MIPS® SIMD Architecture

MIPS® SIMD Architecture (MSA) is designed to support general purpose Single Instruction Multiple Data (SIMD) processing using vectors of 8-, 16-, 32-, and 64-bit integer, 16- and 32-bit fixed-point, or 32- and 64-bit floating-point elements. It is a simple, yet very efficient instruction set built on the same RISC principles pioneered by MIPS.
This whitepaper introduces MSA and describes its key features.

Document Number: MD00926
Revision 1.00
December 4, 2012

MIPS Technologies, Inc.
955 East Arques Avenue
Sunnyvale, CA 94085-4521

Contents

1 Introduction

The MIPS® SIMD Architecture (MSA) module adds new instructions to the industry-standard MIPS architecture that allow efficient parallel processing of vector operations. This functionality is of growing importance across a range of consumer electronics and enterprise applications.

In consumer electronics, while dedicated, non-programmable hardware aids the CPU and GPU by handling heavy-duty multimedia codecs such as H.264, there is a recognized trend toward adding a software-programmable solution in the CPU to handle emerging codecs or a small number of functions not covered by the dedicated hardware. In this way, SIMD can provide increased system flexibility, and the MSA is ideal for these applications.

However, the MSA is not just another multimedia SIMD extension. Rather than focusing on narrowly defined instructions that must have optimized code written manually in assembly language in order to be utilized, the MSA is designed to accelerate compute-intensive applications in conjunction with leveraging generic compiler support.

A new class of emerging applications – including data mining, feature extraction in video, image and video processing, human-computer interaction, and others – have some built-in data parallelism that lends itself well to SIMD. These new compute-intensive applications will not be written in assembly for any specific architecture, but rather in C or OpenCL code using operations on vector data types.

The MSA module was implemented with strict adherence to RISC (Reduced Instruction Set Computer) design principles. From the beginning, MIPS architects designed the MSA with a carefully selected, simple SIMD instruction set that is not only programmer- and compiler-friendly, but also hardware-efficient in terms of speed, area, and power consumption. The simple instructions are also easy to support within high-level languages such as C or OpenCL, enabling fast and simple development of new code, as well as leverage of existing code.

This paper describes the new instructions that comprise the MSA.

2 Overview

The MSA complements the well-established MIPS architecture with a set of more than 150 new instructions operating on 32 vector registers of 8-, 16-, 32-, and 64-bit integer, 16-and 32-bit fixed- point, or 32- and 64-bit floating-point data elements. In the current release, MSA implements 128-bit wide vector registers shared with the 64-bit wide floating-point unit (FPU) registers.

In multi-threaded implementations, MSA allows for fewer than 32 physical vector registers per hardware thread context. The thread contexts have access to as many vector registers as needed, up to the full 32 vector registers set defined by the architecture. When the hardware runs out of physical vector registers, the OS re-schedules the running threads or processes to accommodate the pending requests. The actual mapping of the physical vector registers to the hardware thread contexts is managed by the hardware.

The MSA floating-point implementation is compliant with the IEEE Standard for Floating-Point Arithmetic 754^{TM}-2008. All standard operations are provided for 32-bit and 64-bit floating-point data. 16-bit floating-point storage format is supported through conversion instructions to/from 32-bit floating-point data.

For compare and branch, MSA uses no global condition flags: compare instructions write the results per vector element as all zero or all one bit values. Branch instructions test for zero or not zero element(s) or vector value.

3 Vector Registers

The MSA operates on 32, 128-bit wide vector registers. If both MSA and the scalar floating-point unit (FPU) are present, the 128-bit MSA vector registers extend and share the 64-bit FPU registers. MSA and FPU cannot both be present, unless the FPU has 64-bit floating-point registers.

MSA vector registers have four data formats: byte (8-bit), halfword (16-bit), word (32-bit), doubleword (64-bit). Corresponding to the associated data format, a vector register consists of a number of elements indexed from 0 to n, where the least significant bit of the 0th element is the vector register bit 0 and the most significant bit of the nth element is the vector register bit 127.

When both the FPU and the MSA are present, the floating-point registers are mapped on the corresponding MSA vector registers as the 0th elements.

3.1 Registers Layout

Figure 1 through Figure 4 show the vector register layout for elements of all four data formats, where [n] refers to the nth vector element and, MSB and LSB stand for the element's Most Significant and Least Significant Byte.

Figure 1 MSA Vector Register Byte Elements

127 120	119 112	111 104	103 96	95 88	87 80	79 72	71 64	63 56	55 48	47 40	39 32	31 24	23 16	15 8	7 0
[15]	[14]	[13]	[12]	[11]	[10]	[9]	[8]	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]

Figure 2 MSA Vector Register Halfword Elements

127	112	111	96	95	80	79	64	63	48	47	32	31	16	15	0
[7]		[6]		[5]		[4]		[3]		[2]		[1]		[0]	
MSB	LSB	MSB	LSB	MSB	LSB	MSB	LSB	MSB	LSB	MSB	LSB	MSB	LSB	MSB	LSB

Figure 3 MSA Vector Register Word Elements

127			96	95			64	63			32	31			0
[3]				[2]				[1]				[0]			
MSB			LSB	MSB			LSB	MSB			LSB	MSB			LSB

Figure 4 MSA Vector Register Doubleword Elements

127							64	63							0
[1]								[0]							
MSB							LSB	MSB							LSB

MSA vectors are stored in memory starting from the 0th element at the lowest byte address. The byte order of each element follows the big- or little-endian convention of the system configuration.

3.2 Floating-Point Registers Mapping

The shared FPU register read and write operations are defined as follows:

- A read operation from the floating-point register *r*, where *r* = 0, …, 31, returns the value of the element with index 0 in the vector register *r*. The element's format is word for 32-bit (single precision floating-point) read or double for 64-bit (double precision floating-point) read.
- A write operation of value *V* to the floating-point register *r*, where *r* = 0, …, 31, writes *V* to the element with index 0 in the vector register *r* and writes 0 to all remaining elements. Figure 5 and Figure 6 show the effect of writing a 32-bit (single precision floating-point) and a 64-bit (double precision floating-point) value *V* to a vector register.

Figure 5 FPU Word Write Effect on the MSA Vector Register

127 96	95 64	63 32	31 0
0	0	0	Word value *V*

Figure 6 FPU Doubleword Write Effect on the MSA Vector Register

127 64	63 0
0	Doubleword value *V*

3.3 Register Partitioning

Vector register usage patterns show significant variation between the running threads/processes. A few compute-intensive threads frequently need a lot more vector registers than the vast majority of running threads. The MIPS architecture supports up to nine Virtual Processing Elements (VPEs) which, in essence, are virtual CPUs, each with its own thread context. Rather than outfitting all thread contexts with 32 vector registers, the MSA implements a partitioning scheme, where a pool of k vector registers are used for n thread contexts, with $k < 32 * n$. For example, a four-VPE MIPS CPU could be designed with 72 vector registers instead of the full 128 (32 * 4) registers.

As for any limited hardware resource shared among multiple threads, when all physical vector registers have been allocated, the OS will re-schedule the running threads to free up enough vector registers for the pending requests. The OS is also responsible for saving and restoring the vector registers on software context switching. The actual mapping of the physical registers to the thread contexts is managed by the hardware itself, and it is completely transparent to software.

The hardware/software interface for vector register allocation and software context switching is based on a few MSA control registers and the MSA Access Disabled Exception. MSA control registers keep track of the current thread's vector register state (e.g., allocated, saved, modified), allowing the OS to implement lazy context switching and on-demand allocation.

The performance of a multi-threaded MSA implementation with less than 32 vector registers per thread context depends on the actual register usage at run-time and the OS scheduling strategy. In a typical application, one software thread might use a lot of vector registers for a longer time, while the other threads sporadically use very few. The OS could schedule the most demanding software thread on the same thread context, while time-sharing the other contexts for the software threads with a lighter usage pattern.

4 Instruction Syntax

The MSA assembly language has specific syntax elements to identify the operation/instruction name (ADDS_S for signed saturated add), specify a destination data format (byte, halfword, word, doubleword, or the vector itself), select vector registers ($w0, …, $w31) or general purpose registers ($0, …, $31) operands, and select a single vector register data element or an immediate value.

4.1 Data Format

MSA instructions have two or three register, immediate, or element operands. One of the destination data format abbreviations shown in Table 1 is appended to the instruction name. Note that the data format abbreviation is the same regardless of the instruction's assumed data type. For example, all integer, fixed-point, and floating-point instructions operating on 32-bit elements use the same word (.W in Table 1) data format.

Table 1 Data Format Abbreviations

Data Format	Abbreviation
Byte, 8-bit	.B
Halfword16-bit	.H

Table 1 Data Format Abbreviations

Data Format	Abbreviation
Word, 32-bit	.W
Doubleword, 64-bit	.D
Vector	.V

4.2 Vector Element Selection

MSA instructions select the n^{th} element in the vector register *ws* (*ws[n]* in assembly language) based on the data format *df*. Valid element index values for various data formats and vector register sizes are shown in Table 2.

Table 2 Valid Element Index Values

Data Format	Element Index
Byte	$n = 0, \ldots, 15$
Halfword	$n = 0, \ldots, 7$
Word	$n = 0, \ldots, 3$
Doubleword	$n = 0, 1$

4.3 Examples

Let us assume that vector registers W1 and W2 are initialized to the word values shown in Figure 7, Figure 8, and that general-purpose register R2 is initialized as shown in Figure 9.

Figure 7 Source Vector W1 Values

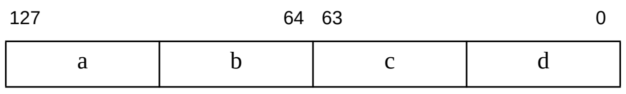


Figure 8 Source Vector W2 Values

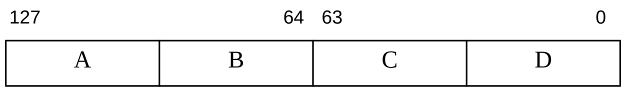
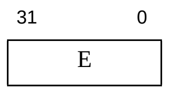


Figure 9 Source GPR 2 Value



Regular MSA instructions operate element-by-element with identical source, target, and destination data types. Figure 10 through Figure 13 have the resulting values of destination vectors W4, W5, W6, and W7 after executing the following sequence of word additions and move instructions:

```
addv.w $w5,$w1,$w2
fill.w $w6,$2
addvi.w $w7,$w1,17
splati.w $w8,$w2[2]
```

Figure 10 Destination Vector W5 Value for ADDV.W Instruction

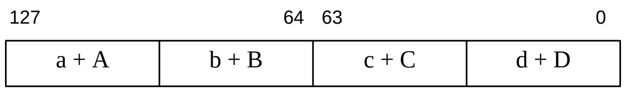


Figure 11 Destination Vector W6 Value for FILL.W Instruction

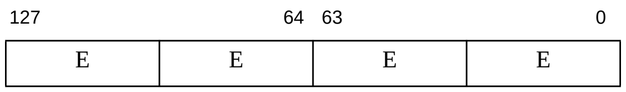


Figure 12 Destination Vector W7 Value for ADDVI.W Instruction

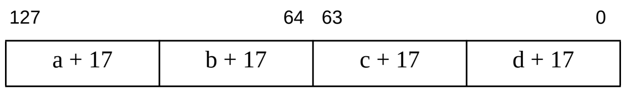
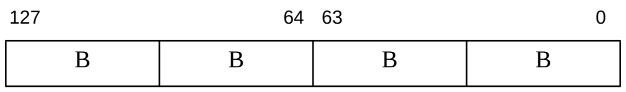


Figure 13 Destination Vector W8 Value for SPLAT.W Instruction

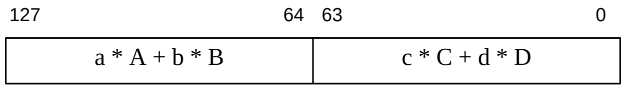


Other MSA instructions operate on adjacent odd/even source elements, generating results on data formats twice as wide. The signed doubleword dot product DOTP_S is such an instruction (see Figure 14):

```
dotp_s.d $w9,$w1,$w2
```

Note that the actual instruction specifies .D (doubleword) as the destination's data format. The data format of the source operands is inferred as being also signed and half the width, i.e. word, in this case.

Figure 14 Destination Vector W9 Value for DOTP_S Instruction



5 Instruction Description

True to the RISC design tradition, the MSA implements simple, homogeneous instructions with explicit functionality. There are no mixed general purpose and vector register operations except for data movement. This simplifies the hardware implementation, and allows for faster and independent execution of scalar and vector instructions.

In the MSA, complex operations that can be implemented by a sequence of two or three existing instructions are not implemented as single instructions. This could increase the code size to some extent, but greatly benefits the execution speed. For example, MSA has no instructions for horizontal arithmetic operations between all elements in the same vector register because these are complex operations easily implemented with few additional element shuffle instructions.

Most MSA instructions operate vector element-by-vector element in a typical SIMD manner. Few instructions handle the operands as bit vectors, because the elements don't make sense (e.g., bitwise logical operations). For certain instructions, the source operand could be a scalar immediate value or a vector element selected by an immediate index. The scalar value is being replicated for all vector elements.

The MSA instruction set implements the following categories of instructions: arithmetic, bitwise, floating-point arithmetic, floating-point compare, floating-point conversions, fixed-point multiplication, branch and compare, load/store, element move, and element shuffle. Each category is briefly described in the following subsections.

5.1 Arithmetic Instructions

Arithmetic instructions (Table 6) include additions and subtractions combined with saturation and absolute value operations. There is also a dedicated saturation instruction for arbitrary clamping at any bit position. Average computing instructions are provided for full precision (i.e. no wrap-around on overflow) add and shift with or without rounding. Minimum and maximum value selection instructions work on signed, unsigned, and absolute values.

Addition, subtraction, minimum, and maximum instructions also take a small, 5-bit constant value to operate across all elements.

Multiply, multiply-add/sub, divide, and remainder (modulo) are defined with operands and results of the same size ranging from bytes to doublewords. A set of dot product instructions perform partitioned multiplication with reduction: essentially a multiply-add or sub on adjacent elements, with the full-precision result double the size (see the example Figure 14).

Bitwise instructions (Table 7) include logical (e.g., AND, OR, NOR, and XOR) operations and shifts. All operate on two vector registers or on a vector register and an immediate constant. More complex logical instructions do selective bit copy from two source vectors to the destination. Leading zero/one bit counting and population counting (all one bits) instructions are available as well.

5.2 Floating-Point Instructions

The MSA floating-point implementation is compliant with the IEEE Standard for Floating-Point Arithmetic 754^{TM}-2008. The floating-point arithmetic operations implemented by dedicated instructions are: addition/subtract, multiply/divide, fused multiply add/sub, base 2 exponentiation and integer logarithm, max/min including for absolute values, and integer rounding (Table 8).

The floating-point compare instructions (Table 9) are similar with the integer comparisons: all compare are quiet and set destination bits to zero (false) or one (true). The floating-point specific unordered relations are supported by a dedicated quiet compare unordered instruction and a complete set of signaling compare instructions.

Format conversion instructions (Table 10) cover single (32-bit) to/from double-precision (64-bit) and single to/from 16-bit floating-point format. Integer and fixed-point conversions are also supported.

In the case of a floating-point exception, each faulting vector element is precisely identified without the need for software emulation for all vector elements.

5.3 Fixed-Point Multiplication Instructions

The fixed-point data formats are Q15 and Q31, i.e. one sign bit and 15 or 31 fractional bits, representing values in the [-1, 1) interval. While the fixed-point add/sub is the regular 2's complement add/sub with saturation, the multiplication operation requires scaling (left shift) with saturation.

The MSA has dedicated fixed-point multiplication instructions (Table 11) with optional rounding.

5.4 Branch and Compare Instructions

Branch and compare instructions (Table 12) are based on truth values: zero for false and non-zero for true. There are no dedicated condition flags.

The compare instructions set the destination element to the truth value of the compare operation for the corresponding source elements. All compare instructions accept a small, 5-bit constant as the second compare operand across all vector elements.

Both branch-on-false and branch-on-true condition instructions are provided, because the vector under test contains multiple truth values that cannot be negated by simply changing the compare operator. As such, there is a pair of branch-on-false (zero) instructions that test if at least one element is zero or if all elements are zero, and a pair of branch-on-true (not zero) instructions that test if all elements are not zero, or if at least one element is not zero.

5.5 Load/Store and Element Move Instructions

The MSA is very flexible and consistent regarding data transfers between the vector registers and the general-purpose registers (GPRs) or memory. Data transfer instructions (Table 13) include vector memory load/store and element move instructions such as vector element data copy to GPR, all vector elements fill with GPR or immediate data, and insert GPR data to a specific element. The load/store instructions require 128-bit (16-byte) memory- address alignment.

All data transfer instructions are typed, i.e., the data format is explicitly specified. This is particularly important for the vector load/store instructions, because it allows any halfword, word, or doubleword data to make the round-trip between GPRs, memory, and vector registers without any need for endian related byte swaps. For example, a store halfword (source) vector register will write the eight halfword values to memory, which then can be loaded as halfwords one-by-one in GPRs, which then can be transferred one-by-one to another (destination) vector register. The source vector register from which the halfword values were initiated is identical to the destination vector register, regardless of the endian memory mode.

5.6 Element Shuffle Instructions

Vector elements can be shuffled based on either a pre-defined pattern or an arbitrary mapping function. Pre-defined patterns are more efficient because no prior set-up is required. Mapping functions provide the most general shuffling, but could take an extra vector register to specify where each source element will be put in the destination vector.

The MSA has both generic mapping and pre-defined pattern-shuffle instructions (Table 14). Pre-defined pattern instructions interleave odd or even elements from two source vectors, or pack all odd or all even elements from two source vectors into the upper half and the lower half of a destination vector.

A second class of predefined patterns are geometrical in nature: the two source vectors seen as byte arrays (of one line by eight columns, two lines by four columns, or four lines by two columns) are horizontally concatenated. The destination is a byte array selected by a sliding window of similar shape (array of one by eight, two by four, or four by two) over the concatenation of the source arrays.

6 GNU Compiler Support

GNU C Compiler (GCC) support for SIMD operations is based on a number of standard pattern names used for code generation. Ideally, the instruction set should implement as many of these operations as possible. In the process of MSA instruction selection and definition, supporting the standard GCC SIMD patterns was one of the most important objectives. Most of these patterns translate directly in single MSA instructions.

Another aspect related to efficient vector code compilation for SIMD architectures is the interoperability between the C language arrays (of scalar data types) and the native vector data types. To support seamless mixing of scalar and vector data types operations, the MSA provides a rich set of typed data transfer instructions.

The MSA is supported by the GNU toolchain starting with GAS (GNU Assembler) 2.22.51 and GCC 4.7.3. The command line options and assembly directives to enable/disable MSA are shown in Table 3.

The GCC options `-mfp64` and `-mhard-float` enforce the compatibility of the calling conventions of MSA and FPU, based on the fact that in the current release, MSA vector registers are shared with the 64-bit wide floating-point unit (FPU) registers.

Table 3 MSA GNU Options and Directives

	GAS		**GCC**	
	Enable	**Disable**	**Enable**	**Disable**
Command Line Options	`-mmsa`	`-mno-msa`	`-mmsa -mfp64 -mhard-float`	`-mno-msa`
Assembly Directives	`.set msa`	`.set nomsa`		

The GCC integer and floating-point vector data types with generic MSA operation support are listed in Table 4 and Table 5.

Table 4 GCC Integer Vector Data Types Supported in MSA

Vector Data Type	C Definition
Vector of 16 signed bytes	`typedef signed char v16i8 __attribute__ ((vector_size(16)));`
Vector of 16 unsigned bytes	`typedef unsigned char v16u8 __attribute__ ((vector_size(16)));`
Vector of 8 signed halfwords	`typedef short v8i16 __attribute__ ((vector_size(16)));`
Vector of 8 unsigned halfwords	`typedef unsigned short v8u16 __attribute__ ((vector_size(16)));`
Vector of 4 signed words	`typedef int v4i32 __attribute__ ((vector_size(16)));`
Vector of 4 unsigned words	`typedef unsigned int v4u32 __attribute__ ((vector_size(16)));`
Vector of 2 signed doublewords	`typedef long long v2i64 __attribute__ ((vector_size(16)));`
Vector of 2 unsigned doublewords	`typedef unsigned long long v2u64 __attribute__ ((vector_size(16)));`

Table 5 GCC Floating-Point Vector Data Types Supported in MSA

Vector Data Type	C Definition
Vector of 4 single precision floating-point values	`typedef float v4f32 __attribute__ ((vector_size(16)));`
Vector of 2double precision floating-point values	`typedef double v2f64 __attribute__ ((vector_size(16)));`

MSA instructions are available to the C/C++ programmer either by the inline assembly `__asm__` directive, by `__builtin_*()` intrinsics, or when using most of the C/C++ operators on vector data types. The list of supported vector C/C++ operators include: `+`, `-`, `*`, `/`, `%`, `^`, `|`, `&`, `<<`, `>>`, `==`, `!=`, `<`, `<=`, `>`, `>=`, `~`.

For example, adding, comparing, or shuffling two single-precision floating-point vectors, as in:

```
v4i32 m, t;
v4f32 a, b, c, s;

a = b + c;
t = b < c;
s = __builtin_shuffle (b, c, m);
```

compiles directly in MSA word floating-point add and compare instructions and word shuffle instruction:

```
fadd.w $w3,$w0,$w1 # a is in $w3, b in $w0, c in $w1
fclt.w $w4,$w0,$w1 # t is in $w4
```

```
vshf.w $w0,$w1,$w2 # m is in $w2, s overwrites $w0
```

Regarding the vector parameter passing conventions, MSA registers are all caller-saved, i.e. temporary registers are not preserved between function calls. The first eight vector parameters are passed in vector registers W4 to W11. When compiled for the MSA, the stack pointer is always aligned to 16 bytes.

7 Evolution

The SIMD architectures have been continuously evolving and likely will continue to do so. However, it remains challenging to program and create compiler support for instructions that continue to grow in complexity over time.

One of the main attributes of the MIPS SIMD Architecture is scalability. The MSA scales nicely with the number of threads using the vector register partitioning scheme. Adding more hardware threads to increase the performance does not result in a proportional increase of the vector registers count.

A wider vector register set of 256 bits is another path to increasing performance. The MSA scales with the vector register width. The instruction set is designed to be independent of the vector register size, allowing for source code (even binary code) compatibility when upgrading to wider vector registers.

With SIMD moving toward mainstream computing, MSA is well positioned to address the emerging compute-intensive applications. MSA is future-proof and extensible through scalability rather than an increase in complexity. The MIPS instruction set has pre-defined scalable extensions that can take advantage of future chips with more gates/transitions available, giving it longevity for multiple generations.

8 MSA Instructions by Category

The following tables list all of the MSA instructions grouped by category: arithmetic, bitwise, floating-point arithmetic, floating-point compare, floating-point conversions, fixed-point, branch and compare, load/store and move, and element permute.

Table 6 MSA Arithmetic Instructions

Mnemonic	Instruction Description
ADDV	Add
ADD_A, ADDS_A	Add/Saturated Add Absolute Values
ADDS_S, ADDS_U	Signed/Unsigned Saturated Add
ASUB_S, ASUB_U	Absolute Value of Signed/Unsigned Subtract
AVE_S, AVE_U	Signed/Unsigned Average
AVER_S, AVER_U	Signed/Unsigned Average with Rounding
DOTP_S, DOTP_U	Signed/Unsigned Dot Product

Table 6 MSA Arithmetic Instructions (Continued)

Mnemonic	Instruction Description
DPADD_S, DPADD_U	Signed/Unsigned Dot Product Add
DPSUB_S, DPSUB_U	Signed/Unsigned Dot Product Subtract
DIV_S, DIV_U	Divide
MADDV	Multiply and Add
MAX_A, MIN_A	Maximum/Minimum of Absolute Values
MAX_S, MAX_U	Signed/Unsigned Maximum
MIN_S, MIN_U	Signed/Unsigned Maximum
MSUBV	Multiply and Subtract
MULV	Multiply
MOD_S/MOD_U	Signed/Unsigned Remainder (Modulo)
SAT_S, SAT_U	Signed/Unsigned Saturate
SUBS_S/SUBS_U	Signed/Unsigned Saturated Subtract
SUBSS_U	Signed Saturated Unsigned Subtract
SUBUS_S	Unsigned Saturated Signed Subtract from Unsigned
SUBV	Subtract

Table 7 MSA Bitwise Instructions

Mnemonic	Instruction Description
AND	Logical And
BCLR	Bit Clear
BINSL, BINSR	Bit Insert Left/Right
BMNZ	Bit Move If Not Zero
BMZ	Bit Move If Zero
BNEG	Bit Negate
BSEL	Bit Select
BSET	Bit Set
NLOC	Leading One Bits Count
NLZC	Leading Zero Bits Count
NOR	Logical Negated Or
PCNT	Population (Bits Set to 1) Count

Table 7 MSA Bitwise Instructions (Continued)

Mnemonic	Instruction Description
OR	Logical Or
SLL	Shift Left
SRA	Shift Right Arithmetic
SRL	Shift Right Logical
XOR	Logical Exclusive Or

Table 8 MSA Floating-Point Arithmetic Instructions

Mnemonic	Instruction Description
FADD	Floating-Point Addition
FDIV	Floating-Point Division
FEXP2	Floating-Point Base 2 Exponentiation
FLOG2	Floating-Point Base 2 Logarithm
FMADD	Floating-Point Fused Multiply-Add
FMAX, FMIN	Floating-Point Maximum/Minimum
FMAX_A, FMIN_A	Floating-Point Maximum/Minimum of Absolute Values
FMUL	Floating-Point Multiplication
FRINT	Floating-Point Round to Integer
FSQRT	Floating-Point Square Root
FSUB	Floating-Point Subtraction

Table 9 MSA Floating-Point Compare Instructions

Mnemonic	Instruction Description
FCUN	Floating-Point Quiet Compare Unordered
FCEQ	Floating-Point Quiet Compare Equal
FCNE	Floating-Point Quiet Compare Not Equal
FCLT	Floating-Point Quiet Compare Less Than
FCGE	Floating-Point Quiet Compare Greater or Equal
FCLE	Floating-Point Quiet Compare Less or Equal
FCGT	Floating-Point Quiet Compare Greater Than

Table 9 MSA Floating-Point Compare Instructions (Continued)

Mnemonic	Instruction Description
FSEQ	Floating-Point Signaling Compare Equal
FSNE	Floating-Point Signaling Compare Not Equal
FSLT	Floating-Point Signaling Compare Less Than
FSGE	Floating-Point Signaling Compare Greater or Equal
FSLE	Floating-Point Signaling Compare Less or Equal
FSGT	Floating-Point Signaling Compare Greater Than
FCLASS	Floating-Point Class Mask

Table 10 MSA Floating-Point Conversion Instructions

Mnemonic	Instruction Description
FEXDO	Floating-Point Down-Convert Interchange Format
FEXUPL, FEXUPR	Left-Half and Right-Half Floating-Point Up-Convert Interchange Format
FFINT_S, FFINT_U	Floating-Point Convert from Signed/Unsigned Integer
FFQL, FFQR	Left-Half and Right-Half Floating-Point Convert from Fixed-Point
FTINT_S, FTINT_U	Floating-Point Convert to Signed/Unsigned Integer
FTQ	Floating-Point Convert to Fixed-Point

Table 11 MSA Fixed-Point Instructions

Mnemonic	Instruction Description
MADD_Q, MADDR_Q	Fixed-Point Multiply and Add without/with Rounding
MSUB_Q, MSUBR_Q	Fixed-Point Multiply and Subtract without/with Rounding
MUL_Q, MULR_Q	Fixed-Point Multiply without/with Rounding

Table 12 MSA Branch and Compare Instructions

Mnemonic	Instruction Description
BNZ	Branch If Not Zero
BZ	Branch If Zero
CEQ	Compare Equal
CLE_S, CLE_U	Compare Less-Than-or-Equal Signed/Unsigned

Table 12 MSA Branch and Compare Instructions (Continued)

Mnemonic	Instruction Description
CLT_S, CLT_U	Compare Less-Than Signed/Unsigned

Table 13 MSA Load/Store and Move Instructions

Mnemonic	Instruction Description
CFCMSA, CTCMSA	Copy from/copy to MSA Control Register
LD	Load Vector
LDI	Load Immediate
LDX	Load Vector Indexed
MOVE	Vector to Vector Move
FILL	Fill Vector from GPR
INSV	Insert GPR to Vector Element
COPY_S, COPY_U	Copy element to GPR Signed/Unsigned
ST	Store Vector
STX	Store Vector Indexed

Table 14 MSA Element Permute Instructions

Mnemonic	Instruction Description
ILVEV, ILVOD	Interleave Even, Odd
ILVL, ILVR	Interleave the Left, Right
PCKEV, PCKOD	Pack Even/Odd Elements
SHF	Set Shuffle
SLD, SLDI	Element Slide
VSHF	Vector shuffle

Template: nW1.04, Built with tags: 2B ARCH FPU_PS FPU_PSandARCH MIPS32