Architecture Specification for Vector Extension to Nios II ISA

Revision 0.8

Draft only, do not distriute widely

Jason Yu

System-On-Chip Research Lab,

Electrical & Computer Engineering,

University of British Columbia

jasony@ece.ubc.ca

May 9, 2008

Contents

1 Introduction

A vector processor is a single-instruction-multiple-data (SIMD) array of virtual processors (VPs). The number of VPs is the same as the vector length (VL). All VPs execute the same operation specified by a single vector instruction. Physically, the VPs are grouped in parallel datapaths called *vector lanes*, each containing a section of the vector register file and a complete copy of all functional units.

This vector architecture is defined as a co-processor unit to the Altera Nios II soft processor. The ISA is designed with the Altera Stratix III family of FPGAs in mind. The architecture of the Stratix III FPGA drove many of the design decisions such as number of vector registers and the supported DSP features.

The instruction set in this ISA borrows heavily from the VIRAM instruction set, which is designed as vector extensions to the MIPS-IV instruction set. A subset of the VIRAM instruction set is adopted, complemented by several new instructions to support new features introduced in this ISA.

Differences of this ISA from the VIRAM ISA are:

- increased number of vector registers,
- different instruction encoding,
- configurable processor parameters,
- sequential memory consistency instead of VP-consistency,
- no barrier instructions to order memory accesses,
- new multiply-accumulate (MAC) units and associated instructions (`vmac`, `vccacc`, `vcczacc`),
- new vector lane local memory and associated instructions (`vldl`, `vstl`),
- new adjacent element shift instruction (`vupshift`),
- new vector absolute difference instruction (`vabsdiff`),
- no support for floating point arithmetic,
- fixed point arithmetic not yet implemented, but defined as a future extension,
- no support for virtual memory or speculative execution.

Table 1: List of configurable processor parameters

Parameter	Description	Typical
NLane	Number of vector lanes	4–128
MVL	Maximum vector length	16–512
VPUW	Processor data width (bits)	8,16,32
MemWidth	Memory interface width (bits)	32, 64, 128
MemMinWidth	Minimum accessible data width in memory	8,16,32
MACL	MAC chain length (0 is no MAC)	0,1,2,4
LMemN	Local memory number of words	0–1024
LMemShare	Shared local memory address space within lane	On/Off
Vmult	Vector lane hardware multiplier	On/Off
Vupshift	Vector adjacent element shifting	On/Off
Vmanip	Vector manipulation instructions (vector insert/extract)	On/Off

1.1 Configurable Architecture

This ISA specifies a set of features for an entire family of soft vector processors with varying performance and resource utilization. The ISA is intended to be implemented by a CPU generator, which would generate an instance of the processor based on a number of user-selectable configuration parameters. An implementation or instance of the architecture is not required to support all features of the specifcation. Table 1 lists the configurable parameters and their descriptions, as well as typical values. These parameters will be referred to throughout the specification.

NLane and *MVL* are the the primary determinants of performance of the processor. They control the number of parallel vector lanes and functional units that are available in the processor, and the maximum length of vectors that can be stored in the vector register file. *MVL* will generally be a multiple of *NLane*. The minimum vector length should be at least 16. *VPUW* and *MemMinWidth* control the width of the VPs and the minimum data width that can be accessed by vector memory instructions. These two parameters have a significant impact on the resource utilization of the processor. The remaining parameters are used to enable or disable optional features of the processor.

1.2 Memory Consistency

The memory consistency model used in this processor is sequential consistency. Order of vector and scalar memory instructions is preserved according to program order. There is no guarantee of ordering between VPs during a vector indexed store, unless an ordered indexed store instruction is used, in which case the VPs access memory in order starting from the lowest vector element.

2 Vector Register Set

The following sections describe the register states in the soft vector processor. Control registers and distributed accumulators will also be described.

2.1 Vector Registers

The architecture defines 64 vector registers directly addressable from the instruction opcode. Vector register zero (`vr0`) is fixed at 0 for all elements.

2.2 Vector Scalar Registers

Vector scalar registers are located in the scalar core of the vector processor. As this architecture targets a Nios II scalar core, the scalar registers are defined by the Nios II ISA. The ISA defines thirty-two 32-bit scalar registers. Vector-scalar instructions and certain memory operations require a vector register and a scalar register operand. Vector scalar register values can also be transferred to and from vector registers or vector control registers using the `vext.vs`, `vins.vs`, `vmstc`, `vmcts` instructions.

Table 2: List of vector flag registers

Hardware Name	Software Name	Contents
$vf0	vfmask0	Primary mask; set to 1 to disable VP operation
$vf1	vfmask1	Secondary mask; set to 1 to disable VP operation
$vf2	vfgr0	General purpose
...	...	...
$vf15	vfgr13	General purpose
$vf16		Integer overflow
$vf17		Fixed point saturate
$vf18		*Unused*
...	...	
$vf29		*Unused*
$vf30	vfzero	All zeros
$vf31	vfone	All ones

2.3 Vector Flag Registers

The architecture defines 32 vector flag registers. The flag registers are written to by comparison instructions and are operated on by flag logical instructions. Almost all instructions in the instruction set support conditional execution using one of two vector masks, specified by a mask bit in most instruction opcodes. The vector masks are stored in the first two vector flag registers. Writing a value of 1 into a VP's mask register will cause the VP to be disabled for operations that specify the mask register. Table 2 shows a complete list of flag registers.

2.4 Vector Control Registers

Table 3 lists the vector control registers in the soft vector processor. The registers in italics hold a static value that is initialized at compile time, and is determined by the configuration parameters of the specific instance of the architecture.

The `vindex` control register holds the vector element index that controls the operation of vector insert and extract instructions. The register is writeable. For vector-scalar insert/extract, `vindex` specifies which data element within the vector register will be written to/read from by the scalar core. For vector-vector insert/extract, `vindex` specifies the index of the starting data element for the vector insert/extract operation.

Table 3: List of control registers

Hardware Name	Software Name	Description
$vc0	VL	Vector length
$vc1	*VPUW*	Virtual processor width
$vc2	vindex	Element index for insert (vins) and extract (vext)
$vc3	vshamt	Fixed point shift amount
...	...	...
$vc28	*ACCncopy*	Number of `vccacc/vcczacc` to sum reduce MVL vector
$vc29	*NLane*	Number of vector lanes
$vc30	*MVL*	Maximum vector length
$vc31	*logMVL*	Base 2 logarithm of MVL
$vc32	vstride0	Stride register 0
...	...	...
$vc39	vstride7	Stride register 7
$vc40	vinc0	Auto-increment Register 0
...	...	...
$vc47	vinc7	Auto-increment Register 7
$vc48	vbase0	Base register 0
...	...	...
$vc63	vbase15	Base register 15

The `ACCncopy` control register specifies how many times the copy-from-accumulator instructions (`vccacc`, `vcczacc`) needs to be executed to sum-reduce an entire MVL vector. If the value is not one, multiple multiply-accumulate and copy-from-accumulator instructions will be needed to reduce a MVL vector. Its usage will be discussed in more detail in Section 2.5.

2.5 Multiply-Accumulators for Vector Sum Reduction

The architecture defines distributed MAC units for multiplying and sum reducing vectors. The MAC units are distributed across the vector lanes, and the number of MAC units can vary across implementations. The `vmac` instruction multiplies two inputs and accumulates the result into accumulators within the MAC units. The `vcczacc` instruction sum reduces the MAC unit accumulator contents, copies the final result to element zero of a vector register, and zeros the accumulators. Together, the two instructions `vmac` and `vcczacc` perform a multiply and sum reduce operation. Multiple vectors can be accumulated and sum reduced by executing `vmac` multiple times. Since the MAC units sum multiplication products internally, they cannot be used for purposes other than multiply-accumulate-sum reduce operations.

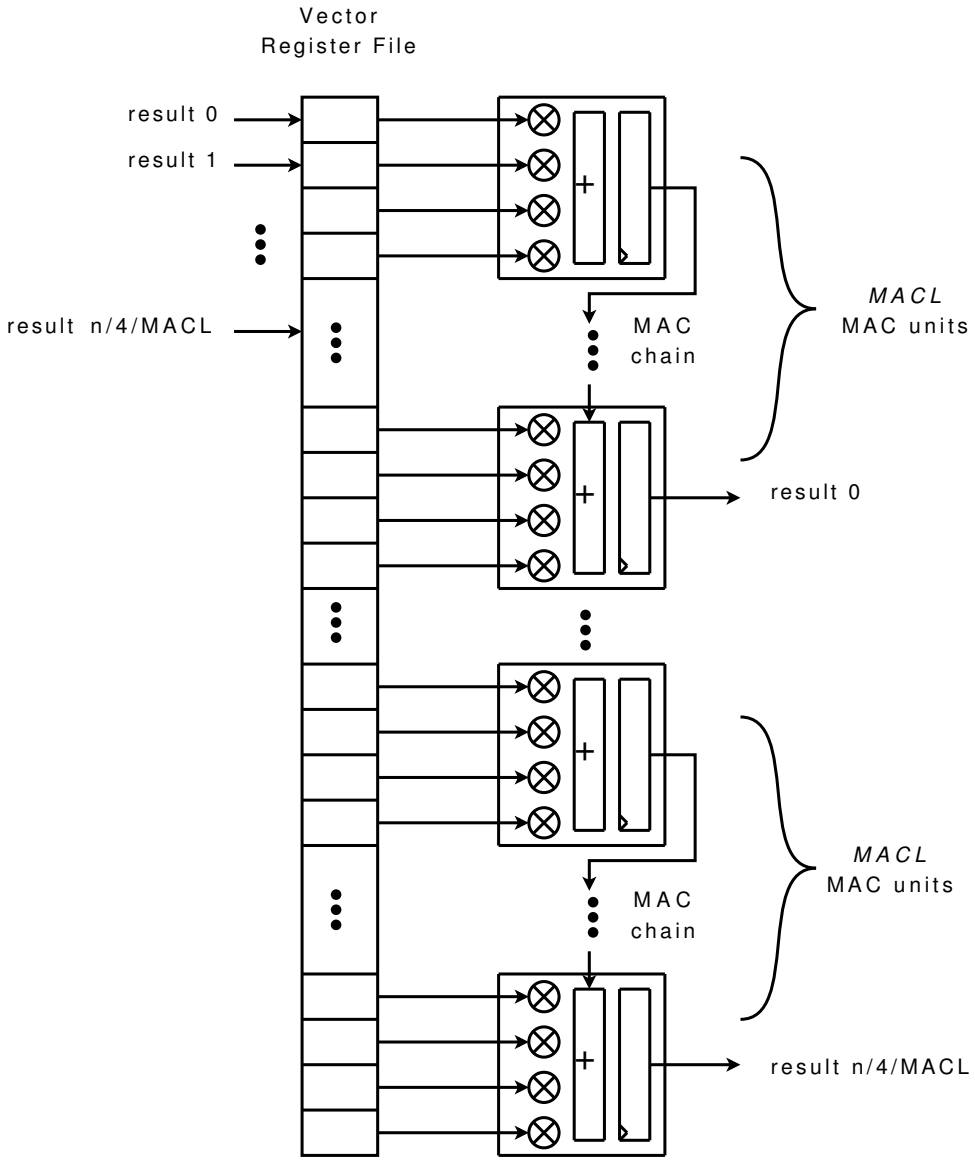


Figure 1: Connection between distributed MAC units and the vector register file

Depending on the number of vector lanes, the `vcczacc` instruction may not be able to sum reduce all MAC unit accumulator contents. In such cases it will instead copy a partially sum-reduced result vector to the destination register. Figure 1 shows how the MAC units generate a result vector and how the result vector is written to the vector register file. The MAC chain length is specified by the *MACL* parameter. The `vcczacc` instruction sets VL to the length of the partial result vector as a side effect, so the partial result vector can be again sum-reduced using the `vmac, vcczacc` sequence. The `ACCncopy` control register specifies how

many times `vcczacc` needs to be executed (including the first) to reduce the entire MVL vector to a single result in the destination register.

2.6 Vector Lane Local Memory

The soft vector architecture supports a vector lane local memory. The local memory is partitioned into private sections for each VP if the *LMemShare* option is off. Turning the option on allows the local memory block to be shared between all VPs in a vector lane. This mode is useful if all VPs need to access the same lookup table data, and allows for a larger table due to shared storage. With *LMemShare*, the *VL* for a local memory write must be less than or equal to *NLane* to ensure VPs do not overwrite each other's data.

The address and data width of the local memory is *VPUW*, and the number of words in the memory is given by *LMemN*. The local memory is addressed in units of *VPUW* wide words. Data to be written into the local memory can be taken from a vector register, or the value from a scalar register can be broadcast to all local memories. A scalar broadcast writes a data value from a scalar register to the VP local memory at an address given by a vector register. This facilitates filling the VP local memory with fixed lookup tables computed by the scalar unit.

3 Instruction Set

The following sections describe in detail the instruction set of the soft vector processor, and different variations of the vector instructions.

3.1 Data Types

The data widths supported by the processor are 32-bit words, 16-bit halfwords, and 8-bit bytes, and both signed and unsigned data tyes. However, not all operations are supported for 32-bit words. Most notably,

32-bit multiply-accumulate is absent.

3.2 Addressing Modes

The instruction set supports three vector addressing modes:

1. Unit stride access
2. Constant stride access
3. Indexed offsets access

The vector lane local memory uses register addressing with no offset.

3.3 Flag Register Use

Almost all instructions can specify one of two vector mask registers in the opcode to use as an execution mask. By default, `vfmask0` is used as the vector mask. Writing a value of 1 into the mask register will cause that VP to be disabled for operations that use the mask. Some instructions, such as flag logical operations, are not masked.

3.4 Instructions

The instruction set includes the following categories of instructions:

1. Vector Integer Arithmetic Instructions
2. Vector Logical Instructions
3. Vector Fixed-Point Arithmetic Instructions
4. Vector Flag Processing Instructions
5. Vector Processing Intructions
6. Memory Instructions

4 Instruction Set Reference

The complete instruction set is listed in the following sections, separated by instruction type. Table 4 describes the possible qualifiers in the assembly mnemonic of each instruction.

Table 4: Instruction qualifiers

Qualifier	Meaning	Notes
op.vv *op*.vs *op*.sv	Vector-vector Vector-scalar Scalar-vector	Vector arithmetic instructions may take one source operand from a scalar register. A vector-vector operation takes two-vector source operands; a vector-scalar operation takes its second operand from the scalar register file; a scalar-vector operation takes its first operand from the scalar register file. The .sv instruction type is provided to support non-commutative operations.
op.b *op*.h *op*.w	1B Byte 2B Halfword 4B Word	The saturate instruction, and all vector memory instructions need to specify the width of integer data.
op.1	Use vfmask1 as the mask	By default, the vector mask is taken from vfmask0. This qualifier selects vfmask1 as the vector mask.

In the following tables, instructions in italics are not yet implemented.

4.1 Integer Instructions

Name	*Mnemonic*	*Syntax*	*Summary*
Absolute Value	`vabs`	`.vv[.1] vD, vA`	Each unmasked VP writes into vD the absolute value of vA.
Absolute Difference	`vabsdiff`	`.vv[.1] vD, vA, vB` `.vs[.1] vD, vA, rS`	Each unmasked VP writes into vD the absolute difference of vA and vB/rS.
Add	`vadd` `vaddu`	`.vv[.1] vD, vA, vB` `.vs[.1] vD, vA, rS`	Each unmasked VP writes into vD the signed/unsigned integer sum of vA and vB/rS.

Subtract	`vsub`
	`vsubu`
		`.vv[.1] vD, vA, vB`
		`.vs[.1] vD, vA, rS`
		`.sv[.1] vD, rS, vB`
			Each unmasked VP writes into vD the signed/unsigned integer result of vA/rS minus vB/rS.

Multiply Hi	`vmulhi` `vmulhiu`	`.vv[.1] vD, vA, vB` `.vs[.1] vD, vA, rS`	Each unmasked VP multiplies vA and vB/rS and stores the upper half of the signed/unsigned product into vD.

Name	*Mnemonic*	*Syntax*	*Summary*
Multiply Low	`vmullo` `vmullou`	`.vv[.1] vD, vA, vB` `.vs[.1] vD, vA, rS`	Each unmasked VP multiplies vA and vB/rS and stores the lower half of the signed/unsigned product into vD.

Integer Divide	*vdiv*
	vdivu
		`.vv[.1] vD, vA, vB`
		`.vs[.1] vD, vA, rS`
		`.sv[.1] vD, rS, vB`
			Each unmasked VP writes into vD the signed/unsigned result of vA/rS divided by vB/rS, where at least one source is a vector.

Shift Right Arithmetic	`vsra`	`.vv[.1] vD, vA, vB`
		`.vs[.1] vD, vA, rS`
		`.sv[.1] vD, rS, vB`
			Each unmasked VP writes into vD the result of arithmetic right shifting vB/rS by the number of bits specified in vA/rS, where at least one source is a vector.

Minimum	`vmin` `vminu`	`.vv[.1] vD, vA, vB` `.vs[.1] vD, vA, rS`	Each unmasked VP writes into vD the minimum of vA and vB/rS.
Maximum	`vmax` `vmaxu`	`.vv[.1] vD, vA, vB` `.vs[.1] vD, vA, rS`	Each unmasked VP writes into vD the maximum of vA and vB/rS.
Compare Equal, Compare Not Equal	`vcmpe` `vcmpne`	`.vv[.1] vF, vA, vB` `.vs[.1] vF, vA, rS`	Each unmasked VP writes into vF the boolean result of comparing vA and vB/rS

Compare Less Than	`vcmplt`
	`vcmpltu`
		`.vv[.1] vF, vA, vB`
		`.vs[.1] vF, vA, rS`
		`.sv[.1] vF, rS, vB`
			Each unmasked VP writes into vF the boolean result of whether vA/rS is less than vB/rS, where at least one source is a vector.

Compare Less Than or Equal	`vcmple`
	`vcmpleu`
		`.vv[.1] vF, vA, vB`
		`.vs[.1] vF, vA, rS`
		`.sv[.1] vF, rS, vB`
			Each unmasked VP writes into vF the boolean result of whether vA/rS is less than or equal to vB/rS, where at least one source is a vector.

Multiply Accumulate	`vmac` `vmacu`	`.vv[.1] vA, vB` `.vs[.1] vD, vA, rS`	Each unmasked VP calculates the product of vA and vB/rS. The products of vector elements are summed, and the summation results are added to the distributed accumulators.

Name	*Mnemonic*	*Syntax*	*Summary*
Compress Copy from Accumulator	`vccacc`	`vD`	The contents of the distributed accumulators are reduced, and the result written into vD. Only the bottom *VPUW* bits of the result are written into vD. If the number of accumulators is greater than *MACL*, multiple partial results will be generated by the accumulate chain, and they are compressed such that the partial results form a contiguous vector in vD. If the number of accumulators is less than or equal to *MACL*, a single result is written into element zero of vD. This instruction is not masked and the elements of vD beyond the partial result vector length are not modified. Additionally, VL is set to the number of elements in the partial result vector as a side effect.
Compress Copy and Zero Accumulator	`vcczacc`	`vD`	The operation is identical to `vccacc`, except the distributed accumulators are zeroed as a side effect.

4.2 Logical Instructions

Name	Mnemonic	Syntax	Summary
And	`vand`	`.vv[.1] vD, vA, vB` `.vs[.1] vD, vA, rS`	Each unmasked VP writes into vD the logical AND of vA and vB/rS.
Or	`vor`	`.vv[.1] vD, vA, vB` `.vs[.1] vD, vA, rS`	Each unmasked VP writes into vD the logical OR of vA and vB/rS.
Xor	`vxor`	`.vv[.1] vD, vA, vB` `.vs[.1] vD, vA, rS`	Each unmasked VP writes into vD the logical XOR of vA and vB/rS.

Shift Left Logical	`vsll`	`.vv[.1] vD, vA, vB`
		`.vs[.1] vD, vA, rS`
		`.sv[.1] vD, rS, vB`
			Each unmasked VP writes into vD the result of logical left shifting vB/rS by the number of bits specified in vA/rS, where at least one source is a vector.

Shift Right Logical	`vsrl`	`.vv[.1] vD, vA, vB`
		`.vs[.1] vD, vA, rS`
		`.sv[.1] vD, rS, vB`
			Each unmasked VP writes into vD the result of logical right shifting vB/rS by the number of bits specified in vA/rS, where at least one source is a vector.

Rotate Right	`vrot`	`.vv[.1] vD, vA, vB`
		`.vs[.1] vD, vA, rS`
		`.sv[.1] vD, rS, vB`
			Each unmasked VP writes into vD the result of rotating vA/rS right by the number of bits specified in vB/rS, where at least one source is a vector.

4.3 Fixed Point Instructions (Future Extension)

Name	*Mnemonic*	*Syntax*	*Summary*
Saturate	*vsat* *vsatu*	{ .b, .h, .w } [.1] vD, vA	Each unmasked VP places into vD the result of saturating vA to a signed/unsigned integer narrower than the VP width. The result is sign/zero-extended to the VP width.
Saturate Signed to Unsigned	*vsatsu*	{ .b, .h, .w } [.1] vD, vA	Each unmasked VP places into vD the result of saturating vA from a signed VP width value to an unsigned value that is as wide or narrower than the VP width. The result is zero-extended to the VP width.
Saturating Add	*vsadd* *vsaddu*	.vv[.1] vD, vA, vB .vs[.1] vD, vA, rS	Each unmasked VP writes into vD the signed/unsigned integer sum of vA and vB/rS. The sum saturates to the VP width instead of overflowing.
Saturating Subtract	*vssub* *vssubu*	.vv[.1] vD, vA, vB .vs[.1] vD, vA, rS .sv[.1] vD, rS, vB	Each unmasked VP writes into vD the signed/unsigned integer subtraction of vA/rS and vB/rS, where at least one source is a vector. The difference saturates to the VP width instead of overflowing.
Shift Right and Round	*vsrr* *vsrru*	[.1] vD, vA	Each unmasked VP writes into vD the right arithmetic/logical shift of vD. The result is rounded as per the fixed-point rounding mode. The shift amount is taken from vcvshamt.
Saturating Left Shift	*vsls* *vslsu*	[.1] vD, vA	Each unmasked VP writes into vD the signed/unsigned saturating left shift of vD. The shift amount is taken from vcshamt.
Multiply High	*vxmulhi* *vxmulhiu*	.vv[.1] vD, vA, vB .vs[.1] vD, vA, rS	Each unmasked VP computes the signed/unsigned integer product of vA and vB/rS, and stores the upper half of the product into vD after arithmetic right shift and fixed-point round. The shift amount is taken from vc_{vshamt}
Multiply Low	*vxmullo* *vxmullou*	.vv[.1] vD, vA, vB .vs[.1] vD, vA, rS	Each unmasked VP computes the signed/unsigned integer product of vA and vB/rS, and stores the lower half of the product into vD after arithmetic right shift and fixed-point round. The shift amount is taken from vc_{vshamt}

Name	*Mnemonic*	*Syntax*	*Summary*
Copy from Accumulator and Saturate	`vxccacc`	`[.1] vD`	The contents of the distributed accumulators are reduced, and the result written into vD. Only the bottom *VPUW* bits of the result are written into vD. If the number of accumulators is greater than *MACL*, multiple partial results will be generated by the accumulate chain, and they are compressed such that the partial results form a contiguous vector in vD. If the number of accumulators is less than or equal to *MACL*, a single result is written into element zero of vD. This instruction is not masked and the elements of vD beyond the partial result vector length are not modified. Additionally, VL is set to the number of elements in the partial result vector as a side effect.
Compress Copy from Accumulator, Saturate and Zero	*vxcczacc*	`vD[.1]`	The operation is identical to `vxccacc`, except the distributed accumulators are zeroed as a side effect.

4.4 Memory Instructions

Name	*Mnemonic*	*Syntax*	*Summary*
Unit Stride Load	`vld` `vldu`	{ `.b` | `.h` | `.w` } `[.1] vD, vbase` `[,vinc]`	The VPs perform a contiguous vector load into vD. The base address is given by the control register vbase, and must be aligned to the width of the data being accessed. The signed increment vinc (default is vinc0) is added to vbase as a side effect. The width of each element in memory is given by the opcode. The loaded value is sign/zero-extended to the VP width.
Unit Stride Store	`vst`	{ `.b` | `.h` | `.w` } `[.1] vA, vbase` `[,vinc]`	The VPs perform a contiguous vector store of vA. The base address is given by vbase (default vbase0), and must be aligned to the width of the data being accessed. The signed increment in vinc (default is vinc0) is added to vbase as a side effect. The width of each element in memory is given by the opcode. The register value is truncated from the VP width to the memory width. The VPs access memory in order.
Constant Stride Load	`vlds` `vldsu`	{ `.b` | `.h` | `.w` } `[.1] vD, vbase,` `vstride [,vinc]`	The VPs perform a strided vector load into vD. The base address is given by vbase (default vbase0), and must be aligned to the width of the data being accessed. The signed stride is given by vstride (default is vstride0). The stride is in terms of elements, not in terms of bytes. The signed increment vinc (default is vinc0) is added to vbase as a side effect. The width of each element in memory is given by the opcode. The loaded value is sign/zero-extended to the VP width.

Name	*Mnemonic*	*Syntax*	*Summary*
Constant Stride Store	`vsts`	`{.b .h .w} [.1] vA, vbase, vstride [,vinc]`	The VPs perform a contiguous store of vA. The base address is given by vbase (default vbase0), and must be aligned to the width of the data being accessed. The signed stride is given by vstride (default is vstride0). The stride is in terms of elements, not in terms of bytes. The signed increment in vinc (default is vinc0) is added to vbase as a side effect. The width of each element in memory is given by the opcode. The register value is truncated from the VP width to the memory width. The VPs access memory in order.
Indexed Load	`vldx` `vldxu`	`{.b .h .w} [.1] vD, vOff, vbase`	The VPs perform an indexed-vector load into vD. The base address is given by vbase (default vbase0), and must be aligned to the width of the data being accessed. The signed offsets are given by vOff and are in units of bytes, not in units of elements. The effective addresses must be aligned to the width of the data in memory. The width of each element in memory is given by the opcode. The loaded value is sign/zero-extended to the VP width.
Unordered Indexed Store	*vstxu*	`{.b .h .w} [.1] vA, vOff vbase`	The VPs perform an indexed-vector store of vA. The base address is given by vbase (default vbase0). The signed offsets are given by vOff. The offsets are in units of bytes, not in units of elements. The effective addresses must be aligned to the width of the data being accessed. The register value is truncated from the VP width to the memory width. The stores may be performed in any order.
Ordered Indexed Store	`vstx`	`{.b .h .w} [.1] vA, vOff vbase`	Operation is identical to vstxu, except that the VPs access memory in order.

Name	*Mnemonic*	*Syntax*	*Summary*
Local Memory Load	`vldl`	`.vv[.1] vD, vA`	Each unmasked VP performs a register-indirect load into vD from the vector lane local memory. The address is specified in vA/rS, and is in units of *VPUW*. The data width is the same as VP width.
Local Memory Store	`vstl`	`.vv[.1] vA, vB` `.vs[.1] vA, rS`	Each unmasked VP performs a register-indirect store of vB/rS into the local memory. The address is specified in vA, and is in units of *VPUW*. The data width is the same as VP width. If the scalar operand width is larger than the local memory width, the upper bits are discarded.
Flag Load	`vfld`	`vF, vbase [,vinc]`	The VPs perform a contiguous vector flag load into vF. The base address is given by vbase, and must be aligned to *VPUW*. The bytes are loaded in little-endian order. This instruction is not masked.
Flag Store	`vfst`	`vF, vbase [,vinc]`	The VPs perform a contiguous vector flag store of vF. The base address is given by vbase, and must be aligned to *VPUW*. A multiple of *VPUW* bits are written regardless of vector length (or more precisely, $\lceil (VL/VPUW) * VPUW \rceil$ flag bits are written). The bytes are stored in little-endian order. This instruction is not masked.

4.5 Vector Processing Instructions

Name	Mnemonic	Syntax	Summary

Merge	`vmerge`	`.vv[.1] vD, vA, vB`
		`.vs[.1] vD, vA, rS`
		`.sv[.1] vD, rS, vB`
			Each VP copies into vD either vA/rS if the mask is 0, or vB/rS if the mask is 1. At least one source is a vector. Scalar sources are truncated to the VP width.

Vector Insert	`vins`	`.vv vD, vA`	The leading portion of vA is inserted into vD. vD must be different from vA. Leading and trailing entries of vD are not touched. The lower vc_{logmvl} bits of vector control register vc_{vindex} specifies the starting position in vD. The vector length specifies the number of elements to transfer. This instruction is not masked.

Vector Extract	`vext`	`.vv vD, vA`	A portion of vA is extracted to the front of vD. vD must be different from vA. Trailing entries of vD are not touched. The lower vc_{logmvl} bits of vector control register vc_{vindex} specifies the starting position in vD. The vector length specifies the number of elements to transfer. This instruction is not masked.
Scalar Insert	`vins`	`.vs vD, rS`	The contents of rS are written into vD at position vc_{vindex}. The lower vc_{logmvl} bits of vc_{vindex} are used. This instruction is not masked and does not use vector length.
Scalar Extract	`vext` `vextu`	`.vs rS, vA`	Element vc_{vindex} of vA is written into rS. The lower vc_{logmvl} bits of vc_{index} are used to determine the element in vA to be extracted. The value is sign/zero-extended. This instruction is not masked and does not use vector length.
Compress	*vcomp*	`[.1] vD, vA`	All unmasked elements of vA are concatenated to form a vector whose length is the population count of the ask (subject to vector length). The result is placed at the front of vD, leaving trailing elements untouched. vD must be different from vA.
Expand	*vexpand*	`[.1] vD, vA`	The first n elements of vA are written into the unmasked positions of vD, where n is the population count of the mask (subject to vector length). Masked positions in vD are not touched. vD must be different from vA.
Vector Element Shift	`vupshift`	`vD, vA`	The contents of vA are shifted up by one element, and the result is written to vD (vD[i] = vA[i+1]). The first element in vA is wrapped to the last element (MVL-1) in vD. This instruction is not masked and does not use vector length.

4.6 Vector Flag Processing Instructions

Name	Mnemonic	Syntax	Summary
Scalar Flag Insert	`vfins`	`.vs vF, rS`	The boolean value of rS is written into vF at position vc_{vindex}. The lower vc_{logmvl} bits of vc_{vindex} are used. This instruction is not masked and does not use vector length.
And	`vfand`	`.vv vFD, vFA, vFB` `.vs vFD, vFA, rS`	Each VP writes into vFD the logical AND of vFA and vFB/rS. This instruction is not masked, but is subject to vector length.
Or	`vfor`	`.vv vFD, vFA, vFB` `.vs vFD, vFA, rS`	Each VP writes into vFD the logical OR of vFA and vFB/rS. This instruction is not masked, but is subject to vector length.
Xor	`vfxor`	`.vv vFD, vFA, vFB` `.vs vFD, vFA, rS`	Each VP writes into vFD the logical XOR of vFA and vFB/rS. This instruction is not masked, but is subject to vector length.
Nor	`vfnor`	`.vv vFD, vFA, vFB` `.vs vFD, vFA, rS`	Each VP writes into vFD the logical NOR of vFA and vFB/rS. This instruction is not masked, but is subject to vector length.
Clear	`vfclr`	`vFD`	Each VP writes zero into vFD. This instruction is not masked, but is subject to vector length.
Set	`vfset`	`vFD`	Each VP writes one into vFD. This instruction is not masked, but is subject to vector length.
Population Count	*vfpop*	`rS, vF`	The population count of vF is placed in rS. This instruction is not masked.
Find First One	*vfff1*	`rS, vF`	The location of the first set bit of vF is placed in rS. This instruction is not masked. If there is no set bit in vF, then the vector length is placed in rS.
Find Last One	*vffl1*	`rS, vF`	The location of the last set bit of vF is placed in rS. The instruction is not masked. If there is no set bit in vF, then the vector length is placed in rS.
Set Before First One	*vfsetbf*	`vFD, vFA`	Register vFD is filled with ones up to and not including the first set bit in vFA. Remaining positions in vF are cleared. If vFA contains no set bits, vFD is set to all ones. This instruction is not masked.

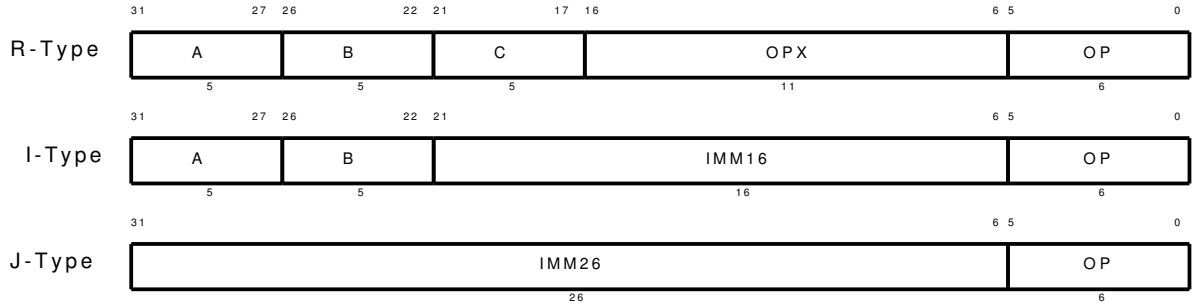
Set Including First One	*vfsetif*	`vFD, vFA`	Register vFD is filled with ones up to and including the first set bit in vFA. Remaining positions in vF are cleared. If vFA contains no set bits, vFD is set to all ones. This instruction is not masked.
Set Only First One	*vfsetof*	`vFD, vFA`	Register vFD is filled with zeros except for the position of the first set bit in vFA. If vFA contains no set bits, vFD is set to all zeros. This instruction is not masked.

4.7 Miscellaneous Instructions

Name	Mnemonic	Syntax	Summary
Move Scalar to Control	`vmstc`	`vc, rS`	Register rS is copied to vc. Writing vc_{vpw} changes vc_{mvl}, vc_{logmvl} as a side effect.
Move Control to Scalar	`vmcts`	`rS, vc`	Register vc is copied to rS.

5 Instruction Formats

The Nios II ISA uses three instruction formats.



The defined vector extension uses up to three 6-bit opcodes from the unused/reserved Nios II opcode space. Each opcode is further divided into two vector instruction types using the OPX bit in the vector instruction opcode. Table 11 lists the Nios II opcodes used by the soft vector processor instructions.

Table 11: Nios II Opcode Usage

Nios II Opcode	OPX Bit	Vector Instruction Type
0x3D	0	Vector register instructions
	1	Vector scalar instructions
0x3E	0	Fixed point instructions
	1	Vector flag, transfer, misc
0x3F	0	Vector memory instructions
	1	Unused except for `vstl.vs`

5.1 Vector Register and Vector Scalar Instructions

The vector register format (VR-type) covers most vector arithmetic, logical, and vector processing instructions. It specifies three vector registers, a 1-bit mask select, and a 7-bit vector opcode. Instructions that take only one source operand use the vA field. Two exceptions are the vector local memory load and store instructions, which also use VR-type instruction format.

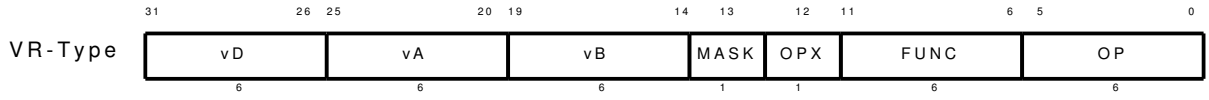


Table 12: Scalar register usage as source or destination register

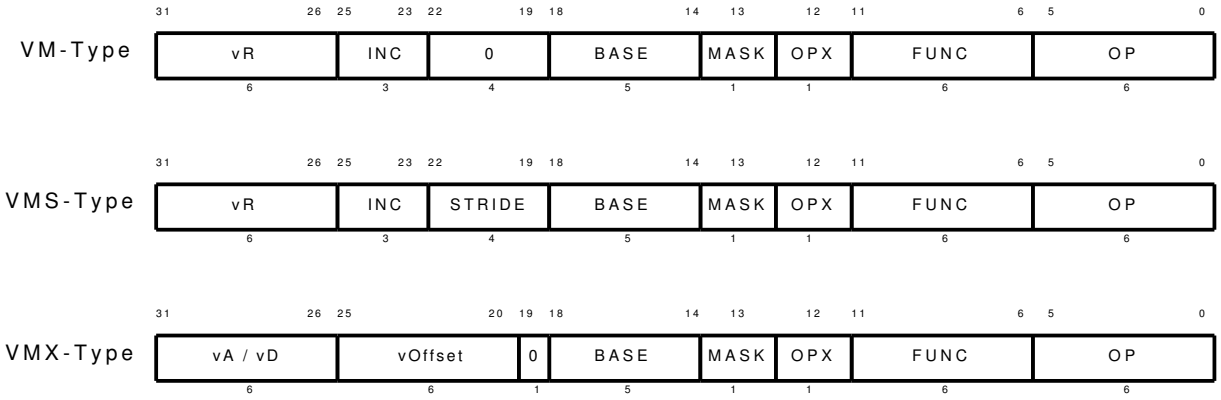
Instruction	Scalar register usage
op.vs	Source
op.sv	Source
vins.vs	Source
vext.vs	Destination
vmstc	Source
vmcts	Destination

Scalar-vector instructions that take one scalar register operand have two formats, depending on whether the scalar register is the source (SS-Type) or destination (SD-Type) of the operation.

Table 12 lists which instructions use scalar register as a source and as a destination.

5.2 Vector Memory Instructions

Separate vector memory instructions exist for the different addressing modes. Each of unit stride, constant stride, and indexed memory access has its own instruction format: VM, VMS, and VMX-type, respectively.



Scalar store to vector lane local memory uses the SS-type instruction format with all zeros in the vD field.

Vector load and store to the local memory use the VR-type instruction format.

5.3 Instruction Encoding

5.3.1 Arithmetic/Logic Instructions

Table 13 lists the function field encodings for vector register instructions. Table 14 lists the function field encodings for scalar-vector and vector-scalar (non-commutative vector-scalar operations). These instructions use the vector-scalar instruction format.

Table 13: Vector register instruction function field encoding (OPX=0)

	[2:0] Function bit encoding for .vv							
[5:3]	000	001	010	011	100	101	110	111
000	vadd	vsub		vmac	vand		vor	vxor
001	vaddu	vsubu		vmacu		vabsdiff		
010	vsra	vcmpeq	vsll	vsrl	vrot	vcmplt	*vdiv*	vcmple
011	vmerge	vcmpneq				vcmpltu	*vdivu*	vcmpleu
100		vmax	vext	vins		vmin	vmulhi	vmullo
101		vmaxu				vminu	vmulhiu	vmullou
110	vccacc	vupshift	*vcomp*	*vexpand*		vabs		
111	vczacc							

Table 14: Scalar-vector instruction function field encoding (OPX=1)

	[2:0] Function bit encoding for .vs							
[5:3]	000	001	010	011	100	101	110	111
000	vadd	vsub		vmac	vand		vor	vxor
001	vaddu	vsubu		vmacu		vabsdiff		
010	vsra	vcmpeq	vsll	vsrl	vrot	vcmplt	*vdiv*	vcmple
011	vmerge	vcmpneq				vcmpltu	*vdivu*	vcmpleu
100		vmax	vext	vins		vmin	vmulhi	vmullo
101		vmaxu	vextu			vminu	vmulhiu	vmullou
	[2:0] Function bit encoding for .sv							
[5:3]	000	001	010	011	100	101	110	111
110	vsra	vsub	vsll	vsrl	vrot	vcmplt	*vdiv*	vcmple
111	vmerge	vsubu				vcmpltu	*vdivu*	vcmpleu

5.3.2 Fixed Point Instructions (Future extension)

Table 15 lists the function field encodings for fixed point arithmetic instructions. These instructions are provided as a specification for future fixed point arithmetic extension.

Table 15: Fixed point instruction function field encoding (OPX=0)

	[2:0] Function bit encoding for fixed-point instructions							
[5:3]	000	001	010	011	100	101	110	111
000	*vsadd*	*vssub*		*vsat*	*vsrr*	*vsls*	*vxmulhi*	*vxmullo*
001	*vsaddu*	*vssubu*		*vsatu*	*vsrru*	*vslsu*	*vxmulhiu*	*vxmullou*
010	*vxccacc*			*vsatsu*				
011	*vxcczacc*							
100	*vsadd.sv*	*vssub.sv*					*vxmulhi.sv*	*vxmullo.sv*
101	*vsaddu.sv*	*vssubu.sv*					*vxmulhiu.sv*	*vxmullou.sv*
110		*vssub.vs*						
111		*vssubu.vs*						

5.3.3 Flag and Miscellaneous Instructions

Table 16 lists the function field encoding for vector flag logic and miscellaneous instructions.

Table 16: Flag and miscellaneous instruction function field encoding (OPX=1)

	[2:0] Function bit encoding for flag/misc instructions							
[5:3]	000	001	010	011	100	101	110	111
000	vfclr	vfset			vfand	vfnor	vfor	vfxor
001	*vffff1*	*vffl1*						
010	*vfsetof*	*vfsetbf*	*vfsetif*					
011			vfins.vs		vfand.vs	vfnor.vs	vfor.vs	vfxor.vs
100								
101	vmstc	vmcts						
110								
111								

5.3.4 Memory Instructions

Table 17 lists the function field encoding for vector memory instructions. The vector-scalar instruction `vstl.vs` is the only instruction that has opcode of 0x3F and OPX bit of 1.

Table 17: Memory instruction function field encoding

	[2:0] Function bit encoding for memory instructions (OPX=0)							
[5:3]	000	001	010	011	100	101	110	111
000	vld.b	vst.b	vlds.b	vsts.b	vldx.b	vstxu.b	vstx.b	
001	vldu.b		vldsu.b		vldxu.b			
010	vld.h	vst.h	vlds.h	vsts.h	vldx.h	vstxu.h	vstx.h	
011	vldu.h		vldsu.h		vldxu.h			
100	vld.w	vst.w	vlds.w	vsts.w	vldx.w	vstxu.w	vstx.w	
101								
110	vldl	vstl	vfld	vfst				
111								
	[2:0] Function bit encoding for memory instructions (OPX=1)							
[5:3]	000	001	010	011	100	101	110	111
110		vstl.vs						