RISC-V Matrix Specification

Version 0.5a, 08/2024: This document is in development.

Table of Contents

Preamble . . . 1
Copyright and license information . . . 2
Contributors . . . 3
1. Introduction . . . 4
2. Implementation-defined Constant Parameters. . . . 5
3. Programmer’s Model . . . 6
    3.1. Matrix Tile Registers . . . 6
    3.2. Matrix Type Register, mtype . . . 7
    3.3. Matrix Tile Size Registers, mtilem/mtilek/mtilen . . . 9
    3.4. Matrix Start Index Register, mstart . . . 9
    3.5. Matrix Control and Status Register, mcsr . . . 9
    3.6. Matrix Context Status in mstatus and sstatus . . . 10
4. Instructions . . . 11
    4.1. Instruction Formats . . . 11
    4.2. Configuration-Setting Instructions . . . 12
        4.2.1. mtype Encoding . . . 14
        4.2.2. ATM/ATK/ATN Encoding . . . 15
        4.2.3. Constraints on Setting mtilem/mtilek/mtilen . . . 16
    4.3. Load and Store Instructions . . . 17
        4.3.1. Load Instructions . . . 17
        4.3.2. Store Instructions . . . 18
        4.3.3. Whole Matrix Load & Store Instructions . . . 19
    4.4. Data Move Instructions . . . 20
        4.4.1. Data Move Instructions between Matrix Registers . . . 20
        4.4.2. Data Move Instructions between Matrix and Integer . . . 21
        4.4.3. Data Move Instructions between Matrix and Float-point . . . 22
        4.4.4. Data Broadcast Instructions . . . 23
        4.4.5. Matrix Transpose Instructions. . . . 24
    4.5. Arithmetic and Logic Instructions . . . 24
        4.5.1. Matrix Multiplication Instructions . . . 24
        4.5.2. Element-Wise Instructions . . . 26
    4.6. Type-Convert Instructions . . . 28
5. Intrinsic Examples . . . 32
    5.1. Matrix multiplication . . . 32
    5.2. Matrix multiplication with left matrix transposed . . . 32
    5.3. Matrix transpose without multiplication . . . 33
6. Standard Matrix Extensions . . . 34

6.1. Zma*b*: Matrix Mode Extension . . . 34
6.2. Zmi4: Matrix 4-bit Integer Extension . . . 34
6.3. Zmi8: Matrix 8-bit Integer Extension . . . 36
6.4. Zmi16: Matrix 16-bit Integer Extension . . . 38
6.5. Zmi32: Matrix 32-bit Integer Extension . . . 40
6.6. Zmi64: Matrix 64-bit Integer Extension . . . 42
6.7. Zmf8e4m3: Matrix 8-bit E4M3 Float Point Extension . . . 44
6.8. Zmf8e5m2: Matrix 8-bit E5M2 Float Point Extension . . . 46
6.9. Zmf8e3m4: Matrix 8-bit E3M4 Float Point Extension . . . 48
6.10. Zmf16e5m10: Matrix 16-bit Half-precision Float-point (FP16) Extension . . . 49
6.11. Zmf16e8m7: Matrix 16-bit BFloat (BF16) Extension . . . 51
6.12. Zmf32e8m23: Matrix 32-bit Float-point Extension . . . 53
6.13. Zmf19e8m10: Matrix 19-bit TensorFloat32 (TF32) Extension . . . 54
6.14. Zmf64e11m52: Matrix 64-bit Float-point Extension . . . 56
6.15. Zmv: Matrix for Vector operations . . . 57
6.15.1. Load Instructions . . . 58
6.15.2. Store Instructions . . . 59
6.15.3. Data Move Instructions . . . 59
6.15.4. Intrinsic Example: Matrix multiplication fused with element-wise vector operation . . . 61
6.16. Zmi2c: Im2col Extension . . . 62
6.16.1. CSRs . . . 62
6.16.2. Configuration Instructions . . . 63
6.16.3. Load Unfold Instructions . . . 63
6.16.4. Intrinsic Example: Conv2D . . . 64
6.16.5. Intrinsic Example: Conv3D . . . 65
6.16.6. Intrinsic Example: MaxPool2D . . . 66
6.16.7. Intrinsic Example: AvgPool2D . . . 67
6.17. Zmc2i: Col2im Extension . . . 68
6.17.1. CSRs . . . 68
6.17.2. Configuration Instructions . . . 68
6.17.3. Store Fold Instructions . . . 68
6.18. Zmsp*: Matrix Sparsity Extension . . . 69
6.18.1. Configuration Instructions . . . 69
6.18.2. Matrix Multiplication Instructions . . . 70
7. Matrix Instruction Listing . . . 72

Preamble

This document is in the Development state

Assume everything can change. This draft specification will change before being accepted as standard, so implementations made to this draft specification will likely not conform to the future standard.

Copyright and license information

Contributors

This RISC-V specification has been contributed to directly or indirectly by:

- Chaoqun Wang
- Chen Chen
- Fujie Fan
- Hui Yao
- Jing Qiu
- Kening Zhang
- Kun Hu
- Mingxin Zhao
- Wengan Shao
- Wenmeng Zhang
- Xianmiao Qu
- Xiaoyan Xiang
- Xin Ouyang
- Xin Yang
- Yunhai Shang
- Zhao Jiang
- Zhiqiang Liu
- Zhiwei liu
- Zhiyong Zhang
- **Your name here**

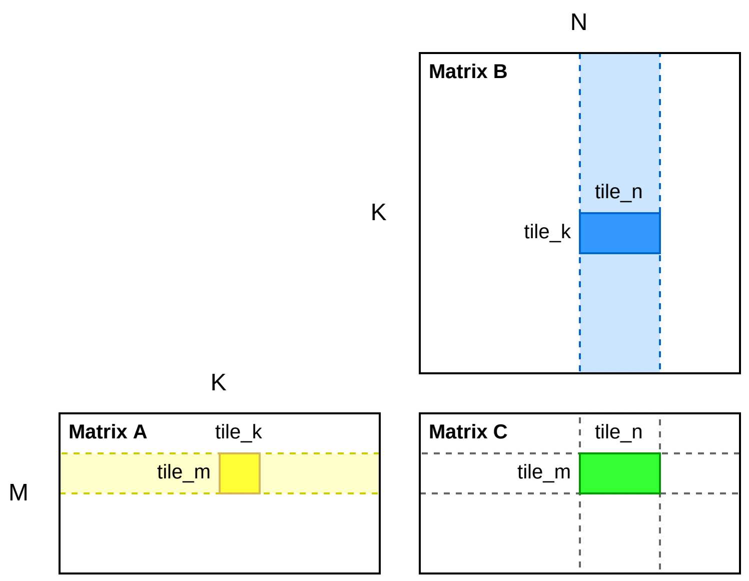
We will be very grateful to the huge number of other people who will have helped to improve this specification through their comments, reviews, feedback and questions.

Chapter 1. Introduction

This document describes the matrix extension for RISC-V.

Matrix extension implement matrix multiplications by partitioning the input and output matrix into tiles, which are then stored to matrix registers.

Tile size usually refers to the dimensions of these tiles. For the operation C = AB in figure below, the tile size of C is mtilem × mtilen, the tile size of A is mtilem × mtilek and the tile size of B is mtilek × mtilen.



Each matrix multiplication instruction computes its output tile by stepping through the K dimension in tiles, loading the required values from the A and B matrices, and multiplying and accumulating them into the output.

Matrix extension is strongly inspired by the RISC-V Vector "V" extension.

Chapter 2. Implementation-defined Constant Parameters

Each hart supporting a matrix extension defines four parameters:

1. The maximum size in bits of a matrix element that any operation can produce or consume, $\mathrm{ELEN} \geq 8$, which must be a power of 2.
2. The number of bits in a single matrix tile register, MLEN, which must be a power of 2, and must be no greater than 2^{32}.
3. The number of bits in a row of a single matrix tile register, RLEN, which must be a power of 2, and must be no greater than 2^{16}.
4. The multiple of length for matrix accumulation registers, AMUL, where the number of bits in a row of a single matrix accumulation register is $\mathrm{RLEN} \times \mathrm{AMUL}$, and the number of bits in a single matrix accumulation register is $\mathrm{MLEN} \times \mathrm{AMUL}$.

Some constraints on these parameters are defined as following.

1. $\mathrm{ELEN} \leq \mathrm{RLEN} \leq \mathrm{MLEN}$, this supports matrix tile size from 1×1 to $2^{16} \times 2^{16}$.
2. For implmentations without widening accumulation space, $\mathrm{AMUL} = 1$.
3. For implmentations with double-widening accumulation space, $\mathrm{AMUL} = 2$.
4. For implmentations with quadruple-widening accumulation space, $\mathrm{AMUL} = 4$.
5. For implmentations with octuple-widening accumulation space, $\mathrm{AMUL} = 8$.
6. AMUL with any other value is not allowed.

Chapter 3. Programmer’s Model

The matrix extension adds 9 unprivileged CSRs and 16 matrix registers to the base scalar RISC-V ISA.

Table 1. Matrix CSRs

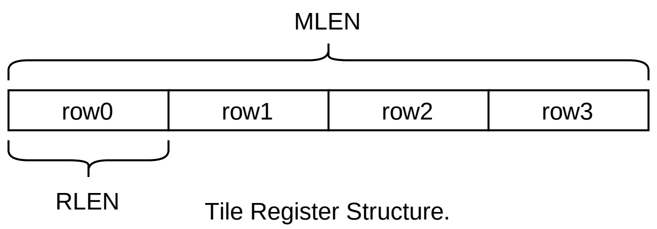
Address	Privilege	Name	Description
0xC40	URO	mtype	Matrix tile data type register.
0xC41	URO	mtilem	Tile length in m direction.
0xC42	URO	mtilen	Tile length in n direction.
0xC43	URO	mtilek	Tile length in k direction.
0xC44	URO	mlenb	MLEN/8 (matrix tile register length in bytes).
0xC45	URO	mrlenb	RLEN/8 (matrix tile register row length in bytes).
0xC46	URO	mamul	AMUL.
0x040	URW	mstart	Start element index.
0x041	URW	mcsr	Matrix control and status register.

3.1. Matrix Tile Registers

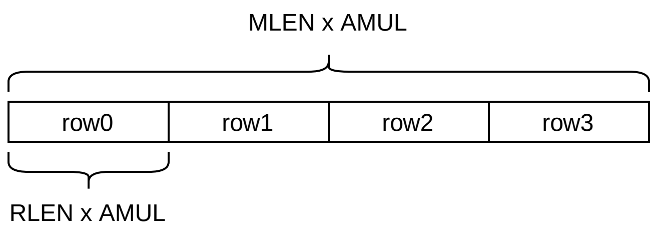
The matrix extension adds 8 architectural **Tile Registers** (tr0-tr7) for input tile matrices and 8 architectural **Accumulation Registers** (acc0-acc7) for output accumulation matrices.

A **Tile Register** has a fixed MLEN bits of state, where each row has RLEN bits. As a result, there are MLEN/RLEN rows for each tile register in logic.

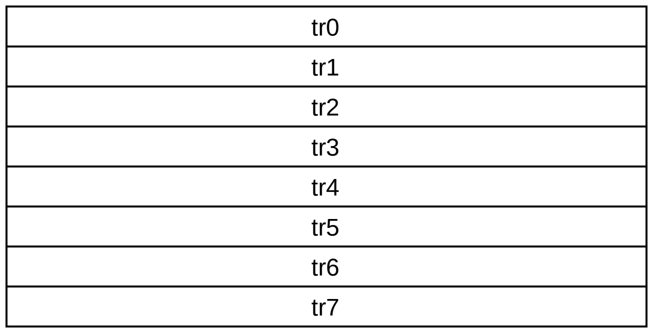
An **Accumulation Register** has a fixed $\mathrm{MLEN} \times \mathrm{AMUL}$ bits of state, where each row has $\mathrm{RLEN} \times \mathrm{AMUL}$ bits. As a result, there are MLEN/RLEN rows for each accumulation register in logic.



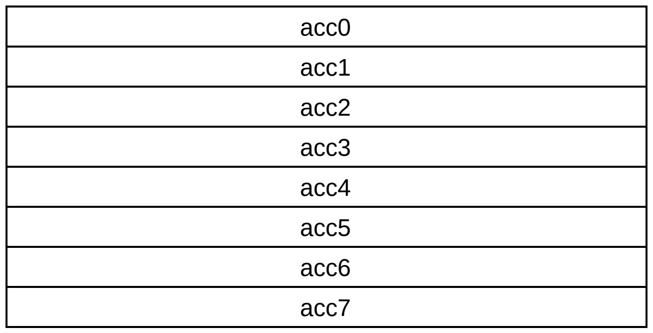
Tile Register Structure.



Accumulation Register Structure.



Tile Register File.



Accumulation Register File.

An input matrix of matrix multiplication instruction only uses one tile register, and large matrix must be split according to the size of tile defined by MLEN and RLEN.

For widening instructions, each output element is wider than input one. To match the width of input and output, an output matrix may be written back to a wider accumulation register whose length are specified by MLEN x AMUL.

3.2. Matrix Type Register, mtype

The read-only XLEN-wide *matrix type* CSR, `mtype`, provides the default type used to interpret the contents of the matrix register file, and can only be updated by `msettype{i|hi}` and field-set instructions. The matrix type determines the organization of elements in each matrix register.

> Allowing updates only via type-set or field-set instructions simplifies the maintenance of `mtype` register state.

The `mtype` register has an `mill` field, an `msew` field, an `mba` field and several type fields. Bits `mtype[XLEN-2:16]` should be written with zero, and non-zero values of this field are reserved.

Table 2. `mtype` register layout

Bits	Name	Description
XLEN-1	mill	Illegal value if set.
XLEN-2:16	0	Reserved if non-zero.
15	mba	Matrix out of bound agnostic.
14	mfp64	64-bit float point enabling.
13:12	mfp32[1:0]	32-bit float point enabling.
11:10	mfp16[1:0]	16-bit float point enabling.
9:8	mfp8[1:0]	8-bit float point enabling.
7	mint64	64-bit integer enabling.
6	mint32	32-bit integer enabling.
5	mint16	16-bit integer enabling.
4	mint8	8-bit integer enabling.
3	mint4	4-bit integer enabling.
2:0	msew[2:0]	Selected element width (SEW) setting.

The `msew` field is used to specify the element width of source operands. It is used to calculate the maximum values of matrix size.

For each type field, a value 0 means the corresponding type is disabled. Write non-zero value to enable matrix multiplication operation of the specified type. 0 will be returned and `mill` will be set if the type is not supported.

For `mint4` field, write 1 to enable 4-bit integer where a 8-bit integer will be treated as a pair of 4-bit integers. 1’b0 will be returned if 4-bit integer is not supported.

For `mint8` field, write 1 to enable 8-bit integer.

For `mint16` field, write 1 to enable 16-bit integer.

For `mint64` field, write 1 to enable 64-bit integer.

For `mfp8` field, write 2’b01 to enable E4M3, 2’b10 to enable E5M2, and 2’b11 to enable E3M4. `mfp8[1:0]` always returns 2’b00 if FP8 is not supported.

For `mfp16` field, write 2’b01 to enable IEEE-754 half-precision float point (E5M10), and write 2’b10 to enable BFloat16 (E8M7). 2’b11 is reserved.

For `mfp32` field, write 2’b01 to enalbe IEEE-754 single-precison float point (E8M23), and write 2’b10 to enable TensorFloat32 (E8M10). 2’b11 is reserved.

For `mfp64` field, write 1 to enable 64-bit double-precision float point. To support FP64 format, the implementation should support "D" extension at the same time. 0 will be returned if FP64 is not supported.

The `mba` field indicates that the out-of-bound elements is undisturbed or agnostic. When `mba` is marked undisturbed (`mba=0`), the out-of-bound elements in a matrix register retain the value it previously held. Otherwise, the out-of-bound elements can be overwritten with any values.

3.3. Matrix Tile Size Registers, mtilem/mtilek/mtilen

The XLEN-bit-wide read-only `mtilem/mtilek/mtilen` CSRs can only be updated by the `msettile{m|k|n}{i}` instructions. The registers holds 3 unsigned integers specifying the tile shapes for tiled matrix.

3.4. Matrix Start Index Register, mstart

The `mstart` read-write CSR specifies the index of the first element to be executed by load/store and element-wise arithmetic instructions. The CSR can be written by hardware on a trap, and its value represents the element on which the trap was taken. The value is the sequential number in row order.

Any legal matrix instruction can reset the `mstart` to zero at the end of excution.

3.5. Matrix Control and Status Register, mcsr

The `mcsr` register has 2 fields, and other bits with non-zero value are reserved.

Table 3. `mcsr` register layout

Bits	Name	Description
XLEN-1:3	0	Reserved if non-zero.
2:1	mmode[1:0]	The mode of matrix multiplication.
0	msat	Integer arithmetic instruction accrued saturation flag.

`mmode` field indicates the mode of matrix multiplication. `mmode = 00` means `C = A x B`, where the source matrices, `A` and `B`, are both organized as the original order. `mmode = 01` means `C = A x BT`, where `B` is transposed. `mmode = 10` means `C = AT x B`, where `A` is transposed.

An implementation can support any combination of these modes, with extensions Zmab, Zmabt and

Zmatb.

If an unsupported `mmode` is set, then any attempt to execute a matrix multiplication instruction will raise an illegal instruction exception.

3.6. Matrix Context Status in mstatus and sstatus

A 2-bit matrix context status field should be added to mstatus and shadowed in status. It is defined analogously to the vector context status field, VS.

Chapter 4. Instructions

4.1. Instruction Formats

The instructions in the matrix extension use 32-bit encoding and a new major opcode OP-M32 (1110111).

Instruction formats are listed below.

Configuration instructions, where the `imm` field supports 10-bit immediate operand.

31 26	25	24 15	14 12	11 7	6 0
funct6	im	imm10	funct3	rd	OP-M32

Configuration Immediate Instructions.

31 26	25	24 20	19 15	14 12	11 7	6 0
funct6	im	00000	rs1	funct3	rd	OP-M32

Configuration Instructions.

Load & store instructions, where `ls` field indicates the type (load or store), and `tr` field indicates if the operand in register is transposed. `eew` field (000-011) indicates the effective element width.

31 26	25	24 20	19 15	14 12	11	10 7	6 0
funct6	ls	rs2	rs1	eew	tr	ms3/td	OP-M32

Load/Store Instructions.

Data move instructions, where `di` field indicates the moving direction.

31 26	25	24 20	19 15	14 12	11 7	6 0
funct6	di	rs2/funct5	rs1/ms1	eew	rd/md	OP-M32

Data Move Instructions.

Arithmetic and logic instructions, where `fp` field indicates if the operation is float point, `sa` field indicates if the result is saturated, and `sn` field indicates if the source operands are signed (for integer). `eew` field indicates the effective element width (000-011 for int8-int64, 111 for int4, and 100 to use `mtype.msew`).

31 26	25	24	20	19	18 15	14 12	11	10 7	6 0
funct6	fp	sa	ms2	sn	ms1	eew	ma	md	OP-M32

Arithmetic & Logic Instructions.

Type-convert instructions, where `fd` field indicates if the destination elements are float point, and `sn` field indicates if the integer operand is signed. `eew` field indicates the effective element width (000-011 for int8-int64, 111 for int4, and 100 to use `mtype.msew`). `nw` field indicates if the destination elements are narrowed or widen.

31 26	25	24 22	21 20	19	18 15	14 12	11	10 7	6 0
funct6	fd	funct3	nw	sn	ms1	eew	0	md	OP-M32

Type-convert Instructions.

4.2. Configuration-Setting Instructions

Due to hardware resource constraints, one of the common ways to handle large-sized matrix multiplication is "tiling", where each iteration of the loop processes a subset of elements, and then continues to iterate until all elements are processed. The Matrix extension provides direct, portable support for this approach.

The block processing of matrix multiplication requires three levels of loops to iterate in the direction of the number of rows of the left matrix (m), the number of columns of the left matrix (k, also the number of rows of the right matrix), and the number of columns of the right matrix (n), given by the application.

The shapes of the matrix tiles to be processed, m (application tile length m or `ATM`), k (`ATK`), n (`ATN`), is used as candidates for `mtilem` / `mtilek` / `mtilen`. Based on microarchitecture implementation and `mmode` setting, hardware returns a new `mtilem` / `mtilek` / `mtilen` value via a general purpose register (usually smaller), also stored in `mtilem` / `mtilek` / `mtilen` CSR, which is the shape of tile per iteration handled by hardware.

For a simple matrix multiplication example, check out the Section Intrinsic Example, which describes how the code keeps track of the matrices processed by the hardware each iteration.

A set of instructions is provided to allow rapid configuration of the values in `mtile*` and `mtype` to match application needs.

The `msettype[i|hi]` instructions set the `mtype` CSR based on their arguments, and write the new value of mtype into rd.

```
msettypei  rd, imm      # rd = new mtype, imm = new mtype[9:0] setting.
msettypehi rd, imm      # rd = new mtype, imm = new mtype[19:10] setting.
msettype   rd, rs1      # rd = new mtype, rs1 = new mtype value.
```

The `mset*` instructions set the specified field of `mtype` without affecting other fields.

```
# Set msew field.
msetsew rd, imm         # rd = new mtype, set msew to imm[2:0].
msetsew rd, e8          # rd = new mtype, imm = 0.
msetsew rd, e16         # rd = new mtype, imm = 1.
msetsew rd, e32         # rd = new mtype, imm = 2.
msetsew rd, e64         # rd = new mtype, imm = 3.

# Set mba field.
```

```
msetba  rd, imm         # rd = new mtype, set mba to imm[0].
msetba  rd, bu          # rd = new mtype, imm = 0.
msetba  rd, ba          # rd = new mtype, imm = 1.

# Set integer type fields.
msetint rd, int4        # rd = new mtype, set mint4 = 1 to enable INT4 type.
msetint rd, int8        # rd = new mtype, set mint8 = 1 to enable INT8 type.
msetint rd, int16       # rd = new mtype, set mint16 = 1 to enable INT16 type.
msetint rd, int32       # rd = new mtype, set mint32 = 1 to enable INT32 type.
msetint rd, int64       # rd = new mtype, set mint64 = 1 to enable INT64 type.

# Set float point type fields.
msetfp  rd, e4m3        # rd = new mtype, set mfp8 = 01 to enable FP8 E4M3 type.
msetfp  rd, e5m2        # rd = new mtype, set mfp8 = 10 to enable FP8 E5M2 type.
msetfp  rd, e3m4        # rd = new mtype, set mfp8 = 11 to enable FP8 E3M4 type.
msetfp  rd, fp16        # rd = new mtype, set mfp16 = 01 to enable FP16 E5M10 type.
msetfp  rd, bf16        # rd = new mtype, set mfp16 = 10 to enable BF16 E8M7 type.
msetfp  rd, fp32        # rd = new mtype, set mfp32 = 01 to enable FP32 E8M23 type.
msetfp  rd, tf32        # rd = new mtype, set mfp32 = 10 to enable TF32 E8M10 type.
msetfp  rd, fp64        # rd = new mtype, set mfp64 = 1 to enable FP64 type.
```

The `munset*` instructions unset the specified field of `mtype` without affecting other fields.

```
munsetint rd, int4      # rd = new mtype, set mint4 = 0 to disable INT4 type.
munsetint rd, int8      # rd = new mtype, set mint8 = 0 to disable INT8 type.
munsetint rd, int16     # rd = new mtype, set mint16 = 0 to disable INT16 type.
munsetint rd, int32     # rd = new mtype, set mint32 = 0 to disable INT32 type.
munsetint rd, int64     # rd = new mtype, set mint64 = 0 to disable INT64 type.

munsetfp  rd, fp8       # rd = new mtype, set mfp8 = 00 to disable FP8 type.
munsetfp  rd, fp16      # rd = new mtype, set mfp16 = 00 to disable FP16 type.
munsetfp  rd, fp32      # rd = new mtype, set mfp32 = 00 to disable FP32 type.
munsetfp  rd, fp64      # rd = new mtype, set mfp64 = 0 to disable FP64 type.
```

The field to be set or unset is specified by inst[18:15] and the value is specified by inst[24:20].

Table 4. Field to be set or unset

inst[18:15]	**field**
0000	msew
0001	mint4
0010	mint8
0011	mint16
0100	mint32

0101	mint64
0110	mfp8
0111	mfp16
1000	mfp32
1001	mfp64
1010	mba

The `msettile{m|k|n}[i]` instructions set the mtilem/mtilek/mtilen CSRs based on their arguments, and write the new value into rd.

```
msettilemi rd, imm         # rd = new mtilem, imm = ATM
msettilem  rd, rs1         # rd = new mtilem, rs1 = ATM
msettileki rd, imm         # rd = new mtilek, imm = ATN
msettilek  rd, rs1         # rd = new mtilek, rs1 = ATN
msettileni rd, imm         # rd = new mtilen, imm = ATK
msettilen  rd, rs1         # rd = new mtilen, rs1 = ATK
```

4.2.1. mtype Encoding

Table 5. `mtype` register layout

Bits	Name	Description
XLEN-1	mill	Illegal value if set.
XLEN-2:16	0	Reserved if non-zero.
15	mba	Matrix out of bound agnostic.
14	mfp64	64-bit float point enabling.
13:12	mfp32[1:0]	32-bit float point enabling.
11:10	mfp16[1:0]	16-bit float point enabling.
9:8	mfp8[1:0]	8-bit float point enabling.
7	mint64	64-bit integer enabling.
6	mint32	32-bit integer enabling.
5	mint16	16-bit integer enabling.
4	mint8	8-bit integer enabling.
3	mint4	4-bit integer enabling.

Bits	Name	Description
2:0	msew[2:0]	Selected element width (SEW) setting.

The new `mtype` value is encoded in the immediate fields of `msettypei` / `msettypehi`, and in the `rs1` register for `msettype`. Each field can be set or unset with `msetsew`, `msetba`, `msetfp`, `msetint`, `munsetfp` and `munsetint` instructions independently.

4.2.2. ATM/ATK/ATN Encoding

There are three values, `TMMAX`, `TKMAX` and `TNMAX`, represent the maximum shapes of the matrix tiles that could be stored in matrix registers, and can be operated on with a single matrix instruction given the current SEW settings.

The values of `TMMAX`, `TKMAX` and `TNMAX` are related to MLEN, RLEN and the configuration of `mmode`.

For `A x B` mode (`mmode=00`),

- TMMAX = MLEN / RLEN
- TKMAX = min(MLEN / RLEN, RLEN / SEW)
- TNMAX = RLEN / SEW

For `A x BT` mode (`mmode=01`),

- TMMAX = MLEN / RLEN
- TKMAX = RLEN / SEW
- TNMAX = MLEN / RLEN

For `AT x B` mode (`mmode=10`),

- TMMAX = min(MLEN / RLEN, RLEN / SEW)
- TKMAX = MLEN / RLEN
- TNMAX = RLEN / SEW

For examples, with `MLEN=256`, `RLEN=64` and `mmode=00`, `TMMAX`, `TKMAX` and `TNMAX` values are shown below.

```
SEW=8,  TMMAX=4, TKMAX=4, TNMAX=8       # 4x4x8 8-bit matmul
SEW=16, TMMAX=4, TKMAX=4, TNMAX=4       # 4x4x4 16-bit matmul
SEW=32, TMMAX=4, TKMAX=2, TNMAX=2       # 4x2x2 32-bit matmul
```

The new tile shape settings are based on `ATM` / `ATK` / `ATN` values, which for `msettile{m|k|n}` is encoded in the rs1 and rd fields.

rd	rs1	`ATM/ATK/ATN` value	Effect on `mtilem/mtilek/mtilen`
-	!x0	Value in `x[rs1]`	Normal tiling
!x0	x0	~0	Set `mtilem/mtilek/mtilen` to `TMMAX/TKMAX/TNMAX`
x0	x0	Value in `mtilem/mtilek/mtilen`	Keep existing `mtilem/mtilek/mtilen` if less than `TMMAX/TKMAX/TNMAX`

For the `msettile{m|k|n}i` instructions, the `ATM` / `ATK` / `ATN` is encoded as a 10-bit unsigned immediate in the rs1.

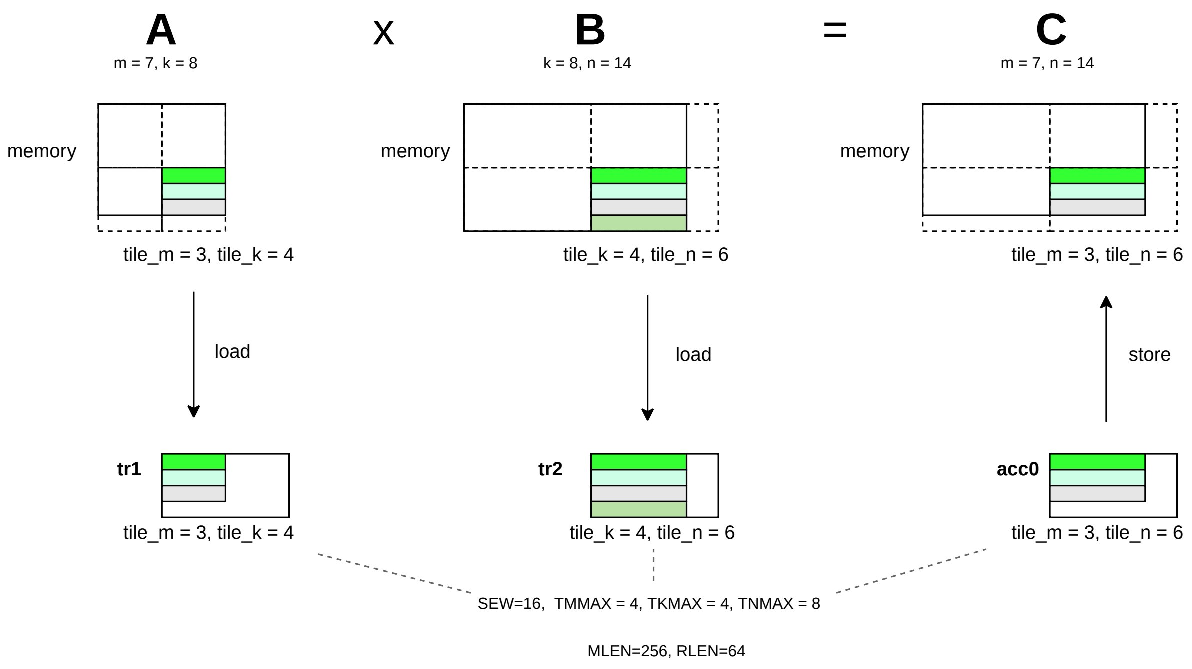
4.2.3. Constraints on Setting mtilem/mtilek/mtilen

The `msettile{m|k|n}[i]` instructions first set `TMMAX/TKMAX/TNMAX` according to the mtype CSR, then set `mtilem/mtilek/mtilen` obeying the following constraints (using `mtilem` & `ATM` & `TMMAX` as an example, and the same with `mtilek` & `ATK` & `TKMAX` and `mtilen` & `ATN` & `TNMAX`):

1. `mtilem = ATM` if `ATM <= TMMAX`
2. `ceil(ATM / 2) <= mtilem <= TMMAX` if `ATM < (2 * TMMAX)`
3. `mtilem = TMMAX` if `ATM >= (2 * TMMAX)`
4. Deterministic on any given implementation for same input ATM and TMMAX values
5. These specific properties follow from the prior rules:
    a. `mtilem = 0` if `ATM = 0`
    b. `mtilem > 0` if `ATM > 0`
    c. `mtilem <= TMMAX`
    d. `mtilem <= ATM`
    e. a value read from `mtilem` when used as the ATM argument to `msettile{m|k|n}{i}` results in the same value in `mtilem`, provided the resultant TMMAX equals the value of TMMAX at the time that `mtilem` was read.

Continue to use `MLEN=256`, `RLEN=64` and `mmode=00` as a example. When SEW=16, TMMAX=4, TKMAX=4, TNMAX=8.

If A is a 7 x 8 matrix and B is a 8 x 14 matrix, we could get `mtilem/mtilek/mtilen` values as show below, in the last loop of tiling.



4.3. Load and Store Instructions

4.3.1. Load Instructions

Load a matrix tile from memory.

```
# md destination, rs1 base address, rs2 row byte stride

# For left matrix, A
# tile size = mtilem * mtilek
mlae8.m  md, (rs1), rs2        #  8-bit left tile load
mlae16.m md, (rs1), rs2        # 16-bit left tile load
mlae32.m md, (rs1), rs2        # 32-bit left tile load
mlae64.m md, (rs1), rs2        # 64-bit left tile load

# For right matrix, B
# tile size = mtilek * mtilen
mlbe8.m  md, (rs1), rs2        #  8-bit right tile load
mlbe16.m md, (rs1), rs2        # 16-bit right tile load
mlbe32.m md, (rs1), rs2        # 32-bit right tile load
mlbe64.m md, (rs1), rs2        # 64-bit right tile load

# For output matrix, C
# tile size = mtilem * mtilen
mlce8.m  md, (rs1), rs2        #  8-bit output tile load
mlce16.m md, (rs1), rs2        # 16-bit output tile load
mlce32.m md, (rs1), rs2        # 32-bit output tile load
```

```
mlce64.m md, (rs1), rs2         # 64-bit output tile load
```

Load a matrix tile from memory, where the matrix on memory is transposed.

```
# md destination, rs1 base address, rs2 row byte stride

# For left matrix, A
# tile size = mtilek * mtilem
mlate8.m  md, (rs1), rs2        #  8-bit left tile load
mlate16.m md, (rs1), rs2        # 16-bit left tile load
mlate32.m md, (rs1), rs2        # 32-bit left tile load
mlate64.m md, (rs1), rs2        # 64-bit left tile load

# For right matrix, B
# tile size = mtilen * mtilek
mlbte8.m  md, (rs1), rs2        #  8-bit right tile load
mlbte16.m md, (rs1), rs2        # 16-bit right tile load
mlbte32.m md, (rs1), rs2        # 32-bit right tile load
mlbte64.m md, (rs1), rs2        # 64-bit right tile load

# For output matrix, C
# tile size = mtilen * mtilem
mlcte8.m  md, (rs1), rs2        #  8-bit output tile load
mlcte16.m md, (rs1), rs2        # 16-bit output tile load
mlcte32.m md, (rs1), rs2        # 32-bit output tile load
mlcte64.m md, (rs1), rs2        # 64-bit output tile load
```

4.3.2. Store Instructions

Store a matrix tile to memory.

```
# ms3 store data, rs1 base address, rs2 row byte stride

# For left matrix, A
# tile size = mtilem * mtilek
msae8.m  ms3, (rs1), rs2        #  8-bit left tile store
msae16.m ms3, (rs1), rs2        # 16-bit left tile store
msae32.m ms3, (rs1), rs2        # 32-bit left tile store
msae64.m ms3, (rs1), rs2        # 64-bit left tile store

# For right matrix, B
# tile size = mtilek * mtilen
msbe8.m  ms3, (rs1), rs2        #  8-bit right tile store
msbe16.m ms3, (rs1), rs2        # 16-bit right tile store
msbe32.m ms3, (rs1), rs2        # 32-bit right tile store
msbe64.m ms3, (rs1), rs2        # 64-bit right tile store
```

```
# For output matrix, C
# tile size = mtilem * mtilen
msce8.m  ms3, (rs1), rs2       #  8-bit output tile store
msce16.m ms3, (rs1), rs2       # 16-bit output tile store
msce32.m ms3, (rs1), rs2       # 32-bit output tile store
msce64.m ms3, (rs1), rs2       # 64-bit output tile store
```

Save a matrix tile to memory, where the matrix on memory is transposed.

```
# ms3 store data, rs1 base address, rs2 row byte stride

# For left matrix, A
# tile size = mtilek * mtilem
msate8.m  ms3, (rs1), rs2      #  8-bit left tile store
msate16.m ms3, (rs1), rs2      # 16-bit left tile store
msate32.m ms3, (rs1), rs2      # 32-bit left tile store
msate64.m ms3, (rs1), rs2      # 64-bit left tile store

# For right matrix, B
# tile size = mtilen * mtilek
msbte8.m  ms3, (rs1), rs2      #  8-bit right tile store
msbte16.m ms3, (rs1), rs2      # 16-bit right tile store
msbte32.m ms3, (rs1), rs2      # 32-bit right tile store
msbte64.m ms3, (rs1), rs2      # 64-bit right tile store

# For output matrix, C
# tile size = mtilen * mtilem
mscte8.m  ms3, (rs1), rs2      #  8-bit output tile store
mscte16.m ms3, (rs1), rs2      # 16-bit output tile store
mscte32.m ms3, (rs1), rs2      # 32-bit output tile store
mscte64.m ms3, (rs1), rs2      # 64-bit output tile store
```

4.3.3. Whole Matrix Load & Store Instructions

Load a whole tile matrix from memory without considering the size.

```
mltre8.m   md, (rs1), rs2      #  8-bit whole matrix load
mltre16.m  md, (rs1), rs2      # 16-bit whole matrix load
mltre32.m  md, (rs1), rs2      # 32-bit whole matrix load
mltre64.m  md, (rs1), rs2      # 64-bit whole matrix load
```

Load a whole accumulation matrix from memory without considering the size.

```
mlacce8.m  md, (rs1), rs2      #  8-bit whole matrix load
```

```
mlacce16.m md, (rs1), rs2       # 16-bit whole matrix load
mlacce32.m md, (rs1), rs2       # 32-bit whole matrix load
mlacce64.m md, (rs1), rs2       # 64-bit whole matrix load
```

Store a whole tile matrix to memory without considering the size.

```
mstre8.m   ms3, (rs1), rs2      #  8-bit whole matrix store
mstre16.m  ms3, (rs1), rs2      # 16-bit whole matrix store
mstre32.m  ms3, (rs1), rs2      # 32-bit whole matrix store
mstre64.m  ms3, (rs1), rs2      # 64-bit whole matrix store
```

Store a whole accumulation matrix to memory without considering the size.

```
msacce8.m  ms3, (rs1), rs2      #  8-bit whole matrix store
msacce16.m ms3, (rs1), rs2      # 16-bit whole matrix store
msacce32.m ms3, (rs1), rs2      # 32-bit whole matrix store
msacce64.m ms3, (rs1), rs2      # 64-bit whole matrix store
```

Whole matrix load and store instructions are usually used for context saving and restoring.

4.4. Data Move Instructions

4.4.1. Data Move Instructions between Matrix Registers

Data move instructions between matrix registers are used to move elements between two tile registers, two accumulation registers, or one tile register and one accumulation register.

```
# md = ms1, md and ms1 are both tile registers.
mmve8.t.t   md, ms1
mmve16.t.t  md, ms1
mmve32.t.t  md, ms1
mmve64.t.t  md, ms1

# md = ms1, md and ms1 are both accumulation registers.
mmve8.a.a   md, ms1
mmve16.a.a  md, ms1
mmve32.a.a  md, ms1
mmve64.a.a  md, ms1

# md[i, rs2 * (RLEN / EEW) + j] = ms1[i, j]
# md is accumulation register and ms1 is tile register.
mmve8.a.t   md, ms1, rs2
mmve16.a.t  md, ms1, rs2
```

```
mmve32.a.t  md, ms1, rs2
mmve64.a.t  md, ms1, rs2

# md[i, j] = ms1[i, rs2 * (RLEN / EEW) + j]
# md is tile register and ms1 is accumulation register.
mmve8.t.a   md, ms1, rs2
mmve16.t.a  md, ms1, rs2
mmve32.t.a  md, ms1, rs2
mmve64.t.a  md, ms1, rs2

# md[i, imm * (RLEN / EEW) + j] = ms1[i, j]
# md is an accumulation register and ms1 is a tile register.
mmvie8.a.t  md, ms1, imm
mmvie16.a.t md, ms1, imm
mmvie32.a.t md, ms1, imm
mmvie64.a.t md, ms1, imm

# md[i, j] = ms1[i, imm * (RLEN / EEW) + j]
# md is a tile register and ms1 is an accumulation register.
mmvie8.t.a  md, ms1, imm
mmvie16.t.a md, ms1, imm
mmvie32.t.a md, ms1, imm
mmvie64.t.a md, ms1, imm
```

4.4.2. Data Move Instructions between Matrix and Integer

Data move instructions between matrix and integer are used to move single element between integer registers and tile registers. Such instructions can change a part of matrix and often used for debug.

```
# x[rd] = ms1[i, j], i = rs2[15:0], j = rs2[XLEN-1:16]
# rd is an integer register and ms1 is a tile register.
mmve8.x.t  rd, ms1, rs2
mmve16.x.t rd, ms1, rs2
mmve32.x.t rd, ms1, rs2
mmve64.x.t rd, ms1, rs2

# md[i, j] = x[rs1], i = rs2[15:0], j = rs2[XLEN-1:16]
# md is a tile register and rs1 is an integer register.
mmve8.t.x  md, rs1, rs2
mmve16.t.x md, rs1, rs2
mmve32.t.x md, rs1, rs2
mmve64.t.x md, rs1, rs2

# x[rd] = ms1[i, j], i = rs2[15:0], j = rs2[XLEN-1:16]
# rd is an integer register and ms1 is an accumulation register.
mmve8.x.a  rd, ms1, rs2
mmve16.x.a rd, ms1, rs2
```

```
mmve32.x.a rd, ms1, rs2
mmve64.x.a rd, ms1, rs2

# md[i, j] = x[rs1], i = rs2[15:0], j = rs2[XLEN-1:16]
# md is an accumulation register and rs1 is an integer register.
mmve8.a.x  md, rs1, rs2
mmve16.a.x md, rs1, rs2
mmve32.a.x md, rs1, rs2
mmve64.a.x md, rs1, rs2
```

The `mmve*.x.t/a` instruction copies a signle SEW-wide element of the matrix register to an integer register, where the element coordinates are specified by rs2. If SEW > XLEN, the least-significat XLEN bits are transferred. If SEW < XLEN, the valud is sign-extended to XLEN bits.

The `mmve*.t/a.x` instruction copies an integer register to an element of the destination matrix register, where the element coordinates are specified by rs2. If SEW < XLEN, the least-significant bits are moved and the upper (XLEN-SEW) bits are ignored. If SEW > XLEN, the valud is sign-extended to SEW bits. The other elements of the tile register are treated as out-of-bound elements, using the setting of `mtype.mba`.

4.4.3. Data Move Instructions between Matrix and Float-point

Float point data move instructions are similar with integer.

```
# f[rd] = ms1[i, j], i = rs2[15:0], j = rs2[XLEN-1:16]
# rd is a float-point register and ms1 is a tile register.
mfmve8.f.t  rd, ms1, rs2
mfmve16.f.t rd, ms1, rs2
mfmve32.f.t rd, ms1, rs2
mfmve64.f.t rd, ms1, rs2

# md[i, j] = f[rs1], i = rs2[15:0], j = rs2[XLEN-1:16]
# md is a tile register and rs1 is a float-point register.
mfmve8.t.f  md, rs1, rs2
mfmve16.t.f md, rs1, rs2
mfmve32.t.f md, rs1, rs2
mfmve64.t.f md, rs1, rs2

# f[rd] = ms1[i, j], i = rs2[15:0], j = rs2[XLEN-1:16]
# rd is a float-point register and ms1 is an accumulation register.
mfmve8.f.a  rd, ms1, rs2
mfmve16.f.a rd, ms1, rs2
mfmve32.f.a rd, ms1, rs2
mfmve64.f.a rd, ms1, rs2

# md[i, j] = f[rs1], i = rs2[15:0], j = rs2[XLEN-1:16]
# md is an accumulation register and rs1 is a float-point register.
```

```
mfmve8.a.f  md, rs1, rs2
mfmve16.a.f md, rs1, rs2
mfmve32.a.f md, rs1, rs2
mfmve64.a.f md, rs1, rs2
```

4.4.4. Data Broadcast Instructions

The first row/column and the first element of a matrix register can be broadcasted to fill the whole matrix.

```
# Broadcast the first row of a tile register to fill the whole matrix.
mbcar.m md, ms1
mbcbr.m md, ms1

# Broadcast the first row of an accumulation register to fill the whole matrix.
mbccr.m md, ms1

# Broadcast the first column of a tile register to fill the whole matrix.
mbcace8.m  md, ms1
mbcace16.m md, ms1
mbcace32.m md, ms1
mbcace64.m md, ms1

mbcbce8.m  md, ms1
mbcbce16.m md, ms1
mbcbce32.m md, ms1
mbcbce64.m md, ms1

# Broadcast the first column of an accumulation register to fill the whole matrix.
mbccce8.m  md, ms1
mbccce16.m md, ms1
mbccce32.m md, ms1
mbccce64.m md, ms1

# Broadcast the first element of a tile register to fill the whole matrix.
mbcaee8.m  md, ms1
mbcaee16.m md, ms1
mbcaee32.m md, ms1
mbcaee64.m md, ms1

mbcbee8.m  md, ms1
mbcbee16.m md, ms1
mbcbee32.m md, ms1
mbcbee64.m md, ms1

# Broadcast the first element of an accumulation register to fill the whole matrix.
mbccee8.m  md, ms1
```

```
mbccee16.m md, ms1
mbccee32.m md, ms1
mbccee64.m md, ms1
```

4.4.5. Matrix Transpose Instructions

Transpose instruction can only be used for square matrix. For matrix A, the sizes of two dimensions are both min(`mtilem`, `mtilek`). Matrix B and C are similar.

```
# Transpose square matrix of tile register..
mtae8.m  md, ms1
mtae16.m md, ms1
mtae32.m md, ms1
mtae64.m md, ms1

mtbe8.m  md, ms1
mtbe16.m md, ms1
mtbe32.m md, ms1
mtbe64.m md, ms1

# Transpose square matrix of accumulation register.
mtce8.m  md, ms1
mtce16.m md, ms1
mtce32.m md, ms1
mtce64.m md, ms1
```

4.5. Arithmetic and Logic Instructions

4.5.1. Matrix Multiplication Instructions

Matrix Multiplication operations take two matrix tiles from matrix **tile registers** specified by `ms1` and `ms2` respectively, and the output matrix tile is a matrix **accumulation register** specified by `md`.

```
# Unigned integer matrix multiplication and add, md = md + ms1 * ms2.
mmau.[dw].mm    md, ms1, ms2        # uint64, output no-widen
mmau.[w].mm     md, ms1, ms2        # uint32, output no-widen
mmau.[h].mm     md, ms1, ms2        # uint16, output no-widen
mwmau.[w].mm    md, ms1, ms2        # uint32, output double-widen
mwmau.[h].mm    md, ms1, ms2        # uint16, output double-widen
mqmau.[b].mm    md, ms1, ms2        # uint8, output quad-widen
momau.[hb].mm   md, ms1, ms2        # uint4, output oct-widen

msmau.[dw].mm   md, ms1, ms2        # uint64, output no-widen and saturated
msmau.[w].mm    md, ms1, ms2        # uint32, output no-widen and saturated
msmau.[h].mm    md, ms1, ms2        # uint16, output no-widen and saturated
```

```
mswmau.[w].mm   md, ms1, ms2        # uint32, output double-widen and saturated
mswmau.[h].mm   md, ms1, ms2        # uint16, output double-widen and saturated
msqmau.[b].mm   md, ms1, ms2        # uint8, output quad-widen and saturated
msomau.[hb].mm  md, ms1, ms2        # uint4, output oct-widen and saturated

# Signed integer matrix multiplication and add, md = md + ms1 * ms2.
mma.[dw].mm     md, ms1, ms2        # int64, output no-widen
mma.[w].mm      md, ms1, ms2        # int32, output no-widen
mma.[h].mm      md, ms1, ms2        # int16, output no-widen
mwma.[w].mm     md, ms1, ms2        # int32, output double-widen
mwma.[h].mm     md, ms1, ms2        # int16, output double-widen
mqma.[b].mm     md, ms1, ms2        # int8, output quad-widen
moma.[hb].mm    md, ms1, ms2        # int4, output oct-widen

msma.[dw].mm    md, ms1, ms2        # int64, output no-widen and saturated
msma.[w].mm     md, ms1, ms2        # int32, output no-widen and saturated
msma.[h].mm     md, ms1, ms2        # int16, output no-widen and saturated
mswma.[w].mm    md, ms1, ms2        # int32, output double-widen and saturated
mswma.[h].mm    md, ms1, ms2        # int16, output double-widen and saturated
msqma.[b].mm    md, ms1, ms2        # int8, output quad-widen and saturated
msoma.[hb].mm   md, ms1, ms2        # int4, output oct-widen and saturated

# Float point matrix multiplication and add, md = md + ms1 * ms2.
mfma.[d].mm     md, ms1, ms2        # 64-bit float point
mfma.[f].mm     md, ms1, ms2        # 32-bit float point
mfma.[hf].mm    md, ms1, ms2        # 16-bit float point

mfwma.[f].mm    md, ms1, ms2        # 32-bit float point, output double-widen
mfwma.[hf].mm   md, ms1, ms2        # 16-bit float point, output double-widen
mfwma.[cf].mm   md, ms1, ms2        # 8-bit float point, output double-widen
mfqma.[cf].mm   md, ms1, ms2        # 8-bit float point, output quad-widen
```

A subset of these instructions is supported according to the implemented standard extensions (Zmi4, Zmi8, etc.).

The field `frm` from `fcsr` indicates the rounding mode of float-point matrix instructions. The encoding is shown below.

frm	Mnemonic	Meaning
000	RNE	Round to Nearest, ties to Even
001	RTZ	Round towards Zero
010	RDN	Round Down (towards $-\infty$)
011	RUP	Round Up (towards $+\infty$)
100	RMM	Round to Nearest, ties to Max Magnitude

101		Invalid
110		Invalid
111		Invalid

4.5.2. Element-Wise Instructions

Matrix element-wise add/sub/multiply instructions. The input and output matrices are both accumulation registers and always with size `mtilem x mtilen`. The element-wise calculation of tile registers can be implemented by combining data move instructions (such as `mmve*.a.t` and `mmve*.t.a`).

```
# Unsigned integer matrix element-wise add.
# md[i,j] = ms1[i,j] + ms2[i,j]
maddu.[hb|b|h|w|dw].mm   md, ms1, ms2
msaddu.[hb|b|h|w|dw].mm  md, ms1, ms2  # output saturated
mwaddu.[hb|b|h|w].mm     md, ms1, ms2  # output double widen

# Signed integer matrix element-wise add.
# md[i,j] = ms1[i,j] + ms2[i,j]
madd.[hb|b|h|w|dw].mm    md, ms1, ms2
msadd.[hb|b|h|w|dw].mm   md, ms1, ms2  # output saturated
mwadd.[hb|b|h|w].mm      md, ms1, ms2  # output double widen

# Unsigned integer matrix element-wise subtract.
# md[i,j] = ms1[i,j] - ms2[i,j]
msubu.[hb|b|h|w|dw].mm   md, ms1, ms2
mssubu.[hb|b|h|w|dw].mm  md, ms1, ms2  # output saturated
mwsubu.[hb|b|h|w].mm     md, ms1, ms2  # output double widen

# Signed integer matrix element-wise subtract.
# md[i,j] = ms1[i,j] - ms2[i,j]
msub.[hb|b|h|w|dw].mm    md, ms1, ms2
mssub.[hb|b|h|w|dw].mm   md, ms1, ms2  # output saturated
mwsub.[hb|b|h|w].mm      md, ms1, ms2  # output double widen

# Integer matrix element-wise minimum.
# md[i,j] = min{ms1[i,j], ms2[i,j]}
mminu.[hb|b|h|w|dw].mm   md, ms1, ms2
mmin.[hb|b|h|w|dw].mm    md, ms1, ms2

# Integer matrix element-wise maximum.
# md[i,j] = max{ms1[i,j], ms2[i,j]}
mmaxu.[hb|b|h|w|dw].mm   md, ms1, ms2
mmax.[hb|b|h|w|dw].mm    md, ms1, ms2

# Integer matrix bit-wise logic.
```

```
mand.mm                  md, ms1, ms2
mor.mm                   md, ms1, ms2
mxor.mm                  md, ms1, ms2

# Integer matrix element-wise shift.
msll.[hb|b|h|w|dw].mm    md, ms1, ms2
msrl.[hb|b|h|w|dw].mm    md, ms1, ms2
msra.[hb|b|h|w|dw].mm    md, ms1, ms2

# Integer matrix element-wise multiply.
# md[i,j] = ms1[i,j] * ms2[i,j]
mmul.[hb|b|h|w|dw].mm    md, ms1, ms2  # signed, returning low bits of product
mmulh.[hb|b|h|w|dw].mm   md, ms1, ms2  # signed, returning high bits of product
mmulhu.[hb|b|h|w|dw].mm  md, ms1, ms2  # unsigned, returning high bits of product
mmulhsu.[hb|b|h|w|dw].mm md, ms1, ms2  # signed-unsigned, returning high bits of
product

# Saturated integer matrix element-wise multiply.
msmul.[hb|b|h|w|dw].mm   md, ms1, ms2  # signed
msmulu.[hb|b|h|w|dw].mm  md, ms1, ms2  # unsigned
msmulsu.[hb|b|h|w|dw].mm md, ms1, ms2  # signed-unsigned

# Widening integer matrix element-wise multiply.
mwmul.[hb|b|h|w].mm      md, ms1, ms2  # signed
mwmulu.[hb|b|h|w].mm     md, ms1, ms2  # unsigned
mwmulsu.[hb|b|h|w].mm    md, ms1, ms2  # signed-unsigned

# Float matrix element-wise add.
# md[i,j] = ms1[i,j] + ms2[i,j]
mfadd.[cf|hf|f|d].mm     md, ms1, ms2
mfwadd.[cf|hf|f].mm      md, ms1, ms2  # output double widen

# Float matrix element-wise subtract.
# md[i,j] = ms1[i,j] - ms2[i,j]
mfsub.[cf|hf|f|d].mm     md, ms1, ms2
mfwsub.[cf|hf|f].mm      md, ms1, ms2  # output double widen

# Float matrix element-wise minimum.
# md[i,j] = min{ms1[i,j], ms2[i,j]}
mfmin.[cf|hf|f|d].mm     md, ms1, ms2

# Float matrix element-wise maximum.
# md[i,j] = max{ms1[i,j], ms2[i,j]}
mfmax.[cf|hf|f|d].mm     md, ms1, ms2

# Float matrix element-wise multiply.
# md[i,j] = ms1[i,j] * ms2[i,j]
mfmul.[cf|hf|f|d].mm     md, ms1, ms2
mfwmul.[cf|hf|f].mm      md, ms1, ms2  # output double widen
```

```
# Float matrix element-wise divide.
# md[i,j] = ms1[i,j] / ms2[i,j]
mfdiv.[cf|hf|f|d].mm     md, ms1, ms2

# Float matrix element-wise square root.
# md[i,j] = ms1[i,j] ^ (1/2)
mfsqrt.[cf|hf|f|d].m     md, ms1
```

There is no matrix-scalar and matrix-vector version for element-wise instructions. Such operations can be replaced by a broadcast instruction and a matrix-matrix element-wise instruction.

4.6. Type-Convert Instructions

The input and output matrices of type-convert instructions are both accumulation registers and always with size `mtilem x mtilen`. The type convert of tile registers can be implemented by combining data move instructions (such as `mmve*.a.t` and `mmve*.t.a`).

```
# Convert integer to integer
mcvt.x.xu.m     md, ms1         # uint to int
mcvt.hb.uhb.m   md, ms1         # uint4 to int4
mcvt.b.ub.m     md, ms1         # uint8 to int8
mcvt.h.uh.m     md, ms1         # uint16 to int16
mcvt.w.uw.m     md, ms1         # uint32 to int32
mcvt.dw.udw.m   md, ms1         # uint64 to int64

mcvt.xu.x.m     md, ms1         # int to uint
mcvt.uhb.hb.m   md, ms1         # int4 to uint4
mcvt.ub.b.m     md, ms1         # int8 to uint8
mcvt.uh.h.m     md, ms1         # int16 to uint16
mcvt.uw.w.m     md, ms1         # int32 to uint32
mcvt.udw.dw.m   md, ms1         # int64 to uint64

mwcvtu.xw.x.m   md, ms1         # uint to double-width uint
mwcvtu.xq.x.m   md, ms1         # uint to quad-width uint
mwcvtu.xo.x.m   md, ms1         # uint to oct-width uint
mwcvtu.b.hb.m   md, ms1         # uint4 to uint8
mwcvtu.h.hb.m   md, ms1         # uint4 to uint16
mwcvtu.w.hb.m   md, ms1         # uint4 to uint32
mwcvtu.h.b.m    md, ms1         # uint8 to uint16
mwcvtu.w.b.m    md, ms1         # uint8 to uint32
mwcvtu.w.h.m    md, ms1         # uint16 to uint32
mwcvtu.dw.w.m   md, ms1         # uint32 to uint64

mwcvt.xw.x.m    md, ms1         # int to double-width int
mwcvt.xq.x.m    md, ms1         # int to quad-width int
mwcvt.xo.x.m    md, ms1         # int to oct-width int
mwcvt.b.hb.m    md, ms1         # int4 to int8
mwcvt.h.hb.m    md, ms1         # int4 to int16
mwcvt.w.hb.m    md, ms1         # int4 to int32
mwcvt.h.b.m     md, ms1         # int8 to int16
mwcvt.w.b.m     md, ms1         # int8 to int32
mwcvt.w.h.m     md, ms1         # int16 to int32
mwcvt.dw.w.m    md, ms1         # int32 to int64

mncvtu.x.xw.m   md, ms1         # double-width uint to single-width uint
mncvtu.x.xq.m   md, ms1         # quad-width uint to single-width uint
mncvtu.x.xo.m   md, ms1         # oct-width uint to single-width uint
mncvtu.hb.b.m   md, ms1         # uint8 to uint4
mncvtu.hb.h.m   md, ms1         # uint16 to uint4
mncvtu.hb.w.m   md, ms1         # uint32 to uint4
mncvtu.b.h.m    md, ms1         # uint16 to uint8
mncvtu.b.w.m    md, ms1         # uint32 to uint8
mncvtu.h.w.m    md, ms1         # uint32 to uint16
mncvtu.w.dw.m   md, ms1         # uint64 to uint32

mncvt.x.xw.m    md, ms1         # double-width int to single-width int
mncvt.x.xq.m    md, ms1         # quad-width int to single-width int
mncvt.x.xo.m    md, ms1         # oct-width int to single-width int
mncvt.hb.b.m    md, ms1         # int8 to int4
mncvt.hb.h.m    md, ms1         # int16 to int4
mncvt.hb.w.m    md, ms1         # int32 to int4
mncvt.b.h.m     md, ms1         # int16 to int8
mncvt.b.w.m     md, ms1         # int32 to int8
mncvt.h.w.m     md, ms1         # int32 to int16
mncvt.w.dw.m    md, ms1         # int64 to int32

# Convert float to float
mfcvt.bf.hf.m   md, ms1         # fp16 to bf16
mfcvt.hf.bf.m   md, ms1         # bf16 to fp16

mfwcvt.fw.f.m   md, ms1         # single-width float to double-width float
mfwcvt.hf.cf.m  md, ms1         # fp8 to fp16
mfwcvt.f.hf.m   md, ms1         # fp16 to fp32
mfwcvt.d.f.m    md, ms1         # fp32 to fp64

mfncvt.f.fw.m   md, ms1         # double-width float to single-width float
mfncvt.cf.hf.m  md, ms1         # fp16 to fp8
mfncvt.hf.f.m   md, ms1         # fp32 to fp16
mfncvt.f.d.m    md, ms1         # fp64 to fp32

# Convert integer to float
mfcvtu.f.x.m    md, ms1         # uint to float
mfcvtu.hf.h.m   md, ms1         # uint16 to fp16
mfcvtu.f.w.m    md, ms1         # uint32 to fp32
mfcvtu.d.dw.m   md, ms1         # uint64 to fp64

mfcvt.f.x.m     md, ms1         # int to float
mfcvt.hf.h.m    md, ms1         # int16 to fp16
mfcvt.f.w.m     md, ms1         # int32 to fp32
mfcvt.d.dw.m    md, ms1         # int64 to fp64

mfwcvtu.fw.x.m  md, ms1         # single-width uint to double-width float
mfwcvtu.fq.x.m  md, ms1         # single-width uint to quad-width float
mfwcvtu.fo.x.m  md, ms1         # single-width uint to oct-width float
mfwcvtu.hf.hb.m md, ms1         # uint4 to fp16
mfwcvtu.f.hb.m  md, ms1         # uint4 to fp32
mfwcvtu.hf.b.m  md, ms1         # uint8 to fp16
mfwcvtu.f.b.m   md, ms1         # uint8 to fp32
mfwcvtu.f.h.m   md, ms1         # uint16 to fp32
mfwcvtu.d.w.m   md, ms1         # uint32 to fp64

mfwcvt.fw.x.m   md, ms1         # single-width int to double-width float
mfwcvt.fq.x.m   md, ms1         # single-width int to quad-width float
mfwcvt.fo.x.m   md, ms1         # single-width int to oct-width float
mfwcvt.hf.hb.m  md, ms1         # int4 to fp16
mfwcvt.f.hb.m   md, ms1         # int4 to fp32
mfwcvt.hf.b.m   md, ms1         # int8 to fp16
mfwcvt.f.b.m    md, ms1         # int8 to fp32
mfwcvt.f.h.m    md, ms1         # int16 to fp32
mfwcvt.d.w.m    md, ms1         # int32 to fp64

mfncvtu.f.xw.m  md, ms1         # double-width uint to float
mfncvtu.hf.w.m  md, ms1         # uint32 to fp16
mfncvtu.f.dw.m  md, ms1         # uint64 to fp32

mfncvt.f.xw.m   md, ms1         # double-width int to float
mfncvt.hf.w.m   md, ms1         # int32 to fp16
mfncvt.f.dw.m   md, ms1         # int64 to fp32

# Convert float to integer
mfcvtu.x.f.m    md, ms1         # float to uint
mfcvtu.h.hf.m   md, ms1         # fp16 to uint16
mfcvtu.w.f.m    md, ms1         # fp32 to uint32
mfcvtu.dw.d.m   md, ms1         # fp64 to uint64

mfcvt.x.f.m     md, ms1         # float to int
mfcvt.h.hf.m    md, ms1         # fp16 to int16
mfcvt.w.f.m     md, ms1         # fp32 to int32
mfcvt.dw.d.m    md, ms1         # fp64 to int64

mfwcvtu.xw.f.m  md, ms1         # single-width float to double-width uint
mfwcvtu.w.hf.m  md, ms1         # fp16 to uint32
mfwcvtu.dw.f.m  md, ms1         # fp32 to uint64

mfwcvt.xw.f.m   md, ms1         # single-width float to double-width int
mfwcvt.w.hf.m   md, ms1         # fp16 to int32
mfwcvt.dw.f.m   md, ms1         # fp32 to int64

mfncvtu.x.fw.m  md, ms1         # double-width float to single-width uint
mfncvtu.x.fq.m  md, ms1         # quad-width float to single-width uint
mfncvtu.x.fo.m  md, ms1         # oct-width float to single-width uint
mfncvtu.hb.hf.m md, ms1         # fp16 to uint4
mfncvtu.hb.f.m  md, ms1         # fp32 to uint4
mfncvtu.b.hf.m  md, ms1         # fp16 to uint8
mfncvtu.b.f.m   md, ms1         # fp32 to uint8
mfncvtu.h.f.m   md, ms1         # fp32 to uint16
mfncvtu.w.d.m   md, ms1         # fp64 to uint32

mfncvt.x.fw.m   md, ms1         # double-width float to single-width int
mfncvt.x.fq.m   md, ms1         # quad-width float to single-width int
mfncvt.x.fo.m   md, ms1         # oct-width float to single-width int
mfncvt.hb.hf.m  md, ms1         # fp16 to int4
mfncvt.hb.f.m   md, ms1         # fp32 to int4
mfncvt.b.hf.m   md, ms1         # fp16 to int8
mfncvt.b.f.m    md, ms1         # fp32 to int8
mfncvt.h.f.m    md, ms1         # fp32 to int16
mfncvt.w.d.m    md, ms1         # fp64 to int32
```

Chapter 5. Intrinsic Examples

5.1. Matrix multiplication

```
void matmul_float16(c, a, b, m, k, n) {
    msettype(e16);                               // use 16bit input matrix element
    for (i = 0; i < m; i += mtilem) {            // loop at dim m with tiling
        mtilem = msettilem(m-i);
        for (j = 0; j < n; j += mtilen) {        // loop at dim n with tiling
            mtilen = msettilen(n-j);

            out = mwsub_mm(out, out)             // clear output reg
            for (s = 0; s < k; s += mtilek) {    // loop at dim k with tiling
                mtilek = msettilek(k-s);

                tr1 = mlae16_m(&a[i][s], k*2);   // load left matrix a
                tr2 = mlbe16_m(&b[s][j], n*2);   // load right matrix b
                out = mfwma_mm(tr1, tr2);        // tiled matrix multiply,
                                                 // double widen output
            }

            out = mfncvt_f_fw_m(out);            // convert widen result
            msce16_m(out, &c[i][j], n*2);        // store to matrix c
        }
    }
}
```

5.2. Matrix multiplication with left matrix transposed

```
void matmul_a_tr_float16(c, a, b, m, k, n) {
    msettype(e16);                               // use 16bit input matrix element
    for (i = 0; i < m; i += mtilem) {            // loop at dim m with tiling
        mtilem = msettilem(m-i);
        for (j = 0; j < n; j += mtilen) {        // loop at dim n with tiling
            mtilen = msettilen(n-j);

            out = mwsub_mm(out, out)             // clear output reg
            for (s = 0; s < k; s += mtilek) {    // loop at dim k with tiling
                mtilek = msettilek(k-s);

                tr1 = mlate16_m(&a[s][i], m*2); // load transposed left matrix a
                tr2 = mlbe16_m(&a[s][j], n*2);   // load right matrix b
                out = mfwma_mm(tr1, tr2);        // tiled matrix multiply,
                                                 // double widen output
            }
            out = mfncvt_f_fw_m(out);           // convert widen result
            msce16_m(out, &c[i][j], n*2);       // store to matrix c
        }
    }
}
```

5.3. Matrix transpose without multiplication

```
void mattrans_float16(out, in, h, w) {
    msettype(e16);                              // use 16bit input matrix element

    for (i = 0; i < h; i += mtilem) {           // loop at dim m with tiling
        mtilem = msettilem(h-i);
        for (j = 0; j < w; j += mtilek) {       // loop at dim k with tiling
            mtilek = msettilek(w-j);

            tr_in = mlae16_m(&in[i][j], w*2);   // load input matrix
            msate16_m(tr_in, &out[j][i], h*2);  // store output matrix
        }
    }
}
```

Chapter 6. Standard Matrix Extensions

6.1. Zma*b*: Matrix Mode Extension

The Zmab extension allows to use `C = A x B` mode for matrix multiplication, where the setting of `mcsr.mmode = 00` is legal.

The Zmabt extension allows to use `C = A x BT` mode for matrix multiplication, where the setting of `mcsr.mmode = 01` is legal.

The Zmatb extension allows to use `C = AT x B` mode for matrix multiplication, where the setting of `mcsr.mmode = 10` is legal.

6.2. Zmi4: Matrix 4-bit Integer Extension

The Zmi4 extension allows to use 4-bit integer as the data type of input matrix elements.

The Zmi4 extension adds a bit `mtype[3]` in `mtype` register.

Table 6. `mtype` register layout

Bits	Name	Description
XLEN-1	mill	Illegal value if set.
XLEN-2:16	0	Reserved if non-zero.
15	mba	Matrix out of bound agnostic.
14	mfp64	64-bit float point enabling.
13:12	mfp32[1:0]	32-bit float point enabling.
11:10	mfp16[1:0]	16-bit float point enabling.
9:8	mfp8[1:0]	8-bit float point enabling.
7	mint64	64-bit integer enabling.
6	mint32	32-bit integer enabling.
5	mint16	16-bit integer enabling.
4	mint8	8-bit integer enabling.
3	mint4	4-bit integer enabling.
2:0	msew[2:0]	Selected element width (SEW) setting.

For `mint4` field, write 1 to enable 4-bit integer where a 8-bit integer will be treated as a pair of 4-bit integers (the size of a row must be even). 0 will be returned and `mtype.mill` will be set if 4-bit integer is not supported.

The `mint4` field can be set with other fields by `msettype[i]` or set independently by `msetint` or `munsetint`.

```
msettypei rd, imm       # rd = new mtype, imm = new mtype setting.
msettype  rd, rs1       # rd = new mtype, rs1 = new mtype value.

msetint   rd, int4      # rd = new mtype, set mint4 = 1 to enable INT4 type.
munsetint rd, int4      # rd = new mtype, set mint4 = 0 to disable INT4 type.
```

As int4 must be in pair, the e8 load/store and data move instructions are reused for int4 data.

The element-wise and type-convert instructions with suffix .hb are added for int4 format.

Four octuple-widen instructions are added to support int4 matrix multiplication. So the output type is always 32-bit integer. As a result, int32 element-wise instructions and type-convert instructions between int4 and int32 must be supported for accumulation registers.

```
# Matrix multiplication instructions.
momau.[hb].mm       md, ms1, ms2
msomau.[hb].mm      md, ms1, ms2

moma.[hb].mm        md, ms1, ms2
msoma.[hb].mm       md, ms1, ms2

# Element-wise instructions.
maddu.[hb|w].mm     md, ms1, ms2
msaddu.[hb|w].mm    md, ms1, ms2
mwaddu.[hb].mm      md, ms1, ms2

madd.[hb|w].mm      md, ms1, ms2
msadd.[hb|w].mm     md, ms1, ms2
mwadd.[hb].mm       md, ms1, ms2

...

mwmul.[hb].mm       md, ms1, ms2
mwmulu.[hb].mm      md, ms1, ms2
mwmulsu.[hb].mm     md, ms1, ms2

# Type-convert instructions.
mcvt.x.xu.m         md, ms1
mcvt.hb.uhb.m       md, ms1
mcvt.xu.x.m         md, ms1
mcvt.uhb.hb.m       md, ms1
```

```
mwcvtu.xw.x.m       md, ms1
mwcvtu.xq.x.m       md, ms1
mwcvtu.xo.x.m       md, ms1
mwcvtu.b.hb.m       md, ms1
mwcvtu.h.hb.m       md, ms1
mwcvtu.w.hb.m       md, ms1

mwcvt.xw.x.m        md, ms1
mwcvt.xq.x.m        md, ms1
mwcvt.xo.x.m        md, ms1
mwcvt.b.hb.m        md, ms1
mwcvt.h.hb.m        md, ms1
mwcvt.w.hb.m        md, ms1

mncvtu.x.xw.m       md, ms1
mncvtu.x.xq.m       md, ms1
mncvtu.x.xo.m       md, ms1
mncvtu.hb.b.m       md, ms1
mncvtu.hb.h.m       md, ms1
mncvtu.hb.w.m       md, ms1

mncvt.x.xw.m        md, ms1
mncvt.x.xq.m        md, ms1
mncvt.x.xo.m        md, ms1
mncvt.hb.b.m        md, ms1
mncvt.hb.h.m        md, ms1
mncvt.hb.w.m        md, ms1
```

6.3. Zmi8: Matrix 8-bit Integer Extension

The Zmi8 extension allows to use 8-bit integer as the data type of input matrix elements.

The Zmi8 extension adds a bit `mtype[4]` in `mtype` register.

Table 7. `mtype` register layout

Bits	Name	Description
XLEN-1	mill	Illegal value if set.
XLEN-2:16	0	Reserved if non-zero.
15	mba	Matrix out of bound agnostic.
14	mfp64	64-bit float point enabling.
13:12	mfp32[1:0]	32-bit float point enabling.
11:10	mfp16[1:0]	16-bit float point enabling.

Bits	Name	Description
9:8	mfp8[1:0]	8-bit float point enabling.
7	mint64	64-bit integer enabling.
6	mint32	32-bit integer enabling.
5	mint16	16-bit integer enabling.
4	mint8	8-bit integer enabling.
3	mint4	4-bit integer enabling.
2:0	msew[2:0]	Selected element width (SEW) setting.

For `mint8` field, write 1 to enable 8-bit integer. 0 will be returned and `mtype.mill` will be set if 8-bit integer is not supported.

The `mint8` field can be set with other fields by `msettype[i]` or set independently by `msetint` or `munsetint`.

```
msettypei rd, imm       # rd = new mtype, imm = new mtype setting.
msettype  rd, rs1       # rd = new mtype, rs1 = new mtype value.

msetint   rd, int8      # rd = new mtype, set mint8 = 1 to enable INT8 type.
munsetint rd, int8      # rd = new mtype, set mint8 = 0 to disable INT8 type.
```

The e8 load/store and data move instructions are used for int8 data.

The element-wise and type-convert instructions with .b suffix are added for int8 format.

Four quadruple-widen instructions are added to support int8 matrix multiplication. So the output type is always 32-bit integer. As a result, int32 element-wise instructions and type-convert instructions between int8 and int32 must be supported for accumulation registers.

```
# Matrix multiplication instructions.
mqmau.[b].mm        md, ms1, ms2
msqmau.[b].mm       md, ms1, ms2

mqma.[b].mm         md, ms1, ms2
msqma.[b].mm        md, ms1, ms2

# Element-wise instructions.
maddu.[b|w].mm      md, ms1, ms2
msaddu.[b|w].mm     md, ms1, ms2
mwaddu.[b].mm       md, ms1, ms2

madd.[b|w].mm       md, ms1, ms2
```

```
msadd.[b|w].mm      md, ms1, ms2
mwadd.[b].mm        md, ms1, ms2

...

mwmul.[b].mm        md, ms1, ms2
mwmulu.[b].mm       md, ms1, ms2
mwmulsu.[b].mm      md, ms1, ms2

# Type-convert instructions.
mcvt.x.xu.m         md, ms1
mcvt.b.ub.m         md, ms1
mcvt.xu.x.m         md, ms1
mcvt.ub.b.m         md, ms1

mwcvtu.xw.x.m       md, ms1
mwcvtu.xq.x.m       md, ms1
mwcvtu.h.b.m        md, ms1
mwcvtu.w.b.m        md, ms1

mwcvt.xw.x.m        md, ms1
mwcvt.xq.x.m        md, ms1
mwcvt.h.b.m         md, ms1
mwcvt.w.b.m         md, ms1

mncvtu.x.xw.m       md, ms1
mncvtu.x.xq.m       md, ms1
mncvtu.b.h.m        md, ms1
mncvtu.b.w.m        md, ms1

mncvt.x.xw.m        md, ms1
mncvt.x.xq.m        md, ms1
mncvt.b.h.m         md, ms1
mncvt.b.w.m         md, ms1
```

6.4. Zmi16: Matrix 16-bit Integer Extension

The Zmi16 extension allows to use 16-bit integer as the data type of input matrix elements.

The Zmi16 extension adds a bit `mtype[5]` in `mtype` register.

Table 8. `mtype` register layout

Bits	Name	Description
XLEN-1	mill	Illegal value if set.
XLEN-2:16	0	Reserved if non-zero.

Bits	Name	Description
15	mba	Matrix out of bound agnostic.
14	mfp64	64-bit float point enabling.
13:12	mfp32[1:0]	32-bit float point enabling.
11:10	mfp16[1:0]	16-bit float point enabling.
9:8	mfp8[1:0]	8-bit float point enabling.
7	mint64	64-bit integer enabling.
6	mint32	32-bit integer enabling.
5	mint16	16-bit integer enabling.
4	mint8	8-bit integer enabling.
3	mint4	4-bit integer enabling.
2:0	msew[2:0]	Selected element width (SEW) setting.

For `mint16` field, write 1 to enable 16-bit integer. 0 will be returned and `mtype.mill` will be set if 16-bit integer is not supported.

The `mint16` field can be set with other fields by `msettype[i]` or set independently by `msetint` or `munsetint`.

```
msettypei rd, imm       # rd = new mtype, imm = new mtype setting.
msettype  rd, rs1       # rd = new mtype, rs1 = new mtype value.

msetint   rd, int16     # rd = new mtype, set mint16 = 1 to enable INT16 type.
munsetint rd, int16     # rd = new mtype, set mint16 = 0 to disable INT16 type.
```

The e16 load/store and data move instructions are used for int16 data.

The element-wise and type-convert instructions with .h suffix are added for int16 format.

Six no-widen and double-widen instructions are added to support int16 matrix multiplication. So the output type is 16-bit or 32-bit integer.

```
# Matrix multiplication instructions.
mmau.[h].mm         md, ms1, ms2
msmau.[h].mm        md, ms1, ms2
mwmau.[h].mm        md, ms1, ms2

mma.[h].mm          md, ms1, ms2
msma.[h].mm         md, ms1, ms2
```

```
mwma.[h].mm          md, ms1, ms2

# Element-wise instructions.
maddu.[h|w].mm       md, ms1, ms2
msaddu.[h|w].mm      md, ms1, ms2
mwaddu.[h].mm        md, ms1, ms2

madd.[h|w].mm        md, ms1, ms2
msadd.[h|w].mm       md, ms1, ms2
mwadd.[h].mm         md, ms1, ms2

...

mwmul.[h].mm         md, ms1, ms2
mwmulu.[h].mm        md, ms1, ms2
mwmulsu.[h].mm       md, ms1, ms2

# Type-convert instructions.
mcvt.x.xu.m          md, ms1
mcvt.h.uh.m          md, ms1
mcvt.xu.x.m          md, ms1
mcvt.uh.h.m          md, ms1

mwcvtu.xw.x.m        md, ms1
mwcvtu.w.h.m         md, ms1
mwcvt.xw.x.m         md, ms1
mwcvt.w.h.m          md, ms1

mncvtu.x.xw.m        md, ms1
mncvtu.h.w.m         md, ms1
mncvt.x.xw.m         md, ms1
mncvt.h.w.m          md, ms1
```

6.5. Zmi32: Matrix 32-bit Integer Extension

The Zmi32 extension allows to use 32-bit integer as the data type of input matrix elements.

The Zmi32 extension adds a bit `mtype[6]` in `mtype` register.

Table 9. `mtype` register layout

Bits	Name	Description
XLEN-1	mill	Illegal value if set.
XLEN-2:16	0	Reserved if non-zero.
15	mba	Matrix out of bound agnostic.

Bits	Name	Description
14	mfp64	64-bit float point enabling.
13:12	mfp32[1:0]	32-bit float point enabling.
11:10	mfp16[1:0]	16-bit float point enabling.
9:8	mfp8[1:0]	8-bit float point enabling.
7	mint64	64-bit integer enabling.
6	mint32	32-bit integer enabling.
5	mint16	16-bit integer enabling.
4	mint8	8-bit integer enabling.
3	mint4	4-bit integer enabling.
2:0	msew[2:0]	Selected element width (SEW) setting.

For `mint32` field, write 1 to enable 32-bit integer. 0 will be returned and `mtype.mill` will be set if 32-bit integer is not supported.

The `mint32` field can be set with other fields by `msettype[i]` or set independently by `msetint` or `munsetint`.

```
msettypei rd, imm       # rd = new mtype, imm = new mtype setting.
msettype  rd, rs1       # rd = new mtype, rs1 = new mtype value.

msetint   rd, int32     # rd = new mtype, set mint32 = 1 to enable INT32 type.
munsetint rd, int32     # rd = new mtype, set mint32 = 0 to disable INT32 type.
```

The e32 load/store and data move instructions are used for int32 data.

The element-wise and type-convert instructions with .w suffix are added for int32 format.

Four no-widen instructions are added to support int32 matrix multiplication. So the output type is always 32-bit integer.

```
# Matrix multiplication instructions.
mmau.[w].mm  md, ms1, ms2      # unsigned int32, output no-widen
msmau.[w].mm md, ms1, ms2      # unsigned int32, output no-widen and saturated

mma.[w].mm   md, ms1, ms2      # signed int32, output no-widen
msma.[w].mm  md, ms1, ms2      # signed int32, output no-widen and saturated

# Element-wise instructions.
maddu.[w].mm        md, ms1, ms2
```

```
msaddu.[w].mm       md, ms1, ms2

madd.[w].mm         md, ms1, ms2
msadd.[w].mm        md, ms1, ms2

...

msmul.[w].mm        md, ms1, ms2
msmulu.[w].mm       md, ms1, ms2
msmulsu.[w].mm      md, ms1, ms2

# Type-convert instructions.
mcvt.x.xu.m         md, ms1
mcvt.w.uw.m         md, ms1
mcvt.xu.x.m         md, ms1
mcvt.uw.w.m         md, ms1
```

6.6. Zmi64: Matrix 64-bit Integer Extension

The Zmi64 extension allows to use 64-bit integer as the data type of input matrix elements.

The Zmi64 extension adds a bit `mtype[7]` in `mtype` register.

Table 10. `mtype` register layout

Bits	Name	Description
XLEN-1	mill	Illegal value if set.
XLEN-2:16	0	Reserved if non-zero.
15	mba	Matrix out of bound agnostic.
14	mfp64	64-bit float point enabling.
13:12	mfp32[1:0]	32-bit float point enabling.
11:10	mfp16[1:0]	16-bit float point enabling.
9:8	mfp8[1:0]	8-bit float point enabling.
7	mint64	64-bit integer enabling.
6	mint32	32-bit integer enabling.
5	mint16	16-bit integer enabling.
4	mint8	8-bit integer enabling.
3	mint4	4-bit integer enabling.
2:0	msew[2:0]	Selected element width (SEW) setting.

For `mint64` field, write 1 to enable 64-bit integer. 0 will be returned and `mtype.mill` will be set if 64-bit integer is not supported.

The `mint64` field can be set with other fields by `msettype[i]` or set independently by `msetint` or `munsetint`.

```
msettypei rd, imm       # rd = new mtype, imm = new mtype setting.
msettype  rd, rs1       # rd = new mtype, rs1 = new mtype value.

msetint   rd, int64     # rd = new mtype, set mint64 = 1 to enable INT64 type.
munsetint rd, int64     # rd = new mtype, set mint64 = 0 to disable INT64 type.
```

The e64 load/store and data move instructions are used for int64 data.

The element-wise and type-convert instructions with .dw suffix are added for int64 format.

Four no-widen instructions are added to support int64 matrix multiplication. So the output type is always 64-bit integer.

```
# Matrix multiplication instructions.
mmau.[dw].mm        md, ms1, ms2
msmau.[dw].mm       md, ms1, ms2

mma.[dw].mm         md, ms1, ms2
msma.[dw].mm        md, ms1, ms2

# Element-wise instructions.
maddu.[dw].mm       md, ms1, ms2
msaddu.[dw].mm      md, ms1, ms2

madd.[dw].mm        md, ms1, ms2
msadd.[dw].mm       md, ms1, ms2

...

msmul.[dw].mm       md, ms1, ms2
msmulu.[dw].mm      md, ms1, ms2
msmulsu.[dw].mm     md, ms1, ms2

# Type-convert instructions.
mcvt.x.xu.m         md, ms1
mcvt.dw.udw.m       md, ms1
mcvt.xu.x.m         md, ms1
mcvt.udw.dw.m       md, ms1
```

If Zmi32 is also supported, the 32-bit widening arithmetic instructions and type convert between int32 and int64 are also provided.

```
# Matrix multiplication instructions.
mwmau.[w].mm        md, ms1, ms2
mwma.[w].mm         md, ms1, ms2

# Element-wise instructions.
maddu.[w].mm        md, ms1, ms2
msaddu.[w].mm       md, ms1, ms2
mwaddu.[w].mm       md, ms1, ms2

madd.[w].mm         md, ms1, ms2
msadd.[w].mm        md, ms1, ms2
mwadd.[w].mm        md, ms1, ms2

...

mwmul.[w].mm        md, ms1, ms2
mwmulu.[w].mm       md, ms1, ms2
mwmulsu.[w].mm      md, ms1, ms2

# Type-convert instructions.
mwcvtu.xw.x.m       md, ms1
mwcvtu.dw.w.m       md, ms1
mwcvt.xw.x.m        md, ms1
mwcvt.dw.w.m        md, ms1

mncvtu.x.xw.m       md, ms1
mncvtu.w.dw.m       md, ms1
mncvt.x.xw.m        md, ms1
mncvt.w.dw.m        md, ms1
```

6.7. Zmf8e4m3: Matrix 8-bit E4M3 Float Point Extension

The Zmf8e4m3 extension allows to use 8-bit float point format with 4-bit exponent and 3-bit mantissa as the data type of input matrix elements.

The Zmf8e4m3 extension uses a 2-bit `mfp8` field, `mtype[9:8]`, in `mtype` register.

Table 11. `mtype` register layout

Bits	Name	Description
XLEN-1	mill	Illegal value if set.
XLEN-2:16	0	Reserved if non-zero.
15	mba	Matrix out of bound agnostic.

Bits	Name	Description
14	mfp64	64-bit float point enabling.
13:12	mfp32[1:0]	32-bit float point enabling.
11:10	mfp16[1:0]	16-bit float point enabling.
9:8	mfp8[1:0]	8-bit float point enabling.
7	mint64	64-bit integer enabling.
6	mint32	32-bit integer enabling.
5	mint16	16-bit integer enabling.
4	mint8	8-bit integer enabling.
3	mint4	4-bit integer enabling.
2:0	msew[2:0]	Selected element width (SEW) setting.

For `mfp8` field, write 01 to enable 8-bit E4M3 float point. 0 will be returned and `mtype.mill` will be set if E4M3 is not supported.

The `mfp8` field can be set with other fields by `msettype[i]` or set independently by `msetfp` or `munsetfp`.

```
msettypei rd, imm       # rd = new mtype, imm = new mtype setting.
msettype  rd, rs1       # rd = new mtype, rs1 = new mtype value.

msetfp    rd, fp8       # rd = new mtype, set mfp8 = 01 to enable E4M3 type.
msetfp    rd, e4m3      # rd = new mtype, set mfp8 = 01 to enable E4M3 type.
munsetfp  rd, fp8       # rd = new mtype, set mfp8 = 00 to disable FP8 type.
```

The e8 load/store and data move instructions are used for E4M3 data.

The element-wise and type-convert instructions with .cf suffix are added for E4M3 format.

A double-widen instruction and a quadruple-widen instruction are added to support E4M3 matrix multiplication. So the output type is 16-bit or 32-bit float point. As a result, fp16/fp32 element-wise instructions and type-convert instructions between E4M3 and fp16/fp32 must be supported for accumulation registers.

```
# Matrix multiplication instructions.
mfwma.[cf].mm       md, ms1, ms2
mfqma.[cf].mm       md, ms1, ms2

# Element-wise instructions.
mfadd.[cf|hf|f].mm  md, ms1, ms2
```

```
mfwadd.[cf|hf].mm   md, ms1, ms2

mfsub.[cf|hf|f].mm  md, ms1, ms2
mfwsub.[cf|hf].mm   md, ms1, ms2

...

mfsqrt.[cf|hf|f].mm md, ms1

# Type-convert instructions.
mfwcvt.fw.f.m       md, ms1
mfwcvt.hf.cf.m      md, ms1
mfwcvt.f.hf.m       md, ms1

mfncvt.f.fw.m       md, ms1
mfncvt.cf.hf.m      md, ms1
mfncvt.hf.f.m       md, ms1
```

6.8. Zmf8e5m2: Matrix 8-bit E5M2 Float Point Extension

The Zmf8e5m2 extension allows to use 8-bit float point format with 5-bit exponent and 2-bit mantissa as the data type of input matrix elements.

The Zmf8e5m2 extension uses a 2-bit `mfp8` field, `mtype[9:8]`, in `mtype` register.

Table 12. `mtype` register layout

Bits	Name	Description
XLEN-1	mill	Illegal value if set.
XLEN-2:16	0	Reserved if non-zero.
15	mba	Matrix out of bound agnostic.
14	mfp64	64-bit float point enabling.
13:12	mfp32[1:0]	32-bit float point enabling.
11:10	mfp16[1:0]	16-bit float point enabling.
9:8	mfp8[1:0]	8-bit float point enabling.
7	mint64	64-bit integer enabling.
6	mint32	32-bit integer enabling.
5	mint16	16-bit integer enabling.
4	mint8	8-bit integer enabling.

Bits	Name	Description
3	mint4	4-bit integer enabling.
2:0	msew[2:0]	Selected element width (SEW) setting.

For `mfp8` field, write 10 to enable 8-bit E5M2 float point. 0 will be returned and `mtype.mill` will be set if E5M2 is not supported.

The `mfp8` field can be set with other fields by `msettype[i]` or set independently by `msetfp` or `munsetfp`.

```
msettypei rd, imm       # rd = new mtype, imm = new mtype setting.
msettype  rd, rs1       # rd = new mtype, rs1 = new mtype value.

msetfp    rd, e5m2      # rd = new mtype, set mfp8 = 10 to enable E5M2 type.
munsetfp  rd, fp8       # rd = new mtype, set mfp8 = 00 to disable FP8 type.
```

The e8 load/store and data move instructions are used for E5M2 data.

The element-wise and type-convert instructions with .cf suffix are added for E5M2 format.

A double-widen instruction and a quadruple-widen instruction are added to support E5M2 matrix multiplication. So the output type is 16-bit or 32-bit float point. As a result, fp16/fp32 element-wise instructions and type-convert instructions between E5M2 and fp16/fp32 must be supported for accumulation registers.

```
# Matrix multiplication instructions.
mfwma.[cf].mm       md, ms1, ms2
mfqma.[cf].mm       md, ms1, ms2

# Element-wise instructions.
mfadd.[cf|hf|f].mm  md, ms1, ms2
mfwadd.[cf|hf].mm   md, ms1, ms2

mfsub.[cf|hf|f].mm  md, ms1, ms2
mfwsub.[cf|hf].mm   md, ms1, ms2

...

mfsqrt.[cf|hf|f].mm md, ms1

# Type-convert instructions.
mfwcvt.fw.f.m       md, ms1
mfwcvt.hf.cf.m      md, ms1
mfwcvt.f.hf.m       md, ms1
```

```
mfncvt.f.fw.m       md, ms1
mfncvt.cf.hf.m      md, ms1
mfncvt.hf.f.m       md, ms1
```

6.9. Zmf8e3m4: Matrix 8-bit E3M4 Float Point Extension

The Zmf8e3m4 extension allows to use 8-bit float point format with 3-bit exponent and 4-bit mantissa as the data type of input matrix elements.

The Zmf8e3m4 extension uses a 2-bit `mfp8` field, `mtype[9:8]`, in `mtype` register.

Table 13. `mtype` register layout

Bits	Name	Description
XLEN-1	mill	Illegal value if set.
XLEN-2:16	0	Reserved if non-zero.
15	mba	Matrix out of bound agnostic.
14	mfp64	64-bit float point enabling.
13:12	mfp32[1:0]	32-bit float point enabling.
11:10	mfp16[1:0]	16-bit float point enabling.
9:8	mfp8[1:0]	8-bit float point enabling.
7	mint64	64-bit integer enabling.
6	mint32	32-bit integer enabling.
5	mint16	16-bit integer enabling.
4	mint8	8-bit integer enabling.
3	mint4	4-bit integer enabling.
2:0	msew[2:0]	Selected element width (SEW) setting.

For `mfp8` field, write 11 to enable 8-bit E3M4 float point. 0 will be returned and `mtype.mill` will be set if E3M4 is not supported.

The `mfp8` field can be set with other fields by `msettype[i]` or set independently by `msetfp` or `munsetfp`.

```
msettypei rd, imm       # rd = new mtype, imm = new mtype setting.
msettype  rd, rs1       # rd = new mtype, rs1 = new mtype value.

msetfp    rd, e3m4      # rd = new mtype, set mfp8 = 11 to enable E3M4 type.
```

```
munsetfp  rd, fp8       # rd = new mtype, set mfp8 = 00 to disable FP8 type.
```

The e8 load/store and data move instructions are used for E3M4 data.

The element-wise and type-convert instructions with .cf suffix are added for E3M4 format.

A double-widen instruction and a quadruple-widen instruction are added to support E3M4 matrix multiplication. So the output type is 16-bit or 32-bit float point. As a result, fp16/fp32 element-wise instructions and type-convert instructions between E3M4 and fp16/fp32 must be supported for accumulation registers.

```
# Matrix multiplication instructions.
mfwma.[cf].mm       md, ms1, ms2
mfqma.[cf].mm       md, ms1, ms2

# Element-wise instructions.
mfadd.[cf|hf|f].mm  md, ms1, ms2
mfwadd.[cf|hf].mm   md, ms1, ms2

mfsub.[cf|hf|f].mm  md, ms1, ms2
mfwsub.[cf|hf].mm   md, ms1, ms2

...

mfsqrt.[cf|hf|f].mm md, ms1

# Type-convert instructions.
mfwcvt.fw.f.m       md, ms1
mfwcvt.hf.cf.m      md, ms1
mfwcvt.f.hf.m       md, ms1

mfncvt.f.fw.m       md, ms1
mfncvt.cf.hf.m      md, ms1
mfncvt.hf.f.m       md, ms1
```

6.10. Zmf16e5m10: Matrix 16-bit Half-precision Float-point (FP16) Extension

The Zmf16e5m10 extension allows to use FP16 format with 5-bit exponent and 10-bit mantissa as the data type of input matrix elements.

The Zmf16e5m10 extension uses a 2-bit `mfp16` field, `mtype[11:10]`, in `mtype` register.

Table 14. `mtype` register layout

Bits	Name	Description
XLEN-1	mill	Illegal value if set.
XLEN-2:16	0	Reserved if non-zero.
15	mba	Matrix out of bound agnostic.
14	mfp64	64-bit float point enabling.
13:12	mfp32[1:0]	32-bit float point enabling.
11:10	mfp16[1:0]	16-bit float point enabling.
9:8	mfp8[1:0]	8-bit float point enabling.
7	mint64	64-bit integer enabling.
6	mint32	32-bit integer enabling.
5	mint16	16-bit integer enabling.
4	mint8	8-bit integer enabling.
3	mint4	4-bit integer enabling.
2:0	msew[2:0]	Selected element width (SEW) setting.

For `mfp16` field, write 01 to enable 16-bit E5M10 float point (FP16). 0 will be returned and `mtype.mill` will be set if FP16 is not supported.

The `mfp16` field can be set with other fields by `msettype[hi]` or set independently by `msetfp` or `munsetfp`.

```
msettypehi rd, imm      # rd = new mtype, imm = new mtype setting.
msettype   rd, rs1      # rd = new mtype, rs1 = new mtype value.

msetfp     rd, fp16     # rd = new mtype, set mfp16 = 01 to enable FP16 type.
munsetfp   rd, fp16     # rd = new mtype, set mfp16 = 00 to disable FP16 type.
```

The e16 load/store and data move instructions are used for FP16 data.

The element-wise and type-convert instructions with .hf suffix are added for FP16 format.

A no-widen instruction and a double-widen instruction are added to support FP16 matrix multiplication. So the output type is 16-bit or 32-bit float point. As a result, fp32 element-wise instructions and type-convert instructions between fp16 and fp32 must be supported for accumulation registers. If integer types are enabled, type-convert instructions between fp16 and integer should also be supported.

```
# Matrix multiplication instructions.
```

```
mfma.[hf].mm        md, ms1, ms2
mfwma.[hf].mm       md, ms1, ms2

# Element-wise instructions.
mfadd.[hf|f].mm     md, ms1, ms2
mfwadd.[hf].mm      md, ms1, ms2

mfsub.[hf|f].mm     md, ms1, ms2
mfwsub.[hf].mm      md, ms1, ms2

...

mfsqrt.[hf|f].mm    md, ms1

# Type-convert instructions.
mfwcvt.fw.f.m       md, ms1
mfwcvt.f.hf.m       md, ms1

mfncvt.f.fw.m       md, ms1
mfncvt.hf.f.m       md, ms1
```

If Zmi* is also supported, the type-convert instructions between integer and fp16/fp32 are also provided.

6.11. Zmf16e8m7: Matrix 16-bit BFloat (BF16) Extension

The Zmf16e8m7 extension allows to use BF16 format with 8-bit exponent and 7-bit mantissa as the data type of input matrix elements.

The Zmf16e8m7 extension uses a 2-bit `mfp16` field, `mtype[11:10]`, in `mtype` register.

Table 15. `mtype` register layout

Bits	Name	Description
XLEN-1	mill	Illegal value if set.
XLEN-2:16	0	Reserved if non-zero.
15	mba	Matrix out of bound agnostic.
14	mfp64	64-bit float point enabling.
13:12	mfp32[1:0]	32-bit float point enabling.
11:10	mfp16[1:0]	16-bit float point enabling.
9:8	mfp8[1:0]	8-bit float point enabling.
7	mint64	64-bit integer enabling.

Bits	Name	Description
6	mint32	32-bit integer enabling.
5	mint16	16-bit integer enabling.
4	mint8	8-bit integer enabling.
3	mint4	4-bit integer enabling.
2:0	msew[2:0]	Selected element width (SEW) setting.

For `mfp16` field, write 10 to enable 16-bit E8M7 float point (BF16). 0 will be returned and `mtype.mill` will be set if BF16 is not supported.

The `mfp16` field can be set with other fields by `msettype[hi]` or set independently by `msetfp` or `munsetfp`.

```
msettypehi rd, imm      # rd = new mtype, imm = new mtype setting.
msettype   rd, rs1      # rd = new mtype, rs1 = new mtype value.

msetfp     rd, bf16     # rd = new mtype, set mfp16 = 10 to enable BF16 type.
munsetfp   rd, fp16     # rd = new mtype, set mfp16 = 00 to disable BF16 type.
```

The e16 load/store and data move instructions are used for BF16 data.

The element-wise and type-convert instructions with .hf suffix are reused for BF16 format.

A no-widen instruction and a double-widen instruction are added to support BF16 matrix multiplication. So the output type is 16-bit or 32-bit float point. As a result, fp32 element-wise instructions and type-convert instructions between bf16 and fp32 must be supported for accumulation registers. If integer types are enabled, type-convert instructions between bf16 and integer should also be supported.

```
# Matrix multiplication instructions.
mfma.[hf].mm        md, ms1, ms2
mfwma.[hf].mm       md, ms1, ms2

# Element-wise instructions.
mfadd.[hf|f].mm     md, ms1, ms2
mfwadd.[hf].mm      md, ms1, ms2

mfsub.[hf|f].mm     md, ms1, ms2
mfwsub.[hf].mm      md, ms1, ms2

...

mfsqrt.[hf|f].mm    md, ms1
```

```
# Type-convert instructions.
mfwcvt.fw.f.m       md, ms1
mfwcvt.f.hf.m       md, ms1

mfncvt.f.fw.m       md, ms1
mfncvt.hf.f.m       md, ms1
```

If Zmi* is also supported, the type-convert instructions between integer and fp16/fp32 are also provided.

If Zmf16e5m10 is also supported, the type-convert instructions between fp16 and bf16 are also provided.

```
# Type-convert instructions.
mfcvt.bf.hf.m       md, ms1
mfcvt.hf.bf.m       md, ms1
```

6.12. Zmf32e8m23: Matrix 32-bit Float-point Extension

The Zmf32e8m23 extension allows to use standard FP32 format with 8-bit exponent and 23-bit mantissa as the data type of input matrix elements.

The Zmf32e8m23 extension uses a 2-bit `mfp32` field, `mtype[13:12]`, in `mtype` register.

Table 16. `mtype` register layout

Bits	Name	Description
XLEN-1	mill	Illegal value if set.
XLEN-2:16	0	Reserved if non-zero.
15	mba	Matrix out of bound agnostic.
14	mfp64	64-bit float point enabling.
13:12	mfp32[1:0]	32-bit float point enabling.
11:10	mfp16[1:0]	16-bit float point enabling.
9:8	mfp8[1:0]	8-bit float point enabling.
7	mint64	64-bit integer enabling.
6	mint32	32-bit integer enabling.
5	mint16	16-bit integer enabling.
4	mint8	8-bit integer enabling.

Bits	Name	Description
3	mint4	4-bit integer enabling.
2:0	msew[2:0]	Selected element width (SEW) setting.

For `mfp32` field, write 01 to enable 32-bit E8M23 float point (FP32). 0 will be returned and `mtype.mill` will be set if FP32 is not supported.

The `mfp32` field can be set with other fields by `msettype[hi]` or set independently by `msetfp` or `munsetfp`.

```
msettypehi rd, imm      # rd = new mtype, imm = new mtype setting.
msettype   rd, rs1      # rd = new mtype, rs1 = new mtype value.

msetfp     rd, fp32     # rd = new mtype, set mfp32 = 01 to enable FP32 type.
munsetfp   rd, fp32     # rd = new mtype, set mfp32 = 00 to disable FP32 type.
```

The e32 load/store and data move instructions are used for FP32 data.

The element-wise and type-convert instructions with .f suffix are added for FP32 format.

A no-widen instructionis added to support FP32 matrix multiplication. So the output type is 32-bit float point. If integer types are enabled, type-convert instructions between fp32 and integer should also be supported.

```
# Matrix multiplication instructions.
mfma.[f].mm         md, ms1, ms2

# Element-wise instructions.
mfadd.[f].mm        md, ms1, ms2
mfsub.[f].mm        md, ms1, ms2
...
mfsqrt.[f].mm       md, ms1
```

If Zmi* is also supported, the type-convert instructions between integer and fp32 are also provided.

6.13. Zmf19e8m10: Matrix 19-bit TensorFloat32 (TF32) Extension

The Zmf19e8m10 extension allows to use TF32 format with 8-bit exponent and 10-bit mantissa as the data type of input matrix elements.

The Zmf19e8m10 extension uses a 2-bit `mfp32` field, `mtype[13:12]`, in `mtype` register.

Table 17. `mtype` register layout

Bits	Name	Description
XLEN-1	mill	Illegal value if set.
XLEN-2:16	0	Reserved if non-zero.
15	mba	Matrix out of bound agnostic.
14	mfp64	64-bit float point enabling.
13:12	mfp32[1:0]	32-bit float point enabling.
11:10	mfp16[1:0]	16-bit float point enabling.
9:8	mfp8[1:0]	8-bit float point enabling.
7	mint64	64-bit integer enabling.
6	mint32	32-bit integer enabling.
5	mint16	16-bit integer enabling.
4	mint8	8-bit integer enabling.
3	mint4	4-bit integer enabling.
2:0	msew[2:0]	Selected element width (SEW) setting.

For `mfp32` field, write 10 to enable 19-bit E8M10 float point (TF32). 0 will be returned and `mtype.mill` will be set if TF32 is not supported.

The `mfp32` field can be set with other fields by `msettype[hi]` or set independently by `msetfp` or `munsetfp`.

```
msettypehi rd, imm      # rd = new mtype, imm = new mtype setting.
msettype   rd, rs1      # rd = new mtype, rs1 = new mtype value.

msetfp     rd, tf32     # rd = new mtype, set mfp32 = 10 to enable TF32 type.
munsetfp   rd, fp32     # rd = new mtype, set mfp32 = 00 to disable FP32 type.
```

TF32 implementions are designed to achieve better performance on matrix multiplications and convolutions by rounding input Float32 data to have 10 bits of mantissa, and accumulating results with FP32 precision, maintaining FP32 dynamic range.

So when Zmtf32 is used, Float32 is still used as the input and output data type for matrix multiplication.

The e32 load/store and data move instructions are used for TF32 data.

The element-wise and type-convert instructions are not supported for TF32 format.

A no-widen instruction is added to support TF32 matrix multiplication. So the output type is always 32-bit float point (FP32). As a result, fp32 element-wise instructions must be supported for accumulation registers. If integer types are enabled, type-convert instructions between fp32 and integer should also be supported.

```
mfma.[f].mm  md, ms1, ms2       # 19-bit float point, output no-widen
```

There is no double-widen version for TF32 matrix multiplication (a double-widen version for standard FP32 is supported by Zmf64e11m52 extension).

6.14. Zmf64e11m52: Matrix 64-bit Float-point Extension

The Zmf64e11m52 extension allows to use standard FP64 format with 11-bit exponent and 52-bit mantissa as the data type of input matrix elements.

The Zmf64e11m52 extension uses a 1-bit `mfp64` field, `mtype[14]`, in `mtype` register.

Table 18. `mtype` register layout

Bits	Name	Description
XLEN-1	mill	Illegal value if set.
XLEN-2:16	0	Reserved if non-zero.
15	mba	Matrix out of bound agnostic.
14	mfp64	64-bit float point enabling.
13:12	mfp32[1:0]	32-bit float point enabling.
11:10	mfp16[1:0]	16-bit float point enabling.
9:8	mfp8[1:0]	8-bit float point enabling.
7	mint64	64-bit integer enabling.
6	mint32	32-bit integer enabling.
5	mint16	16-bit integer enabling.
4	mint8	8-bit integer enabling.
3	mint4	4-bit integer enabling.
2:0	msew[2:0]	Selected element width (SEW) setting.

For `mfp64` field, write 1 to enable 64-bit E11M52 float point (FP64). 0 will be returned and `mtype.mill` will be set if FP64 is not supported.

The `mfp64` field can be set with other fields by `msettype[hi]` or set independently by `msetfp` or `munsetfp`.

```
msettypehi rd, imm      # rd = new mtype, imm = new mtype setting.
msettype   rd, rs1      # rd = new mtype, rs1 = new mtype value.

msetfp     rd, fp64     # rd = new mtype, set mfp64 = 1 to enable FP64 type.
munsetfp   rd, fp64     # rd = new mtype, set mfp64 = 0 to disable FP64 type.
```

The e64 load/store and data move instructions are used for FP64 data.

The element-wise and type-convert instructions with .d suffix are added for FP64 format.

A no-widen instruction is added to support FP64 matrix multiplication. The output type is always 64-bit float point (FP64). If integer types are enabled, type-convert instructions between fp64 and integer should also be supported.

```
mfma.[d].mm         md, ms1, ms2
```

If Zmi* is also supported, the type-convert instructions between integer and fp64 are also provided.

If Zmf32e8m23 is also supported, the 32-bit widening arithmetic instructions and type convert between fp32 and fp64 are also provided.

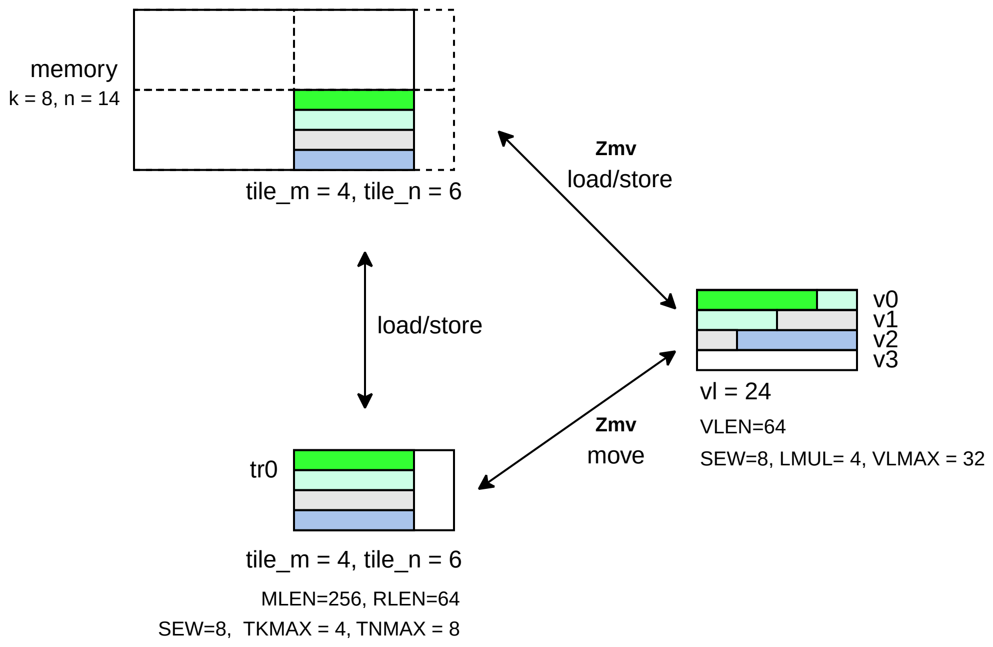
```
# Matrix multiplication instructions.
mfwma.[f].mm        md, ms1, ms2

# Element-wise instructions.
mfwadd.[f].mm       md, ms1, ms2
mfwsub.[f].mm       md, ms1, ms2
...
mfwmul.[f].mm       md, ms1, ms2

# Type-convert instructions.
mfwcvt.fw.f.m       md, ms1
mfwcvt.d.f.m        md, ms1
mfncvt.f.fw.m       md, ms1
mfncvt.f.d.m        md, ms1
```

6.15. Zmv: Matrix for Vector operations

The Zmv extension is defined to provide matrix support with the RISC-V Vector "V" extension.

The Zmv extension allows to load matrix tile slices into vector registers, and move data between slices of a matrix register and vector registers.

The data layout examples of registers and memory in Zmv are shown below.



6.15.1. Load Instructions

```
# vd destination, rs1 base address, rs2 row byte stride
# lmul / (eew/sew) rows or columns

# for left matrix, a
mlae8.v    vd, (rs1), rs2 #  8-bit tile slices load to vregs
mlae16.v   vd, (rs1), rs2 # 16-bit tile slices load to vregs
mlae32.v   vd, (rs1), rs2 # 32-bit tile slices load to vregs
mlae64.v   vd, (rs1), rs2 # 64-bit tile slices load to vregs

# for right matrix, b
mlbe8.v    vd, (rs1), rs2 #  8-bit tile slices load to vregs
mlbe16.v   vd, (rs1), rs2 # 16-bit tile slices load to vregs
mlbe32.v   vd, (rs1), rs2 # 32-bit tile slices load to vregs
mlbe64.v   vd, (rs1), rs2 # 64-bit tile slices load to vregs

# for output matrix, c
mlce8.v    vd, (rs1), rs2 #  8-bit tile slices load to vregs
mlce16.v   vd, (rs1), rs2 # 16-bit tile slices load to vregs
mlce32.v   vd, (rs1), rs2 # 32-bit tile slices load to vregs
mlce64.v   vd, (rs1), rs2 # 64-bit tile slices load to vregs
```

6.15.2. Store Instructions

```
# vs3 store data, rs1 base address, rs2 row byte stride
# lmul / (eew/sew) rows or columns

# for left matrix, a
msae8.v    vs3, (rs1), rs2 #  8-bit tile slices store from vregs
msae16.v   vs3, (rs1), rs2 # 16-bit tile slices store from vregs
msae32.v   vs3, (rs1), rs2 # 32-bit tile slices store from vregs
msae64.v   vs3, (rs1), rs2 # 64-bit tile slices store from vregs

# for right matrix, b
msbe8.v    vs3, (rs1), rs2 #  8-bit tile slices store from vregs
msbe16.v   vs3, (rs1), rs2 # 16-bit tile slices store from vregs
msbe32.v   vs3, (rs1), rs2 # 32-bit tile slices store from vregs
msbe64.v   vs3, (rs1), rs2 # 64-bit tile slices store from vregs

# for output matrix, c
msce8.v    vs3, (rs1), rs2 #  8-bit tile slices store from vregs
msce16.v   vs3, (rs1), rs2 # 16-bit tile slices store from vregs
msce32.v   vs3, (rs1), rs2 # 32-bit tile slices store from vregs
msce64.v   vs3, (rs1), rs2 # 64-bit tile slices store from vregs
```

6.15.3. Data Move Instructions

For data moving between vector and matrix, the vsew of vector must equal to msew of matrix.

The number of elements moved is min(VLEN/SEW * VLMUL, matrix_size).

- For matrix A, matrix_size = mtilem * mtilek.
- For matrix B, matrix_size = mtilek * mtilen.
- For matrix C, matrix_size = mtilem * mtilen.

```
# Data move between matrix register rows and vector registers.

# vd[(i - rs2) * mtilek + j] = md[i, j], i = rs2 .. rs2 + mtilem - 1
mmvare8.v.m   vd, ms1, rs2
mmvare16.v.m  vd, ms1, rs2
mmvare32.v.m  vd, ms1, rs2
mmvare64.v.m  vd, ms1, rs2

# vd[(i - rs2) * mtilen + j] = md[i, j], i = rs2 .. rs2 + mtilek - 1
mmvbre8.v.m   vd, ms1, rs2
mmvbre16.v.m  vd, ms1, rs2
mmvbre32.v.m  vd, ms1, rs2
mmvbre64.v.m  vd, ms1, rs2
```

```
# vd[(i - rs2) * mtilen + j] = md[i, j], i = rs2 .. rs2 + mtilem - 1
mmvcre8.v.m   vd, ms1, rs2
mmvcre16.v.m  vd, ms1, rs2
mmvcre32.v.m  vd, ms1, rs2
mmvcre64.v.m  vd, ms1, rs2

# md[i, j] = vd[(i - rs2) * mtilek + j], i = rs2 .. rs2 + mtilem - 1
mmvare8.m.v   md, vs1, rs2
mmvare16.m.v  md, vs1, rs2
mmvare32.m.v  md, vs1, rs2
mmvare64.m.v  md, vs1, rs2

# md[i, j] = vd[(i - rs2) * mtilen + j], i = rs2 .. rs2 + mtilek - 1
mmvbre8.m.v   md, vs1, rs2
mmvbre16.m.v  md, vs1, rs2
mmvbre32.m.v  md, vs1, rs2
mmvbre64.m.v  md, vs1, rs2

# md[i, j] = vd[(i - rs2) * mtilen + j], i = rs2 .. rs2 + mtilem - 1
mmvcre8.m.v   md, vs1, rs2
mmvcre16.m.v  md, vs1, rs2
mmvcre32.m.v  md, vs1, rs2
mmvcre64.m.v  md, vs1, rs2

# Data move between matrix register columns and vector registers.

# vd[(j - rs2) * mtilem + i] = md[i, j], j = rs2 .. rs2 + mtilek - 1
mmvace8.v.m   vd, ms1, rs2
mmvace16.v.m  vd, ms1, rs2
mmvace32.v.m  vd, ms1, rs2
mmvace64.v.m  vd, ms1, rs2

# vd[(j - rs2) * mtilek + i] = md[i, j], j = rs2 .. rs2 + mtilen - 1
mmvbce8.v.m   vd, ms1, rs2
mmvbce16.v.m  vd, ms1, rs2
mmvbce32.v.m  vd, ms1, rs2
mmvbce64.v.m  vd, ms1, rs2

# vd[(j - rs2) * mtilem + i] = md[i, j], j = rs2 .. rs2 + mtilen - 1
mmvcce8.v.m   vd, ms1, rs2
mmvcce16.v.m  vd, ms1, rs2
mmvcce32.v.m  vd, ms1, rs2
mmvcce64.v.m  vd, ms1, rs2

# md[i, j] = vd[(j - rs2) * mtilem + i], j = rs2 .. rs2 + mtilek - 1
mmvace8.m.v   md, vs1, rs2
mmvace16.m.v  md, vs1, rs2
mmvace32.m.v  md, vs1, rs2
```

```
mmvace64.m.v  md, vs1, rs2

# md[i, j] = vd[(j - rs2) * mtilek + i], j = rs2 .. rs2 + mtilen - 1
mmvbce8.m.v   md, vs1, rs2
mmvbce16.m.v  md, vs1, rs2
mmvbce32.m.v  md, vs1, rs2
mmvbce64.m.v  md, vs1, rs2

# md[i, j] = vd[(j - rs2) * mtilem + i], j = rs2 .. rs2 + mtilen - 1
mmvcce8.m.v   md, vs1, rs2
mmvcce16.m.v  md, vs1, rs2
mmvcce32.m.v  md, vs1, rs2
mmvcce64.m.v  md, vs1, rs2
```

6.15.4. Intrinsic Example: Matrix multiplication fused with element-wise vector operation

```
void fused_matmul_relu_float16(c, a, b, m, k, n) {
    msettype(e16);                             // use 16bit input matrix element
    for (i = 0; i < m; i += tile_m) {          // loop at dim m with tiling
        tile_m = msettilem(m-i);
        for (j = 0; j < n; j += tile_n) {      // loop at dim n with tiling
            tile_n = msettilen(n-j);

            out = mwsub_mm(out, out)           // clear acc reg
            for (s = 0; s < k; s += tile_k) {  // loop at dim k with tiling
                tile_k = msettilek(k-s);

                tr1 = mlae16_m(&a[i][s]);      // load left matrix a
                tr2 = mlbe16_m(&b[s][j]);      // load right matrix b
                out = mfwma_mm(tr1, tr2);      // tiled matrix multiply,
                                               // double widen output
            }

            out = mfncvt_f_fw_m(out, m2);      // convert widen result to single

            for (s = 0; s < tile_m; s += rows) {
                // max rows could move into 8 vregs
                rows = min(tile_m - s, 8*vlenb/rlenb);
                vsetvl(tile_n*rows, e16, m8);

                v1 = mmvcr_v_m(out, s);        // move out rows to vreg
                v1 = vfmax_vf(0.f, v1);        // vfmax.vf for relu

                msce16_v(v1, &c[i+s][j], n);   // store output tile slices
            }
        }
    }
}
```

6.16. Zmi2c: Im2col Extension

Im2col stands for Image to Column, and is an implementation technique of computing Convolution operation (in Machine Learning) using GEMM operations.

The Zmi2c extension allows to perform im2col operation on-the-fly, by a set of new load instructions.

The **Load Unfold** instructions allows to load and extract sliding local blocks from memory into the matrix tile registers.

6.16.1. CSRs

The matrix extension adds 7 unprivileged CSRs (moutsh, minsh, mpad, mstdi, minsk, moutsk, mpadval) to the base scalar RISC-V ISA.

Table 19. New matrix CSRs

Address	Privilege	Name	Description
0xC47	URO	moutsh	Fold/unfold output shape.
0xC48	URO	minsh	Fold/unfold input shape.
0xC49	URO	mpad	Fold/unfold padding parameters.
0xC4A	URO	mstdi	Fold/unfold sliding strides and dilations.
0xC4B	URO	minsk	Fold/unfold sliding kernel position of input.
0xC4C	URO	moutsk	Fold/unfold sliding kernel position of output.
0xC4D	URO	mpadval	Fold/unfold padding value, default to zero.

Table 20. `minsh` `moutsh` register layout

Bits	Name	Description
XLEN:32	0	Reserved
31:16	shape[1]	shape of dim 1, height
15:0	shape[0]	shape of dim 0, width

Table 21. `mpad` register layout

Bits	Name	Description
XLEN:32	0	Reserved
31:24	mpad_top	Padding added to up side of input
23:16	mpad_bottom	Padding added to bottom side of input
15:8	mpad_left	Padding added to left side of input
7:0	mpad_right	Padding added to left side of input

Table 22. `mstdi` *register layout*

Bits	Name	Description
XLEN:32	0	Reserved
31:24	mdil_h	Height spacing of the kernel elements
23:16	mdil_w	Weight spacing of the kernel elements
15:8	mstr_h	Height stride of the convolution
7:0	mstr_w	Weight stride of the convolution

Table 23. `minsk` `moutsk` *register layout*

Bits	Name	Description
XLEN:32	0	Reserved
31:16	msk[1]	Sliding kernel position of dim 1, height
15:0	msk[0]	Sliding kernel position of dim 0, width

6.16.2. Configuration Instructions

```
msetoutsh  rd, rs1, rs2 # set output shape(rs1), strides and dilations(rs2)
msetinsh   rd, rs1, rs2 # set input shape(rs1) and padding(rs2)
msetsk     rd, rs1, rs2 # set fold/unfold sliding positions, insk(rs1), outsk(rs2)
msetpadval rd, rs1      # set fold/unfold padding value
```

6.16.3. Load Unfold Instructions

The **Load Unfold** instructions allows to load and extract sliding local blocks from memory into the matrix tile registers. Similar to PyTorch, for the case of two input spatial dimensions this operation is sometimes called `im2col`.

```
# md destination, rs1 base address, rs2 row byte stride
# for left matrix, a
mlufae8.m    md, (rs1), rs2
mlufae16.m   md, (rs1), rs2
mlufae32.m   md, (rs1), rs2
mlufae64.m   md, (rs1), rs2

# for left matrix, b
mlufbe8.m    md, (rs1), rs2
mlufbe16.m   md, (rs1), rs2
mlufbe32.m   md, (rs1), rs2
mlufbe64.m   md, (rs1), rs2

# for left matrix, c
mlufce8.m    md, (rs1), rs2
mlufce16.m   md, (rs1), rs2
mlufce32.m   md, (rs1), rs2
mlufce64.m   md, (rs1), rs2
```

6.16.4. Intrinsic Example: Conv2D

```
void conv2d_float16(c, a, b, outh, outw, outc, inh, inw, inc,
        kh, kw, pt, pb, pl, pr, sw, dh, dw) {
    m = outh * outw;
    k = kh * kw * inc;
    n = outc;

    msettype(e16);      // use 16bit input matrix element

    // set in/out shape, sliding strides and dilations, and padding
    msetoutsh(outh << 16 | outw, dh << 24 | dw << 16 | sh << 8 | sw);
    msetinsh(inh << 16 | inw, pt << 24 | pb << 16 | pl << 8 | pr);

    for (i = 0; i < m; i += tile_m) {              // loop at dim m with tiling
        tile_m = msettilem(m-i);

        outh_pos = i / outw;
        outw_pos = i - outh_pos * outw;

        for (j = 0; j < n; j += tile_n) {          // loop at dim n with tiling
            tile_n = msettilen(n-j);

            out = mwsub_mm(out, out)               // clear output reg
            for (skh = 0; skh < kh; skh++) {       // loop for kernel height
                inh_pos = outh_pos * sh - pt + skh * dh;
                for (skw = 0; skw < kw; skw++) {    // loop for kernel width
                    inw_pos = outw_pos * sw - pl + skw * dw;
                    // set sliding position
                    msetsk(inh_pos << 16 | inw_pos, skw * dw << 16 | outw_pos)

                    // loop for kernel channels
                    for (skc = 0; skc < inc; skc += tile_k) {
                        tile_k = msettilek(inc-skc);

                        tr1 = mlufae16_m(&a[inh_pos][inw_pos][skc]);
                                                    // load and unfold input blocks
                        tr2 = mlbe16_m(&b[s][j]);   // load right matrix b
                        out = mfwma_mm(tr1, tr2);   // tiled matrix multiply,
                                                    // double widen output
                    }
                }
            }

            out = mfncvt_f_fw_m(out, m2);           // convert widen result
            msce16_m(out, &c[i][j], n*2);           // store to matrix c
        }
    }
}
```

6.16.5. Intrinsic Example: Conv3D

```
void conv3d_float16(c, a, b, outh, outw, outc, ind, inh, inw, inc,
        kd, kh, kw, pt, pb, pl, pr, sw, dh, dw) {
    m = outh * outw;
    k = kd * kh * kw * inc;
    n = outc;

    msettype(e16);      // use 16bit input matrix element

    // set in/out shape, sliding strides and dilations, and padding
    msetoutsh(outh << 16 | outw, dh << 24 | dw << 16 | sh << 8 | sw);
    msetinsh(inh << 16 | inw, pt << 24 | pb << 16 | pl << 8 | pr);

    for (i = 0; i < m; i += tile_m) {               // loop at dim m with tiling
        tile_m = msettilem(m-i);

        outh_pos = i / outw;
        outw_pos = i - outh_pos * outw;

        for (j = 0; j < n; j += tile_n) {           // loop at dim n with tiling
            tile_n = msettilen(n-j);

            out = mwsub_mm(out, out)                // clear output reg
            for (skd = 0; skd < kd; skd++) {        // loop for kernel *depth*
                for (skh = 0; skh < kh; skh++) {    // loop for kernel height
                    inh_pos = outh_pos * sh - pt + skh * dh;
                    for (skw = 0; skw < kw; skw++) {    // loop for kernel width
                        inw_pos = outw_pos * sw - pl + skw * dw;

                        msetsk(inh_pos << 16 | inw_pos, skw * dw << 16 | outw_pos)
                                                        // set sliding position

                        for (skc = 0; skc < inc; skc += tile_k) {
                            tile_k = msettilek(inc-skc);

                            tr1 = mlufae16_m(&a[skd][inh_pos][inw_pos][skc]);
                                                        // load and unfold blocks
                            tr2 = mlbe16_m(&b[s][j]);   // load right matrix b
                            out = mfwma_mm(tr1, tr2);   // tiled matrix multiply,
                                                        // double widen output
                        }
                    }
                }
            }

            out = mfncvt_f_fw_m(out, m2);   // convert widen result
            msce16_m(out, &c[i][j], n*2);   // store to matrix c
        }
    }
}
```

6.16.6. Intrinsic Example: MaxPool2D

```
void maxpool2d_float16(out, in, outh, outw, outc, inh, inw, inc,
        kh, kw, pt, pb, pl, pr, sw, dh, dw) {
    m = outh * outw;
    n = outc;

    msettype(e16);      // use 16bit input matrix element

    // set in/out shape, sliding strides and dilations, and padding
    msetoutsh(outh << 16 | outw, dh << 24 | dw << 16 | sh << 8 | sw);
    msetinsh(inh << 16 | inw, pt << 24 | pb << 16 | pl << 8 | pr);

    for (i = 0; i < m; i += tile_m) {           // loop at dim m with tiling
        tile_m = msettilem(m-i);

        outh_pos = i / outw;
        outw_pos = i - outh_pos * outw;

        for (j = 0; j < n; j += tile_n) {       // loop at dim n with tiling
            tile_n = msettilen(n-j);

            m_out = mfmv_s_f(tr_out, -inf)     // move -inf to output reg
            m_out = mbcce_m (tr_out)           // fill -inf to output reg
            for (skh = 0; skh < kh; skh++) {    // loop for kernel height
                inh_pos = outh_pos * sh - pt + skh * dh;
                for (skw = 0; skw < kw; skw++) {        // loop for kernel width
                    inw_pos = outw_pos * sw - pl + skw * dw;

                    msetsk(inh_pos << 16 | inw_pos, skw * dw << 16 | outw_pos)
                                                    // set sliding position

                    // load and unfold matrix blocks
                    m_in = mlufce16_m(&in[inh_pos][inw_pos][j]);
                    m_out = mfmax_mm(m_out, m_in);
                }
            }

            msce16_m(tr_out, &out[i][j], n*2);  // store to matrix c
        }
    }
}
```

6.16.7. Intrinsic Example: AvgPool2D

```
void avgpool2d_float16(out, in, outh, outw, outc, inh, inw, inc,
        kh, kw, pt, pb, pl, pr, sw, dh, dw) {
    m = outh * outw;
    n = outc;

    msettype(e16);      // use 16bit input matrix element

    // set in/out shape, sliding strides and dilations, and padding
    msetoutsh(outh << 16 | outw, dh << 24 | dw << 16 | sh << 8 | sw);
    msetinsh(inh << 16 | inw, pt << 24 | pb << 16 | pl << 8 | pr);

    // set divider
    m_div = mfmv_s_f(m_div, kh*kw)
    m_div = mbcce_m (m_div)

    for (i = 0; i < m; i += tile_m) {   // loop at dim m with tiling
        tile_m = msettilem(m-i);

        outh_pos = i / outw;
        outw_pos = i - outh_pos * outw;

        for (j = 0; j < n; j += tile_n) {   // loop at dim n with tiling
            tile_n = msettilen(n-j);

            m_out = mwsub_mm(m_out, m_out)          // clear output reg
            for (skh = 0; skh < kh; skh++) {        // loop for kernel height
                inh_pos = outh_pos * sh - pt + skh * dh;
                for (skw = 0; skw < kw; skw++) {    // loop for kernel width
                    inw_pos = outw_pos * sw - pl + skw * dw;

                    msetsk(inh_pos << 16 | inw_pos, skw * dw << 16 | outw_pos)
                                                    // set sliding position

                    // load and unfold matrix blocks
                    m_in = mlufce16_m(&in[inh_pos][inw_pos][j]);
                    m_out = mfadd_mm(m_out, m_in);
                }
            }

            m_out = mfdiv_mm(m_out, m_div);
            msce16_m(m_out, &out[i][j], n*2);      // store to matrix c
        }
    }
}
```

6.17. Zmc2i: Col2im Extension

The Zmc2i extension allows to perform Column to Image operation on-the-fly, by a set of new store instructions.

6.17.1. CSRs

The Zmc2i extension reuses 7 unprivileged CSRs (moutsh, minsh, mpad, mstdi, minsk, moutsk, mpadval) of Zmi2c.

6.17.2. Configuration Instructions

The Zmc2i extension reuses all configuration instructions of Zmi2c.

6.17.3. Store Fold Instructions

The **Store Fold** instructions allows to store and combine an array of sliding local blocks from the matrix tile regstiers into memory. Similar to PyTorch, for the case of two output spatial dimensions this operation is sometimes called `col2im`.

```
# ms3 destination, rs1 base address, rs2 row byte stride
# for left matrix, a
msfdae8.m    ms3, (rs1), rs2
msfdae16.m   ms3, (rs1), rs2
msfdae32.m   ms3, (rs1), rs2
msfdae64.m   ms3, (rs1), rs2

# for left matrix, b
msfdbe8.m    ms3, (rs1), rs2
msfdbe16.m   ms3, (rs1), rs2
msfdbe32.m   ms3, (rs1), rs2
msfdbe64.m   ms3, (rs1), rs2

# for left matrix, c
msfdce8.m    ms3, (rs1), rs2
msfdce16.m   ms3, (rs1), rs2
msfdce32.m   ms3, (rs1), rs2
msfdce64.m   ms3, (rs1), rs2
```

6.18. Zmsp*: Matrix Sparsity Extension

The Zmspa extension allows to perform 2:4 sparsing for left matrix.

The Zmspb extension allows to perform 2:4 sparsing for right matrix.

The Zmsp* extension adds two unprivileged CSRs, two configuration instructions, and two groups of matrix multiplication instructions, both for left matrix and right matrix.

Table 24. Sparsity CSRs

Address	Privilege	Name	Description
0xC4E	URO	mtsp	Source tile register for sparsing index.
0xC4F	URO	mdsp	The direction of sparsity (0 for row and 1 for column).

6.18.1. Configuration Instructions

The Zmsp* extension adds two configuration instruction to configure the source index register and sparsity direction.

```
# Set sparsing index source register.
msettspi  rd, imm   # rd = new mtsp, imm = register index
msettsp   rd, rs1   # rd = new mtsp, rs1 = register index

# Set sparsity direction.
msetdspi  rd, imm   # rd = new mdsp, imm = direction
msetdsp   rd, rs1   # rd = new mdsp, rs1 = direction
```

An implementation may support one of sparsity directions or both two directions. The `msetdsp[i]` always returns the supported direction.

6.18.2. Matrix Multiplication Instructions

The Zmspa extension adds a group of matrix multiplication instructions for left matrix sparsity.

```
# Unigned integer sparsing matrix multiplication and add, md = md + ms1 * ms2.
mmau.spa.[dw].mm   md, ms1, ms2     # left matrix is sparsing
mmau.spa.[w].mm    md, ms1, ms2     # left matrix is sparsing
mmau.spa.[h].mm    md, ms1, ms2     # left matrix is sparsing
mqmau.spa.[b].mm   md, ms1, ms2     # left matrix is sparsing
momau.spa.[hb].mm  md, ms1, ms2     # left matrix is sparsing

msmau.spa.[dw].mm  md, ms1, ms2     # left matrix is sparsing
msmau.spa.[w].mm   md, ms1, ms2     # left matrix is sparsing
msmau.spa.[h].mm   md, ms1, ms2     # left matrix is sparsing
msqmau.spa.[b].mm  md, ms1, ms2     # left matrix is sparsing
msomau.spa.[hb].mm md, ms1, ms2     # left matrix is sparsing

# Signed integer sparsing matrix multiplication and add, md = md + ms1 * ms2.
mma.spa.[dw].mm    md, ms1, ms2     # left matrix is sparsing
mma.spa.[w].mm     md, ms1, ms2     # left matrix is sparsing
mma.spa.[h].mm     md, ms1, ms2     # left matrix is sparsing
mqma.spa.[b].mm    md, ms1, ms2     # left matrix is sparsing
moma.spa.[hb].mm   md, ms1, ms2     # left matrix is sparsing

msma.spa.[dw].mm   md, ms1, ms2     # left matrix is sparsing
msma.spa.[w].mm    md, ms1, ms2     # left matrix is sparsing
msma.spa.[h].mm    md, ms1, ms2     # left matrix is sparsing
msqma.spa.[b].mm   md, ms1, ms2     # left matrix is sparsing
msoma.spa.[hb].mm  md, ms1, ms2     # left matrix is sparsing

# Float point sparsing matrix multiplication and add, md = md + ms1 * ms2.
mfma.spa.[d].mm    md, ms1, ms2     # left matrix is sparsing
mfma.spa.[f].mm    md, ms1, ms2     # left matrix is sparsing
mfma.spa.[hf].mm   md, ms1, ms2     # left matrix is sparsing

mfwma.spa.[f].mm   md, ms1, ms2     # left matrix is sparsing
mfwma.spa.[hf].mm  md, ms1, ms2     # left matrix is sparsing
mfwma.spa.[cf].mm  md, ms1, ms2     # left matrix is sparsing
mfqma.spa.[cf].mm  md, ms1, ms2     # left matrix is sparsing
```

The Zmspb extension adds a group of matrix multiplication instructions for right matrix sparsity.

```
# Unigned integer sparsing matrix multiplication and add, md = md + ms1 * ms2.
mmau.spb.[dw].mm   md, ms1, ms2     # right matrix is sparsing
mmau.spb.[w].mm    md, ms1, ms2     # right matrix is sparsing
mmau.spb.[h].mm    md, ms1, ms2     # right matrix is sparsing
mqmau.spb.[b].mm   md, ms1, ms2     # right matrix is sparsing
momau.spb.[hb].mm  md, ms1, ms2     # right matrix is sparsing

msmau.spb.[dw].mm  md, ms1, ms2     # right matrix is sparsing
msmau.spb.[w].mm   md, ms1, ms2     # right matrix is sparsing
msmau.spb.[h].mm   md, ms1, ms2     # right matrix is sparsing
msqmau.spb.[b].mm  md, ms1, ms2     # right matrix is sparsing
msomau.spb.[hb].mm md, ms1, ms2     # right matrix is sparsing

# Signed integer sparsing matrix multiplication and add, md = md + ms1 * ms2.
mma.spb.[dw].mm    md, ms1, ms2     # right matrix is sparsing
mma.spb.[w].mm     md, ms1, ms2     # right matrix is sparsing
mma.spb.[h].mm     md, ms1, ms2     # right matrix is sparsing
mqma.spb.[b].mm    md, ms1, ms2     # right matrix is sparsing
moma.spb.[hb].mm   md, ms1, ms2     # right matrix is sparsing

msma.spb.[dw].mm   md, ms1, ms2     # right matrix is sparsing
msma.spb.[w].mm    md, ms1, ms2     # right matrix is sparsing
msma.spb.[h].mm    md, ms1, ms2     # right matrix is sparsing
msqma.spb.[b].mm   md, ms1, ms2     # right matrix is sparsing
msoma.spb.[hb].mm  md, ms1, ms2     # right matrix is sparsing

# Float point sparsing matrix multiplication and add, md = md + ms1 * ms2.
mfma.spb.[d].mm    md, ms1, ms2     # right matrix is sparsing
mfma.spb.[f].mm    md, ms1, ms2     # right matrix is sparsing
mfma.spb.[hf].mm   md, ms1, ms2     # right matrix is sparsing

mfwma.spb.[f].mm   md, ms1, ms2     # right matrix is sparsing
mfwma.spb.[hf].mm  md, ms1, ms2     # right matrix is sparsing
mfwma.spb.[cf].mm  md, ms1, ms2     # right matrix is sparsing
mfqma.spb.[cf].mm  md, ms1, ms2     # right matrix is sparsing
```

Chapter 7. Matrix Instruction Listing

Table 25. Configuration Instructions

Format	31 26	25	24 20	19 15	14 12	11 7	6 0
	funct6	**im**	**imm**	**rs1**	**funct3**	**rd**	**opcode**
msettype	000000	0	00000	rs1	100	rd	OP-M32
msettypei	000000	1	imm[9:0]		100	rd	OP-M32
msettypehi	000000	1	imm[9:0]		101	rd	OP-M32
msetsew	000000	1	setval	field	110	rd	OP-M32
msetint	000000	1	setval	field	110	rd	OP-M32
munsetint	000000	1	setval	field	110	rd	OP-M32
msetfp	000000	1	setval	field	110	rd	OP-M32
munsetfp	000000	1	setval	field	110	rd	OP-M32
msetba	000000	1	setval	field	110	rd	OP-M32
msettilem	000001	0	00000	rs1	101	rd	OP-M32
msettilemi	000001	1	imm[9:0]		101	rd	OP-M32
msettilek	000001	0	00000	rs1	110	rd	OP-M32
msettileki	000001	1	imm[9:0]		110	rd	OP-M32
msettilen	000001	0	00000	rs1	100	rd	OP-M32
msettileni	000001	1	imm[9:0]		100	rd	OP-M32
msettsp	000000	0	00000	rs1	111	rd	OP-M32
msettspi	000000	1	00000	imm	111	rd	OP-M32
msetdsp	000001	0	00000	rs1	111	rd	OP-M32
msetdspi	000001	1	00000	imm	111	rd	OP-M32
msetoutsh	000010	0	rs2	rs1	100	rd	OP-M32
msetinsh	000010	0	rs2	rs1	101	rd	OP-M32
msetsk	000010	0	rs2	rs1	110	rd	OP-M32
msetpadval	000010	0	00000	rs1	111	rd	OP-M32

Table 26. Load/Store Instructions

Format	31 26	25	24 20	19 15	14 12	11	10 7	6 0
	funct6	**ls**	**rs2**	**rs1**	**eew**	**tr**	**md**	**opcode**
mlae8.m	000001	0	rs2	rs1	000	0	md	OP-M32
mlae16.m	000001	0	rs2	rs1	001	0	md	OP-M32
mlae32.m	000001	0	rs2	rs1	010	0	md	OP-M32
mlae64.m	000001	0	rs2	rs1	011	0	md	OP-M32
mlbe8.m	000010	0	rs2	rs1	000	0	md	OP-M32
mlbe16.m	000010	0	rs2	rs1	001	0	md	OP-M32
mlbe32.m	000010	0	rs2	rs1	010	0	md	OP-M32
mlbe64.m	000010	0	rs2	rs1	011	0	md	OP-M32
mlce8.m	000000	0	rs2	rs1	000	0	md	OP-M32
mlce16.m	000000	0	rs2	rs1	001	0	md	OP-M32
mlce32.m	000000	0	rs2	rs1	010	0	md	OP-M32
mlce64.m	000000	0	rs2	rs1	011	0	md	OP-M32
mltre8.m	000011	0	rs2	rs1	000	0	md	OP-M32
mltre16.m	000011	0	rs2	rs1	001	0	md	OP-M32
mltre32.m	000011	0	rs2	rs1	010	0	md	OP-M32
mltre64.m	000011	0	rs2	rs1	011	0	md	OP-M32
mlate8.m	000001	0	rs2	rs1	000	1	md	OP-M32
mlate16.m	000001	0	rs2	rs1	001	1	md	OP-M32
mlate32.m	000001	0	rs2	rs1	010	1	md	OP-M32
mlate64.m	000001	0	rs2	rs1	011	1	md	OP-M32
mlbte8.m	000010	0	rs2	rs1	000	1	md	OP-M32
mlbte16.m	000010	0	rs2	rs1	001	1	md	OP-M32
mlbte32.m	000010	0	rs2	rs1	010	1	md	OP-M32
mlbte64.m	000010	0	rs2	rs1	011	1	md	OP-M32
mlcte8.m	000000	0	rs2	rs1	000	1	md	OP-M32
mlcte16.m	000000	0	rs2	rs1	001	1	md	OP-M32
mlcte32.m	000000	0	rs2	rs1	010	1	md	OP-M32

mlcte64.m	000000	0	rs2	rs1	011	1	md	OP-M32
mlacce8.m	000011	0	rs2	rs1	000	1	md	OP-M32
mlacce16.m	000011	0	rs2	rs1	001	1	md	OP-M32
mlacce32.m	000011	0	rs2	rs1	010	1	md	OP-M32
mlacce64.m	000011	0	rs2	rs1	011	1	md	OP-M32
msae8.m	000001	1	rs2	rs1	000	0	ms3	OP-M32
msae16.m	000001	1	rs2	rs1	001	0	ms3	OP-M32
msae32.m	000001	1	rs2	rs1	010	0	ms3	OP-M32
msae64.m	000001	1	rs2	rs1	011	0	ms3	OP-M32
msbe8.m	000010	1	rs2	rs1	000	0	ms3	OP-M32
msbe16.m	000010	1	rs2	rs1	001	0	ms3	OP-M32
msbe32.m	000010	1	rs2	rs1	010	0	ms3	OP-M32
msbe64.m	000010	1	rs2	rs1	011	0	ms3	OP-M32
msce8.m	000000	1	rs2	rs1	000	0	ms3	OP-M32
msce16.m	000000	1	rs2	rs1	001	0	ms3	OP-M32
msce32.m	000000	1	rs2	rs1	010	0	ms3	OP-M32
msce64.m	000000	1	rs2	rs1	011	0	ms3	OP-M32
mstre8.m	000011	1	rs2	rs1	000	0	ms3	OP-M32
mstre16.m	000011	1	rs2	rs1	001	0	ms3	OP-M32
mstre32.m	000011	1	rs2	rs1	010	0	ms3	OP-M32
mstre64.m	000011	1	rs2	rs1	011	0	ms3	OP-M32
msate8.m	000001	1	rs2	rs1	000	1	ms3	OP-M32
msate16.m	000001	1	rs2	rs1	001	1	ms3	OP-M32
msate32.m	000001	1	rs2	rs1	010	1	ms3	OP-M32
msate64.m	000001	1	rs2	rs1	011	1	ms3	OP-M32
msbte8.m	000010	1	rs2	rs1	000	1	ms3	OP-M32
msbte16.m	000010	1	rs2	rs1	001	1	ms3	OP-M32
msbte32.m	000010	1	rs2	rs1	010	1	ms3	OP-M32
msbte64.m	000010	1	rs2	rs1	011	1	ms3	OP-M32

mscte8.m	000000	1	rs2	rs1	000	1	ms3	OP-M32
mscte16.m	000000	1	rs2	rs1	001	1	ms3	OP-M32
mscte32.m	000000	1	rs2	rs1	010	1	ms3	OP-M32
mscte64.m	000000	1	rs2	rs1	011	1	ms3	OP-M32
msacce8.m	000011	1	rs2	rs1	000	1	ms3	OP-M32
msacce16.m	000011	1	rs2	rs1	001	1	ms3	OP-M32
msacce32.m	000011	1	rs2	rs1	010	1	ms3	OP-M32
msacce64.m	000011	1	rs2	rs1	011	1	ms3	OP-M32
mlae8.v	100001	0	rs2	rs1	000	0	md	OP-M32
mlae16.v	100001	0	rs2	rs1	001	0	md	OP-M32
mlae32.v	100001	0	rs2	rs1	010	0	md	OP-M32
mlae64.v	100001	0	rs2	rs1	011	0	md	OP-M32
mlbe8.v	100010	0	rs2	rs1	000	0	md	OP-M32
mlbe16.v	100010	0	rs2	rs1	001	0	md	OP-M32
mlbe32.v	100010	0	rs2	rs1	010	0	md	OP-M32
mlbe64.v	100010	0	rs2	rs1	011	0	md	OP-M32
mlce8.v	100000	0	rs2	rs1	000	0	md	OP-M32
mlce16.v	100000	0	rs2	rs1	001	0	md	OP-M32
mlce32.v	100000	0	rs2	rs1	010	0	md	OP-M32
mlce64.v	100000	0	rs2	rs1	011	0	md	OP-M32
msae8.v	100001	1	rs2	rs1	000	0	ms3	OP-M32
msae16.v	100001	1	rs2	rs1	001	0	ms3	OP-M32
msae32.v	100001	1	rs2	rs1	010	0	ms3	OP-M32
msae64.v	100001	1	rs2	rs1	011	0	ms3	OP-M32
msbe8.v	100010	1	rs2	rs1	000	0	ms3	OP-M32
msbe16.v	100010	1	rs2	rs1	001	0	ms3	OP-M32
msbe32.v	100010	1	rs2	rs1	010	0	ms3	OP-M32
msbe64.v	100010	1	rs2	rs1	011	0	ms3	OP-M32
msce8.v	100000	1	rs2	rs1	000	0	ms3	OP-M32

msce16.v	100000	1	rs2	rs1	001	0	ms3	OP-M32
msce32.v	100000	1	rs2	rs1	010	0	ms3	OP-M32
msce64.v	100000	1	rs2	rs1	011	0	ms3	OP-M32
mlufae8.m	100001	0	rs2	rs1	000	1	md	OP-M32
mlufae16.m	100001	0	rs2	rs1	001	1	md	OP-M32
mlufae32.m	100001	0	rs2	rs1	010	1	md	OP-M32
mlufae64.m	100001	0	rs2	rs1	011	1	md	OP-M32
mlufbe8.m	100010	0	rs2	rs1	000	1	md	OP-M32
mlufbe16.m	100010	0	rs2	rs1	001	1	md	OP-M32
mlufbe32.m	100010	0	rs2	rs1	010	1	md	OP-M32
mlufbe64.m	100010	0	rs2	rs1	011	1	md	OP-M32
mlufce8.m	100000	0	rs2	rs1	000	1	md	OP-M32
mlufce16.m	100000	0	rs2	rs1	001	1	md	OP-M32
mlufce32.m	100000	0	rs2	rs1	010	1	md	OP-M32
mlufce64.m	100000	0	rs2	rs1	011	1	md	OP-M32
msfdae8.m	100001	1	rs2	rs1	000	1	ms3	OP-M32
msfdae16.m	100001	1	rs2	rs1	001	1	ms3	OP-M32
msfdae32.m	100001	1	rs2	rs1	010	1	ms3	OP-M32
msfdae64.m	100001	1	rs2	rs1	011	1	ms3	OP-M32
msfdbe8.m	100010	1	rs2	rs1	000	1	ms3	OP-M32
msfdbe16.m	100010	1	rs2	rs1	001	1	ms3	OP-M32
msfdbe32.m	100010	1	rs2	rs1	010	1	ms3	OP-M32
msfdbe64.m	100010	1	rs2	rs1	011	1	ms3	OP-M32
msfdce8.m	100000	1	rs2	rs1	000	1	ms3	OP-M32
msfdce16.m	100000	1	rs2	rs1	001	1	ms3	OP-M32
msfdce32.m	100000	1	rs2	rs1	010	1	ms3	OP-M32
msfdce64.m	100000	1	rs2	rs1	011	1	ms3	OP-M32

Table 27. Data Move Instructions

Format	31 26	25	24 20	19 15	14 12	11 7	6 0
	funct6	**di**	**rs2**	**rs1**	**eew**	**rd**	**opcode**
mmve8.a.t	000100	0	rs2	ms1	000	md	OP-M32
mmve16.a.t	000100	0	rs2	ms1	001	md	OP-M32
mmve32.a.t	000100	0	rs2	ms1	010	md	OP-M32
mmve64.a.t	000100	0	rs2	ms1	011	md	OP-M32
mmve8.t.a	000100	1	rs2	ms1	000	md	OP-M32
mmve16.t.a	000100	1	rs2	ms1	001	md	OP-M32
mmve32.t.a	000100	1	rs2	ms1	010	md	OP-M32
mmve64.t.a	000100	1	rs2	ms1	011	md	OP-M32
mmvie8.a.t	000100	0	imm	ms1	100	md	OP-M32
mmvie16.a.t	000100	0	imm	ms1	101	md	OP-M32
mmvie32.a.t	000100	0	imm	ms1	110	md	OP-M32
mmvie64.a.t	000100	0	imm	ms1	111	md	OP-M32
mmvie8.t.a	000100	1	imm	ms1	100	md	OP-M32
mmvie16.t.a	000100	1	imm	ms1	101	md	OP-M32
mmvie32.t.a	000100	1	imm	ms1	110	md	OP-M32
mmvie64.t.a	000100	1	imm	ms1	111	md	OP-M32
mmve8.x.t	000101	0	rs2	ms1	000	rd	OP-M32
mmve16.x.t	000101	0	rs2	ms1	001	rd	OP-M32
mmve32.x.t	000101	0	rs2	ms1	010	rd	OP-M32
mmve64.x.t	000101	0	rs2	ms1	011	rd	OP-M32
mmve8.t.x	000101	1	rs2	rs1	000	md	OP-M32
mmve16.t.x	000101	1	rs2	rs1	001	md	OP-M32
mmve32.t.x	000101	1	rs2	rs1	010	md	OP-M32
mmve64.t.x	000101	1	rs2	rs1	011	md	OP-M32
mmve8.x.a	000101	0	rs2	ms1	100	rd	OP-M32
mmve16.x.a	000101	0	rs2	ms1	101	rd	OP-M32
mmve32.x.a	000101	0	rs2	ms1	110	rd	OP-M32

mmve64.x.a	000101	0	rs2	ms1	111	rd	OP-M32
mmve8.a.x	000101	1	rs2	rs1	100	md	OP-M32
mmve16.a.x	000101	1	rs2	rs1	101	md	OP-M32
mmve32.a.x	000101	1	rs2	rs1	110	md	OP-M32
mmve64.a.x	000101	1	rs2	rs1	111	md	OP-M32
mfmve8.x.t	000110	0	rs2	ms1	000	rd	OP-M32
mfmve16.x.t	000110	0	rs2	ms1	001	rd	OP-M32
mfmve32.x.t	000110	0	rs2	ms1	010	rd	OP-M32
mfmve64.x.t	000110	0	rs2	ms1	011	rd	OP-M32
mfmve8.t.x	000110	1	rs2	rs1	000	md	OP-M32
mfmve16.t.x	000110	1	rs2	rs1	001	md	OP-M32
mfmve32.t.x	000110	1	rs2	rs1	010	md	OP-M32
mfmve64.t.x	000110	1	rs2	rs1	011	md	OP-M32
mfmve8.f.a	000110	0	rs2	ms1	100	rd	OP-M32
mfmve16.f.a	000110	0	rs2	ms1	101	rd	OP-M32
mfmve32.f.a	000110	0	rs2	ms1	110	rd	OP-M32
mfmve64.f.a	000110	0	rs2	ms1	111	rd	OP-M32
mfmve8.a.f	000110	1	rs2	rs1	100	md	OP-M32
mfmve16.a.f	000110	1	rs2	rs1	101	md	OP-M32
mfmve32.a.f	000110	1	rs2	rs1	110	md	OP-M32
mfmve64.a.f	000110	1	rs2	rs1	111	md	OP-M32
mmve8.t.t	000111	0	00000	ms1	000	md	OP-M32
mmve16.t.t	000111	0	00000	ms1	001	md	OP-M32
mmve32.t.t	000111	0	00000	ms1	010	md	OP-M32
mmve64.t.t	000111	0	00000	ms1	011	md	OP-M32
mmve8.a.a	000111	0	00001	ms1	000	md	OP-M32
mmve16.a.a	000111	0	00001	ms1	001	md	OP-M32
mmve32.a.a	000111	0	00001	ms1	010	md	OP-M32
mmve64.a.a	000111	0	00001	ms1	011	md	OP-M32

mbcar.m	000111	0	10001	ms1	000	md	OP-M32
mbcbr.m	000111	0	10010	ms1	000	md	OP-M32
mbccr.m	000111	0	10000	ms1	000	md	OP-M32
mbcace8.m	000111	0	10101	ms1	000	md	OP-M32
mbcace16.m	000111	0	10101	ms1	001	md	OP-M32
mbcace32.m	000111	0	10101	ms1	010	md	OP-M32
mbcace64.m	000111	0	10110	ms1	011	md	OP-M32
mbcbce8.m	000111	0	10110	ms1	000	md	OP-M32
mbcbce16.m	000111	0	10110	ms1	001	md	OP-M32
mbcbce32.m	000111	0	10110	ms1	010	md	OP-M32
mbcbce64.m	000111	0	10110	ms1	011	md	OP-M32
mbccce8.m	000111	0	10100	ms1	000	md	OP-M32
mbccce16.m	000111	0	10100	ms1	001	md	OP-M32
mbccce32.m	000111	0	10100	ms1	010	md	OP-M32
mbccce64.m	000111	0	10100	ms1	011	md	OP-M32
mbcaee8.m	000111	0	11001	ms1	000	md	OP-M32
mbcaee16.m	000111	0	11001	ms1	001	md	OP-M32
mbcaee32.m	000111	0	11001	ms1	010	md	OP-M32
mbcaee64.m	000111	0	11010	ms1	011	md	OP-M32
mbcbee8.m	000111	0	11010	ms1	000	md	OP-M32
mbcbee16.m	000111	0	11010	ms1	001	md	OP-M32
mbcbee32.m	000111	0	11010	ms1	010	md	OP-M32
mbcbee64.m	000111	0	11010	ms1	011	md	OP-M32
mbccee8.m	000111	0	11000	ms1	000	md	OP-M32
mbccee16.m	000111	0	11000	ms1	001	md	OP-M32
mbccee32.m	000111	0	11000	ms1	010	md	OP-M32
mbccee64.m	000111	0	11000	ms1	011	md	OP-M32
mtae8.m	000111	0	11101	ms1	000	md	OP-M32
mtae16.m	000111	0	11101	ms1	001	md	OP-M32

mtae32.m	000111	0	11101	ms1	010	md	OP-M32
mtae64.m	000111	0	11101	ms1	011	md	OP-M32
mtbe8.m	000111	0	11110	ms1	000	md	OP-M32
mtbe16.m	000111	0	11110	ms1	001	md	OP-M32
mtbe32.m	000111	0	11110	ms1	010	md	OP-M32
mtbe64.m	000111	0	11110	ms1	011	md	OP-M32
mtce8.m	000111	0	11100	ms1	000	md	OP-M32
mtce16.m	000111	0	11100	ms1	001	md	OP-M32
mtce32.m	000111	0	11100	ms1	010	md	OP-M32
mtce64.m	000111	0	11100	ms1	011	md	OP-M32

Table 28. Zmv Extension Data Move Instructions

Format	31 26	25	24 20	19	18 15	14 12	11	10 7	6 0
	funct6	**di**	**rs2**	**var**	**rs1**	**eew**	**var**	**rd**	**opcode**
mmvare8.v.m	100110	0	rs2	0	ms1	100	vd		OP-M32
mmvare16.v.m	100110	0	rs2	0	ms1	101	vd		OP-M32
mmvare32.v.m	100110	0	rs2	0	ms1	110	vd		OP-M32
mmvare64.v.m	100110	0	rs2	0	ms1	111	vd		OP-M32
mmvbre8.v.m	100110	0	rs2	1	ms1	100	vd		OP-M32
mmvbre16.v.m	100110	0	rs2	1	ms1	101	vd		OP-M32
mmvbre32.v.m	100110	0	rs2	1	ms1	110	vd		OP-M32
mmvbre64.v.m	100110	0	rs2	1	ms1	111	vd		OP-M32
mmvcre8.v.m	100110	0	rs2	0	ms1	000	vd		OP-M32
mmvcre16.v.m	100110	0	rs2	0	ms1	001	vd		OP-M32
mmvcre32.v.m	100110	0	rs2	0	ms1	010	vd		OP-M32
mmvcre64.v.m	100110	0	rs2	0	ms1	011	vd		OP-M32
mmvare8.m.v	100110	1	rs2	vs1		100	0	md	OP-M32
mmvare16.m.v	100110	1	rs2	vs1		101	0	md	OP-M32
mmvare32.m.v	100110	1	rs2	vs1		110	0	md	OP-M32
mmvare64.m.v	100110	1	rs2	vs1		111	0	md	OP-M32

mmvbre8.m.v	100110	1	rs2	vs1		100	1	md	OP-M32
mmvbre16.m.v	100110	1	rs2	vs1		101	1	md	OP-M32
mmvbre32.m.v	100110	1	rs2	vs1		110	1	md	OP-M32
mmvbre64.m.v	100110	1	rs2	vs1		111	1	md	OP-M32
mmvcre8.m.v	100110	1	rs2	vs1		000	0	md	OP-M32
mmvcre16.m.v	100110	1	rs2	vs1		001	0	md	OP-M32
mmvcre32.m.v	100110	1	rs2	vs1		010	0	md	OP-M32
mmvcre64.m.v	100110	1	rs2	vs1		011	0	md	OP-M32
mmvace8.v.m	100111	0	rs2	0	ms1	100	vd		OP-M32
mmvace16.v.m	100111	0	rs2	0	ms1	101	vd		OP-M32
mmvace32.v.m	100111	0	rs2	0	ms1	110	vd		OP-M32
mmvace64.v.m	100111	0	rs2	0	ms1	111	vd		OP-M32
mmvbce8.v.m	100111	0	rs2	1	ms1	100	vd		OP-M32
mmvbce16.v.m	100111	0	rs2	1	ms1	101	vd		OP-M32
mmvbce32.v.m	100111	0	rs2	1	ms1	110	vd		OP-M32
mmvbce64.v.m	100111	0	rs2	1	ms1	111	vd		OP-M32
mmvcce8.v.m	100111	0	rs2	0	ms1	000	vd		OP-M32
mmvcce16.v.m	100111	0	rs2	0	ms1	001	vd		OP-M32
mmvcce32.v.m	100111	0	rs2	0	ms1	010	vd		OP-M32
mmvcce64.v.m	100111	0	rs2	0	ms1	011	vd		OP-M32
mmvace8.m.v	100111	1	rs2	vs1		100	0	md	OP-M32
mmvace16.m.v	100111	1	rs2	vs1		101	0	md	OP-M32
mmvace32.m.v	100111	1	rs2	vs1		110	0	md	OP-M32
mmvace64.m.v	100111	1	rs2	vs1		111	0	md	OP-M32
mmvbce8.m.v	100111	1	rs2	vs1		100	1	md	OP-M32
mmvbce16.m.v	100111	1	rs2	vs1		101	1	md	OP-M32
mmvbce32.m.v	100111	1	rs2	vs1		110	1	md	OP-M32
mmvbce64.m.v	100111	1	rs2	vs1		111	1	md	OP-M32
mmvcce8.m.v	100111	1	rs2	vs1		000	0	md	OP-M32

mmvcce16.m.v	100111	1	rs2	vs1	001	0	md	OP-M32
mmvcce32.m.v	100111	1	rs2	vs1	010	0	md	OP-M32
mmvcce64.m.v	100111	1	rs2	vs1	011	0	md	OP-M32

Table 29. Matrix Multiplication Instructions

Format	31 26	25	24	23 20	19	18 15	14 12	11	10 7	6 0
	funct6	**fp**	**sa**	**ms2**	**sn**	**ms1**	**eew**	**ma**	**md**	**opcode**
mmau.mm	001000	0	0	ms2	0	ms1	100	1	md	OP-M32
mmau.h.mm	001000	0	0	ms2	0	ms1	001	1	md	OP-M32
mmau.w.mm	001000	0	0	ms2	0	ms1	010	1	md	OP-M32
mmau.dw.mm	001000	0	0	ms2	0	ms1	011	1	md	OP-M32
msmau.mm	001000	0	1	ms2	0	ms1	100	1	md	OP-M32
msmau.h.mm	001000	0	1	ms2	0	ms1	001	1	md	OP-M32
msmau.w.mm	001000	0	1	ms2	0	ms1	010	1	md	OP-M32
msmau.dw.mm	001000	0	1	ms2	0	ms1	011	1	md	OP-M32
mma.mm	001000	0	0	ms2	1	ms1	100	1	md	OP-M32
mma.h.mm	001000	0	0	ms2	1	ms1	001	1	md	OP-M32
mma.w.mm	001000	0	0	ms2	1	ms1	010	1	md	OP-M32
mma.dw.mm	001000	0	0	ms2	1	ms1	011	1	md	OP-M32
msma.mm	001000	0	1	ms2	1	ms1	100	1	md	OP-M32
msma.h.mm	001000	0	1	ms2	1	ms1	001	1	md	OP-M32
msma.w.mm	001000	0	1	ms2	1	ms1	010	1	md	OP-M32
msma.dw.mm	001000	0	1	ms2	1	ms1	011	1	md	OP-M32
mfma.mm	001000	1	0	ms2	0	ms1	100	1	md	OP-M32
mfma.hf.mm	001000	1	0	ms2	0	ms1	001	1	md	OP-M32
mfma.f.mm	001000	1	0	ms2	0	ms1	010	1	md	OP-M32
mfma.d.mm	001000	1	0	ms2	0	ms1	011	1	md	OP-M32
mwmau.mm	001001	0	0	ms2	0	ms1	100	1	md	OP-M32
mwmau.h.mm	001001	0	0	ms2	0	ms1	001	1	md	OP-M32
mwmau.w.mm	001001	0	0	ms2	0	ms1	010	1	md	OP-M32

mswmau.mm	001001	0	1	ms2	0	ms1	100	1	md	OP-M32
mswmau.h.mm	001001	0	1	ms2	0	ms1	001	1	md	OP-M32
mswmau.w.mm	001001	0	1	ms2	0	ms1	010	1	md	OP-M32
mwma.mm	001001	0	0	ms2	1	ms1	100	1	md	OP-M32
mwma.h.mm	001001	0	0	ms2	1	ms1	001	1	md	OP-M32
mwma.w.mm	001001	0	0	ms2	1	ms1	010	1	md	OP-M32
mswma.mm	001001	0	1	ms2	1	ms1	100	1	md	OP-M32
mswma.h.mm	001001	0	1	ms2	1	ms1	001	1	md	OP-M32
mswma.w.mm	001001	0	1	ms2	1	ms1	010	1	md	OP-M32
mfwma.mm	001001	1	0	ms2	0	ms1	100	1	md	OP-M32
mfwma.cf.mm	001001	1	0	ms2	0	ms1	000	1	md	OP-M32
mfwma.hf.mm	001001	1	0	ms2	0	ms1	001	1	md	OP-M32
mfwma.f.mm	001001	1	0	ms2	0	ms1	010	1	md	OP-M32
mqmau.mm	001010	0	0	ms2	0	ms1	100	1	md	OP-M32
mqmau.b.mm	001010	0	0	ms2	0	ms1	000	1	md	OP-M32
msqmau.mm	001010	0	1	ms2	0	ms1	100	1	md	OP-M32
msqmau.b.mm	001010	0	1	ms2	0	ms1	000	1	md	OP-M32
mqma.mm	001010	0	0	ms2	1	ms1	100	1	md	OP-M32
mqma.b.mm	001010	0	0	ms2	1	ms1	000	1	md	OP-M32
msqma.mm	001010	0	1	ms2	1	ms1	100	1	md	OP-M32
msqma.b.mm	001010	0	1	ms2	1	ms1	000	1	md	OP-M32
mfqma.mm	001010	1	0	ms2	1	ms1	100	1	md	OP-M32
mfqma.cf.mm	001010	1	0	ms2	1	ms1	000	1	md	OP-M32
momau.mm	001011	0	0	ms2	0	ms1	100	1	md	OP-M32
momau.hb.mm	001011	0	0	ms2	0	ms1	111	1	md	OP-M32
msomau.mm	001011	0	1	ms2	0	ms1	100	1	md	OP-M32
msomau.hb.mm	001011	0	1	ms2	0	ms1	111	1	md	OP-M32
moma.mm	001011	0	0	ms2	1	ms1	100	1	md	OP-M32
moma.hb.mm	001011	0	0	ms2	1	ms1	111	1	md	OP-M32

msoma.mm	001011	0	1	ms2	1	ms1	100	1	md	OP-M32
msoma.hb.mm	001011	0	1	ms2	1	ms1	111	1	md	OP-M32

Table 30. Sparsing Matrix Multiplication Instructions

Format	31 26	25	24	23 20	19	18 15	14 12	11	10 7	6 0
	funct6	**fp**	**sa**	**ms2**	**sn**	**ms1**	**eew**	**sp**	**md**	**opcode**
mmau.spa.mm	101000	0	0	ms2	0	ms1	100	0	md	OP-M32
mmau.spb.mm	101000	0	0	ms2	0	ms1	100	1	md	OP-M32
mmau.spa.h.mm	101000	0	0	ms2	0	ms1	001	0	md	OP-M32
mmau.spb.h.mm	101000	0	0	ms2	0	ms1	001	1	md	OP-M32
mmau.spa.w.mm	101000	0	0	ms2	0	ms1	010	0	md	OP-M32
mmau.spb.w.mm	101000	0	0	ms2	0	ms1	010	1	md	OP-M32
mmau.spa.dw.mm	101000	0	0	ms2	0	ms1	011	0	md	OP-M32
mmau.spb.dw.mm	101000	0	0	ms2	0	ms1	011	1	md	OP-M32
msmau.spa.mm	101000	0	1	ms2	0	ms1	100	0	md	OP-M32
msmau.spb.mm	101000	0	1	ms2	0	ms1	100	1	md	OP-M32
msmau.spa.h.mm	101000	0	1	ms2	0	ms1	001	0	md	OP-M32
msmau.spb.h.mm	101000	0	1	ms2	0	ms1	001	1	md	OP-M32
msmau.spa.w.mm	101000	0	1	ms2	0	ms1	010	0	md	OP-M32
msmau.spb.w.mm	101000	0	1	ms2	0	ms1	010	1	md	OP-M32
msmau.spa.dw.mm	101000	0	1	ms2	0	ms1	011	0	md	OP-M32
msmau.spb.dw.mm	101000	0	1	ms2	0	ms1	011	1	md	OP-M32
mma.spa.mm	101000	0	0	ms2	1	ms1	100	0	md	OP-M32
mma.spb.mm	101000	0	0	ms2	1	ms1	100	1	md	OP-M32
mma.spa.h.mm	101000	0	0	ms2	1	ms1	001	0	md	OP-M32
mma.spb.h.mm	101000	0	0	ms2	1	ms1	001	1	md	OP-M32
mma.spa.w.mm	101000	0	0	ms2	1	ms1	010	0	md	OP-M32
mma.spb.w.mm	101000	0	0	ms2	1	ms1	010	1	md	OP-M32
mma.spa.dw.mm	101000	0	0	ms2	1	ms1	011	0	md	OP-M32

mma.spb.dw.mm	101000	0	0	ms2	1	ms1	011	1	md	OP-M32
msma.spa.mm	101000	0	1	ms2	1	ms1	100	0	md	OP-M32
msma.spb.mm	101000	0	1	ms2	1	ms1	100	1	md	OP-M32
msma.spa.h.mm	101000	0	1	ms2	1	ms1	001	0	md	OP-M32
msma.spb.h.mm	101000	0	1	ms2	1	ms1	001	1	md	OP-M32
msma.spa.w.mm	101000	0	1	ms2	1	ms1	010	0	md	OP-M32
msma.spb.w.mm	101000	0	1	ms2	1	ms1	010	1	md	OP-M32
msma.spa.dw.mm	101000	0	1	ms2	1	ms1	011	0	md	OP-M32
msma.spb.dw.mm	101000	0	1	ms2	1	ms1	011	1	md	OP-M32
mfma.spa.mm	101000	1	0	ms2	1	ms1	100	0	md	OP-M32
mfma.spb.mm	101000	1	0	ms2	1	ms1	100	1	md	OP-M32
mfma.spa.hf.mm	101000	1	0	ms2	1	ms1	001	0	md	OP-M32
mfma.spb.hf.mm	101000	1	0	ms2	1	ms1	001	1	md	OP-M32
mfma.spa.f.mm	101000	1	0	ms2	1	ms1	010	0	md	OP-M32
mfma.spb.f.mm	101000	1	0	ms2	1	ms1	010	1	md	OP-M32
mfma.spa.d.mm	101000	1	0	ms2	1	ms1	011	0	md	OP-M32
mfma.spb.d.mm	101000	1	0	ms2	1	ms1	011	1	md	OP-M32
mwmau.spa.mm	101001	0	0	ms2	0	ms1	100	0	md	OP-M32
mwmau.spb.mm	101001	0	0	ms2	0	ms1	100	1	md	OP-M32
mwmau.spa.h.mm	101001	0	0	ms2	0	ms1	001	0	md	OP-M32
mwmau.spb.h.mm	101001	0	0	ms2	0	ms1	001	1	md	OP-M32
mwmau.spa.w.mm	101001	0	0	ms2	0	ms1	010	0	md	OP-M32
mwmau.spb.w.mm	101001	0	0	ms2	0	ms1	010	1	md	OP-M32
mswmau.spa.mm	101001	0	1	ms2	0	ms1	100	0	md	OP-M32
mswmau.spb.mm	101001	0	1	ms2	0	ms1	100	1	md	OP-M32
mswmau.spa.h.mm	101001	0	1	ms2	0	ms1	001	0	md	OP-M32
mswmau.spb.h.mm	101001	0	1	ms2	0	ms1	001	1	md	OP-M32
mswmau.spa.w.mm	101001	0	1	ms2	0	ms1	010	0	md	OP-M32
mswmau.spb.w.mm	101001	0	1	ms2	0	ms1	010	1	md	OP-M32

mwma.spa.mm	101001	0	0	ms2	1	ms1	100	0	md	OP-M32
mwma.spb.mm	101001	0	0	ms2	1	ms1	100	1	md	OP-M32
mwma.spa.h.mm	101001	0	0	ms2	1	ms1	001	0	md	OP-M32
mwma.spb.h.mm	101001	0	0	ms2	1	ms1	001	1	md	OP-M32
mwma.spa.w.mm	101001	0	0	ms2	1	ms1	010	0	md	OP-M32
mwma.spb.w.mm	101001	0	0	ms2	1	ms1	010	1	md	OP-M32
mswma.spa.mm	101001	0	1	ms2	1	ms1	100	0	md	OP-M32
mswma.spb.mm	101001	0	1	ms2	1	ms1	100	1	md	OP-M32
mswma.spa.h.mm	101001	0	1	ms2	1	ms1	001	0	md	OP-M32
mswma.spb.h.mm	101001	0	1	ms2	1	ms1	001	1	md	OP-M32
mswma.spa.w.mm	101001	0	1	ms2	1	ms1	010	0	md	OP-M32
mswma.spb.w.mm	101001	0	1	ms2	1	ms1	010	1	md	OP-M32
mfwma.spa.mm	101001	1	0	ms2	1	ms1	100	0	md	OP-M32
mfwma.spb.mm	101001	1	0	ms2	1	ms1	100	1	md	OP-M32
mfwma.spa.hf.mm	101001	1	0	ms2	1	ms1	001	0	md	OP-M32
mfwma.spb.hf.mm	101001	1	0	ms2	1	ms1	001	1	md	OP-M32
mfwma.spa.f.mm	101001	1	0	ms2	1	ms1	010	0	md	OP-M32
mfwma.spb.f.mm	101001	1	0	ms2	1	ms1	010	1	md	OP-M32
mqmau.spa.mm	101010	0	0	ms2	0	ms1	100	0	md	OP-M32
mqmau.spb.mm	101010	0	0	ms2	0	ms1	100	1	md	OP-M32
mqmau.spa.b.mm	101010	0	0	ms2	0	ms1	000	0	md	OP-M32
mqmau.spb.b.mm	101010	0	0	ms2	0	ms1	000	1	md	OP-M32
msqmau.spa.mm	101010	0	1	ms2	0	ms1	100	0	md	OP-M32
msqmau.spb.mm	101010	0	1	ms2	0	ms1	100	1	md	OP-M32
msqmau.spa.b.mm	101010	0	1	ms2	0	ms1	000	0	md	OP-M32
msqmau.spb.b.mm	101010	0	1	ms2	0	ms1	000	1	md	OP-M32
mqma.spa.mm	101010	0	0	ms2	1	ms1	100	0	md	OP-M32
mqma.spb.mm	101010	0	0	ms2	1	ms1	100	1	md	OP-M32
mqma.spa.b.mm	101010	0	0	ms2	1	ms1	000	0	md	OP-M32

mqma.spb.b.mm	101010	0	0	ms2	1	ms1	000	1	md	OP-M32
msqma.spa.mm	101010	0	1	ms2	1	ms1	100	0	md	OP-M32
msqma.spb.mm	101010	0	1	ms2	1	ms1	100	1	md	OP-M32
msqma.spa.b.mm	101010	0	1	ms2	1	ms1	000	0	md	OP-M32
msqma.spb.b.mm	101010	0	1	ms2	1	ms1	000	1	md	OP-M32
mfqma.spa.mm	101010	1	0	ms2	1	ms1	100	0	md	OP-M32
mfqma.spb.mm	101010	1	0	ms2	1	ms1	100	1	md	OP-M32
mfqma.spa.cf.mm	101010	1	0	ms2	1	ms1	000	0	md	OP-M32
mfqma.spb.cf.mm	101010	1	0	ms2	1	ms1	000	1	md	OP-M32
momau.spa.mm	101011	0	0	ms2	0	ms1	100	0	md	OP-M32
momau.spb.mm	101011	0	0	ms2	0	ms1	100	1	md	OP-M32
momau.spa.hb.mm	101011	0	0	ms2	0	ms1	111	0	md	OP-M32
momau.spb.hb.mm	101011	0	0	ms2	0	ms1	111	1	md	OP-M32
msomau.spa.mm	101011	0	1	ms2	0	ms1	100	0	md	OP-M32
msomau.spb.mm	101011	0	1	ms2	0	ms1	100	1	md	OP-M32
msomau.spa.hb.mm	101011	0	1	ms2	0	ms1	111	0	md	OP-M32
msomau.spb.hb.mm	101011	0	1	ms2	0	ms1	111	1	md	OP-M32
moma.spa.mm	101011	0	0	ms2	1	ms1	100	0	md	OP-M32
moma.spb.mm	101011	0	0	ms2	1	ms1	100	1	md	OP-M32
moma.spa.hb.mm	101011	0	0	ms2	1	ms1	111	0	md	OP-M32
moma.spb.hb.mm	101011	0	0	ms2	1	ms1	111	1	md	OP-M32
msoma.spa.mm	101011	0	1	ms2	1	ms1	100	0	md	OP-M32
msoma.spb.mm	101011	0	1	ms2	1	ms1	100	1	md	OP-M32
msoma.spa.hb.mm	101011	0	1	ms2	1	ms1	111	0	md	OP-M32
msoma.spb.hb.mm	101011	0	1	ms2	1	ms1	111	1	md	OP-M32

Table 31. Element-wise Arithmetic & Logic Instructions

Format	31 26	25	24	23 20	19	18 15	14 12	11	10 7	6 0
	funct6	**fp**	**sa**	**ms2**	**sn**	**ms1**	**eew**	**ma**	**md**	**opcode**
maddu.mm	001000	0	0	ms2	0	ms1	100	0	md	OP-M32

maddu.b.mm	001000	0	0	ms2	0	ms1	000	0	md	OP-M32
maddu.h.mm	001000	0	0	ms2	0	ms1	001	0	md	OP-M32
maddu.w.mm	001000	0	0	ms2	0	ms1	010	0	md	OP-M32
maddu.dw.mm	001000	0	0	ms2	0	ms1	011	0	md	OP-M32
msaddu.mm	001000	0	1	ms2	0	ms1	100	0	md	OP-M32
msaddu.b.mm	001000	0	1	ms2	0	ms1	000	0	md	OP-M32
msaddu.h.mm	001000	0	1	ms2	0	ms1	001	0	md	OP-M32
msaddu.w.mm	001000	0	1	ms2	0	ms1	010	0	md	OP-M32
msaddu.dw.mm	001000	0	1	ms2	0	ms1	011	0	md	OP-M32
madd.mm	001000	0	0	ms2	1	ms1	100	0	md	OP-M32
madd.b.mm	001000	0	0	ms2	1	ms1	000	0	md	OP-M32
madd.h.mm	001000	0	0	ms2	1	ms1	001	0	md	OP-M32
madd.w.mm	001000	0	0	ms2	1	ms1	010	0	md	OP-M32
madd.dw.mm	001000	0	0	ms2	1	ms1	011	0	md	OP-M32
msadd.mm	001000	0	1	ms2	1	ms1	100	0	md	OP-M32
msadd.b.mm	001000	0	1	ms2	1	ms1	000	0	md	OP-M32
msadd.h.mm	001000	0	1	ms2	1	ms1	001	0	md	OP-M32
msadd.w.mm	001000	0	1	ms2	1	ms1	010	0	md	OP-M32
msadd.dw.mm	001000	0	1	ms2	1	ms1	011	0	md	OP-M32
mfadd.mm	001000	1	0	ms2	1	ms1	100	0	md	OP-M32
mfadd.hf.mm	001000	1	0	ms2	1	ms1	001	0	md	OP-M32
mfadd.f.mm	001000	1	0	ms2	1	ms1	010	0	md	OP-M32
mfadd.d.mm	001000	1	0	ms2	1	ms1	011	0	md	OP-M32
mwaddu.mm	001001	0	0	ms2	0	ms1	100	0	md	OP-M32
mwaddu.b.mm	001001	0	0	ms2	0	ms1	000	0	md	OP-M32
mwaddu.h.mm	001001	0	0	ms2	0	ms1	001	0	md	OP-M32
mwaddu.w.mm	001001	0	0	ms2	0	ms1	010	0	md	OP-M32
mwadd.mm	001001	0	0	ms2	1	ms1	100	0	md	OP-M32
mwadd.b.mm	001001	0	0	ms2	1	ms1	000	0	md	OP-M32

mwadd.h.mm	001001	0	0	ms2	1	ms1	001	0	md	OP-M32
mwadd.w.mm	001001	0	0	ms2	1	ms1	010	0	md	OP-M32
mfwadd.mm	001001	1	0	ms2	1	ms1	100	0	md	OP-M32
mfwadd.hf.mm	001001	1	0	ms2	1	ms1	001	0	md	OP-M32
mfwadd.f.mm	001001	1	0	ms2	1	ms1	010	0	md	OP-M32
msubu.mm	001010	0	0	ms2	0	ms1	100	0	md	OP-M32
msubu.b.mm	001010	0	0	ms2	0	ms1	000	0	md	OP-M32
msubu.h.mm	001010	0	0	ms2	0	ms1	001	0	md	OP-M32
msubu.w.mm	001010	0	0	ms2	0	ms1	010	0	md	OP-M32
msubu.dw.mm	001010	0	0	ms2	0	ms1	011	0	md	OP-M32
mssubu.mm	001010	0	1	ms2	0	ms1	100	0	md	OP-M32
mssubu.b.mm	001010	0	1	ms2	0	ms1	000	0	md	OP-M32
mssubu.h.mm	001010	0	1	ms2	0	ms1	001	0	md	OP-M32
mssubu.w.mm	001010	0	1	ms2	0	ms1	010	0	md	OP-M32
mssubu.dw.mm	001010	0	1	ms2	0	ms1	011	0	md	OP-M32
msub.mm	001010	0	0	ms2	1	ms1	100	0	md	OP-M32
msub.b.mm	001010	0	0	ms2	1	ms1	000	0	md	OP-M32
msub.h.mm	001010	0	0	ms2	1	ms1	001	0	md	OP-M32
msub.w.mm	001010	0	0	ms2	1	ms1	010	0	md	OP-M32
msub.dw.mm	001010	0	0	ms2	1	ms1	011	0	md	OP-M32
mssub.mm	001010	0	1	ms2	1	ms1	100	0	md	OP-M32
mssub.b.mm	001010	0	1	ms2	1	ms1	000	0	md	OP-M32
mssub.h.mm	001010	0	1	ms2	1	ms1	001	0	md	OP-M32
mssub.w.mm	001010	0	1	ms2	1	ms1	010	0	md	OP-M32
mssub.dw.mm	001010	0	1	ms2	1	ms1	011	0	md	OP-M32
mfsub.mm	001010	1	0	ms2	1	ms1	100	0	md	OP-M32
mfsub.hf.mm	001010	1	0	ms2	1	ms1	001	0	md	OP-M32
mfsub.f.mm	001010	1	0	ms2	1	ms1	010	0	md	OP-M32
mfsub.d.mm	001010	1	0	ms2	1	ms1	011	0	md	OP-M32

mwsubu.mm	001011	0	0	ms2	0	ms1	100	0	md	OP-M32
mwsubu.b.mm	001011	0	0	ms2	0	ms1	000	0	md	OP-M32
mwsubu.h.mm	001011	0	0	ms2	0	ms1	001	0	md	OP-M32
mwsubu.w.mm	001011	0	0	ms2	0	ms1	010	0	md	OP-M32
mwsub.mm	001011	0	0	ms2	1	ms1	100	0	md	OP-M32
mwsub.b.mm	001011	0	0	ms2	1	ms1	000	0	md	OP-M32
mwsub.h.mm	001011	0	0	ms2	1	ms1	001	0	md	OP-M32
mwsub.w.mm	001011	0	0	ms2	1	ms1	010	0	md	OP-M32
mfwsub.mm	001011	1	0	ms2	1	ms1	100	0	md	OP-M32
mfwsub.hf.mm	001011	1	0	ms2	1	ms1	001	0	md	OP-M32
mfwsub.f.mm	001011	1	0	ms2	1	ms1	010	0	md	OP-M32
mminu.mm	001100	0	0	ms2	0	ms1	100	0	md	OP-M32
mminu.b.mm	001100	0	0	ms2	0	ms1	000	0	md	OP-M32
mminu.h.mm	001100	0	0	ms2	0	ms1	001	0	md	OP-M32
mminu.w.mm	001100	0	0	ms2	0	ms1	010	0	md	OP-M32
mminu.dw.mm	001100	0	0	ms2	0	ms1	011	0	md	OP-M32
mmaxu.mm	001100	0	1	ms2	0	ms1	100	0	md	OP-M32
mmaxu.b.mm	001100	0	1	ms2	0	ms1	000	0	md	OP-M32
mmaxu.h.mm	001100	0	1	ms2	0	ms1	001	0	md	OP-M32
mmaxu.w.mm	001100	0	1	ms2	0	ms1	010	0	md	OP-M32
mmaxu.dw.mm	001100	0	1	ms2	0	ms1	011	0	md	OP-M32
mmin.mm	001100	0	0	ms2	1	ms1	100	0	md	OP-M32
mmin.b.mm	001100	0	0	ms2	1	ms1	000	0	md	OP-M32
mmin.h.mm	001100	0	0	ms2	1	ms1	001	0	md	OP-M32
mmin.w.mm	001100	0	0	ms2	1	ms1	010	0	md	OP-M32
mmin.dw.mm	001100	0	0	ms2	1	ms1	011	0	md	OP-M32
mmax.mm	001100	0	1	ms2	1	ms1	100	0	md	OP-M32
mmax.b.mm	001100	0	1	ms2	1	ms1	000	0	md	OP-M32
mmax.h.mm	001100	0	1	ms2	1	ms1	001	0	md	OP-M32

mmax.w.mm	001100	0	1	ms2	1	ms1	010	0	md	OP-M32
mmax.dw.mm	001100	0	1	ms2	1	ms1	011	0	md	OP-M32
mfmin.mm	001100	1	0	ms2	1	ms1	100	0	md	OP-M32
mfmin.hf.mm	001100	1	0	ms2	1	ms1	001	0	md	OP-M32
mfmin.f.mm	001100	1	0	ms2	1	ms1	010	0	md	OP-M32
mfmin.d.mm	001100	1	0	ms2	1	ms1	011	0	md	OP-M32
mfmax.mm	001100	1	1	ms2	1	ms1	100	0	md	OP-M32
mfmax.hf.mm	001100	1	1	ms2	1	ms1	001	0	md	OP-M32
mfmax.f.mm	001100	1	1	ms2	1	ms1	010	0	md	OP-M32
mfmax.d.mm	001100	1	1	ms2	1	ms1	011	0	md	OP-M32
msmulu.mm	001101	0	1	ms2	0	ms1	100	0	md	OP-M32
msmulu.b.mm	001101	0	1	ms2	0	ms1	000	0	md	OP-M32
msmulu.h.mm	001101	0	1	ms2	0	ms1	001	0	md	OP-M32
msmulu.w.mm	001101	0	1	ms2	0	ms1	010	0	md	OP-M32
msmulu.dw.mm	001101	0	1	ms2	0	ms1	011	0	md	OP-M32
mmul.mm	001101	0	0	ms2	1	ms1	100	0	md	OP-M32
mmul.b.mm	001101	0	0	ms2	1	ms1	000	0	md	OP-M32
mmul.h.mm	001101	0	0	ms2	1	ms1	001	0	md	OP-M32
mmul.w.mm	001101	0	0	ms2	1	ms1	010	0	md	OP-M32
mmul.dw.mm	001101	0	0	ms2	1	ms1	011	0	md	OP-M32
msmul.mm	001101	0	1	ms2	1	ms1	100	0	md	OP-M32
msmul.b.mm	001101	0	1	ms2	1	ms1	000	0	md	OP-M32
msmul.h.mm	001101	0	1	ms2	1	ms1	001	0	md	OP-M32
msmul.w.mm	001101	0	1	ms2	1	ms1	010	0	md	OP-M32
msmul.dw.mm	001101	0	1	ms2	1	ms1	011	0	md	OP-M32
mfmul.mm	001101	1	0	ms2	1	ms1	100	0	md	OP-M32
mfmul.hf.mm	001101	1	0	ms2	1	ms1	001	0	md	OP-M32
mfmul.f.mm	001101	1	0	ms2	1	ms1	010	0	md	OP-M32
mfmul.d.mm	001101	1	0	ms2	1	ms1	011	0	md	OP-M32

mmulhu.mm	001110	0	0	ms2	0	ms1	100	0	md	OP-M32
mmulhu.b.mm	001110	0	0	ms2	0	ms1	000	0	md	OP-M32
mmulhu.h.mm	001110	0	0	ms2	0	ms1	001	0	md	OP-M32
mmulhu.w.mm	001110	0	0	ms2	0	ms1	010	0	md	OP-M32
mmulhu.dw.mm	001110	0	0	ms2	0	ms1	011	0	md	OP-M32
mmulh.mm	001110	0	0	ms2	1	ms1	100	0	md	OP-M32
mmulh.b.mm	001110	0	0	ms2	1	ms1	000	0	md	OP-M32
mmulh.h.mm	001110	0	0	ms2	1	ms1	001	0	md	OP-M32
mmulh.w.mm	001110	0	0	ms2	1	ms1	010	0	md	OP-M32
mmulh.dw.mm	001110	0	0	ms2	1	ms1	011	0	md	OP-M32
mmulhsu.mm	001110	0	1	ms2	0	ms1	100	0	md	OP-M32
mmulhsu.b.mm	001110	0	1	ms2	0	ms1	000	0	md	OP-M32
mmulhsu.h.mm	001110	0	1	ms2	0	ms1	001	0	md	OP-M32
mmulhsu.w.mm	001110	0	1	ms2	0	ms1	010	0	md	OP-M32
mmulhsu.dw.mm	001110	0	1	ms2	0	ms1	011	0	md	OP-M32
msmulsu.mm	001110	0	1	ms2	1	ms1	100	0	md	OP-M32
msmulsu.b.mm	001110	0	1	ms2	1	ms1	000	0	md	OP-M32
msmulsu.h.mm	001110	0	1	ms2	1	ms1	001	0	md	OP-M32
msmulsu.w.mm	001110	0	1	ms2	1	ms1	010	0	md	OP-M32
msmulsu.dw.mm	001110	0	1	ms2	1	ms1	011	0	md	OP-M32
mfdiv.mm	001110	1	0	ms2	1	ms1	100	0	md	OP-M32
mfdiv.hf.mm	001110	1	0	ms2	1	ms1	001	0	md	OP-M32
mfdiv.f.mm	001110	1	0	ms2	1	ms1	010	0	md	OP-M32
mfdiv.d.mm	001110	1	0	ms2	1	ms1	011	0	md	OP-M32
mwmulu.mm	001111	0	0	ms2	0	ms1	100	0	md	OP-M32
mwmulu.b.mm	001111	0	0	ms2	0	ms1	000	0	md	OP-M32
mwmulu.h.mm	001111	0	0	ms2	0	ms1	001	0	md	OP-M32
mwmulu.w.mm	001111	0	0	ms2	0	ms1	010	0	md	OP-M32
mwmul.mm	001111	0	0	ms2	1	ms1	100	0	md	OP-M32

mwmul.b.mm	001111	0	0	ms2	1	ms1	000	0	md	OP-M32
mwmul.h.mm	001111	0	0	ms2	1	ms1	001	0	md	OP-M32
mwmul.w.mm	001111	0	0	ms2	1	ms1	010	0	md	OP-M32
mfwmul.mm	001111	1	0	ms2	1	ms1	100	0	md	OP-M32
mfwmul.hf.mm	001111	1	0	ms2	1	ms1	001	0	md	OP-M32
mfwmul.f.mm	001111	1	0	ms2	1	ms1	010	0	md	OP-M32
mand.mm	010000	0	0	ms2	0	ms1	100	0	md	OP-M32
mor.mm	010000	0	1	ms2	0	ms1	100	0	md	OP-M32
mxor.mm	010000	0	1	ms2	1	ms1	100	0	md	OP-M32
mfsqrt.mm	010000	1	0	0	1	ms1	100	0	md	OP-M32
mfsqrt.hf.mm	010000	1	0	0	1	ms1	001	0	md	OP-M32
mfsqrt.f.mm	010000	1	0	0	1	ms1	010	0	md	OP-M32
mfsqrt.d.mm	010000	1	0	0	1	ms1	011	0	md	OP-M32
msll.mm	010001	0	0	ms2	0	ms1	100	0	md	OP-M32
msll.b.mm	010001	0	0	ms2	0	ms1	000	0	md	OP-M32
msll.h.mm	010001	0	0	ms2	0	ms1	001	0	md	OP-M32
msll.w.mm	010001	0	0	ms2	0	ms1	010	0	md	OP-M32
msll.dw.mm	010001	0	0	ms2	0	ms1	011	0	md	OP-M32
msrl.mm	010001	0	1	ms2	0	ms1	100	0	md	OP-M32
msrl.b.mm	010001	0	1	ms2	0	ms1	000	0	md	OP-M32
msrl.h.mm	010001	0	1	ms2	0	ms1	001	0	md	OP-M32
msrl.w.mm	010001	0	1	ms2	0	ms1	010	0	md	OP-M32
msrl.dw.mm	010001	0	1	ms2	0	ms1	011	0	md	OP-M32
msra.mm	010001	0	1	ms2	1	ms1	100	0	md	OP-M32
msra.b.mm	010001	0	1	ms2	1	ms1	000	0	md	OP-M32
msra.h.mm	010001	0	1	ms2	1	ms1	001	0	md	OP-M32
msra.w.mm	010001	0	1	ms2	1	ms1	010	0	md	OP-M32
msra.dw.mm	010001	0	1	ms2	1	ms1	011	0	md	OP-M32

Table 32. Type Convert Instructions

Format	31 26	25	24 22	21 20	19	18 15	14 12	11	10 7	6 0
	funct6	**fd**	**f3**	**nw**	**sn**	**ms1**	**eew**	**ma**	**md**	**opcode**
mcvt.x.xu.m	011000	0	000	00	1	ms1	100	0	md	OP-M32
mcvt.hb.uhb.m	011000	0	000	00	1	ms1	111	0	md	OP-M32
mcvt.b.ub.m	011000	0	000	00	1	ms1	000	0	md	OP-M32
mcvt.h.uh.m	011000	0	000	00	1	ms1	001	0	md	OP-M32
mcvt.w.uw.m	011000	0	000	00	1	ms1	010	0	md	OP-M32
mcvt.dw.udw.m	011000	0	000	00	1	ms1	011	0	md	OP-M32
mcvt.xu.x.m	011000	0	001	00	1	ms1	100	0	md	OP-M32
mcvt.uhb.hb.m	011000	0	001	00	1	ms1	111	0	md	OP-M32
mcvt.ub.b.m	011000	0	001	00	1	ms1	000	0	md	OP-M32
mcvt.uh.h.m	011000	0	001	00	1	ms1	001	0	md	OP-M32
mcvt.uw.w.m	011000	0	001	00	1	ms1	010	0	md	OP-M32
mcvt.udw.dw.m	011000	0	001	00	1	ms1	011	0	md	OP-M32
mwcvtu.xw.x.m	011000	0	000	01	0	ms1	100	0	md	OP-M32
mwcvtu.xq.x.m	011000	0	001	01	0	ms1	100	0	md	OP-M32
mwcvtu.xo.x.m	011000	0	010	01	0	ms1	100	0	md	OP-M32
mwcvtu.b.hb.m	011000	0	000	01	0	ms1	111	0	md	OP-M32
mwcvtu.h.hb.m	011000	0	001	01	0	ms1	111	0	md	OP-M32
mwcvtu.w.hb.m	011000	0	010	01	0	ms1	111	0	md	OP-M32
mwcvtu.h.b.m	011000	0	000	01	0	ms1	000	0	md	OP-M32
mwcvtu.w.b.m	011000	0	001	01	0	ms1	000	0	md	OP-M32
mwcvtu.w.h.m	011000	0	000	01	0	ms1	001	0	md	OP-M32
mwcvtu.dw.w.m	011000	0	000	01	0	ms1	010	0	md	OP-M32
mwcvt.xw.x.m	011000	0	000	01	1	ms1	100	0	md	OP-M32
mwcvt.xq.x.m	011000	0	001	01	1	ms1	100	0	md	OP-M32
mwcvt.xo.x.m	011000	0	010	01	1	ms1	100	0	md	OP-M32
mwcvt.b.hb.m	011000	0	000	01	1	ms1	111	0	md	OP-M32
mwcvt.h.hb.m	011000	0	001	01	1	ms1	111	0	md	OP-M32

mwcvt.w.hb.m	011000	0	010	01	1	ms1	111	0	md	OP-M32
mwcvt.h.b.m	011000	0	000	01	1	ms1	000	0	md	OP-M32
mwcvt.w.b.m	011000	0	001	01	1	ms1	000	0	md	OP-M32
mwcvt.w.h.m	011000	0	000	01	1	ms1	001	0	md	OP-M32
mwcvt.dw.w.m	011000	0	000	01	1	ms1	010	0	md	OP-M32
mncvtu.x.xw.m	011000	0	000	10	0	ms1	100	0	md	OP-M32
mncvtu.x.xq.m	011000	0	001	10	0	ms1	100	0	md	OP-M32
mncvtu.x.xo.m	011000	0	010	10	0	ms1	100	0	md	OP-M32
mncvtu.hb.b.m	011000	0	000	10	0	ms1	000	0	md	OP-M32
mncvtu.hb.h.m	011000	0	001	10	0	ms1	001	0	md	OP-M32
mncvtu.hb.w.m	011000	0	010	10	0	ms1	010	0	md	OP-M32
mncvtu.b.h.m	011000	0	000	10	0	ms1	001	0	md	OP-M32
mncvtu.b.w.m	011000	0	001	10	0	ms1	010	0	md	OP-M32
mncvtu.h.w.m	011000	0	000	10	0	ms1	010	0	md	OP-M32
mncvtu.w.dw.m	011000	0	000	10	0	ms1	011	0	md	OP-M32
mncvt.x.xw.m	011000	0	000	10	1	ms1	100	0	md	OP-M32
mncvt.x.xq.m	011000	0	001	10	1	ms1	100	0	md	OP-M32
mncvt.x.xo.m	011000	0	010	10	1	ms1	100	0	md	OP-M32
mncvt.hb.b.m	011000	0	000	10	1	ms1	000	0	md	OP-M32
mncvt.hb.h.m	011000	0	001	10	1	ms1	001	0	md	OP-M32
mncvt.hb.w.m	011000	0	010	10	1	ms1	010	0	md	OP-M32
mncvt.b.h.m	011000	0	000	10	1	ms1	001	0	md	OP-M32
mncvt.b.w.m	011000	0	001	10	1	ms1	010	0	md	OP-M32
mncvt.h.w.m	011000	0	000	10	1	ms1	010	0	md	OP-M32
mncvt.w.dw.m	011000	0	000	10	1	ms1	011	0	md	OP-M32
mfcvt.bf.hf.m	011001	1	000	00	0	ms1	001	0	md	OP-M32
mfcvt.hf.bf.m	011001	1	000	00	1	ms1	001	0	md	OP-M32
mfwcvt.fw.f.m	011001	1	001	01	0	ms1	100	0	md	OP-M32
mfwcvt.hf.cf.m	011001	1	001	01	0	ms1	000	0	md	OP-M32

mfwcvt.f.hf.m	011001	1	001	01	0	ms1	001	0	md	OP-M32
mfwcvt.d.f.m	011001	1	001	01	0	ms1	010	0	md	OP-M32
mfncvt.f.fw.m	011001	1	001	10	0	ms1	100	0	md	OP-M32
mfncvt.cf.hf.m	011001	1	001	10	0	ms1	001	0	md	OP-M32
mfncvt.hf.f.m	011001	1	001	10	0	ms1	010	0	md	OP-M32
mfncvt.f.d.m	011001	1	001	10	0	ms1	011	0	md	OP-M32
mfcvtu.f.x.m	011001	1	010	00	0	ms1	100	0	md	OP-M32
mfcvtu.hf.h.m	011001	1	010	00	0	ms1	001	0	md	OP-M32
mfcvtu.f.w.m	011001	1	010	00	0	ms1	010	0	md	OP-M32
mfcvtu.d.dw.m	011001	1	010	00	0	ms1	011	0	md	OP-M32
mfcvt.f.x.m	011001	1	010	00	1	ms1	100	0	md	OP-M32
mfcvt.hf.h.m	011001	1	010	00	1	ms1	001	0	md	OP-M32
mfcvt.f.w.m	011001	1	010	00	1	ms1	010	0	md	OP-M32
mfcvt.d.dw.m	011001	1	010	00	1	ms1	011	0	md	OP-M32
mfwcvtu.fw.x.m	011001	1	010	01	0	ms1	100	0	md	OP-M32
mfwcvtu.hf.b.m	011001	1	010	01	0	ms1	000	0	md	OP-M32
mfwcvtu.f.h.m	011001	1	010	01	0	ms1	001	0	md	OP-M32
mfwcvtu.d.w.m	011001	1	010	01	0	ms1	010	0	md	OP-M32
mfwcvt.fw.x.m	011001	1	010	01	1	ms1	100	0	md	OP-M32
mfwcvt.hf.b.m	011001	1	010	01	1	ms1	000	0	md	OP-M32
mfwcvt.f.h.m	011001	1	010	01	1	ms1	001	0	md	OP-M32
mfwcvt.d.w.m	011001	1	010	01	1	ms1	010	0	md	OP-M32
mfncvtu.f.xw.m	011001	1	010	10	0	ms1	100	0	md	OP-M32
mfncvtu.hf.w.m	011001	1	010	10	0	ms1	010	0	md	OP-M32
mfncvtu.f.dw.m	011001	1	010	10	0	ms1	011	0	md	OP-M32
mfncvt.f.xw.m	011001	1	010	10	1	ms1	100	0	md	OP-M32
mfncvt.hf.w.m	011001	1	010	10	1	ms1	010	0	md	OP-M32
mfncvt.f.dw.m	011001	1	010	10	1	ms1	011	0	md	OP-M32
mfcvtu.x.f.m	011001	0	010	00	0	ms1	100	0	md	OP-M32

mfcvtu.h.hf.m	011001	0	010	00	0	ms1	001	0	md	OP-M32
mfcvtu.w.f.m	011001	0	010	00	0	ms1	010	0	md	OP-M32
mfcvtu.dw.d.m	011001	0	010	00	0	ms1	011	0	md	OP-M32
mfcvt.x.f.m	011001	0	010	00	1	ms1	100	0	md	OP-M32
mfcvt.h.hf.m	011001	0	010	00	1	ms1	001	0	md	OP-M32
mfcvt.w.f.m	011001	0	010	00	1	ms1	010	0	md	OP-M32
mfcvt.dw.d.m	011001	0	010	00	1	ms1	011	0	md	OP-M32
mfwcvtu.xw.f.m	011001	0	010	01	0	ms1	100	0	md	OP-M32
mfwcvtu.w.hf.m	011001	0	010	01	0	ms1	001	0	md	OP-M32
mfwcvtu.dw.f.m	011001	0	010	01	0	ms1	010	0	md	OP-M32
mfwcvt.xw.f.m	011001	0	010	01	1	ms1	100	0	md	OP-M32
mfwcvt.w.hf.m	011001	0	010	01	1	ms1	001	0	md	OP-M32
mfwcvt.dw.f.m	011001	0	010	01	1	ms1	010	0	md	OP-M32
mfncvtu.x.fw.m	011001	0	010	10	0	ms1	100	0	md	OP-M32
mfncvtu.b.hf.m	011001	0	010	10	0	ms1	001	0	md	OP-M32
mfncvtu.h.f.m	011001	0	010	10	0	ms1	010	0	md	OP-M32
mfncvtu.w.d.m	011001	0	010	10	0	ms1	011	0	md	OP-M32
mfncvt.x.fw.m	011001	0	010	10	1	ms1	100	0	md	OP-M32
mfncvt.b.hf.m	011001	0	010	10	1	ms1	001	0	md	OP-M32
mfncvt.h.f.m	011001	0	010	10	1	ms1	010	0	md	OP-M32
mfncvt.w.d.m	011001	0	010	10	1	ms1	011	0	md	OP-M32
mfwcvtu.fq.x.m	011001	1	110	01	0	ms1	100	0	md	OP-M32
mfwcvtu.hf.hb.m	011001	1	110	01	0	ms1	111	0	md	OP-M32
mfwcvtu.f.b.m	011001	1	110	01	0	ms1	000	0	md	OP-M32
mfwcvt.fq.x.m	011001	1	110	01	1	ms1	100	0	md	OP-M32
mfwcvt.hf.hb.m	011001	1	110	01	1	ms1	111	0	md	OP-M32
mfwcvt.f.b.m	011001	1	110	01	1	ms1	000	0	md	OP-M32
mfwcvtu.fo.x.m	011001	1	111	01	0	ms1	100	0	md	OP-M32
mfwcvtu.f.hb.m	011001	1	111	01	0	ms1	111	0	md	OP-M32

mfwcvt.fo.x.m	011001	1	111	01	1	ms1	100	0	md	OP-M32
mfwcvt.f.hb.m	011001	1	111	01	1	ms1	111	0	md	OP-M32
mfncvtu.x.fq.m	011001	0	110	10	0	ms1	100	0	md	OP-M32
mfncvtu.hb.hf.m	011001	0	110	10	0	ms1	001	0	md	OP-M32
mfncvtu.b.f.m	011001	0	110	10	0	ms1	010	0	md	OP-M32
mfncvt.x.fq.m	011001	0	110	10	1	ms1	100	0	md	OP-M32
mfncvt.hb.hf.m	011001	0	110	10	1	ms1	001	0	md	OP-M32
mfncvt.b.f.m	011001	0	110	10	1	ms1	010	0	md	OP-M32
mfncvtu.x.fo.m	011001	0	111	10	0	ms1	100	0	md	OP-M32
mfncvtu.hb.f.m	011001	0	111	10	0	ms1	010	0	md	OP-M32
mfncvt.x.fo.m	011001	0	111	10	1	ms1	100	0	md	OP-M32
mfncvt.hb.f.m	011001	0	111	10	1	ms1	010	0	md	OP-M32