines whether a macroinstruction can be restarted following a machine check fault. The machine check code examines the noretry flag, and information from other sources, to determine if the instruction that caused a fault can be restarted after restoring the PC and any modified GPRs.

The noretry flag is hardware cleared during the first microword of every macroinstruction. It must then be set by the first microword of the instruction that performs an operation that could prevent the instruction from being properly restarted following a fault.

Examples of non-retryable operations include writes to GPRs and IPRs (except GPR auto-increment/decrement) and memory writes. Note that writes to GPRs during auto-increment/decrement operations are backed up by RLOG registers in the FADS MCAs and are, therefore, retryable.

The noretry flag resides in the UTRP MCA and is set at canonical T9 by the I_NORETRY field.

3.3.4.2 Microbranch Latency - Due to the pipeline effect, there is normally a 3 cycle delay before microbranch conditions generated by a microword are available to the UBRS MCAs for testing. Figure 3-7 illustrates the pipeline latency for conditional microbranching.



NOTES:

1. ALL MICROBRANCH CONDITIONS ARE AVAILABLE TO THE UBRS MCAS AT CANONICAL T10 RELATIVE TO THE CONDITION PRODUCING MICROWORD (T4 RELATIVE TO CONDITION TESTING MICROWORD).

2. THE TWO MICROWORDS BETWEEN THE CONDITION PRODUCING AND THE CONDITION TESTING MICROWORDS ARE ONLY ALLOWED TO PERFORM OPERATIONS WHICH WILL NOT AFFECT THE MICROBRANCH CONDITIONS (MICROPROGRAMMING RESTRICTION).

3. MICROTRAPS OR STALLS DURING THE BRANCH LATENCY PERIOD DO NOT AFFECT THE CORRECT FUNCTIONING OF THE MICROBRANCH. HOWEVER, DYNAMIC MICROBRANCH CONDITIONS (SEE TEXT) ARE NOT PRESERVED ACROSS IB DECODE CYCLES.

MKV86-1216

Figure 3-7 Microbranch Latency

There are two basic types of microbranch conditions (relative to microbranch latency):

- Dynamic
- Static

Dynamic conditions, such as the WBUS <N,Z,C,V> bits, are only valid for one cycle and can be tested only by the third microword after the one that produces them. Static conditions, such as the state flags, are valid indefinitely and may be tested by the third and any later microword. Dynamic microbranch conditions are indicated by the "@" character in Table 3-4.

3.3.5 Microsubroutine Calls And Returns

Refer to Figure 3-8. The IBox hardware supports microsubroutine calls and returns through the use of the microstack. Subroutines may be nested up to a depth of 15 calls, after which the stack wraps back, overwriting previous entries. Note that the microstack is exclusively for subroutine calls/returns, not for microtrap related operations.

The I_USTACK microword field and the UTRP MCA control the operation of the microstack. The I_USTACK field specifies the operation (call or return), the UTRP MCA maintains the microstack pointer and supplies the stack write enable signal.

3.3.5.1 Normal Microstack Operation - During normal operation (no subroutine call or return), the address of each new microword is loaded in the microstack location specified by the microstack pointer. This occurs at the same time the microaddress is loaded into the CS0 RAM address latches (canonical T3). As long as the I_USTACK field of the new microword does not request a subroutine call or return, the UTRP MCA will keep the microstack pointer constant so that the next microaddress will simply overwrite the one previously stored in the selected microstack location.

3.3.5.2 Microsubroutine Calls - On a subroutine call, the address of the calling microword is loaded in the stack and the stack pointer is incremented by one to point to the next location. This effectively pushes the address of the calling microword onto the stack. The I_NEXT field of the calling microword, which specifies the subroutine starting address, is then selected by the UBRS MCAs and sent to the CS0 RAM address latches.

The microprogrammers can also also specify conditional multiple-entry point subroutine calls by encoding a subroutine call and a conditional microbranch in the same microword. In this case, the condition bits are ORed with the I_NEXT field of the calling microword, generating one of several subroutine entry points.

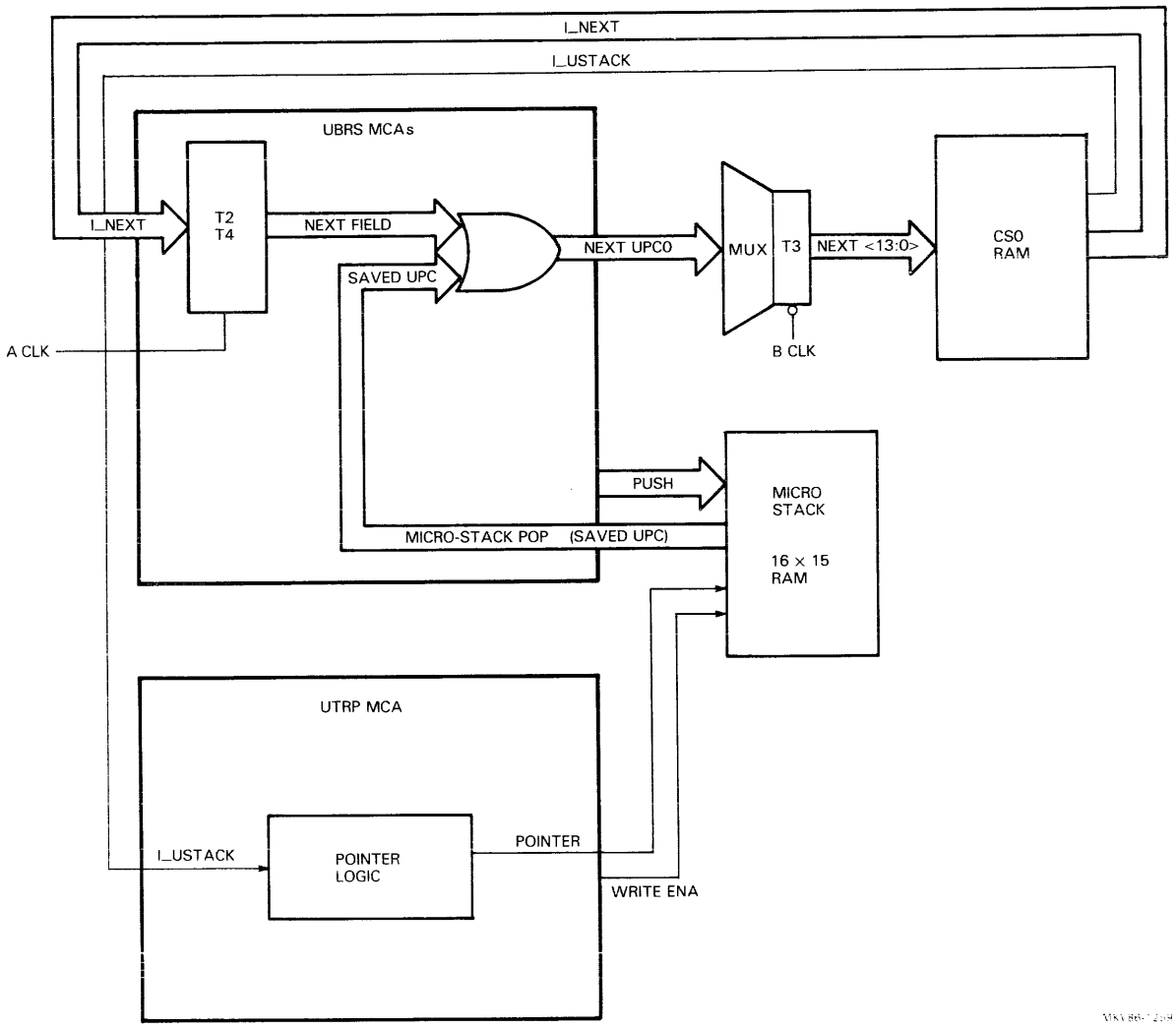


Figure 3-8 Microstack Operation

3.3.5.3 Microsubroutine Returns - On a subroutine return, the microstack pointer is first decremented by one to point to the stack entry containing the address of the original calling microword. This address is then popped from the stack, sent to the UBRS MCAs, and ORed with the 5 low order bits from the I_NEXT field of the returning microword. The resulting microaddress is then sent to the CS0 RAM address latches.

Note that by ORing the popped microaddress with the low 5 bits of the I_NEXT field of the returning microword allows a single subroutine to have up to 32 return points. To implement the return, the following constraint is applied to the address of the calling microword:

> If one (or more) of the five low order address bits of the calling microword is to be ORed with the corresponding bit from the I_NEXT field of the return microword, the address bit(s) must be zero.

For example, if I_NEXT bit <0> is set in the return microword, address bit <0> of the calling microword must be 0. (Single point returns can use any combination of the low five address bits in this manner.)

Microprogrammers can also specify conditional, multiple-return points for a subroutine by encoding a conditional microbranch in the return microword. In this case, the UBRS MCAs form the return point address by ORing the popped microaddress with the OR of the I_NEXT <4:0> bits from the returning microword and the enabled microbranch conditions.

3.3.6 Microtraps

Refer to Figures 3-9 and 3-10 for the following discussions.

Microtraps are hardware detected conditions which prevent the current microword from executing properly. When a microtrap occurs, hardware alters the current microcode flow by generating a microtrap vector to a fixed control store address, overriding the address that would have otherwise been selected. The microtrap vector, which is based on the type of trap detected, forces microcode to enter the appropriate trap handler microroutine to service the condition.

Microtraps can be caused by serious system faults, such as a machine check due to a control store parity error, or by conditions expected during normal macroinstruction execution. For example, microtraps are used extensively by the CBox memory management logic to call routines that service such faults as TB misses and access control violations.

Microtraps are prioritized such that if two or more microtraps occur in the same cycle, the higher priority microtrap is serviced and the others ignored. Nested traps (traps within traps) are not supported. Instead, trap service routines and trap priorities are arranged such that a second trap is taken only after the first trap is serviced. Machine check traps (such as a control store parity error), however, cannot be controlled in this manner since they are unpredictable.

It is the responsibility of the priority encoder logic in the UTRP MCA to monitor all microtrap conditions and to generate the microvector of the highest priority condition present. The following table lists the various microtrap conditions and the corresponding microvectors.

Table 3-6 Microtrap Conditions And Vectors

Vector	Microtrap Condition	Priority
03E0	Microbreakpoint Hit	Highest
03C0	Machine Check	
03A0	VA Parity Error	
0380	TB Tag Parity Error	
0360	Reserved for ECO trap	
0340	Illegal FP Operand	
0320	FP Rounding Error (addition)	
0300	FP Rounding Error (multiply)	
02E0	Integer Overflow	
02C0	TB Miss	
02A0	TB ACV	
0280	Modify Bit Not Set	
0260	Page Cross	
0240	Unaligned Page Cross	
0220	Unaligned Address	
0200	Conditional macrobranch	Lowest

Machine Check Microtraps

The UTRP MCA generates a common vector of 03C0 for certain types of machine check exceptions. Depending on the type of fault, a machine check may be reported by one of four mechanisms:

1. Microtrap vector from the UTRP MCA
2. Special address from the IB decoder logic
3. Interrupt vector from the INPR MCA
4. Current microword I_NEXT field

The following table lists the machine check exceptions reported by the microtrap mechanism.

Table 3-7 Machine Check Microtrap Conditions

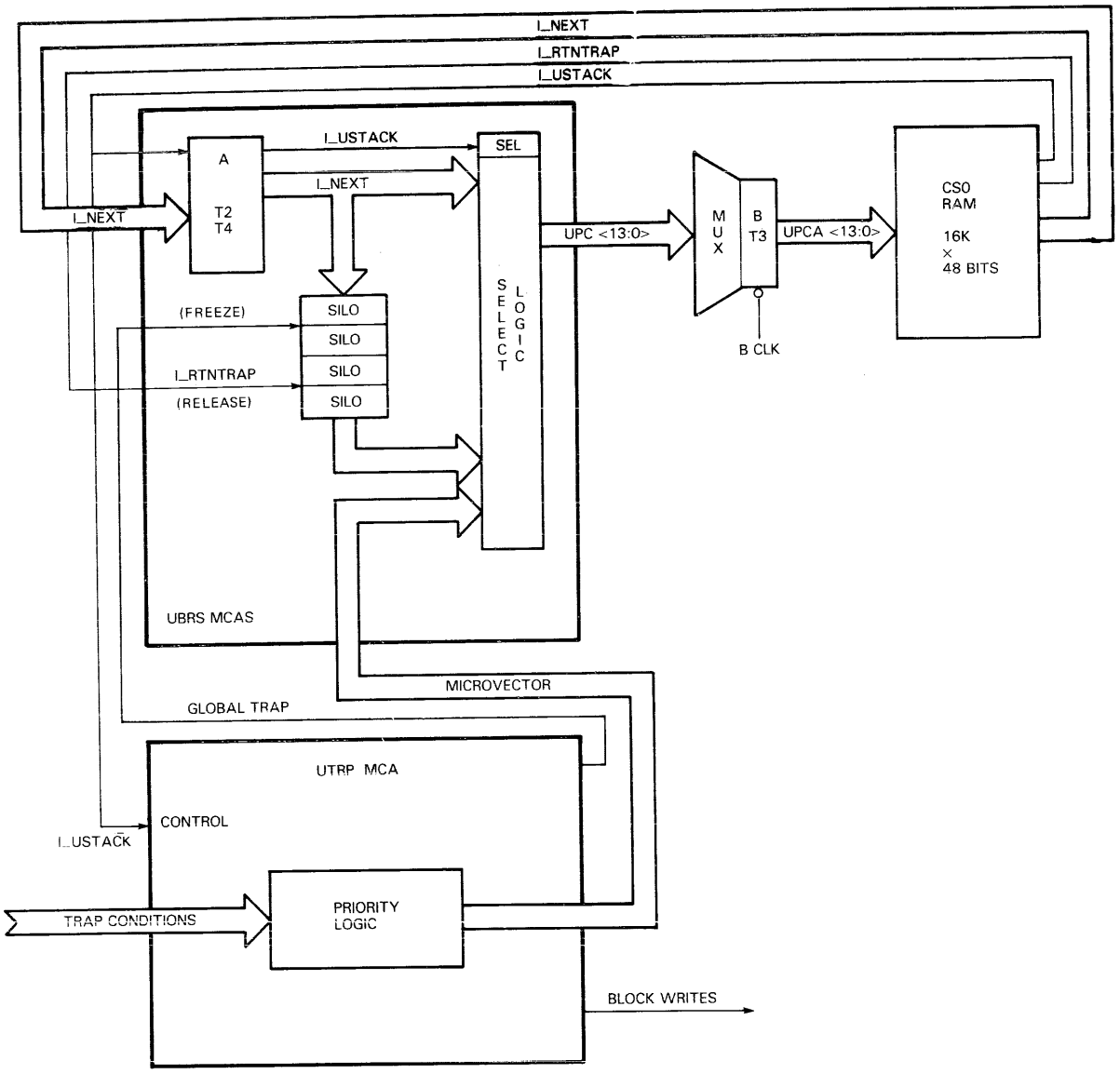
Cache Data Bus Parity Errors
Cache Tag Parity Errors
Control Store RAM Parity Errors
Console Receive Data Parity Error
Decoder RAM Parity Error
IB Data Bus Parity Error
Main ALU Parity Errors
Processor Register Parity Errors
TB Data Parity Error

Note that VA parity errors and TB tag parity errors are also machine check exceptions but have their own microtrap vectors (03A0 and 0380).

3.3.6.1 Microtrap Servicing - Microtrap conditions are generated at various times of a microword but are not honored until canonical T10 by the UTRP MCA. For example, a machine check trap due to a decoder RAM parity error is sensed at T4 while a TB miss microtrap is sensed at T9 time. Microtrap conditions sensed early in a microword are propagated through a series of latches so that all conditions arrive at the priority encoder in the UTRP MCA at canonical T10 of the trap causing microword.

When a microtrap occurs, the hardware must:

1. Preserve the current pipeline state
2. Preserve the current CPU state
3. Service the microtrap
4. Return from the trap
5. Restore CPU state to proper post trap condition



NOTES: WHEN A MICROTRAP OCCURS, THE UTRP MCA OUTPUTS THE:

- MICROVECTOR OF THE APPROPRIATE TRAP HANDLER ROUTINE
- GLOBAL TRAP SIGNAL TO FREEZE THE MICRO-PC SILO (SAVES THE UPCS OF THE FOUR MICROWORDS IN THE PIPELINE THAT FOLLOWED THE TRAP CAUSING MICROWORD)
- BLOCK WRITES SIGNALS TO INHIBITS WRITES BY THE FOUR MICROWORDS IN THE SHADOW OF THE TRAP.

THE LAST MICROWORD OF THE TRAP HANDLER ROUTINE RETURNS FROM THE TRAP BY ISSUING EITHER:

- I_RTNTRAP-RETURN TO ORIGINAL FLOW, REQUES TRAPPED MICROWORDS.
- I_USTACK/RLS.SILO-ABORT ORIGINAL FLOW, RELEASES MICRO-PC SILO, DOES NOT REQUE TRAPPED MICROWORDS.

Figure 3-9 Microtrap Servicing

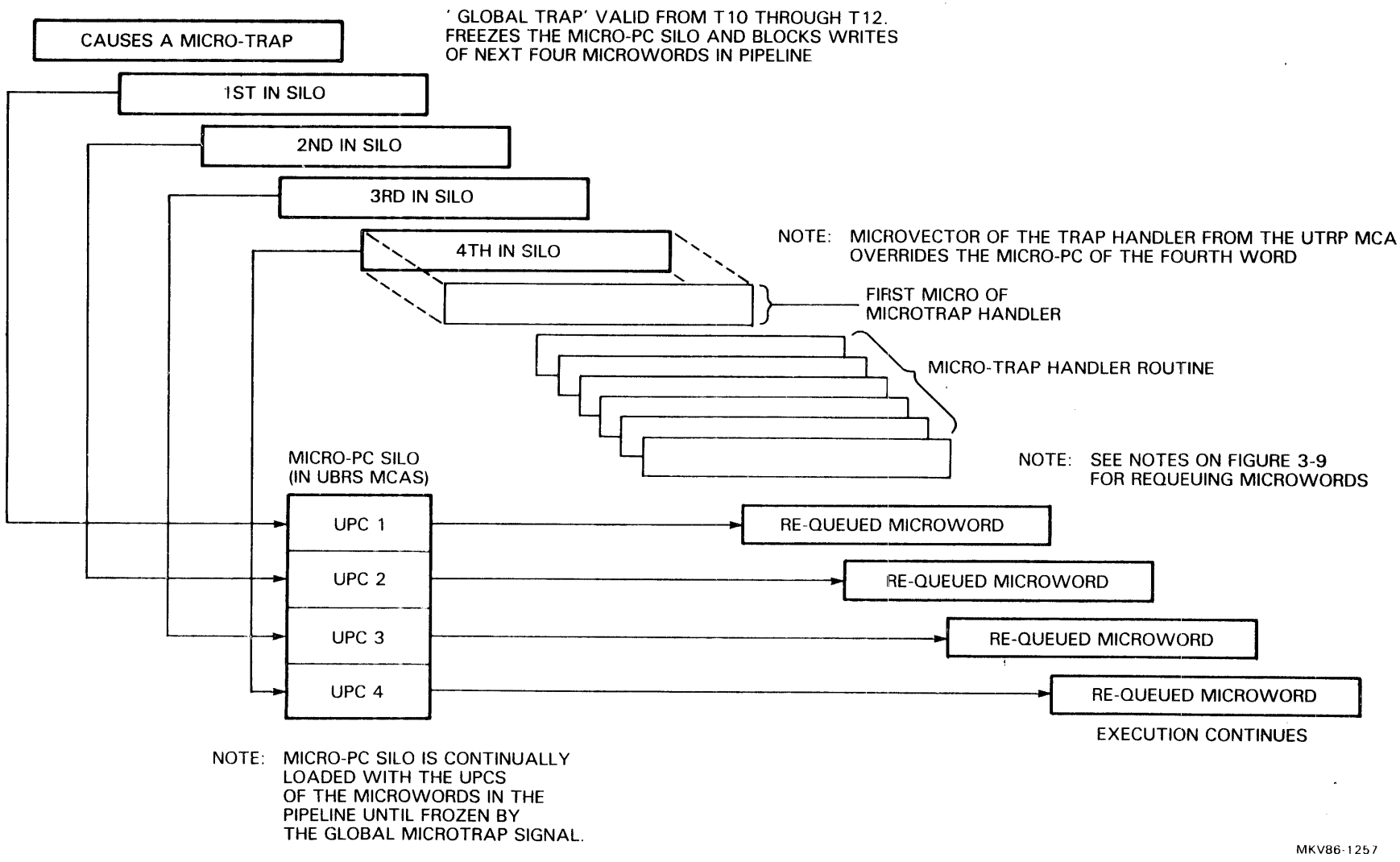


Figure 3-10 Microtrap Latency

Preserving The Current Pipeline State

Due to microcode pipelining, three additional microwords are already started and the address of a fourth microword generated by the time the UTRP MCA responds to a trap. To allow the original microcode flow to resume on a trap return, the addresses of these four microwords are saved in the micro-PC silo maintained by the UBRS MCAs.

During normal microcode flow, the micro-PC silo is continually loaded with the address of each microword executed. When a microtrap occurs, the GLOBAL UTRAP signals from the UTRP MCA inhibit further loading of the silo. Since the silo is four words deep, the addresses of the four microwords following the trap causing microword are thus saved in the silo. On a normal trap return, the UBRS MCAs requeue these four addresses, allowing the original microcode flow to continue.

Preserving The Current CPU State

The three microwords already in the pipeline after the one that caused the trap must be prevented from altering the CPU state so that the trap service routine can properly service the fault. (Note that the fourth microword in pipeline has not yet started, only the address is formed and saved in the micro-PC silo.)

The current CPU state is preserved by either saving potentially corruptible data in various silos in the CPU or by inhibiting the writes that would otherwise be performed by the other three microwords already in the pipeline.

The UTRP MCA issues the GLOBAL UTRAP CONT <2:0> signals to inform the rest of the CPU that a trap occurred and to save critical CPU state conditions in the appropriate silos.

The UTRP MCA also issues the BLOCK WRITE CONT <3:0> signals to inhibit all writes starting at T11 relative to the trap causing microword.

Servicing The Microtrap

When a microtrap occurs, the UTRP MCA outputs the appropriate vector for the trap service routine along with the GLOBAL UTRAP signal to the UBRS MCAs. This forces the UBRS MCAs to select the microtrap vector as the source of the next microaddress.

Since the microword that caused the trap has already completed (unsuccessfully) by the time the UTRP MCA honors the trap, the trap service routine is responsible for correcting the fault (if possible) such that the net result is the same as if the microword completed successfully.

Returning From A Microtrap

Some microtrap service routines return to the original microcode flow while others abort the original flow and enter a new flow. Routines of the first type include those that service memory management related faults, such as a TB miss. Routines of the second type include those that service more serious faults, such as a machine check.

The settings of the I_RTNTRAP bit and the I_USTACK field in the last microword of the trap service routine determine how the routine is to return from the trap. The I_RTNTRAP bit informs the UBRS MCAs and the UTRP MCA whether a normal return is in effect, the I_USTACK field is used to release the micro-PC silo.

To return to the original microcode flow, the I_RTNTRAP bit and the I_USTACK field are encoded as follows:

I_RTNTRAP/ENABLE, I_USTACK/RLS.SILO

On a normal trap return, the four microaddresses that were saved in the micro-PC silo are sequentially output by the UBRS MCAs and sent to the CS0 RAM addresses. This requeues the four microwords that followed the one that caused the trap.

The only exception to this is that IB decoder generated addresses are not saved in the micro-PC silo. When an address is generated by the decoder, the corresponding location in the micro-PC silo is flagged with a DECODER MARKER (derived from the I_DECODER microword bit). Then, if on the trap return a silo location is so marked, the selection of silo as the microaddress source is terminated and the next address is selected from the decoder.

Trap service routines abort the original microcode flow by issuing the I_USTACK/RLS.SILO order by itself. This releases the micro-PC silo, re-enabling it for future traps.

The silo release micro-order may be issued in any microword of a trap routine with the following constraint:

> The trap return (I_RTNTRAP) must be issued in the very next microword after one that performs a potential trap causing operation from which the microcode may want to return.

Trap routines normally encode the silo release and the trap return orders in the same microword. Trap routines that do not return but also perform potential trap causing operations must release the first trap in the next microword after the one containing the trap causing operation.

3.3.6.2 Disabling Microtraps - The console can disable microtraps by setting a Disable Microtrap bit in the console interface logic on the CLK module. If this bit is set, all microtraps, including machine checks traps, are inhibited. The bit can only be written by the console and is normally cleared.

3.3.7 Console Supplied Microaddresses

Refer to Figure 3-11. The console has the ability to write a microaddress in the micromatch register of the UBRS MCAs. The contents of this register are compared with current UPC <13:00> and, if the two are equal, a MICROMATCH signal is sent back to the console.

The console generated address is selected in response to an explicit request from the console and takes precedence over all other sources.

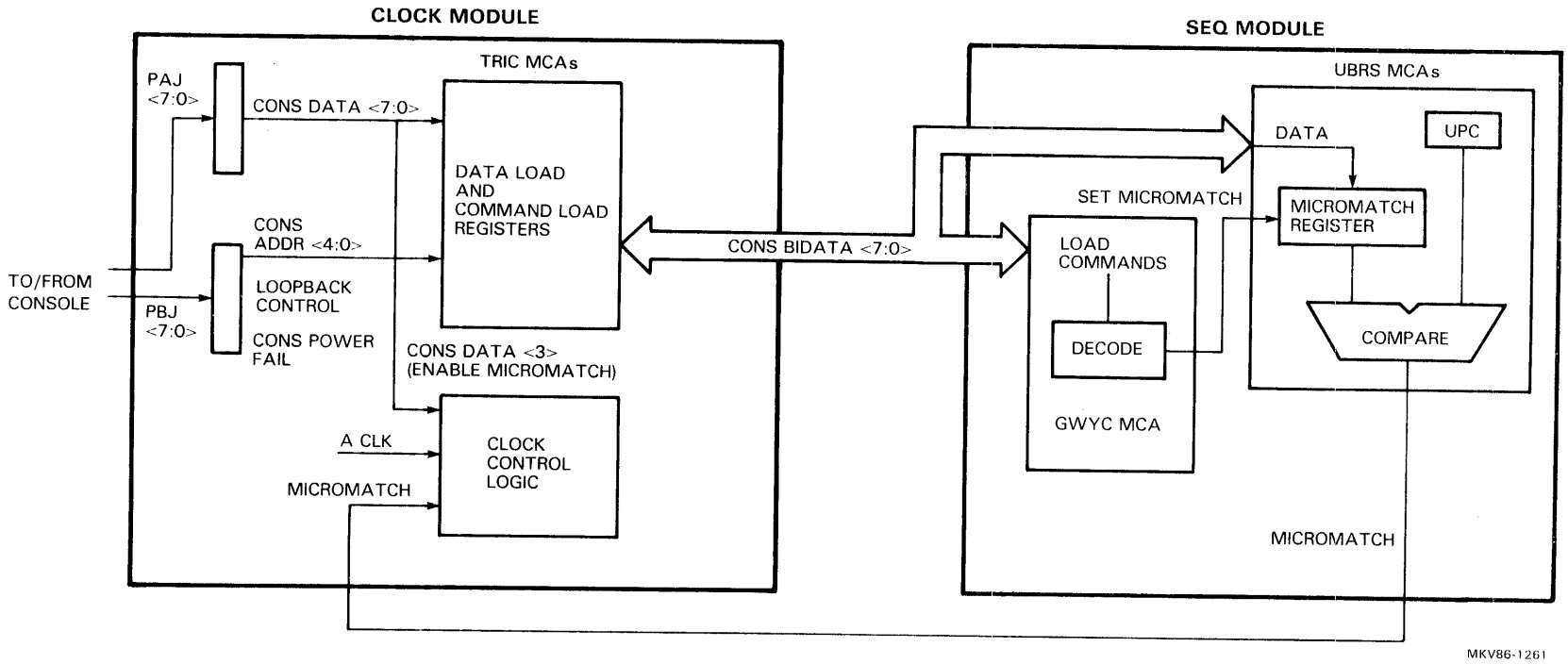


Figure 3-11 Console-Supplied Microaddress

3.4 MACROINSTRUCTION DECODING

The macroinstruction decode process entails selecting a block of I-stream data from the instruction buffer (IB), decoding the opcode and current operand specifier, and generating the entry point address for the microroutine required to service the specifier. After each specifier of the instruction is processed, the IB is logically shifted so that the next specifier (if any) is made available for decoding. Once all specifiers are processed, the entry point address for the execute code of the instruction is then generated.

Refer to Figure 3-12. The description of the instruction decode process begins with the operation of the IB.

3.4.1 Initializing the IB (IB Flush)

When the VAX PC is modified by other than its incremented value (for example, summed with a branch offset), the IB and the decoder must be initialized. This "IB Flush" operation is invoked by microcode to ensure that the I-stream data for the next instruction to be executed is processed properly.

There are two types of IB Flush operations:

1. Full flush - Used by microroutines that handle PC control instructions, interrupts, exceptions, and other macro services

2. Partial flush - Used by microtrap service routines

3.4.1.1 Full IB Flush - An example of when a full IB flush is required is the execution of a successful macrobranch instruction.

When a successful macrobranch is executed, the CBox will deliver the new I-stream data to the IB starting with the longword containing the opcode of the new instruction. Since there is no longer a correlation between data entering the IB and its current address pointers, it is highly likely that the next decode cycle will access the opcode for the wrong instruction. To prevent this, all microroutines that service macrobranch instructions issue a full IB flush before exiting. This initializes the IB's address inputs and forces the Decoder to wait for the CBox to deliver the new I-stream.

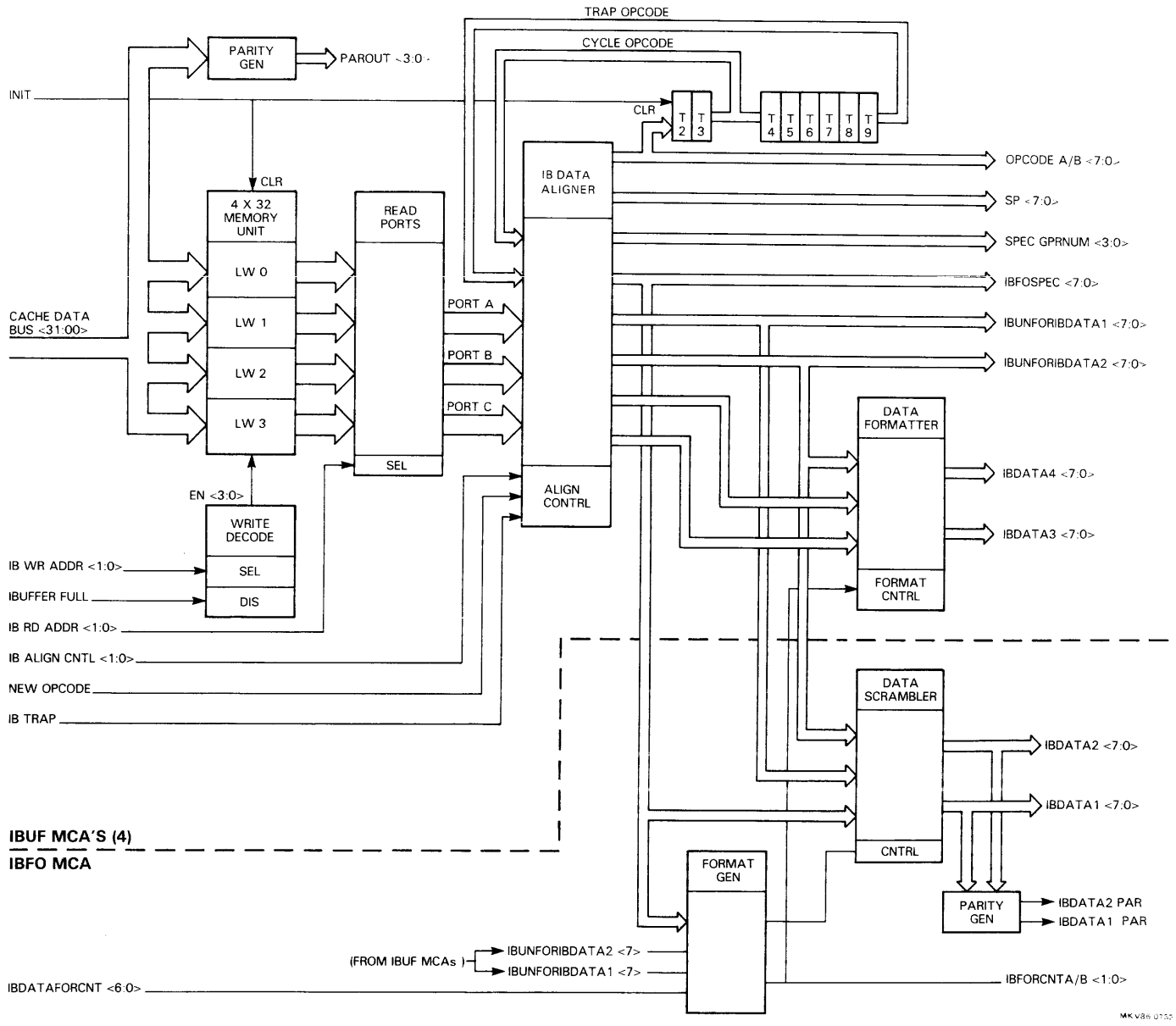


Figure 3-12 Instruction Buffer Logic

3.4.1.2 IB Flush Logic - Refer to Figure 3-13. The VAX Branch logic of the IBST MCA determines if a full IB flush is required by checking the I_MISC microword field. When this field is in the range of 20 to 2F, one of the following microroutines is being executed:

I_MISC	Microservice Routine for:
20 to 2E	Simple conditional branches (eg: BEQL, BNEQ) Loop control instructions (eg: ACBx, AOBLEQ)
2F	Unconditional branches (BRB, BRW) Subroutine instructions (BSBB, JSB, RSB) Interrupts, exceptions, CPU init,... etc.

The I_MISC settings in the range 20 to 2F and the instruction(s) they represent are listed below.

I_MISC	Instruction	I_MISC	Instruction
20	BGRT	28	BGTRU
21	BLEQ, SOBGTR	29	BLEQU
22	BGEQ	2A	BCC
23	BLSS, SOBGEQ	2B	BCS
24	BNEQ	2C	AOBLEQ, ACBx
25	BEQL	2D	AOBLSS
26	BVC	2E	BBx, BBxx
27	BVS	2F	BRx, BSxx

When a conditional branch is executed, I_MISC specifies which PSL CC bit(s) the VBRL should test to see if the branch should be taken (the PSL CC bits are also contained in the CCBR MCA).

Example:

If: I_MISC = 24
Then: Test PSL Z bit for BNEQ instruction

Z Bit	Meaning	Operation
Set	Branch fail	VBRL negates IB FLUSH. IBox executes next instruction from IB.
Clear	Branch success	VBRL asserts IB FLUSH. EBox sums branch offset to VAX PC. IBox waits for the new I-stream to enter IB.

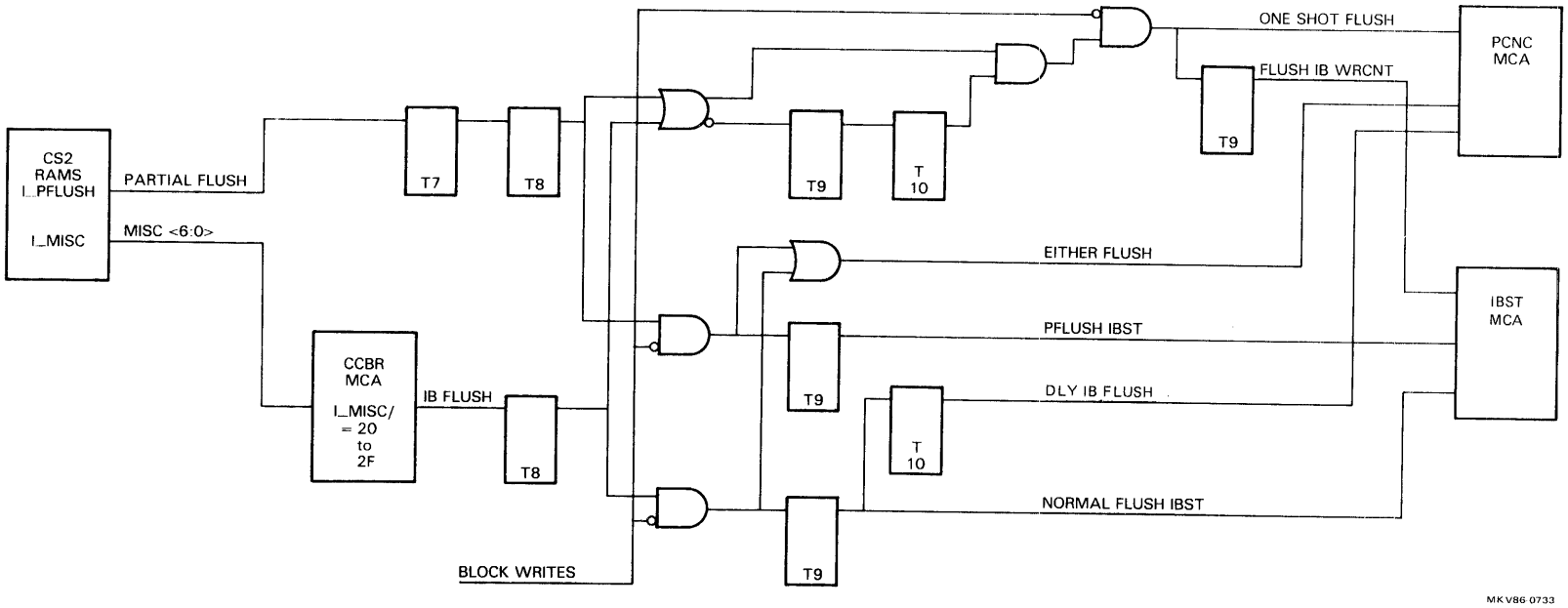


Figure 3-13 IB Flush Logic

If an unconditional branch is executed (I_MISC equal to 2F), the VBRL does not test any CC bits but immediately asserts IB FLUSH to force an unconditional flush.

Refer to Table 3-8. When IB FLUSH is asserted, the discrete DEC module logic issues several IB flush related signals to the IBST MCA and to the PCNC MCA (the signals are listed in chronological order):

1. EITHER FLUSH
2. NORMAL FLUSH IBST
3. ONE SHOT FLUSH
4. DLY IB FLUSH
5. FLUSH IB WRCNT

These signals initialize the IB and the Decoder by forcing most outputs of the IBST and PCNC MCAs into a known state.

The column labeled FULL FLUSH in Table 3-8 lists the IBST and PCNC MCA outputs after a full IB flush. The mnemonic "Idep" means that the output is instruction dependent and remains unchanged until a new instruction enters the IB.

Table 3-8 IBST And PCNC MCA Outputs After An IB Flush

IBST MCA

Output Signal	State: Full Flush	State: Partial Flush
IB WRADDR <1:0>	Clear	Clear
IB PE <1:0>	Clear	Clear
INDEXED OR INDEX PC	Clear	Clear
LW ACCEPTED	Clear	Clear
SEQ LW	Clear	Clear
SLOW BRANCH	Clear	Restored
SPEC NO <2:0>	Clear	Restored
SPEC WAS RMODE	Clear	Clear
TWO BYTE	Clear	Restored
NEW OPCODE	Set	Restored
SILO NEW OPCODE	Set	Restored
SPECIAL ADDR ENBL	Set	Restored
DLY OPN XOR IBS	Idep	Idep
PRED DATA SIZE <1:0>	Idep	Restored
SPEC TYPE <3:0>	Idep	Idep

PCNC MCA

IB RD ADDR <1:0>	Clear	Clear
IB FULL	Clear	Clear
PC INC <2:0>	Clear	Clear
IB ALIGN CNTL <1:0>	Load w/ VA FILE <1:0>	
IB EMPTY	Set	Set
IB DEC NOOP	Set	Set
IB DEC NOOP OR PE	Set	Set
OP IS FD	Idep	Idep
SET BRANCH	Idep	Idep

The IBST MCA and PCNC MCA output state changes relevant to an IB flush are described below. The other changes will be covered later.

Table 3-9 IB Flush Relative State Changes - IBST And PCNC MCAs

State change	Function
Clearing IB WR ADDR <1:0> IB RD ADDR <1:0>	Forces the new I-stream to be written to and read from the IB starting with its longword 0 location.
Loading IB ALIGN CNTL <1:0> with VA FILE <1:0>	Ensures that the first byte read from the IB will be the opcode byte of the new instruction. VA FILE <1:0> always equal VAX PC <1:0> after an IB flush and therefore point to the opcode byte in the first longword delivered by Cache.
Asserting NEW OPCODE	Forces the IB to select the opcode byte from its memory unit instead of from its "Cycle Opcode Register".
Clearing SPEC NO <2:0>	Forces the Decoder to generate the entry address for the microroutine that services the first specifier of the new instruction.

3.4.1.3 Partial IB Flush - A partial flush is invoked when microword bit I_PFLUSH is set. This bit is sent as the signal PARTIAL FLUSH to the same logic used for a full IB flush. The only difference is that the logic will issue the signal PFLUSH IBST instead of NORMAL FLUSH IBST and DLY IB FLUSH.

PFLUSH IBST forces the IBST MCA to "restore" several IB state indicators (operand data size, specifier number, etc.) to the state they were in prior to the time that a microtrap occurs (see column labeled PARTIAL FLUSH in Table 3-8. This allows microtrap service routines to recover from traps that occur during instruction execution.

The IB state indicators are stored in a silo internal to the IBST MCA when a microtrap is detected.

3.4.2 I-Stream Prefetching

I-stream prefetching enhances instruction execution speed by keeping the IB supplied with I-stream data. The operation is a function of the CBox and is described in detail in the CBox section of this manual.

3.4.2.1 General Description - The CBox maintains a Physical Instruction Buffer Address register (PIBA) which contains the address of the next longword to be delivered to the IB. The PIBA is initially loaded with the new PC formed when there is a change in the I-stream flow (eg: after a successful branch instruction) and is updated when a longword is accepted by the IB.

During each of its idle cycles, the CBox fetches the longword pointed to by the PIBA offers it to the IB on the CACHE DATA BUS. If the IB accepts the data, the PIBA is incremented by four to point to the next longword which is offered to the IB in the next CBox idle cycle.

The CBox assumes each longword is accepted by the IB unless the IBox asserts IB FULL. In this case, PIBA incrementing is suspended and the last referenced longword is offered to the IB. Prefetching resumes once a longword is processed from the IB and the IBox negates IB FULL.

3.4.2.2 Refilling Cache - Each time a PIBA request is made, there exists the possibility that Cache will not contain the required data. In this event, Cache will report a "read miss", forcing the CBox to obtain the data from NMI memory by performing an operation known as a PIBA "refill."

PIBA refilling entails fetching a hexaword (32 bytes) from the NMI (a longword at a time) and storing the data in Cache. There are two key points about the PIBA refill operation relevant to I-stream decoding:

1. The NMI hexaword is transferred as two octawords

2. The first longword returned by NMI is always the one requested when the Cache read miss occurred

When it receives the first longword, the CBox simultaneously stores it in Cache and offers it to the IB.

If the PIBA was octaword aligned when the read miss occurred, the next three longwords of the first octaword are stored in Cache and offered to the IB. This allows the IBox to continue the decode process while the rest of the data is being stored in Cache.

3.4.3 Loading The IB

Refer to Figure 3-14.

The IBUF MCAs always assume the CACHE DATA BUS carries I-stream data destined for the IB. This means that they will attempt to store data from the bus on every B clock. (The IB load operation is asynchronous to the pipeline in that no microword field controls the IB.)

The IBUFFER FULL signal from the PCNC MCA determines if data should enter the IB. When this signal is negated, it indicates that the IB longword addressed by IB WRADDR <1:0> is "empty" and can be loaded with data from the CACHE DATA BUS. (An IB longword is considered empty if ALL the bytes in the longword are processed.)

The Write Decode logic of the IBUF MCAs checks IBUFFER FULL on every A clock. If the signal is negated, the logic will decode IB WRADDR <1:0> and clock the longword from the CACHE DATA BUS into the selected IB location on the following B clock.

3.4.3.1 IB Write Control - The IB load method has one drawback: it allows the IB to capture data from the CACHE DATA BUS even if the data are intended for the EBox. (The CACHE DATA BUS also supplies data for EBox operations.)

If IB WRADDR <1:0> were to change state each time Cache sent data to the EBox, the IB would be loaded with the wrong data. Since this would corrupt the instruction decode process, the IBST MCA must ensure that the write address only changes state when data on the CACHE DATA BUS is destined for the IB. To do this, the IBST MCA monitors five signals:

Signal	Source	Description
DEST IS IB	CBox	CACHE DATA BUS carries I-stream data destined for the IB
CACHE DATA VALID	CBox	Last Cache reference resulted in a cache hit. Signal also forced true when source of data is NMI.
MEMORY BROKEN	CBox	CBox detected some type of parity error (TB, Cache, NMI, etc.)
FLUSH IB WRCNT	IBox	IB Flush operation in progress
IBUFFER FULL	IBox	IB cannot accept another longword

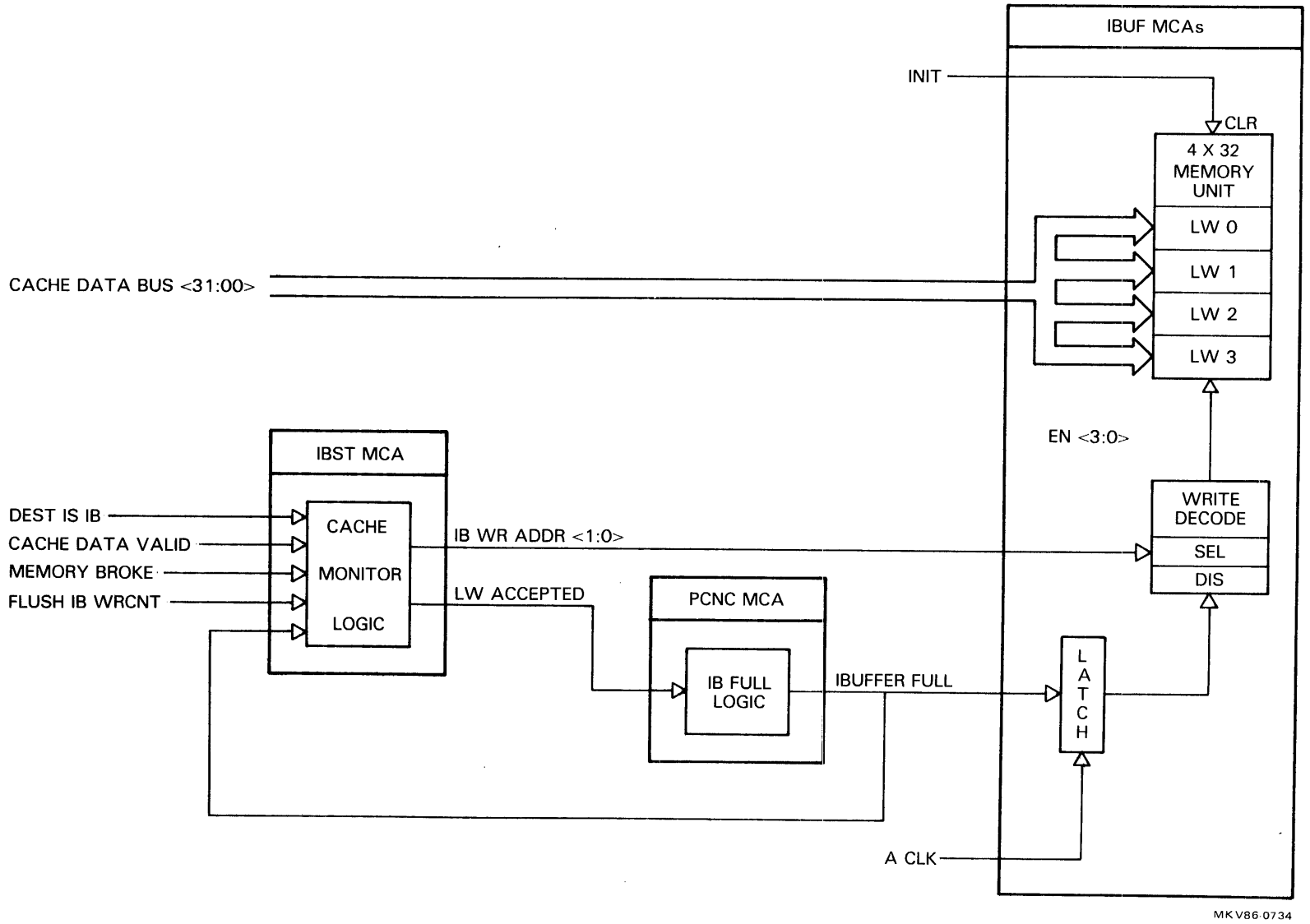


Figure 3-14 IB Load Logic

3.4.3.2 Cache Monitor Logic - When these signals are properly conditioned, the Cache Monitor logic in the IBST MCA asserts the signal LW ACCEPTED to indicate that a longword has entered the IB. It then increments IB WRADDR <1:0> by one to point to the next IB location to receive data from Cache.

The following chart indicates the state of LW ACCEPTED and IB WRADDR <1:0> for any given condition.

			LW ACCEPTED	IB WRADDR <1:0>
FLUSH	CACHE DATA VALID	1		
IB WRCNT	DEST IS IB	1	1	Previous
	MEMORY BROKE	0		value + 1
0	IBUFFER FULL	0		
	Other combinations		0	Previous value
1	Don't care		0	Cleared

Note that IB WRADDR <1:0> are unconditionally cleared when FLUSH IB WRCNT is asserted. This ensures that the first longword from the CBox is loaded in the IB's longword 0 location.

From this point on, the IBST MCA will increment IB WRADDR <1:0> by one each time a longword on the CACHE DATA BUS is destined for and loaded into the IB. Once all four IB longwords are loaded, the write address is "wrapped back" to again point to the IB's longword 0 location.

Since IB WRADDR <1:0> can only change state when the CACHE DATA BUS is destined for the IB, they will always point to an "empty" IB location (assuming IBUFFER FULL is negated). Thus, the problem of the IB trying to capture data on every B clock is resolved; the new longword simply replaces one that was already processed by the IBox.

3.4.3.3 IB Full Logic - When a longword enters the IB, the assertion of LW ACCEPTED causes the PCNC MCA to increment a counter which tracks the number of longwords available for decoding. This "IB Full" counter is incremented by one when a longword enters the IB and is decremented by one when a longword is processed from the IB.

If the CBox supplies data to the IB faster than the IBox can process the data, causing the counter to overflow, the PCNC MCA will issue IBUFFER FULL to prevent any more data from entering the IB.

3.4.3.4 IB Load Example - Although I-stream data always enters the IB longword aligned, the VAX architecture imposes no such restrictions on the PC of an instruction. Since this means that the I-stream for a new instruction may begin on any byte boundary, the first longword from the CBox may include data associated with some other instruction.

Figure 3-15 shows how I-stream data for a MOVL #©X12345678, R0 instruction would enter the IB. The xx's represent I-stream data associated with other instructions.

Assumptions:

1. IB Flush just completed
2. Starting PC for MOVL is 1003

Note that the first longword (address 1000) is stored in the IB's LW 0 location. This is always the case after an IB flush. Also note that although the instruction only occupies seven bytes in memory, it takes three IB LW locations to contain the data. The consequences of this aspect of the IB will become apparent in the next discussion.

```
Assembler Syntax  -  MOVL #^X12345678, R0
Machine code      -  50 12345678 8F D0

Instruction as    -  Address    Contents
stored in memory

                     1000       D0 xx xx xx
                     1004       34 56 78 8F
                     1008       xx xx 50 12
```

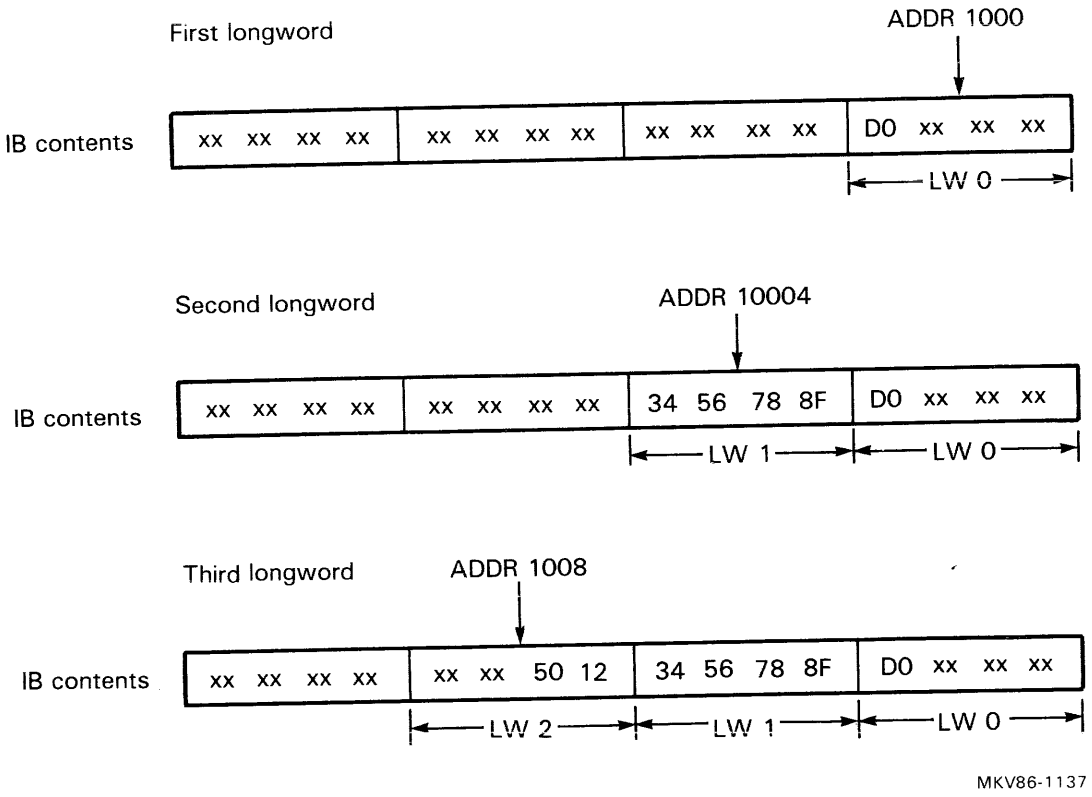


Figure 3-15 I-Stream Data Entering The IB

3.4.4 Reading The IB

Although I-stream data enters the IB a longword at a time, the IB is treated as a sixteen byte buffer when read. The IB read operation entails selecting six consecutive bytes from the IB and outputting the data to the rest of the IBox. These bytes consist of the:

- Opcode
- Current specifier (if any)
- Up to 4 extension bytes (eg: immediate data)

The opcode byte is sent to the Decoder RAMs (DRAMS) to form part of their address inputs. It is also saved in a "Cycle Opcode" register during the first decode cycle of the instruction. The cycle opcode register then becomes the source of the opcode for subsequent decode cycles.

The specifier byte is sent to the IBST MCA which will determine the operand specifier's:

- Size (byte, word, longword)
- Type (literal, register mode, etc.)
- Access mode (read, write, etc.)
- Position in the instruction (1 to 6)

Both bytes are sent to the PCNC MCA which determines the general class of the instruction and computes the amount of IB data "consumed" in each decode cycle.

The four extension bytes, which are sign extended for byte and word size specifiers, are sent to the EBox on the IB DATA BUS. Since 6 consecutive bytes are always read from the IB, this data may or may not be related to the current specifier. Microcode determines if the EBox is to use all four bytes from the bus.

3.4.4.1 Pipeline Timing - IB reads are performed during canonical T1. This allows the Decoder to generate the entry address for the microroutine that will service the specifier (or opcode) during T2 time and have it ready for the micro-PC address latches by T3.

3.4.4.2 IB Read Ports - The Read Ports select nine consecutive bytes from the IB Memory unit starting with the longword pointed to by IB RD ADDR <1:0>. The bytes are output on three read ports to the IB Data Aligner. Ports A and B are both four bytes wide, port C is one byte wide.

The following table shows the relationship between IB RD ADDR <1:0> and data selected by the IB Read Ports.

Table 3-10 IB Read Address/IB Read Port Source

IB RD ADDR <1:0>	PORT A Source	PORT B Source	PORT C Source
0 0	LW 0	LW 1	LW 2 Byte 0
0 1	LW 1	LW 2	LW 3 Byte 0
1 0	LW 2	LW 3	LW 0 Byte 0
1 1	LW 3	LW 0	LW 1 Byte 0

Note that the read ports perform a "wrap around" when the read address is equal to 2 or 3. This allows the read ports to always access nine consecutive bytes regardless of the starting longword address.

The reason the read ports select nine bytes at a time is in case the first specifier of an instruction is being decoded and the opcode byte occupies byte 3 of a LW.

Since the opcode byte can end up in any position in a longword and is "consumed" in the decode cycle of the first specifier, up to six bytes of IB data may be required to process the first operand:

- Opcode
- First specifier
- Up to four extension bytes

Example:

- Instruction just executed ended in byte 2 of LW 2
- Next instruction to be executed is:

  MOVL #©X12345678, B©04 (R0)

The I-stream for the MOVL would be loaded into the IB as shown below. (The xx's indicate I-stream data associated with other instructions.)

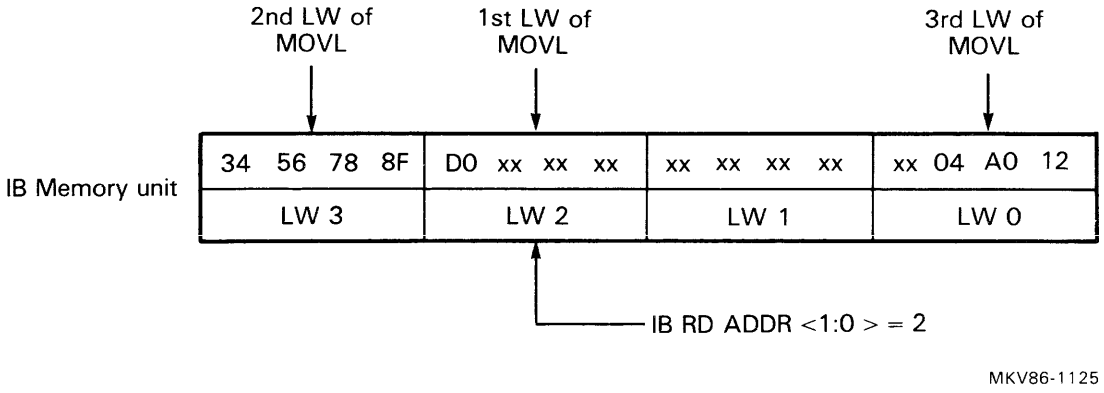
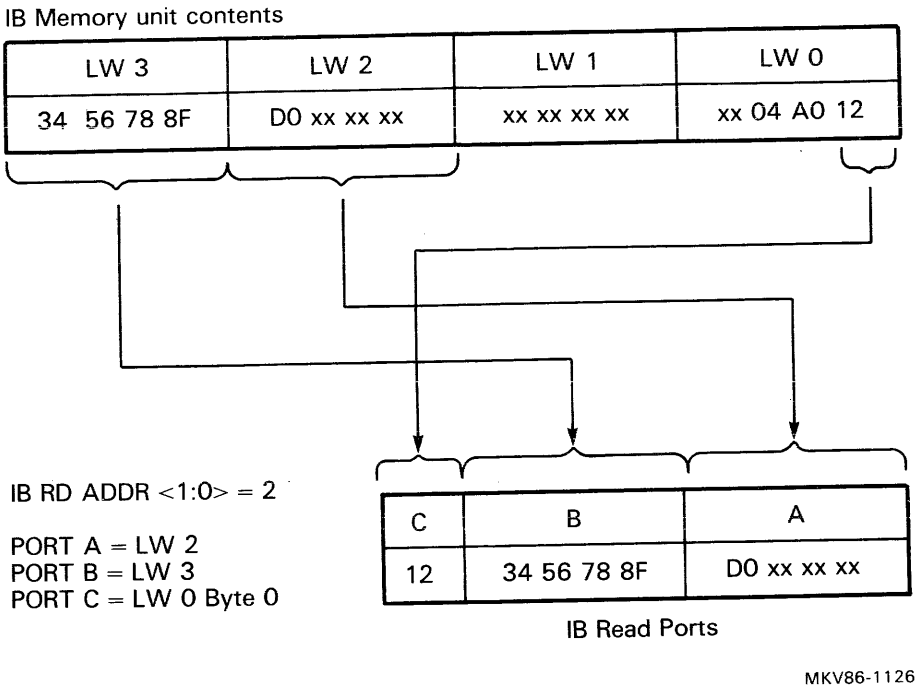


Figure 3-16 IB Memory Unit Contents For MOVL Example

First Decode Cycle - MOVL #©X12345678, B©04(R0)

Figure 3-17 shows the data selected by the read ports with a read address of 2 in the first decode cycle of the MOVL.

Notice that by performing a wrap around function, the read ports output the I-stream for the MOVL in the same order it was delivered by the CBox. Also, by selecting nine bytes at a time, they can read all six bytes required by the decode cycle even though the opcode occupied byte 3 of a LW.



Note:

If the previous instruction caused a change in the I-stream flow (eg: branch or jump), the I-stream for the MOVL would still be output from the IB. However, the data would be ignored since the IB would not be "logically" shifted at canonical T3 time of the last microword of the previous instruction.

Figure 3-17 IB Read Port Example - Part 1

MOVL #©X12345678, B©04(R0)

First Decode Cycle

Second Decode Cycle - MOVL #©X12345678, B©04(R0)

While the first specifier is being processed, IB RD ADDR <1:0> are updated to point to the IB longword containing the second operand specifier.

The next figure shows the data selected by the Read Ports in the second decode cycle.

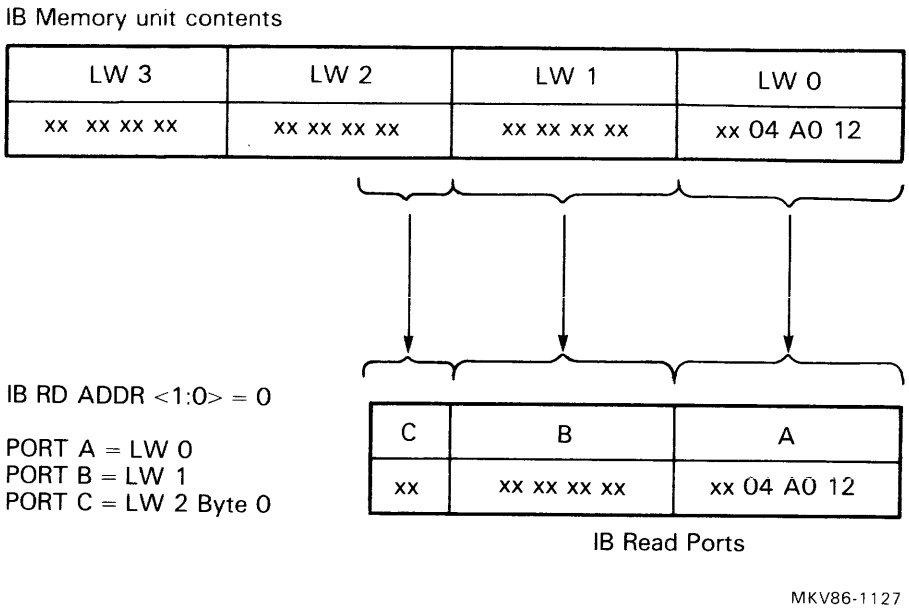


Figure 3-18 IB Read Port Example - Part 2

Note:

LWs 2 and 3 are shown containing data associated with some other instruction since new I-stream data will most likely be delivered to the IB while the first specifier is being decoded.

Figure 3-18 IB Read Port Example - Part 2

MOVL #©X12345678, B©04(R0)

Second Decode Cycle

Figures 3-17 and 3-18 serve to illustrate two important aspects concerning the IB Read Ports:

1. The opcode byte appears in PORT A in the first decode cycle of each new instruction

2. By selecting nine bytes at a time, the Read Ports can access all I-stream data associated with a specifier in one decode cycle

Exceptions:

1. Since the second byte of a two byte opcode is the true opcode, the first byte is given its own decode cycle. The second byte will appear in PORT A in the next decode cycle.

   Except for this, instructions with two byte opcodes are processed the same as any other instructions. (Note: Microcode treats the first byte of a two byte opcode as if it were a NOP macroinstruction.)

2. Indexed and big immediate (quad/octaword, double, grand and huge) specifiers require additional decode cycles.

3.4.4.3 IB Data Aligner - The six data aligner muxes are each one byte wide. They select six consecutive bytes from an eleven byte input field consisting of:

- Nine bytes from the IB Read Ports
- The Cycle Opcode Register
- The Trap Opcode Silo

The following figure shows the aligner muxes, the data they output and the destination of the data. The indicator "internal" means that the data from a mux remains internal to the IBUF MCAs.

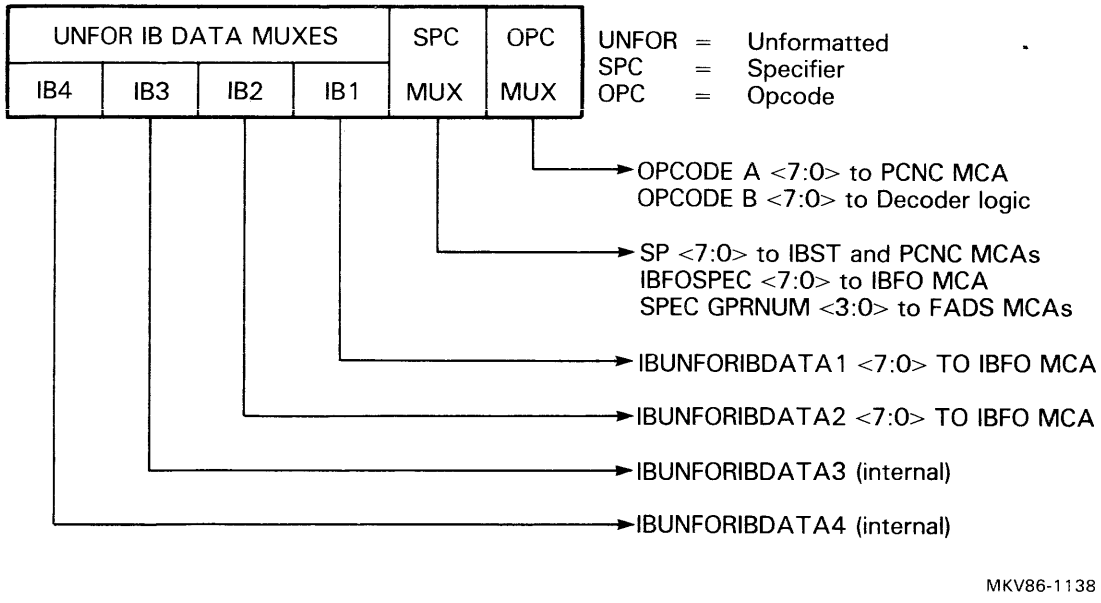


Figure 3-19 IB Data Aligner Muxes

Data Aligner Control

The Data Aligner Muxes are controlled by the following signals:

- o IB ALIGN CNTL <1:0> - PCNC MCA
- o NEW OPCODE - IBST MCA
- o IB TRAP - discrete DEC module logic

The next table shows the relationship between the control signals and the data selected by the muxes.

Table 3-11 Data Aligner Control Signals/Data Selection

IB TRAP = 0							
IB ALIGN CNTL <1:0>	IB Data Aligner Muxes						
	IB4	IB3	IB2	IB1	SPC	OPC	
0 0	B-1	B-0	A-3	A-2	A-1	A-0	CYC
0 1	B-2	B-1	B-0	A-3	A-2	A-1	OPC
1 0	B-3	B-2	B-1	B-0	A-3	A-2	REG
1 1	C	B-3	B-2	B-1	B-0	A-3	
NEW OPCODE = 1 (First decode cycle)							
NEW OPCODE = 0 (Subsequent cycles)							
IB TRAP = 1, NEW OPCODE = x (don't care)							
IB ALIGN CNTL <1:0>	IB Data Aligner Muxes						
	IB4	IB3	IB2	IB1	SPC	OPC	
x x	Don't care					TRAP OPCODE SILO	

Note: A-0, A-1, etc. represent the IB Read Port and the byte in the port selected by a mux. For example:
If: IB TRAP =0, NEW OPCODE =1, IB ALIGN CNTL <1:0> =00
Then: OPC MUX selects PORT A Byte 0 (A-0)

Data Aligner Operation

The overall Data Aligner operation will be illustrated using the MOVL #©X12345678, B©04(R0) instruction as an example.

The MOVL instruction was assumed to be loaded in the IB as follows:

IB Memory Unit

34 56 78 8F	D0 xx xx xx	xx xx xx xx	xx 04 A0 12
LW 3	LW 2	LW 1	LW 0

In the first decode cycle, IB RD ADDR <1:0> pointed to the longword containing the opcode byte (LW 2). They were then updated in the second cycle to point to the longword containing the second specifier (LW 0).

Figures 3-20 and 3-21 show the bytes selected by the IB Data Aligner muxes for each decode cycle of the MOVL.

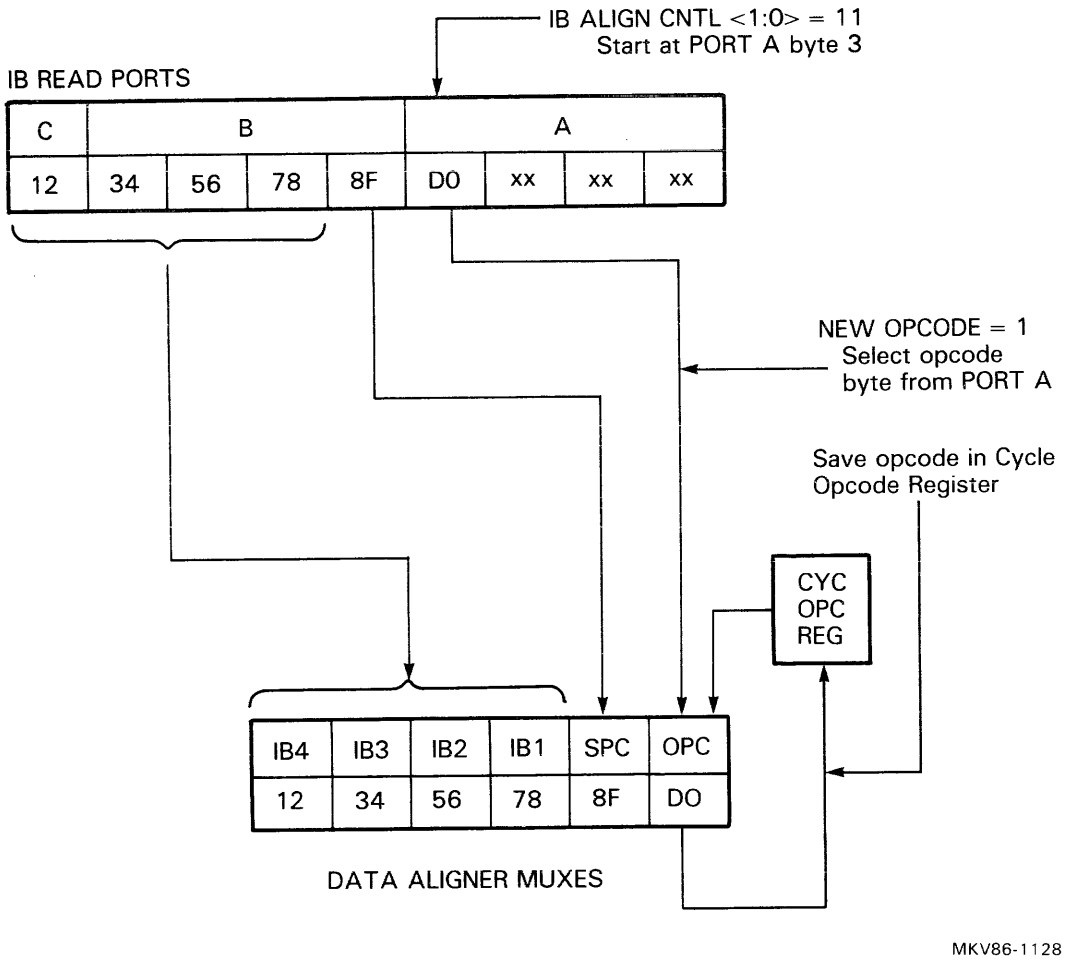


Figure 3-20 IB Data Aligner Output Example - Part 1

MOVL #©X12345678, B©04(R0)

First Decode Cycle

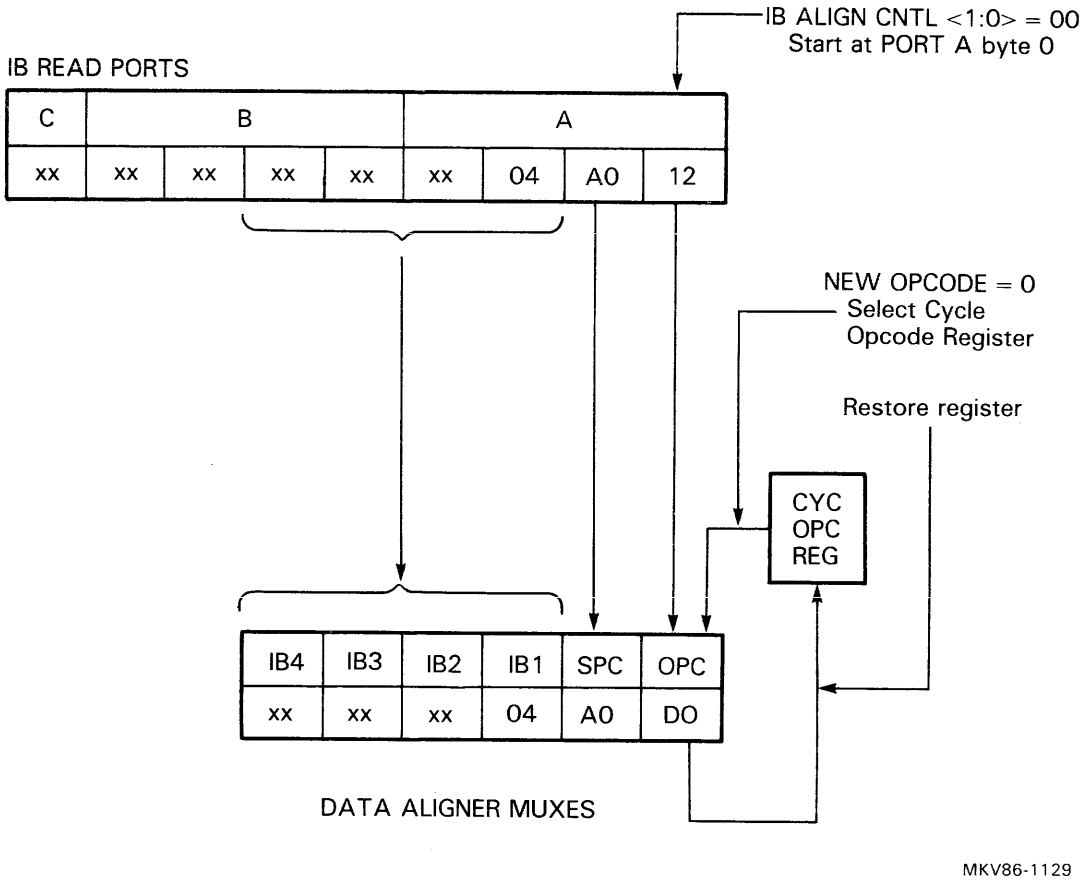


Figure 3-21 IB Data Aligner Output Example - Part 2

MOVL #©X12345678, B©04(R0)

Second Decode Cycle

OPC Mux Source Selection

Refer to Figure 3-22.

The selection of the source input to the OPC mux determines which bytes the SPC and the IB1 to IB4 muxes select from the Read Ports.

There are six possible source inputs to the OPC mux:

- Four PORT A bytes
- Cycle Opcode Register
- Trap Opcode Silo

Figure 3-20 shows that NEW OPCODE is asserted in the first decode cycle of the MOVL. This forces the OPC mux to select the PORT A byte pointed to by IB ALIGN CNTL <1:0>.

IB ALIGN CNTL <1:0> always equal VAX PC <1:0> in the first cycle of an instruction and therefore point to the opcode byte. The opcode is thus selected from the appropriate PORT A byte and stored in the Cycle Opcode register.

Figure 3-21 shows that in the second cycle, IB ALIGN CNTL <1:0> point to the byte that precedes the second specifier instead of the specifier itself. This is because the OPC mux must be able to select the opcode from PORT A in the first cycle and then from the Cycle Opcode register in each subsequent cycle. To reduce the amount of logic needed to do this, IB ALIGN CNTL <1:0> are conditioned to point to a "dummy" byte in PORT A in every decode cycle except the first.

With the starting byte position established by IB ALIGN CNTL <1:0>, the next five consecutive bytes from the Read Ports are selected by the SPC mux and the IB1 to IB4 muxes.

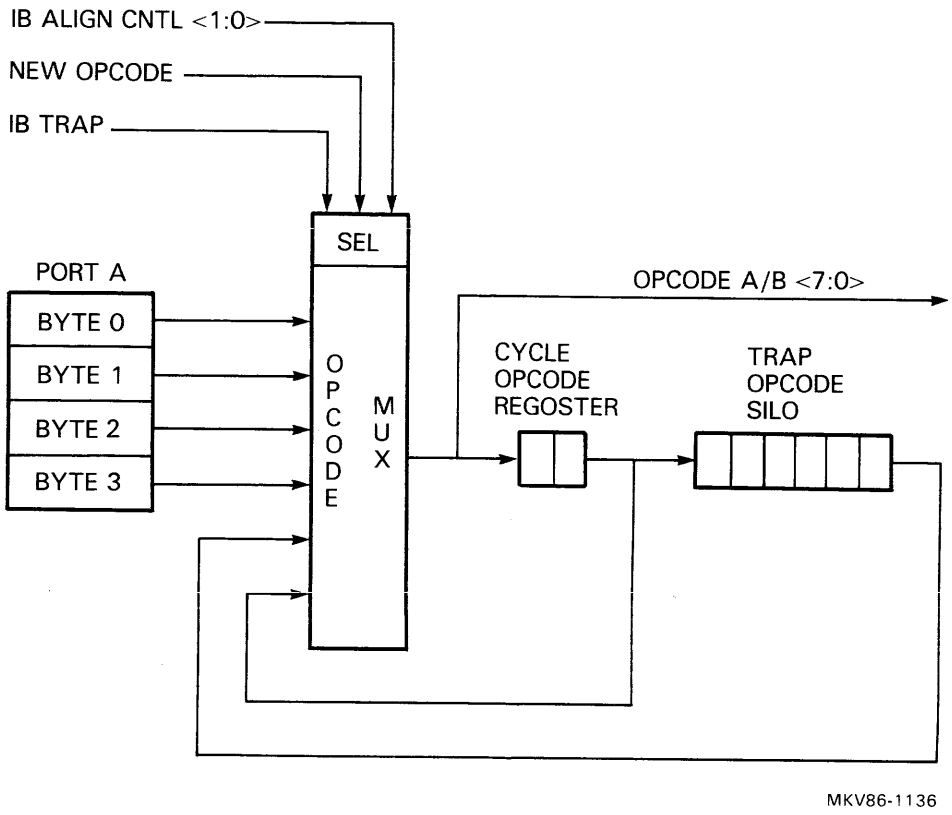


Figure 3-22 Opcode Mux Sources

SPC Mux Source Selection

The SPC mux always selects the specifier byte in the decode cycle associated with a specifier. The only exception to this is when the specifier is indexed or is a big immediate.

Indexed and big immediate specifies both require additional decode cycles to fully process the operand. In this case, the SPC mux will select data as follows:

Specifier Type	SPC MUX Action
Index mode	Select index in first decode cycle then base in second cycle (if base is a big immediate, the following also applies).
Big immediate	Select specifier byte in first cycle then low order byte of each subsequent longword in the following cycles.

IB1 - IB4 Mux Source Selection

Data selected by the IB1 to IB4 muxes is considered unformatted since it may or may not be related to the current specifier.

In the first cycle of the MOVL, all four muxes contain data related to the first specifier. Since data formatting is not required, the four extension bytes are sent unmodified to the EBox on the IB DATA BUS.

In the second cycle, only the IB1 mux contains data related to the specifier. Since the byte displacement (04) is to be summed with GPR R0 to form a longword address, it must be sign extended before being sent to the IB DATA BUS. This sign extending is a function performed by the Data Scrambler and Data Formatter logic. Once the displacement is properly sign extended, it is then sent to the EBox on the IB DATA BUS.

Note that the EBox does not always require all four bytes from the IB DATA BUS to be related to the current specifier. For example, to process the first specifier of a MOVB #©X12, R0 instruction, the EBox only uses the low byte of the bus and ignores the upper three.

Cycle Opcode Register Selection

In the second cycle of the MOVL, NEW OPCODE is negated to force the OPC mux to select the Cycle Opcode Register. During this cycle, the opcode is selected from the Cycle Opcode Register, passed through the OPC mux and stored back in the register.

Since a MOVL instruction only has two operands, this opcode "recycling" is performed only once, in the decode cycle for the second specifier. However, if the instruction were, say, a MOVP, the opcode would be recycled several times, in the second and each subsequent specifier decode cycle of the instruction.

For instructions with two byte opcodes, the NEW OPCODE signal stays asserted in the second cycle since the first byte requires its own decode cycle. NEW OPCODE is then negated in the third and subsequent cycles.

Most instructions use the Cycle Opcode Register again after the last specifier is processed. This is to allow the Decoder to examine the opcode and generate the entry point address for the execute code. However, certain instructions, such as MOVL, do not need this cycle since they combine the execute code in the routine of the last specifier. These "Optimized" instructions are covered later.

Trap Opcode Silo Selection

The Trap Opcode Silo is selected as input to the OPC mux when IB TRAP is asserted.

The Trap Opcode Silo provides a copy of the opcode as it was 4 cycles ago which corresponds to the delay between the occurrence of a trap and the start of the microroutine that services it. A four cycle old copy of the opcode is required since by the time a trap is detected, the IB has usually been "shifted" beyond the problem causing specifier.

(Every two latches in Figure 3-22 equal one cycle. Thus, it takes four cycles for the opcode to get from the OPC mux to the output of the silo.)

Trap service routines use the Trap Opcode Silo along with certain data saved in the IBST MCA to restore IB state indicators to the state they were in at the time the trap occurred. The IB itself is flushed and refilled with the I-stream data pointed to by a copy of the VAX PC which is saved in a similar manner in a Trap PC Silo maintained by the EBox.

3.4.4.4 IB Data Formatter and Data Scrambler - One of the functions of the IB is to supply the EBox with I-stream embedded data such as short literals, immediates, branch offsets and displacements, and absolute addresses. This function is performed during canonical T2 through T5 by the data formatter and data scrambler.

Data Formatter/Scrambler Logic

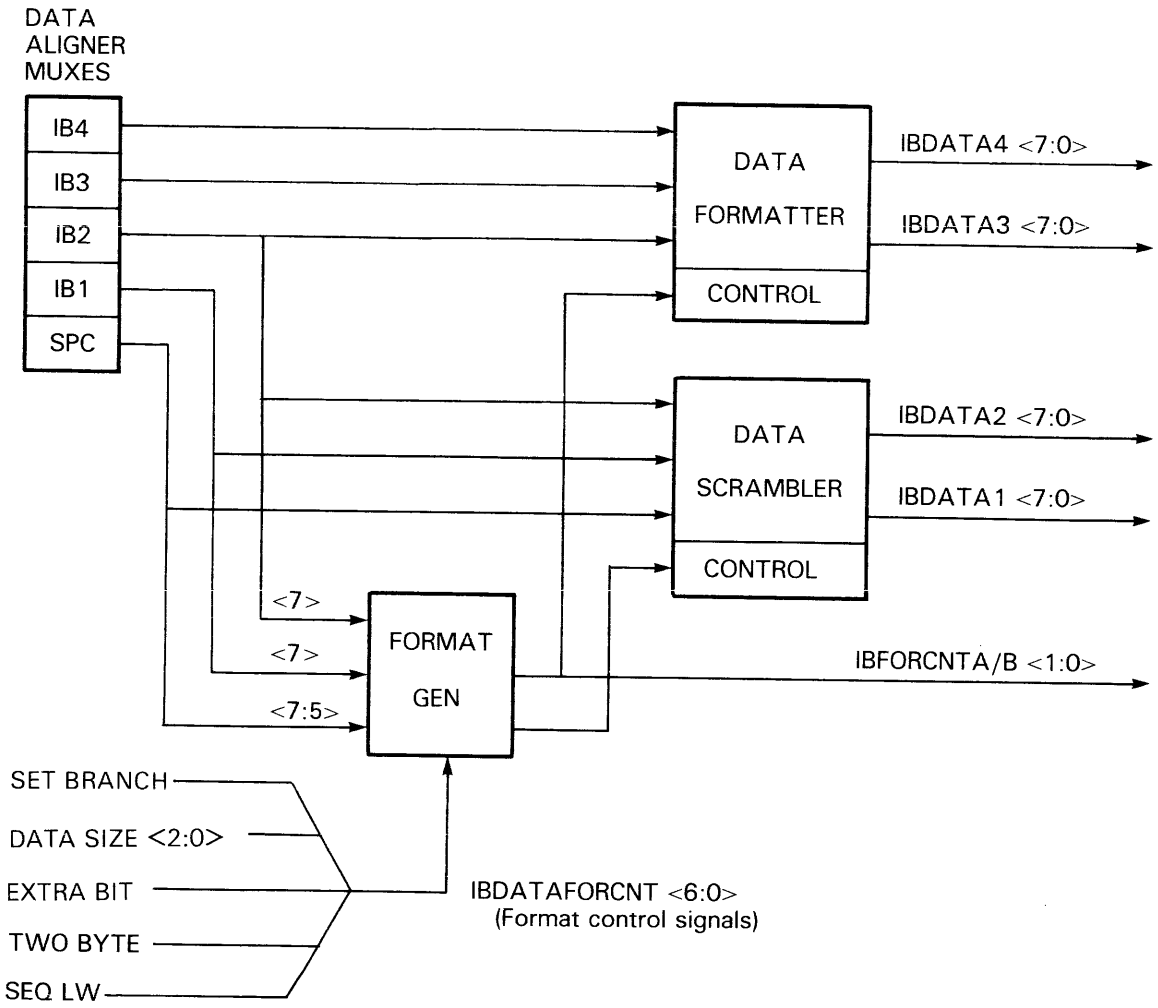
Refer to Figure 3-23.

The Data Formatter and Data Scrambler consist mostly of muxes. They receive the unformatted IB data from the IB1 to IB4 muxes and the specifier byte from the SPC mux. Then, based on the addressing mode of the specifier and various control signals, they format the data and send it to the EBox on the IB DATA BUS.

IB Format Control Inputs

The format control inputs to the Format Generator, IBDATAFORCNT <6:0>, represent the general instruction class and the data size of the current specifier.

Table 3-12 lists the state of the format control signals for any given data type. The order in which a signal appears in the table indicates its logical precedence. That is, SET BRANCH has the highest precedence followed by SEQ LW. When these signals are both negated, the remaining inputs indicate the data type of the specifier.



Notes:

1. The Format Generator and the Data Scrambler reside in the IBFO MCA. The Data Formatter resides in the IBUF MCAs.

2. Only bit <7> from IB1 and IB2 and bits <7:5> from SPC are sent to the Format Generator.

Figure 3-23 IB Data Formatter And Data Scrambler Logic

Table 3-12 IB Data Format Control Signals/Functions

Signal	Function
SET BRANCH	Indicates current specifier is the branch offset of a conditional, unconditional, or loop control branch instruction. (See notes.)
DATA SIZE <2:0>	Indicates data size of current specifier: 000, 011 = Not used 001 = Word 010 = Longword, F-Float 100 = Quadword, D-Float, G-Float 101 = Byte 110 = Octaword, H-Float (except 1st specifier) 111 = Octaword, H-Float (1st specifier only)
EXTRA BIT	Distinguishes between byte and word sized branch offsets (see SET BRANCH) and, with TWO BYTE, between the various data types when the DATA SIZE <2:0> bits are equal to 010, 100, 110 or 111.
TWO BYTE	Indicates instruction has two byte opcode (see entry for EXTRA BIT).
SEQ LW	Indicates the current decode cycle is processing a subsequent longword of a big immediate specifier (quad/octaword, double, grand or huge data).

Notes:

1. SET BRANCH is also asserted in the first decode cycle of a two byte opcode and in first, and only, cycle of an instruction that has no operands (eg: NOP). This prevents the Format Generator from interpreting the second byte of a two byte opcode or the opcode of a new instruction as a specifier.

2. SET BRANCH is not asserted for branch displacement specifiers, such as B©D(Rn). Specifiers of this type are decoded from their addressing mode bits, SPC <7:5>.

Table 3-13 IB Format Control Signals/Specifier Data Type

Set Branch	SEQ LW	Data Size	Extra Bit	Two Byte	Data Type
		001	-	-	Word
		010	0	-	Longword
			1	-	F-Float
		100	0	-	D-Float
0	0		1	0	Quadword
			1	1	G-Float
		101	-	-	Byte
		110	0	-	Octaword
			1	-	H-Float
		111	0	-	Octaword
			1	-	H-Float
	1	-	-	-	Subsequent LWs
1	-	-	0	-	Branch byte
			1		Branch word

SET BRANCH asserted

When SET BRANCH is asserted, the EXTRA BIT signal represents the data size of a branch offset. Depending on the state of this input, the Format Generator determines the sign of the offset from either:

- SPC <7> - byte offset
- IB1 <7> - word offset

Once it establishes the size and sign of the offset, the Format Generator causes the Data Scrambler and Data Formatter to output the sign extended offset to the IB DATA BUS. (The Data Scrambler and Data Formatter outputs will be shown later.)

SET BRANCH negated, SEQ LW asserted

This combination indicates that the IB is to output a subsequent longword of a big immediate specifier. Since no further formatting of the data is required, the contents of the SPC mux (low byte) and the IB1 to IB3 muxes (upper three bytes) are sent, unmodified, to the IB DATA BUS.

SET BRANCH and SEQ LW both negated

When SET BRANCH and SEQ LW are both negated, the remaining format control inputs determine the specifier data type.

Data Type	How Determined
Byte, Word	DATA SIZE <2:0> alone specifies the data size since there is only one 8 bit and one 16 bit data type.
Longword, F-Float	DATA SIZE <2:0> specifies a 32-bit data type, EXTRA BIT tells which one.
Quadword, D, G-Float	DATA SIZE <2:0> specifies a 64-bit data type, EXTRA BIT and TWO BYTE combined tell which one
Octaword, H-Float	DATA SIZE <2:0> specifies a 128-bit data type, EXTRA BIT tells which one

Defining the Specifiers Mode and Sign

To completely define a specifier, the Format Generator must determine its addressing mode and, for branch offsets, its sign. It does this by examining SPC <7:5>, IB1 <7>, and IB2 <7> (SPC bit <4> is not used by the Format Generator).

SPC <7:5> usually equals the specifier addressing mode bits and, as such, define the basic specifier type.

Since raw data need only be formatted when the specifier extension is less than a longword, only three addressing modes are considered:

Addressing Mode	SPC <7:5>	Data Formatter/Scrambler Operation
Literal	00x	SPC mux contains literal. Data Scrambler selects SPC mux as input and outputs data on IB DATA BUS either unmodified (integer literal), or "scrambled" (FP literal). (FP literal format is shown later.)
Byte displ and deferred. Byte relative and deferred.	101	IB1 mux contains displacement byte. Format Generator derives sign from IB1 <7>, forces Data Scrambler and Data Formatter to output sign extended displacement to IB DATA BUS.
Word displ and deferred. Word relative and deferred	110	As above except word displacement in IB1 and IB2 muxes. IB2 <7> has sign.

All other addressing modes either have longword extensions or no extension bytes at all. Data for these modes is therefore output to the IB DATA BUS in the same format received from the Data Aligner.

Data Formatter and Data Scrambler Outputs

Table 3-14 shows which bytes the Data Formatter and Data Scrambler output to the IB DATA BUS. The mnemonics SPC and IB1 to IB4 refer to the data selected from the IB Data Aligner muxes. SIGN refers to sign extended branch offsets or displacements. The indicator "x" means that the data being output is not related to the current specifier (but will have good parity).

Table 3-15 shows the "scrambled" format of floating point short literals. Microcode "unscrambles" the data in the EBox.

3.4.4.5 IB Read Example - Figure 3-24 (4 pages) shows the entire IB read operation for the decode cycles of the following instruction:

```
ADDL3 #@X12345678, B@04 (R3), R0
```

Assumptions:

1. Instruction loaded into IB starting at byte 1 of longword 0
2. No microtraps or interrupts occur during instruction execution

The Data Formatter and the Data Scrambler are shown collectively in the example as the FORMATTER.

Table 3-14 IB Data Formatter And Data Scrambler Output

Specifier Type or Instruction class	Data Formatter		Data Scrambler	
	IBDATA4	IBDATA3	IBDATA2	IBDATA1
Literal modes				
All integer types	x	x	0	SPC
F and D Float	x	x	See notes	
G and H Float	x	x	See notes	
Immediate mode				
Byte	x	x	x	IB1
Word	x	x	IB2	IB1
Longword	IB4	IB3	IB2	IB1
Subsequent LWs	IB3	IB2	IB1	SPC
Displacement modes				
Byte	SIGN	SIGN	SIGN	IB1
Word	SIGN	SIGN	IB2	IB1
Longword	IB4	IB3	IB2	IB1
Branch instructions				
Byte offset	SIGN	SIGN	SIGN	SPC
Word offset	SIGN	SIGN	IB1	SPC

Notes:

1. IB only supplies first longword for quad/octaword and for D, G and H-Float literals; microcode supplies upper longwords.

2. H-Float literals are output in same format as G-float literals (Table 3-15). Microcode performs format conversion in the EBox.

3. First longword of big immediates and address for PC absolute address mode (9F) are selected from same source muxes (IB1 to IB4) as longword immediates.

4. Entries for displacement mode specifiers also apply to PC relative and relative deferred modes.

Table 3-15 Floating Point Short Literal Formats

	IBDATA2	IBDATA1
	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
	0 1 0 0 0 0 x x	x x x x 0 0 0 0
F and D Float	<----------> SPC <5:0>	
	0 1 0 0 0 0 0 0	0 x x x x x x 0
G and H Float		<----------> SPC <5:0>

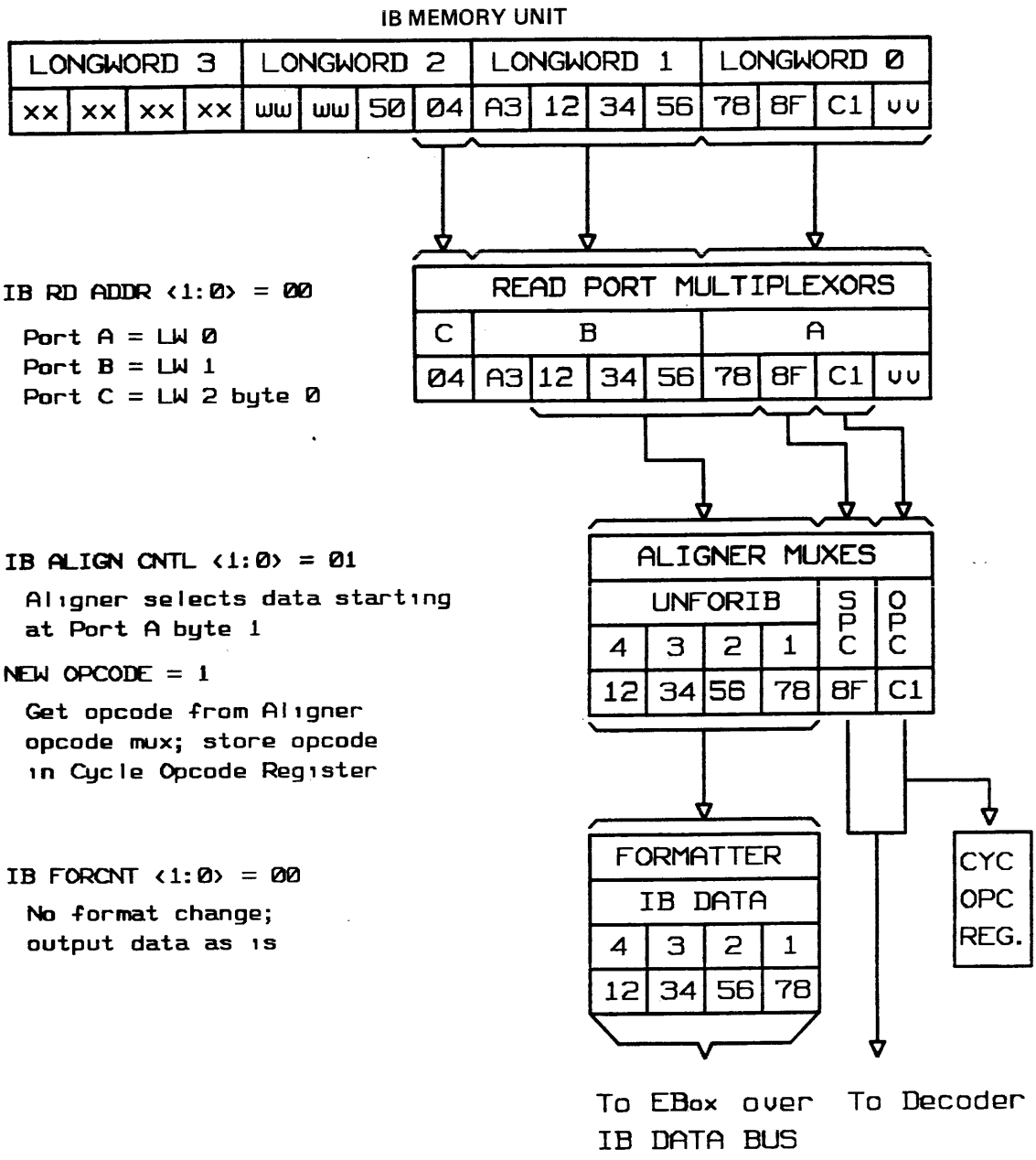


Figure 3-24 IB Read Example (Sheet 1 Of 4)

ADDL3 #©X12345678, B©04 (R3), R0

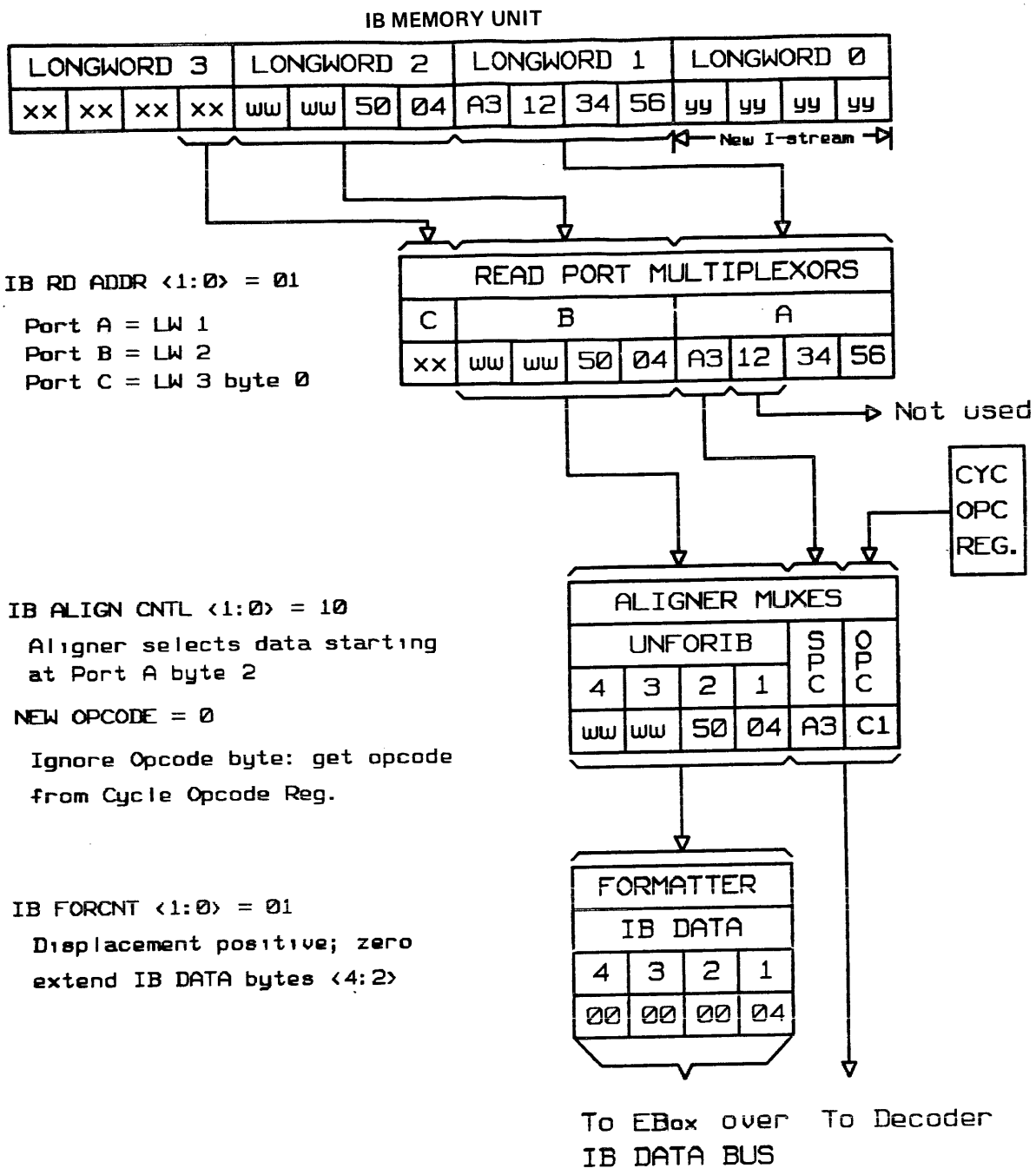


Figure 3-24 IB Read Example (Sheet 2 of 4)

ADDL3 #©X12345678, B©04 (R3), R0

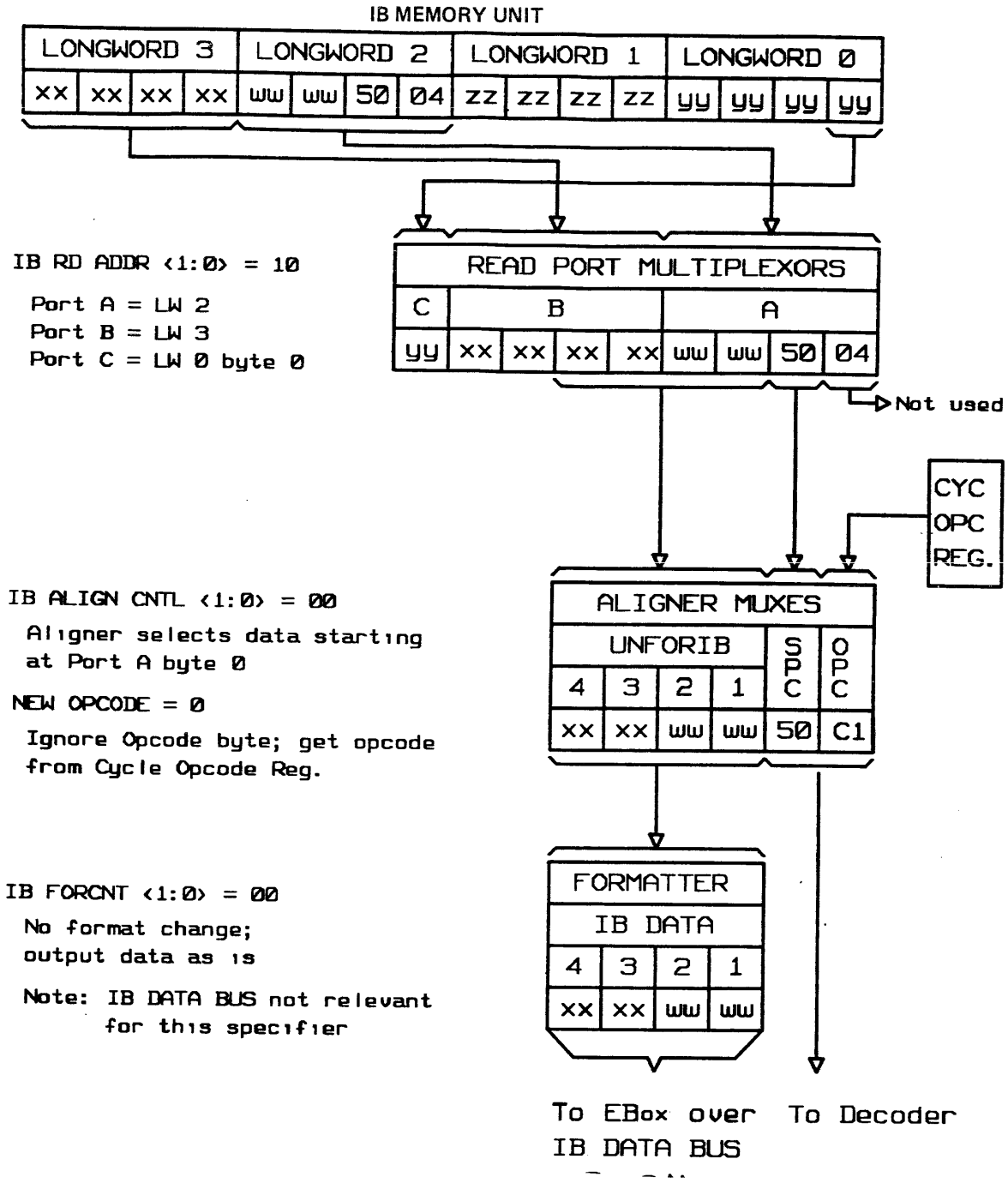
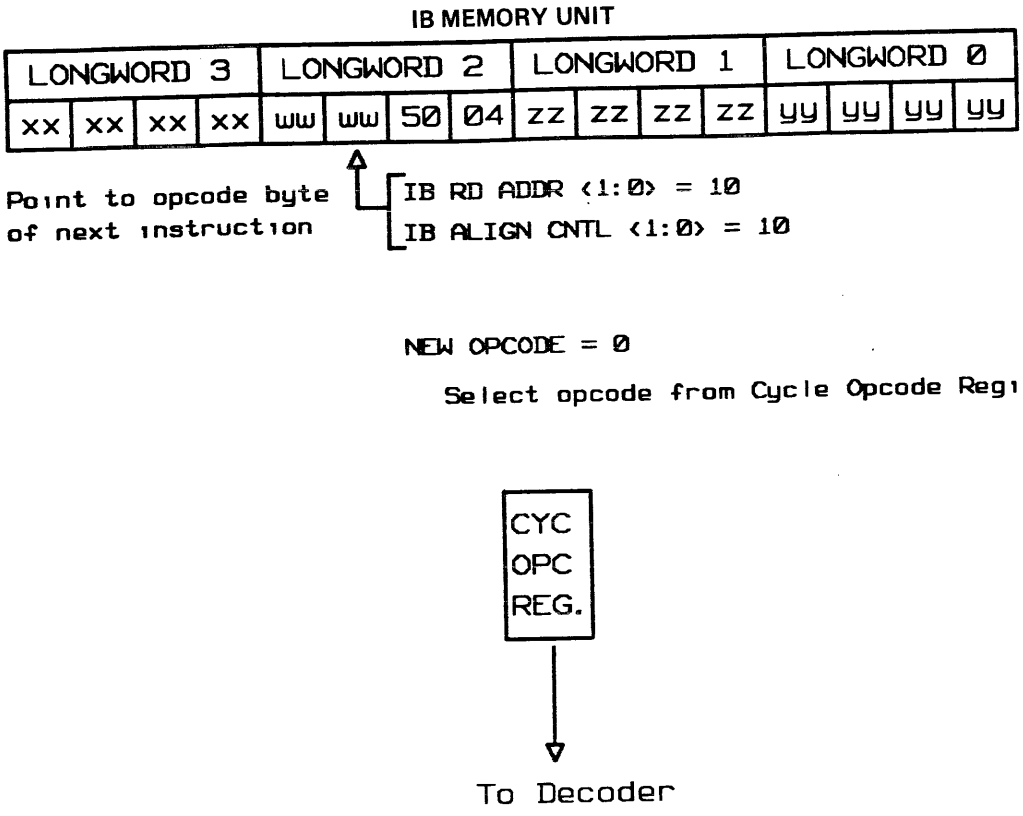


Figure 3-24 IB Read Example (Sheet 3 of 4)

ADDL3 #©X12345678, B©04 (R3), R0



NOTE: All outputs from IB except for the opcode are don't care during the opcode decode cycle

Figure 3-24 IB Read Example (Sheet 4 of 4)

ADDL3 #©X12345678, B©04 (R3), R0

3.4.5 IB Manager Operations

The IB Manager consists of the PCNC MCA and part of the IBST MCA.

The PCNC MCA (Figure 3-25) supplies the IB read address and alignment control inputs and computes the amount of IB data consumed during every decode cycle. It also decodes the opcode and current operand specifier and supplies some of the addressing inputs to the Decoder RAMs.

The IBST MCA supplies the IB write address and detects when a longword has entered the IB.

The following text describes the operations performed by the PCNC MCA. The IB manager related functions of the IBST MCA were discussed in preceding sections.

3.4.5.1 IB Read Address Logic -

Initializing IB RD ADDR <1:0> and IB ALIGN CNTL <1:0>

When microcode invokes an IB Flush, the signals EITHER FLUSH and ONE SHOT FLUSH are asserted. This causes the IB Read Address logic to initialize IB RD ADDR <1:0> and IB ALIGN CNTL <1:0> as follows:

- IB RD ADDR <1:0> = 0 0
- IB ALIGN CNTL <1:0> = VAX PC <1:0>

EITHER FLUSH causes the mux to the lower right of the IB Read Address logic to select the two inputs hardwired to logical 0 and the two that receive VA FILE <1:0>, which equal VAX PC <1:0> after a flush, from the EBox. The mux outputs are latched to become OLD PC <3:0>.

ONE SHOT FLUSH forces the Read Address logic to issue NO DATA IN IB to the IB Full logic which responds with NO LW AVAIL. This signal causes the Read Address logic to select OLD PC <3:2> as the source for IB RD ADDR <1:0> and OLD PC <1:0> as the source for IB ALIGN CNTL <1:0>.

NOTE

IB RD ADDR <1:0> and IB ALIGN CNTL <1:0> can be thought of as a four bit address into the IB and will be referred to collectively as the "IB pointer" in the rest of the text.

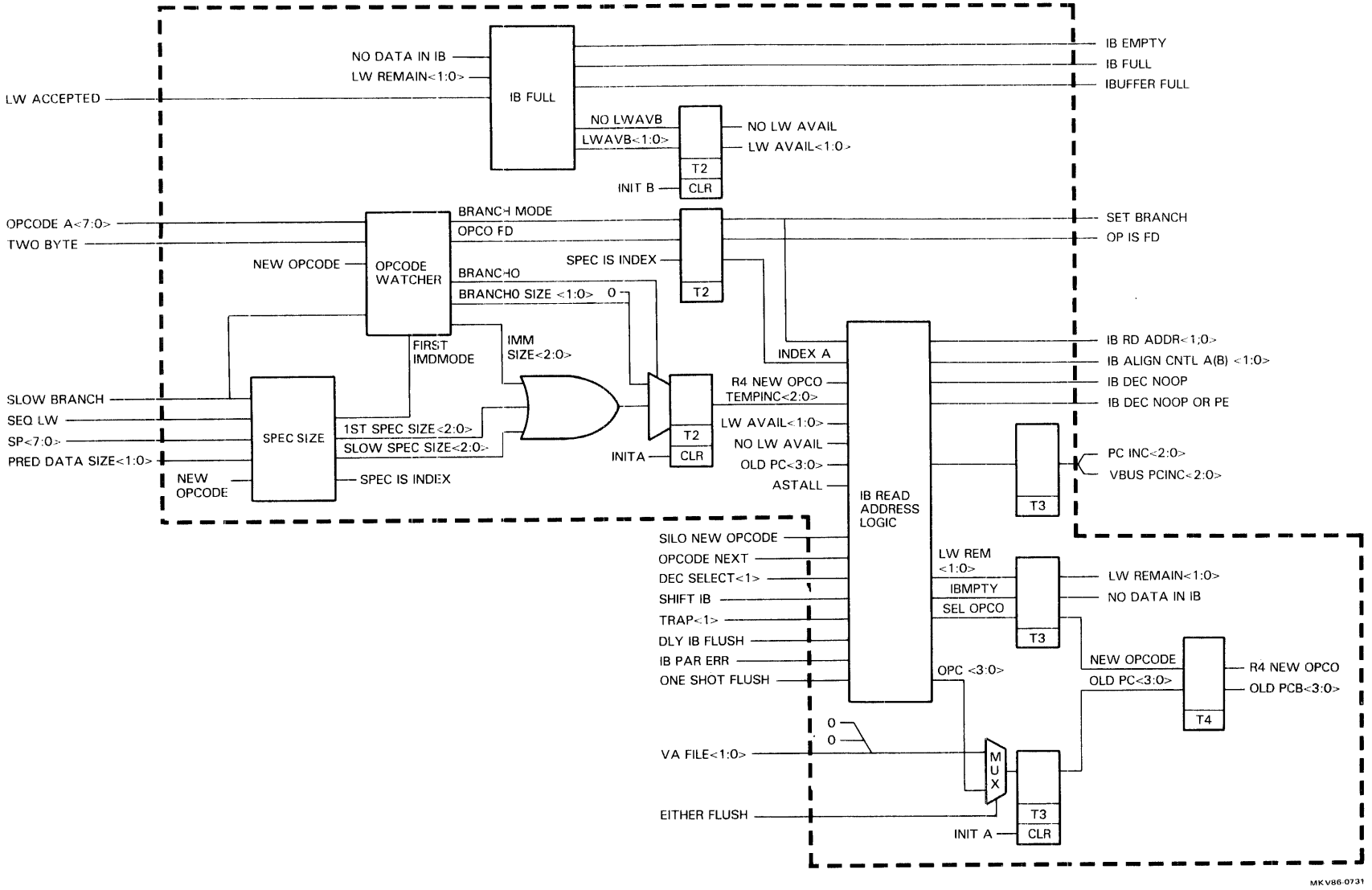


Figure 3-25 PCNC MCA Block Diagram

Computing Amount of IB Data Required

At the end of an IB flush, the CBox will deliver new I-stream data to the IB. When the first longword arrives and the IBST MCA asserts LW ACCEPTED, the IB Full logic negates NO LW AVAIL. This enables the IB Read Address logic to check if the IB contains enough I-stream data for the first decode cycle of the new instruction.

The amount of IB data required for any given decode cycle is indicated as a temporary increment value on the TEMPINC <2:0> bits. These bits, which become the PC INC <2:0> value sent to the EBox, are derived from one of two sources:

TEMPINC <2:0> Sources

Source	Cycle Active	Function
Opcode Watcher	First	Decode opcode byte and compute PC increment amount for branch offsets and immediate mode specifiers that occupy first specifier position. Note: Also active in second decode cycle for instructions with two byte opcodes.
Specifier Size logic	All	Compute PC increment amount in first cycle for those specifiers not handled by Opcode Watcher Compute PC increment amount for all subsequent specifiers.

3.4.5.2 Opcode Watcher Logic - The Opcode Watcher is enabled by the signal NEW OPCODE. This signal is asserted in the first cycle of every instruction and, for the FD class two byte opcodes, along with TWO BYTE in the second decode cycle.

When NEW OPCODE is asserted, the Opcode Watcher checks the signal FIRST IMDMODE to see if the first specifier is immediate mode.

If FIRST IMDMODE is negated, the Opcode Watcher then determines if the instruction falls into one of the following categories:

- No operands - NOP, HALT, etc.
- FD class two byte opcode - ADDG2, MOVO, etc.
- Branch offset in first specifier - BNEQ, BSBW, etc.

If the instruction falls into any of these categories, the Opcode Watcher will indicate the amount of IB data required for the first decode cycle on the BRANCH0 SIZE <1:0> bits as follows:

BRANCH0 SIZE <1:0>	Instruction Class
0 1	No operands or first byte of two byte opcode
1 0	Branch with byte offset (eg: BNEQ, BSBB)
1 1	Branch with word offset (eg: BRW, BSBW)
0 0	Otherwise

Note that BRANCH0 SIZE <1:0> equal 2 for a branch instruction with a byte offset and 3 for a word offset. This is because the opcode byte is always consumed along with the first specifier bytes in the first decode cycle of an instruction. (The only exception is the first byte of a two byte opcode which is given in its own decode cycle.)

In addition to encoding BRANCH0 SIZE <1:0>, the Opcode Watcher also asserts the following signals:

Signal	Function
BRANCH0	Forces TEMPINC mux to select BRANCH0 SIZE <1:0>. (TEMPINC bit <2> is forced to logic 0; IB Read Address logic requires 3 increment bits.)
SET BRANCH	Prevents IB Read Address logic from using signal INDEX from Spec Size logic. SET BRANCH also asserted when a subsequent specifier is a branch offset (eg: last specifier of ACBB.)
OP IS FD	Asserted if instruction has FD class two byte opcode. OP IS FD is stored in the IB State Silo (IBST MCA) and returned as TWO BYTE in second cycle of instruction to allow Opcode Watcher to decode second byte of two byte opcode.

When FIRST IMDMODE is asserted, the Opcode Watcher must first decode the opcode to determine the data size operated on by the instruction. It then computes the amount of IB data required and encodes the amount on the IMM SIZE <2:0> bits.

Since the amount of IB data consumed in the first decode cycle includes the opcode, the first specifier, and up to four specifier extension bytes, the IMM SIZE value ranges from three to six as follows:

Extension Size	IMM SIZE <2:0>	Example
Byte	011	MOVB #©X12, R0
Word	100	MOVW #©X1234, R0
Longword	110	MOVL #©X12345678, R0

Note: The PC increment value for the subsequent longwords of big immediate specifiers is computed by the Spec Size logic.

3.4.5.3 Specifier Size Logic -

First Decode Cycle

If the first specifier is not of a type handled by the Opcode Watcher, the Specifier Size logic will provide the PC increment value as 1ST SPEC SIZE <2:0>.

Since some specifiers have no extension bytes and others have up to four, the 1ST SPEC SIZE <2:0> value can range from two (opcode, first specifier) to six (opcode, specifier, longword extension) as follows:

1ST SPEC SIZE <2:0> Values

New Opcode	SP <7:0>	1ST SPEC Size <2:0>
	00 - 8F	010
	90 - 9E	010
1	9F	110
	A0 - BF	011
	C0 - DF	100
	E0 - FF	110
0	x	000

Subsequent Decode Cycle

The Specifier Size logic provides the PC increment value in subsequent decode cycles (2nd to 6th specifiers) as SLOW SPEC SIZE <2:0>.

The SLOW SPEC SIZE <2:0> value is based on the current specifier byte and on three control signals from the IBST MCA. Table 3-16 describes the control signals, and Table 3-17 indicates the SLOW SPEC SIZE <2:0> value for each specifier type.

Table 3-16 Specifier Size Logic Control Signals

Signal	Indication
SLOW BRANCH	Second to sixth specifier is a branch offset (eg: last specifier of all ACBx instructions is a branch offset). Derived from RAM SLOW BRANCH from Decoder RAM.
PRED DATA SIZE <1:0>	Size of a branch offset or immediate mode specifier that occupies second to sixth specifier position. Derived from PRED DATA SIZE <2:0> from Decoder RAM.
SEQ LW	Subsequent longword of a big immediate specifier is being processed. Derived from IBST DATA SIZE <2:0> from Decoder RAM.

Notes:

1. The signals are not used during the second cycle of FD class two byte opcodes (NEW OPCODE still asserted).

2. The signals are available in current decode cycle but indicate data type/size for next specifier.

3. The SLOW BRANCH signal also forces the Opcode Watcher to unconditionally assert the SET BRANCH signal.

Table 3-17 Slow Spec Size <2:0> Values

Slow Branch	SP <7:0>	PRED Data Size <1:0>	SEQ LW	Slow Spec Size <2:0>
	00 to 8E	-	-	0 0 1
	8F	0 0	0	1 0 1
			1	1 0 0
		1 0	-	0 1 0
		1 1	-	0 1 1
0	90 to 9E	-	-	0 0 1
	9F	-	-	1 0 1
	A0 to BF	-	-	0 1 0
	C0 to DF	-	-	0 1 1
	E0 to FF	-	-	1 0 1
1	xx	0 0	-	1 0 0
		0 1	-	0 0 1
		1 0	-	0 1 0

3.4.5.4 Checking TEMPINC <2:0> Validity - Since IB reads occur on every B clock, there is a possibility that the TEMPINC at value could be based on I-stream data left over from a previous instruction. This means that before it can use the TEMPINC value, the IB Read Address logic must first ensure that the value is based on current IB data.

The read address logic determines the validity of the TEMPINC <2:0> value based on the signals NO LW AVAIL, LW AVAIL <1:0> and OLD PC <1:0>.

The signals NO LW AVAIL and LW AVAIL <1:0> are output from the counter which the IB Full logic (top of Figure 3-25) maintains to keep track of the number of longwords in the IB available for decoding:

NO LW AVAIL	Asserted if all longwords currently in the IB have already been processed (IB is "empty").
LW AVAIL <1:0>	Encoded with the number of IB longwords yet to be decoded. Only valid if NO LW AVAIL is negated.

NO LW AVAIL Asserted

In this case, the IB Read Address logic knows that the TEMPINC value is invalid since it could only be based on old IB data. The IB Read Address logic will therefore ignore LW AVAIL <1:0> and OLD PC <1:0> and initiate a "Decoder Stall" operation (described later).

NO LW AVAIL Deasserted

The deasserted state of NO LW AVAIL only informs the IB Read Address logic that the IB contains some valid data, not if it contains enough data for the current decode cycle.

To determine if the IB contains enough data, the IB Read Address logic first computes the number of IB bytes available from the OLD PC <1:0> bits (starting IB byte position) and the LW AVAIL <1:0> bits (total number of IB longwords available). It then compares this amount to the TEMPINC <2:0> value.

If the IB contains enough data, the IB Read Address logic may or may not use the TEMPINC <2:0> value in the current decode cycle depending on whether the IB pointer is to be updated or remain unchanged for the next cycle.

If the IB does not contain enough data, the IB Read Address logic will initiate the "Decoder Stall" operation described below.

3.4.5.5 Decoder Stall - When the IB does not contain enough data for the current decode cycle, the IB Read Address logic will initiate a "Decoder Stall" operation by asserting the signal IB DEC NOOP OR PE.

The assertion of IB DEC NOOP OR PE causes the Special Address Encoder logic to output the address of an OS.IB.STALL microword in place of the address that the Decoder would have otherwise generated (opcode or specifier entry address).

The only function of the OS.IB.STALL microword is to request another IB decode cycle (I_DECODER bit set). This effectively "stalls" the decode process for one cycle and allows the CBox more time to deliver the required data to the IB.

If the CBox fails to supply a new longword of I-stream data by the end of the "stall" cycle or if more than one longword is required, the IB Read Address logic will keep IB DEC NOOP OR PE asserted, forcing the Special Address Encoder to again generate the OS.IB.STALL microword address. This operation will continue until the CBox delivers enough I-stream data for the current decode cycle.

3.4.5.6 Modifying the IB Pointer - When the IB contains enough data for the current decode cycle, the IB Read Address logic examines its other inputs to determine if the IB pointer should be modified by the TEMPINC <2:0> value or remain in its current state for the next cycle.

Figure 3-26 is a conceptual block diagram of the IB Read Address logic that controls the IB pointer. The logic at the top of the figure determines if the IB pointer should point to the opcode byte in the first decode cycle of an instruction or to the byte preceding the next specifier for a subsequent cycle. The logic at the bottom of the figure determines if the pointer will stay in its current state or be modified by the TEMPINC <2:0> value.

Table 3-18 shows the relationship between the inputs to the IB Read Address logic and the state of the IB pointer for the next decode cycle. The mnemonic OPC refers to the OLD PC <3:0> bits, TPC refers to TEMPINC <2:0>. The indicator "-" means that the given input signal is "don't care". For example, if DLY IB FLUSH is true (1), all other signals are ignored.

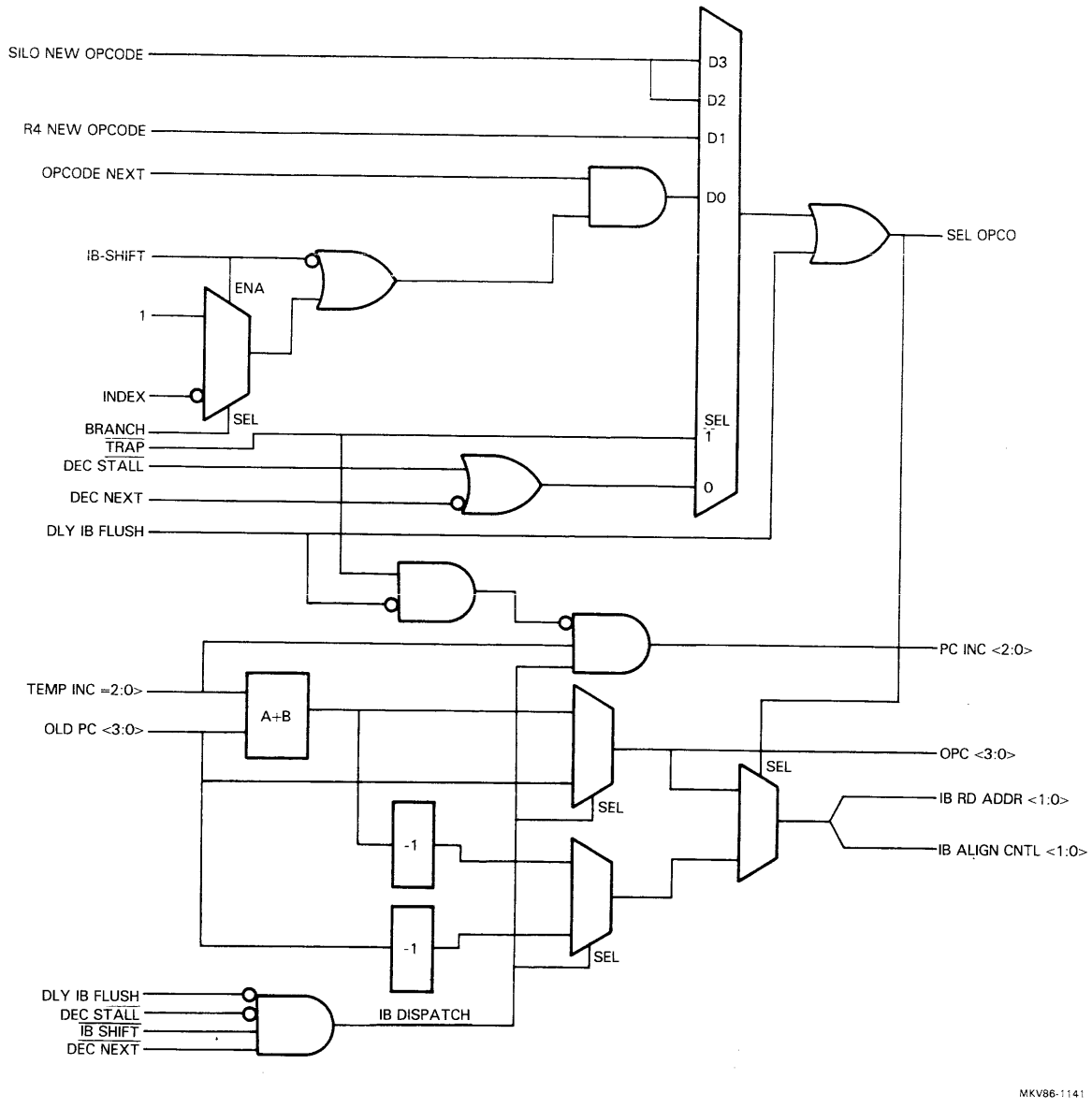


Figure 3-26 Simplified IB Read Address Logic

DLY IB FLUSH	TRAP	DEC STALL	DEC SELECT	IB SHIFT	SET BRANCH	INDEX	OPCODE NEXT	R4 NEW OPCODE	SILO NEW OPCODE	New IB Pointer	REMARKS
1										OPC	Full IB Flush
0	1								1 0	OPC OPC -1	Trap during first decode cycle Subsequent cycle
	0	1						1 0		OPC OPC -1	Stall on first decode cycle Subsequent cycle
		0	0					1 0		OPC OPC -1	No IB decode - first cycle Subsequent cycle
			1	0			1 0			OPC OPC -1	Execute cycle - most inst. Shouldn't happen
				1	0	0	1 0			OPC + TPC OPC + TPC -1	Execute cycle - optimized inst. Normal subsequent spec. cycles
						1				OPC + TPC -1	Index mode specifer
					1		1 0			OPC + TPC OPC + TPC -1	Last specifer branch offset Shouldn't happen

OPC = OLD PC <3:0>
TPC = TEMPINC <2:0>

MKV86-1140

3.4.5.7 IB Read Address Logic Control Signals - This section describes the IB Read Address logic control signals in the order in which they appear in Table 3-18.

DLY IB FLUSH

This signal is asserted during a full IB flush to force the unmodified OLD PC <3:0> bits to become the new IB pointer. (The OLD PC <3:0> bits are initialized during a full or partial flush to point to the opcode of the new instruction delivered to the IB.)

TRAP

The TRAP signal indicates that a microtrap occurred.

Microtraps are detected late in the pipeline (canonical T10 time). Thus, the assertion of TRAP does not necessarily mean that the current instruction caused the trap.

To determine which cycle was in progress at the time of a trap, the IB Read Address logic saves the state of the signal SILO NEW OPCODE.

SILO NEW OPCODE is a four cycle old copy of the NEW OPCODE signal that the IBST MCA issues at the start of a new instruction and, as such, indicates which decode cycle was in progress as follows:

SILO NEW OPCODE	Decode Cycle in Progress
Asserted	First cycle
Negated	Subsequent cycle

When TRAP is asserted, SILO NEW OPCODE is output as SEL OPCO, latched (see Figure 3-25), and returned as R4 NEW OPCODE.

During the ensuing microtrap routine, both TRAP and DEC SELECT are negated, forcing R4 NEW OPCODE to be recirculated until the routine exits. (Microcode convention does not allow "nested" traps or decode cycles within trap routines.)

At the end of the microtrap routine, a partial IB flush is invoked which initializes the OLD PC <3:0> bits. R4 NEW OPCODE is then used to select the appropriate IB pointer source:

Trap During	R4 NEW OPCODE	IB Pointer Source	Reason
First Cycle	True	Unmodified OLD PC	Points to opcode byte of trapped instruction
Subsequent Cycle	False	OLD PC-1	Points to byte preceding specifier being processed when trap occurred

In either case, the IB pointer will address the proper IB byte in the first decode cycle after the trap is released.

NOTE

Microtraps that occur in the "shadow" of a full flush (DLY IB FLUSH asserted) are ignored by the logic.

DEC STALL

Indicates that a Decoder stall condition is present.

When DEC STALL is true, the IB Read Address logic uses R4 NEW OPCODE to determine if the stall occurred during the first decode cycle of a new instruction or in some subsequent cycle:

Stall during	R4 NEW OPCODE	IB Pointer Source
First cycle	True	Unmodified OLD PC
Subsequent cycle	False	OLD PC -1

R4 NEW OPCODE is also used when DEC STALL and DEC SELECT are both negated. This condition is the norm when there is no Decoder stall or IB decode request. The selection of the source for the new IB pointer is as described above.

IB Quiescent State

The above operations have one thing in common: they all force the IB pointer to stay in its current state until an IB "shift" is requested along with DEC SELECT (see below) and the IB contains enough data.

To ensure that the IB pointer does not change state for the next cycle, the IB Read Address logic recirculates the OLD PC <3:0> value. It does this by latching OPC <3:0> and returning the value in the next cycle as OLD PC <3:0> (see Figure 3-25 for the external latches.)

The reason for recirculating the OLD PC <3:0> value in this manner is that if the IB pointer mux were to be used instead, the assertion of SEL OPCO would cause the pointer to be decremented by 1 in the next cycle. (The logic would select and decrement the OLD PC-1 value.)

The following text describes the remaining inputs to the Read Address logic assuming that DLY IB FLUSH, TRAP, and DEC STALL are all negated.

DEC SELECT

This signal is asserted each time microcode issues a decode request and causes two operations to occur:

1. Microsequencer selects Decoder generated entry address for the specifier or opcode microroutine

2. PCNC MCA modifies (if required) the IB pointer in preparation for the next decode cycle

When it receives DEC SELECT, the IB Read Address logic first examines the state of SHIFT IB from the Decoder RAMs (DRAMs) to determine if it should keep the IB pointer in its current state or modify it for the next cycle.

SHIFT IB

SHIFT IB is always asserted by the DRAMs in the first decode cycle of every instruction and whenever the next block of I-stream data is to be shifted out of the IB.

If SHIFT IB is asserted, the IB Read Address logic selects either the OLD PC + TEMPINC value or the OLD PC + TEMPINC-1 value as the source of the new IB pointer. If the signal is negated, either the unmodified OLD PC value or the OLD PC-1 value is selected.

The selection of one of these sources depends on the state of three other input signals which are used in conjunction with SHIFT IB:

SET BRANCH	Asserted if the specifier is the branch offset of a PC control (BNEQ, BBC, etc.) or a loop control (ACBB, AOBLEQ, etc.) instruction. Also asserted in the first, and only, decode cycle of an instruction that has no specifiers (HALT, NOOP, etc.), and in the first cycle of an instruction with a two byte opcode.
INDEX	Asserted if the specifier is index mode.
OPCODE NEXT	Asserted by the DRAMs if the next decode cycle will be the first cycle of a new instruction.

Note that SHIFT IB and OPCODE NEXT, like all other DRAM outputs, are issued during the current decode cycle but control operations for the next cycle. Therefore, the signals are available prior to the time the logic receives DEC SELECT from microcode.

The following assumes SHIFT IB is asserted when the current microcode routine issues DEC SELECT.

SET BRANCH

Since a branch offset is always the last specifier of a macrobranch instruction, the next decode cycle will always be the first cycle of a new instruction.

If SET BRANCH is true, the IB Read Address logic ignores the INDEX signal, which is meaningless is this case, and selects the OLD PC + TEMPINC value. This will be the pointer to the opcode byte of the next instruction should the branch fail. (On a branch success, the ensuing IB flush operation would initialize the pointer.)

This method is also used to modify the IB pointer for the second byte of a FD class two byte opcode and to point to the opcode of the instruction following one that has no specifiers.

NOTE

> As long as the DRAMs are properly loaded and no hardware failure exists, the case in which SHIFT IB and SET BRANCH are both asserted and OPCODE NEXT negated should never occur.

INDEX

The assertion of INDEX with SET BRANCH negated means that the current specifier is indexed mode.

In this case, the IB Read logic will ignore OPCODE NEXT and select the OLD PC + TEMPINC-1 value as the new IB pointer. This will ensure that the pointer will address the base operand in the second decode cycle of the specifier.

(It takes at least two cycles to process an index mode specifier: one for the index operand and one for the base operand. If the base is a big immediate, additional cycles are required, one for each subsequent longword.)

The INDEX signal is especially important when the last specifier of an instruction is indexed since the base operand byte would otherwise be interpreted as the opcode of a new instruction.

OPCODE NEXT

When SET BRANCH and INDEX are both negated, the state of OPCODE NEXT determines whether the IB pointer should address the opcode byte of a new instruction or a subsequent specifier of the current instruction.

If all specifiers of the current instruction have yet to be processed, the DRAMs keep OPCODE NEXT negated. This forces the IB Read Address logic to select the OLD PC + TEMPINC-1 value which points to the byte preceding the next specifier to be processed.

After the last specifier is processed, the DRAMs negate SHIFT IB and assert OPCODE NEXT. This forces the IB Read Address logic to select the unmodified OLD PC value since this value points to the opcode byte of the next instruction.

NOTE

> OPCODE NEXT also causes the IBST MCA to issue NEW OPCODE to the IBUF MCAs. However, the IBUF MCAs will not use this signal until the next decode cycle.

Optimized Instructions

The only exception to the operand specifier processing described above is the decode cycle for the last specifier of "optimized" instructions such as a MOVL. In this cycle, the DRAMs assert both SHIFT IB and OPCODE NEXT to force the selection of the OLD PC + TEMPINC value. This is because the execute code for optimized instructions is incorporated in the last specifiers microroutine. Thus, there is no decode cycle to generate the entry point address for the execute microcode.

3.4.5.8 Computing Number Of IB Longwords Consumed - When it updates the IB pointer, the IB Read Address logic must also compute the number of IB longwords consumed during the decode cycle. It does this by:

1. Summing TEMPINC <2:0> (bytes requested) to OLD PC <1:0> (starting byte position). This yields the number of IB longword boundaries that will be crossed accessing the current specifier.

2. Subtracting above sum from LW AVAIL <1:0>. This effectively compares the number of longwords requested to the number actually available in the IB.

NOTE

LW AVAIL <1:0> only have meaning if NO LW AVAIL is false. In addition, with NO LW AVAIL false, the bits always equal 1 less than the number of longwords actually available. This is, they equal 0 if 1 longword is available, 1 if 2 two longwords are available, etc.

Based on the result of this comparison, the IB Read Address logic then performs one of the operations below (the symbol ">" means greater than; "<" means less than).

Result	Amount of Data in IB	IB Read Address Logic Response
= 0	Just enough	Clear LW REMAIN <1:0> and assert NO DATA IN IB. IB Full logic will issue NO LW AVAIL to cause a Decoder stall in the next cycle if the CBox does not deliver a new longword in time.
> 0	More than enough	Negate NO DATA IN IB and indicate number of longwords remaining as LW REMAIN <1:0>. LW REMAIN <1:0> become LW AVAIL <1:0> in the next cycle if the CBox does not deliver a new longword.
< 0	Not enough	Return LW AVAIL <1:0> and NO LW AVAIL to the IB Full logic as LW REMAIN <1:0> and NO DATA IN IB. Note: The IB Read Address logic would have already invoked a Decoder stall by asserting IB DEC NOOP OR PE.

3.4.6 Instruction Decoder Operation

Refer to Figure 3-27. The instruction decode logic consists of the following components:

- 4K x 17 bit Decoder RAM (DRAM)
- Special Address Encoder
- Part of the IBST MCA

Primary Functions

- Decode the opcode and current operand specifier
- Generate the entry point address for all operand specifier microroutines
- After all specifiers are processed, generate the entry point address for the routine that performs the actual work of the instruction (the execute code)
- Monitor "special" conditions which may affect instruction execution and generate the entry point address for the routine which services the condition
- Assist the PCNC and IBST MCAs in controlling the IB

Decoder generated microaddresses, as with all microaddresses, are 14 bits wide. Two copies of the microaddress are sent to the microsequencer, DEC UADDRA <13:00> and DEC UADDRB <13:00>. This reduces signal loading of the CS0 micro-PC address latches.

3.4.6.1 Pipeline Timing Considerations - The Decoder operates asynchronous to the pipeline in that it is not explicitly controlled by microcode. However, there is one microword bit, I_DECODER, which effectively couples the Decoder to the pipeline.

I_DECODER is set in the last microword of every specifier and opcode routine to initiate the next IB decode cycle. The bit is output from its CS0 RAM data latch as the signal DEC SELECT which is sent to the micro-PC address latches.

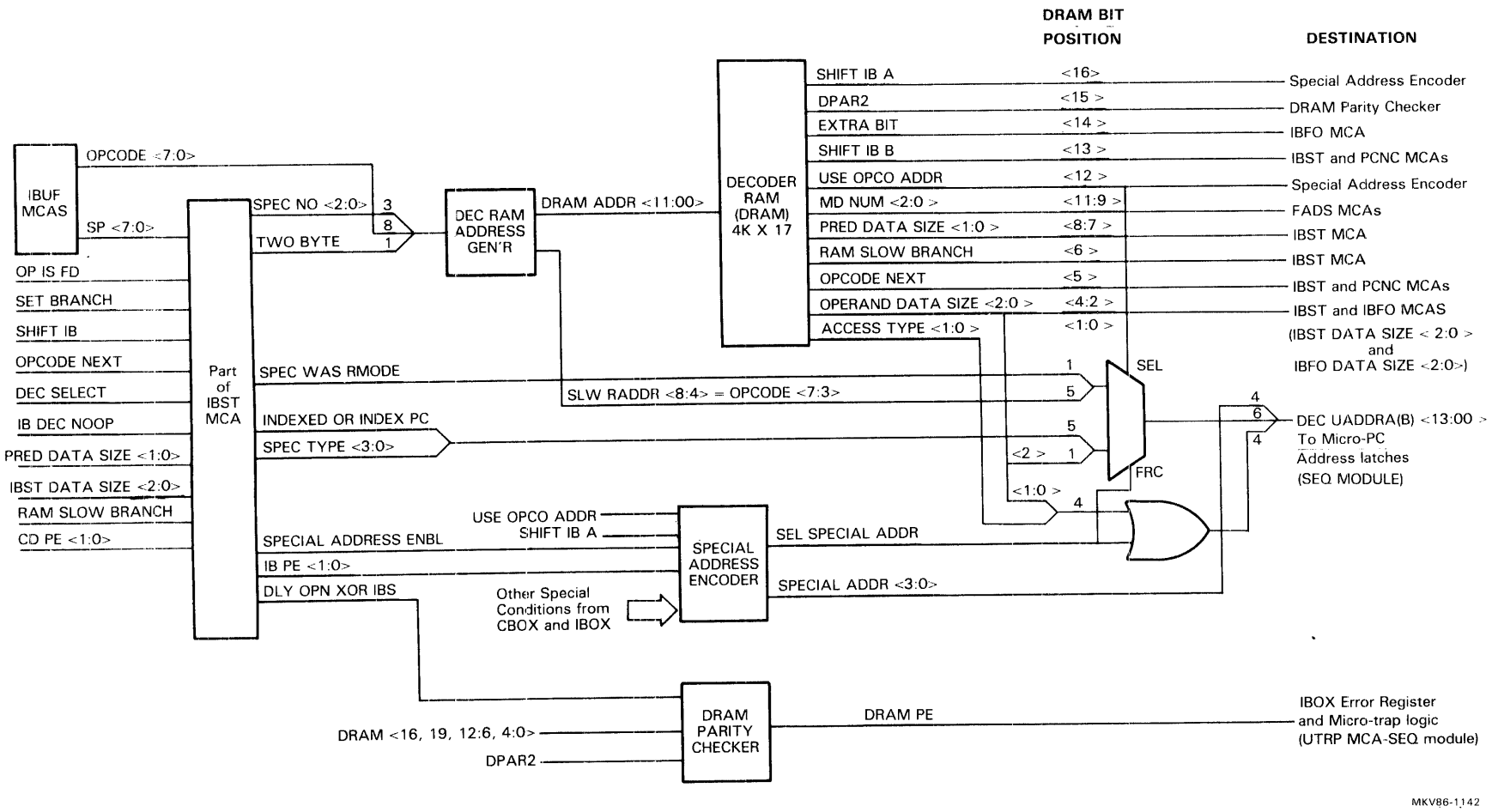


Figure 3-27 Instruction Decode Logic

The state of the DEC SELECT signal determines the source of the next microaddress selected by the micro-PC address latches:

DEC SELECT	Next Microaddress Source
Asserted	Decoder logic. This operation is commonly referred to as a "Dec Next".
Negated	Microsequencer logic (UBRS MCAs)

Decoder to Pipeline Basic Timing Relationship

I_DECODER is output from the CS0 RAM at canonical T3 of the current microword. The bit is then latched as DEC SELECT at T4 and presented to the micro-PC address latches at T5.

Canonical T3 to T5 of the current microword correspond with T1 to T3 of the next microword to be executed. It is during these time states that the IB, the Decoder, and the micro-PC address latches perform their respective functions:

Time	Operation
T1	IB outputs next block of I-stream data
T2	Decoder generates entry address for next specifier (or opcode) routine.
T3	CS0 micro-PC address latches select Decoder supplied address if DEC SELECT is asserted (I_DECODER set).

If DEC SELECT and the signal SHIFT IB from the DRAM are both asserted, the IB will output the next block of I-stream data to be processed. The decoder will then generate the entry address for the next specifier routine (or execute code if no more specifiers) and make it available to the micro-PC latches by the next T3 time.

If either DEC SELECT or SHIFT IB is negated, the IB will output the same block of I-stream data. In this case, the Decoder will output the same address in the next cycle.

3.4.6.2 Operand Specifier Entry Point Addresses - The entry point address for an operand specifier microroutine is derived from three sources.

Figure 3-28 shows the format of the entry point address and the source of each bit.

Table 3-19 briefly describes how the address bits are derived and used.

Table 3-20 shows the relationship between the operand specifiers data size and its access type. The notation used in Table 3-20 is the same as that used on the VAX programming card. For example, .RW means that the operand is read access, data size is word.

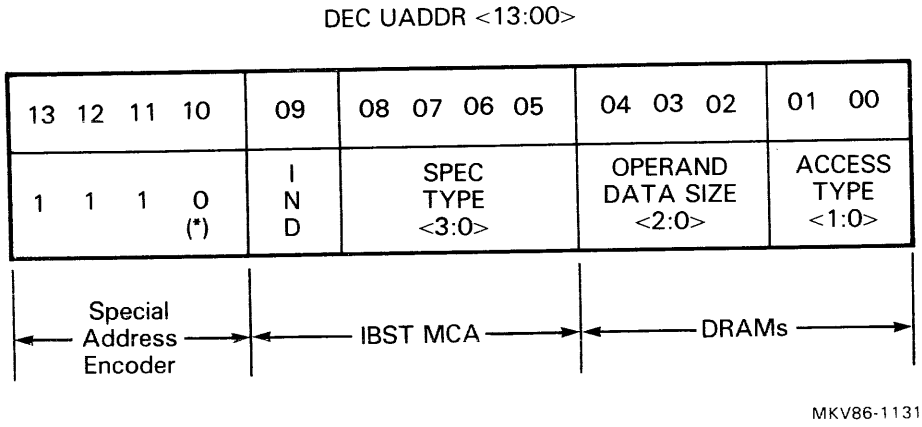


Figure 3-28 Operand Specifier Entry Address Format

Table 3-19 Operand Specifier Entry Address Bit Descriptions

Bit(s)	How Derived
13:10	Forced to state indicated by Special Address Encoder if no special condition is pending (discussed later).
(*)	Bit 10 is DRAM signal USE OPCO ADDR which is negated for all specifiers except branch offsets. Branch offsets do not have their own decode cycles since they are serviced by the execute microcode (see text).
9	INDEXED OR INDEX PC from the IBST MCA. Asserted if the current decode cycle is to process the base operand of an index mode specifier. (The index byte would have been processed in the previous cycle; see text.)
08:05	Based on either the specifier byte, SP <7:0>, from the IB or, for the subsequent longwords of big immediates, on the IBST DATA SIZE <2:0> bits from the DRAM. Represents either the specifier type or which longword of a big immediate specifier is being processed: 0 - Literal 1 - PC absolute (all sizes) 2 - PC relative (all sizes) 3 - PC relative deferred (all sizes) 4 - Index (index byte, not the base) 5 - Register 6 - Register deferred 7 - Auto-decrement 8 - Auto-increment 9 - Auto-increment deferred A - Displacement (all sizes) B - Displacement deferred (all sizes) C - Immediate: Byte, Word, Longword, F-float or, 1st LW of Quad/Octaword, D/G/H-float. D - Immediate: 2nd LW of Quadword, D/G-float or, 4th LW of Octaword, H-float. E - Immediate: 2nd LW of Octaword, H-float. F - Immediate: 3rd LW of Octaword, H-float.
04:00	DRAM bits <4:0>. Encoded with the data size and access mode of the current operand except for certain special cases. See Table 3-20.

Table 3-20 Operand Data Size/Access Type Correlation

OPERAND DATA SIZE <2:0> --> DEC UADDR <4:2>	ACCESS TYPE <1:0> --> DEC UADDR <1:0>	Operand Specifier Notation or Special Case
0	0	Not used - Reserved for TRAP vector address space
	1	Not used - Reserved for future expansion
	2	Illegal opcode
	3	Not used - Reserved for future expansion
1	0	.RW
	1	.MW
	2	.AW
	3	.WW - Also, .MW for ADAWI only
2	0	.RL, .RF
	1	.ML, .MF
	2	.AL, .AF
	3	.WL - Except for special .WLs below
3	0	Special - .WL and set CCs' for optimized instructions
	1	Special - .WL 3rd spec of EDIV; 4th spec of EMODs'
	2	.VB - Normal case (eg: BBC through BBSSI)
	3	.VB - Abnormal case (eg: CMPV, CMPZV)
4	0	.RD, .RG, .RQ
	1	.MD, .MG, .MQ
	2	.AD, .AG, .AQ
	3	.WD, .WG, .AQ
5	0	.RB
	1	.MB
	2	.AB
	3	.WB
6	0	.RH, .RO - Except first specifier
	1	.MH - There are no .MO specifiers
	2	.AH, .AO
	3	.WH, .WO
7	0	.RH, .RO - First specifier only
	1	.MH - ACBH only
	2	Illegal - Should never be encoded
	3	Illegal - Reserved for Special Addresses

Operand Specifier Entry Point Address Labels

Each operand specifier entry address is assigned a unique label in the OPSP.MIC (OPerand SPecifier MICrocode) module of the microcode.

Table 3-21 shows the operand specifier entry address label format and the mnemonics used in each label field.

Example: Specifier is (Rn)+, not indexed, read access, longword

.B

Address	Label
3908	OS.AINC.NI.RD.LF

Reserved Addressing Mode Faults

Certain specifier addressing mode and access type combinations result in reserved addressing mode faults. For example, REGISTER mode with ADDRESS access.

When a reserved addressing mode fault occurs, the Decoder will still generate an entry address for the specifier. However, all routines entered in this manner contain code to immediately transfer control to a reserved addressing mode fault service routine.

The reserved addressing mode fault service routine resides in the IANDE.MIC module. The routine starts at the label IE.FLT.ILL.ADR and is entered by the GOTO macroexpression as follows:

Example: Specifier is RMODE, not indexed, address access, longword

Address	Label	Microword contents
38AA	OS.REG.NI.ADR.LF	GOTO [IE.FLT.ILL.ADR]

Table 3-21 Operand Specifier Entry Address Symbolic Labels

Label format: OS.AAA.BBB.CCC.DDD

Field	Meaning	Mnemonics		
OS	OPSP.MIC module	N/A		
AAA	Address mode	LIT	Literal	General Register Addressing Modes
		IND	Indexed	
		REG	Register	
		RDEF	Register deferred	
		ADEC	Auto-decrement	
		AINC	Auto-increment	
		AIDEF	Auto-increment deferred	
		DIS	Displacement	
		DSDEF	Displacement deferred	
		IMM	Immediate	PC Addressing Modes
		ABS	Absolute	
		REL	Relative	
		RLDEF	Relative deferred	
BBB	Index mode	I	Indexed	
		NI	Not indexed	
CCC	Access mode	ADR	Address	
		ILL2B	Illegal opcode	
		MOD	Modify	
		RD	Read	
		WRT	Write, normal case	
		WRTCC	Write and set CCs' - used for optimized instructions	
		WRTNLST	Write/not last specifier - 3rd spec of EDIV, 4th spec of EMODs'	
		VLD	Vield - last operand only	
		VLDRD	Vield - not last operand	
DDD	Data size	BYTE	Byte	
		WORD	Word	
		LF	Longword and F-float	
		DGQ	D/G-float, Quadword	
		HO	H-float, Octaword (not first operand)	
		HOFST	H-float, Octaword (first operand, read only)	
		H3	H-float 3rd operand (modify only)	
		ODD1	Special; used with WRTCC, WRTNLST, VLD and VLDRD	
		ODD2	Special; used with ILL2B	

The major functions of the IE.FLT.ILL.ADR routine are to:

1. Restore the VAX GPRs to the state they were in prior to the fault

2. Form the exception vector of 1C (hex) into the System Control Block (SCB)

When the fault routine exits, it passes control to a routine which will push the PSL and PC onto the appropriate stack and report the fault to the system software. Software will determine if the instruction can be restarted or should be aborted.

One entry address per specifier

With the following exceptions, the Decoder will generate only one entry address for each specifier:

1. No entry address is generated for a branch offset. The offset is processed by the execute code.

2. Index mode specifiers require two entry addresses, one for the index and one for the base. (If the base is a big immediate, item 3 also applies.)

3. Big immediate (but not big literal) mode specifiers require a separate entry address for each longword.

   Big literals only require one entry address since the routine that services the specifier supplies the additional longwords.

Branch offsets

There are two general classes of macrobranch instructions:

Branch Class	Description	Decoder Action
Simple branch (BNEQ, BRB, BRW, etc.)	Offset is the first, and only, specifier	Immediately generate entry address for the execute code.
Loop control (ACBx, BBx, AOBx, etc.)	Offset is always the last specifier	Generate entry address for execute code after next to last specifier is processed.

In either case, the branch offset is sign extended, output to the IB DATA BUS, and summed with the VAX PC during the execute code.

Index mode specifiers

The signal INDEXED OR INDEX PC from the IBST MCA determines if the Decoder is to generate the entry point address for the index or for the base operand microroutine.

Indexed or Index PC	Indication
Negated	If SPEC TYPE <3:0> = 4, specifier is an index operand. Otherwise, specifier is not indexed.
Asserted	Specifier is the base operand.

Big immediate specifiers

Since Quadword, D-Float, and G-Float data are all 64 bits wide, the value represented by OPERAND DATA SIZE <2:0> for these data types is the same. This means that the routine that services the first longword of a Quadword big immediate is also the one used for the first longword of a D or G-float immediate. This also applies to Octaword and H-Float data since these data types are both 128 bits wide.

As each longword of a big immediate specifier is processed, the IBST MCA updates the SPEC TYPE <3:0> bits so that a different routine is entered to service the next longword. This ensures that the EBox logic will be conditioned according to the longword being processed.

The IBUF MCAs and the IBFO MCA ensure that each longword is properly formatted before being sent to the EBox.

Specifier routine length

Some specifier types can be handled by a single microword while others require two or more microwords:

- Specifier types serviced by single microword routines

  Register and Register deferred.
  All displacement modes except deferred.
  All PC relative modes except deferred.
  Literals and immediates not bigger than a longword.

- Specifier types serviced by multiple microword routines

  Auto-increment (except PC auto-increment).
  Auto-decrement.
  All deferred modes except register deferred.
  PC absolute mode.
  Big literals and immediates.
  All indexed modes.

Microword bit I_DECODER is always set in the last microword of every specifier routine. For single word routines, this means that the decode cycle for the next specifier is requested only one clock cycle after the current routine starts (next T3 time). For multiple word routines, at least two cycles elapse before the next decode is requested.

In either case, if the IB does not contain enough data for the next decode cycle, the Special Address Encoder will continually output the address of an OS.IB.STALL microword (3BFF), stalling the decoder until the CBox delivers the required data to the IB.

3.4.6.3 Opcode Entry Point Microaddresses - After all operand specifiers of an instruction are processed (except for branch offsets), the Decoder then generates the entry address for the execute code.

The only exception to this are the "optimized" instructions, such as a MOVL, which do not have a separate opcode decode cycle since they incorporate the execute code in the last specifier routine.

Figure 3-29 shows the format of the opcode entry address and the source of each address bit. Table 3-22 briefly describes how the bits are derived and used.

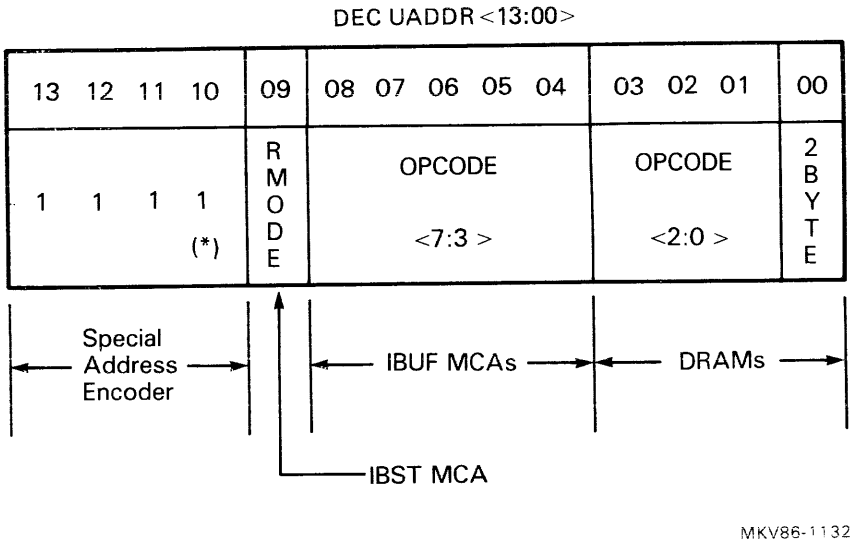


Figure 3-29 Opcode Entry Address Format

Table 3-22 Opcode Entry Address Bit Descriptions

Bit(s)	How Derived
13:10	Forced to state indicated by Special Address Encoder if no special condition is pending. Bit 10 is actually DRAM signal USE OPCO ADDR which is asserted for all opcode entry addresses.
9	SPEC WAS RMODE from IBST MCA. Asserted if last specifier of instruction was register mode.
08:04	Latched copy of OPCODE <7:3> from IB. One byte opcode - opcode itself Two byte opcode - bits take on the two values in order
03:01	DRAM bits <03:01>. Equal opcode <2:0>.
0	DRAM bit <0>. One byte opcode = 0 Two byte opcode = 1

Two Opcode Entry Points Per Instruction

Most instructions have two execute routines: one when the last operand required a memory data transfer, and one when the transfer involved a VAX GPR. The SPEC WAS RMODE bit determines which routine is entered:

SPEC Was RMODE	Execute Routine Entered
Asserted	Register transfer
Negated	Memory transfer

Two opcode entry points exist even for instructions with no operands and for the simple branch instructions; microcode requires there be a valid first microword at every entry point.

Exception: Instructions whose last specifier is ADDRESS access, such as an ADDP4

Reason: RMODE mode with ADDRESS access is a reserved addressing mode which would be detected while decoding the specifier. Therefore, the entry address for the register transfer code could never be generated.

Opcode Entry Address Symbolic Label

The symbolic label assigned to an opcode entry address ends with .MEM for the memory transfer code and .REG for the register transfer code.

Example: Instruction - ADDL2

SPEC Was RMODE	Address	Label
Asserted	3F80	INT.ADDL2.REG
Negated	3D80	INT.ADDL2.MEM

While operand specifier routines all reside in one module (OPSP.MIC), execute routines reside in several modules:

Module Name	Instruction Types
CHARSTR.MIC	Character String and CRC
CONTROL.MIC	PC, loop and subroutine control; CASE, JMP
DECIMAL.MIC	Decimal string
EDITPC.MIC	Edit
FLOAT.MIC	Floating point
INTLOG.MIC	Integer and Logical
LDSV.MIC	Load/Save Process Context
MULDIV.MIC	Integer Multiply and Divide
MXPR.MIC	Move to/from Privileged Register
PCALL.MIC	Procedure Call/Return
QUEUE.MIC	Queue
VIELD.MIC	Variable length bit field

Special Cases For Entering Execute Code

The decoder logic immediately generates an opcode entry point address without a preceding specifier address for the following instruction types:

- Those with no specifiers (BPT, HALT, NOP, etc.)
- Simple branches (BNEQ, BEQL, BRB, BSBB, etc.)
- Illegal one byte opcodes (57, 59, 5A, 5B, 77)
- Reserved escape opcodes (FE and FF)
- First byte of two byte opcode (FD)

Instructions With No Specifiers and Simple Branches

The DRAM immediately issues USE OPCO ADDR, SHIFT IB and OPCODE NEXT.

USE OPCODE ADDR forces the DEC UADDR mux to select the opcode entry address for the instruction.

SHIFT IB and OPCODE NEXT instruct the IB to output the opcode and first specifier of the next instruction. This allows the Decoder to start decoding the next instruction while microcode processes the current one.

Exception: Conditional branch instructions that result in a branch success and unconditional branches cause microcode to invoke a full IB flush. This prevents the Decoder from decoding any IB data until the CBox delivers the new I-stream.

Illegal One Byte Opcodes

USE OPCO ADDR is negated and DRAM bits <4:0> are encoded to a value of 2 to indicate an illegal opcode (see Table 3-19).

The negation of USE OPCO ADDR usually means that the DEC UADDR mux is to select a specifier entry address. For illegal opcodes, this implies that the Decoder will form a separate opcode entry address for every possible specifier combination (a microcode requirement).

All microwords addressed in this manner contain code to immediately transfer control to a microroutine that services reserved opcodes. This routine, which resides in the IANDE.MIC module, starts at symbolic label IE.FLT.RES.OPCD and is entered by the GOTO macroexpression:

Example:

Instruction - Any illegal opcode
Specifier - RMODE, not indexed, access type and data size don't care

Address	Label	Microword contents
38A2	OS.REG.NI.ILL2B.ODD2	GOTO [IE.FLT.RES.OPCD]

The IE.FLT.RES.OPCD routine is similar to the routine used to service reserved addressing mode faults (IE.FLT.ILL.ADR). The main difference is that the exception vector into the SCB is 10 (hex) instead of 1C.

The DRAM <4:0> value of 2 not only leads to the formation of an illegal opcode entry address, but also causes ILLEGAL OPCODE to be asserted. The microbranch logic (UBRS MCAs) of the microsequencer uses the signal to distinguish illegal addressing mode from illegal opcode exceptions.

Reserved Escape Opcodes

These instructions are also illegal one byte opcodes, but are treated in a similar manner as instructions with no specifiers:

1. USE OPCO ADDR - Asserted

   SPEC WAS RMODE determines if the entry address for the .MEM or the .REG code should be generated:

Opcode	SPEC Was RMODE	Address	Label
FE	Negated	3DFC	IE.ESCE.MEM
	Asserted	3FFC	IE.ESCE.REG
FF	Negated	3DFE	IE.ESCF.MEM
	Asserted	3FFE	IE.ESCF.REG

2. SHIFT IB and OPCODE NEXT - Asserted

   IB shifts out opcode and first specifier of next instruction.

Note that the entry points reside in the IANDE.MIC module. In each case, the only function of the microword is to form the exception vector of 10 (hex) and then pass control to the illegal opcode routine (the same one used for illegal one byte opcodes).

Although OPCODE NEXT and SHIFT IB are asserted, the new data output from the IB is not used; the exception handler routine will overwrite the IB.

Two Byte Opcodes

DRAM signals USE OPCO ADDR, SHIFT IB and OPCODE NEXT are all asserted for the first byte (FD) of a two byte opcode. (The byte is treated as if it were a macro NOP instruction.)

As with reserved escape opcodes, there are two entry points for the first byte of a two byte opcode (also in the IANDE.MIC module):

Address	Label	Microword contents
3FFA	IE.ESCD.REG	GOTO DECODER
3DFA	IE.ESCD.MEM	GOTO DECODER

The GOTO DECODER expression (microword bit I_DECODER set) means that the only function of the microword is to request a "Dec Next" cycle for the second, or true, opcode byte of the instruction.

As it forms the address for the FD microword, the Decoder also prepares for the second opcode byte by performing the following:

1. SHIFT IB and OPCODE NEXT instruct the IB to shift out the second opcode byte and the first specifier bytes.

2. PCNC MCA issues OP IS FD to indicate instruction has a two byte opcode

3. IBST MCA saves OP IS FD as the signal TWO BYTE and inhibits its normal incrementing of the SPEC NO <2:0> bits.

The TWO BYTE signal is affixed to the DRAM address pointer for the next, and all subsequent, decode cycles. This enables the DRAM to distinguish the second byte of a two byte opcode from its one byte opcode look alike (eg: the second byte of CVTDH is the same as the opcode byte of CVTWL). Otherwise, the DRAMs would output the wrong data for each specifier.

If the second opcode byte is an illegal opcode, DRAM <4:0> will be equal to 2. This will cause the Decoder to generate the entry address for an illegal opcode microroutine the same way it does for illegal one byte opcodes.

3.4.6.4 Special Microaddresses - Refer to Figure 3-31.

The Special Address Encoder monitors several CPU conditions which may affect normal instruction execution. These conditions may be caused by events that occur external to the current instruction, such as an interrupt, or they may be caused by the instruction itself, such as a Decoder Stall.

When a special condition is detected, the Special Address Encoder:

1. Disables the Decoder from outputting the entry address for the next specifier or opcode routine.

2. Generates the entry point address for the routine required to service the condition

Special Address Generation

Special condition addresses are also output on the DEC UADDR <13:00> lines. Bits <09:00> are forced set, bits <13:10> receive SPECIAL ADDRESS <3:0> which are encoded with the level of the highest priority condition present.

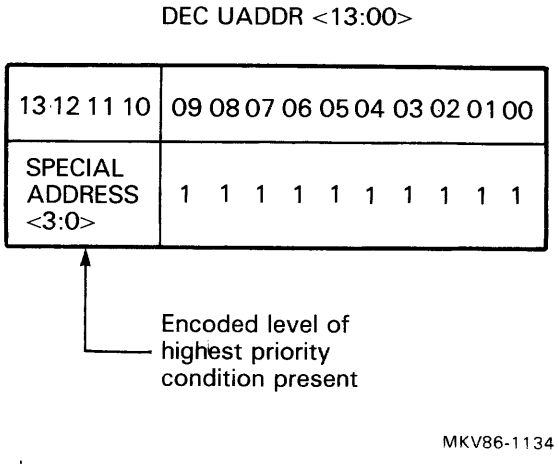


Figure 3-30 Special Microaddress Format

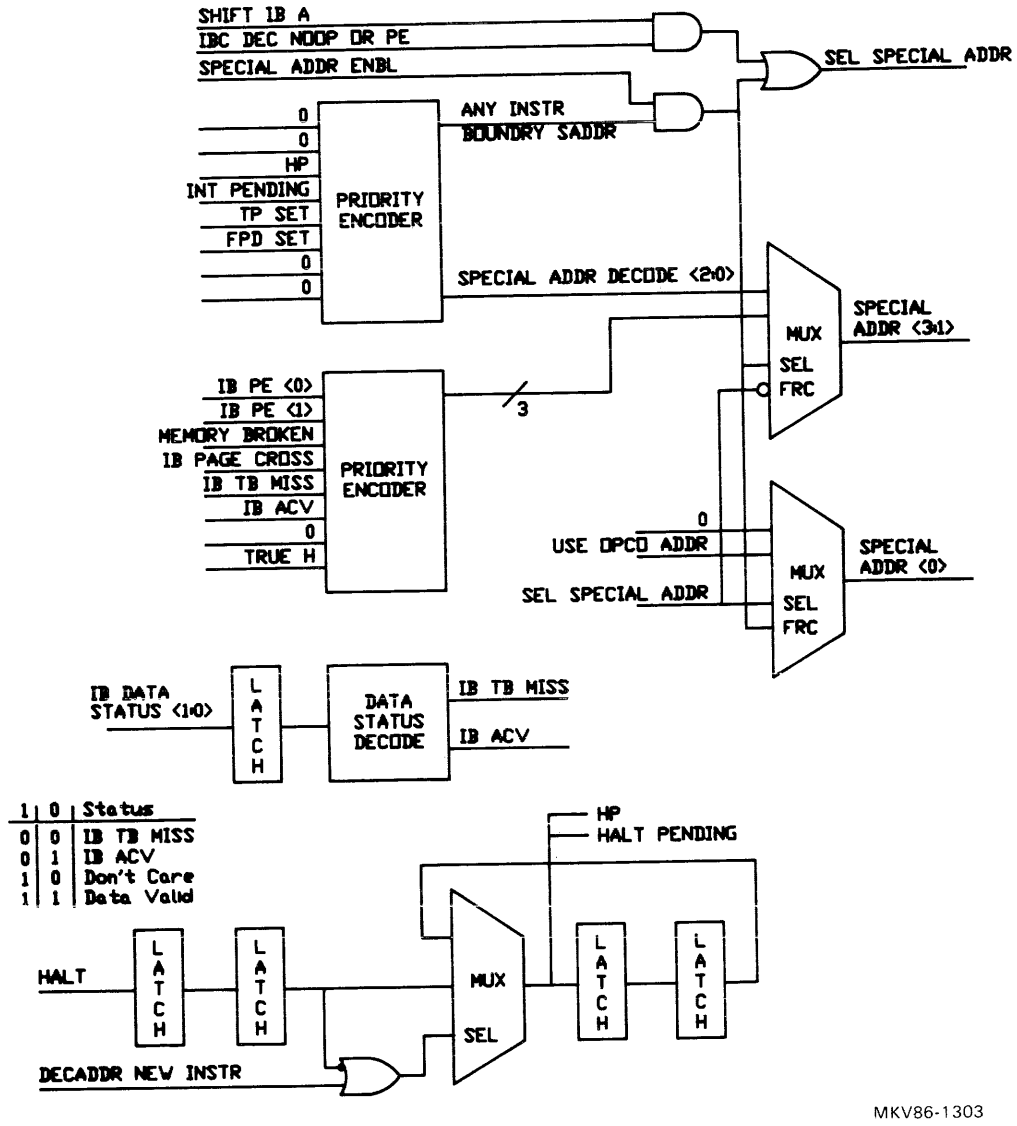


Figure 3-31 Special Address Encoder Logic

Special Condition Event Classes

The Special Address Encoder categorizes special conditions into two event classes and services each class at defined times:

Event Class	When Serviced	Example
External to current instruction	Instruction boundaries	Interrupt
Caused by current instruction	Next IB decode cycle	Decoder Stall

If the next decode cycle coincides with an instruction boundary and a special condition is pending in each event class, the instruction boundary event takes precedence and is selected in the decode cycle.

Table 3-23 lists the special conditions by class and, within each class, by relative priority. It also shows the encoded SPECIAL ADDRESS <3:0> field and the DEC UADDR <13:00> generated for each condition.

Table 3-23 Special Microaddress Conditions

Conditions Serviced at Instruction Boundaries

Condition	Priority	Special ADDR <3:0>	DEC UADDR <13:0>
Not used	Highest	0001	07FF
Not used		0011	0FFF
Halt Pending		0101	17FF
Interrupt Pending		0111	1FFF
Trace Pending		1001	27FF
First Part Done		1011	2FFF
Reserved (Note 1)		1101	37FF
Reserved (Note 1)	Lowest	1111	3FFF

Conditions Serviced on Next IB Decode Cycle

Condition	Priority	Special ADDR <3:0>	Decoder UADDR <13:0>
IB PE <0> (Note 2)	Highest	0000	03FF
IB PE <1> (Note 2)		0010	0BFF
IB Memory Broken		0100	13FF
IB Page Cross		0110	1BFF
IB TB Miss		1000	23FF
IB ACV		1010	2BFF
Not used		1100	33FF
IB Stall (Note 3)	Lowest	1110	3BFF

NOTES

1. Special Address <3:0> cannot equal 1101 or 1111:

   1101 - Conflicts with User control store space
   1111 - Conflicts with Opcode entry point space

2. IB PE <0> = Parity error, lower word of IB longword
   IB PE <1> = Parity error, upper word of IB longword

   If both IB PEs' occur at the same time (double error), IB PE <0> is reported.

3. Although the IB Stall address, 3BFF, is in the address range assigned to specifiers, the address is allowed since no specifier generates an entry address of 3BFF.

Instruction Boundary Special Address Selection

Instruction boundaries are indicated by the signal SPECIAL ADDR ENBL from the IBST MCA. This signal is issued under the same conditions used by the PCNC MCA to determine when a new instruction is to be read from the IB.

Refer to Table 3-18.

The entries OPC and OPC+TPC in the column "New IB Pointer" correspond to the times SPECIAL ADDR ENBL is issued. With few exceptions, even the names of the control signals are the same:

Signal Listed in Table 3-18	Signal Used by IBST MCA
DLY IB FLUSH	IB FLUSH
DEC STALL	IB DEC NOOP (actually the same signal)
R4 NEW OPCODE	R4 NEW IB OPCODE (internal IBST MCA signal)

There is one additional condition imposed on SPECIAL ADDR ENBL: it cannot be issued if TWO BYTE is asserted. This prevents an instruction boundary condition from being signaled twice for a two byte opcode.

IB Decode Special Address Selection

These special conditions are sensed when the signals SHIFT IB from the DRAM and IB DEC NOOP OR PE from the PCNC MCA are asserted. This means that the encoder will only generate a special address if the IB either does not contain enough data for the current decode cycle or the data is flagged with an error of some sort.

The next table describes which special conditions are serviced during IB decode cycles that do not coincide with instruction boundaries.

Table 3-24 Special Conditions Serviced During IB Decode Cycles

Condition	Meaning
IB PE <1:0>	CBox sent a longword on the CACHE DATA BUS with good parity but the DEC module sensed bad parity when the data entered the IB. The parity error is reported in the next decode cycle. IB PEs only indicate that some IB longword received bad parity, not which one. Therefore, the error may or may not be related to the next IB longword read. IB PEs are indications of problems with the CACHE DATA BUS or with the IBUF MCAs. This is because the CBox asserts MEMORY BROKE when it senses bad parity on data read from cache or from the NMI. Since the assertion of this signal prevents data from entering the IB, no IB PE would be reported.
IB Memory Broken	CBox detected an error in the cache/memory subsystem (TB, cache, NMI, etc.) and was unable to prefetch the next longword of the I-stream. A longword is sent to, but not loaded in, the IB (see the entry for IB PE <1:0> above).

Table 3-24 Special Conditions Serviced During IB Decode Cycles (Cont)

Condition	Meaning
IB Page Cross	CBox detected a page boundary crossing while fetching the next longword of the I-stream. The longword is loaded in the IB but is "tagged" with PIBA PAGE CROSS. (PIBA is the Physical Instruction Buffer Address register in the CBox.) A longword tagged with a "page cross" is usually not related to the current I-stream. The microcode will check if the page cross is legal and, if so, instruct the CBox to form a new PIBA and continue prefetching the I-stream. The tagged IB longword location is overwritten with the first longword of the new page.
IB TB Miss	Last I-stream prefetch request resulted in a TB miss. TB misses are common, usually non-fatal page faults. The microcode will enter a memory management routine to fetch the appropriate PTE and service the miss.
IB ACV	The last I-stream prefetch request resulted in a TB Access Control Violation (ACV). ACVs may or may not be fatal. Microcode determines the fault severity and takes appropriate action.
IB Stall	Not enough IB data for current decode cycle. Special address encoder generates the address for an OS.IB.STALL microword whose only function is to again issue an IB decode request. The special address encoder will continually generate the address for the OS.IB.STALL microword until the CBox delivers the required data.

NOTE

IB PE <1:0> and MEMORY BROKE are indications of hardware problems and are reported to the IBox Error Register (IBER). The IBER is saved along with the EBox and CBox error registers (EBER and CBER) and other relevant data (PSL, PC, etc.) in a microcode data structure known as the Machine Check Error Bank. Refer to the VAX 8800 Machine Check Interpretation Guide (EK-KA88H-UG).

3.4.6.5 IBST MCA Signals Related To Instruction Decoding - The following text describes those IBST MCA signals which were either not covered in preceding sections or need additional explanation.

SPEC NO <2:0>

Function:

- Form the upper three read address inputs to the Decoder RAMs
- Indicate which specifier of instruction is to be processed next.

Operation:

SPEC NO <2:0> are cleared at the start of every instruction. They are then incremented by one when DEC SELECT and SHIFT IB are asserted and IB DEC NOOP is negated. This means that the bits will be equal to 0 in the first specifier decode cycle, 1 in second cycle, and so forth.

Exception: The bits are not incremented in subsequent cycles of indexed or big immediate specifiers or in the first cycle of a two byte opcode.

When the opcode cycle of an instruction is to be performed, SPEC NO <2:0> usually reflect the number of operands in the instruction.

Since the bits are cleared at the start of an instruction, this means that they will point to a DRAM location for a specifier that does not exist. In this case, the selected DRAM location is not encoded with specifier data, but with data necessary to generate and select the entry point address for the opcode routine.

For example, the SPEC NO <2:0> value will be equal to 2 in the opcode cycle of an ADDL2 instruction.

A value of 2 for an instruction with three or more operands would mean that the DRAM word selected is the one containing data for the third operand. However, since ADDL2 only has two operands, the DRAM word selected will contain the data required by the opcode cycle.

Exception: The last specifier of a loop control instruction (ACBx, AOBLEQ, etc.) is always the branch offset. Since this specifier does not have its own decode cycle, the SPEC NO <2:0> value for the specifier is the one used in the opcode cycle.

SPEC TYPE <3:0>

Function:

- Encoded to represent the current specifier type (literal, register mode, etc.).
- Become DEC UADDR <8:4> of an operand entry point address (see Figure 3-28 and Table 3-19.

Operation:

The SPEC TYPE <3:0> bits are normally based on the current specifier byte, SP <7:0>, from the IB.

The exception to this is during the additional cycles of big immediate specifiers in which SP <7:0> does not represent a specifier but is the first byte of a subsequent longword (Table 3-14).

In this case, the IBST MCA must remember that a big immediate is being processed and the data size of the operand. Otherwise, it could easily mistake the first byte of a subsequent longword as a new specifier.

When it first decodes the SP <7:0> byte and determines that the current specifier is immediate mode, the IBST MCA:

1. Encodes the SPEC TYPE <3:0> bits with a value of C (hex)

   This is always the case in the first decode cycle of immediate mode specifiers (Table 3-19).

2. Gates the SPEC TYPE value just generated with DRAM bits IBST DATA SIZE <2:0>

IBST DATA SIZE <2:0> represent the data size of the current operand and are sent to the IBST MCA the same time the entry point address for the first longword is being formed.

Although the IBST DATA SIZE <2:0> bits arrive to late to affect the SPEC TYPE value in the first cycle, the IBST MCA uses them to generate the value for the next, and each subsequent, cycle.

When IBST DATA SIZE <2:0> equal 4, 6 or 7, indicating the operand is bigger than a longword (Table 3-20), the IBST MCA will:

3. Issue SEQ LW to the PCNC MCA

4. Inhibit incrementing SPEC NO <2:0>

5. Encode SPEC TYPE <3:0> with the value appropriate for the next decode cycle (Table 3-20)

The IBST MCA performs steps 3 to 5 once for D-float, G-float and for Quadword big immediates and three times for H-float and Octaword. In either case, normal specifier decoding resumes after the last longword is processed.

DLY OPN XOR IBS

This signal is the logical "XOR" of the DRAM signals OPCODE NEXT and SHIFT IB.

DLY OPN XOR IBS is sent along with the other DRAM outputs to discrete parity checking logic on the DEC module. If bad parity is sensed in the DRAM, the parity check logic will issue the signal DRAM PE. This signal is latched in the IBox Error Register (IBER) and is sent to the microtrap logic on the SEQ module.

3.4.6.6 Decoder RAM (DRAM) - The 4K x 17 bit DRAM serves basically as a look-up table of opcode and operand specifier entry point microaddresses. The DRAM is loaded by the VAX Console during the system initialization sequence.

There is one DRAM word for every specifier of an instruction and one for the opcode. In addition, a DRAM word is allocated to every illegal one byte and two byte opcode.

Exceptions - There is no corresponding DRAM word for the:

- Opcode of an optimized instruction
- Branch specifier of a branch class instruction

DRAM Read Address

The DRAM is indexed by an 11 bit read address which is comprised of three fields. The following figure shows the address format and the source of each field.

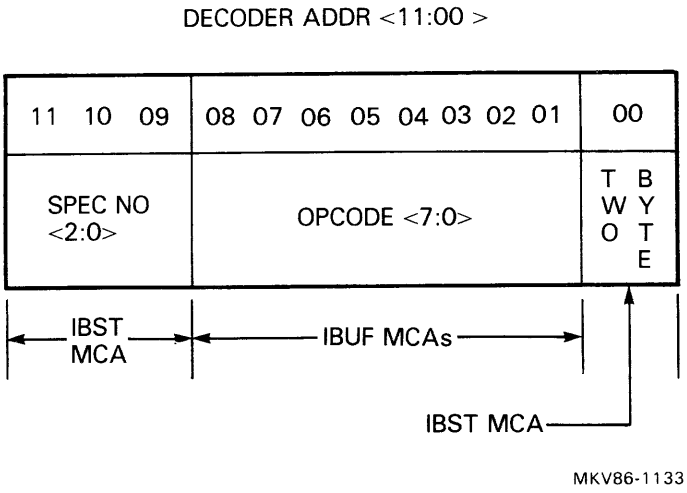


Figure 3-32 Decoder RAM Read Address Format

There are six copies of the DRAM read address: DECODER ADDR A <11:00> to DECODER ADDR F <11:00>. This reduces loading of the read address lines and speeds the selection of certain DRAM bits which are required early in the decode cycle.

Copies A, B, and D each feed a single DRAM chip. Copies E and F each feed seven chips.

Copy C does not address the DRAM. Instead, bits <8:4>, which are equal to OPCODE <7:3>, are sent directly to the DEC UADDR mux. Bits <8:1> are sent to the "visibility bus" as VBUS OPCODE <7:0>.

DRAM Output Signals

Figure 3-33 shows the 17 DRAM output bits and the signal name assigned to each bit. Table 3-25 briefly describes each signal.

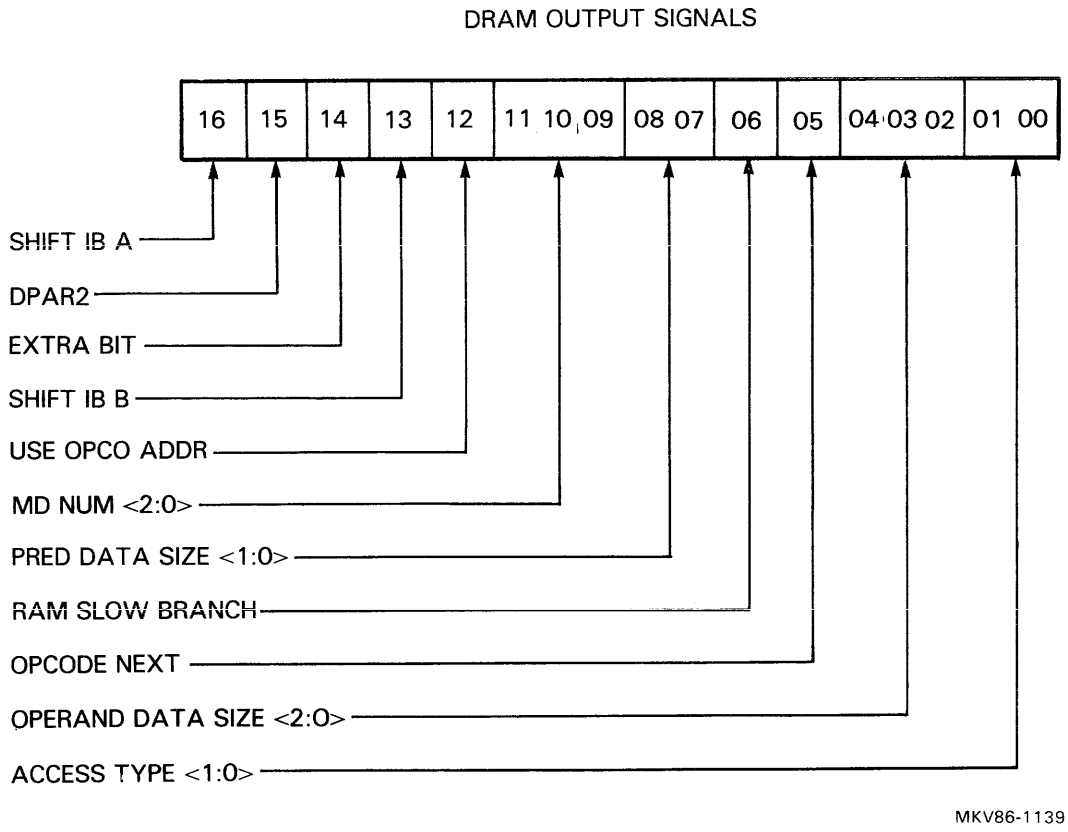


Figure 3-33 Decoder RAM Output Signals

Table 3-25 Decoder RAM Output Signal Descriptions

DRAM Signal	Description
SHIFT IB A	Gated with IB STALL OR PE (originally IB DEC NOOP OR PE from PCNC MCA). Forces Special Address Encoder to check for special conditions when IB is shifted.
DPAR2	DRAM parity bit (odd parity) When a DRAM word has bad parity, the signal DRAM PE is asserted. This signal is recorded in bit 5 of the IBox Error Register (IBER) and is reported to the microtrap logic as machine check condition bit IBox MC COND <0>.
EXTRA BIT	Allows IBFO MCA to distinguish between byte and word size offsets and between different data types of 32, 64 and 128 bit operands.
SHIFT IB B	Signals PCNC MCA and IBST MCA when next block of I-stream data is to be shifted out of IB.
USE OPCO ADDR	Forces DEC UADDR <13:00> mux to select opcode entry address for current instruction. Also see entry for OPCODE NEXT.

Table 3-25 Decoder RAM Output Signal Descriptions (Cont)

DRAM Signal	Description
MD NUM <2:0>	Specifies which EBox Memory Data Register (MDR) is to receive data for specifiers that request data from memory or from a VAX GPR. A maximum of 6 MDRs can be assigned to an instruction, one per specifier. Encoding MD NUMs in the DRAM allows specifier microroutines to refer to MDRs implicitly as MD[MDNUM], instead of explicitly as MD[MD0].
PRED DATA SIZE <1:0>	Specifies data size for branch offsets and immediate mode specifiers that occupy second through sixth specifier positions. Bits are output in current decode cycle but indicate data size of next operand. Bits are latched in IBST MCA and presented to PCNC MCA in T3 of current specifier cycle. Allows PCNC MCA to generate appropriate PC increment amount for next specifier in time for next decode cycle.
RAM SLOW BRANCH	Asserted if next specifier will be the branch offset of a PC loop control instruction (eg: ACBB, AOBLEQ,... etc.)

Table 3-25 Decoder RAM Output Signal Descriptions (Cont)

DRAM Signal	Description
OPCODE NEXT	Asserted if the next decode cycle will be the first one of a new instruction. The OPCODE NEXT and USE OPCO ADDR signals are both asserted in the opcode decode cycle of an instruction but are used during different canonical times: • USE OPCO ADDR Used by DEC UADDR muxes during T2 so that the opcode entry address will be available to the micro-PC address latches by T3. • OPCODE NEXT Latched in PCNC and IBST MCAs during T3. PCNC MCA uses the signal to establish the IB pointer for the next decode cycle. IBST MCA uses the signal to generate NEW OPCODE which forces IB to output opcode and first specifier of next instruction. Asserting both signals simultaneously allows the Decoder to decode the next instruction while microcode processes the execute code of the current one.

Table 3-25 Decoder RAM Output Signal Descriptions (Cont)

DRAM Signal	Description
OPERAND DATA SIZE <2:0>	These bits have dual functionality: ● Specifier cycle Represent current operand data size ● Opcode cycle Bits <1:0> = opcode <1:0> Bit <2> = 0
ACCESS TYPE <1:0>	These bits also have dual functionality: ● Specifier cycle Represent current operand access mode ● Opcode cycle Bit <1> = opcode bit <0> Bit <0> = 0 if one byte opcode 1 if two byte opcode

3.4.7 Optimized Instructions

Optimized instructions incorporate the execute code and last specifier routine into a single microroutine to yield a one cycle saving in instruction execution time.

Optimized instructions fall into two general classes:

1. Simple moves - MOVAx, MOVL, MOVZBL and MOVZWL
2. Simple branches - BEQ, BNEQ, BRB, BRW, etc.

3.4.7.1 Simple Move Instructions - Simple move instructions do not require execute code since no operation is performed on the destination operand, which is of the WRITE LONG AND SET CC's type (see Table 3-20. When the last specifier routine issues the I_DECODER bit, the Decoder will output the entry address for the first specifier of the next instruction instead of the opcode entry address for the current one.

Although the simple move instructions exclude the opcode cycle, there are some specifier addressing modes which defeat the one cycle saving in execution time.

Examples:

Instruction	Reason for Lost Optimization
MOVL (R0), R1	Cache latency fetching first operand forces "optimized" code to wait an extra cycle.
MOVL R0, 04(R1)	Not enough data ports in the EBox to perform the write operation in one cycle.

NOTE

The MOVZBL and MOVZWL instructions both zero extend the source operand in the routine that handles the operand. Except as noted above, this is done without a performance penalty.

3.4.7.2 Simple Branch Instructions - These instructions only take one microword to execute since there is no decode cycle for the branch specifier. Instead, the Decoder will immediately output the entry address for the execute code.

The execute code is responsible for sign extending the branch offset and sending it to the EBox on the IB Data Bus. The execute code also encodes a microbranch condition recipe for the microsequencer

NOTE

> Loop control instructions (ACBx, AOBLEQ, etc.) also exclude the branch specifier cycle. However, they are not considered optimized since they require additional clock cycles to calculate the new index, generate the condition code bits and test the branch condition.

3.5 MACROBRANCH INSTRUCTIONS

Branch instructions are part of a larger group of instructions that potentially modify the VAX PC. These "PC Control" instructions also include subroutine control (BSBx, JSB, RSB), case (CASEx) and the procedure call/return (CALLx, RET) instructions.

The intent of this section to present the basic hardware/ microcode concepts involved with servicing PC control instructions by using the branch instructions as examples. Refer to the microcode listings and the VAX Nautilus 8800 Microcode Interpretation Guide (EK-KA88E-UG) for more information.

3.5.1 Branch Instruction Basics

When the VAX PC is modified by a branch offset, the resultant change to the I-stream flow usually means that the IB will not contain the next instruction to be executed. Since there is no way to predict this in advance, all microroutines that service branch instructions contain code to conditionally, or unconditionally:

1. Initialize (flush) the IB
2. Sum the updated PC with the branch offset
3. Generate the new physical PC
4. Initiate a Cache prefetch operation

On a successful branch, the above operations are carried out and the new instruction is fetched from Cache and delivered to the IB. The decode logic will start processing the new instruction when the last microword of the branch routine requests a "Dec Next" cycle (I_DECODER microword bit set).

On an unsuccessful branch, the above operations are inhibited and the next instruction in the IB is executed.

3.5.2 Branch Instruction Classes

There are, in general, three branch instruction classes:

1. Unconditional branches - BRB, BRW, JMP, etc.

2. Short conditional branches - BEQL, BNEQ, etc.

3. Long conditional branches - ACBB, AOBLEQ, BBC, etc.

3.5.3 Unconditional Branches

An unconditional branch only requires a single microword to execute. However, due to the time required to generate a new physical PC and to fetch new I-stream from Cache, there is always a delay before the next instruction is delivered to the IB.

During the delay incurred fetching new I-stream data from Cache, the pipeline must be "padded" with a series of "Noop" microwords. This prevents the microsequencer from selecting the Decoder as the source of the next microaddress until the new instruction enters the IB. Otherwise, the wrong instruction would be executed next.

Figure 3-34 shows the pipeline state during the execution of a BRB instruction. (The reason for showing six microwords will be addressed in the text).

Table 3-26 lists the symbolic labels, micro-orders and operations performed by the BRB execute code.

NOTE

All branch instructions have two execute code entry points: one if the last specifier of the previous instruction was register mode, and one if it was memory mode. This example assumes that the last specifier was memory mode.

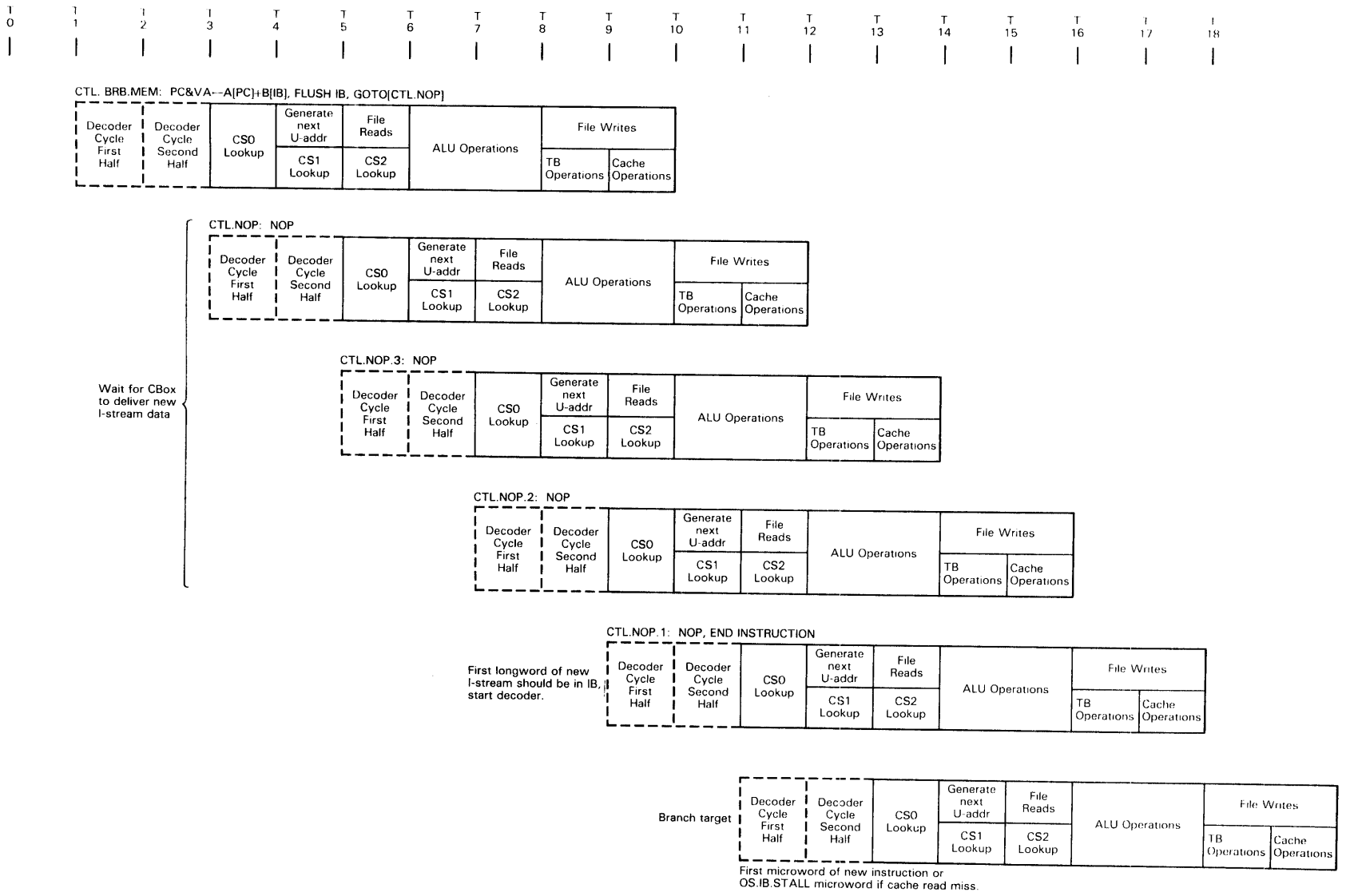


Figure 3-34 Pipeline State for a BRB Instruction

Table 3-26 Execute Code For A BRB Instruction

Microword	Micro-order	Function
CTL.BRB.MEM	PC & VA <- A[PC] + B[IB]	Add sign extended branch offset (IB DATA BUS) to updated PC, store sum in PC and VA registers. Translate virtual PC to physical PC and prefetch new I-stream data.
	FLUSH IB	Unconditional IB Flush
	GOTO [CTL.NOP]	Enter No-op routine
CTL.NOP	NOP	Pad pipeline
CTL.NOP.3	NOP	Pad pipeline
CTL.NOP.2	NOP	Pad pipeline
CTL.NOP.1	NOP, END INSTRUCTION	Issue "Decoder Next".

The first microword of the BRB code performs the actual work of the instruction. The next three microwords "pad" the pipeline, allowing time for the CBox to fetch and deliver the first longword of the new I-stream to the IB.

The macro expression END INSTRUCTION in the fifth microword indicates that microword bit I_DECODER is set to force the micro-PC address muxes to select the Decoder as the source of the next microaddress.

Cache/Decoder Timing

The timing relationship between the arrival of the first longword in the IB and the assertion of the I_DECODER bit determines whether the sixth microword to be executed will be the first microword of the new instruction or the "Decoder Stall" micro-word, OS.IB.STALL.

Table 3-27 lists the events that occur during each canonical time of the CTL.BRB.MEM microword starting with T8.

Table 3-27 Microword CTL.BRB.MEM Event Timing

Time	Event
T8	EBox outputs new virtual PC to CBox Translation Buffer (TB).
T8 - T9	TB generates new physical PC.
T9 - T10	Cache read operation requested.
T10	Cache outputs first longword of new I-stream to IB.
T11	First longword stored in IB.
T12	Decoder generates entry address for first specifier of new instruction.
T13	Entry address latched in micro-PC address latches.

Canonical T11, T12, and T13 relative to CTL.BRB.MEM correspond with T3, T4, and T5 relative to CTL.NOP.1. It is during these canonical times that the I_DECODER bit is:

1. Read from the CS0 RAM

2. Latched in the CS0 RAM data latches

3. Presented to the micro-PC address muxes

If the CBox can deliver the first longword by T11, the Decoder will be able to generate the entry address for the first specifier of the new instruction in T12 and have it for ready for the micro-PC address muxes by T13.

If the CBox cannot deliver the first longword on time, the Special Address Encoder will disable the Decoder and output the address of the OS.IB.STALL microword (the Decoder always generates an address, even if it is based on wrong IB data).

The Special Address Encoder will continually output the address of the OS.IB.STALL microword until the CBox delivers the first longword to the IB. Once this happens, the Decoder will resume control and supply the entry address for the first specifier of the new instruction.

3.5.4 Short Conditional Branches

Short conditional branches (BEQL, BNEQ, etc.) are to only perform the branch function if a certain condition is true. The execute codes for these instructions are similar in structure in that they all test the PSL condition code bits to determine if the branch should be taken. The only difference is the PSL condition code bit(s) under test. For example, a BEQL instruction will branch if the PSL <Z> bit is set while a BNEQ will branch if the bit is clear.

Branch Recipes

The execute microwords of short conditional branches specify the PSL condition code bit(s) under test by encoding a "branch recipe" in the I_MISC field. The branch recipe is of the form:

I_MISC <6:0> = 010xxxx

Table 3-28 lists the I_MISC field settings and PSL condition code bit(s) under test for all macrobranch instructions.

Table 3-28 I_MISC Field Settings For Macrobranch Recipes

I_MISC	Instruction	Take Branch If
20	BGRT	PSL <N OR Z> = 0
21	BLEQ, SOBGTR	PSL <N OR Z> = 1
22	BGEQ	PSL <N> = 0
23	BLSS, SOBGEQ	PSL <N> = 1
24	BNEQ	PSL <Z> = 0
25	BEQL	PSL <Z> = 1
26	BVC	PSL <V> = 0
27	BVS	PSL <V> = 1
28	BGTRU	PSL <C OR Z> = 0
29	BLEQU	PSL <C OR Z> = 1
2A	BCC	PSL <C> = 0
2B	BCS	PSL <C> = 1
2C	AOBLEQ, ACBx	WBUS <N> XOR ALU <V> = 1
2D	AOBLSS	((WBUS <N> XOR ALU <V>) OR WBUS <Z>) = 1
2E	BBx, BBxx	WBUS <Z> = 1
2F	BRX, BSxx	UNCONDITIONAL

Note:

The I_MISC field is monitored by the CCBR MCA. Refer to Figure 3-37.

Short Conditional Branch Execute Code

Short conditional branches, like the unconditional branches, are also executed by a single microword. The execute code for a BEQL is shown below. Except for the PSL condition code bit(s) under test, the codes for other short conditional branches are similar in structure.

Microword	Micro-Orders
CTL.BEQL.REG or CTL.BEQL.MEM	COND.PC & VA <- A[PC] + B[IB], LOAD PC & FLUSH IB IF[PSL<Z>.EQ.1], END INSTRUCTION

The first micro-order stipulates to conditionally load the PC and VA registers in the EBox with the sum of the up-dated PC and the sign extended branch displacement. If the branch is successful, the new PC is then to be routed from the EBox, over the VA BUS, and loaded into the VA latch in the CBox.

The second micro-order specifies to load the new PC and flush the IB only if the PSL <Z> bit is set; otherwise, do nothing.

The third micro-order indicates that the I_DECODER microword bit is set to end the instruction and request a decoder next cycle.

Conditionally loading the PC and VA registers is effectively a guess that the branch will succeed. If the branch is successful, the CBox will load the new PC in the VA latch, form the physical PC, and fetch the new I-stream data. If the branch fails, the IBox will issue the signal COND BR FAIL (see Figure 3-37) to inhibit the CBox from loading the new PC in the VA latch.

Pipeline Timing Consideration

Note that the execute code for a conditional branch does not call the CTL.NOP routine to "pad" the pipeline as does the execute code for an unconditional branch. Instead, it immediately asserts the I_DECODER bit to request another decode cycle.

Since the Decoder is not under microcode control, there is no way to stop it from decoding the I-stream data in the IB. This means that when the I_DECODER bit is set, the Decoder will supply what it thinks is the correct entry address for the next specifier (or opcode) to be processed to the micro-PC address latches.

If the branch fails, there is no change to the I-stream flow and the IB should contain the next instruction to be executed (assuming Cache hits on the I-stream). The Decoder will therefore generate the next address based on the correct I-stream data already in the IB. In this case, the branch instruction effectively becomes a NOP.

On a branch success, however, there is a change to the I-stream flow and the IB will not contain the next instruction to be executed. The Decoder generated address, which is always based on the IB's current contents, will therefore be based on the wrong next instruction.

The timing relationship between the I_DECODER bit and the I_MISC field of a microword is the key factor to what happens when a conditional branch results in a branch success.

Refer to microword CTL.BEQL.REG shown at the top of Figure 3-35.

The I_DECODER bit is available at T5 time of CTL.BEQL.REG while the I_MISC field is not available until T8 time. This means that before the I_MISC field can determine whether a branch should be taken, the Decoder will have already:

1. Decoded the next block of I-stream data currently in the IB

2. Generated the address of microword "U"

3. Sent the address to the micro-PC address muxes

Also, by the time I_MISC is actually used, microword "U" will have generated the address for microword "V" which will have generated the address for microword "W".

Therefore, before the CTL.BEQL.REG microword progresses to the point where it can test the branch condition, three erroneous microwords will already be in various stages of execution.

Conditional Branch Microtrap

Since microwords "U", "V" and "W" are already started by the time the branch condition is tested, the hardware must inhibit the microwords from performing their normal write operations on a branch success. If this is not done, the register or memory location written may contain the wrong data for the next instruction to be executed.

The IBox hardware inhibits the writes of microwords "U", "V" and "W" by generating the microtrap condition "COND BR SUCCESS". As in the case with all microtraps, this causes the microtrap logic to issue the GLOBAL UTRAP and BLOCK WRITES signals. The BLOCK WRITES signals inhibit the writes to the appropriate logic elements.

The COND BR SUCCESS microtrap condition is handled by a single word routine at the microtrap vector of 0200 (hex). This vector is the address of the CTL.TRAP.COND.BR microword shown in Figure 3-35.

The only function of the CTL.TRAP.COND.BR microword is to release the trap silos and to request another "Decoder Next" cycle. The code for the CTL.TRAP.COND.BR microword is given below.

Microword	Micro-Orders
CTL.TRAP.COND.BR	CLEAR TRAP, END INSTRUCTION

Note that since the CTL.BEQL.REG microword loaded the PC and flushed the IB and that the three microwords in the "shadow" of the trap are not to be restarted, the CTL.TRAP.COND.BR microword need not perform these functions as would other microtrap service routines.

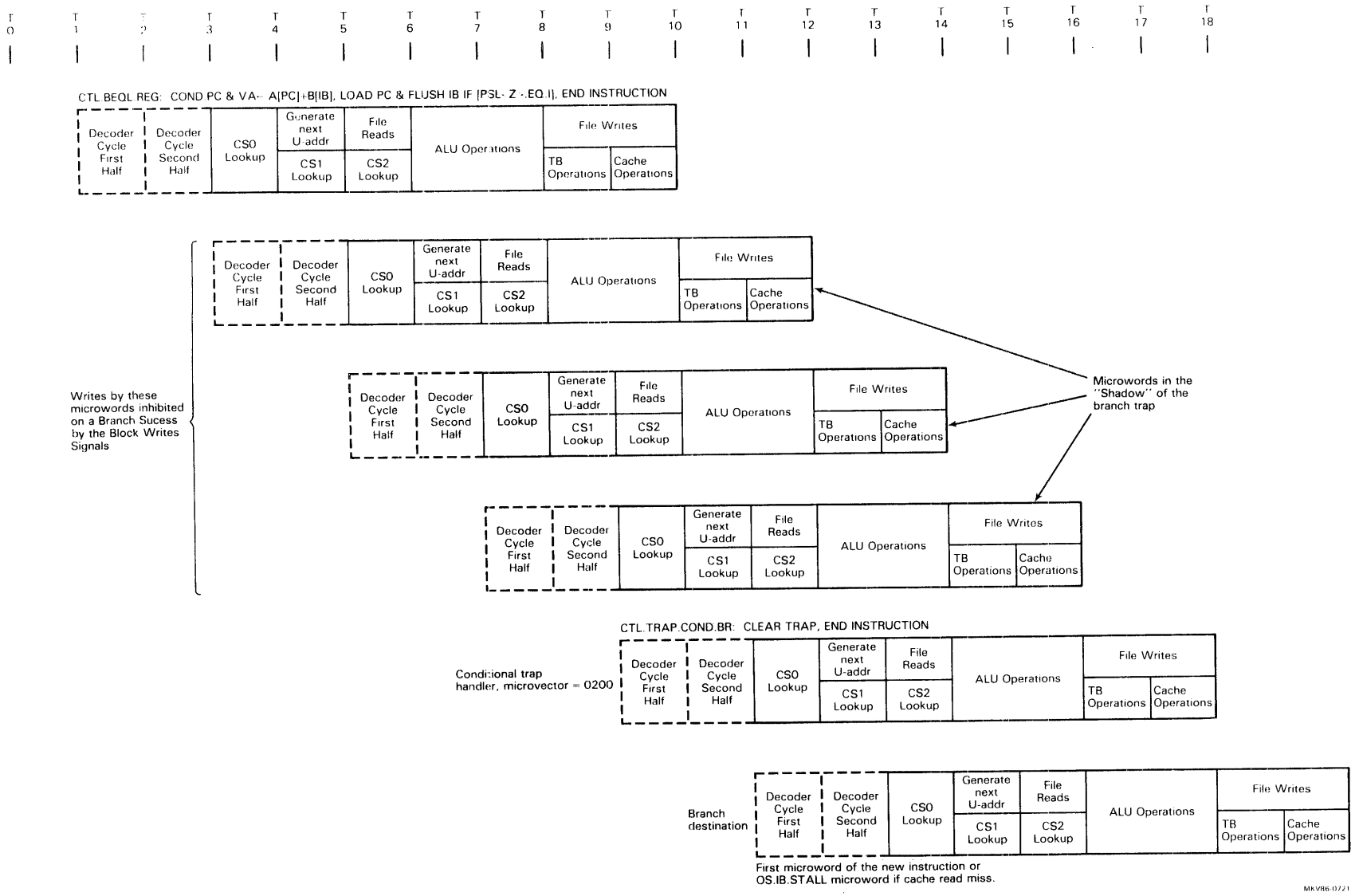


Figure 3-35 Pipeline State for a Successful BEQL Instruction

3.5.5 Long Conditional Branches

Long conditional branches include the following instruction types:

- Loop control - ACBx, AOBxxx, SOBxxx
- Branch on bit - BBx, BBxx, BBxxx
- Branch on low bit - BLBC, BLBS
- Multi-way branching - CASEx

The execute codes for long conditional branches vary in size from a few to several microwords. For example, the code for a BLBC is only three microwords long while the BBCCI code is more than ten.

Optimized Code

Some long conditionals are executed using "optimized" code to enhance instruction execution speed on a branch success. The first microword of these instructions save the current PC in a microcode temporary register and unconditionally load the new PC and flush the IB. This "guess" that the branch will succeed reduces the number of cycles that would otherwise be wasted waiting for Cache to deliver new I-stream data on a branch success. If the guess proves wrong, the PC will be reloaded with its saved value later on by a microword that includes the "LOAD PC & FLUSH IB IF[]" conditional branch expression.

The first microword of a long conditional that is not optimized will either save the new PC or the branch offset in a microcode temporary register and then transfer control to a routine that determines if the branch should be taken. In this case, the last microword of the code must include the "LOAD PC & FLUSH IB IF[]" expression.

PSL Condition Code Recipes

Some long conditionals, such as the loop control instructions, perform arithmetic operations to modify the PSL CC bits and then determine if the branch should be taken based on the new bit settings. Other long conditionals need only examine the current bit settings to determine branch success/fail.

The setting/clearing of the PSL CC bits is controlled by the I_MISC field of a microword. Instructions that modify the CC bits include a microword that contains the expression "SETCC []" which indicates the PSL CC "recipe" encoded in the I_MISC field.

Table 3-29 lists the various SETCC [] expressions, the encoded I_MISC field value and the new PSL CC bit settings.

Sample Execute Code

Refer to Figure 3-36 and Table 3-30.

The AOBLEQ instruction is typical of an long conditional branch whose execute code:

1. Saves the current PC
2. Unconditionally loads the new PC and flushes the IB.
3. Performs an arithmetic operation to modify the PSL CC bits.
4. Reloads the saved PC if the branch should not have been taken.

The first microword of the AOBLEQ execute code saves the current PC in an EBox Memory Data Register (MDR 6), performs the unconditional load PC/flush IB and requests a Cache read for new I-stream data at the predicted branch target address.

The second microword increments the index operand by one and modifies the PSL condition code bits according to the condition code "recipe" specified by the expression SETCC [OP6] (see Table 3-29).

The third microword subtracts the new index value from the limit and returns the WBUS and ALU condition codes that result to the IBox.

The forth microword determines if the guess that the branch will be successful was correct with the LOAD PC & FLUSH IB IF[] expression. It also checks whether an integer overflow trap is to be taken when an integer overflow occurs and the PSL IV bit is set. (In the case of an AOBLEQ instruction, an integer overflow will occur if the index was the largest positive integer before being incremented).

If the initial guess that the branch will be successful was correct and no integer overflow trap is to be taken, the AOBLEQ code will wait one more cycle for Cache to deliver the new I-stream by executing the CTL.NOP.1 microword. Otherwise, the IBox will force the microcode to enter either the conditional branch or the integer overflow trap handler routine.

Table 3-29 I_MISC Field Settings For PSL Condition Code Recipes

	New N	New Z	New V	New C	Notes
OP1=1F FROM.WBUS=1F	WBUS<3>	WBUS<2>	WBUS<1>	WBUS<0>	Set CC bits from WBUS
OP2=1	*WBUS<N>	*WBUS<Z>	*ALU<V>	*ALU<C>	C <- carry out
OP3=4	*WBUS<N>	*WBUS<Z>	0	C	
OP4=3	ALU<V> XOR *WBUS<N>	*WBUS<Z>	0	not (*ALU<C>)	C <- borrow in
OP5=5	*WBUS<N>	*WBUS<Z>	0	0	
OP6=0	*WBUS<N>	*WBUS<Z>	*ALU<V>	C	
OP7=1C	*WBUS<N>	*WBUS<Z> AND Z	0	C	
OP8=14	*WBUS<N>	*WBUS<Z>	0	C	
OP9=8	*WBUS<N>	Z	0	0	
OP10=15	*WBUS<N>	*WBUS<Z>	V	0	
OP11=0D	N	Z	*ALU<V>	0	
OP12=1D	N	*WBUS<Z> AND Z	0	0	
OP13=18 SET.N=18	1	Z	V	C	Set N
OP14=10 CLR.N=10	0	Z	V	C	Clear N
OP15=1A SET.V=1A	N	Z	1	C	Set V
OP16=1E	N	*WBUS<Z> AND Z	V	C	

Table 3-29 IMISC Field Settings For PSL Condition Code Recipes (Cont)

	New N	New Z n	New V	New C	Notes
OP17=0E	N	Z	NOT (*WBUS<Z>) OR V	C	
OP18=0F	N	Z	NOT (*WBUS<Z>)	C	
OP19=0B	N	*WBUS<Z>	0	0	
OP20=16	*WBUS<N>	*WBUS<Z>	0	0	
OP21=6	*WBUS<N>	*WBUS<Z>	V	0	
OP22=7	*WBUS<N>	*WBUS<Z>	1	0	
OP23=2	*WBUS<N>	*WBUS<Z>	*ALU<V>	not (*ALU<C>)	C <- borrow in
OP24=9	*WBUS<N>	0	0	0	
OP25=11 CLR.Z=11	N	0	V	C	Clear Z
OP26=1B SET.C=1B	N	Z	V	1	Set C
OP27=13 CLR.C=13	N	Z	V	0	Clear C
OP28=12 CLR.V=12	N	Z	0	C	Clear V
OP29=0A	0	*WBUS<Z>	0	0	
OP30=19 SET.Z=19	N	1	V	C	Set Z
OP31=17					Open recipe
OP32=0C	0	*WBUS<Z>	0	C	

"*" - State of bit is size dependent. Data size given by I_SIZE field of microword.

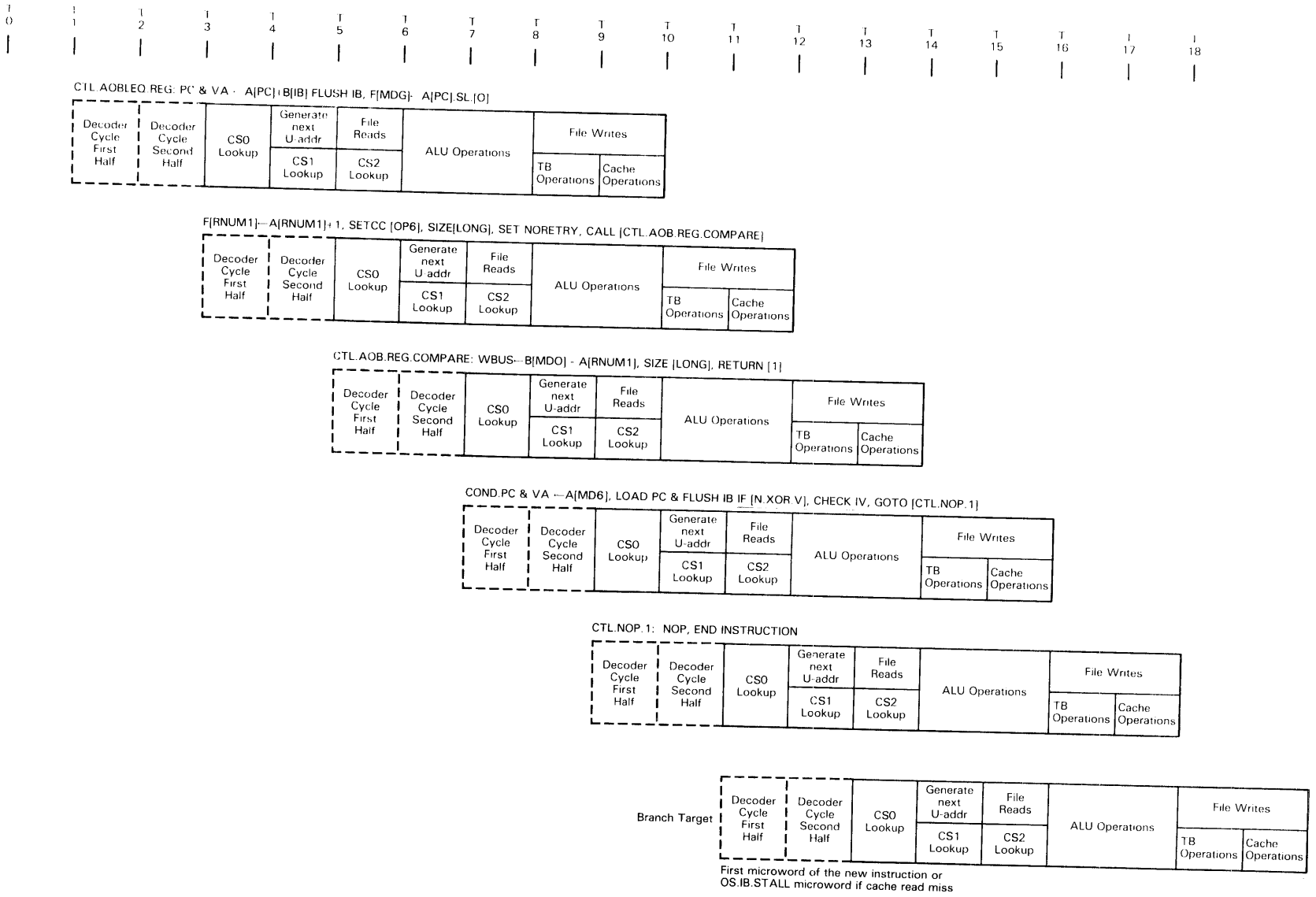


Figure 3-36 Pipeline State for a Successful AOBLEQ Instruction

Table 3-30 Execute Code For A AOBLEQ Instruction

```
CTL.AOBLEQ.REG:
  ;---------------------------------;
  PC & VA <- A[PC] + B[IB] FLUSH IB, ; Uncond load PC/flush IB.
  F[MD6] <- A[PC].SL.[0]             ; Save current PC.

=0
  ;0--------------------------------;
  F[RNUM1] <- A[RNUM1] + 1,          ; Increment index.
  SETCC [OP6], SIZE [LONG],          ; Set PSL CC bits.
  SET NORETRY,                       ;
  CALL [CTL.AOB.REG.COMPARE]         ;

CTL.AOB.REG.COMPARE:
  ;---------------------------------;
  WBUS <- B[MD0] - A[RNUM1],         ; Compare limit - index
  SIZE [LONG],                       ;
  RETURN [1]                         ; Return to caller

  ;1--------------------------------; Return point.
  COND.PC & VA <- A[MD6],            ; Reload saved PC if branch
                                     ; should not have been taken.
  LOAD PC & FLUSH IB IF[N.XOR.V],    ; Reload PC if incremented
                                     ; index greater than limit.
  CHECK IV,                          ; Check for integer overflow.
  GOTO [CTL.NOP.1]                   ; On branch success, need one
                                     ; more cycle due to IB flush.
```

Note: The code given here is in the order in which the microwords are executed, not the order in which the microwords appear in the listings.

3.5.6 Condition Code And Macro Branch Logic

The CCBR MCA supports macroinstruction execution by maintaining the hardware images of the PSL CC bits. It also maintains 7 state flags which provide firmware writers with one of the methods available for controlling microcode flow.

Refer to Figure 3-37.

Sized Branch Logic

The Sized Branch logic generates a set of size dependent microbranch conditions based on intermediate, or "raw", condition codes from the EBox and on the I_SIZE field of the current microword.

The "raw" condition codes from by the EBox can represent byte, word, or longword size operations. The I_SIZE field of the current microword indicates the data size:

I_SIZE	Data Size
0	Not Used
1	Byte
2	Word
3	Longword

The sized microbranch conditions are fed to the PSL Condition Code logic, where they may be stored as the new PSL CC bits, and to the VAX Branch logic, where they help to determine macrobranch success/fail. In addition, the sized conditions are also output to the microbranch logic (UBRS MCAs) where they may be tested as microbranch conditions by some later microword.

PSL Condition Code Logic

The PSL Condition Code logic sets the PSL CC bits based on the recipe encoded in the I_MISC field (Table 3-29). Note that the CC bits will be left unchanged if the I_MISC field is encoded with other than a PSL CC recipe. In addition, if the I_MISC field is not required by the current microword, it is encoded with the NOP value normally used to inhibit changes to the state flags (I_MISC = 3F).

The new CC bits may be derived from the outputs of the Sized Branch logic or from the current CCs'. They may also be directly set/cleared from the WBUS or by the I_MISC recipe. If the new CCs are to be based on the raw condition codes from the EBox, the PSL CC recipe is always specified by the same microword that causes the EBox to produce the conditions. Note that the opcode of the current macroinstruction is not considered when setting the condition codes; only the recipe.

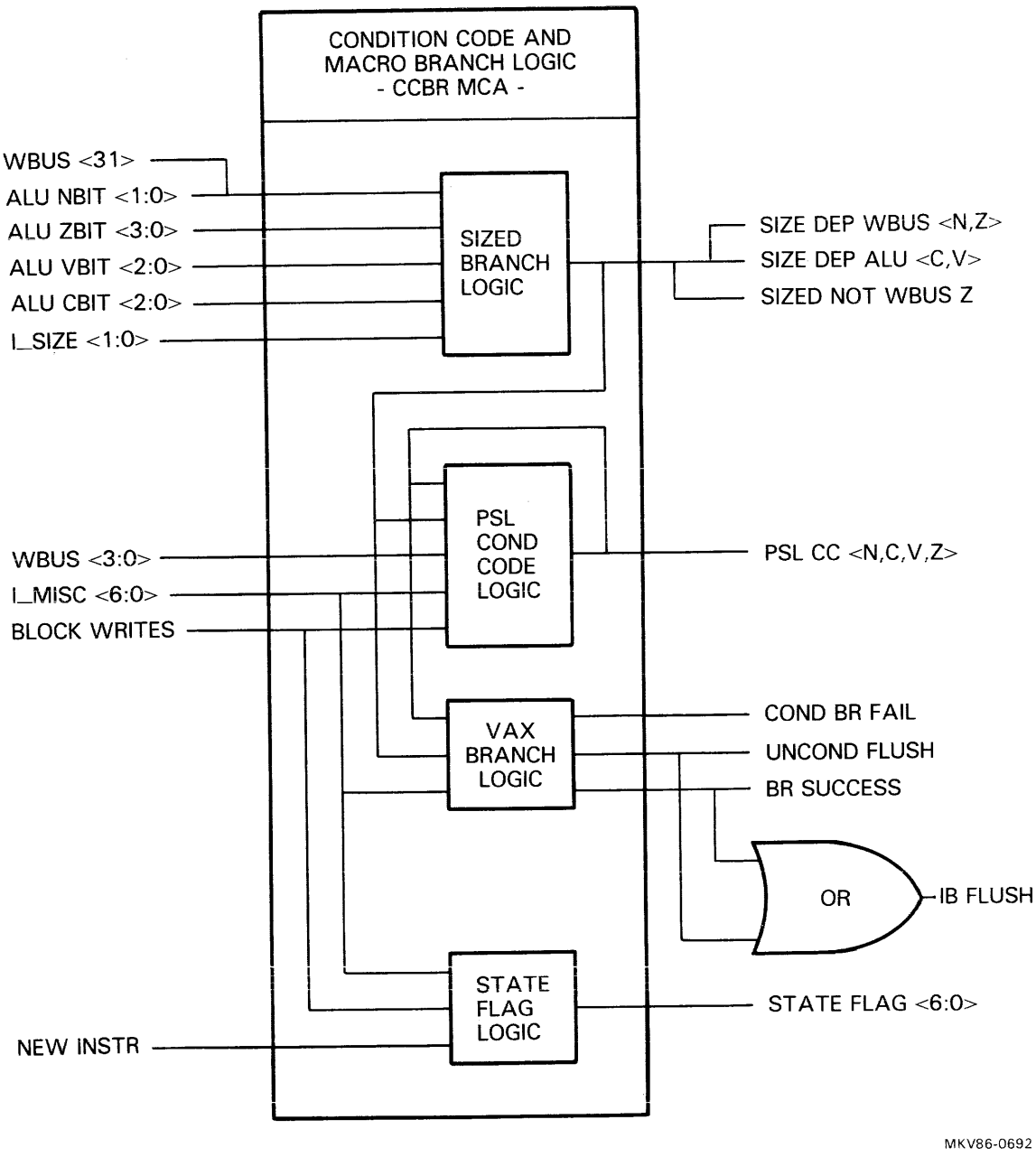


Figure 3-37 Condition Code And Macro Branch Logic

The PSL CC bits are available to the VAX Branch logic where they are tested to determine macrobranch success/fail and to the microbranch logic where they may be tested as microbranch conditions.

VAX Branch Logic

The VAX Branch logic examines the sized branch conditions, the CC bits and the macrobranch recipe encoded in the I_MISC field (Table 3-28). From these inputs, the VAX Branch logic then generates the appropriate macrobranch control signals BR SUCC, UNCOND FLUSH and COND BR FAIL.

State Flag Logic

The State Flag logic maintains 7 microcode state flags which provide firmware writers with more flexibility in controlling microcode flow. The state flags are set/cleared as specified by the I_MISC field.

Table 3-31 I_MISC Field Settings For State Flag Control

30	Clear Flag 0	38	Set Flag 0
31	Clear Flag 1	39	Set Flag 1
32	Clear Flag 2	3A	Set Flag 2
33	Clear Flag 3	3B	Set Flag 3
34	Clear Flag 4	3C	Set Flag 4
35	Clear Flag 5	3D	Set Flag 5
36	Clear Flag 6	3E	Set Flag 6
37	Clear All Flags	3F	No change (NOP)

Note from the above table that the state flags can only be set one at a time but may be cleared individually or as a group. The flags are typically set-up by one microroutine and then tested as microbranch conditions by a latter routine. The flags are all cleared during the first microword of every macroinstruction by the signal NEW INSTR.

3.6 SPECIAL REGISTER ADDRESSING

The File Address Slice (FADS) MCAs supply the addressing for the A and B port inputs to EBox main ALU. They also record changes made to GPRs during auto-increment and auto-decrement operations, and provide fast access to operands requiring multiple GPRs.

The main ALU inputs include the EBox register file (RGF), the slow data file (SDF), the PC and VA registers, the Cache Data Bus, the IB Data Bus, and the Bypass Bus.

Refer to Figure 3-38. The FADS MCAs contain the MDNUM, RNUM1, RNUM2, and RLOG registers, and the file read and write address control logic. The FADS MCAs allow microcode to specify register addresses explicitly or implicitly through the RNUM1, RNUM2, MDNUM, or the RLOG registers.

3.6.1 RNUM1 And RNUM2 Registers

The RNUM1 and RNUM2 registers are both 4 bits wide.

The GPR number of the current specifier is available during the first microword of a specifier routine and may be used to address the register file, or be saved in the RNUM1 or RNUM2 registers.

Since the GPR number is available only in the first microword of a specifier routine, it is stored in the RNUM1 register for use by later microwords. This allows the microprogrammers to write generic specifier flows without needing to remember which GPR is in use.

The RNUM1 register is also used for reading operands bigger than a longword from GPRs, for all register writes (except a few that use RNUM2), and for communicating with the RLOG.

The function of the RNUM2 register is similar to RNUM1, but its use is specialized. It records the GPR number used in the first of the two write specifiers for EDIV and EMODx instructions.

3.6.2 RLOG Register

The RLOG is a six stage shift register which records changes made to GPRs during autoincrement and autodecrement operations. This enables the microcode to "roll back" the GPRs if a fault occurs during macroinstruction execution.

Constants required by microcode to implement GPR autoincrement/ autodecrement operations are loaded in certain SDF locations when the system is initialized. Specifier flows add the constants, RLOG restoration code subtracts the constants. For example, autoincrement adds the appropriate constant (1, 2, 4, etc.) to the GPR, and saves the GPR number and the increment amount in the RLOG. When restoring the register, the same GPR address is used, but the value is subtracted.

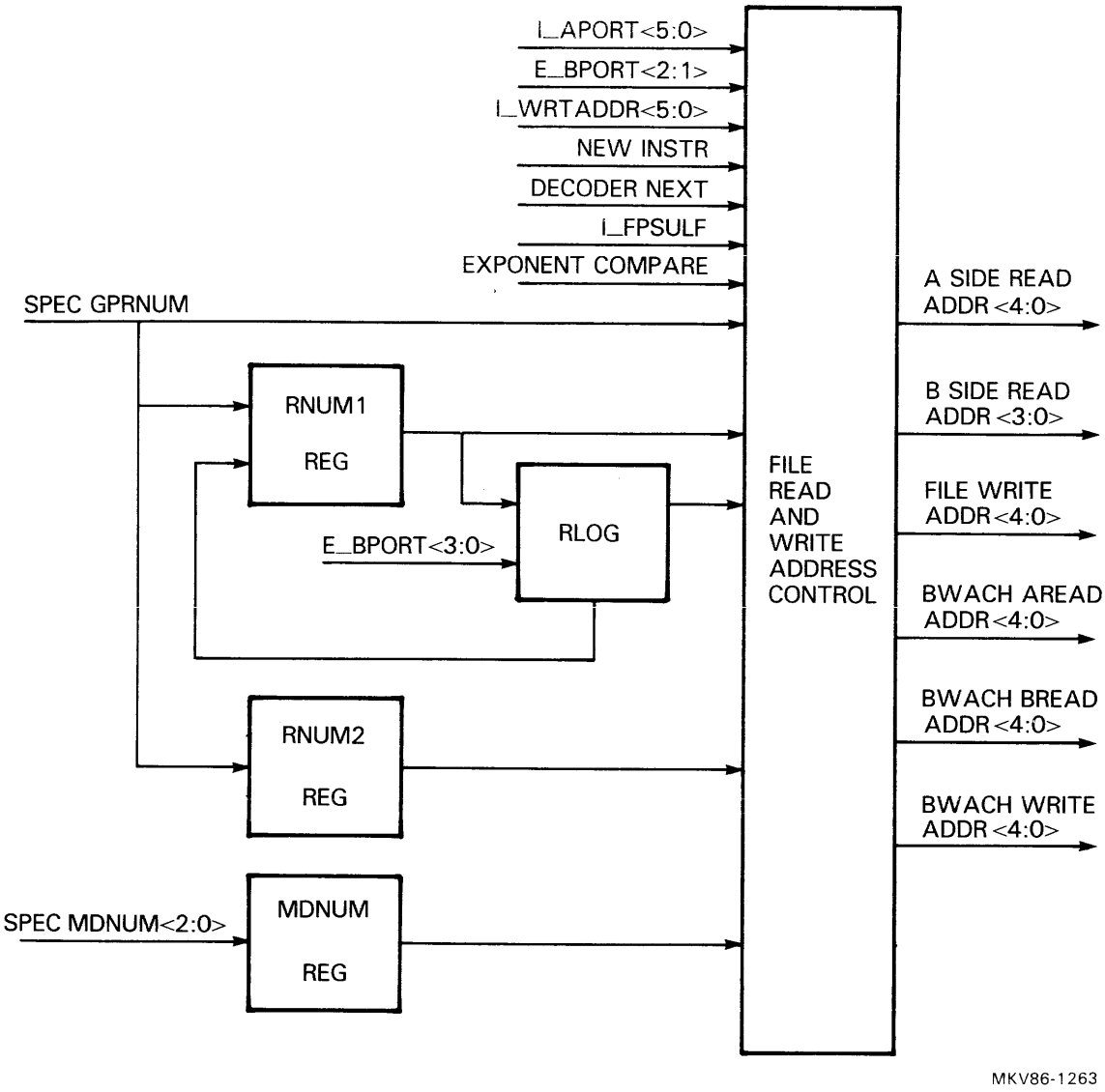


Figure 3-38 File Address Slice MCAs

Note that only autoincrement/decrement addressing modes use the RLOG due to access limitation to specifier GPRs. The result is that during use of the SP, the memory operation is performed first, and then the SP is changed. This ensures that if the memory operation fails, the SP change will not have taken place, and will not need to be corrected.

To allow quick access to large sized operands, file addresses are also indexed by some constants in hardware (RNUM1 by 1,2,3 and MDNUM by 1). For a floating point operation, file addresses may be modified to effect a swap of operands on the two ports of main ALU (floating point shuffle).

3.6.3 MDNUM Register

The MDNUM saves the address of the EBox memory data register (MDR) which is to receive data from the Cache Data Bus while processing an operand. A different MDR number is supplied by the decoder RAMs for each specifier.

The MDR number is a function of the opcode and current specifier number and eliminates the need for agreement between different opcodes concerning the location of the specifiers. The exception is that the MOVL, MOVAx, MOVZBL, MOVZWL group must agree in order to be optimized.

The microcoders decide which specifier of which opcode goes in which MDR. The MDNUM value can be:

- Merged into a cache read command so that there is no need for separate routines for each MDR
- Substituted for the register destination field of the microword so that specifier routines can drop operands in the right MD

Both uses serve to deliver operands into known MDRs for use by the instructions execute code.

The MDR number is valid the entire time a specifier is being processed. It may be used for either of its two jobs in the decoder generated microword, and in any later microword including the last one of the specifier routine. It is unpredictable in the opcode routine.

3.7 INTERRUPTS

This section describes the VAX 8800 interrupt servicing mechanism from the hardware point of view. Refer to the VAX 8800 Microcode Interpretation Guide (EK-KA88E-UG) for detailed information on how microcode and software handle interrupts.

3.7.1 Interrupt Requests

Interrupt requests can be generated by external devices, such as the NBIs and memory, by internal CPU conditions, such as the CPU power fail, or by software (MTPR SIRR).

Each interrupt request is assigned a specific interrupt priority level (IPL) as defined by the VAX architecture. The IPL of a device (or condition) is the priority the device must have in order to interrupt the current macroinstruction flow. If the IPL of the device is greater than the IPL of the current process, an interrupt will occur, causing the CPU to suspend the current process and to enter the appropriate interrupt service routine for the device.

There are 31 interrupt priority levels for the VAX 8800 system, 15 software and 16 hardware.

Software IPLs are numbered 01 to 0F and are implemented entirely by microcode. These IPLs are devoted totally to software use. There are no hardware device interrupts at these levels.

Hardware IPLs are numbered 10 to 1F. Interrupt levels 10 to 17 are reserved for external devices (NBIs, memory, console, etc.), levels 18 to 1F are reserved for internal CPU conditions (power fail and serious faults).

Table 3-32 lists the IPLs for the hardware interrupt requests. Note that devices that interrupt at the same level are listed in order of priority. For example, the interval timer interrupt has precedence over the two NBI BR 6 interrupts.

3.7.2 Interrupt Servicing

Refer to Figure 3-39.

The interrupt logic, which is part of the INPR MCA, monitors all hardware interrupt requests and generates a five-bit interrupt identification code, INTR ID <4:0>, to represent the level of the highest pending request. The identification code is tested by the microsequencer as a microbranch condition, allowing multi-way branching to the various interrupt service routines. The signal INTR PENDING is asserted to indicate INTR ID <4:0> validity.

Table 3-32 Hardware Interrupt Priority Levels

Interrupt Device or Condition	IPL (Hex)	Priority
Unassigned	1F	Highest
Power Fail	1E	
Machine Check	1D	
NMI Fault	1C	
Unassigned	1B-18	
NBI 0, NMI BR7	17	
NBI 1, NMI BR7	17	
Interval Timer	16	
NBI 0, NMI BR6	16	
NBI 1, NMI BR6	16	
NBI 0, NMI BR5	15	
NBI 1, NMI BR5	15	
Memory, NMI BR5	15	
NBI 0, NMI BR4	14	
NBI 1, NMI BR4	14	
Console Receive	14	
Console Transmit	14	
Other Processor	14	
Unassigned	13-10	Lowest

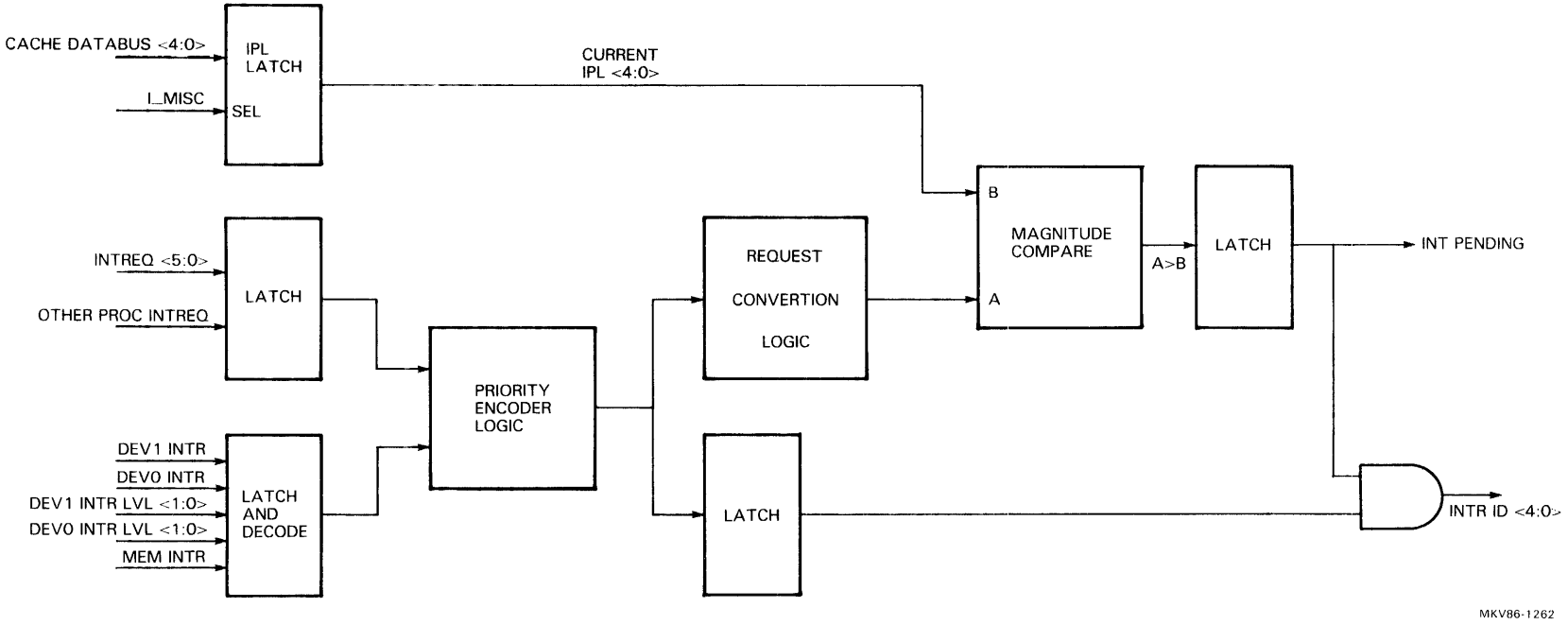


Figure 3-39 Interrupt Logic Simplified Block Diagram

The encoded INTR ID <4:0> value of a device (or condition) is not directly related to the IPL of the device. Table 3-33 contrasts the encoded INTR ID <4:0> values and the IPLs.

Pending interrupts are honored at two different times: between instructions (after one instruction has finished and before the next has stored any results or made any memory references), and at well-defined times during the execution of long instructions (for example, a MOVP).

The special address encoder (part of the IB decoder logic) checks for interrupts at instruction boundaries, the microsequencer checks for interrupts during the execution of long instructions. These logic elements both use the INTR PENDING bit to determine if the interrupt is to be honored.

The VAX 8800 system supports two I/O subsystems with the NBI I/O adapters (NBIA/NBIB module pair). Each NBI can an interrupt at any of four NMI BR levels (BR 4 to 7). The signals DEV0 INTR and DEV1 INTR signify the validity of the interrupt level encoded on the DEV0 INTR LVL <1:0> and DEV1 INTR LVL <1:0> lines.

Note that in a dual processor system, only one CPU is allowed to respond to interrupts generated by NMI connected devices (NBIs and memory). The NMI interrupt control register (see Chapter 1) of each CPU is set up at system boot time to insure that only one CPU will accept NMI interrupts.

The INTREQ <5:0> lines represent interrupts from the following:

Bit	Device or Condition
5	CPU power fail
4	CBox error (machine check)
3	NMI fault
2	Console receive
1	Console transmit
0	Interval clock

The INTREQ <5:0> lines are latched to provide a time window for priority arbitration. The priority arbitration logic is updated every clock cycle to reflect the current state of the devices and conditions which may generate interrupts.

Table 3-33 Interrupt ID Codes/IPLs

Interrupt Device or Condition	INTR ID <4:0>	IPL (Hex)
Power fail	11110	1E
Machine Check	11100	1D
NMI fault	11010	1C
NBI 0, NMI BR7	11000	17
NBI 1, NMI BR7	10110	17
Interval Timer	10100	16
NBI 0, NMI BR6	10010	16
NBI 1, NMI BR6	10000	16
NBI 0, NMI BR5	01110	15
NBI 1, NMI BR5	01100	15
Memory, NMI BR5	01010	15
NBI 0, NMI BR4	01000	14
NBI 1, NMI BR4	00110	14
Console Receive	00100	14
Console Transmit	00010	14
Other processor	00001	14
Passive release	00000	N/A

NOTE

All other INTR ID <4:0> codes are illegal and cause microcode to enter the machine check routine.

3.8 CONSOLE GATEWAY CONTROL

The gateway control (GWYC) MCA controls data transfers between IPRs resident in the console interface logic of the CLK module and the rest of VAX 8800 CPU. Note that each IBox in the dual processor environment has its own interface to the CLK module.

The GWYC MCA also controls the loading of the following CPU data structures during system initialization:

- CPU control store RAMS
- Cache control store RAMs
- Decoder RAMs
- Micromatch register

Figure 3-40 is a simplified block diagram of the gateway control logic.

3.8.1 Loading Control Store And Decoder RAMs

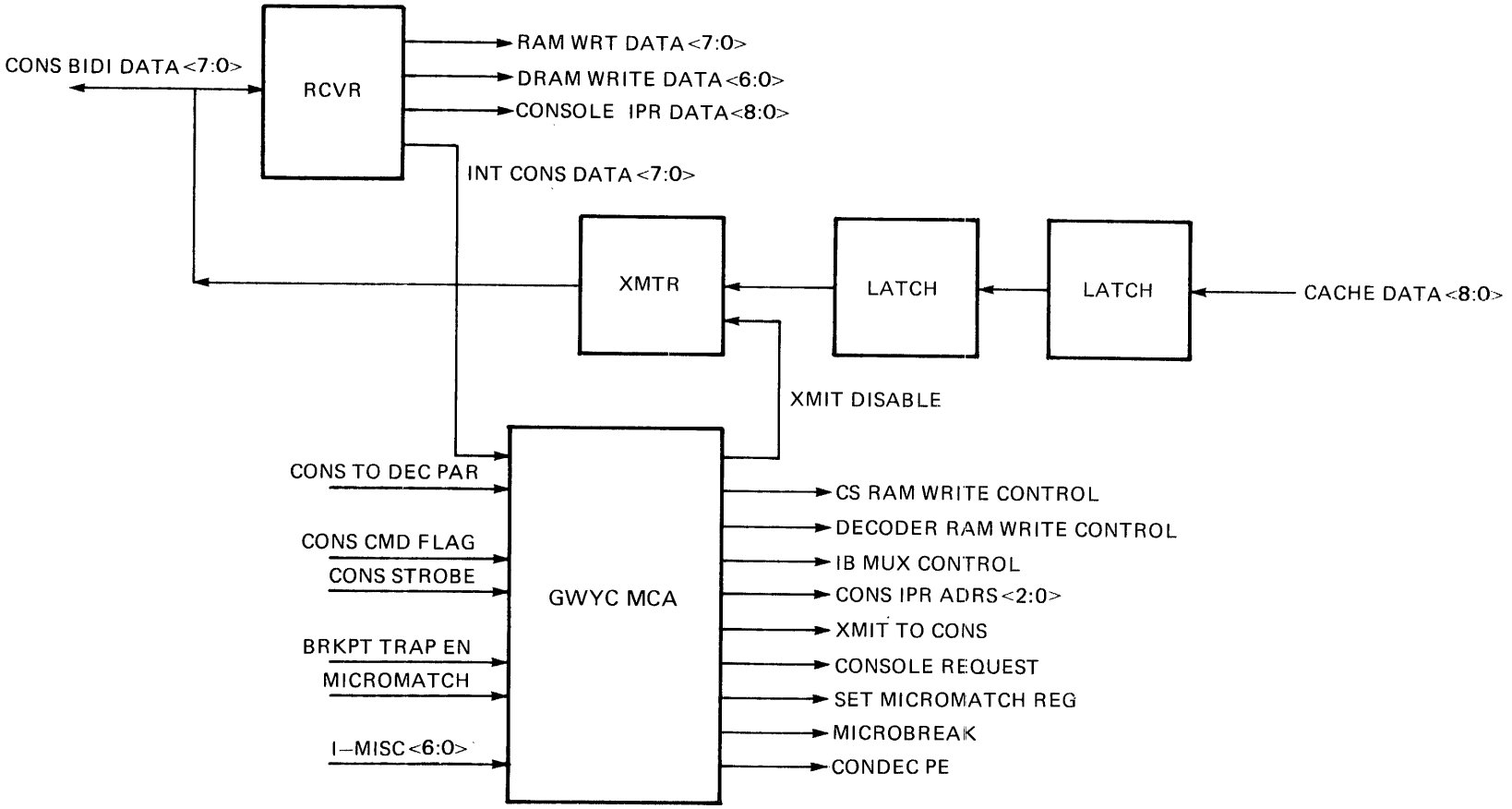
The console loads the CS RAMS and the decoder RAMS from the console Winchester disk through the bidirectional Cons Bidi Data bus which links the console interface logic on the CLK module to the IBox.

The console specifies the operation to be performed by issuing a command byte to the GWYC. Addresses and data are transferred to the GWYC in bytes and is controlled with a strobe signal from the console interface.

The basic sequence for loading the RAMs and DRAMs is:

- Write physical segment count for RAMs or DRAMs to be loaded (physical segment count defines the number of bytes to be loaded)
- Point VBUS to parity error bits
- Write RAM or DRAM address
- Write RAM or DRAM data (in bytes)
- Check Parity

The gateway logic buffers the console data and distributes a separate copy to each IBox module along with a write control and a module specific strobe signal to the destination module. During the time the GWYC is writing an address, a specific SET ADDRESS signal is generated for each DRAM, cache control store, or CPU control store data structure. Refer to Chapter 2 of the Console section of this manual for a description of the RAM/DRAM loading process.


CONS BIDI DATA<7:0>
RCVR
RAM WRT DATA<7:0>
DRAM WRITE DATA<6:0>
CONSOLE IPR DATA<8:0>
INT CONS DATA<7:0>
XMTR
LATCH
LATCH
CACHE DATA<8:0>
XMIT DISABLE
CONS TO DEC PAR
CONS CMD FLAG
CONS STROBE
BRKPT TRAP EN
MICROMATCH
I-MISC<6:0>
GWYC MCA
CS RAM WRITE CONTROL
DECODER RAM WRITE CONTROL
IB MUX CONTROL
CONS IPR ADRS<2:0>
XMIT TO CONS
CONSOLE REQUEST
SET MICROMATCH REG
MICROBREAK
CONDEC PE
MKV86-1264

3.8.2 Starting The Micromachine

Starting the micromachine depends on the interaction between the console interface on the CLK module and the GWYC MCA.

The console defines the operation to the GWYC MCA through with a command byte after the micromatch register (in the UBRS MCAs) is loaded with the appropriate microaddress.

The GWYC MCA selects the micromatch register as the source of the microaddress with the CONSOLE REQUEST signal. The console then initializes the microcode by bursting the clocks the appropriate number of times. When the microcode pipeline is initialized, the micromachine is started by unblocking the clocks.

3.8.3 Data Transfer With Console Resident IPRs

The GWYC MCA also controls data transfers between the CPU and the console interface resident IPRs. This function occurs only when the micromachine is running and is specified by the I_MISC field of the microword.

The GWYC MCA generates the IB MUX CONTROL signal to enable the selection of console data when IPRs are being read. When data is to be written to IPRs, the GWYC MCA generates the XMIT signal to enable the transfer of data to the console interface. During the IPR write and read functions, the GWYC MCA specifies the byte address of the IPR with the IPR ADDRESS lines.

3.8.4 Breakpoint Microtrap

Hardware supports a breakpoint feature which allows the console to stop the VAX CPU based on a console command. In this stopped state, it is possible for the console to examine the state of the CPU via the Vbus. The breakpoint feature is accomplished by specifying a breakpoint microaddress in the micromatch register of the UBRS MCAs and request one of two actions:

- Stop on Match
- Trap on Match

The micromatch register allows the console to set a breakpoint at any microaddress except an IB decoder generated addresses.

Once the breakpoint feature is established, hardware constantly compares the contents of the micro-PC with the breakpoint microaddress. When a match is found, the micromachine is either stopped by disabling the CPU clocks or a microtrap is generated.

The console specifies to the GWYC MCA whether a micromatch should generate a microtrap through the assertion of the BRKPT TRAPEN (breakpoint trap enable) signal.

3.8.5 Console Data Parity Check

The GWYC MCA checks parity on inbound console data and performs one of two actions in the event of a parity error. The controlling factor for handling a parity error is the current state of the micromachine. If the micromachine is stopped the console is alerted of the parity error by the assertion of the PAROUT signal. If the micromachine is running the parity error results in a machine check microtrap.

SECTION 7
EXECUTION BOX LOGIC (EBOX)

1.1 GENERAL

Figure 1-1 is a basic block diagram showing the logical placement of the execution unit (EBox) in the VAX 8800 CPU kernel.

The EBox performs logical and arithmetic functions under the direct control of the CPU microcode from the instruction unit (IBox) logic. It is the IBox that fetches and decodes macroinstructions from memory and initiates the microcode routines that control the CPU kernel.

The main functions of the EBox are to:

- Perform the logical shifts or rotates, and the integer or floating-point operations required to execute the VAX native instruction set and update the program counter (PC).
- Transfer data to and from registers in the cache unit (CBox) and the instruction unit (IBox).
- Process data from the CBox or IBox and pass it to the CBox along with the virtual address.
- Generate condition code branch information and pass it to the IBox microsequencer logic.
- Generate or check parity on data received from the CBox or IBox or from operators internal to the EBox.

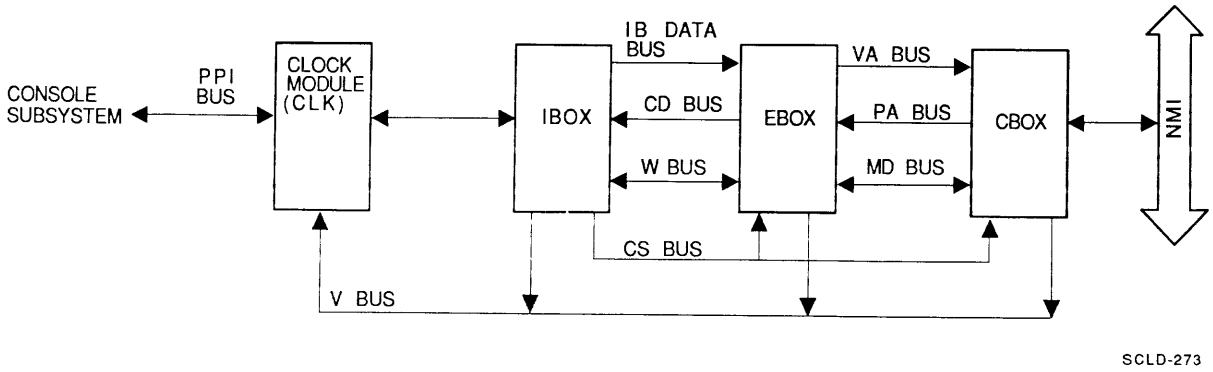


Figure 1-1 VAX 8800 CPU Kernel Block Diagram

1.1.1 EBox Organization

Figure 1-2 is a basic block diagram of the EBox. The EBox consists of three modules that form the following logic:

Data Path Slice (SLC) Modules

The slice module section provides the main address and data routing circuitry for the CPU kernel. It also maintains most of the privileged internal processor registers (privileged IPRs).

- The slice 1 module (SLC1) processes the upper data word, bits <31:16>.
- The slice 0 module (SLC0) processes the lower data word, bits <15:00>.

The slice modules also contain the main arithmetic logic unit (Main ALU), which performs arithmetic operations on both integer and floating-point data.

Shifter (SHR) Module

The SHR provides additional MCA and ALU operators that perform the shift/rotate and multiply/divide operations on integer or floating-point data and manipulate the signed exponent of a floating datum.

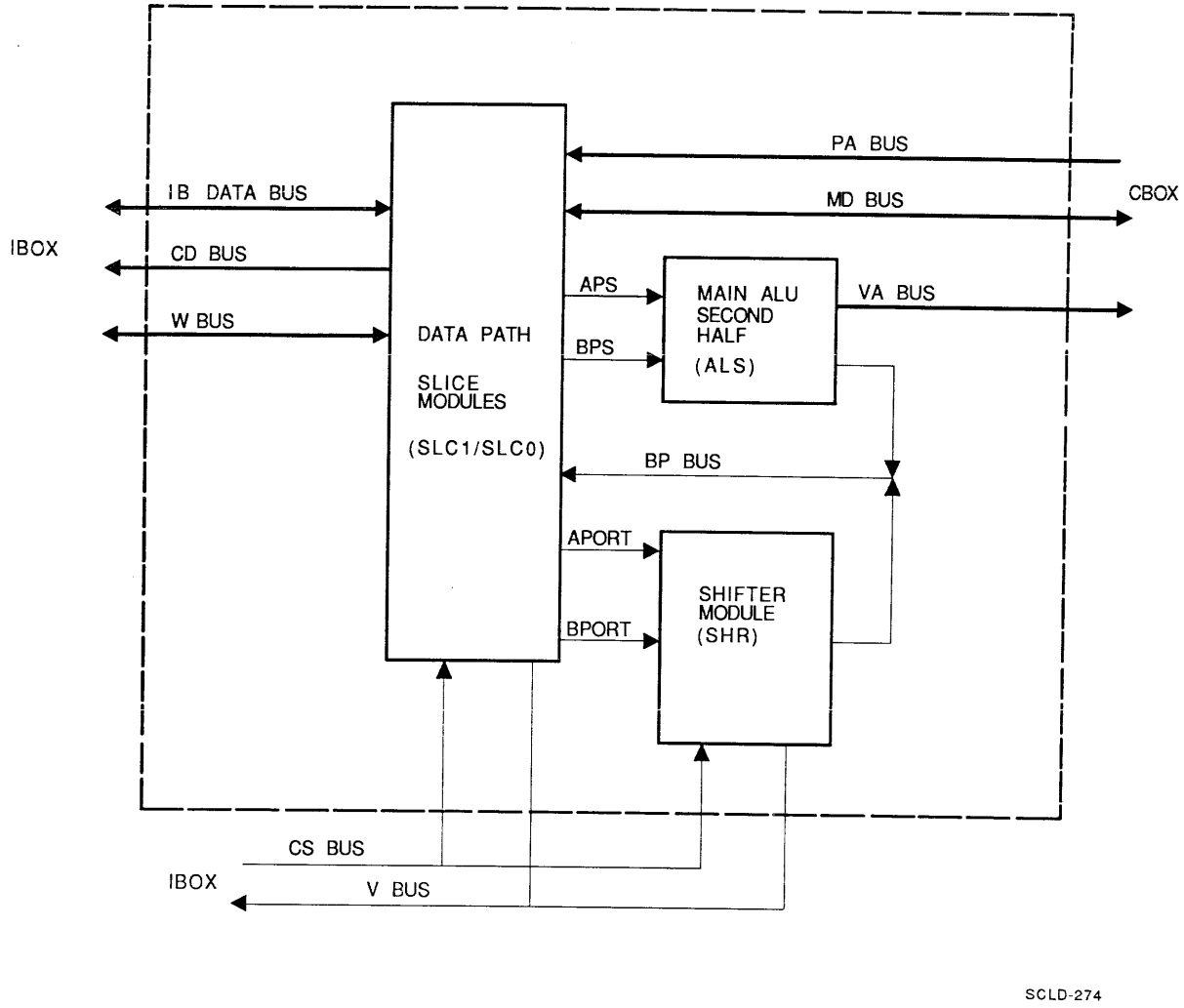


Figure 1-2 Execution Unit (EBox) Block Diagram

1.1.2 EBox Operators

The EBox contains the arithmetic logic unit (ALU) and macrocell array (MCA) elements, called operators, that perform the logical or arithmetic functions requested by the VAX native instruction set.

EBox operators are connected in parallel. Operand data is applied to all of the operators, but only the one that applies to the task is used by the microcode. For operations that require multiple cycles, the results of one cycle can be passed directly to the same, or another, operator to complete the calculation.

1.1.2.1 Main ALU -- The main ALU is located on the slice modules and consists of two halves.

ALU First Half (ALF)
The ALF performs the main multiplexing functions for the rest of the EBox and contains the partial sum logic for the first part of an arithmetic operation.

ALU Second Half (ALS)
The ALS contains the final sum and carry logic, which outputs the result of an arithmetic operation. The outputs from the ALS and operators located on the SHR are all connected in parallel on the bypass (BP) bus.

The ALF passes the first part of an arithmetic operation to the ALS on the A and B partial select (APS and BPS) lines. It also passes the first and second operand of an integer or floating-point operation to the SHR module over the APORT and BPORT lines. The selected operator then passes the result on the BP bus back to the data paths (slice modules) where the data can be stored or passed along to the CBox or IBox.

1.1.2.2 Cache Data Path (CDP) and Bus Watcher/Decoder (BWD) -- The CDP logic is located on the slice modules for fast access. Another section of EBox logic called the bus watcher/decoder is located on the decoder (DEC) module in the IBox.

1.2 SLICE MODULE (SLC1/SLC0) FUNCTIONS

The data path logic is formed by two slice modules (SLC1 and SLC0), each of which provides a 16-bit slice of the following 32-bit logic:

- Parity Generator/Checker (PAR)
- Register File (RGF)
- Slow Data File (SDF)
- Program Counter (PC) Subsystem
- Cache Data Path (CDP)
- Main Arithmetic Logic Unit (Main ALU)

1.2.1 Parity Generator/Checker (PAR)

The PAR receives the result of an operation from the main ALU or from the SHR module on the bypass (BP) bus. It passes the result to the SDF, RGF, or CDP over the write (W) bus.

The PAR performs the following logical functions:

- Contains the carry-save logic that stores the longword carry-out from the first cycle of a main ALU operation, allowing it to be used as a carry-in on the next cycle.
- Asserts the four highest and four lowest W bus bits to the IBox where they are used for microbranch test conditions.
- Generates the zero (Z) and negative (N) condition code bits for a byte, word, and longword and passes them to the IBox.
- Generates byte parity on the received BP bus data and asserts it on the W bus from where it is stored (with the W bus data) in the destination location.
- Combines the nibble parity generated by the main ALU to form byte parity.
- Generates parity on data from the main ALU and compares it with the received parity bits. If an error is detected, it sets a bit in the parity error register for the byte and generates a trap.

1.2.2 Register File (RGF)

The RGF consists of 32 high data rate longword registers with byte parity. It is used to maintain:

- Fifteen general-purpose registers (GPRs), except the PC register.
- Nine temporary registers (TEMPs) that are used as scratchpad registers by the microcode.
- Eight memory data registers (MDRs) that store data from cache memory.

The GPRs are available to all software levels. The TEMPS and the MDRs are only accessible by the microcode.

The RGF stores autoincrement/autodecrement results (in GPRs) or the partial results of long arithmetic operations (in TEMP registers).

The RGF can also be enabled to perform a floating-point shuffle (FPS). This can be used to sort two floating-point operands according to their exponent size.

1.2.3 Slow Data File (SDF)

The slow data file (SDF) and register file (RGF) are both used by the microcode for temporary data storage and for maintaining many of the VAX architectural registers.

The SDF provides 256 low data rate longword registers with byte parity. It is used to maintain:

- Most of the internal processor registers (IPRs)
- The data path constants
- Additional microcode scratchpad registers
- Temporary data when the RGF temporary (TEMP) registers are full
- Registers reserved for microdiagnostic test patterns

The IPRs are available only to the privileged software. All other registers are only accessed by the microcode. The SDF operates under timing restraints that inhibit further access to it for three cycles after a write operation.

1.2.4 Program Counter (PC) Subsystem

The PC subsystem maintains the PC, the PC incrementer, the backup PC and trap PC, and a register called the virtual address (VA) file.

These registers have the following functions:

PC	Supplies the translation buffer (TB) in the CBox with the virtual address for each operating code (op code), operand specifier (op spec), and operand in the instruction stream (I-stream).
PC Incrementer	Updates the PC by adding an increment value equal to the size of the I-stream data being processed (the I-size). The increment value is between 0 and 6 and is supplied by the PC increment generator from the IBox.
Backup PC	Stores the PC for the op code of a macroinstruction, restoring the PC if the instruction causes a macroexception (a reserved operand fault, for example). This allows the software service routines to examine the op code of a failing instruction and service the fault.
Trap PC	Maintains a recent history of PC activity, providing the microtrap service routines with a copy of the PC that was active at the time a microtrap occurred (a TB miss, for example).
VA File	Holds a copy of the last virtual address sent to the virtual address latch in the CBox. Serves as the backup if the address causes a microtrap.

The two highest-order bits are sent to the IBox sequencer (SEQ) module for use in the microbranch logic.

The two lowest-order bits are sent to the SHR module for use in byte alignment. They are also sent to the instruction buffer (IB) logic in the IBox for use in byte alignment after an IB flush.

1.2.5 Cache Data Path (CDP)

Although functionally part of the CBox, the cache data path (CDP) is located on the slice modules. The CDP consists of the cache data buffer (CDBF) and cache data store (CDS).

The CDBF can be written with modified data from the W bus or with memory data from the CBox on the memory data (MD) bus. From there, the data is written to the CDS RAM.

CDS data can be written to the IBox on the cache data (CD) bus or to the CBox on the MD bus (from where it is written to memory). A bypass multiplexer asserts new data to those lines while it is being written to CDS RAM.

1.2.6 Main Arithmetic Logic Unit (Main ALU)

The main ALU is a 32-bit adder that performs the main multiplexing functions for the EBox (see Section 1.1.1). It performs the following arithmetic functions and operations:

- Addition and subtraction with propagated carries and borrows on integer, floating-point, or decimal string data.
- Generates carry (C) and overflow (V) condition codes on the data and passes them to the IBox.
- Provides masking for the hidden bit and overflow bit positions on floating-point instructions.
- Logical AND, OR, and XOR operations.
- Passes the results of an operation to the PAR MCAs over the BP bus. The results are then passed on the W bus to the RGF or SDF, or to the cache data path (CDP) or IBox.
- For a CDP destination, the virtual address is passed to the CBox translation buffer. The VA file contents are also sent as a backup in the event that the virtual address causes a trap.

1.2.7 Bus Watcher/Decoder (BWD)

The BWD controls the bypassing operation, which selects the outputs for the BP bus and selects the inputs to the main ALU. When the BWD detects a condition that could prevent data from arriving at its destination in time for the current operation, it overrides the microcode fields that normally select the main ALU inputs, and enables the BP bus for the transfer.

1.3 SHIFTER MODULE (SHR) FUNCTIONS

The shifter module (SHR) receives 32 or 64 bits of data from the SLC logic on the APORT and BPORT and returns a 16 or 32-bit result on the BP bus. It provides the following sets of operators:

- Shifter (SHF)
- Floating-Point (FP) Support
  - Priority Encoder (PEN)
  - Shift ALU (SALU)
  - Exponent ALU (XALU)
- Multiplier/Divider (MULT)

The CPU microcode makes extensive use of the SHF, MULT, and main ALU operators in executing the VAX native instruction set (including all floating-point operations). However, the SHF performs several decimal string conversions while the PEN, SALU, and XALU provide hardware functions that support the F_ or D_floating and G_floating or integer data types.

1.3.1 Shifter (SHF)

The SHF shift matrix extracts a 16 or 32-bit output from a 32 or 64-bit input while processing the following three data types:

1. Integer
2. Floating-Point (FP)
3. Decimal String

1.3.1.1 Integer Data -- The SHF performs a right or left logical shift (or rotate) or an arithmetic shift of integer data. On an arithmetic right shift of more than 0, the output is sign-extended.

1.3.1.2 Floating-Point Data -- The SHF performs a normalize, align, or right or left shift of the fraction field of a floating-point datum. A fraction field shift is based on the shift count received from the FP support logic (PEN or SALU) on the shift count bus.

1.3.1.3 Decimal String Data -- The SHF performs the following decimal string data conversions:

- 32-bit trailing --> 16-bit packed
- 16-bit packed --> 32-bit trailing
- 32-bit packed --> 32-bit integer
- 32-bit integer --> 32-bit packed

1.3.2 Floating-Point (FP) Support

The FP support logic processes the sign and exponent fields of the F_floating, D_floating, and G_floating data types. It consists of three elements, each of which consists of a single MCA:

1. Priority Encoder (PEN)
2. Shift ALU (SALU)
3. Exponent ALU (XALU)

1.3.2.1 Priority Encoder (PEN) -- The PEN consists of the following logical subsections:

PE Logic	This logic scans the mantissa field of a floating datum to find the most significant 1 bit. It then passes the shift count necessary to normalize the number to the shifter (SHF) and to the exponent ALU (XALU).
Round Select Logic	During an FP alignment, this logic saves the last two bits shifted out of the data field for use as the rounding bits in a normalization.
Sticky Bit	This logic examines the eight most significant bits shifted out during an FP alignment. If all 8 bits are 0, the logic sets a carry-in bit for the main ALU.
Fast Normalize	This logic performs a 0 or 1-bit right or left shift of the low-order byte of a floating number, and passes the result to the rounding increment logic.
Rounding Increment	This logic takes the normalized 8 bits from the fast normalize logic and adds a 0 or 1 to it, according to the rounding bits and the operation (add or subtract). When enabled, it passes the result to the low-order byte of the floating number.

1.3.2.2 Shift ALU (SALU) -- The SALU subtracts the exponents of two FP operands and generates a shift count based on the difference. This shift count is sent to the shifter (SHF), which then shifts the fraction field of the smaller operand in order to align the operands.

1.3.2.3 Exponent ALU (XALU) -- The XALU performs arithmetic operations on the exponent fields of the floating data applied to its inputs. It adjusts the exponent field of the result according to the shift count from the PE logic.

1.3.3 Multiplier/Divider (MULT)

The MULT consists of eight 8-bit MCAs that are a custom implementation of very large scale integration (VLSI) technology.

The MULT produces a 64-bit multiply or 32-bit divide result. It improves the speed of both integer and floating-point multiplication by:

- Using an eight-bit-at-a-time multiply algorithm that generates eight result bits per MULT cycle. Eight cycles are required to produce the 32-bit result.
- Producing the correct two's complement results for integer data so that no premultiply or post-multiply sign bit correction is necessary.

It uses a one-bit-at-a-time division algorithm that generates two quotient bits per cycle. The quotient bits are subtracted from each other at the end of each division loop to produce the true quotient.

1.4 EBOX REGISTERS

Table 1-1 lists the privileged internal processor registers (IPRs) maintained in the EBox slow data file (SDF). The register numbers are expressed in hexadecimal (hex) notation. Software access is shown as read/write (R/W) or read-only (R/O).

Table 1-1 Privileged IPRs Maintained by the EBox

Register Name	Mnemonic	Number	Access
VAX Architectural Registers			
Kernel Stack Pointer	KSP	00	R/W
Executive Stack Pointer	ESP	01	R/W
Supervisor Stack Pointer	SSP	02	R/W
User Stack Pointer	USP	03	R/W
Interrupt Stack Pointer	ISP	04	R/W
P0 Base Register	P0BR	08	R/W
P0 Length Register	P0LR	09	R/W
P1 Base Register	P1BR	0A	R/W
P1 Length Register	P1LR	0B	R/W
System Base Register	SBR	0C	R/W
System Length Register	SLR	0D	R/W
Process Control Block Base	PCBB	10	R/W
System Control Block Base	SCBB	11	R/W
Interrupt Priority Level	IPL	12	R/W
Asynchronous System Trap Level	ASTLVL	13	R/W
Software Interrupt Summary	SISR	15	R/W
VAX 8800-Specific Registers			
Machine Check Status Register	MCSTS	26	R/W
System Identification Register	SID	3E	R/O
Revision Register 1	REVR1	86	R/O
Revision Register 2	REVR2	87	R/O

Notes:

1. Refer to the VAX Architecture Handbook for descriptions of the VAX Architectural IPRs.

2. The IPL register resides in the IBox as part of the PSL hardware image and in the SDF as part of the PSL software image. The microcode maintains both images equally.

1.4.1 P0LR, P1LR, and SLR Internal Bit Formats

P0LR, P1LR, and SLR are stored in the SDF in a different format than seen by the software because of memory management microcode requirements. The format changes are listed in Table 1-2. Conversion takes place when the MTPR and MFPR instructions are executed.

Table 1-2 P0LR, P1LR, and SLR Internal Formats

Register	Internal Format
P0LR and SLR	The software format multiplied by 512.
P1LR	The largest virtual address in P1 space (7FFFFFFF), minus the software format multiplied by 512.

1.4.2 VAX 8800-Specific Registers

The EBox contains four registers that are unique to the VAX 8800 system.

1.4.2.1 Machine Check Status Register (MCSTS) -- The MCSTS stores status information for the machine check microcode and the VAX/VMS operating system as shown in Figure 1-3. Table 1-3 provides bit descriptions.

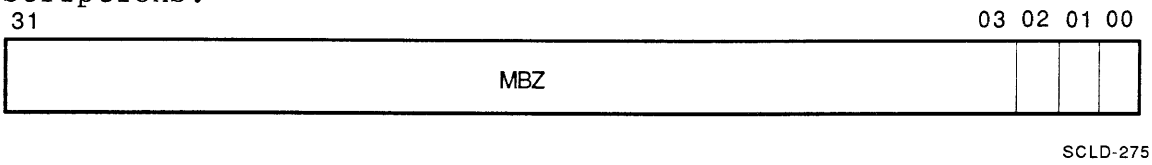


Figure 1-3 Machine Check Status Register (MCSTS)

Table 1-3 Machine Check Status Register (MCSTS) Bit Descriptions

Bit(s)	Name	Description
<31:03>	Must be Zeros (MBZ)	Unused and must be zeros.
<00>	Abort (ABT)	Set by the Machine Check (MC) microcode when an instruction is aborted. Used by VMS to determine if the error caused a fault or abort.
<01>	Enter Machine Check (MC)	Indicates that the MC microcode was entered but did not complete. Checked when the MC microcode starts to service an error. If set, the MC microcode passes control to the double error halt microcode. Otherwise, the MC microcode sets ENTER MC and processes the error.
<02>	Enter VMS	Indicates that the VMS handler was entered but did not complete. Checked when the MC microcode finishes servicing an error. If set, the VMS handler did not finish processing the previous error and the MC microcode passes control to the double error halt code. Otherwise, the MC microcode resets the ENTER MC bit, sets the ENTER VMS bit and passes control to the VMS handler. When VMS successfully processes the error, it resets the ENTER VMS bit.

1.4.2.2 System Identification (SID) Register -- As shown in Figure 1-4, the SID register holds the system identification codes that identify a specific VAX system. Table 1-4 describes the bit fields.

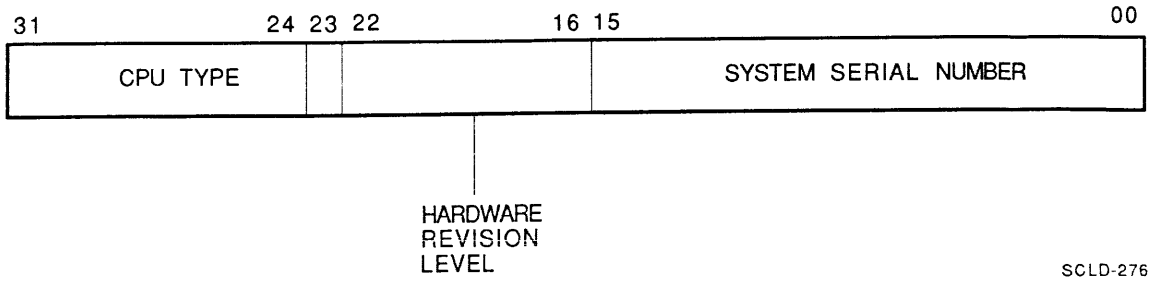


Figure 1-4 System Identification (SID) Register

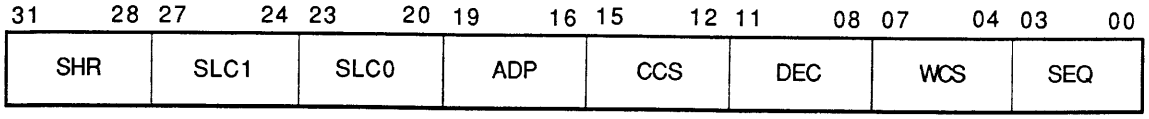
Table 1-4 System Identification (SID) Register Bit Field Descriptions

Bit(s)	Name	Description
<31:24>	CPU Type	Hardwired on the backplane in order to define a specific VAX processor. The value is equal to 00000110 (six) for the VAX 8800 system.
<23>	Left/Right	Defines the physical location and logical identification of the CPU in the system cabinet: 0 = Right CPU 1 = Left CPU
<22:16>	Hardware Revision Level	Kernel hardware revision level. Changed when a hardware revision impacts the VMS operating system or diagnostics.
<15:00>	System Serial Number	Hardwired jumpers on the backplane. Equal to the system serial number imprinted on the cabinet nameplate. It is the same for both CPUs on a dual-processor system.

1.4.2.3 Revision Registers (REVR1 and REVR2) -- REVR1 and REVR2 indicate (in decimal) the revision levels of the modules and other hardware components, and also some of the software components that are loaded during system initialization.

Figure 1-5 shows the REVR1 bit format, and Table 1-5 describes the bit fields.

Figure 1-6 shows the REVR2 bit format, and Table 1-6 describes the bit fields.



SCLD-277

Figure 1-5 Revision Register 1 (REVR1)

Table 1-5 Revision Register 1 (REVR1) Bit Field Descriptions

Bit Field	Unit	Module Revision Level
<31:28>	EBox	Shifter module (SHR)
<27:24>	EBox	Slice 1 module (SLC1)
<23:20>	EBox	Slice 0 module (SLC0)
<19:16>	CBox	Address Data Path module (ADP)
<15:12>	CBox	Cache Control Sequencer module (CCS)
<11:08>	IBox	Decoder module (DEC)
<07:04>	IBox	Writeable Control Store module (WCS)
<03:00>	IBox	Sequencer module (SEQ)

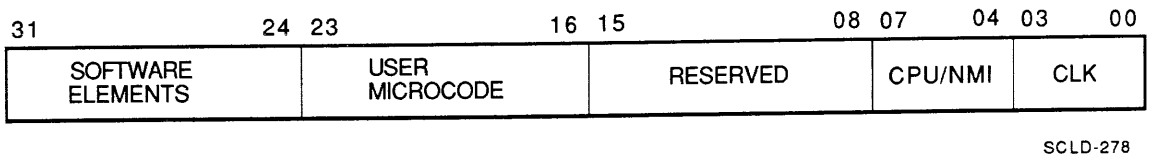


Figure 1-6 Revision Register 2 (REVR2)

Table 1-6 Revision Register 2 (REVR2) Bit Field Descriptions

Bit Field	Name	Definition
<31:24>	Software Elements	Revision level of the software components loaded at system initialization time from the console storage medium. The software components include: - CPU Control Store firmware - Cache Control Store firmware - Constants loaded to the SDF - Data loaded into the I-decode RAMs (Changes made to individual components are incorporated into a complete package before being released.)
<23:16>	User Microcode	Revision level of the user microcode loaded by the user to the WCS.
<15:08>	Reserved	
<07:04>	CPU/NMI	Revision level of the CPU/NMI backplane.
<03:00>	Clock Module	Revision level of the Clock (CLK) module.

1.4.2.4 EBox Parity Error Register (EBER) -- The EBER indicates parity errors detected on the APORT and BPORT buses and provides pointers to the source of the errors.

The microcode stores the EBER contents along with the contents of similar registers in the CBox and IBox in an SDF data structure called the machine check error bank (MCEB). When a machine check occurs, several machine check-related registers are written to the MCEB. The MCEB is then pushed onto the stack to provide software access to the various registers. For further information on the MCEB, refer to the machine check description in the IBox section of this manual.

The EBER is located in the parity generator/checker (PAR) and is only accessible by the microcode. It is described in further detail in Chapter 2.

CHAPTER 2
FUNCTIONAL DESCRIPTION

2.1 GENERAL

The EBox makes extensive use of high density emitter-collector logic (ECL) for its main logical functions. Most of this ECL logic is based in macrocell array (MCA) chips that make up the main logical operators.

This chapter describes the EBox logic to the block diagram level and is intended for use with the module print sets.

2.2 SLICE MODULE (SLC1/SLC0) DESCRIPTION

Figure 2-1 is a block diagram of the 32-bit data path logic formed by the SLC1 and SLC0 modules. Each module provides a 16-bit slice of the following EBox operators:

- Parity Generator/Checker (PAR)
- Register File (RGF)
- Slow Data File (SDF)
- Program Counter (PC) Subsystem
- Cache Data Path (CDP)
- Main Arithmetic Logic Unit (Main ALU)

2.2.1 Parity Generator/Checker (PAR)

Figure 2-2 is a block diagram of the parity generator/checker (PAR) logic, which is formed by two PAR MCAs on each slice module. Each MCA passes one byte of data to the rest of the data paths and generates byte parity for storage with, or checks on, the received data.

The PAR provides the following logical functions:

- Parity Generator
- Parity Checker
- Parity Error (PE) Register
- Carry Save Logic

Table 2-1 describes the PAR input and output signals.

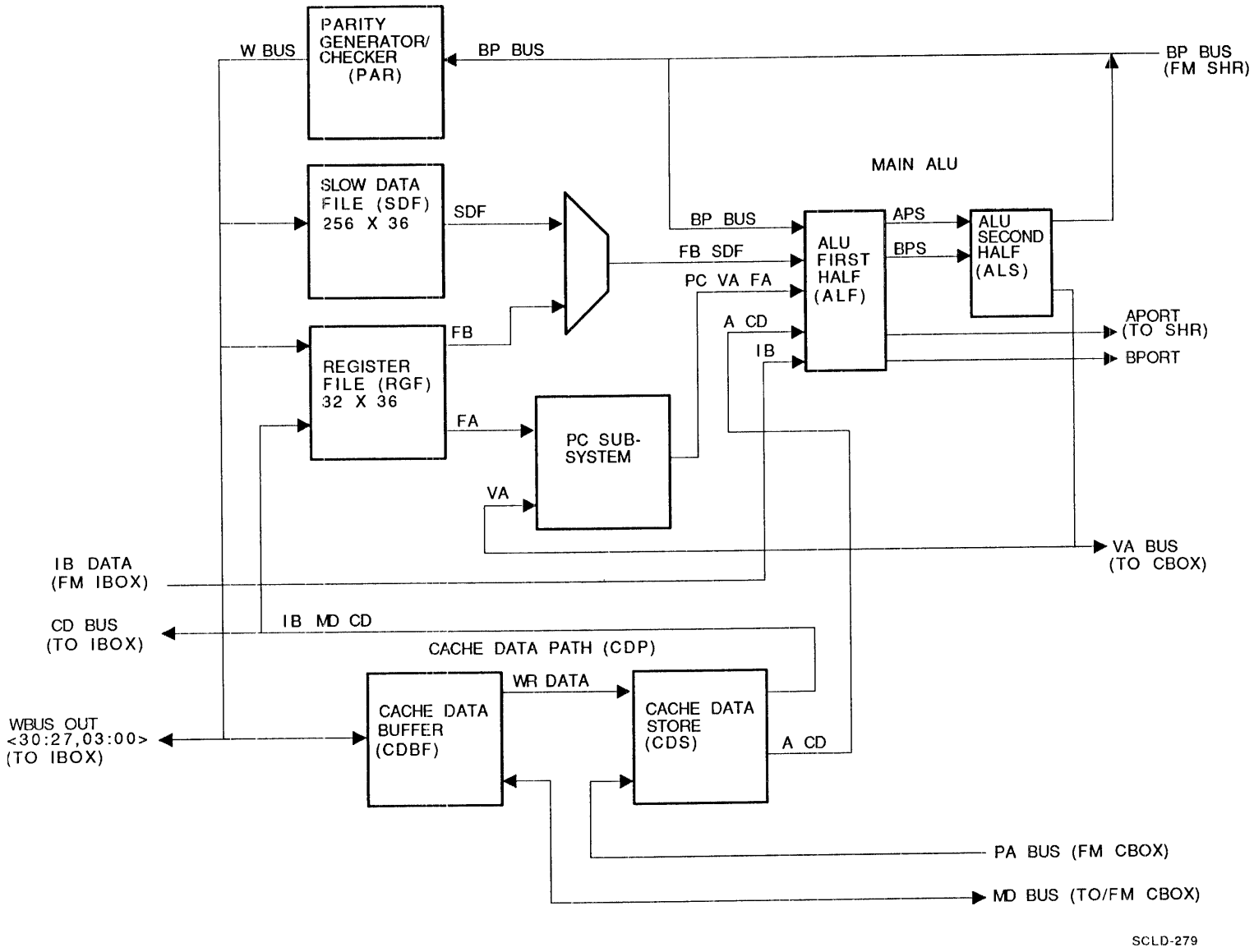


Figure 2-1 Slice Module (SLC1/SLC0) Block Diagram

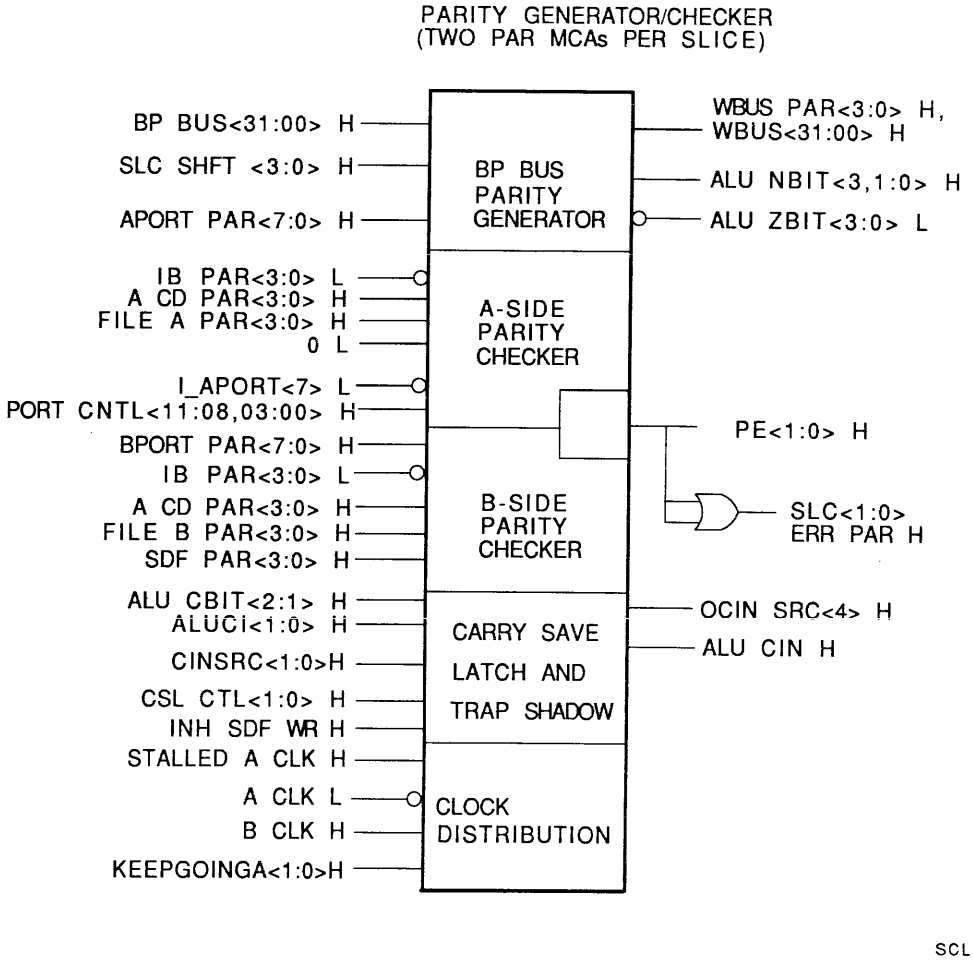


Figure 2-2 Parity Generator/Checker (PAR) Block Diagram

Table 2-1 Parity Generator/Checker (PAR) Signal Descriptions

Signal Name	Description	Valid
Data Input and Parity Control Signals		
Bypass Bus (BP BUS<31:00> H)	The PAR section receives longword data from the main ALU or the shifter (SHR) module ALUs on the bypass (BP) bus.	T7 to T9
Slice Shift (SLC SHFT<3:0> H)	E_SHFT<4:3,1:0> from the microword control the PAR MCA parity register (see Table 2-5).	T5 to T7

Table 2-1 Parity Generator/Checker (PAR) Signal Descriptions (Cont)

Signal Name	Description	Valid
Data and Parity Output Signals		
Write Bus (W BUS<31:00> H)	The W bus passes received BP bus data to the slow data file, register file, and BP cache data path. W BUS <30:27,03:00> H are asserted to the sequencer (SEQ) module as W BUS OUT<30:27,03:00> H for microbranch conditions.	T8 to T10
Write Bus Parity (W BUS PAR<3:0> H)	Byte parity is generated on received BP bus data and asserted as the W bus byte parity bits (one bit from each PAR MCA). W BUS PAR<3:0> are written to the slow data file, register file, or cache data path along with the received BP bus data.	T8 to T10
ALU Negative Bit (ALU NBIT<3,1:0> H)	Asserted to the IBox, which produces the negative (N) condition code bit for a byte, word, or longword. ALU NBIT<3> H is asserted from PAR 3 as W BUS OUT<31> H.	T8 to T10
ALU Zero Bit (ALU ZBIT<3:0> L)	Asserted to the IBox, which combines the signals and produces the zero (Z) condition code bit for a byte, word, or longword. NOTE N and Z byte values are generated on the BP bus inputs and are not valid when reading the parity error register.	T8 to T10
Parity Error (PE<1:0> H)	ORed on each slice module. Asserts the slice parity error signals SLC1 PAR ERR H and SLC0 PAR ERR H to the sequencer (SEQ) module to generate a machine check. (These bits do not include cache data path parity errors.)	T8 to T10

Table 2-1 Parity Generator/Checker (PAR) Signal Descriptions (Cont)

Signal Name	Description	Valid
A-Side Input Signals		
APORT Parity (APORT PAR<7:0> H)	APORT nibble parity is generated by each of the eight main ALU first-half (ALF) MCAs on an ALU operation.	T6 to T8
IB Parity (IB PAR<3:0> L)	Instruction buffer (IB) byte parity from the IBox.	T5 to T7
Cache Data Parity (A CD PAR<3:0> H)	Byte parity from the cache data path (CDP).	T5 to T7
File A Parity (FILE A PAR<3:0> H)	File A byte parity from the register file (RGF).	T5 to T7
I_APORT<7> ($\overline{\text{APORT}}$<7> L)	I_APORT<7> from the microword ($\overline{\text{SLC1}}$/SLC0 CS0 DATA<1> L signal on the backplane). Used with PORT CNTL<11:08,03:00> to select the A-side byte data/parity bit source (see Table 2-2).	T4 to T6
APORT Control (PORT CNTL<11:08, 03:00> H)	A-side port control from the (DEC) module. Used with I_APORT<7>, above, to select the A-side byte data/parity source.	T5 to T7
B-Side Input Signals		
BPORT Parity (BPORT PAR<7:0> H)	BPORT nibble parity is generated by each of the eight main ALU first-half (ALF) MCAs.	T6 to T8
IB Parity (IB PAR<3:0> L)	Instruction buffer (IB) byte parity from the IBox.	T5 to T7
Cache Data Parity (A CD PAR<3:0> H)	Byte parity from the cache data path (CDP).	T5 to T7
File B Parity (FILE B PAR<3:0> H)	File B byte parity from the register file (RGF).	T5 to T7
Slow Data File Parity (SDF PAR<3:0> H)	Byte parity from the slow data file (SDF).	T5 to T7

Table 2-1 Parity Generator/Checker (PAR) Signal Descriptions (Cont)

Signal Name	Description	Valid
Slow Data File Write (SDF WRITE L)	From the SDF, inhibits parity checking during a write to the SDF. Valid from T8 to T10 of the microinstruction requesting the SDF write.	T8 to T10
E_BPORT<8> (B RD MODE L)	E_BPORT<8> from the microword (asserted as SLC1/SLC0 CS0 DATA<6> L on the backplane). Used with PORT CNTL<15:12,07:04> to select the B-side byte data/parity input source (see Table 2-3).	T5 to T7
BPORT Control (PORT CNTL<15:12, 07:04> H)	B-side port control from the decoder (DEC) module. Used with E_BPORT<8>, above, to select the B-side byte data/parity input source.	T5 to T7
Carry Save and Trap Shadow Signals		
ALU Carry Bit (ALU CBIT<2:1> H)	Asserted from the upper ALU byte to the upper PAR byte on the same slice module. Valid from T7 to T9 of the instruction producing a word carry.	T7 to T9
Output Carry In Source (OCIN SRC<4> H)	Asserted from the upper PAR byte of each slice module. Asserted as CIN SRC<1> H to the carry save logic in the upper PAR byte of the other module.	T6 to T8
ALU Carry In (ALUCI<1:0> H)	E_ALUCI<1:0> from the microword. Used with CSL CTL<0> to select the next ALU CIN carry source (see Table 2-6).	T5 to T7
ALU Carry In (ALU CIN H)	Asserted from upper PAR byte to both ALU bytes on the same slice module. Valid from T6 to T8 of the microinstruction using the carry in.	T6 to T8

Table 2-1 Parity Generator/Checker (PAR) Signal Descriptions (Cont)

Signal Name	Description	Valid
Carry In Source (CINSRC<1:0> H)	CINSRC<1> H is OCIN SRC<4> H from the upper PAR byte of the other slice module. Valid from T7 to T9 of the microinstruction generating the carry. CINSRC<0> H is CIN SRC1<0> H from the SHR guard bit logic to both SLC1 and SLC0. Valid from T7 to T9 of the microinstruction generating the carry.	T7 to T9
Carry Save Latch Control (CSL CTL<1:0> H)	CSL CTL<1> H is E_LDCSL from the microword and is used to load the carry save latch. CSL CTL<0> H is FPOP from the upper ALU byte on the same slice module. Used with ALUCI<1:0> to select the integer carry or floating-point carry as the source for the next ALU CIN.	T6 to T8
Inhibit SDF Write (INH SDF WR H)	Asserted by the PC subsystem following a microinstruction causing a global trap. Valid from T11 to T17; this is a half-cycle delayed and 3-cycle extended trap signal.	T11 to T17
Clock Distribution Control		
Keepgoing (KEEPGOINGA <1:0> H)	Holds the cache data parity A latches open (A CD PAR<3:0> outputs) during a stall (see Table 2-4).	T4 to T6

2.2.1.1 Parity Generator -- The parity generator logic performs the following functions:

- Receives data from the main ALU on the BP bus if ALUENBP is asserted. Otherwise, it receives data from the SHR module or receives only zeros. It distributes received data on the W bus to the RGF, SDF, and CDP.

- Generates byte parity on the received BP bus data and distributes it on the W bus from where it may be written to the RGF, SDF, or CDP. Zero parity may be forced for maintenance operations.

- Generates the negative (N) and zero (Z) bit for each byte and passes them to the IBox where they are combined to form the N and Z condition code bits of the processor status longword (PSL).

2.2.1.2 Parity Checker -- The parity checker logic performs the following operations on the main ALU data:

- When the main ALU performs an add or subtract operation, it generates nibble parity on the data asserted to its A-side and B-side. It outputs the nibble parity to the A-side and B-side of the parity checker where it is combined to form byte parity.

- The parity checker compares the byte parity bits with those from the data sources applied to the A-side and B-side of the main ALU. The byte parity error signals from both slices are ORed and asserted to the sequencer (SEQ) module to generate a machine check.

A-Side Checker

The A-side checker compares the main ALU APORT parity output bits with the parity bits from the following sources:

- Instruction Buffer (IB) parity bits
- Cache Data A (A CD) output parity bits
- File A (FILE A) output parity bits

I_APORT<7> and the port control fields PORT CNTL<11:08,03:00> from the decoder (DEC) module control the A-side checker as shown in Table 2-2. Only the lowest bit-pair is described; the functions for the other 3 pairs are identical.

B-Side Checker

The B-side checker compares the main ALU BPORT parity output bits with the parity bits from the following sources:

- Instruction Buffer (IB) parity bits
- Cache Data A (A CD) output parity bits
- File B (FILE B) output parity bits
- Slow Data File (SDF) parity bits

The BRDMODE bit and the port control fields PORT CNTL<15:12,07:04> from the decoder (DEC) module control the B-side checker as shown in Table 2-3. Only the lowest bit-pair is described; the functions for the other 3 pairs are identical.

Other Checker Functions:

- The parity checker is disabled during SDF writes and for A-side data that involves a floating-point mask (a function that changes the data before asserting it on the APORT lines). (The parity checker may also be disabled for maintenance functions.)

- The first parity error on either side generates an EBox parity error trap. The error for that side is reported in the parity error (PE) register and that side of the register is locked. Further errors on the same side are not recorded but are serviced by an EBox parity error machine check.

- A microinstruction that generates bad parity completes the write to the destination whether the data is correct or not. Further writes are blocked by the trap shadow.

- Cache data (CD) parity checks are maintained during a stall, depending on the state of the CD ready or CD valid bits and the memory data register (MD) valid bit as shown in Table 2-4.

Table 2-2 A-Side Port Control

I_APORT<7>	PORT CNTL <1	0>	Selected Data/Parity Bit Source
0	0	0	Virtual Address (VA) or Program Counter (PC)
1	0	0	File A
-	0	1	Instruction Buffer (IB)
-	1	0	Bypass (BP) Bus
-	1	1	Cache Data (CD) Bus

Table 2-3 B-Side Port Control

BRDMODE	PORT CNTL <5	4>	Selected Data/Parity Bit Source
0	0	0	Slow Data File (SDF)
1	0	0	File B
-	0	1	Instruction Buffer (IB)
-	1	0	Bypass (BP) Bus
-	1	1	Cache Data (CD) Bus

Table 2-4 Keepgoing Conditions for A CD PAR<3,1>

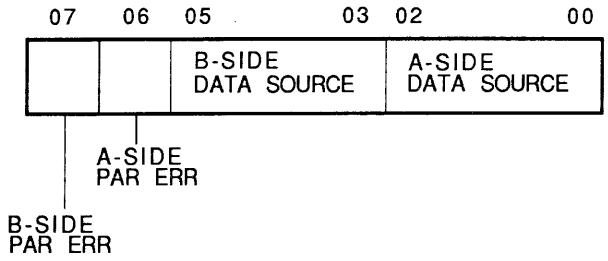
CD Valid	CD Ready	MD Valid	Keepgoing
1	1	-	0
-	-	1	0
0	-	0	1
-	0	0	1

2.2.1.3 EBox Parity Error Register (EBER) -- Figure 2-3 shows the EBER bit format and functions. This register occurs in the MCA for each byte and peforms the following functions:

- Monitors the ALU input select lines to determine the source of a byte parity error.
- Records the first byte parity error that occurs (for the A-side or the B-side) and indicates the source of the data that produced the error.

Table 2-5 shows how the E_SHFT field of the microword controls the EBER register logic.

A parity error on either side generates a parity error trap and is stored, locking that side of the register. Further errors generate machine checks. The effects of parity errors can only be inhibited by disabling all traps from the console.



B-SIDE DATA SOURCE VALUE

0 = SDF
1 = CACHE - IB MD CD BUS
2 = BP BUS (NO PAR CHECK)
3 = CACHE - A CD BUS
4 = RGF - FILE B

A-SIDE DATA SOURCE VALUE

0 = VA OR PC (NO PAR CHECK)
1 = CACHE - IB MD CD BUS
2 = BP BUS (NO PAR CHECK)
3 = CACHE - A CD BUS
4 = RGF - FILE A

SCLD-281

Figure 2-3 EBox Parity Error Register (EBER)

Table 2-5 E_SHFT<4:0> Control of the EBox Parity Error Register (EBER)

E_SHFT Bits <4 3 2 1 0>	Register Function
1 1 - 0 0	Inhibits A-side parity checking during a floating-point mask operation.
1 1 - 0 1	Forces zero W bus parity during a maintenance operation.
1 1 - 1 0	Clears (opens) the parity error register.
1 1 - 1 1	Reads (closes/latches) the parity error register onto the W bus.

2.2.1.4 Carry Save Logic -- The carry save logic contains the following functional elements:

- The carry-in multiplexer
- The carry save latch (CSL)
- The floating-point carry selection latch

Carry Save Functions

The PAR MCA for the upper byte on each slice module uses the CSL to preserve word and longword carries from an ALU operation. A latched carry-out is then used as a carry in on the next ALU cycle:

- A latched integer carry-out from ALU bit <31> (CSL<31>) is carried into ALU bit <00> on the next integer cycle.
- A latched floating-point carry-out from ALU bit <15> (CSL<15>) is carried into ALU bit <16> on the next floating-point cycle.

Table 2-6 shows how the carry-in multiplexer source is controlled from the EALUCI<1:0> field of the microword. The microword and hardware both select the next carry in. The microword can select a 0, 1, the CSL, or the guard bit from the SHR module. The floating-point operation (FPOP) bit from the ALU defines whether the cycle is an integer or floating-point operation.

Traps

On a trap, backup latches are used to save the contents of the carry save and floating-point carry selection latches. The inhibit SDF write (INH SDF WR) signal from the PC subsystem indicates a trap shadow, loading the backup latches, and also multiplexing the backup values to the carry save and floating-point carry selection latches. The latches are not loaded by the trap routine.

Table 2-6 E_ALUCI<1:0> Control of the ALU Carry-In Source

FPOP	ALUCI Bits <1	0>	ALU Carry-In Source
1	0	0	ALU1<16> <-- FP CIN
0	0	1	ALU0<00> <-- 0
1	1	0	ALU0<16> <-- 1
0	1	1	ALU0<00> <-- CSL<31> (Integer)
1	1	1	ALU1<16> <-- CSL<15> (FP)

2.2.2 Register File (RGF)

The RGF File B data outputs are multiplexed with the SDF data outputs and passed to the B-side of the main ALU (Figure 2-1). However, the parity outputs for both RGF File A and File B, and for the SDF, are connected to the A-side and B-side inputs of the PAR parity checker.

Figure 2-4 is a block diagram of the register file (RGF) logic, which provides thirty-two 32-bit registers with byte parity. The RGF consists of a set of write-enable delay latches that drive two custom ECL muliple-register (MPR) MCAs per slice.

- The RGF is used for all of the general-purpose registers (GPRs) except the program counter. The remaining 17 locations are used by the microprogram as memory management and scratchpad registers.

- Two separate outputs (File A and File B) carry 32-bit data to the A-side and B-side of the main ALU.

- Two separate read addresses and two separate write addresses may be applied to the RGF RAM at once. Two read and two write functions may take place at the same time with the following limitations:

  - Two reads may access the same location at the same time.

  - Two writes to the same location at the same time produces unspecified results.

  - The W bus has byte, word, or longword access to all 32 locations.

  - The CD bus has longword access to the lowest 8 locations. These are the memory data (MD) registers and are not used as general-purpose registers.

Table 2-7 shows the RGF memory address allocation. Table 2-8 describes the RGF read and write signals.

2.2.2.1 Floating-Point Shuffle (FPS) -- Under control of the microcode and the SHR module, the IBox performs an FP shuffle by swapping the RGF read addresses. A shuffle occurs when FPS is enabled (EFPSUFL from the microword) and the XALU compare (XALUCC H) signal is asserted by the XALU on the SHR module.

Bit <1> of either read address is also complemented if the XALU compare signal is asserted and address bit <2> is a 1. This swaps the quadword/longword scratch locations for read purposes, but only in file areas that are defined with address bit <2> as a 1. The function is mainly used to sort two floating-point operands in order according to their exponent size.

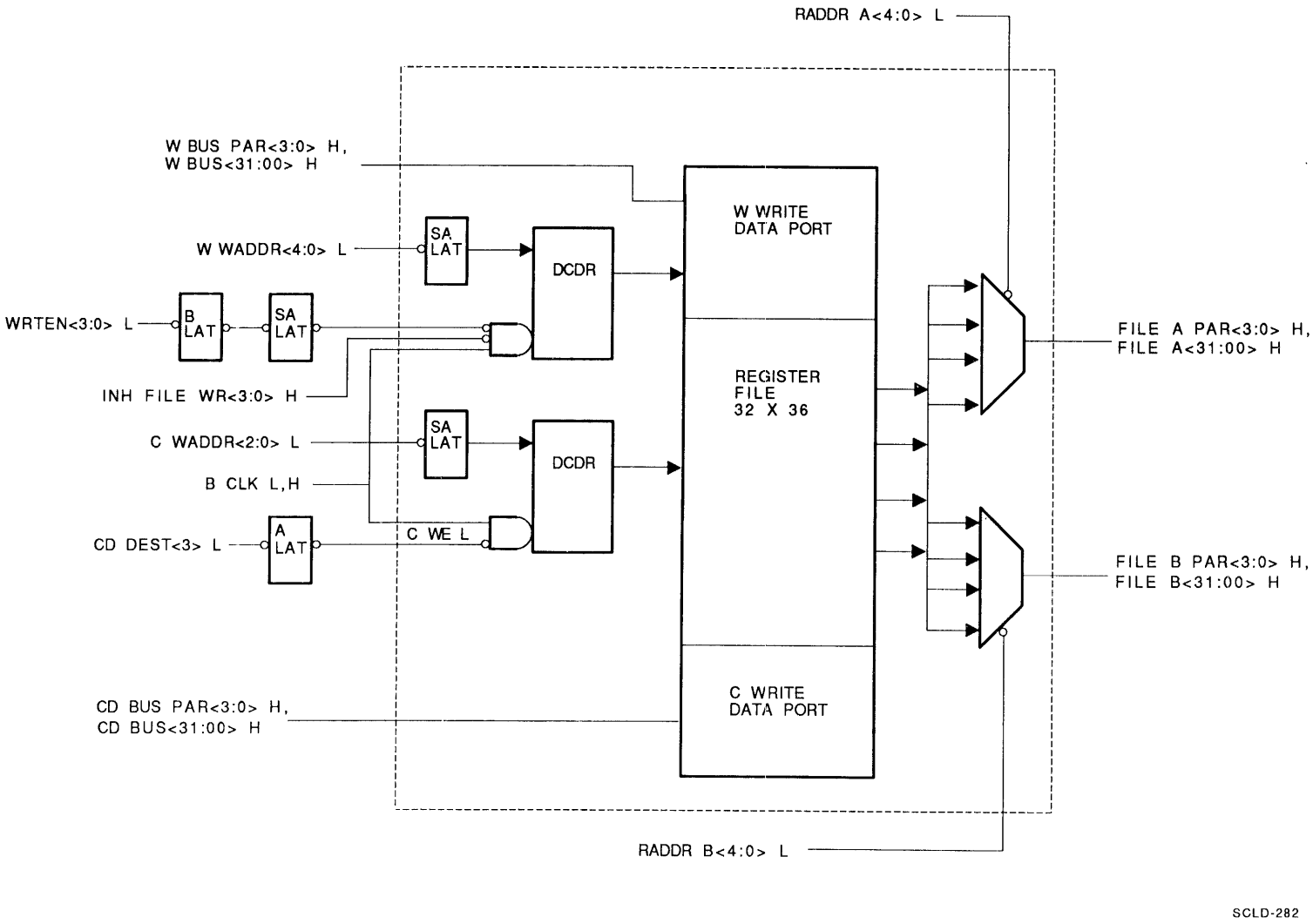


Figure 2-4 Register File (RGF) Block Diagram

2.2.2.2 Memory Data (MD) Registers -- All MD registers are set valid in the first microinstruction of a new macroinstruction. However, an individual MD register may be invalidated by the first microinstruction.

MD registers may be read and written as scratch registers unless invalidated.

An invalid MD register expects that cache will eventually perform a write to the location and the register cannot be used until the cache write is no longer pending. This is determined in one of two ways:

- Cache has written its result in the MD register.
- A trap occurs where cache informs the microcode that the requested data will not be written (no writes are pending to the MD register).

2.2.2.3 Traps and Stalls -- A trap takes priority over a stall. In the first case above, the MD register becomes valid and can be used. In the second case, the MD will not be valid until the beginning of a new macroinstruction. In either case, a new macroinstruction cannot be started until there are no cache writes pending to the MD registers.

An RGF write normally starts at T9. However, the write enable signals must be delayed. The write enable for the cache side is delayed one-half cycle by one latch. The write enable for the W bus side is delayed by one cycle by two latches.

Table 2-7 Register File (RGF) Address Allocation

Location	Register Name
0-D	General-Purpose Registers (GPRs)
E	GPR Stack Pointer
F	Memory Write Data Temp
10-12	Memory Management Temps
13-17	Scratchpad Registers
18-1E	Memory Data (MD) Registers
1F	MD Memory Mangagement Register

Table 2-8 Register File (RGF) Signal Descriptions

Signal Name	Description	Valid
A-Port Data Read Signals		
File A Data (FILE A<31:00> H)	Read data from the RGF A-port.	T5 to T7
File A Parity (FILE A PAR<3:0> H)	Byte parity bits for the RGF A-port read data.	T5 to T7
Read Address A (RADDR A<4:0> L)	ASIDE READ ADDR<4:0> L from the microsequencer.	T4 to T6
B-Port Data Read Signals		
File B Data (FILE B<31:00> H)	Read data from the RGF B-port.	T5 to T7
File B Parity (FILE B PAR<3:0> H)	Byte parity bits for the RGF B-port read data.	T5 to T7
Read Address B (RADDR B<4:0> L)	Concatenation of E_BPORT<4> from the microword and BSIDE READ ADDR<3:0> from the microsequencer.	T4 to T6
W Bus Data Write Signals		
W Bus Data (W BUS<31:00> H)	Write data from the PAR logic on the same slice module.	T8 to T10
W Bus Parity (W BUS PAR<3:0> H)	W bus byte parity bits from the same slice module.	T8 to T10
W Bus Address (W WADDR<4:0> L)	FILE WRITE ADDR<4:0> from the decoder (DEC) module.	T7 to T9
W Bus Write Enable (WRTEN<3:0> L)	E_WRTEN<3:0> from the microword.	T6 to T8

Table 2-8 Register File (RGF) Signal Descriptions (Cont)

Signal Name	Description	Valid
Inhibit File Writes (INH FILE WR<3:0> H)	BLOCK WRITES<5:2> from the microsequencer inhibit byte writes to the RGF during a trap shadow. Valid from T10 to T16 of the trapping instruction. RGF writes occur at T9, therefore: - T11 of the trapping instruction blocks T9 of the first micro-instruction in the trap shadow - T13 blocks T9 of the second microinstruction - T15 blocks T9 of the third microinstruction	T10 to T16
CD Bus Data Write Signals		
CD Bus Data (CD BUS<31:00> H)	IB CD MD<31:00> from the cache data path (CDP) logic.	T8 to T10
CD Bus Parity (CD BUS PAR<3:0> H)	IB CD MD PAR<3:0> from the cache data path (CDP) logic.	T8 to T10
Cache Write Address (C WADDR<2:0> L)	CACHE DATA DEST<2:0> on the same slice module.	T7 to T9
Cache Write Enable (C WE L)	CACHE DATA DEST<3> on the same slice module.	T8 to T10

2.2.3 Slow Data File (SDF)

The microcode uses the SDF for the following types of register functions:

- Page table base registers
- Error registers
- Stack pointers
- Masks
- Constants
- Additional scratchpad locations

Figure 2-5 is a block diagram of the slow data file (SDF) logic, which provides 256 32-bit registers with byte parity.

Table 2-9 describes the SDF read and write signals.

2.2.3.1 **Writes** -- On a write, the SDF address latches at T5 and is delayed for two cycles through two sets of B CLK and STALLED A CLK latches. SDWRTEN is asserted at T7, and data from the main ALU or SHR is received on the W bus at T8.

2.2.3.2 **Reads** -- An SDF read occurs at T5 so the address is gated directly to the SDF RAM.

Reads occur from T5 (B CLK) to T6 (A CLK) and are inhibited during a SDF write, which occurs during a modified A CLK at T10. For a write followed by a read to the same location, the microcode performs the write at T10, which then corresponds to T8 of the next microinstruction, T6 of the second, and T4 of the third. Therefore, the read occurs at T5 of the reading microinstruction, three cycles after the write.

2.2.3.3 **Stalls and Traps** -- The A latches stall when the stall signal is asserted.

During a trap, SDF writes are disabled until the first microinstruction of the trap routine can begin, 3 microcycles later.

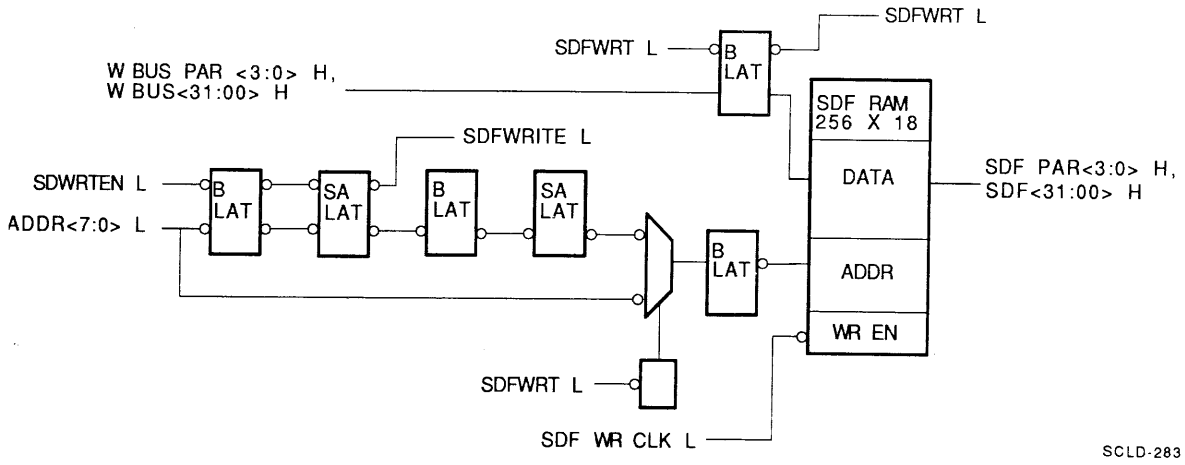


Figure 2-5 Slow Data File (SDF) Block Diagram

Table 2-9 Slow Data File (SDF)
Signal Descriptions

Signal Name	Description	Valid
Output Signals		
SDF Data (SDF<31:00> H)	Read data from the SDF RAM is multiplexed and asserted to the B-side of the main ALU as FB SDF<31:00> H.	T5 to T7
SDF Parity (SDF PAR<3:0> H)	Byte parity from the SDF RAM is asserted to the PAR parity check inputs.	T5 to T7
SDF Write Signals	The SDF write enable signal (SDWRTEN L) from the IBox is delayed one cycle and asserted to the PC trap logic as SDFWRITE L. There, it is gated with an extended trap signal and returned to the SDF logic as SDFWRT L. Under control of the PC trap logic, SDFWRT L selects either the read or write address and is valid from T9 to T11. It is delayed by a half cycle and sent to the clock distribution logic as SDFWR L where it is gated with A CLK to form the SDF write clock SDF WR CLK L, which is valid from T10 to T12.	T8 to T10

Table 2-9 Slow Data File (SDF)
Signal Descriptions (Cont)

Signal Name	Description	Valid
Input Signals		
Read/Write Address (ADDR<7:0> L)	ADDR<7:0> L is a concatenation of E_BPORT<7:4> from the microword and BSIDE READ ADDR<3:0> from the decoder (DEC) module.	T4 to T6
Write Bus (W BUS<31:00> H)	Asserts latched ALU output or parity error register data to the SDF RAM write inputs.	T8 to T10
Write Bus Parity (W BUS PAR<1:0> H)	W bus data source parity from the PAR logic outputs to the SDF RAM write inputs.	T8 to T10
SDF Write Enable (SD WRT EN L)	Asserted by E_SDWRTEN of the microword. Selects the read/write address and is eventually gated with A CLK to form the SDF write clock (SDF WR CLK L, see Output Signal description).	T6 to T8

2.2.4 Program Counter (PC) Subsystem

Figure 2-6 is a block diagram of the program counter (PC) subsystem, which contains the following main logical elements:

- PC VA FA Multiplexer
- Virtual Address (VA) File
- Trap Shadow Logic
- Program Counter (PC), Backup PC, and Trap PC

The slice section contains four PC MCAs, each of which operates on one byte of data or address information. The SLC1 module contains two PC2 type MCAs. The SLC0 module contains one PC2 and one PC1 type MCA. The PC1 MCA operates on byte 0 (PC bits <07:00>).

Table 2-10 describes the PC subsystem input and output signals.

2.2.4.1 PC VA FA Multiplexer -- The PC VA FA multiplexer connects to the A-side input of the main ALU. Its inputs are selected by the VA WRT and APORT<7:6> bits of the microword as shown in Table 2-11.

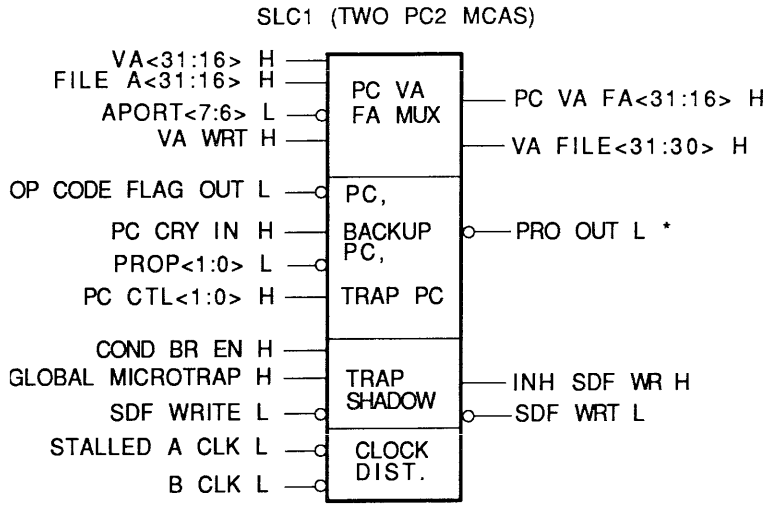
2.2.4.2 Virtual Address (VA) File -- The VA file is loaded from the main ALU and is a duplicate of the VA register on the cache control sequencer (CCS) module. The VA file holds the virtual address for a data transfer request to cache memory. If the request results in a trap (a TB miss, for example), the VA file contents are used for the repair and retry of the request.

Traps

When a trap occurs, INH VAWR from the trap logic causes the VA file to hold its current address until the trap routine is ready to execute. The trap logic also inhibits the VA file bypass during the first microinstruction of a trap routine, following a trap shadow.

In the EBox timing, micro-operations can overlap where the VA file may be physically read first and then written. If a VA read cycle is requested immediately after a write cycle, the VAWRT signal selects the VA file write input data through the VA bypass bus.

2.2.4.3 Trap Shadow Logic -- The trap shadow logic consists of concatenated latches that cause a delay of 2 1/2 cycles when a trap occurs. When this logic receives the global microtrap signal from the IBox, it produces a series of signals on succeeding cycles that are used during the trap shadow, and on the first microinstruction of the trap routine.



*NOTE: THIS SIGNAL IS PROP<2> FROM BYTE 2 TO BYTE 3.

SCLD-284

Figure 2-6 Program Counter (PC) Subsystem Block Diagram (Sheet 1 of 2)

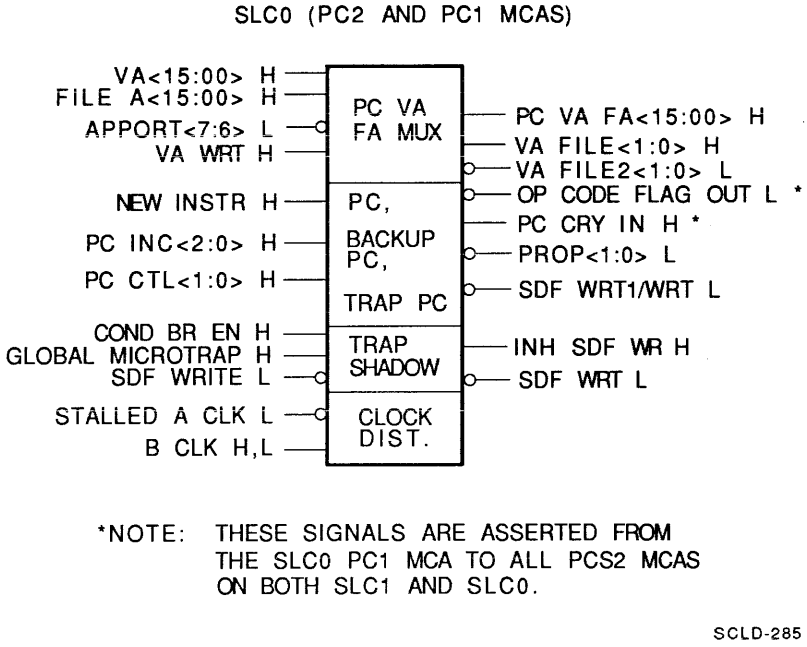


Figure 2-6 Program Counter (PC) Subsystem Block Diagram (Sheet 2 of 2)

2.2.4.4 Program Counter (PC) -- The PC is incremented with a value of 0 to 6 as determined by the decoder (DEC) module and is latched at T4 of the currently executing macroinstruction.

During operation, the PC makes use of the following logical functions:

- PC Incrementer
- Backup PC
- Trap PC
- PC Steering Multiplexer

PC Incrementer

The PC is incremented by a value of 0 to 6 by adding the increment value to the three lowest-order bits. The resulting carry-out, with any preceding byte boundary propagates, determines whether PC<07:04>, PC<15:08>, PC<23:16>, and/or PC<31:24>, are to be incremented by 1.

Backup PC

When a macroexception is generated, the PC is restored from the backup PC to point to the offending op code.

The backup PC points to the op code before the trap PC. The backup PC is then loaded during T11 to T13 of the first microinstruction of a new macroinstruction.

The op code flag (NEW INSTR) signals the beginning of a new macroinstruction. Set during the first cycle of the current macroinstruction, it enables writing to the backup PC. Clearing the op code flag is higher in priority than setting the op code bit so that the correct PC value is not lost if the last microinstruction of a macroinstruction causes a trap.

Trap PC

If a microtrap occurs, the PC is backed up to its correct state from the trap PC (just before the beginning of the microcycle following the one that caused the trap). This is the correct value for the return to the microinstruction following the one that had the problem.

When a trap occurs, the hardware forces the PC to be loaded with the trap PC value during the second microinstruction in the trap shadow. It then reloads the PC with itself during the third and final microinstruction in the trap shadow. The hardware and microcode prevent the PC from being changed so that the return from trap routine value is not changed.

PC Steering Multiplexer

The following signals are loaded to the PC through the steering multiplexer:

- VA Bus
- VA File
- PC
- Backup PC
- Trap PC

Table 2-12 summarizes the five bits that control the multiplexer in the following priority order:

1. LD PC INC and LD TRAP PC from the trap logic (internal to the PC MCAs) - Highest

2. COND BR EN from the decoder (DEC) module

3. E_PCCTRL<1:0> from the microword - Lowest

A microinstruction that loads a new value into the PC must not produce a trap from which there can be a return. (The trap PC would not hold the correct value.)

To improve performance on conditional branches, the IBox assumes that the branch will succeed and sets up for the new I-stream by writing the new address to the VA file. Meanwhile, the microcode assumes that the branch will fail and proceeds with the microinstructions in the current sequence. If the branch succeeds, the IBox issues the Trap signal, blocking the results of the EBox's assumed failure to branch logic. It also issues a Conditional Branch Successful signal that causes the PC to be loaded from the VA file instead of the Trap PC.

Table 2-10 Program Counter (PC) Subsystem Signal Descriptions

Signal Name	Description	Valid
PC Virtual Address/File A Multiplexer Input/Output Signals		
PC VA FA Out (PC VA FA<31:00> H)	PC VA FA multiplexer outputs to the A-side of the main ALU.	T5 to T7
Virtual Address In (VA<31:00> H)	Virtual address from the main ALU VA bus. Valid from T7 to T9 of the microinstruction that generates it.	T7 to T9
File A Data In (FILE A<31:00> H)	Register file data from the File A port.	T5 to T7
A-Port Control (APORT<7:6> L)	I_APORT<7:6> from the microword select the PC, VA, or FA for the main ALU A-side inputs (see Table 2-11).	T4 to T6
VA Write (VA WRT H)	E_VAWRT from the microword selects either the VA file or VA file bypass when selected by APORT<7:6> and enables writes to the VA file (see Table 2-11).	T6 to T8
Virtual Address File Output Signals		
VA File MSBs (VA FILE<31:30> H)	The two most significant bits (MSBs) are used by the sequencer (SEQ) module microbranch logic.	T9 to T11
VA File LSBs (VA FILE<1:0> H)	The two least significant bits (LSBs) are used by the IBox for byte alignment after an IB flush.	T8 to T10
VA File 2 LSBs (VA FILE2<1:0> L)	The two least significant bits (LSBs) are also used by the shifter (SHR) module for read/write data byte alignment.	T9 to T11

Table 2-10 Program Counter (PC) Subsystem Signal Descriptions (Cont)

Signal Name	Description	Valid
Program Counter (PC) Input/Ouput Signals		
New Instruction (NEW INSTR H)	OP FLAG H from the decoder (DEC) module to the PC1 MCA on SLC0. Flags the op code location for the next instruction decode.	T4 to T6
Opcode Flag (OPCODE FLAG OUT L)	Issued from the PC1 MCA on SLC0. Indicates a new instruction to all PC2 MCAs and enables writes to the backup PC.	T9 to T11
PC Carry In (PC CRY IN H)	CARRY OUT from the low order 3-bit adder in the PC1 MCA to all PC2 MCAs. Used with PRO OUT (PROP<2>) and PROP<1:0> to increment the next highest order PC byte by 1.	T4 to T6
PC Increment (PC INC<2:0> H)	From the IBox only to the PC1 MCA. Selects the increment value (0 to 6) for the low order 3-bit PC1 adder.	T3 to T5
Propogate (PROP<2:0> L)	A PROP bit is asserted by a PC2 MCA when all eight PC bits of the byte are 1.	T4 to T6
PC Control (PC CTL<1:0> H)	E_PCCTL<1:0> from the microword select the PC input source. Used with internal PC MCA signals (see Table 2-12).	T2 to T4
Trap Shadow Input/Output Signals		
Conditional Branch Enable (COND BR EN H)	From the IBox. Used with the GLOBAL MICROTRAP signal to indicate that a conditional branch is to be taken. The new PC value is loaded from the VA file instead of the trap PC.	T10 to T12

Table 2-10 Program Counter (PC) Subsystem Signal Descriptions (Cont)

Signal Name	Description	Valid
Global Microtrap (GLOBAL MICROTRAP H)	TRAP SIGNAL H from the IBox is valid from T10 to T12 of the microinstruction that caused the trap. Causes the following PC MCA functions to occur during a trap shadow: 1. The PC is loaded with the contents of the trap PC during the second microinstruction in the trap shadow. Valid from T11 to T13. 2. VA file bypass is inhibited so that the first microinstruction of a trap routine can access the VA file contents. Valid from T12 to T14. 3. The PC is loaded with an incremented value. Valid from T13 to T15. A write to the VA file and backup PC are also inhibited during the trapping microinstruction and during the trap shadow. Valid from T10 to T15.	T10 to T12
Inhibit SDF Write (INH SDF WR H)	Asserted to the PAR MCAs during a trap shadow to preserve the pretrap values of the parity error registers, carry save latch, and FP carry selection latch. Valid from T11 to T16 of the microinstruction that caused the trap.	T11 to T16
Write Control Signals to the Slow Data File (SDF)		
SDF Write In (SDF WRITE L)	SD WRT EN from the IBox goes to the SDF logic for each slice where it is delayed 1 cycle. The output, SDF WRITE, is asserted to the trap logic of both PC MCAs (on the same slice) and is valid from T8 to T10.	

Table 2-10 Program Counter (PC) Subsystem Signal Descriptions (Cont)

Signal Name	Description	Valid
SDF Write Out (SDF WRT L)	In each PC MCA, SDF WRITE is gated with the negated state of GLOBAL MICROTRAP to obtain the SDF WRT output signal. In the SDF logic for each slice, SDF WRT selects either the fast or slow address for a write or read of the SDF RAM. Valid from T8 to T10.	
SDF Write	From the SDF logic, SDF WRT is (SDF WR L) asserted as SDF WR to the slice clock logic. There it is ANDed with A CLK to form the write clock (SDF WR CLK L) for the SDF RAM. Valid from T8 to T10. With GLOBAL MICROTRAP asserted, SDF WRT and SDF WR are inhibited for three cycles, blocking SDF writes during a trapping microinstruction and trap shadow. Valid from T10 to T15.	

Table 2-11 E_VAWRT and I_APORT<7:6> Control of PC VA FA Multiplexer Input Selection

E_VAWRT	I_APORT <7 6>	These Inputs are Gated to the Outputs
-	0 0	PC
0	0 1	VA File
1	0 1	VA File Bypass
-	1 0	File A Data

Table 2-12 PC Multiplexer Input Selection PC Multiplexer Select Signals

COND BR EN	PC CTL <1 0>	LD TRAP PC	LD PC INC	Selected Input
-	0 0	0	0	Incremented PC
-	0 1	0	0	VA Bus
-	1 0	0	0	Backup PC
-	- -	0	1	Incremented PC
0	- -	1	-	Trap PC
1	- -	1	-	VA File

2.2.5 Cache Data Path (CDP)

The cache memory section of the CBox consists of the following three main logical groups:

1. Cache Tag Store (CTS)
2. Cache Tag Valids (CTV)
3. Cache Data Path (CDP)

While logically part of the cache memory logic, the cache data path (CDP) group is physically located on the EBox slice modules. For further information on cache memory functions, refer to the CBox section of this manual.

Figure 2-7 is a block diagram of the CDP logic which contains the following elements:

- Cache Data Buffer (CDBF) MCA logic
- Cache Data Store (CDS) RAM logic

2.2.5.1 Cache Data Buffer (CDBF) -- Each slice module contains two cache data buffer (CDBF) MCAs (Figure 2-7, Sheet 1), each of which passes one byte of memory or CPU data and parity to or from the CDS RAM.

Table 2-13 describes the CDBF input and output signals.

The CDBF MCAs make up the following data path registers:

• Write Data Buffer	One longword and byte parity
• Delay-Write Buffer	One longword and byte parity
• Write Buffer/Output Buffer	One octaword (four longwords, each) with byte parity

Write Data Buffer

The write data buffer consists of a B-latch that drives the write data (WR DATA) bus with data received from one of the following sources:

- The MD bus from the IBox
- The W bus from the slice PAR logic
- The W bus, delayed one cycle by the delay-write buffer

Delay-Write Buffer

The delay-write buffer receives write bus (W BUS) data from the slice PAR logic, passing it to the write data buffer and write buffer.

Write Buffer/Output Buffer

Four 36-bit buffers (data and parity) act as a small cache for writes to the same octaword. The write buffer (all four longwords) is parallel-loaded to the output buffer, which drives the MD BUS lines.

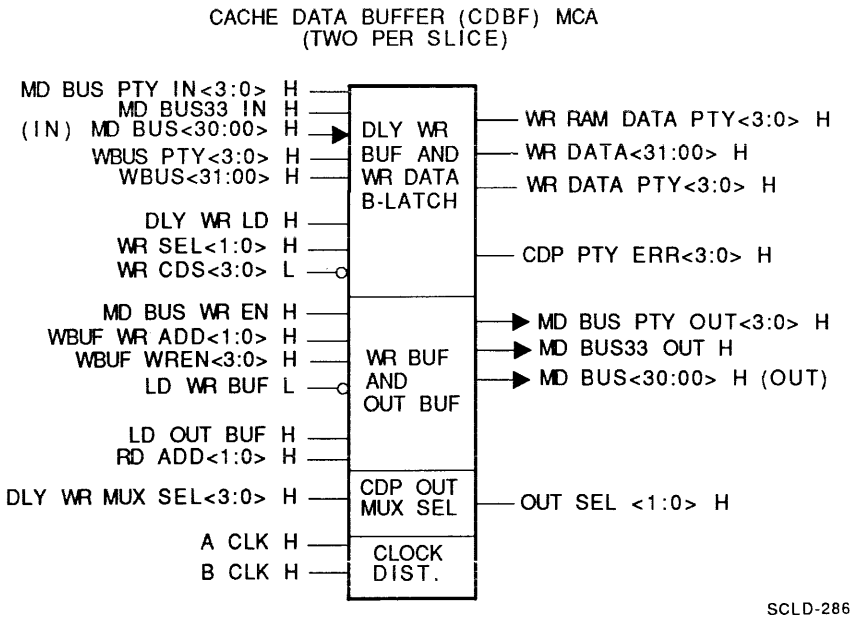


Figure 2-7 Cache Data Path (CDP) Block Diagram (Sheet 1 of 2)

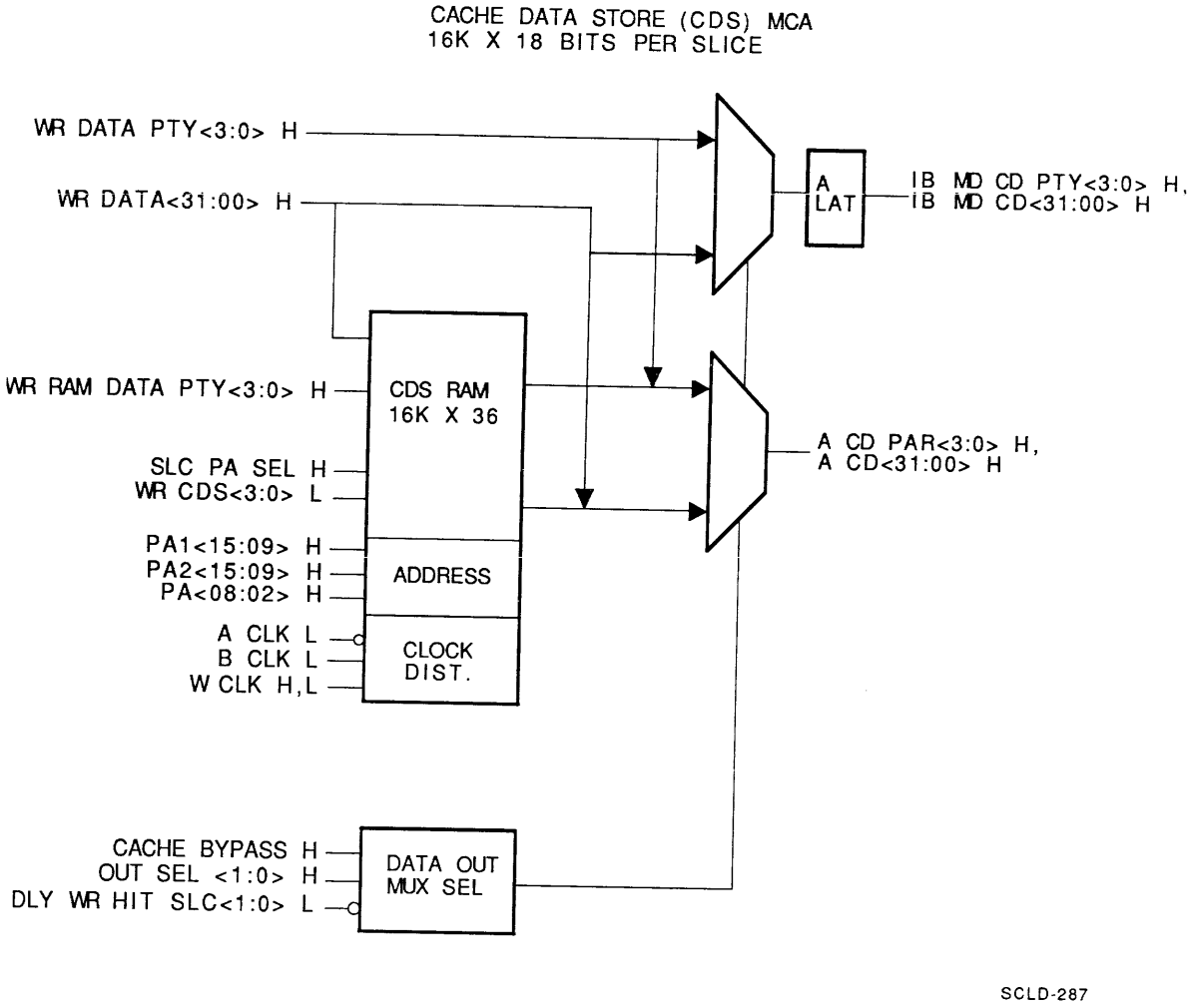


Figure 2-7 Cache Data Path (CDP) Block Diagram (Sheet 2 of 2)

2.2.5.2 Cache Data Store (CDS) -- Each slice module contains one 16K X 18-bit CDS RAM set (Figure 2-7, Sheet 2) that stores and/or passes two bytes of memory read/write data and byte parity.

Table 2-14 describes the CDS RAM logic signals. The logic consists of the following elements:

- CDS RAM memory, 32K X 36 bits
- Two sets of data/bypass multiplexers, one of which uses A-latched outputs.

Table 2-15 shows the control signal functions for the cache data output multiplexers.

Table 2-13 Cache Data Buffer (CDBF) Signal Description

Signal Name	Description
Write Data B-Latch Outputs	
Write Data (WR DATA<31:00> H)	Carries the write data B-latch contents to the CDS RAM and to the EBox and IBox. The B-latch is loaded from one of the following data sources (depending on the states of WR SEL<1:0>): - The received W BUS value - The delay-write buffer - The received MD BUS value from the CBox
Write RAM Data Parity (WR RAM DATA PTY<3:0> H)	Carries the write data B-latch byte parity bits to the CDS RAM. These B-latch bits are loaded with the byte parity value from the selected data source.
Write Data Parity (WR DATA PTY<3:0> H)	With WR CDS<3:0> negated, byte parity errors are inhibited from CDP PTY ERR<3:0>. Byte parity is generated on the B-latch contents and asserted on WR DATA PTY<3:0> to the EBox and IBox. With any WR CDS<3:0> bits asserted, the selected write data B-latch byte parity bits are asserted on WR DATA PTY<3:0>. Byte parity errors are reported on CDP PTY ERR<3:0>.
CDP Parity Error (CDP PTY ERR<3:0> H)	Byte parity is generated on the B-latch contents and compared with the byte parity bits from the data source. With any WR CDS<3:0> bits asserted, selected byte parity errors are reported on CDP PTY ERR<3:0>.

Table 2-13 Cache Data Buffer (CDBF) Signal Description (Cont)

Signal Name	Description
Inputs to the Delay-Write Buffer and Write Data B-Latch	
Memory Data Bus 33 (MD BUS33 IN H)	Asserted from the CBox to bit <31> of the write data B-latch. From there, it is written to the CDS RAM as WR DATA<31>.
Memory Data Bus (In) (MD BUS<30:00> H)	The bidirectional lines pass memory data from the CBox to the write data B-latch.
Memory Data Bus Parity (MD BUS PTY IN<3:0> H)	Memory data parity from the CBox is passed to the write data B-latch.
Write Bus Data (W BUS<31:00> H)	ALU data asserted by the PAR logic is received on the W bus and is asserted to the write data B-latch and the delay-write buffer. The delay-write buffer is asserted to the write data B-latch and the write buffer.
Write Bus Parity (W BUS PTY<3:0> H)	Byte parity generated by the PAR logic (on the ALU data) is received on the W bus. It is asserted to the write data B-latch and the delay-write buffer along with the write data. The delay-write buffer parity bits are asserted to the write data B-latch and the write buffer along with the write data.
Delay Write Load (DLY WR LD H)	Asserted, causes received W BUS data to be clocked to the delay-write buffer. It is written on the following cycle to the write data B-latch or the write buffer.

Table 2-13 Cache Data Buffer (CDBF) Signal Description (Cont)

Signal Name	Description
Write Select (WR SEL<1:0> H)	Select one of the following data and parity input sources for the write data B-latch: WR SEL <1 0> Data Source 0 0 Delay-write buffer - 1 MD bus 1 0 W bus From there, the data is written to CDS RAM or passed to the IBox.
Write Cache Data Store (WR CDS<3:0> L)	Control parity checking on writes to CDS RAM. Negated, byte parity generated on the B-latch contents is asserted on WR DATA PTY<3:0>. Byte parity errors are inhibited from CDP PTY ERR<3:0>. Asserted, the write data B-latch byte parity bits are asserted on WR DATA PTY<3:0>. Byte parity errors are reported on CDP PTY ERR<3:0>.
Output Buffer Outputs	
Memory Data Bus 33 (MD BUS33 OUT H)	Asserted from bit <31> of the output buffer to the CBox.
Memory Data Bus (Out) (MD BUS<30:00> H)	The bidirectional lines pass data from the output buffer to the CBox.
Memory Data Bus Parity (MD BUS PTY OUT<3:0> H)	Byte parity from the output buffer is passed to the CBox.

Table 2-13 Cache Data Buffer (CDBF) Signal Description (Cont)

Signal Name	Description
Write Buffer and Output Buffer Inputs	
MD Bus Write Enable (MD BUS WR EN H)	Asserted, enables the MD bus drivers for the output buffer. The data and parity bits are passed on the MD BUS to the NMI data paths in the CBox.
Write Buffer Write Address (WBUF WR ADD<1:0> H)	Selects one of four 32-bit write buffer registers for a write operation. (Selects the longword within an octaword as specified by PA<3:2>.)
Write Buffer Write Enable (WBUF WREN<3:0> H)	Write buffer byte mask. Enables valid bytes to be written from the delay-write buffer to the selected write buffer.
Load Write Buffer (LD WR BUF L)	Writes selected write buffer bytes from the delay-write buffer during A CLK.
Load Output Buffer (LD OUT BUF H)	Causes all four write buffers to be written to the output buffers one A CLK after a write to the selected write buffer. Deasserted, causes output buffer data to be held for writing to the NMI through the NMI data paths in the CBox.
Read Address (RD ADD<1:0> H)	Selects one of four 32-bit output buffers for assertion on the MD BUS to the NMI data paths in the CBox.
CDP Output Multiplexer Control	
Delay Write Multiplexer Select (DLY WR MUX SEL<3:0> H)	Passed as the signals Output Select A for slice 1 (OUT SEL A<1:0>) and Output Select B for slice 0 (OUT SEL B<1:0>) to the CDS RAM output multiplexer. These bits are driven by the cache write valids. Used with DLY WR HIT SLC to make delay write hit/read cycles work in one cycle (see Table 2-14).

Table 2-14 Cache Data Store (CDS) Signal Description

Signal Name	Description
CDS Multiplexer Output Signals	
Cache Data Bus (IB MD CD<31:00> H)	The CDS RAM data outputs are output through an A-latch multiplexer to the IBox and the register file. Write data from the WR DATA bus is asserted for a bypass when a write is immediately followed by a read to the same location.
Cache Data Bus Parity (IB MD CD PTY<31:00> H)	The CDS RAM parity outputs are output through an A-latch multiplexer to the IBox and the register file along with the data. Parity from the WR DATA bus is asserted for a bypass when a write is immediately followed by a read to the same location.
Cache Data Bus A (A CD<31:00> H)	The CDS RAM data outputs are multiplexed and passed unlatched to the A-side and B-side of the main ALU first half (ALF). Write data from the WR DATA bus is asserted for a bypass when a write is immediately followed by a read to the same location.
Cache Data Bus A Parity (A CD<31:00> H)	The CDS RAM parity outputs are multiplexed and passed unlatched to the A-side and B-side of the main ALU first half (ALF). Parity from the WR DATA bus is asserted for a bypass when a write is immediately followed by a read to the same location.

Table 2-14 Cache Data Store (CDS) Signal Description (Cont)

Signal Name	Description
CDS RAM and RAM Logic Input Signals	
Write Data (WR DATA<31:00> H)	Passes the write data B-latch contents from the CDBF to the CDS RAM and the output multiplexers (see Table 2-13).
Write RAM Data Parity (WR RAM DATA PTY<3:0> H)	Passes the write data B-latch byte parity bits from the CDBF to the CDS RAM.
Write Data Parity (WR DATA PTY<3:0> H)	Passes the write data B-latch byte parity bits, or the byte parity generated on the write data B-latch contents, from the CDBF to the output multiplexers. (See the WR CDS<3:0> description, Table 2-13.)
Slice Physical Address Select (SLC PA SEL H)	Negated, the PA2<15:09> PABH address is selected for addressing the CDS RAM. Asserted, the PA1<15:09> and PA<08:02> fast TB addresses are selected for addressing the CDS RAM.
Write CDS (WR CDS<3:0> L)	Used with write clock (W CLK) to produce the byte write pulses for CDS RAM.
Physical Address 1 (PA1<15:09> H)	Physical address bits <15:09> from the physical address high buffer (PAHB) in the translation buffer (TB).
Physical Address 2 (PA2<15:09> H)	Physical address bits <15:09> from the fast address register.
Physical Address (PA<08:02> H)	Bits <08:02> of the virtual address latch from the fast address register.

Table 2-14 Cache Data Store (CDS) Signal Description (Cont)

Signal Name	Description
Output Multiplexer Control	
CACHE BYPASS H	Asserted, enables WR DATA bus inputs directly to the output multiplexers. Passes CDS RAM write data from the W BUS to the IBox.
Delay Write Hit Slice (DLY WR HIT SLC L)	Asserted for the slice on a match between delay-write address bits <15:02> and the corresponding address of the current cycle. Used with DLY WR MUX SEL to assert read data from the delay-write buffer instead of CDS RAM when a write is followed by a read to the same location (cache is not yet updated). Enables the WR DATA bus inputs directly to the output multiplexers (valid only when OUT SEL A,B<1:0> equals 01).
Output Select A/B (OUT SEL <1:0> H)	Selects WR DATA bus or CDS RAM data for output from the CDS output multiplexers. OUT SEL<1> controls the upper byte and OUT SEL<0> controls the lower byte for each slice (see Table 2-15).

Table 2-15 CDS Output Multiplexer Control for Each Slice

CACHE BYPASS	DLY WRT HIT SLC	OUT SEL <1>	<0>	Selected Input Asserted from the Multiplexer Output Bytes Upper Mux Byte	Lower Mux Byte
0	0	-	-	CDS RAM Data	CDS RAM Data
0	1	0	0	CDS RAM Data	CDS RAM Data
0	1	0	1	CDS RAM Data	WR DATA Bus
0	1	1	0	WR DATA Bus	CDS RAM Data
0	1	1	1	WR DATA Bus	WR DATA Bus
1	-	-	-	WR DATA Bus	WR DATA Bus

2.2.6 Main Arithmetic Logic Unit (Main ALU)

Figure 2-8 shows the main ALU logic, which consists of the following two types of MCAs:

1. ALU First Half (ALF) — Four ALF MCAs per slice. Each process 4 bits (1 nibble) of data, passing a partial result to the ALS along with carry propagate/-generate information.

2. ALU Second Half (ALS) — Two ALS MCAs per slice. Each process 1 byte of data, completing an arithmetic function and outputting the final result.

Table 2-16 describes the ALF input/output and control signals.

Table 2-17 describes the ALS input/output and control signals.

2.2.6.1 ALU First Half (ALF) -- The ALF stage processes the first part of an add (ADD) or subtract (SUB) operation. It passes a partial result to the ALS along with the carry generate/propagate information the ALS uses to complete the operation. Each pair of ALF chips passes nibble parity to an ALS chip, which combines the bits to form byte parity.

The following buses apply data to the A-side and/or the B-side of the ALF, where the data is internally latched and selected:

PC VA FA Bus — RGF File A data, virtual address, or program counter information is preselected in the PC subsystem and applied to the A-side.

A CD Bus — Data coming from the W bus or MD bus for writing to cache, bypasses the cache data store (CDS) and is applied to the A-side and B-side.

BP Bus — Bypass bus data is applied to the A-side and B-side.

IB Data Bus — Instruction Buffer information is applied to the A-side and B-side.

FB SDF Bus — RGF File B or SDF data is preselected by a multiplexer and applied to the B-side.

A-Side Input Select (ASEL<1:0>)

Each pair of the port control bits (PORT CNTL<11:10><09:08><03:02><01:00> from the bus watcher MCA on the DEC module) selects one 4-bit field on the A-side of an ALF MCA as shown in Table 2-18.

B-Side Input Select (BSEL<1:0>)

Each pair of the port control bits (PORT CNTL<15:14><13:12><07:06><05:04> from the bus watcher MCA on the DEC module) selects one 4-bit field on the B-side of an ALF MCA as shown in Table 2-19. BSEL<2> (EBRDMODE from the microword) selects data from either the RGF File B or SDF lines.

Stalls

All of the main ALU A latches may be stalled except for the A CD bus input latches. The bus watcher MCA on the DEC module asserts the cache data ready (CD READY) and memory data register valid (MD VALID) signals while the cache data valid (CD VALID) signal is asserted by cache. As shown in Table 2-20, these signals are encoded into the KEEPGOING signal, which determines whether the CD latches stall for either side while the other side is waiting for data. They also determine when the stall will be released.

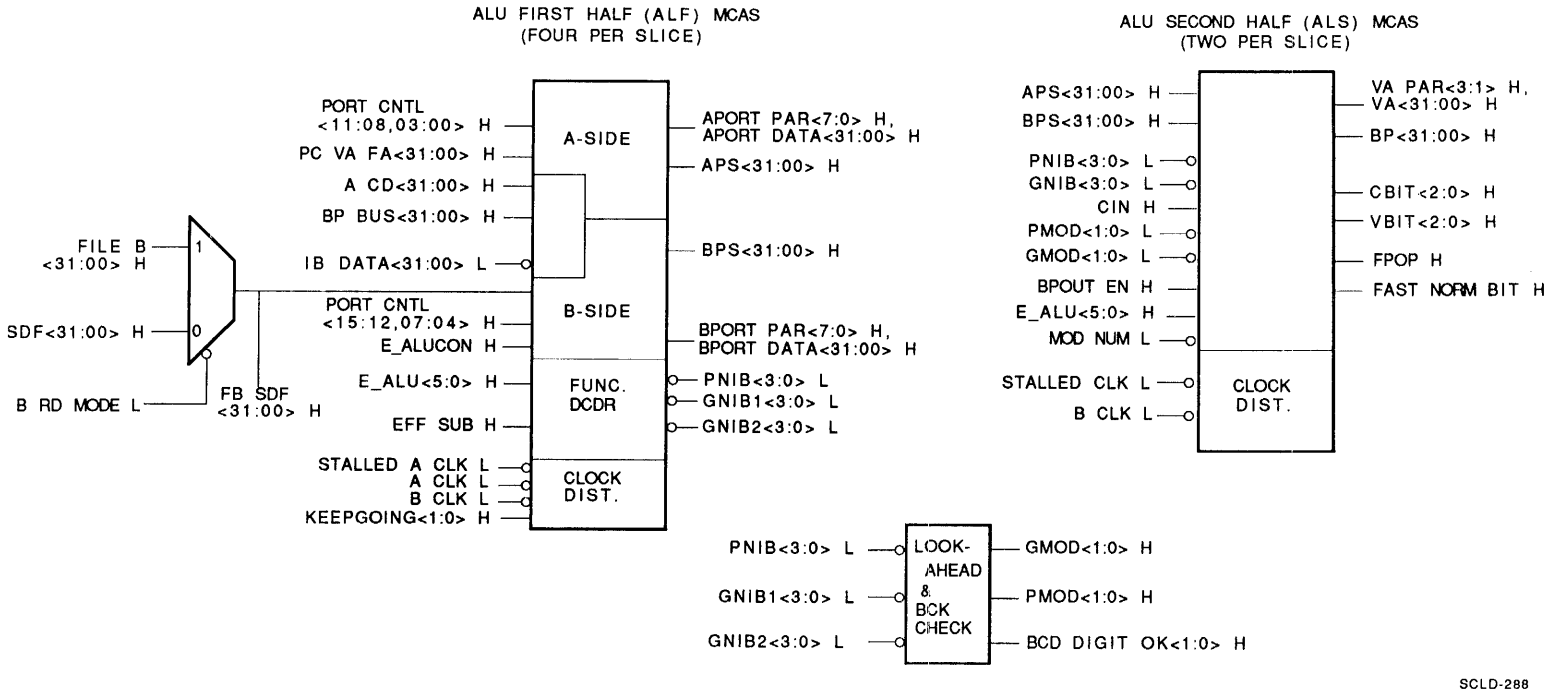


Figure 2-8 Main Arithmetic Logic Unit (Main ALU) Block Diagram

2.2.6.2 Main ALU Functions -- The main ALU logic performs addition (ADD) and subtraction (SUB) of the following types of numbers:

- Integer
- Floating-Point
- Packed Decimal

ALU Function Control

Table 2-21 lists the ALU functions that take place under control of EALU<5:0> from the microword and are used to perform the following kinds of operations:

- Performs a floating-point mask of A-side data.
- Generates Integer and Floating-Point carries.
- Generates Overflow (V) and Carry (C) condition code bits on byte, word, and longword boundaries (ignored by the microcode if not relevant).
- Generates nibble parity on A-side and B-side data, which is checked against the source input data parity in the PAR.
- Outputs the ALU data or result on the VA bus and BP bus (when enabled by BPOUT EN).

Table 2-16 ALU First Half (ALF) Signal Descriptions

Signal Name	Description	Valid
Data Output Signals - A-Side		
A Partial Sum (APS<31:00> H)	Passes a partial A-side sum to the ALS on the same slice.	T6 to T8
A Port Data (APORT DATA<31:00> H)	Passes the selected A-side input data to the shifter (SHR) module.	T6 to T8
A Port Parity (APORT PAR<7:0> H)	Passes nibble parity generated on the selected A-side input data to the PAR on the same slice.	T6 to T8
Data Output Signals - B-Side		
B Partial Sum (BPS<31:00> H)	Passes a partial B-side sum to the ALS on the same slice.	T6 to T8
B Port Data (BPORT DATA<31:00> H)	Passes the selected B-side input data to the shifter (SHR) module.	T6 to T8
B Port Parity (BPORT PAR<7:0> H)	Passes nibble parity generated on the selected B-side input data to the PAR on the same slice.	T6 to T8
Data Input Signals - A-Side		
PC Multiplexer (PC VA FA<31:00> H)	From the PC subsystem to the A-side, selects PC, VA, or FA information (see Table 2-11).	T5 to T7
Data Input Signals - B-Side		
File B/Slow Data File (FB SDF<31:00> H)	Multiplexed data from the RGF File B or SDF data outputs, selected by B RD MODE (EBPORT<8> from the microword (see Table 2-19).	T5 to T7

Table 2-16 ALU First Half (ALF) Signal Descriptions (Cont)

Signal Name	Description	Valid
Data Input Signals - A-Side and B-Side		
A Cache Data (A CD<31:00> H)	From the unlatched CD multiplexer. Inputs cache read data or cache write/read bypass data.	T5 to T7
Bypass Bus (BP BUS<31:00> H)	Driven by the ALS or an ALU on the SHR module; inputs to the PAR to become the W bus. Inputs directly to the ALF in order to bypass the W bus when accepting data from the SHR.	T5 to T7
IB Data Bus (IB DATA<31:00> L)	Instruction buffer operand data from the IBox.	T5 to T7
Control Input Signals		
ALU Port Control (PORT CNTL<15:00> H)	A-Side -- Each pair of the PORT CNTL bits <11:10><09:08><03:02> <01:00> from the DEC module selects the A-side byte inputs for the slice. (ASEL<3:2,1:0>, see Table 2-18.)	T5 to T7
	B-Side -- Each pair of the PORT CNTL bits <15:14><13:12><07:06> <05:04> from the DEC module selects the byte inputs for the slice. (BSEL<3:2,1:0>). B RD MODE is EBPORT<8> from the microword (see Table 2-19).	T5 to T7
ALU Constant (EALUCON H)	EALUCON from the microword, when asserted, forces a constant of 4 to be read by the least significant nibble on the B-side and a value of 0 by all other ALF MCAs. (A value of 0000 0004 is read from the B-side instead of the data input value.)	T5 to T7

Table 2-16 ALU First Half (ALF) Signal Descriptions (Cont)

Signal Name	Description	Valid
Function Decode Signals		
ALU Function Control (EALU<3:0> H)	EALU<3:0> from the microword control the ALF partial sum functions in conjunction with the ALS functions (see Table 2-21).	T5 to T7
Effective Subtract (EFF SUB H)	Indicates a Floating-Point Shuffle (FPS) operation. A special case that converts an FP add to a subtract in the ALF partial sum.	T5 to T7
Generate Nibble (GNIB<3:0> L)	Nibble carry generates to the ALS look-ahead logic on the same slice.	T6 to T8
Propagate Nibble (PNIB<3:0> L)	Nibble carry propagates to the ALS look-ahead logic on the same slice.	T6 to T8
Generate Module (GMOD<1:0> H)	Cross-module word carry generates to the ALS MCAs on the other slice. (GMOD<1:0> are a fanout of the same signal.)	T6 to T8
Propagate Module (PMOD<1:0> H)	Cross-module word carry propagates to the ALS MCAs on the other slice. (PMOD<1:0> are a fanout of the same signal.)	T6 to T8
BCD DIGIT OK <1:0> H	Indicates that the nibble contains a valid decimal digit.	T6 to T8
Clock Distribution Control		
KEEPGOING<1:0> H	Asserted during a stall. Holds the A-side or B-side A CLK latches open until the expected data is available.	T6 to T8

Table 2-17 ALU Second Half (ALS) Signal Descriptions

Signal Name	Description	Valid
Output Signals		
Virtual Address Bus (VA<31:00> H)	Virtual address to the Cbox.	T7 to T9
Virtual Address Parity (VA PAR<3:0> H)	Virtual address parity to the Cbox.	T7 to T9
Bypass Bus (BP<31:00> H)	Bypass bus data from the ALS or the SHR module inputs directly to the PAR and ALF MCAs.	T7 to T9
Carry Bits (CBIT<2:0> H)	Carry (C) bits from ALF bits <31,15,07>.	T7 to T9
Overflow Bits (VBIT<2:0> H)	Overflow (O) bits calculated on byte, word, and longword boundaries.	T7 to T9
Floating-Point Opcode (FPOP H)	Indicates that the current instruction is floating-point. Controls the carry save logic in the next higher byte PAR MCA on the same slice.	T6 to T8
Fast Normalize Bit (FAST NORM BIT H)		
Input Signals		
A Partial Sum (APS<31:00> H)	Partial sum nibble inputs from the ALF A-side.	T6 to T8
B Partial Sum (BPS<31:00> H)	Partial sum nibble inputs from the ALF B-side.	T6 to T8
Carry In (CIN H)	Carry input from the next lower byte PAR MCA on the same slice.	T6 to T8
Bypass Output Enable (BPOUT EN H)	EALENBP from the microword. Enables the selected ALS outputs to the bypass bus.	T6 to T8
ALU Function Control (ALU<5:0> H)	EALU<5:0> from the microword. Selects the ALU function to be performed (see Table 2-21).	T5 to T7
Module Number (MOD NUM L)	Hardwired high or low. Identifies the slice for each ALS MCA.	

Table 2-18 A-Side Select (ASEL) Input Control Signals

ASEL Bits <3,1>	ASEL Bits <2,0>	Selected Input Data
0	0	PC VA FA Bus
0	1	IB DATA Bus
1	0	BP Bus
1	1	A CD Bus

Table 2-19 B-Side Select (BSEL) Input Control Signals

B RD MODE<2>	BSEL Bits <3,1>	BSEL Bits <2,0>	Selected Input Data
0	0	0	SDF data
1	0	0	RGF File B data
-	0	1	IB DATA bus data
-	1	0	BP Bus data
-	1	1	A CD Bus data

Table 2-20 Keepgoing/Stall Conditions

Input Conditions CD Valid	CD Ready	MD Valid	Keepgoing State*
-	0	0	1
0	-	0	1
-	-	1	0
1	1	-	0

* Note: 1 = Keepgoing, 0 = Stall.

Table 2-21 EALU<5:0> Field Control of the Main ALU

EALU<5:0> Value	General ALU Function	ALF Results APS	BPS	Carry Enable
00	A+B+Carry (FP, no mask)	AorB	-(AandB)	APSandBPS, 1
01	A+B+Carry (F & D mask)	AorB	-(AandB)	APSandBPS, 1
02	A+B+Carry (G mask)	AorB	-(AandB)	APSandBPS, 1
03	A+B+Carry (H mask)	AorB	-(AandB)	APSandBPS, 1
04	A-B-1+Carry (FP, no mask)	-(-AandB)	-(Aand-B)	APSandBPS, 1
05	A-B-1+Carry (F & D mask)	-(-AandB)	-(Aand-B)	APSandBPS, 1
06	A-B-1+Carry (G mask)	-(-AandB)	-(Aand-B)	APSandBPS, 1
07	A-B-1+Carry (H mask)	-(-AandB)	-(Aand-B)	APSandBPS, 1
0F	2's Compl. A (0-A-1+Carry)	-A	F	APSandBPS, 1
10	A+B+Carry Integer	AorB	-(AandB)	APSandBPS, 1
14	A-B-1+Carry Integer	-(-AandB)	-(Aand-B)	APSandBPS, 1
18	BCD 1st Cycle Add	- -	- -	APSandBPS, 1
1A	Inc A (A+Carry)	A	F	APSandBPS, 1
1C	Inc B (B+Carry)	B	F	APSandBPS, 1
1D	B-A-1+Carry Integer	-(Aand-B)	-(-AandB)	APSandBPS, 1
1E	BCD 1st Cycle Sub	- -	- -	APSandBPS, 1
20	AorB	AorB	-(AandB)	APS , 0
24	notAandB	-(-AandB)	-(Aand-B)	-APS , 0
2A	Pass A	A	F	APS , 0
2B	BCD 2nd Cycle	AorB	-(AandB)	APS , 1
2C	Pass B	B	F	APS , 0
2D	Aand.notB	-(Aand-B)	-(-AandB)	-APS , 0
2F	notA	-A	F	APS , 0
30	AandB	AorB	-(AandB)	-BPS , 0
39	AxorB	AorB	-(AandB)	APSandBPS, 0
3A	Force VA parity to 0	A	F	APSVA PAR=0, 0
3C	Clear	B	F	-BPS , 0
3F	Dec. A (A-Carry)	-A	F	-APS , 1

2.3 SHIFTER MODULE (SHR) DESCRIPTION

The SHR is a functional extension of the slice modules, providing additional ALUs and MCA operators that are connected in parallel with the main ALU on the SLC modules. The SHR processes a 32 or 64-bit operand on the APORT/BPORT lines and returns a 16 or 32-bit result on the bypass (BP) bus. Figure 2-9 identifies the logical operators provided by the SHR:

- Shifter (SHF)
- Floating-Point (FP) Support
  - Priority Encoder (PEN)
  - Shift ALU (SALU)
  - Exponent ALU (XALU)
- Multiplier/Divider (MULT)

2.3.1 Shifter (SHF)

The shifter (SHF) contains two types of MCA logic. Figure 2-10 shows the shift MCA (SHFT) logic, which consists of eight MCAs. It also shows the output gating that drives the BP bus lines through a set of NOR gates located on the SHR module. Figure 2-11 shows the shift control MCA (SHC) signals and two other signals from the microcode that control the SHFT MCAs. The SHF shifts integer or floating-point data and provides several other functions, one of which is to perform direct conversions between some of the decimal string formats. Its main hardware functions perform the following operations:

- Logical Shift or Rotate
- Arithmetic Shift
- Decimal String Conversion
- FP Normalize
- FP Align

The SHF mainly processes 64-bit operands but has several functions that also apply to 32-bit operands. Only a 32-bit output is available. For 32-bit data, the result is selected for the operand applied to the APORT or BPORT. For 64-bit data, the upper and lower 32 bits of the result are gated to the BP bus on different cycles.

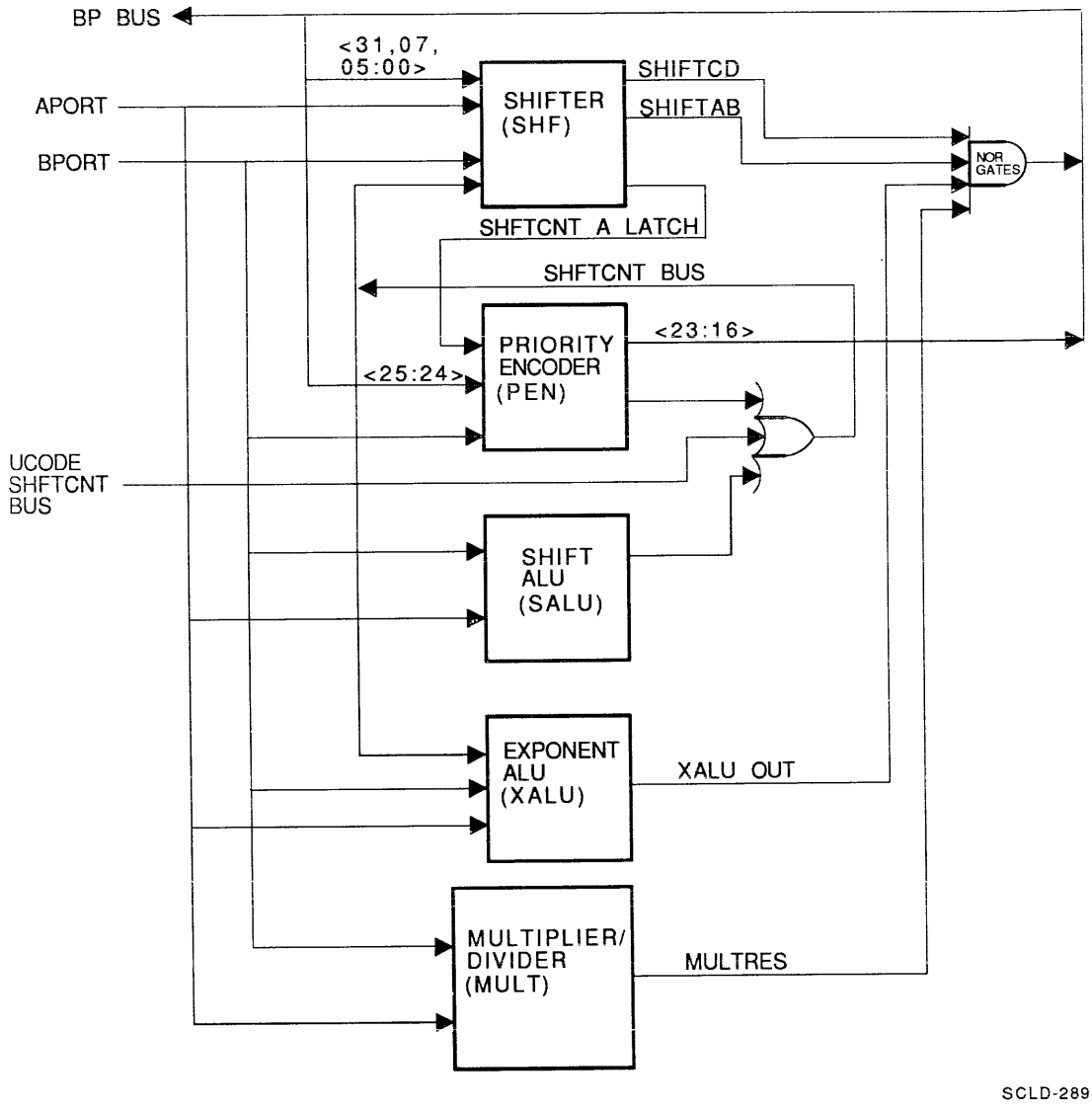


Figure 2-9 Shifter Module (SHR) Block Diagram

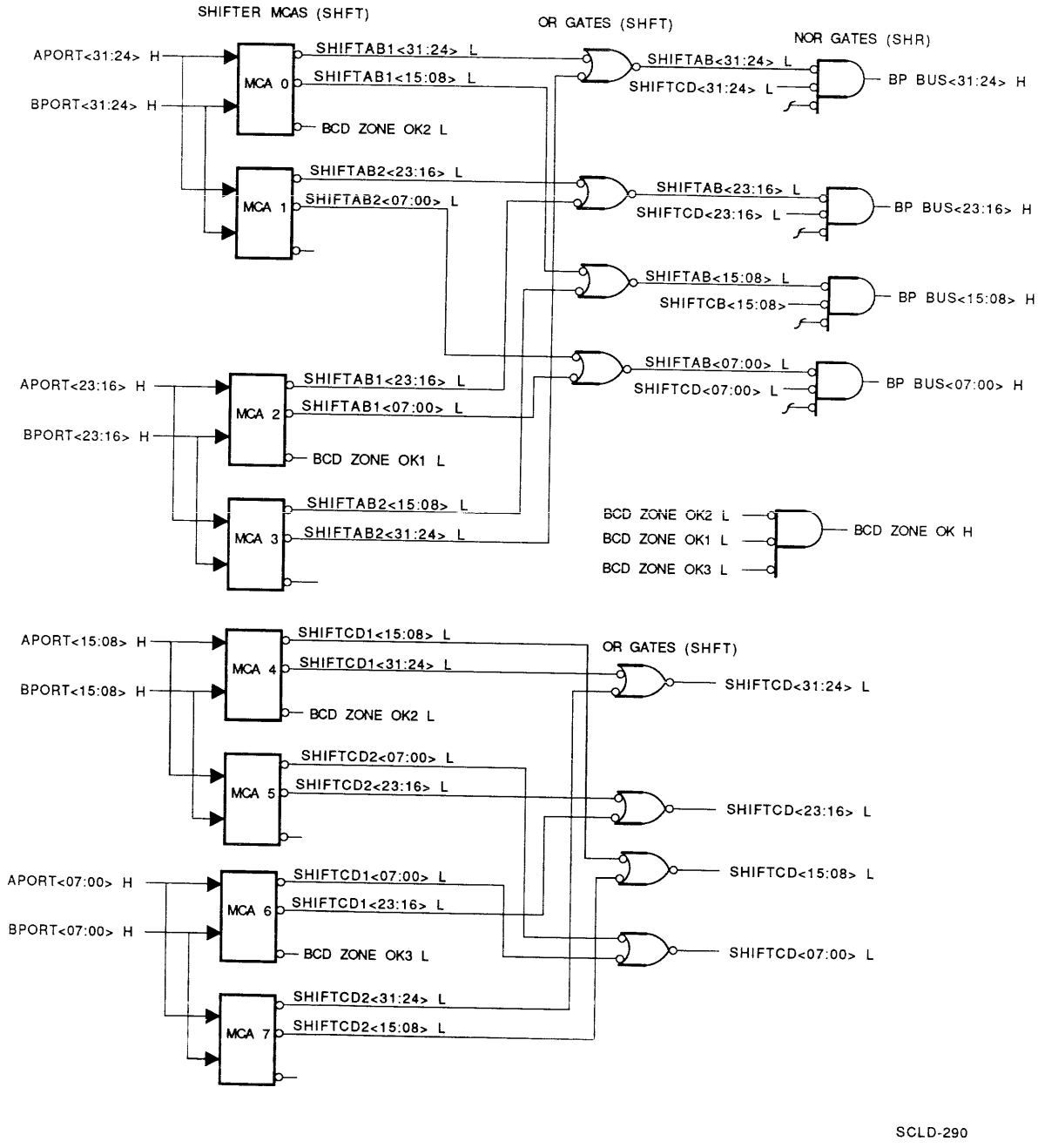


Figure 2-10 Shift MCA (SHFT) Logic and Gating Block Diagram

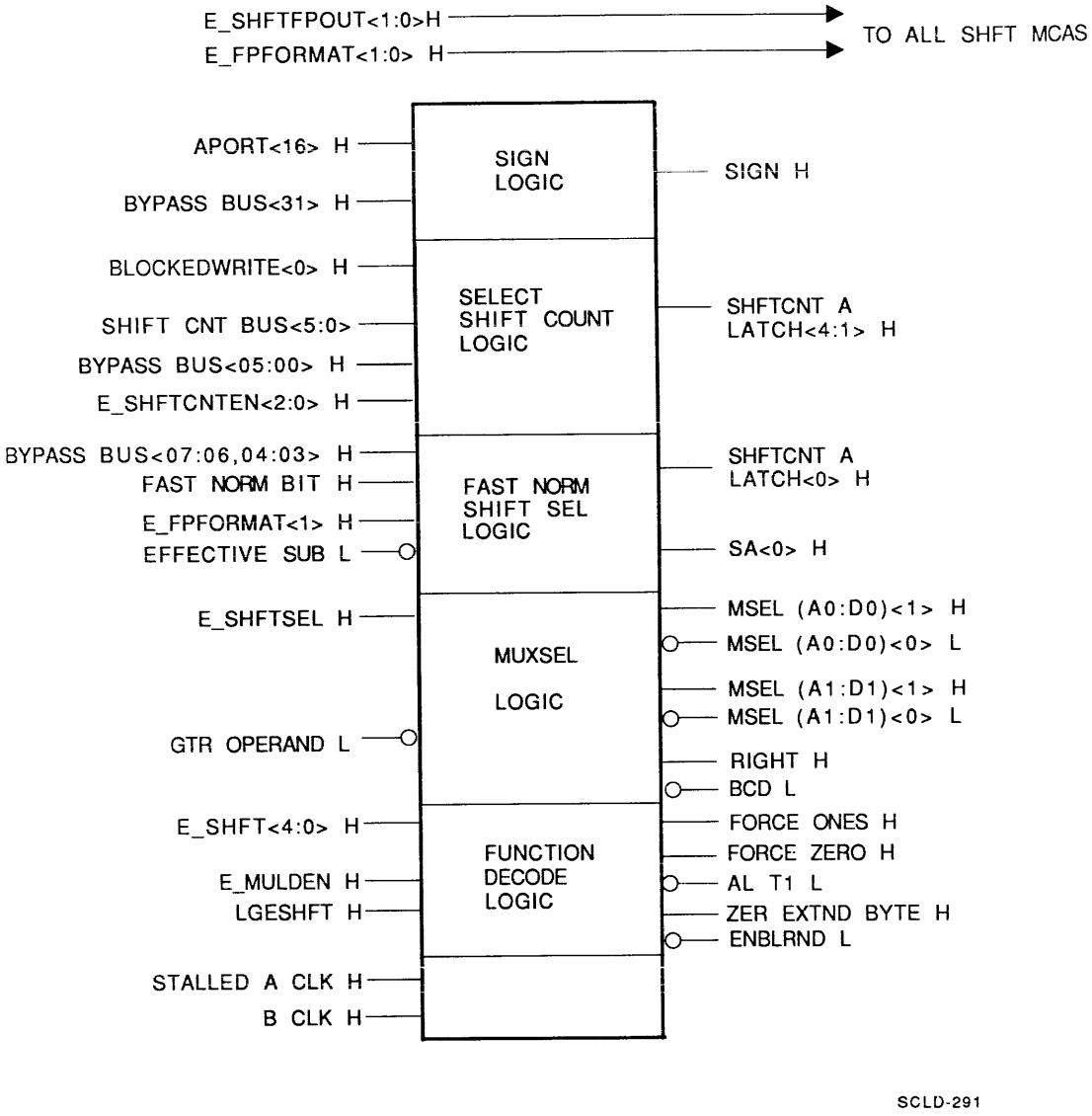


Figure 2-11 Shift Control MCA (SHC) Block Diagram

2.3.1.1 Shift Count Bus -- The shift count bus (SHIFT CNT BUS<5:0>) is a wire-OR of the three sets of signals shown in Figure 2-12. These signals are described in Table 2-22.

Table 2-22 Shift Count Bus Signals and Source

Signal Bus Name	Source of the Shift Count/Amount
SALU SHIFT CNT BUS<5:0>	Six bits from the SALU.
PE SHIFT CNT BUS<3:0>	Four bits from the PEN.
UCODE SHIFT CNT BUS<5:0>	Six bits from the microcode (the complement of ESHFTCNT<5:0>) or from the virtual address byte selection bits (VA<1:0>) when enabled by ESHFTCNTEN<2:1>. VA<1:0> are passed to shift count bits <4:3> (the remaining bits <5,2:0> are asserted as zeros).

The shift amount passed to the SHC or XALU is normally controlled by the microcode. For certain types of operations, the SHC passes the A-latched shift amount (SHFTCNT A LATCH<4:0>) to the PEN. In addition, certain floating-point operations through the XALU leave a value on the BP bus that can be readily used as the shift count amount by a following cycle.

2.3.1.2 General Function Selection -- Table 2-23 lists the SHF functions selected by the ESHFT<4:0> field of the microword. Table 2-24 shows how the ESHFTSEL bit is used with the ESHFT<4:0> field in selecting the result of a 32-bit operation on an APORT or BPORT operand and passing it to the BP bus. For the result of a 64-bit FP operation, the most significant (MS) and least significant (LS) 32 bits are passed to the BP bus on successive cycles.

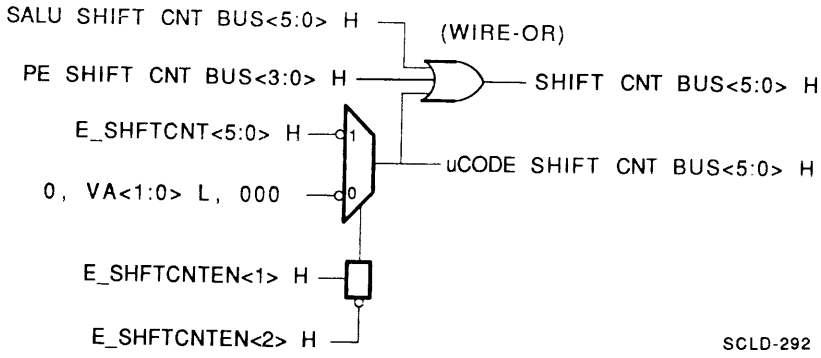


Figure 2-12 Shift Count Bus Signal and Gating Block Diagram

Table 2-23 ESHFT<4:0> Field Selection of Shifter (SHF) MCA Logic Functions

ESHFT<4:0> Value (Hex)	Selected General Function
00	Left rotate 64 bits by SHFTCNT value
01	Right rotate 64 bits by SHFTCNT value
02	Arithmetic left shift 64 bits by SHFTCNT value (same as logical shift left)
03	Arithmetic right shift 64 bits by SHFTCNT value (sign saved in SIGN SAVE register on a previous code 12 cycle)
04	Logical left shift 64 bits by SHFTCNT value
05	Logical right shift 64 bits by SHFTCNT value
06	Logical left shift 32 bits by SHFTCNT value
07	Logical right shift 32 bits by SHFTCNT value
08	FP normalize by SHFTCNT value
09	Reserved
0A	FP fast normalize MS half (FASTNORMMS)
0B	FP fast normalize LS half (FASTNORMLS)
0C	FP left shift 64 bits by SHFTCNT value
0D	FP right shift 64 bits by SHFTCNT value
0E	FP right align two 32-bit operands by SHFTCNT value
0F	FP right align 64-bit operand by SHFTCNT value
10	Zero-extend byte to longword (SHFTCNT = 0)
11	Zero-extend word to longword (SHFTCNT = 0)
12	Logical left shift 64 bits by SHFTCNT value and load SIGN SAVE register from BP bus bit <31>
13	Short arithmetic right shift 64 bits (0 to 7 places)
14	Pass -- Logical left shift by 0

Table 2-23 ESHFT<4:0> Field Selection of Shifter (SHF) MCA Logic Functions (Cont)

ESHFT<4:0> Value (Hex)	Selected General Function
15	ClearMult -- Clear BP bus when using the MULT logic
16	BCD functions (SHFTCNT must = 0, see Section 2.3.1.8)
17	Shifter off (force ones to NOR gates)
18--1B	Reserved
1C	Inhibit APORT parity check (used with ALU FP masking)
1D	Force Wbus parity to zero
1E	Clear parity error, open parity error register
1F	Read Ebox parity error register

Table 2-24 ESHFTSEL Selection of a Result Output to the BP Bus

ESHFT<4:0> Value	ESHFTSEL Equal to 0 Selects	ESHFTSEL Equal to 1 Selects	Operand or Data Size
0--5, 12, 13	LS half of the shifted result	MS half of the shifted result	64 bits
6, 7, 10, 11, 14	APORT as input	BPORT as input	32 bits
B, C, D, F	MS half of the FP shifted result	LS half of the FP shifted result	64 bits
A	BPORT as input	APORT as input	32 bits
B	BPORT as input	Illegal Function	32 bits

Note: The ESHFTSEL bit is ignored on the following operations:

Decimal conversion
Clear parity error
FP right align two 32-bit operands

2.3.1.3 Logical Shift or Rotate -- Each SHFT MCA processes or passes one byte of data, performing a 0 to 7-bit shift in one cycle. To perform shifts of more than 7 bits, complementary functions are shared between the even and odd MCAs as determined by the chip identification code (ID codes 0 through 7). The ID code for each SHFT MCA is hardwired to the personality select (PS) input pins.

Example of a Logical Shift of APORT or BPORT data follows:

To perform a 2-bit left shift, MCAs 0, 2, 4, and 6 shift the data left two places. The upper two bits are lost and zeros are shifted into the lower two bits. However, MCAs 1, 3, 5, and 7 shift the data right six places. MCAs 1 and 3 shift zeros into the upper six bits, while MCAs 5 and 7 shift in ones to satisfy the AND requirement by the NOR gates to the BP bus. (The MULT and XALU MCAs are unused in the operation and force 1s to satisfy this condition.) The outputs of MCA0 are then inclusive-ORed with MCA3 to assert the shifted byte on BP BUS<31:24>. For a right rotate, MCA1 asserts bits <31:30> (from the APORT or BPORT). For a right shift, it asserts zeros for those bits.

2.3.1.4 Arithmetic Shift -- An arithmetic shift performs the same functions as a logical shift except that, on an arithmetic right shift, the data is sign-extended from the SIGN SAVE latch. Two cycles are required. The first cycle operates on ESHFT<4:0> code 12, which asserts the data on the BP bus, shifting it left by the necessary number of places to place the sign bit on BP bus bit <31>. The SIGN SAVE latch is loaded from BP bus bit <31> and the actual right shift must take place on a subsequent cycle.

The SIGN SAVE latch is normally not altered except by the 12 code or when corrupted by an FP fast normalize. Its state is held over traps and should not be altered by a trap service routine unless it is first saved (by an arithmetic right shift), then restored.

Arithmetic Short Shift

The arithmetic short shift function can right shift a 64-bit integer from 0 to 7 places, a function that is useful when converting bit addresses to byte or word addresses.

2.3.1.5 Decimal String Conversion -- A value of 16 in the ESHFT<4:0> field of the microword selects the conversion of BCD data between the decimal string formats (except packed decimal to/from leading separate numeric). Conversion takes place on the APORT operand. SHFTCNT A LATCH<4:0> must assert a value of zero. Table 2-25 shows how EFPFORMAT<1:0> select conversion between the trailing numeric, packed, and integer formats.

Table 2-25 EFPFORMAT<1:0> Field Control of Decimal String Data Conversions

EFPFORMAT<1:0> Value (Hex)	ESHFT<4:0> Value Equal to 16 If No, Select	If Yes, Convert
0	No Mask	32-bit trailing --> 16-bit packed Output is on the lower 16 bits of the BP bus. The microcode branch flag BCD ZONE OK H is asserted if the upper nibble of each byte in the trailing format is a 3.
1	F, D Mask	16-bit packed --> 32-bit trailing Performs a conversion on the lower 16 bits of the APORT.
2	G Mask	32-bit packed <--> 32-bit integer Converts one longword format to the other by swapping the two end bytes and swapping the two center bytes.
3	H Mask	Reserved

2.3.2 Floating-Point (FP) Support

The SHR module provides three operators that support the floating-point calculations made by the microcode. Each operator is a single MCA:

- Priority Encoder (PEN)
- Shift ALU (SALU)
- Exponent ALU (XALU)

These operators are mainly concerned with operations on the F_, D_, and G_Floating data formats shown in Figure 2-13. (H_Floating operations are performed entirely by the microcode.) Sections of each MCA logic may also be used by the microcode for general arithmetic or register functions.

The PEN receives FP data on the BPORT and performs three separate operations:

1. Priority encodes floating data in order to find the most significant (MS) 1-bit in the mantissa or to aid the microcode in finding the byte that contains it.

2. Rounds and performs a 0 or 1-bit shift of the least significant (LS) byte of an FP normalization.

3. Stores and generates the rounding bits for a floating add (ADD) or subtract (SUB), possibly sending a carry-in (plus 1) request to the Main ALU.

The SALU and XALU are both controlled by some of the same microcode fields and receive the exponent of both operands:

- The APORT receives the exponent of the first operand.
- The BPORT receives the exponent of the second operand.

The SALU is mainly concerned with the addition or subtraction of two floating operands:

- Produces the resulting sign of the fraction and returns it to the SLC0 module on BP BUS<15>.

- Subtracts the exponents and generates a shift count based on the difference. Sends the result to the shifter logic (SHF), which aligns the operands by shifting the fraction field of the smaller.

- Generates most of the FP condition code bits for microcode branching.

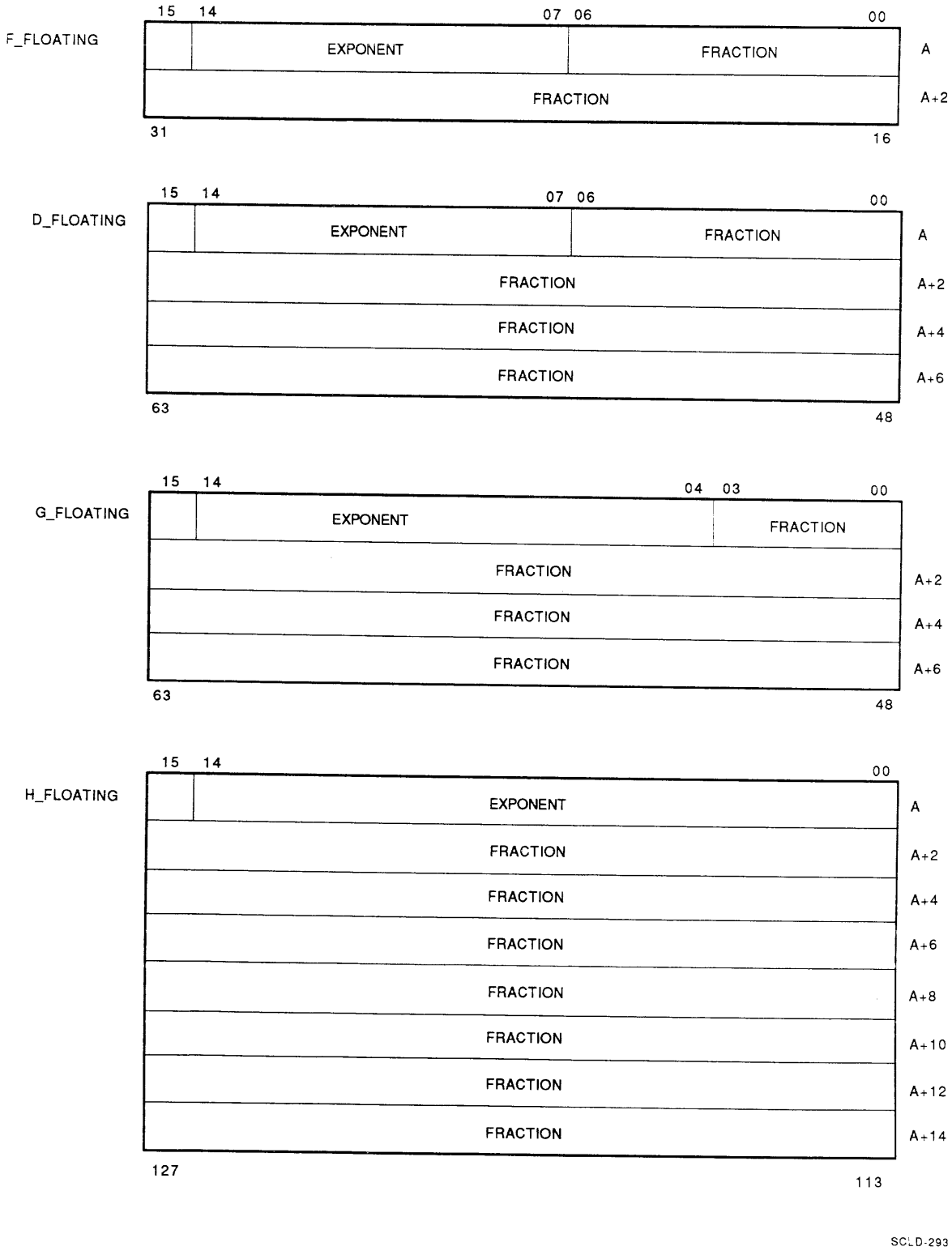


Figure 2-13 VAX-11 Floating-Point Formats

The XALU is mainly concerned with a multiply or divide of two floating operands:

- Performs 12-bit exponent arithmetic in the exponent register (XREG). Returns the result to the SLC0 module on BP BUS<14:04>.
- Generates one FP condition code bit for microcode branching.

2.3.2.1 Priority Encoder (PEN) MCA -- Figure 2-14 identifies the main functional areas of the PEN logic:

PE Logic	Tests the mantissa of a floating datum to find the most significant 1 bit, then passes the shift count value needed to the SHF to normalize the fraction.
Sticky Bit	Examines the eight most significant bits shifted out on an FP align. If all eight bits are 0, the carry-in bit for the main ALU is set.
Guard and Round Bit Select Logic	Saves the last two bits shifted out of the data field on an FP align. These bits are used as the rounding bits on a normalization.
Fast Normalize	Increment (INCR) logic input. Performs a 0 or 1-bit right or left shift of the LS byte of a floating number, and passes the result to the rounding increment logic.
Rounding Increment	Increment (INCR) logic output. Takes the normalized 8-bit result from the fast normalize logic and adds a 0 or 1 according to the rounding bits and operation (ADD or SUB). When enabled, it passes the result to the LS byte of the floating number on BP BUS<23:16>.

Function Decoder

The PE function field (EPEFUNC<2:0>) selects the PEN operating functions as shown in Table 2-26.

Input Multiplexer (INMUX)

The INMUX aligns the BPORT value for an operation on an F_/D_ or G_ format input as shown in Figure 2-15. The LS bit position of the exponent represents the hidden bit and is aligned with the MS bit of the fraction. This data is passed to the rest of the logic as PEDATA<20:00>.

Priority Encoder (PE)

The E logic tests the mantissa of a floating number (including the hidden bit) to find the most significant 1-bit. It then passes the shift amount needed for a normalize to the SHF and XALU as shown in Table 2-27. If a 1-bit is not found on the first pass, it asserts the ALL ZERO status signal to the microcode.

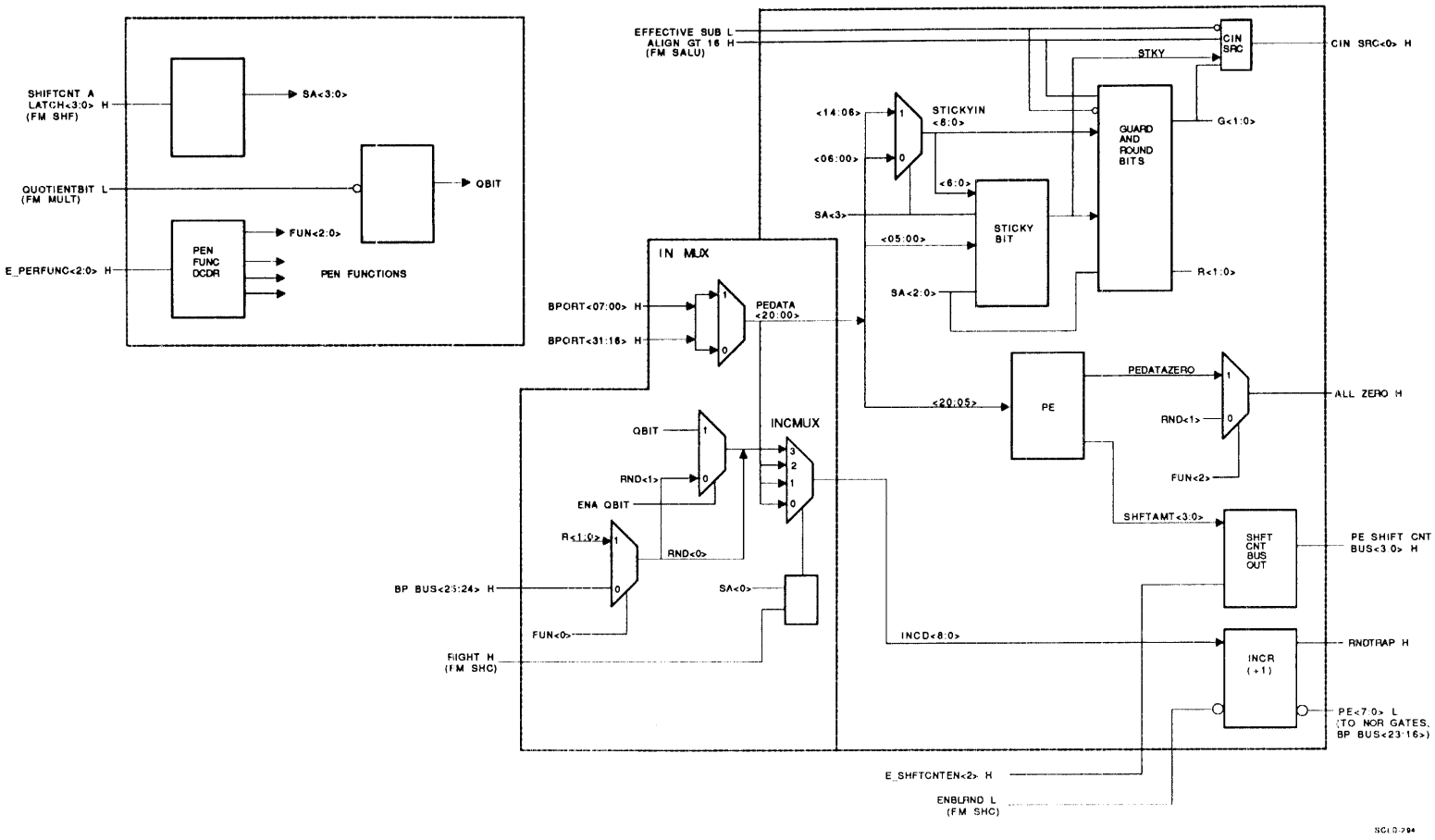


Figure 2-14 Priority Encoder (PEN) Block Diagram

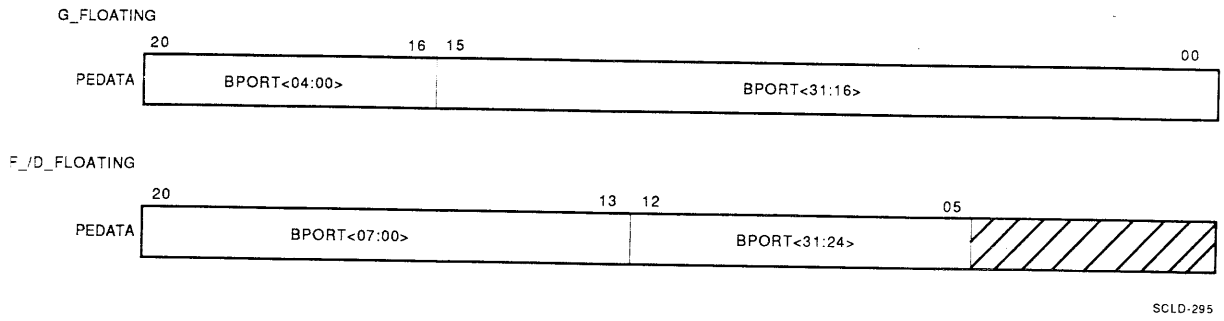


Figure 2-15 INMUX Mapping of the BPORT Input Data

Table 2-26 EPEFUNC Field Selection of PEN Functions

EPEFUNC <2 1 0>	Function	Comments
0 0 0	Left shift into LS byte, 0 or 1 place	No rounding
0 0 1	Left shift into LS byte, 0 or 1 place	Add rounding increment
0 1 0	Load round bits RND<1:0> from BP BUS<15:14>	
0 1 1	Load round bits RND<1:0> from R<1:0>	
1 0 0	SHFTCNT BUS <-- PE value (F_/D_ format)	
1 0 1	SHFTCNT BUS <-- PE value (G_ format)	
1 1 0	SHFTCNT BUS <-- 1, BP BUS<16> <-- QBIT	Divide
1 1 1	SHFTCNT BUS <-- 8	Multiply

Rounding Incrementer (INCR)

The increment multiplexer (INCMUX) aligns the lowest nine INMUX bits for the incrementer as shown in Table 2-28. For a fast normalization (FAST NORM), the INCMUX outputs are directed by the state of SHIFTCNT A LATCH<0> (SA<0>) from the SHF:

SA<0> = 0 Passes the true state of PEDATA<7:0> and RND<1>.
SA<0> = 1 Shifts the value right or left by one place, according to the RIGHT signal (0 = left, 1 = right).

With E_PEFUNC<0> asserted, the INCR adds INCD<0> to INCD<1>. With ENBLRND asserted, it passes the rounded result of INCD<8:1> to PE<7:0> (for assertion to BP BUS<23:16>). RNDTRAP is asserted to the microcode if adding the rounding bit caused a carry-out from INCD<8>.

Sticky, Guard, and Round Bits

Inputs to this logic are controlled by the shift count A-latch (SA<3:0>) value asserted by the SHF as shown in Table 2-29. The sticky logic asserts STKY to the Carry-In Source (CIN SRC) gating and to the Guard and Round Bit logic when any of the tests are true.

Table 2-27 Priority Encoder (PE) Results Passed to the Shift Count Bus

Input Conditions	Logical Test Results
PEDATA<20:13> Equal to 0	Assert SHFTCNT<3> and test PEDATA<12:05>. Then, if: E_PEFUNC<2> = 1, assert ALL ZERO if PEDATA<12:05> are also equal to 0. E_PEFUNC<2> = 0, assert ALL ZERO if rounding bit RND<1> is set.
PEDATA<20:13> Not Equal to 0	Negate SHFTCNT<3> and test PEDATA<20:13>. In either case, assert the final shift count value as follows:

PE Test Bit Value <7 6 5 4 3 2 1 0>	Output Passed to SHFTCNT<2:0>
1 - - - - - - -	0 0 0
0 1 - - - - - -	0 0 1
0 0 1 - - - - -	0 1 0
0 0 0 1 - - - -	0 1 1
0 0 0 0 1 - - -	1 0 0
0 0 0 0 0 1 - -	1 0 1
0 0 0 0 0 0 1 -	1 1 0
0 0 0 0 0 0 0 1	1 1 1
0 0 0 0 0 0 0 0	1 1 1

Note: E_SHFTCNTEN<2> must be asserted to pass the SHFTCNT<3:0> result to PE SHIFT CNT BUS<3:0>.

Table 2-28 Increment Multiplexer Data (INCD) Selection to the Incrementer (INCR)

SA<0>	RIGHT	INCR Bits <8 - - - 2>	<1>	<0>
0	-	PEDATA Bits <7 - - - 1>	<0>	RND<1>
1	0	PEDATA Bits <8 - - - 2>	<1>	<0>
1	1	PEDATA Bits <6 - - - 0>	<*>	RND<0>
			└	ENA QBIT: 0 = RND<1> (Code 110) 1 = QBIT

Note: If E_PEFUNC<0> = 0, RND<1:0> assert BP BUS<25:24>. If $\overline{E_PEFUNC<0>}$ = 1, RND<1:0> assert R<1:0> (round bits); R<1> (RND<$\overline{1}$>) added to INCR<1> and INCR<8:1> are asserted as PE<7:0>. Assert RNDTRAP if INCR<8> produced a carry-out.

Table 2-29 Sticky Bit Logic Input and Test Selection

Input Selection	Selected Input or Test
SA<3> Equal to 1	Asserts PEDATA<14:06> to the Guard and Round Bit logic on STICKYIN<8:0>. Sticky logic performs an OR of PEDATA<05:00> where any bit on a 1 asserts STKY.
SA<3> Equal to 0	Asserts PEDATA<06:00> to the Sticky logic and the Guard and Round Bit logic. (Zeros are asserted on STICKYIN<8:7>.) Sticky logic performs a selective OR of PEDATA<06:00>, where a 1-bit asserts STKY as selected by SA<2:0>:

SA Bits <2 1 0>	Selects OR Test of PEDATA Bits	Comments
0 0 0	- -	STKY negated
0 0 1	<0>	STKY asserted
0 1 0	<1:0>	if any bit is
0 1 1	<2:0>	a 1
1 0 0	<3:0>	
1 0 1	<4:0>	
1 1 0	<5:0>	
1 1 1	<6:0>	

Table 2-30 G<1:0> Guard Bit Input Selection

SA Bits <2 1 0>	STICKYIN Bits Selected for Assertion as G<1:0> <8 7 6 5 4 3 2 1 0>
0 0 0	- - - - - - - X X
0 0 1	- - - - - - X X -
0 1 0	- - - - - X X - -
0 1 1	- - - - X X - - -
1 0 0	- - - X X - - - -
1 0 1	- - X X - - - - -
1 1 0	- X X - - - - - -
1 1 1	X X - - - - - - -

Note: Xs indicate the two STICKYIN bits selected for assertion as guard bits G<1:0> (and for possible assertion as the round bits R<1:0>).

Table 2-31 Round Bit R<1:0> Input Selection

EFF SUB	ALIGN GT 16	STKY	Bits Selected as the R<1:0> Outputs
0	1	-	(Outputs deselected)
0	0	-	R<1:0> <-- G<1:0>
1	1	-	R<1:0> <-- Ones
1	0	0	R<1:0> <-- G<1:0> (Complement of G<1:0>)
1	0	1	R<1> <-- XOR of G<1:0> and R<0> <-- G<0>

2.3.2.2 Shift ALU (SALU) -- In addition to their special FP functions, the SALU and XALU may be used by the microcode for general arithmetic or register operations. They are both controlled by some of the same microcode fields as shown in Table 2-32.

Figure 2-16 identifies the main functional areas of the SALU logic:

- SALU Function Decoder (SFND)
- SALU Sign (SSGN)
- SALU Input Select (SINS)
- SALU Symmetrical Difference (SDIF)
- SALU Output Inverter (SOPI)
- SALU Branch Logic (SBRL)

SALU Function Decoder (SFND)

The SALU/XALU function field of the microword (E_SXALUFN<5:0>) controls the SALU functions as shown in Table 2-33.

Condition Codes

The SFND logic does not produce any microbranch condition codes. However, each of the other logical areas produces codes that are asserted to the SBRL logic, where they are either used directly or to generate other types of codes.

SALU Sign (SSGN)

The SSGN logic receives the fractional signs of the first and second floating operand on APORT<15> and BPORT<15>. For a floating add or subtract, the SSGN produces the sign of the resulting fraction, as shown in Table 2-34, and writes it to the sign latch.

Sign Latch -- The sign latch is a 1-bit recirculating register consisting of a stalled A-clock and B-clock latch. The sign latch holds the resulting sign of the fraction for assertion to BP BUS<15> or for use as a microbranch condition. The stalled A-clock latch (ASIGN LATCH) value is passed to the SBRL.

SALU Input Select (SINS)

The SINS consists of a set of multiplexers that receive the exponent of the first operand on the APORT<14:04> inputs and receive the exponent of the second operand on the BPORT<14:04> inputs.

Table 2-32 SALU and XALU Control Signals from the Microcode

Signal Name	Bit Description
CS0 RAM Segment (SALU)	
EFPSUFL	Asserted, enables floating-point shuffle to the IBox.
CS1 RAM Segment (SALU and XALU)	
ESXALUFN<5:3>	Selects the general SALU/XALU functions. (Described in Tables 2-33 and 2-38.)
ESXALUFN<2:1>	Selects specialized functions according to the microcode operation.
ESXALUFN<0>	Selects the input/output format: 0 = Operand is F_ or D_Floating 1 = Operand is G_Floating or Integer
ESXBYEN	SALU -- Asserted, enables the SIGN LATCH value (the resulting sign of the fraction) to BP BUS<15>. XALU -- Asserted, enables the resulting exponent (or result of the operation) to BP BUS<14:04>.
CS1 RAM Segment (SALU)	
ESHFTCNTEN<2>	1 = Enables the SALU SHIFT CNT<5:0> value to the shift count bus.
CS2 RAM Segment (SALU)	
ERECIPE<1:0>	Selects one of four microbranch condition code groups from the branch multiplexer.
ESIGNWR	1 = Writes the sign to the SIGN LATCH and enables the reserved operand trap check logic. 0 = Inverts the shift amount. (Is used if WRONG SHIFT was set on the first pass through the SALU.)

Table 2-33 ESXALUFN Field Control of the SALU Functions

E_SXALUFN <$\overline{5}$ 4 3 2 1>	Exponent Operation or Function	Comments
See Note 1.		
0 0 0 0 0	BP BUS<15> <-- SIGN LATCH	Note 2.
| 0 1	BP BUS<15> <-- SIGN LATCH; SHFTCNT <-- SHFT AMT	ADD
| 1 0	BP BUS<15> <-- SIGN LATCH	
0 0 0 1 1	BP BUS<15> <-- SIGN LATCH; SHFTCNT <-- SHFT AMT	SUB
See Note 3.		
0 0 1 - -	SIGN LATCH <-- SIGN LATCH XORed with Sign of Fraction	MUL
0 1 0 - -	SIGN LATCH <-- SIGN LATCH XORed with Sign of Fraction	MUL
0 1 1 - -	SIGN LATCH <-- SIGN LATCH XORed with Sign of Fraction	MUL
1 0 0 - -	SIGN LATCH <-- Sign of the Larger Fraction	
1 0 1 0 0	SIGN LATCH <-- 1	
| 0 1	SIGN LATCH <-- APORT<15>	
| 1 0	SIGN LATCH <-- 0	
1 0 1 1 1	SIGN LATCH <-- BPORT<$\overline{\overline{15}}$>	Note 4.
1 1 0 - -	SIGN LATCH <-- SIGN LATCH XORed with Sign of Fraction	MUL
1 1 1 - -	SIGN LATCH <-- SIGN LATCH XORed with Sign of Fraction	DIV

Notes:

1. ESXALUFN<0> -- Equal to 0, selects the format for F_/D_Floating. Equal to 1, selects the format for G_Floating/Integer.

2. SIGN LATCH -- The sign latch value is passed to BP BUS<15> if E_SXBYEN is asserted. Otherwise, the operation is performed but the result is not sent.

3. Write SIGN -- The reserved operand trap (RES OP TRAP) check is enabled when E_SIGNWR is asserted.

4. BPORT<$\overline{\overline{15}}$> -- If the result generates a reserved operand trap, the BPORT sign is not inverted.

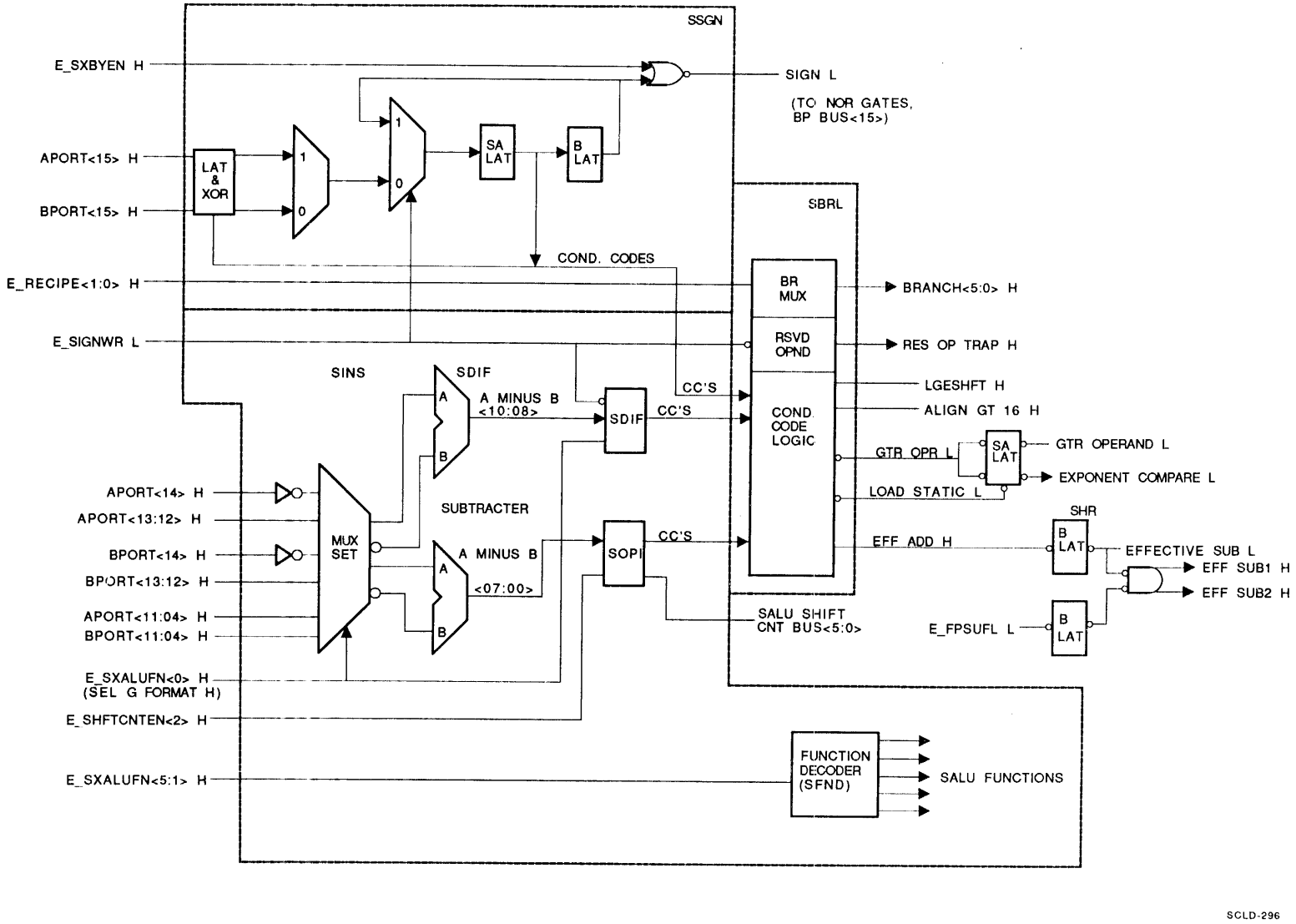


Figure 2-16 Shift ALU (SALU) Block Diagram

Table 2-34 Resulting Sign of the Fraction

Operation	Resulting Sign	Resulting Operation
(+A) + (+B)	Plus	Effective ADD
(+A) - (+B)	Plus, if A > B; Minus, if B > A	Effective SUB
(+A) + (-B)	Plus, if A > B; Minus, if B > A	Effective SUB
(+A) - (-B)	Plus	Effective ADD
(-A) + (+B)	Minus, if A > B; Plus, if B > A	Effective SUB
(-A) - (+B)	Minus	Effective ADD
(-A) + (-B)	Minus	Effective ADD
(-A) - (-B)	Minus, if A > B; Plus, if B > A	Effective SUB

Table 2-35 SALU Selection of the APORT and BPORT Inputs

Floating Format	SINS Outputs to the SDIF Subtracter <10 09 08 07 06 05 04 03 02 01 00>	Comments
APORT or BPORT Inputs to the SINS		
G_	<$\overline{14}$ $\overline{14}$ 13 12 11 10 09 08 07 06 05 04>	
F_/D_	<$\overline{14}$ 13 12 $\overline{14}$ 13 12 11 10 09 08 07>	
Exponent Input Value		
G_	1 0 0 0 0 0 - - - - -	Small, see
F_/D_	1 0 0 1 0 0 - - - - -	Note 1.
G_	0 1 1 1 1 1 1 1 1 1 1	Large, see
F_/D_	0 1 1 0 1 1 1 1 1 1 1	Note 2.
G_	1 0 0 0 0 0 0 0 0 0 0	Zero, see
F_/D_	1 0 0 1 0 0 0 0 0 0 0	Note 3.

Notes:
1. Asserts OVR_UND POSS ADD if EFFECTIVE ADD = 0 and both exponents have a value of less than 64. (Adding the exponents may result in a floating underflow.)
2. Asserts OVR_UND POSS ADD if EFFECTIVE ADD = 1 and either exponent has a value of all ones. (Adding the exponents may result in a floating overflow.)
3. Asserts B ZERO EXPA if the APORT exponent is zero. Asserts B ZERO EXPB if the BPORT exponent is zero.

Exponent Sign

VAX architecture uses the inverted state of the exponent sign to produce a true positive zero. Therefore, APORT<14> and BPORT<14> are both inverted on input to the SINS logic.

SINS Input Registers and SDIF Subtracter

The APORT and BPORT operands are each multiplexed to a B-latch register according to a selected format as shown in Table 2-35. Both registers are used in tests that assert condition codes to the SBRL. The 1 side of the APORT latches (which are part of the input MUX SET) are applied to the A-side of the SDIF adder. However, the 0 side of the BPORT latches are applied to the B-side of the adder, making it a full-time subtracter.

SALU Symmetrical Difference (SDIF)

The SDIF receives the upper three bits of a subtraction result (A MINUS B<10:08>). It also receives the carry outputs from bits <10> and <07>, which are used to produce and assert condition codes to the SBRL.

SALU Output Inverter (SOPI)

The SOPI receives the lower eight bits of a subtraction result (A MINUS B<07:00>) and produces the shift amount required for a floating align. It also uses these bits in tests that produce and assert condition codes to the SBRL.

SALU Branch Logic (SBRL)

The SBRL branch multiplexer receives the condition codes produced by the other logical sections, either using them directly or with signals that produce other kinds of codes. The SBRL outputs one of the four code groups shown in Table 2-36, under the control of the E_RECIPE<1:0> field. Table 2-37 describes the logical states or conditions that are passed to the microbranch logic.

Most of the SBRL input signals are static condition codes that are loaded to a set of stalled A-clock latches. The latch states remain the same until they are again loaded by one of two SALU functions:

E_SXALUFN<5:3> = 101
E_SXALUFN<5:1> = 000X1

Reserved Operand Trap

The RES OP TRAP signal is one of the nonstatic condition codes. Enabled by ESIGNWR, it is asserted if the sign of either fraction is negative but its exponent is zero.

Table 2-36 A-Latched Condition Code Inputs to the Branch Multiplexer

E_RECIPE Bits <1:0>	Branch Multiplexer Outputs to the BRANCH<5:0> B-Latch BRANCH<5>	BRANCH<4>	BRANCH<3>
1 1	A_EFF ADD	A_ANY OP ZERO	A_BOTH OP ZERO
1 0	A_GTR 64 ALIGN	A_GTR 32 ALIGN	A_ANY OP ZERO
0 1	A_SIGN LATCH*	A_ZERO OP B	A_SIGN OF EXP*
0 0	A_GTR 32 ALIGN	A_EFF ADD	A_ALIGN 1632
<1:0>	**BRANCH<2>**	**BRANCH<1>**	**BRANCH<0>**
1 1	A_SIGN LATCH*	0	0
1 0	A_EFF ADD	A_WRONG SHIFT	A_BOTH OP ZERO
0 1	A_ZERO OP A	A_GTR 32 ALIGN	0
0 0	A_OVR OR ANY OP_ZERO	A_UNKNOWN_WRONG LARGE	A_FULL NORM

* Nonstatic Condition Codes

Table 2-37 Microbranch Condition Code Description

Signal Name	Function
A_EFF ADD	Asserted or negated depending on the states shown in Table 2-34.
A_SIGN LATCH	Asserts the fractional sign result.
A_ANY OP ZERO	Either or both of the exponents are zero.
A_BOTH OP ZERO	Both of the exponents are zero.
A_GTR 64 ALIGN	The result of an exponent subtraction will be greater than 64.
A_GTR 32 ALIGN	The result of an exponent subtraction will be greater than 32.
A_ALIGN 1632	The result of an exponent subtraction will be in the range of 16 to 32.
A_WRONG SHIFT	The calculated shift amount is wrong. (The microcode must force the correct result from the subtractor.)
A_ZERO OP A	The APORT exponent is zero.
A_ZERO OP B	The BPORT exponent is zero.
A_SIGN OF EXP	The sign of the BPORT exponent. (Useful for converting from floating to integer format.)
A_OVR OR ANY_OP_ZERO	The result of an exponent subtraction may be too large for the size of the register. (A floating overflow or underflow may result.)
A_UNKNOWN_WRONG_LARGE	Which operand is larger is unknown if the exponents are equal on an effective subtract (anything other than an effective add). However, the second operand may be the larger of the two fractions.
A_FULL NORM	A shift greater than 1 is necessary to normalize. (The microcode must then scan the smaller fraction to find the first 1-bit.)

* Nonstatic Condition Codes

2.3.2.3 **Exponent ALU (XALU)** -- Figure 2-17 shows the location of the main data multiplexers and identifies the main functional areas of the XALU logic:

- XALU Function Decoder (XALF)
- Exponent Register (XREG)
- XALU Shifter (XALS)
- XALU Adder (XALA)

XALU Function Decoder (XALF)

The microcode signals that control the XALU come from the CS1 RAM segment.

- Table 2-38 shows the general XALU functions that are defined by ESXALUFN<5:3>.
- Table 2-39 shows the functions selected by E_SXALUFN<2:0> when E_SXALUFN<5:3> are equal to all zeros.
- Table 2-40 shows the functions selected by E_SXALUFN<2:0> when E_SXALUFN<5:3> contain a code other than zero.

Exponent Register (XREG)

The XREG is a 12-bit recirculating register that consists of a stalled A-clock register (REGXA) and a B-clock register (REGXB). The XREG holds the first operand for a subsequent operation or holds the result for assertion to the BP bus. REGXA is used for condition code (CC) tests that assert XALUCC directly to the IBox (Table 2-41).

REGXB contents can be passed to the BP bus or asserted to the A-side of the adder. The adder can receive the true or inverted value of REGXB, or a value of zero (Table 2-38). A global trap inhibits REGXB from being loaded.

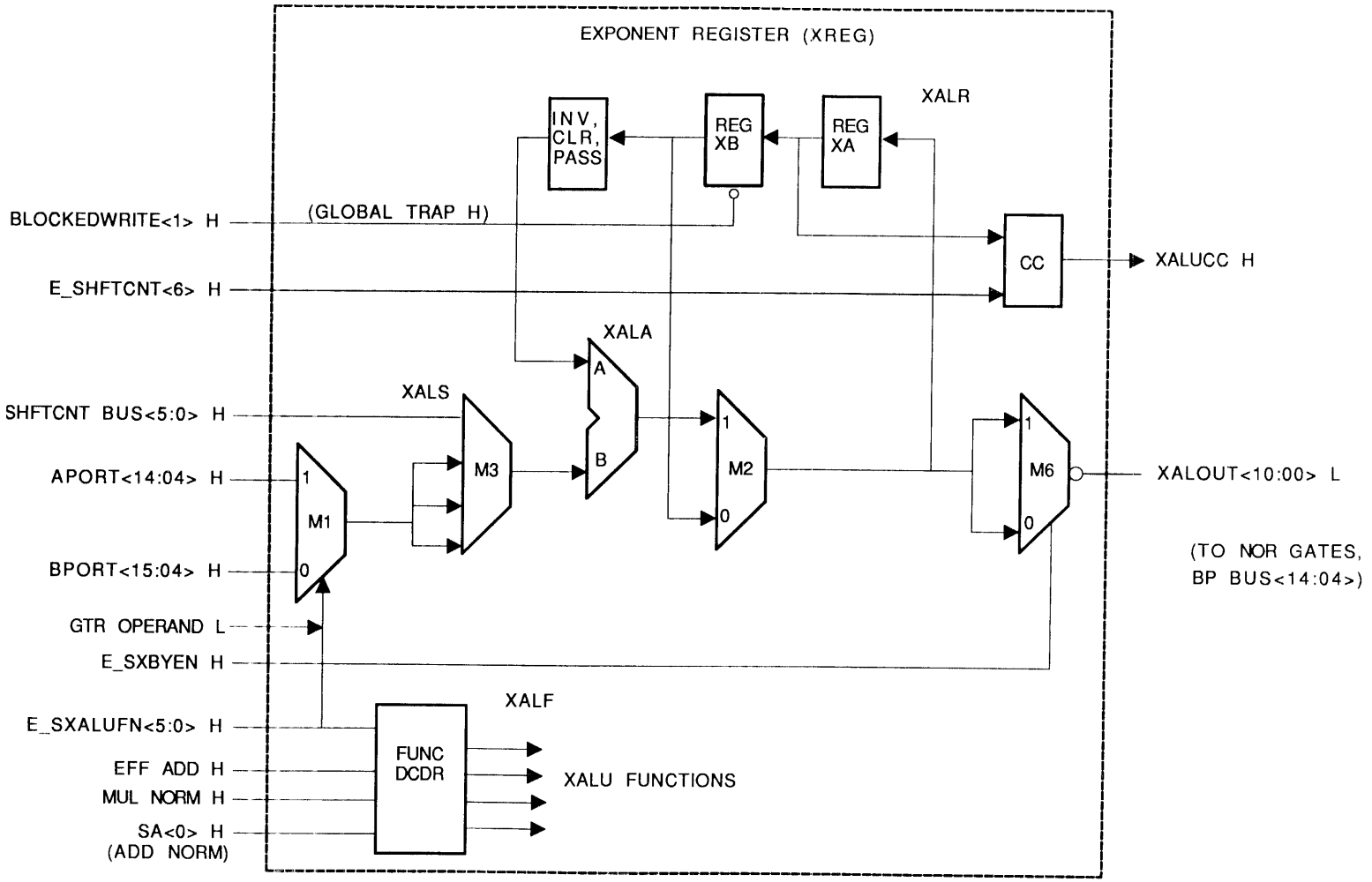


Figure 2-17 Exponent ALU (XALU) Block Diagram

Table 2-38 ESXALUFN<5:3) Control of the General XALU Functions

ESXALUFN <$\overline{5}$ 4 3 2>	Function Name	Exponent Operation or Function
0 0 0 -	Bypass	BP BUS<14:04> <-- XREG<10:00> (See Note 1)
0 0 1 -	Add	XREG<11:00> <-- XREG<11:00> + INPUT<11:00>
0 1 0 -	Increment	XREG<11:00> <-- XREG<11:00> + 1
0 1 1 -	Add-Increment	XREG<11:00> <-- XREG<11:00> + INPUT<11:00> + 1
0 0 0 -	Sel. Exponent	XREG<11:00> <-- Larger of APORT or BPORT value
1 0 1 -	Pass	XREG<11:00> <-- INPUT<11:00>
1 1 0 0	Decrement	XREG<11:00> <-- XREG<11:00> - 2
1 1 0 1	Decrement	XREG<11:00> <-- XREG<11:00> - 1
1 1 1 -	Subtract	XREG<11:00> <-- XREG<11:00> + INPUT$\overline{<11:00>}$ + 1

Notes:

1. The output is passed to the BP bus if E_SXBYEN is asserted by the microcode. Otherwise, the operation is performed but the result is not sent.

2. The Invert/Clear/Pass logic has the following effects on the XREG data. With E_SXALUFN<5:3> equal to:

   1 1 1 — Invert - Assert XREG<11:00> for subtraction.

   1 0 0, 1 0 1 — Clear - Assert zeros to pass the true input value though the adder.

   All Other Codes — Pass - Assert the contents of XREG<11:00> for an exponent operation.

Table 2-39 XALU Functions with E_SXALUFN<5:3> Equal to 000

E_SXALUFN <2 1 0>	General XALU Function	Comments
0 0 0	BP Bus <-- XREG	F_/D_
0 0 1	BP Bus <-- XREG	G_
0 1 0	Reserved	
0 1 1	BP Bus <-- XREG	Integer (No Bias)
1 0 0	BP Bus <-- XREG +1/-1	F_/D_ ADD/SUB Fast Norm
1 0 1	BP Bus <-- XREG +1/-1	G_ ADD/SUB Fast Norm
1 1 0	BP Bus <-- XREG -1	F_/D_ MUL Fast Norm
1 1 1	BP Bus <-- XREG -1	G_ MUL Fast Norm

Notes:

E_SXBYEN must be asserted to send the result to the BP bus. Otherwise, the operation is performed but the result is not sent.

SA<0> from the shift control MCA (SHC) determines whether the XREG contents are sent to the BP bus modified (on a 1) or unmodified (on a 0). If modified, EFF ADD from the SALU then selects an exponent correction of plus one (+1), if asserted, or minus one (-1), if negated.

ADD NORM from the SHF logic or MUL NORM from the MULT logic causes M2 to select the adder instead of the XREG.

Table 2-40 XALU Functions with ESXALUFN<5:3> Not Equal to 000

E_SXALUFN <2 1 0>	XALU Input Function	Comments
0 0 0	INPUT <-- SHFTCNT BUS, 1's Complement	
0 0 1	INPUT <-- APORT, 2's Complement	11 Bits
0 1 0	INPUT <-- APORT, Bias F_/D_Floating	See Note.
0 1 1	INPUT <-- APORT, Bias G_Floating	See Note.
1 0 0	INPUT <-- SHFTCNT BUS	
1 0 1	INPUT <-- BPORT, 2's Complement	12 Bits
1 1 0	INPUT <-- BPORT, Bias F_/D_Floating	See Note.
1 1 1	INPUT <-- BPORT, Bias G_Floating	See Note.

Note:

The SALU performs a reserved operand check if E_SIGNWR is asserted.

Table 2-41 XALU Condition Code (XALUCC) Tests

E_SXALUFN Bits <$\overline{5}$ 4 3 2 1>	Conditions that Assert XALUCC
0 0 0 - -	XREG<11> is set (negative fraction).
1 0 1 - -	
0 1 0 - 0	XREG<11:00> are zero.
1 1 0 - 0	
0 1 0 - 1	The fraction and exponent sign bits are
1 1 0 - 1	not equal and E_SHFTCNT<6> is:
0 0 1 - -	
0 1 1 - -	0 = XOR bits REGXA<08:07> (F_/D_)
1 0 0 - -	
1 1 1 - -	1 = XOR bits REGXA<11:10> (G_/Integer)

XALU Shifter (XALS)

The XALS has two sets of input multiplexers that select the input data and pass it (shifted or unshifted) to the B-side of the adder:

- Input Multiplexer (M1) - M1 passes the selected input port data and format to M3 (Table 2-42). On the first pass, GTR OPERAND from the SALU selects the larger of the two operands.

- Shift Multiplexer (M3) - The M3 outputs are passed to the B-side of the adder (Table 2-43). The M3 input data and format are selected from one of two sources:

  - The true or negative (one's complement) value from the shift count bus.

  - The true or shifted value of the first or second operand from M1.

XALU Output Multiplexers

The XALU has two multiplexers (M2 and M6) that participate in an exponent operation with the adder and XREG, or pass a result to the bypass (BP) bus:

- Adder Multiplexer (M2) -- M2 may pass the complete 12-bit adder output to the exponent register. It may also pass the 11-bit adder or XREG value (bits <10:00>) to M6 for output to the BP bus (Table 2-44).

- Output Multiplexer (M6) -- M6 passes the correct format of the result received from M2 (Table 2-45) and inverts the exponent sign bit back to its complemented value. (The M6 outputs are enabled if E_SXBYEN is asserted by the microcode.)

Table 2-42 M1 Inputs Passed to M3

E_SXALUFN <3 2>	GTR OPERAND	M1 Multiplexer Output Bits <15 14 13 12 11 10 09 08 07 06 05 04>
0 0	1	
0 1	1	APORT<14 14 13 12 11 10 09 08 07 06 05 04>
1 0	-	
0 0	0	
0 1	0	BPORT<15 14 13 12 11 10 09 08 07 06 05 04>
1 1	-	

Note:

APORT<14> is applied to the inputs for both M1 bits <15> and <14>. The larger of the two operands is selected on the first pass.

Table 2-43 M3 Inputs Passed to the Adder B-side

E_SXALUFN <2 1 0>	Input Source	M3 Multiplexer Outputs <11 10 09 08 07 06 05 04 03 02 01 00>
0 0 0	SHFTCNT BUS	1 1 1 1 1 1 <05 04 03 02 01 00>
1 0 0	SHFTCNT BUS	0 0 0 0 0 0 <05 04 03 02 01 00>
X 0 1	M1 PORT (G)	<15 14 13 12 11 10 09 08 07 06 05 04>
X 1 0	M1 PORT (F/D)	<14 14 14 14 14 13 12 11 10 09 08 07>
X 1 1	M1 PORT (G)	<14 14 13 12 11 10 09 08 07 06 05 04>

|
+--- Bias (invert) the sign of the remainder (BIAS REM).

Note: Zeros are asserted to the adder when ESXALUFN<5:3> = 000, 010, 110

The above values are true for all other codes.

Table 2-44 M2 Outputs to M6 or the XREG

E_SXALUFN <2 1>	E_SXBYEN	ADDNORM	MULNORM	M2 Inputs Passed to the Outputs	Comments
E_SXALUFN<5:3> Not Equal to 000					
- -	-	-	-	ADDER<11:00>	
E_SXALUFN<5:3> Equal to 000					
0 0	-	-	-	XREG<11:00>	
0 1	0	-	-	XREG<11:00>	Invert XREG<10>
0 1	1	-	-	XREG<11:00>	Clear XREG<10> for integer operation
1 0	0	-	-	XREG<11:00>	
1 1	0	-	-	XREG<11:00>	
1 0	1	0	-	XREG<11:00>	
1 0	1	1	-	ADDER<11:00>	
1 1	1	-	0	XREG<11:00>	
1 1	1	-	1	ADDER<11:00>	

Notes: Bit <11> is used by the adder, XREG, and condition code (CC) logic but is not sent to the BP bus. ADDNORM is bit <0> of the shift count A-latch (SA<0>) from the SHF. MULNORM comes from the MULT logic.

Table 2-45 M2 Data Passed to the BP Bus by M6

E_SXALUFN<0> Value	Output Format	M6 Outputs to the BP Bus <14 13 12 11 10 09 08 07 06 05 04>
0	G_ or Integer	<10 09 08 07 06 05 04 03 02 01 00>
1	F_ or D_	<07 06 05 04 03 02 01 00> - - -

Note: The M6 outputs are enabled when E_SXBYEN is asserted by the microcode.

2.3.3 Multiplier/Divider (MULT)

The MULT section consists of eight custom multiplier (MULTIPR) chips, each of which operates on one byte of data.

2.3.3.1 Data Interface Signals -- Figure 2-18, Sheet 1, identifies the data input and loop signals used in multiply (MUL) operations. It also shows the output gating that drives the BP bus through the NOR gates.

2.3.3.2 Carry and Control Signals -- Figure 2-18, Sheet 2, identifies the control signals and the carry propagate/generate signals. It also identifies the data shift signals used in divide (DIV) operations.

Table 2-46 describes the microcode bit fields that control the MULT logical and arithmetic functions.

Table 2-47 describes the MULT operations that take place under the control of the E_MULTDIV<4:0> microcode field.

Table 2-48 describes the functions of the MULTIPR chip signal ports as they are used in the MULT logic.

2.3.3.3 Logical and Arithmetic Functions -- The MULT accepts floating operands of up to 64 bits from the APORT and BPORT (although the exponent may be masked).

As shown in Figure 2-18, Sheet 1, the MS byte of the fraction field is loaded to the MS byte of the MULT logic, while the sign and exponent field is loaded to the LS byte. Thus, an 8-bit right shift is accomplished on the load. A MULT load takes 2 cycles, with the guard bits loaded on the first cycle and the operand loaded on the second.

An E_MULDIV code of 1, 2, or 3 (hex) loads the multiplicand or divisor with 64 bits. The sign and exponent field of the operand is forced to zero and the hidden bit is inserted into the LS bit position of the exponent when a "load with mask" is specified.

The NOP code 1E or 1F (hex) pauses the MULT and saves its internal states, allowing a multiply or divide operation to be interrupted and later continued. On a later cycle, the BP bus asserts the quotient or multiplier byte, so care is required on the cycle following a pause to prevent this data from being corrupted.

When reading the quotient is required (E_PEFUNC<2:0> = 6), the SHF must be doing an FP left normalize (E_SHFT<4:0> = 8) by 1 bit. This is so the normalize shift will enable the PE to output the LS byte and the PEN will substitute the quotient (instead of the rounding bit) for the FP left normalize.

When the MULT is enabled and the SHF is not used (when doing a LOAD of the multiplier, for example), the BP bus would normally assert all ones. This is because of the requirement that unused operators assert ones and enable the NOR gates, allowing the active operator to drive the BP bus. However, a special SHF function, CLEAR_MULT (E_SHFT<4:0> = 15), forces the BP bus to all zeros, allowing the ALU to drive the BP bus (with data to be tested for zero).

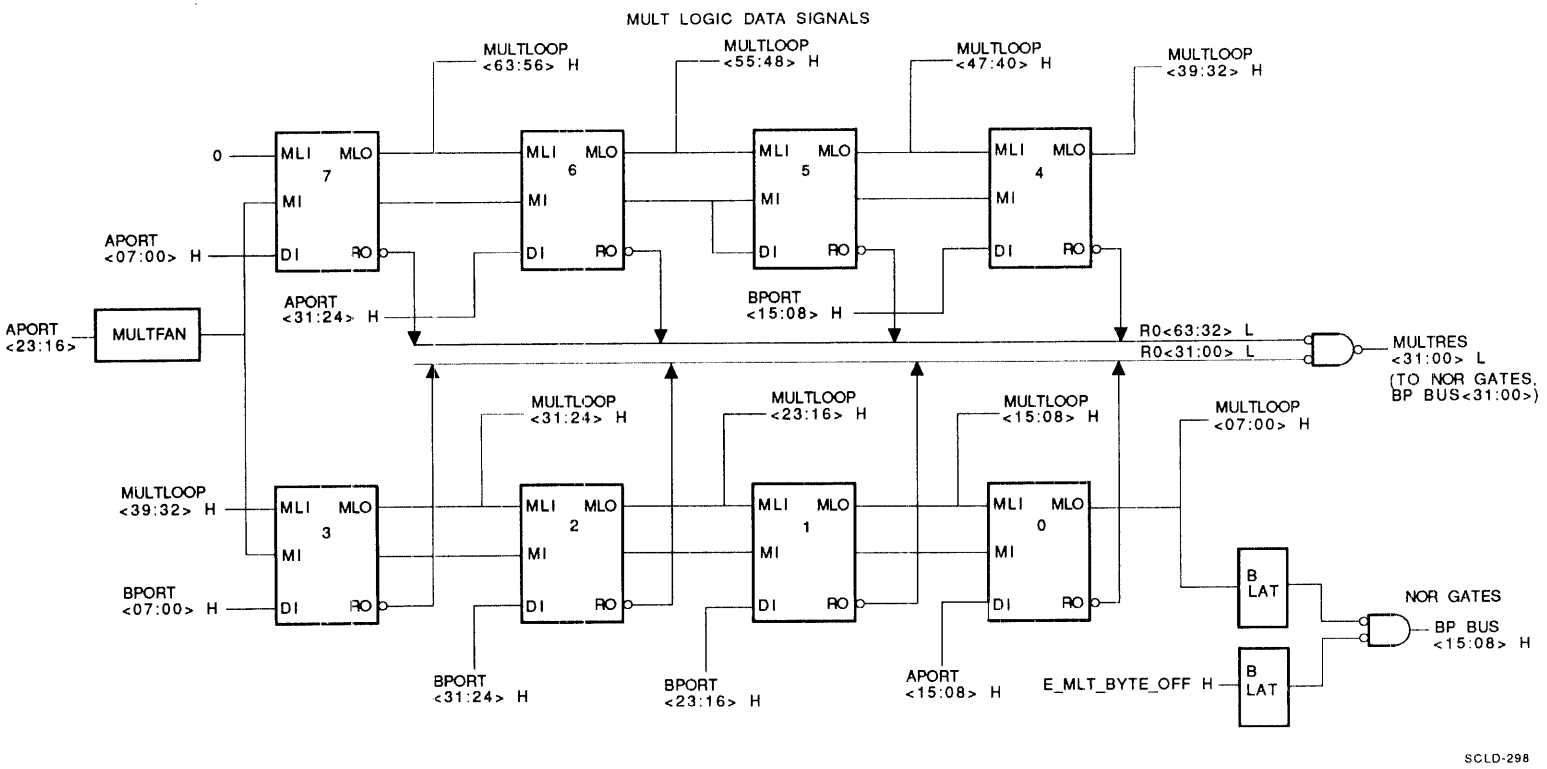


Figure 2-18 Multiplier/Divider (MULT) Block Diagram
(Sheet 1 of 2)

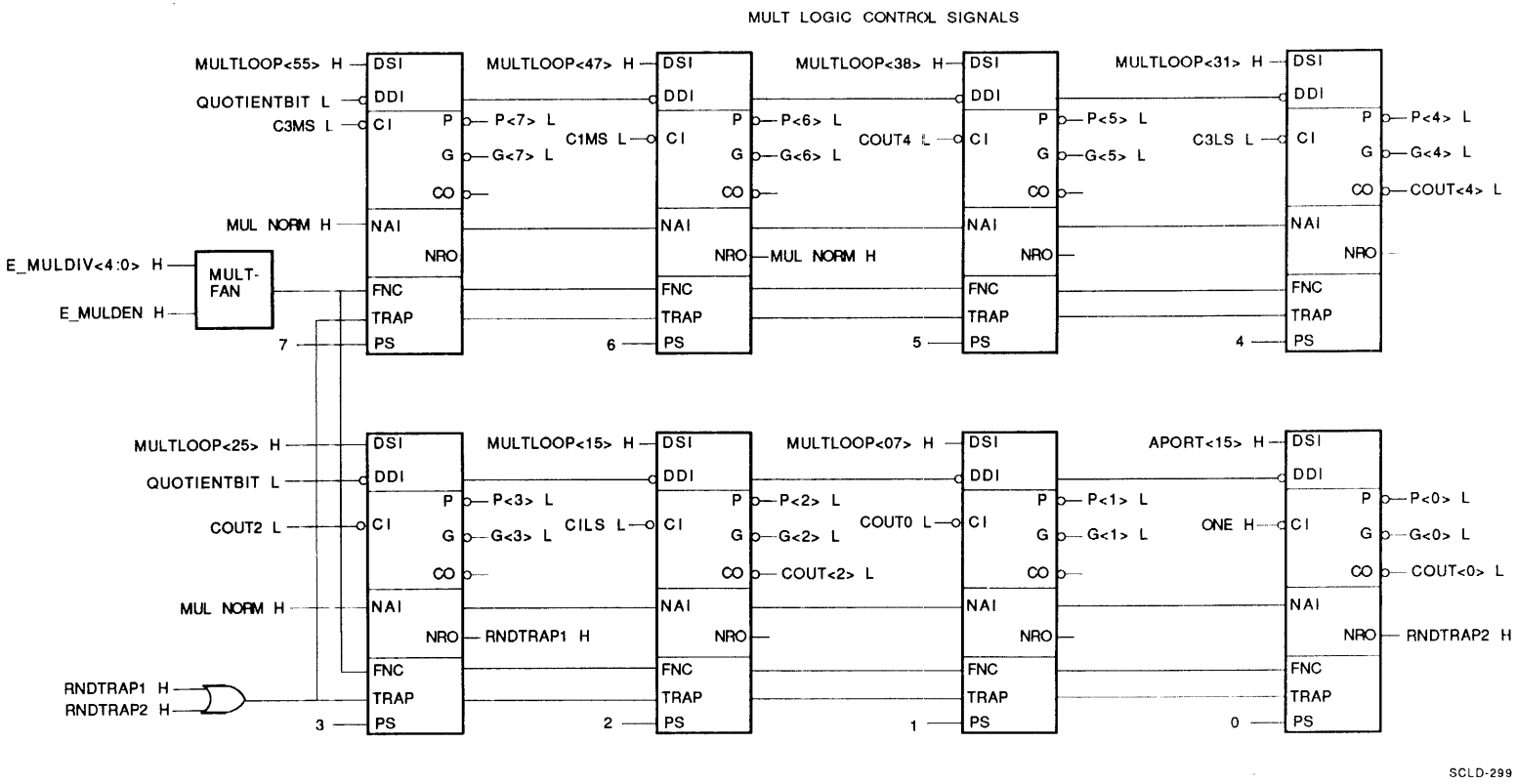


Figure 2-18 Multiplier/Divider (MULT) Block Diagram
(Sheet 2 of 2)

Table 2-46 Multiplier/Divider (MULT) Control Signals from the Microcode

Signal Name	Signal Functions
E_MULDIV<4:0> H	These are the lower five bits of the shift count (E_SHFTCNT) field and are applied to the SHF and XALU from the microcode shift count bus.
E_MULDEN H	This bit has the following effects: 0 = Enables the MULT under the control of the E_MULDIV<4:0> field. 1 = Disables the MULT and preserves internal states in order to resume execution later (Has the same effect as an E_MULDIV code of all 1s.)
E_MLT_BYTE_OFF H	This bit has the following effects: 1 = Inhibits the LS byte from being written to BP BUS<15:08>. 0 = Outputs the LS byte to BP BUS<15:08> if other required conditions are met. (The SHF must be doing an 8-bit FP right shift with E_FPFORMAT<1:0> and E_SHFTFPOUT<1:0> both equal to 3.)

Table 2-47 E_MULDIV Field Control of the MULT Functions

E_MULTDIV Bits <4 3 2 1 0>	Name Mnemonic	Multiply/Divide Function
0 0 0 0 0		Reserved.
0 0 0 0 1	LOAD	Load 64-bit multiplicand/divisor, no masking.
0 0 0 1 0	LOAD G	Load 64-bit multiplicand/divisor with G_format masking; load hidden bit.
0 0 0 1 1	LOAD D	Load 64-bit multiplicand/divisor with F_/D_format masking; load hidden bit.
0 0 1 0 0	STORE F	Store 32-bit normalized and rounded result with F_format masking. Check for Round Trap.
0 0 1 0 1	STORE MS	Store MS 32 bits of the result with no normalize, masking, rounding, or Round Trap check.
0 0 1 1 0	STORE MS D	Store MS 32 bits of normalized and rounded result with D_format masking. Check for Round Trap.
0 0 1 1 1	STORE MS G	Store MS 32 bits of normalized and rounded result with G_format masking. Check for Round Trap.
0 1 0 0 0	LAST MLT	Last multiply cycle.
0 1 0 0 1	STORE LS	Store LS 32 bits of the result with no masking.
0 1 0 1 0	STORE LS G	Store LS 32 bits of the G_format result.
0 1 0 1 1	STORE LS D	Store LS 32 bits of the D_format result.
0 1 1 0 0	FIRST MLT	First multiply loop. (Microcode supplies the LS multiplier byte.)
0 1 1 0 1	MAIN MLT	Main multiply loop. (Microcode supplies the next multiplier byte.)

Table 2-47 E_MULDIV Field Control of the MULT Functions (Cont)

E_MULTDIV Bits <4 3 2 1 0>	Name Mnemonic	Multiply/Divide Function
01110 to 10111		Reserved.
1 1 0 0 0	LOAD DD	Load Dividend and do first division loop.
1 1 0 0 1	MAIN DIV	Main division loop.
1 1 0 1 0	LAST DIV	Last division loop.
11011 to 11101		Reserved.
11110 and 11111	NOP	Preserve internal states and inhibit writes to the BP BUS.

Table 2-48 MULT Logic Signal Port Function Description

MULTIPR Port Name	Signal Function
Data I/O Signals	
Multiplier Loop In/Out (MLI/MLO)	On a multiply cycle, each byte of the accumulating product is right-shifted to the next LS byte.
Data In (DI)	A multiplicand or divisor of up to 56 bits may be loaded through the DI port. The MS word is applied to the APORT and the LS word to the BPORT, and the exponent is masked out. For a DIV operation, the divisor is loaded first. The dividend is then loaded through the DI port on a following cycle.
Multiplier In (MI)	For a MUL operation, one byte of the multiplier is applied to the MI input on each cycle, starting with the LS byte. Each multiplier byte is supplied by the microcode on APORT<23:16> and is asserted to all chips from the MULT fanout (MULTFAN) logic.
Result Out (RO)	The MS 32 bits (RO<63:32>) and the LS 32 bits (RO<31:00>) of the final result are passed to the BP bus on separate cycles.
Divider Shift In (DSI)	On a divide cycle, the accumulating quotient is left-shifted one bit at a time to the next MS byte.

Table 2-48 MULT Logic Signal Port Function Description (Cont)

MULTIPR Port Name	Signal Function
Carry and Look-Ahead Signals	
Carry-Out (CO), Carry-In (CI)	A carry-out from byte 4, 2, or 0 is passed directly to the carry-in port of the next MS byte. A carry-in for each of the other bytes comes from the carry look-ahead logic.
Propagate (P), Generate (G)	The carry propagate and generate signals from all chips drive the carry look-ahead logic.
Divider Data In (DDI)	QUOTIENTBIT L is passed from the carry look-ahead logic to all chips: The first cycle of a divide subtracts the dividend from the divisor. If the result is negative, QUOTIENTBIT is asserted and the next cycle does an add of the partial remainder. If the result is positive, the next cycle does a subtract.
Control I/O Signals	
Normalize In (NAI)	MUL NORM is asserted if the MS bit is not in the hidden bit position, causing a 1-bit left shift.
Normalize/Round Out (NRO) and TRAP	When RNDTRAP is asserted by byte 3 or 0, the internal states of all chips are preserved and the RO outputs disabled. The MULT is thus disabled for the next 3 cycles (during pipeline latency).
Function (FNC)	E_MULDIV<4:0> and E_MULDEN from the microcode are applied to all chips through the MULT fanout (MULTFAN) logic. (See Tables 2-46 and 2-47.)
Position Select (PS)	Each chip has a 3-bit PS input hardwired to the code that defines its order of byte significance. (Byte 7 is MS; Byte 0 is LS.)

2.3.3.4 Multiplier Operation -- Starting with the LS multiplier byte, the microcode supplies the stored multiplicand with a multiplier byte on each cycle, generating a new partial result. Each multiplier byte is applied to the MULT on APORT<23:16>, and each partial result is passed to the next MS chip on the multiplier loop (MULTLOOP) lines.

The first multiplier byte is required on the first multiply cycle using E_MULDIV code C (hex). Thereafter, each of the next higher bytes is applied on each of the following main multiply cycles using code D (hex). On the last multiply loop, using code 8 (hex), a byte containing the multiplier sign bits for all bytes is sent to the multiply logic (the multiplier must have been sign-extended by one byte to obtain the required sign bits). (For floating-point, the mantissa is always positive so a byte of all zeros can be sent.)

At the end of the first and main multiply loops, the LS byte of the product generated thus far is passed to the BP bus on bits <15:08>. (Because this byte is sent during register write time of the current microinstruction, it appears to be on the bus in the middle of the ALU cycle of the next microinstruction. For a floating-point multiply, this byte is usually discarded. For an integer multiply, this byte represents part of the required result, with a MICROCODE RESTRICTION that no traps may be provoked if this is sent to the BP bus.)

The MS part of the result is stored in the multiplier logic after the last multiply cycle and can be sent over the BP bus using the STORE function codes 4--7, and 9--B (hex). The internal result is 64 bits long but the MS and LS 32 bits can be sent to the BP bus on separate cycles. The output can also be masked, allowing the sign and exponent fields from the SALU and XALU to be merged with the multiplier result on the BP bus.

The result can also be normalized and rounded. If a normalization is performed, the hardware signal MUL NORM is asserted to the XALU to correct the exponent by a decrement of 1. A rounding increment is only performed on the LS byte of the product if a carry-out from the rounded result occurs, producing a round trap. The ROUND TRAP microsubroutine then fixes the result by propagating the carry through the remainder.

Because the G_format mantissa is not byte-aligned, it is necessary to first prescale one of the operands of a G_ times G_format multiply. This involves a shift of either the multiplicand or the multiplier by three bits left, then three bits back, and filling with zeros.

Two cycles are required to store an F_format multiply result with normalization and rounding. This is because the rounding occurs late in the multiply cycle. The STORE order is performed twice, with the result of the first STORE being ignored and the result of the second STORE being valid.

2.3.3.5 Divider Operation -- The division algorithm is based on a nonrestoring technique that generates one quotient bit per cycle. Both operands must be positive.

To start the division algorithm, the divisor is first loaded using code 1, 2, or 3 (hex), the same functions used to load the multiplicand.

The dividend is then loaded and the first division cycle is performed using code 18 (hex). A hardwired left shift of 8 bits is involved in this load. To compensate, the dividend must first have been right-rotated by 8 bits. At the end of the first division cycle, the MS quotient bit is output at register write time and written to BP BUS<16>. The division algorithm then continues, using the main division loop function code 19 (hex), which generates one quotient bit per cycle.

On the last cycle, the last divide loop is coded and the remainder is generated. The remainder is correct if it is positive. If not, it must be corrected by adding the divisor to it. The quotient then requires that a value of 1 be subtracted from it (if the remainder was corrected). Function codes 5 and 9 (hex) are then used to output the remainder.

For an aid in implementing the H_format divide, it is possible to divide an arbitrarily large dividend by the 56-bit divisor. On each division cycle, APORT<15> is shifted into the LS bit of each partial result (under normal operation these bits should be zero). The MS part of the dividend is initially loaded into the multiplier/divider. Thereafter, the remaining bits are shifted in one bit at a time on each successive cycle.

The LS part of the dividend must be shifted left by 1 bit on each division cycle. This works in conjunction with the 1-bit shift required for quotient accumulation. As the dividend is shifted in from its MS end, the new quotient bits are shifted into the LS end. By supplying the correct part of the dividend at the right time, it is possible to perform divides of large dividends.

SECTION 8
CACHE BOX LOGIC (CBOX)

CHAPTER 1
INTRODUCTION

1.1 CACHE BOX SYSTEM DESCRIPTION

As Figure 1-1 illustrates, the VAX 8800 cache box (CBox) consists of three subsystems:

1. Translation buffer (TB)
2. Cache
3. NMI interface

The TB is a hardware mechanism that speeds virtual address-to-physical address translations.

NOTE
Cache and memory accesses cannot be made without a physical address.

The TB holds calculated address translations for future use. If the translation for a virtual address exists in the TB, the data is taken as the physical address. And, if the required address translation is unavailable, a microtrap will occur and cause a trap routine to perform the required translation and write it into the TB. Subsequent use of the virtual address reads the address translated as data out of the TB.

In addition to the physical address, the TB also holds physical page control information (that is, protection field and modify bit). It performs a hardware check on these functions.

The TB receives a virtual or physical address as VA <31:00>, plus an instruction of how to process the address as CACHE CMD MDNUM <03:02> from the EBox. Within the TB, VA <31:00> bits cause an address vector to be produced as PA <15:02>. This vector points to a data entry in the cache.

The TB also produces a PA3 <29:16> address that is used in TAG and matching MCAs in the cache. The PA3 <29:00> output of the TB is sent to the NMI interface where it is stored whenever a cache reference is made and is used to write the cache with refill data when a cache miss occurs. The TB receives refill/invalidate addresses as REF/INVAL <28:02> from the NMI interface when cache miss data is received from the NMI as NMI DATA <31:00>.

The cache is a hardware mechanism that provides the CPU fast access to frequently-used data. It is addressed by a physical address vector (PA <15:02>) produced in the TB. If the referenced data is in the cache, it will be read from the cache and no memory request is required. However, if the referenced cache data is unavailable, a request to memory is made to obtain it. Returning refill data is sent to the requester and is also written into cache. Subsequent use of the data causes it to be read out of the cache.

The cache is "read allocate only"; a new cache location is created only after a read miss occurs, and a write updates the cache if it is a cache hit. However, if a write does hit in cycle, the next possible cache request is stalled. Instead, a "delay write algorithm" is used to update the cache on the next write cycle.

The cache is written by the EBox (by means of WBUS <31:00>) and refilled by the NMI interface with MD BUS <31:00> data by means of the MD BUS. It sends data to the EBox ALU as A CD BUS <31:00>. It also sends CACHE DATA BUS <31:00> data bits to the EBox register file and to the IBox.

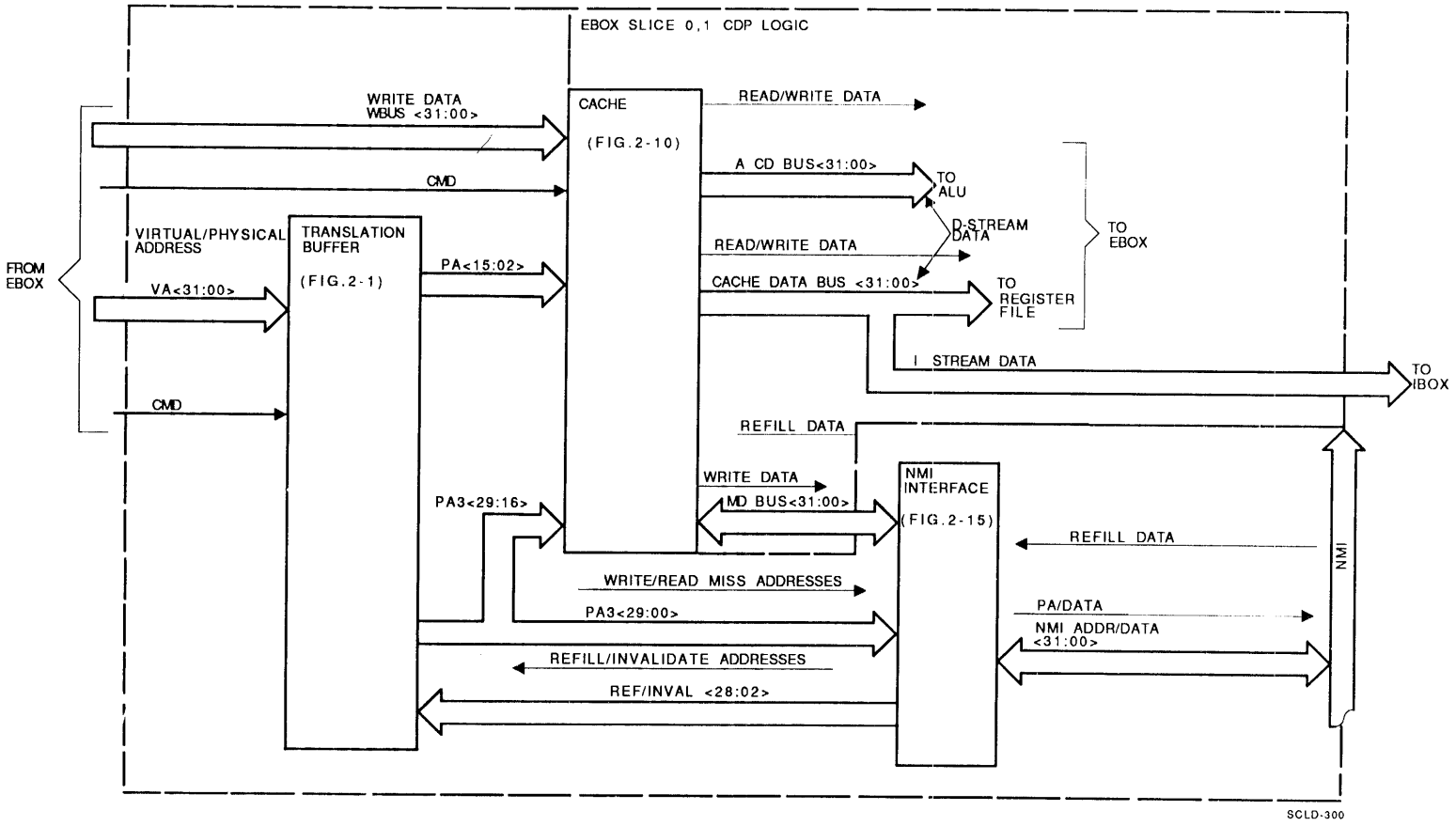


Figure 1-1 CBox - Block Diagram

The NMI interface is the CPU interface to memory and I/O subsystems. All requests to memory and I/O subsystems are processed by the NMI interface.

The NMI interface provides the control and data path by which the CPU communicates on the NMI. When a cache read misses, the NMI interface uses the read-miss address to build an NMI command/address transaction and then sends the address to memory. The TB and cache then become free to process additional CPU requests, while the NMI interface handles the transaction to memory. When memory data arrives, the NMI interface:

1. Takes control of the cache
2. Loads the data into the cache data store
3. Validates the cache tag valids with the new tag address.

When the CPU executes a write, the NMI interface transfers the write data to memory without making the CPU wait for a write completion to memory.

The destination of cache or memory data defines its usage. Any data destined for the EBox is data stream, or D-stream data. Data destined for the IBox is instruction stream, or I-stream data. I-stream data is held in a 4-longword instruction buffer (IB). The content of the IB is always interpreted as macroinstructions by the IBox. It is used by the IBox decoder as the basis for forming entry points into the VAX 8800 microcode and, thus, starting microcode execution. Normally, an I-stream refers to the contiguous set of longwords within a page currently being fetched. A "new" I-stream is when the next longword being fetched is not the next contiguous one.

Data destined for the EBox is used as data or an address calculation. It can only be requested by microcode and is always written into a memory data (MD) register. The MD number is a register specification field in the microcode and is used to select one of eight possible MD registers for the destination of the read data.

All D-stream read accesses must specify a known MD register number; the MD number must be sent along with the D-stream data to the EBox.

The EBox uses the MD number to write the data into the selected register in a GPR file. When microcode requests a cache read destined for an MD register, the MD register is invalidated by EBox control logic; the MD register becomes valid when the data returns. Whenever a reference is made to an EBox MD register that it is invalid, the CPU MD stalls, and waits for the read.

1.2 CBOX OPERATION

The CBox operates per instructions in the VAX 8800 microcode. These functions consist of data accesses (that is, read or write and "housekeeping" functions that support the CBox hardware). The CBox-specific microcode that is used to request CBox functions is the CACHE command.

Figure 1-2 illustrates CBox cycle timing. The CACHE command is received at T7. The half-cycle T7-T8 is used to decode the command and form the appropriate control signals. The address associated with the access is available at T9 from the ALU. The T9-T10 half-cycle is used to look up the TB, and the physical address becomes available at T9. The cache tag and data store are accessed during T9-T10. At T10, the CACHE HIT signal and cache data are available. The cycle T8-T10 of a microinstruction is the CBox (or cache) cycle for that instruction. Every microinstruction has a cache cycle at the canonical time. The instruction and the CBox determine where a microinstruction performs any function.

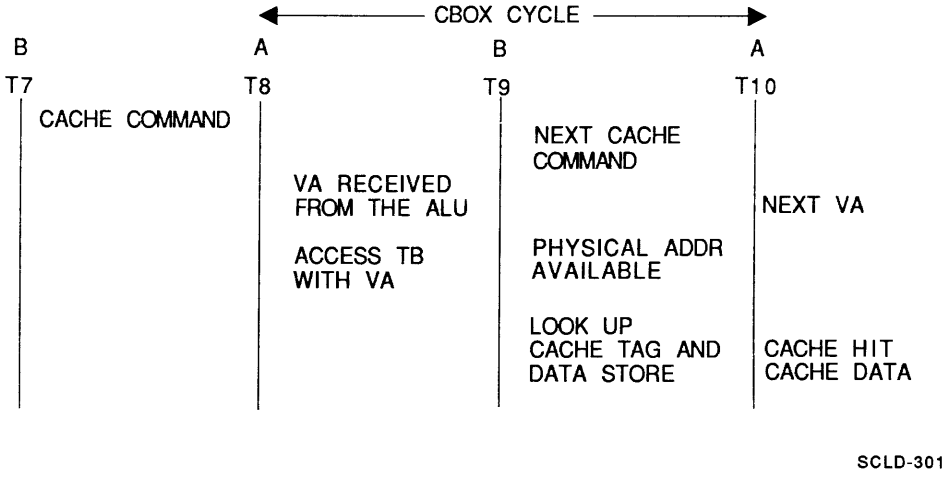


Figure 1-2 CBox Cycle Timing

Basically, the address and command flow is similar for all CACHE commands. If a read miss occurs, the NMI interface must process the read miss and eventually send a read transaction out to the memory to obtain the required data for the cache. All write requests are processed by the NMI interface. Whether a CACHE command needs to be processed by the NMI interface does not affect microinstruction processing, unless the NMI interface becomes full or read data is used before it is received from memory.

The cache microfield consists of several overlapping fields, depending on the function of the field. Each valid field combination specifies the following information:

- CACHE command - An encoding of the type of CBox function being requested (for example, read, write). Additionally, it also specifies whether a physical address or virtual address is being used.
- TB command - Decoded to control the generation of memory management traps.
- MD number - Only relevant for read cycles, when it specifies the destination for read data. On any other type of cycle, a nonvalid MD number is specified.
- Size - Access size, if relevant.

The CBox is transparent to the microcode. Whether the data being written to (or being delivered) is in memory or in the cache does not affect its functionality.

Relative to the CPU, the CACHE command part of the microinstruction is completed in one cycle; a CACHE command can be presented to the CBox on every machine cycle. The microcode and the CBox determine whether any function is performed as a result of this. If the CBox is unavailable for a particular microinstruction, the CPU will stall. The default state of the microfield is interrupted as a "no operation" request.

1.2.1 CBox Cycles

The CBox performs certain functions, or cycles, based on the command it receives in the VAX 8800 microcode. Table 1-1 lists and describes the CBox cycles.

Table 1-1 CBox Cycles

CBox Cycle	CPU Command	NMI Function	PIBA
Quiescent	Noop	None	Missed in cache
Read	Read	None	No arbitration *
Write	Write	None	No arbitration *
PIBA	Noop	None	Yes
TB command	TB command	Don't care	Yes
Refill	Stall if cmd**	Refill	No arbitration *
Invalidate	Stall if cmd**	Invalidate	No arbitration *
Register returns	Stall if cmd**	None†	No arbitration *

* PIBA does not have arbitration; it does not make an access.

** CPU command is overridden. If there is a CPU command, the CPU will stall until the NMI function completes. Not applicable to TB commands.

† Not really an NMI activity, but a completion of the register read that uses the NMI address/data paths.

1.2.1.1 Quiescent State -- The quiescent state is a control mechanism the PIBA accesses for I-stream data delivery. The following conditions set the quiescent state:

- Quiescent on the last cycle - The state was set and no condition for clearing the state has occurred.
- LD PC or PIBA cache miss - Whenever an I-stream fetch in the cache misses, additional fetching is stopped until the quiescent state is cleared.
- Page cross - When an I-stream fetch crosses a page boundary, further access is prevented until an LD PC command sets up the new I-stream fetch on another page with a new translated address.

The following conditions clear the quiescent state.

- LD PC with cache hits - This starts a new I-stream fetch.
- Second octaword of a current IB refill return - The IB has been incremented, the refill data has been sent to the IB, and further PIBA accesses can be made without conflicts in I-stream fetching.

1.2.1.2 Read Cycle -- A read cycle can be requested only by microcode. The microcode specifies an MD number, which is the destination of the read data and the size of the read. The address of the read is latched at the input of the TB.

If memory management errors are detected, a trap is generated. If the address does not cause a trap, and the read data is in the cache, the read is a cache hit and it takes only one cycle to complete. The microcode request is received at T7 and decoding is started. The VA is taken from the latched output of the TB at T8. The address translation RAM in the TB is "looked up" in the T8 cycle.

At T9, the physical address is selected and latched at the output of the TB. During T9, a tag and data store in the cache are addressed. The data becomes available at T10. If the read caused a cache hit, the data is taken by the EBox and eventually written to the specified MD register. If the read misses in the cache, the CBox NMI interface initiates a request to memory for the data.

1.2.1.3 Write Cycles -- Write cycles are requested by microcode. The microcode specifies the write size, but does not specify an MD number because read data is not expected. The address is taken from a VA latch at the TB input, and the data is supplied to the cache by means of the WBus.

Writes can be with a virtual or physical address. A virtual address requires use of the TB for translation. If any memory management problems occur, a trap will be generated. Otherwise, the TB supplies the physical address needed for the cache access. If the EBox specifies a physical address, the TB is bypassed and the address is taken directly out of the VA Latch at the TB input.

The CBox implements a buffered write-through algorithm to control writes to memory. Write-through means that writes go out to memory as soon as possible. Buffered means the NMI interface collects these writes, buffers them, and then sends them to memory.

The CBox implements a delay-write algorithm. Two cycles are required to complete a write that updates the cache. The first cycle is used as the look-up cycle to determine if the write hit in the cache. If the write hit, the write must update the cache. And, if the write did not hit, the write does not update the cache.

If two write cycles occur consecutively, any CBox request that follows the write must stall for one cycle while the write completes the update of the cache. Thus, the CBox implements a "delay write algorithm". If a write hits in the cache it updates the cache only during the next occurrence of a write (that is, the write to cache is delayed until the next write).

The first cycle of the write performs a cache look-up. During the following cycle, the physical address of the write is loaded into a delay-write address buffer (in the TB) and the data is loaded into a delay-write data buffer (in the cache). If the next cycle is a read, both the write address in the delay-write address buffer and the data in the delay-write data buffer remain unchanged. If the next cycle is a write, the write performs a look-up by addressing a tag store in the cache. The delay-write address in the TB is then sent to the cache data store, and the data in the TB delay-write buffer is written to the cache, if it was previously a hit.

NOTE

If the write was previously a miss, cache control logic prevents the write from updating the cache.

1.2.1.4 The PIBA -- The PIBA refers to the contents of a PIBA address latch in the TB. The PIBA always holds the address of the next longword of I-stream data to be fetched. On each successful delivery of IB data, the PIBA in the TB is incremented to point to the next longword. The TB is not used when the PIBA addresses contiguous longwords within the page boundary. If the PIBA crosses a page boundary, it will not be incremented any further, and the IB will be informed. The memory management microcode then issues a LD PC command to start a new I-stream fetch on the new page.

1.2.1.5 TB Cycles -- TB cycles make exclusive use of the TB. They have three functions:

1. Writing the TB
2. Flushing the TB
3. Checking the TB

Writing the TB means loading the page table entry (PTE) into the TB as the result of a TB miss. The VA latch is taken as the source of the address and the data.

The TB can be flushed. This means invalidating one or more TB entries. Any reference to an invalidated TB entry causes a TB miss. Typically, this is used to clear the TB after a CPU reset, or after a context switch.

1.2.1.6 Refill Operation -- When a reference misses in the cache, the CBox sends a read request for the data to memory or an I/O device. Data returned by memory or an I/O device, is termed the refill, or returned data. For the duration of the returning data, the CBox performs refill cycles, passing data to the IBox or EBox and writing the data into the cache data store. The number of longwords returned, and, hence, the number of cycles required, can be one longword or two octawords. The size of the refill expected depends on the type of the original read reference.

A read request is sent to memory or an I/O device because of:

- A D-stream read miss
- An I-stream read miss
- An I/O read

The address for the refill is the physical address. It is used to address the cache tag and cache data store during the cache refill cycles.

Data is returned from memory on the NMI address/data lines. The received data is sent (by means of the MD-bus) to the cache data store and the cache data bus. The first longword is always the data missed in the original reference. For D-stream or I/O reads, this is always passed to the EBox. I-stream data is sent to the IBox only if it can accept the data. All the refill data is written to the cache data store and set valid, except if:

- The refill is for an I/O device
- The cache is off, or
- An error occurred during the refill

The size of the refill data determines the number of refill cycles required. The CBox determines the size of the refill expected by the type of read reference made. If it has been waiting for a longword return and, instead, a hexword is returned, this is an error condition.

A hexaword refill requires a minimum of 10 cycles for completion. It consists of four data consecutive cycles (termed "first-octaword refill"), a minimum gap of two nonrefill cycles, and then four additional, consecutive data refill cycles (termed "second-octaword refill"). A refill size of one longword takes only one cycle to complete.

1.2.1.7 Invalidate Cycle -- The NMI interface interprets writes on the NMI, with any ID other than its own, as an invalidate cycle. The address accompanying the write is taken from the NMI and then sent to the TB as the source for the cache physical address. This is then used as the physical address to look up the cache tag store.

Invalidates can take one to two cycles in the CBox. The first cycle is used as the look-up cycle for a cache hit. If the invalidate address misses, then the block being modified is not in the cache and no further action is necessary. If it was a cache hit, then modified data is in the cache and it must be invalidated; subsequent accesses must access the data from the memory.

The address in the refill/invalidate cycle is made available for two cycles. The cache data store is not affected during a invalidate cycle.

1.2.2 CBox Stalls

A stall is a CPU condition, whereby the current set of microinstructions in the pipeline cannot continue processing due to unavailability of certain resources or data.

Each latch in the CPU that is required to retain information (for the current instruction) when the CPU stalls, is clocked by a "stalled A clock." During normal operation, the stalled A clock functions as A clocks. When a stall happens, the stalled A clocks are blocked, thereby retaining the state of the "stalled" latches the cycle before the stall occurred. Thus, microinstructions that cause a stall can be completed after the stalling conditions are cleared.

There are two types of stalls:

1. VA stalls

2. MD stalls

VA stalls occur when the CPU requests a cache cycle but the cache data path is being used for other functions (for example, refill, invalidates, or register returns). The CPU must stop and present the address again during the next cycle.

MD stalls occur when the CPU tries to use some data that was requested by a previous cycle and the data has not yet arrived from the cache (for example, because the reference missed in the cache). The CPU must wait until the data arrives and then execute the microoperation.

NOTE

The CPU can wait for more than one MD at the same time.

2.1 CBOX SUBSYSTEMS DESCRIPTION

2.1.1 Translation Buffer

Because the VAX 8800 system uses virtual addresses, every virtual address must be translated to its actual physical address in memory; the CBox translation buffer (TB) performs the required translations. However, the CBox TB does not always perform address translations; sometimes it receives physical addresses that require no translation.

The translation buffer receives:

- Read and write addresses from the EBox
- Refill/invalidate addresses from the NMI interface

It produces the following types of address vectors:

- Read
- TB read miss
- Cache read miss
- Write
- Delay-write algorithm
- Physical instruction buffer
- Cache TAG

The TB data inputs consist of:

- Refill/invalidate data (addresses) from the NMI interface
- Write data for the TB RAM

As Figure 2-1 illustrates, the TB consists of:

- An input VA latch
- A TB RAM
- A TB match MCA
- A TB RAM bypass
- A PA latch

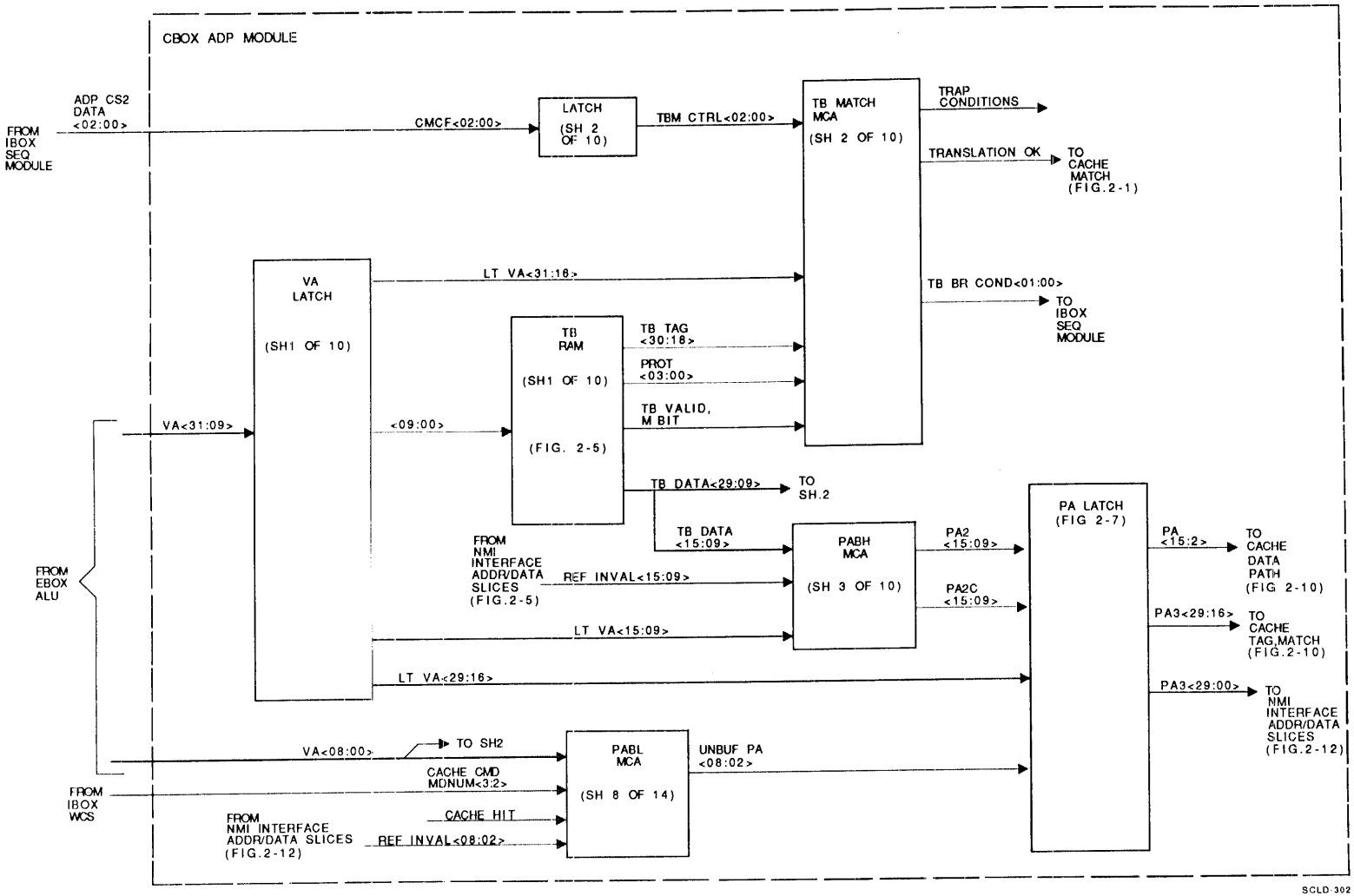


Figure 2-1 Translation Buffer - Block Diagram

Basically, a virtual address is received as VA bits <31:00> from the EBox, plus cache command CMCF <02:00> field bits that determine TB operation. The virtual address in VA bits <31:09> is held in an input VA latch and then distributed to the TB match MCA, the TB RAM, and the TB RAM bypass logic. If the TB RAM contains a physical-to-virtual address translation for the EBox VA <31:00> input:

- The required physical address is accessed from the RAM.
- The accessed translation is checked for validity in the TB match MCA.
- Presence of the following fault conditions are checked:
  - Page crossing
  - Access violation
  - Memory trap
  - Modify bit trap
- A physical address vector is assembled in the PA latch and sent to the cache data store as PA <15:02> bits.

However, when a physical address is received from the EBox, the TB RAM is not used; the TB RAM bypass logic produces the physical address input to the PA latch.

When the TB RAM is to be used for an address translation, and a check determines that it does not contain a valid translation, the EBox writes the data into the TB RAM and the physical-address-access procedure is repeated to produce the required cache data store physical address vector.

Of the virtual/physical address VA<31:0> bits received from the EBox:

- VA <31:09> specifies a virtual page frame number (VPN).
- The corresponding physical translation for the VPN is the page frame number (PFN).
- Data within a page (virtual or physical) pointed to by the PFN is specified by VA<8:0>.

Thus, a virtual address input is comprised of a VPN (VA <31:09>) and VA<8:0>, and a full physical address is comprised of the PFN and VA<8:0>. Figure 2-2 illustrates the virtual address fields.

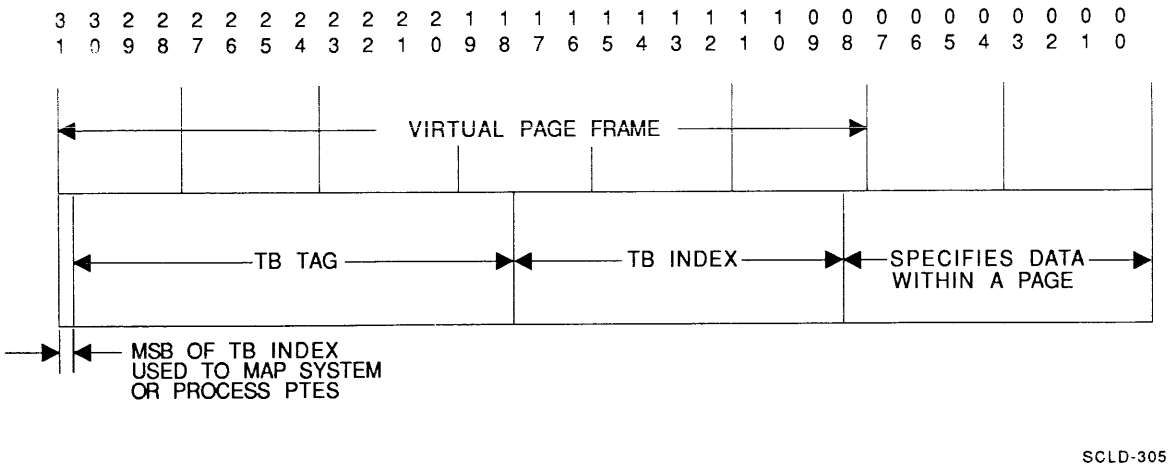


Figure 2-2 Virtual Address Fields

The PFN is a part of a page table entry (PTE) data structure that holds information specific to the page. The CBox TB RAM, in addition to holding the PFN, also holds a subset of the information included in the PTE (the protection field and the modify bit).

2.1.1.1 VA Latch -- The VA latch (Figure 2-1) is the holding mechanism for high-order bits <31:09> of the VA <31:00> input received from the EBox. It is clocked with a STALLED A CLK and loaded by means of LD VA EN from the PABL MCA. During a VA stall, it holds bits <31:09> until the cache can process the command.

As Figure 2-1 illustrates, the VA latch sends:

- LT VA <31:16> bits to the TB match MCA where it is compared with the TB RAM TB TAG <30:18> output bits to determine if a TB hit has occurred.
- VA <31,17:09> bits as FVA <09:00>, TVA <09:00>, SVA <09:00>, and PVA <09:00> to the TB RAM to access a translated (physical) address and also produce control/status signals.
- LT VA <29:16> bits to the PA latch.

2.1.1.2 TB RAM -- The TB RAM provides a cache of 1K calculated virtual-to-physical page table entries (PTEs), consisting of 512 system PTEs and 512 process PTEs. It receives data as bits LT VA <30:18> and addresses as bits FVA <09:00>, TVA <09:00>, SVA <09:00>, and PVA <09:00> from the VA latch, plus write enables and a flush signal from the cache control MCA (Figure 2-10).

The TB RAM directly maps the VPN bits <31:09> into 1K TB entries. Bit VA<31> is used to map one half of the TB RAM for system translations and the other half for process translations. Bits VA<17:9> are used to map the system and process PTEs into the 512 TB RAM entries.

NOTE

As the TB RAM translates pages, bits VA<8:0> do not need to be translated.

Whether or not the TB RAM is used for an address translation is determined by setting a memory management enable (MME) bit that is received from the cache control logic. When the MME bit is set, all virtual address references use the TB RAM for translations and memory management traps are allowed to occur (unless otherwise defined in the TB command). When the MME bit is not set, the TB RAM is not used for translations; all addresses are taken as physical addresses from the TB RAM bypass logic and memory management traps are blocked from occurring.

Within the TB RAM, the four sets of bits <09:00> received from the VA latch are used to access a translated address that was written into the RAM, and perform checks on the access being made. And, if the VA <31:09> input from the EBox contains a virtual address, the TB RAM is used to produce a PA1 output. The PA1 output comprises the upper physical address bits <15:09> that are used to form the PA <15:02> address vector for the cache data store.

When the EBox sends a physical instead of a virtual address (in the VA <31:00> input), the TB RAM is not used to produce bits <15:09> for the cache address vector. Instead, the TB RAM bypass logic is used; bits LT VA <15:09> from the VA latch are used in the PABH (physical address bypass - high) MCA to produce the upper bits (PA2 <15:09>) of the physical address. This would then be combined with the buffered output PA <08:02> of the PABL (physical address bypass - low) MCA to produce the PA <15:02> address bits for the cache.

Whenever a translation access is made, the TB RAM TB TAG <30:18> bits are compared against the LT VA <31:16> bits in the TB match MCA to determine if the address produced by the translation buffer is valid data. And, if it is determined to be valid, the TB match MCA sends a TRANSLATION OK signal to the cache match MCA. The cache match MCA then determines if the cache data entry being pointed to by the physical address vector contains the required data.

Figure 2-3 is a simplified diagram that illustrates the TB RAM write sequence.

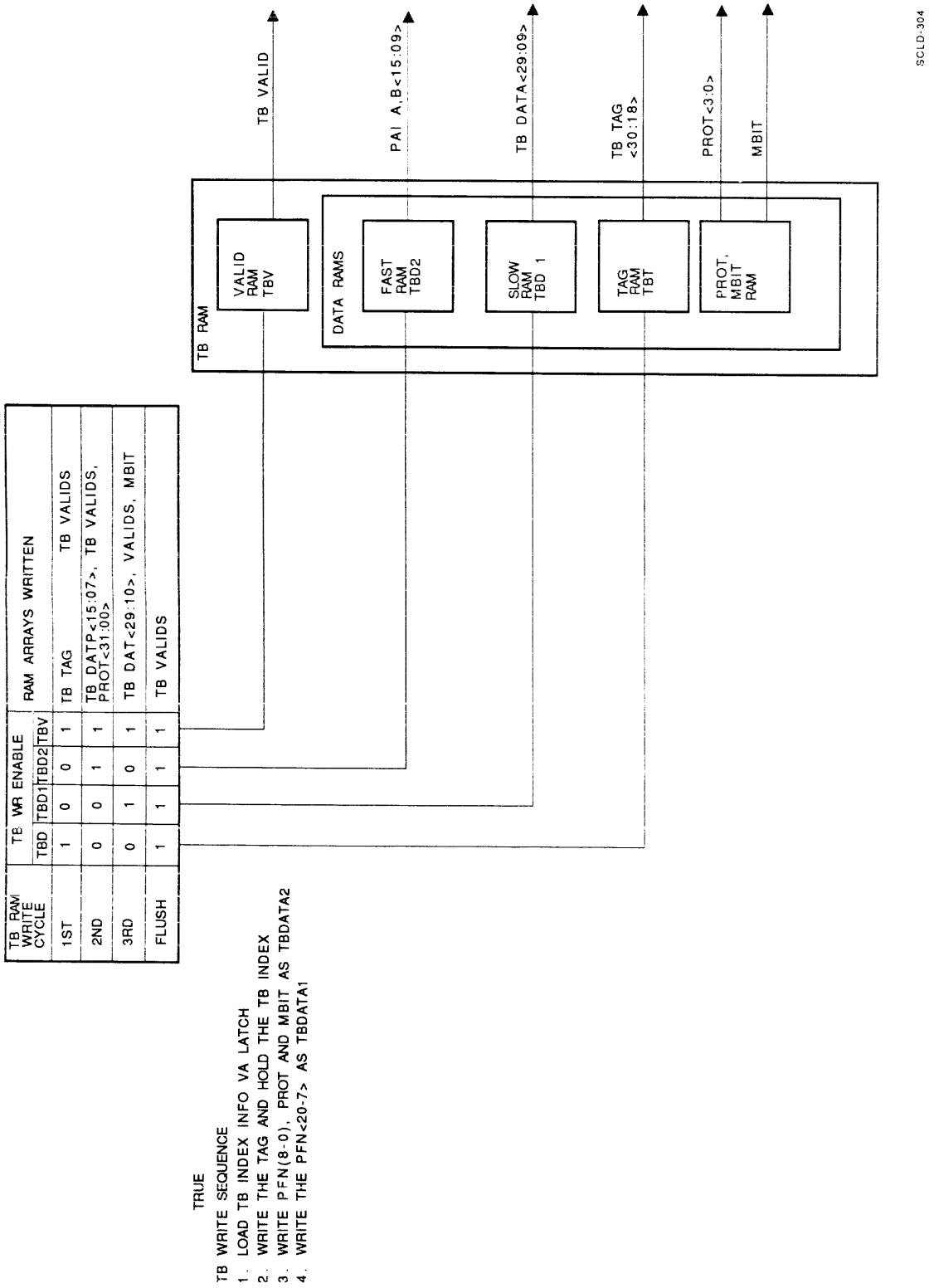


Figure 2-3 TB - Write Sequence Diagram

As Figure 2-3 illustrates, the TB RAM is:

- Flushed
- Written
- Checked

The TB RAM is flushed to invalidate (clear) one or more of its data entries. Typically, it is flushed after a CPU reset or a context switch has been made.

The TB RAM is written with page table entries after the TB match MCA has determined that it does not contain valid data for the address reference being made.

Table 2-1 correlates the encoding of the PROTection field <03:00> bits to the type of access allowed.

Table 2-1 PROTection Field <03:00> Coding and Access Allowed

Protect Codes <3	2	1	0>	Access Allowed K	E	S	U
0	0	0	0	NONE	NONE	NONE	NONE
0	0	0	1	RES	RES	RES	RES
0	0	1	0	RW	NONE	NONE	NONE
0	0	1	1	R	NONE	NONE	NONE
0	1	0	0	RW	RW	RW	RW
0	1	0	1	RW	RW	NONE	NONE
0	1	1	0	RN	R	NONE	NONE
0	1	1	1	R	R	NONE	NONE
1	0	0	0	RW	RW	RW	NONE
1	0	0	1	RW	RW	R	NONE
1	0	1	0	RW	R	R	NONE
1	0	1	1	R	R	R	NONE
1	1	0	0	RW	RW	RW	R
1	1	0	1	RW	RW	R	R
1	1	1	0	RW	R	R	R
1	1	1	1	R	R	R	R

2.1.1.3 TB Match MCA -- The TB match MCA (Figure 2-4) checks the validity of the TB RAM virtual address translations. As Figure 2-4 illustrates, the TB match MCA checks the TB RAM TB TAG <30:18> output bits against the LT VA <31:16> bits from the VA latch. If the two inputs do not match, or if the TB VALID bit output of the TB RAM has not been set, a TB MISS TRAP error signal is generated, indicating that the translation referenced by the EBox is not in the TB RAM. If the TB RAM contains a valid translation, a TRANSLATION OK signal is sent to the cache control.

The TB match MCA is controlled by means of a TBM CTRL <02:00> field received from the EBox and a MME signal from the cache control.

Table 2-2 provides a correlation of the TB match MCA operations to the type of TB match control bits TBM CTRL <02:00> and the setting of the MME bit.

Table 2-2 TB Match MCA Operation Coding

TB State	Encoding <1>	<TBS>
TB MISS	0	0
ACV	0	1
MBIT	1	0
TB OK	1	1
MME OFF	1	1

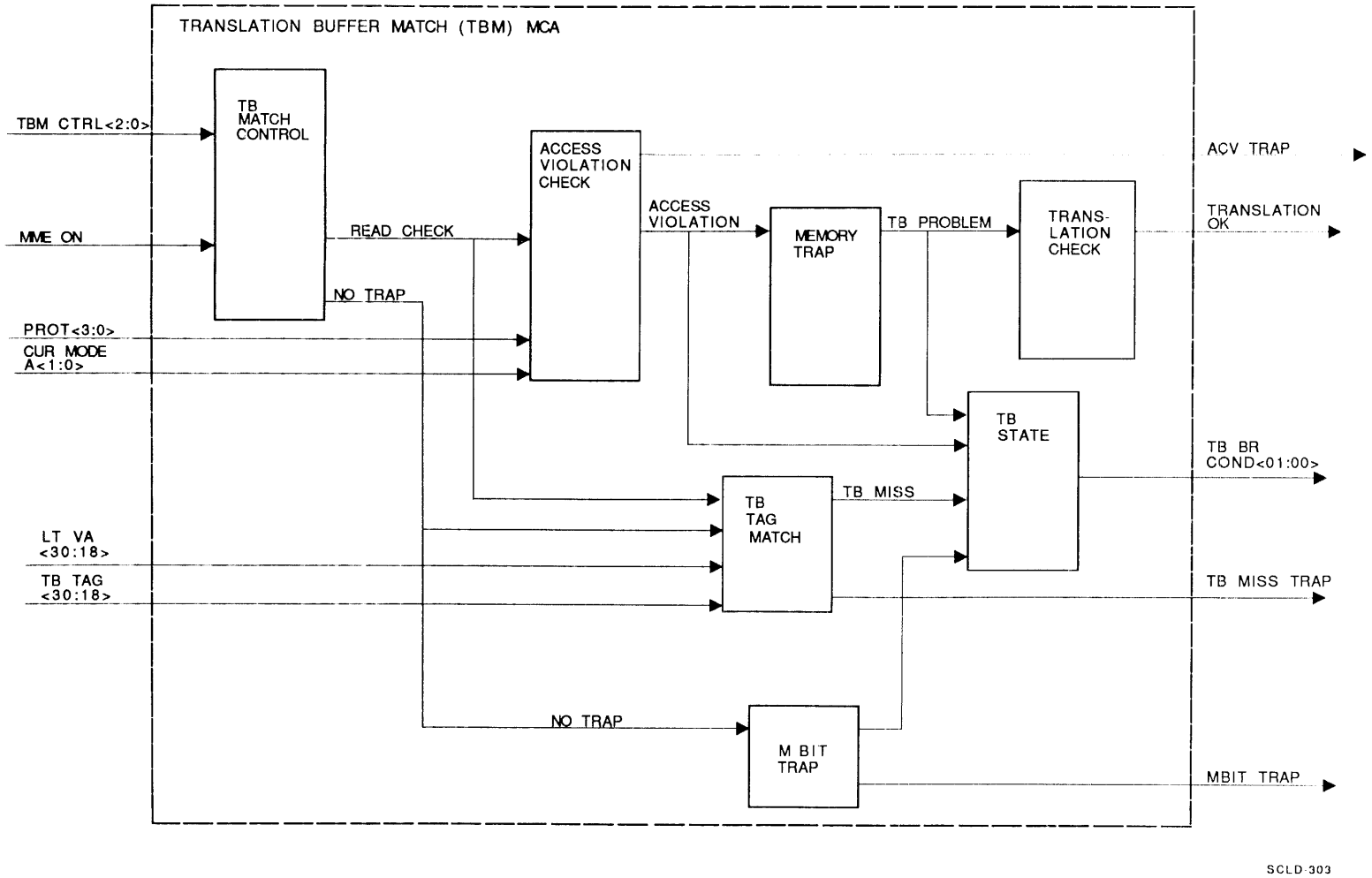


Figure 2-4 TB Match MCA - Simplified Block Diagram

2.1.1.4 TB RAM Bypass -- The TB RAM bypass logic produces a physical address input to the PA latch whenever the TB RAM has been turned off. It consists of a PABH MCA, a PABL MCA, and part of logic in the cache match MCA. The TB RAM bypass logic produces the following types of addresses:

- PIBA
- Delay write
- Refill/Invalidate

PIBA Addresses

A PIBA address contains a physical address that points to the next longword of I-stream data to be fetched. Because the PIBA is used to store a physical address, and fetches contiguous longwords from memory, an address translation need only be done once per page. The address translation is performed and the PIBA is loaded whenever:

- There is a Jump
- A conditional branch is successfully taken, or
- A PIBA page crossing has occurred

Delay-Write Addresses

A delay-write address is the address of a write waiting to be written into the cache. The address held in the TB RAM bypass corresponds to the data held in the delay-write address buffer.

DELAY WR ADDR <15:09> is loaded during a CPU write operation, with either TB DATA <15:09> bits, or the LT VA <15:09> input bits to the PABH MCA. This delay-write address is used to update the cache data store if the write was determined to be a hit in the cache, or to check subsequent CPU reads to determine if they hit in the same cache data store location that is waiting to be updated.

Since this MCA contains only the <15:09> bits of the delay-write address (PABL MCA contains low bits <08:02>), it generates the PART2 DELAY WR HIT signal that is used by the cache match MCA to generate the DELAY WRITE HIT signal.

NOTE

The PABH MCA DELAY WR ADDRS latch is loaded during an A clock, and is gated by the signal DELAY WR LOAD.

Refill/Invalidate Addresses

When a reference misses in the cache, the CBox sends a read request for the data to the memory or the I/O device. Data returned by the memory or the I/O device is refill, or returned data. For the duration of the returning data input, the CBox performs refill cycles, sending data to the IBox or the EBox and writing the data into the cache data store.

The refill address is always held in the NMI interface address/data slices PA file PIBA Q2 or Read Q (because it was first a miss). The address is selected from one of these queues, and then sent to and latched in the TB RAM bypass MCAs refill/invalidate latches for the duration of the refill. This address is selected as the PA within these MCAs for input to the data PA latch and the tag PA latch during the refill cycle.

Within the TB RAM bypass logic:

- The PABL MCA produces the refill address low bits <15:09>.
- The PABH MCA produces the refill address high bits <15:09>.
- The cache tag MCA checks the refill address to determine if a refill match exists.

The REFILL/INVAL address is stored in the three MCAs, and the PABL MCA increments the low three bits of the address to point to the proper longword of the hexword that is being written to the cache during the refill cycle. The REFILL/INVAL address is received by the PABH MCA on the B CLK, and is loaded into an A latch, which has a clock gated by the signal LD REFILL/INVAL. As with the PIBA, bits <15:09> of the REFILL/INVAL ADDR do not have to be incremented, so this section of the PABH MCA simply provides latching for these bits.

During an invalidate operation, the REFILL/INVAL address is loaded into the three MCAs with the address (which had been checked for a cache hit) that was present on the NMI, and then used to clear the valid bit in the appropriate cache locations.

Figures 2-5 and 2-6, respectively, are simplified block diagrams of the TBR RAM bypass PABH and PABL MCAs. They illustrate how the PIBA, delay-write, and refill/invalidate addresses are produced.

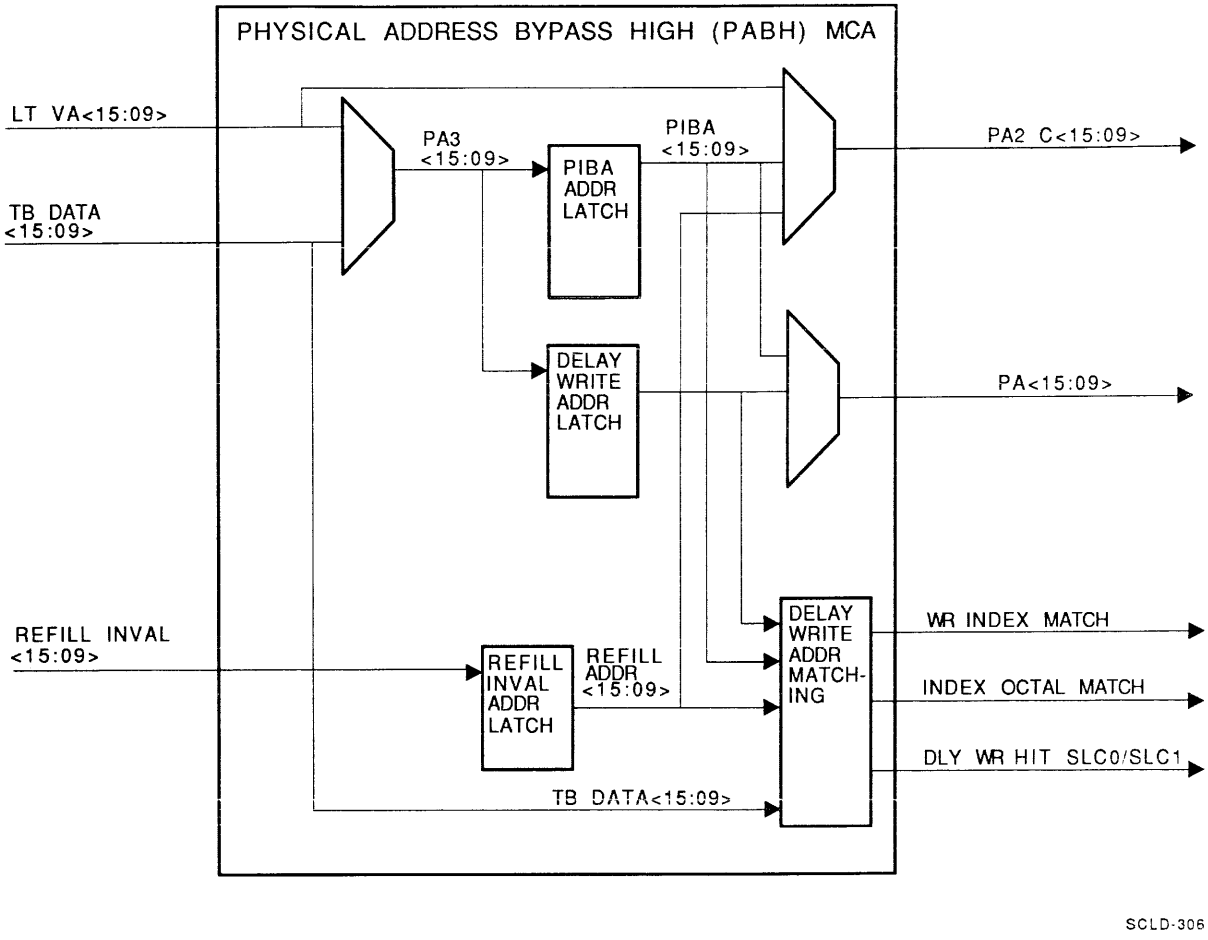


Figure 2-5 PABH MCA - Simplified Block Diagram

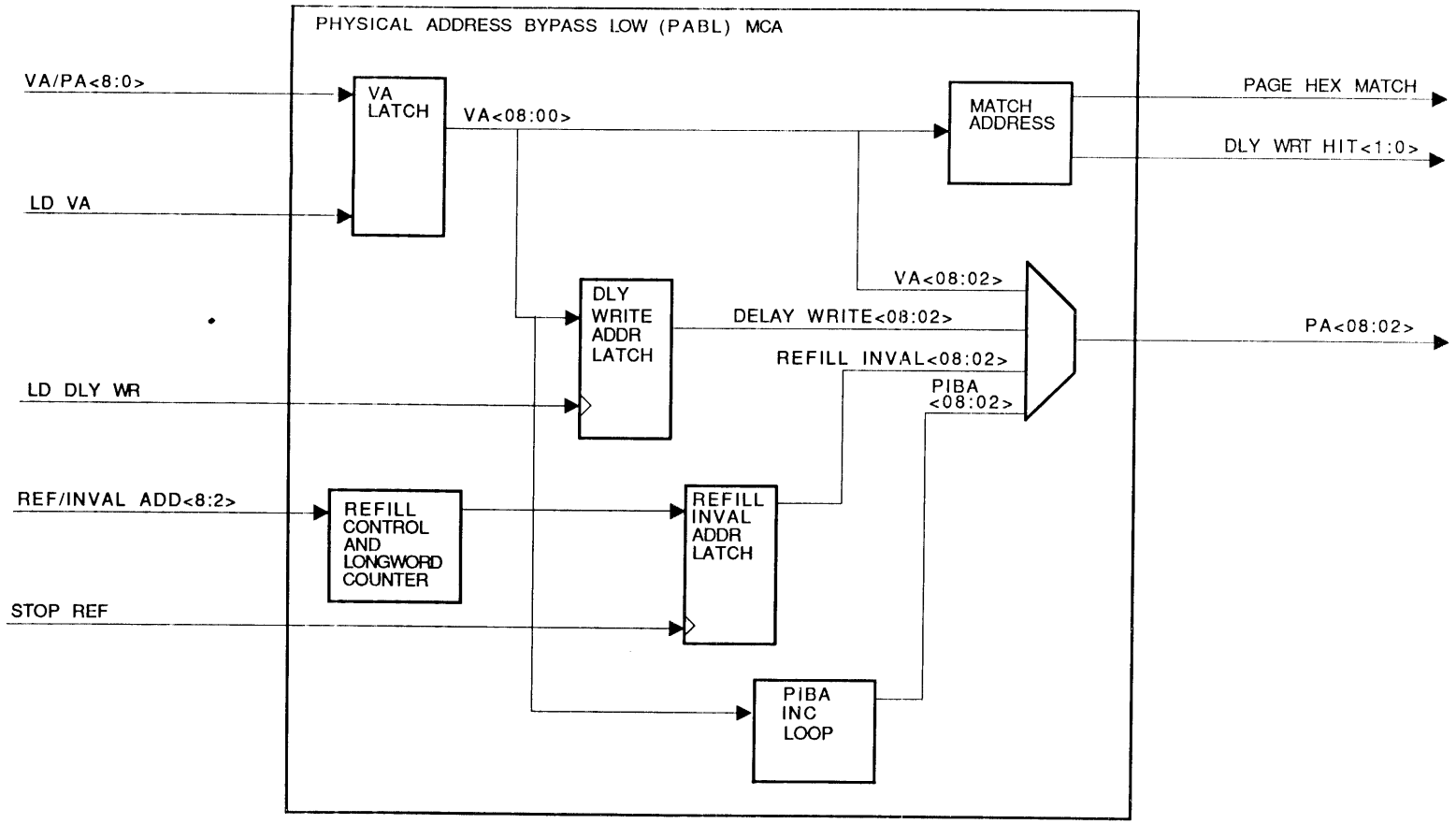


Figure 2-6 PABL MCA - Simplified Block Diagram

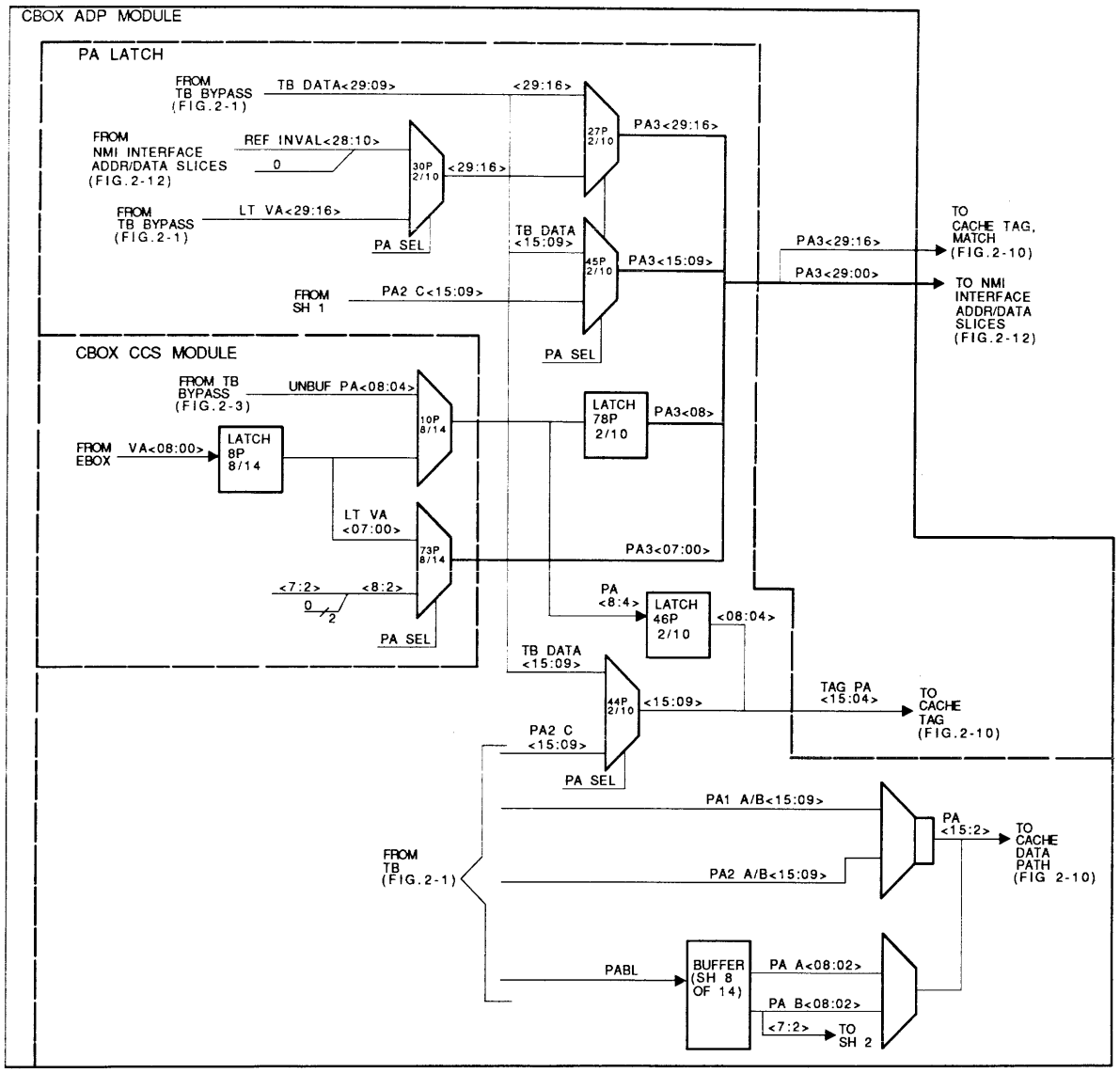
PABL MCA functions include:

- Storage and control of the PIBA bits <08:00>
- Storage of DELAY WRITE ADDRESS bits <08:00>, and the generation of the DLY WRT HIT signal for a match in this section of the address
- Storage and control of REFILL/INVAL ADDR bits <08:00>
- Detection of four microtrap conditions:
- Unaligned
- Page cross
- Unaligned page cross

2.1.1.5 PA Latch -- The PA latch produces the following address vectors:

- PA <15:02> for the cache data store
- PA 3 <29:16> for the cache match MCA and cache tag MCA
- PA 3 <29:00> that is sent to the NMI interface address/data slices
- TAG PA <15:04> that is sent to the cache TAG MCA

Figure 2-7 is a simplified diagram of the PA latch.



SCLD-308

Figure 2-7 PA Latch - Logic Diagram

The data PA latch produces the cache data store look-up address vector PA <15:02>. It has the same timing as the tag PA latch, but is, in the case of writes, loaded from a different source (the delay-write address buffer in both the PABH MCA and the PABL MCA).

Because the data PA latch is used only to index the cache data store, it does not need any bits beyond the cache index bits (that is, PA <29:16> bits are not needed).

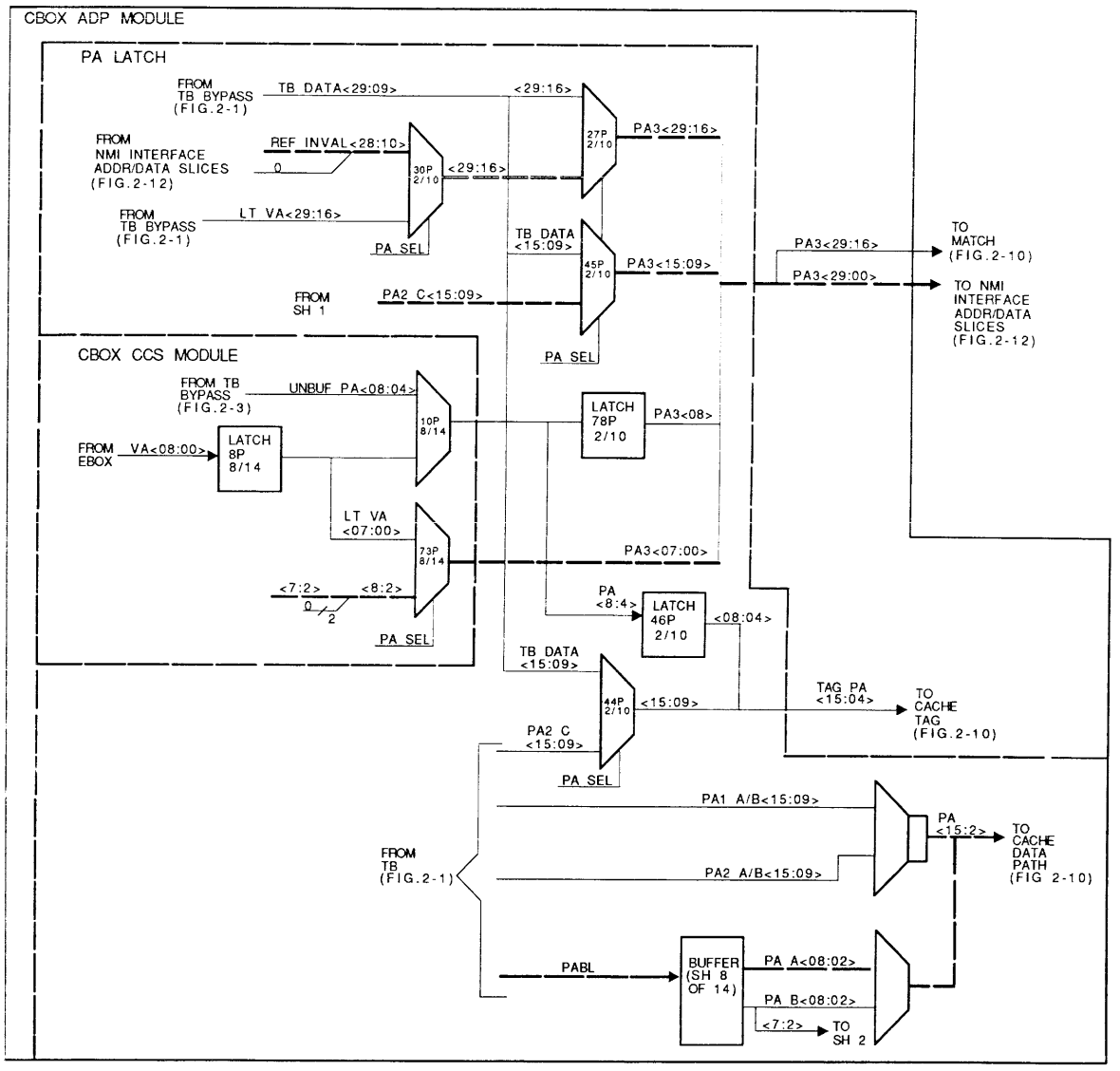
The data PA latch is loaded from:

- The TB RAM with bits PA1 <15:09> (or from the TB RAM bypass PABH MCA with bits PA2 <15:09>) during a cache read operation.
- The delay-write address buffer in PABL for CPU writes with PA 1 <08:02>.

The tag PA latch produces the cache tag MCA look-up address vector TAG PA <15:04>. The tag PA latch holds the same address as the data PA latch during all CBox cycles (except the write cycle).

The tag PA latch is loaded from the TB RAM with TB DATA <15:09> bits (or TB RAM bypass PABH MCA with bits PA2 C <15:09>) for CPU reads and "new PC" reads.

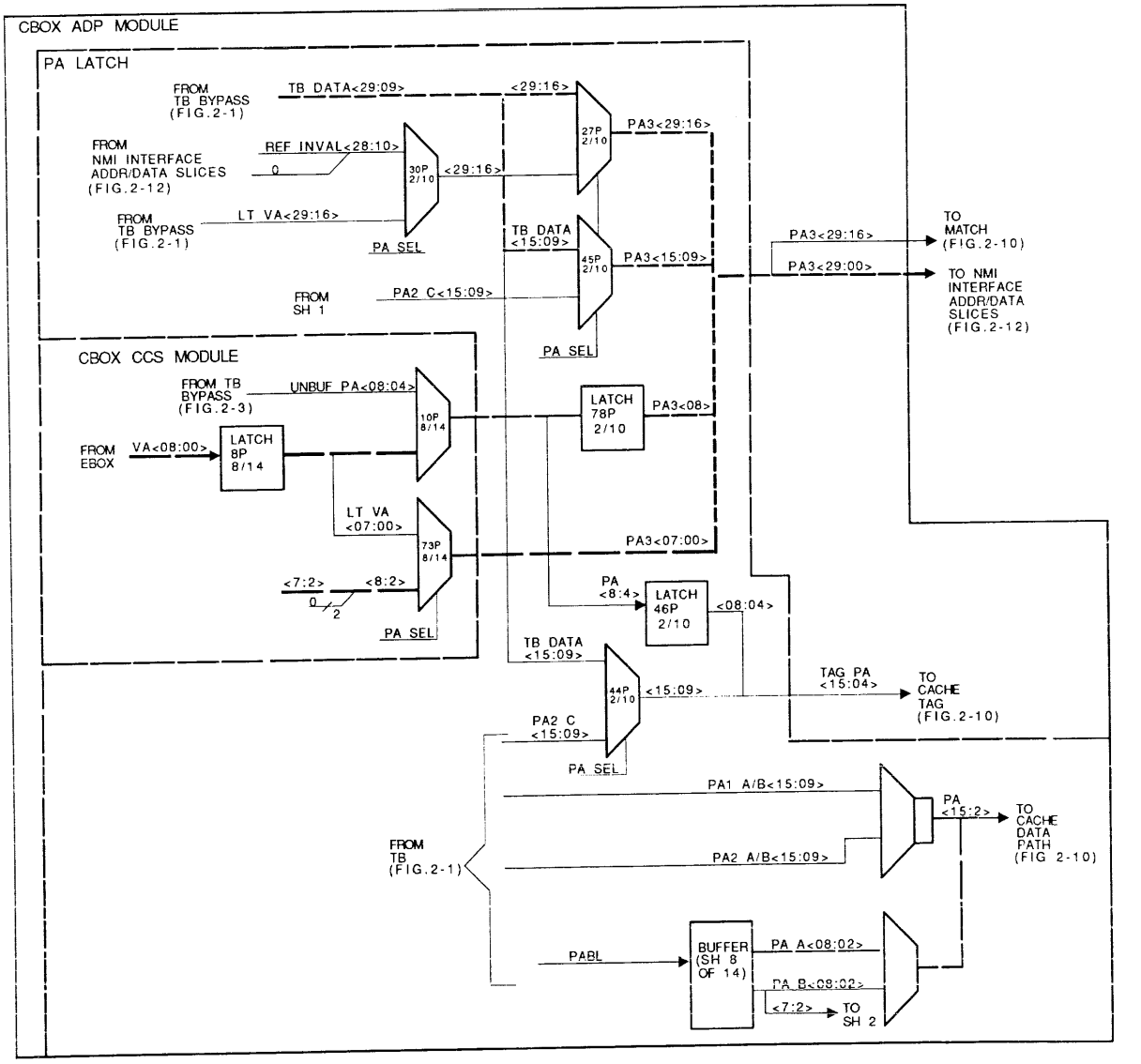
Figure 2-8 illustrates the bit routing through the TB PA latch when a refill address is received from the NMI address/data slices (Figure 2-12).



SCLD-308B

Figure 2-8 PA Latch Bit Routing - Refill Cycle

Figure 2-9 illustrates the bit routing of the TB RAM the TB DATA output through the PA latch when the EBox makes a virtual address reference to the CBox.



SCLD-308C

Figure 2-9 PA Latch Bit Routing - VA Reference

2.2 CBOX SUBSYSTEMS DESCRIPTION

2.2.1 Cache

As Figure 2-10 illustrates, the cache consists of:

- Data path logic
- Tag MCA
- Match MCA
- Control MCA
- MD number MCA

2.2.1.1 **Cache Data Path Logic** -- The cache data path logic is used to move data out of the CBox to the IBox and the EBox. It consists of:

- A delay-write data latch
- A write data output multiplexer
- A cache data store
- An ALU bypass multiplexer
- A write buffer

Basically,

- Data is read from the cache data store as bits RD <08:00> and multiplexed by means of the ALU bypass multiplexer as cache DATA bus <31:00> bits to the EBox register file and the IBox, or

- It is multiplexed to the EBox ALU as A CD <31:00>

Alternately, the cache data store can be bypassed (by means of a bypass multiplexer) and EBox write data is sent as WR DATA CDB <08:00> bits to these two destinations.

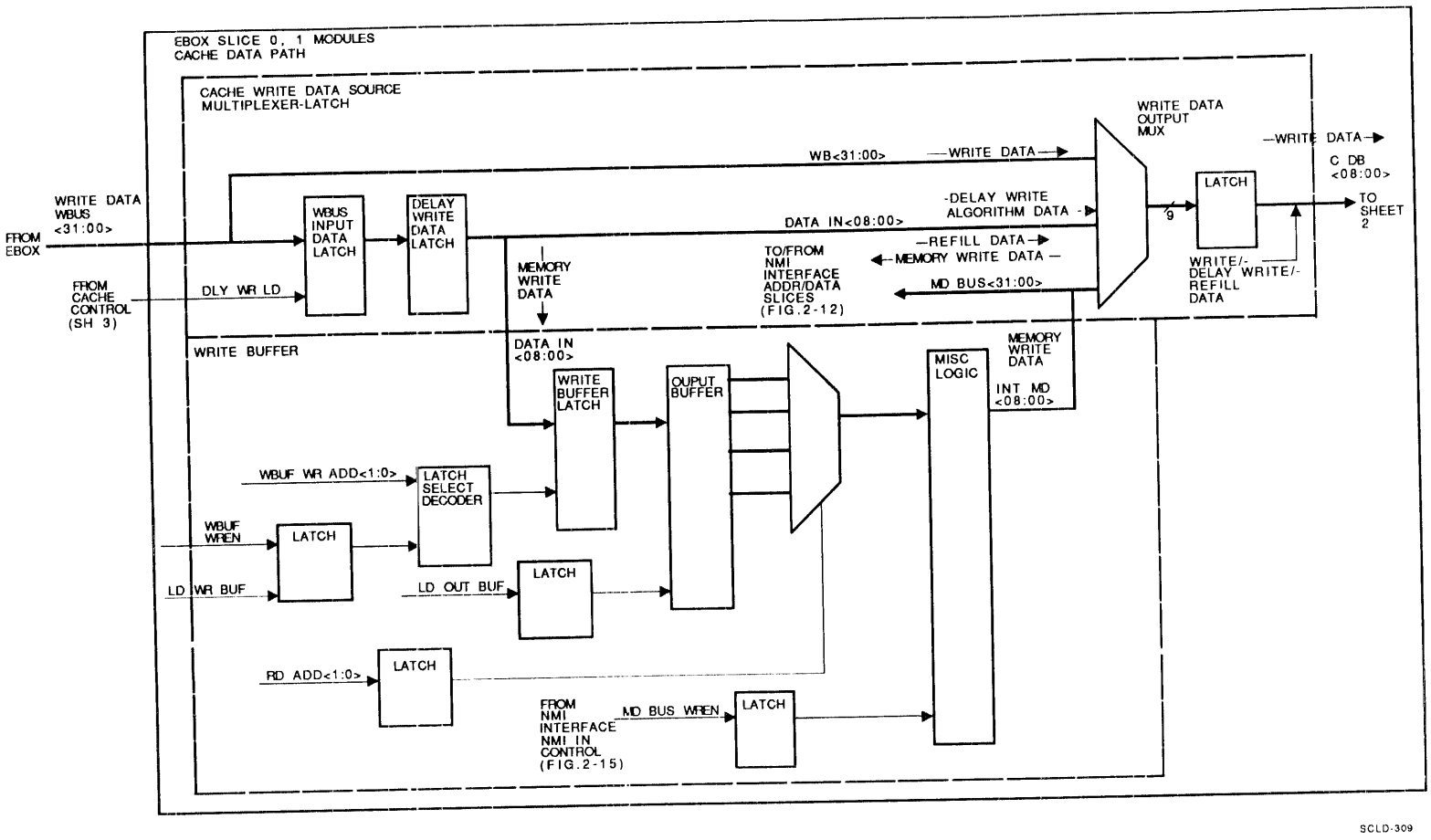


Figure 2-10 Cache - Block Diagram (Sheet 1 of 3)

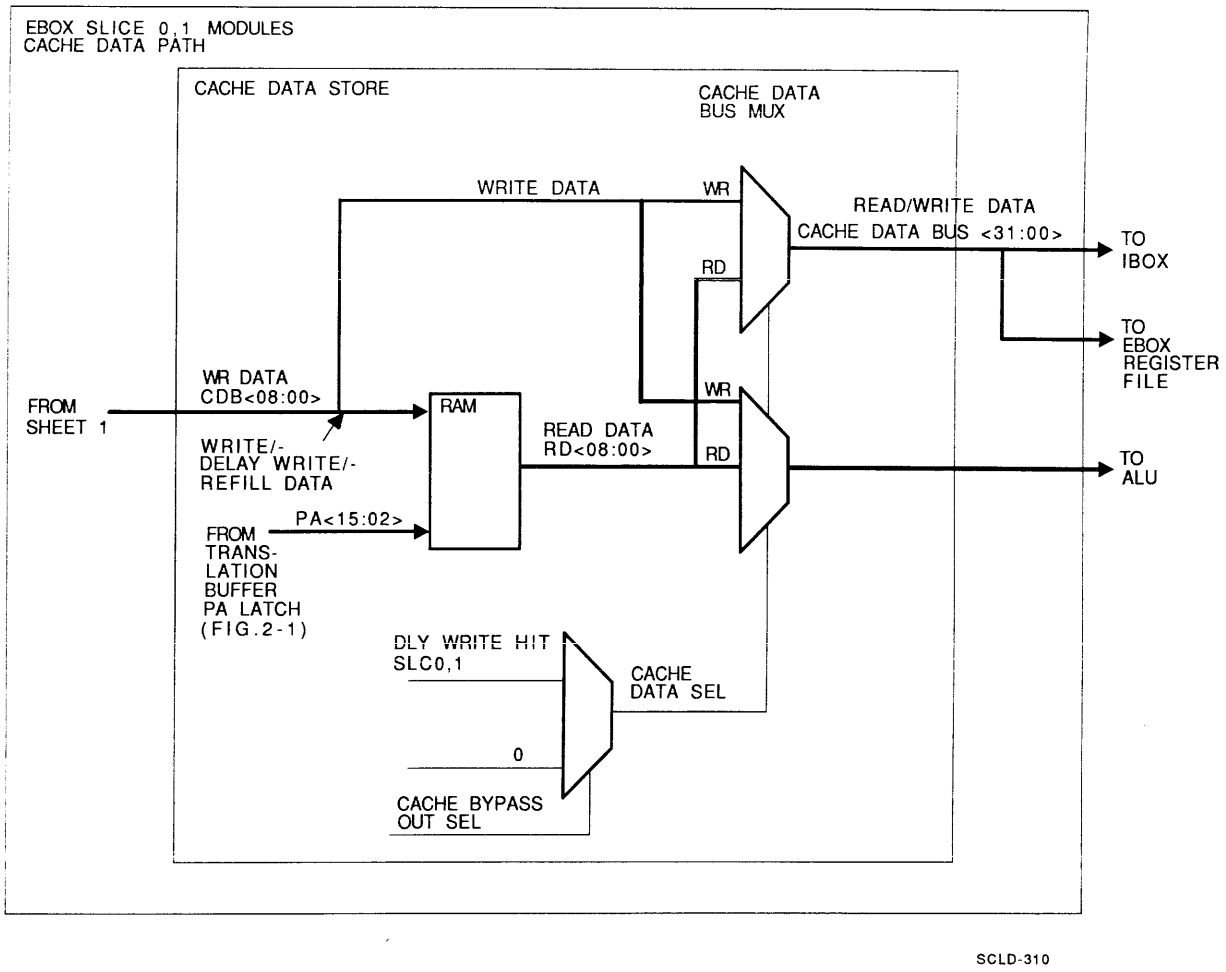


Figure 2-10 Cache - Block Diagram (Sheet 2 of 3)

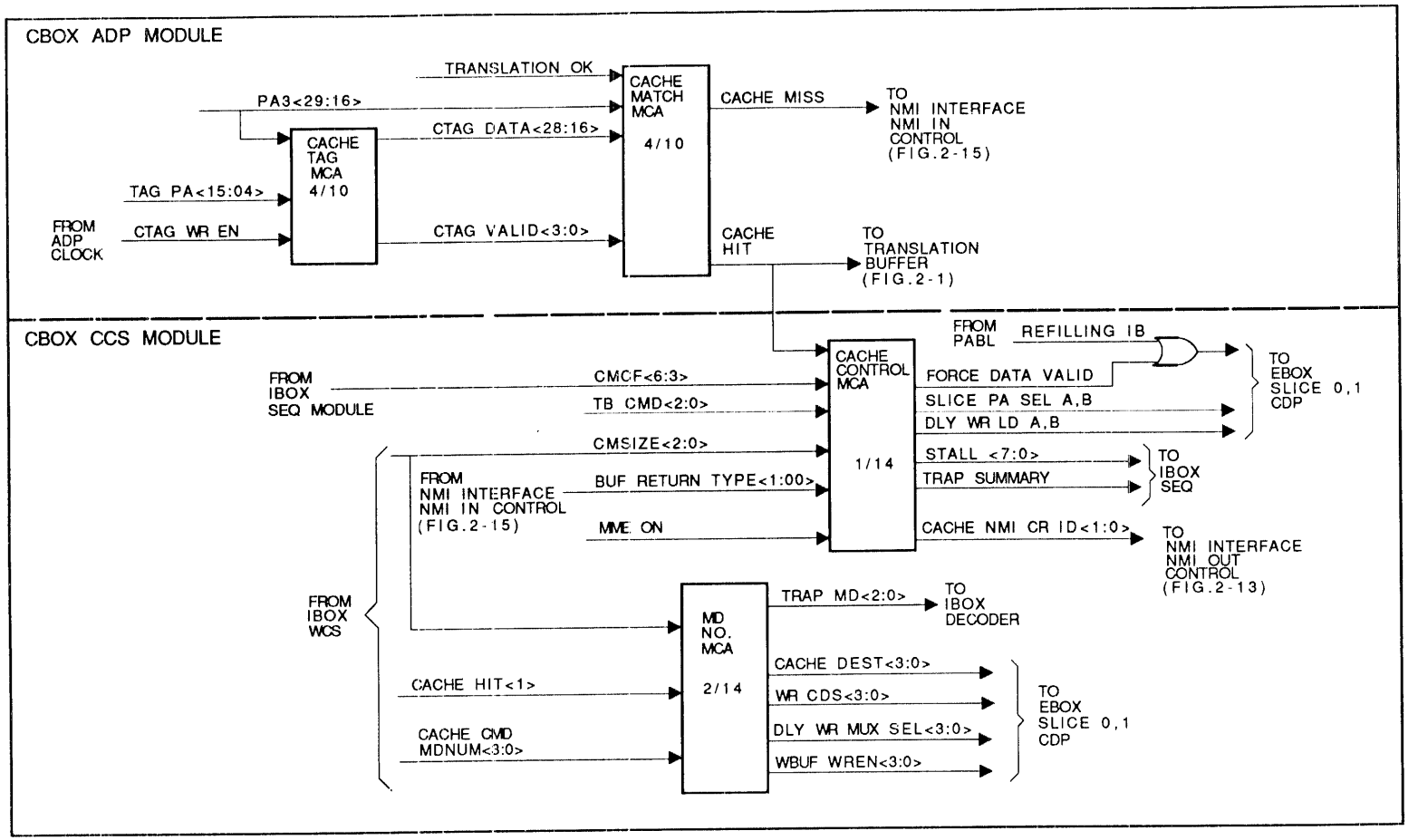


Figure 2-10 Cache - Block Diagram (Sheet 3 of 3)

The cache data store is loaded with WR DATA CDB <08:00> from a latch at the output of the cache write data source multiplexer. The data source multiplexer is used to select:

- Data directly from the EBox as WB <31:00> during a cache write.

- A delayed EBox input from the delay-write data buffer as DATA IN <08:00> during a delay-write-algorithm operation.

- INTernal MD <08:00> from the NMI interface (by means of the MD bus) when memory data must be written to the cache RAM.

  During a write operation to memory, the delay-write data buffer DATA IN <08:00> output is buffered in a write buffer, put on the MD bus as MD bus <31:00>, and then written to memory by means of the NMI address/data slices and the NMI. Incoming WBus data <31:00> bits are either:

  1. Sent directly to the write data output multiplexer as WB <31:00>, or

  2. Latched in the delay-write data latch (if a delay write is requested), the DATA IN <08:00> output of which can be sent to either:

     a. The write data output multiplexer, or

     b. The write buffer

The delay-write data buffer provides storage of data received from the WBus until:

- The EBox executes another write (at which time the data is written into the cache data store on the next write)

- A read operation requires data to be placed into the cache data store

The delay-write data buffer is loaded during a delay-write algorithm by means of a DLY WR LD control signal from the cache control MCA.

The cache data store holds the corresponding data for addresses held in the cache tag MCA. It is written from:

- The WBus

- The delay-write data buffer, or

- The MD bus

The cache data bypass and cache ALU MD bypass multiplexers located at the output of the cache data store are used to select between the output of the cache data store and the cache write data source latch. This allows the data being written to the cache data store to also be sent to the EBox and IBox if necessary, or in the case of a delay-write hit, to be taken directly out of the delay-write data buffer.

The ALU MD bypass allows data to be sent more quickly to the ALU from the cache data store, instead of through the MD registers. It contains the same information as the cache data bus, but the data is unlatched and, therefore, driven slightly earlier (the input latch of the ALU has the same timing as the latch that drives the cache data bus).

The write buffer is a 1-octaword (16-byte) write-only, write-back cache that is used for writing data to the cache data store or VAX 8800 memory. It allows the following types of large data blocks:

- Quadword
- Octaword
- Grand-floating
- Huge-floating
- Character strings
- Stack writes

to be grouped into blocks that correspond to the memory block size. This minimizes the number of required main memory write cycles.

The write buffer consists of:

- Input latch selection logic
- 16-byte write buffer
- Output buffer
- Output multiplexer

Basically, the write buffer is written with a longword. The latch sections are then selected for writing to the output buffer by WBUF WR ADD <01:00> bits from the translation buffer after being enabled by WBUF WREN control signal [from the cache control logic (MD number MCA)].

Longwords in the output buffer are read (multiplexed) onto the MD bus as MD BUS <31:00> bits by means of the MD BUS WR EN and WR BUF RD ADD <01:00> control signals.

NOTE

During the time that the output buffer is being multiplexed onto the MD bus, the write buffer can again be accepting a new longword of data, thus providing approximately a 200-ns delay between availability of data.

2.2.1.2 **Cache Tag MCA** -- The cache tag MCA is used to check if data requested in the current reference is in the cache. It consists of a cache TAG RAM, plus several registers. The RAM is written with PA 3 <29:16> bits and addressed by means of PA <15:04> bits (from the translation buffer). The CTAG VALID <03:00> output indicates which octaword within a cache block is valid.

2.2.1.3 **Cache Match MCA** -- The catch match MCA:

- Detects cache "hits"
- Provides storage for part of the PIBA address and part of the delay cache write buffer address

The cache match MCA consists of:

- Cache match
- Refill match
- Write buffer match logic

Basically, CTAG DATA <28:16> and PA 3 <29:16> inputs are compared for a match. If they do match, they are gated with CTAG VALID <3:0> as a TAG match condition to produce a CACHE HIT signal. If a TAG match condition does not exist, a CACHE MISS signal is sent to the NMI interface (Figure 2-11).

NOTE

The FORCE CACHE MISS input signal from the cache control MCA can also produce the CACHE MISS output.

Additionally, there are partial "delay-write hit" terms from the PABH and PABL MCAs. The major outputs of the cache match MCA are several primary "hit" signals that are combined with other CBox states in order to cause a stall, to start an NMI sequence to fetch read data, or to mark data "valid" for the IBox and EBox.

The cache match MCA, in addition to checking the cache tag match, also checks for "hits" on the delay cache write buffer and on the cache write buffer. Hitting on the delay cache write buffer on a read means that the longword being requested is within a valid cache block, but one or more bytes of the longword in the cache data RAMs is "stale" and must be updated with information from the delay cache write buffer before the data can be returned to the requester. Hitting on the write buffer means that the longword being requested is NOT within a valid cache block, but some information from that cache block, which will be fetched, is in the write buffer.

Both the write buffer and the delay cache write buffer, if valid, contain information from the same address block. Thus, the distinction between delay cache write buffer hit, and write buffer hit is the cache hit signal. When a refill hits the delay-write address, the delay-write hit valid bit must be cleared (since the data in the cache is no longer stale).

When PA3 <29:16> is received by the cache match MCA, it is latched for forwarding to the PIBA latch and the delay cache write latch, and it is also used immediately by the hit Logic and the PA3 parity logic. If the PA3 represented a new PC (or a new translation for the PC), the PIBA latch loaded. Once the PIBA latch is loaded, all PIBA cache cycles match with the PIBA latch instead of the received PA3 (selected by the match data multiplexer). Similarly, whenever the received PA3 corresponds to a write, the delay cache write latch loaded.

The cache match MCA hit logic compares the match field (that is, bits <29:16> of the physical address) of the current reference to output of the cache tag RAMs. This comparison is always a miss if MATCH DATA<29>=[1], since I/O address space is referenced this way.

The hit logic contains two match arrays, one to compare the match data with the cache tag DATA and one to compare the match data with the delay-write address. The delay-write address compare happens relatively early while the cache tag data compare ultimately determines the time of the output hit signals. In order to speed up the cache match signal for the parity check of the cache data, the cache tag parity bits received from the RAMs are compared with the match data parity (available and checked early). Since there is no TB lookup during PIBA cycles or invalidates, TB OK is forced true, and PA3 parity is forced OK.

2.2.1.4 **Cache Control MCA** -- The cache control MCA provides a general control of CBox operation. It decodes cache command field bits CMCF <6:3> and CMSIZE <2:0> into control signals for the cache data path and the translation buffer PA latch. It also sends stall and trap status data to the IBox.

The cache control MCA performs cache arbitration. Arbitration in the cache is effected on a priority basis. The NMI activities are of the highest priority level and the CPU has the MPXT level. The following is the priority for each cycle type in descending order:

- Refill/invalidate cycle. Requested by the NMI in control (function MCA) to either return read miss data or to invalidate a cache block. Refill and invalidate cannot happen at the same time.

- Register return. Requested by the NMI out control (microsequencer MCA) to complete a register read sequence.

- CPU request. Microcode request to read, write, etc.

- Noop. The default request from the microcode is no operation. When this is true, two further cycles could occur. The PIBA cycle, uses the cache only when the microcode is no operation and there is no NMI activity. If the PIBA cycle missed in the cache, then the quiescent state is set, which prevents further PIBA cycles from occurring until refill or I-stream change occurs. Neither the PIBA or quiescent participates in arbitrating for the CBox. Additionally, the quiescent state is a mechanism for controlling the PIBA, and does not affect operation of the other cycles (that is, quiescent can be set when the microcode makes requests).

Which function is granted use of the cache data path defines what the CBox is performing (for example, if refill occurs then the CBox is said to be performing a refill cycle).

NOTE

If the NMI interface is busy processing writes out to the NMI and the PIBA is quiescent, the CBox is then in a quiescent state.

2.2.1.5 **MD Number MCA** -- The MD number MCA receives the number (as CACHE CMD MDNUM bits <3:00>) of the memory data (MD) register that cache data is to be written to, and sends it as CACHE DESTination <03:00> to the EBox. When the translation buffer TB match MCA generates a MEM TRAP signal, the number of the register (that should have received the cache data) content is sent to the NMI out control (microsequencer MCA) as TRAP MD <02:00>, and is used when the required read miss data is received from memory.

2.3 CBOX SUBSYSTEMS DESCRIPTION

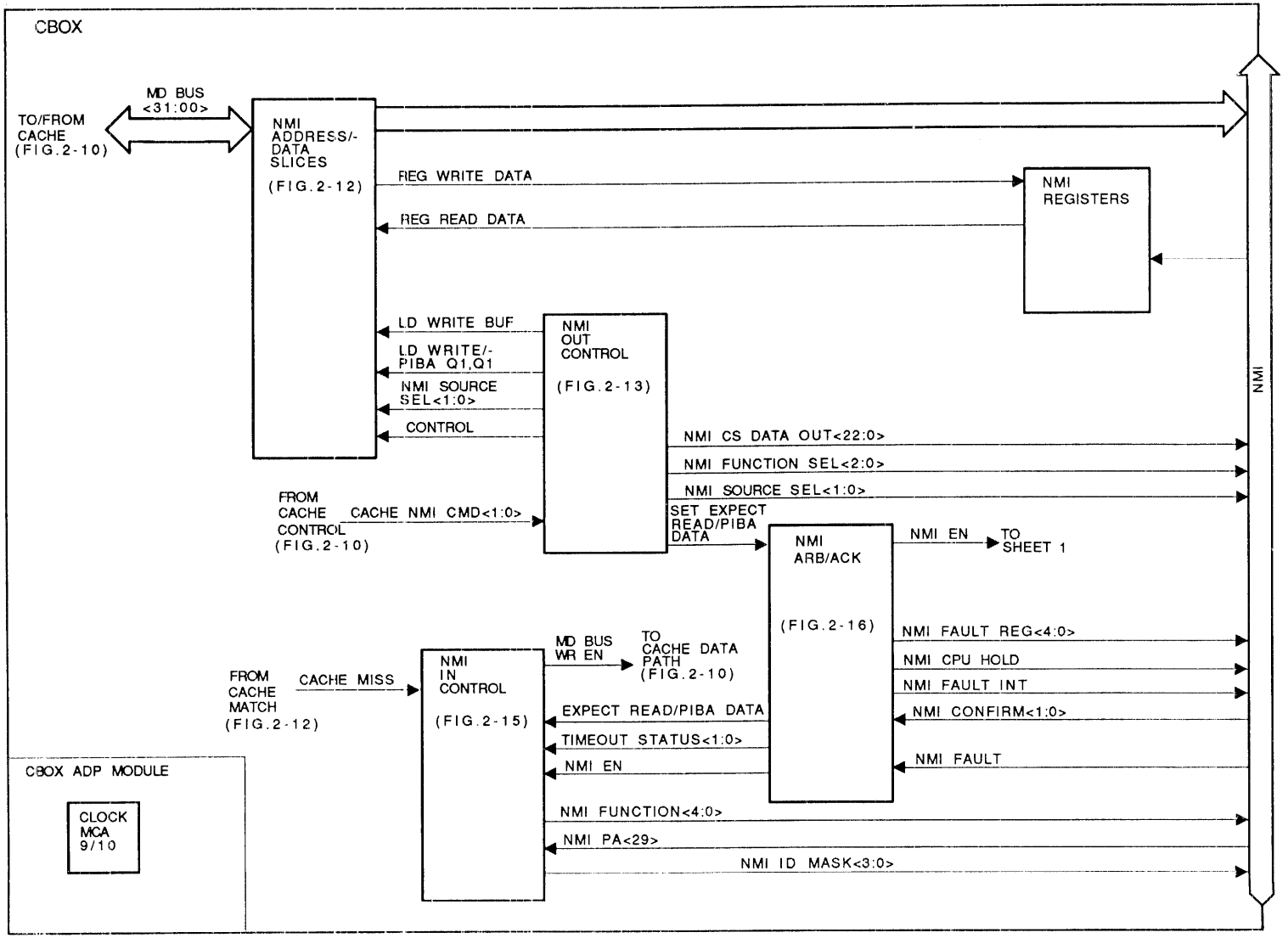
2.3.1 NMI Interface

The NMI interface provides a data path and control function with which the CPU communicates on the NMI. When a cache read misses, the NMI interface uses the read-missed address received from the translation buffer (TB) to build an NMI command/address transaction and sends it to memory. The TB and the cache then become available to process additional CPU requests, while the NMI interface processes the memory transaction. When the read-miss data arrives from memory, the NMI interface takes control of the cache and loads the data into the cache data store. It also validates the cache tag valids with the new tag address.

When the CPU executes a write, the NMI interface transfers the write data to memory without making the CPU wait for write completion to memory.

As Figure 2-11 illustrates, the NMI interface consists of:

- Address/data slices
- NMI out control
- NMI in control
- NMI arbitration/acknowledgment logic
- NMI hardware registers



SCLD-312

Figure 2-11 MNI Interface - Block Diagram

2.3.1.1 **NMI Address/Data Slices** -- There are eight NMI address/data slice MCAs; six are located on the address data path (ADP) module, and the other two are located on the cache control sequence (CCS) module. Their function is to send:

- Read-miss/write addresses and write data to memory
- Read-miss data (that is received from memory) to the cache
- Read-miss addresses to the translation buffer

Figure 2-12 is a simplified diagram of the NMI address/data slice MCAs. As Figure 2-12 illustrates, the MCAs basically consist of:

- A physical address (PA) file
- An NMI address/data bus
- An MD-bus

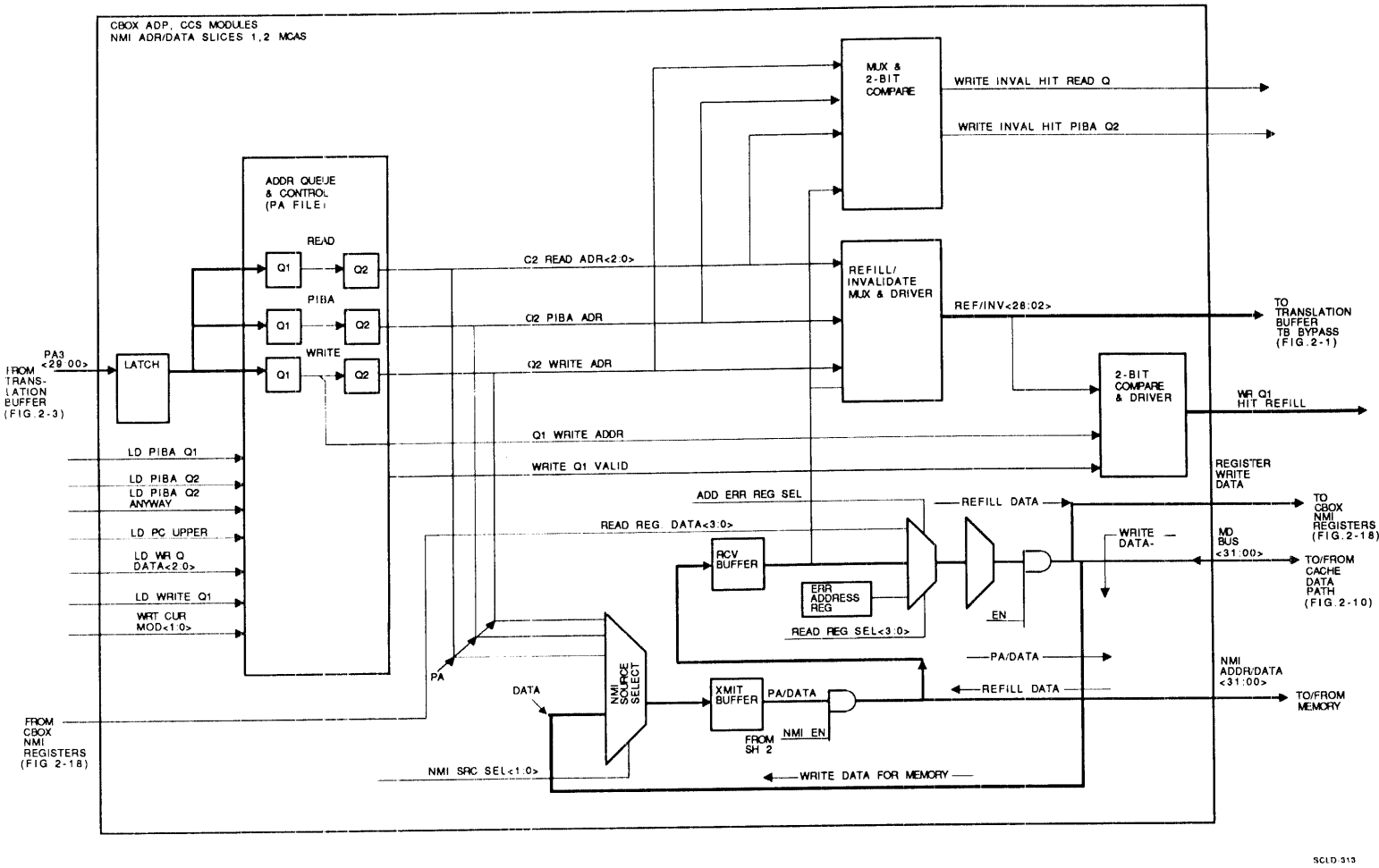


Figure 2-12 NMI Address/Data Slices MCA - Simplified Block Diagram

PA File

The PA fill consists of three, two-deep queues:

1. Read Q1, Q2
2. PIBA Q1, Q2
3. Write Q1, Q2

Briefly,

- Memory reads load the read (miss) queue. The read queue holds read miss addresses destined for the EBox. When the read-miss data is received from memory, the read-miss address in the read queue is sent to the TB. The TB then produces an address vector for the cache and the data is transferred from the NMI address/data slices MD BUS to the cache data store.

- PIBA references load the PIBA miss queue. PIBA queues Q1 and Q2 hold read miss addresses destined for the IBox.

- A memory write operation loads the write queue. Write queues Q1 and Q2 are used during write requests; the address is sent with the data (taken from the cache write buffer) as part of a write transaction.

The CPU stalls if it attempts to do a reference to a full queue. The file is read at conceptually two different times. First, to start the memory transaction, an address is taken out of the appropriate register in the PA file and placed on the NMI address wires in the same cycle as the command.

NOTE

For a write, this is the last time the address needs to be used unless a retry is necessary. For reads, when the NMI sequencer recognizes that a read return sequence is taking place, it loads the refill/invalidate address port with the refill address to load the refill sequencer.

When a cache read miss occurs, and if the PA file read queue Q2 is empty, the miss address is transferred from read queue Q1 to Q2. The NMI out control (microsequencer MCA) then uses the address in read queue Q2 for the read transaction. If a second miss occurs, the address of the miss remains in input queue Q1 until the data from the first read has been received from memory.

The two-deep read queue enables the CPU to continue processing while a miss is outstanding. If a second miss occurs while the first miss is still out, a bit in the stall logic is set that indicates "stall if read." When there are two read misses outstanding, the CPU VA stalls on read. When the data comes back from memory, the miss address in read Q2 is placed in the refill/invalidate multiplexer and driver logic and sent to the translation buffer as REFILL/INVAL <28:02>.

The PIBA queue functions similarly to the read queue, except that the high-order bits are loaded when there is a:

- Branch
- PC change, or
- Page cross PC

When the PIBA queue is initially loaded, the upper bits go to output queue Q2. If there is another PC change and no miss has occurred, the high-order bits are again transferred to Q2. The lower bits, PIBA<8:2> are loaded in PIBA Q2 in the cycle that the miss occurred in.

If there is a branch while PIBA Q2 is full, and waiting for read data, the new PIBA high-order bits (PIBA <29:09>) are loaded in PIBA Q1.

NOTE

This can happen when the PIBA prefetches ahead of instruction execution.

When the data for the address in PIBA Q2 is returned, the content of PIBA Q1 is moved to PIBA Q2.

The cache write buffer (Figure 2-10) is 2-octawords deep, and can support two independent transactions. Therefore, the related PA file write queue in the NMI address/data slice MCAs (Figure 2-12) is two addresses (queues) deep. The first write of a sequence to load the cache write buffer will load the input side (Q1) of the queue. When either:

- The explicit microcode bit to validate memory is set, or
- The write sequence crosses an octaword boundary:
  - The cache write buffer (Figure 2-10) is copied to the cache output write buffer.
  - An address in the write queue Q1 is copied to the output write queue Q2.
  - An NMI write sequence is requested from the NMI out control (Figure 2-13).

When the NMI out control sets up to do the command address cycle of the NMI write sequence, it examines the write mask buffer for the output write buffer and modifies the lower bits of the write address to fall on the natural data type boundary, (that is, the write buffer can be loaded with byte writes, but be an octaword write to the memory).

The address in output write queue Q2 is multiplexed by means of the NMI source select multiplexer (by means of the NMI out control NMI SRC SEL <01:00> signal) and is then sent to the NMI as NMI ADD/DATA <31:00>. The address is held in write Q2 until an acknowledgment from the last write to memory is received. If the acknowledgment is received, the register valid bit is cleared so the next write can be processed. However, if no acknowledgment is received, the address in the PA file is used for a memory-write retry until it completes successfully, or a timeout occurs.

NMI Address/Data Bus

The NMI address/data bus is sourced in the NMI address/data slice's MCAs. It is a multiplexed address and data bus and is the major interconnect between the CPU and the memory and I/O devices. It is used by the NMI in control (function MCA) and as an input port to the cache, and is used by the NMI out control (microsequencer MCA) output as a cache output port.

If, in a given cycle, the NMI out control, has not requested and obtained use of the NMI address/data bus, or, if the NMI in control is not processing a refill sequence, receive logic in the NMI address/data slices processes the received memory information as both address and data. The refill/invalidate wires in the NMI address/data slices are driven to load the refill register in the translation buffer. The MD bus is then used to send the received data to the cache data path where it is loaded into a cache write latch. This allows the NMI in control (function MCA) sufficient time to determine what type of NMI transaction is currently being processed. If it is a refill or an invalidate transaction, the data will be at the correct place (cache write latch) when the NMI in control needs it.

During a CPU-initiated transaction, the NMI address/data bus is driven with addresses from the PA file and data received from the MD bus (from the cache).

MD Bus

The MD bus is a bidirectional data path that provides an interface between the CPU and the NMI. The NMI out control uses it to move write data from the cache data path to the NMI address/data bus and refill data from memory to the cache. The NMI out control (microsequencer) uses the MD bus for register reads and writes.

MD bus arbitration is performed in the arbitration/acknowledgment control (Figure 2-16). When the NMI in control is not requesting the MD bus, the MD bus is enabled for the NMI in control refill path. While the NMI in control is determining if the received function is read return data, the data is loaded into the MD bus receive latch.

2.3.1.2 **NMI Out Control** -- The NMI out control:

- Controls the NMI address/data slice MCAs PA file queues
- Controls the cache:
  - Write buffer
  - Output buffer
- Sends NMI transactions to memory and I/O devices

Figure 2-13 is a simplified block diagram of NMI out control. It illustrates that it consists of:

- A next address chip
- A microsequencer MCA
- An NMI control store

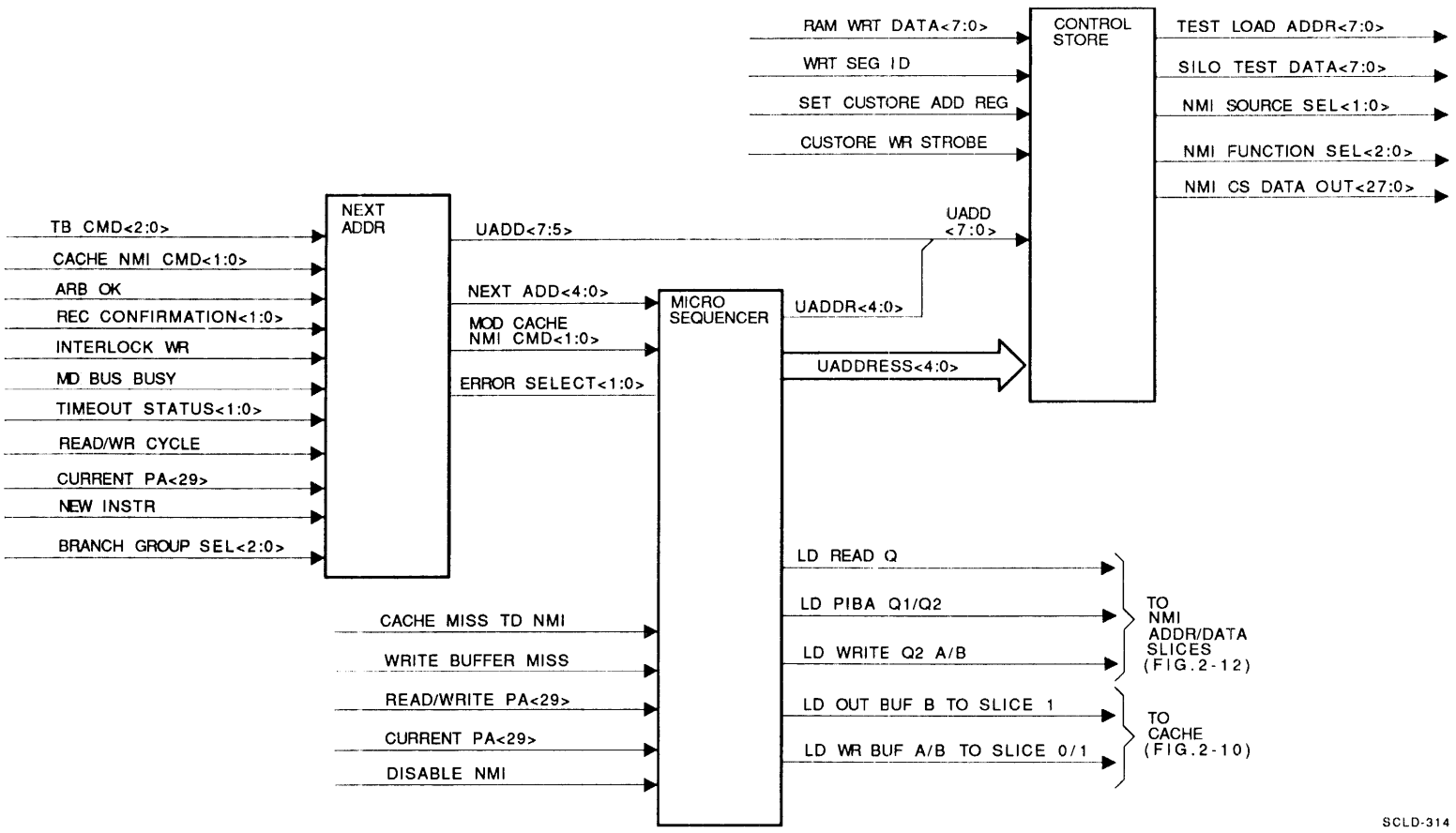


Figure 2-13 NMI Out Control - Simplified Block Diagram

The NMI microsequencer is a microcoded sequencer that handles command traffic to the NMI. It contains logic that holds the state of the NMI address/data slice MCAs PA file and maintains the order CPU memory requests. It is in control of all commands from the CPU that are destined to the NMI, and performs some error handling and CBox register reads and writes.

The outputs of the control store directly control the NMI address/data slices, some part of the cache, and provide the next address and branch selects to the sequencer itself.

The microsequencer sends read and write commands to the memory and I/O devices for the CPU. These commands are made up of several steps:

- The priority and ordering logic produce the address of the next pending instruction.
- The sequencer fetches the address of the command from the PA file, writes it to the NMI bus, and starts the appropriate timers. For writes, write data is moved to the NMI bus.
- The sequencer checks the received confirmation lines to see if the receiver got the data. For a write, the situation is the same in relation to pending writes, but a second pending read (from the PIBA queue, for example) can be done while the sequencer waits for the read confirmation.
- When the confirmation of the transaction is received from the NMI, the sequencer sets or clears the appropriate state. For writes, this means clearing the valid bit in the write queue state and moving the Q1 address to Q2. For reads, this means getting the appropriate EXPECT READ DATA bit in the function MCA.

The microsequencer MCA contains logic that monitors the loading of the read, write, and PIBA queues in the NMI address/data slices. It also includes logic that keeps PA file reads and writes in order, by giving them a 2-bit number as they are received by the microsequencer MCA. The number is incremented whenever a read that missed is followed by a write. This ensures that there can be no more than a difference of 1 in the numbers of the commands at the front of the queues at any one time. Thus, at a given time, if a read and a write have the same number, the write goes first.

Within the microsequencer MCA, next command select logic looks at the numbers associated with each of the three PA file queues and their valid bits. If more than one is valid, then it uses two blocks of logic to determine which one goes first. One block checks if the write number is less than or equal to the CPU read number; the other checks if the write number is less than or equal to the PIBA read number. If the write is valid and less than or equal to any other valid command, the write is the next command to be processed, otherwise, the command with the lowest number is processed. If both CPU read and PIBA read are lowest (and valid), then the order is CPU read, then PIBA read.

Dispatch logic in the microsequencer MCA generates dispatch microaddresses from the control store based on command types in the read, write, and PIBA queue state logic. It prioritizes pending requests and sends an address to the control store, initiating the first cycle of the sequence.

There are two groups of requests that can be pending.

1. NMI timeouts, a delay-write hit, and a read finish

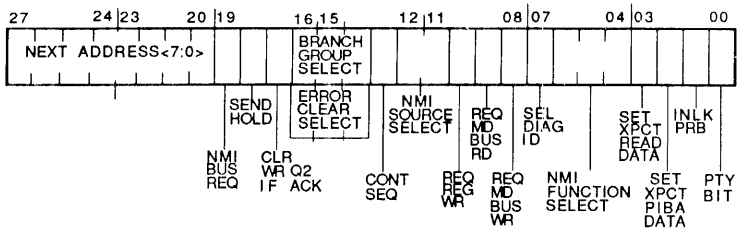
2. Pending requests involved with the PA file

A third path for generating the first cycle of a sequence functions as a bypass path around the PA file queue state logic for commands from the CPU. Because the PA file queue state is loaded in only cycle, a cycle can be saved if nothing else must be serviced by the sequencer, and the command must be sent to the NMI address/date bus.

The first group has the highest priority for the NMI microsequencer. The highest priority is the group of NMI timeouts. The NMI microsequencer in response to a timeout clears the appropriate address location in the PA file to allow further processing to go through and load the timed-out address in the error register. The next request that the sequencer handles is read finish. If the CPU does two read misses to the same cache block, the NMI microsequencer passes the read back into the cache rather than go to memory again after the first one comes back. The lowest priority in this group is the delay-write hit. The NMI microsequencer does a masked read/write into the cache and delivers the data to the ALU and MD registers in the IBox.

If any of the PA file queue entries are waiting for the microsequencer MCA, they are processed if none of the above requests are pending. Since reads and writes have to be kept in order (to avoid various types of stale data problems) simply prioritizing reads and writes will not work. Instead, each location in the PA file is given a number as it comes from the CPU. The next command logic looks at these numbers and selects the next command. Besides being processed in order by the NMI microsequencer, the commands are also finished in order. Thus, the receiver (memory or an I/O device) must confirm that it received the read command or the write command and all of the write data. This is handled by the way the microcode flow is written and the branch on the received confirmation. Hence, a longword write takes four lines of microcode.

Figure 2-14 illustrates the format of the control store microword.



<0> PARITY
Odd parity on the 28 microword bits

<1> INTERLOCK ARB
Re-arbitration for READ LOCK after a WRITE UNLOCK is detected.

<2> SET EXPECT PIBA DATA
Used to set the EXPECT PIBA DATA state bit in FUNC MCA during PIBA READ cycles.

<3> SET EXPECT READ DATA
Used to set the EXPECT READ DATA state bit in FUNC MCA during CPU READ cycles.

<6:4> NMI FUNCTION SELECT
Used by the FUNC MCA to generate NMI FUNCTION CODES.

0 NO-OP
1 DIAGNOSTIC FUNCTION
2 READ
3 READ INTERLOCK
4 WRITE
5 WRITE UNNLOCK
6 PIBA
7 WRITE DATA

<7> SELECT DIAGNOSTIC ID
Select DIAGNOSTIC ID for transmission on the NMI.

1 No Operation
0 Select DIAGNOSTIC ID

<8> REQUEST MD BUS WRITE
Requests the SLC modules to send WRITE DATA on the MD BUS.

1 No Operation
0 Request data

<9> REQUEST MD BUS READ
During Cbox Register Read this bit is used to request the use of the cache, set the REFILL/INVAL select lines to select the READ QUEUE and enable the data onto the MD BUS.

1 No Operation
0 Request MD BUS read

<10> REQUEST REGISTER WRITE
Used during Cbox REGISTER WRITES to enable the actual writing of the registers.

1 No Operation
0 Request Register Write

<12:11> NMI SOURCE SELECT
Select the source of information for transmission on the NMI ADDRESS DATA Lines. It is also decoded and used to start the timeout counter and determine whether ID or MASK should be sent on the NMI.

0 MD BUS
1 READ ADDRESS
2 WRITE ADDRESS
3 PIBA ADDRESS

<13> CONTINUE SEQUENCE
Select between the DISPATCH and the NEXT ADDRESS microaddress.

0 Dispatch
1 Continue sequence

<16:14> BRANCH GROUP SELECT
Selects branch bits for replacement of next microaddress <2:0>. Also performs ERROR CLEAR function in the NMI interface, and ERROR ADDRESS REGISTER (EAR) loading.

<17> CLEAR WRITE Q2 IF ACK
Used to clear the WRITE QUEUE when CONFIRMATION is received from the NMI device.

<18> SEND HOLD
Used to assert the CPU HOLD line on the NMI during write transactions.

<19> NMI BUS REQUEST
Used to asseri NMI BUS REQUEST when arbitrating for use of the NMI bus.

<27:20> NEXT ADDRESS
Because the NEXT MICROADDRESS for continuation of NMI microcode sequences.

	BRANCH ADDR<2> components	BRANCH ADDR<1> components	BRANCH ADDR<0> components	REBOOT	ERROR CLEAR	LD EAR
0	NEXT UADDRS<2>	NEXT UADDRS<1>	NEXT UADDRS<0>	No branch	WRITE	------
1	NEXT UADDRS<2>	RCVD CONF<1>	RCVD CONF<0>		PIBA	------
2	NEXT UADDRS<2>	0	MD BUS BUSY		READ	------
3	NEXT UADDRS<2>	0	MEMORY BUSY		------	------
4	NEXT UADDRS<2>	INTR IF TIMEOUT	Write Unlock		WRITE	------
5	NEXT UADDRS<2>	INTR IF TIMEOUT	ARB OK		PIBA	------
6	Write Unlock	RCVD CONF<1>	RCVD CONF<0>	Select Error Clr & set up LD EAR.	READ	LOAD
7	NEXT UADDRS<2>	0	0		------	LOAD

SCLD-315

Figure 2-14 Control Store Microword Format Diagram

2.3.1.3 **NMI In Control** -- The NMI in control controls data entering the CBox from the NMI (Figure 2-15). It consists of a mask/ID MCA and a function MCA.

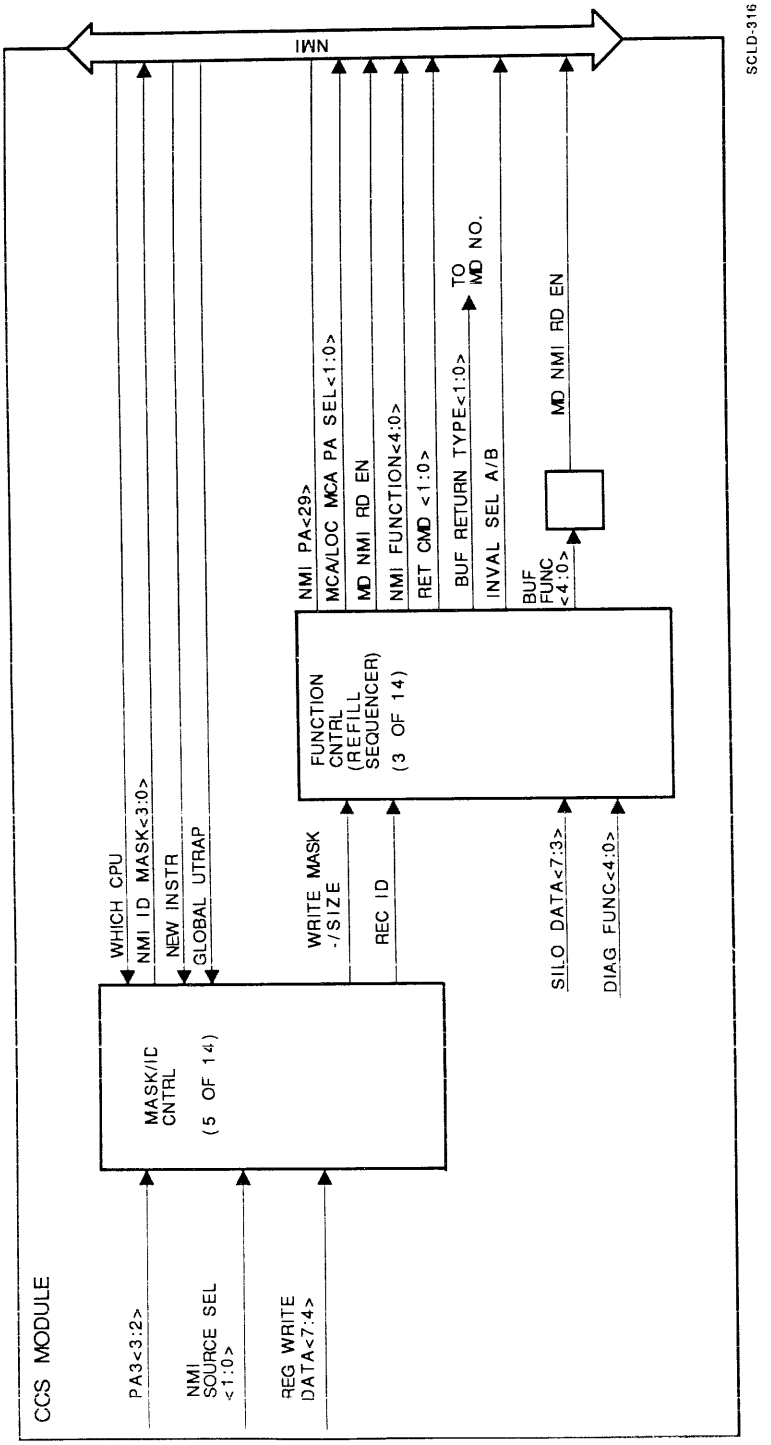


Figure 2-15 NMI In Control - Block Diagram

Mask/ID MCA

The mask/ID MCA stores the valid (or state) queues for the cache write buffers indicating which bytes are valid. The outputs of this chip include the mask/ID field for the NMI, state information for the microsequencer, and information needed for write transactions for the function MCA. Within the mask/ID MCA, valid queue encode logic looks at VA<03:00>, the CPU WRITE command, and the size of the reference to determine what bytes are being written in the NMI address/data slices PA file write buffer, and then sets the corresponding bit in valid queue 1. Conditions are monitored in this block to determine whether or not to move Queue 1 to Queue 2 and whether or not to clear Queue 1. Within the mask/ID MCA, valid queue decode logic provides information needed for the NMI write transaction. From the data in valid Queue 2 logic, the following is determined:

- The size of the NMI write transaction
- The starting address (VA<3:2>) of that write (00 for an octaword, 00 or 10 for quadword, and any for longword writes)
- Whether or not the transaction is a masked or unmasked write
- If the write is masked, the proper mask for each data cycle

The mask/ID MCA contains ID logic that:

- Generates the NMI MASK/ID field during command cycles in which the CBox is the commander using the CPU number and control lines from the CBox microsequencer
- Monitors the MASK/ID field at times when the CBox is transmitting to ensure that the ID was transmitted properly, (otherwise, a multiple transmitter fault has occurred)
- Monitors the MASK/ID field during NMI command cycles for return data to the CBox

The mask/ID MCA also calculates parity on the MASK/ID field of the NMI control. This term is then combined with the parity calculated by the function MCA on the function field in the acknowledgment/arbitration MCA to complete the parity on the NMI control field.

Function MCA

The function MCA is in control of data and invalidates received from the NMI bus. It is completely synchronized with NMI traffic. When the received memory data is refill data for the cache, a check is made to see if read data is expected for that ID, then the function MCA requests and automatically obtains use of the cache and writes the refill data into it.

Any writes to memory space cause the function MCA to perform a two-cycle invalidate. The function MCA (Figure 2-15) serves two functions in the cache:

1. It is the interface to the NMI FUNCTION <4:0> field.

2. It is the controller for received NMI traffic.

In the first case, it is the output path for the NMI microsequencer to drive the FUNCTION field to other devices on the NMI. In the second case, it produces all the control to direct the cache control MCA in loading refill data, or invalidating the cache.

Function Encode -- The output portion of the function MCA is controlled by the microsequencer MCA. One field is output from the control store that selects the function to go to the bus. For writes, these lines are encoded with the mask bit and WRITE SIZE <1:0> lines from the mask/ID MCA. Reads are encoded from the current read function from the microsequencer MCA.

Function Read State -- The function MCA has four state bits associated with read commands. They are:

1. EXPECT READ DATA

   This bit is set by the NMI microsequencer when the confirmation from the receiver of the READ command is returned. When the read data is sent back, the function MCA checks this bit to make sure the data is expected for this ID before writing the data into the cache. The function MCA clears this bit during the last data return cycle. If the reference has a timeout, the NMI microsequencer clears this bit when it enters the timeout service routine.

2. READ SIZE

   This bit is loaded with PA<29> when the read goes to the NMI bus. If PA<29>=0, the read size is hex; if PA<29<=1, then the read size is long. The function MCA uses this bit to determine how much data is to be returned. It is cleared in the same fashion as expect read data.

3. VALIDATE RETURN DATA

   This bit keeps the cache from storing stale data in the following case. If the CPU does a read miss followed by a write to the same cache half-block (or an invalidate occurs to that block), the data it requires from the cache is not the data returned by the read, but the data that had been written to memory. The specific data requested in the read miss is sent to the CPU, but only a longword. This bit is set by address checking logic in the NMI ADD/DATA MCAs and is cleared by the function MCA at the same time as the EXPECT READ DATA BIT. It causes the function not to validate the returned block in the cache.

4. READ Q1 HIT Q2

   This bit keeps the NMI microsequencer from doing two reads to memory for the same half block. If the CPU executes two consecutive reads to the same cache half-block (read quad, for instance), the correct data for the second read is contained in the return for the first read. Instead of reading to memory, the NMI microsequencer directs the read back to the CPU.

   This bit is set by the address checking logic in the NMI ADD/DATA MCAs if the expect read data line is set. It is cleared by the NMI microsequencer when the read finish sequence is executed.

Function PIBA State -- The function MCA has four state bits associated with PIBA commands.

1. EXPECT PIBA DATA -- Same as for read data.

2. PIBA SIZE -- Same as read data, except that it is set by PIBA PA<29>.

3. VALIDATE cache -- Same as read data, except that the check is done against the PIBA Q2 address.

4. OLD/NEW -- This bit is unique to the PIBA state. If, while the PIBA miss is out, a PC change occurs, the PIBA in the PIBA address register is no longer the same one that caused the miss. This means that while the data is being written into the cache, it does not want to be sent to the IB. This bit is set by a PC change while the EXPECT READ DATA bit is set and cleared by the function MCA during the last data return cycle.

Function Decode Logic

The input logic decodes the received function, examines the received ID if READ RETURN and PA <29> if write, and decides what to do. To do a READ RETURN, the received ID has to be checked against the EXPECTING READ DATA bit. If, for that ID (one ID is for PIBA READ, one for CPU READ), it is not expecting data, the UNEXPECTED READ FAULT line is raised. If it is expecting data, the function MCA requests and wins the cache and does the refill.

For a CPU read, the function MCA:

- In the first cycle, loads the refill/invalidate register to write the cache with the data that has been loaded into the cache write latch.

- Also in the first cycle, reads out the MD NUMber of the reference that caused the miss currently being refilled. This directs the first word to the requesting MD register.

- Increments the refill pointer for every cycle the received NMI function indicates READ RETURN DATA CONT, not incremented on PAUSE.

- When the refill counter has reached eight, clears the EXPECT READ RETURN DATA bit if there have been no errors during the sequence.

If the refill is in response to a PIBA READ miss, then the function MCA responds slightly differently in one of two ways. Along with the EXPECT PIBA READ DATA bit in the function MCA, there is a bit that says whether or not there has been a PC change, while the read was out. If there was a PC change then the data is put in the cache just as the read data was with the exception of reading out an MD NUMber. If there was not a PC change, then the function MCA puts out a signal that says step the PIBA with the refill. So if the IB is not full, it will get a longword a cycle until either it becomes full or the refill crosses the wrapped boundary.

During an invalidate operation, the function MCA:

- In the first cycle, loads the refill/invalidate register and looks up the cache tag pointed to by that address.
- In the next cycle, invalidates the line if the previous cycle's lookup was a hit. If it missed, then this cycle will be given to another requester.

2.3.1.4 NMI Arbitration/Acknowledgment

Acknowledgment/Arbitration MCA

The NMI acknowledgment field is used by the NMI "responder" to notify the "commander" of the status of the data transfer. The NMI arbitration field is used to determine the "commander" of the next command cycle. Also included in the MCA is the timeout logic, the fault logic, and the NMI parity logic.

Arbitration Logic

CPU0 has the lowest NMI priority and, in a dual-processor arrangement, CPU1 is just above it. This piece of logic determines if the CPU has been granted the arbitration cycle. This is determined by checking the NMI arbitration lines coming into the chip to see if a device of higher priority is arbitrating for the next cycle. If the CPU number is '0', it is the lowest device in the scheme and it may assume that it has won the bus at this point if no one else wants the bus. However, CPU1 must make sure that its own arbitration line has been asserted the cycle before it desires the bus. This ensures that no other device of lower priority (CPU0) thinks it has the bus. In summary, CPU0 may assume it has won the bus when no one else is arbitrating. CPU1 must additionally check to be sure that its own arbitration line has been asserted, and if it has not, to assert it and try again next cycle. Finally, the arbitration logic sends NMI hold if the transaction is longer than one cycle, and notifies the microsequencer if the bad memory busy response occurs.

Acknowledgment Logic

The NMI arbitration/acknowledgment control (Figure 2-16) has transaction confirmation codes:

- AC
- OK
- BUSY
- BUSY INTERLOCKED
- NO RESPONSE

These codes are transmitted by the receiver one cycle after it receives a command or date. When the NMI microsequencer is a commander, it treats BUSY, BUSY INTERLOCKED, and NO RESPONSE like they were busy responses and keeps retrying the command. When the function MCA is receiving read return data, it only sends ACK OK. The CPU is never busy to returned data.

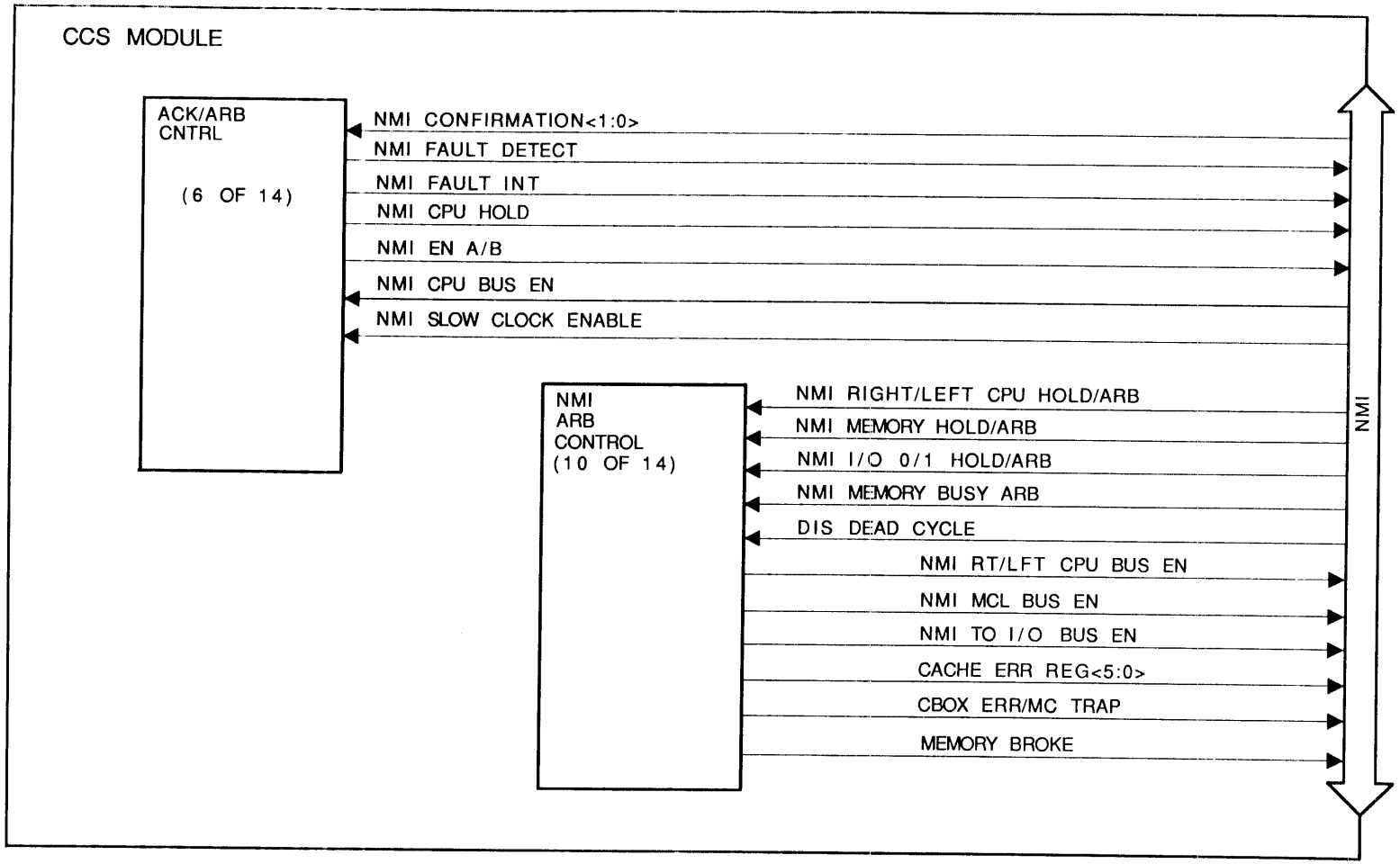
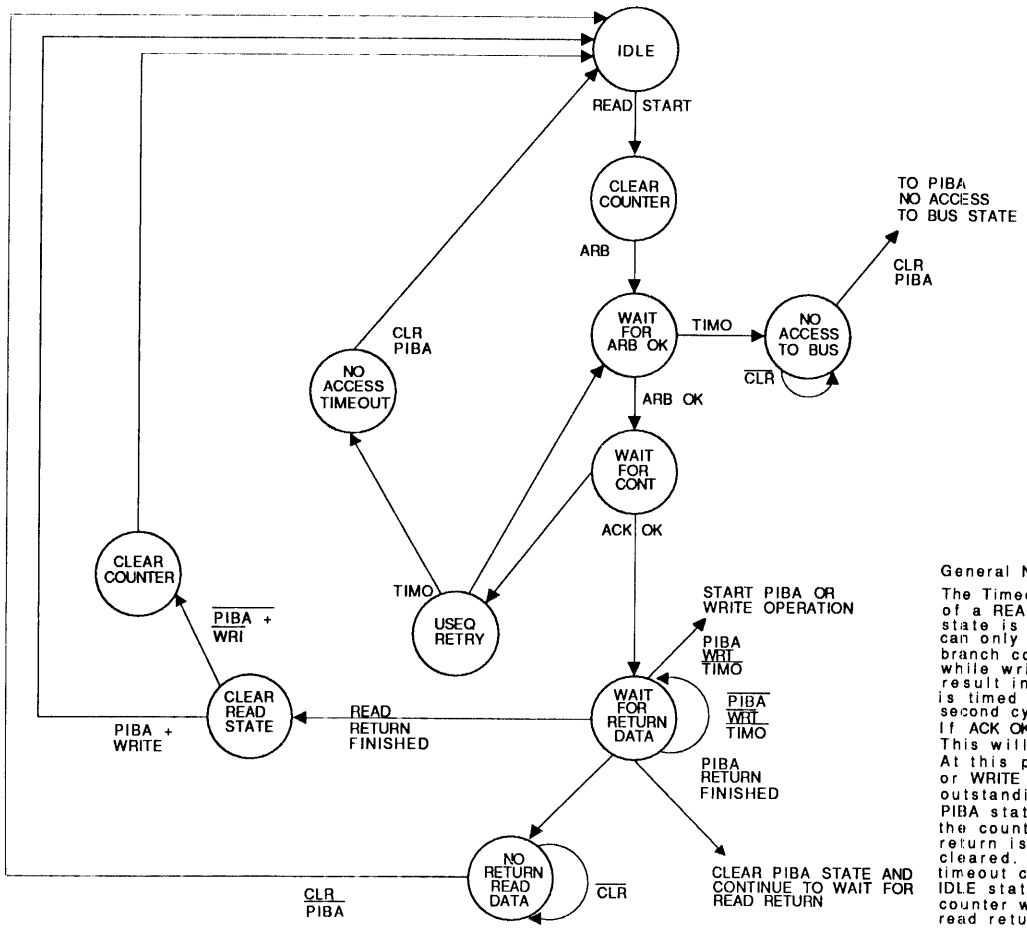


Figure 2-16 NMI Arbitration/Acknowledgment Control - Simplified Block Diagram

Timeout Logic

Basically, this block is a counter that is given a slow clock from the NMI and monitors the three types of CBox transactions (read, write, and PIBA read) for timeouts. For the read and PIBA read, there are two periods to the transaction that are timed. The first is from the time the NMI microsequencer starts the read until the responder confirms the command with ACK OK. This means the READ command can timeout either because there was no access to the NMI bus or the responder was busy or not there. When positive confirmation is received, the timer is restarted to wait for the data. In the first cycle of read return data, the timer associated with that data is set back to the idle state. For writes, only the first case of timeout is necessary. In the event of a timeout, the NMI microsequencer sets the timer to the idle state and clears the address queue of the timed-out transaction. Timeout locks the fault register and NMI silo for examination by system software. These timeouts occur when cycles of the slow clock pass without the transaction being completed. For diagnostic purposes, the timers can be switched from running off the slow clock (system clock divided by approximately one thousand) to running off the normal system clock for very fast timeouts.

Figure 2-17 illustrates the timeout flow.



General Notes on Timeout Counter:

The Timeout counter is cleared on the first attempt of a READ, WRITE, or PIBA cycle. The WAIT FOR ARB OK state is a tight loop in the Cache microcode, which can only be broken by ARB OK or a timeout. These are branch conditions that the microsequencer checks while writing for ARB OK. A timeout at this point should result in a NO ACCESS TO BUS code. The WAIT FOR CONT state is timed by the Cache microcode, waiting for the second cycle after the command/address is sent on the NMI. If ACK OK is set, the EXPECT READ DATA bit will be set. This will result in the WAIT FOR RETURN DATA state. At this point, the microsequencer can begin a PIBA or WRITE operation, or if there is a PIBA return outstanding, the PIBA RETURN can come back and clear the PIBA state. If there is no PIBA or WRITE to be done, the counter will count until timeout occurs or the read return is finished. At this time, the read state will be cleared. If there is no other cycle in progress, the timeout counter will be disabled, and return to the IDLE state. If a PIBA read is outstanding, the counter will continue to count until the PIBA read returns, or it times out.

SCLD-318

Figure 2-17 Timeout - Flow Diagram

2.3.1.5 **CBox NMI Registers** -- Figure 2-18 illustrates CBox registers that are read/written from/to the NMI (directly from the NMI and indirectly by means of the NMI address/data slices).

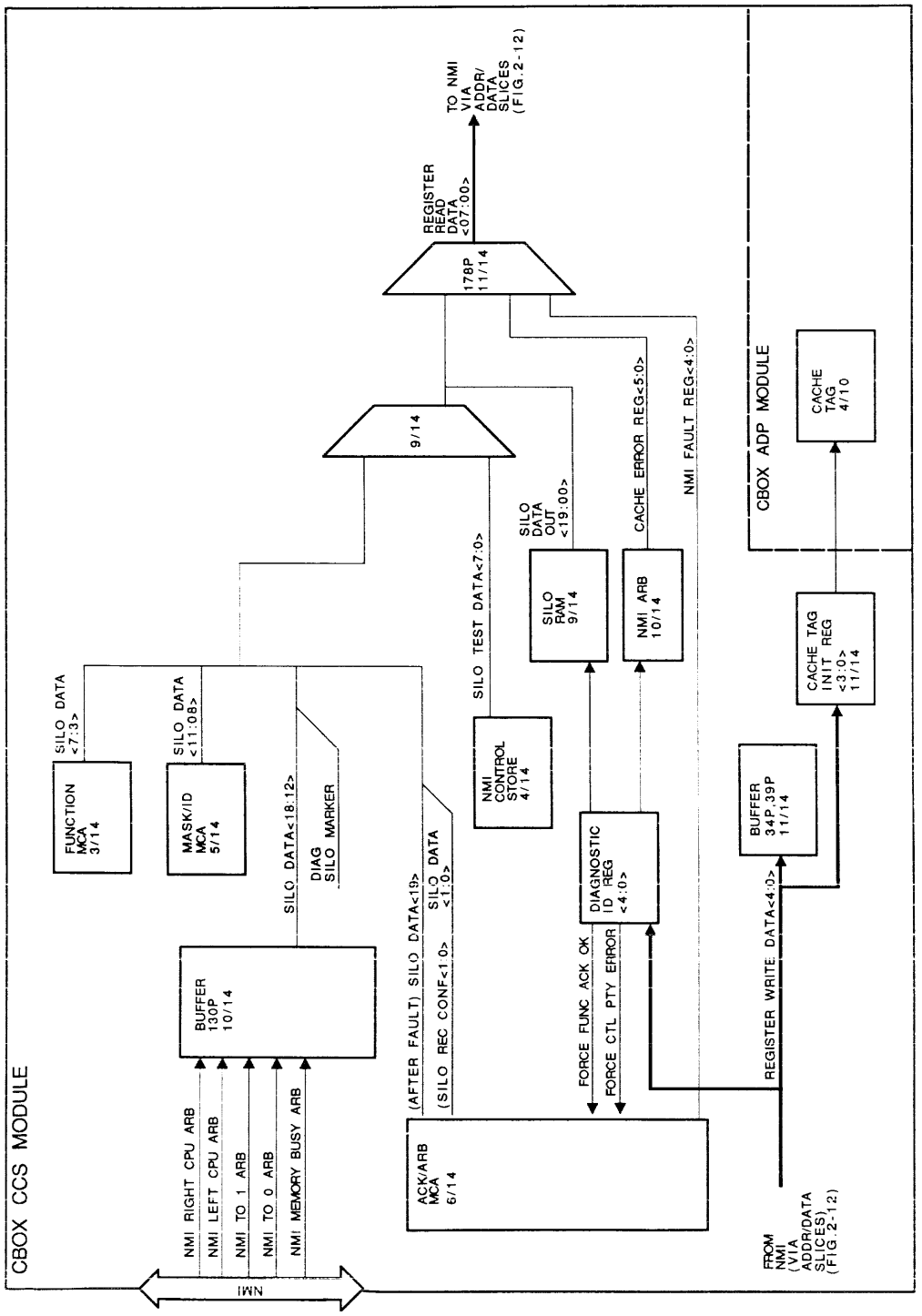


Figure 2-18 CBox NMI Registers - Location Diagram

Cache Register

The cache control register controls the overall operation of the cache. It is the only READ/WRITE register in the CBox.

Figure 2-19 illustrates and Table 2-3 describes the format of the Cache Register.

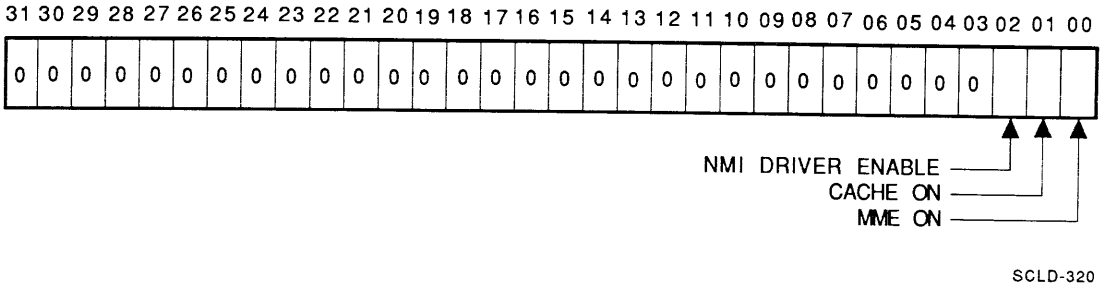


Figure 2-19 Cache Register - Bit Format

Table 2-3 Cache Register - Bit Descriptions

Bit	Name	Description
<2>	NMI DRIVER ENABLE	When NMI DRIVER ENABLE is cleared, it prevents the NMI ENABLE signal from being set in the CBox, in effect turning off the NMI drivers. This permits some diagnostic functions to be performed without disturbing the NMI bus. This READ/WRITE bit is cleared by CPU INIT. **NOTE** **This bit must be set for normal CPU operation to occur.**
<1>		CACHE ON When this bit is set, cache operation is enabled. When it is cleared, all cache references will be misses. Additionally, all NMI D-stream reads will be reads to conserve NMI bus cycles. This READ/WRITE bit is cleared by CPU INIT.

Table 2-3 Cache Register - Bit Descriptions (Cont)

Bit	Name	Description
<0>	MME ON	This memory management enable bit enables the virtual memory management hardware in the translation buffer. The operation of this bit is generally defined by VAX 8800 architecture. When this bit is cleared, the virtual address range of the system is 30 bits rather than 32 bits. The mapping of virtual physical addresses, with MME cleared, is PA<31:30> = [00], PA<29:00> = VA<29:00>. When MME is cleared, the translation buffer does not perform protection checking or page cross checking. This read/write bit is cleared by CPU INIT.

Cache Error Register

The cache error register holds the error bits for the various parity checking networks in the CBox -- other than the NMI interface. Included are:

- VA bus parity
- Physical address parity (this generally means TB parity)
- Memory data parity, which is the parity of the MD bus information received at the cache data path end
- Cache tag parity
- TB tag parity
- NMI microsequencer parity
- NMI data parity error on:
  - Data destined for an EBox MD register
  - BAD READ DATA that is not prefetch data
  - Bad PIBA data for the current PIBA

The entire cache error register locks on the first error, and is unlocked by a CBox register write to address VA = 20 (hexadecimal).

The cache error register is cleared by CPU INIT.

Cache Error Register - Byte 2 -- Figure 2-20 illustrates and Table 2-4 describes the format of the Cache Error Register - Byte 2.

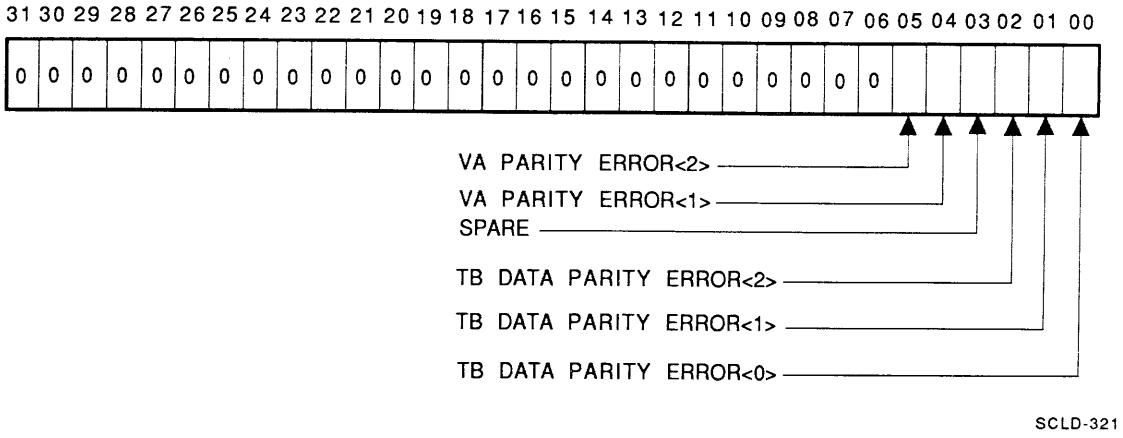


Figure 2-20 Cache Error Register Byte 2 - Bit Format

Table 2-4 Cache Error Register Byte 2 - Bit Descriptions

Bit	Name	Description
<05:04>	VA PARITY ERROR <2:1>	These bits are set to indicate that there was a parity error in the VA on bits <31:16> or bits <15:09>, respectively.
<03>	Spare	Read as a ZERO.
<02:00>	TB DATA PARITY ERROR <2:0>	These bits are set on a parity error in the translation buffer (TB) data. The bits pertain to the following fields of the TB data: TB DATA<29:23> TB DATA<22:16> TB DATA<15:09>

Cache Error Register - Byte 1 -- Figure 2-21 illustrates and Table 2-5 describes the format of the Cache Error Register - Byte 1.

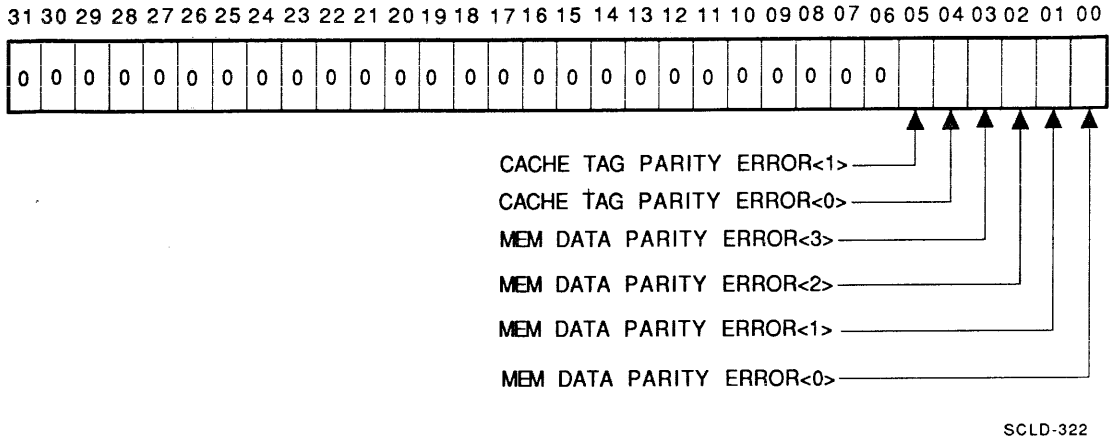


Figure 2-21 Cache Error Register Byte 1 - Bit Format

Table 2-5 Cache Error Register Byte 1 - Bit Descriptions

Bit	Name	Description
<05:04>	CACHE TAG PARITY ERROR<1:0>	These bits are set when there is a parity error detected on the cache tag data <28:24> or <23:16>, respectively.
<03:00>	MEMORY DATA PARITY ERROR<3:0>	These bits indicate that a parity error was detected on MD bus data by the cache data buffer. There is one parity error bit for each byte of data.

Cache Error Register - Byte 0 -- Figure 2-22 illustrates and Table 2-6 describes the format of the Cache Error Register - Byte 0.

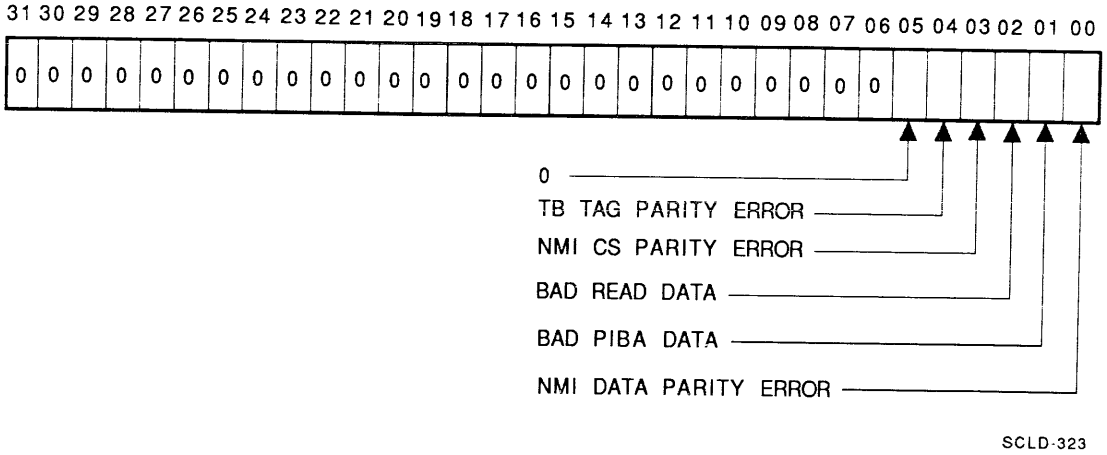


Figure 2-22 Cache Error Register Byte 0 - Bit Format

Table 2-6 Cache Error Register Byte 0 - Bit Descriptions

Bit	Name	Description
<05>	Read as a ZERO	
<04>	TB TAG PARITY ERROR	This bit is set when there is a parity error detected in the TB tag RAMs. It is the logical OR of parity check conditions for the TB tag, MBIT, PROTection, and VALID fields.
<03>	NMI CS PARITY ERROR	This indicates that there was a parity error in the 28 bits of the CBox NMI control store data.
<02>	BAD READ DATA	BAD READ DATA is set when the return data for the CPU read miss received the BAD DATA NMI function code (which indicates that the data that was being returned from the memory was uncorrectable) and the data was not prefetch read data.
<01>	BAD PIBA DATA	BAD PIBA DATA is set when the return data for the CPU PIBA miss received the BAD DATA NMI function code (which indicates that the data that was being returned from the memory was uncorrectable) and the data was a return for the current I-stream.

Table 2-6 Cache Error Register Byte 0 - Bit Descriptions (Cont)

Bit	Name	Description
<00>	NMI DATA PARITY ERROR	NMI DATA PARITY ERROR is set when there is a parity error on return data that was destined for an MD register, or data that was destined for the current PIBA.

NMI Fault/Status Register

NMI Fault/Status Register - Byte 1 -- Figure 2-23 illustrates and Table 2-7 describes the format of the NMI Fault/Status Register - Byte 1.

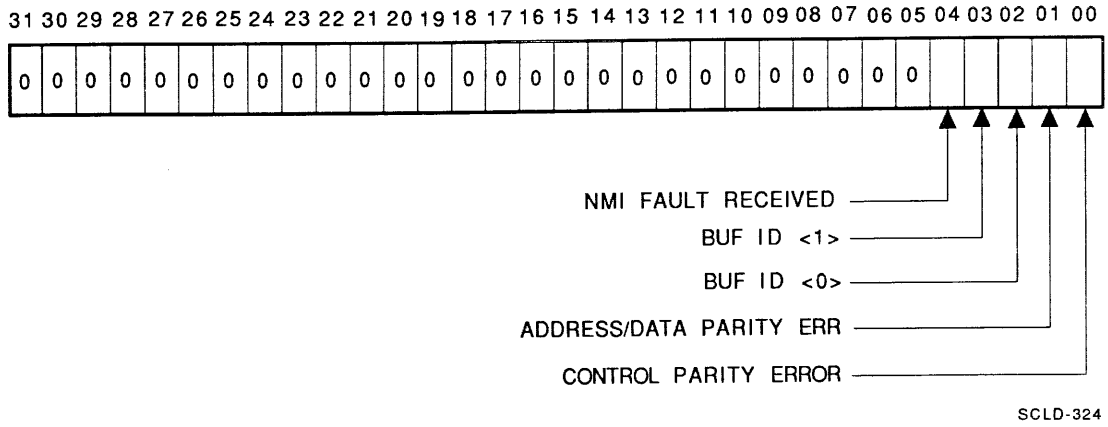


Figure 2-23 NMI Fault/Status Register Byte 1 - Bit Format

Table 2-7 NMI Fault/Status Register Byte 1 - Bit Descriptions

Bit	Name	Description
<04>	NMI FAULT RECEIVED	This bit shows the state of the NMI FAULT wire at the time the register is read. If this bit is not set, bits <1:0> of NMI fault/status register byte 1 and bits <4:3> of NMI fault/status register byte (that the CPU detects) have no meaning.

Table 2-7 NMI Fault/Status Register Byte 1 - Bit Descriptions (Cont)

Bit	Name	Description
<03:02>	BUF ID <1:0>	BUF ID <1:0> is an encoded field used to determine which of the transaction buffers had a timeout condition. The encoding is shown below: BUF <1> ID <0> Buffer Code 0 0 No Timeout 0 1 Write Timeout 1 0 Read Timeout 1 1 PIBA Timeout
<01:00>	PARITY ERROR FAULT	ADDRESS/DATA PARITY ERROR and CONTROL PARITY ERROR indicate that the CBox detected a parity error on the ADDRESS/DATA or CONTROL parity lines on the NMI. These are NMI FAULT conditions, and are held from the time the fault was detected until the NMI FAULT line is cleared (by a transaction to a memory CSR). If the NMI FAULT received bit is not set, these bits have no meaning. The information that is latched in these bits pertains to the cycle on the NMI in which the fault was detected, not the cycle in which the FAULT line was asserted.

NMI Fault/Status Register - Byte 0 -- Figure 2-24 illustrates and Table 2-8 describes the format of the NMI Fault/Status Register - Byte 0.

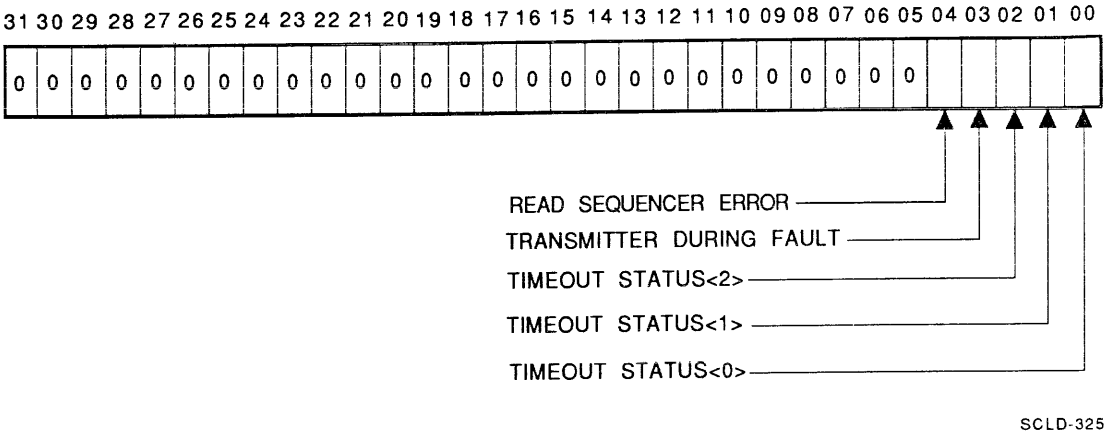


Figure 2-24 NMI Fault/Status Register Byte 0 - Bit Format

Table 2-8 NMI Fault/Status Register Byte 0 - Bit Descriptions

Bit	Name	Description
<04>	READ SEQUENCE ERROR Fault	Read Sequence Error is another of the NMI FAULT conditions. It is set when read return data is sent to the CPU when no READ command was outstanding. This is also one of the NMI FAULT REGISTER bits. If NMI FAULT RECEIVED is not set, this bit has no meaning.
<03>	TRANSMITTER DURING FAULT	TRANSMITTER DURING FAULT indicates that the CPU was transmitting on the NMI in the cycle that caused the fault. It is undefined if the NMI FAULT RECEIVED bit is not set.
<02:00>	TIMEOUT STATUS <2:0>	Three-bit field for the type of timeout that occurred. The following table shows the encoding of those bits:

<2>	<1>	<0>	Timeout Code
0	0	0	No Timeout
0	0	1	Reserved
0	1	0	Interlock Timeout
0	1	1	No Return Read Data
1	0	0	No Access - no response
1	0	1	No Access to bus
1	1	0	No Access - interlocked
1	1	1	No Access - busy

NMI Error Address Register

The NMI error address register holds the address of a CPU NMI transaction that has timed out on the NMI. It is loaded with the address from the PA FILE under the control of the NMI control store microcode when a timeout occurs. This is done by asserting the NMI source select bits and strobing the LOAD ERROR ADDRESS REGISTER signal at the correct time. The NMI error address register is read only to the CPU microcode. Figure 2-25 illustrates and Table 2-9 describes the format of the NMI Error Address Register.

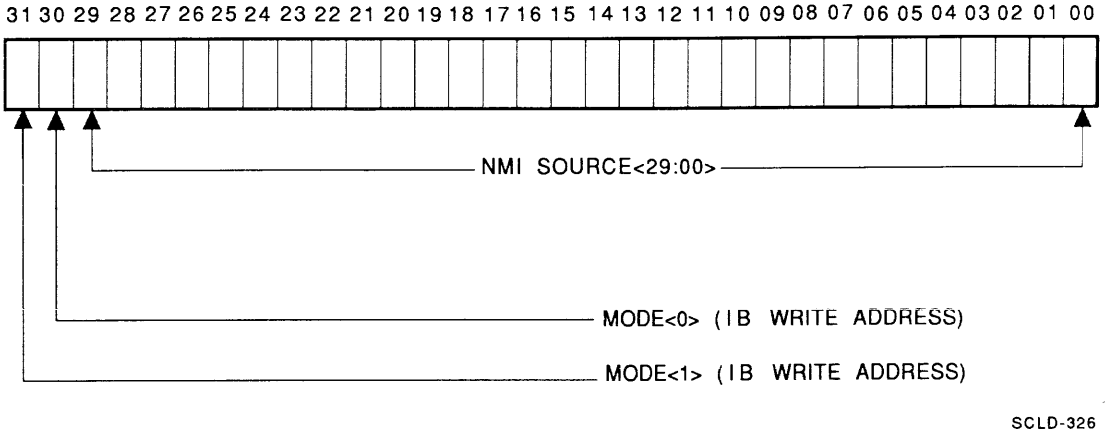


Figure 2-25 NMI Error Address Register - Bit Format

Table 2-9 NMI Error Address Register - Bit Descriptions

Bit	Name	Description
<31:30>	ERROR ADDRESS REGISTER	In the NMI PA file, the CPU ACCESS MODE for the write is stored along with the write address. If there is a timeout of a write operation on the NMI, bits <31:30> of the error address register are loaded with these saved access mode bits at the same time the write address is being loaded into bits <29:00>. These bits are undefined for read and PIBA timeouts, and should be ignored. Only bits <29:00> contain valid address information for these types.

Table 2-9 NMI Error Address Register - Bit Descriptions (Cont)

Bit	Name	Description
		For diagnostic purposes, there is the capability to load the MD bus data (the source of the NMI write data) into the error address register. When this diagnostic capability is used, bits <31:30> will contain the two most significant bits of the MD bus data. This diagnostic function can only be accessed through the use of a special CBox microcode sequence, which diagnostic programmers create and use in their testing.
<29:00>	Error Address Register	Normally these bits contain address bits <29:00> of an address that has timed out on the NMI. This register is loaded under control of the CBox microcode.

NMI Silo

The NMI silo is a 255-cycle history file of NMI events preceding a fault, plus the events for the faulted cycle. The NMI silo is normally written every cycle with the contents of a selected group of NMI fields that indicate the type of transaction taking place in that cycle. Address and data information are not saved. When the FAULT signal on the NMI is driven, the NMI silo can be prevented from further loading -- thus capturing information about the faulting cycle and its predecessors. When the NMI silo is locked, reads of the silo return the contents of the most recent cycles, starting with the faulting cycle, with successive reads returning older information. In other words, when the silo is in running mode, the address counter increments after each write; and when the silo is locked, the address counter decrements after each read.

The contents of this register are UNDEFINED if the silo is not locked.

This register is READ ONLY for all bit locations.

NMI Silo - Byte 2 -- Figure 2-26 illustrates and Table 2-10 describes the format of the NMI Silo - Byte 2.

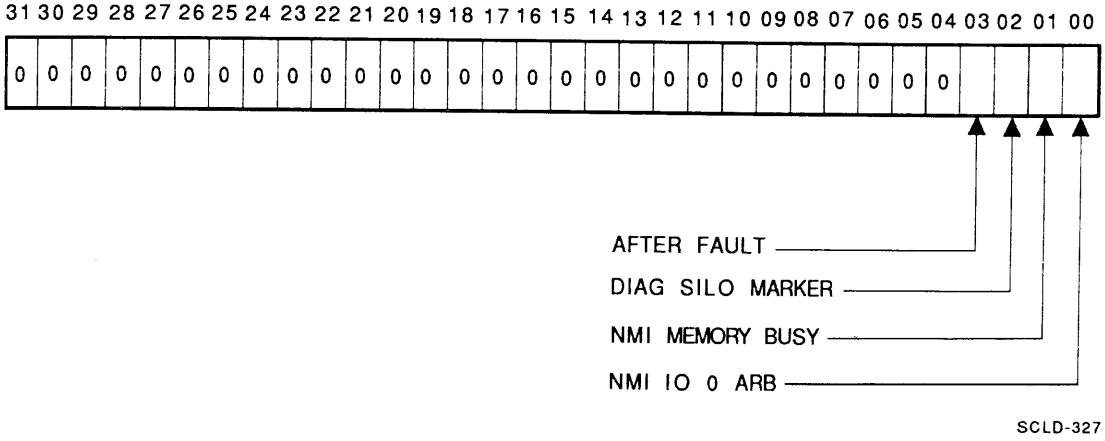


Figure 2-26 NMI Silo Byte 2 - Bit Format

Table 2-10 NMI Silo Byte 2 - Bit Descriptions

Bit	Name	Description
<03>	AFTER FAULT	The AFTER FAULT signal is asserted for one cycle after NMI FAULT is deasserted on the NMI. In the silo, this bit indicates the first line of information added to the silo after the deassertion of the NMI fault wire. This bit distinguishes information added to the silo leading to two separate faults. In general, it will only be set when the system has a high error rate.
<02>	DIAG SILO MARKER	The DIAG SILO MARKER is a synonym for the LOAD ADDRESS ERROR REGISTER bit that comes out of the CBox control store. It helps diagnostic programmers determine the start of certain special diagnostic sequences when examining the silo after a diagnostic test.
<01:00>	NMI Arbitration lines	The NMI MEMORY BUSY and NMI IO 0 ARB are two of the NMI arbitration lines.

NMI Silo - Byte 1 -- Figure 2-27 illustrates and Table 2-11 describes the format of the NMI Silo - Byte 1.

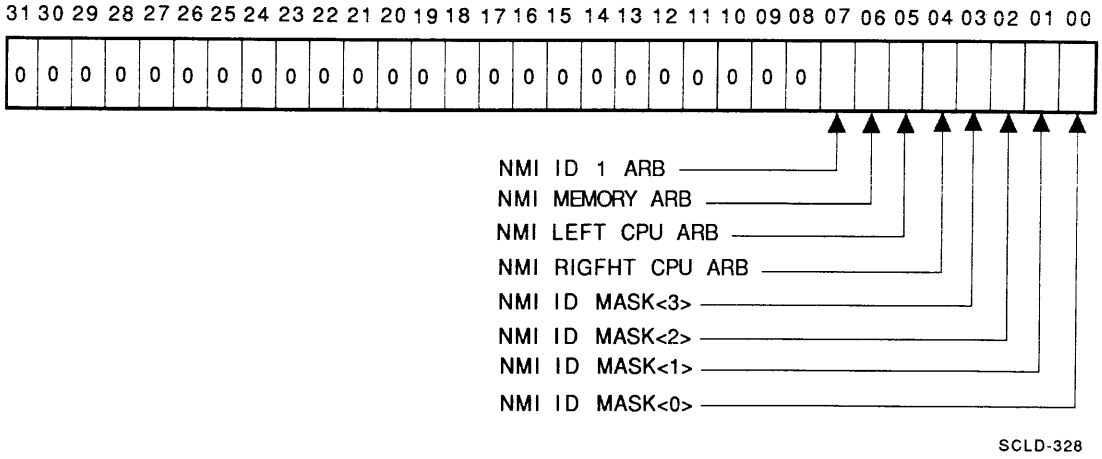


Figure 2-27 NMI Silo Byte 1 - Bit Format

Table 2-11 NMI Silo Byte 1 - Bit Descriptions

Bit	Name	Description
<07:04>	NMI Arbitration lines	These bits capture the state of various arbitration lines for devices on the bus.
<03:00>	NMI ID MASK<3:0>	The NMI ID MASK bits contain the ID of the commander during NMI command/address cycles, and the byte mask for write data cycles.

NMI Silo - Byte 0 -- Figure 2-28 illustrates and Table 2-12 describes the format of the NMI Silo - Byte 0.

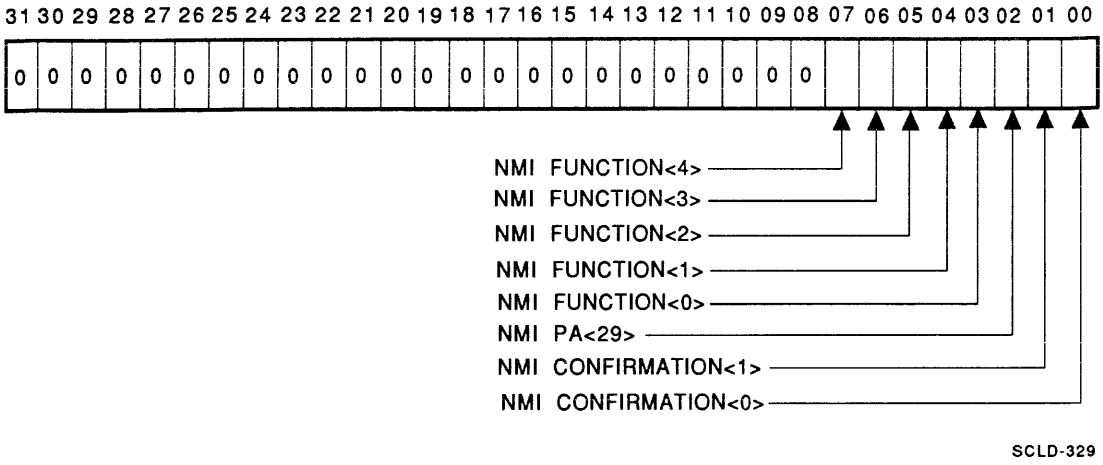


Figure 2-28 NMI Silo Byte 0 - Bit Format

Table 2-12 NMI Silo Byte 0 - Bit Descriptions

Bit	Name	Description
<07:03>	NMI FUNCTION <4:0>	The function field on the NMI specifies the type of bus transfer for the cycle.
<02>	NMI PA <29>	This bit records the state of address/data bit <29>, which is of interest during command/address transactions because it indicates that I/O address space is being accessed.
<01:00>	NMI CONFIRMATION	These are the confirmation lines, which are the response to a command/address transfer two cycles earlier.

Cache TAG Initialization Register

Figure 2-29 illustrates and Table 2-13 describes the format of the Cache TAG Initialization Register.

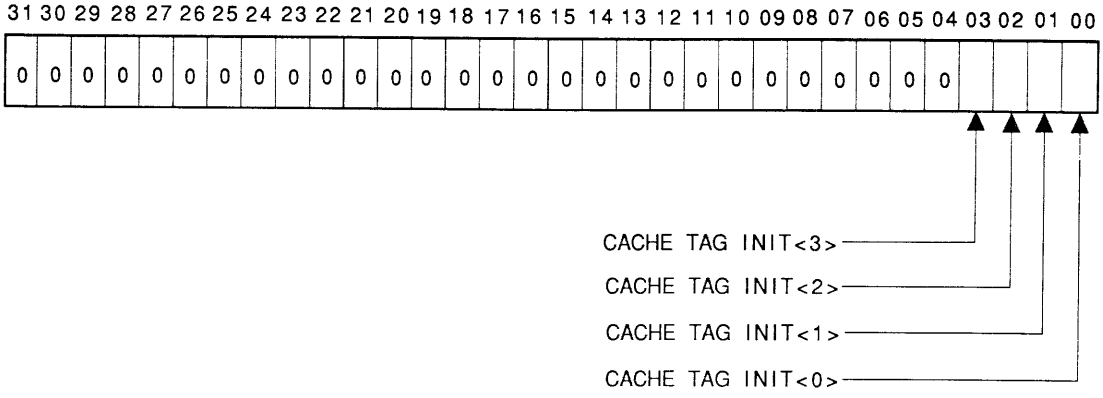


Figure 2-29 Cache TAG Initialization Register - Bit Format

Table 2-13 Cache TAG Initialization Register - Bit Descriptions

Bit	Name	Description
<03:00>	CACHE TAG INIT<3:0>	These bits are used in conjunction with the cache INIT microcode command to load the cache tag valid bits. The main purpose is to initialize the cache tag RAMS after powerup by putting good parity and all zero valid bits into the cache. The cache TAG bits are cleared by CPU INIT. The CACHE TAG INIT bits must be cleared before normal CPU operation.

Diagnostic ID Register

Figure 2-30 illustrates and Table 2-14 describes the format of the Diagnostic ID Register.

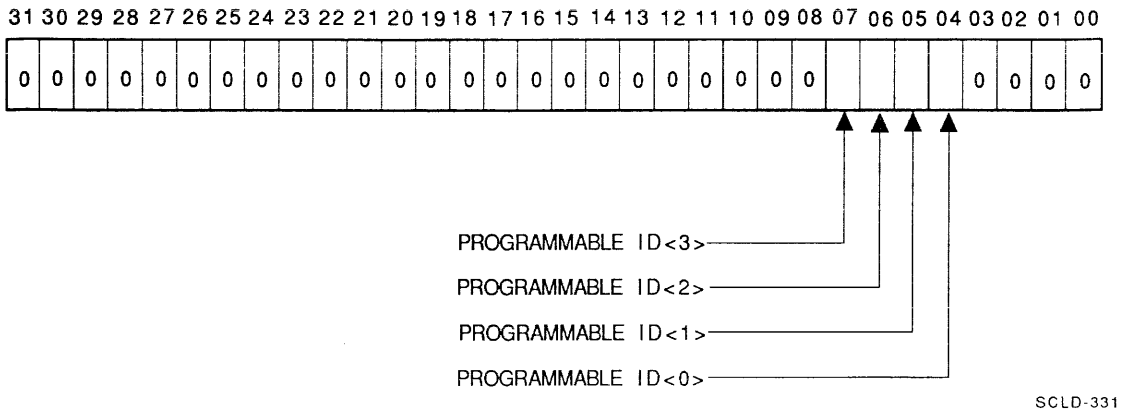


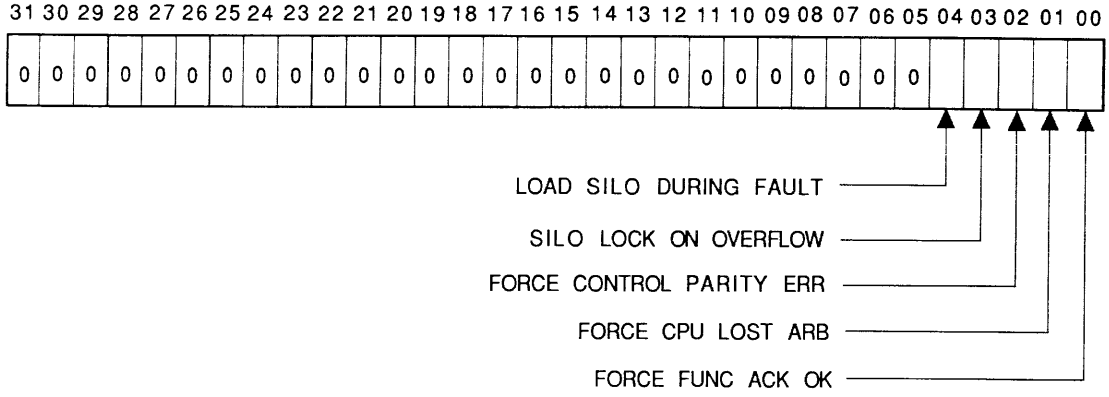
Figure 2-30 Diagnostic ID Register - Bit Format

Table 2-14 Diagnostic ID Register - Bit Descriptions

Bit	Name	Description
<07:04>	PROGRAMMABLE ID <3:0>	This is a diagnostic feature that allows the diagnostic programmer to replace normal ID of an NMI transfer with another ID to perform some error checking functions in the CBox. The programmable ID is enabled by the DIAGNOSTIC ID SELECT bit out of the CBox microcode. The PROGRAMMABLE ID bits are cleared by CPU INIT.

Diagnostic Control Register

Figure 2-31 illustrates and Table 2-15 describes the format of the Diagnostic Control Register.



SCLD-332

Figure 2-31 Diagnostic Control Register - Bit Format

Table 2-15 Diagnostic Control Register - Bit Descriptions

Bit	Name	Description
<04>	LOAD SILO DURING FAULT	Prevents the SILO from locking on NMI FAULT. This bit is asserted low: that is, 0 = asserted. It is cleared by CPU initialization so LOAD SILO DURING FAULT is asserted after CPU INIT. This bit must be deasserted (by writing a one to it) to cause the silo to lock properly on FAULT.

Table 2-15 Diagnostic Control Register - Bit Descriptions (Cont)

Bit	Name	Description
<03>	SILO LOCK ON OVERFLOW	Causes the NMI silo to stop loading when the silo address counters overflow. This permits the silo to be stopped even in the absence of the NMI FAULT signal, which is the normal mechanism for locking the silo. This bit is asserted low: that is, 0 = asserted. It is cleared by CPU initialization so SILO LOCK ON OVERFLOW is asserted after CPU INIT. This bit must be deasserted (by writing a one to it) to allow the silo to correctly continue to load (until FAULT is received.)
<02>	FORCE CONTROL PARITY ERR	Forces the NMI control parity generation logic to transmit bad parity along with the current FUNCTION/ID MASK transfer on the NMI. Cleared by CPU INIT.
<01>	FORCE CPU LOST ARB	Forces the CPU to get ARB OK from the NMI bus arbitration logic. Cleared by CPU INIT.
<00>	FORCE FUNC ACK OK	This enables diagnostic programmers to generate the ACK OK signal, which indicates that the CPU has sent a command to the NMI and received positive acknowledgment to the command, which permits testing of some hardware in the CBox without actually having to use the NMI bus.

Digital Equipment Corporation • Bedford, MA 01730