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 1.03
April 8, 2014

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 C++ 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™-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 0^{th} element is the vector register bit 0 and the most significant bit of the n^{th} 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 0^{th} 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 n^{th} vector element and, MSB and LSB stand for the element's Most Significant and Least Significant Byte.

Figure 1 MSA Vector Register Byte Elements

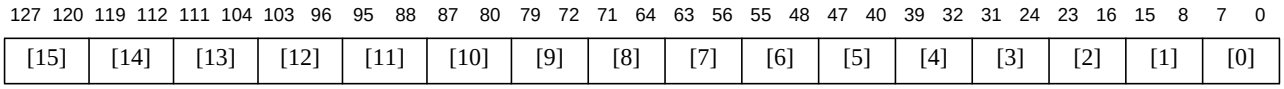


Figure 2 MSA Vector Register Halfword Elements

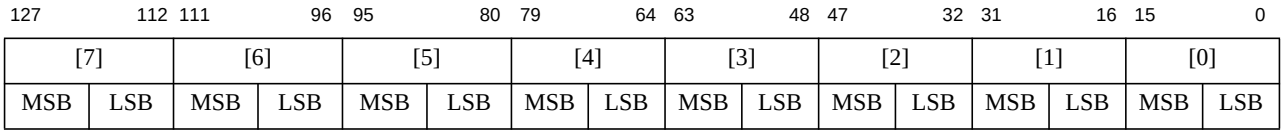


Figure 3 MSA Vector Register Word Elements

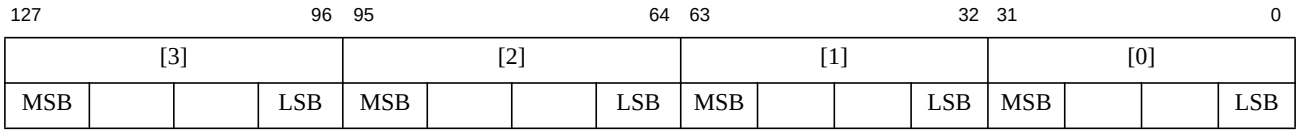
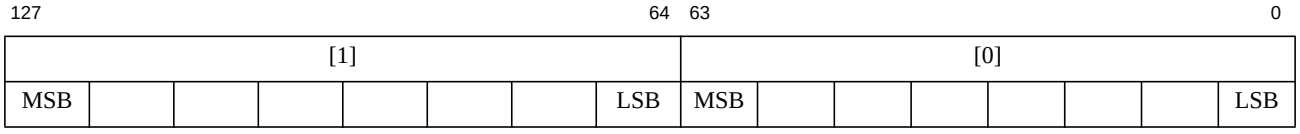


Figure 4 MSA Vector Register Doubleword Elements



MSA vectors are stored in memory starting from the 0^{th} 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 *A* to the floating-point register *r*, where *r* = 0, …, 31, writes *A* to the element with index 0 in the vector register *r* and all remaining elements are **UNPREDICTABLE**. 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 *A* to a vector register.

Figure 5 FPU Word Write Effect on the MSA Vector Register

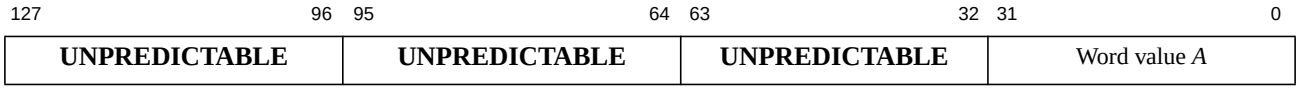
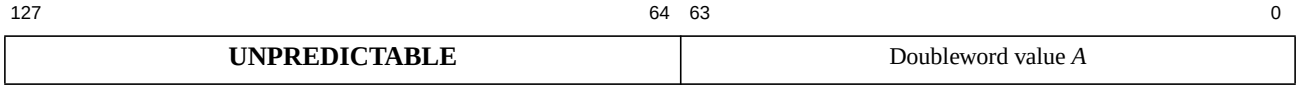


Figure 6 FPU Doubleword Write Effect on the MSA Vector Register



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 DF abbreviations shown in Table 1 is appended to the assembler instruction name INSN[1] as in INSN.DF. 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.

1. Instructions names and data format abbreviations are case insensitive.

Table 1 Data Format Abbreviations

Data Format	Abbreviation
Byte, 8-bit	b
Halfword16-bit	h
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, …, 15
Halfword	*n* = 0, …, 7
Word	*n* = 0, …, 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

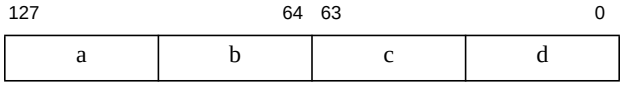


Figure 8 Source Vector $w2 Values

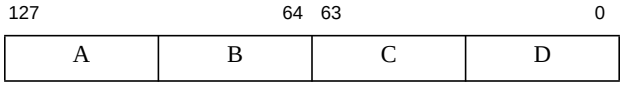
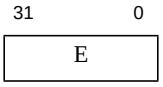


Figure 9 Source GPR $2 Value



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

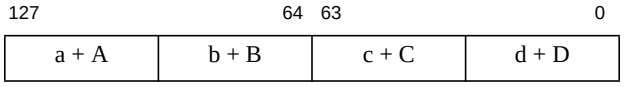


Figure 11 Destination Vector $w6 Value for FILL.W Instruction

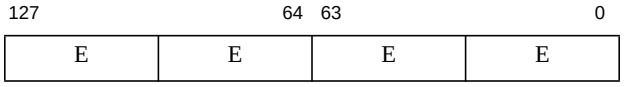


Figure 12 Destination Vector $w7 Value for ADDVI.W Instruction

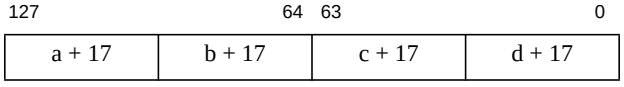
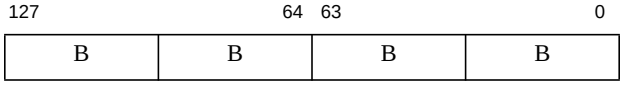


Figure 13 Destination Vector $w8 Value for SPLAT.W Instruction

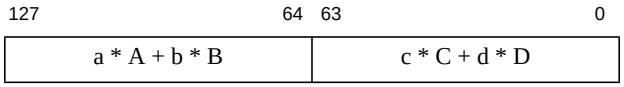


Other MSA instructions operate on adjacent odd/even source elements, generating results on data formats twice as wide. The signed doubleword dot product DOTP_S is such an instruction (see Figure 14):

```
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 GNU C 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.

5.1 Vector Data Types and Intrinsics

The GCC integer and floating-point vector data types with generic MSA operation support as listed in Table 3 and Table 4 are defined in the compiler provided header `<msa.h>`.

It is recommended aligning the vector data to the size of the vector registers. MSA could operate on vectors of any alignment, but there will always be multiple cycles performance loss for each load/store of data not aligned to the size of the vector registers. For interoperability with the standard C arrays, the minimum alignment of a vector data type should be the size of the element type.

Note that any element aligned MSA vector data type `t` has an explicit data format suffix `df` as defined in Table 1 in the format `t_df`.

Table 3 GCC Integer Vector Data Types Supported in MSA

Vector Data Type	Alignment	C Definition
Vector of signed bytes	Vector (16-bytes)	`typedef signed char` **`v16i8`** `__attribute__((vector_size(16), aligned(16)));`
	Byte	`typedef signed char` **`v16i8_b`** `__attribute__((vector_size(16), aligned(1)));`
Vector of unsigned bytes	Vector (16-bytes)	`typedef unsigned char` **`v16u8`** `__attribute__((vector_size(16), aligned(16)));`
	Byte	`typedef unsigned char` **`v16u8_b`** `__attribute__((vector_size(16), aligned(1)));`
Vector of signed halfwords	Vector (16-bytes)	`typedef short` **`v8i16`** `__attribute__((vector_size(16), aligned(16)));`
	Halfword (2-bytes)	`typedef short` **`v8i16_h`** `__attribute__((vector_size(16), aligned(2)));`
Vector of unsigned half-words	Vector (16-bytes)	`typedef unsigned short` **`v8u16`** `__attribute__((vector_size(16), aligned(16)));`
	Halfword (2-bytes)	`typedef unsigned short` **`v8u16_h`** `__attribute__((vector_size(16), aligned(2)));`
Vector of signed words	Vector (16-bytes)	`typedef int` **`v4i32`** `__attribute__((vector_size(16), aligned(16)));`
	Word (4-bytes)	`typedef int` **`v4i32_w`** `__attribute__((vector_size(16), aligned(4)));`
Vector of unsigned words	Vector (16-bytes)	`typedef unsigned int` **`v4u32`** `__attribute__((vector_size(16), aligned(16)));`
	Word (4-bytes)	`typedef unsigned int` **`v4u32_w`** `__attribute__((vector_size(16), aligned(4)));`
Vector of signed doublewords	Vector (16-bytes)	`typedef long long` **`v2i64`** `__attribute__((vector_size(16), aligned(16)));`
	Doubleword (8-bytes)	`typedef long long` **`v2i64_d`** `__attribute__((vector_size(16), aligned(8)));`
Vector of unsigned double-words	Vector (16-bytes)	`typedef unsigned long long` **`v2u64`** `__attribute__((vector_size(16), aligned(16)));`
	Doubleword (8-bytes)	`typedef unsigned long long` **`v2u64_d`** `__attribute__((vector_size(16), aligned(8)));`

Table 4 GCC Floating-Point Vector Data Types Supported in MSA

Vector Data Type	Alignment	C Definition
Vector of single precision floating-point values	Vector (16-bytes)	`typedef float` **`v4f32`** `__attribute__((vector_size(16), aligned(16)));`
	Word (4-bytes)	`typedef float` **`v4f32_w`** `__attribute__((vector_size(16), aligned(4)));`
Vector of double precision floating-point values	Vector (16-bytes)	`typedef double` **`v2f64`** `__attribute__((vector_size(16), aligned(16)));`
	Doubleword (8-bytes)	`typedef double` **`v2f64_d`** `__attribute__((vector_size(16), aligned(8)));`

MSA instructions are available to the C programmer through inline assembly, intrinsics, or vector operators:

- The inline assembly `__asm__` directive is documented by the *Assembler Instructions with C Expression Operands* section in the *Using the GNU GCC Compiler Collection* documentation. The operand constraint for the MSA vector registers is `'f'`. For example, this sequence adds and compares 2 floating point vectors using the assembler FADD.W and FSLT.W instructions:

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

__asm__ volatile (
    "   fadd.w  %w[a],%w[b],%w[c]   \n"
    "   fslt.w  %w[t],%w[b],%w[c]   \n"
    : [a] "=&f"(a), [t] "=&f"(t)
    : [b]   "f"(b), [c]   "f"(c));
```

- The intrinsics are declared by the compiler provided header `<msa.h>`, see Section 6 “Instruction Description”. For example, the same sequence from above to add and compare 2 floating point numbers can be written using `fadd_w()` and `fslt_w()` intrinsics and the compiler will generate FADD.W and FSLT.W instructions:

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

a = fadd_w(b, c);
t = fslt_w(b, c);
```

- The list of supported vector C operators include: `+`, `-` (binary, unary), `*` (multiplication, indirection), `/`, `%`, `^`, `|`, `&` (bitwise ‘and’, reference), `<<`, `>>`, `==`, `!=`, `<`, `<=`, `>`, `>=`, `~`, [] (array subscript), and the ternary operator `?:`. For more details, see the *Vector Extensions* section in the *Using the GNU GCC Compiler Collection* documentation. For example, the above examples of adding and comparing 2 floating point numbers can be written using the + and < operators and the same FADD.W and FSLT.W instructions being generated:

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

a = b + c;
t = b < c;
```

5.2 MSA ABI Extensions

The base O32, N32, and N64 MIPS ABIs are extended to allow efficient use of the vector registers and instructions defined by MSA. The MSA ABI extensions are compatible with the base ABIs in the sense that existing binaries run unchanged on systems supporting MSA, i.e. there are no incompatibilities between the base O32, N32, and N64 and the corresponding MSA extended ABI.

In particular, MSA ABI extensions don't change the base ABI data types layout / alignment, don't introduce new callee-saved (aka saved) registers, and preserve the call-clobbered (aka temporary) or callee-saved status of the aliased floating-point registers. However, vector data types are considered part of the MSA ABI by default.

5.2.1 ABI Requirements

To be compatible with the MSA hardware, an ABI extension for MSA has to support 32 64-bit floating point registers and a stack frame aligned to the size of the vector registers.

N32 and N64 ABIs satisfy the 64-bit floating point registers requirement. The O32 ABI also permits the use of 64-bit floating point registers with the command line option `-mfr1`.

It is possible to adjust the stack alignment at run time using an existing compiler mechanism called dynamic stack realignment. Any ABI that does not meet the MSA stack alignment will use dynamic stack re-alignment. For example, the 16-byte stack alignment of N32 and N64 ABIs is enough for MSA's 128-bit vector registers. O32 has to do dynamic stack re-alignment in this case.

5.2.2 Command Line Options and Assembler Directives

Compiling for MSA, i.e. using the MSA defined instructions and vector registers, is enabled by the `-mmsa` command line option. A function compiled for MSA is referred to as a MSA function. This is the list of various actions related to the `-mmsa` command line option:

- On O32 ABI, `-mmsa` sets the floating-point registers mode to 64-bits, mode which is normally selected by the command line option `-mfr1`.
- O32 ABI requires 16-byte dynamic stack re-alignment when `-mmsa` is present.
- Using vector types without the `-mmsa` option results in a warning stating that a no MSA instructions will be emitted. This warning can be disabled by the command line option `-mno-msa`.
- Implicit conversions between vectors with different number of elements and/or incompatible element types are not allowed with `-mmsa`. Using `-flax-vector-conversions` command line option with `-mmsa` is signaled as an error.

The `-mmsa` command line option defines the pre-processor symbol `__mips_msa` as in

```
#define __mips_msa 1
```

5.2.3 Base O32 Compatibility Mode

A contingency plan to ensure that MSA can be used with pre-existing base O32 objects using 32-bit floating-point registers is also required to give access to 64-bit floating-point registers mode for small regions of code contained within one function.

Essentially, this feature known as “compatibility mode” switches a function into 64-bit floating-point registers mode in the prologue and switches back to 32-bit floating-point registers mode in the epilogue. It also ensures that any function calls are made in 32-bit floating-point registers mode. Full details of this feature are beyond the scope of this document.

The command line options to trigger the compatibility mode are `-mmsa -mfr0`. These options are sufficient to say that the function must operate as if in 32-bit floating-point registers mode and therefore has to use the compatibility feature to enable use of the MSA instruction set.

5.2.4 Vector and Floating-Point Register Usage

The MSA vector registers are temporary, i.e. all live vector registers must be saved before calling a function. This ensures MSA functions can call any other function and also maintain compatibility with future MSA extensions. Note that compilers need to preserve the aliased callee-saved floating-point registers as specified by the base ABIs: even $f20, $f22, …, $f30 for O32 and N32, and $f24, $f25, …, $f30, $f31 for N64. For example, if the vector register $w30 is used, the aliased floating point register $f30 has to be preserved under all ABIs.

Compiling for MSA does not change the base ABI’s vector calling conventions. Vector data types passed or returned by value don’t use the MSA vector registers. Rather, passing and returning vectors by value follow the calling conventions of the base ABI. Floating-point registers are also passed and returned as specified by the base ABIs. For functions with variable arguments, no vector registers are used to pass vector parameters. This falls back to the original variable argument passing scheme from the base ABIs.

The base ABIs incur a substantial overhead when handling vector arguments by value. It is highly recommended to pass and return pointers to vector data types instead.

6 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.

The following sections briefly describe all MSA instructions with the mnemonics, compiler intrinsics, and, if applicable, the equivalent C expressions. For the complete instruction set descriptions see the MSA manuals.

The mnemonics are not shown with all supported data formats. For example, a slightly more complete description of the add vector instruction ADDV should list the byte, halfword, word, and doubleword syntax, the associated intrinsics, and examples of C expressions using both the + operator and the intrinsic (see Table 5). Obviously the vector alignment is not relevant, so all combination of vector aligned data, e.g., `v4u32`, and element aligned data, e.g., `v4u32_w`, are valid but not shown to keep the table at a reasonable size.

If for whatever reason the INSN.DF instruction intrinsic is not available, the compiler generates an external call in the same data format prefixed by `__builtin_` as in `__builtin_`*`insn_df`*`()`.

Table 5 msa_addv() Intrinsic Formats

Mnemonic	Compiler Intrinsic	C Examples

ADDV.B	`v16i8 addv_b(v16i8, v16i8)`
	`v16u8 addv_b(v16u8, v16u8)`
		`v16i8 a, b, c;`
		`c = a + b;`
		`c = addv_b(a, b);`

ADDV.H	`v8i16 addv_h(v8i16, v8i16)`
	`v8u16 addv_h(v8u16, v8u16)`
		`v8i16 a, b, c;`
		`c = a + b;`
		`c = addv_h(a, b);`

ADDV.W	`v4i32 addv_w(v4i32, v4i32)`
	`v4u32 addv_w(v4u32, v4u32)`
		`v4i32 a, b, c;`
		`c = a + b;`
		`c = addv_w(a, b);`

ADDV.D	`v2i64 addv_w(v2i64, v2i64)`
	`v2u64 addv_w(v2u64, v2u64)`
		`v2i64 a, b, c;`
		`c = a + b;`
		`c = addv_d(a, b);`

6.1 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.

Table 6 MSA Arithmetic Instructions

Mnemonic	Compiler Intrinsic	C Expression	Instruction Description
ADDV.*df*[1]	*V*[2] addv_*df*(*V*,*V*)	*v*[3] + *v*	Add
ADDVI.*df*	*V* addvi_*df*(*V*,*K*[4])	*v* + *k*[5]	Add Immediate (immediate value is unsigned)
ADD_A.*df*	*V* add_a_*df*(*V*,*V*)		Add Absolute Values
ADDS_A.*df*	*V* adds_a_*df*(*V*,*V*)		Saturated Add Absolute Values
ADDS_S.*df*	*V* adds_s_*df*(*V*,*V*)		Signed Saturated Add
ADDS_U.*df*	*V* adds_u_*df*(*V*,*V*)		Unsigned Saturated Add
HADD_S.*df*	*V* hadd_s_*df*(*W*[6],*W*)		Signed Horizontal Add
HADD_U.*df*	*V* hadd_u_*df*(*W*,*W*)		Unsigned Horizontal Add
ASUB_S.*df*	*V* asub_s_*df*(*V*,*V*)		Absolute Value of Signed Subtract
ASUB_U.*df*	*V* asub_u_*df*(*V*,*V*)		Absolute Value of Unsigned Subtract
AVE_S.*df*	*V* ave_s_*df*(*V*,*V*)		Signed Average
AVE_U.*df*	*V* ave_u_*df*(*V*,*V*)		Unsigned Average
AVER_S.*df*	*V* aver_s_*df*(*V*,*V*)		Signed Average with Rounding
AVER_U.*df*	*V* aver_u_*df*(*V*,*V*)		Unsigned Average with Rounding
DOTP_S.*df*	*V* dotp_s_*df*(*W*,*W*)		Signed Dot Product
DOTP_U.*df*	*V* dotp_u_*df*(*W*,*W*)		Unsigned Dot Product
DPADD_S.*df*	*V* dpadd_s_*df*(*V*,*W*,*W*)		Signed Dot Product Add
DPADD_U.*df*	*V* dpadd_u_*df*(*V*,*W*,*W*)		Unsigned Dot Product Add
DPSUB_S.*df*	*V* dpsub_s_*df*(*V*,*W*,*W*)		Signed Dot Product Subtract
DPSUB_U.*df*	*V* dpsub_u_*df*(*V*,*W*,*W*)		Unsigned Dot Product Subtract
DIV_S.*df*	*V* div_s_*df*(*V*,*V*)	*v* / *v*	Signed Divide

Table 6 MSA Arithmetic Instructions (Continued)

Mnemonic	Compiler Intrinsic	C Expression	Instruction Description
DIV_U.*df*	*V* div_u_*df*(*V*,*V*)	*v* / *v*	Unsigned Divide
MADDV.*df*	*V* maddv_*df*(*V*,*V*)	*v* + *v* * *v*	Multiply-Add
MAX_A.*df*	*V* max_a_*df*(*V*,*V*)		Maximum of Absolute Values
MIN_A.*df*	*V* min_a_*df*(*V*,*V*)		Minimum of Absolute Values
MAX_S.*df*	*V* max_s_*df*(*V*,*V*)		Signed Maximum
MAXI_S.*df*	*V* maxi_s_*df*(*V*,*K*)		Signed Immediate Maximum
MAX_U.*df*	*V* max_u_*df*(*V*,*V*)		Unsigned Maximum
MAXI_U.*df*	*V* maxi_u_*df*(*V*,*K*)		Unsigned Immediate Maximum
MIN_S.*df*	*V* min_s_*df*(*V*,*V*)		Signed Maximum
MINI_S.*df*	*V* mini_s_*df*(*V*,*K*)		Signed Immediate Maximum
MIN_U.*df*	*V* min_u_*df*(*V*,*V*)		Unsigned Maximum
MINI_U.*df*	*V* mini_u_*df*(*V*,*K*)		Unsigned Immediate Maximum
MSUBV.*df*	*V* msubv_*df*(*V*,*V*)	*v* - *v* * *v*	Multiply-Subtract
MULV.*df*	*V* mulv_*df*(*V*,*V*)	*v* * *v*	Multiply
MOD_S.*df*	*V* mod_s_*df*(*V*,*V*)	*v* % *v*	Signed Remainder (Modulo)
MOD_U.*df*	*V* mod_u_*df*(*V*,*V*)	*v* % *v*	Unsigned Remainder (Modulo)
SAT_S.*df*	*V* sat_s_*df*(*V*,*V*)		Signed Saturate
SAT_U.*df*	*V* sat_u_*df*(*V*,*V*)		Unsigned Saturate
SUBS_S.*df*	*V* subs_s_*df*(*V*,*V*)		Signed Saturated Subtract
SUBS_U.*df*	*V* subs_u_*df*(*V*,*V*)		Unsigned Saturated Subtract
HSUB_S.*df*	*V* hsub_s_*df*(*W*,*W*)		Signed Horizontal Subtract
HSUB_U.*df*	*V* hsub_u_*df*(*W*,*W*)		Unsigned Horizontal Subtract
SUBSUU_S.*df*	*V* subsuu_s_*df*(*V*,*V*)		Signed Saturated Unsigned Subtract (both arguments are unsigned, the result is signed)
SUBSUS_U.*df*	*V* subsus_u_*df*(*V*,*V*)		Unsigned Saturated Signed Subtract from Unsigned (the first argument is unsigned, the second is signed, and the result is unsigned)
SUBV.*df*	*V* subv_*df*(*V*,*V*)	*v* - *v*	Subtract
SUBVI.*df*	*V* subvi_*df*(*V*,*K*)	*v* - *k*	Subtract Immediate (immediate value is unsigned)

1. *df* – supported data format abbreviation, see Table 1.
2. *V* – vector type of integer elements (signed or unsigned based on the instruction's semantics)
3. *v* – vector variable of type *V*
4. *K* – integer constant (signed or unsigned based on the instruction's semantics) type
5. *k* – 5-bit constant of type *K*
6. *W* – vector type of integer elements (signed or unsigned based on the instruction's semantics) half the size of the elements in *V*

Table 7 MSA Bitwise Instructions

Mnemonic	Compiler Intrinsic	C Expression	Instruction Description
AND.V	*V*[1] and_v(*V*,*V*)	*v*[2] & *v*	Logical And
ANDI.B	*V* andi_b(*V*,*K*[3])	*v* & *k*[4]	Logical And Immediate
BCLR.*df*[5]	*V* bclr_*df*(*V*,*V*)		Bit Clear
BCLRI.*df*	*V* bclri_*df*(*V*,*K*)		Bit Clear Immediate
BINSL.*df*	*V* binsl_*df*(*V*,*V*)		Insert Left of Bit Position
BINSLI.*df*	*V* binsli_*df*(*V*,*K*)		Insert Left of Immediate Bit Position
BINSR.*df*	*V* binsr_*df*(*V*,*V*)		Insert Right of Bit Position
BINSRI.*df*	*V* binsri_*df*(*V*,*K*)		Insert Right of Immediate Bit Position
BMNZ.V	*V* bmnz_v(*V*,*V*,*V*)		Bit Move If Not Zero
BMNZI.B	*V* bmnzi_b(*V*,*V*,*K*)		Bit Move If Not Zero Immediate
BMZ.V	*V* bmz_v(*V*,*V*,*V*)		Bit Move If Zero
BMZI.B	*V* bmzi_b(*V*,*V*,*K*)		Bit Move If Zero Immediate
BNEG.*df*	*V* bneg_*df*(*V*,*V*)		Bit Negate
BNEGI.*df*	*V* bnegi_*df*(*V*,*K*)		Bit Negate Immediate
BSEL.V	*V* bsel_v(*V*,*V*)		Bit Select
BSELI.B	*V* bseli_b(*V*,*K*)		Bit Select Immediate
BSET.*df*	*V* bset_*df*(*V*,*V*)		Bit Set
BSETI.*df*	*V* bseti_*df*(*V*,*K*)		Bit Set Immediate
NLOC.*df*	*V* nloc_*df*(*V*,*V*)		Leading One Bits Count
NLZC.*df*	*V* nlzc_*df*(*V*,*V*)		Leading Zero Bits Count
NOR.V	*V* nor_v(*V*,*V*)		Logical Negated Or
NORI.B	*V* nori_b(*V*,*K*)		Logical Negated Or Immediate
PCNT.*df*	*V* pcnt_*df*(*V*,*V*)		Population (Bits Set to 1) Count
OR.V	*V* or_v(*V*,*V*)	*v* | *v*	Logical Or
ORI.B	*V* ori_b(*V*,*K*)	*v* | *k*	Logical Or Immediate
XOR.V	*V* xor_v(*V*,*V*)	*v* ^ *v*	Logical Or
XORI.B	*V* xori_b(*V*,*K*)	*v* ^ *k*	Logical Or Immediate
SLL.*df*	*V* sll_*df*(*V*,*V*)	*v* << *v*	Shift Left
SLLI.*df*	*V* slli_*df*(*V*,*K*)	*v* << *k*	Shift Left Immediate
SRA.*df*	*V* sra_*df*(*V*,*V*)	*v* >> *v*	Shift Right Arithmetic

Table 7 MSA Bitwise Instructions (Continued)

Mnemonic	Compiler Intrinsic	C Expression	Instruction Description
SRAI.*df*	`V srai_df(V,K)`	`v >> k`	Shift Right Arithmetic Immediate
SRAR.*df*	`V srar_df(V,V)`		Shift Right Arithmetic with Rounding
SRARI.*df*	`V srari_df(V,K)`		Shift Right Arithmetic with Rounding Immediate
SRL.*df*	`V srl_df(V,V)`	`v >> v`	Shift Right
SRLI.*df*	`V srli_df(V,K)`	`v >> k`	Shift Right Immediate
SRLR.*df*	`V srlr_df(V,V)`		Shift Right with Rounding
SRLRI.*df*	`V srlri_df(V,K)`		Shift Right with Rounding Immediate

1. `V` – vector type of integer elements
2. `v` – vector variable of type `V`
3. `K` – integer constant type suitable for the instruction's semantics
4. `k` – constant of type `K`
5. *df* – supported data format abbreviation, see Table 1.

6.2 Floating-Point Instructions

The MSA floating-point implementation is compliant with the IEEE Standard for Floating-Point Arithmetic 754™-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 set destination bits to zero (false) or one (true). The floating-point specific unordered relations are supported by dedicated quiet compare unordered instructions 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.

Table 8 MSA Floating-Point Arithmetic Instructions

Mnemonic	Compiler Intrinsic	C Expression	Instruction Description
FADD.*df*[1]	`F`[2] `fadd_df(F,F)`	`f`[3] `+ f`	Floating-Point Addition
FDIV.*df*	`F fdiv_df(F,F)`	`f / f`	Floating-Point Division
FEXP2.*df*	`F fexp2_df(F,V`[4]`)`		Floating-Point Base 2 Exponentiation
FLOG2.*df*	`F flog2_df(F,F)`		Floating-Point Base 2 Logarithm

Table 8 MSA Floating-Point Arithmetic Instructions (Continued)

Mnemonic	Compiler Intrinsic	C Expression	Instruction Description
FMADD.*df*	`F fmadd_df(F,F)`	`f + f * f`	Floating-Point Fused Multiply-Add
FMSUB.*df*	`F fmsub_df(F,F)`	`f - f * f`	Floating-Point Fused Multiply-Subtract
FMAX.*df*	`F fmax_df(F,F)`		Floating-Point Maximum
FMIN.*df*	`F fmin_df(F,F)`		Floating-Point Minimum
FMAX_A.*df*	`F fmax_a_df(F,F)`		Floating-Point Maximum of Absolute Values
FMIN_A.*df*	`F fmin_a_df(F,F)`		Floating-Point Minimum of Absolute Values
FMUL.*df*	`F fmul_df(F,F)`	`f * f`	Floating-Point Multiplication
FRCP.*df*	`F frcp_df(F,F)`		Approximate Floating-Point Reciprocal
FRINT.*df*	`F frint_df(F,F)`		Floating-Point Round to Integer
FRSQRT.*df*	`F frsqrt_df(F,F)`		Approximate Floating-Point Reciprocal of Square Root
FSQRT.*df*	`F fsqrt_df(F,F)`		Floating-Point Square Root
FSUB.*df*	`F fsub_df(F,F)`	`f - f`	Floating-Point Subtraction

1. *df* – supported data format abbreviation, see Table 1.
2. `F` – vector type of floating-point elements
3. `f` – vector variable of type `F`
4. `V` – vector type of signed integer elements

Table 9 MSA Floating-Point Compare Instructions

Mnemonic	Compiler Intrinsic	C Expression	Instruction Description
FCLASS.*df*[1]	`V`[2] `fclass_df(F`[3]`)`		Floating-Point Class Mask
FCAF.*df*	`V fcaf_df(F,F)`		Floating-Point Quiet Compare Always False
FCUN.*df*	`V fcun_df(F,F)`		Floating-Point Quiet Compare Unordered
FCOR.*df*	`V fcor_df(F,F)`		Floating-Point Quiet Compare Ordered
FCEQ.*df*	`V fceq_df(F,F)`		Floating-Point Quiet Compare Equal
FCUNE.*df*	`V fcune_df(F,F)`		Floating-Point Quiet Compare Unordered or Not Equal
FCUEQ.*df*	`V fcueq_df(F,F)`		Floating-Point Quiet Compare Unordered or Equal
FCNE.*df*	`V fcne_df(F,F)`		Floating-Point Quiet Compare Not Equal
FCLT.*df*	`V fclt_df(F,F)`		Floating-Point Quiet Compare Less Than
FCULT.*df*	`V fcult_df(F,F)`		Floating-Point Quiet Compare Unordered or Less Than
FCLE.*df*	`V fcle_df(F,F)`		Floating-Point Quiet Compare Less Than or Equal

Table 9 MSA Floating-Point Compare Instructions (Continued)

Mnemonic	Compiler Intrinsic	C Expression	Instruction Description
FCULE.*df*	*V* fcule_*df*(*F*,*F*)		Floating-Point Quiet Compare Unordered or Less Than or Equal
FSAF.*df*	*V* fsaf_*df*(*F*,*F*)		Floating-Point Signaling Compare Always False
FSUN.*df*	*V* fsun_*df*(*F*,*F*)		Floating-Point Signaling Compare Unordered
FSOR.*df*	*V* fsor_*df*(*F*,*F*)		Floating-Point Signaling Compare Ordered
FSEQ.*df*	*V* fseq_*df*(*F*,*F*)	*f*[4] == *f*	Floating-Point Signaling Compare Equal
FSUNE.*df*	*V* fsune_*df*(*F*,*F*)		Floating-Point Signaling Compare Unordered or Not Equal
FSUEQ.*df*	*V* fsueq_*df*(*F*,*F*)		Floating-Point Signaling Compare Unordered or Equal
FSNE.*df*	*V* fsne_*df*(*F*,*F*)	*f* != *f*	Floating-Point Signaling Compare Not Equal
FSLT.*df*	*V* fslt_*df*(*F*,*F*)	*f* < *f*	Floating-Point Signaling Compare Less Than
FSULT.*df*	*V* fsult_*df*(*F*,*F*)		Floating-Point Signaling Compare Unordered or Less Than
FSLE.*df*	*V* fsle_*df*(*F*,*F*)	*f* <= *f*	Floating-Point Signaling Compare Less Than or Equal
FSULE.*df*	*V* fsule_*df*(*F*,*F*)		Floating-Point Signaling Compare Unordered or Less Than or Equal

1. *df* – supported data format abbreviation, see Table 1.
2. *V* – vector type of integer elements
3. *F* – vector type of floating-point elements
4. *f* – vector variable of type *F*

Table 10 MSA Floating-Point Conversion Instructions

Mnemonic	Compiler Intrinsic	C Expression	Instruction Description
FEXUPL.*df*[1]	*F*[2] fexupl_*df*(*G*[3])		Left-Half Floating-Point Format Up-Convert
FEXUPR.*df*	*F* fexupr_*df*(*G*)		Right-Half Floating-Point Format Up-Convert
FEXDO.*df*	*G* fexdo_*df*(*F*,*F*)		Floating-Point Format Down-Convert
FFINT_S.*df*	*F* ffint_s_*df*(*V*[4])		Floating-Point Convert from Signed Integer
FFINT_U.*df*	*F* ffint_u_*df*(*V*)		Floating-Point Convert from Unsigned Integer
FFQL.*df*	*F* ffql_*df*(*W*[5])		Left-Half Floating-Point Convert from Fixed-Point
FFQR.*df*	*F* ffqr_*df*(*W*)		Right-Half Floating-Point Convert from Fixed-Point
FTINT_S.*df*	*V* ftint_s_*df*(*V*,*V*)		Floating-Point Round and Convert to Signed Integer

Table 10 MSA Floating-Point Conversion Instructions (Continued)

Mnemonic	Compiler Intrinsic	C Expression	Instruction Description
FTINT_U.*df*	*V* ftint_u_*df*(*V*,*V*)		Floating-Point Round and Convert to Unsigned Integer
FTRUNC_S.*df*	*V* ftrunc_s_*df*(*F*)		Floating-Point Truncate and Convert to Signed Integer
FTRUNC_U.*df*	*V* ftrunc_u_*df*(*F*)		Floating-Point Truncate and Convert to Unsigned Integer
FTQ.*df*	*W* ftq_*df*(*F*,*F*)		Floating-Point Round and Convert to Fixed-Point

1. *df* – supported data format abbreviation, see Table 1.
2. *F* – vector type of floating-point elements
3. *G* – vector type of floating-point elements half the size of the elements in *F*
4. *V* – vector type of integer or fixed-point (based on the instruction's semantics) elements the same size as the elements in *F*
5. *W* – vector type of integer or fixed-point (based on the instruction's semantics) elements half the size of the elements in *F*

6.3 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 with optional rounding (Table 11).

Table 11 MSA Fixed-Point Instructions

Mnemonic	Compiler Intrinsic	C Expression	Instruction Description
MADD_Q.*df*[1]	*V*[2] madd_q_*df*(*V*,*V*,*V*)		Fixed-Point Multiply and Add
MADDR_Q.*df*	*V* maddr_q_*df*(*V*,*V*,*V*)		Fixed-Point Multiply and Add with Rounding
MSUB_Q.*df*	*V* msub_q_*df*(*V*,*V*,*V*)		Fixed-Point Multiply and Subtract
MSUBR_Q.*df*	*V* msubr_q_*df*(*V*,*V*,*V*)		Fixed-Point Multiply and Subtract with Rounding
MUL_Q.*df*	*V* mul_q_*df*(*V*,*V*)		Fixed-Point Multiply
MULR_Q.*df*	*V* mulr_q_*df*(*V*,*V*)		Fixed-Point Multiply with Rounding

1. *df* – supported data format abbreviation, see Table 1.
2. *V* – vector type of fixed-point elements

6.4 Branch and Compare Instructions

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

The compare instructions (Table 12) 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.

Table 12 MSA Compare Instructions

Mnemonic	Compiler Intrinsic	C Expression	Instruction Description
CEQ.*df*[1]	`V`[2] `ceq_df(V,V)`	`v`[3] `== v`	Compare Equal
CEQI.*df*	`V ceqi_df(V,K`[4]`)`	`v == k`[5]	Compare Equal Immediate
CLE_S.*df*	`V cle_s_df(V,V)`	`v <= v`	Compare Less-Than-or-Equal Signed
CLEI_S.*df*	`V clei_s_df(V,K)`	`v <= k`	Compare Less-Than-or-Equal Signed Immediate
CLE_U.*df*	`V cle_u_df(V,V)`	`v <= v`	Compare Less-Than-or-Equal Unsigned
CLEI_U.*df*	`V clei_u_df(V,K)`	`v <= k`	Compare Less-Than-or-Equal Unsigned Immediate
CLT_S.*df*	`V clt_s_df(V,V)`	`v < v`	Compare Less-Than Signed
CLTI_S.*df*	`V clti_s_df(V,K)`	`v < k`	Compare Less-Than Signed Immediate
CLT_U.*df*	`V clt_u_df(V,V)`	`v < v`	Compare Less-Than Unsigned
CLTI_U.*df*	`V clti_u_df(V,K)`	`v < k`	Compare Less-Than Unsigned Immediate

1. *df* – supported data format abbreviation, see Table 1.
2. `V` – vector type of integer elements (signed or unsigned based on the instruction's semantics)
3. `v` – vector variable of type `V`
4. `K` – integer constant (signed or unsigned based on the instruction's semantics) type
5. `k` – 5-bit constant of type `K`

Both branch-on-false and branch-on-true condition instructions are provided (Table 13) 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. There are no intrinsics for control flow statements.

Table 13 MSA Branch Instructions

Mnemonic	Instruction Description
BNZ.V	Branch If Not Zero (at least one bit is not zero)

Table 13 MSA Branch Instructions (Continued)

Mnemonic	Instruction Description
BZ.V	Branch If Zero (all bits are zero)
BNZ.*df*[1]	Branch If Not Zero (all elements are not zero)
BZ.*df*	Branch If Zero (at least one element is zero)

1. *df* – supported data format abbreviation, see Table 1.

Based on the branch-on-false and branch-on-true instructions, the intrinsics in Table 14 assign the truth value of vector conditions to scalars. For example the C statement,

```
int n = test_bnz_h(v)
```

is compiled to 3 assembly instructions:

```
    bnz.h $w0,1f
    li    $2,1
    li    $2,0
1:
```

This sequence sets GPR $2 (scalar `n` in C) to 1 if the condition tested by BNZ.H is true, i.e. the code branches if all halfword elements in vector register $w0 (vector `v` in C) are not zero.

Table 14 MSA Scalar Branch Condition Intrinsics

Compiler Intrinsic	C Expression	Intrinsic Description
N[1] `test_bnz_v(`*V*[2]`)`		Test Branch Not Zero condition: return 1 if at least one bit is not zero, otherwise return 0
N `test_bz_v(`*V*`)`		Test Branch Zero condition: return 1 if all bits are zero, otherwise return 0
N `test_bnz_`*df*[3]`(`*V*`)`		Test Branch Not Zero condition: return 1 if all elements are not zero, otherwise return 0
N `test_bz_`*df*`(`*V*`)`		Test Branch Zero condition: return 1 if at least one element is zero, otherwise return 0

1. *N* – scalar integer type
2. *V* – vector type of integer, floating-point, or fixed-point elements
3. *df* – supported data format abbreviation, see Table 1.

The scalar branch condition intrinsics are legal in `if()` statements, where the assignment of 1 or 0 will often be eliminated leaving just the corresponding BNZ/BZ instruction.

6.5 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 15) 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 do not 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 half-words 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.

Table 15 MSA Load/Store and Move Instructions

Mnemonic	Compiler Intrinsic	C Expression	Instruction Description
CFCMSA	*N*[1] `cfcmsa(`*K*[2]`)`		Copy from MSA Control Register
CTCMSA	`void ctcmsa(`*N*`,`*K*`)`		Copy to MSA Control Register
LD.*df*[3]	*V*[4] `ld_`*df*`(*`*V*`)`	*v*[5] `= *`*pv*[6]	Load Vector
LDI.*df*	*V* `ldi_`*df*`(`*K*`)`	*v* `= (`*V*`){`*k*[7]`,…,`*k*`}`	Load Immediate
MOVE.V	*V* `move_v(`*V*`)`	*v* `=` *v*	Vector to Vector Move
SPLAT.*df*	*V* `splat_`*df*`(`*V*`,`*N*`)`	*v* `= (`*V*`){`*v*`[`*n*[8]`],…,`*v*`[`*n*`]}`	Replicate Vector Element
SPLATI.*df*	*V* `splati_`*df*`(`*V*`,`*K*`)`	*v* `= (`*V*`){`*v*`[`*k*`],…,`*v*`[`*k*`]}`	Replicate Vector Element Immediate
FILL.*df*	*V* `fill_`*df*`(`*N*`)`	*v* `= (`*V*`){`*n*`,…,`*n*`}`	Fill Vector from GPR
INSERT.*df*	*V* `insert_`*df*`(`*V*`,`*K*`,`*N*`)`	*v*`[`*k*`] =` *n*	Insert GPR to Vector Element
INSVE.*df*	*V* `insve_`*df*`(`*V*`,`*K*`,`*V*`)`	*v*`[`*k*`] =` *v*`[0]`	Insert Vector element 0 to Vector Element
COPY_S.*df*	*N* `copy_s_`*df*`(`*V*`,`*K*`)`	*n* `=` *v*`[`*k*`]`	Copy element to GPR Signed
COPY_U.*df*	*N* `copy_u_`*df*`(`*V*`,`*K*`)`	*n* `=` *v*`[`*k*`]`	Copy element to GPR Unsigned
ST.*df*	*V* `st_`*df*`(*`*V*`,`*V*`)`	`*`*pv* `=` *v*	Store Vector

1. *N* – scalar integer type
2. *K* – integer constant type suitable for the instruction's semantics
3. *df* – supported data format abbreviation, see Table 1.
4. *V* – vector type of integer, floating-point, or fixed-point elements
5. *v* – vector variables of type *V*
6. *pv* – pointer to a vector of type *V*
7. *k* – integer constant of type *K*
8. *n* – integer variable of type *N*

6.6 Element Permute 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 16). 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.

Note that the MSA VSHF instruction is semantically compatible with the architecture independent GCC intrinsic `__builtin_shuffle()`.

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.

Table 16 MSA Element Permute Instructions

Mnemonic	Compiler Intrinsic	C Expression	Instruction Description
ILVEV.*df*[1]	*V*[2] `ilvev_`*df*`(`*V*`,`*V*`)`		Interleave Even
ILVOD.*df*	*V* `ilvod_`*df*`(`*V*`,`*V*`)`		Interleave Odd
ILVL.*df*	*V* `ilvl_`*df*`(`*V*`,`*V*`)`		Interleave Left
ILVR.*df*	*V* `ilvr_`*df*`(`*V*`,`*V*`)`		Interleave Right
PCKEV.*df*	*V* `pckev_`*df*`(`*V*`,`*V*`)`		Pack Even Elements
PCKOD.*df*	*V* `pckod_`*df*`(`*V*`,`*V*`)`		Pack Odd Elements
SHF.*df*	*V* `shf_`*df*`(`*V*`,`*K*[3]`)`		Set Shuffle
SLD.*df*	*V* `sld_`*df*`(`*V*`,`*N*`)`		Element Slide
SLDI.*df*	*V* `sldi_`*df*`(`*V*`,`*K*`)`		Element Slide Immediate
VSHF.*df*	*V* `vshf_`*df*`(`*V*`,`*V*`,`*V*`)`		Vector shuffle

1. *df* – supported data format abbreviation, see Table 1.
2. *V* – vector type of integer, floating-point, or fixed-point elements
3. *K* – integer constant type suitable for the instruction's semantics

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 and multithreading 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.

Template: nW1.03, Built with tags: 2B ARCH FPU_PS FPU_PSandARCH MIPS32