10 VAX VECTOR ARCHITECTURE

This chapter describes an extension to the VAX architecture for integrated vector processing. Some VAX vector architecture departs from the traditional VAX scalar architecture, especially in the areas of UNPREDICTABLE results, vector processor exceptions, and instruction/memory synchronization.

10.1 Introduction to VAX Vector Architecture

Implementation of the VAX vector architecture is optional. VAX processors that do implement the vector architecture do so as specified in this chapter. Operating system software may emulate the vector architecture on processors that omit this feature.

On VAX processors that omit the vector architecture, vector instructions result in a reserved-instruction fault.

The vector architecture features include additional instructions, vector registers, and vector control registers.

All descriptions and examples of vector instructions in this chapter use the assembler notation form of instructions, as described in Section 10.5. The number and order of operands for the assembler notation differs from that defined in the instruction stream format. See Section 10.3 and Section 10.5 for additional information.

10.2 VAX Vector Architecture Registers

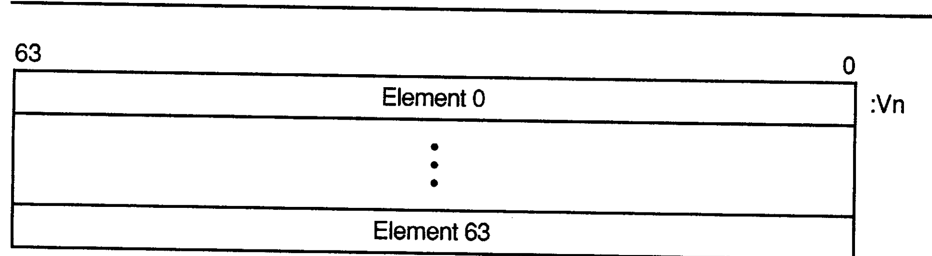
This section identifies and describes the vector, vector control, and internal processor registers used in processing vector architecture operations.

10.2.1 Vector Registers

There are 16 vector registers, V0 to V15. Each vector register contains 64 elements numbered 0 to 63. Each element is 64 bits wide. Figure 10–1 depicts a vector register.

A vector instruction that performs a register-to-register operation is defined as a **vector operate instruction**. A vector operate instruction that reads or writes F_floating data, or integer data for shifts or integer arithmetic operations, reads bits <31:0> of each source element and writes bits <31:0> of each destination element. Bits <63:32> of the destination are UNPREDICTABLE for F_floating, integer arithmetic, and shift instructions.

Figure 10–1 Vector Register



Vector logical instructions read bits <31:0> of each source element and write the result into bits <31:0> of each destination element; bits <63:32> of the destination element receive bits <63:32> of the corresponding element of the Vb source operand.

For vector instructions that read longword data from memory into a vector register (VLDL and VGATHL), bits <63:32> of the destination elements are UNPREDICTABLE.

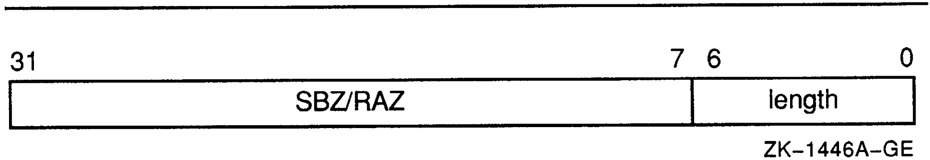
If the same vector register is used as both source and destination in a Gather Memory Data into Vector Register (VGATH) instruction, the result of the VGATH instruction is UNPREDICTABLE.

For the IOTA vector instruction, bits <63:32> of the destination elements are UNPREDICTABLE.

10.2.2 Vector Control Registers

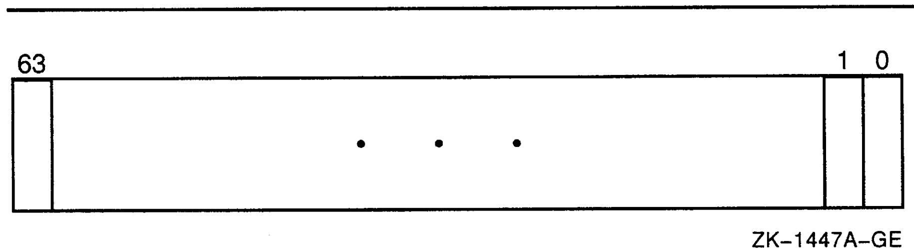
The 7-bit Vector Length Register (VLR), shown in Figure 10–2, limits the highest vector element to be processed by a vector instruction. VLR is loaded prior to executing the vector instruction using a Move to Vector Processor (MTVP) instruction. The value in VLR may range from 0 to 64. If the vector length is zero, no vector elements are processed. If a vector instruction is executed with a vector length greater than 64, the results are UNPREDICTABLE. Elements beyond the vector length in the destination vector register are not modified.

Figure 10–2 Vector Length Register (VLR)



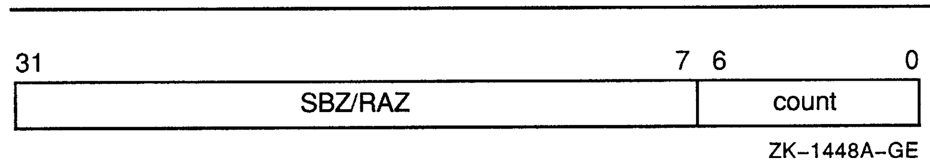
The Vector Mask Register (VMR), shown in Figure 10–3, has 64 bits, each corresponding to an element of a vector register. Bit <0> corresponds to vector element 0. See Section 10.3.1 for information on masked operations.

Figure 10–3 Vector Mask Register (VMR)



The 7-bit Vector Count Register (VCR), shown in Figure 10–4, receives the length of the offset vector generated by the IOTA instruction.

Figure 10–4 Vector Count Register (VCR)



These registers are read and written by Move from/to Vector Processor (MFVP/MTVP) instructions.

10.2.3 Internal Processor Registers

The vector processor contains the following internal processor registers (IPRs) that can be accessed by the scalar processor using MTPR/MFPR instructions:

- Vector Processor Status Register (VPSR)
- Vector Arithmetic Exception Register (VAER)
- Vector Memory Activity Check (VMAC)
- Vector Translation Buffer Invalidate All (VTBIA)

- Vector State Address Register (VSAR)

The VPSR is shown in Figure 10–5, and is described in Table 10–1.

Figure 10–5 Vector Processor Status Register (VPSR)

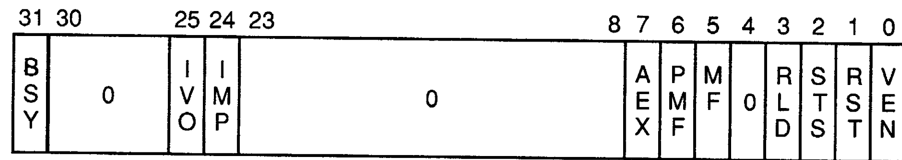


Table 10–1 Description of the Vector Processor Status Register (VPSR)

Extent	Type	Description
<0>	R/W	Vector Processor Enabled (VEN). The vector processor is enabled by writing a one to this bit. Writing a zero disables the vector processor. If VPSR<VEN> is cleared by software while VPSR<BSY> is set, then once the new state of VPSR becomes synchronized with subsequent vector instructions, no more instructions are sent and the vector processor completes execution of all pending instructions in its instruction queue. See Section 10.6.3, Vector Processor Disabled, for more details.
<1>	W	Vector Processor State Reset (RST). Writing a one to this bit clears VPSR and VAER. If VPSR<RST> is set by software while VPSR<BSY> is set, the operation of the vector processor is UNDEFINED. This bit is read as zero (RAZ).
<2>	W	Vector State Store (STS). Writing a one to this bit initiates storing of implementation-specific vector state information to memory using the address in VSAR for the asynchronous method of handling memory management exceptions. If the synchronous method is implemented, write operations to VPSR<STS> are ignored. This bit is RAZ.
<3>	W	Vector State Reload (RLD). Writing a one to this bit initiates reloading of implementation-specific vector state information from memory using the address in VSAR for the asynchronous method of handling memory management exceptions. If the synchronous method is implemented, write operations to VPSR<RLD> are ignored. This bit is RAZ.
<4>	R	0
<5>	R/W1C	Memory Fault (MF). This bit is set by the vector processor when there is a memory reference to be retried due to an asynchronous memory management exception. Writing a one to VPSR<MF> clears it. Writing a zero to VPSR<MF> has no effect. If the synchronous method of handling memory management exceptions is implemented, this bit is always zero.
<6>	R/W1C	Pending Memory Fault (PMF). This bit is set by the vector processor when an asynchronous memory management exception is pending. Writing a one to VPSR<PMF> clears it. Writing a zero to VPSR<PMF> has no effect. If the synchronous method of handling memory management exceptions is implemented, this bit is always zero.

(continued on next page)

Table 10–1 (Cont.) Description of the Vector Processor Status Register (VPSR)

Extent	Type	Description
<7>	R/W1C	Vector Arithmetic Exception (AEX). This bit is set by the vector processor when disabling itself due to an arithmetic exception. Information regarding the nature of the exception can be found in VAER. Writing a one to VPSR<AEX> clears VPSR<AEX> and VAER. Writing a zero to VPSR<AEX> has no effect.
<23:8>	R	0
<24>	R/W1C	Implementation-Specific Hardware Error (IMP). This bit is set by the vector processor when disabling itself due to an implementation-specific hardware error. Writing a one to VPSR<IMP> clears it. Writing a zero to VPSR<IMP> has no effect. An implementation may choose not to implement VPSR<IMP>. In this case, writing VPSR<IMP> with either value must have no effect and must not generate any error. Also, its value when read must be zero.
<25>	R/W1C	Illegal Vector Opcode (IVO). This bit is set by the vector processor when disabling itself due to receiving an illegal vector opcode. Writing a one to VPSR<IVO> clears it. Writing a zero to VPSR<IVO> has no effect. An implementation may choose not to implement VPSR<IVO>. In this case, writing VPSR<IVO> with either value must have no effect and must not generate any error. Also, its value when read must be zero.
<30:26>	R	0
<31>	R	Vector Processor Busy (BSY). When this bit is set, the vector processor is executing vector instructions. When it is clear, the vector processor is idle, or the vector processor has suspended instruction execution due to an asynchronous memory management exception or hardware error. Writing to VPSR<BSY> has no effect.

Table 10–2 shows the possible settings of VPSR<3:0> in the same MTPR instruction, and the resulting action for the vector processor. The state of the vector processor is determined by the encoding of Vector Processor Enabled (VPSR<VEN>) and Vector Processor Busy (VPSR<BSY>). The vector processor state for possible encodings is shown in Table 10–3.

Table 10–2 Possible VPSR<3:0> Settings for MTPR

RLD	STS	RST	VEN	Meaning
0	0	0	0	Disable vector processor
0	0	0	1	Enable vector processor
0	0	1	0	Reset state and disable vector processor
0	0	1	1	Reset state and enable vector processor
0	1	0	0	Store state (must disable vector processor)
1	0	0	0	Reload state and disable vector processor
1	0	0	1	Reload state and then enable vector processor

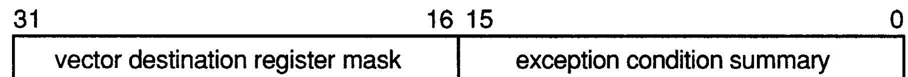
Table 10–3 State of the Vector Processor

VEN	BSY	Meaning
0	0	The vector processor is not executing any instructions now, and either has no pending instructions or will not execute pending instructions. No more instructions should be sent.
0	1	The vector processor is executing at least one pending instruction. No more instructions should be sent.
1	0	The vector processor is not executing any instructions now, and either has no pending instructions or will not execute pending instructions. New instructions can be sent to the vector processor.
1	1	The vector processor is executing at least one instruction now. New instructions can be sent.

Note that because the vector and scalar processors can execute asynchronously, a VPSR state transition may not be seen immediately by the scalar processor. After performing an MTPR to VPSR, software must then issue an MFPR from VPSR to ensure that the new state of VPSR (and VAER if cleared by VPSR<RST>) will affect the execution of subsequently issued vector instructions. The MFPR in this case will not complete until the new state of the vector processor becomes visible to the scalar processor. If software does not issue the MFPR, then it is UNPREDICTABLE whether this synchronization between the new state of VPSR (and VAER) and subsequently issued vector instructions occurs.

The VAER, shown in Figure 10–6, is a read-only register used to record information regarding vector arithmetic exceptions. Table 10–4 shows the encoding for the exception condition types. The destination register mask field of VAER records which vector registers have received default results due to arithmetic exceptions. VAER<16+n> corresponds to vector register Vn, where n is between 0 and 15. For more information, refer to Section 10.6.2, Vector Arithmetic Exceptions.

Figure 10–6 Vector Arithmetic Exception Register (VAER)



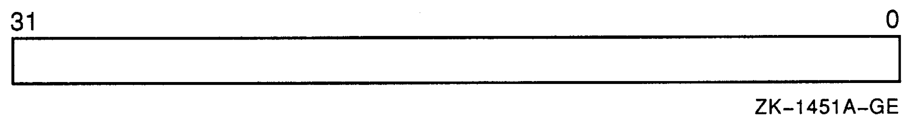
ZK–1450A–GE

Table 10–4 VAER Exception Condition Summary Word Encoding

Bit	Exception Condition
<0>	Floating underflow
<1>	Floating divide by zero
<2>	Floating reserved operand
<3>	Floating overflow
<4>	0
<5>	Integer overflow
<15:6>	0

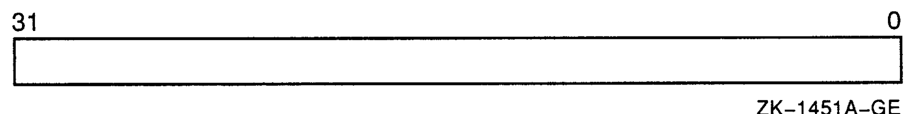
The Vector Memory Activity Check (VMAC) register, shown in Figure 10–7, is used to guarantee the completion of all prior vector memory accesses. For more information on this function of VMAC, refer to section Section 10.7.2.2. An MFPR from VMAC also ensures that all hardware errors encountered by previous vector memory instructions are reported before the MFPR completes. For more information on this function of VMAC, refer to Section 10.9, Hardware Errors. The value returned by MFPR from VMAC is UNPREDICTABLE.

Figure 10–7 Vector Memory Activity Check (VMAC) Register



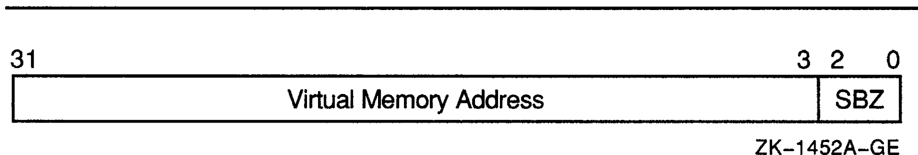
The Vector Translation Buffer Invalidate All (VTBIA) register, shown in Figure 10–8, is a write-only register that may be omitted in some implementations. If the vector processor contains its own translation buffer, moving zero into VTBIA using the MTPR instruction invalidates the entire vector translation buffer. For more information, refer to Section 10.8, Memory Management.

Figure 10–8 Vector Translation Buffer Invalidate All (VTBIA) Register



The Vector State Address Register (VSAR), shown in Figure 10–9, is a read/write register that contains a quadword-aligned virtual address of memory assigned by software for storing implementation-specific vector hardware state when the asynchronous method of handling memory management exceptions is implemented. The length of this memory area is implementation specific. Software must guarantee that accessing the memory pointed to by the address does not result in a memory management exception. If the synchronous method of handling memory management exceptions is implemented, this register is omitted. For more information, refer to Section 10.6.1, Vector Memory Management Exception Handling.

Figure 10–9 Vector State Address Register (VSAR)



With the exception of VPSR (and VAER), an MTPR to any other writable vector internal processor register (IPR) ensures that the new state of the IPR affects the execution of all subsequently issued vector instructions. Vector instructions issued before an MTPR to any writable vector IPR are unaffected by the new state of the IPR (and any implicitly changed vector IPR) except in one case: when the MTPR sets VPSR<RST> while VPSR<BSY> is set. (See Table 10–1 for more details.)

Except for the following two cases, the operations of the scalar and vector processors are UNDEFINED after execution of MTPR to a read-only vector IPR, MTPR to a nonexistent vector IPR, MTPR of a nonzero value to a MBZ field, or MTPR of a reserved value to a vector IPR. The preferred implementation is to cause reserved-operand fault.

- If an implementation supports an optional vector processor, but the vector processor is not installed, MTPR to VPSR has no effect.
- If an implementation supports an optional vector processor, but either the vector processor is not installed, or the scalar/vector processor pair uses a common translation buffer (TB), MTPR to VTBIA has no effect.

In each of these cases, MTPR is implemented as a no-op.

Except for the following two cases, the operations of the scalar and vector processors are UNDEFINED after execution of MFPR from a nonexistent vector IPR, or MFPR from a write-only vector IPR. The preferred implementation is to cause reserved-operand fault.

- If an implementation supports an optional vector processor, but the vector processor is not installed, MFPR from VPSR returns zero.
- If an implementation supports an optional vector processor, but the vector processor is not installed, MFPR from VMAC has no effect.

The internal processor register (IPR) assignments for these registers are found in Table 10–5.

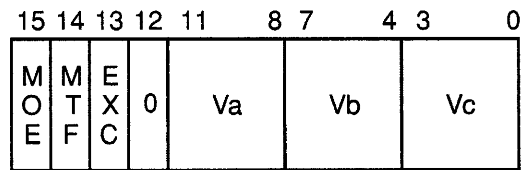
Table 10–5 IPR Assignments

Offset (Hex)	IPR
90	VPSR
91	VAER
92	VMAC
93	VTBIA
94	VSAR
95–9B	Reserved for vector architecture use
9C–9F	Reserved for vector implementation use

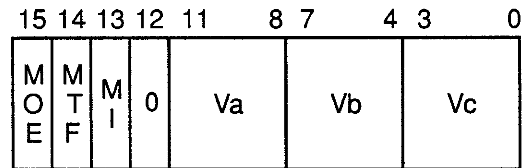
10.3 Vector Instruction Formats

Vector instructions use 2-byte opcodes and normal VAX operand specifiers. For more information on VAX operand specifiers, refer to the *VAX Architecture Reference Manual*. The vector registers to be used by a vector instruction are specified by the vector control word operand. The MFVP, MTVP, and Synchronize Vector Memory Access (VSYNC) instructions do not use a vector control word operand. The general format of the vector control word operand is shown in Figure 10–10. Table 10–6 describes the fields of the vector control word operand (cntrl). The actual format of the vector control word operand is instruction dependent. (Refer to the instruction descriptions later in this chapter for more detail.) The vector control word operand is passed by the VAX scalar processor to the vector processor.

Figure 10–10 Vector Control Word Operand (cntrl)



or



ZK-1453A-GE

Table 10–6 Description of the Vector Control Word Operand

Extent	Description
<3:0>	Vc. This field selects the vector register to be used as the Vc operand. For the Vector Floating Compare (VCMP) instruction, it specifies the compare function.
<7:4>	Vb. This field selects the vector register to be used as the Vb operand.
<11:8>	Va. This field selects the vector register to be used as the Va operand. For the Vector Convert (VVCVT) instruction, it specifies the convert function.
<12>	0
<13>	Modify Intent (MI). Used only in Load Memory Data into Vector Register (VLD) and VGATH instructions. instructions to indicate that a majority of the memory locations being loaded by the VLD or VGATH will later be stored into by VST/VSCAT instructions. This bit is optional to implement. See Section 10.3.3, Modify Intent bit, for more details.
<13>	Exception Enable (EXC). Used only in vector integer and floating-point instructions to enable integer overflow and floating underflow, respectively.
<14>	Match True/False (MTF). When masked operations have been enabled (cntrl<MOE> EQL 1), only elements for which the corresponding VMR bit matches cntrl<MTF> are operated on. See previous description. Cntrl<MTF> is also used by the VMERGE and IOTA instructions.
<15>	Masked Operation Enable (MOE). This bit enables operations under the control of the Vector Mask Register (VMR) for vector instructions. When set, masked operations are enabled, and only elements for which the corresponding VMR bit matches cntrl<MTF> are operated on. If cntrl<MOE> is clear, all elements are operated upon. In either case, the Vector Length Register (VLR) limits the highest element operated upon.

The vector control word operand may determine some or all of the following:

- Enabling of masked operations
- Enabling of floating underflow for floating-point instructions and integer overflow for integer operations
- Which vector registers to use as sources, destinations, or both
- Which type of operation to perform (for the convert and compare instructions)

10.3.1 Masked Operations

Masked operations are enabled by the use of cntrl<15:14> of the vector control word operand. Cntrl<15> is the Masked Operation Enable (MOE) bit, and cntrl<14> is the Match True/False (MTF) bit. When cntrl<MOE> is set, masked operations are enabled. Only elements for which the corresponding Vector Mask Register (VMR) bit matches cntrl<MTF> are operated upon. If cntrl<MOE> is clear, all elements are operated upon. In either case, the Vector Length Register (VLR) limits the highest element operated upon.

Cntrl<MOE> should be zero for VMERGE and IOTA instructions; otherwise the results are UNPREDICTABLE. Both the Vector Mask Register (VMR) and the Match True/False bit (cntrl<MTF>) are always used by these instructions. VMERGE and IOTA operate upon vector register elements up to the value specified in VLR.

10.3.2 Exception Enable Bit

The vector processor does not use the IV and FU bits in the processor status longword (PSL) to enable integer overflow and floating underflow exception conditions. These exception conditions are enabled or disabled on a per instruction basis for vector integer and floating-point instructions by bit <13> in the vector control word operand (cntrl<EXC>). When cntrl<EXC> is set, floating underflow is enabled for vector floating-point instructions, and integer overflow is enabled for vector integer instructions. When cntrl<EXC> is clear, floating underflow and integer overflow are disabled. Note that for VLD/VGATH instructions bit<13> is used and labeled differently.

10.3.3 Modify Intent Bit

The Modify Intent (MI) bit is used by the software to indicate to the vector processor that a majority of the memory locations being loaded by VLD /VGATH instructions will later be stored into, and so become modified, by VST/VSCAT instructions. When informed of software's intent to modify, some vector processor implementations can optimize the vector loads and stores performed on these locations.

The MI bit resides in bit<13> of the vector control word operand (cntrl<MI>) and is used only in VLD and VGATH instructions. A vector processor implementation is not required to implement cntrl<MI>.

For vector processors that implement cntrl<MI>, software uses the bit in a VLD or VGATH instruction in the following way:

- By setting cntrl<MI> to zero, software indicates that less than a majority of the locations loaded by the VLD/VGATH instructions will later be stored into by VST/VSCAT instructions.
- By setting cntrl<MI> to 1, software indicates that a majority of the locations loaded by the VLD/VGATH instructions will later be stored into by VST/VSCAT instructions.

Vector processors that do not implement cntrl<MI> ignore the setting of this bit in the control word for VLD and VGATH.

The results of VLD/VGATH and VST/VSCAT are unaffected by the setting of cntrl<MI>. This includes memory management, where access-checking is done with read intent for VLD/VGATH even if cntrl<MI> is set. However, incorrectly setting cntrl<MI> can prevent the optimization of these instructions.

10.3.4 Register Specifier Fields

The Va (cntrl<11:8>), Vb (cntrl<7:4>), and Vc (cntrl<3:0>) fields of the vector control word operand are generally used to select vector registers. Some vector instructions use these fields to encode other instruction-specific information as shown later in this section.

10.3.5 Vector Control Word Formats

Depending on the instruction, the vector control word can specify up to two vector registers as sources, and one vector register as a destination. Other information may be encoded in the vector control word, as shown in Figure 10–11a to Figure 10–11n. Bits that are shown as "0" should be zero (SBZ). Execution of vector instructions with illegal, inconsistent, or unspecified control word fields produces UNPREDICTABLE results.

Figure 10–11a depicts the vector control word for VLDL and VLDQ.

Figure 10–11b depicts the vector control word for VSTL and VSTQ.

Figure 10–11c depicts the vector control word for VGATHL and VGATHQ.

Figure 10–11d depicts the vector control word for VSCATL and VSCATQ.

Figure 10–11e depicts the vector control word for VVADDL/F/D/G, VVSUBL/F/D/G, VVMULL/F/D/G, and VVDIVF/D/G.

Figure 10–11f depicts the vector control word for VVSLLL, VVSRLL, VVBISL, VVXORL, and VVBICL. Cntrl<EXC> should always be zero for these instructions, otherwise the results are UNPREDICTABLE.

Figure 10–11g depicts the vector control word for VVCMPL/F/D/G. The Vc field (cntrl<3:0>) is used to specify the compare function.

Figure 10–11h depicts the vector control word for VVCVT. The Va field (cntrl <11:8>) is used to specify the convert function.

Figure 10–11i depicts the vector control word for VVMERGE.

Figure 10–11j depicts the vector control word for VSADDL/F/D/G, VSSUBL /F/D/G, VSMULL/F/D/G, and VSDIVF/D/G.

Figure 10–11k depicts the vector control word for VSSLLL, VSSRLL, VSBISL, VSXORL, and VSBICL. Cntrl<EXC> should be zero for these instructions; otherwise, the results are UNPREDICTABLE.

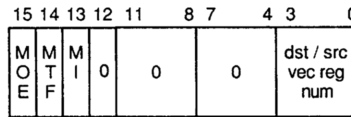
Figure 10–11l depicts the vector control word for VSCMPL/F/D/G. The Vc field (cntrl<3:0>) is used to specify the compare function.

Figure 10–11m depicts the vector control word for VSMERGE.

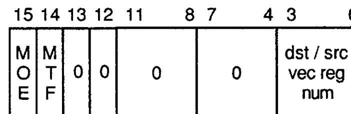
Figure 10–11n depicts the vector control word for IOTA.

Figure 10–11 Vector Control Word Format

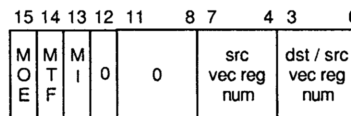
a. Vector Control Word Format for VLDL and VLDQ



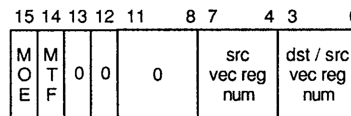
b. Vector Control Word Format for VSTL and VSTQ



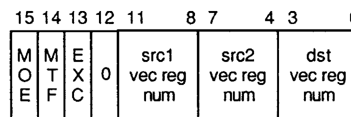
c. Vector Control Word Format for VGATHL and VGATHQ



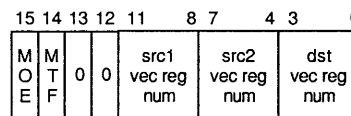
d. Vector Control Word Format for VSCATL and VSCATQ



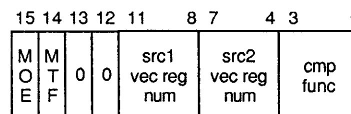
e. Vector Control Word Format for VVADDL/F/D/G, VVSUBL/F/D/G, and VVDIVF/D/G



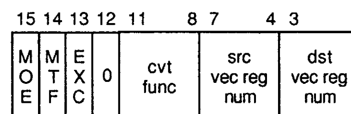
f. Vector Control Word Format for VVSLLL, VVSRLL, VVBISL, VVXORL, and VVBICL



g. Vector Control Word Format for VVCMPL/F/D/G



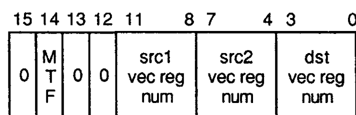
h. Vector Control Word Format for VVCVT



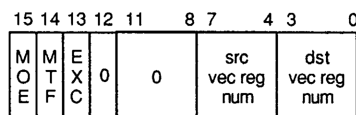
(continued on next page)

Figure 10–11 (Cont.) Vector Control Word Format

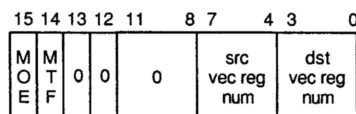
i. Vector Control Word Format for VVMERGE



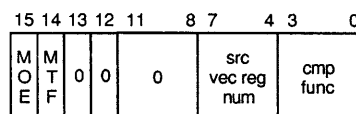
j. Vector Control Word Format for VSADDL/F/D/G, VSSUBL/F/D/G, VSMULL/F/D/G and VSDIVF/D/G



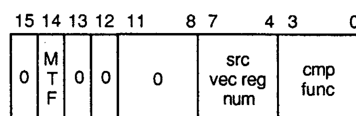
k. Vector Control Word Format for VSSLLL, VSSRLL, VSBISL, VSXORL, and VSBICL



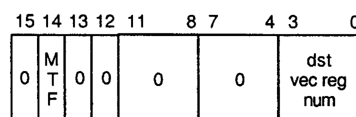
l. Vector Control Word Format for VSCMPL/F/D/G



m. Vector Control Word Format for VSMERGE



n. Vector Control Word Format for IOTA



ZK-1454A-GE

10.3.6 Restrictions on Operand Specifier Usage

Certain restrictions are placed on the addressing mode combinations usable within a single vector instruction. These combinations involve the logically inconsistent simultaneous use of a value as both a source operand (that is, a .rw, .rl, or .rq operand) and an address. Specifically, if within the same instruction the contents of register Rn is used as both a part of a source operand and as an address in an addressing mode that modifies Rn (that is, autodecrement, autoincrement, or autoincrement deferred), the value of the scalar source operand is UNPREDICTABLE.

Use of short literal mode for the scalar source operand of a vector floating-point instruction causes UNPREDICTABLE results.

If a Store Vector Register Data into Memory (VST) or Scatter Memory Data into Vector Register (VSCAT) instruction overwrites anything needed for calculation of the memory addresses to be written, the result of the VST or VSCAT is UNPREDICTABLE.

If the same vector register is used as both source and destination in a Gather Memory Data into Vector Register (VGATH) instruction, the result of the VGATH is UNPREDICTABLE.

When the addressing mode of the BASE operand used in a VLD, VST, VGATH, or VSCAT instruction is immediate, the results of the instruction are UNPREDICTABLE.

10.3.7 VAX Condition Codes

The vector instructions do not affect the condition codes in the processor status longword (PSL) of the associated scalar processor.

10.3.8 Illegal Vector Opcodes

An illegal vector opcode is defined as a vector opcode to which no vector processor function is currently assigned. Opcodes that are not identified in Appendix D as vector opcodes are neither decoded nor executed by the vector processor.

An implementation is permitted to report an illegal vector opcode in one of the following ways:

1. Reserved-instruction fault. This is the recommended implementation.
2. Illegal vector opcode. The vector processor disables itself and sets VPSR<IVO>. The remainder of the vector processor state is left unmodified.

The way in which a particular illegal vector opcode is reported is implementation specific.

10.4 Assembler Notation

The assembler notation uses a format that is different from the operand specifiers for the vector instructions. The number and order of operands is not the same as the instruction-stream format. For example, vector-to-vector addition is denoted by the assembler as "VVADDL V1, V2, V3" instead of "VVADDL X123". The assembler always generates immediate addressing mode (I#constant) for vector control word operands. The assembler notation for vector instructions uses opcode qualifiers to select whether vector processor exception conditions are enabled or disabled, and to select the value of cntrl<MTF> in masked, VMERGE, and IOTA operations. The appropriate opcode is followed by a slash (/). The following qualifiers are supported:

- The qualifier U enables floating underflow. The qualifier V enables integer overflow. Both of these qualifiers set cntrl<EXC>. The default is no vector processor exception conditions are enabled.

- The qualifier 0 denotes masked operation on elements for which the Vector Mask Register (VMR) bit is 0. The qualifier 1 denotes masked operation on elements for which the VMR bit is 1. Both qualifiers set cntrl<MOE>. The default is no masked operations.
- For the VMERGE and IOTA instructions only, the qualifier 0 denotes cntrl<MTF> is 0. The qualifier 1 denotes cntrl<MTF> is 1. Cntrl<MTF> is 1 by default. Cntrl<MOE> is not set in this case.
- For the VLD and VGATH instructions only, the qualifier M indicates modify intent (cntrl<MI> is 1). The default is no modify intent (cntrl<MI> is 0).

The following examples use several of these qualifiers:

```
VVADDF/1    V0, V1, V2    ;Operates on elements with mask bit set
VVMULD/0    V0, V1, V2    ;Operates on elements with mask bit clear
VVADDL/V    V0, V1, V2    ;Enables exception conditions
                           (integer overflow here)
VVSUBG/U0   V0, V1, V2    ;Enables floating underflow and
                          ;Operates on elements with mask bit clear

VLDL/M      base,#4,V1    ;Indicates Modify Intent
```

10.5 Execution Model

A typical processor consists of a VAX scalar processor and its associated vector processor, which contains vector registers and vector function units. The scalar and vector processors may execute asynchronously. The VAX scalar processor decodes both scalar and vector instructions following the operand specifier evaluation rules stated in the *VAX Architecture Reference Manual*, but executes only the scalar instructions. The scalar processor passes the information required to execute a vector instruction to the vector processor. This information may include the vector opcode, scalar source operands, and vector control words. The vector processor performs the required operation, such as loading data from memory, storing data to memory, or manipulating data already loaded into its vector registers.

The scalar processor may decode a vector instruction before checking whether the vector processor should receive it. Exceptions on vector instruction operands may occur during this decoding and may be taken before the attempt to send the decoded instruction to the vector processor. The scalar processor performs one of the following operations when sending a decoded vector instruction to the vector processor. Recall that because the vector and scalar processors can execute asynchronously, a VPSR state transition may not be seen immediately by the scalar processor.

- If the scalar processor views the vector processor as enabled (the scalar processor sees VPSR<VEN> as set), the decoded vector instruction is sent to the vector processor. The vector processor queues instructions sent by the scalar processor until they can be executed.

- If the scalar processor views the vector processor as disabled (the scalar processor sees VPSR<VEN> as clear), attempting to send the decoded vector instruction to the vector processor results in a vector processor disabled fault.

The following flow details how vector instruction decode proceeds from the scalar processor:

```
DO WHILE (the scalar processor has a decoded vector instruction for
          the vector processor)
  IF (the vector processor is viewed as disabled -- the scalar processor
     sees VPSR<VEN> as clear) THEN

     enter the vector processor disabled fault handler.
  ELSE
     IF (asynchronous memory management handling is implemented
        AND VPSR<PMF> is set) THEN
        enter the memory management exception handler.
        {The vector processor clears VPSR<PMF>.}
     ELSE

        BEGIN
          {If asynchronous memory management handling is
          implemented and VPSR<MF> is set, the vector processor
          clears VPSR<MF>, and retries the faulting memory
          reference before any new vector instructions in the
          queue are executed.}
          IF (the vector processor instruction queue is not full) THEN
             BEGIN
               Send the decoded instruction to the vector processor
               for execution.
               IF (the decoded instruction is a vector memory access
                   instruction AND synchronous memory management
                   handling is implemented) THEN
                  ensure instruction completion without the occurrence
                  of memory management exceptions.
             END
        END
END
```

If asynchronous memory management handling is implemented, and VPSR<MF> is set when the scalar processor sends the vector processor an instruction, the vector processor clears VPSR<MF>, and retries the faulting memory reference before any new vector instructions in the queue are executed.

The VAX scalar processor need not wait for the vector processor to complete its operation before processing other instructions. Thus, the scalar processor could be processing other VAX instructions while the vector processor is performing vector operations. However, if the scalar processor issues an MFVP instruction to the vector processor, the scalar processor must wait for the MFVP result to be written before processing other instructions.

Because the scalar and vector processors may execute asynchronously, it is possible to context switch the scalar processor before the vector processor is idle. Software is responsible for ensuring that scalar and vector memory management remains synchronized, and that all exceptions get reported in the context of the process where they occurred. This is achieved by

making sure all vector memory accesses complete, and then disabling the vector processor before any scalar context switch.

The vector processor may have its own translation buffer (TB) and cache and may have separate paths to memory, or it may share these resources with the scalar processor.

10.5.1 Access Mode Restrictions

In general, processes are expected to use the vector processor in only one mode. However, multimode use of the vector processor by a process is allowed. Software decides whether to allow vector processor exceptions from vector instructions executed in a previous access mode to be reported in the current mode. The preferred method is to report all vector processor exceptions in the access mode where they occurred. This is achieved by requiring a process that uses the vector processor to execute a SYNC instruction before changing to an access mode where additional vector instructions are executed.

For correct access checking of vector memory references, the vector processor must know the access mode in effect when a vector memory access instruction is issued by the scalar processor.

10.5.2 Scalar Context Switching

With the addition of a vector processor, the required steps in performing a scalar context switch change. The following procedure outlines the required method software should use for scalar context switching:

1. Disable the vector processor so that no new vector instructions will be accepted. Writing zero to the VPSR using the MTPR instruction clears VPSR<VEN>, and disables the vector processor without affecting VPSR<31:1>. (See Section 10.6.3, Vector Processor Disabled, for more details.)
2. Ensure that no more vector memory read or write operations can occur. Reading the VMAC internal processor register (IPR) using the MFPR instruction does the required scalar/vector memory synchronization without any exceptions being reported. Reading VMAC also ensures that all unreported hardware errors encountered by previous vector memory instructions are reported before the MFPR completes. For more information on this function of VMAC, refer to Section 10.9, Hardware Errors.
3. Set a software scalar-context-switch flag and perform a normal scalar processor context switch, for example SVPCTX, and so on, leaving the vector processor state as is.

Although not required by the architecture, software may wait for VPSR<BSY> to be clear after disabling the vector processor when performing a scalar context switch, which provides the following advantages:

- The vector processor can not be executing non-memory-access instructions from the previous process while a normal scalar context

switch to a new process is being performed—which may be desirable to an operating system.

- All unreported hardware errors encountered by previous non-memory-access instructions will be reported by the time the vector processor clears VPSR<BSY> and thus known to software before scalar-context switching continues (refer to Section 10.9, Hardware Errors, for more details).
- The MFPR from VPSR used to read VPSR<BSY> also ensures that the scalar processor views the vector processor as disabled.

If software does not wait for VPSR<BSY> to be clear, it is possible that while a normal scalar context switch to a new process is being performed, the vector processor may still be executing non-memory-access instructions from the previous process.

The required steps for Vector Context Switching are discussed in Section 10.6.4, Handling Disabled Faults and Vector Context Switching.

10.5.3 Overlapped Instruction Execution

To improve performance, the vector processor may overlap the execution of multiple instructions—that is, execute them concurrently. Further, when no data dependences are present, the vector processor may complete instructions out of order relative to the order in which they were issued. A vector processor implementation can perform overlapped instruction execution by having separate function units for such operations as addition, multiplication, and memory access. Both data-dependent and data-independent instructions can be overlapped; the former by a technique known as chaining, which is described in the next section. In many instances, overlapping allows an operation from one instruction to be performed in any order with respect to an operation of another instruction.

When vector arithmetic exceptions occur during overlapped instruction execution, exception handling software may not see the same instruction state and exception information that would be returned from strictly sequential execution. Most notably, the VAER could indicate the exception conditions and destination registers of a number of vector instructions that were executing concurrently and encountered exceptions. Exception reporting during chained execution is discussed further in Section 10.5.3.1.

To ensure correct program results and exception reporting, the architecture does place requirements on the ordering among the operations of one vector instruction and those of another. The primary goal of these requirements is to ensure that the results obtained from both the overlapped and strictly sequential execution of data-dependent instructions are identical. A secondary goal is to establish places within the instruction stream where software is guaranteed to receive the reporting of exceptions from a chain of data-dependent instructions.

In many cases, these requirements ensure the obvious: for example, an output vector register element of one arithmetic instruction must be computed before it can be used as an input element to a subsequent instruction. But, a number of the things ensured are not obvious: for

example, a Memory Instruction Synchronization (MSYNC) instruction must report exceptions encountered in generating a value of Vector Mask Register (VMR) that is used in a previously issued masked store instruction.

To precisely define the requirements on the ordering among operations, Section 10.5.3.3 discusses the "dependence" among their results (the vector register elements and control register bits produced by the operations).

10.5.3.1 Vector Chaining

The architecture allows vector chaining, where the results of one vector instruction are forwarded (chained) to another before the input vector of the first instruction has been completely processed. In this way, the execution of data-dependent vector instructions may be overlapped. Thus, chaining is an implementation-dependent feature that is used to improve performance.

With some restrictions stated below, the vector processor may chain a number of instructions. Usually, each instruction is performed by a separate function unit. The number and types of instructions allowed within a chained sequence (often referred to as a "chain") are implementation dependent. Typically, implementations will attempt to chain sequences of two or three instructions such as: operate-operate, operate-store, load-operate, operate-operate-store, and load-operate-store. Load-operate-operate-store may also be possible.

The following is an example of a sequence that an implementation will often chain:

```
VVADDF  V0, V1, V2
VVMULF  V2, V3, V4
```

The destination of the VVADDF is a source of the succeeding VVMULF. The VVMULF begins executing when the first sum element of the VVADDF is available.

A number of instructions within a chained sequence can encounter exceptions. For each instruction that encounters an exception, the vector processor records the exception condition type and destination register number in the Vector Arithmetic Exception Register (VAER). When the last instruction within the chain completes, the VAER will show the exception condition type and destination register number of all instructions that encountered exceptions within the chain. Furthermore, when the vector processor disabled fault is finally generated for the exceptions, the VAER may also indicate exception state for instructions issued after the last instruction within the chain. This effect is possible due to the asynchronous exception-reporting nature of the vector processor.

Furthermore, for each instruction that encounters an exception within a chain, the default result, as defined in Section 10.6.2, is forwarded as the source operand to the next instruction. This has the effect that default results and exceptions can propagate down through a chain. Note that the default result of one instruction may be overwritten by another instruction before the exception is taken.

Consider the following:

```
VVADDG V1, V2, V3   ;gets Floating Overflow
VVGEQG V3, V4       ;gets Floating Reserved Operand
VVMULG V4, V5, V3   ;overwrites V3
```

For the previous example, assume that an exception is taken after the completion of the VVMULG. The VAER will indicate: Floating Overflow and Floating Reserved Operand exception condition types; and V3 as a destination register. However, no default result will be found in the appropriate element of V3 because it has been overwritten by the VVMULG.

The architecture allows a vector load to be chained into a vector operate instruction provided the operate instruction can be suspended and resumed to produce the correct result if the vector load gets a memory management exception. Consider this example:

```
VLDL    A, #4, V0
VVADDF  V0, V1, V1
```

In synchronous memory management mode, the VVADDF cannot be chained into the VLDL until the VLDL is ensured to complete without a memory management exception. This occurs because the scalar processor is not even allowed to issue the VVADDF to the vector processor until the memory management checks for the VLDL have been completed. In asynchronous memory management mode, the VVADDF may be chained into the VLDL prior to the completion of memory management exception checking. This is possible because a memory management exception in asynchronous memory management mode provides sufficient state to restart both the VLDL and the VVADDF when the memory management exception is corrected.

The architecture allows a vector operate instruction to be chained into a store instruction. If the vector operate instruction encounters an arithmetic exception, the exception condition type and destination register number are recorded in the Vector Arithmetic Exception Register (VAER). The default result generated by that instruction (in some cases an encoded reserved operand) may be written to memory by the store instruction before the exception is reported.

10.5.3.2 Register Conflict

When overlapping the execution of instructions, the vector processor must deal with register conflict. This occurs when one instruction is intending to write a register while previously issued instructions are reading from that register. The following is an example of vector register conflict:

```
VVADDF  V1, V2, V3
VVMULF  V4, V5, V1
```

In the example, the VVADDF and VVMULF cannot both begin execution simultaneously because the elements of V1 generated by the VVMULF would overwrite the original elements of V1 required as input by the VVADDF. However, a vector processor implementation can still overlap the execution of these two instructions in a number of ways. One way would be by not starting the VVMULF until the first element of V1 has been read by the VVADDF. In this manner, as the VVADDF reads the

next elements from V1 and V2, the VVMULF writes its product into the previous element of V1. This process continues until all the elements have been processed by both instructions. The VVADDF will finish execution while the VVMULF still has at least one product to store.

In the case of the Vector Mask Register (VMR), the vector processor ensures that register conflict does not occur. This is often accomplished by making a copy of the VMR value under which a pending vector instruction is to execute, and using this copy when execution begins. This allows the vector processor to begin executing an instruction that writes VMR before it completes prior instructions that read VMR.

10.5.3.3 Dependences Among Vector Results

In order to achieve correct results and exception reporting during overlapped execution, the vector processor must maintain certain dependences among the register elements and control register bits produced by various vector instructions. Because of the vector processor's asynchronous exception reporting nature and out-of-order completion of instructions, these dependences differ from those ensured by the VAX scalar processor. In addition, these dependences are at the level of vector register elements and vector control register bits; rather than at the level of vector registers and vector control registers.

Among other things, these dependences determine the exception reporting nature of the MFVP instruction. The value of the vector control register (VCR, VLR, VMR<31:0>, VMR<63:32>) delivered by an MFVP depends upon the value of certain vector register elements and vector control register bits. Unreported exceptions that occur in the production of these elements and control register bits are reported by the vector processor prior to the completion of the MFVP from the vector control register.

The dependences are expressed formally for the various classes of vector instructions by the tables of pseudo-code in this section. These are the only dependences that software should rely upon the vector processor to ensure.

A vector processor implementation is allowed to ensure more than just these dependences providing that this larger set of dependences yields correct results and exception reporting.

Note: Note the implications of the following sequence for Table 10–7, Table 10–8, Table 10–9, Table 10–10, Table 10–11, Table 10–12, Table 10–13, and Table 10–14:

```
VVSUBF V5, V6, V7
VVADDF V1, V2, V7
VVMULF V7, V7, V3
VVDIVF V1, V4, V7
```

Implicit in statements of the form: "result DEPENDS on B" is the requirement that the result depends only on the value of "B" generated by the most immediate previously issued instruction relative to the result's own generating instruction. For instance, in the following example, the V3 produced by the VVMULF has the dependence: "V3[i] DEPENDS on V7[i]". This means that the value

of V3[i] produced by the VVMULF depends only on the value of V7[i] produced by the VVADDF.

Table 10–7 Dependences for Vector Operate Instructions

Instructions	Dependence

VVADDx, VSADDx, VVSUBx, VSSUBx, VVMULx, VSMULx, VVDIVx, VSDIVx, VVCVTxy, VVBICL, VSBICL, VVBISL, VSBISL, VVXORL, VSXORL, VVSLLL, VSSLLL, VVSRLL, VSSRLL	for i = 0 to VLR-1
	    begin
	    Vc[i] DEPENDS on VLR;
	    if {MOE EQL 1} then Vc[i] DEPENDS on VMR<i>;
	       if ({MOE EQL 1} AND {VMR<i> EQL MTF}) OR {MOE EQL 0} then
	          begin
	          Vc[i] DEPENDS on Vb[i];
	          if {Vector-Vector Operation} AND NOT {VVCVTxy} then
	             Vc[i] DEPENDS on Va[i];
	          end;
	    end;

Table 10–8 Dependences for Vector Load and Gather Instructions

Instructions	Dependence

VLDx, VGATHx	for i = 0 to VLR-1
	    begin
	    Vc[i] DEPENDS on VLR;
	    if {MOE EQL 1} then Vc[i] DEPENDS on VMR<i>;
	    if ({MOE EQL 1} AND {VMR<i> EQL MTF}) OR {MOE EQL 0} then
	        if VGATH then
	           begin
	           Vc[i] DEPENDS on Vb[i];
	           k = BASE + Vb[i];
	           end
	        else
	           k = BASE + i * STRIDE;
	    Vc[i] DEPENDS on LOAD_COMPLETED(k);
	    end;

Table 10–9 Dependences for Vector Store and Scatter Instructions

Instructions	Dependence
VSTx, VSCATx	

```
j = 0;
for i = 0 to VLR-1
    begin
    if ( {MOE EQL 1} AND {VMR<i> EQL MTF} ) OR {MOE EQL 0} then
        begin
        if {MOE EQL 1} then ELEMENT_STORED[j] depends on VMR<i>;
        ELEMENT_STORED[j] DEPENDS on Vc[i];
        ELEMENT_STORED[j] DEPENDS on VLR;
        if VSCAT then
           begin
           ELEMENT_STORED[j] DEPENDS on Vb[i];
           k = BASE + Vb[i];
           end
        else
           k = BASE + i * STRIDE;
        STORE_COMPLETED(k) DEPENDS on ELEMENT_STORED[j];
        j = j+1;
        end;
    end;
```

Table 10–10 Dependences for Vector Compare Instructions

Instructions	Dependence
VVCMPx, VSCMPx	

```
for i = 0 to VLR-1
    begin
    VMR<i> DEPENDS on VLR;
    if {MOE EQL 1} then VMR<i> DEPENDS on VMR<i>
    if ( {MOE EQL 1} AND {VMR<i> EQL MTF} ) OR {MOE EQL 0} then
        begin
        VMR<i> DEPENDS on Vb[i];
        if VVCMP then VMR<i> DEPENDS on Va[i];
        end;
    end;
```

Table 10–11 Dependences for Vector MERGE Instructions

Instructions	Dependence

VVMERGE, VSMERGE	for i = 0 to VLR-1
	begin
	Vc[i] DEPENDS on VLR;
	Vc[i] DEPENDS on VMR<i>;
	if {VMR<i> EQL MTF} then
	begin
	if VVMERGE then Vc[i] DEPENDS on Va[i];
	end
	else
	Vc[i] DEPENDS on Vb[i];
	end;

Table 10–12 Dependences for IOTA Instruction

Instruction	Dependence

IOTA	j = 0;
	for i = 0 to VLR-1
	begin
	Vc[j] DEPENDS on VLR;
	if {VMR<i> EQL MTF} then
	begin
	Vc[j] DEPENDS on VMR<0..i>;
	j = j+1;
	end;
	end;
	VCR DEPENDS on VMR<0..VLR-1>;

Table 10–13 Dependences for MFVP Instructions

Instructions	Dependence
MSYNC	DEPENDS on the following: • All STORE_COMPLETED(x) of previously issued VST and VSCAT instructions • All LOAD_COMPLETED(X) of previously issued VLD and VGATH instructions
SYNC	DEPENDS on the vector register elements and vector control register bits produced and stored by all previous vector instructions
MFVMRLO	DEPENDS on VMR<0..31>
MFVMRHI	DEPENDS on VMR<32..63>
MFVCR	DEPENDS on VCR
MFVLR	DEPENDS on VLR

Table 10–14 Miscellaneous Dependences

Item	Dependence
VSYNC	Depends on nothing, but for each memory location, x forces all subsequent LOAD_COMPLETED(x) and STORE_COMPLETED(x) to DEPEND on all previous LOAD_COMPLETED(x) and STORE_COMPLETED(x).
MTVP	DEPENDS on nothing.
Value of a memory location	The value of a memory location DEPENDS on nothing and is not DEPENDED on by any vector instruction.
Transitive dependence	`if {a DEPENDS on b} AND {b DEPENDS on c} then a DEPENDS on c`

10.6 Vector Processor Exceptions

There are two major classes of vector processor exceptions as follows:

- Vector memory management
  - Access control violation

    Vector access control violation
    Vector alignment
    Vector I/O space reference
  - Translation not valid
  - Modify
- Vector Arithmetic
  - Floating underflow
  - Floating divide by zero
  - Floating reserved operand
  - Floating overflow
  - Integer overflow

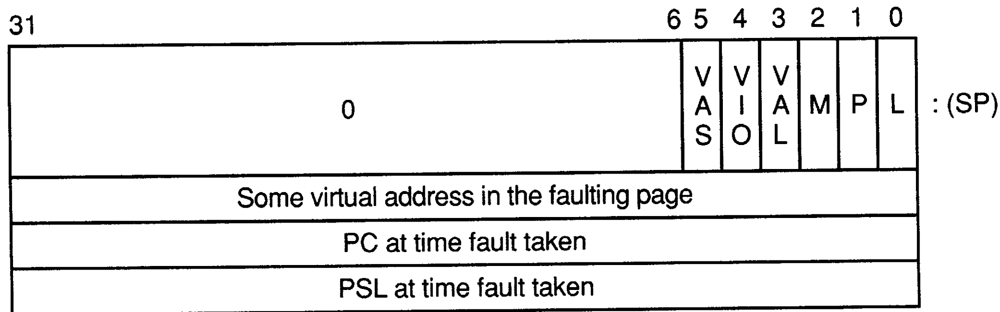
  Floating underflow and integer overflow can be disabled on a per-instruction basis by clearing cntrl<EXC>.

Vector processor arithmetic exceptions cause the vector processor to disable itself (see Section 10.6.3, Vector Processor Disabled). The vector processor does not disable itself for vector processor memory management exceptions.

10.6.1 Vector Memory Management Exception Handling

Vector processor memory management exceptions are taken through the system control block (SCB) vector for their scalar counterparts. Figure 10–12 illustrates the memory management fault stack frame that contains the memory management fault parameter.

Figure 10–12 Memory Management Fault Stack Frame (as Sent by the Vector Processor)



- The length (L) bit, the Page Table Entry (PTE) reference (P) bit, and the modify or write intent (M) bit are defined in the *VAX Architecture Reference Manual*. Vector processor memory management exceptions set these bits in the same way as required for scalar memory management exceptions.
- The vector alignment exception (VAL) bit must be set when an access control violation has occurred due to a vector element not being properly aligned in memory.
- The vector I/O space reference (VIO) bit is set by some implementations to indicate that an access control violation has occurred due to a vector instruction reference to I/O space.
- The vector asynchronous memory management exception (VAS) bit must be set to indicate that a vector processor memory management exception has occurred when the follwing asynchronous memory management scheme is implemented.

If more than one kind of memory management exception could occur on a reference to a single page, then access control violation takes precedence over both translation not valid and modify. If more than one kind of access control violation could occur, the precedence of vector access control violation, vector alignment exception, and vector I/O space reference is UNPREDICTABLE.

The architecture allows an implementation to choose one of two methods for dealing with vector processor memory management exceptions. The two methods are as follows:

- Synchronous memory management handling and restart from the beginning.
- Asynchronous memory management handling and store/reload implementation-specific state using VSAR.

With the synchronous method, no new instructions are processed by the vector or the scalar processor until the vector memory access instruction is guaranteed to complete without incurring memory management exceptions. In such an implementation, the vector memory access instruction is backed up when a memory management exception occurs and a normal VAX memory management (access control violation, translation not valid, modify) fault taken with the program counter (PC) pointing to the faulting vector memory access instruction. If the synchronous method is implemented, VSAR is omitted. After fixing the vector processor memory management exception, software may REI back to the faulting vector instruction. Alternately, software may context switch to another process. For further details, see Section 10.6.4.

With the asynchronous method, vector memory management exceptions set VPSR<PMF> and VPSR<MF>. The vector processor does not inform the scalar processor of the exception condition; the scalar processor continues processing instructions. All pending vector instructions that have started execution are allowed to complete if their source data is valid. The scalar processor is notified of an exception condition or conditions when it sends the next vector instruction to the vector processor and a normal VAX memory management fault is taken. The saved PC points to this instruction, which is not the vector memory access instruction that incurred the memory management exception. At this point, the vector processor clears VPSR<PMF>. After fixing the vector processor memory management exception, software may allow the current scalar /vector process to continue. Before vector processor instruction execution resumes using state that already exists in the vector processor, the vector processor clears VPSR<MF> and the faulting memory reference is retried. Alternately, software may context switch to another process. For further details, see Section 10.6.4.

When a vector processor memory management exception is encountered by a VLD or VGATH instruction, the contents of the destination vector register elements are UNPREDICTABLE. When a vector processor memory management exception is encountered by a VSTL or VSCAT instruction, it is UNPREDICTABLE whether the vector processor writes any result location for which an exception did not occur. In either case, if the fault condition can be eliminated by software and the instruction restarted, then the vector processor will ensure that all destination register elements or result locations are written.

10.6.2 Vector Arithmetic Exceptions

Vector operate instructions are always executed to completion, even if a vector arithmetic exception occurs. If an exception occurs, a default result is written. The default result is as follows:

- The low-order 32 bits of the true result for integer overflow.
- Zero for floating underflow if exceptions are disabled.

- An encoded reserved operand for floating divide by zero, floating overflow, reserved operand, and enabled floating underflow. (See Section 10.13.1.) For vector convert instructions that convert floating-point data to integer data, where the source element is a reserved operand, the value written to the destination element is UNPREDICTABLE.

The exception condition type and destination register number are always recorded in the Vector Arithmetic Exception Register (VAER) when a vector arithmetic exception occurs. Refer to Section 10.2.3, Internal Processor Registers, for more information.

10.6.3 Vector Processor Disabled

As a result of error conditions or software control, the vector processor signals the scalar processor not to issue any more vector instructions. The vector processor is disabled when this signal is generated and its state is reflected in VPSR<VEN>. Because the scalar and vector processors can execute asynchronously, the scalar processor may not receive this signal immediately. As a result, the scalar processor may continue to view the vector processor as enabled and send it vector instructions. Once the scalar processor receives this signal, it will view the vector processor as disabled and will not send it any more vector instructions (including MFVP/MTVP). While the vector processor is disabled, and in the absence of hardware errors, it will complete all pending instructions in its instruction queue including those sent by the scalar processor after the vector processor became disabled.

The vector processor can either disable itself or be disabled by software. The following error conditions cause the vector processor to disable itself:

- Vector arithmetic exception (flagged by VPSR<AEX>)
- Hardware error (flagged by VPSR<IMP> in some implementations)
- On some implementations, receipt of an illegal vector opcode (flagged by VPSR<IVO>)

In these cases, the vector processor clears VPSR<VEN> and flags the error condition by setting the appropriate bit in VPSR. (See Table 10–1.)

Software disables the vector processor by writing a zero into VPSR<VEN> using an MTPR instruction. Once the vector processor is disabled, only software can enable it. The software does this by writing a one to VPSR<VEN> using an MTPR. Recall that after performing an MTPR to VPSR, software must then issue an MFPR from VPSR to ensure that the new state of VPSR will affect the execution of subsequently issued vector instructions. The MFPR will not complete in this case until the new state of the vector processor becomes visible to the scalar processor.

When the vector processor disables itself due to a hardware error, it is implementation dependent whether the vector processor completes any pending vector instruction. However, in this case, the vector processor ensures when it is reenabled that all incompleted instructions have been flushed from the instruction queue.

If the scalar processor attempts to issue a vector instruction after it views the vector processor as disabled, then a vector processor disabled fault occurs. The vector processor disabled fault uses SCB offset 68 (hex). The exception handling software (running on the scalar processor) can then read the vector internal processor registers (IPRs) with MFPR instructions to determine what exception conditions are recorded in the vector processor and if the vector processor is still busy processing other unfinished instructions.

Once the scalar processor views the vector processor as disabled, the only operations that can be issued to the vector processor are MTPR and MFPR to and from the vector IPRs.

10.6.4 Handling Disabled Faults and Vector Context Switching

The following flow outlines the required steps for handling a vector processor disabled fault.

If the new process executing on the scalar processor has a vector instruction to execute, saving and restoring the state of the vector processor—that is, vector context switching—is done as part of handling a subsequent vector processor disabled fault.

If a vector processor disabled fault occurs and the current scalar process is also the current vector process, then software must perform the following procedure:

1 Obtain the vector processor status by reading the VPSR using the MFPR instruction.

2 Perform the following checks to see if any of these conditions caused the vector processor to be disabled. If any of these conditions exist, a decision to not continue this flow may occur.

   a. If VPSR<IVO> is set, then write one to clear VPSR<IVO> using the MTPR instruction, and report an illegal vector opcode error.

   b. If VPSR<IMP> is set, then write one to clear VPSR<IMP> using the MTPR instruction, and report an implementation-specific error.

   c. If VPSR<AEX> is set, then write one to clear VPSR<AEX> using the MTPR instruction, and enter the vector arithmetic exception handler with information in VAER.

3 If the software scalar-context-switch flag is set, indicating that a scalar context switch has been done, then perform the following:

   a. Make sure the vector processor has access to correct P0LR, P0BR, P1LR, and P1BR values.

   b. If any vector translation buffer needs to be invalidated, then write zero into the VTBIA IPR using the MTPR instruction. Vector translation buffer flushing is required if the process was swapped out and the mapping change has not yet been made known to the vector translation buffer.

   c. Clear the software scalar-context-switch flag.

4 Enable the vector processor by writing one to VPSR<VEN> using the MTPR instruction. Ensure the new state of the vector processor becomes visible to the scalar processor by reading VPSR with the MFPR instruction.

5 REI to retry the vector instruction at the time of the vector processor disabled fault. If there is an asynchronous memory management exception pending, it is taken when that vector instruction is reissued to the vector processor.

If a vector processor disabled fault occurs and the current scalar process is not the current vector process, then software must perform the following procedure:

1 Check if there is a current vector process. If there is one, then perform the following procedure:

   a. Wait for VPSR<BSY> to be clear using the MFPR instruction.

   b. Perform the following check to see if this condition caused the vector processor to be disabled. If this condition exists, a decision to not continue this flow may occur.

      1 If VPSR<IMP> is set, then report an implementation-specific error.

      2 If VPSR<IVO> is set, then set a software IVO flag for this process. The illegal vector opcode error is handled when this process next tries to execute in the vector processor.

      3 If VPSR<AEX> is set, then set a software AEX flag for this process, and save vector arithmetic exception state from VAER using the MFPR instruction. Any vector arithmetic exception conditions are handled when this process next tries to execute in the vector processor.

   c. At this point there cannot be a synchronous memory management exception pending. But, if asynchronous memory management handling is implemented, there may be an asynchronous memory management exception pending. Because scalar/vector memory synchronization was required before scalar context switching, all such pending exceptions are known at this time. So, if VPSR<PMF> is set, then perform the following procedure:

      1 Set a software asynch-memory-exception-pending flag for this process.

      2 Store implementation-specific vector state in memory starting at the address in VSAR by writing one to VPSR<STS> using the MTPR instruction.

   d. Reset the vector processor state to clear VAER and VPSR, and enable the vector processor. Writing a one to both VPSR<RST> and VPSR<VEN> using the same MTPR instruction accomplishes this. Ensure the new state of the vector processor becomes visible to the scalar processor by reading VPSR with the MFPR instruction.

   e. Store the current vector (V0–V15) and vector control (VLR, VMR, and VCR) register values using VST and MFVP instructions.

f. Read the VMAC IPR using the MFPR instruction. This ensures scalar/vector memory synchronization and that all hardware errors encountered by previous vector memory instructions have been reported.

2 Make the current scalar process also the current vector process.

3 Clear the software scalar-context-switch flag.

4 Make sure the vector processor has access to correct P0LR, P0BR, P1LR, and P1BR values, and invalidate any vector translation buffer by writing zero to the VTBIA IPR using the MTPR instruction.

5 Load the saved vector (V0–V15) and vector control (VLR, VMR, and VCR) register values using VLD and MTVP instructions.

6 If the software IMP, IVO, or AEX flags for this process are set, perform the following procedure:

a. Disable the vector processor by writing zero to VPSR<VEN> using the MTPR instruction. Ensure the new state of the vector processor becomes visible to the scalar processor by reading VPSR with the MFPR instruction.

b. If set, clear the software IMP flag for this process and finish handling the implementation-specific error. A decision to not continue this flow may occur.

c. If set, clear the software IVO flag for this process and report an illegal vector opcode error occurred. A decision to not continue this flow may occur.

d. If set, clear the software AEX flag for this process and enter the vector arithmetic exception handler with saved VAER state. A decision to not continue this flow may occur.

7 If the software async-memory-exception-pending flag for this process is set, perform the following procedure:

a. Clear the software async-memory-exception-pending flag for this process.

b. Send the vector processor the memory address that points to implementation-specific vector state for this process by writing VSAR using the MTPR instruction.

c. Reload the implementation-specific vector state for this process and leave the vector processor enabled by writing one to both VPSR<RLD> and VPSR<VEN> using the same MTPR instruction. From this state, the vector processor determines if VPSR<PMF>, VPSR<MF>, or both need to be set, and does it. Ensure the new state of the vector processor becomes visible to the scalar processor by reading VPSR with the MFPR instruction.

8 REI to retry the vector instruction at the time of the vector processor disabled fault. If there is an asynchronous memory management exception pending, it is taken when that vector instruction is reissued to the vector processor.

10.6.5 MFVP Exception Reporting Examples

This section gives examples of Move from Vector Processor (MFVP) exception reporting that are ensured by the vector processor. The rules used to determine the correct result for each example are found in: the tables of dependences found in Section 10.5.3.3, the description of MSYNC in Section 10.7.2, and the description of MFVP in Section 10.15.

Examples of Exceptions That Cause MSYNC to Fault

The following examples illustrate which exceptions are ensured by the vector processor to always cause MSYNC to fault:

1

```
VVMULF     V1, V1, V2
VVADDF     V3, V2, V3
MTVLR      #1
VSTL       V2, A,  #4
VVCVTFD            V2, V3
MSYNC      R0
```

The MSYNC faults if exceptions occur in the production of V2[0] by the VVMULF or in the storage of V2[0] by the VSTL. MSYNC need not fault if exceptions occur in the production of: V2[1..VLR-1] by the VVMULF, V3[0..VLR-1] by the VVADDF, or V3[0..VLR-1] by the VVCVTFD.

2

```
VVADDF     V1, V1, V0
VLDL       A,  #4, V0
MSYNC      R0
```

The MSYNC faults if exceptions occur in the loading of V0[0..VLR-1] from memory. MSYNC need not fault if exceptions occur in the production of V0[0..VLR-1] by the VVADDF.

3

```
VVADDF     V1, V1, V2
VLDL       A,  #4, V1
MSYNC      R0
```

The MSYNC faults if exceptions occur in the loading of V1[0..VLR-1] from memory. MSYNC need not fault if exceptions occur in the production of V2[0..VLR-1] by the VVADDF.

4

```
VVMULF     V1, V1, V2
VVGTRF     V2, V3
VSTL/1     V0, A,  #4
MSYNC      R0
```

The MSYNC faults if exceptions occur: in the production of V2[0..VLR-1] by the VVMULF, in the production of VMR<0..VLR-1> by the VVGTRF, or in the storage by the VSTL/1 of elements of V0 for which the corresponding VMR bit is one.

Examples of Exceptions the Processor Reports Prior to MFVP Completion

The following examples illustrate which exceptions the vector processor will report prior to the completion of an MFVP from a vector control register:

1

```
VLDL      A, #4, V1
VVMULF    V1, V1, V2
MTVLR     #1
VVGTRF    V2, V3
MFVMRHI          R1
MFVMRLO          R2
```

Unreported exceptions that occur: in the loading of V1[0] from memory by the VLDL, in the production of V2[0] by the VVMULF, and VMR<0> by the VVGTRF are reported by the vector processor prior to the completion of the MFVMRLO. The vector processor need not at that time report any exceptions that occur in the loading of V1[1..63] from memory by the VLDL or in the production of V2[1..63] by the VVMULF. Note that the vector processor need not report any exceptions before completing MFVMRHI.

2

```
VVGTRF    V0, V1
MTVMRLO   #patt
MFVMRLO          R1
```

For any value of “i” in the range of 0 to 31 inclusive: the value of VMR<i> delivered by MFVMRLO only depends on the value placed into VMR<i> by the MTVMRLO. As a result, the vector processor need not report exceptions that occur in the production of VMR by the VVGTRF prior to completing the MFVMRLO.

3

```
VVMULF/1  V1, V1, V2
MTVMRLO          #patt
MFVMRLO          R1
```

For any value of “i” in the range of 0 to 31 inclusive: the value of VMR<i> delivered by MFVMRLO only depends on the value placed into VMR<i> by the MTVMRLO. As a result, the vector processor need not report exceptions that occur in the production of V2[0..VLR-1] by the VVMULF/1 prior to completing the MFVMRLO.

4

```
MTVLR     #64
VVMULF    V0, V0, V2
VVGTRF    V0, V2
MTVLR     #32
IOTA      #str, V4
MFVCR     R1
```

Prior to the completion of the MFVCR, the vector processor must report any exceptions that occurred in the production of V2[0..31] by the VVMULF and VMR<0..31> by the VVGTRF. Note that VCR produced by an IOTA depends only on VMR<0..VLR-1>. Recall that no exceptions can occur in the production of V4[0..VCR-1] by IOTA.

5

```
MTVLR      #64
VLDL       A, #4, V2
VVGTRF     V0, V1
VSGTRF/1   #3.0, V2
MFVMRLO            R1
```

For any value of "i" in the range of 0 to 31 inclusive: prior to the completion of the MFVMRLO, the vector processor must report any exceptions that occurred: in the loading of V2[i] from memory for which V0[i] is greater than V1[i], in the production of VMR<0..31> by the VVGTRF, and in the production of VMR<0..31> by the VSGTRF/1.

6

```
VVMULF     V1, V1, V1
VSTL       V1, base, #str
MTVMRLO    base
MFVMRLO            R1
```

In this example, the value of VMR<31:0> delivered by MFVMRLO only depends on the value placed into VMR<31:0> by the MTVMRLO – whether this value is V1[0] or the previous value of the location is UNPREDICTABLE. As a result, the vector processor need not report exceptions that occur in the production of V1 by the VVMULF or in the storage of V1 by the VSTL.

10.7 Synchronization

For most cases, it is desirable for the vector processor to operate concurrently with the scalar processor so as to achieve good performance. However, there are cases where the operation of the vector and scalar processors must be synchronized to ensure correct program results. Rather than forcing the vector processor to detect and automatically provide synchronization in these cases, the architecture provides software with special instructions to accomplish the synchronization. These instructions synchronize the following:

- Exception reporting between the vector and scalar processors
- Memory accesses between the scalar and vector processors
- Memory accesses between multiple load/store units of the vector processor

Software must determine when to use these synchronization instructions to ensure correct results.

The following sections describe the synchronization instructions.

10.7.1 Scalar/Vector Instruction Synchronization (SYNC)

A mechanism for scalar/vector instruction synchronization between the scalar and vector processors is provided by SYNC, which is implemented by the MFVP instruction. SYNC allows software to ensure that the exceptions of previously issued vector instructions are reported before the scalar processor proceeds with the next instruction. SYNC detects both arithmetic exceptions and asynchronous memory management exceptions

and reports these exceptions by taking the appropriate VAX instruction fault. Once it issues the SYNC, the scalar processor executes no further instructions until the SYNC completes or faults.

In beginning the execution of SYNC, the vector processor determines if any previously issued vector instruction has encountered exceptions which have yet to be reported to the scalar processor. If so, the SYNC is faulted; otherwise, the vector processor waits for either of the following conditions to be true:

- A pending or currently executing vector instruction encounters an exception—in which case the SYNC faults
- The vector processor determines that all pending and currently executing vector instructions (including memory instructions in asynchronous memory management mode) will execute to completion without encountering vector exceptions. In that case the SYNC completes.

When SYNC completes, a longword value (which is UNPREDICTABLE) is returned to the scalar processor. The scalar processor writes the longword value to the scalar destination of the MFVP and then proceeds to execute the next instruction. If the scalar destination is in memory, it is UNPREDICTABLE whether the new value of the destination becomes visible to the vector processor until scalar/vector memory synchronization is performed.

When SYNC faults, it is not completed by the vector processor and the scalar processor does not write a longword value to the scalar destination of the MFVP. Also, depending on the exception condition encountered, the SYNC itself takes either a vector processor disabled fault or memory management fault. If both faults are encountered while the vector processor is performing SYNC, then the SYNC itself takes a vector processor disabled fault. Note that it is UNPREDICTABLE whether the vector processor is idle when the fault is generated. After the appropriate fault has been serviced, the SYNC may be returned to through an REI.

SYNC only affects the scalar/vector processor pair that executed it. It has no effect on other processors in a multiprocessor system.

10.7.2 Scalar/Vector Memory Synchronization

Scalar/vector memory synchronization allows software to ensure that the memory activity of the scalar/vector processor pair has ceased and the resultant memory write operations have been made visible to each processor in the pair before the pair's scalar processor proceeds with the next instruction. Two ways are provided to ensure scalar/vector memory synchronization: using MSYNC, which is implemented by the MFVP instruction, and using the MFPR instruction to read the VMAC (Vector Memory Activity Check) internal processor register (IPR). Section 10.7.2.1 discusses MSYNC in detail. Section 10.7.2.2 discusses VMAC in detail.

Scalar/vector memory synchronization does not mean that previously issued vector memory instructions have completed; it only means that the vector and scalar processors are no longer performing memory operations. While both VMAC and MSYNC provide scalar/vector memory synchronization, MSYNC performs significantly more than just that function. In addition, VMAC and MSYNC differ in their exception behavior.

Note that scalar/vector memory synchronization only affects the scalar/vector processor pair that executed it. It has no effect on other processors in a multiprocessor system. Scalar/vector memory synchronization does not ensure that the write operations made by one scalar/vector pair are visible to any other scalar or vector processor. Software can make data visible and shared between a scalar/vector pair and other scalar and vector processors by using the mechanisms described in the *VAX Architecture Reference Manual*. Software must first make a memory write operation by the vector processor visible to its associated scalar processor through scalar/vector memory synchronization before making the write operation visible to other processors. Without performing this scalar/vector memory synchronization, it is UNPREDICTABLE whether the vector memory write will be made visible to other processors even by the mechanisms described in the *VAX Architecture Reference Manual*.

Lastly, waiting for VPSR<BSY> to be clear does not guarantee that a vector write operation is visible to the scalar processor.

10.7.2.1 Memory Instruction Synchronization (MSYNC)

While MSYNC performs scalar/vector memory synchronization, it does more than that. MSYNC allows software to ensure that all previously issued memory instructions of the scalar/vector processor pair are complete and their results made visible before the scalar processor proceeds with the next instruction.

MSYNC is implemented through the nonprivileged MFVP instruction. Arithmetic and asynchronous memory management exceptions encountered by previous vector instructions can cause MSYNC to fault.

Once it issues MSYNC, the scalar processor executes no further instructions until MSYNC completes or faults.

MSYNC completes when the following events occur:

- All previously issued scalar and vector memory instructions have completed.
- All resultant memory write operations (scalar write operations and vector store operations) have been made visible to both the scalar and vector processor.
- No exception that should cause MSYNC to fault has occurred. (See the next paragraph.)

MSYNC faults when any unreported exception has occurred in the production or storage of any result (vector register element or vector control register bit) that MSYNC depends upon. Such results include all elements loaded or stored by a previously issued vector memory instruction as well as any element or control register bit that these elements depend upon.

It is UNPREDICTABLE whether MSYNC faults due to exceptions that occur in the production and storage of results (vector register elements and vector control register bits) that MSYNC does not depend upon. Software should not rely on such exceptions being reported by MSYNC for program correctness.

When MSYNC completes, a longword value (which is UNPREDICTABLE) is returned to the scalar processor, which writes it to the scalar destination of the MFVP. The scalar processor then proceeds to execute the next instruction. If the scalar destination is in memory, it is UNPREDICTABLE whether the new value of the destination becomes visible to the vector processor until another scalar/vector memory synchronization instruction is performed.

When MSYNC faults, it is not ensured that all previously issued scalar and vector memory instructions have finished. In this case, the scalar processor writes no longword value to the scalar destination of the MFVP. Depending on the exception encountered by the vector processor, the MSYNC takes a vector processor disabled fault or memory management fault. Note that it is UNPREDICTABLE whether the vector processor is idle when the fault is generated. After the fault has been serviced, the MSYNC may be returned to through an REI.

Section 10.5.3.3 gives the necessary rules and examples to determine what vector control register elements and vector control register bits MSYNC depends upon.

10.7.2.2 Memory Activity Completion Synchronization (VMAC)

Privileged software needs a way to ensure scalar/vector memory synchronization that will not result in any exceptions being reported. Reading the VMAC internal processor register (IPR) with the privileged MFPR instruction is provided for these situations. It is especially useful for context switching.

Once a MFPR from VMAC is issued by the scalar processor, the scalar processor executes no further instructions until VMAC completes, which it does when the following events occur:

- All vector and scalar memory activities have ceased.
- All resultant memory write operations have been made visible to both the scalar and vector processor.
- A longword value (which is UNPREDICTABLE) is returned to the scalar processor.

After writing the longword value to the scalar destination of the MFPR, the scalar processor then proceeds to execute the next instruction. If the scalar destination is in memory, it is UNPREDICTABLE whether the new value of the destination becomes visible to the vector processor until another scalar/vector memory synchronization operation is performed.

As stated in Section 10.7.2, Scalar/Vector Memory Synchronization, the ceasing of vector and scalar memory activities does not mean that previously issued vector memory instructions have completed. For example, consider a vector memory instruction that has suspended execution due to an asynchronous memory management exception or hardware error. Once it becomes suspended, the instruction will write no further elements and its memory activity will cease. As a result, a subsequently issued VMAC will complete as soon as those write operations that were made by the memory instruction before it was suspended are visible to both the scalar and vector processor. But, after the completion of the VMAC, the memory instruction is not completed and remains suspended.

Vector arithmetic and memory management exceptions of previous vector instructions never fault an MFPR-from-VMAC and never suspend its execution.

10.7.3 Other Synchronization Between the Scalar and Vector Processors

Synchronization between the scalar and vector processors also occurs in the following situations:

- In the absence of pending vector arithmetic exceptions, reading a vector control register using the MFVP instruction waits for all previous write operations to that register to complete. In addition, the scalar processor must wait for the MFVP result to be written before processing other instructions. An MFVP instruction that reads a vector control register must fault if there is any unreported exception that has occurred in the production of the value of the control register.
- Writing to VTBIA or VSAR with MTPR causes the new state of the changed vector IPR to affect the execution of all subsequently issued vector instructions.
- Reading from VPSR with MFPR after writing to VPSR with MTPR causes the new state of VPSR (and VAER if cleared by VPSR<RST>) to affect the execution of subsequently issued vector instructions.

10.7.4 Memory Synchronization Within the Vector Processor (VSYNC)

The vector processor may concurrently execute a number of vector memory instructions through the use of multiple load/store paths to memory. When it is necessary to synchronize the accesses of multiple vector memory instructions the MSYNC instruction can be used; however, there are cases for which this instruction does more than is needed. If it is known that only synchronization between the memory accesses of vector instructions is required, the VSYNC instruction is more efficient.

VSYNC orders the conflicting memory accesses of vector-memory instructions issued after VSYNC with those of vector-memory instructions issued before VSYNC. Specifically, VSYNC forces the access of a memory location by any subsequent vector-memory instruction to wait for (depend upon) the completion of all prior conflicting accesses of that location by previous vector-memory instructions.

VSYNC does not have any synchronizing effect between scalar and vector memory access instructions. VSYNC also has no synchronizing effect between vector load instructions because multiple load accesses cannot conflict. It also does not ensure that previous vector memory management exceptions are reported to the scalar processor.

10.7.5 Required Use of Memory Synchronization Instructions

Table 10–15 shows for all possible pairs of vector or scalar read and write operations to a common memory location, whether one of the scalar/vector memory synchronization instructions or the VSYNC instruction must be issued after the first reference and before the second. Since the MSYNC instruction also includes the VSYNC function, it can always be used instead of VSYNC.

In general, these rules apply to any sequence of instructions that access a common memory location, no matter how many other vector or scalar instructions are issued between the first instruction that accesses the common location and the second instruction that accesses the same location. For example, the following code sequence depicts a vector load followed by a scalar write operation to the same memory location. Between these two instructions are other scalar/vector instructions that do not access the common memory location. A scalar/vector memory synchronization instruction (MSYNC or VMAC) must be executed sometime after the vector read operation and before the scalar write operation to the common location. (Here MSYNC is shown.)

```
VLDL   A, #4, V0
              .
other scalar/vector instructions
that do not access A
              .
MSYNC  Dst
MOVL   R0, A
```

In most cases, MSYNC is the preferred method for ensuring scalar/vector memory synchronization. However, there are special cases, usually encountered by an operating system, when VMAC is more appropriate.

Cases when scalar/vector memory synchronization is required are as follows:

- After a vector instruction that stores to memory and before a peripheral (I/O) data transfer of the stored location is initiated by an application program. This ensures that the value stored will be transferred to the output device. The application must ensure that this requirement is met by using MSYNC. Using VMAC in this case is not sufficient because unlike MSYNC, VMAC does not ensure that all previous vector memory instructions have successfully completed.

- After a vector instruction that stores to memory and before the associated scalar processor can execute a HALT instruction. This ensures that a read operation or modify operation by another processor will access the updated memory value. VMAC is the preferred method for this case.
- Before the vector processor state is saved as a result of power failure. A read or modify operation of the same memory must read the updated value (provided that the duration of the power failure does not exceed the maximum nonvolatile period of the main memory). Also, software is responsible for saving any pending vector processor exception status. VMAC is the preferred method for this case.
- Before a context switch. Software is responsible for ensuring that the vector processor has completed all its memory accesses before performing a context switch. Software is also responsible for saving any pending vector processor exception status. VMAC is the preferred method for this case.

The scalar/vector memory synchronization instructions are the only ones that guarantee that the memory operations of the vector and scalar processors are synchronized. Write operations to I/O space, changes in access mode, machine checks, interprocessor interrupts, execution of a HALT, REI, or interlocked instruction do not make the results of vector instructions that write to memory visible to the scalar processor, I/O subsystem, or other processors. Execution of a scalar/vector memory synchronization instruction must precede any of these mechanisms to ensure synchronization of all system components.

Table 10–15 Possible Pairs of Read and Write Operations When Scalar/Vector Memory Synchronization (M) or VSYNC (V) Is Required Between Instructions That Reference the Same Memory Location

First Reference Second Reference	Scalar Scalar	Scalar Vector	Vector Scalar	Vector Vector
Operation Sequence				
Read, Read	No[1,2]	No[1]	No[1]	No[1]
Read, Write	No[2]	No[3]	M	V[5]
Write, Read	No[2]	M[4]	M	V
Write, Write	No[2]	M[4]	M	V

[1]Scalar/vector memory synchronization or VSYNC is never required between two read accesses to a memory location.

[2]Scalar/vector memory synchronization is never required between two accesses by the VAX scalar processor to a memory location.

[3]The scalar read is synchronous and will have completed before a vector memory operation is issued.

[4]Although a scalar write operation is a synchronous instruction, scalar/vector memory synchronization is required to ensure that the written data is visible to the vector processor before the vector memory reference is executed.

[5]See Section 10.7.5.1 for the conditions when VSYNC is not required between a vector memory read/write pair.

10.7.5.1 When VSYNC Is Not Required

There exist conditions when VSYNC is not required between conflicting vector memory accesses. A VSYNC is not required before a vector memory store instruction (VST/VSCAT) if, for each memory location to be accessed by the store, both of the following conditions are met:

- Each of the store's accesses to the location does not conflict with any access to the location by previously issued vector store instructions. Conflict is avoided in this case because one of the following events occurred:
  - The location is not shared.
  - All accesses to the location by previous store instructions were forced to complete by the issue of an MSYNC or VMAC.
- Each of the store's accesses to the location does not conflict with any access to the location by previously issued vector load (VLD /VGATH) instructions. Conflict is avoided in this case because one of the following events occurred:
  - The location is not shared.
  - All accesses to the location by previous load instructions were forced to complete by the issue of an MSYNC or VMAC.
  - Each of the store's accesses to the location depends on the completion (as seen by the vector processor) of all accesses to the location by previous LOAD instructions. (The examples immediately following demonstrate this concept.)

In all other cases of conflicting vector memory accesses, VSYNC is necessary to ensure correct results.

Examples Where VSYNC Is Not Required

In the following examples, VSYNC is not required because both of the previous conditions have been met for each location accessed by the store instruction:

1

```
VLDL        A, #4, V0
VSTL        V0, A, #4
```

2

```
VLDL        A, #4, V0
VSSUBL      R0, V0, V1
VSTL        V1, A, #4
```

3

```
VLDL/0        A, #4 ,V0
VSMULL/0      #3, V0, V0
VLDL/1        A, #4 ,V1
VVMULL/1      V1, V1, V1
VVMERGE/1     V1, V0, V2
VSTL          V2, A, #4
```

4

```
VLDL         A, #4 ,V0
VSGTRF      #0, V0
VLDL/1      B, #4, V1
VLDL/0      C, #4, V2
VVMERGE/0   V2, V1, V3
VSTL        V3, A, #4
```

Examples Where VSYNC Is Required

In the following examples, VSYNC is required before the vector memory store instruction:

1

```
VLDL/1      A,#4,V0
VSLSSL      #0,V1
VSYNC
VSTL/1      V1,A,#4
```

If the VSYNC is not included, V0 could contain incorrect data at the end of the sequence since the vector processor is allowed to begin the VSTL before the VLDL is finished. This occurs because there is no dependence between the VMR value used by the VLDL and the VSTL.

2

```
VLDL        A, #4, V0
VVMERGE/0   V0, V1, V1
VSYNC
VSTL        V1, A, #4
```

Unless the programmer can ensure that the VMR mask being used by the VVMERGE will force the access of each location by the VSTL to depend on the access to that location by the VLDL, a VSYNC is required. Note that in general, when masked operations provide a conditional path of dependence between conflicting memory accesses, a VSYNC is usually necessary to ensure correct results.

3

```
VSTL        V1, A, #4
MTVLR       #32
VSYNC
VLDL        A+128, #4, V2
```

In this example, the VSTL writes locations A to A+255 and the VLDL reads locations A+128 to A+255. Without the VSYNC, the vector processor is allowed to start reading locations A+128 to A+255 for the VLDL before the vector processor completes (or even starts) writing locations A+128 to A+255 for the VSTL. Consequently, V2[0:31] will not contain V1[32:63], which is the intended result. Note that the rules on when VSYNC is not required (found in Section 10.7.5.1) only apply to waiving the use of VSYNC prior to VST/VSCAT instructions.

4

```
VGATHL      A, V2, V0     ; let at least two elements
                          ; of V2 be equal
VVMULL      V9, V0, V1
VSYNC
VSCATL      V1, A, V2
```

The VSYNC is needed in this example because the VSCATL may store elements of V1 into a common location before the VGATHL has finished loading that location into all the appropriate elements of V0. As a result, elements of V0 fetched from the same location may be unequal. Suppose in the example that V2[0] = V2[63] = 0 and that the original value of location A before the sequence starts is X. Then it is possible without the VSYNC that V0[63] = X*V9[0] and that (A)= V1[63] = V9[63]*V9[0]*X after the sequence completes.

5

```
VLDL        A, #0, V0
VVMULL      V9, V0, V1
VSYNC
VSTL        V1, A, #0
```

The VSYNC is needed in this example because the VSTL may store elements of V1 into A before the VLDL has finished loading all elements of V0 from A. As a result, the elements of V0 may be unequal and so produce incorrect results.

10.8 Memory Management

The vector processor may include its own translation buffer and maintain its own copies of SBR, SLR, SPTEP, P0BR, P0LR, P1BR, and P1LR as a group, or may use the scalar processor's memory management unit. Hardware implementations must ensure that MTPR to these registers update the copy retained by the vector processor. Changes to P0BR, P0LR, P1BR, and P1LR due to a LDPCTX do not update the copies in the vector processor. Before software enables the vector processor again, explicit MTPRs to P0BR, P0LR, P1BR, and P1LR are required to guarantee correct operation.

An MTPR to TBIS must also invalidate the corresponding TB entry in the vector processor, and an MTPR to TBIA must also invalidate the entire TB in the vector processor. However, the vector TB is not invalidated by a LDPCTX instruction. Software can use an MTPR to the Vector TB Invalidate All (VTBIA) register to invalidate only the vector TB. An MTPR to VTBIA results in no operation on a processor that uses a common TB for the scalar and vector processors.

Updates to memory management registers and invalidates of translation buffer entries in the vector processor take place even when the vector processor is disabled (VPSR<VEN> is clear). However, the vector processor may load translation buffer entries only when the vector processor is executing a vector memory access instruction.

The vector processor implements the modify-fault option if its scalar processor implements the virtual-machine option.

Vector memory access instructions must not be used to read or write page tables. If a vector instruction is used to read or write page tables, the results are UNPREDICTABLE.

Vector instructions are not allowed to reference I/O space. If a vector instruction references I/O space, the results are UNPREDICTABLE.

Issuing vector instructions with memory management disabled causes the operation of the vector processor to be UNDEFINED. Disabling memory management when the vector processor is busy (VPSR<BSY> is set) also causes the operation of the vector processor to be UNDEFINED.

10.9 Hardware Errors

A vector processor implementation may experience error conditions (such as chip malfunctions, parity errors, or bus errors) that prevent it from executing and completing instructions and from which it cannot recover through its own means. Such errors are termed hardware errors and may occur at anytime, even when the vector processor is already disabled. Vector processor hardware errors do not normally halt the scalar processor.

At some point after the error condition occurs, the vector processor reports the error to the scalar processor. The reporting may be accomplished through a machine check; or by disabling the vector processor, setting VPSR<IMP>, and generating a vector processor disabled fault when the next vector instruction is issued. After the error is reported, the

appropriate software handler will be invoked to diagnose the vector processor and to determine the severity of the hardware error and whether the vector processor can be restarted.

During execution, software may wish to force the reporting of hardware errors encountered by previous vector instructions before issuing further ones. This can be accomplished by reading the VMAC internal processor register (IPR) and by waiting for VPSR<BSY> to become clear.

An MFPR from VMAC ensures that all pending vector memory instructions have finished or are suspended by an asynchronous memory management exception, and that all vector-processor hardware errors encountered by these instructions are reported by the time the MFPR completes. Errors are handled as follows:

- If the errors are reported by machine check, then the exception is taken either upon the VMAC itself, or upon the instruction immediately following the VMAC.
- If the errors are reported through VPSR<IMP>, the vector processor sets VPSR<IMP> and disables itself by the time the scalar processor completes VMAC. Subsequently, a vector processor disabled fault will occur when the next vector instruction is issued. A read of VPSR immediately after the VMAC completes will find the vector processor disabled and VPSR<IMP> set.

Waiting for VPSR<BSY> to become clear before issuing further instructions ensures that all previous non-memory-access instructions have been finished or are suspended by an asynchronous memory management exception, and that all vector-processor hardware errors encountered by these instructions are reported by the time VPSR<BSY> becomes clear. Errors are handled as follows:

- If the errors are reported by machine check, then the exception is taken either upon the first instruction during which the new state of VPSR<BSY> becomes visible to the scalar processor or upon the instruction immediately thereafter.
- If the errors are reported through VPSR<IMP>, the vector processor sets VPSR<IMP> and disables itself by the time it clears VPSR<BSY>. Subsequently, a vector processor disabled fault will occur when the next vector instruction is issued. The first MFPR instruction which reads VPSR<BSY> as clear will also read VPSR<VEN> as clear and VPSR<IMP> as set.

VMAC does not ensure that hardware errors encountered by pending non-memory-access instructions will be reported. Waiting for VPSR<BSY> to become clear does not ensure that vector-processor hardware errors encountered by vector memory instructions are reported.

Software can force the reporting of hardware errors encountered during the execution of previous vector instructions (both memory and non-memory) by waiting for VPSR<BSY> to become clear and then by issuing an MFPR from VMAC. This technique can be used during scalar context switching to cause hardware errors resulting from the execution of vector

instructions for the current process to be reported before that process is context-switched.

10.10 Vector Memory Access Instructions

There are alignment, stride, address specifier context, and access mode considerations for the vector memory access instructions.

10.10.1 Alignment Considerations

Vector memory access instructions require their vector operands to be naturally aligned in memory. Longwords must be aligned on longword boundaries. Quadwords must be aligned on quadword boundaries. If any vector element is not naturally aligned in memory, an access control violation occurs. For further details, see Section 10.6.1, Vector Memory Management Exception Handling.

The scalar operands need not be naturally aligned in memory.

10.10.2 Stride Considerations

A vector's stride is defined as the number of memory locations (bytes) between the starting address of consecutive vector elements. A contiguous vector that has longword elements has a stride of four; a contiguous vector that has quadword elements has a stride of eight.

10.10.3 Context of Address Specifiers

The base address specifier used by the vector memory access instructions is of byte context, regardless of the data type. Arrays are addressed as byte strings. Index values in array specifiers are multiplied by one, and the amount of autoincrement or autodecrement, when either of these modes is used, is one.

10.10.4 Access Mode

A vector memory access instruction is executed using the access mode in effect when the instruction is issued by the scalar processor.

10.10.5 Memory Instructions

This section describes VAX vector architecture memory instructions.

VLD

Load Memory Data into Vector Register

FORMAT

VLDL *[/M[0 | 1]]* *base, stride, Vc*

VLDQ *[/M[0 | 1]]* *base, stride, Vc*

ARCHITECTURE

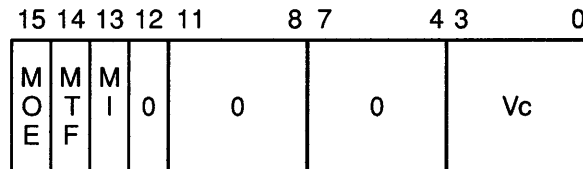
Format

opcode *cntrl.rw, base.ab, stride.rl*

opcodes

34FD	VLDL	Load Longword Vector from Memory to Vector Register
36FD	VLDQ	Load Quadword Vector from Memory to Vector Register

vector control word



exceptions

access control violation
translation not valid
vector alignment

DESCRIPTION

The source operand vector is fetched from memory and is written to vector destination register Vc. The length of the vector is specified by VLR. The virtual address of the source vector is computed using the base address and the stride. The address of element i (0 LEQU i LEQU (VLR-1)) is computed as {base+{i*stride}}. The stride can be positive, negative, or zero.

In VLDL, bits <31:0> of each destination vector element receive the memory data and bits <63:32> are UNPREDICTABLE.

If any vector element operated upon is not naturally aligned in memory, a vector alignment exception occurs.

The results of VLD are unaffected by the setting of cntrl<MI>. For more details about the use of cntrl<MI>, see Section 10.3.3, Modify Intent bit.

If the addressing mode of the BASE operand is immediate, the results of the instruction are UNPREDICTABLE.

An implementation may load the elements of the vector in any order, and more than once. When a vector processor memory management exception occurs, the contents of the destination vector elements are UNPREDICTABLE.

VGATH

Gather Memory Data into Vector Register

FORMAT

VGATHL *[/M[0 | 1]]* *base, Vb, Vc*

VGATHQ *[/M[0 | 1]]* *base, Vb, Vc*

ARCHITECTURE

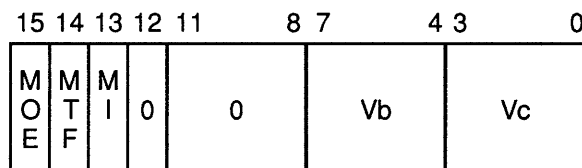
Format

opcode *cntrl.rw, base.ab*

opcodes

35FD	VGATHL	Gather Longword Vector from Memory to Vector Register
37FD	VGATHQ	Gather Quadword Vector from Memory to Vector Register

vector_control_word



exceptions

access control violation
translation not valid
vector alignment

DESCRIPTION

The source operand vector is fetched from memory and is written to vector destination register Vc. The length of the vector is specified by VLR. The virtual address of the vector is computed using the base address and the 32-bit offsets in vector register Vb. The address of element i (0 LEQU i LEQU (VLR-1)) is computed as {base+Vb[i]}. The 32-bit offset can be positive, negative, or zero.

In VGATHL, bits <31:0> of each destination vector element receive the memory data and bits <63:32> are UNPREDICTABLE.

If any vector element operated upon is not naturally aligned in memory, a vector alignment exception occurs.

The results of VGATH are unaffected by the setting of cntrl<MI>. For more details about the use of cntrl<MI>, see Section 10.3.3, Modify Intent bit.

If the addressing mode of the BASE operand is immediate, the results of the instruction are UNPREDICTABLE.

An implementation may load the elements of the vector in any order, and more than once. When a vector processor memory management exception occurs, the contents of the destination vector elements are UNPREDICTABLE.

If the same vector register is used as both source and destination, the result of the VGATH is UNPREDICTABLE.

VST

Store Vector Register Data into Memory

FORMAT

VSTL *[/0 | 1]* *Vc, base, stride*

VSTQ *[/0 | 1]* *Vc, base, stride*

ARCHITECTURE

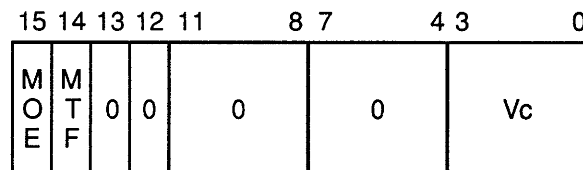
Format

opcode *cntrl.rw, base.ab, stride.rl*

opcodes

9CFD	VSTL	Store Longword Vector from Vector Register to Memory
9EFD	VSTQ	Store Quadword Vector from Vector Register to Memory

vector_control_word



exceptions

access control violation
translation not valid
vector alignment
modify

DESCRIPTION

The source operand in vector register Vc is written to memory. The length of the vector is specified by the Vector Length Register (VLR). The virtual address of the destination vector is computed using the base address and the stride. The address of element i (0 LEQU i LEQU (VLR-1)) is computed as {base+{i*stride}}. The stride can be positive, negative, or zero.

If any vector element operated upon is not naturally aligned in memory, a vector alignment exception occurs.

For a nonzero stride value, an implementation may store the vector elements in parallel; therefore the order in which these elements are stored is UNPREDICTABLE. Furthermore, if the nonzero stride causes

result locations in memory to overlap, then the values stored in the overlapping result locations are also UNPREDICTABLE.

For a stride value of zero, the highest numbered register element destined for the single memory location becomes the final value of that location.

When a vector processor memory management exception occurs, it is UNPREDICTABLE whether the vector processor writes any result location for which an exception did not occur. If the fault condition can be eliminated by software and the instruction restarted then the vector processor will ensure that all destination locations are written.

If the destination vector overlaps the vector instruction control word, base, or stride operand, the result of the instruction is UNPREDICTABLE.

If the addressing mode of the BASE operand is immediate, the results of the instruction are UNPREDICTABLE.

VSCAT

Scatter Vector Register Data into Memory

FORMAT

VSCATL *[/0 | 1]* *Vc, base, Vb*

VSCATQ *[/0 | 1]* *Vc, base, Vb*

ARCHITECTURE

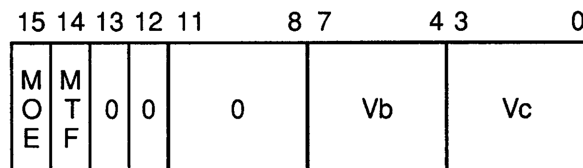
Format

opcode *cntrl.rw, base.ab*

opcodes

9DFD	VSCATL	Scatter Longword Vector from Vector Register to Memory
9FFD	VSCATQ	Scatter Quadword Vector from Vector Register to Memory

vector_control_word



ZK–1460A–GE

exceptions

access control violation
translation not valid
vector alignment
modify

DESCRIPTION

The source vector operand Vc is written to memory. The length of the vector is specified by the Vector Length Register (VLR) register. The virtual address of the destination vector is computed using the base address operand and the 32-bit offsets in vector register Vb. The address of element i (0 LEQU i LEQU (VLR-1)) is computed as {base+Vb[i]}. The 32-bit offset can be positive, negative, or zero.

If any vector element operated upon is not naturally aligned in memory, a vector alignment exception occurs.

An implementation may store the vector elements in parallel; therefore, the order in which elements are stored to different memory locations is UNPREDICTABLE. In the case where multiple elements are destined for the same memory location, the highest numbered element among them becomes the final value of that location.

When a vector processor memory management exception occurs, it is UNPREDICTABLE whether the vector processor writes any result location for which an exception did not occur. If the fault condition can be eliminated by software and the instruction restarted, then the vector processor will ensure that all destination locations are written.

If the destination vector overlaps the vector instruction control word or base operand, the result of the instruction is UNPREDICTABLE.

If the addressing mode of the BASE operand is immediate, the results of the instruction are UNPREDICTABLE.

10.11 Vector Integer Instructions

This section describes VAX vector architecture integer instructions.

VADDL

Vector Integer Add

FORMAT

vector + vector:

VVADDL *[/0 | 1]* *Va, Vb, Vc*

scalar + vector:

VSADDL *[/0 | 1]* *scalar, Vb, Vc*

ARCHITECTURE

Format

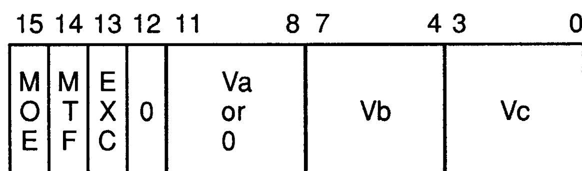
vector + vector: *opcode* *cntrl.rw*

scalar + vector: *opcode* *cntrl.rw, addend.rl*

opcodes

80FD	VVADDL	Vector Vector Add Longword
81FD	VSADDL	Vector Scalar Add Longword

vector_control_word



exceptions

integer overflow

DESCRIPTION

The scalar addend or Va operand is added, elementwise, to vector register Vb and the 32-bit sum is written to vector register Vc. Only bits <31:0> of each vector element participate in the operation. Bits <63:32> of the elements of vector register Vc are UNPREDICTABLE. The length of the vector is specified by the Vector Length Register (VLR).

If integer overflow is detected and cntrl<EXC> is set, the exception type and destination register number are recorded in the Vector Arithmetic Exception Register (VAER) and the vector operation is allowed to complete. On integer overflow, the low-order 32 bits of the true result are stored in the destination element.

VCMPL

Vector Integer Compare

FORMAT

vector–vector:

{ **VVGTRL** | **VVEQLL** | **VVLSSL** | **VVLEQL** | **VVNEQL** | **VVGEQL** } *[/0 | 1]* *Va, Vb*

scalar–vector:

{ **VSGTRL** | **VSEQLL** | **VSLSSL** | **VSLEQL** | **VSNEQL** | **VSGEQL** } *[/0 | 1]* *src, Vb*

ARCHITECTURE

Format

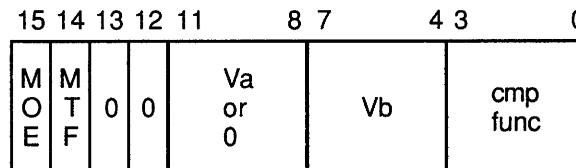
vector–vector: *opcode* *cntrl.rw*

scalar–vector: *opcode* *cntrl.rw, src.rl*

opcodes

C0FD	VVCMPL	Vector Vector Compare Longword
C1FD	VSCMPL	Vector Scalar Compare Longword

vector_control_word



ZK–1462A–GE

The condition being tested is determined by cntrl<2:0>, as follows:

Value of cntrl<2:0>	Meaning
0	Greater than
1	Equal
2	Less than
3	Reserved[1]
4	Less than or equal
5	Not equal
6	Greater than or equal
7	Reserved[1]

[1]Vector integer compare instructions that specify reserved values of cntrl<2:0> produce UNPREDICTABLE results.

DESCRIPTION

The scalar or Va operand is compared, elementwise, with vector register Vb. The length of the vector is specified by the Vector Length Register (VLR). For each element comparison, if the specified relationship is true, the Vector Mask Register bit (VMR<i>) corresponding to the vector element is set to one; otherwise, it is cleared. If cntrl<MOE> is set, VMR bits corresponding to elements that do not match cntrl<MTF> are left unchanged. VMR bits beyond the vector length are left unchanged. Only bits <31:0> of each vector element participate in the operation.

VMULL

Vector Integer Multiply

FORMAT

*vector * vector:*

VVMULL *[/V[0 | 1]]* *Va, Vb, Vc*

*scalar * vector:*

VSMULL *[/V[0 | 1]]* *scalar, Vb, Vc*

ARCHITECTURE

Format

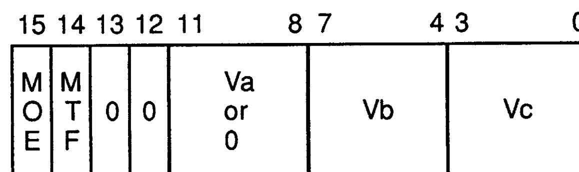
*vector * vector:* *opcode* *cntrl.rw*

*scalar * vector:* *opcode* *cntrl.rw, mulr.rl*

opcodes

A0FD	VVMULL	Vector Vector Multiply Longword
A1FD	VSMULL	Vector Scalar Multiply Longword

vector_control_word



exceptions

integer overflow

DESCRIPTION

The scalar multiplier or vector operand Va is multiplied, elementwise, by vector operand Vb and the least significant 32 bits of the signed 64-bit product are written to vector register Vc. Only bits <31:0> of each vector element participate in the operation. Bits <63:32> of the elements of vector register Vc are UNPREDICTABLE. The length of the vector is specified by the Vector Length Register (VLR).

If integer overflow is detected and cntrl<EXC> is set, the exception condition type and destination register number are recorded in the Vector Arithmetic Exception Register (VAER) and the vector operation is allowed to complete. On integer overflow, the low-order 32 bits of the true result are stored in the destination element.

VSUBL

Vector Integer Subtract

FORMAT

vector–vector:

VVSUBL *[/V[0 | 1]]* *Va, Vb, Vc*

scalar–vector:

VSSUBL *[/V[0 | 1]]* *scalar, Vb, Vc*

ARCHITECTURE

Format

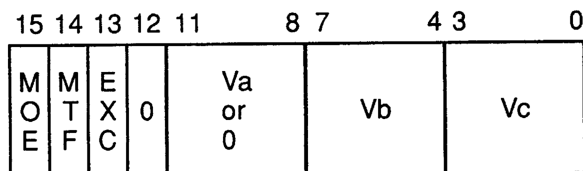
vector–vector: *opcode* *cntrl.rw*

scalar–vector: *opcode* *cntrl.rw, min.rl*

opcodes

88FD	VVSUBL	Vector Vector Subtract Longword
89FD	VSSUBL	Vector Scalar Subtract Longword

vector_control_word



exceptions

integer overflow

DESCRIPTION

The vector operand Vb is subtracted, elementwise, from the scalar minuend or vector operand Va. The 32-bit difference is written to vector register Vc. Only bits <31:0> of each vector element participate in the operation. Bits <63:32> of the elements of vector register Vc are UNPREDICTABLE. The length of the vector is specified by the Vector Length Register (VLR).

If integer overflow is detected and cntrl<EXC> is set, the exception condition type and destination register number are recorded in the Vector Arithmetic Exception Register (VAER) and the vector operation is allowed

to complete. On integer overflow, the low-order 32 bits of the true result are stored in the destination element.

10.12 Vector Logical and Shift Instructions

This section describes VAX vector architecture logical and shift instructions.

VBIC, VBIS, and VXOR

Vector Logical Functions

FORMAT

vector op vector:

{ **VVBISL** | **VVXORL** | **VVBICL** } *[/V[0 | 1]]* *Va, Vb, Vc*

vector op scalar:

{ **VSBISL** | **VSXORL** | **VSBICL** } *[/V[0 | 1]]* *scalar, Vb, Vc*

ARCHITECTURE

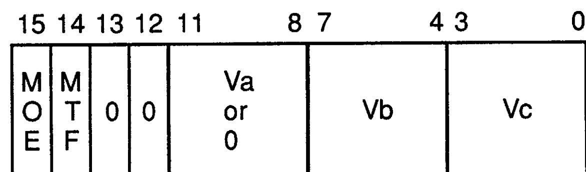
Format

vector op vector:	*opcode*	*cntrl.rw*
vector op scalar:	*opcode*	*cntrl.rw, src.rl*

opcodes

C8FD	VVBISL	Vector Vector Bit Set Longword
E8FD	VVXORL	Vector Vector Exclusive-OR Longword
CCFD	VVBICL	Vector Vector Bit Clear Longword
C9FD	VSBISL	Vector Scalar Bit Set Longword
E9FD	VSXORL	Vector Scalar Exclusive-OR Longword
CDFD	VSBICL	Vector Scalar Bit Clear Longword

vector_control_word



exceptions

None.

DESCRIPTION

The scalar src or vector operand Va is combined, elementwise, using the specified Boolean function, with vector register Vb and the result is written to vector register Vc. Only bits <31:0> of each vector element participate in the operation. Bits <63:32> of the elements of Vb are written into bits <63:32> of the corresponding elements of Vc. The length of the vector is specified by the Vector Length Register (VLR).

VSL

Vector Shift Logical

FORMAT

vector shift count:

{ **VVSRLL** / **VVSLLL** } *[/V[0 | 1]]* *Va, Vb, Vc*

scalar shift count:

{ **VSSRLL** / **VSSLLL** } *[/V[0 | 1]]* *cnt, Vb, Vc*

ARCHITECTURE

Format

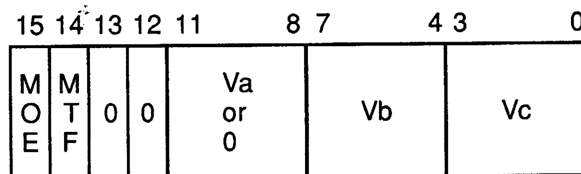
vector shift count: *opcode* *cntrl.rw*

scalar shift count: *opcode* *cntrl.rw, cnt.rl*

opcodes

E0FD	VVSRLL	Vector Vector Shift Right Logical Longword
E4FD	VVSLLL	Vector Vector Shift Left Logical Longword
E1FD	VSSRLL	Vector Scalar Shift Right Logical Longword
E5FD	VSSLLL	Vector Scalar Shift Left Logical Longword

vector_control_word



exceptions

None.

DESCRIPTION

Each element in vector register Vb is shifted logically left or right 0 to 31 bits as specified by a scalar count operand or vector register Va. The shifted results are written to vector register Vc. Zero bits are propagated into the vacated bit positions. Only bits <4:0> of the count operand and bits <31:0> of each Vb element participate in the operation. Bits <63:32> of the elements of vector register Vc are UNPREDICTABLE. The length of the vector is specified by the Vector Length Register (VLR).

10.13 Vector Floating-Point Instructions

The VAX vector architecture provides instructions for operating on F_floating, D_floating, and G_floating operand formats. The floating-point arithmetic instructions are add, subtract, compare, multiply, and divide. Data conversion instructions are provided to convert operands between D_floating, G_floating, F_floating, and longword integer.

Rounding is performed using standard VAX rounding rules. The accuracy of the vector floating-point instructions matches that of the scalar floating-point instructions. Refer to the section on floating-point instructions in the *VAX Architecture Reference Manual* for more information.

10.13.1 Vector Floating-Point Exception Conditions

All vector floating-point exception conditions occur asynchronously with respect to the scalar processor. These exception conditions do not interrupt the scalar processor. If the exception condition is enabled, then the exception condition type and destination register number are recorded in the Vector Arithmetic Exception Register (VAER), and a reserved operand in the format of the instruction's data type is written into the destination register element. Encoded in this reserved operand is the exception condition type. After recording the exception and writing the appropriate result into the destination register element, the instruction encountering the exception continues executing to completion.

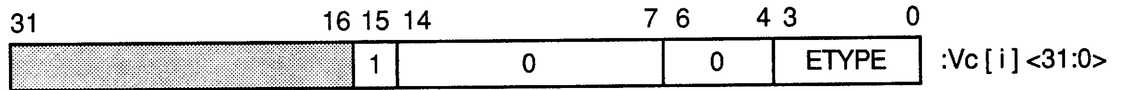
If a vector convert floating to integer instruction encounters a source element that is a reserved operand, an UNPREDICTABLE result rather than a reserved operand is written into the destination register element.

Figure 10–13 shows the encoding of the reserved operand that is written for vector floating-point exceptions. Consistent with the definition of a reserved operand (as defined in Section 10.13.2, Floating-Point Instructions) the sign bit (bit <15>) is one and the exponent (bits <14:7> for F_floating and D_floating, and bits <14:4> for G_floating) is zero. When the reserved operand is written in F_floating or D_floating format, bits <6:4> are also zero. The exception condition type (ETYPE) is encoded in bits <3:0>, as shown in Table 10–16. If a reserved operand is divided by zero, both ETYPE bits may be set. The state of all other bits in the result (denoted by shading) is UNPREDICTABLE.

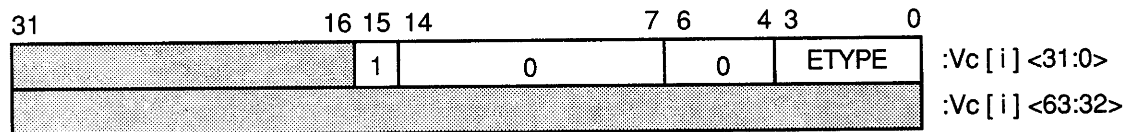
If the floating underflow exception condition is suppressed by cntrl<EXC>, a zero result is written to the destination register element and no further action is taken. Floating overflow, floating divide by zero, and floating reserved operand are always enabled.

Figure 10–13 Encoding of the Reserved Operand

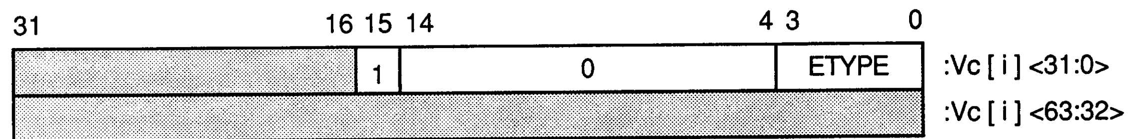
a. F_floating



b. D_floating



c. G_floating



ZK–1464A–GE

Table 10–16 Encoding of the Exception Condition Type (ETYPE)

Bit	Exception Condition Type
<0>	Floating underflow
<1>	Floating divide by zero
<2>	Floating reserved operand
<3>	Floating overflow

10.13.2 Floating-Point Instructions

This section describes VAX vector architecture floating-point instructions.

VADD

Vector Floating Add

FORMAT

vector + vector:

{ **VVADDF** | **VVADDD** | **VVADDG** } *[/U[0 | 1]]* *Va, Vb, Vc*

scalar + vector:

{ **VSADDF** | **VSADDD** | **VSADDG** } *[/U[0 | 1]]* *scalar, Vb, Vc*

ARCHITECTURE

Format

vector + vector:

opcode *cntrl.rw*

scalar + vector (F_floating):

opcode *cntrl.rw, addend.rl*

scalar + vector (D_ and G_floating):

opcode *cntrl.rw, addend.rq*

opcodes

84FD	VVADDF	Vector Vector Add F_Floating
85FD	VSADDF	Vector Scalar Add F_Floating
86FD	VVADDD	Vector Vector Add D_Floating
87FD	VSADDD	Vector Scalar Add D_Floating
82FD	VVADDG	Vector Vector Add G_Floating
83FD	VSADDG	Vector Scalar Add G_Floating

vector_control_word

15	14	13	12	11 ... 8	7 ... 4	3 ... 0
MOE	MTF	EXC	0	Va or 0	Vb	Vc

ZK-1461A-GE

exceptions

floating overflow
floating reserved operand
floating underflow

DESCRIPTION

The source addend or vector operand Va is added, elementwise, to vector register Vb and the sum is written to vector register Vc. The length of the vector is specified by the Vector Length Register (VLR).

In VxADDF, only bits <31:0> of each vector element participate in the operation. Bits <63:32> of the destination vector elements are UNPREDICTABLE.

If a floating underflow occurs when cntrl<EXC> is set or if a floating overflow or floating reserved operand occurs, an encoded reserved operand is stored as the result and the exception condition type and destination register number are recorded in the Vector Arithmetic Exception Register (VAER). The vector operation is then allowed to complete. If cntrl<EXC> is clear, zero is written to the destination element when an exponent underflow occurs and no other action is taken.

VCMP

Vector Floating Compare

FORMAT

vector–vector: { **VVGTRF**, **VVGTRD**, **VVGTRG**, **VVEQLF**, **VVEQLD**, **VVEQLG**, **VVLSSF**, **VVLSSD**, **VVLSSG**, **VVLEQF**, **VVLEQD**, **VVLEQG**, **VVNEQF**, **VVNEQD**, **VVNEQG**, **VVGEQF**, **VVGEQD**, **VVGEQG** } *[/U[0 | 1]]* *Va, Vb*

scalar–vector: { **VSGTRF**, **VSGTRD**, **VSGTRG**, **VSEQLF**, **VSEQLD**, **VSEQLG**, **VSLSSF**, **VSLSSD**, **VSLSSG**, **VSLEQF**, **VSLEQD**, **VSLEQG**, **VSNEQF**, **VSNEQD**, **VSNEQG**, **VSGEQF**, **VSGEQD**, **VSGEQG** } *[/U[0 | 1]]* *src, Vb*

ARCHITECTURE

Format

vector–vector:

opcode *cntrl.rw*

scalar–vector (F_floating):

opcode *cntrl.rw, src.rl*

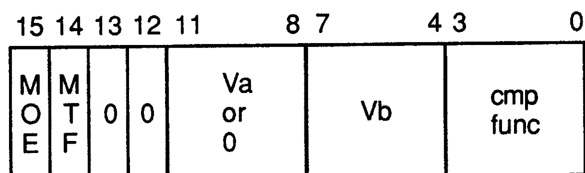
scalar–vector (D_ and G_floating):

opcode *cntrl.rw, src.rq*

opcodes

C4FD	VVCMPF	Vector Vector Compare F_floating
C5FD	VSCMPF	Vector Scalar Compare F_floating
C6FD	VVCMPD	Vector Vector Compare D_floating
C7FD	VSCMPD	Vector Scalar Compare D_floating
C2FD	VVCMPG	Vector Vector Compare G_floating
C3FD	VSCMPG	Vector Scalar Compare G_floating

vector_control_word



The condition being tested is determined by cntrl<2:0>, as follows:

Value of cntrl<2:0>	Meaning
0	Greater than
1	Equal
2	Less than
3	Reserved[1]
4	Less than or equal
5	Not equal
6	Greater than or equal
7	Reserved[1]

[1]Vector integer compare instructions that specify reserved values of cntrl<2:0> produce UNPREDICTABLE results.

Note: Cntrl<3> should be zero; if it is set, the results of the instruction are UNPREDICTABLE.

exceptions

floating reserved operand

DESCRIPTION

The scalar or vector operand Va is compared, elementwise, with vector register Vb. The length of the vector is specified by the Vector Length Register (VLR). For each element comparison, if the specified relationship is true, the Vector Mask Register bit (VMR<i>) corresponding to the vector element is set to one, otherwise it is cleared. If cntrl<MOE> is set, VMR bits corresponding to elements that do not match cntrl<MTF> are left unchanged. VMR bits beyond the vector length are left unchanged. If an element being compared is a reserved operand, VMR<i> is UNPREDICTABLE. In VxCMPF, only bits <31:0> of each vector element participate in the operation.

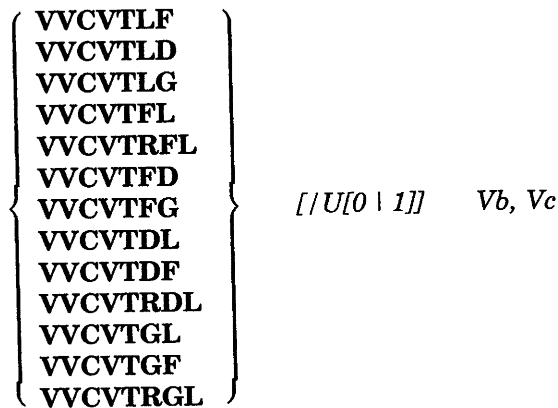
If a floating reserved operand exception occurs, the exception condition type is recorded in the Vector Arithmetic Exception Register (VAER) and the vector operation is allowed to complete.

Note that for this instruction, no bits are set in the VAER destination register mask when an exception occurs.

VVCVT

Vector Convert

FORMAT



ARCHITECTURE

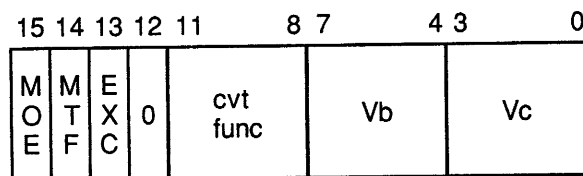
Format

opcode *cntrl.rw*

opcodes

ECFD VVCVT Vector Convert

vector_control_word



Cntrl<11:8> specifies the conversion to be performed, as follows:

cntrl<11:8>	Meaning
1 1 1 1	CVTRGL (Convert Rounded G_Floating to Longword)
1 1 1 0	Reserved[1]
1 1 0 1	CVTGF (Convert Rounded G_Floating to F_Floating)

[1]Vector convert instructions that specify reserved values of cntrl<11:8> produce UNPREDICTABLE results.

cntrl<11:8>	Meaning
1 1 0 0	CVTGL (Convert Truncated G_Floating to Longword)
1 0 1 1	Reserved[1]
1 0 1 0	CVTRD (Convert Rounded D_Floating to Longword)
1 0 0 1	CVTDF (Convert Rounded D_Floating to F_Floating)
1 0 0 0	CVTDL (Convert Truncated D_Floating to Longword)
0 1 1 1	CVTFG (Convert F_Floating to G_Floating (exact))
0 1 1 0	CVTFD (Convert F_Floating to D_Floating (exact))
0 1 0 1	CVTRF (Convert Rounded F_Floating to Longword)
0 1 0 0	CVTFL (Convert Truncated F_Floating to Longword)
0 0 1 1	CVTLG (Convert Longword to G_Floating (exact))
0 0 1 0	CVTLD (Convert Longword to D_Floating (exact))
0 0 0 1	CVTLF (Convert Rounded Longword to F_Floating)
0 0 0 0	Reserved[1]

[1]Vector convert instructions that specify reserved values of cntrl<11:8> produce UNPREDICTABLE results.

exceptions

floating overflow
floating reserved operand
floating underflow
integer overflow

DESCRIPTION

The vector elements in vector register Vb are converted and results are written to vector register Vc. Cntrl<11:8> specifies the conversion to be performed. The length of the vector is specified by the Vector Length Register (VLR). Bits <63:32> of Vc are UNPREDICTABLE for instructions that convert from D_floating or G_floating to F_floating or longword. When CVTRGL, CVTRDL, and CVTRFL round, the rounding is done in sign magnitude, before conversion to two's complement.

If an integer overflow occurs when cntrl<EXC> is set, the low-order 32 bits of the true result are written to the destination element as the result, and the exception condition type and destination register number are recorded in the Vector Arithmetic Exception Register (VAER). The vector operation is then allowed to complete. If integer overflow occurs when cntrl<EXC> is clear, the low-order 32 bits of the true result are written to the destination element, and no other action is taken.

For vector convert floating to integer, where the source element is a reserved operand, the value written to the destination element is UNPREDICTABLE. In addition, the exception type and destination register number are recorded in the VAER. The vector operation is then allowed to complete.

For vector convert floating to floating instructions, if floating underflow occurs when cntrl<EXC> is clear, zero is written to the destination element, and no other action is taken. The vector operation is then allowed to complete.

For vector convert floating to floating instructions, if floating underflow occurs with cntrl<EXC> set or if a floating overflow or reserved operand occurs, an encoded reserved operand is written to the destination element, and the exception condition type and destination register number are recorded in the VAER. The vector operation is then allowed to complete.

VDIV

Vector Floating Divide

FORMAT

vector / vector:

{ **VVDIVF** / **VVDIVD** / **VVDIVG** } *[/U[0 | 1]]* *Va, Vb, Vc*

scalar / vector:

{ **VSDIVF** / **VSDIVD** / **VSDIVG** } *[/U[0 | 1]]* *scalar, Vb, Vc*

ARCHITECTURE

Format

vector/vector:

opcode *cntrl.rw*

scalar/vector (F_floating):

opcode *cntrl.rw, divd.rl*

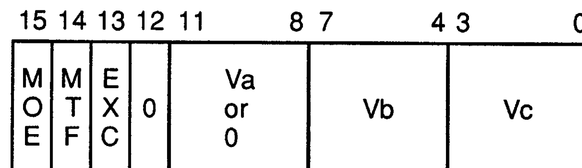
scalar/vector (D_ and G_floating):

opcode *cntrl.rw, divd.rq*

opcodes

ACFD	VVDIVF	Vector Vector Divide F_floating
ADFD	VSDIVF	Vector Scalar Divide F_floating
AEFD	VVDIVD	Vector Vector Divide D_floating
AFFD	VSDIVD	Vector Scalar Divide D_floating
AAFD	VVDIVG	Vector Vector Divide G_floating
ABFD	VSDIVG	Vector Scalar Divide G_floating

vector_control_word



exceptions

floating divide by zero
floating overflow
floating reserved operand
floating underflow

DESCRIPTION

The scalar dividend or vector register Va is divided, elementwise, by the divisor in vector register Vb and the quotient is written to vector register Vc. The length of the vector is specified by the Vector Length Register (VLR).

In VxDIVF, only bits <31:0> of each vector element participate in the operation; bits <63:32> of the destination vector elements are UNPREDICTABLE.

If a floating underflow occurs when cntrl<EXC> is set or if a floating overflow, divide by zero or reserved operand occurs, an encoded reserved operand is stored as the result and the exception condition type and destination register number are recorded in the Vector Arithmetic Exception Register (VAER). The vector operation is then allowed to complete. If cntrl<EXC> is clear, zero is written to the destination element when an exponent underflow occurs and no other action is taken.

VMUL

Vector Floating Multiply

FORMAT

*vector * vector:*

{ **VVMULF** | **VVMULD** | **VVMULG** } *[/U[0 | 1]]* *Va, Vb, Vc*

*scalar * vector:*

{ **VSMULF** | **VSMULD** | **VSMULG** } *[/U[0 | 1]]* *scalar, Vb, Vc*

ARCHITECTURE

Format

vector * vector:

opcode *cntrl.rw*

scalar * vector (F_floating):

opcode *cntrl.rw, mulr.rl*

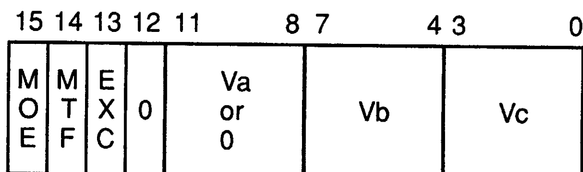
scalar * vector (D_ and G_floating):

opcode *cntrl.rw, mulr.rq*

opcodes

A4FD	VVMULF	Vector Vector Multiply F_floating
A5FD	VSMULF	Vector Scalar Multiply F_floating
A6FD	VVMULD	Vector Vector Multiply F_floating
A7FD	VSMULD	Vector Scalar Multiply D_floating
A2FD	VVMULG	Vector Vector Multiply G_floating
A3FD	VSMULG	Vector Scalar Multiply G_floating

vector_control_word



exceptions

floating overflow
floating reserved operand
floating underflow

DESCRIPTION

The multiplicand in vector register Vb is multiplied, elementwise, by the scalar multiplier or vector operand Va and the product is written to vector register Vc. The length of the vector is specified by the Vector Length Register (VLR).

In VxMULF, only bits <31:0> of each vector element participate in the operation. Bits <63:32> of the destination vector elements are UNPREDICTABLE.

If a floating underflow occurs when cntrl<EXC> is set or if a floating overflow or reserved operand occurs, an encoded reserved operand is stored as the result and the exception condition type and destination register number are recorded in the Vector Arithmetic Exception Register (VAER). The vector operation is then allowed to complete. If cntrl<EXC> is clear, zero is written to the destination element when an exponent underflow occurs and no other action is taken.

VSUB

Vector Floating Subtract

FORMAT

vector–vector:

{ **VVSUBF** / **VVSUBD** / **VVSUBG** } *[/U[0 | 1]]* *Va, Vb, Vc*

scalar–vector:

{ **VSSUBF** / **VSSUBD** / **VSSUBG** } *[/U[0 | 1]]* *scalar, Vb, Vc*

ARCHITECTURE

Format

vector–vector:

opcode *cntrl.rw*

scalar–vector (F_floating):

opcode *cntrl.rw, min.rl*

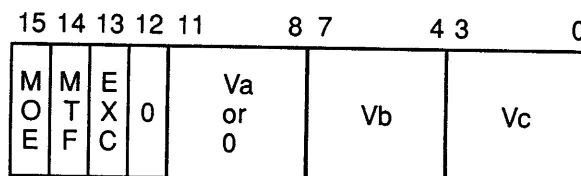
scalar–vector (D_ and G_floating):

opcode *cntrl.rw, min.rq*

opcodes

8CFD	VVSUBF	Vector Vector Subtract F_floating
8DFD	VSSUBF	Vector Scalar Subtract F_floating
8EFD	VVSUBD	Vector Vector Subtract D_floating
8FFD	VSSUBD	Vector Scalar Subtract D_floating
8AFD	VVSUBG	Vector Vector Subtract G_floating
8BFD	VSSUBG	Vector Scalar Subtract G_floating

vector_control_word



exceptions

floating overflow
floating reserved operand
floating underflow

DESCRIPTION

Vector register Vb is subtracted, elementwise, from the scalar minuend or vector register Va and the difference is written to vector register Vc. The length of the vector is specified by the Vector Length Register (VLR).

In VxSUBF, only bits <31:0> of each vector element participate in the operation; bits <63:32> of the destination vector elements are UNPREDICTABLE.

If a floating underflow occurs when cntrl<EXC> is set or if a floating overflow or reserved operand occurs, an encoded reserved operand is stored as the result and the exception condition type and destination register number are recorded in the Vector Arithmetic Exception Register (VAER). The vector operation is then allowed to complete. If cntrl<EXC> is clear, zero is written to the destination element when an exponent underflow occurs and no other action is taken.

10.14 Vector Edit Instructions

This section describes VAX vector architecture edit instructions.

VMERGE

Vector Merge

FORMAT

vector vector merge:

VVMERGE *[/0 | 1]* *Va, Vb, Vc*

vector scalar merge:

{ **VSMERGE**, **VSMERGEF**, **VSMERGED**, **VSMERGEG** } *[/0 | 1]* *src, Vb, Vc*

ARCHITECTURE

Format

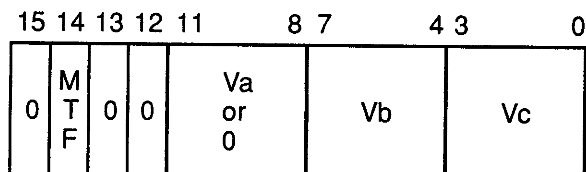
vector-vector: *opcode* *cntrl.rw*

vector-scalar: *opcode* *cntrl.rw,src.rq*

opcodes

EEFD	VVMERGE	Vector Vector Merge
EFFD	VSMERGE	Vector Scalar Merge

vector_control_word



exceptions

None.

DESCRIPTION

The scalar src or vector operand Va is merged, elementwise, with vector register Vb and the resulting vector is written to vector register Vc. The length of the vector operation is specified by the Vector Length Register (VLR).

For each vector element, i, if the corresponding Vector Mask Register bit (VMR<i>) matches cntrl<MTF>, src or Va[i] is written to the destination vector element Vc[i]. If VMR<i> does not match cntrl<MTF>, Vb[i] is written to the destination vector element.

IOTA

Generate Compressed Iota Vector

FORMAT

IOTA *[/0 | 1]* *stride, Vc*

ARCHITECTURE

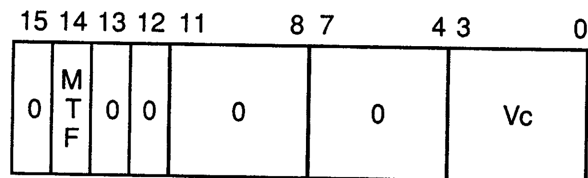
Format

opcode *cntrl.rw, stride.rl*

opcodes

EDFD	IOTA	Generate Compressed Iota Vector

vector_control_word



exceptions

None.

DESCRIPTION

IOTA constructs a vector of offsets for use by the vector gather/scatter instructions VGATH and VSCAT.

IOTA first generates an iota vector of length VLR using the stride operand. An iota vector is a vector whose first element is zero and whose subsequent elements are spaced by the stride increment. The stride can be positive, negative, or zero. For example:

```
0*stride, 1*stride, 2*stride, 3*stride, ..., {VLR-1}*stride
```

The iota vector is then compressed using the contents of the Vector Mask Register (VMR). Elements of the iota vector for which the corresponding Vector Mask Register bit matches cntrl<MTF> are written in contiguous elements of the destination vector register Vc. Only bits <31:0> of each iota and destination vector element participate in the operation. Bits <63:32> of the destination vector elements are UNPREDICTABLE.

The number of elements written to Vc is returned in the Vector Count Register (VCR). The values of elements in the destination vector register between the new value of VCR and the vector length are UNPREDICTABLE.

Note: **If a large value is specified for the stride.rl operand, there is a chance for integer overflow during calculation of the "tmp <- tmp + stride" step. In this case, the overflow is ignored. For example:**

```
tmp  <-  tmp  + stride

Value of tmp before above step:  FFFFFF00
Value of Stride:                 FFFFFF00

Value of tmp + stride:         1 FFFFFE00

Since the overflow is ignored, the new value of tmp
is FFFFFE00.
```

10.15 Miscellaneous Instructions

This section describes VAX vector architecture miscellaneous instructions.

MFVP

Move from Vector Processor

FORMAT

{ MFVCR, MFVLR, MFVMRLO, MFVMRHI, SYNCH, MSYNCH } *dst*

ARCHITECTURE

Format

opcode *regnum.rw, dst.wl*

opcodes

31FD	MFVP	Move from Vector Processor

vector_control_word

None.

exceptions

None.

MFVP instructions that specify reserved values of the regnum operand produce UNPREDICTABLE results.

DESCRIPTION

This instruction can be used to read the Vector Count, Length, and Mask Registers, and to synchronize a scalar processor with its associated vector processor.

When the scalar processor issues an MFVP instruction to the vector processor, the scalar processor waits for the MFVP result to be written before processing other instructions.

MFVP from VCR or VLR does not read that register until all previous write operations to the register are completed. MFVP from VMR<31:0> or VMR<63:32> does not read that longword of VMR until all previous write operations to the same longword of VMR are completed; however, this is not true for previous write operations to the other longword.

SYNC allows software to ensure that the unreported exceptions of all previously issued vector instructions (including vector memory instructions in asynchronous memory management mode) are detected and reported to the scalar processor before the scalar processor proceeds with further instructions. For more details about SYNC and its exception reporting nature refer to Section 10.7.1, Scalar/Vector Instruction Synchronization.

MSYNC allows software to ensure that all previously issued memory instructions of the scalar/vector processor pair are complete before the scalar processor proceeds with further instructions. For more details about MSYNC and its exception reporting nature, refer to Section 10.7.2, Memory Instruction Synchronization.

The value of the vector control register (VCR, VLR, VMR<31:0>, VMR<63:32>) delivered by an MFVP depends upon the value of certain vector register elements and vector control register bits. Unreported exceptions that occur in the production of these elements and control register bits are reported by the vector processor prior to the completion of the MFVP from the vector control register.

In addition, there are vector register elements and vector control register bits that the value of a vector control register delivered by an MFVP does not depend upon. It is UNPREDICTABLE whether unreported exceptions that occur in the production of these elements and control register bits are reported by the vector processor prior to the completion of the MFVP from the vector control register. Software must not rely upon the reporting of these exceptions prior to the completion of the MFVP for the correctness of program results.

Section 10.5.3.3, Dependences Among Vector Results, gives the necessary rules to determine what vector control register elements and vector control register bits the value of a vector control register delivered by an MFVP depends upon. Examples of MFVP exception reporting using these rules are found in Section 10.6.5.

When a vector arithmetic exception or memory management exception (in asynchronous memory management mode) is reported prior to the completion of an MFVP, the following occur:

- The operation of the MFVP does not complete.
- No longword result is written to the scalar destination of the MFVP by the scalar processor.
- The MFVP itself (rather than the next vector instruction) takes either a vector processor disabled fault or a memory management fault.

After the appropriate fault has been serviced, the MFVP may be returned to through an REI. If both exception conditions are encountered by an MFVP, then the MFVP itself takes a vector processor disabled fault. In this case, after the vector processor disabled fault has been serviced, returning to the MFVP instruction will cause the asynchronous memory management exception to be reported.

MTVP

Move to Vector Processor

FORMAT

{ **MTVCR** / **MTVLR** / **MTVMRLO** / **MTVMRHI** } *src*

ARCHITECTURE

Format

opcode *regnum.rw, src.rl*

opcodes

A9FD	MTVP	Move to Vector Processor

vector_control_word

None.

exceptions

None.

Move to Vector Processor instructions that specify reserved values of the regnum operand produce UNPREDICTABLE results.

DESCRIPTION

This instruction can be used to write the Vector Count, Length, and Mask Registers.

The new value of VCR, VLR, or VMR does not affect any prior instructions. The new value remains in effect for all subsequent vector instructions executed until a new value is loaded.

VSYNC

Synchronize Vector Memory Access

FORMAT

VSYNCH

ARCHITECTURE

Format

opcode *regnum.rw*

opcodes

A8FD	VSYNC	Synchronize Vector Memory Access

vector_control_word

None.

exceptions

None.

Synchronize Vector Memory Access instructions that specify reserved values of the regnum operand produce UNPREDICTABLE results.

DESCRIPTION

The VSYNC instruction can be used to synchronize memory access within the vector processor. The instruction allows software to order the conflicting memory accesses of vector-memory instructions issued after VSYNC with those of vector-memory instructions issued before VSYNC. Specifically, VSYNC forces the access of a memory location by any subsequent vector-memory instruction to wait for (depend upon) the completion of all prior conflicting accesses of that location by previous vector-memory instructions. See Section 10.7.1 for more details.

See Section 10.7.5, Required Use of Memory Synchronization Instructions, for the conditions when VSYNC is not required before a vector store instruction.