MIPS Vector Architecture Manual
Revision : 1.3

Corinna G. Lee
Department of Electrical and Computer Engineering
University of Toronto

June 17, 1997

Contents

List of Figures

1 MIPS Instruction Formats

	31 26	25 21	20 16	15 11	10 6	5 0
J-type	OPCODE	TARGET				
I-type	OPCODE	RS	RT	IMMEDIATE		
R-type	OPCODE	RS	RT	RD	SHAMT	FUNCTION

Cop1(FPA) F-type	010001	1	FORMAT	FT	FS	FD	FUNCTION

	31 26	25 21		20 16	15 11	10 6	5 0
Cop2(VP) T-type	010010	0	FORMAT	RT/FT	RS	VD/0	000000
				oth reg	VP reg	VP reg	
Cop2(VP) M-type	010010	0	1100	VT	RS/VS/0	RD	FUNCTION
				data	stride	base	
Cop2(VP) V-type	010010	1	FORMAT	RT/FT/VT/0	RS/FS/VS	VD/0	FUNCTION
				source	source	dest	

2 Overview of Vector Processor (VP)

The vector processor operates as a coprocessor unit of the MIPS instruction set architecture and extends the ISA to execute memory and computational instructions in vector mode. The computational instructions include integer arithmetic, logical, shift, and floating-point instructions.

Similar to the FPA (floating-point accelerator), the VP continually monitors the instruction stream. If an instruction is a coprocessor 2 instruction (opcode=010010), then the VP executes the instruction; otherwise, the VP ignores it. Because all vector loads and stores are defined within the coprocessor 2 instruction set, the VP ignores any `lwc2` and `swc2` instructions.

VP Features

32-bit and 64-bit Operation. The VP contains 32 32-element, 64-bit vector registers. A vector register can hold integer, IEEE single-precision, or IEEE double-precision values. Integer and single-precision values are 32 bits while double-precision values are 64 bits.

Control Registers. There are three 32-bit control registers for specifying the length of a vector operation (`vlr`), storing the results of a vector compare instruction (`vcond`), and providing access to floating-point exception handling capabilities (`vcsr`).

Load/Store Instruction Set. Like the CPU and FPA, the VP uses a load/store instruction set. All vector instructions are *started* in a single cycle but may execute for several cycles. Their execution can be overlapped with other scalar or vector instructions.

Single-Chip Implementation. It is expected that the CPU, FPA, and VP are all integrated on a single chip because vector instructions can use values from the integer and floating-point register files as source operands. In addition, the coprocessor interface is tightly-coupled in a manner similar to the CPU-FPA interface.

2.1 VP Registers

The VP contains two types of registers: general-purpose vector registers and vector control registers.

General-Purpose Vector Registers. There are 32 vector registers: `$vv0` to `$vv31`[1]. Each vector register contains 32 64-bit elements. All elements of `$vv0` are defined to hold the constant zero.

A vector register can hold integer, IEEE single-precision, or IEEE double-precision values. Different vector registers can store different data formats at the same time but all elements of one vector register are considered to have the same data format when the register is accessed.

[1]The symbol `$vv` is used rather than `$v` or `v` to avoid confusion with the symbolic register names `v0` and `v1` which are used for integer registers `$2` and `$3`.

An element of a vector register can be copied to an integer or floating-point register using the `vtint` and `vtfp` instructions, respectively. Conversely, a scalar register can be copied to an element of a vector register using the `vfint` and `vffp` instructions. These instructions are a renamed and expanded version of the `mfc2` and `mtc2` instructions. See Section 2.2.1 for details about these instructions.

Vector Control Registers. Although MIPS coprocessors can have as many as 32 control registers, the VP coprocessor defines, for now, only three:

`vlr` (`vcr2`): The vector length register specifies the number of operations to execute for a vector instruction.

If a vector instruction is issued when `vlr`=0, then no operations are executed. If a vector instruction is issued when `vlr` is greater than an implementation's maximum vector length, a vector-instruction exception is raised.

Reads or writes of `vlr` do not affect vector instructions in progress.

`vcond` (`vcr4`): The vector condition register holds the results of the most recently executed vector compare `flag` instruction (`flt`, `feq`, etc). There is one bit corresponding to each comparison result, with the least significant bit representing the result of operating on vector element zero.

Reads of `vcond` are interlocked and guaranteed to return the most recent value. Write are also interlocked and thus are not affected by previously issued vector instructions that may still be executing.

`vcsr` (`vcr31`): The vector status/control register is used to control and monitor floating-point exceptions and establish rounding modes for floating-point execution. The `vcsr` has the same format and functionality as the FPA's `fcsr` except the the `vcsr` does *not* have a condition (`C`) bit.

The vector control registers can be accessed with the `vctint` instruction (also known as `cfc2`) and the `vcfint` (`ctc2`) instruction.

2.2 Summary of VP Instruction Set

The VP instructions are specified using the coprocessor unit 2 (`cop2`) operation instructions. Like all MIPS instructions, every VP instruction consists of a single word (32 bits) aligned on a word boundary. There is only one instruction format which is used slightly differently for transfer, memory, and computational instructions; hence, three instruction types are defined. Figure 1 shows the VP instruction format and how it is used for these three instruction classes.

Transfer instructions are used to transfer data between VP registers and integer or floating-point registers without having to go through memory. Memory instructions transfer data between the general-purpose vector registers and memory. The effective address is computed using integer registers and possibly a vector register, depending upon the addressing mode used. Computational instructions perform operations, such as arithmetic, logical,

	31 26		25 21	20 16	15 11	10 6	5 0
Cop2(VP) T-type	010010	0	FORMAT	RT/FT	RS	VD/0	000000
				oth reg	VP reg	VP reg	
Cop2(VP) M-type	010010	0	1100	VT	RS/VS/0	RD	FUNCTION
				data	stride	base	
Cop2(VP) V-type	010010	1	FORMAT	RT/FT/VT/0	RS/FS/VS	VD/0	FUNCTION
				source	source	dest	

Field Name	Function
FORMAT	a 4-bit format field specifying the data format (integer, single-precision, or double-precision) and the source type (vector/vector, vector/scalar, or scalar/vector) of the vector instruction
VS,VT,VD	5-bit vector register specifiers
RS,RT,RD	5-bit integer register specifiers
FS,FT	5-bit floating-point register specifiers
FUNCTION	a 6-bit function field that specifies the operation to execute

Figure 1: VP Instruction Format and Instruction Types

Instruction Type	Source Type	Data Format i&u(&$\mathcal{F}$)	s(&$\mathcal{F}$)	d(&$\mathcal{F}$)	TOTAL(&$\mathcal{F}$)
transfer	—	4 = 4	2	2	8
memory	—	18+12 =30	6	6	42
computational	vv	21+ 6 (+ 5)=27(+ 5)	17(+25)	17(+25)	61 (+55)
	vs	20+ 6 (+ 5)=26(+ 5)	13(+ 3)	13(+ 3)	52 (+11)
	sv	7+ 4 (+ 5)=11(+ 5)	10(+ 3)	10(+ 3)	31 (+11)
	TOTAL	70+28(+15)=98(+15)	48(+31)	48(+31)	194(+77)=271

Figure 2: Summary of Number of Vector Instructions

shifts, transfers, etc., on register data and store the results in a vector register. Source operands can come from vector registers or a scalar integer or floating-point register.

Almost every scalar memory and computational instruction has a vector version. A vector instruction has the same functionality as its scalar counterpart with the following exceptions:

- the source operands can be either vector registers or scalar registers;
- the destination operand is a vector register;
- the specified operation is executed `vlr` times (i.e., the amount specified in the vector length register); and
- three addressing modes are defined for vector memory instructions.

In particular, vector floating-point instructions are similar in functionality to scalar floating-point instructions with regards to supporting the IEEE Standard for Binary Floating-Point Arithmetic. Moreover, floating-point data formats and number definitions are the same as those used for the FPA.

In total, there are 271 VP instructions defined: 8 transfer instructions, 42 vector memory instructions, and 221 vector computational instructions. Of these, all the transfer and memory instructions plus 144 of the computational instructions are currently implemented, while the rest will be implemented at a later date. Figure 2 summarizes the number of vector instructions by instruction type, source type and data format. Counts for future vector instructions ($\mathcal{F}$) are given in parentheses. Note that signed (i) and unsigned (u) operations are defined for the integer source type.

The following subsections provide a summary of the transfer instructions as well as the vector memory and vector computational instructions. The VP instructions are classified in this manner because each instruction class uses the single instruction format in a slightly different way. In addition, the following subsection give the assembly form of each instruction class. In the assembly form, vector registers are specified as `$vv`k where k is the register number. Integer registers are written as `$`k or with their special register names (eg., `a0, a1, v0, v1,` etc.), while floating-point registers are given as `$f`k.

Assembly Form	Functionality	Description
`vtint` RT, RS(VD)	RT ← VD[RS]	Copy RS element of vector register VD into integer register RT.
`vfint` RT, RS(VD)	VD[RS] ← RT	Copy integer register RT into RS element of vector register VD.
`vtfp.s` FT, RS(VD)	FT ← VD[RS]<31:0>	Copy least-significant 32 bits of RS element of vector register VD into floating-point register FT.
`vffp.s` FT, RS(VD)	VD[RS]<31:0> ← FT	Copy floating-point register FT into least significant half of RS element of vector register VD.
`vtfp.d` FT, RS(VD)	FT ← VD[RS]<31:0> ; FT+1 ← VD[RS]<63:32>	Copy RS element of vector register VD into double-precision floating-point register FT. FT must be an even number to denote a double-precision register.
`vffp.d` FT, RS(VD)	VD[RS]<31:0> ← FT ; VD[RS]<63:32> ← FT+1	Copy double-precision floating-point register FT into RS element of vector register VD. FT must be an even number to denote a double-precision register.
`vctint` RT, RS	RT ← `vcr`[RS]	Copy vector control register RS into integer register RT. Field VD is set to 0.
`vcfint` RT, RS	`vcr`[RS] ← RT	Copy integer register RT into vector control register RS. Field VD is set to 0.

Figure 3: VP Transfer Instructions

2.2.1 Transfer Instructions

These instructions transfer data between the VP registers and the integer registers on the CPU and the floating-point registers on the FPA (COP1). The instruction format for the transfer instructions is as follows:

	31 26	25	21	20 16	15 11	10 6	5 0
Cop2(VP) T-type	010010	0	FORMAT	RT/FT	RS	VD/0	000000
				oth reg	VP reg	VP reg	

The RS and VD fields specify the VP register to transfer to or from, while the RT/FT field specifies the register on the CPU or FPA.

There are eight transfer instructions in total: two that transfer between the general-purpose vector registers (GPVR) and the integer registers, four for transferring between the GPVRs and the floating-point registers, and two to transfer between the vector control registers and integer registers. Figure 3 gives the assembly form for the eight instructions and describes their functionality.

Addressing Mode	Assembly Form	Effective Address of i^{th} Access
unit stride	FUNCTION VT, (RD)	RD+$i \times s$ where s is the number of bytes for one access (eg., 1 for byte accesses, 2 for half-word accesses, etc.). Field RS is set to 0.
constant stride	FUNCTION VT, RS(RD)	RD+$i \times s$ where s is the contents of integer register RS.
indexed	FUNCTION VT, VS(RD)	RD+VS[i].

Figure 4: Addressing Modes for Vector Memory Instructions

2.2.2 Vector Memory Instructions

Vector memory instructions copy data between memory and vector registers. Seven different data formats can be accessed: byte(b), half-word(h), word(w), unsigned byte(bu), unsigned half-word(hu), single-precision(.s), and double-precision(.d).

The instruction format for vector memory instructions is as follows:

	31 26	25	21	20 16	15 11	10 6	5 0
Cop2(VP) M-type	010010	0	1100	VT	RS/VS/0	RD	FUNCTION
				data	stride	base	

The fields RD and RS/VS are used to compute the effective addresses and specify the base address and stride (as a byte offset), respectively. The VT field specifies the vector data register that is to be loaded into or stored from.

The functionality of an individual access is the same as its scalar counterpart with the exception of how the address is computed. The address of an individual element i ($i = 0 \ldots$`vlr`) is computed by adding a multiple of a *byte* offset or stride value (RS/VS) to the base (RD) register. Three different addressing modes are defined to specify the stride s. Figure 4 describes the three addressing modes and also gives the assembly form for each. The FUNCTION field specifies the opcode mnemonic to use for a particular memory instruction.

The following is a summary of the opcode mnemonics for vector memory instructions:

vector loads with unit stride:	`vlb,vlh,vlw, vlbu,vlhu, vl.s,vl.d`
vector stores with unit stride:	`vsb,vsh,vsw, vsbu,vshu, vs.s,vs.d`
vector loads with constant stride:	`vlbst,vlhst,vlwst, vlbust,vlhust, vlst.s,vlst.d`
vector stores with constant stride:	`vsbst,vshst,vswst, vsbust,vshust, vsst.s,vsst.d`
vector loads with indexed offsets:	`vlbx,vlhx,vlwx, vlbux,vlhux, vlx.s,vlx.d`
vector stores with indexed offsets:	`vsbx,vshx,vswx, vsbux,vshux, vsx.s,vsx.d`

The opcode mnemonics are also specified in the FUNCTION bit encoding table for vector memory instructions (see Figure 15).

2.2.3 Vector Computational Instructions

Vector computational instructions operate on one or two source operands to produce a vector result. Most of the VP instructions fall into this instruction class. The functionality of an individual operation is the same as its scalar counterpart. Four different data formats can be operated on: word, unsigned word(u), single-precision(.s), and double-precision(.d).

In addition, there are three categories of vector computational instructions based on the type of the source operands:

- `.vv`: Both source operands are **vector** registers.
- `.vs`: The first source operand is a **vector** register and the second source operand is a **scalar** register.
- `.sv`: The first source operand is a **scalar** register and the second source operand is a **vector** register.

Almost every scalar instruction defined for the `special` and `cop1`(FPA) opcodes has a `.vv` and a `.vs` version. For instructions with only one source operand, the `.vv` version is used where VS specifies the source operand and VT is set to 0. Non-commutative operations that cannot be synthesized without using an extra register[2] also have a `.sv` version defined.

Although not all `.sv` instructions are *defined* in the ISA, functional support for the undefined ones is provided by the assembler. The assembler will replace an undefined `.sv` instruction with its equivalent `.vs` version if it is a commutative operation or macro-expand it to a sequence of instructions that provides the same functionality. Section 2.2.5 describes the equivalent instructions and macro expansions for the `.sv` pseudo-instructions.

The instruction format for vector computational instructions is as follows:

[2]For example, "`movz.sv` VD, RS,RT" can be synthesized as "`beq` RS,`$0, next ; nop ; add.vv` VD, VT,`$vv0 ; next:` " and hence `movz.sv` is not defined in the VP ISA. On the other hand, because synthesizing "`sub.sv vd, rs,vt`" would require using another vector register ("`expand vtemp,`RS; `sub.vv` VD, `vtemp,`VT"), `sub.sv` is defined.

FORMAT	Assembly Form		Computation of i^{th} Result
`.vv`	FUNCTION`.vv`	VD, VS,VT	VD$[i]$ = VS$[i]$ FUNCTION VT$[i]$
`.vs`	FUNCTION`.vs`	VD, VS,RT	VD$[i]$ = VS$[i]$ FUNCTION RT
`.sv`	FUNCTION`.sv`	VD, RS,VT	VD$[i]$ = RS FUNCTION VT$[i]$
`.{s,d}.vv`	FUNCTION`.s.vv`	VD, VS,VT	VD$[i]$ = VS$[i]$ FUNCTION VT$[i]$
	FUNCTION`.d.vv`	VD, VS,VT	
`.{s,d}.vs`	FUNCTION`.s.vs`	VD, VS,FT	VD$[i]$ = VS$[i]$ FUNCTION FT
	FUNCTION`.d.vs`	VD, VS,FT	
`.{s,d}.sv`	FUNCTION`.s.sv`	VD, FS,VT	VD$[i]$ = FS FUNCTION VT$[i]$
	FUNCTION`.d.sv`	VD, FS,VT	

Figure 5: Vector Computational Instructions

	31 26	25	25 21	20 16	15 11	10 6	5 0
Cop2(VP) V-type	010010	1	FORMAT	RT/FT/VT/0	RS/FS/VS	VD/0	FUNCTION
				source	source	dest	

The source operands are specified by the RS/FS/VS and RT/FT/VT fields. At least one of the source operands must be a vector register while the other one may be a scalar or vector register. The destination operand is always a vector register and is specified by VD. The FORMAT field specifies the source type and data format of the operation to be performed.

Figure 5 summarizes the operation of the different classes of vector instructions and provides the assembly form. The opcode mnemonics are specified in the FUNCTION bit encoding tables for vector integer instructions (see Figure 16) and vector floating-point instructions (see Figure 17).

2.2.4 Special Vector Computational Instructions

Several subclasses of computational instructions function somewhat differently from what is described above. These are the `conditional move` instructions, `flag` instructions, and `extract` and `expand` instructions.

Conditional Move Instructions. These are the vector versions of the scalar `conditional move` instructions defined for the MIPS IV architecture. The vector versions can be used to execute `if`-converted source code.

The basic functionality of the vector `conditional move` is as follows. The RT/FT/VT register is conditionally copied to the VD register only if the corresponding element in the VS register satisfies the specified condition relative to zero. If the condition is not satisfied, the corresponding element in VD is left unchanged. Figure 6 gives the RTL functional description of the `conditional move` instructions.

Figure 6 also lists the vector `conditional move` instructions that are defined. These instructions use the same assembly form as the regular computational instructions (see Fig-

FORMAT	Assembly Form		Computation of i^{th} Result
`.vv`	`mov<cond>z.vv`	VD, VS,VT	if (VS[i] <cond> 0) {VD[i] = VT[i]}
`.vs`	`mov<cond>z.vs`	VD, VS,RT	if (VS[i] <cond> 0) {VD[i] = RT}
`.{s,d}.vs`	`mov<cond>z.{s,d}.vs`	VD, VS,FT	if (VS[i] <cond> 0) {VD[i] = FT}

Data Format	Suffix	Conditional Move Instructions movgtz	movgez	movltz	movlez	movz	movnz
integer	`.{vv,vs}`	X	X	X	X	X	X
floating-point	`.{s,d}.{vv,vs,sv}`	X	X	X	X	X	X

Figure 6: Vector Conditional Move Instructions

ure 5. A total of 48 `conditional move` instructions are defined based on 6 different conditions that can be tested (`gtz`, `gez`, `ltz`, `lez`, `z`, `nz`), 3 data formats (signed, single-precision, double-precision), and 3 source types (vector/vector, vector/scalar, scalar/vector).

Note that there is no scalar/vector version for integer data. This is because these can be synthesized using a branch instruction and an unconditional vector move. Figure 10 in Section 2.2.5 gives the instruction sequences for synthesizing the `.sv` version for integer data. Scalar/vector `conditional moves`, on the other hand, are defined for floating-point data because floating-point comparisons to zero require the use of another register.

Conditional moves are not devined for unsigned data because these either can be tested using a signed version (`z, nz, gtz`), are always true (`gez`), or do not make sense for unsigned data (`ltz, lez`).

Flag Instructions. These instructions are similar to the `set` instructions except the result of each comparison is stored in one bit of the `vcond` vector control register rather than in an element of a general-purpose vector register. It is expected that the `vcond` register would then be used as the mask register when masked vector instructions are included in the ISA at a future date. Multiple vector masks are possible without having to recompute them by transferring masks between the `vcond` register and the integer scalar registers using the `vctint` and `vcfint` instructions.

The `flag` instructions specify two source operands but no destination register since the destination is implicitly the `vcond` register. Hence the encoding for these instructions sets the VD field to 0. Figure 7 gives the RTL functional description of the `flag` instructions.

Figure 7 also lists the vector `flag` instructions that are defined and their assembly form. There are 2 conditions that can be tested (`ge`, `lt`) for all the possible data formats (signed, unsigned, single-precision, double-precision) and source types (`.vv, .vs, .sv`). In addition, integer and floating-point data can be tested for equality (`eq`) for `.vv` and `.vs` operand types. In total, there are 30 vector `flag` instructions.

Missing combinations (eg., equality for unsigned values or equality for `.sv` integers) are provided as pseudo-instructions (see Section 2.2.5).

FORMAT	Assembly Form	Computation of i^{th} Result
.vv	f<cond>.vv VS,VT	vcr[vcond]$< i >$ = VS[i] <cond> VT[i]
.vs	f<cond>.vs VS,RT	vcr[vcond]$< i >$ = VS[i] <cond> RT
.sv	f<cond>.sv RS,VT	vcr[vcond]$< i >$ = RS <cond> VT[i]

Data Format	Suffix	Flag Instructions		
		fge	feq	flt
integer	.{vv,vs}	X	X	X
	.sv	X		X
unsigned	.{vv,vs}	X		X
	.sv	X		X
floating-point	.{s,d}.{vv,vs}	X	X	X
	.sv	X		X

Figure 7: Vector Flag Instructions

Assembly Form		Computation of i^{th} Result
extract.vs	VD, VS,RT	VD[i] = VS[RT + i]
expand.sv	VD, RS	VD[i] = RS
expand.sv.s	VD, FS	VD[i] = FS
expand.sv.d	VD, FS	VD[i]$_{<31:0>}$ = FS ; VD[i]$_{<63:32>}$ = FS+1

Figure 8: Vector Extract and Expand Instructions

Extract and Expand Instructions. These are special vector instructions that have no equivalent scalar version. Figure 8 summarizes the functionality of these instructions.

The `extract` instruction copies `vlr` elements from the VS register beginning at the RTth element into VD. Because this instruction merely copies bits from one vector register to another, there is only one version (i.e., `extract.vs`) of it to handle all the data formats.

The `expand` instructions copy the RS or FS scalar register into the first `vlr` elements of the VD vector register. The VT field is unused and set to 0. Because the source operand could be an integer register, single-precision floating-point register, or double-precision register, there are three versions of the `expand` instruction. For double-precision data, the FS register must be an even number.

2.2.5 Assembler-Supported Pseudo-Instructions

This section describes the pseudo-instructions whose functionality is supported by the assembler. The assembly form of pseudo-instructions is the same as that of defined VP in-

Pseudo-Instruction Type	Pseudo-Instruction	Equivalent Instruction Sequence

no branch in pseudo-instruction	`j FOO`
	`lbai.v $vv1, $12,$14`
		`vlb $vv1, ($12)`
		`j FOO`
		`  addu $12, $12,$14`

with branch in pseudo-instruction	`add $12, $13,$14`
	`j FOO`
	`  movgtz.sv $vv1, $15,$vv2`
		`blez $15, NEXT`
		`  add $12, $13,$14`
		`movz.vv $vv1, $vv0,$vv2`
		`next: j FOO`
		`  nop`

Figure 9: Pseudo-Instructions in Branch Delay Slots

structions and, as far as the assembly-level programmer is concerned, there is little need to distinguish between pseudo- and real instructions. The main difference between the two is that pseudo-instructions are not implemented directly in hardware but can be implemented with an existing VP instruction or sequence of instructions using only the registers specified in the pseudo-instruction. Upon encountering a pseudo-instruction, the assembler replaces it with its equivalent instruction sequence.

If a pseudo-instruction appears in the delay slot of a branch and the pseudo-instruction expands to several instructions, the assembler has two options depending upon if the expansion itself contains a branch. Figure 9 shows examples of the two possibilities. If the expansion does *not* contain a branch, all but the last instruction in the sequence are placed before the branch and the last instruction is placed in the delay slot. For a pseudo-instruction that does include a branch, the assembler places all the instructions before the first branch and attempts to fill the delay slot of the pseudo-instruction's branch.

The pseudo-instructions serve to make the VP instruction set completely orthogonal for the assembly-level programmer. All operations are defined for the 4 data formats (signed integer, unsigned integer, single-precision floating-point, double-precision floating-point) and the three source types (vector/vector, vector/scalar, scalar/vector). The remainder of this section describes how the pseudo-instructions are supported by the assembler.

There are a total of 104 assembler-supported pseudo-instructions. These can be grouped into five categories: commutative instructions, `.sv` conditional moves, conditional moves and flags for unsigned integers, unconditional moves, and Torrent instructions.

Commutative Instructions. The first category of pseudo-instructions is composed of those instructions which are commutative. There are 18 such operations defined in the VP architecture. The assembler implements these instructions by merely switching the scalar and vector operands and then emitting the appropriate `.vs` instruction:

$$\_\_\_\_.\texttt{sv}\ \mathrm{VD,RT,VS} \Longrightarrow \_\_\_\_.\texttt{vs}\ \mathrm{VD,VS,RT}$$

Pseudo-Instruction	Equivalent Instruction Sequence

`movgtz.sv` VD, RS,VT	`blez` RS, `next`
	`nop`
	`movz.vv` VD, `$vv0`,VT
	`next:`

`movgez.sv` VD, RS,VT	`bltz` RS, `next`
	`nop`
	`movz.vv` VD, `$vv0`,VT
	`next:`

`movltz.sv` VD, RS,VT	`bgez` RS, `next`
	`nop`
	`movz.vv` VD, `$vv0`,VT
	`next:`

`movlez.sv` VD, RS,VT	`bgtz` RS, `next`
	`nop`
	`movz.vv` VD, `$vv0`,VT
	`next:`

`movz.sv` VD, RS,VT	`bnez` RS, `next`
	`nop`
	`movz.vv` VD, `$vv0`,VT
	`next:`

`movnz.sv` VD, RS,VT	`beqz` RS, `next`
	`nop`
	`movz.vv` VD, `$vv0`,VT
	`next:`

Figure 10: `.sv` Conditional Move Pseudo-Instructions

commutative integer instructions:	`add.sv`, `addu.sv`, `mul.sv`, `mulu.sv`, `seq.sv`,`feq.sv`, `and.sv`, `or.sv`, `xor.sv`, `nor.sv`
commutative floating-point instructions:	`adds.sv`, `muls.sv`, `seqs.sv`, `feqs.sv`,`addd.sv`, `muld.sv`, `seqd.sv`, `feqd.sv`

`.sv` Conditional Moves. There are 6 conditional move pseudo-instructions: `movgtz.sv`, `movgez.sv`, `movltz.sv`, `movlez.sv`, `movz.sv`, `movnz.sv`. Figure 10 shows how a `.sv` conditional move is expanded into several instructions. In addition, `moveqz` has been defined as an alias for `movz` and `movnez` has been defined as an alias for `movnz`.

Conditional Moves and Flags for Unsigned Integers. Figure 11 shows how all of the conditional moves and flags for unsigned integers can be accomplished with an appropriate

Operation	Pseudo-Instruction		Emitted Instruction	
move > 0	`movgtzu.vv`	VD,VS,VT	`movnz.vv`	VD,VS,VT
	`movgtzu.vs`	VD,VS,RT	`movnz.vs`	VD,VS,RT
	`movgtzu.sv`	VD,RT,VS	`movnz.sv`	VD,RT,VS
move $\neq 0$	`movnzu.vv`	VD,VS,VT	`movnz.vv`	VD,VS,VT
	`movnzu.vs`	VD,VS,RT	`movnz.vs`	VD,VS,RT
	`movnzu.sv`	VD,RT,VS	`movnz.sv`	VD,RT,VS
move $= 0$	`movzu.vv`	VD,VS,VT	`movz.vv`	VD,VS,VT
	`movzu.vs`	VD,VS,RT	`movz.vs`	VD,VS,RT
	`movzu.sv`	VD,RT,VS	`movz.sv`	VD,RT,VS
flag equal	`fequ.vv`	VD,VS,VT	`feq.vv`	VD,VS,VT
	`fequ.vs`	VD,VS,RT	`feq.vs`	VD,VS,RT
	`fequ.sv`	VD,RT,VS	`feq.sv`	VD,RT,VS

Figure 11: Conditional Moves and Flag Instructions for Unsigned Integers

Pseudo-Instruction		Emitted Instruction	
`mov.vv`	VD,VT	`movz.vv`	VD,$vv0,VT
`movu.vv`	VD,VT	`movz.vv`	VD,$vv0,VT
`movs.vv`	VD,VT	`movz.vv`	VD,$vv0,VT
`movd.vv`	VD,VT	`movz.vv`	VD,$vv0,VT

Figure 12: Unconditional Moves

substitution of integer moves and flags.

Unconditional Moves. In addition to the conditional move instructions, four unconditional move instructions have been defined. As shown in Figure 12, all four moves, regardless of the target data-type, use the `integer` conditional move. The floating-point moves are treated as integers because, unlike `$0` and `$vv0`, `$f0` is not defined to be zero: a `movz.vv` VD,`$f0`,VT would not produce the intended result.

Torrent Instructions. The remainder of the pseudo-instructions allow the assembler to interpret Berkeley's Torrent instruction set. For additional information on the Torrent instruction set, please refer to the Torrent Architecture Manual.[3]

[3]Asanovic, Krste, et al. Torrent Architecture Manual. Rev. 2.13. International Computer Science Institute: Berkeley, 1995.

Torrent indexed load and store instructions.

Pseudo-Instruction		Emitted Instruction	
`lbx.v`	VT,RD,VS	`vlbx`	VT,VS(RD)
`lhx.v`	VT,RD,VS	`vlhx`	VT,VS(RD)
`lwx.v`	VT,RD,VS	`vlwx`	VT,VS(RD)
`lbux.v`	VT,RD,VS	`vlbux`	VT,VS(RD)
`lhux.v`	VT,RD,VS	`vlhux`	VT,VS(RD)
`sbx.v`	VT,RD,VS	`vsbx`	VT,VS(RD)
`shx.v`	VT,RD,VS	`vshx`	VT,VS(RD)
`swx.v`	VT,RD,VS	`vswx`	VT,VS(RD)
`sbux.v`	VT,RD,VS	`vsbux`	VT,VS(RD)
`shux.v`	VT,RD,VS	`vshux`	VT,VS(RD)

Torrent strided load and store instructions.

Pseudo-Instruction		Emitted Instruction	
`lbst.v`	VT,RD,RS	`vlbst`	VT,RS(RD)
`lhst.v`	VT,RD,RS	`vlhst`	VT,RS(RD)
`lwst.v`	VT,RD,RS	`vlwst`	VT,RS(RD)
`lbust.v`	VT,RD,RS	`vlbust`	VT,RS(RD)
`lhust.v`	VT,RD,RS	`vlhust`	VT,RS(RD)
`sbst.v`	VT,RD,RS	`vsbst`	VT,RS(RD)
`shst.v`	VT,RD,RS	`vshst`	VT,RS(RD)
`swst.v`	VT,RD,RS	`vswst`	VT,RS(RD)
`sbust.v`	VT,RD,RS	`vsbust`	VT,RS(RD)
`shust.v`	VT,RD,RS	`vshust`	VT,RS(RD)

Torrent auto-increment load and store instructions.

Pseudo-Instruction		Emitted Instruction	
`lbai.v`	VT,RD,RT	`vlb` `addu`	VT,(RD) RD,RD,RT
`lhai.v`	VT,RD,RT	`vlh` `addu`	VT,(RD) RD,RD,RT
`lwai.v`	VT,RD,RT	`vlw` `addu`	VT,(RD) RD,RD,RT
`lbuai.v`	VT,RD,RT	`vlbu` `addu`	VT,(RD) RD,RD,RT
`lhuai.v`	VT,RD,RT	`vlhu` `addu`	VT,(RD) RD,RD,RT
`sbai.v`	VT,RD,RT	`vsb` `addu`	VT,(RD) RD,RD,RT
`shai.v`	VT,RD,RT	`vsh` `addu`	VT,(RD) RD,RD,RT
`swai.v`	VT,RD,RT	`vsw` `addu`	VT,(RD) RD,RD,RT
`sbuai.v`	VT,RD,RT	`vsbu` `addu`	VT,(RD) RD,RD,RT
`shuai.v`	VT,RD,RT	`vshu` `addu`	VT,(RD) RD,RD,RT

Torrent control register instructions.

`ctvu` RT,RS $\Longrightarrow$ `vcfint` RT,RS
`cfvu` RT,RS $\Longrightarrow$ `vctint` RT,RS

Torrent conditional moves.

Operation	Pseudo-Instruction	Emitted Instruction
move $\neq 0$	`cmvnez.vv` VD,VS,VT `cmvnez.vs` VD,VS,RT	`movnz.vv` VD,VS,VT `movnz.vs` VD,VS,RT
move ≥ 0	`cmvgez.vv` VD,VS,VT `cmvgez.vs` VD,VS,RT	`movgez.vv` VD,VS,VT `movgez.vs` VD,VS,RT
move ≤ 0	`cmvlez.vv` VD,VS,VT `cmvlez.vs` VD,VS,RT	`movlez.vv` VD,VS,VT `movlez.vs` VD,VS,RT
move $= 0$	`cmveqz.vv` VD,VS,VT `cmveqz.vs` VD,VS,RT	`movz.vv` VD,VS,VT `movz.vs` VD,VS,RT
move < 0	`cmvltz.vv` VD,VS,VT `cmvltz.vs` VD,VS,RT	`movltz.vv` VD,VS,VT `movltz.vs` VD,VS,RT
move > 0	`cmvgtz.vv` VD,VS,VT `cmvgtz.vs` VD,VS,RT	`movgtz.vv` VD,VS,VT `movgtz.vs` VD,VS,RT

Torrent vector insert/extract instructions.

`vext.v` VD,VS,RT $\Longrightarrow$ `extract.vs` VD,VS,RT
`vext.s` RT,VD,RS $\Longrightarrow$ `vtint` RT,RS(VD)
`vins.s` RT,VD,RS $\Longrightarrow$ `vfint` RT,RS(VD)

Torrent shift instructions. These instructions are interpreted in the same manner as the corresponding commands in the VP instruction set (without the 'v').

Pseudo-Instruction	Emitted Instruction
`sllv.vv` VD,VS,VT `sllv.vs` VD,VS,RT `sllv.sv` VD,RS,VT	`sll.vv` VD,VS,VT `sll.vs` VD,VS,RT `sll.sv` VD,RS,VT
`srlv.vv` VD,VS,VT `srlv.vs` VD,VS,RT `srlv.sv` VD,RS,VT	`srl.vv` VD,VS,VT `srl.vs` VD,VS,RT `srl.sv` VD,RS,VT
`srav.vv` VD,VS,VT `srav.vs` VD,VS,RT `srav.sv` VD,RS,VT	`sra.vv` VD,VS,VT `sra.vs` VD,VS,RT `sra.sv` VD,RS,VT

Torrent Fixed-Point Support. Very little support has been added, as yet, for synthesizing the Torrent fixed-point arithmetic instructions. The following two instruction mappings have been defined, to satisfy our own internal requirements for running MachSUIF-generated code:

Pseudo-Instruction	Emitted Instruction
`fxadd.vv` VD,VS,VT,RD	`nop`

`fxmul.vv` VD,VS,VT,RD	`srl $at,` RD`,0x18`
	`andi $at, $at,0x3`
	`bnezl $at,` NEXT
	` mulu.vv` VD, VS,VT
	`mulu.vv` VD, VS,VT
	`next:`

	31 … 26	25	… 21	20 … 16	15 … 11	10 … 6	5 … 0
Cop2(VP) T-type	010010	0	FORMAT	RT/FT	RS	VD/0	000000
				oth reg	VP reg	VP reg	
Cop2(VP) M-type	010010	0	1100	VT	RS/VS/0	RD	FUNCTION
				data	stride	base	
Cop2(VP) V-type	010010	1	FORMAT	RT/FT/VT/0	RS/FS/VS	VD/0	FUNCTION
				source	source	dest	

Figure 13: VP Instruction Format and Instruction Types

22:21 FORMAT Bit Encoding (ignore MSB of column headers)

25:23	000	001	010	011
000	vtint	vtfp.s	vctint$^{\mho}$	
001	vfint	vffp.s	vcfint$^{\mho}$	
010		vtfp.d		
011	$\boxed{\text{vmem}}$	vffp.d		
100	$\boxed{\text{vv}}$	$\boxed{\text{vs/sv}}$		
101				
110	$\boxed{\text{s.vv}}$	$\boxed{\text{s.vs/s.sv}}$		
111	$\boxed{\text{d.vv}}$	$\boxed{\text{d.vs/d.sv}}$		

Figure 14: FORMAT Bit Encoding

3 Opcode Bit Encoding of COP2(VP) Instructions

This section provides the encoding for the VP instructions. The VP instruction format and types are shown in Figure 13. Figure 14 gives the bit encoding for the FORMAT field. Boxed entries indicate instructions that also use the FUNCTION field to specify the operation to execute.

FUNCTION bit encodings for the boxed FORMAT entries are listed on the following pages. In the FORMAT table and the tables on the following pages, certain instructions are specially marked. Most instructions are binary and have no special markings. Other instructions are marked to identify unary operations (eg., cvt.s$^{\mho}$), noncommutative operations that cannot be synthesized without using an extra register (eg., sub$^{\neq}$), and operations to be implemented in the future (eg., *feq*$^{\mathcal{F}}$). For unary operations, the source register that is unused depends upon the instruction type and is set to 0 as indicated in Figure 13.

These tables also give the opcode mnemonics for the VP instructions. The mnemonics for the transfer functions are given in the FORMAT table (Figure 14). For the memory instructions, the opcode mnemonics are given in the table for the `vmem` FUNCTION bit encodings. The other highlighted entries (i.e., when bit 25 is equal to 1) in the FORMAT table specify the computational instructions. For these instructions, the full opcode mnemonic is obtained by appending FORMAT to the mnemonic given in the corresponding FUNCTION bit encoding table. The suffix specifies both the data format (s,d) and the source type (vv,vs,sv)

5:3 \ 2:0	000	001	010	011	100	101	110	111
	FUNCTION Bit Encoding for vmem (Mvmem1)							
000	vl (vector load with unit stride)							
001	vs (vector store) with unit stride							
010	vlst (vector load with constant stride)							
011	vsst (vector store with constant stride)							
100								
101								
110	vlx (vector load with indexed offsets)							
111	vsx (vector store with indexed offsets)							

5:3 \ 2:0	000	001	010	011	100	101	110	111
	FUNCTION Bit Encoding for vmem (Mvmem2)							
000	vlb	vlh		vlw	vlbu	vlhu	vl.s	vl.d
001	vsb	vsh		vsw	vsbu	vshu	vs.s	vs.d
010	vlbst	vlhst		vlwst	vlbust	vlhust	vlst.s	vlst.d
011	vsbst	vshst		vswst	vsbust	vshust	vsst.s	vsst.d
100								
101								
110	vlbx	vlhx		vlwx	vlbux	vlhux	vlx.s	vlx.d
111	vsbx	vshx		vswx	vsbux	vshux	vsx.s	vsx.d

Figure 15: FUNCTION Encodings for Vector Memory Instructions (vmem)

FUNCTION Bit Encoding for .vv (Ivvfmt)

5:3 \ 2:0	000	001	010	011	100	101	110	111
000	add	addu	mul	mulu	sub$^{\neq}$	subu$^{\neq}$	div$^{\neq}$	divu$^{\neq}$
001	movgtz		movgez		movltz		movlez	
010	seq		*feq*$^{\mathcal{F}}$		movz		movnz	
011	sge$^{\neq}$	sgeu$^{\neq}$	*fge*$^{\neq\ \mathcal{F}}$	*fgeu*$^{\neq\ \mathcal{F}}$	slt$^{\neq}$	sltu$^{\neq}$	*flt*$^{\neq\ \mathcal{F}}$	*fltu*$^{\neq\ \mathcal{F}}$
100	sll$^{\neq}$		srl$^{\neq}$	sra$^{\neq}$	and	or	xor	nor
101								
110								
111								

FUNCTION Bit Encoding for .vs (Ivsfmt)

5:3 \ 2:0	000	001	010	011	100	101	110	111
000	add	addu	mul	mulu	sub$^{\neq}$	subu$^{\neq}$	div$^{\neq}$	divu$^{\neq}$
001	movgtz		movgez		movltz		movlez	
010	seq		*feq*$^{\mathcal{F}}$		movz		movnz	
011	sge$^{\neq}$	sgeu$^{\neq}$	*fge*$^{\neq\ \mathcal{F}}$	*fgeu*$^{\neq\ \mathcal{F}}$	slt$^{\neq}$	sltu$^{\neq}$	*flt*$^{\neq\ \mathcal{F}}$	*fltu*$^{\neq\ \mathcal{F}}$
100	sll$^{\neq}$	extract	srl$^{\neq}$	sra$^{\neq}$	and	or	xor	nor
101								

FUNCTION Bit Encoding for .sv (Ivsfmt)

5:3 \ 2:0	000	001	010	011	100	101	110	111
110	sll$^{\neq}$	*expand*$^{\mho\ \mathcal{F}}$	srl$^{\neq}$	sra$^{\neq}$	sub$^{\neq}$	subu$^{\neq}$	div$^{\neq}$	divu$^{\neq}$
111	sge$^{\neq}$	sgeu$^{\neq}$	*fge*$^{\neq\ \mathcal{F}}$	*fgeu*$^{\neq\ \mathcal{F}}$	slt$^{\neq}$	sltu$^{\neq}$	*flt*$^{\neq\ \mathcal{F}}$	*fltu*$^{\neq\ \mathcal{F}}$

Figure 16: FUNCTION Encodings for Integer Data Format (.vv,.vs,.sv)

FUNCTION Bit Encoding for s.vv and d.vv (Svvfmt,Dvvfmt)

5:3 \ 2:0	000	001	010	011	100	101	110	111
000	add		mul		sub$^{\neq}$		div$^{\neq}$	
001	movgtz$^{\neq}$		movgez$^{\neq}$		movltz$^{\neq}$		movlez$^{\neq}$	
010	seq		*feq*$^{\mathcal{F}}$		movz$^{\neq}$		movnz$^{\neq}$	
011	sge$^{\neq}$		*fge*$^{\neq\,\mathcal{F}}$		slt$^{\neq}$		*flt*$^{\neq\,\mathcal{F}}$	
100	cvt.s$^{\mho}$	cvt.d$^{\mho}$		cvt.w$^{\mho}$				
101	sqrt$^{\mho}$	*rsqrt*$^{\mho\,\mathcal{F}}$		*recip*$^{\mho\,\mathcal{F}}$	*round.w*$^{\mho\,\mathcal{F}}$	*trunc.w*$^{\mho\,\mathcal{F}}$	*ceil.w*$^{\mho\,\mathcal{F}}$	*floor.w*$^{\mho\,\mathcal{F}}$
110	*c.f*$^{\mho\,\mathcal{F}}$	*c.un*$^{\mho\,\mathcal{F}}$	*c.eq*$^{\mho\,\mathcal{F}}$	*c.ueq*$^{\mho\,\mathcal{F}}$	*c.olt*$^{\mho\,\mathcal{F}}$	*c.ult*$^{\mho\,\mathcal{F}}$	*c.ole*$^{\mho\,\mathcal{F}}$	*c.ule*$^{\mho\,\mathcal{F}}$
111	*c.sf*$^{\mho\,\mathcal{F}}$	*c.ngle*$^{\mho\,\mathcal{F}}$	*c.seq*$^{\mho\,\mathcal{F}}$	*c.ngl*$^{\mho\,\mathcal{F}}$	*c.lt*$^{\mho\,\mathcal{F}}$	*c.nge*$^{\mho\,\mathcal{F}}$	*c.le*$^{\mho\,\mathcal{F}}$	*c.ngt*$^{\mho\,\mathcal{F}}$

FUNCTION Bit Encoding for s.vs and d.vs (Svsfmt,Dvsfmt)

5:3 \ 2:0	000	001	010	011	100	101	110	111
000	add		mul		sub$^{\neq}$		div$^{\neq}$	
001	movgtz$^{\neq}$		movgez$^{\neq}$		movltz$^{\neq}$		movlez$^{\neq}$	
010	seq		*feq*$^{\mathcal{F}}$		movz$^{\neq}$		movnz$^{\neq}$	
011	sge$^{\neq}$		*fge*$^{\neq\,\mathcal{F}}$		slt$^{\neq}$		*flt*$^{\neq\,\mathcal{F}}$	
100								
101								

FUNCTION Bit Encoding for s.sv and d.sv (Svsfmt,Dvsfmt)

5:3 \ 2:0	000	001	010	011	100	101	110	111
110	movgtz$^{\neq}$	*expand*$^{\mho\,\mathcal{F}}$	movgez$^{\neq}$		sub$^{\neq}$	movltz$^{\neq}$	div$^{\neq}$	movlez$^{\neq}$
111	sge$^{\neq}$		*fge*$^{\neq\,\mathcal{F}}$		slt$^{\neq}$	movz$^{\neq}$	*flt*$^{\neq\,\mathcal{F}}$	movnz$^{\neq}$

Figure 17: FUNCTION Encodings for Single- and Double-Precision Floating-Point Data Formats ({s,d}.{vv,vs,sv})