C38xxVector Processor Specification

Document No. 417-001248-000

Revision 1.1
April 2, 1991

INTERNAL USE ONLY

PROPRIETARY DISCLAIMER

This document is **proprietary.** As such, it is **not** approved for field or customer distribution. It is approved for **internal use only.** Distribution or use outside CONVEX is **strictly prohibited.**

CONVEX Computer Corporation

6 The Control Queues (NQ) 6-1
6.1 Queue inputs 6-1
6.2 Level 2 outputs 6-1
6.3 Level 3 outputs 6-1
6.4 Back-door outputs for Add and Multiply pipes 6-2
6.5 Back-door outputs for the Load pipe 6-2
7 Back-Door Controller (BD) 7-1
7.1 Add pipe back-door logic (page 1) 7-1
7.2 Multiply pipe back-door logic (page 1) 7-1
7.3 Load pipe back-door logic (page 1) 7-2
7.3.1 Load BD control signals 7-2
7.3.2 The last level of the load pipe control queue 7-3
7.3.3 Clock Extend generation (page 1) and distribution (page 2) 7-3
8 The Function Pipes 8-1
8.1 The Add Function Pipe (AFP) 8-1
8.1.1 The AFC controller 8-1
8.1.2 The NDIVs on AFP 8-4
8.1.3 The NFAD_A and NMISC_A on the AFP 8-5
8.1.4 The AFP result mux (page 4) 8-6
8.2 The Multiply Function Pipe (MFP) 8-6
8.2.1 The MFC controller 8-6
8.2.2 The NMULs on MFP 8-8
8.2.3 The NFAD_M and NMISC_M on the MFP 8-9
8.2.4 The MFP result mux (page 3) 8-9
10 The Output Staging Controller (OSCTL) 10-1
10.1 Normal mode operation 10-1
10.2 Fault mode operation 10-2
11 The Status Logic (STAT) 11-1
11.1 Compare Results 11-1
11.2 NVP PSW Bits 11-1
12 Clock Generation Logic (NVP_CLK) 12-1
12.1 Clock generation 12-1
12.1.1 Clock fanout 12-1
12.1.2 Reduction of 3x clock to 1x (page 1) 12-1
12.2 Hard error generation (page 1) 12-2
12.3 RSLT_PARITY_ENABLE (page 1) 12-2
12.4 SRAM write controls 12-2
12.4.1 SRAM Read Mode 12-3
12.4.2 SRAM Write Mode 12-3
12.4.3 SRAM Output Disable mode 12-3

13 Scan/Context controller (NVP_CLK) 13-1
13.1 Functional overview of context switches 13-1
13.1.1 Context save (fault) 13-1
13.1.2 Context restore (rtnc) 13-2
13.2 Hardware implementation of the context controller 13-2
13.2.1 The context state machine 13-3
13.2.2 Control signals from the state machine 13-5
13.3 Hardware implementation for rtnc 13-6
13.4 Generation of Scan and Context controls (page 4) 13-6
14 Microcode 14-1
14.1 Notational Conventions 14-1
14.2 The VD Microinstruction 14-2
14.3 The VD Microlanguage 14-6
14.3.1 The VL0 Constructs 14-6
14.3.2 The Register Allocation Constructs 14-6
14.3.3 The Register Mapping Constructs 14-7
14.3.4 The VM Constructs 14-7
14.3.5 The Scalar Function Constructs 14-8
14.3.6 The Pipe Length Constructs 14-8
14.3.7 The Rate_2X Construct 14-8
14.3.8 The Function Pipe Constructs 14-8
14.3.9 The Parity Construct 14-11
14.4 VD Examples 14-11
14.4.1 ADD.D_V 14-11
14.5 The UA Microword 14-11
14.6 The UA Microlanguage 14-14
14.6.1 VRF Operand Select Construct 14-14
14.6.2 Counter Control Construct 14-14
14.6.3 Pipe Control Construct 14-15
14.6.4 The Last Element Identification Construct 14-15
14.6.5 The Test Condition Construct 14-16
14.6.6 Branch Constructs 14-16
14.6.7 The Parity Construct 14-17
14.7 UA Examples 14-17
14.7.1 UA_VOPV 14-17
14.7.2 UA_VOPV_ACC 14-17
14.7.3 UA_CPRS 14-18
14.8 The UL Microword 14-18
14.9 The UL Microlanguage 14-21
14.9.1 VRF Operand Select Construct 14-21

14.9.2 Counter Control Construct 14-21
14.9.3 Pipe Control Construct 14-22
14.9.4 The Last Element Identification Construct 14-22
14.9.5 The Test Condition Construct 14-23
14.9.6 Branch Constructs 14-23
14.9.7 The Source Control Construct 14-24
14.9.8 The Destination Constructs 14-24
14.9.9 The Parity Construct 14-25
14.10 UL Examples 14-25
14.10.1 UL_LD_V 14-25
14.10.2 UL_LD_V_MASK 14-25
14.10.3 UL_STVI_V_ACC 14-25
14.10.4 UL_MOV_S_V 14-26
15 Inside NVD 15-1
15.1 Overview 15-1
15.2 Caveats 15-1
15.3 Chaining Check 15-1
15.4 Read Port Allocation and Hazard Check 15-1
15.5 Write Port Allocation and Hazard Check 15-4
15.6 Function Pipe Check 15-5
15.7 Read Overruns Write Check 15-5
15.8 Write Overruns Read Check 15-7
15.9 VM Allocation and Chaining Hazard Checking 15-9
15.9.1 VM Chaining Check 15-9
15.9.2 Serial/Parallel VM hazards 15-12
15.9.3 VM Write Port Hazards 15-12
15.9.4 VL Change Hazard 15-12
15.10 VM Rate_2x Hazard Check 15-12
15.11 VM Accelerate/Normal Selection and Edit Hazards 15-13
15.11.1 VM Accelerate/Normal Selection 15-13
15.11.2 Edit Hazard Checking 15-13
15.12 Merge Hazard Check 15-15
15.13 PSW Hazard Logic 15-15
Signal List .... Appendix A

List of Figures

Figure 2-1 – C38xx Vector Processor 2-2
Figure 2-2 – VD state machine 2-3
Figure 2-3 – IPCTL instruction queue 2-7
Figure 2-4 – Scalar to Vector Dispatch interface 2-8
Figure 3-1 – Scalar to Vector bus interface timing 3-1
Figure 3-2 – Input Staging 3-3
Figure 4-1 – Vector Register Files 4-2
Figure 4-2 – NVRF RAM Bank Organization 4-3
Figure 5-1 – NVM Block Diagram 5-3
Figure 6-1 – NQ Block Diagram 6-3
Figure 8-1 – The Add Function Pipe (AFP) 8-1
Figure 8-2 – The Multiply Function Pipe (MFP) 8-7
Figure 10-1 – Vector to Scalar bus interface 10-2
Figure 12-1 – Writing the UA WCS SRAMs 12-4
Figure 13-1 – Mapping of ring segments to SCAN_OUT_BUS<31..0> 13-2
Figure 13-2 – Context State Machine Transition Diagram 13-4
Figure 1-1 – Chaining Check 15-2
Figure 1-2 – Read Port Allocation and Hazard Check 15-3
Figure 1-3 – Write Port Allocation and Hazard Check 15-6
Figure 1-4 – Function Pipe Check 15-7
Figure 1-5 – Read Overruns Write Hazard Check 15-8
Figure 1-6 – Write Overruns Read Hazard Check 15-10
Figure 1-7 – VM Allocation and Chaining Check 15-11
Figure 1-8 – VM Rate 2x Hazard Check 15-13
Figure 1-9 – VM Accelerate/Normal Selection and Edit Hazard Check 15-14
Figure 1-10 – Merge Hazard Check and PSW Hazard Logic 15-15

List of Tables

Table 2-1 Vector Dispatch to NSP Interface Signals 2-1
Table 3-1 Scalar-to-Vector Interface Signals 3-1
Table 3-2 IS_MUX modes 3-4
Table 4-1 Mapping of NVRF OP<7..0> to System OP<63..0> 4-5
Table 4-2 Mapping of NVRF PAR<1..0> to System PAR<7..0> 4-5
Table 4-3 Mapping of NVRF PAR<1..0> to NIBBLE_PAR<7..0> 4-6
Table 5-1 Steering of LQ_RSLT_DAT<31..0> by L_DST_CTL<2..0> 5-5
Table 5-2 Selection for NVM pins VM_DAT<31..0> 5-5
Table 8-1 NDIV Output Enables 8-4
Table 10-1 Vector to Scalar interface signals 10-1
Table 10-2 OSCTL control signals 10-2
Table 11-1 Vector to Scalar PSW bits 11-1
Table 12-1 System interface to NVP_CLK 12-1
Table 12-2 100474 SRAM Write/Select Modes 12-2
Table 13-1 NVP Context Control Inputs 13-1
Table 13-2 NVP Scan Control Inputs 13-6
Table 14-1 VL0 Constructs 14-6
Table 14-2 Register Allocation Constructs 14-6
Table 14-3 Register Mapping Constructs 14-7
Table 14-4 VM Constructs 14-7
Table 14-5 VD Scalar Function Constructs 14-8
Table 14-6 VD Pipe Length Constructs 14-8
Table 14-7 VD Function Pipe Constructs 14-9
Table 14-8 UA VRF Operand Select Construct 14-14
Table 14-9 UA Counter Control Construct 14-15
Table 14-10 UA PIpe Control Construct 14-15
Table 14-11 UA Last Element Identification Construct 14-16
Table 14-12 UA Test Condition Construct 14-16
Table 14-13 UA Branch Constructs 14-16
Table 14-14 UL VRF Operand Select Construct 14-21
Table 14-15 UL Counter Control Construct 14-22
Table 14-16 UL Pipe Control Construct 14-22
Table 14-17 UL Last Element Identification Construct 14-22
Table 14-18 UL Test Condition Construct 14-23
Table 14-19 UL Branch Constructs 14-23
Table 14-20 UL Source Control Construct 14-24
Table 14-21 UL Destination Constructs 14-24

1 Introduction

The C38xx Vector Processor (NVP) participates in the execution of all vector instructions executed by a processor. It contains the vector registers, vector mask register, and vector function units for execution of vector instructions. The NVP receives instructions across an instruction bus. It receives data from the scalar processor and memory system across a data bus $SP_VP.DATA_{63..0}$ and returns data to the scalar processor and memory across a data bus $VP_SP.DATA_{63..0}$.

There are three pipes within the Vector Processor: the "Add" pipe, the "Multiply" pipe and the "Load" pipe. The Add pipe can perform all integer arithmetic/compare, floating point arithmetic/compare, divide and square root, logical and vector edit operations with the exception of multiplies. The Multiply pipe can perform all integer arithmetic/compare, floating point arithmetic/compare, multiplies, logical and vector edit operations with the exception of divides and square roots. The terms "Add" and "Multiply" are really misnomers carried over from C2 days. The two pipes are identical except that divide/sqrt can only be done on the "Add" pipe, and multiplies can only be done on the "Multiply" pipe. The Load pipe performs all vector/VM/VS load/store operations.

1.1 Overview

A block diagram of the C38xx Vector Processor is shown in Figure 2-1 on page 2-2. The scalar processor dispatches instructions to the Vector Dispatch (VD) unit of the VP. The VD unit selects the pipe to execute the instruction, determines whether the vector register file ports and other required resources are available, and determines whether chaining and/or accelerated execution may occur. The VD body also waits for a seed from the scalar unit if it is required. When all the required conditions are met, the VD body dispatches the instruction to the selected pipe controller (UA, UM, or UL). This dispatch includes an entry point micro-address, vector register and port selects, source and destination data sizes, execution rate, and the scalar seed. The pipe controller then may execute the instruction without performing any resource checks.

Each pipe has its own controller (UA/A_VM/ABD, UM/M_VM/MBD and UL/L_VM/LBD) which contains a microsequencer, a copy of the VM register (all three copies of the VM register are identical at all times), a control queue and a backdoor controller. Each pipe controller is responsible for reading vector elements from the Vector Register File (VRF) body, sending these elements through its operation pipe (memory for the Load pipe), and writing the results back into the vector register file.

The Add Function Pipe (AFP) and Multiply Function Pipe (MFP) bodies contain all of the function units and thus perform all the "processing" within the Vector Processor, and are controlled by the UA and UM controllers respectively. The Input Staging (IS/ISCTL) unit receives and queues all data from the scalar processor and memory system. This data comes on the $SP_VP.DATA_{63..0}$ bus. Control of the IS unit is shared between the VD and UL bodies. All three pipe controllers may send data to the scalar processor. Only the load pipe sends data to, and receives data from memory. The Output Staging Control (OSCTL) body handles the handshakes for these transfers, while the $VP_SP.DATA_{63..0}$ bus is used for the data transfer.

1.2 Pipe Nomenclature

In order to keep the many stages of pipelining in the VP organized, a convention for naming pipes

and pipeline levels was defined and strictly followed. Signals within a pipe are prefixed by "A", "M" or "L" for the add, multiply and load pipes respectively. This prefix is followed by a string indicating the pipeline level: "_", "1_", "2_", "3_", "Q_", and "BD_". The first three strings refer to 0, 1, 2, and 3 levels after the control store and dispatch registers. The "Q_" refers to the output of each pipes control queue, while "bd_" is one level after the queue level (the backdoor level). Thus "*A_ACTIVE*" is the active bit for the add pipe microsequencer, "*A3_ACTIVE*" is delayed three clocks, "*AQ_ACTIVE*" is at the exit of the control queue, and "*ABD_ACTIVE*" is in the backdoor registers.

1.3 Organization of this Specification

The functional specification is divided into sections that correspond to the major functional subsections of the NVP. Many of the descriptions will refer to the NVP system block diagram which is a D size drawing that should be attached to the end of this specification. References to the detailed inner workings of gate arrays will be found in the individual gate array specifications.

A high level block diagram of the C38xx Vector Processor (NVP) is shown in Figure 2-1 on page 2-2. This diagram shows the basic functional blocks of the NVP. It also gives a general idea of the flow of data and control on the NVP, and it shows the major functional sections of the NVP.The major functional sections to be described include:

- The vector dispatch (referred to as VD) function.
- The input staging (IS) function.
- The vector register files (VRFs).
- The function pipe micro-controllers (UA, UM, and UL) and NVMs.
- The control queues (NQ).
- The back door controller (BD).
- The function pipes.
- Output staging controller (OSCTL)
- The clock logic.
- The context switch controller.

1.4 Physical Characteristics

The NVP is packaged as a single 18" by 20" multiwire circuit board with four wiring layers. All board internal and processor interface signals are 100k ECL compatible. The NVP board goes only in the CPU cabinet.

The following sections describe the each of the units in more detail.

2 Vector Dispatch

The Vector Dispatch (VD) logic receives instructions from the instruction processor on the scalar unit via the IPCTL controller. Part of the IPCTL is implemented in discrete logic on VD schematic page 3, and part is internal to the NVD gate array. The interface signals between the NSP and the VD logic are listed in Table 2-1 on page 2-1. These instructions are dispatched to the function pipe microcontrollers on the vector processor when all conflicts for resources are resolved. Resources primarily are: VRF read ports, VRF write ports, VM write port, Multiply and Add function pipes, the scalar-to-vector interface, and the vector-to-scalar interface. For those instructions that are capable of executing on either the Add or Multiply pipe, the vector dispatch logic determines which function pipe an instruction will execute on.

Table 2-1 Vector Dispatch to NSP Interface Signals

Signal	Description
SP_VP.DISP_EP$_{10..0}$	Instruction type (Entry point to lookup table)
SP_VP.DISP_IREG$_{2..0}$	Vector register I number (Vi field of instruction)
SP_VP.DISP_JREG$_{2..0}$	Vector register J number (Vj field of instruction)
SP_VP.DISP_KREG$_{2..0}$	Vector register K number (Vk field of instruction)
SP_VP.DISP_PSW_HAZ	PSW hazard on this instruction
SP_VP.IEEE	IEEE mode for floating point
SP_VP.DISP_RDY	Dispatch ready from NSP
VP_SP.DISP_REQ_NEXT	NVP ready to accept dispatch on next cycle
VP_SP.PSW_HAZ	PSW hazard on active vector instruction
VP_SP.IDLE	NVP is idle

The Vector Dispatch function is implemented in a 15K GaAs gate array (NVD), 13 1Kx4 RAMs for the VD control store, and about two dozen other discrete parts. The NVD array does most of the hazard checking and resource allocation. The discrete parts contain the state machines, buffering, and other miscellaneous functions.

The IPCTL controller (implemented in discrete parts) performs all of the required handshaking with the instruction processor on the scalar processor (NSP) board and attempts to maintain a queue of instructions such that an instruction will always be available as soon as the vector dispatch logic is ready. The actual instruction queue is a 3 entry queue that is internal to the NVD array.

When the IPCTL indicates that an instruction is available and the VD is not currently dispatching an instruction, the VD begins a dispatch cycle. A state transition diagram for the VD state machine is shown in Figure 2-2 on page 2-3. Dispatch requires at least two clock cycles and is not pipelined. In those two cycles it reads the VD control store (VD WCS) to determine the required resources for the instruction, checks the availability of all of those resources and prepares to allocate the resources that the instruction will require. If the resources are not available, the dispatch will be held off until they become available.

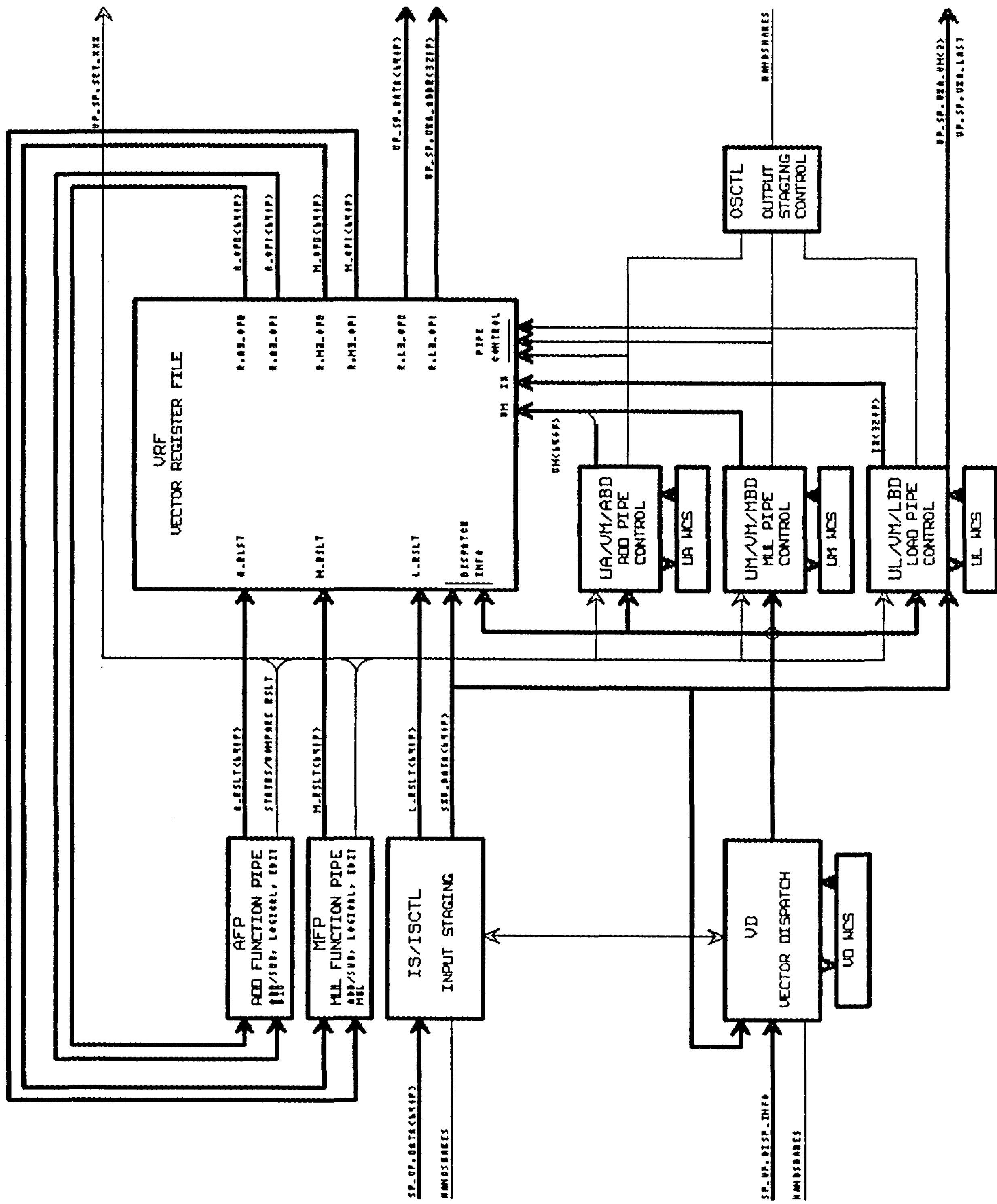
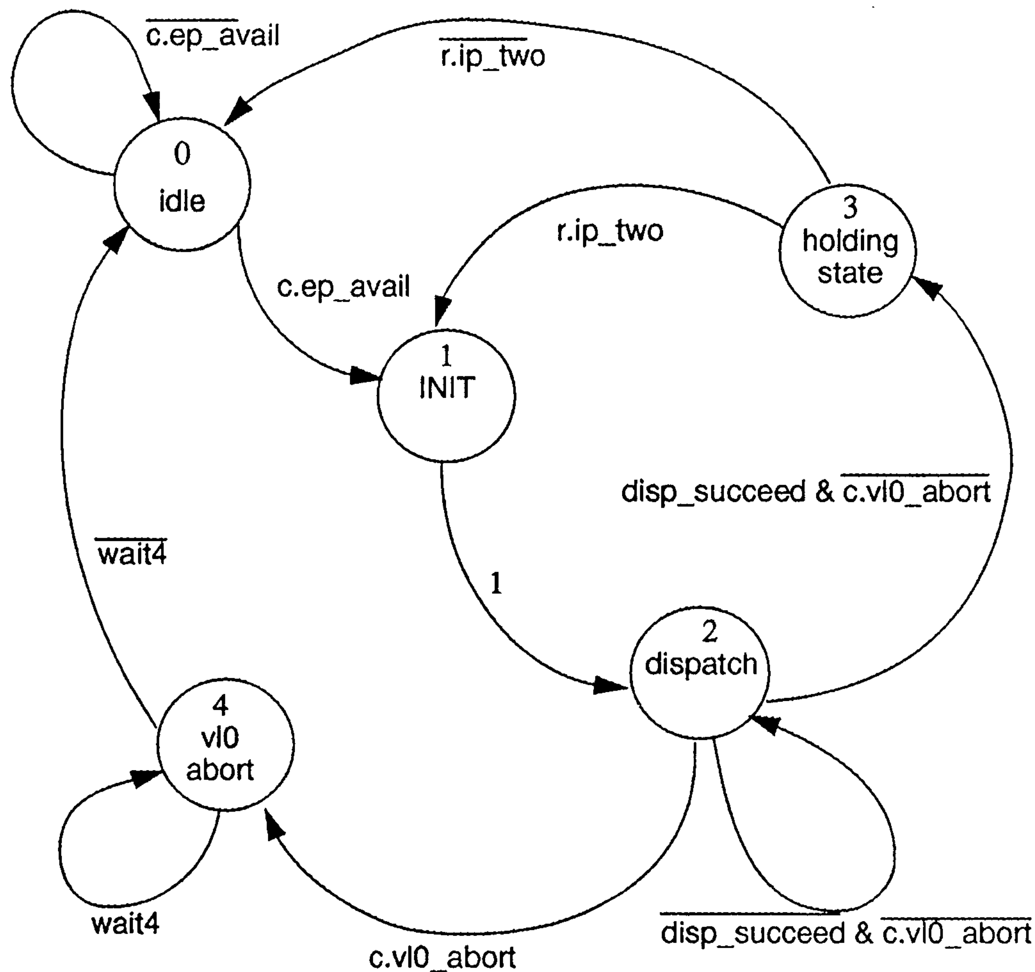


Figure 2-1 C38xx Vector

Figure 2-2 VD state machine



wait4 = $\overline{\text{sxv_ok}}$ & c.req_sxv_wait
c.ep_avail = r.ip_dval + (r.req_prev & sp_vp.disp_rdy)
NOTE: ext_ck_xtnd causes states 2, 3 and 4 to hold

Resource availability checks fall into 7 main categories:

1) Read port check/allocation
2) Write port check/allocation
3) Function pipe check
4) Chaining checks
5) Scalar port checks
6) Rate 2x and VM accelerate checks
7) Miscellaneous hazard checks

Read port check and allocation logic ascertains whether read ports on the requested registers in the VRF are available, and allocates them if they are. Since there are two read ports on each register pair, either one can be assigned to an instruction. If an instruction requests to read the

same register twice (e.g., *or.w v0,v0,v1* to copy register v0 to register v1), only one read port is assigned to the instruction and the VRF is commanded to use the same read port for both, leaving the other read port free for use of the register by another instruction.

Write port check and allocation logic determines whether the requested write port in the VRF is available, and allocates it if it is. The requested write port may already be allocated to any of the three pipes, and either the front-door or back-door of any of those pipes may be using the write port. If a front-door is using the write port, dispatch is blocked. If a back-door is using the write port, dispatch is blocked *unless* the instruction in dispatch has a pipe length greater than or equal to the instruction that is currently hogging the write port.

Function pipe check logic determines whether the requested function pipe is in use. If the pipe is in use it blocks dispatch unless the dispatching instruction has a pipe length greater than the remaining pipe length of the executing instruction.

Chaining check logic determines whether an executing instruction is writing to one of the registers (VRF or VM) that the dispatching instruction needs to read, and blocks dispatch until that executing instruction has written at least one result into that register. This approach assumes that all of the C38xx function pipes run in lock-step. That is, that either all of the function pipes advance one step or all of the function pipes advance zero steps on each clock.

Scalar port checks control access to the scalar-to-vector (SXV) and vector-to-scalar (VXS) interfaces. Only one instruction which requires the SXV interface is allowed to execute at any time. This alleviates scalar data ordering problems, since an instruction that uses SXV data knows that any SXV data that is available belongs to it. Only one instruction which requires the VXS/VXM (vector-to-memory) interface is allowed to execute at any time. This alleviates scalar and memory data ordering problems. Instructions which would violate the above conditions are delayed in dispatch until the resource becomes available. This function also interfaces with the Input Staging function (using *SXV_DVAL* and *VD_SXV_POP*) to get scalar data for those instructions that require them and to load the VL register.

Rate 2x and VM accelerate checks determine whether a dispatching instruction may cause problems as a result of consuming or producing data at an accelerated rate. Rate 2x and VM accelerated instructions may consume data and produce results at a rate that is faster than other instructions executing on the vector unit. These fast instructions can read data more rapidly than it is being produced by a normal instruction. In a chaining situation this could result in the fast instruction reading past the data that is being written by the normal instruction or it may begin consuming from vector elements that have not yet been written to. This is prohibited by this check. In this case a rate 2x instruction is prohibited from executing until the offending instruction completes; the under mask instruction would be forced to go at the normal rate rather than at the accelerated rate, but is dispatched immediately if all resources are available.

The other hazard is the reverse of the above. The fast instruction can produce results that overwrite elements that have yet to be read by a preceding instruction (e.g., *add.l v0,v1,v1* followed by *mul.w v2,v3,v0* where the *add.l* goes at the normal rate and the *mul.w* runs at rate 2x and could write into elements of v0 that the *add.l* has not had a chance to read.) This is prohibited by the check. The dispatch acts in the same way as the above condition.

Miscellaneous checks are a set of small checks that do not fall into one of the above categories. These checks are as follows:

- VM write port checking and allocation. Both the MUL and ADD pipes can write VM bits serially on C38xx. The LOAD pipe can parallel write the VM register. The VM must be allocated properly so that the order of writes is preserved.
- Simultaneous serial and parallel accesses to the VM register are prohibited.
- Edit instructions that do not produce or consume data at the single cycle rate must not interfere with instructions that do.
- The VAG check prohibits a new instruction which stores data to memory from dispatching if the NVP's memory request is full.

There is a great deal of complexity in the details of the vector dispatch unit. The best and most precise description is available in the nvd.isp behavioral model for the NVD gate array and in the vd_glue.isp behavioral model for the non-NVD parts of the VD function.

2.1 The VD State Machine

The state transition diagram for the VD state machine is shown in Figure 2-2 on page 2-3. The VD state machine is implemented primarily in PAL 5266 and a 100e142 register on VD page 3. The PAL implements most of the equations for the state machine and the register keeps the state. The equations that are not fully formed in PAL 5266 are for *DISP_SUCCEED*, and *C.VL0_ABORT.* The *DISP_SUCCEED* equation is partially formed into *PART_SUCCEED_NOK* and *PART_SUCCEED_OK* by PAL 5250 on VD page 1. They are composed of the terms of the *DISP_SUCCEED* equation that do not include *NVD_DISP_OK* and the terms that do include *NVD_DISP_OK*, respectively. This partial formation is required do to the shortage of inputs on PAL 5266, and the fact that there is insufficient time to run *NVD_DISP_OK* through two PALs before setting up to the 100e142 register.

The VD state machine starts in state 0, the idle state, and waits for an instruction to come from the scalar processor. This is indicated by the assertion of *R.IP_DVAL* from the IPCTL state machine or the assertion of both *R.REQ_PREV* and *SP_VP.DISP_RDY*. This sends the machine from state 0 to state 1. State 1 is a delay state which allows time for the dispatch RAM to be read. The machine proceeds unconditionally from state 1 to state 2. State 2 is the dispatch state. The state machine waits in state 2 until all hazards are cleared (indicated by the signal *DISP_SUCCEED)* and the instruction can be dispatched, or until the *C.VL0_ABORT* signal is asserted which indicates that the instruction is to be dumped on the floor instead of dispatched. (Some instructions, such as *add v1,v2,v3*, are not dispatched if VL=0 since there is no work to be done by the vector unit). If *EXT_CK_XTND* is asserted during state 2, the state machine will remain in state 2 regardless of the condition of other signals.

If the machine exits state 2 due to a *DISP_SUCCEED*, it proceeds to state 3, which is a holding state that allows time for the handshaking between the VD state machine and the ISCTL state machine for the SXV bus. This avoids a condition where two instructions in a row require SXV data and there is only one piece of SXV data currently available and the first instruction takes it and the second instruction thinks that the SXV data is still available because the ISCTL state machine has not had time to de-assert *SXV_DVAL* and so the second instruction dispatches without its piece of SXV data. Whew. From state 3 the VD state machine proceeds without delay to state 1 if another instruction is available, or back to the idle state (state 0) if another instruction is not available. *EXT_CK_XTND* will force it to stay in state 3.

In the case where the VD state machine exited state 2 with a vl0_abort, it proceeds to state 4. In state 4 the state machine waits for the instruction's piece of SXV data, if it has one, and then

proceeds back to the idle state. It cannot go directly to state 1 for the same reason that state 3 exists. *EXT_CK_XTND* will force the machine to hold in state 4

2.2 Final Dispatch Logic

The final level of the vector dispatch function is implemented in PAL 5250 and the 100e142 register on VD page 1. The PAL performs the final combination of the signals required to dispatch an instruction. It generates the *DISP_SUCCEED* that indicates that an instruction is to be dispatched. It generates the combinatorial version of the *A_DISPATCH*, *M_DISPATCH* and *L_DISPATCH* signals which indicate a dispatch to the particular pipe. It generates the *PART_SUCCEED* signals which are used by the VD state machine. It generates *C.VD_DISPATCH*, which is a test point that indicates that an instruction has executed which runs only in the VD unit and not on one of the pipes (e.g., *mov.w s0,VL*). The register delays most of these signals. It is also used for a pair of control signals (*VD_SXV_POP* and *R.VL_EN*) from VD page 3.

2.3 SXV Data with Instructions

Some instructions, such as *add.s Vi,Sj,Vk* are scalar-vector instructions. They require that a piece of SXV data be provided for the instruction. The dispatch RAM fields *REQ_SXV*, *REQ_SXV_POP*, and *REQ_SXV_WAIT* are used for these instructions. *REQ_SXV* indicates that the instruction uses SXV data. *REQ_SXV_WAIT* indicates that the instruction should not be dispatched if *SXV_DVAL* is not asserted. *REQ_SXV_POP* indicates that VD should wait for *SXV_DVAL* before dispatching, and then assert *VD_SXV_POP* when the instruction dispatches.

The effect of this is that for scalar-vector instructions, if *REQ_SXV_POP* is asserted, then vector dispatch is what pops the SXV data.

2.4 The IPCTL State Machine and Instruction Queue

The IPCTL controller contains the controller that maintains the vector dispatch instruction queue. The instruction queue is a three entry queue of instructions that resides inside the NVD gate array. The instructions in this queue are the concatenation of the instruction fields from the NSP: $SP_VP.DISP_EP_{10..0}$, $SP_VP.DISP_IREG_{2..0}$, $SP_VP.DISP_JREG_{2..0}$, $SP_VP.DISP_KREG_{2..0}$, *SP_VP.DISP_PSW_HAZ* and *SP_VP.IEEE*. A diagram of the IPCTL instruction queue is shown in Figure 2-3 on page 2-7

The IPCTL controller works by maintaining a two-bit read pointer ($R.IP_RD_{1..0}$) and a two-bit write pointer ($R.IP_WR_{1..0}$). If a new instruction comes in from the scalar unit, it is written in the register pointed to by the write pointer and the write pointer is advanced. If an instruction is popped off of the queue by the vector dispatch state machine (*VD_POP*), the data comes from the location pointed to by the read pointer and the read pointer is advanced. The IPCTL queue is prevented from overflowing by keeping some extra state that indicates whether there are one, two (*R.IP_TWO*), or three (*R.IP_FULL*) entries.

The IPCTL state machine is implemented primarily in PAL 5201 and a 100e142 register on VD page 3. The instruction queue that is maintained by the IPCTL state machine is located inside the NVD gate array. It is a three entry queue. The entries are referred to as R0, R1 and R2. Writes to R0, R1 and R2 are enabled by *C.R0_EN*, *C.R1_EN* and *C.R2_EN*, respectively, which are direct decodes of $R.IP_WR_{1..0}$ through PAL 5265. Reading from the queue is done using $R.IP_RD_{1..0}$ which directly selects a mux inside the NVD array.

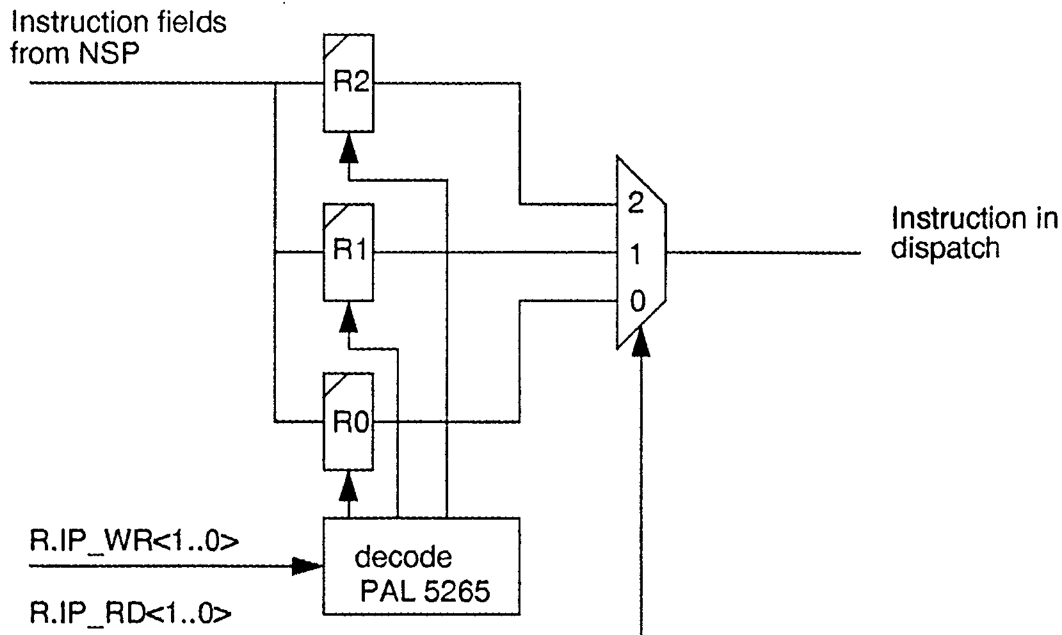


Figure 2-3 IPCTL instruction queue

The IPCTL controller also maintains a three entry queue of PSW hazard bits that is external to the NVD gate array. The PSW queue is implemented in PAL 5258 and a 100e142 register on VD page 3. It is managed in the same way as the instruction queue inside the NVD array and contains the PSW hazard bit that goes with the instruction that is in the respective entry in the queue that is inside the NVD array.

The handshaking with the NSP is accomplished through the signals *SP_VP.DISP_RDY* and *VP_SP.DISP_REQ_NEXT*. These handshakes follow the usual REQ_NEXT/RDY format used on the NVP. Figure 2-1 on page -8 shows the relationship between the handshakes and the instruction information. *SP_VP.DISP_RDY* indicates that the NSP has an instruction available on the current cycle. *VP_SP.DISP_REQ_NEXT* indicates that the IPCTL will accept an instruction on the next cycle if *SP_VP.DISP_RDY* is asserted. The only time a transfer takes place is on a cycle where *SP_VP.DISP_RDY* is asserted when *VP_SP.DISP_REQ_NEXT* was asserted on the previous cycle.

2.5 The VD Dispatch RAMs

These RAMs contain the vector dispatch lookup table (microcode) that the VD logic uses to determine the parameters and resources required for an instruction. A detailed description of the VD microcode formats is given in Appendix TBD.

The VD lookup table is composed of fourteen 100474 ECL 1k x 4 SRAMs (see VD page 2). The address bus $IP_VP_EP_{9..0}$ has two true copies and two inverted copies for timing purposes. The duplicate copies of the address bus are driven by 100e112 buffers. The data from the RAMs

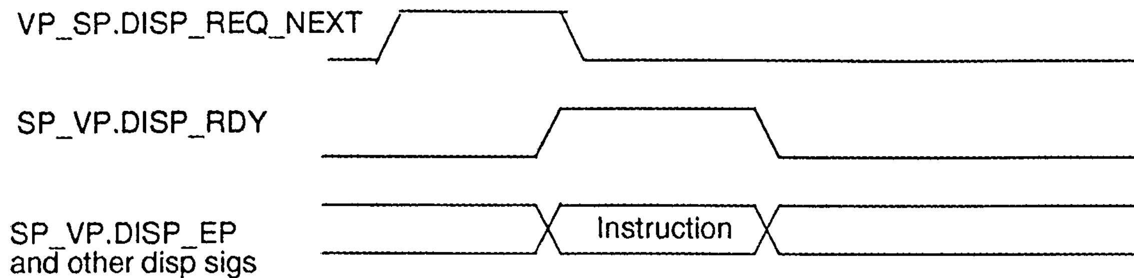


Figure 2-4 Scalar to Vector Dispatch interface

DISP_RAM$_{53..0}$ is driven into the NVD gate array to be registered and part of it is captured in a discrete 100e142 register on VD page 4. The registers that are driven by *DISP_RAM*$_{53..0}$ internal and external to the NVD array are the micro-instruction registers for vector dispatch.

Writing of the dispatch RAMs during system initialization is accomplished by scanning the write data onto the data inputs of the RAMs and activating the *VD_WCS_CS** and *VD_WCS_WE** signals via the scan ring. The data inputs of the RAMs are driven by *DISP_RAM*$_{26..0}$ and *R.LOAD_DAT*$_{26..0}$. The *DISP_RAM*$_{26..0}$ bus is driven backwards by the NVD gate array during this operation and supplies half of the write data. The other half is provided by *R.LOAD_DAT*$_{26..0}$, which comes from three 100e142 registers on VD page 4. These registers are used for no other purpose than to provide data to write the dispatch RAMs.

2.6 PSW Hazard Logic

The NVD array internally keeps track of PSW hazards on instructions that are executing on one of the function pipes. It uses this information to generate *NVD_PSW_HAZ*.

On page 3 is the logic that determines whether there is a PSW hazard on an instruction that is waiting in the instruction queue, or on the instruction that is currently in dispatch. The signal *C.Q_PSW_HAZ* generated by PAL 5265 indicates that there is a PSW hazard bit set on an instruction in the instruction queue and that queue entry is valid. This signal is loaded into a 100e142 (which is part of the VD micro-instruction register) as *R.Q_PSW_HAZ* when the VD state machine begins a dispatch. This signal is then run through PAL 5251 to generate *C.RAMREG_PSW_HAZ* which indicates that the instruction in dispatch has a PSW hazard. *R.RAMREG_PSW_HAZ* is a registered version of this. The signals *NVD_PSW_HAZ*, *C.Q_PSW_HAZ*, *C.RAM_REG_PSW_HAZ* and *R.RAM_REG_PSW_HAZ* are combined by an OR gate to generate *C.VP_PSW_HAZ* which indicates that an instruction somewhere on the NVP has a PSW hazard. This signal is registered to generate *VP_SP.PSW_HAZ* for the NSP.

2.7 Other VD Logic

The Vector Length (VL) register for the NVP is on Page 3. It is implemented in a 100e142 register. The loading of this register is caused by *C.VL_EN*. The dispatch logic needs to know when VL=0 for certain instructions. The signal *C.VL0** (active low) is created for this purpose as follows: The

$VL_{7..0}$ bus is OR'ed by a pair of OR gates and a wire-OR to generate *C.VL0** which is an indication that VL != 0.

3 Input Staging

Input Staging receives and queues data from the scalar processor and memory. A block diagram for input staging is shown in Figure 3-2 on page 3-3. In the schematics it is implemented in the blocks IS72 and ISCTL. Memory (MXV) data and scalar (SXV) data are queued separately. The SXV queue is three levels deep, which provides enough overrun after an NVP clock extend to absorb incoming data until the *VP_SP.SXV_REQ_NEXT* signal can be negated (plus one more transfer). The MXV queue is 16 levels deep, which avoids a gridlock condition in the memory system that will not be described here. Parity is checked on all entries of the SXV queue, and a parity error will cause all queue entries to be held. Parity is checked only on the output of the MXV queue and will have the same result as a parity error on the SXV queue. A list of the interface signals for the scalar to vector data bus is given in Table 3-1 on page 3-1.

Table 3-1 Scalar-to-Vector Interface Signals

SP_VP.DATA$_{63..0}$	Scalar and memory data (SXV and MXV data)
SP_VP.PAR$_{7..0}$	Parity on the above
SP_VP.MXV_RDY	Handshake for MXV data
SP_VP.SXV_RDY	Handshake for SXV data
VP_SP.MXV_REQ_NEXT	Handshake for MXV data
VP_SP.SXV_REQ_NEXT	Handshake for SXV data

The handshakes for SXV and MXV transfers follow the REQ_NEXT/RDY format that is commonly used on the C38xx machine. A diagram of the handshakes is shown in Figure 3-1 on page 3-1. A successful transfer requires that the REQ_NEXT signal be active on the cycle before the RDY is active. The data is valid on the cycle that the RDY is asserted. Since *VP_SP.DATA*$_{63..0}$ carries both SXV and MXV data from the NSP, it is not possible for *SP_VP.MXV_RDY* and *SP_VP.SXV_RDY* to be asserted at the same time.

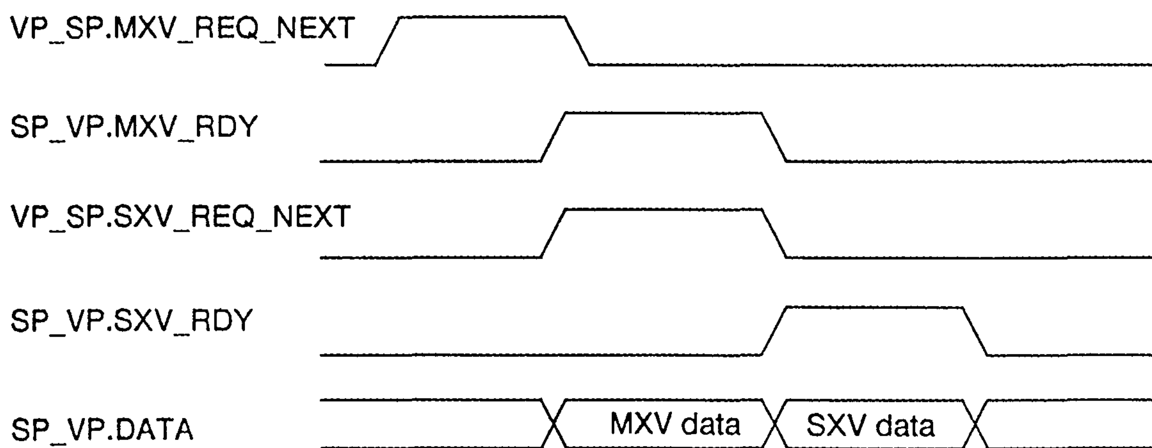


Figure 3-1 Scalar to Vector bus interface timing

The input staging data path is bit sliced into two 15K NIS gate arrays (IS0 and IS1), each array implementing 36 bits (four bytes plus parity). IS0 handles bits 16 through 31 and bits 48 through 63 or each 64-bit bus. IS1 handles bits 0 through 15 and bits 32 through 47. MXV control is internal to the NIS gate arrays. SXV control and the context mux are external. The SXV control is in the body called ISCTL. The context mux is in the body called IS_MUX.

3.1 The NIS gate arrays

Data from the scalar processor and memory system come to the NVP over the *SP_VP.DATA*$_{63..0}$ data bus (with parity *SP_VP.PAR*$_{7..0}$). The handshake signals *SP_VP.SXV_RDY* and *SP_VP.MXV_RDY* indicate to the NVP whether the data on the *SP_VP.DATA*$_{63..0}$bus is from the scalar processor or the memory system, respectively. Based on these control signals, the data is directed to either the SXV queue or the MXV queue for the NVP to pull out as needed.

Scalar data is driven onto the SXV bus (*SXV_DAT*$_{63..0}$). When data is available it may be popped by either the vector dispatch unit (*VD_SXV_POP*), the load microsequencer or the load backdoor controller (both using *LQ_SXV_POP* as generated in the BD controller). Memory data or scalar data may be driven onto the load pipe result bus (*LQ_RSLT_DAT*$_{63..0}$). Load pipe result data may only be popped by the load backdoor controller. The *LQ_RSLT_DAT*$_{63..0}$ bus goes only to the VRFs.

The *SCAN_OUT*$_{31..0}$ bus is driven by the IS_MUX. It contains context data to be sent to the NSP during a fault. See Section 3.2 on page 3-2.

SXV_RD$_{1..0}$ is a mux select for the three entry SXV queue. It is driven by the ISCTL controller. See Section 3.4 on page 3-5. The ISCTL controller also drives *SXV_WR*$_{1..0}$. This is a pointer that tells the NIS array which entry of the SXV queue should be loaded.

The *LQ_RSLT_SEL*$_{1..0}$ signal comes from the BD controller. It selects the type of data that will be placed on the *LQ_RSLT_DAT*$_{63..0}$ bus. The following list shows the relationship between the two.

LQ_RSLT_SEL$_{1..0}$	*LQ_RSLT_DAT*$_{63..0}$
0	MXV data
1	Word duplicated MXV data (for loads of VM)
2	SXV data
3	Word duplicated SXV data (for loads of VM)

The *CNTX_MODE* and *CNTX_RESTORE* signals select the type of data to come out of the *SXV_DAT*$_{63..0}$ bus. These signals are asserted at different times during context saves (faults) and restores (rtnc). When *CNTX_MODE* is asserted the NIS arrays put context data onto the *SXV_DAT*$_{63..0}$ bus. If *CNTX_RESTORE* is also asserted then the data is context restore data from the CX_IN register which itself was loaded from the *SP_VP.DATA*$_{63..0}$ bus. If *CNTX_RESTORE* is not asserted, then the data is from the CX_OUT register which is context save data from the IS_MUX, and addresses from the context address counter in the NIS array.

3.2 The IS_MUX

The IS_MUX controls the connectivity of the scan ring on the NVP. It receives and distributes context restore data from the NSP via the NIS arrays. It receives context save (fault) data from the registers on the NVP and sends them to the NIS arrays to be passed on to the NSP via the NVRF gate arrays. The IS_MUX provides three modes of connection as listed in Table 3-2 on

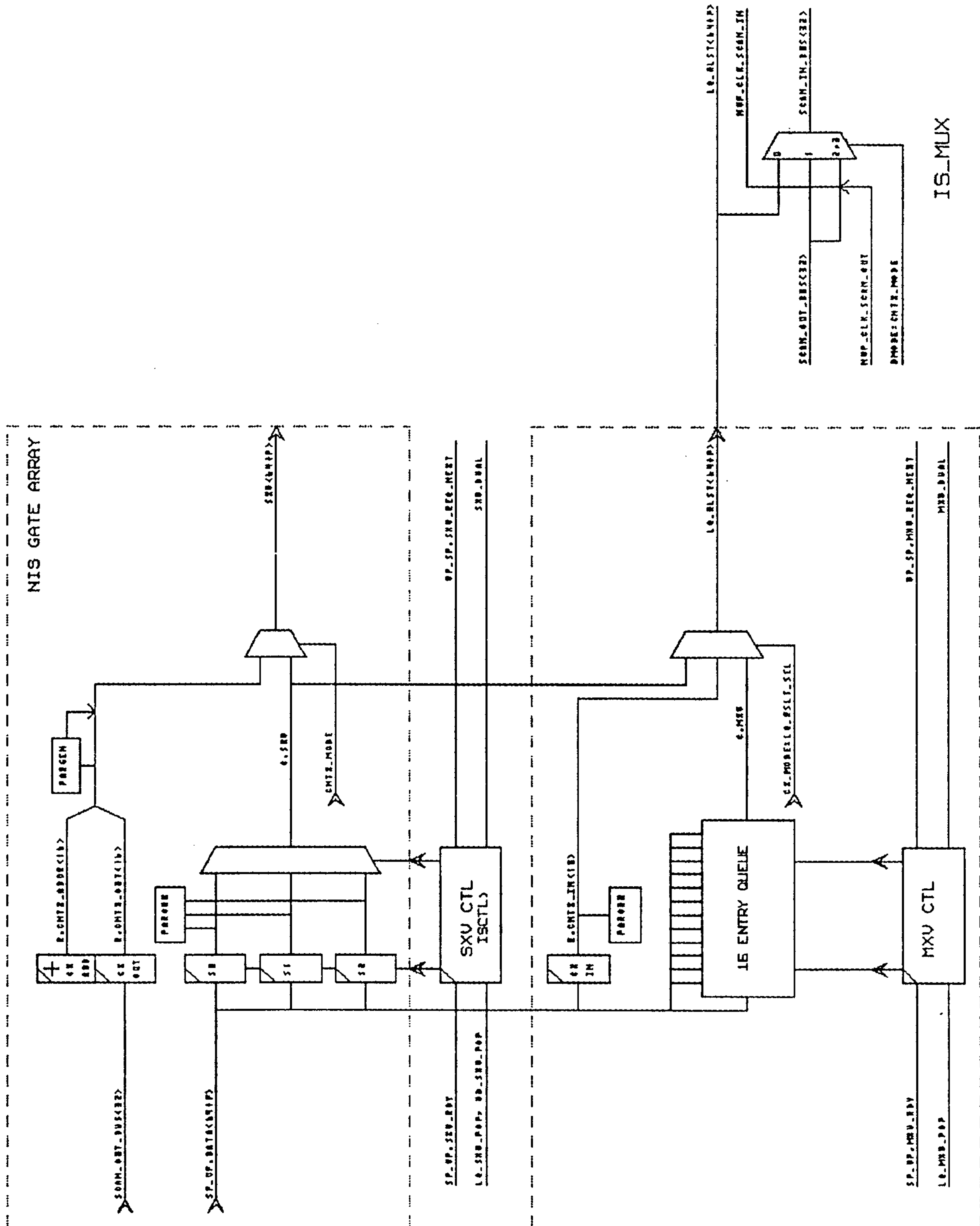


Figure 3-2 Input Staging

page 3-4.

Table 3-2 IS_MUX modes

Mode	Scan connectivity
recirculate	Each gate array scan_out connected to its own scan_in. (For faults)
rtnc	Gate array scan_in's driven by $SXV_DAT_{31..0}$. (For rtnc)
dmode	Entire scan chain connected in a single ring. (For diagnostic scan)

The **recirculate** mode occurs when *DMODE*=0 and *CNTX_SAVE*=1. This is the mode that is used during a fault. Each gate array scan_out is connected to its own scan_in so that when the fault is finished the state of the gate array is the same as it was before the fault. The scan_out data is also sent to memory via the NIS and NVRF gate arrays.

The **rtnc** mode occurs when *DMODE*=0 and *CNTX_SAVE*=0. This mode is used during a context restore (rtnc). The scan_in of each gate array, and the scan input of the discrete register portion of the scan ring, are driven by $SXV_DAT_{31..0}$ from the NIS gate arrays. The NIS gate arrays receive the returning context data from the NSP via the NIS arrays. See Section 3.3 on page 3-4 for further information.

The **dmode** occurs when *DMODE*=1. This mode is used during diagnostic scan and system initialization. The scan ring of the entire board is connected into a single long ring driven by *XC_VP.SCAN_IN* and ending in *VP_XC.SCAN_OUT*. For example, *NQ_SCAN_OUT* connects to *NVD_SCAN_IN*.

The IS_MUX is implemented in 100e171 4:1 multiplexers. The signals *DMODE* and *CNTX_SAVE* provide the select signals for the mux's as described above. The data inputs to the mux's are the scan outputs of all of the gate arrays and of the discrete ring (ext_ring) which are combined into the signal $SCAN_OUT_BUS_{31..0}$ and $SXV_DAT_{31..0}$. The $SCAN_OUT_BUS_{31..0}$ is context save data that goes to the NIS gate arrays, out the *LQ_RSLT_DAT* bus, into the NVRF arrays, and on to the NSP on the *VP_SP.DATA* bus. The data output from the mux's is the $SCAN_IN_BUS_{31..0}$ which is the scan inputs of all of the gate arrays and of the discrete ring. This path is used to send data back to all of the registers on the NVP.

3.3 Input Staging during Faults and RTNC's

The input staging logic has a special function during faults. It is the section of the NVP where the context data is accumulated and sent to the NSP during a fault, and where context data is sent back to its source during a return from fault.

Special registers and data paths are used only during faults and return from faults. During faults, the CX_OUT register stages context data collected from the gate arrays and external registers. A counter generates an index stream (0+4n where "n" is the transfer number) that accompanies the data. These address/data pairs are driven onto the $LQ_RSLT_DAT_{63..0}$ bus to the VRF gate arrays, where they are registered again and sent to memory as an STVI stream on $VP_SP.DATA_{31..0}$ and $VP_SP.VXA_ADDR_{31..0}$.

During return from faults (rtnc), the CX_IN register loads data from the $SP_VP.DATA_{63..0}$ bus which is then driven onto the $SXV_DAT_{31..0}$ bus, through the IS_MUX, and back to the gate arrays and external registers. The context mux allows scan ring data to be either recirculated (fault), loaded from $SXV_DAT_{31..0}$ (rtnc), or chained together into one ring (diagnostic mode).

3.4 The ISCTL controller

The ISCTL controller contains the controller for the SXV (scalar-to-vector) transfer queue. The controller works by maintaining a two-bit read pointer (*$SXV_RD_{1..0}$*) and a two-bit write pointer (*$SXV_WR_{1..0}$*). If new SXV data comes in from the scalar unit, it is written in the register pointed to by the write pointer and the write pointer is advanced. If SXV data is read by the load pipe (*LQ_SXV_POP*) or vector dispatch (*VD_SXV_POP*), the data comes from the location pointed to by the read pointer and the read pointer is advanced. The SXV queue is prevented from overflowing by keeping some extra state that indicates whether there are one, two (*SXV_TWO*), or three (*SXV_FULL*) entries.

The ISCTL state machine is implemented in PAL 5201 and a pair of 100e142 registers on ISCTL page 1. The SXV queue that is maintained by the ISCTL state machine is located inside the NIS gate arrays (IS0 and IS1). It is a three entry queue. Writing to the queue entries in the NIS arrays is enabled by the *$SXV_WR_{1..0}$* pointer which selects the location to be written and the *SXV_FULL* signal which disables writing. Reading from the SXV queue is done using *$SXV_RD_{1..0}$* which directly selects a mux inside the NIS arrays.

The ISCTL state machine also receives *C_MXV_DVAL* (which is the un-registered version of *MXV_DVAL*) from the IS0 NIS gate array, and registers it to make multiple copies of *MXV_DVAL*.

4 Vector Register File

The Vector Register File (VRF) is composed of eight NVRF gate arrays. The NVRF's together contain the eight 128 by 64-bit (plus parity) vector registers: v0 through v7. They also contain a read crossbar that allows any of the eight vector registers to be directed to either of the operand buses for any of the three function pipes. They contain a write crossbar which allows the result bus from any of the three function pipes to write into any of the eight vector registers. They contain registers which store scalar seeds for vector/scalar operations (e.g., *mul.w v0,s1,v4*). They contain operand multiplexers which allow vector register data, scalar data, identity data, or result data to be sent out on the operand buses of the function pipes.

A block diagram for the Vector Register File is shown in Figure 4-1 on page 4-2. The numbers inside the registers indicate the pipeline level to which the register corresponds. The VRF contains the memories for the vector registers, along with the data selection and staging required to allow the three microcontrollers to perform their operations as independently as possible. There are four 256 location by 72 bit (10 bit per gate array) vector RAM banks: V0/V4, V1/V5, V2/V6, and V3/V7. Each bank is divided into two 256x36 bit (256x5 bit per gate array) halves (macros) which are separately addressed, as shown in Figure 4-2 on page 4-3. The upper and lower words of each even vector element are swapped between the two halves. This allows pairs of byte/halfword/word vector elements to be read from the bank each clock, thus providing the bandwidth to run "single 2X double". Even longword elements are word swapped on writing and reading. Each RAM macro has internal registers on the address, write enable, read data and write data lines, and is triple cycled to allow two reads and a write each system clock.

There are three sets of dispatch registers within the VRF, one for each pipe. When an instruction is dispatched , the "frontdoor" dispatch registers in the VRF for that pipe are loaded with selects and controls for the instruction. These dispatch registers route read addresses and controls from the microsequencer to the vector RAM banks via the read control crossbar. The dispatch registers are also piped two register levels to route read data from the vector RAM banks to the pipe data path via the read crossbar. These piped dispatch registers also control the swappers, operand selection muxes and data registers of the pipe.

A *XQ_PIPE_CTL* code causes the "backdoor" dispatch registers for a pipe to be loaded. These backdoor dispatch registers route pipe results and controls to the vector RAM banks via the write crossbar. Parity is checked on pipe results while they are being written into the RAM banks.

The VRF is bit sliced into eight 30K gate arrays (NVRF) with RAM, each array implementing 4 bits plus parity of the upper word and 4 bits plus parity of the lower word of every longword.

4.1 The NVRF arrays

The vector register file is implemented in eight NVRF gate arrays in the VRF72 section of the schematics. Each of the NVRF arrays handles 4 bits (plus parity) of the upper word and 4 bits of the lower word out of each longword. Thus the NVRF labeled VRF0 (on VRF72 page 4) handles bits 63..60 and 31..28 out of each longword. Figure 4-1 on page 4-2 shows the structure of the VRFs. The individual NVRF gate arrays are slices of the VRF with the same structure as the figure.

As to the parity bits for the operand buses: Each gate array stores 4 bits of each word plus a 5th bit for parity, that implies that the 8 NVRFs have 8 parity bits per word, where only 4 are necessary.

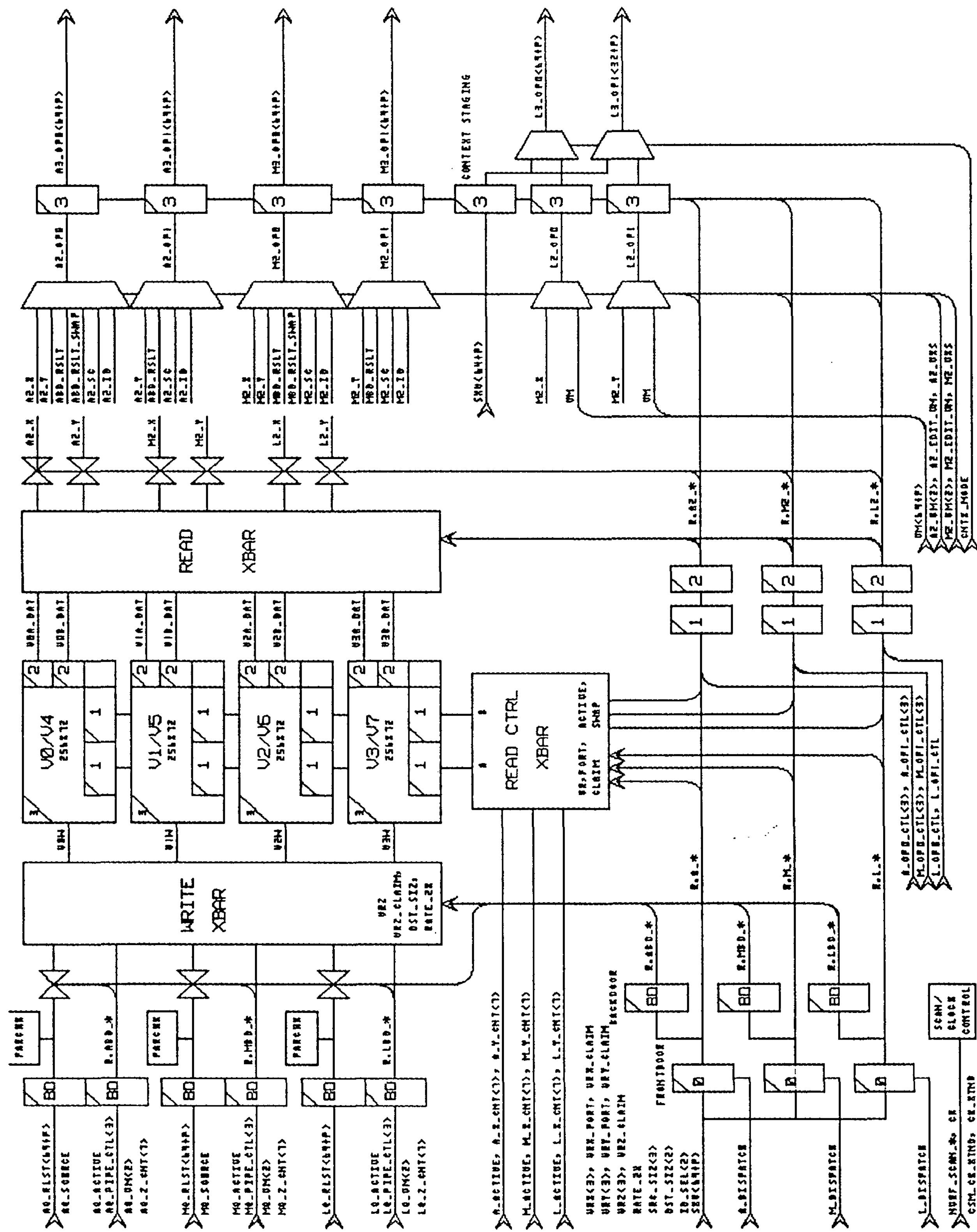


Figure 4-1 Vector Register Files

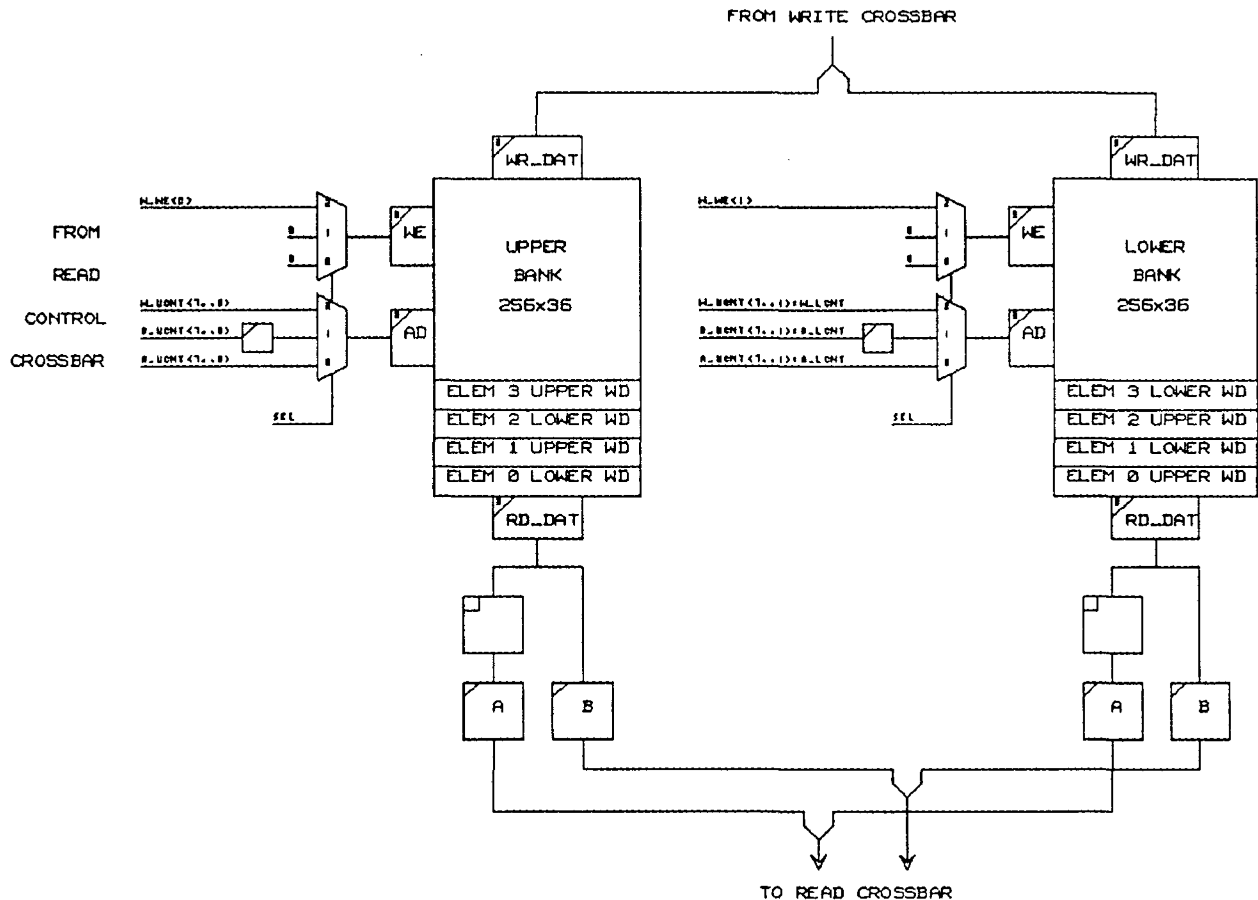


Figure 4-2 NVRF RAM Bank Organization

The way that this is resolved is for NVRFs 1,3,5 and 7 to generate byte parity ($X3_OPX_PAR_{7..0}$) for the operand buses ($X3_OPX_DAT_{63..0}$). The remaining four NVRFs 0,2,4 and 6 provide nibble parity for the 4 bits of data that they provide. The nibble parity (for the add and multiply buses) is collected and sent to 100e141 registers on VRF72 page 1. More information about the nibble parity is provided in Section 4.2 on page 4-5.

The signals *A_DISPATCH, M_DISPATCH* and *L_DISPATCH* come from Vector Dispatch (VD) and indicate a dispatch on the respective pipe and that the dispatch information is valid and should be loaded in. This dispatch information comes on $VRX_{2..0}$, *VRX_PORT, VRX_CLAIM,* $VRY_{2..0}$, *VRY_PORT, VRY_CLAIM,* $VRZ_{2..0}$, *VRZ_CLAIM, IEEE, RATE_2X,* $SRC_SIZ_{2..0}$, $DST_SIZ_{1..0}$, $ID_SEL_{2..0}$, and the appropriate bits of $SXV_DAT_{63..0}$. The dispatch information is used throughout execution of the instruction.

The $VM_DAT_{7..0}$ input to the NVRFs is driven by the NVM gate arrays. It may contain SXV data for a *mov Si,Sj,Vk* type of instruction, VM register data for a *st.x VM* type of instruction, or addresses for ordinary and under-mask stores. See the signal description in the signal appendix for $L_SRC_CTL_{2..0}$ for an exact description of what goes on the *VM_DAT* bus.

The *RSLT_PAR_ERR* input is driven by PALs 5200 on VRF72 page 3. It indicates that a parity

error has occurred on either $AQ_RSLT_DAT_{63..0}$, $MQ_RSLT_DAT_{63..0}$, or $LQ_RSLT_DAT_{63..0}$. The signal stops the result registers inside the NVRFs so that they can be scanned to determine what the data and parity was the produced the error.

The $AQ_RSLT_DAT_{7..0}$ and $MQ_RSLT_DAT_{7..0}$ inputs are the appropriate sub-ranges of the result buses from the add and multiply pipes, respectively. The $LQ_RSLT_DAT_{7..0}$ is the appropriate sub-range of the load pipe result data from the NIS gate arrays.

The $AQ_VM_{1..0}$, $MQ_VM_{1..0}$ and $LQ_VM_{1..0}$ signals are the returning mask bits that are used to enable or disable writing of result data to the vector register files. During a 64-bit operation, only VM<0> is used and applies to the entire longword on the result bus. During rate_2x operations (32-bits or smaller), VM<0> applies to the word that is on the upper half of the result bus which corresponds to an even vector address; VM<1> applies to the word that is on the lower half of the result bus which corresponds to an odd vector address;

The *AQ_ACTIVE*, *MQ_ACTIVE* and *LQ_ACTIVE* signals indicate that the data and control signals at the back-door level of the pipe are valid. In particular, they indicate the validity of the $AQ_PIPE_CTL_{2..0}$, $MQ_PIPE_CTL_{2..0}$ and $LQ_PIPE_CTL_{2..0}$ control signals, which the NVRF will ignore if the active signal is not asserted.

The $AQ_Z_CNT_{6..0}$, $MQ_Z_CNT_{6..0}$ and $LQ_Z_CNT_{6..0}$ signals represent the VRF write address for data on the three result buses. The NQ gate array drives these buses. These addresses do not necessarily count consecutively due to the edit, move scalar-to-vector and under-mask accelerated instructions. During rate_2x operations these addresses always represent the even address of the pair of operands, the odd operand going to the Z_CNT address plus one.

The *A_ACTIVE*, *M_ACTIVE* and *L_ACTIVE* signals indicate that the front-door is running and that the front-door level control signals are valid. These signals include $A_OP0_CTL_{1..0}$, $A_OP1_CTL_{1..0}$, $M_OP0_CTL_{1..0}$, $M_OP1_CTL_{1..0}$, $L_OP0_CTL_{1..0}$ and $L_OP1_CTL_{1..0}$. These OP_CTL signals are more fully described in the signal description in Appendix A on page -1.

The $A_X_CNT_{6..0}$, $A_Y_CNT_{6..0}$, $M_X_CNT_{6..0}$, $M_Y_CNT_{6..0}$, $L_X_CNT_{6..0}$ and $L_Y_CNT_{6..0}$ signals represent the VRF read addresses at the front-door level for each of the three pipes.

The *A2_EDIT_VM* and *M2_EDIT_VM* signals are the VM bits for the edit instructions (*mask*, *merge*, *cprs*) and causes the NVRF arrays to swap the correct vector register data onto the operand bus.

The $A2_VM_{1..0}$ and $M2_VM_{1..0}$ bits come from the NQ array. They are used in reduction under-mask operations (e.g., *sum.t V0*). During such operations if the mask bit is not set for a particular element it cannot be included in a calculation, and so the NVRF arrays use these VM bits to insert an identity element in place of the vector register contents.

The NVRF arrays source a number of signals. The primary operand buses for the function pipes are $A3_OP0_DAT_{7..0}$, $A3_OP1_DAT_{7..0}$, $M3_OP0_DAT_{7..0}$ and $M3_OP1_DAT_{7..0}$ with parity bits $A3_OP0_PAR_{1..0}$, $A3_OP1_PAR_{1..0}$, $M3_OP0_PAR_{1..0}$ and $M3_OP1_PAR_{1..0}$. As described at the beginning of this section, the eight operand bits coming out of each NVRF are actually composed of four bits from each word of a longword. Thus, for example, the NVRF labeled VRF0 (on VRF72 page 4) has pins labeled $A3_OP0_DAT_{3..0}$ which drive the board signal $A3_OP0_DAT_{31..28}$ and pins labeled $A3_OP0_DAT_{7..4}$ which drive the board signal $A3_OP0_DAT_{63..60}$. Table 4-1 on page 4-5 shows the mapping between the operand 7..0 outputs of the NVRFs and the full 64-bit operand

bus for each of the 8 NVRF gate arrays.

Table 4-1 Mapping of NVRF OP<7..0> to System OP<63..0>

VRF	OP<7..4> Upper Word	OP<3..0> Lower Word	VRF72 Page
0	63..60	31..28	4
1	59..56	27..24	5
2	55..52	23..20	6
3	51..48	19..16	7
4	47..44	15..12	8
5	43..40	11..8	9
6	39..36	7..4	10
7	35..32	3..0	11

The mapping of the parity bits for the operand buses is similar to that for the data bits. One of the parity bits (0) is byte parity for one of the bytes in the upper word and the other (1) is byte parity for one of the bytes in the lower word. Table 4-2 on page 4-5 shows the mapping between the parity 1..0 outputs of the NVRFs and the full 8-bit parity bus for the four NVRFs that drive byte parity.

Table 4-2 Mapping of NVRF PAR<1..0> to System PAR<7..0>

VRF	PAR<1> Upper Word	PAR<0> Lower Word	VRF72 Page
1	4	0	5
3	5	1	7
5	6	2	9
7	7	3	11

The NVRFs drive the *L3_OP0_DAT*$_{7..0}$ and *L3_OP1_DAT*$_{3..0}$ with parity bits *L3_OP0_PAR*$_{1..0}$ and *L3_OP1_PAR*. As with the add and multiply operands, the eight operand bits coming out of each NVRF are actually composed of four bits from each word of a longword. Table 4-1 on page 4-5 shows the mapping between the operand 7..0 outputs of the NVRFs and the full 64-bit operand bus for each of the 8 NVRF gate arrays and Table 4-2 on page 4-5 shows the mapping for the parity bits as well. The *L3_OP0_DAT*$_{63..0}$ bus is **identical** to *VP_SP.DATA*$_{63..0}$. The *L3_OP1_DAT*$_{31..0}$ bus is **identical** to *VP_SP.VXA_ADDR*$_{31..0}$. Pardon the naming inconsistency.

The *RSLT_PART_SYND*$_{1..0}$ output is a pair of combined parity syndrome bits over the nibbles of all three result buses (*AQ_RSLT_DAT*, *MQ_RSLT_DAT* and *LQ_RSLT_DAT*) that each NVRF receives. *RSLT_PART_SYND<0>* applies to the upper nibble and *RSLT_PART_SYND<1>* to the lower. The isp model describes the exact formation of the syndrome bits from the result data and parity.

4.2 Nibble Parity (page 1)

Nibble parity on *A3_OP0_DAT*$_{7..0}$, *A3_OP1_DAT*$_{7..0}$, *M3_OP0_DAT*$_{7..0}$ and *M3_OP1_DAT*$_{7..0}$ is accumulated in 100e141 registers on VRF72 page 1. The nibble parity for these buses is *A3_OP0_NIBBLE_PAR*$_{7..0}$, *A3_OP1_NIBBLE_PAR*$_{7..0}$, *M3_OP0_NIBBLE_PAR*$_{7..0}$ and *M3_OP1_NIBBLE_PAR*$_{7..0}$, respectively. Table 4-2 on page 4-5 shows the mapping between the parity 1..0 outputs of the NVRFs and the full 8-bit nibble parity bus for the four NVRFs that drive nibble parity. The nibble parity is available only for the most significant nibble (bits 7..4) of a byte since it comes from NVRFs 0, 2, 4 and 6.

Table 4-3 Mapping of NVRF PAR<1..0> to NIBBLE_PAR<7..0>

VRF	PAR<1> Upper Word	PAR<0> Lower Word	VRF72 Page
0	4	0	4
2	5	1	6
4	6	2	8
6	7	3	10

The nibble parity coming from the NVRF arrays is at the third pipline level and is so labeled (e.g., A3, M3). The first rank of 100e141's is at the fourth pipeline level; the same level as the XBUS and YBUS registers inside the function units. The function units check parity at this level and register a parity error on the next clock (i.e., at the fifth pipeline level). The parity error signal exits the function unit and is combined in the NVP_CLK body into the signal *FU_ERROR.* If *FU_ERROR* is asserted, all of the 100e141's are put into hold mode. The second rank of 100e141's is at the fifth pipeline level and thus have the nibble parity for the data that caused a parity error in the XBUS or YBUS registers of the function units.

This nibble parity is useful for determining exactly which of the 8 NVRF arrays provided the data that caused the parity error. This is done as follows:

- The function unit with a parity error will have its PAR_ERR signal asserted. This allows determination of the function unit that has a parity error.
- The *R.XBUS_PAR_ERR* or *R.YBUS_PAR_ERR* in that function unit will be set. This allows determination of which longword caused the parity error.
- The data in *R.XBUS_DATA* or *R.YBUS_DATA* and the parity in *R.XBUS_PAR* or *R.YBUS_PAR* will show which byte caused the parity error. This narrows the possible source of the bad data down to two NVRF arrays.
- The nibble parity in the level 5 register on VRF72 page 1 will show if the most significant nibble of the bad byte caused the parity error. This narrows the possible source of the bad data down to an individual NVRF array.

4.3 Result Parity Error logic (Page 3)

This hardware combines the *RSLT_PART_SYND* signals from all eight NVRFs to create the *RSLT_PAR_ERR* signal which indicates a parity error on one of the result buses ($AQ_RSLT_DAT_{63..0}$, $MQ_RSLT_DAT_{63..0}$ or $LQ_RSLT_DAT_{63..0}$). The hardware is composed of a 100e166 comparator and two 5200 PALs. The comparator combines the *RSLT_PART_SYND* signals to produce *RSLT_SYND_ERR* which is the actual indication of a parity error. The *RSLT_SYND_ERR* signal goes into the PALs where it can be disabled by *DMODE*, *CNTX_MODE*, or enabled by *RSLT_PAR_ENABLE.* The PALs fan out copies of *RSLT_PAR_ERR* which return to the NVRFs to stop the result registers and to the NVP_CLK logic to assert *VP_XC.HARD_ERROR.*

4.4 Other VRF logic

Pages 2 and 3 of VRF72 contain signal buffering for timing reasons.

5 The Pipe Controllers (UA, UM and UL) and the NVMs

The three pipe controllers are each composed of an NVM gate array (referred to as A_VM, M_VM and L_VM for obvious reasons), nine 100474 ECL 5ns SRAMs, and a couple of discrete parts. The functionality of the three controllers is very similar, particularly for the UA and UM. To avoid needless repetition, only the add pipe controller will be described, except when there are differences between the pipes.

Each NVP pipe controller contains a microsequencer, a set of vector element counters, a copy of the VM register, a control queue, and a backdoor controller. A block diagram for a pipe controller is shown in Figure 5-1 on page 5-3. The control store RAM and miscellaneous logic shown in the dashed area is implemented in discrete parts. The control queues for all three microsequencers are implemented in the 15K gate array NQ. The remainder of the logic is implemented in a 30K gate array (NVM) for each microcontroller.

The microsequencer controls the X, Y, Z and VM counters, selects operands from the VRF, generates a code to control operations farther down the pipe, and determines several "last element" conditions. The X and Y counters create vector register read addresses that are sent to the NVRF arrays. The Z counter creates vector register write addresses that are queued in the NQ array in parallel with the data passing through the function pipes. The VM counter addresses the VM register and differs from the X and Y counts during edit operations.

5.1 The UA, UM and UL Logic Pages

The UA, UM and UL logic pages are nearly identical and this description will be of the UA, except where another pipe differs.

The UA logic page contains the logic for the add pipe microsequencer that is not contained in the A_VM gate array. There are nine 100474 ECL 1k x 4 5ns SRAMs which compose the UA control store. These RAMs drive the AF bus which is the micro-instruction word for the microsequencer. They are written from the $VM_DAT_{31..0}$ bus. This is driven by $VM_DAT_{63..32}$ in the case of UA, $VM_DAT_{31..0}$ in the case of UL, and $M_VM_DAT_{31..0}$ in the case of UM. The $VM_DAT_{63..0}$ bus is used for other purposes during normal operation, but is used to write the control stores during system initialization. The $M_VM_DAT_{31..0}$ bus is used only to write the control stores during system initialization.

A 100e142 register is used to contain the external portion of the UIR (micro-instruction register). This external register is needed for timing reasons. A 100e160 parity generator generates a parity syndrome bit (*A_UIR_PAR_EXT*) on the external portion of the UIR. This syndrome bit is used inside the NVM array in combination with parity from the internal portion of the UIR to detect parity errors across the entire UIR.

The 100e101 OR gate that is used to disable the clock of the 100e142 register when a parity error is detected on the UIR (*A_UIR_PAR_ERR*) or during a clock extend (*EXT_CK_XTND*).

5.2 The NVM gate arrays

The NVM gate arrays contain the core of the microsequencer, the vector element counters, a copy

of the VM register, and various other logic.

5.2.1 Dispatch Inputs

The dispatch signal *A_DISPATCH* is asserted by the VD logic to indicate that an instruction dispatch is taking place to the add pipe microsequencer. The NVM will load in the dispatch information which includes $EP_{5..0}$, *RATE_2X*, $VL_{7..0}$, $PL_{2..0}$, $OPCODE_{9..0}$, *IEEE*, *VM_POL* and *VM_FORCE*.

The $EP_{5..0}$ signal is the microcode entry point from the vector dispatch logic. When an instruction dispatch is received by an NVM array, it shifts the *EP* signal left by 3 and uses this as the first microcode address.

The other dispatch signals are fairly well described in the signal description Appendix A on page -1.

5.2.2 Micro-instruction Word Inputs

The signals $AF_CNT_CTL_{3..0}$ (or $LF_CNT_CTL_{2..0}$), $AF_PIPE_CTL_{2..0}$, $AF_LAST_{2..0}$ (or $LF_LAST_{3..0}$), $AF_BR_ADDR_{9..0}$, *AF_TEST_POL* and *AF_TEST_SEL* come from the UA control store RAMs on the UA page of the schematics. These signals are micro-instruction signals and control the basic operations of the microsequencer. $AF_CNT_CTL_{3..0}$ tells the NVM what type of address sequence to produce on the $A_X_CNT_{6..0}$, $A_Y_CNT_{6..0}$ and $A1_Z_CNT_{6..0}$ ports. $AF_PIPE_CTL_{2..0}$ tells the NVM what type of pipe operation is to be performed at each level of the pipe. It is passed through the NVM and out as $A1_PIPE_CTL_{2..0}$ into the control queue in the NQ array. $AF_LAST_{2..0}$ indicates to the NVM that the last microcode cycle is occurring so that the NVM can generate its "last" output signals. $AF_BR_ADDR_{9..0}$ is the branch address. *AF_TEST_POL* selects whether a conditional microcode branch condition is true or false. *AF_TEST_SEL* selects what type of condition is used for a conditional microcode branch.

The L_VM gate array uses the *SXV_DVAL* signal which indicates that there is valid data in the SXV queue in the NIS arrays. The load pipe is the only pipe that can take multiple pieces of SXV data. The add and load pipe can receive a single piece of SXV data, but the VD controller performs the handshakes with the ISCTL for them and provides the SXV data at dispatch time.

5.2.3 VM Write Port Inputs

The *SQ_ACTIVE*, $SQ_PIPE_CTL_{2..0}$, *SQ_RATE_2X*, $SQ_VM_{1..0}$, $SQ_Z_CNT_{7..0}$ and $SQ_CT_{1..0}$ inputs come from the STAT block of the schematics. These signals are used in the serial writing of the VM register. Since only one pipe can write to the VM register at a time, the STAT controller selects back-door signals from either the add pipe or multiply pipe and drives them onto the SQ signals. The *SQ_ACTIVE* signal indicates the validity of the rest of the SQ signals. $SQ_PIPE_CTL_{2..0}$ indicates the mode of writing to the VM register and causes a clear of VM above VL at the end of a VM write operation. *SQ_RATE_2X* indicates whether VM bits should be written as pairs or single bits. $SQ_VM_{1..0}$ are the control VM bits that indicate whether a write should take place or not. These bits are used during compare under-mask operations. $SQ_Z_CNT_{7..0}$ is the address of the element of the VM register to be written. $SQ_CT_{1..0}$ are the VM bits to be written back into the VM register during a VM write.

L_SRC_CTL
R.2_SRC_CTL<2>
LQ_RSLT_DAT<32+P>
L_ACTIVE
L_DST_CTL<3>
R.L1_*
VS
R.X_CNT<7>
R.VS<32>
MUL
R.2_IX<32>
VM_DAT<32+P>
VM Parallel RD/WR
Serial RD/WR
R.VM<128>
R.ACC_START<8>
R.RATE_2X
R.VM_POL
Accelerate Encoder
C.ACC_NEXT<8>
R.ACC_CNT<8>
SQ_ACTIVE
SQ_PIPE_CTL<3>
SQ_RATE_2X
SQ_VM<2>
SQ_Z_CNT<8>
SQ_CT<2>
R.BD_*
Bit Munge
C.VM<2>
C.EDIT_VM
R.OPCODE<10>
R.IEEE
R.ACTIVE
R.PIPE_CTL<3>
R.PL<3>
R.RATE_2X
R.Z_CNT<8>
C.LAST_ELEM
X1_VM<2>
X2_EDIT_VM
X2_OPCODE<10>
X2_IEEE
X1_ACTIVE
X1_PIPE_CTL<3>
X1_PL<3>
X1_RATE_2X
X1_Z_CNT<8>
X1_LAST_ELEM
R.VM_CNT<8>
C.VM_VALID<2>
R.VM_FORCE, R.VM_POL, R.RATE_2X
LQ_RSLT_DAT<0>
SXV_DVAL
(R.VL - R.VM_CNT) <= 2 or 4
R.RATE_2X
R.VL == 1 or 2
R.TEST_SEL<3>
R.TEST_POL
R.TEST
VM_FORCE
VM_POL
RATE_2x
PL<3>
OPCODE<10>
IEEE
R.*
R.ACC_CNT<8>
R.VL<8>
R.CNT_CTL<4>
LQ_RSLT_DAT<7..0>
C.ACC_NEXT<8>
R.RATE_2X
C.EDIT_VM
Count CTL
X_X_CNT<7>
X_Y_CNT<7>
R.Z_CNT<8>
R.VM_CNT<8>
R.ACC_START<8>
C.VM_VALID<2>
EP<6>
R.EP<6>
VL<8>
R.VL<8>
Dec
R.VL_CASE<4>
X_DISPATCH
XQ_LD_BD
Next Uaddr Gen
X_RDY
X_UADDR<10>
X_UPC<10>
X_ACTIVE
R.LD_BD_PEND
R.BR_INST<3>
R.BR_ADDR<10>
R.TEST
R.ACTIVE
R.UPC<10>
XF_LAST<4>
XF_PIPE_CTL<3>
XF_CNT_CTL<4>
XF_BR_INST<3>
XF_TEST_POL
XF_TEST_SEL<3>
XF_BR_ADDR<10>
R.*
R.LAST<4>
R.BR_INST<3>
R.ACTIVE
R.TEST
Last Decode
X_LAST_RD
X_LAST_WR
X_LAST_SXV
X_LAST_VXS
C.LAST_ELEM
X_UIR_PAR_EXT
Parity Check
X_UIR_PAR_ERR
UX_PAR_ERR

Figure 5-1 NVM Block Diagram

5.2.4 Parity Signals

The *A_UIR_PAR_EXT* input is a parity syndrome bit for the externally stored portion of the micro-instruction register on the UA schematic page. This signal is combined with the parity syndrome on the internal portion of the micro-instruction register to produce *A_UIR_PAR_ERR*.

5.2.5 Control Queue Outputs

The NVM arrays drive control signals that go to the control queue in the NQ array and also perform some control at the first few pipeline stages. These signals are *A1_ACTIVE*, $A1_PIPE_CTL_{2..0}$, $A1_VM_{1..0}$, *A2_EDIT_VM*, $A2_OPCODE_{9..0}$, $A1_Z_CNT_{7..0}$, *A2_IEEE*, $A1_PL_{2..0}$, *A1_RATE_2X* and *A1_LAST_ELEM*. The load pipe only drives *L1_ACTIVE*, $L1_PIPE_CTL_{2..0}$, $L1_Z_CNT_{7..0}$, $L1_VM_{1..0}$ and *L1_LAST_ELEM*. Most of these signals have the same function at this level that they had at the AF level or the dispatch level. $A1_VM_{1..0}$ was generated inside the NVM array by reading from the VM register (if the pipe is running an under-mask instruction) at the location that corresponds to the $A_X_CNT_{6..0}$ output. $A1_Z_CNT_{7..0}$ is the address that data will be written back to after going through the function pipe.

5.2.6 Front-door Level Outputs

$A_X_CNT_{6..0}$ and $A_Y_CNT_{6..0}$ are the vector element addresses that go to the NVRF array's vector register read port addresses. *A_LAST_RD*, *A_LAST_VXS*, *A_LAST_WR* and *L_LAST_SXV* are signals that indicate to the VD logic when the microsequencer is done with the VRF and VM read ports (*A_LAST_RD*), the VXS port (*A_LAST_VXS*), the VRF and VM write ports (*A_LAST_WR*), and the SXV port (*L_LAST_SXV*). The *A_ACTIVE*, *A_CLR_ACTIVE* and *A_RDY* signals indicate to the VD logic the state of the microsequencer itself so that another instruction may be dispatched to it when it is available.

5.2.7 Micro-address Outputs

$A_UADDR_{9..0}$ is the micro-address that the microsequencer is currently executing. This signal directly drives the address pins of the UA RAMs. $A_UPC_{9..0}$ is a registered version of $A_UADDR_{9..0}$ and is used solely for test purposes. It does not go to any other device (except terminators). It contains the previous micro-address at all times. If *A_UIR_PAR_ERR* is asserted, then $A_UPC_{9..0}$ is the address which contained the data that caused the parity error.

5.2.8 VM (and VS) Parallel Write Signals

For a parallel write to the VM register the data comes from the $LQ_RSLT_DAT_{31..0}$ bus, which allows 32 bits of the 128-bit VM register to be written at a time. Data to be written into the VS (vector stride) register also comes from the $LQ_RSLT_DAT_{31..0}$ bus. $L_DST_CTL_{2..0}$ selects which portion of the VM register is written, or if it is the VS register that is to be written according to Table 5-1 on page 5-5.

Table 5-1 Steering of LQ_RSLT_DAT<31..0> by L_DST_CTL<2..0>

CTL<2..0>	Write to
3	VS<31..0>
4	VM<127..96>
5	VM<95..64>
6	VM<63..32>
7	VM<31..0>

5.2.9 $L_SRC_CTL_{1..0}$ and $VM_DAT_{31..0}$

The $VM_DAT_{31..0}$ bus from the NVMs is combined to drive the board level $VM_DAT_{63..0}$, which is driven by the A_VM ($VM_DAT_{63..32}$) and the L_VM ($VM_DAT_{31..0}$). This bus goes to the NVRF arrays where they can either send it to the NSP or use it internally. The bus is used to store VM data and to provide address indexes during vector store and load under-mask operations.

During parallel reads of the VM register, the data is placed on $VM_DAT_{63..0}$. The portion of the VM register that is put on this bus is determined by $L_SRC_CTL_{1..0}$ and by an internal register (only accessible via scan) on the NVM array *R.VM_SEL*. During vector store and load under-mask operations the NVM gate arrays put out 32-bit address indexes onto the VM_DAT bus. The indexes are the product of VS (vector stride) times the X count. See the nvm.isp model for details.

Table 5-2 Selection for NVM pins VM_DAT<31..0>

SRC_CTL	R.VM_SEL	VM_DAT
0	X	Address indexes
1	0	VM<127..96>
1	1	VM<95..64>
2	0	VM<63..32>
2	1	VM<31..0>

Table 5-2 on page 5-5 shows how $L_SRC_CTL_{1..0}$ and *R.VM_SEL* are used to select the contents of the $VM_DAT_{31..0}$ bus from each gate array. In normal operation the A_VM gate array has *R.VM_SEL* = 0 and the L_VM gate array has *R.VM_SEL* = 1. This results in contiguous 64-bit segments of the VM register being stored on $VM_DAT_{63..0}$. It also implies that when indexes are on $VM_DAT_{63..0}$ that 32-bits are replicated on the upper and lower portions of the bus.

5.2.10 Misc. NVM signals

The *UA_WCS_WE** signal is used during system initialization to write the control store RAMs. In the NVM arrays it forces the NVM internal register $R.UPC_{9..0}$ onto $A_UADDR_{9..0}$ which is the address bus of the RAMs.

A_UIR_PAR_ERR indicates that the UIR (including both internal and external portions) has a parity error on this cycle. It causes the address and UIR registers to be held. *UA_PAR_ERR* is a registered version of *A_UIR_PAR_ERR* that is used to generate the NVP board level hard error signal *VP_XC.HARD_ERROR.*

6 The Control Queues (NQ)

The control queues for the three pipe controllers reside in a single NQ (15K GaAs) gate array. A block diagram of the NQ gate array is shown in Figure 6-1 on page 6-3. Virtually all of the inputs to the NQ array come from level 1 outputs of the NVM gate arrays in the three pipe controllers. The NQ receives this control information, delays it, and puts out control information to the function pipes, the VRFs, and the BD controller. For the add and multiply pipes, the control information is inserted into the queue at a point determined by the pipe length (e.g., $A1_PL_{2..0}$) and is read out at the head of the queue. For the load pipe, control information is stacked in the queue until data returns from memory and is read out at the head of the queue.

The purpose of the control queues is to provide a mechanism on the NVP that allows control information to be generated by the front-door controller and to be staged in parallel with the data. It is not necessary for the front-door controller to be running after it starts the last operation, since all of the control information necessary at the back-door level is in the control queue. It is not necessary to start the back-door controller until data actually arrives at the back door level. The control and data should flow in parallel through the various pipeline levels.

The load pipe queue is somewhat different. While the add and multiply queues have their length fixed for each instruction, the load pipe queue is a variable length queue. Entries are made into the queue for each memory request that the load pipe makes. The entries are de-queued whenever a piece of memory data returns to the NVP.

6.1 Queue inputs

The control queue receives pipe level 1 inputs from the A_VM, M_VM and L_VM gate arrays on the UA, UM and UL controllers. These inputs have prefixes A1, M1 or L1, respective to their pipes. The UL controller sends a level 0 active signal *L_ACTIVE*, while the other pipes send level 1 active signals (*A1_ACTIVE, M1_ACTIVE*).

6.2 Level 2 outputs

These are the outputs prefixed by A2, M2 or L2. The level 2 outputs are equivalent to the queue inputs, but have been registered for one clock. The level 2 control signals are used by sections of the NVP that are receiving level 2 data. Included are the OSCTL controller, the NVRFs, and the AFP and MFP function pipes.

100e112 OR gates are used to provide buffering for the $A2_VM_{1..0}$ and $M2_VM_{1..0}$ signals since they have to drive all eight NVRFs.

6.3 Level 3 outputs

These outputs are the load pipe signals $L3_VM_{1..0}$ and *L3_LAST_ELEM.* These signals go to the OSCTL controller. They are used during load and store operations. $L3_VM_{1..0}$ are the VM bits for load and store data which are sent to the NSP and are used by the address generator there. *L3_VM<0>* is for the MSW and *L3_VM<1>* is for the LSW. They indicate whether the word is valid and should be stored or loaded. *L3_LAST_ELEM* indicates the last piece of data and/or VM bits of a load pipe operation.

6.4 Back-door outputs for Add and Multiply pipes

These are the outputs prefixed AQ and MQ. The back-door level outputs from the control queue go to all of the places where back-door level data goes, including the NVRFs and the NVMs; and to where back-door level control is needed, including the BD controller and the VD logic.

6.5 Back-door outputs for the Load pipe

These outputs are prefixed by LN. The LN outputs are one stage shy of actually being back-door signals. The LN signals a registered in the BD controller where they become LQ signals which are actually at the back-door level. The exception to this is *LN_MEM_FULL* which goes to the NVD gate array for use in a dispatch check.

The load pipe queue is loaded on each cycle where *L2_ACTIVE* was asserted on the previous cycle. Data is removed from the queue when *LN_POP* is asserted. The queue gets deeper when the queue is loaded and *LN_POP* is not asserted.

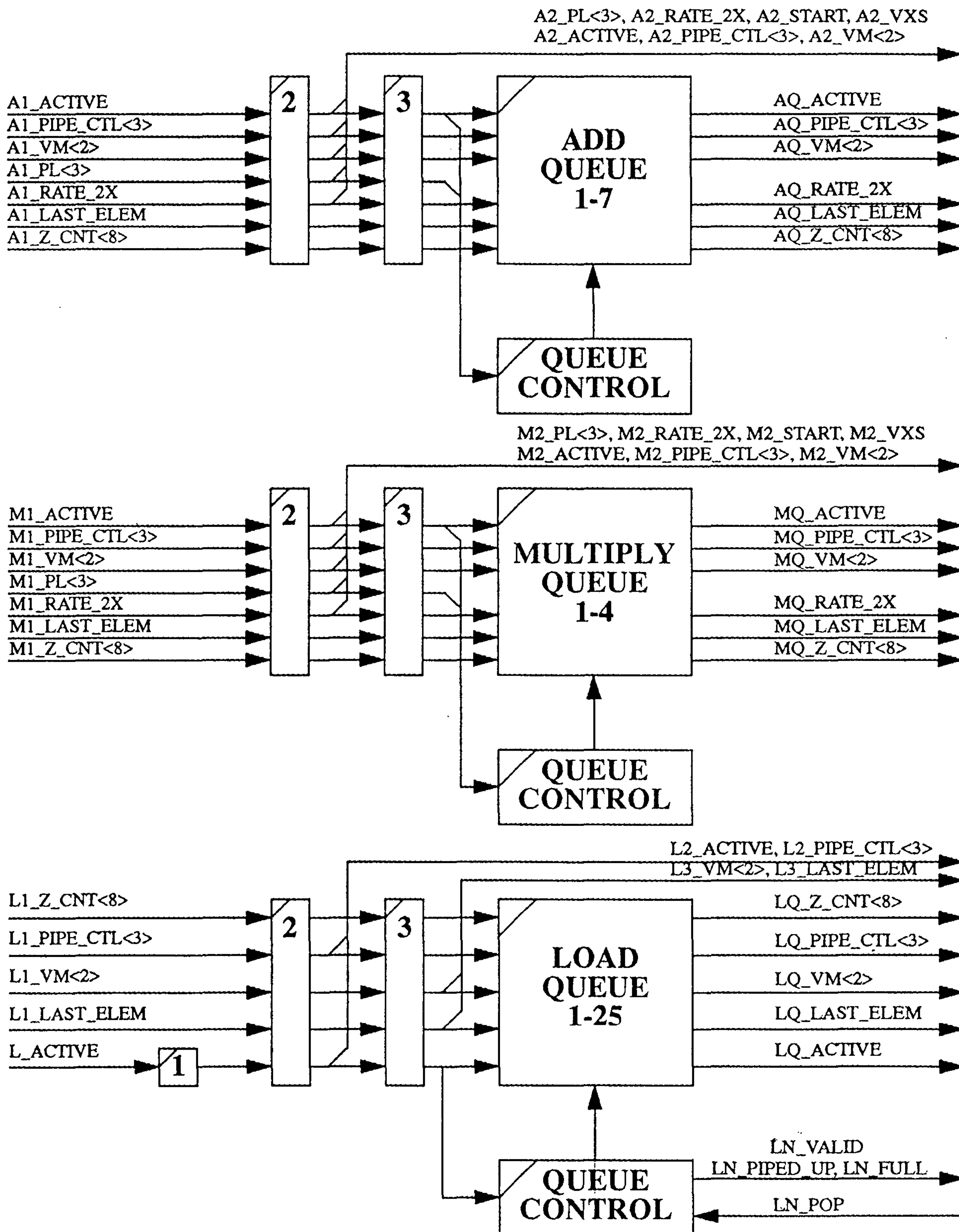


Figure 6-1 NQ Block Diagram

7 Back-Door Controller (BD)

The back-door controller (BD) is a logic block that contains back-door level control for all three pipes. Things described as "back-door" are data and control logic that are at the pipeline level just prior to writing result or load data into the VRF or VM bits from a compare into the VM register. The BD controller decodes control signals coming out of the control queues in the NQ array. It implements the last level of the load pipe control queue and part of the control for that queue. The BD controller also contains the generation and distribution of clock extend signals for the entire NVP.

The BD controller is implemented with seven PALs, three 100e141 registers, a 100e142 register, some 100e101 OR gates, and eleven 100e111 buffers for distribution of clock extend signals.

7.1 Add pipe back-door logic (page 1)

The back-door logic for the add pipe is implemented in one of the PALs 5203 and the 100e142 register. The BD logic for the add pipe primarily is used to decode some of the add pipe control signals that come out of the NQ array. These decoded signals are then sent to the destinations where add pipe back-door level control is needed. The add pipe BD logic also has one bit of state, *R.ABD_STATE*, that it uses internally.

The add pipe BD logic receives *AQ_LAST_ELEM*, $AQ_PIPE_CTL_{2..0}$, and *AQ_ACTIVE* from the NQ array. All of these signals are staged versions of control signals that were generated by the UA controller and staged a number of cycles equal to the pipe length for the instruction. PAL 5203 combines these signals and *R.ABD_STATE* to form *AQ_VM_WR*, *AQ_LD_BD*, *AQ_FIRST_WR*, *AQ_LAST_WR* and *C.ABD_STATE*.

AQ_VM_WR is generated by PAL 5203. It indicates that the add pipe is writing to the VM register at the back door level. This signal goes to the STAT controller to select between add and multiply pipe signals for writing into the VM register.

AQ_LD_BD is generated by PAL 5203. It indicates that an operation on the add pipe has reached the back-door level. The VD logic uses this signal to allocate back-door level resources for the add pipe.

AQ_FIRST_WR is generated by PAL 5203. It indicates that the add pipe has made the first write of an operation to a vector register or the VM register. The VD logic uses this signal to allow an instruction to chain. That is, it allows an instruction to start on another pipe which uses results of this add pipe operation as operands.

AQ_LAST_WR is generated by PAL 5203. It indicates that the add pipe has made the last write of an operation to a vector register or the VM register. The VD logic uses this signal to free up VRF and VM write port resources that were allocated to an instruction.

R.ABD_STATE is a state signal that indicates that something is running at the back-door level in the add pipe which will write to either the VRF or VM register.

7.2 Multiply pipe back-door logic (page 1)

The back-door logic for the multiply pipe is implemented in one of the PALs 5203 and the 100e142

register. The BD logic for the multiply pipe is essentially identical to that of the add pipe. The functionality of the multiply pipe BD logic can be understood from the description of the add pipe BD logic in Section 7.1 on page 7-1. The signals exactly correspond except for *AQ_VM_WR*, which is a mux select for the STAT logic and would be the **inverse** of *MQ_VM_WR* if there were an *MQ_VM_WR*. There is no *MQ_VM_WR*.

7.3 Load pipe back-door logic (page 1)

The back-door logic for the load pipe is implemented in PALs 5204, 5205 and 5209, the three 100e141 registers, and the 100e101 OR gates. The load pipe BD controller contains significantly more logic than those for the add or multiply pipes. The BD logic for the load pipe is used to decode or stage some of the load pipe control signals that come out of the NQ array. In several cases, the last level of the control queue for the load pipe is implemented here, rather than in the NQ array, for timing reasons. The decoded and staged signals are then sent to the destinations where load pipe back-door level control is needed: primarily in the VRFs and VD logic. The load pipe BD logic also has one bit of state, *R.LBD_STATE*, that it uses internally.

7.3.1 Load BD control signals

LN_POP is generated by PAL 5204. It indicates that the load pipe back-door is removing an entry from the load pipe control queue in the NQ array. The last level of this queue is actually implemented in the 100e141 registers on BD page 1. If *LN_POP* is not asserted (during normal operation) the 100e141 registers hold their value, and the portion of the queue inside the NQ array is stopped. If *LN_POP* is not asserted, any new entries for the load pipe control queue will increase the depth of the queue. If *LN_POP* is asserted, the 100e141 registers are enabled, and the load pipe control queue advances.

C.LQ_POP is generated by PAL 5204. It indicates that the back-door level of the load pipe controller is going to remove data from either the SXV or MXV queue. This signal is used in the creation of *LN_POP*.

$LQ_RSLT_SEL_{1..0}$ is generated by PAL 5209. It goes to the NIS arrays to select what type of data will be put on the $LQ_RSLT_DAT_{63..0}$ bus which is the load pipe result bus to the VRF.

LQ_SXV_POP and *LQ_MXV_POP* are generated by PAL 5209. They go to the ISCTL and NIS gate arrays respectively. They indicate that the load pipe is taking data from the SXV or MXV queue.

LQ_LD_BD is generated by PAL 5205. It indicates that an operation on the load pipe has reached the back-door level and is preparing to store into the VRF. The VD logic uses this signal to allocate back-door level resources for the load pipe.

LQ_FIRST_WR is generated by PAL 5205. It indicates that the load pipe has made the first write of an operation to a vector register. The VD logic uses this signal to allow an instruction to chain. That is, it allows an instruction to start on another pipe which uses load data as operands.

LQ_LAST_WR is generated by PAL 5205. It indicates that the load pipe has made the last write of an operation to a vector register. The VD logic uses this signal to free up VRF and VM write port resources that were allocated to an instruction.

LQ_LAST_SXV is generated by PAL 5205. It indicates that the load pipe has made the last SXV transfer for an instruction. The VD logic uses this signal to free SXV port allocation. Only one instruction at a time can be running which has an SXV read pending.

R.LBD_STATE is a state signal that indicates that something is running at the back-door level in the load pipe which will write to the VRF.

C.LQ_ACTIVE is generated by PAL 5204 and registered to create *LQ_ACTIVE*. (*LQ_ACTIVE.BD*, *LQ_ACTIVE.BD1* and *LQ_ACTIVE.BD2* are duplicate copies of *LQ_ACTIVE* for timing purposes). It indicates that there is a valid operation running on the load pipe at the back-door level.

C.L_ACTIVE.BD is generated by PAL 5203 and is registered to create *L_ACTIVE.BD*. *L_ACTIVE.BD* is a copy of the signal *L_ACTIVE* that is driven by the L_VM NVM array. It is generated on BD page 1 for timing reasons. The PAL and the 100e142 register compose a set-reset flip-flop which is set by *L_DISPATCH* and cleared by either *L_CLR_ACTIVE* or *L_RDY*.

7.3.2 The last level of the load pipe control queue

The last level of the load pipe control queue is implemented in 100e141 registers on BD page 1. This generally amounts to receiving next-to-last-level signals (prefixed by LN) and registering them to produce back-door level signals (prefixed by LQ). For timing reasons this cannot be done in the NQ array. The LN and LQ signals are $LN_Z_CNT_{6..0}$ and $LQ_Z_CNT_{6..0}$, $LN_PIPE_CTL_{2..0}$ and $LQ_PIPE_CTL_{2..0}$, $LN_VM_{1..0}$ and $LQ_VM_{1..0}$, *LN_LAST_ELEM* and *R.LQ_LAST_ELEM*.

These registers can be stopped or scanned by the *SEL1* and *SEL0* signals. The following table lists the control codes. The 100e101 OR/NOR gates create the appropriate select signals from *LN_POP*, *SCAN_LEFT* and *EXT_CK_XTND*.

SEL0	SEL1	Function
0	0	Load
0	1	Shift
1	1	Hold
1	0	Broken (shift right is not used on the NVP)

Four control signals also come out of the NQ gate array at the next-to-last level. These are *LN_PIPED_UP*, *LN_VALID*, *LN_MEM_FULL* and *LN_FULL*. *LN_PIPED_UP* is generated by the NQ array and indicates that the load pipe control queue is filled with requests for memory data or that the UL controller has finished making requests for memory data. It is registered to create *R.LQ_PIPED_UP* which is used during *ldvi* instructions so that the NVP fills the memory system with requests before accepting any memory data. The BD controller will not assert *LQ_POP* during an *ldvi* until *R.LQ_PIPED_UP* is asserted.

LN_VALID indicates that some operation is occurring on the load pipe past the front-door level. It indicates the validity of the other back-door level load pipe control signals.

LN_MEM_FULL is an indication to the vector dispatch logic that the NQ is nearly full of requests to memory. Vector dispatch uses this to avoid dispatching instructions that could cause deadlock in the memory system.

LN_FULL is also generated by the NQ array and indicates that the load pipe control queue in the NQ array is completely full. It is registered to generate *R.LQ_FULL* which is used by the BD controller to generate clock extends when the NVP cannot make any requests for memory data without overrunning its load pipe control queue.

7.3.3 Clock Extend generation (page 1) and distribution (page 2)

The BD controller receives all of the information required to generate all of the clock extend signals

used on the NVP. It generates these clock extend signals and buffers them so that almost all of the clock extend signals go to only a single destination. There are five types of clock extends used on the NVP. They are generated by PALs 5207 and 5208 on BD page 1. Buffering is provided by eleven 100e111 buffers on BD page 2.

CK_XTND is the general clock extend signal for gate arrays. It is used during normal operations to stop most of the clocks on the NVP when one of the following occurs:

- A load or ldvi instruction is running on the load pipe and no MXV data is available when expected.
- The NVP is attempting to send data to the NSP on $VP_SP.DATA_{63..0}$, but the NSP is not accepting the data. This is indicated to the BD controller by *OS_CK_HOLD* being asserted.
- A divide instruction is executing on the add pipe, but no divide result is available from an NDIV. This is indicated to the BD controller by *DIV_CK_HOLD* being asserted.

Some clocks are not stopped by *CK_XTND.* These are primarily the handshakes and overrun queues between the NSP and NVP, and the internal core of the NDIV array.

EXT_CK_XTND is the general clock extend signal for eclips registers. It is essentially the same as *CK_XTND* except that it is also asserted when *EXT_CK_HOLD* is asserted during faults.

FREE_CK_XTND is the clock extend for gate arrays that stops any registers that are not stopped by *CK_XTND.* It is used during faults. All registers must be stopped during faults, but *CK_XTND* does not stop all registers on all gate arrays.

FREE_EXT_CK_XTND is the clock extend for eclips registers that stops any registers that are not stopped by *EXT_CK_XTND.* It is used during faults. All registers must be stopped during faults, but *EXT_CK_XTND* does not stop all eclips registers.

8 The Function Pipes

8.1 The Add Function Pipe (AFP)

The Add function pipe receives operand data from the vector register file and control information from the UA microsequencer, performs operations on the data, and returns the data results to the vector register file and the compare results to the STAT controller. The add pipe performs integer and floating point addition, subtraction, division, square root, comparisons, type conversions, and edit operations. It is composed of the AFC controller, six NDIV arrays, an NMISC array, an NFAD array, and multiplexers for the result bus. A block diagram of the AFP is shown in Figure 8-1 on page 8-1.

For timing reasons, the eight function units drive two result buses - $A_RSLT_DAT_1_{63..0}$ and $A_RSLT_DAT_0_{63..0}$ - which are combined by the AFP multiplexer to form the AFP result bus $AQ_RSLT_DAT_{63..0}$. The AFC controller generates the select for this multiplexer. The drivers of $A_RSLT_DAT_0_{63..0}$ are NDIV0, NDIV1, NFAD_A and NMISC_A. The drivers of $A_RSLT_DAT_1_{63..0}$ are NDIV2, NDIV3, NDIV4 and NDIV5.

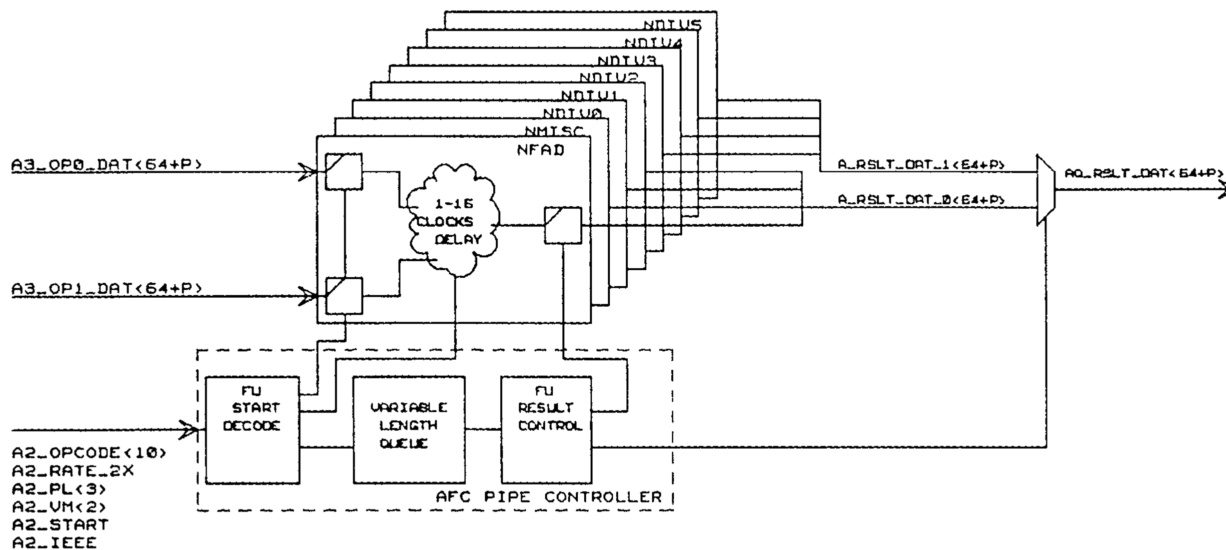


Figure 8-1 The Add Function Pipe (AFP)

8.1.1 The AFC controller

The AFC controller receives control information at the second pipeline level from the UA controller, registers it, uses the registered version to start the function units, and enqueues the control information. The output of the queue is used to enable the outputs of the function units, control the result bus multiplexer, and - in the case of divide/square-root operations - to generate clock extends.

The AFC controller is composed of nine 100e142 registers, eight PALs, five 100e155 multiplexers, and a 100e122 buffer.

8.1.1.1 The Level 3 Registers (page 1)

On AFC page 1 there are two 100e142 registers that receive the control information *A2_START*, $A2_OPCODE_{7..0}$, $A2_PL_{2..0}$, and *A2_RATE_2X* from the A_VM and NQ array. These signals are delayed one clock to align them with the operand data on $A3_OP0_DAT_{63..0}$ and $A3_OP1_DAT_{63..0}$. *A2_IEEE* goes to another 100e142 on page 1. *A2_START* indicates that valid operands and control signals are at the A2 level. These registers also have $C.FU_TYPE_{1..0}$ as an input which is created by PAL 5263 on AFC page 2. It is $A2_OPCODE_{9..8}$ if *A2_START* is asserted or 0b00 if *A2_START* is not asserted. This keeps the AFC from doing bizarre things when the pipe should be idle.

The registers also have $C.A_PTR_{2..0}$ as an input. The registered version, $R.A_PTR_{2..0}$, is a modulo 6 counter that indicates which of the NDIV arrays should be started on this cycle if the pipe is running a divide or square-root operation.

$C.FU_VM_{1..0}$ is generated on page 2 and is registered on page 1 to create $FU_VM_{1..0}$. These are the VM bits, at the third pipeline level, that are sent to the function units. They cause the function units to enable or disable their status bit outputs so that PSW bits are not asserted during under-mask instructions for operations where the VM bits are not set.

8.1.1.2 The AFC Control Queue (mostly page 1)

The AFC control queue is implemented in four 100e142 registers and six 100e155 multiplexers on AFC page 1 and one 100e142 register on page 3. The queue selection controls are generated in PAL 5263 on AFC page 2.

The AFC control queue is a six entry queue. Entries are inserted at any point in the queue and are read out at the head of the queue ($R.QUEUE1_{5..0}$). The normal progression of the queue is from $R.QUEUE6_{5..0}$ to $R.QUEUE5_{5..0}$ and so on down to $R.QUEUE1_{5..0}$. The point where the entries are inserted into the queue is based on the pipe length of the current operation ($R.PL_{2..0}$). The insertion point is controlled by enabling one bit of $QUEUE_SEL_{4..0}$ which is generated in PAL 5263 on page 2. For example, if $R.PL_{2..0} = 4$, then $QUEUE_SEL_{4..0}$ = 0b00100 and the next clock loads the new queue entry into $R.QUEUE3_{5..0}$.

The 6-bit entries in the queue are composed of $R.FU_TYPE_{1..0}$: *R.RATE_2X* : $R.A_PTR_{2..0}$. The queue entries contain all of the information that is required at the output of the function pipe to enable the function unit outputs, control the result bus multiplexer, and generate clock extends. $R.FU_TYPE_{1..0}$ indicates what type of function unit the result will be coming from. $R.A_PTR_{2..0}$ indicates which of the NDIV arrays to expect a result from at the end of the pipe if the operation is a divide or square-root. *R.RATE_2X* indicates whether the result will come from more than one NDIV if the operation is a divide or square-root.

8.1.1.3 Function Unit input level signals (page 2)

The AFC controller generates the signals that are required at the input level of the function units to start the function unit and to accept its operand data from the appropriate half of the bus in rate_2x mode. The start signals include *NMISC_START* and *NDIV_START_5* through *NDIV_START_0*, which are generated by PAL 5257; and *NFAD_START* which is generated by PAL 5260. The start signals are connected to the *UIR1_XXX_REQ* signal of the respective function unit. The start signals are based on $R.FU_TYPE_{1..0}$ for all function units and *R.RATE_2X* and $R.A_PTR_{2..0}$ for the NDIVs.

On page 3, the *A2_RATE_2X* signal is registered to create *NDIV_ODD_OPS*. This signal is sent to NDIVs 1, 3 and 5 on their input pin *XYU_XYL_SEL*. During a rate_2x operation these three NDIVs receive their 32-bit operands from the upper half of the operand bus (i.e., $A3_OP0_DAT_{63..32}$) and must switch it to the lower half of the bus so that they can operate on it. Assertion of *XYU_XYL_SEL* causes this to happen.

8.1.1.4 Function Unit output level signals (page 2)

The AFC controller generates the signals that are required at the output level of the function units to retrieve data from them, enable their outputs onto the result bus, and control the result mux. Each of the function units requires the assertion of its *XXX_RSLT_SEL* pin in order to free its output register and disable the status bits from being driven eternally. These pins are driven by *NFAD_RSLT_SEL*, and *NDIV_RSLT_SEL_5* through *NDIV_RSLT_SEL_0*, all of which are generated by PALs on AFC page 2. All of these selects are based on $R.QUEUE1_{5..0}$. *NMISC_RSLT_SEL* is directly connected to *NMISC_RSLT_NEXT* on logic page NMISC_A. This is due to the fact that the MFC is always ready to accept an output on the next cycle after the result is ready.

The output enables for the function units on the add pipe are also generated on page 2. These include *NFAD_RSLT_OE*, *NMISC_RSLT_OE*, *NDIV_RSLT_UOE_O_1*, *NDIV_RSLT_UOE_E_1*, *NDIV_RSLT_LOE_O_1*, *NDIV_RSLT_LOE_E_1*, *NDIV_RSLT_UOE_O_0*, *NDIV_RSLT_UOE_E_0*, *NDIV_RSLT_LOE_O_0* and *NDIV_RSLT_LOE_E*_0. The *NFAD_RSLT_OE* and *NMISC_RSLT_OE* signals enable the entire bus for the NFAD and NMISC function units since both of these function units drive the entire 64-bit result bus even in rate_2x mode (they are able to perform two 32-bit operations simultaneously).

The NDIV output enables are a bit more complicated. *NDIV_RSLT_UOE_E_1* and *NDIV_RSLT_LOE_E_1* are upper and lower output enables. The "E_1" part of the name indicates that in rate_2x mode the NDIVs drive the even half (hence the "E") of the $A_RSLT_DAT_1_{63..0}$ (hence the "1") bus. The even half is - contrary to convention - the lower half: $A_RSLT_DAT_1_{31..0}$. The same reasoning follows for the other enables: UOE enables the upper half of the bus, LOE enables the lower half of the bus, "E" means that this NDIV drives the lower half of the result bus during rate_2x operations, "O" means that this NDIV drives the upper half of the result bus during rate_2x operations, "0" means that this NDIV drives $A_RSLT_DAT_0_{63..0}$, and "1" means that this NDIV drives $A_RSLT_DAT_1_{63..0}$. Table 8-1 on page 8-4 shows how the divider output enables control the buses.

Table 8-1 NDIV Output Enables

Unit	Bits	Enable	Bus
NDIV0	63..32	UOE_E_0	$A_RSLT_DAT_0_{63..32}$
	31..0	LOE_E_0	$A_RSLT_DAT_0_{31..0}$
NDIV1	63..32	UOE_O_0	$A_RSLT_DAT_0_{63..32}$
	31..0	LOE_O_0	$A_RSLT_DAT_0_{31..0}$
NDIV2	63..32	UOE_E_0	$A_RSLT_DAT_0_{63..32}$
	31..0	LOE_E_0	$A_RSLT_DAT_0_{31..0}$
NDIV3	63..32	UOE_O_0	$A_RSLT_DAT_0_{63..32}$
	31..0	LOE_O_0	$A_RSLT_DAT_0_{31..0}$
NDIV4	63..32	UOE_E_0	$A_RSLT_DAT_1_{63..32}$
	31..0	LOE_E_0	$A_RSLT_DAT_1_{31..0}$
NDIV5	63..32	UOE_O_0	$A_RSLT_DAT_1_{63..32}$
	31..0	LOE_O_0	$A_RSLT_DAT_1_{31..0}$

8.1.1.5 Divider clock extend logic (page 3)

The C38xx Vector Processor is not capable of providing a divide or square-root result on every clock. Since the normal mode of operation for the NVP is to do an operation on every clock, some special action must be taken when this is not possible. The divider clock extend logic causes the clocks on the NVP to be extended during periods when an NDIV result is expected but not available.

During a divide operation the NDIVs are started on a rotating basis based on the value of $R.A_PTR_{2..0}$. Once an NDIV is started, PALs 5261 and 5268 generate *C.Dx_BUSY* for each of the NDIVs which is registered to create *R.Dx_BUSY*. These signals amount to a set-reset flip-flop which is set when the divider is started (*R.NDIV_START<0>* & *R.NDIV_UIR2_VAL*) and cleared when the divider indicates that it is finished (*NDIV_RSLT_NEXT_0*). PALs 5261, 5268 and 5262 combine the busy signals, the last two queue entries ($R.QUEUE2_{5..0}$ and $R.QUEUE1_{5..0}$), and the *NDIV_RSLT_NEXT_x* signals to determine whether a result will be available from the NDIV that is pointed to by the output of the queue; and if no result will be available, to create *C.CK_XTND*. This signal is registered to produce *DIV_CK_HOLD* which goes to the BD controller to generate clock extends for the entire NVP.

Since the *C.CK_XTND* signal is registered before being sent accross the board, the AFC controller must operate one cycle in advance of the clock extends that it might generate. This is the reason for the *C.CK_XTND* equation including $R.QUEUE2_{5..0}$. The AFC controller also can use this information to stop an operation one cycle after it started it by de-asserting *NDIV_UIR2_VAL*, which it does in the case of clock extends.

8.1.2 The NDIVs on AFP

The AFP performs division and square-root operations using six NDIV custom function units. Even with six NDIVs, the NVP cannot produce a result on every cycle. Consequently, clock extends are used to stop the pipelines on the entire NVP while NDIV finishes its operations. See Section 8.1.1.5 on page 8-4 for a description of the clock extend generation. The different NDIVs drive onto two different result buses - $A_RSLT_DAT_1_{63..0}$ and $A_RSLT_DAT_0_{63..0}$ which are combined by the AFP multiplexer to form the AFP result bus $AQ_RSLT_DAT_{63..0}$. NDIV0 and NDIV1are on $A_RSLT_DAT_0_{63..0}$. NDIV2, NDIV3, NDIV4 and NDIV5 are on $A_RSLT_DAT_1_{63..0}$.

Most of the control signals to the NDIVs are fairly well described in Section 8.1.1 on page 8-1.

The operands to the NDIVs come from the VRFs on the *A3_OP0_DAT*$_{63..0}$ and *A3_OP1_DAT*$_{63..0}$ buses. The operands are at the third pipeline level.

The status bits coming out of the NDIVs are bundled together in buses named *AQ_xxx* (where *xxx* is the particular status bit) with the status bits from the NMISC_A and NFAD_A function units and sent to the STAT body where they are OR'ed together, registered, and sent to the NSP as *VP_SP.SET_xxx*. The status bits are only asserted when an operation was started on the NDIV, the VM bit is set, and the output is enabled.

The NDIVs check parity on every cycle on their XBUS and YBUS input registers. Parity checking is enabled by setting the *R.PARITY_ENABLE* bit on the gate array scan ring. Each gate array has such a bit. If parity checking is enabled and a parity error occurs on either the XBUS or YBUS input register, the NDIV asserts its *PAR_ERR* signal and holds the contents of the XBUS and YBUS input registers. The parity error outputs are bundled together with the parity error bits from the NMISC_A and NFAD_A into the bus *AFP_PAR_ERR*$_{7..0}$. This bus is sent to the NVP_CLK body where it is combined with all of the parity error signals on the NVP.

The NDIV arrays have two clock extend inputs. *FREE_CK_XTND* is the clock extend used during context switch operations and stops all registers in the NDIV. *CK_XTND* is the clock extend used during normal operations when data is not available somewhere on the NVP. It stops the input and output registers, but allows the core of the divider to continue iterating.

8.1.3 The NFAD_A and NMISC_A on the AFP

The AFP performs all operations other than division and square-root operations using one NFAD custom function unit and one NMISC gate array. The NFAD on the add pipe is called NFAD_A. It performs floating point adds, subtracts, negates and sums. The NMISC does everything that the NFAD and NDIVs do not do. The NMISC on the add pipe is called NMISC_A. The NFAD_A and NMISC_A both drive their results onto *A_RSLT_DAT_0*$_{63..0}$ which passes through the AFP multiplexer to form the AFP result bus *AQ_RSLT_DAT*$_{63..0}$

Most of the control signals to the NMISC_A and NFAD_A are fairly well described in Section 8.1.1 on page 8-1.

The operands to the NMISC_A and NFAD_A come from the VRFs on the *A3_OP0_DAT*$_{63..0}$ and *A3_OP1_DAT*$_{63..0}$ buses. The operands are at the third pipeline level.

The status bits coming out of the function units are bundled together in buses named *AQ_xxx* (where *xxx* is the particular status bit) with the status bits from the six NDIV function units and sent to the STAT body where they are OR'ed together, registered, and sent to the NSP as *VP_SP.SET_xxx*. The status bits are only asserted when an operation was started on the function unit, the appropriate VM bit is set, and the output is enabled. Since the NFAD and NMISC arrays are capable of performing two 32-bit operations simultaneously during rate_2x mode but only have one set of status bit outpus, the *FU_VM*$_{1..0}$ inputs are staged inside the gate array and used to enable the correct set of status bits.

The NMISC and NFAD function units check parity on every cycle on their XBUS and YBUS input registers. Parity checking is enabled by setting the *R.PARITY_ENABLE* bit on the gate array scan ring. Each gate array has such a bit. If parity checking is enabled and a parity error occurs on either the XBUS or YBUS input register, the function unit asserts its *PAR_ERR* signal and holds the contents of the XBUS and YBUS input registers. The parity error outputs are bundled together

with the parity error bits from the NDIVs into the bus $AFP_PAR_ERR_{7..0}$. This bus is sent to the NVP_CLK body where it is combined with all of the parity error signals on the NVP.

8.1.4 The AFP result mux (page 4)

The AFP result mux is composed of twelve 100e155 2:1 multiplexers and is on AFP page 4. The inputs are two result buses from the function units -$A_RSLT_DAT_1_{63..0}$ and $A_RSLT_DAT_0_{63..0}$ - which are combined by the AFP multiplexer to form the AFP result bus $AQ_RSLT_DAT_{63..0}$. The AFC controller generates the select for this multiplexer. The drivers of $A_RSLT_DAT_0_{63..0}$ are NDIV0, NDIV1, NFAD_A and NMISC_A. The drivers of $A_RSLT_DAT_1_{63..0}$ are NDIV2, NDIV3, NDIV4 and NDIV5.

8.2 The Multiply Function Pipe (MFP)

The Multiply function pipe receives operand data from the vector register file and control information from the UM microsequencer, performs operations on the data, and returns the data results to the vector register file and the compare results to the STAT controller. The multiply pipe performs integer and floating point addition, subtraction, multiplication, comparisons, type conversions, and edit operations. It is composed of the MFC controller, four NMUL arrays, an NMISC array, an NFAD array, and multiplexers for the result bus. A block diagram of the MFP is in Figure 8-2 on page 8-7.

For timing reasons, the six function units drive two result buses - $M_RSLT_DAT_1_{63..0}$ and $M_RSLT_DAT_0_{63..0}$ which are combined by the MFP multiplexer to form the MFP result bus $MQ_RSLT_DAT_{63..0}$. The MFC controller generates the select for this multiplexer. The drivers of $M_RSLT_DAT_0_{63..0}$ are NMUL0, NMUL1 and NFAD_M. The drivers of $M_RSLT_DAT_1_{63..0}$ are NMUL2, NMUL3 and NMISC_M.

8.2.1 The MFC controller

The MFC controller receives control information at the second pipeline level from the UM controller, registers it, uses the registered version to start the function units, and enqueues the control information. The output of the queue is used to enable the outputs of the function units and control the result bus multiplexer.

The MFC controller is composed of five 100e142 registers, four PALs, three 100e155 multiplexers, and a 100e122 buffer.

8.2.1.1 The Level 3 Registers (page 1)

On MFC page 1 there are two 100e142 registers that receive the control information *M2_START*, $M2_OPCODE_{7..0}$, $M2_PL_{2..0}$, *M2_IEEE* , and *M2_RATE_2X* from the M_VM and NQ array. These signals are delayed one clock to align them with the operand data on $M3_OP0_DAT_{63..0}$ and $M3_OP1_DAT_{63..0}$. *M2_START* indicates that valid operands and control signals are at the M2 level. These registers also have $C.FU_TYPE_{1..0}$ as an input which is created by PAL 5255 on MFC page 2. It is $M2_OPCODE_{9..8}$ if *M2_START* is asserted or 0b00 if *M2_START* is not asserted. This keeps the MFC from doing bizarre things when the pipe should be idle.

The registers also have $C.M_PTR_{1..0}$ as an input. The registered version, $R.M_PTR_{1..0}$, is a modulo 4 counter that indicates which of the NMUL arrays should be started on this cycle if the pipe is

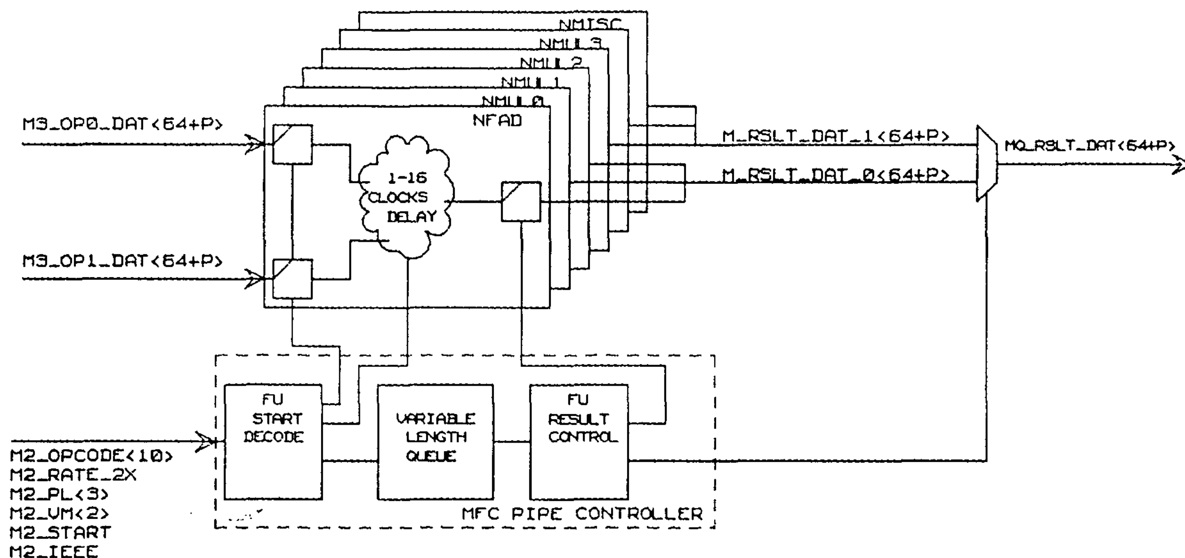


Figure 8-2 The Multiply Function Pipe (MFP)

running a multiply operation.

$C.FU_VM_{1..0}$ is generated on page 2 and is registered on page 1 to create $FU_VM_{1..0}$. These are the VM bits, at the third pipeline level, that are sent to the function units. They cause the function units to enable or disable their status bit outputs so that PSW bits are not asserted during under-mask instructions for operations where the VM bits are not set.

8.2.1.2 The MFC Control Queue (page 1)

The MFC control queue is implemented in three 100e142 registers and three 100e155 multiplexers on MFC page 1. The queue selection controls ($QUEUE_SEL_{2..0}$)are generated in PAL 5256 on MFC page 2.

The MFC control queue is a four entry queue. Entries are inserted at any point in the queue and are read out at the head of the queue ($R.QUEUE1_{4..0}$). The normal progression of the queue is from $R.QUEUE4_{4..0}$ to $R.QUEUE3_{4..0}$ and so on down to $R.QUEUE1_{4..0}$. The point where the entries are inserted into the queue is based on the pipe length of the current operation ($R.PL_{2..0}$). The insertion point is controlled by enabling one bit of $QUEUE_SEL_{2..0}$ which is generated in PAL 5256 on page 2. For example, if $R.PL_{2..0} = 3$, then $QUEUE_SEL_{4..0} = 0b010$ and the next clock loads the new queue entry into $R.QUEUE2_{4..0}$.

The 5-bit entries in the queue are composed of $R.FU_TYPE_{1..0}$: $R.RATE_2X$: $R.M_PTR_{1..0}$. The queue entries contain all of the information that is required at the output of the function pipe to enable the function unit outputs and control the result bus multiplexer. $R.FU_TYPE_{1..0}$ indicates what type of function unit the result will be coming from. $R.M_PTR_{1..0}$ indicates which of the NMUL arrays to expect a result from at the end of the pipe if the operation is a multiply. $R.RATE_2X$

indicates whether the result will come from more than one NMUL if the operation is a multiply.

8.2.1.3 Function Unit input level signals (page 2)

The MFC controller generates the signals that are required at the input level of the function units to start the function unit and to accept its operand data from the appropriate half of the bus in rate_2x mode. The start signals include *NMISC_START* which is generated by PAL 5256; *NMUL_START_3* through *NMUL_START_0*, which are generated by PAL 5253; and *NFAD_START* which is generated by PAL 5253. The start signals are connected to the *UIR1_XXX_REQ* signal of the respective function unit. The start signals are based on $R.FU_TYPE_{1..0}$ for all function units and *R.RATE_2X* and $R.M_PTR_{1..0}$ for the NMULs.

On page 2, *R.RATE_2X* and $R.M_PTR_{1..0}$ are used by PAL 5253 to create *NMUL_ODD_OPS_1* and *NMUL_ODD_OPS_3*. These signals are sent to NMULs 1 and 3, respectively, on their input pin *XYU_XYL_SEL*. During a rate_2x operation these two NMULs receive their 32-bit operands from the upper half of the operand bus (i.e., $M3_OP0_DAT_{63..32}$) and must switch it to the lower half of the bus so that they can operate on it. Assertion of *XYU_XYL_SEL* causes this to happen.

8.2.1.4 Function Unit output level signals (page 2)

The MFC controller generates the signals that are required at the output level of the function units to retrieve data from them, enable their outputs onto the result bus, and control the result mux. Each of the function units requires the assertion of its *xxx_RSLT_SEL* pin in order to free its output register and disable the status bits from being driven eternally. *NFAD_RSLT_SEL* is generated by PAL 5256 on MFC page 2. This select is based on $R.QUEUE1_{4..0}$. *NMISC_RSLT_SEL* is directly connected to *NMISC_RSLT_NEXT* on logic page NMISC_M. *NMUL_RSLT_SEL_3* through *NMUL_RSLT_SEL_0* are directly connected to the *NMUL_RSLT_SEL* inputs for the respective function units. This is due to the fact that the MFC is always ready to accept the output on the next cycle after it is finished for the NMISC and NMUL function units.

The output enables for the function units on the multiply pipe are also generated on page 2. These include *NFAD_RSLT_OE*, *NMISC_RSLT_OE*, *NMUL_RSLT_UOE_3*, *NMUL_RSLT_UOE_2*, *NMUL_RSLT_UOE_1*, *NMUL_RSLT_UOE_0*, *NMUL_RSLT_LOE_3*, *NMUL_RSLT_LOE_2*, *NMUL_RSLT_LOE_1*, and *NMUL_RSLT_LOE_0*. The *NFAD_RSLT_OE* and *NMISC_RSLT_OE* signals enable the entire bus for the NFAD and NMISC function units since both of these function units drive the entire 64-bit result bus even in rate_2x mode (they are able to perform two 32-bit operations simultaneously). The NMUL output enables are seperate for the upper and lower halves of the bus. *NMUL_RSLT_UOE_3* and *NMUL_RSLT_LOE_3* are upper and lower output enables for NMUL3.

8.2.2 The NMULs on MFP

The MFP performs multiply operations using four NMUL custom function units. The different NMULs drive onto two different result buses - $M_RSLT_DAT_1_{63..0}$ and $M_RSLT_DAT_0_{63..0}$ which are combined by the MFP multiplexer to form the MFP result bus $MQ_RSLT_DAT_{63..0}$. NMUL0 and NMUL1 are on $M_RSLT_DAT_0_{63..0}$. NMUL2 and NMUL3 are on $M_RSLT_DAT_1_{63..0}$.

Most of the control signals to the NMULs are fairly well described in Section 8.1.1 on page 8-1.

The operands to the NMULs come from the VRFs on the $M3_OP0_DAT_{63..0}$ and $M3_OP1_DAT_{63..0}$ buses. The operands are at the third pipeline level.

The status bits coming out of the NMULs are bundled together in buses named *MQ_xxx* (where *xxx* is the particular status bit) with the status bits from the NMISC_M and NFAD_M function units and sent to the STAT body where they are OR'ed together, registered, and sent to the NSP as *VP_SP.SET_xxx* . The status bits are only asserted when an operation was started on the NMUL, the VM bit is set, and the output is enabled.

The NMULs check parity on every cycle on their XBUS and YBUS input registers. Parity checking is enabled by setting the *R.PARITY_ENABLE* bit on the gate array scan ring. Each gate array has such a bit. If parity checking is enabled and a parity error occurs on either the XBUS or YBUS input register, the NMUL asserts its *PAR_ERR* signal and holds the contents of the XBUS and YBUS input registers. The parity error outputs are bundled together with the parity error bits from the NMISC_M and NFAD_M into the bus $MFP_PAR_ERR_{5..0}$. This bus is sent to the NVP_CLK body where it is combined with all of the parity error signals on the NVP.

8.2.3 The NFAD_M and NMISC_M on the MFP

The MFP performs all operations other than multiply operations using one NFAD custom function unit and one NMISC gate array. The NFAD on the multiply pipe is called NFAD_M. It performs floating point adds, subtracts, negates and sums. The NMISC does everything that the NFAD and NMULs do not do. The NMISC on the multiply pipe is called NMISC_M. The NFAD_M drives its results onto $M_RSLT_DAT_0_{63..0}$ and NMISC_M drives its results onto $M_RSLT_DAT_1_{63..0}$ which are combined by the MFP multiplexer to form the MFP result bus $MQ_RSLT_DAT_{63..0}$

Most of the control signals to the NMISC_M and NFAD_M are fairly well described in Section 8.1.1 on page 8-1.

The operands to the NMISC_M and NFAD_M come from the VRFs on the $M3_OP0_DAT_{63..0}$ and $M3_OP1_DAT_{63..0}$ buses. The operands are at the third pipeline level.

The status bits coming out of the function units are bundled together in buses named *MQ_xxx* (where *xxx* is the particular status bit) with the status bits from the four NMUL function units and sent to the STAT body where they are OR'ed together, registered, and sent to the NSP as *VP_SP.SET_xxx* . The status bits are only asserted when an operation was started on the function unit, the appropriate VM bit is set, and the output is enabled. Since the NFAD and NMISC arrays are capable of performing two 32-bit operations simultaneously during rate_2x mode but only have one set of status bit outpus, the $FU_VM_{1..0}$ inputs are staged inside the gate array and used to enable the correct set of status bits.

The NMISC and NFAD function units check parity on every cycle on their XBUS and YBUS input registers. Parity checking is enabled by setting the *R.PARITY_ENABLE* bit on the gate array scan ring. Each gate array has such a bit. If parity checking is enabled and a parity error occurs on either the XBUS or YBUS input register, the function unit asserts its *PAR_ERR* signal and holds the contents of the XBUS and YBUS input registers. The parity error outputs are bundled together with the parity error bits from the NMULs into the bus $MFP_PAR_ERR_{5..0}$. This bus is sent to the NVP_CLK body where it is combined with all of the parity error signals on the NVP.

8.2.4 The MFP result mux (page 3)

The MFP result mux is composed of twelve 100e155 2:1 multiplexers and is on MFP page 3. The inputs are two result buses from the function units - $M_RSLT_DAT_1_{63..0}$ and $M_RSLT_DAT_0_{63..0}$ -

which are combined by the MFP multiplexer to form the MFP result bus *MQ_RSLT_DAT*$_{63..0}$. The MFC controller generates the select for this multiplexer. The drivers of *M_RSLT_DAT_0*$_{63..0}$ are NMUL0, NMUL1 and NFAD_M. The drivers of *M_RSLT_DAT_1*$_{63..0}$ are NMUL2, NMUL3 and NMISC_M.

10 The Output Staging Controller (OSCTL)

The OSCTL controller manages the Vector-to-Scalar (VXS) transfers from the add, multiply and load pipes, and Vector-to-Memory (VXM) transfers from the load pipe. It also manges transfer of NVP context data to memory during faults. The data for these transfers moves on $VP_SP.DATA_{63..0}$ and $VP_SP.VXA_ADDR_{31..0}$. The controller generates clock extend signals for the NVP when data is available to transfer to the NSP but the NSP is not accepting the data. The OSCTL is implemented in two PALS, a 100e142 register, and a 100e158 2:1 mux.

A list of the vector to scalar interface signals is given in Table 10-1 on page 10-1.

Table 10-1 Vector to Scalar interface signals

Signal	Description
$VP_SP.DATA_{63..0}$	Vector data
$VP_SP.PAR_{7..0}$	Parity for same
$VP_SP.VXA_ADDR_{31..0}$	Address indexes
$VP_SP.VXA_ADDR_PAR_{4..0}$	Parity for same
$VP_SP.VXA_VM_{1..0}$	VM bits for vector data
VP_SP.VXA_LAST	Indicates last data transfer
VP_SP.VXS_RDY	Handshake for VXS data
VP_SP.VXM_RDY	Handshake for VXM data
SP_VP.VX_REQ_NEXT	Handshake for VXM and VXS data

The NVP is designed such that there is no more than one instruction running at any time that does a VXS or VXM transfer. (During a fault, nothing else is running). Hence, the OSCTL only needs to manage one type of transfer at a time. The handshakes for the transfer are the usual REQ_NEXT/RDY variety. The controller asserts *VP_SP.VXS_RDY* if it has scalar data to send, and *VP_SP.VXM_RDY* if it has memory data to send. The NSP responds by asserting *SP_VP.VX_REQ_NEXT* to accept data of either type. In order for a valid transfer to occur, the REQ_NEXT signal has to be asserted on the cycle before the RDY signal. Otherwise, no transfer takes place. A diagram of the vector to scalar interface handshakes is shown in Figure 10-1 on page 10-2. This diagram is not intended to imply that VXM and VXS transfers actually happen on consecutive cycles.

10.1 Normal mode operation

The OSCTL determines whether a VXS or VXM transfer is occurring from the state of the A2, M2 and L2 control signals from the three pipes. If *A2_ACTIVE* is asserted, then $A2_PIPE_CTL_{2..0}$ is valid. Likewise for *M2_ACTIVE* and $M2_PIPE_CTL_{2..0}$, and *L2_ACTIVE* and $L2_PIPE_CTL_{2..0}$. If the ACTIVE signal is not asserted, then the PIPE_CTL signal is ignored. Table 10-2 on page 10-2 shows the interpretation of the control signals by the OSCTL.

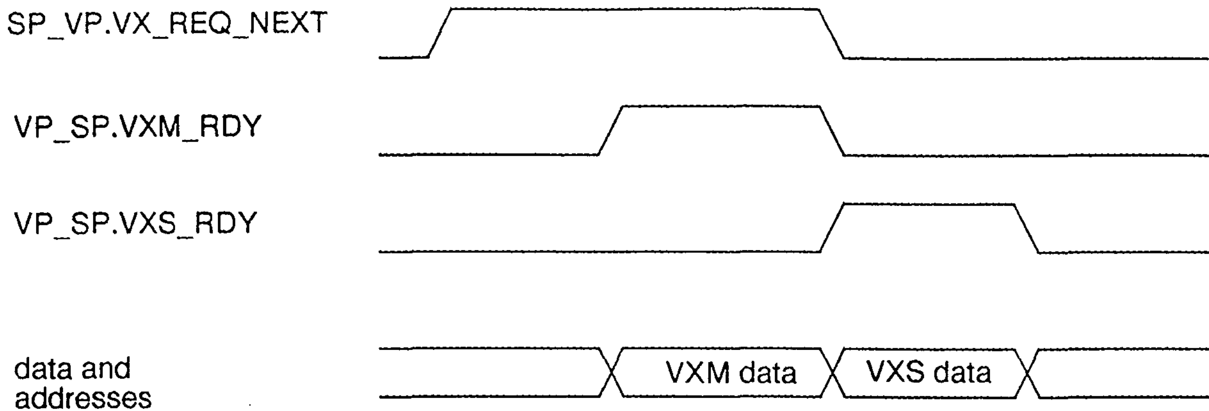


Figure 10-1 Vector to Scalar bus interface

Table 10-2 OSCTL control signals

Xfer type	pipe	Control Signal Value
VXS	Add	*A2_PIPE_CTL*$_{2..0}$0x7
VXS	Multiply	*M2_PIPE_CTL*$_{2..0}$0x7
VXS	Load	*L2_PIPE_CTL*$_{2..0}$0x7
VXM	Load	*L2_PIPE_CTL*$_{2..0}$0x2

The controller generally asserts the RDY signals (*VP_SP.VXS_RDY* and *VP_SP.VXM_RDY*) if the input control signals indicate that a transfer is to be made. If the controller is ready to make a transfer and the NSP does not assert *SP_VP.VX_REQ_NEXT*, the controller asserts *OS_CK_HOLD* which goes to the BD controller and causes clock extends to be asserted on the NVP. This is done because the NVP operates in lock step, and if one portion of the pipeline is stopped (in this case, output staging), then the entire pipeline must be stopped.

During VXS transfers, only a single element is sent across the *VP_SP.DATA*$_{63..0}$ bus. During VXM transfers, however, a variable number of transfers can occur. During the last of these VXM transfers, the OSCTL receives *L3_LAST_ELEM* and passes it to the NSP as *VP_SP.VXA_LAST* which indicates to the NSP that the last VXM transfer has occurred.

10.2 Fault mode operation

During faults, the context controller handles the handshaking of context data which is being sent to memory, and the OSCTL merely passes them along. The 100e158 mux switches over to the context mode handshakes which are driven by the context controller in the NVP_CLK body. *VP_SP.VXM_RDY* is driven by *SAVE_RDY*, which indicates that a peice of context data is available. *VP_SP.VXS_RDY* is set to zero. *VP_SP.VXA_LAST* is driven by *SAVE_LAST* from the context controller. *VP_SP.VXA_VM<1>* is set to zero. *VP_SP.VXA_VM<0>* is driven by *SAVE_VM* from the context controller.

The OSCTL controller itself does not participate in the context transfers, nor does it cause clock extends. The context controller performs these functions. If the NSP indicates that it is not ready

to accept data by de-asserting *SP_VP.VX_REQ_NEXT*, the context controller generates the clock extends.

11 The Status Logic (STAT)

The status logic (STAT) contains two different functions dealing with two types of status. The first type is the status that results from compare operations that is used to write into the VM register. Compare operations can be performed on the add and multiply pipes (not simulataneously). The STAT logic has a muliplexer that selects which pipe's compare results will be sent back to the NVM gate arrays on the "SQ" bus.

The other type of status that the STAT logic deals with is the PSW bits that generated by the function units result from operations on the function pipes. These PSW bits are accumulated by the STAT logic, registered, and sent to the NSP on the "VP_SP.SET" signals. The PSW bits that are sent to the NSP are listed in Table 11-1 on page 11-1.

Table 11-1 Vector to Scalar PSW bits

VP_SP.SET_SIV	Integer overflow PSW bit
VP_SP.SET_UN	Floating-point underflow PSW bit
VP_SP.SET_OV	Floating-point overflow PSW bit
VP_SP.SET_RO	Floating-point PSW bit
VP_SP.SET_SDZ	Integer divide-by-zero PSW bit
VP_SP.SET_FDZ	Floating-point divide-by-zero PSW bit
VP_SP.SET_FSN	Floating-point sqare-root-of-negative-number PSW bit

11.1 Compare Results

The results of compare operations on the add and multiply pipe are reduced to a single bus in the STAT logic using 100e155 2:1 multiplexers. The add and multiply pipes can both perform compare operations (e.g., *le.w v0,v1*), but the VM register has a single write port. The STAT logic must select between the compare results from the add and multiply pipe. For example, the STAT mux recieves both $AQ_Z_CNT_{7..0}$ and $MQ_Z_CNT_{7..0}$. If *AQ_VM_WR* is a "one", then the add pipe is the one doing the writing to the VM register, and $AQ_Z_CNT_{7..0}$ is selected on to $SQ_Z_CNT_{7..0}$. *AQ_VM_WR* is sourced by the BD controller, which uses the PIPE_CTL signals to determine which pipe is writing to the VM register (if either one is). The SQ signals are sent to the NVM arrays to control writing into the VM register copy in each of those arrays.

11.2 NVP PSW Bits

The STAT logic receives the PSW bits from all of the function units on both function pipes. For each of the types of PSW bits (i.e., RO), the STAT logic OR's all of the possible sources of that PSW bit using 100e101 OR gates. This produces the a set bit for each of the PSW types (i.e., *C.SET_RO*). The set bit is registered to produce the PSW signal that is sent to the NSP (i.e., *VP_SP.SET_RO*). The NSP uses these "VP_SP.SET" signals to set the corresponding bit in the PSW if *VP_SP.PSW_HAZ* is asserted.

12 Clock Generation Logic (NVP_CLK)

The clock generation and distribution logic are in the schematic body NVP_CLK on Pages 1 and 2. These pages also contain the parity error accumulation logic, the generation of SRAM write enables, and some registering of board level signals. Pages 3 and 4 have the context controller which is described in another section. The interface signals to the NVP_CLK logic that come from of the NVP to the NVP_CLK or leave the board from the NVP_CLK are listed in Table 12-1 on page 12-1.

Table 12-1 System interface to NVP_CLK

XC_VP.CLOCK_3X	Board clock running at 3x rate
*XC_VP.CLOCK_3X**	Complement of XC_VP.CLOCK_3X
XC_VP.SCAN_CTL$_{2..0}$	Board scan controls
XC_VP.SCAN_IN	Board scan input
VP_XC.HARD_ERROR	Hard error
VP_XC.SCAN_OUT	Board scan out

12.1 Clock generation

The NVP_CLK logic receives the 3x system clock *XC_VP.CLOCK_3X* and generates the appropriate rate and phase clocks for all of the clocked devices on the NVP. The method of clock numbering is that at each level of buffering another number is appended to the name. For example, the first buffered version of the 3x clock is *CK3.0*, the second is *CK3.0.0*, and the third is *CK3.0.0.0*. Clocks that are at the 3x rate are prefixed CK3. All others are at the 1x rate.

12.1.1 Clock fanout

The clocks are all sourced from *XC_VP.CLOCK_3X* and each clock line at every level of the clock fanout tree has a single source and a single destination. Every clock that goes to a clocked device goes through three layers of buffering.

XC_VP.CLOCK_3X is received from the XCL and goes directly to a 100e111 buffer which creates the first level clocks *CK3.x* (where "x" is 0 through 3). *CK3.0* goes to another buffer to generate *CK3.0.0* which is buffered again to generate *CK3.0.0.0* which is the 3x clock for the NVRF gate arrays. *CK3.1* goes to a 100e111 buffer to generate clocks *CK.1.x* which are buffered again to generate the clocks that go to all of the gate arrays except the NVRFs and to all of the discrete registers on the NVP except the phase generation register. *CK3.2* runs the phase generation 100e451 register on page 1.

12.1.2 Reduction of 3x clock to 1x (page 1)

The 3x clock is reduced to a 1x clock for most clocked devices on the NVP. This is accomplished by gating out 2 out of 3 negative pulses in the 3x clock. The one pulse left out of three means that the clock has been divided by three. The logic that accomplishes this is 6 bits of a 100e451 register, a 100e158 2:1 mux, and the 100e111 buffer that produces clocks *CK.1.x*.

The 100e451 register is connected in a 6-bit ring during normal operation. This ring is named *PHASE_GEN_1X*$_{5..0}$. The ring is clocked at the 3x rate by *CK3.2*. The ring is initialized to 0b110110. On each clock, the ring rotates so that it goes through the sequence: 110110, 101101, 011011. From this it can be seen that *PHASE_GEN_1X<5>* is zero on every third 3x clock and one on the

first and second 3x clocks. PHASE_GEN_1X<5> is passed through the 100e158 2:1 mux and becomes CK_EN*. CK_EN* goes to the enable of the 100e111 buffer that produces CK.1.x. The enable OR's CK_EN* with CK3.1 with the result that two out of three negative pulses are removed from CK3.1 resulting in a 1x clock at the output of the 100e111.

The reason that the PHASE_GEN_1X is 6 bits long instead of three is so that the CAST hardware can perform two consecutive clocks in any phase order that it needs to.

During diagnostic scan, this configuration changes. The signal ALL_SCAN_LEFT is asserted, which causes the 100e158 mux that generates CK_EN* to switch to a floating input, causing CK_EN* to be zero, which results in the 1x clocks running the same rate as XC_VP.CLOCK_3X. ALL_SCAN_LEFT also switches the 100e158 mux that drives the register input for PHASE_GEN_1X<0> so that the phase generation ring is connected in the scan chain instead of being an isolated recirculating ring.

12.2 Hard error generation (page 1)

NVP_CLK page 1 has 100e101 OR gates to OR all of the parity error signals from across the NVP to generate C.HALT, which is registered to generate VP_XC.HARD_ERROR. The parity error signals from all of the function units are OR'ed to generate FU_ERROR, which is sent to schematic section VRF72 to halt the registers that save nibble parity from the VRFs. See Section 4.2 on page 4-5 for a description of nibble parity for the VRFs.

12.3 RSLT_PARITY_ENABLE (page 1)

RSLT_PARITY_ENABLE is a scan accessible register which enables parity checking of result data to the NVRF gate arrays. If this signal is a zero, RSLT_PAR_ERR will never be asserted and the NVRFs will not hold their data registers in the event of a parity error. The parity error will be ignored.

12.4 SRAM write controls

NVP_CLK page 1 has the logic required to enable writing the SRAMs that are used for the UA, UM, UL and VD writable control stores (WCS). The function is implemented in a 100e142 register and PAL 5226. The control stores are implemented 100474 1k x 4 ECL SRAMs. Each of the RAMs has a chip select (CS*) and a write enable (WE*). The operation of the CS* and WE* inputs to the RAMs are shown in Table 12-2 on page 12-2.

Table 12-2 100474 SRAM Write/Select Modes

CS*	WE*	Mode
1	X	Output disable
0	0	Write
0	1	Read

The SRAM write control logic on NVP_CLK page 1 generates separate chip select and write enable signals for all four of the writable control stores. For the UA they are UA_WCS_CS* and UA_WCS_WE*. For the UM they are UM_WCS_CS* and UM_WCS_WE*. For the UL they are UL_WCS_CS* and UL_WCS_WE*. For the VD they are VD_WCS_CS* and VD_WCS_WE*.

PAL 5226 generates these CS* and WE* signals from the registered signals R.UA_WE, R.UM_WE, R.UL_WE, R.VD_WE, and R.WCS_WP<0>. These signals are all accessible only from the scan ring.

R.WCS_WE must also be set to one when any of the SRAMs are to be written. It causes *CNTX_MODE* and *CNTX_CK_HOLD* to be asserted. These go to the BD controller and cause the clock extends to be asserted, stopping almost all of the registers on the NVP. The result is that the address and data for the SRAM are held despite the fact that the RAM write occurs in scan_normal mode.

12.4.1 SRAM Read Mode

In normal (read) mode we should have CS*=0 and WE*=1 according to Table 12-2 on page 12-2. For the UA WCS, PAL 5226 creates this condition when *R.UA_WE* is zero. The controls for the other WCS's operate on the same basis. Therefore, when not writing the RAMs, *R.UA_WE*, *R.UM_WE*, *R.UL_WE* and *R.VD_WE* should all be zero.

12.4.2 SRAM Write Mode

To write the 100474 SRAMs we should set CS*=0 and WE*=0 according to Table 12-2 on page 12-2. This mode is used on the NVP when it is desired to write the writable control stores. All four WCS's are implemented in the same way, so the UA control store will be used as an example. For the UA WCS, PAL 5226 creates CS*=0 and WE*=0 for the UA SRAMs when *R.UA_WE* is one and *R.WCS_WP<0>* is zero. In preparation to write the UA WCS, *R.UA_WE* should be scan initialized to a one, and *R.WCS_WP*$_{1..0}$ should be scan initialized to 0b01. After the scan controls switch from shift_left back to scan_normal, *R.WCS_WP*$_{1..0}$ becomes a ring that cycles between 0b01 and 0b10. This causes *R.WCS_WP<0>* to toggle, which results in results in UA_WCS_CS* toggling. The result is that UA_WCS_WE*=0 and UA_WCS_CS* toggling on each clock cycle, with the SRAM controls alternating between "Output disable" and "Write" modes.[1]

Figure 12-1 on page 12-4 shows the relationship of all of these signals for a write of the UA WCS. The controls for the other WCS's operate on the same basis as those for the UA WCS. When not writing the RAMs, *R.UA_WE*, *R.UM_WE*, *R.UL_WE* and *R.VD_WE* should all be zero.

12.4.3 SRAM Output Disable mode

The 100474 SRAMs are put in output disable mode any time that CS*=1 according to Table 12-2 on page 12-2. This happens in two cases on the NVP. When *ALL_SCAN_LEFT* is asserted, the 100474 SRAM is put into output disable mode where CS*=1 and WE*=1 for each WCS. The output disable mode is also entered on every other clock during WCS writes when CS*=1 and WE*=0 for the particular RAMs that are being written.

---

1. This is different from the 100474 data sheet specification of write mode. The data sheet shows CS* being held low and WE* being a write pulse bracketed by CS*. The NVP implementation has WE* being held low while CS* is the write pulse.

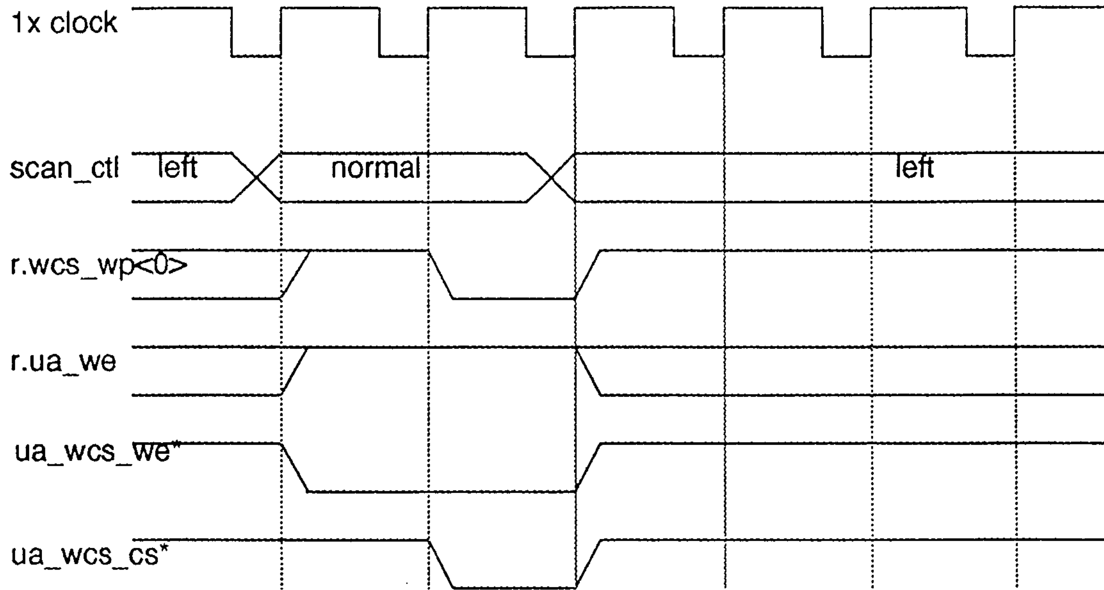


Figure 12-1 Writing the UA WCS SRAMs

13 Scan/Context controller (NVP_CLK)

The context controller logic is implemented in schematic body NVP_CLK on pages 3 and 4. The context controller performs the tasks required to save NVP context on faults and to restore NVP context on returns from fault. It contains a state machine and two counters that oversee the context operations, and several PALs to decode the scan and context control signals for the gate arrays.

13.1 Functional overview of context switches

During a context save (also know as a **fault**), the function of the context controller is to remove all of the register state from the NVP and send this state to be stored in memory. After a fault, the NVP should be in an idle state. During a context restore (also known as a **rtnc**), the function of the context controller is to receive context data from memory and restore all of the register state on the NVP. After a rtnc, the NVP should be in exactly the same state that it was in before the fault occurred who's context was restored in the rtnc. It should appear that the fault and rtnc never even happened.

The context mode of the NVP is under control of the NSP via the signal *SP_VP.CNTX_CTL*$_{1..0}$. The function of this control signal is shown in Table 13-1 on page 13-1. The NVP enters the appropriate fault mode immediately after recieving the context control signal.

Table 13-1 NVP Context Control Inputs

SP_VP.CNTX_CTL	Context Mode
0	Context Normal (CNTX_NORMAL)
1	Context Hold (CNTX_HOLD)
2	Context Reset (CNTX_RESET)
3	Context Left (CNTX_LEFT)

13.1.1 Context save (fault)

Both context save and restore use the scan ring to move data in and out of registers. In diagnostic scan, the scan ring is a single ring with over 15,000 bits. During context switches, the scan ring is broken into pieces. Each gate array is a separate piece. The eclips registers are in one contiguous ring (the "ext_ring"). These ring pieces are stacked up in parallel in the IS_MUX to form *SCAN_OUT_BUS*$_{31..0}$. Figure 13-1 on page 13-2 shows which gate array (and ext_ring) drives which bit of *SCAN_OUT_BUS*$_{31..0}$. It also shows the relative lengths of each of the ring segments. For example, each of the eight NVRF gate arrays has a context ring length of 532 bits. They drive *SCAN_OUT_BUS*$_{7..0}$. The context data is actually stored in this fashion in memory, with one end of each segment lined up with all of the others and junk in the bits that are beyond the length of each segment. This permits the data to be restored directly, since on the short segments the junk bits will scan clear through the ring segment and out the other end, leaving the correct data in the gate array (or ext_ring).

Once the segments of the scan ring are collected in *SCAN_OUT_BUS*$_{31..0}$, they proceed to the NIS gate arrays. The NIS gate arrays stage the context data and generate address indexes that the address generator on the NSP will use when storing the context data in memory. The NIS gate arrays send that staged context data and addresses out on *SXV_DAT*$_{31..0}$, which is received by the NVRF gate arrays. The NVRF arrays stage the context data and put it out on *VP_SP.DATA*$_{31..0}$,

Figure 13-1 Mapping of ring segments to SCAN_OUT_BUS<31..0>

and stage the address indexes and put them out on *VP_SP.VXA_ADDR*$_{31..0}$.

While context data is being shifted out of the gate arrays and onto *SCAN_OUT_BUS*$_{31..0}$, it is also being recirculated back into the gate array that it came from (e.g., *NVD_SCAN_OUT* is connected to *NVD_SCAN_IN* in addition to driving *SCAN_OUT_BUS<29>*). When all of the context data has been shifted out of the gate arrays, they have the same state that they did before the beginning of the fault. At this point the context controller asserts *CNTX_CTL*=CNTX_RESET to all of the gate arrays, leaving them in an idle state as described above. The ext_ring is not recirculated, but rather has all zeros scanned into it during a fault, and thus ends up in the idle state after a fault.

Since the ring segments are not all the same length, the context controller asserts clock extends to the gate arrays at various times, so that all of the arrays scan out their data with the last bit of their rings aligned. This also allows for each gate array ring to be recirculated by its exact length.

13.1.2 Context restore (rtnc)

Context restore operates in a manner similar to faults. Instead of scan data being collected from the ring segments and sent to memory, it is received from memory and distributed back to the ring segments. The context restore data comes to the NVP over *SP_VP.DATA*$_{31..0}$ and is staged in the NIS gate arrays which send it out on *SXV_DAT*$_{31..0}$. This bus goes through the IS_MUX which sends it back to the scan inputs of each of the gate arrays and the ext_ring.

13.2 Hardware implementation of the context controller

The context controller is implemented on NVP_CLK pages 3 and 4. There is a state machine implemented in PAL 5222 and a 100e142 register. A block length counter (*R.LCNT*$_{7..0}$) is implemented in PALs 5220 and 5227, a 100e142 register, and two sections of a 100e101 OR gate. A block counter (*R.BCNT*$_{3..0}$) is implemented in PAL 5221 and a 100e142 register. Various other parts of the context controller are scattered in the registers and PALs on page 3.

The scan and context controls are generated on NVP_CLK page 4 from the state of the context controller.

This section will describe the implementation of the context controller. The best functional description of the context controller is to be found in the nvp_clk.isp behavioral model.

13.2.1 The context state machine

The context state machine is an eight-state state machine. Its state is represented by $R.STATE_{2..0}$ which is generated by PAL 5222 and stored registered in a 100e142. The state transition diagram is shown in Figure 13-2 on page 13-4.

The block and length counts are used by the context controller to manage the clock extends which are necessary since the ring segments are not of uniform length, but must be aligned. The block count ($R.BCNT_{3..0}$) is the number of different sized ring segments. Each of the gate array types has a different ring length, and the ext_ring has yet another length. The length count ($R.LCNT_{7..0}$) is set to the difference between a segment length and the next shorter segment length (except for $R.BCNT_{3..0}$=1, when $R.LCNT_{7..0}$ is set to zero to allow the sate machine to end the fault operation). For example, the first block ($R.BCNT_{3..0}$=12) begins with the longest segment (NIS) and ends with the next shorter segment (NQ). The NIS context ring length is 692, the NQ (which is the next shorter) is 648, so the initial $R.LCNT_{7..0}$ value for the first block is 692 - 648 - 1 = 43. The NIS gate array is scanning during this first block, but the NQ gate array has its clock extend asserted and is not scanning during the first block. The length counter gets loaded each time it reaches zero, at which time the block counter is decremented. Each time this happens, the next shorter gate array (or the ext_ring) has its clock extend de-asserted; until at the end, all gate arrays and the ext_ring are scanning.

The state machine is normally in the idle state (state 0) until $SP_VP.CNTX_CTL_{1..0}$ is set to 0x1 (VCX_SAVE) which indicates that a fault is starting. It then goes to state 7. PAL 5221 decodes state 7 and sets the block count ($BCNT_{3..0}$) to 12, which is the number of different size ring segments (see Figure 13-1 on page 13-2). Clock extends to the context ring registers are asserted during state 7. The state machine proceeds unconditionally to state 6.

In state 6, PAL 5220 and 5227 cause the length counter ($LCNT_{7..0}$) to be loaded, which is based on the current $R.BCNT_{3..0}$. The state machine proceeds unconditionally to state 5.

The state machine holds in state 5 while $R.LCNT_{7..0}$ is decremented. When $R.LCNT_{7..0}$ reaches zero *R.LDONE* is asserted. When *R.LDONE* is asserted, the state machine goes to state 4.

In state 4, the decision is made as to whether all of the context data has been scanned out. If $R.BCNT_{3..0}$ reaches 0x1, then *R.BDONE* is asserted. This does not happen until after the state machine exits state 4, so $R.BCNT_{3..0}$ is actually equal to zero in state four when *R.BDONE* is asserted. *R.BDONE*=1 indicates that all of the context data has been scanned out. If this is the case, the machine proceeds to state 3. Otherwise, the state machine goes back to state 6 to continue storing context data.

State 3 is a delay state that allows a cycle for the context control signals to change. The state machine proceeds unconditionally to state 2.

In state 2, the context controller causes the context control signals to the gate arrays (e.g.,

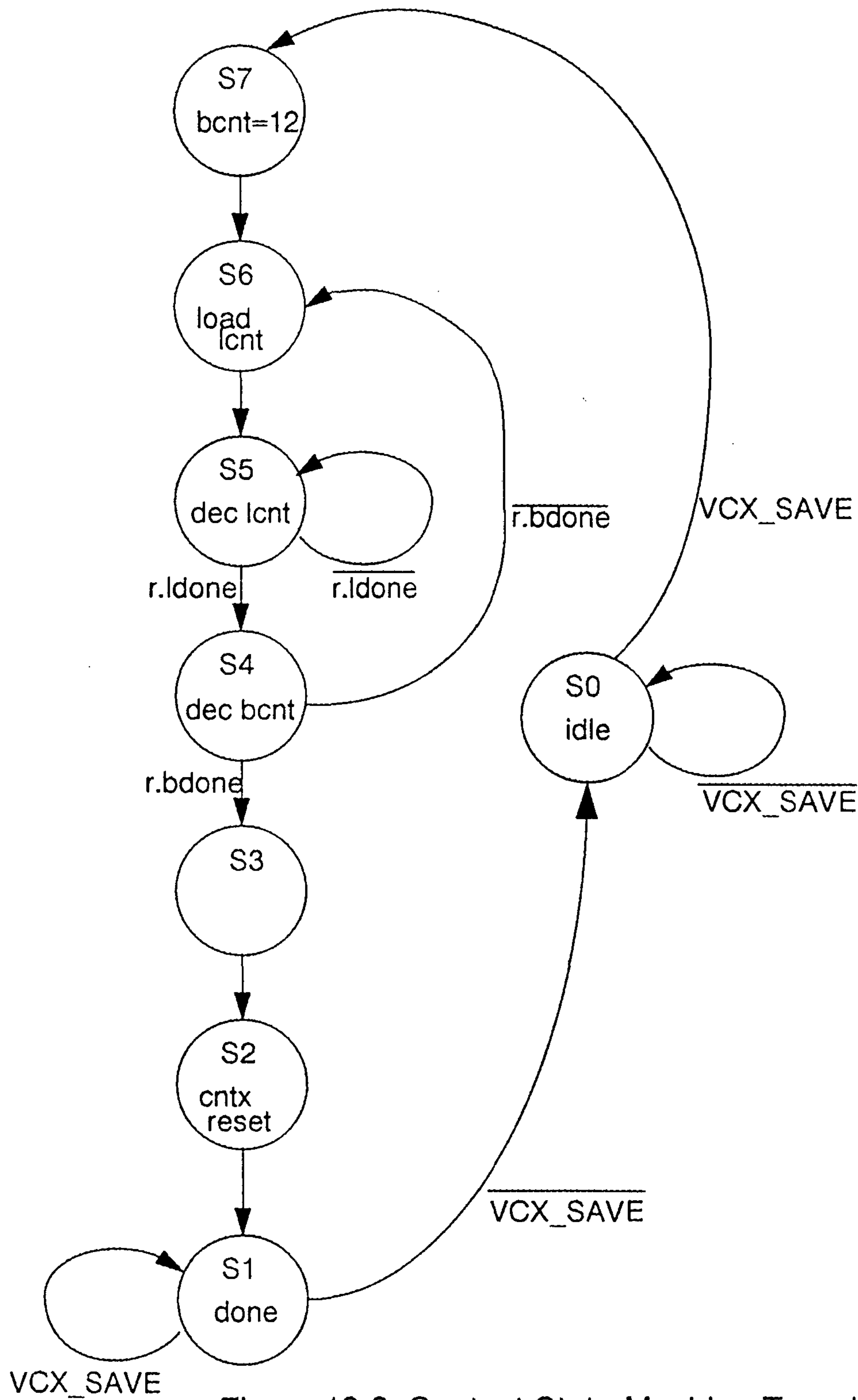


Figure 13-2 Context State Machine Transition Diagram

NVD_CNTX_CTL$_{1..0}$) to be set to 0x2 (CNTX_RESET). The gate arrays reset any of their internal state that is required for them to be put in an idle state. The state machine proceeds unconditionally to state 1.

State 1 is the "done" state of the context controller state machine. In this state the NVP has finished sending all of its context data to memory, and has returned the NVP to an idle state. The state machine waits in state 1 for the NSP to change *SP_VP.CNTX_CTL*$_{1..0}$ from 0x1 (VCX_SAVE) to 0x0 (VCX_NORMAL), at which point it proceeds to state 0, the idle state.

13.2.2 Control signals from the state machine

The state machine generates a number of control signals that are used to communicate with the NSP (via the OSCTL controller) and to manipulate the scan rings on the NVP.

Several control signals are used to control the interface with the NSP during faults. They are sent through the OSCTL controller to manage the VXM type transfers. The context data being sent as VXM data operates on the REQ_NEXT/RDY format used by the OSCTL controller and shown in Figure 10-1 on page 10-2. *SAVE_RDY* is sent through the OSCTL and on to the NSP as *VP_SP.MXV_RDY*, indicating that a piece of context data is available. The NSP indicates that it is ready to accept a piece of the context data on the next cycle by asserting *SP_VP.VX_REQ_NEXT*. If this is not asserted, the context controller can cause clock extends by asserting *CNTX_CK_HOLD* to the BD controller. Similarly, *CSM_CK_HOLD* is asserted under similar circumstances, and during rtnc when *SP_VP.MXV_RDY* is not asserted which indicates that no context restore data is available for the NVP.

It should be noted here, that for *R.BCNT*$_{3..0}$=0x1, that *R.LCNT*$_{7..0}$ is initialized to 0x0. This means that when the state machine is running with *R.BCNT*$_{3..0}$=0x1, it is in a special state where the machine is still running, but no scanning is going on. The state machine uses this state to do some "cleaning up" at the end of a fault.

SAVE_VM is also sent through the OSCTL and on to the NSP as *VP_SP.VXA_VM<0>*. This signal is used to cause valid VM bits to be asserted during the fault. It is the registered version of *R.SAVE_LAST_NEXT* which is the registered version of *C.SAVE_LAST_NEXT* which is asserted when *R.BCNT*$_{3..0}$ > 0x1.

SAVE_LAST is also sent through the OSCTL and on to the NSP as *VP_SP.VXA_LAST*, which indicates that the last piece of fault data is going to be transferred. *SAVE_LAST* is a registered version of *R.SAVE_LAST_NEXT* which is a registered version of *C.SAVE_LAST_NEXT* which is asserted when *R.BCNT*$_{3..0}$ = 0x1.

CNTX_MODE and its many copies are asserted when *SP_VP.CNTX_CTL*$_{1..0}$ is non-zero, or when *R.WCS_WE* from NVP_CLK page 1 is asserted. It indicates that the NVP is running in context mode, or that the NVP is in control store write mode.

CNTX_SAVE is asserted when *SP_VP.CNTX_CTL*$_{1..0}$ =0x2, which indicates that a fault is occurring on the NVP.

CNTX_RESTORE is asserted when *SP_VP.CNTX_CTL*$_{1..0}$ =0x3, which indicates that a rtnc is occurring on the NVP.

13.3 Hardware implementation for rtnc

The logic on the context control pages of NVP_CLK participates in context restores (rtnc), but not in the same way that it does in faults. During faults, the state machine runs to manage the accumulation of context data so that it is properly aligned, and manages the interface to the NSP. During rtnc, the state machine is idle, and the return data is already properly aligned since it was sent out that way. The logic in PAL 5222 and a 100e142 register still manage the interface to the NSP.

Since the context restore data that comes back to the NVP from memory during a rtnc is already properly aligned, the only thing that needs to be done during a rtnc is to send the data to the correct destination, and make sure that no bubbles are allowed in the restore data when the NSP misses a handshake.

Context restore data comes across the $SP_VP.DATA_{31..0}$ bus, through the NIS gate arrays and out onto the $SXV_DAT_{31..0}$ bus. The IS_MUX receives $SXV_DAT_{31..0}$ and sends it to the scan inputs of each of the gate arrays and the ext_ring. The data is organized in the same fashion as it was sent out, as shown in Figure 13-1 on page 13-2.

The context restore data coming to the NVP follows the usual REQ_NEXT/RDY format. The handshakes are *SP_VP.MXV_RDY* and *VP_SP.MXV_REQ_NEXT*. *VP_SP.MXV_REQ_NEXT* is asserted at all times during a rtnc. Thus, the NVP will take data on each cycle on which *SP_VP.MXV_RDY* is asserted. When *SP_VP.MXV_RDY* is not asserted and there is no restore data available, the context controller asserts *C.CSM_CK_HOLD* which is registered and sent to the BD controller, causing clock extends to be asserted. Therefore, the context restore only proceeds when data is available for it.

13.4 Generation of Scan and Context controls (page 4)

NVP_CLK page 4 has four PALs that generate the context control signals for the gate arrays and the scan control signals for the eclips registers. Another PAL (5228) generates the scan control signals for all of the gate arrays. Two 100e111 buffers fan out the gate array scan control signals.

PALs 5224, 5225 and 5264 generate the gate array context controls (e.g., $NVD_CNTX_CTL_{1..0}$) based on state from the context controller and $XC_VP.SCAN_CTL_{2..0}$. The scan control signals that come from the XCL board and their function on the NVP are listed in Table 13-2 on page 13-6.

Table 13-2 NVP Scan Control Inputs

XC_VP.SCAN_CTL	Scan Function
0	Normal Mode
1	- unused
2	- unused
3	- unused
4	- unused
5	Last Scan Left
6	Load Mode (CAST)
7	Board Left

PAL 5223 generates multiple copies of *SCAN_LEFT* (for timing purposes) which causes the eclips registers on the ext_ring to shift; both for context shifts and diagnostic shifts. Also generated is *ALL_SCAN_LEFT* (which is identically equal to *XC_VP.SCAN_CTL<0>*), which causes the eclips registers which are not on the context ring to shift. These registers are in the clock generation

logic and the context controller. PAL 5223 also generates *C.EXT_CK_HOLD*, which is a context controller signal that is sent to the BD controller to generate clock extends for the ext_ring during faults.

PAL 5228 creates the scan control signals for all of the gate arrays (e.g., $NVD_SCAN_CTL_{1..0}$). The scan controls are a direct decoding of $XC_VP.SCAN_CTL_{2..0}$. These are identical for all of the gate arrays except the NVRF. The difference is that when $XC_VP.SCAN_CTL_{2..0}$=0x5 (SCAN_LAST_LEFT) all of the gate arrays except NVRF have their scan controls set to 0x3 (SCAN_LEFT) while the NVRF has its scan control set to 0x0 (SCAN_NORMAL). This is because the NVRF scans for one extra cycle after SCAN_LEFT is taken away. The scan controls are buffered and fanned out by a pair of 100e111 buffers.

14 Microcode

There are 4 different microcode files used within the NVP. The VD microcode controls the vector dispatch logic, the UL microcode controls the load pipe controller, the UA microcode controls the add pipe controller, and the UM microcode controls the multiply pipe controller. The UA and UM microcodes are identical and will be referred to in this document as the UA microcode. The purpose of this chapter is to provide enough explanation so that the microcode listings can be read and understood. A definition of the microword is presented, giving the microinstruction field names and defining their encodings. Constructs of the microlanguage are also presented along with descriptions of how they control the hardware.

14.1 Notational Conventions

There are some rules and conventions which apply to all microlanguage constructs. A language construct may be referred to as a *microop*, short for micro-operation. This is the name the CONVEX microassembler gives to the initial definition of the syntax of a microlanguage construct. A microop consists of a name optionally followed by a list of parameters enclosed in parentheses. Parameter identification is by position in the parameter list. For example, a construct with three parameters

FOO(X,Y,Z)

in which the first two are to be unspecified must show the third parameter preceded by two commas to indicate the two missing parameters:

FOO(,,Z)

Commas need not be shown for unspecified parameters following the right-most specified parameter. For example, a four parameter construct with only the third parameter specified could be written in either of these forms:

FOO(,,Z,)
FOO(,,Z)

In the following subsections, all language constructs appear in upper case. Parameters for the construct are lower case.

A shorthand notation is used in the tables which detail the microlanguage constructs. If a construct has multiple parameters (e.g. FOO(X,Y)), rather than listing all possible combinations of all parameters, each parameter is enumerated with a "—" for the other parameters. This indicates that all other values for the other parameters may be substituted. For example, if the FOO(X,Y) construct has potential values of X1 and X2 for the X parameter and Y1, Y2, and Y3 for the Y parameter, the constructs would be listed like this:

FOO(X1,—)
FOO(X2,—)
FOO(—,Y1)
FOO(—,Y2)
FOO(—,Y3)

where the "--"s indicate the presence of another parameter.

14.2 The VD Microinstruction

The VD microcode controls the dispatch logic on the NVP. The entry point for the microcode comes from the scalar processor through the IPCTL logic. The microcode tells the dispatch logic all of the information it needs to decide when an instruction can be dispatched. For example, information such as which pipes the instruction can run on, what resources does it need, and does it run in accelerated mode or at rate_2x. Some of the fields in the VD microword are used by other parts of the NVP, too, such as the entry point for the pipe controller microcode.

The VD microinstruction fields are listed below, in alphabetical order.

Field Name	Width	Pos.	Description
EP	6	45-40	The microcode entry point for the pipe controllers. This signal tells the pipe controllers where to start reading their microcode ram. The actual address will have three zeroes appended to it by the pipe controller.
F@A_DISPATCH	1	6	The current instruction can be dispatched to the Add Pipe. The Add Pipe can execute integer and floating point addition, subtraction, division, square root, comparisons, type conversions, and edit operations.
F@DEMAND_VM_ACC_RD	1	38	For the current instruction the VM must be read in accelerated mode. This is used for reductions under mask, compress operations, and it is used to cause a VM overrun check for the Move Vector-to-Scalar instruction.
F@DEMAND_VM_ACC_WR	1	39	For the current instruction the VM must be written in accelerated mode. This is used for expand operations, and it is used to cause a VM overrun check for the Move Scalar-to-Vector instruction.
F@DISP_VL0	1	51	The current instruction must be dispatched even if the VL = 0.
F@DST_SIZ	2	25-24	The size of the data to be written back into the VRF at the end of the current instruction. The destination size may be different from the source size for operations such as type converts. 00 - Byte 01 - Half-word 10 - Word (or Single-Precision Floating point) 11 - Long-word (or Double-Precision Floating point)

Field	Width	Bits	Description
F@ID_SEL	3	23-21	Tells the VRF which identity element to use for operations which use them. 000 - Integer zero 001 - Integer one 010 - Minimum representable number 011 - Maximum representable number 100 - All ones
F@LOAD_VL	1	50	The current instruction loads the VL register.
F@L_DISPATCH	1	4	The current instruction must be dispatched to the Load Pipe. The Load Pipe executes all load, store, move, and population count operations.
F@L_VM_PARALLEL	1	53	The current instruction will read or write the VM register in the Load Pipe controller in parallel mode. Used on loads, moves and population count operations where the VM is the source or target register.
F@MERGE	1	7	The current instruction is a merge operation.Used by the NVD array to determine when to dispatch a merge instruction.
F@M_DISPATCH	1	5	The current instruction can be dispatched to the Multiply Pipe. The Multiply Pipe can execute integer and floating point addition, subtraction, multiplication, comparisons, type conversions, and edit operations.
F@NO_CHAIN	1	52	UNUSED in current hardware.
F@PARITY	1	0	Parity on the dispatch microword.
F@PIPE_LENGTH	3	10-8	The pipe length of the current instruction. Used for the Load and Multiply pipes.
F@RATE_2X	1	29	The current instruction operates in 2X mode. This means that the operand length is byte, halfword, word, or short precision floating point. The dispatch logic must make sure that this instruction won't overrun a previous instruction or a following instruction.
F@REQ_SXV	1	48	The current instruction needs a scalar-to-vector transfer. The dispatch logic uses this signal to make sure that only one instruction is using the SXV transfer resource at a time.
F@REQ_SXV_POP	1	47	The current instruction is using the SXV transfer bus and the dispatch logic will accept the data from the

			bus.
F@REQ_SXV_WAIT	1	46	The current instruction is using the SXV transfer bus and the dispatch logic should wait until the first piece of data is available (i.e. SXV_DVAL = 1) before dispatching the instruction.
F@REQ_VM_ACC	1	37	The current instruction can run in accelerated mode. The dispatch logic has to check to make sure that there is no overrun condition before dispatching the instruction in accelerated mode. It will dispatch the instruction in non-accelerated mode if it passes all of the checks except the accelerated check.
F@REQ_VM_RD	1	36	The current instruction needs to read the VM. The dispatch logic must make sure that parallel and serial reads of the VM are not occurring at the same time and that the current instruction will chain properly with an executing instruction that is writing to the VM.
F@REQ_VM_WR	1	35	The current instruction needs to write the VM. The dispatch logic must perform VM write port checks before dispatching this instruction. Both the Multiply Pipe and Add Pipe can serially write the VM and the Load Pipe can parallel write it. The VM must be allocated properly so that the order of writes is preserved.
F@REQ_VRI	1	3	The current instruction requires the use of vector register read port VRI. In some instructions the order of VRI and VRJ doesn't matter and in others it is very important. The VRI dispatch port is translated to the VRX read port at dispatch.
F@REQ_VRJ	1	2	The current instruction requires the use of vector register read port VRJ. In some instructions the order of VRI and VRJ doesn't matter and in others it is very important. The VRJ dispatch port is translated to the VRY read port at dispatch.
F@REQ_VRK	1	1	The current instruction requires the use of vector register K which is a symbol for the vector register write port. The dispatch logic must block dispatch of the current instruction if the front-door for an executing instruction is using the write port or if a back-door is using it unless the current instruction has a pipe length greater than or equal to the instruction using the write port. The VRK dispatch port is translated to the VRZ write port at dispatch.

Field	Width	Bits	Description
F@REQ_VXS	1	49	The current instruction needs a vector-to-scalar transfer. The dispatch logic uses this signal to make sure that only one instruction is using the VXS transfer resource at a time.
F@RI_SEL	2	32-31	This signal is used to map the input register address to the VRI read port in the dispatch logic. The value of this field tells the dispatch logic which register address should be sent to the VRI(X) read port. 00 - Register I 01 - Register J 10 - Register K
F@RJ_SEL	1	30	This signal is used to map the input register address to the VRJ read port in the dispatch logic. The value of this field tells the dispatch logic which register address should be sent to the VRJ(Y) read port. 00 - Register J 01 - Register K
F@SPARE0	1	54	UNUSED.
F@SPARE1	1	55	UNUSED.
F@SRC_SIZ	3	28-26	The size of the data operands for the current instruction. The source size may be different from the destination size for operations such as type converts. 000 - Byte 001 - Half-word 010 - Word 011 - Long-word 110 - Single Precision 111 - Double Precision
F@VD_VM_POL	1	34	The polarity of VM bits for the current instruction. This bit is sent on to the pipe controllers when the instruction is dispatched. 0 - False 1 - True
F@VM_FORCE	1	33	The current instruction is forced to not run under-mask. This signal is sent on to the pipe controller when the instruction is dispatched.
FU	2	20-19	These bits tell the function unit what kind of operation to perform. They are passed from the dispatch logic through the pipe controllers to the

			function units. 00 - Floating point Add 01 - Misc. 10 - Floating point divide if Add Pipe Floating point multiply if Multiply Pipe 11 - Unused
OP	8	18-11	This is the opcode for the function units. It is passed from the dispatch logic to the pipe controllers to the function unit controllers. See the gate array specs for the function units for the exact decode of what each unit will do with the different opcodes.

14.3 The VD Microlanguage

14.3.1 The VL0 Constructs

There are two constructs VL0 and NO_VL0 which tell the dispatch logic whether an instruction can be dispatched with a VL of zero. They set the microsignal F@DISP_VL0 and are shown in Table 14-1. The default in the dispatch logic is to not dispatch an instruction when VL = 0.

Table 14-1 VL0 Constructs

construct	encodings
VL0	F@DISP_VL0 = 1
NO_VL0	F@DISP_VL0 = 0

14.3.2 The Register Allocation Constructs

These constructs are used to set the microsignals F@REQ_VRI, F@REQ_VRJ, and F@REQ_VRK. The signals are used to tell the dispatch logic which VRF read and write ports are needed by the instruction. The constructs are shown in Table 14-2.

Table 14-2 Register Allocation Constructs

construct	encodings
REG_I	F@REQ_VRI = 1
REG_J	F@REQ_VRJ = 1
REG_K	F@REQ_VRK = 1
REG_IJ	F@REQ_VRI = 1, F@REQ_VRJ = 1
REG_IK	F@REQ_VRI = 1, F@REQ_VRK = 1
REG_JK	F@REQ_VRJ = 1, F@REQ_VRK = 1

14.3.5 The Scalar Function Constructs

These constructs control scalar-to-vector and vector-to-scalar dispatch checks. They tell the dispatch logic that an instruction needs one of these interfaces, whether to wait for the first piece of data to arrive before dispatching, and whether the dispatch logic should take the first piece of data. The constructs are shown in Table 14-5.

Table 14-5 VD Scalar Function Constructs

construct	encodings
SXV	F@REQ_SXV = 1, F@REQ_SXV_WAIT = 1
VD_SXV	F@REQ_SXV = 1, F@REQ_SXV_WAIT = 1, F@REQ_SXV_POP = 1
VXS	F@REQ_VXS = 1

14.3.6 The Pipe Length Constructs

These constructs simply assign the pipe length of the given instruction to the microsignal F@PIPE_LENGTH. They are shown in Table 14-6.

Table 14-6 VD Pipe Length Constructs

construct	encodings
PL2	F@PIPE_LENGTH = 2
PL3	F@PIPE_LENGTH = 3
PL4	F@PIPE_LENGTH = 4
PL5	F@PIPE_LENGTH = 5
PL6	F@PIPE_LENGTH = 6
PL7	F@PIPE_LENGTH = 7

14.3.7 The Rate_2X Construct

The **R2X** construct simply assigns a 1 to F@RATE_2X.

14.3.8 The Function Pipe Constructs

These constructs do two main things. They tell the dispatch logic which function pipes the instruction can be dispatched to and they tell the function units what type of operation to do and what kind of data will be provided. They are shown in Table 14-7.

REG_IJK	F@REQ_VRI = 1, F@REQ_VRJ = 1, F@REQ_VRK = 1

14.3.3 The Register Mapping Constructs

These constructs assign values to the signals F@RI_SEL and F@RJ_SEL which control muxes on the NVP. These muxes take input register addresses from the NSP and direct them to the appropriate VRF read port. There are 3 possible addresses which can come in I, J, and K. These can be mapped to either I or J in the NVP. These constructs are shown in Table 14-3.

Table 14-3 Register Mapping Constructs

construct	encodings
I_TO_I	F@RI_SEL = 0
J_TO_I	F@RI_SEL = 1
K_TO_I	F@RI_SEL = 2
J_TO_J	F@RJ_SEL = 0
K_TO_J	F@RJ_SEL = 1

14.3.4 The VM Constructs

These constructs control the dispatch microsignals that have to do with the VM register. They tell the dispatch logic whether the instruction needs to read or write the VM register, whether the instruction needs to run in accelerated mode, the polarity of the VM bits for under mask operations, and whether the instruction should be forced to not run under mask. The constructs are shown in Table 14-4.

Table 14-4 VM Constructs

construct	encodings
NOMASK	F@VM_FORCE = 1
MASK	F@VM_FORCE = 0, F@REQ_VM_RD = 1
EDIT	F@VM_FORCE = 1, F@REQ_VM_RD = 1
VM_ACC	F@REQ_VM_ACC = 1
VMR	F@REQ_VM_RD = 1
VMW	F@REQ_VM_WR = 1
VM_TRUE	F@VM_POL = 1
VM_FALSE	F@VM_POL = 0

Table 14-7 VD Function Pipe Constructs

construct	encodings
FADD(datatype)	F@A_DISPATCH = 1 F@M_DISPATCH = 1 FU = 0H F@DST_SIZE = datatype F@SRC_SIZE = datatype
MISC(datatype)	F@A_DISPATCH = 1 F@M_DISPATCH = 1 FU = 1H F@DST_SIZE = datatype F@SRC_SIZE = datatype
CMP(datatype)	F@A_DISPATCH = 1 F@M_DISPATCH = 1 FU = 1H F@DISP_VL0 = 1 F@DST_SIZE = datatype F@SRC_SIZE = datatype
CVT(sdatatype,ddatatype)	F@A_DISPATCH = 1 F@M_DISPATCH = 1 FU = 1H F@DST_SIZE = ddatatype F@SRC_SIZE = sdatatype
FADD_RED(datatype)	F@A_DISPATCH = 1 F@M_DISPATCH = 1 FU = 0H F@DISP_VL0 = 1 F@RI_SEL = 2 F@REQ_VRI = 1 F@REQ_SXV = 1 F@REQ_SXV_WAIT = 1 F@REQ_SXV_POP = 1 F@REQ_VXS = 1 F@DST_SIZE = datatype F@SRC_SIZE = datatype

MISC_RED(datatype)	F@A_DISPATCH = 1 F@M_DISPATCH = 1 FU = 1H F@DISP_VL0 = 1 F@RI_SEL = 2 F@REQ_VRI = 1 F@REQ_SXV = 1 F@REQ_SXV_WAIT = 1 F@REQ_SXV_POP = 1 F@REQ_VXS = 1 F@DST_SIZE = datatype F@SRC_SIZE = datatype
PROD(datatype)	F@M_DISPATCH = 1 FU = 2H F@DISP_VL0 = 1 F@RI_SEL = 2 F@REQ_VRI = 1 F@REQ_SXV = 1 F@REQ_SXV_WAIT = 1 F@REQ_SXV_POP = 1 F@REQ_VXS = 1 F@DST_SIZE = datatype F@SRC_SIZE = datatype
MUL(datatype)	F@M_DISPATCH = 1 FU = 2H F@DST_SIZE = datatype F@SRC_SIZE = datatype
DIV(datatype)	F@A_DISPATCH = 1 FU = 2H F@DST_SIZE = datatype F@SRC_SIZE = datatype
LD(datatype)	F@L_DISPATCH = 1 F@DST_SIZE = datatype
ST(datatype)	F@L_DISPATCH = 1 F@REQ_VXS = 1 F@SRC_SIZE = datatype

14.3.9 The Parity Construct

The ENDLINE construct simply assigns parity to the F@PARITY signal. Its name is symbolic of the end of the line in the microcode. There really is no format for the construct, parity is simply assigned to the signal.

14.4 VD Examples

Since the VD microcode has a single entry for each instruction that enters the vector processor it is easier to simply read the microcode and look at the constructs in this document than it is to try and explain a number of examples. Given this, only one example will be shown for the VD microcode.

14.4.1 ADD.D_V

The microcode for this instruction reads as follows:

FADD(DP) REG_IJK NOMASK PL2 OP=020H EP=UA_VOPV

The **FADD** construct tells the dispatch logic that it can dispatch to the add or multiply pipe by setting F@A_DISPATCH and F@M_DISPATCH, that the source and destination are double precision floating point numbers by setting F@DST_SIZ and F@SRC_SIZ, and that the instruction is to run on the NFAD by setting FU. **REG_IJK** tells the dispatch logic that the instruction will need all three VRF ports by setting F@REQ_VRI, F@REQ_VRJ and F@REQ_VRK. **NOMASK** sets F@VM_FORCE so the instruction will not run under mask. **PL2** sets F@PIPE_LENGTH to 2. OP is set to floating point add by the **OP** construct. The **EP** construct sets the entry point for the pipe controller microcode to UA_VOPV.

14.5 The UA Microword

The UA and UM microwords are identical so for the remainder of this document only the UA will be discussed.

The UA microcode controls the Add Pipe pipe controller which is comprised of a portion of the A_VM gate array and a small amount of discrete logic. The entry point for the microcode is passed to the A_VM by the VD microcode. The actual address has 3 zeros appended on the least significant 3 digits allowing for 8 addresses per entry point. The fields of the microword contain information used by the controller to control the X, Y, Z, and VM counts used by the instruction, to control the output muxes on the VRF for determining what data enters the Add Function Unit, to determine when the end of an instruction has arrived, and to control the address of the next microinstruction.

The UA microinstruction fields are listed below in alphabetical order.

A_BR_ADDR	10	9-0	This signal tells the microcode where to branch to if the A_BR_INST selects this field as the source of the next microword.
A_BR_INST	3	16-14	This signal tells the microcontroller where to go next.

- 000 - Unconditional branch to A_BR_ADDR
- 001 - Conditional branch to A_BR_ADDR.
- 010 - Unconditional jump to dispatch EP.
- 011 - Conditional jump to dispatch EP.
- 10X - Conditionally loop or next address.
- 11X - Multi-way branch from A_BR_ADDR based on VL.

Field	Width	Bits	Description
A_CNT_CTL	4	26-23	Tells what type of address sequence the AVM should produce for the VRF.

- 0001 - Clear X, Y, Z, and VM counts. (Initialize).
- 0010 - Increment X, Y, Z, and VM counts by 1 (or by 2 if rate_2x). (Normal mode).
- 0011 - Set X, Y, Z, and VM counts to address of next set VM bit. (Accelerate Mode).
- 0100 - Set Z count to VL. Hold X, Y, and VM counts. (Post-write cleanup).
- 0101 - Set X and Z counts to LQ_RSLT_DAT<8..0>. Hold Y and VM counts. (Load/Store vector element).
- 1000 - Increment X count by 1 (or by 2 if rate_2x). Hold Y, Z, and VM counts. (Compress in accelerate mode).
- 1001 - Increment X count. Hold Y, Z, and VM counts. (Compress)
- 1010 - Increment Z and VM counts by 1 (or by 2 if rate_2x). Hold X and Y counts. (Expand in accelerate mode).
- 1011 - Increment X count. Increment Z and VM counts by 1 (or by 2 if rate_2x). Hold Y count. (Expand).
- 1100 - See isp model for explanation of X and Y counts. Increment Z and VM counts. (Merge mode).
- All others - Hold X, Y, Z, and VM counts.

Field	Width	Bits	Description
A_LAST_BD	1	19	Indicates that the last microcode cycle is occurring and that the AVM should generate the last_wr or last_sxv signal.
A_LAST_RD	1	18	Indicates that the last microcode cycle is occurring and that the AVM should generate the last_rd signal.
A_LAST_VXS	1	17	Indicates that the last microcode cycle is occurring and that the AVM should generate the last_vxs signal.
A_OP0_CTL	3	31-29	This signal goes directly to the NVRF gate arrays. It selects what type of data will be placed on the

Field	Width	Bits	Description
			A3_OP1_CTL<2..0> bus. 000 - Integer zero. 001 - Scalar data. 010 - If a2_edit_vm then Y data, otherwise X data. (See NVRF section.) 011 - If a2_edit_vm then scalar data, otherwise X data. (See NVRF section.) 100 - Add pipe result data. 101 - Word-swapped add pipe result data. 110 - Vector Register X data. Masked if necessary. 111 - Identity element.
A_OP1_CTL	3	31-29	This signal goes directly to the NVRF gate arrays. It selects what type of data will be placed on the A3_OP1_CTL<2..0> bus. 000 - Integer zero. 001 - Integer -1. 010 - Scalar data. 011 - Scalar data. Masked if necessary. 100 - Add pipe result data. 101 - Integer -1. 110 - Vector Register Y data. Masked if necessary. 111 - Identity element.
A_PARITY	1	35	Parity on the UA microword.
A_PIPE_CTL	3	22-20	Pipe control codes. 000 - Unused. 001 - Load back door. Done at start of operation. 010 - Start operation. Used for reductions. 011 - Start operation. Used during pipe-down of reductions. 100 - Write to VRF. Used during normal vector operations. 101 - Zero out VM register above VL. Done at end of compares. 110 - Write to VM register. Used during compares. 111 - Do a VXS transfer. Used at end of reductions.
A_SPARE	2	28-27	Unused.
A_TEST_POL	1	13	Selects whether a conditional microcode branch condition is made on the true condition or the false condition in the UA controller.

A_TEST_SEL	3	12-10	Selects which type of condition will be used for a conditional microcode branch on the UA controller. 000 - VL <= 1for rate_1x, VL <= 2 for rate_2x. 001 - VL - VM count <= 2 for rate_1x, VL-VM count <= 4 for rate_2x. 010 - SXV_DVAL asserted 011 - Operation is rate_2x 100 - LQ_RSLT_DAT<0>

14.6 The UA Microlanguage

14.6.1 VRF Operand Select Construct

The **VRF** construct specifies what type of operand will be placed on the A3_OP0_DAT bus and A3_OP1_DAT bus by the NVRF arrays. The construct assigns values to the A_OP0_CTL and A_OP1_CTL microsignals which in turn control muxes on the output of the NVRF arrays. The format for the **VRF** construct is shown in Table 14-8.

Table 14-8 UA VRF Operand Select Construct

construct	encodings
VRF(,—)	A_OP0_CTL = 0
VRF(SC,—)	A_OP0_CTL = 1
VRF(MERG_V,—)	A_OP0_CTL = 2
VRF(MERG_S,—)	A_OP0_CTL = 3
VRF(RSLT,—)	A_OP0_CTL = 4
VRF(RSLT_L,—)	A_OP0_CTL = 5
VRF(X,—)	A_OP0_CTL = 6
VRF(ID,—)	A_OP0_CTL = 7
VRF(—)	A_OP1_CTL = 0
VRF(—,SC)	A_OP1_CTL = 2
VRF(—,SC_ID)	A_OP1_CTL = 3
VRF(—,RSLT)	A_OP1_CTL = 4
VRF(—,Y)	A_OP1_CTL = 6
VRF(—,ID)	A_OP1_CTL = 7

14.6.2 Counter Control Construct

The **CNT** construct specifies what the counters on the AVM should do.It assigns the A_CNT_CTL field of the microword. The format of the **CNT** construct is shown in Table 14-9.

Table 14-9 UA Counter Control Construct

construct	encodings
CNT(CLR)	A_CNT_CTL=01H
CNT(INC)	A_CNT_CTL=02H
CNT(SCAN)	A_CNT_CTL=03H
CNT(VL)	A_CNT_CTL=04H
CNT(CPRS_SCAN)	A_CNT_CTL=08H
CNT(CPRS)	A_CNT_CTL=09H
CNT(XPND_SCAN)	A_CNT_CTL=0AH
CNT(XPND)	A_CNT_CTL=0BH
CNT(MERG)	A_CNT_CTL=0CH

14.6.3 Pipe Control Construct

The **PIPE** construct assigns the A_PIPE_CTL signal of the microword which is passed through the control pipe. The format for the **PIPE** construct is shown in Table 14-10.

Table 14-10 UA Pipe Control Construct

construct	encodings
PIPE(LD_BD)	A_PIPE_CTL=01H
PIPE(START)	A_PIPE_CTL=02H
PIPE(FSTART)	A_PIPE_CTL=03H
PIPE(WR_VRF)	A_PIPE_CTL=04H
PIPE(ZERO_VM)	A_PIPE_CTL=05H
PIPE(WR_VM)	A_PIPE_CTL=06H
PIPE(VXS)	A_PIPE_CTL=07H

14.6.4 The Last Element Identification Construct

The **LAST** construct identifies which last signals should be generated by the LVM at the appropriate time. The construct assigns values to all three of the last signals A_LAST_VXS, A_LAST_RD, and A_LAST_BD. The format for the **LAST** construct is shown in Table 14-11.

Table 14-11 UA Last Element Identification Construct

construct	encodings
LAST(RD,—,—)	A_LAST_RD = 1
LAST(VXS,—,—)	A_LAST_VXS = 1
LAST(BD,—,—)	A_LAST_BD = 1;
LAST(—,RD,—)	A_LAST_RD = 1
LAST(—,VXS,—)	A_LAST_VXS = 1
LAST(—,BD,—)	A_LAST_BD = 1
LAST(—,—,RD)	A_LAST_RD = 1
LAST(—,—,VXS)	A_LAST_VXS = 1
LAST(—,—,BD)	A_LAST_BD = 1

14.6.5 The Test Condition Construct

The **TEST** construct is used to determine what test condition should be used when one is needed. The construct sets two signals A_TEST_SEL and A_TEST_POL.The format of the **TEST** construct is shown in Table 14-12.

Table 14-12 UA Test Condition Construct

construct	encodings
TEST(VL_LE_1)	A_TEST_SEL = 0, A_TEST_POL = 0
TEST(DF_LE_2)	A_TEST_SEL = 1, A_TEST_POL = 0
TEST(RATE_2X)	A_TEST_SEL = 3, A_TEST_POL = 0
TEST(VL_LE_1@)	A_TEST_SEL = 0, A_TEST_POL = 1
TEST(DF_LE_2@)	A_TEST_SEL = 1, A_TEST_POL = 1
TEST(RATE_2X@)	A_TEST_SEL = 3, A_TEST_POL = 1

14.6.6 Branch Constructs

There are numerous constructs used to branch in the UA microcode. They all are used to assign values to the signals A_BR_INST and A_BR_ADDR. They are shown in Table 14-13.

Table 14-13 UA Branch Constructs

construct	encodings

GOTO(addr)	A_BR_INST = 0, A_BR_ADDR = addr
GOTO_IF(addr)	A_BR_INST = 1, A_BR_ADDR = addr
DISP	A_BR_INST = 2
DISP_IF	A_BR_INST = 3, A_BR_ADDR = current microcode address
CASE_TEST(addr)	A_BR_INST = 4, A_BR_ADDR = addr
CASE_VL(addr)	A_BR_INST = 6, A_BR_ADDR = addr
NEXT	same as GOTO(current address + 1)
NEXT_IF	same as GOTO_IF(current address + 1)

14.6.7 The Parity Construct

The ENDLINE construct simply assigns parity to the A_PARITY signal. Its name is symbolic of the end of the line in the microcode. There really is no format for the construct, parity is simply assigned to the signal.

14.7 UA Examples

14.7.1 UA_VOPV

This is the basic operation of vector register source and destination. The microcode for this type of instruction is:

```
CNT(CLR) PIPE(LD_BD) TEST(VL_LE_1) NEXT;

VRF(X,Y) CNT(INC) PIPE(WR_VRF) LAST(RD,BD) TEST(DF_LE_2) DISP_IF;
```

The first microword clears the X, Y and Z counts inside the NVM using the **CNT** construct, sets up the loading of back-door control register in the NVM, NVD, and NVRF arrays using the **PIPE** construct, sets up a test for VL <= 1 using the **TEST** construct and branches to the next line. The test for VL <= 1 is set up in the first microword because it takes a cycle before the test is actually performed.

The second microword performs the operation. The **VRF** construct tells the VRF arrays to put the data from the X and Y read ports on to the operand bus, the **CNT** construct increments by 1 each of the X, Y and Z counters in the NVM which control the element address being read from and written into the VRF, the **PIPE** construct tells the NVRF arrays to write the results to the VRF, the **LAST** construct tells NVM to generate the last read and last back-door signals at the appropriate time, and the **TEST** construct sets the ending condition to be when VL - VM count is less than or equal to 2.

14.7.2 UA_VOPV_ACC

This is the basic accelerated operation with vector registers being the source and destination. The microcode for this type of instruction is:

```
CNT(CLR) NEXT;

CNT(SCAN) NEXT;
```

CNT(SCAN) PIPE(LD_BD) TEST(VL_LE_1) NEXT;

VRF(X,Y) CNT(SCAN) PIPE(WR_VRF) LAST(RD,BD) TEST(DF_LE_2) DISP_IF;

The first microword clears the X, Y and Z counts inside the NVM using the **CNT** construct. This construct also clears the r.acc_start register which indicates the starting position for the next scan of the VM.

The second microword tells the NVM to find the first VM bit(s) for the first operation by switching the **CNT** construct parameter to scan.

The third microword continues with the scanning using the **CNT** construct, sets the register r.lq_bd_pend in the VM using the **PIPE** construct, and sets up a test for VL <= 1 using the **TEST** construct. This is similar to the first word of the previous example other than the counter control.

The final microword performs the operation just like the previous example expect that instead of incrementing the counters the counters are set to the next good VM bit location by the scanner.

14.7.3 UA_CPRS

This is the compress operation. The microcode for this type of instruction is:

CNT(CLR) NEXT;

CNT(CPRS_SCAN) NEXT;

CNT(CPRS_SCAN) PIPE(LD_BD) TEST(VL_LE_1) NEXT;

VRF(X) CNT(CPRS) PIPE(WR_VRF) LAST(RD,BD) TEST(DF_LE_2) DISP_IF;

The first microword clears the X, Y and Z counts inside the NVM using the **CNT** construct. This construct also clears the r.acc_start register which indicates the starting position for the next scan of the VM.

The second microword tells the NVM to find the first VM bit(s) for the first operation by switching the **CNT** construct parameter to CPRS_SCAN. In this mode only the X counter is updated with the address of the VM bits.

The third microword continues with the scanning using the **CNT** construct, sets up the loading of back-door control registers using the **PIPE** construct, and sets up a test for VL <= 1 using the **TEST** construct. Again, only the X counter is updated.

The final microword performs the operation by using the CNT construct to scan and update the X counter and increment the Z counter. Since the compress instruction only uses 1 read port the **VRF** construct only uses the X parameter. The **PIPE**, **LAST**, and **TEST** constructs are used just like the previous two examples.

14.8 The UL Microword

The UL microcode controls the Load/Store pipe controller which is comprised of a portion of the L_VM gate array and a small amount of discrete logic. The entry point for the microcode is passed to the L_VM by the VD microcode. The actual address has 3 zeros appended on the least significant 3 digits allowing for 8 addresses per entry point. The fields of the microword contain

information used by the controller to control the X, Y, Z, and VM counts used by the instruction, to control the output muxes on the VRF for determining what data enters the Load/Store logic, to determine when the end of an instruction has arrived, and to control the address of the next microinstruction. There are fields in the UL microword which make it unique from the other two pipe controller microcodes. These fields control the writing of the VM register and the muxes which determine what data is driven on the VM_DAT bus. This ensures that the three copies of the VM and VS registers are always identical.

The UL microinstruction fields are listed below in alphabetical order.

Field	Width	Bits	Description
L_BR_ADDR	10	9-0	This signal tells the microcode where to branch to if the L_BR_INST selects this field as the source of the next microword.
L_BR_INST	3	16-14	This signal tells the microcontroller where to go next. 000 - Unconditional branch to L_BR_ADDR 001 - Conditional branch to L_BR_ADDR. 010 - Unconditional jump to dispatch EP. 011 - Conditional jump to dispatch EP. 10X - Conditionally loop or next address. 11X - Multi-way branch from L_BR_ADDR based on VL.
L_CNT_CTL	3	26-24	Tells what type of address sequence the LVM should produce for the VRF. 001 - Clear X, Y, Z, and VM counts. (Initialize). 010 - Increment X, Y, Z, and VM counts by 1 or by 2 if rate_2x. (Normal mode). 011 - Set X, Y, Z, and VM counts to address of next set VM bit. (Accelerate Mode). 100 - Set Z count to VL. Hold X, Y, and VM counts. (Post-write cleanup). 101 - Set X and Z counts to LQ_RSLT_DAT<8..0>. Hold Y and VM counts. (Load vector element).
L_DST_CTL	3	29-27	Tells the LVM array what to do with the LQ_RSLT_DAT bus. 011 - Write to VS. 100 - Write to VM<127..96>. 101 - Write to VM<95..64>. 110 - Write to VM<63..32>. 111 - Write to VM<31..0>.
L_LAST_ELEM	1	19	Indicates that the last microcode cycle is occurring and that the LVM should generate the last_elem

signal.

Field	Width	Bits	Description
L_LAST_RD	1	18	Indicates that the last microcode cycle is occurring and that the LVM should generate the last_rd signal.
L_LAST_VXS	1	17	Indicates that the last microcode cycle is occurring and that the LVM should generate the last_vxs signal.
L_LAST_WR	1	20	Indicates that the last microcode cycle is occurring and that the LVM should generate the last_wr signal.
L_OP0_CTL	1	34	Tells the NVRF arrays what to output on the L3_OP0_DAT bus. 0 - VRF X port data. (Normal operation). 1 - VM data. (Used for stores of VM).
L_OP1_CTL	1	33	Tells the NVRF arrays what to output on the L3_OP1_DAT bus. 0 - VRF Y port data. (Normal operation). 1 - Address index. (Used for stores under mask.)
L_PARITY	1	35	Parity on the UL microword.
L_PIPE_CTL	3	23-21	Pipe control codes. 000 - Unused. 001 - Load back door. Done at start of operation. 010 - VXM transfer. (Stores of all kinds). 011 - Unused. 100 - Write to VRF from memory data. (Vector loads). 101 - Write to VRF from scalar data. (Move scalar to vector element). 110 - VXM and write VRF. (Load vector of indices and loads under mask). 111 - VXS transfer. (Store VM. Move vector element to scalar.)
L_SRC_CTL	3	32-30	Tells the L_VM array what to put out on the VM_DAT bus. 000 - Address index. (Load/Store under mask.) 001 - Unused. 010 - VM<127..64>. (Store VM upper.) 011 - VM<63..0>. (Store VM lower.) 100 - SXV<31..0>. (Move scalar to vector element.

			Load VM.) 101 - SXV<63..32>. (Move scalar to vector element. Load VM.)
L_TEST_POL	1	13	Selects whether a conditional microcode branch condition is made on the true condition or the false condition in the UL controller.
L_TEST_SEL	3	12-10	Selects which type of condition will be used for a conditional microcode branch on the UL controller. 000 - VL <= 1for rate_1x, VL <= 2 for rate_2x. 001 - VL - VM count <= 2 for rate_1x, VL-VM count <= 4 for rate_2x. 010 - SXV_DVAL asserted 011 - Operation is rate_2x 100 - LQ_RSLT_DAT<0>

14.9 The UL Microlanguage

14.9.1 VRF Operand Select Construct

The **VRF** construct specifies what type of operand will be placed on the L3_OP0_DAT bus and L3_OP1_DAT bus by the NVRF arrays. The construct assigns values to the L_OP0_CTL and L_OP1_CTL microsignals which in turn control muxes on the output of the NVRF arrays. The format for the **VRF** construct is shown in Table 14-8.

Table 14-14 UL VRF Operand Select Construct

construct	encodings
VRF(,—)	L_OP0_CTL = 0
VRF(X,—)	L_OP0_CTL = 0
VRF(VM,—)	L_OP0_CTL = 1
VRF(—)	L_OP1_CTL = 0
VRF(—,Y)	L_OP1_CTL = 0
VRF(—,IX)	L_OP1_CTL = 1

14.9.2 Counter Control Construct

The **CNT** construct specifies what the counters on the LVM should do.It assigns the L_CNT_CTL field of the microword. The format of the **CNT** construct is shown in Table 14-9.

Table 14-15 UL Counter Control Construct

construct	encodings
CNT(CLR)	L_CNT_CTL = 01H
CNT(INC)	L_CNT_CTL = 02H
CNT(SCAN)	L_CNT_CTL = 03H
CNT(VL)	L_CNT_CTL = 04H
CNT(RSLT)	L_CNT_CTL = 05H

14.9.3 Pipe Control Construct

The **PIPE** construct assigns the L_PIPE_CTL signal of the microword which is passed through the control pipe. The format for the **PIPE** construct is shown in Table 14-10.

Table 14-16 UL Pipe Control Construct

construct	encodings
PIPE(LD_BD)	L_PIPE_CTL=01H
PIPE(VXM)	L_PIPE_CTL=02H
PIPE(WR_VRF_M)	L_PIPE_CTL=04H
PIPE(WR_VRF_S)	L_PIPE_CTL=05H
PIPE(VXM.WR_VRF_M)	L_PIPE_CTL=06H
PIPE(VXS)	L_PIPE_CTL=07H

14.9.4 The Last Element Identification Construct

The **LAST** construct identifies which last signals should be generated by the LVM at the appropriate time. The construct assigns values to all four of the last microsignals L_LAST_VXS, L_LAST_RD, L_LAST_WR, and L_LAST_ELEM. The format for the **LAST** construct is shown in Table 14-11.

Table 14-17 UL Last Element Identification Construct

construct	encodings
LAST(RD,—,—)	L_LAST_RD = 1
LAST(VXS,—,—)	L_LAST_VXS = 1
LAST(WR,—,—)	L_LAST_WR = 1
LAST(ELEM,—,—)	L_LAST_ELEM = 1

LAST(---,RD,---)	L_LAST_RD = 1
LAST(---,VXS,---)	L_LAST_VXS = 1
LAST(---,WR,---)	L_LAST_WR = 1
LAST(---,ELEM,---)	L_LAST_ELEM = 1
LAST(---,---,RD)	L_LAST_RD = 1
LAST(---,---,VXS)	L_LAST_VXS = 1
LAST(---,---,WR)	L_LAST_WR = 1
LAST(---,---,ELEM)	L_LAST_ELEM = 1

14.9.5 The Test Condition Construct

The **TEST** construct is used to determine what test condition should be used when one is needed. The construct sets two signals L_TEST_SEL and L_TEST_POL.The format of the **TEST** construct is shown in Table 14-18.

Table 14-18 UL Test Condition Construct

construct	encodings
TEST(VL_LE_1)	L_TEST_SEL = 0, L_TEST_POL = 0
TEST(DF_LE_2)	L_TEST_SEL = 1, L_TEST_POL = 0
TEST(SXV_DVAL)	L_TEST_SEL = 2, L_TEST_POL = 0
TEST(RATE_2X)	L_TEST_SEL = 3, L_TEST_POL = 0
TEST(RSLT_0)	L_TEST_SEL = 4, L_TEST_POL = 0
TEST(VL_LE_1@)	L_TEST_SEL = 0, L_TEST_POL = 1
TEST(DF_LE_2@)	L_TEST_SEL = 1, L_TEST_POL = 1
TEST(SXV_DVAL@)	L_TEST_SEL = 2, L_TEST_POL = 1
TEST(RATE_2X@)	L_TEST_SEL = 3, L_TEST_POL = 1
TEST(RSLT_0@)	L_TEST_SEL = 4, L_TEST_POL = 1

14.9.6 Branch Constructs

There are numerous constructs used to branch in the UL microcode. They all are used to assign values to the signals L_BR_INST and L_BR_ADDR. They are shown in Table 14-13.

Table 14-19 UL Branch Constructs

construct	encodings
GOTO(addr)	L_BR_INST = 0, L_BR_ADDR = addr
GOTO_IF(addr)	L_BR_INST = 1, L_BR_ADDR = addr

DISP	L_BR_INST = 2
DISP_IF	L_BR_INST = 3, L_BR_ADDR = current microcode address
CASE_TEST(addr)	L_BR_INST = 4, L_BR_ADDR = addr
CASE_VL(addr)	L_BR_INST = 6, L_BR_ADDR = addr
NEXT	same as GOTO(current address + 1)
NEXT_IF	same as GOTO_IF(current address + 1)

14.9.7 The Source Control Construct

The **SRC** construct controls a mux on the three NVM arrays which selects what data will be output on the VM_DAT bus. The actual VM_DAT bus used by the NVP is made up of 32 bits from the AVM(63..32) and 32 bits from the LVM(31..0). The format for the **SRC** construct is shown in Table 14-20.

Table 14-20 UL Source Control Construct

construct	encodings
SRC(IX)	L_SRC_CTL = 0
SRC(VM_01)	L_SRC_CTL = 2
SRC(VM_23)	L_SRC_CTL = 3
SRC(SXV_L)	L_SRC_CTL = 4
SRC(SXV_U)	L_SRC_CTL = 5

14.9.8 The Destination Constructs

The signal L_DST_CTL is used to tell all three of the NVM arrays what to do with the LQ_RSLT_DAT bus. The constructs used to assign values to this signal are shown in Table 14-21.

Table 14-21 UL Destination Constructs

construct	encodings
WR_VS	L_DST_CTL = 3
WR_VM0	L_DST_CTL = 4
WR_VM1	L_DST_CTL = 5
WR_VM2	L_DST_CTL = 6
WR_VM3	L_DST_CTL = 7

14.9.9 The Parity Construct

The ENDLINE construct simply assigns parity to the L_PARITY signal. Its name is symbolic of the end of the line in the microcode. There really is no format for the construct, parity is simply assigned to the signal.

14.10 UL Examples

14.10.1 UL_LD_V

This is the basic load vector operation. The microcode for this operation is:

```
CNT(CLR) PIPE(LD_BD) TEST(VL_LE_1) NEXT;

CNT(INC) PIPE(WR_VRF_M) LAST(ELEM) TEST(DF_LE_2) DISP_IF;
```

The first microword clears the X, Y and Z counts inside the NVM using the **CNT** construct, sets up the loading of back-door control register in the NVM, NVD, and NVRF arrays using the **PIPE** construct, sets up a test for VL <= 1 using the **TEST** construct and branches to the next line. The test for VL <= 1 is set up in the first microword because it takes a cycle before the test is actually performed.

The second microword performs the operation. The **CNT** construct increments by 1 each of the X, Y and Z counters in the NVM which control the element written into the VRF, the **PIPE** construct tells the NVRF arrays to write the results to the VRF from the memory bus, the **LAST** construct tells NVM to generate the last element signal at the appropriate time, and the **TEST** construct sets the ending condition to be when VL - VM count is less than or equal to 2.

14.10.2 UL_LD_V_MASK

This is the basic load vector operation under mask. The microcode for this operation is:

```
CNT(CLR) PIPE(LD_BD) TEST(VL_LE_1) NEXT;

VRF(,IX) CNT(INC) PIPE(VXM.WR_VRF_M) SRC(IX) LAST(RD,VXS,ELEM) TEST(DF_LE_2)
DISP_IF;
```

The first microword operates in the same way as in the first example.

The second microword performs the operation. The **VRF** construct tells the NVRF arrays to put the vector index onto theVP_SP.VXA_ADDR bus to be used by the vector address generator on the NSP, the **CNT** construct increments by 1 each of the X, Y and Z counters in the NVM which control the element written into the VRF, the **PIPE** construct tells the NVRF arrays to write the results to the VRF from the memory bus and to send the address bus to the NSP, the SRC construct tells the NVM array to send the vector index out on the VM_DAT bus, the **LAST** construct tells NVM to generate the last read, last vxs and last element signals at the appropriate time, and the **TEST** construct sets the ending condition to be when VL - VM count is less than or equal to 2.

14.10.3 UL_STVI_V_ACC

This is the accelerated store vector of indices operation. The microcode for this operation is:

```
CNT(CLR) NEXT;

CNT(SCAN) NEXT;

CNT(SCAN) TEST(VL_LE_1) NEXT;

VRF(,Y) CNT(SCAN) PIPE(VXM) LAST(RD,VXS,ELEM) TEST(DF_LE_2) DISP_IF;
```

The first microword clears the X, Y and Z counts inside the NVM using the **CNT** construct. This also clears the r.acc_start register which is used in the VM accelerator.

The second microword tells the NVM to find the first active VM bit(s) for the first store by switching the **CNT** construct parameter to SCAN.

The third microword continues by scanning for the next VM bit(s) using the **CNT** construct and uses the **TEST** construct to test for VL less than or equal to 1.

The final microword performs the operation. The **VRF** construct tells the NVRF arrays to put the read port Y data onto theVP_SP.VXA_ADDR bus to be used by the vector address generator on the NSP, the **CNT** construct scans for the next VM bit(s) and sets the Y counter in the NVM which controls the element being read from the VRF, the **PIPE** construct tells the NVRF arrays to send the address bus to the NSP, the **LAST** construct tells NVM to generate the last read, last vxs and last element signals at the appropriate time, and the **TEST** construct sets the ending condition to be when VL - VM count is less than or equal to 2.

14.10.4 UL_MOV_S_V

This is the move scalar-to-vector operation. The microcode for this operation is:

```
PIPE(LD_BD) SRC(SXV_L) CNT(RSLT) NEXT;

PIPE(WR_VRF_S) LAST(ELEM) DISP;
```

The first microword controls the load of the address into the NVM. The back-door control registers are loaded with the **PIPE** construct, the input staging logic puts the element address onto the LQ_RSLT_DAT because of the **SRC** construct and the element address is loaded into the Z counter on the NVM with the **CNT** construct.

The second microword controls the actual loading of the data. The PIPE construct tells the NVRFs to write the VRF from the scalar data bus and the LAST construct tells the NVM to generate the last element signal.

Note that the NVD will not dispatch this instruction to the rest of the NVP until the element address is available. The NVP will clock extend from that point until the scalar data arrives due to the back-door logic.

15 Inside NVD

15.1 Overview

This section is intended to provide a more detailed look at the gory details of the vector dispatch checks. The reason that this is not included in the main dispatch section of the text is that it goes to levels of detail that are beyond the scope of the rest of the document. The reason that it is included at all is to assist persons who really need to know the excruciating details at some future date, since the author is in the process of forgetting them himself.

15.2 Caveats

As always, for an exact detailed description, one should turn to the isp model. Since vector dispatch is a control nightmare and the functionality may not always be obvious from the isp model, this section of the specification will seek to explain what types of resource checks are made in the dispatch logic. Use will be made of some drawings which **may not be entirely accurate** due to the fact that they were made before the logic was designed rather than afterwards and have not been updated. Also, they are ugly because they were captured from GED. The drawings are used here to help visualize the function of the logic, not to be an exact representation. Caveat Emptor.

15.3 Chaining Check

The chaining check insures that data is available for a second instruction which needs data from a previous instruction, or "chains" with a previous instruction. For example, if we have the code sequence

```
add.lv0,v1,v2
mul.lv2,v3,v4
```

We see that the *add.l* writes to v2 and that the *mul.l* reads data out of v2. Clearly, the *mul* cannot start until the *add* has started writing its data into v2. In fact, the *mul* can start reading data from v2 as soon as the *add* writes the first element into v2.

A diagram of the chaining check logic is shown in Figure 1-1 on page 15-2. There are eight set-reset registers: one for each of the vector registers. A register gets set when an instruction is dispatched which writes into a vector register. The VRK<2:0> field selects the register to be set. The registers are cleared when an instruction makes its first write into its destination vector register. The xBD_VRK<2:0> field selects the register to be cleared for each of the three pipes. The xBD_WR_ACTIVE and xQ_FIRST_WR signals indicate that the first write is taking place.

The actual dispatch check takes place when an instruction requests read registers (either VRI or VRJ). The VRI<2:0> and VRJ<2:0> fields select multiplexers which pass through the state of the set-reset register corresponding to the selected vector register. If the set-reset register is set and the instruction is requesting that vector register, then there is a chaining hazard.

These vector register chaining hazards are combined with chaining hazards for the VM register (described later) to create the overall chaining hazard condition.

15.4 Read Port Allocation and Hazard Check

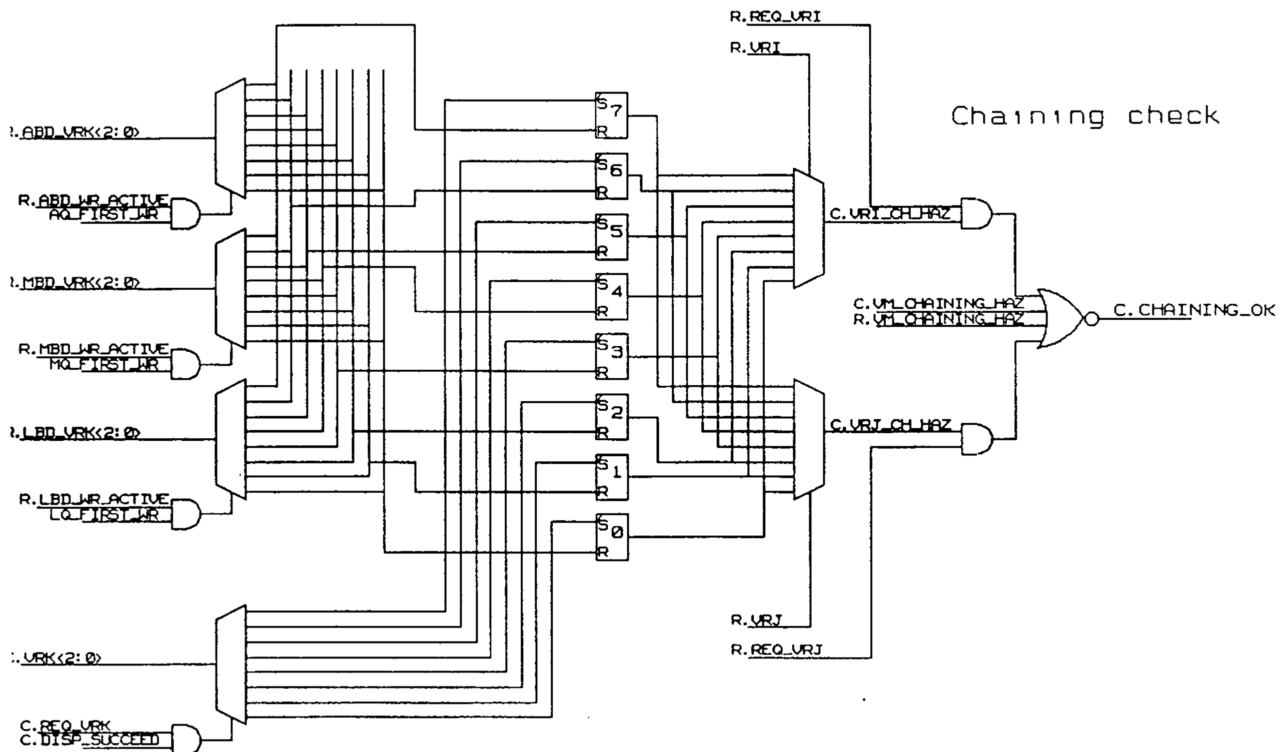


Figure 1-1 Chaining Check

Read port allocation and hazard checking is performed in order to insure that no read port resource is over-allocated. The C38xx vector processor has four vector register banks with two read ports per bank. With three function pipes it is clearly possible to require more read ports into a vector register bank that actually exist; hence this check. A diagram of the read port allocation and hazard check logic is shown in Figure 1-2 on page 15-3.

Read port allocation is performed by taking the requested read register fields VRI<2:0> and VRF<2:0> and stripping off the MSB since the register banks are stacked (i.e., v0 and v4 use the same read ports). The stripped register fields VRI<1:0> and VRJ<1:0> are compared against the already allocated register ports R.xFP_VRx<3,1:0> (bit 3 indicating which of the two read ports was allocated). The default is for the A read port to be taken by an instruction. If the A read port is taken, then the B port is selected for the instruction. If both ports are already taken, then a hazard exists and the instruction is not dispatched until the hazard is cleared.

For example, let C.VRI<2:0>=7 and R.AFP_VRI<3,1:0>=3 and R.MFP_VRJ<3,1:0>=7. Then read port 3A is taken by R.AFP_VRI which causes C.A_PORT_TAKEN_I to be asserted. Read port 3B is taken by R.MFP_VRJ which causes C.B_PORT_TAKEN_I to be asserted. These two together cause C.BLOCKED_ON_VRI to be asserted which causes C.READ_PORTS_OK to go away, indicating a hazard. Now, if M_ACTIVE goes away, then port 3B is no longer taken, C.B_PORT_TAKEN_I goes away causing C.BLOCKED_ON_VRI to go away allowing C.READ_PORTS_OK to be asserted. The C.A_PORT_TAKEN_I is equivalent to the C.VRX_PORT signal which indicates that the dispatching instruction is to use the 3B read port.

Figure 1-2 Read Port Allocation and Hazard Check

This signal also becomes bit 3 of the VRI<3:0> signal which is fed back into the read port reservation registers R.xFP_VRx<3:0>.

When an instruction is dispatched, the VRI<3:0> and VRJ<3:0> signals are registered to indicate the reservation of read ports. Another reservation signal R.x_RES_VRx is also registered at dispatch that indicates that the port is actually being used, since not all instructions use two read ports.

Another hazard that can occur is if one read port is available on a bank and an instruction requests both ports. This hazard is checked by comparing the two full read addresses to see if the request is on the same register, or just on the same register bank. If the two requests are on the same bank, then there is a hazard. If they are on the exact same register (except for a *merge* instruction), then the dispatch logic assigns both read requests to the same port since the read port addresses will be identical at all times.

This leads us to the discussion of an unusual feature not mentioned so far: read register duplication override. If we have an instruction such as *or v0,v0,v1* which wants to read from the same register twice, the dispatch logic will only allocate one vector register read port to this instruction and the VRFs will be instructed to read both operand buses from the same read port. The *merge vi,vj,vk* instruction is an exception to this since its data from vi and vj are read at different rates and one read port is not adequate.

15.5 Write Port Allocation and Hazard Check

Vector register write port allocation and hazard checking is performed in order to insure that no write port resource is over-allocated. The C38xx vector processor has four vector register banks with one write per bank. More than one pipe may request to write to the same register or register bank at the same time, requiring this allocation and checking. A diagram of the write port allocation and hazard check logic is shown in Figure 1-3 on page 15-6.

Since there is only one possible write port for a particular instruction to take, allocation does not amount to anything more than saving the vk field of an instruction after an instruction is dispatched. The register field is saved in the register R.xFP_VRK<2:0> (although only bits 1:0 are relevant due to register stacking). A set-reset register R.x_RES_VRK is also activated to indicate that the register is actually being used. This register is cleared when the operation reaches the back-door level as indicated by xQ_LD_BD. At this point the register field is moved into the R.xBD_VRK<2:0> register. A set-reset register R.xBD_WR_ACTIVE is set to indicate that the register is being used at the back-door level.

The write port hazard checking is broken up into two pieces: one for the front-door and one for the back door. (If you recall, a pipe can be dispatched an instruction and start the micro-controller with that new instruction whilst a previous instruction is finishing up in the back-door of the same pipe.) The front-door check is used while the micro-controller for a pipe is still running. The back-door check is used when the micro-controller for a pipe has finished running an instruction, but write controls are still in the queue for that instruction.

The front-door write port hazard check amounts to comparing the two LSBs of the requested write register VRK<1:0> against the ports already allocated to the function pipes which are currently in use. This creates the C.x_WRITING signals, any of which indicates a hazard on the requested write port.

The back-door write port hazard check involves the same type of register number comparison

used in the front-door check except that the comparison address is now R.xBD_VRK<1:0> instead of R.xFP_VRF<1:0>. In addition, the pipe-length of the instruction in dispatch is compared against the pipe-length of the instruction running on the back door of each pipe. This allows an instruction to be allocated a write port that is already in use IF the instruction that is currently running in the back-door will complete before the dispatching instruction reaches the back-door level. This occurs if the dispatching pipe-length is greater than the pipe-length of the instruction that is currently running in the back-door.

15.6 Function Pipe Check

The function pipe hazard check is used to insure that there will be no conflict at any level of the function pipe pipeline. One way to do this is to not start an instruction on a pipe until the previous instruction is fully finished. This is wasteful, however, and the function pipe check allows two instructions to be running on a pipe provided that the second will not overtake the first in the pipeline. This check is only used for the add and multiply pipes since the load pipe uses a variable-depth queue. A diagram of the function pipe hazard logic is shown in Figure 1-4 on page 15-7.

This check assumes that no new instruction can start on a function pipe until the micro-controller goes idle from the previous instruction. This is true since vector dispatch will not dispatch an instruction to a pipe until the x_RDY signal is asserted. Once the micro-controller for the previous instruction goes idle, the number operations left in the back-door of the function pipe is equal to the pipe-length for that instruction. One implication of this is that the previous instruction has loaded a number of control entries into the control queue that is equal to its pipe length. If the new instruction has a pipe-length that is shorter than the previous instruction, it will insert its control entries into the control queue in front of the entries for the previous instruction. That would be bad. A similar problem is that the vector register or VM write port is tied up for a number of cycles equal to the pipe length of the previous instruction, but the new instruction wants to use the write port in a number of cycles equal to its pipe length. If the new one is shorter than the old one then there is a conflict for the write port.

So, the function pipe check consists of insuring that the new instruction has a pipe length that is longer than what is left in the back door of the function pipe by the previous instruction. This is done by loading a register with the pipe length of an instruction when the instruction starts. As long as the instruction is running in the front door, the pipe length keeps getting loaded into the register. When the instruction finishes in the front_door (as indicated by the absence of x_ACTIVE), the pipe length register starts getting decremented on every cycle (until it is zero). This register is compared with the pipe length of the instruction in dispatch. If the registered value is greater than the new pipe length there is a hazard.

15.7 Read Overruns Write Check

In general, the C38xx vector processor runs in lock-step so that there is no danger of things happening out of order. There are a couple of cases where things do not happen quite this way. One of them is rate_2x which runs faster than the normal rate. Another is in the VM accelerate mode (VMA) which has the ability to run at an indeterminately faster rate than normal. One possibility is that a rate_2x or VMA instruction can be chained to a normal instruction and attempt to read operands faster than they are being produced by the normal instruction. This constitutes a read-overruns-write (ROW) hazard. A diagram of the vector register read-overruns-write hazard checking is shown in Figure 1-5 on page 15-8.

The ROW check is composed of checks for both rate_2x hazards and VMA hazards, which share

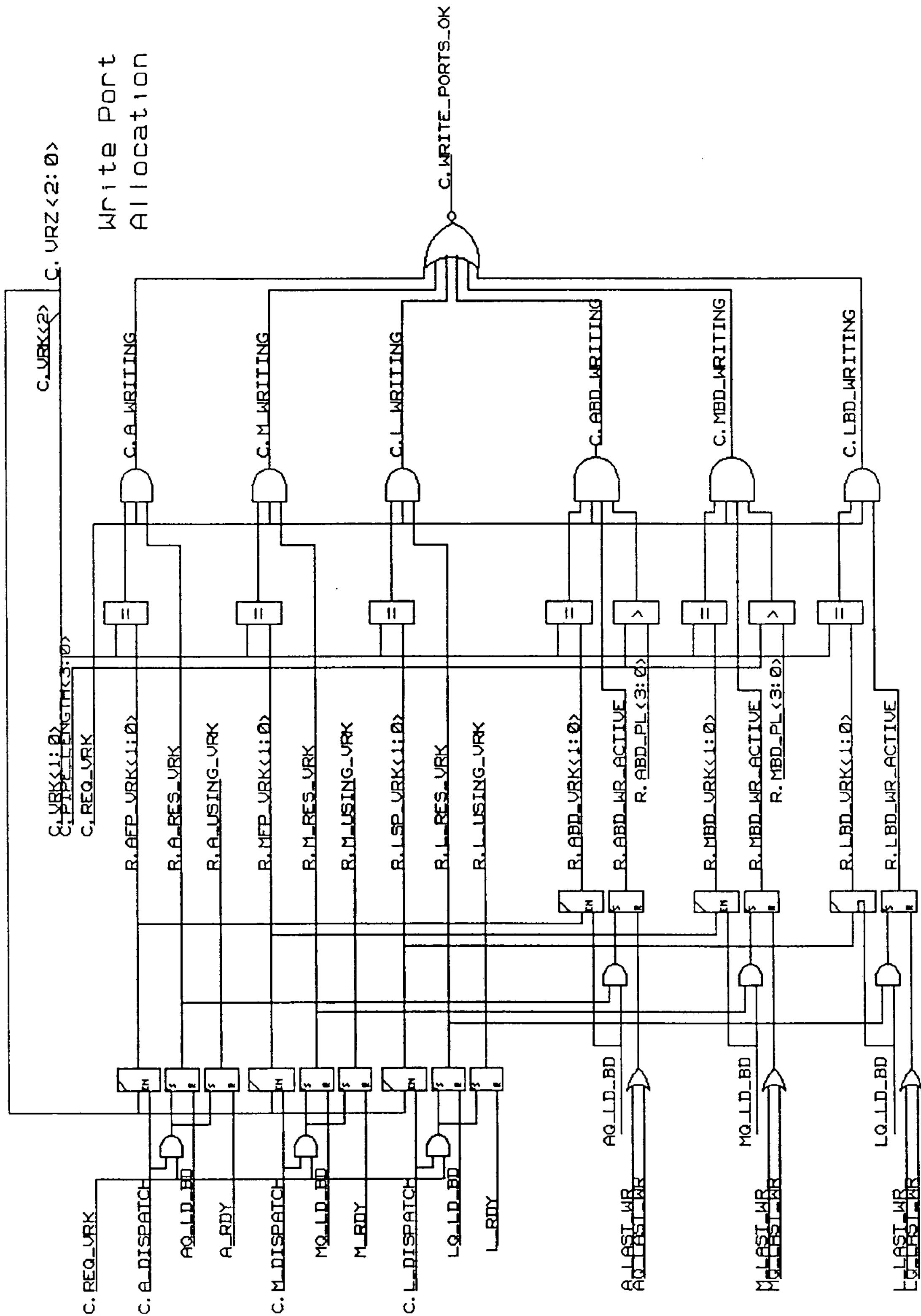


Figure 1-3 Write Port Allocation and Hazard

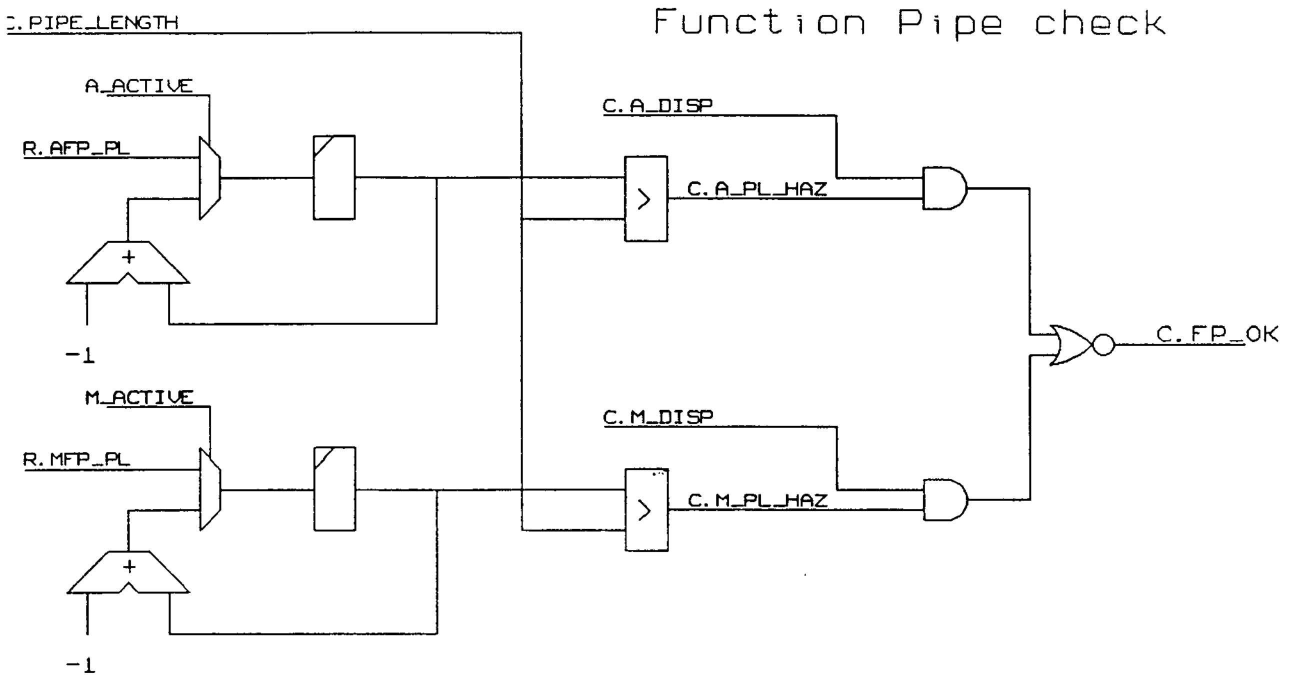


Figure 1-4 Function Pipe Check

the same architecture and some hardware. In general a rate_2x ROW hazard occurs if the instruction to be dispatch is a rate_2x instruction while a register it is trying to read is being written at the normal rate. A VMA ROW hazard exists if the dispatching instruction **could** run in the VM accelerated mode while a register it is trying to read is being written in a non-accelerated mode or in accelerated mode with the opposite VM polarity. The rate_2x ROW hazard is a hard hazard which causes an instruction to not dispatch until the hazard is cleared. The VMA ROW hazard allows the under-mask instruction to dispatch but causes it to not run in accelerated mode.

The first part of the ROW hazard check is to compare the requested read registers against the write registers of the instructions that are already running. A match generates the C.RD_ON_x_WR. In the case of the rate_2x ROW check this is combined with the rate being requested for the dispatching instruction and the rate being run on the function pipe to generate the hazard for each pipe. In the case of the VMA ROW check, the C.RD_ON_x_WR is combined with signals indicating that the dispatching instruction wants to run in accelerated mode to produce a hazard that keeps the instruction from dispatching in accelerated mode, but allows it to dispatch in non-accelerated mode. (This is reasonable since if the instruction that is running is non-accelerated, the dispatching instruction can run non-accelerated and be covered by the instruction that is already running.)

15.8 Write Overruns Read Check

The write-overruns-read check is similar in nature to the read-overruns-write check described in

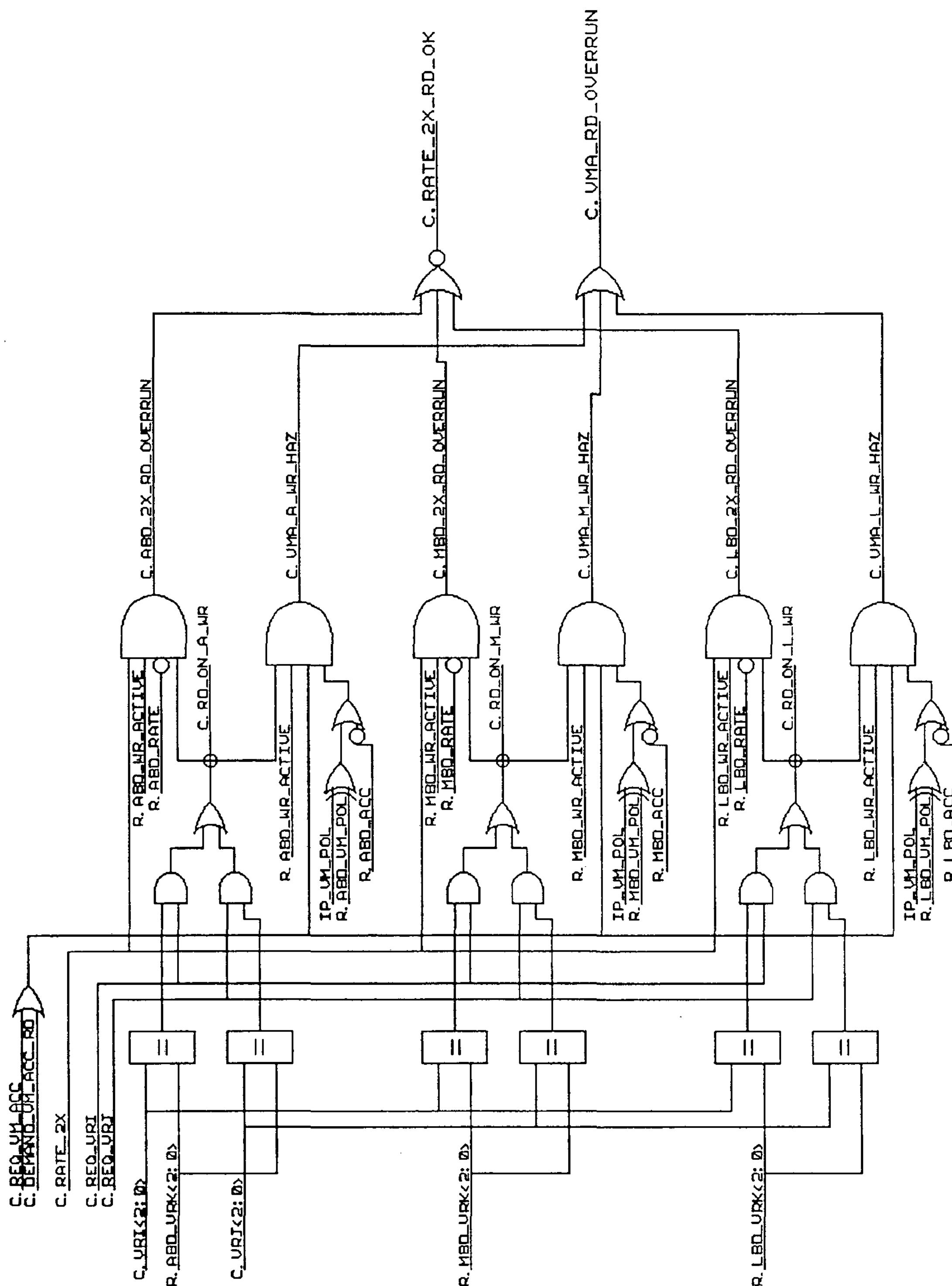


Figure 1-5 Read Overruns Write Hazard Check

the previous paragraph. The risk here is that a rate_2x or VMA instruction can require writing into a register that is currently being written by a previous instruction which is running in a normal mode and the new instruction may attempt to write results faster than they are being read by the normal instruction, thus corrupting the data of the previous instruction. This constitutes a write-overruns-read (WOR) hazard. A diagram of the vector register write-overruns-read hazard checking is shown in Figure 1-6 on page 15-10. Like the ROW hazard checking, the WOR hazard checking shares some hardware between the rate_2x and VMA hazards.

In general a rate_2x WOR hazard occurs if the instruction to be dispatch is a rate_2x instruction while the register it needs to write is being read at the normal rate by a running instruction. A VMA WOR hazard exists if the dispatching instruction **could** run in the VM accelerated mode while a register it needs to write is being read in a non-accelerated mode or in accelerated mode with the opposite VM polarity. The rate_2x WOR hazard is a hard hazard which causes an instruction to not dispatch until the hazard is cleared. The VMA WOR hazard allows the under-mask instruction to dispatch but causes it to not run in accelerated mode.

The registers that are currently being read by the three pipes are compared against the requested write register of the dispatching instruction. A match generates the C.WR_ON_x_RD. In the case of the rate_2x WOR check this is combined with the rate being requested for the dispatching instruction and the rate being run on the function pipe to generate the hazard for each pipe. In the case of the VMA WOR check the C.WR_ON_x_RD is combined with signals indicating that the dispatching instruction wants to run in accelerated mode to produce a hazard that keeps the instruction from dispatching in accelerated mode, but allows it to dispatch in non-accelerated mode.

15.9 VM Allocation and Chaining Hazard Checking

This check takes care of four basic problems for the VM register: Conflicts for serial and parallel writes to the VM register, conflicts over use of the serial write port into the VM register, dangers posed by changing the VL while a VM read is in progress, and chaining of instructions through the VM register. A drawing of these checks is shown in Figure 1-7 on page 15-11

15.9.1 VM Chaining Check

The VM chaining hazard check is similar to the vector register chaining check, except that there is only one VM register as compared to the eight vector registers. Also, only the add and multiply pipe have the ability to write serially into the VM register. In implementation, the chaining hazard check shares hardware with the VM write port allocation. When an instruction is dispatched on the add or multiply pipe that will write serially to the VM register, a set-reset register R.x_VM_WR is set to indicate that the VM write port has been allocated and by implication a write into the VM register is pending. No chaining is possible while R.x_VM_WR is asserted. This register is cleared by xQ_LD_BD which indicates that the VM write has reached the back-door level.

At this point a set-reset register R.xBD_VM_WR_PEND is set which indicates that the write is at the back-door level but that it hasn't yet actually written into the VM register. Chaining through the VM register is still prohibited until xQ_FIRST_WR is asserted and clears this register.

Parallel writing via the load pipe is also included in the chaining check despite the fact that it is not chaining in the strict sense. When the load pipe is dispatched with an instruction that writes the VM register it is by definition a parallel write. The set-reset register R.L_VM_WR is set on this dispatch. No chaining is possible until this register is cleared by the L_LAST_WR signal indicating

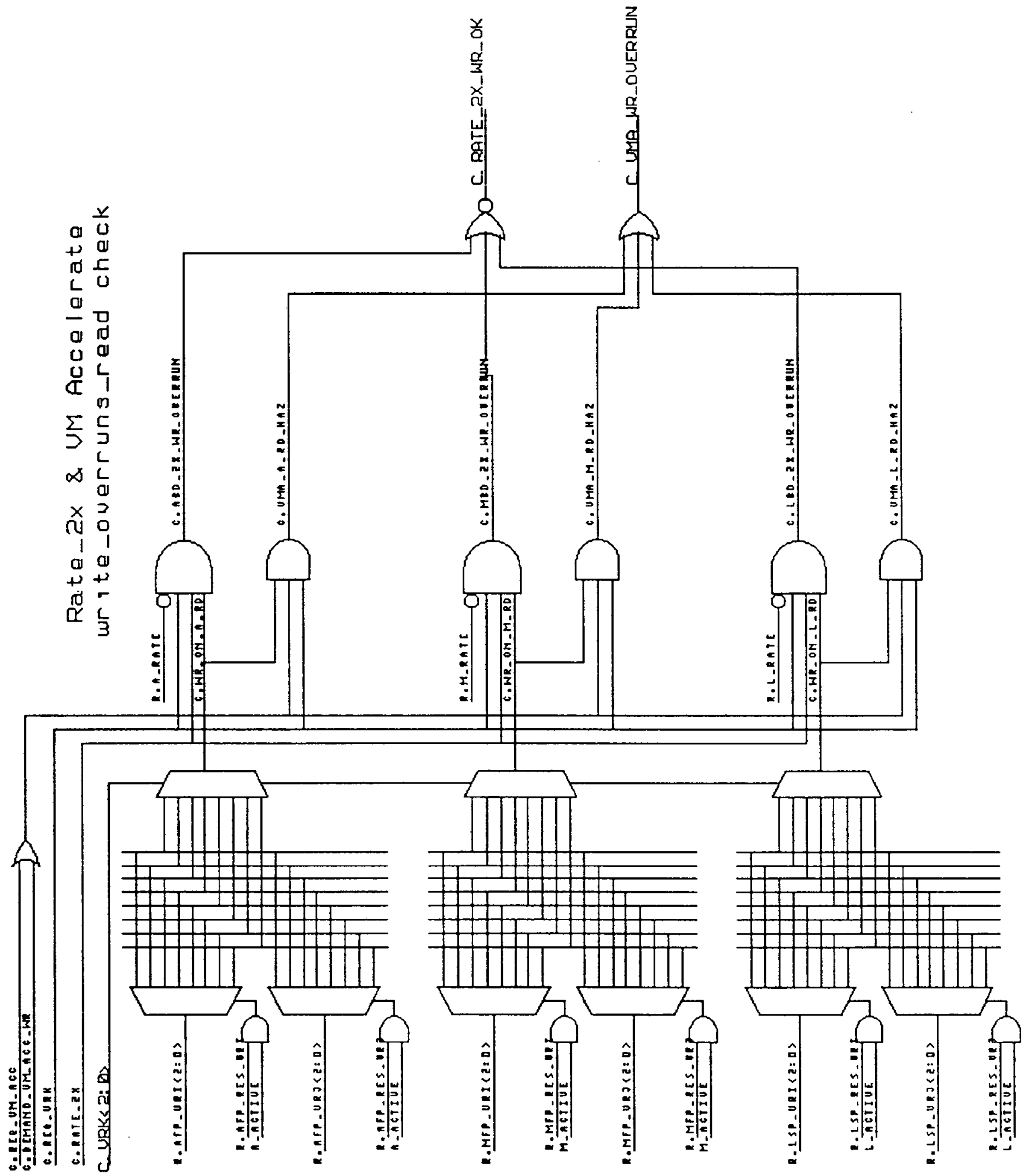


Figure 1-6 Write Overruns Read Hazard Check

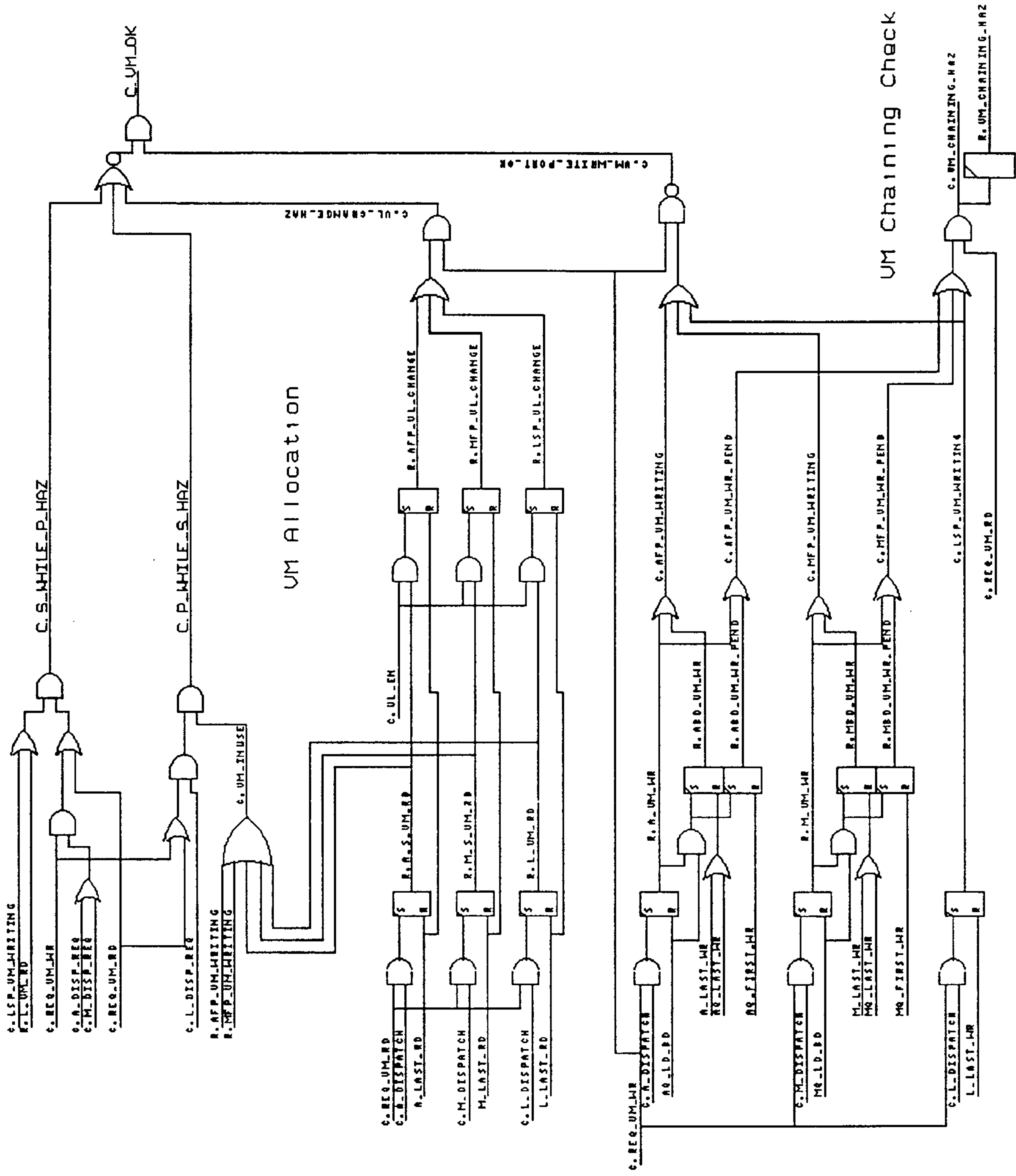


Figure 1-7 VM Allocation and Chaining Check

that the parallel write is complete.

15.9.2 Serial/Parallel VM hazards

The serial/parallel VM hazards deal with potential conflicts between serial and parallel accesses of the VM register. The first hazard of this type is to attempt a serial access to the VM register while a parallel access is already underway. If the serial access was a write and was allowed to dispatch it could corrupt the data that was being read in parallel. If the serial access was a read and was allowed to dispatch its VM contents could be corrupted by a parallel write that was not complete.

The second hazard of this type is to attempt a parallel access to the VM register while a serial access is already underway. If the parallel access trying to dispatch was a write it could corrupt any serial read that was already running. If the parallel access trying to dispatch was a read it could get the wrong data if a serial write was still in progress. Words are not sufficient to describe the implementation of these hazard checks.

15.9.3 VM Write Port Hazards

The VM write port allocation shares some hardware with the VM chaining check described above. In addition to the hardware described above that sets R.x_VM_WR for the add and multiply pipes there is a set-reset register R.xBD_VM_WR which is set when the write transitions from the front-door to the back-door as indicated by the xQ_LD_BD signal. As long as either of these set-reset registers is set, the VM serial write port is in use and no instruction can dispatch which would use it. The R.L_VM_WR signal is also used here also, to avoid conflicts between serial and parallel writes to the VM register.

15.9.4 VL Change Hazard

The VL change hazard is required due to the fact that the Convex ISA requires that the VM register be cleared above VL upon completion of a serial write. The hazard results if an instruction is dispatched that serially reads the VM register with some VL and then another instruction wants to dispatch that serially writes the VM register with a shorter VL. Normally this would be acceptable since the write would stay behind the read. However, if the write has a shorter VL, it may complete before the serial read and clear VM locations that the read is still using. That would be bad.

The VL change hazard logic keeps track of instructions that are reading the VM register. If a load of VL occurs while VM read is running a set-reset register R.xFP_VL_CHANGE is set and not cleared until the last read on that pipe. This prohibits dispatching any instruction that wants to write the VM register. This check does not know what the old and new VL values are and so produces a hazard whenever VL changes during a VM read regardless of whether the VL was increased or decreased.

15.10 VM Rate_2x Hazard Check

The rate 2x hazard check for the VM register is basically the same in structure as that for the vector registers. The check is broken up into read-overruns-write and write-overruns-read sections. Each of these is almost identical in structure to the corresponding vector register checks. The main difference is that there is only one serial write port into the VM register and there is a read port for each pipe permanently allocated. As a result no comparisons are necessary. A drawing of the rate 2x check for the VM register is shown in Figure 1-8 on page 15-13

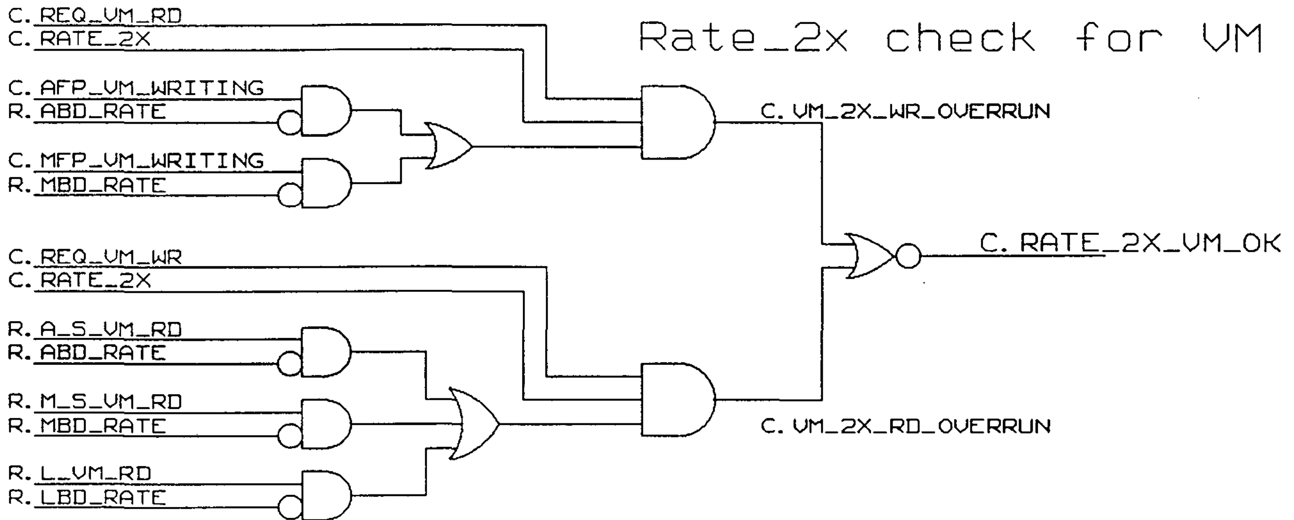


Figure 1-8 VM Rate 2x Hazard Check

15.11 VM Accelerate/Normal Selection and Edit Hazards

The VM accelerate/normal selection collects VMA hazards and munges the micro-controller entry point to force the mode that the instruction should run in. The logic to control edit hazards is included with this since it shares some hardware. A drawing of this logic is shown in Figure 1-9 on page 15-14.

15.11.1 VM Accelerate/Normal Selection

The possible hazards which could keep an under-mask instruction from running accelerated are collected to generate the C.NO_ACC signal. This is combined with the ENABLE_VM_ACC signal and the request for accelerated mode to generate the C.VM_ACC signal which indicates that the instruction should run accelerated if asserted. This signal is run through a mux which makes it the LSB of the micro-controller entry point if the instruction is requesting to run accelerated. Otherwise the entry point from the dispatch table is passed through unaltered.

15.11.2 Edit Hazard Checking

The edit hazard checking checks VM accelerate hazards for all instructions which have DEMAND_VM_ACC_RD or DEMAND_VM_ACC_WR in the dispatch table. This was originally designed in to be a VM accelerate hazard check for the compress and expand (CPRS and XPND) instructions. These two instructions are brain-damaged in that they read at a different rate than they write. The compress instruction reads its data in accelerated mode but writes it sequentially. The expand instruction reads data in sequential (non-accelerated) mode but writes it in accelerated mode. This two-mode behavior is coercive – the instructions cannot be run without the accelerated mode. Some other instructions also use the demand-accelerate modes in order to force hazards on the VM register.

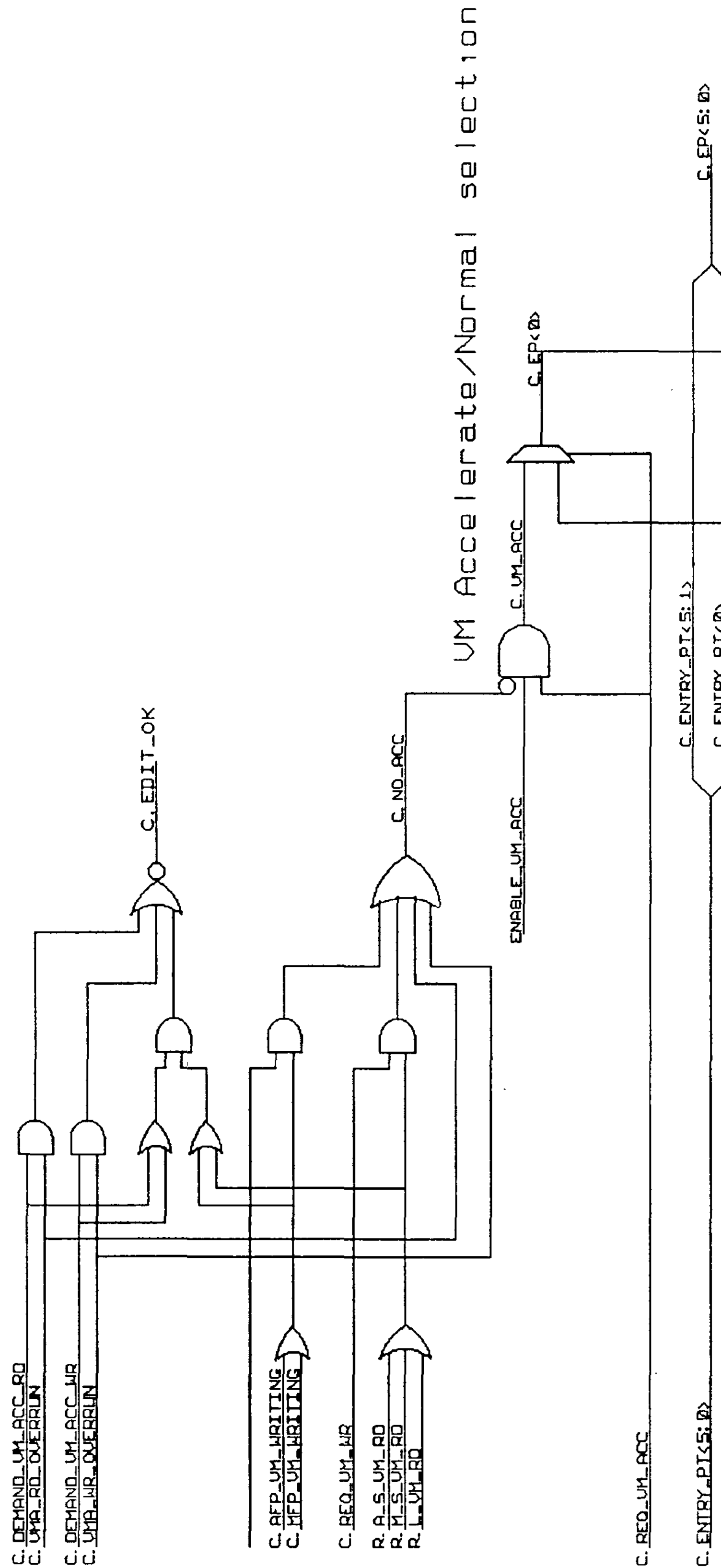


Figure 1-9 VM Accelerate/Normal Selection and Edit Hazard Check

15.12 Merge Hazard Check

The merge hazard is a special case. The merge (MERG) instruction is an abomination that reads two registers at a rate dependent on the contents of the VM register and writes the destination register at the normal rate. Once a merge instruction has started no other instruction is allowed to dispatch until the merge is finished. The logic keeps track of whether there is a merge instruction running on the front or back doors of either the add or multiply pipe and creates a hazard if there is. A drawing of the merge hazard check logic is shown in Figure 1-10 on page 15-15.

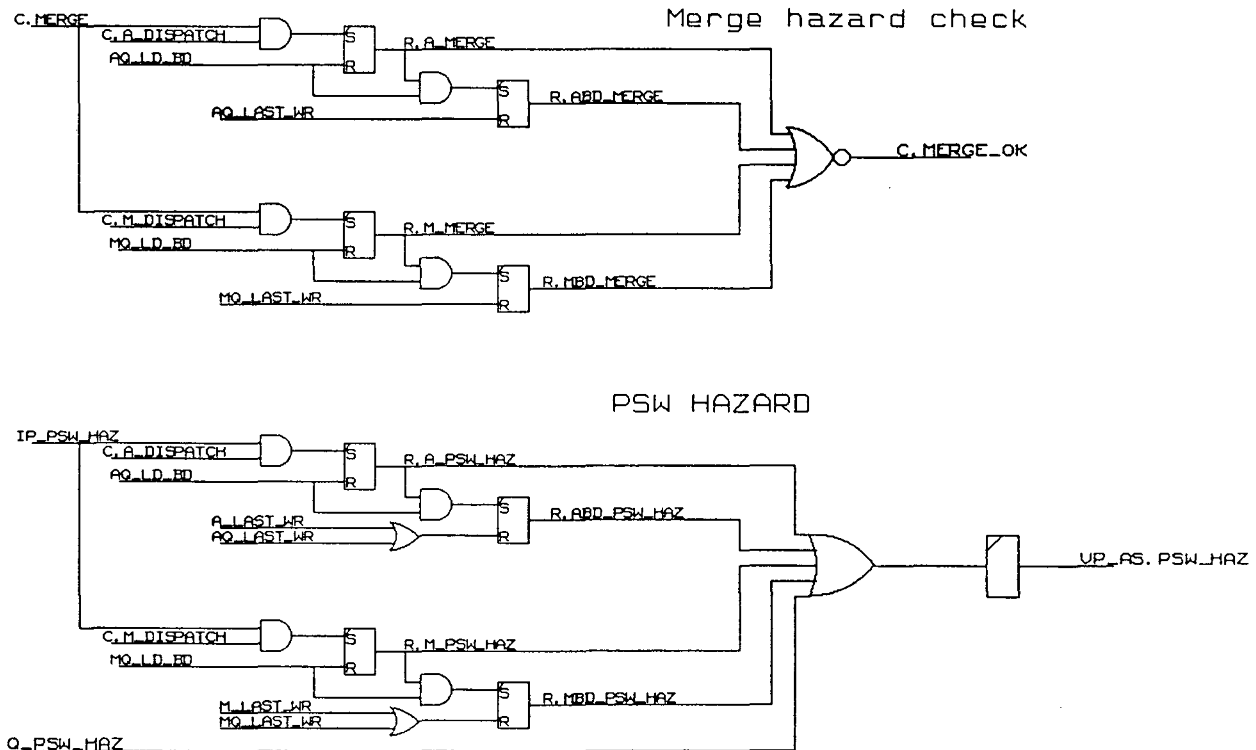


Figure 1-10 Merge Hazard Check and PSW Hazard Logic

15.13 PSW Hazard Logic

The PSW hazard logic is in two parts. One part of it is in the external glue logic. That part keeps track of whether an instruction that is waiting in the dispatch queue or in the process of being dispatched has a PSW hazard associated with it. The other part of the PSW hazard logic is internal to the NVD gate array and it keeps track of whether an instruction running on either the add or multiply pipe has a PSW hazard associated with it. (The load pipe performs no instructions which can have a PSW hazard). A drawing of the NVD-internal portion of the PSW hazard logic is shown in Figure 1-10 on page 15-15.

Appendix A Signal List

About terminology. As has been noted elsewhere in this specification, the naming terminology for signals generally follows the levels of the vector pipeline. For example, signals prefixed by X_ (where "X" is either A, M or L) are generated by the pipe controller itself. Signals prefixed by X1_ are at the first pipeline level after the controller, and so on for X2_, X3_,etc. Signals prefixed by XQ_ are at the output of the pipeline and are about to be written back (or control the write-back) into the VRFs or VM register.

Signals with the prefix "C." are combinatorial signals. They often go directly to a register. If you do not see the "C." version of the signal in this list, look for the signal name without the "C." or possibly with an "R." instead of the "C.".

Signals with a suffix of a period and a letter or number, such as "CNTX_MODE.A", are duplicate copies of the signal without the period and the letter. In this case, CNTX_MODE.A is a duplicate of CNTX_MODE, CNTX_MODE.B, etc.

Occasionally, two different signals are given the same name, but are in different sections of the schematics. This is most likely true in the AFC and MFC controllers. The signals are prefixed by "(AFC)" or "(MFC)" or some other unique identifier in the signal descriptions. My apologies for having given multiple signals the same name.

A1_ACTIVE

Level 1 active signal for the add pipe. Goes directly from NVM to NQ array to be entered in the control queue. After being delayed to the A3 level, it controls loading of the a_queue internal to the NQ array.

A1_LAST_ELEM

Level 1 "last element" signal for the add pipe. Goes directly from NVM to NQ array to be entered in the control queue. See AQ_LAST_ELEM.

A1_PIPE_CTL<2..0>

Level 1 pipe control signal for the add pipe. Goes directly from NVM to NQ array to be entered in the control queue. See AQ_PIPE_CTL.

A1_PL<2..0>

Level 1 pipe length signal for the add pipe. Goes directly from NVM to NQ array to be entered in the control queue. The pipe length is used in the NQ array to set the depth of the control queue. The value of A1_PL<2..0> is equal to the number of clocks from the A3 level to the AQ level.

A1_RATE_2X

Level 1 rate 2x signal for the add pipe. Goes directly from NVM to NQ array to be entered in the control queue. See AQ_RATE_2X.

A1_VM<1..0>

Level 1 VM bits for the add pipe. Goes directly from NVM to NQ array to be entered in the control queue. See AQ_VM<1..0>.

A1_Z_CNT<7..0>

Level 1 z count signal for the add pipe. The Z count is the VRF write port address that will be used for returning data. Goes directly from NVM to NQ array to be entered in the control queue. See AQ_Z_CNT<7..0>.

A2_ACTIVE

Level 2 active signal for the add pipe. Along with A2_PIPE_CTL, tells the OSCTL that the add pipe is invoved in a VXS transfer.

A2_EDIT_VM

Level 2 VM bit for the add pipe used in vector edit operations (e.g., MASK, MERGE, COMPRESS).

A2_IEEE

Level 2 IEEE bit for the add pipe. Goes from the NQ to the AFC controller. Tells the AFC to perform IEEE floating point operations if asserted.

A2_OPCODE<9..0>

Level 2 opcode bits for the add pipe. See the gate array appendix of this spec for opcode definitions.

A2_PIPE_CTL<2..0>

Along with A2_ACTIVE, tells the OSCTL controller that a VXS transfer is taking place on the load pipe if A2_PIPE_CTL = 0x7. Other values are don't-care for the OSCTL controller.

A2_PL<2..0>

Tells the AFC controller the pipe length of the current operation. The PL is equal to the number of clocks from the A3 level to the AQ level.

A2_RATE_2X

Tells the AFC controller whether the current operation is a 2x or 1x operation.

A2_START

Tells the AFC controller that a piece of data is available at level 2 in the add pipe.

A2_VM<1..0>

These VM bits enable the status bits coming out of the function units in the add pipe.

A2_VXS

Tells the NVRF arrays that a piece of scalar data from the add pipe should be put out on the L3_OP0_DAT bus for a VXS transfer.

A3_IEEE

Delayed version of the IEEE bit in the AFC controller to align it with the data that will be coming from the VRF on the A3_OP?_DAT bus.

A3_OP0_DAT<63..0>

A3_OP1_DAT<63..0>

Add pipe operands coming from the VRFs to the add pipe function units. The three indicates that the data is at level 3 of the pipeline.

A3_OP0_NIBBLE_PAR<7..0>

A3_OP1_NIBBLE_PAR<7..0>

Nibble parity for add pipe operand 0. This parity will be captured in discrete registers. If the operand data has a parity error detected when it reaches the function unit, this nibble parity (in conjunction with the normal byte parity) will allow localizing the source of the parity error to one of the 8 NVRF gate arrays.

A3_OP0_PAR<7..0>

A3_OP1_PAR<7..0>

Add pipe operand parity coming from the VRFS to the add pipe function units.

A4_OP0_NIB_PAR<7..0>

A4_OP1_NIB_PAR<7..0>

Delayed version of A3_OP?_NIB_PAR<7..0>. This register is aligned in the pipeline with the XBUS and YBUS input registers of the function units.

A5_OP0_NIB_PAR<7..0>

A5_OP1_NIB_PAR<7..0>

Twice delayed version of A3_OP?_NIB_PAR<7..0>. In the event of a parity error on data in the XBUS or YBUS register of a function unit on the add pipe, the register generating this signal will contain the nibble parity for the data that caused the parity error.

AFP_PAR_ERR<7..0>

Parity error signals for all of the AFP function units.

AF_BR_ADDR<9..0>

Branch address for the UA controller.

AF_BR_INST<2..0>

Branch type for the UA controller.

INST<2..0>	Type
0	Unconditional branch to AF_BR_ADDR<9..0>.
1	Conditional branch to AF_BR_ADDR<9..0>
2	Unconditional jump to dispatch EP
3	Conditional jump to dispatch EP
4,5	Conditionally loop or next address
6,7	Multi-way branch from AF_BR_ADDR<9..4> based on VL

AF_CNT_CTL<3..0>

Type of address sequence for VM to produce for the NVRFs.

CTL<3..0>	Type
1	Clear X, Y, Z, and vm counts. (Initialize.)
2	Increment X, Y, Z, and vm counts by 1 or 2 if rate_2x. (Normal mode)
3	set X, Y, Z, and vm counts to address of next set vm bit. (Accelerate mode)
4	Hold X, Y and vm counts. Set Z = VL. (Post-write cleanup)
5	Set X and Z to LQ_RSLT_DAT<8..0>. (load/store vector element)
8	Accelerate mode X. Hold Y, Z, and vm counts. (Compress in accelerate mode)
9	Increment X. Hold Y, Z, and vm counts. (Compress)
A	Hold X and Y. Accelerate mode Z and vm counts. (Expand in accelerate mode)
B	Increment X. Accelerate mode Z and VM. Hold Y. (Expand)
C	Merge mode. See isp model.

AF_EXT<8..0>

Control store signal grouping that contains un-registered versions of parity, A_OP0_CTL<2..0>, A_OP1_CTL<2..0>, and two spare bits.

AF_LAST<2..0>

Indicates that the last microcode cycle is occuring so that the NVM can generate the last_rd, last_vxs, last_wr, last_elem, last_sxv control signals.

LAST	Type
<0>	Last VRF read
<1>	Last VXS transfer
<2>	Not used on add pipe
<3>	Last VRF write or SXV transfer

AF_PIPE_CTL<2..0>

Pipe control codes.

CTL<2..0>	Pipe operation
1	Load back door. Done at start of operation.
2	Start operation. Used for reductions.
3	Start operation. Used during pipe-down of reductions.
4	Write to VRF. Used during normal vector operations.
5	Zero out VM register above VL. Done at end of compares.
6	Write to VM register. Used during compares.
7	Do a VXS transfer. Used at end of reductions.

AF_TEST_POL

Selects whether a conditional microcode branch condition is true or false for the UA controller.

AF_TEST_SEL<2..0>

Selects what type of condition is used for a conditional microcode branch on the UA controller.

SEL<2..0>	Condition type
0	VL <= 1 for rate_1x, VL <= 2 for rate_2x.
1	VL - vm count <= 2 for rate_1x, VL - vm count <=4 for rate_2x.
2	SXV_DVAL asserted.
3	operation is rate_2x
4	LQ_RSLT_DAT<0>

ALL_SCAN_LEFT

Scan control signal telling discrete registers that are not on the context ring (all of which are in NVP_CLK) to shift left.

AQ_ACTIVE

Back-door level active signal for the add pipe. Indicates to various parts of the NVP that an operation is indeed occuring at the back-door level on the add pipe, and that other signals at this level are valid.

AQ_CT<1..0>

VM bits generated by the add pipe during compare operations. These will be written into the VM register.

AQ_FDZ<5..0>

Floating point divide-by-zero status signals from the six NDIV arrays on the add pipe.

AQ_FSN<5..0>
: Floating point square-root of a negative number status signals from the six NDIV arrays on the add pipe.

AQ_LAST_ELEM
: Indicates that the element at the back-door level of the add pipe is the last element in an operation.

AQ_LAST_WR
: Indicates that the last write to a vector register of an operation is taking place on the add pipe.

AQ_LD_BD
: Indicates that an add pipe operation is beginning to use the back door controller.

AQ_OV<7..0>
: Overflow status signals from all of the function units on the add pipe.

AQ_PIPE_CTL<2..0>
: Back door level pipe control signal for the add pipe. See AF_PIPE_CTL<2..0> for description.

AQ_RATE_2X
: Indicates that the operation at the back door level of the add pipe is a rate_2x operation.

AQ_RO<7..0>
: Floating point reserved-operand status signal from all of the function units on the add pipe.

AQ_RSLT_DAT<63..0>
: Result data from the add function pipe function units.

AQ_RSLT_PAR<7..0>
: Result bus parity from the add function pipe function units.

AQ_SDZ<5..0>
: Integer divide-by-zero status signals from the six NDIV arrays on the add pipe.

AQ_SIV<6..0>
: Integer overflow status signals from the function units on the add pipe.

AQ_SOURCE<5..0>
: This is the registered version of the last level of the AFC control queue. These bits indicate the source of the data that is on the AQ_RSLT_DAT<63..0> bus. If the result data causes a parity error at the NVRF gate arrays, the source of the error can be determined.

SOURCE<5..4>	Function unit type
0	NFAD
1	NMISC
2	NDIV
3	It's broken

SOURCE<3> Rate_2x

SOURCE<2..0>	Pointer to NDIV
0	NDIV0 (and NDIV1 if Rate_2x)
1	NDIV1
2	NDIV2 (and NDIV3 if Rate_2x)
3	NDIV3
4	NDIV4 (and NDIV5 if Rate_2x)
5	NDIV5

AQ_UN<7..0>
: Underflow status signals from the function units on the add pipe.

AQ_VM<1..0>
: VM bits for the add pipe at the back door level. These tell the NVRF whether to write back the results or not during an under-mask operation.

AQ_VM_WR
: Tells the STAT unit to select back door information from the add pipe (if asserted) or the multiply pipe. The selected information is used for writing the VM register.

AQ_Z_CNT<7..0>

Back door level Z count for the add pipe. The Z count is the VRF write port address that will be used for returning data (AQ_RSLT_DAT).

A_ACTIVE

Add pipe active signal from the add pipe controller. Indicates that the add pipe controller (UA) is active and sending valid control signals.

A_CLR_ACTIVE

Indicates to the NVD that the A_ACTIVE signal is about to go away. The NVD array keeps its own internal copy of A_ACTIVE for timing reasons and needs to know when to shut it off.

A_DISP

A vector dispatch signal that indicates that the instruction in dispatch can run on the add pipe.

A_DISPATCH

Indicates that an instruction is being dispatched on the add pipe. All of the dispatch information is valid.

A_LAST_RD

Indicates that the last vector register read of an instruction is finished and the pipe is ready to surrender the VRF read ports.

A_LAST_VXS

Indicates that the last VXS transfer for an instruction is complete and that the add pipe is ready to surrender its claim (if it made one) on the VXS port.

A_LAST_WR

Indicates that the last vector register write of an instruction is finished and the pipe is ready to surrender the VRF write ports.

A_MUX_SEL

Selects either A_RSLT_DAT_1<63..0> (if asserted) or A_RSLT_DAT_0<63..0> to be switched onto the AQ_RSLT_DAT bus.

A_OP0_CTL<2..0>

Operand 0 control for the NVRF arrays. This selects what type of data will be placed on the A3_OP0_DAT<63..0> bus.

CTL<2..0>	Data selected
0	integer zero
1	scalar data
2	if a2_edit_vm then Y data. otherwise x data. (see NVRF section)
3	if a2_edit_vm then scalar data. otherwise x data. (see NVRF section)
4	add pipe result data
5	word-swapped add pipe result data
6	vector register X data. masked if necessary. (see NVRF section)
7	identity element

A_OP1_CTL<2..0>

Operand 1 control for the NVRF arrays. This selects what type of data will be placed on the A3_OP1_DAT<63..0> bus.

CTL<2..0>	Data selected
0	integer zero
1	integer -1
2	scalar data
3	scalar data. masked if necessary.
4	add pipe result data
5	integer -1
6	vector register Y data. masked if necessary. (see NVRF section)
7	identity element

A_RDY

Add pipe controller ready for dispatch.

A_RSLT_DAT_0<63..0>

Add pipe result data from either NDIV0, NDIV1, NFAD_A or NMISC_A.

A_RSLT_DAT_1<63..0>

Add pipe result data from either NDIV3, NDIV4, NDIV4 or NDIV5

A_RSLT_PAR_0<7..0>

Add pipe result data parity from either NDIV0, NDIV1, NFAD_A or NMISC_A.

A_RSLT_PAR_1<7..0>

Add pipe result data from parity either NDIV3, NDIV4, NDIV4 or NDIV5

A_UADDR<9..0>

Current micro-address of the UA controller.

A_UIR_PAR_ERR

Parity error signal for the add pipe micro-controller's micro-instruction register. Part of this register is internal to the A_VM array and part of it is in a register in the UA controller.

A_UIR_PAR_EXT

A parity syndrome bit for the externally stored portion of the add pipe micro-instruction register. This signal goes into the A_VM array to be combined with the parity syndrome on the internal portion of the add pipe micro-instruction register. The result is A_UIR_PAR_ERR.

A_UPC<9..0>

A registered version of the A_UADDR<9..0> signal. This register is stopped in the event of a parity error in the A_VM gate array. If the parity error is in the micro-instruction register, A_UPC<9..0> should have the address from which the bad data came.

A_X_CNT<6..0>

X port vector register address for the add pipe. Tells the NVRFs which element to pull out of the register file.

A_Y_CNT<6..0>

Y port vector register address for the add pipe. Tells the NVRFs which element to pull out of the register file.

BD_SCAN_OUT

Scan out signal from the BD body.

C. VD_DISPATCH

A test signal that indicates that the vector dispatch unit is going through a dispatch cycle that does not dispatch to any of the function pipes. This type of dispatch happens only during a load of the VL register.

C.A_DISPATCH

Indicates that an instruction is going to be dispatched on the add pipe on the next cycle.

C.D02_BUSY_NEXT_X0

C.D02_BUSY_NEXT_X1

C.D35_BUSY_NEXT_X0

C.D35_BUSY_NEXT_X1

These signals are partial computations of a larger equation that indicate that the divider will be busy on the next cycle that the AFC controller is waiting for data from. This generates the DIV_CK_HOLD signal that in turn generates CK_XTND for the board.

C.HALT

This is a logical OR of all of the parity error signals on the NVP. It becomes VP_XC.HARD_ERROR on the next cycle.

C.IDLE

Indicates that the NVP will be idle on the next cycle. This is the combinatorial version of VP_SP.IDLE.

C.IP_REQ_NEXT

Indicates that the NVP will accept an instruction from the NSP on the next cycle if one is available. This is the combinatorial version of VP_SP.DISP_REQ_NEXT.

C.LQ_POP

If this signal is NOT asserted, then the load back-door controller is NOT going to take data. It does not leave the LBD controller and is used in some other equations.

C.L_DISPATCH

Indicates that an instruction is going to be dispatched on the load pipe on the next cycle.

C.M_DISPATCH

Indicates that an instruction is going to be dispatched on the multiply pipe on the next cycle.

C.PSW0

C.PSW1

C.PSW2

These signals correspond to the R0, R1 and R2 entries in the IPCTL queue, except that they indicate that a PSW hazard is associated with same numbered entry in the IPCTL queue. These are used in the production of the VP_SP.PSW_HAZ signal.

C.Q_PSW_HAZ

Indicates that there is a valid instruction in the IPCTL queue which has a PSW hazard associated with it.

C.R0_EN

C.R1_EN

C.R2_EN

These signals enable the three registers (R0, R1, R2) in the IPCTL queue which are internal to the NVD array. Those registers contain vector instructions from the NSP.

C.SET_FDZ

C.SET_FSN

C.SET_OV

C.SET_RO

C.SET_SDZ

C.SET_SIV

C.SET_UN

These signals are the OR of the respective status signals from the function units on the NVP. They are registered to generate the VP_SP.SET_XXX version of the signals.

C.VL0_ABORT

This vector dispatch signal indicates that VL=0, and that the instruction to be dispatched does not dispatch when VL=0, and therefore the dispatch should be aborted.

C.VP_PSW_HAZ

Indicates that the VP has an instruction that is generating a PSW hazard. This signal is registered to generate VP_SP.PSW_HAZ.

C.VXM

Internal to the OSCTL controller. Indicates that the load pipe is running and attempting to do a VXM transfer.

C.VXS

Internal to the OSCTL controller. Indicates that the load pipe is running and attempting to do a VXS transfer.

CHAINING_OK

A test point signal from the NVD array that indicates that the instruction in dispatch has passed the chaining check.

CK_EN*

A clock generation enable that gates the 3x clock down to a 1x clock during normal operation.

CK_XTND

The general clock extend signal for the NVP that goes to the gate arrays on the board.

CNTX_CK_HOLD

The clock hold from the context state machine (in NVP_CLK). This is used to generate the many clock extend signals during a context switch.

CNTX_MODE

Indicates that the NVP is undergoing a context switch. Asserted whenever SP_VP.CNTX_CTL is in any mode other than 00.

CNTX_OR_PREV

Indicates that CNTX_MODE is asserted during the current cycle or was asserted one cycle earlier.

CNTX_RESET

Indicates that SP_VP.CNTX_CTL=0x2. This causes the gate arrays to have their context controls set to CONTEXT_RESET.

CNTX_RESTORE

Indicates that the NVP is undergoing the restore phase of a context switch. Causes the NIS arrays to put context restore data onto the SXV_DAT bus.

CNTX_SAVE

Indicates that the NVP is undergoing the save phase of a context switch. Causes the IS_MUX to break the board scan chain into pieces to feed to the NSP, 32 bits at a time.

CSM_CK_HOLD

This signal is generated by the context state machine to stop the clocks during a context switch if restore data is not available from the NSP during a context restore, or if the NSP is not prepared to accept data from the NVP during a context save.

CSM_CK_XTND.A<7..0>

The clock extend generated for the parts of the NVP that are normally running during a clock extend in context mode.

C_MXV_DVAL

A combinatorial version of MXV_DVAL that is generated by the NIS gate arrays and registered externally for timing reasons. See MXV_DVAL.

DIAG_OK

A diagnostic pin from the NVD gate array. Based on an internal enable register, this pin can show the status of any of the dispatch checks in the NVD.

DISP_RAM<53..0>

This is the data from the dispatch RAM in the vector dispatch logic. It is also used to write data back into the dispatch RAM during system initialization.

DISP_SUCCEED

Indicates that an instruction was dispatched on any pipe or that a load VL completed.

DIV_CK_HOLD

This hold signal is generated in the AFC to create clock extends when the a divide/sqrt result is not available.

DMODE

Diagnostic mode. Indicates that the XC_VP.SCAN_CTL<2..0> is not equal to 00.

DST_SIZ<1..0>

Destination size. A dispatch signal that indicates the size of the data to be written back into the VRF at the end of an operation. The destination size may be different than the source size for such operations as type converts.

SIZ<1..0>	Destination Size
0	byte
1	half-word
2	word
3	long-word

D_CK_XTND

See CK_XTND.

D_CSM_CK_XTND

See CSM_CK_XTND.

D_EXT_CK_XTND

See EXT_CK_XTND

D_FREE_CK_XTND

See FREE_CK_XTND.

D_FREE_EXT_CK_XTND

See FREE_EXT_CK_XTND

EP<5..0>

The micro-code entry point for the UA, UM and UL controllers. This signal is active at dispatch time and tells the controllers the address at which to begin. The actual address used by the controllers is EP<5..0> with 3 zeros appended.

EXT_CK_HOLD

This signal is used to generate EXT_CK_XTND. It causes the discrete (non-gate-array) registers to stop during a context switch when they are not supposed to be scanning.

EXT_CK_XTND

This clock extend stops the discrete (non-gate-array) registers when they should not be clocked.

EXT_SCAN_IN

Scan input at the beginning of the discrete portion of the NVP scan ring. This is actually connected to the scan input for the UM controller.

EXT_SCAN_OUT

Scan output at the end of the discrete portion of the NVP scan ring. It comes out of the VD logic.

FREE_CK_XTND

A clock extend that is used for gate arrays that need to have a free running clock during normal operation when the rest of the NVP is clock extended, but that need to have their clocks extended during a context switch. An example of this is the NDIV gate arrays that need to have their internal parts running to generate results while the rest of the NVP is clock extended waiting for results from the NDIVs.

FREE_EXT_CK_XTND

A clock extend that is used for discrete registers that need to have a free running clock during normal operation when the rest of the NVP is clock extended, but that need to have their clocks extended during a context switch. An example of this is the IPCTL controller in the VD logic which needs to free run in order to properly handshake with the NSP.

FU_ERROR

Indicates a parity error from one of the function units. This is used to stop the nibble parity registers. See A5_OP1_NIB_PAR<7..0> for the use of the nibble parity registers.

(AFC) FU_OPCODE<7..0>

(MFC) FU_OPCODE<7..0>

Level 3 opcode bits for the add and multiply pipes. I shouldn't have given them the same name, I suppose. See the gate array appendix of this spec for opcode definitions.

(AFC) FU_VM<1..0>

(MFC) FU_VM<1..0>

Level 3 VM bits for the add and multiply pipes. I shouldn't have given these the same name, either. These VM bits enable the status bits coming out of the function units in.

GA_SCAN_CTL<1..0>

An un-buffered version of the scan control signal for all of the gate arrays.

ID_SEL<2..0>

A dispatch signal telling the NVRF arrays which identity element to use (if this is an operation which uses identities).

ID<2..0>	Identity Element
0	integer zero
1	integer one
2	minimum representable number
3	maximum representable number
4	all ones

IEEE

A dispatch signal indicating that the operation is to take place in IEEE mode (if it is a floating point operation).

IP_VP_EP<9..0>

The dispatch table lookup address. It is the entry point into the VD micro-code.

IS_PAR_ERR<1..0>

Parity error signals from the two NIS gate arrays.

L1_ACTIVE

Level 1 active signal for the load pipe. Goes directly from NVM to NQ array to be entered in the control queue. After being delayed to the L3 level, it controls loading of the l_queue internal to the NQ array.

L1_LAST_ELEM

Level 1 "last element" signal for the load pipe. Goes directly from NVM to NQ array to be entered in the control queue. See LQ_LAST_ELEM.

L1_VM<1..0>

Level 1 VM bits for the load pipe. Goes directly from NVM to NQ array to be entered in the control queue. See LQ_VM<1..0>.

L1_Z_CNT<7..0>

Level 1 Z count signal for the load pipe. The Z count is the VRF write port address that will be used for returning data. Goes directly from NVM to NQ array to be entered in the control queue. See LQ_Z_CNT<7..0>.

L2_ACTIVE

Level 2 active signal for the load pipe. Along with L2_PIPE_CTL, tells the OSCTL that the load pipe is invoved in a VXS or VXM transfer.

L2_PIPE_CTL<2..0>

Along with L2_ACTIVE, tells the OSCTL controller that a VXS transfer is taking place on the load pipe if L2_PIPE_CTL = 0x7, or that a VXM transfer is taking place if L2_PIPE_CTL = 0x2 or 0x6;

L3_LAST_ELEM

Indicates to the OSCTL that the last element of a store operation is occuring so that the OSCTL can assert VP_SP.VXA_LAST.

L3_VM<1..0>

These VM bits are sent to the NSP on VP_SP.VXA_VM<1..0> for store under-mask operations. Remember that VM<0> corresponds to the most significant word.

LF_BR_ADDR<9..0>

Branch address for the UL controller.

LF_BR_INST<2..0>

Branch type for the UL controller.

INST<2..0>	Type
0	Unconditional branch to LF_BR_ADDR<9..0>
1	Conditional branch to LF_BR_ADDR<9..0>
2	Unconditional jump to dispatch EP
3	Conditional jump to dispatch EP
4,5	Conditionally loop or next address
6,7	Multi-way branch from LF_BR_ADDR<9..4> based on VL

LF_CNT_CTL<3..0>

Type of address sequence for VM to produce for the NVRFs.

CTL<3..0>	Type
1	Clear X, Y, Z, and vm counts. (Initialize.)
2	Increment X, Y, Z, and vm counts by 1 or 2 if rate_2x. (Normal mode)
3	set X, Y, Z, and vm counts to address of next set vm bit. (Accelerate mode)
4	Hold X, Y and vm counts. Set Z = VL. (Post-write cleanup)
5	Set X and Z to LQ_RSLT_DAT<8..0>. (load vector element)

LF_EXT<8..0>

Control store signal grouping that contains un-registered versions of parity, L_OP0_CTL, L_OP1_CTL, L_SRC_CTL<2..0>, L_DST_CTL<2..0> and two spare bits.

LF_LAST<3..0>

Indicates to the NVM that the last microcode cycle is occuring so that it can generate the last_rd, last_vxs, last_wr, last_elem, last_sxv control signals.

LAST	Type
<0>	Last VRF read
<1>	Last VXS transfer
<2>	Last element of a load or store (MXV or VXM transfer)
<3>	Last VRF write or SXV transfer

LF_PIPE_CTL<2..0>

Pipe control codes.

CTL<2..0>	Pipe operation
1	Load back door. Done at start of operation.
2	VXM transfer. (Stores of all kinds)
4	Write to VRF from memory data. (Vector loads)
5	Write to VRF from scalar data. (Move scalar to vector element)
6	VXM and write VRF. (Load under mask and load vector of indexes)
7	VXS transfer. (Store VM. Move Vector element to Scalar.)

LF_TEST_POL

Selects whether a conditional microcode branch condition is true or false for the UL controller.

LF_TEST_SEL<2..0>

Selects what type of condition is used for a conditional microcode branch on the UL controller.

SEL<2..0>	Condition type
0	VL <= 1 for rate_1x, VL <= 2 for rate_2x.
1	VL - vm count <= 2 for rate_1x, VL - vm count <=4 for rate_2x.
2	SXV_DVAL asserted.
3	operation is rate_2x
4	LQ_RSLT_DAT<0>

LN_FULL

Indicates that the l_queue in the NQ array is filled with requests to memory. During an LDVI this causes the NVP to not clock extend due to lack of MXV data until the memory system is full of requests.

LN_LAST_ELEM

Indicates that the element at the back-door level of the load pipe is the last element in an operation. Used to generate LQ_LAST_SXV, LQ_LAST_WR, etc.

LN_MEM_FULL

Indicates that the l_queue in the NQ array is nearly filled with requests to memory. Used by VD logic to avoid dispatching instructions that could cause deadlock in the memory system.

LN_PIPED_UP

Indicates that the l_queue in the NQ array is filled with requests to memory. This is used during LDVI operations to insure that the NVP fills memory with requests before starting to accept memory return data.

LN_PIPE_CTL<2..0>

Next-to-back-door level pipe control signal for the load pipe. See LF_PIPE_CTL<2..0> for description.

LN_POP

Causes a "pop" from the l_queue in the NQ array.

LN_VALID

Indicates that there is at least one entry in the l_queue in the NQ array. That means that some sort of load pipe operation is occuring that is past the front-door level.

LN_VM<1..0>

Next-to-back-door level VM bits for the load pipe. See LQ_VM<1..0>.

LN_Z_CNT<7..0>

Next-to-back-door level Z count for the load pipe. See LQ_Z_CNT<7..0>.

LQ_ACTIVE

Back-door level active signal for the load pipe. Indicates to the VRFs that an operation is occuring at the back-door level on the load pipe, and that other signals at this level are valid.

LQ_FIRST_WR

Indicates that the load pipe has made its first result write to the VRFs. This is used by the VD logic in its chaining check.

LQ_LAST_SXV

Indicates that the load pipe has made its last SXV transfer for an instruction and is ready to surrender its claim to the SXV port.

LQ_LAST_WR

Indicates that the last vector register write of an instruction is finished and the pipe is ready to surrender the VRF write ports.

LQ_LD_BD

Indicates that an load pipe operation is beginning to use the back door controller.

LQ_MXV_POP

Indicates that the load pipe is removing a peice of data from the MXV queue.

LQ_PIPE_CTL<2..0>

Back-door level pipe control signal for the load pipe. See LF_PIPE_CTL<2..0> for description.

LQ_RSLT_DAT<63..0>

LQ_RSLT_PAR<7..0>

Return data and parityfor the load pipe. This is for MXV or SXV data, depending on the condition of LQ_RSLT_SEL<1..0>.

LQ_RSLT_SEL<1..0>

Selects the type of data to be placed on the LQ_RSLT_DAT<63..0> bus.

SEL<1..0>	Data
0	MXV data
1	Word duplicated MXV data (for loads of VM)
2	SXV data
3	Word duplicated SXV data (for loads of VM)

LQ_SXV_POP

Indicates that the load pipe is removing a peice of data from the SXV queue.

LQ_VM<1..0>

VM bits for the load pipe at the back door level. These tell the NVRF whether to write back the results or not during an under-mask operation.

LQ_Z_CNT<6..0>

Back door level Z count for the load pipe. The Z count is the VRF write port address that will be used for returning data (LQ_RSLT_DAT).

L_ACTIVE

Load pipe active signal from the load pipe controller. Indicates that the load pipe controller (UL) is active and sending valid control signals.

L_CLR_ACTIVE

Indicates to the NVD that the L_ACTIVE signal is about to go away. The NVD array keeps its own internal copy of L_ACTIVE for timing reasons and needs to know when to shut it off.

L_DISPATCH

Indicates that an instruction is being dispatched on the load pipe. All of the dispatch information is valid.

L_DST_CTL<2..0>

Directs the NVM gate arrays as to where the data on the LQ_RSLT_DAT<31..0> bus should be written.

CTL<2..0>	Write to
3	VS<31..0>
4	VM<127..96>
5	VM<95..64>
6	VM<63..32>
7	VM<31..0>

L_LAST_RD

Indicates that the last vector register read of an instruction is finished and the pipe is ready to surrender the VRF read ports.

L_LAST_SXV

Indicates that the last SXV transfer for an instruction is complete and that the load pipe is ready to surrender its claim (if it made one) on the SXV port.

L_LAST_VXS

Indicates that the last VXS transfer for an instruction is complete and that the load pipe is ready to surrender its claim (if it made one) on the VXS port.

L_LAST_WR

Indicates that the last vector register write of an instruction is finished and the pipe is ready to surrender the VRF write ports.

L_OP0_CTL

Causes the NVRF arrays to select what data will be output on the L3_OP0_DAT bus.

OP0_CTL	L3_OP0_DAT
0	VRF X port data (Normal operations)
1	VM data (Stores of VM)

L_OP1_CTL

Causes the NVRF arrays to select what data will be output on the L3_OP1_DAT bus.

OP1_CTL	L3_OP1_DAT
0	VRF Y port data (Normal operations)
1	Address index (stores under mask)

L_RDY

Load pipe controller ready for dispatch.

L_SRC_CTL<2..0>

Causes the NVM arrays to select what data will be put out on the VM_DAT<31..0> bus. This bus is passed through the NVRF arrays and out onto the L3_OP?_DAT buses.

CTL<2..0>	Data
0	Address index (Load/Store under mask)
2	VM<127..64> (Store VM upper)
3	VM<63..0> (Store VM lower)
4	SXV<31..0> (Move scalar to vector element. Load VM.)
5	SXV<63..32> (Move scalar to vector element. Load VM.)

L_UADDR<9..0>

Current micro-address of the UL controller.

L_UIR_PAR_ERR

Parity error signal for the load pipe micro-controller's micro-instruction register. Part of this register is internal to the L_VM array and part of it is in a register in the UL controller.

L_UIR_PAR_EXT

A parity syndrome bit for the externally stored portion of the load pipe micro-instruction register. This signal goes into the L_VM array to be combined with the parity syndrome on the internal portion of the load pipe micro-instruction register. The result is L_UIR_PAR_ERR.

L_UPC<9..0>

A registered version of the L_UADDR<9..0> signal. This register is stopped in the event of a parity error in the L_VM gate array. If the parity error is in the micro-instruction register, L_UPC<9..0> should have the address from which the bad data came.

L_X_CNT<6..0>

X port vector register address for the load pipe. Tells the NVRFs which element to pull out of the register file.

L_Y_CNT<6..0>

Y port vector register address for the load pipe. Tells the NVRFs which element to pull out of the register file.

M1_ACTIVE

Level 1 active signal for the multiply pipe. Goes directly from NVM to NQ array to be entered in the control queue. After being delayed to the M3 level, it controls loading of the m_queue internal to the NQ array.

M1_LAST_ELEM

Level 1 "last element" signal for the multiply pipe. Goes directly from NVM to NQ array to be entered in the control queue. See MQ_LAST_ELEM.

M1_PIPE_CTL<2..0>

Level 1 pipe control signal for the multiply pipe. Goes directly from NVM to NQ array to be entered in the control queue. See MQ_PIPE_CTL.

M1_PL<2..0>

Level 1 pipe length signal for the multiply pipe. Goes directly from NVM to NQ array to be entered in the control queue. The pipe length is used in the NQ array to set the depth of the control queue. The value of M1_PL<2..0> is equal to the number of clocks from the M3 level to the MQ level.

M1_RATE_2X

Level 1 rate 2x signal for the multiply pipe. Goes directly from NVM to NQ array to be entered in the control queue. See MQ_RATE_2X.

M1_VM<1..0>

Level 1 VM bits for the multiply pipe. Goes directly from NVM to NQ array to be entered in the control queue. See MQ_VM<1..0>.

M1_Z_CNT<7..0>

Level 1 Z count signal for the multiply pipe. The Z count is the VRF write port address that will be used for returning data. Goes directly from NVM to NQ array to be entered in the control queue. See MQ_Z_CNT<7..0>.

M2_ACTIVE

Level 2 active signal for the multiply pipe. Along with M2_PIPE_CTL, tells the OSCTL that the multiply pipe is invoved in a VXS transfer.

M2_EDIT_VM

Level 2 VM bit for the multiply pipe used in vector edit operations (e.g., MASK, MERGE, COMPRESS).

M2_IEEE

Level 2 IEEE bit for the multiply pipe. Goes from the NQ to the MFC controller. Tells the MFC to perform IEEE floating point operations if asserted.

M2_OPCODE<9..0>

Level 2 opcode bits for the multiply pipe. See the gate array appendix of this spec for opcode definitions.

M2_PIPE_CTL<2..0>

Along with M2_ACTIVE, tells the OSCTL controller that a VXS transfer is taking place on the multiply pipe if M2_PIPE_CTL = 0x7. Other values are don't-care for the OSCTL controller.

M2_PL<2..0>

Tells the MFC controller the pipe length of the current operation. The PL is equal to the number of clocks from the A3 level to the AQ level.

M2_RATE_2X

Tells the MFC controller whether the current operation is a 2x or 1x operation.

M2_START

Tells the MFC controller that a piece of data is available at level 2 in the multiply pipe.

M2_VM<1..0>

These VM bits enable the status bits coming out of the function units in the multiply pipe.

M2_VXS

Tells the NVRF arrays that a piece of scalar data from the multiply pipe should be put out on the L3_OP0_DAT bus for a VXS transfer.

M3_IEEE

Delayed version of the IEEE bit in the MFC controller to align it with the data that will be coming from the VRF on the M3_OP?_DAT bus.

M3_OP0_DAT<63..0>

M3_OP1_DAT<63..0>

Multiply pipe operands coming from the VRFs to the multiply pipe function units. The three indicates

that the data is at level 3 of the pipeline.

M3_OP0_PAR<7..0>

M3_OP1_PAR<7..0>

Multiply pipe operand parity coming from the VRFS to the multiply pipe function units.

M3_OP0_NIBBLE_PAR<7..0>

M3_OP1_NIBBLE_PAR<7..0>

M4_OP0_NIB_PAR<7..0>

M4_OP1_NIB_PAR<7..0>

M5_OP0_NIB_PAR<7..0>

M5_OP1_NIB_PAR<7..0>

See the definitions for the corresponding add pipe signals.

MFC_SCAN_OUT

Scan out signal from the MFC controller. Goes to the scan input of the AFC controller.

MFP_PAR_ERR<5..0>

Parity error signals for all of the MFP function units.

MF_BR_ADDR<9..0>

Branch address for the UM controller.

MF_BR_INST<2..0>

Branch type for the UM controller. See AF_BR_INST<2..0> for bit definitions.

MF_CNT_CTL<3..0>

Type of address sequence for VM to produce for the NVRFs. See AF_CNT_CTL<3..0> for bit definitions.

MF_EXT<8..0>

Control store signal grouping that contains un-registered versions of parity, M_OP0_CTL<2..0>, M_OP1_CTL<2..0>, and two spare bits.

MF_LAST<2..0>

Indicates to the NVM the last microcode cycle is occuring so that it can generate the last_rd, last_vxs, last_wr, last_elem, last_sxv control signals. See AF_LAST<2..0> for bit definitions.

MF_PIPE_CTL<2..0>

Multiply pipe control codes. See AF_PIPE_CTL<2..0> for bit definitions.

MF_TEST_POL

Selects whether a conditional microcode branch condition is true or false for the UM controller.

MF_TEST_SEL<2..0>

Selects what type of condition is used for a conditional microcode branch on the UM controller. See AF_TEST_SEL<2..0> for bit definitions.

MQ_ACTIVE

Back-door level active signal for the multiply pipe. Indicates to various parts of the NVP that an operation is indeed occuring at the back-door level on the multiply pipe, and that other signals at this level are valid.

MQ_CT<1..0>

VM bits generated by the multiply pipe during compare operations. These will be written into the VM register.

MQ_FIRST_WR

Indicates that the multiply pipe has made its first result write to the VRFs. This is used by the VD logic in its chaining check.

MQ_LAST_ELEM

Indicates that the element at the back-door level of the multiply pipe is the last element in an operation.

MQ_LAST_WR

Indicates that the last write to a vector register of an operation is taking place on the multiply pipe.

MQ_LD_BD

Indicates that an multiply pipe operation is beginning to use the back door controller.

MQ_OV<5..0>

Overflow status signals from all of the function units on the multiply pipe.

MQ_PIPE_CTL<2..0>

Back door level pipe control signal for the multiply pipe. See MF_PIPE_CTL<2..0> for description.

MQ_RATE_2X

Indicates that the operation at the back door level of the multiply pipe is a rate_2x operation.

MQ_RO<5..0>

Floating point reserved-operand status signal from all of the function units on the multiply pipe.

MQ_RSLT_DAT<63..0>

Result data from the multiply function pipe function units.

MQ_RSLT_PAR<7..0>

Result bus parity from the multiply function pipe function units.

MQ_SIV<4..0>

Integer overflow status signals from the function units on the multiply pipe.

MQ_SOURCE<4..0>

This is the registered version of the last level of the MFC control queue. These bits indicate the source of the data that is on the MQ_RSLT_DAT<63..0> bus. If the result data causes a parity error at the NVRF gate arrays, the source of the error can be determined.

SOURCE<4..3>	Function unit type
0	NFAD
1	NMISC
2	NMUL
3	it's broken

SOURCE<2>	Rate_2x

SOURCE<1..0>	Pointer to NMUL
0	NMUL0 (and NMUL1 if Rate_2x)
1	NMUL1
2	NMUL2 (and NMUL3 if Rate_2x)
3	NMUL3

MQ_UN<5..0>

Underflow status signals from the function units on the multiply pipe.

MQ_VM<1..0>

VM bits for the multiply pipe at the back door level. These tell the NVRF whether to write back the results or not during an under-mask operation.

MQ_Z_CNT<7..0>

Back door level Z count for the multiply pipe. The Z count is the VRF write port address that will be used for returning data (MQ_RSLT_DAT).

MXV_DVAL

Indicates that there is valid data in the MXV queue.

M_ACTIVE

Multiply pipe active signal from the multiply pipe controller. Indicates that the multiply pipe controller (UM) is active and sending valid control signals.

M_CLR_ACTIVE

Indicates to the NVD that the M_ACTIVE signal is about to go away. The NVD array keeps its own internal copy of M_ACTIVE for timing reasons and needs to know when to shut it off.

M_DISP

A vector dispatch signal that indicates that the instruction in dispatch can run on the multiply pipe.

M_DISPATCH

Indicates that an instruction is being dispatched on the multiply pipe. All of the dispatch information is valid.

M_LAST_RD

Indicates that the last vector register read of an instruction is finished and the multiply pipe is ready to surrender the VRF read ports.

M_LAST_VXS

Indicates that the last VXS transfer for an instruction is complete and that the multiply pipe is ready to surrender its claim (if it made one) on the VXS port.

M_LAST_WR

Indicates that the last vector register write of an instruction is finished and the multiply pipe is ready to surrender the VRF write ports.

M_MUX_SEL

Selects either M_RSLT_DAT_1<63..0> (if asserted) or M_RSLT_DAT_0<63..0> to be switched onto the MQ_RSLT_DAT<63..0> bus.

M_OP0_CTL<2..0>

Operand 0 control for the NVRF arrays. This selects what type of data will be placed on the M3_OP0_DAT<63..0> bus. See A_OP0_CTL<2..0> for bit definitions.

M_OP1_CTL<2..0>

Operand 1 control for the NVRF arrays. This selects what type of data will be placed on the M3_OP1_DAT<63..0> bus. See A_OP1_CTL<2..0> for bit definitions.

M_RDY

Multiply pipe controller ready for dispatch.

M_RSLT_DAT_0<63..0>

Multiply pipe result data from either NMUL0, NMUL1 or NFAD_A.

M_RSLT_DAT_1<63..0>

Multiply pipe result data from either NMUL3, NMUL4 or NMISC_A.

M_RSLT_PAR_0<7..0>

Multiply pipe result data parity from either NMUL0, NMUL1 or NFAD_A.

M_RSLT_PAR_1<7..0>

Multiply pipe result data from parity either NMUL3, NMUL4 or NMISC_A.

M_UADDR<9..0>

Current micro-address of the UM controller.

M_UIR_PAR_ERR

Parity error signal for the multiply pipe micro-controller's micro-instruction register. Part of this register is internal to the M_VM array and part of it is in a register in the UM controller.

M_UIR_PAR_EXT

A parity syndrome bit for the externally stored portion of the multiply pipe micro-instruction register. This signal goes into the M_VM array to be combined with the parity syndrome on the internal portion of the multiply pipe micro-instruction register. The result is M_UIR_PAR_ERR.

M_UPC<9..0>

A registered version of the M_UADDR<9..0> signal. This register is stopped in the event of a parity error in the M_VM gate array. If the parity error is in the micro-instruction register, M_UPC<9..0> should have the address from which the bad data came.

M_VM_DAT<31..0>

M_VM_PAR<3..0>

This is the M_VM array's version of the VM_DAT<31..0> bus. It is used to allow the loading of the UM control store during system initialization. It is not used in normal system operation.

M_X_CNT<6..0>

X port vector register address for the multiply pipe. Tells the NVRFs which element to pull out of the register file.

M_Y_CNT<6..0>

Y port vector register address for the multiply pipe. Tells the NVRFs which element to pull out of the register file.

NDIV_ODD_OPS

Used during a rate_2x NDIV operation to cause the NDIVs that it is connected to to take their 32-bit operands from the upper half of the operand buses.

NDIV_ODD_RSLT

Used during a rate_2x NDIV operation to cause the NDIVs that it is connected to to take their 32-bit results from the upper half of the result bus.

NDIV_RSLT_LOE_E_0

NDIV_RSLT_LOE_E_1

NDIV_RSLT_LOE_O_0

NDIV_RSLT_LOE_O_1

NDIV_RSLT_UOE_E_0

NDIV_RSLT_UOE_E_1

NDIV_RSLT_UOE_O_0

NDIV_RSLT_UOE_O_1

Used to enable the output of the NDIVs on the next. The UOEs enable the upper word of the 64-bit bus and the LOEs enable the lower half. The "E" or "O" in the postfix indicates whether the NDIV supplies the even or odd word in a rate_2x operation. The "0" or "1" at the end indicates whether the NDIV drives the A_RSLT_DAT_0 or A_RSLT_DAT_1 bus.

NDIV_RSLT_NEXT_0

NDIV_RSLT_NEXT_1

NDIV_RSLT_NEXT_2

NDIV_RSLT_NEXT_3

NDIV_RSLT_NEXT_4

NDIV_RSLT_NEXT_5

Indicates that the NDIV (with the same number suffix as this signal) will have a result available on the next cycle. The AFC controller uses this signal to determine if it needs to clock extend the NVP if a result will not be available.

NDIV_RSLT_SEL_0

NDIV_RSLT_SEL_1

NDIV_RSLT_SEL_2

NDIV_RSLT_SEL_3

NDIV_RSLT_SEL_4

NDIV_RSLT_SEL_5

Indicates that the result data is being taken from the NDIV (with the same number suffix as this signal). The NDIVs use this to manage their output register.

NDIV_START_0

NDIV_START_1

NDIV_START_2

NDIV_START_3

NDIV_START_4

NDIV_START_5

Start an operation in the NDIV (with the same number suffix as this signal).

NDIV_UIR2_VAL

Indicates that the operation started on an NDIV on the last cycle should proceed. This signal will be de-asserted when an NDIV operation was started on the last cycle, but the NVP went into a clock extend state which caused the NDIV to get bad data.

(AFP) NFAD_RSLT_OE

(MFP) NFAD_RSLT_OE

Output enable for the NFAD function units.

(AFP) NFAD_RSLT_SEL

(MFP) NFAD_RSLT_SEL

Indicates that the result data is being taken from the NFAD function unit. An NFAD will clear its status bits after assertion of this signal. This avoids having spurious status driving the status returns after an operation is done.

(AFP) NFAD_START

(MFP) NFAD_START

Start an operation on an NFAD.

(AFP) NMISC_RSLT_NEXT

(MFP) NMISC_RSLT_NEXT

Indicates that a result will be available on an NMISC next cycle. This is tied directly back to the NMISC_RSLT_SEL on the NMISC gate array to enable the ouputs.

(AFP) NMISC_RSLT_OE

(MFP) NMISC_RSLT_OE

Output enable for the NMISC function units.

(AFP) NMISC_START

(MFP) NMISC_START

Indicates that an operation is being started on an NMISC function unit.

NMUL_ODD_OPS_1

NMUL_ODD_OPS_3

Used during a rate_2x NMUL operation to cause the NMULs that it is connected to to take their 32-bit operands from the upper half of the operand buses.

NMUL_ODD_RSLT_1

NMUL_ODD_RSLT_3

Used during a rate_2x NMUL operation to cause the NMULs that it is connected to to take their 32-bit results from the upper half of the result bus.

NMUL_RSLT_LOE_0

NMUL_RSLT_LOE_1

NMUL_RSLT_LOE_2

NMUL_RSLT_LOE_3

NMUL_RSLT_UOE_0

NMUL_RSLT_UOE_1

NMUL_RSLT_UOE_2

NMUL_RSLT_UOE_3

Used to enable the output of the NMULs on the next. The UOEs enable the upper word of the 64-bit bus and the LOEs enable the lower half. The postfix indicates which of the four NMULs is to be selected.

NMUL_RSLT_NEXT_0

NMUL_RSLT_NEXT_1

NMUL_RSLT_NEXT_2

NMUL_RSLT_NEXT_3

Indicates that a result will be available on an NMUL next cycle. This is tied directly back to the NMUL_RSLT_SEL on the NMUL gate array to enable the ouputs.

NMUL_START_0

NMUL_START_1

NMUL_START_2

NMUL_START_3

Indicates that an operation is to be started on an NMUL function unit.

NVD_DISP_OK

A vector dispatch signal that indicates that all of the hazard checks, with the exception of the SXV port check, have been completed inside the NVD gate array.

NVD_IDLE

Indicates that the NVD gate array believes that the vector processor is idle. The NVD array keeps track of whether the front and back doors are busy for each of the three pipes.

NVD_PSW_HAZ

Indicates that an instruction that is in dispatch or running has a PSW hazard associated with it.

NVP_CLK_SCAN_OUT

Scan out from the NVP_CLK logic.

OPCODE<9..0>

A dispatch signal that is the opcode for the function units. The pipe controllers will pass it along to the function units.

OSCTL_SCAN_OUT

Scan out from the OSCTL controller. Goes to the scan input for the ISCTL controller.

OS_CK_HOLD

This signal is generated by the OSCTL controller and is used to generate clock extends. It indicates that the NVP is attempting a VXS or VXM transfer and that the NSP is not accepting the data.

PART_SUCCEED_NOK

PART_SUCCEED_OK

These two signals are partially formed versions of DISP_SUCCEED, without the NVD_DISP_OK term included. The creation of DISP_SUCCEED for the state machine is done in the PAL 181-005266 for timing reasons. NVD_DISP_OK is a critical signal and cannot go through two PALs and make timing. The state machine equations are too complicated to fit in the PAL without having DISP_SUCCEED partially formed elsewhere.

PHASE_GEN_1X<5..0>

This six bit register is the clock gating ring. This ring normally contains 0b011011 which circulates around the ring, gating the 3x system clock down to a 1x clock to be used by everything on the NVP except the NDIV arrays.

PL<2..0>

A dispatch signal that indicates the pipe length of an instruction for add and multiply pipe operations.

(AFC) QUEUE_SEL<4..0>

(MFC) QUEUE_SEL<2..0>

These signals manage the control queues in the AFC and MFC controllers. They select the point of insertion into the control queue.

R.ABD_STATE

Indicates that something is running in the back-door section of the add pipe.

R.A_PTR<2..0>

A modulo 6 counter that indicates which of the NDIV arrays should be started during the next cycle (if it is a divide/square-root operation). This pointer is put into the control queue and selects which NDIV the result should come from when it pops out the other end.

R.A_START

The function unit start signal. This is at the A3 pipe level.

R.BCNT<3..0>

Block count for the context controller. The block count is decremented each time the context controller starts scanning a new block.

R.BDONE

Indicates that the block count equals 1. When this block is finished the context scan is complete.

R.CNTX_CTL<1..0>

Registered version of SP_VP.CNTX_CTL<1..0>

R.D0_BUSY

R.D1_BUSY

R.D2_BUSY

R.D3_BUSY

R.D4_BUSY

R.D5_BUSY

Indicates that the same numbered NDIV is running but is not ready to produce a result.

R.DISP_SUCCEED

Registered version of the DISP_SUCCEED signal. Indicates to the NVD array that the dispatch cycle is complete.

R.DISP_VL0

The current instruction should dispatch even if VL=0.

(AFC) R.FU_TYPE<1..0>

(MFC) R.FU_TYPE<1..0>

Indicates which type of function unit on which the instruction is supposed to run. This is at the third pipe level.

TYPE<1..0>	Function unit type
0	NFAD
1	NMISC
2	NMUL on multiply pipe, NDIV on add pipe
3	it's broken

R.IP_DVAL

Indicates that there is a valid instruction in the IPCTL queue.

R.IP_FULL

Indicates that there are 3 instructions in the IPCTL queue.

R.IP_RD<1..0>

The read pointer for the IPCTL queue.

R.IP_REQ_PREV

A state machine signal for the IPCTL controller indicating that the IPCTL asserted VP_SP.DISP_REQ_NEXT on the previous cycle.

R.IP_TWO

Indicates that there are 2 instructions in the IPCTL queue.

R.IP_WR<1..0>

The write pointer for the IPCTL queue.

R.KILL_DISP

Keeps the dispatch signals from being asserted when the VD state machine is not in state 2.

R.LBD_STATE

Indicates that something is running in the back-door section of the load pipe.

R.LCNT<7..0>

Block length count for the context controller. It is loaded at the beginning of each new block and then decremented each un-extended clock during a context switch.

R.LDONE

Indicates that block length count is zero. This means the block is done.

R.LOAD_DAT<25..0>

An external register that is used solely for the purpose of loading the VD dispatch RAM during system initialization.

R.LOAD_VL

Indicates that the VL register is to be loaded.

R.LQ_FULL

Registered version of LN_FULL. At the back-door level.

R.LQ_LAST_ELEM

Registered version of LN_LAST_ELEM. At the back-door level.

R.LQ_PIPED_UP

Registered version of LN_PIPED_UP. At the back-door level.

R.L_DISP_REQ

Indicates that the instruction in dispatch is to run on the load pipe.

R.MBD_STATE

Indicates that something is running in the back-door section of the multiply pipe.

R.M_PTR<1..0>

A modulo 4 counter that indicates which of the NMUL arrays should be started during the next cycle (if it is a multiply operation). This pointer is put into the control queue and selects which NMUL the result should come from when it pops out the other end.

R.M_START

The function unit start signal. This is at the M3 pipe level.

R.NDIV_START<5..0>

Indicates which of the NDIVs was started on the last cycle.

(AFC) R.PL<2..0>

(MFC) R.PL<2..0>

Pipe-length of the instruction running in the function pipe. This signal is at the third pipe level.

R.PSW0

R.PSW1

R.PSW2

These three registers correspond to the three entries in the IPCTL queue. The R.PSW? bit is the SP_VP.DISP_PSW_HAZ bit for the corresponding entry in the IPCTL queue.

R.RAMREG_PSW_HAZ

Indicates that an instruction is in dispatch and that it has a psw hazard associated with it.

R.RAM_PSW_HAZ

Indicates that an instruction is in the first cycle of dispatch and that it has a psw hazard associated with it.

R.RAM_REG_EN

Enables loading of the micro-instruction register for the dispatch RAM. This occurs only on the first cycle of dispatch.

(AFC) R.RATE_2X

(MFC) R.RATE_2X

Indicates whether the instruction running on the function pipe is running at rate_2x. This signal is at the third pipe level.

R.REQ_SXV

A VD UIR signal. Indicates that the dispatching instruction requires one or more SXV transfers.

R.REQ_SXV_POP

A VD UIR signal. Indicates that the vector dispatch unit will take data from the SXV queue.

R.REQ_SXV_WAIT

A VD UIR signal. Indicates that the vector dispatch must waid for SXV_DVAL before dispatching the instruction to the function pipe.

R.SAVE_LAST_NEXT
: Asserted when R.BCNT<3..0> = 1. Indicates that the context switch is almost through.

R.SAVE_RDY_NEXT
: Indicates that the context state machine is running and has data to send to the NSP. This is asserted when the context state R.STATE<2..0> = 5.

R.SAVE_VM_NEXT
: Registered to produce SAVE_VM which is passed through OSCTL to the NSP as the VM bit to go with the context store data.

R.SCAN_IN
: Registered version of XC_VP.SCAN_IN. This register helps system timing, and is also required since the last bit coming in during scan is indeterminate and cannot be used for any function requiring known state.

R.STATE<2..0>
: The state of the context controller state machine.

R.UA_WE

R.UL_WE

R.UM_WE
: Scan ring signals used to generate write enables for the UA, UL and UM control stores.

R.VD_MACH_BUSY
: Indicates that the VD state machine is in a dispatch cycle (not in state 0).

R.VD_STATE<2..0>
: State of the VD state machine.

R.VD_WE
: Scan ring signal used to generate write enables for the VD control store.

R.VL0*
: Active low signal that indicates that VL=0 (when low).

R.VL_EN
: Enable signal for the VL register.

R.VXM_RDY
: Indicates that the NVP has VXM data to transfer (for memory). This signal becomes VP_SP.VXM_RDY when not in context mode.

R.VXS_RDY
: Indicates that the NVP has VXS data to transfer (for memory). This signal becomes VP_SP.VXM_RDY when not in context mode.

R.VX_REQ
: Registered version of SP_VP.VX_REQ_NEXT. The OSCTL controller uses this signal to determine whether the NSP is accepting data from the NVP.

R.WCS_WE
: Control store write enable. This is on the scan ring and is used to enable writes to the control store during system initialization.

R.WCS_WP<1..0>
: Control store write pulse signal. This two bit register rotates allowing a train of single-cycle write pulses.

RAM_STOP
: The enable signal for the portion of the dispatch RAM UIR that is external to the NVD gate array.

RATE_2X
: A dispatch signal that indicates (if active) that the dispatching instruction is to run at rate_2x.

RATE_2X_OK
: A test point signal from the NVD array that indicates that the instruction in dispatch has passed the

rate_2x check.

READ_PORTS_OK
: A test point signal from the NVD array that indicates that the instruction in dispatch has passed the read port allocation.

RSLT_PARITY_ENABLE
: Enables parity errors on the result bus to the NVRFs (RSLT_PAR_ERR). The parity is checked across all eight NVRFs and is combined externally, where this signal can disable it.

RSLT_PART_SYND_L<7..0>

RSLT_PART_SYND_U<7..0>
: Nibble parity syndrome on the result buses returning to the NVRFs.

RSLT_PAR_ERR
: Indicates a parity error on either AQ_RSLT_DAT, MQ_RSLT_DAT or LQ_RSLT_DAT. This causes the result registers to stop on the NVRFs, and assertion of VP_XC.HARD_ERROR. It can be disabled by RSLT_PARITY_ENABLE.

RSLT_SYND_ERR
: Indicates a parity error on either AQ_RSLT_DAT, MQ_RSLT_DAT or LQ_RSLT_DAT.

SAVE_LAST
: This signal becomes VP_SP.VXA_LAST during a context save. It indicates the last transfer of a context save.

SAVE_RDY
: This signal becomes VP_SP.VXM_RDY during a context save. It indicates that context save data from the NVP is available.

SAVE_VM
: This signal becomes VP_SP.VXA_VM<0> during a context save. It indicates that the data being stored is valid.

SCAN_IN_BUS<31..0>
: The scan_in signals for all of the gate arrays, and the scan_in for the discrete ring (EXT_SCAN_IN). See the IS_MUX drawing for the ordering of this bus.

SCAN_LEFT
: This is the scan control signal for the discrete registers (100e141 and 100e142).

SP_VP.CNTX_CTL<1..0>
: Context control signal from the NSP.

CTL<1..0>	mode
0	Normal mode
1	Context hold
2	Context reset
3	Context left shift

SP_VP.DATA<63..0>
: Data bus from the NSP. This bus carries both SXV and MXV transfers. It is parity protected by SP_VP.PAR<7..0>.

SP_VP.DISP_EP<10..0>
: Entry point for a vector instruction from the NSP. This uniquely identifies the type of instruction. It is used as an entry point into the VD dispatch RAM.

SP_VP.DISP_IREG<2..0>

SP_VP.DISP_JREG<2..0>

SP_VP.DISP_KREG<2..0>
: Register numbers for a vector instruction from the NSP. These correspond to the Vi, Vj and Vk fields of a vector instruction.

SP_VP.DISP_PSW_HAZ
: PSW hazard indication for a vector instruction from the NSP. Asserted if the instruction to be dispatched causes a PSW hazard.

SP_VP.DISP_RDY

Indicates that the NSP has an instruction available to be run on the NVP.

SP_VP.IEEE

IEEE mode bit for a vector instruction. Asserted if the instruction to be dispatched should run in IEEE mode.

SP_VP.MXV_RDY

Indicates that the NSP has memory (MXV) data available for the NVP.

SP_VP.PAR<7..0>

Parity bits for SP_VP.DATA<63..0>.

SP_VP.SXV_RDY

Indicates that the NSP has scalar (SXV) data available for the NVP.

SP_VP.VX_REQ_NEXT

Indicates that the NSP will accept a data transfer from the NVP on the next cycle if the NVP has data to send. VP_SP.VXM_RDY and VP_SP.VXS_RDY indicate whether the transfer is VXM or VXS respectively.

SQ_ACTIVE

Back-door level active signal for the NVM arrays. This is used during VM register writes to enable those writes.

SQ_CT<1..0>

Back-door level result VM bits for the NVM arrays. This is the actual data to be written into the VM register during VM register writes.

SQ_PIPE_CTL<2..0>

Back-door level pipe control signal for the NVM arrays. This is used during VM register writes to enable those writes or to clear the VM register above VL. It is generated by a multiplexer that selects either AQ_PIPE_CTL<2..0> or MQ_PIPE_CTL<2..0>. See AF_PIPE_CTL<2..0> for a decoding.

SQ_RATE_2X

Back-door level rate_2x signal for the NVM arrays. This is used during VM register writes to determine whether pairs of bits should be written (if active) or one bit at a time.

SQ_VM<1..0>

Back-door level control VM bits for the NVM arrays. These bits enable writing back into the VM register for compare-under-mask operations.

SQ_Z_CNT<7..0>

Back-door level address for the NVM arrays. This address determines what position in the NVM register is written to.

SRC_SIZ<2..0>

Source size. A dispatch signal that indicates the size of the data to be read out of the VRF. It is also used to determine which identity element to use. The destination size may be different than the source size for such operations as type converts.

SIZ<1..0>	Destination Size
0	byte
1	half-word
2	word
3	long-word
6	single precision
7	double precision

STAT_SCAN_OUT

Scan out from the STAT logic. Goes to the scan input of the OSCTL controller.

SXV_DAT<63..0>

Output of the SXV queue in normal operation. During a context save it provides context data and addresses that will be sent through the NVRFs and out on the L3_OP?_DATA buses. During a context restore it carries the context restore data that will be put on the scan_in pins of the gate arrays and discrete parts.

SXV_DVAL

Indicates that there is valid data on the SXV_DAT<63..0> bus.

SXV_FULL

Indicates that there are 3 entries in the SXV queue.

SXV_OK

A test point signal from the NVD array that indicates that the instruction in dispatch has passed the SXV port check.

SXV_PAR<7..0>

Parity bits for the SXV_DAT<63..0> bus.

SXV_RD<1..0>

The read pointer for the SXV queue.

SXV_REQ_NEXT

This signal becomes VP_SP.SXV_REQ_NEXT except during context mode, when VP_SP>SXV_REQ_NEXT is forced to zero.

SXV_TWO

Indicates that there are two entries in the SXV queue.

SXV_WR<1..0>

The write address for the SXV queue. Data on the SP_VP.DATA bus will be written at this address unless SXV_FULL is asserted.

UA_PAR_ERR

Indicates that there was a parity error on the UA micro-instruction register. Probably indicates bad data in the UA control store RAM.

UA_SCAN_OUT

Scan out from the UA logic. Goes to the scan input of the UL logic.

UL_PAR_ERR

Indicates that there was a parity error on the UL micro-instruction register. Probably indicates bad data in the UL control store RAM.

UL_SCAN_OUT

Scan out from the UL logic. Goes to the scan input of the BD logic.

UM_PAR_ERR

Indicates that there was a parity error on the UM micro-instruction register. Probably indicates bad data in the UM control store RAM.

UM_SCAN_OUT

Scan out from the UM logic. Goes to the scan input of the UA logic.

VD_PAR_ERR

Indicates that there was a parity error on the VD micro-instruction register. Probably indicates bad data in the VD dispatch RAM.

VD_POP

Indicates that the VD state machine is taking an instruction out of the IPCTL queue. This only happens when the VD state machine is in state 1.

VD_SXV_POP

Indicates that the vector dispatch unit is removing a piece of data from the SXV queue.

VL<7..0>

The vector length (VL) register.

VL_STOP

The enable for the VL register.

VM_DAT<63..0>

This bus is driven by the VM gate arrays, through the NVRFs, and out onto the L3_OP?_DAT buses. See L_SRC_CTL<1..0> for a description of what goes on this bus.

VM_FORCE

A dispatch signal that indicates (if active) that the dispatching instruction is to run not under-mask.

VM_OK

A test point signal from the NVD array that indicates that the instruction in dispatch has passed the multiple VM register checks.

VM_PAR<7..0>

Parity for the VM_DAT<63..0> bus.

VM_POL

A dispatch signal that sets the polarity of the instruction being dispatched, if it is an under-mask instruction. (1=TRUE; 0=FALSE).

VP_SP.DATA<63..0>

Data bus from the NVP to the NSP. It may carry either VXM or VXS data.

VP_SP.DISP_REQ_NEXT

Indicates that the NVP is ready to accept a vector instruction from the NSP on the next cycle.

VP_SP.IDLE

Indicates that the entire NVP is idle, and that there are no vector instruction dispatches pending in the IPCTL queue.

VP_SP.MXV_REQ_NEXT

Indicates that the NVP is ready to accept MXV data into the MXV queue on the next cycle.

VP_SP.PAR<7..0>

Parity bits for VP_SP.DATA<63..0>.

VP_SP.PSW_HAZ

Indicates that an instruction on the NVP (executing, dispatching, or in the IPCTL queue) has a PSW hazard associated with it.

VP_SP.SET_FDZ

An instruction executing on the NVP has caused a floating point divide-by-zero.

VP_SP.SET_FSN

An instruction executing on the NVP has caused a floating point square root of a negative number.

VP_SP.SET_OV

An instruction executing on the NVP has caused an overflow.

VP_SP.SET_RO

An instruction executing on the NVP has caused a floating point reserved operand.

VP_SP.SET_SDZ

An instruction executing on the NVP has caused a integer divide-by-zero.

VP_SP.SET_SIV

An instruction executing on the NVP has caused a integer overflow.

VP_SP.SET_UN

An instruction executing on the NVP has caused an underflow.

VP_SP.SXV_REQ_NEXT

Indicates that the NVP is ready to accept SXV data into the SXV queue on the next cycle.

VP_SP.VXA_ADDR<31..0>

Addresses or indexes for the vector address generator on the NSP. This bus is driven by data from a vector register during vector-of-index operations, by indexes from the NVM arrays during load/ stores under-mask, and by context addresses during context saves.

VP_SP.VXA_ADDR_PAR<3..0>

Parity for VP_SP.VXA_ADDR<31..0>

VP_SP.VXA_LAST

Indicates the last address or data transfer of a load or store operation or of a context save.

VP_SP.VXA_VM<1..0>

VM bits for addresses on the VP_SP.VXA_ADDR<31..0> bus. These bits tell the vector address generator on the NSP whether the addresses being transfered are valid. The addresses can be longword addresses. VM<0> applies to the MSW.

VP_SP.VXM_RDY

Indicates that the NVP is ready to transfer VXM (memory) data on the VP_SP.DATA<63..0> bus.

VP_SP.VXS_RDY

Indicates that the NVP is ready to transfer VXS (scalar) data on the VP_SP.DATA<63..0> bus.

VP_XC.HARD_ERROR

Indicates that the NVP has a parity error somewhere.

VP_XC.SCAN_OUT

Board scan out for the NVP.

VRF_CNTX_MODE

Indicates to the NVRF arrays that the NVP is going to go into (or out of) context mode on the next cycle.

VRF_GLUE_SCAN_OUT

Scan out from the discrete logic in the VRF72 body. Goes to the scan input of the STAT logic.

VRX<2..0>

A dispatch signal that tells the NVRFs which vector register the instruction will read from for X data. This is usually equivalent to the Vi register field of a vector instruction.

VRX_CLAIM

A dispatch signal that indicates that the instruction will actually use the X read register.

VRX_PORT

A dispatch signal that tells the NVRFs which of the two read ports (A or B) on a vector register pair will be used to read for X data.

VRY<2..0>

A dispatch signal that tells the NVRFs which vector register the instruction will read from for Y data. This is usually equivalent to the Vj register field of a vector instruction.

VRY_CLAIM

A dispatch signal that indicates that the instruction will actually use the Y read register.

VRY_PORT

A dispatch signal that tells the NVRFs which of the two read ports (A or B) on a vector register pair will be used to read for Y data.

VRZ<2..0>

A dispatch signal that tells the NVRFs which vector register the instruction will write its result data to. This is usually equivalent to the Vk register field of a vector instruction.

VRZ_CLAIM

A dispatch signal that indicates that the instruction will actually write result data back to VRZ<2..0>.

VXS_OK

A test point signal from the NVD array that indicates that the instruction in dispatch has passed the VXS port check.

WRITE_PORTS_OK

A test point signal from the NVD array that indicates that the instruction in dispatch has passed the VRF write port allocation.

XC_VP.CLOCK_3X

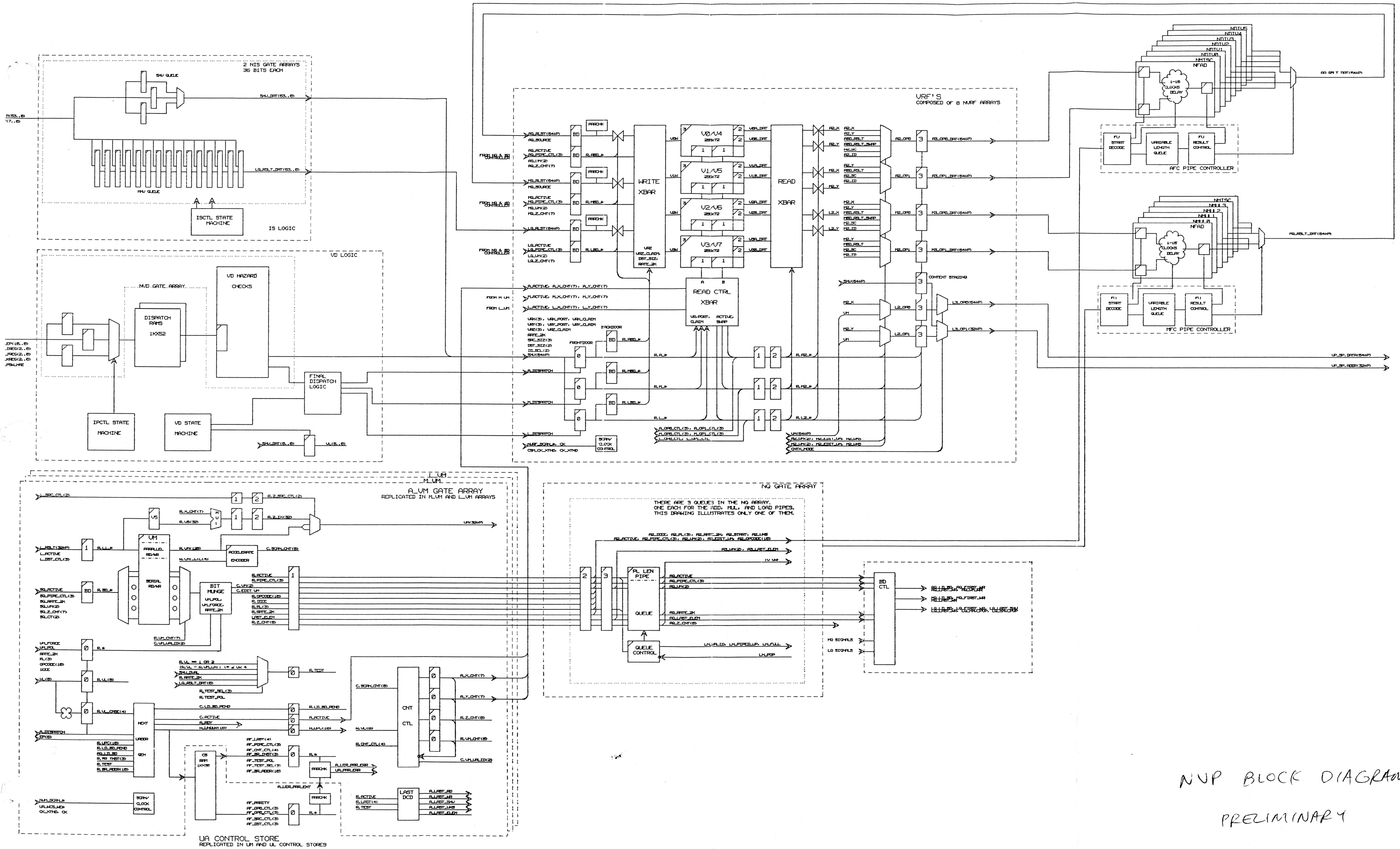
XC_VP.CLOCK_3X*

System clock to the NVP.

XC_VP.SCAN_CTL<2..0>

System level scan control.

CTL<2..0>	mode
0	NORMAL
5	SCAN_LAST_LEFT
6	SCAN_LOAD
7	SCAN_LEFT



2 NIS GATE ARRAYS
36 BITS EACH
ISCTL STATE MACHINE
IS LOGIC
VD LOGIC
NVD GATE ARRAY
DISPATCH RAMS
1KX52
VD HAZARD CHECKS
FINAL DISPATCH LOGIC
IPCTL STATE MACHINE
VD STATE MACHINE
VRF'S
COMPOSED OF 8 NVRF ARRAYS
WRITE XBAR
V0/V4
V1/V5
V2/V6
V3/V7
READ XBAR
READ CTRL XBAR
CONTEXT STAGING
NFAD
1-15 CLOCKS DELAY
VARIABLE LENGTH QUEUE
AFC PIPE CONTROLLER
MFC PIPE CONTROLLER
A_VM GATE ARRAY
REPLICATED IN M_VM AND L_VM ARRAYS
BIT MUNGE
CNT CTL
LAST DCD
UA CONTROL STORE
REPLICATED IN UM AND UL CONTROL STORES
NQ GATE ARRAY
THERE ARE 3 QUEUES IN THE NQ ARRAY.
ONE EACH FOR THE ADD, MUL, AND LOAD PIPES.
THIS DRAWING ILLUSTRATES ONLY ONE OF THEM.
PL LEN PIPE
QUEUE
QUEUE CONTROL
BD CTL
NVP BLOCK DIAGRAM
PRELIMINARY