



RISC-V Matrix Specification

Stream Computing

Version 0.4b, 06/2024: 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	9
4. Instructions	11
4.1. Instruction Formats	11
4.2. Configuration-Setting Instructions	13
4.2.1. mtype Encoding	15
4.2.2. ATM/ATK/ATN Encoding	15
4.2.3. Constraints on Setting mtilem/mtilek/mtilen	16
4.3. Load and Store Instructions	18
4.3.1. Load Instructions	18
4.3.2. Store Instructions	19
4.3.3. Whole Matrix Load & Store Instructions	20
4.4. Data Move Instructions	21
4.4.1. Data Move Instructions between Matrix and Integer	21
4.4.2. Data Move Instructions between Matrix and Float-point	21
4.4.3. Data Broadcast Instructions	22
4.4.4. Matrix Transpose Instructions	23
4.5. Arithmetic and Logic Instructions	23
4.5.1. Matrix Multiplication Instructions	23
4.5.2. Element-Wise Instructions	25
4.6. Type-Convert Instructions	28
5. Intrinsic Examples	30
5.1. Matrix multiplication	30
5.2. Matrix multiplication with left matrix transposed	30
5.3. Matrix transpose without multiplication	31
6. Standard Matrix Extensions	32
6.1. Zmi4: Matrix 4-bit Integer Extension	32

6.2. Zmi8: Matrix 8-bit Integer Extension	33
6.3. Zmi16: Matrix 16-bit Integer Extension	34
6.4. Zmi32: Matrix 32-bit Integer Extension	35
6.5. Zmi64: Matrix 64-bit Integer Extension	37
6.6. Zmf8e4m3: Matrix 8-bit E4M3 Float Point Extension.	38
6.7. Zmf8e5m2: Matrix 8-bit E5M2 Float Point Extension.	39
6.8. Zmf8e3m4: Matrix 8-bit E3M4 Float Point Extension.	40
6.9. Zmf16e5m10: Matrix 16-bit Half-precision Float-point (FP16) Extension	42
6.10. Zmf16e8m7: Matrix 16-bit BFloat (BF16) Extension	43
6.11. Zmf32e8m23: Matrix 32-bit Float-point Extension	44
6.12. Zmf19e8m10: Matrix 19-bit TensorFloat32 (TF32) Extension	45
6.13. Zmf64e11m52: Matrix 64-bit Float-point Extension	47
6.14. Zmv: Matrix for Vector operations	48
6.14.1. Load Instructions	48
6.14.2. Store Instructions	49
6.14.3. Data Move Instructions.	49
6.14.4. Intrinsic Example: Matrix multiplication fused with element-wise vector operation .	51
6.15. Zmi2c: Im2col Extension	52
6.15.1. CSRs	52
6.15.2. Configuration Instructions	54
6.15.3. Load Unfold Instructions	54
6.15.4. Intrinsic Example: Conv2D.	54
6.15.5. Intrinsic Example: Conv3D.	55
6.15.6. Intrinsic Example: MaxPool2D	56
6.15.7. Intrinsic Example: AvgPool2D.	57
6.16. Zmc2i: Col2im Extension	58
6.16.1. CSRs	59
6.16.2. Configuration Instructions	59
6.16.3. Store Fold Instructions	59
6.17. Zmsp: Matrix Sparsity Extension	59
7. Matrix Instruction Listing	62

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

This specification is licensed under the Creative Commons Attribution 4.0 International License (CC-BY 4.0). The full license text is available at creativecommons.org/licenses/by/4.0/.

Copyright © 2022-2024 Stream Computing Inc.

Contributors

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

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

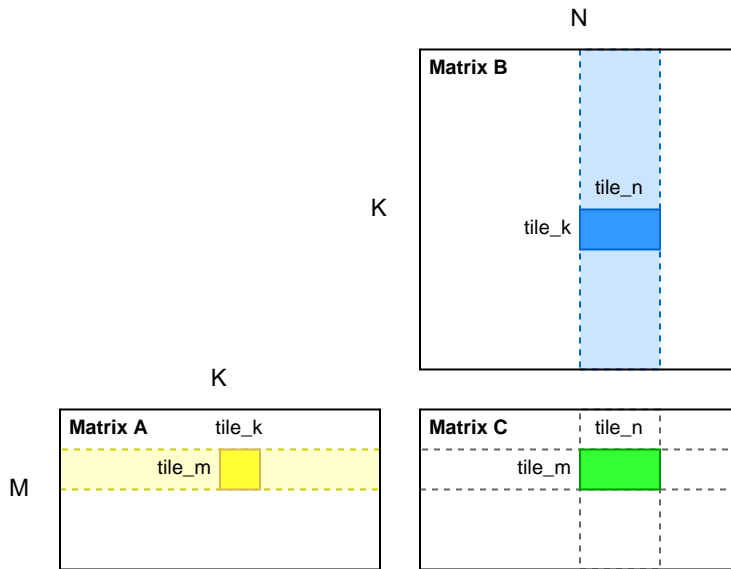
We will be very grateful to the huge number of other people who will have helped to improve this specification through their comments, reviews, feedback and questions.

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 $m_{tilem} \times m_{tilen}$, the tile size of A is $m_{tilem} \times m_{tilek}$ and the tile size of B is $m_{tilek} \times m_{tilen}$.



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 three parameters:

1. The maximum size in bits of a matrix element that any operation can produce or consume, $ELEN \geq 8$, 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. $ELEN \leq RLEN \leq MLEN$, this supports matrix tile size from 1×1 to $2^{16} \times 2^{16}$.

Chapter 3. Programmer's Model

The matrix extension adds 8 unprivileged CSRs and 16 matrix tile registers to the base scalar RISC-V ISA.

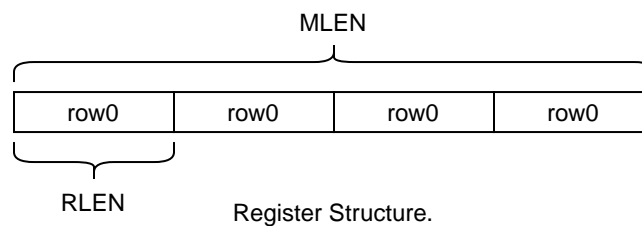
Table 1. Matrix CSRs

Address	Privilege	Name	Description
0xXXX	URO	mtype	Matrix tile data type register.
0xXXX	URO	mtilem	Tile length in m direction.
0xXXX	URO	mtilen	Tile length in n direction.
0xXXX	URO	mtilek	Tile length in k direction.
0xXXX	URO	mlemb	MLEN/8 (matrix tile register length in bytes).
0xXXX	URO	mrlenb	RLEN/8 (matrix tile register row length in bytes).
0xXXX	URW	mstart	Start element index.
0xXXX	URW	mcsr	Matrix control and status register.

3.1. Matrix Tile Registers

The matrix extension adds 16 architectural **Tile Registers** (tr0-tr15) for input and output tile 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.



tr0
tr1
tr2
tr3
tr4
tr5
tr6
tr7
tr8
tr9
tr10
tr11
tr12
tr13
tr14
tr15

Tile 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.

For widening instructions, each output element is wider than input one. An implementation should support double-width output for float-point and quad-width output for integer. To match the width of input and output, an output matrix may be written back to several registers: 2 registers for double-width output and 4 registers for quad-width output.

Widening instructions use continuous registers as their destination. For example, a double-widen instruction uses **td** and **td+1** as the output registers. As a result, the destination register index must be even (i.e., tr0, tr2, tr4, etc.). Odd-indexed **td** is reserved. A quadruple-widen instruction uses **td**, **td+1**, **td+2** and **td+3** as the output registers. As a result, the destination register index must be a multiple of 4 (i.e., tr0, tr4, tr8, etc.). Other-indexed **td** is reserved. An octuple-widen instruction uses continuous tile registers from **td** to **td+7** as the output registers. As a result, the destination register index must be a multiple of 8 (i.e., tr0 and tr8). Other-indexed **td** is reserved.

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 maintenance of the **mtype** register state.

The **mtype** register has a **mill** field, a **msew** field, a **ba** 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. 0 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 enable IEEE-754 single-precision float point (E8M23), and write 2'b10 to enable TensorFlow32 (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 tile register retain the value they 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 hold 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 execution.

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 $C = A \times B$, where the source matrices, **A** and **B**, are both organized as the original order. **mmode = 01** means $C = A \times B^T$, where **B** is transposed. **mmode = 10** means $C = A^T \times B$, where **A** is transposed.

3.6. Matrix Context Status in **mstatus** and **sstatus**

A 2-bit matrix context status field should be added to **mstatus** and shadowed in **sstatus**. 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 64-bit encoding and a new major opcode OP-M64.

Instruction formats are listed below.

Configuration instructions, where **ew** field, inst[19:17], is fixed to 100. The **imm** field supports 32-bit immediate operand.

63											44	43					39	38					32				
imm[31:12]											sub-funct5				funct7												
31											20	19	17	16	12	11					7	6					0
imm[11:0]											111		rd		OP-M64				0111111								

Configuration Immediate Instructions.

63																	44	43	39	38					32		
0...000																sub-funct5				major-funct7							
31																	25	24	20	19	17	16	12	11	7	6	0
0000000								rs1				111	rd				OP-M64				0111111						

Configuration Instructions.

Data move instructions, where **di** field indicates the moving direction, and **rc** field indicates the moving dimension (row or column).

63												44 43				39 38				32			
0...000												sub-funct5				major-funct7							
31 30 29			25 24			20 19			17 16			12 11		7 6		0							
di rc		rs2			rs1/ts1			eew		rd/td			OP-M64				0111111						

Data Move Instructions.

Load & store instructions, where **ls** field indicates the type (load or store), and **tr** field indicates if the operand in register is transposed. **ew** field (000-011) indicates the effective element width. **mt** field indicates the type of matrix (00 for output/accumulation matrix, 01 for left matrix and 10 for right matrix).

63												44 43				39 38				32			
0...000												sub-funct5				major-funct7							
31 30 29			25 24			20 19			17 16			12 11		7 6		0							
ls	tr	rs2				rs1				eew		ts3/td		OP-M64				0111111					

Load/Store Instructions.

Matrix multiplication instructions, where **fp** field indicates if the operation is float point, and **sn** field indicates if the source operands are signed (for integer). **ew** field indicates the effective element width (000-011 for int8-int64, 111 for int4, and 100 to use **msew** field of **mtype** CSR). **frm** field

indicates the rounding mode of float-point result, where the encoding is the same as RVF. **spm** field indicates the sparsing mode (01 for left matrix sparsing, 10 for right matrix sparsing, and 00 without sparsing) and **tsp** field indicates the index register. **enw** means if the destination elements are narrowed or widen (111 for half narrowing, 001 for double widening, 010 for quadruple widening, 011 for octuple widening, and 000 without change of width).

63		57		56	52		51	49		48	46		45	44	43	39		38	32				
0000000					tsp			enw		frm		spm		sub-funct5			major-funct7						
31		30		29		25		24	20		19	17		16		12		11	7		6	0	
fp	sn	ts2			ts1			eew		td			OP-M64			0111111							

Matrix Multiplication Instructions.

Element-wise instructions, where **fp** field indicates if the operation is float point, and **sn** field indicates if the source operands are signed (for integer). **eew** field indicates the effective element width (000-011 for int8-int64 and 100 to use **msew** field of **mtype** CSR). **frm** field indicates the rounding mode of float-point result, where the encoding is the same as RVF. **opm** field indicates the operand mode (00 for double-matrix operands, 01 for single-matrix operands, 10 for matrix-scalar operands and 11 for matrix-immediate operands). **enw** means if the destination elements are narrowed or widen. **imm** field supports 17-bit immediate operand.

63				52 51				49 48				46 45 44 43				39 38				32							
imm[16:5]								enw				frm				opm				sub-funct5				major-funct7			
31 30 29				25 24				20 19				17 16				12 11				7 6				0			
fp		sn		imm[4:0]				ts1				eew				td				OP-M64				0111111			

Element-wise Immediate Instructions.

63				52 51				49 48				46 45 44 43				39 38				32							
000000000000								enw		frm		opm		sub-funct5				major-funct7									
31 30 29				25 24				20 19				17 16				12 11				7 6				0			
fp		sn		ts2				ts1				eew		td				OP-M64				0111111					

Element-wise Instructions.

Type-convert instructions, where **fd** field indicates if the destination operation is float point, and **sn** field indicates if the integer operand is signed. **eew** field indicates the effective element width (000-011 for int8-int64 and 100 to use **msew** field of **mtype** CSR). **frm** field indicates the rounding mode of float-point result, where the encoding is the same as RVF. **lmul** field indicates the group size of source registers. **enw** means if the destination elements are narrowed or widen.

63				52 51		49 48		46 45 44 43		39 38		32			
000000000000						enw		frm		lmul		sub-funct5		major-funct7	
31 30 29			25 24		20 19		17 16		12 11		7 6		0		
fd sn		00000		ts1		eew		td		OP-M64		0111111			

Type-convert Instructions.

4.2. Configuration-Setting Instructions

Due to hardware resource constraints, one of the common ways to handle large-sized matrix multiplications 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]** 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 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 mseg field.
msetseg rd, imm        # rd = new mtype, mseg = imm[2:0].
msetba  rd, imm        # rd = new mtype, mba = imm[0].

# 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, {fp8|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[23:20]` and the value is specified by `inst[63:44]`.

Table 4. Field to be set or unset

inst[23:20]	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.

<code>msettilemi rd, imm</code>	# rd = new mtilem, imm = ATM
<code>msettilem rd, rs1</code>	# rd = new mtilem, rs1 = ATM
<code>msettileki rd, imm</code>	# rd = new mtilek, imm = ATN
<code>msettilek rd, rs1</code>	# rd = new mtilek, rs1 = ATN
<code>msettileni rd, imm</code>	# rd = new mtilen, imm = ATK
<code>msettilen rd, rs1</code>	# 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.
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**),

- $TMAX = MLEN / RLEN$
- $TKMAX = \min(MLEN / RLEN, RLEN / SEW)$
- $TNMAX = RLEN / SEW$

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

- $TMAX = MLEN / RLEN$
- $TKMAX = RLEN / SEW$
- $TNMAX = MLEN / RLEN$

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

- $TMAX = \min(MLEN / RLEN, RLEN / SEW)$
- $TKMAX = MLEN / RLEN$
- $TNMAX = RLEN / SEW$

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

SEW=8, TMAX=4, TKMAX=4, TNMAX=8	# 4x4x8 8-bit matmul
SEW=16, TMAX=4, TKMAX=4, TNMAX=4	# 4x4x4 16-bit matmul
SEW=32, TMAX=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 TMAX/TKMAX/TNMAX
x0	x0	Value in mtilem/mtilek/mtilen	Keep existing mtilem/mtilek/mtilen if less than TMAX/TKMAX/TNMAX

For the **msettile{m|k|n}i** instructions, the **ATM / ATK / ATN** is encoded as a 32-bit unsigned immediate in the **rs1**.

4.2.3. Constraints on Setting **mtilem/mtilek/mtilen**

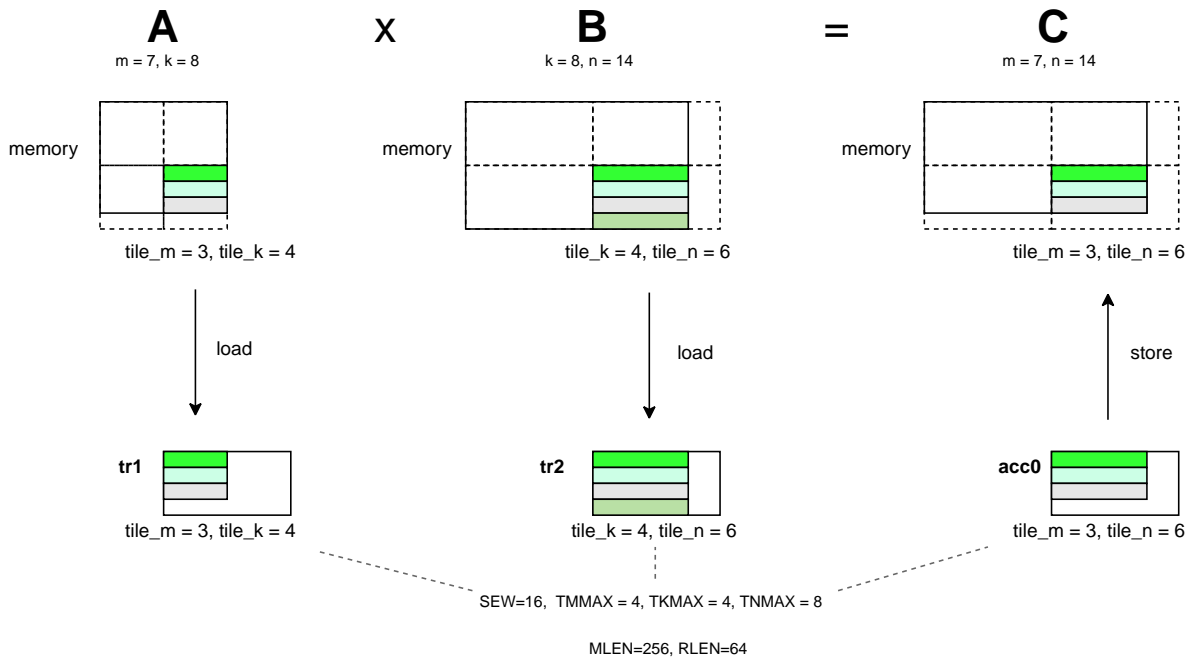
The **msettile{m|k|n}[i]** instructions first set **TMAX/TKMAX/TNMAX** according to the **mtype** CSR,

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.

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

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

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

# For output matrix, C
# tile size = mtilem * mtilen
mlce8.m  td, (rs1), rs2     # 8-bit output tile load
mlce16.m td, (rs1), rs2     # 16-bit output tile load
mlce32.m td, (rs1), rs2     # 32-bit output tile load
mlce64.m td, (rs1), rs2     # 64-bit output tile load
```

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

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

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

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

```

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

```

4.3.2. Store Instructions

Store a matrix tile to memory.

```

# ts3 store data, rs1 base address, rs2 row byte stride

# For left matrix, A
# tile size = mtilem * mtilenk
msae8.m  ts3, (rs1), rs2      # 8-bit left tile store
msae16.m ts3, (rs1), rs2     # 16-bit left tile store
msae32.m ts3, (rs1), rs2     # 32-bit left tile store
msae64.m ts3, (rs1), rs2     # 64-bit left tile store

# For right matrix, B
# tile size = mtilenk * mtilen
msbe8.m  ts3, (rs1), rs2      # 8-bit right tile store
msbe16.m ts3, (rs1), rs2     # 16-bit right tile store
msbe32.m ts3, (rs1), rs2     # 32-bit right tile store
msbe64.m ts3, (rs1), rs2     # 64-bit right tile store

# For output matrix, C
# tile size = mtilen * mtilen
msce8.m  ts3, (rs1), rs2      # 8-bit output tile store
msce16.m ts3, (rs1), rs2     # 16-bit output tile store
msce32.m ts3, (rs1), rs2     # 32-bit output tile store
msce64.m ts3, (rs1), rs2     # 64-bit output tile store

```

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

```

# ts3 store data, rs1 base address, rs2 row byte stride

# For left matrix, A
# tile size = mtilenk * mtilen
msate8.m  ts3, (rs1), rs2      # 8-bit left tile store
msate16.m ts3, (rs1), rs2     # 16-bit left tile store
msate32.m ts3, (rs1), rs2     # 32-bit left tile store
msate64.m ts3, (rs1), rs2     # 64-bit left tile store

```

```

# For right matrix, B
# tile size = mtilen * mtilek
msbte8.m  ts3, (rs1), rs2      # 8-bit right tile store
msbte16.m ts3, (rs1), rs2     # 16-bit right tile store
msbte32.m ts3, (rs1), rs2     # 32-bit right tile store
msbte64.m ts3, (rs1), rs2     # 64-bit right tile store

# For output matrix, C
# tile size = mtilen * mtilen
mscte8.m  ts3, (rs1), rs2      # 8-bit output tile store
mscte16.m ts3, (rs1), rs2     # 16-bit output tile store
mscte32.m ts3, (rs1), rs2     # 32-bit output tile store
mscte64.m ts3, (rs1), rs2     # 64-bit output tile store

```

4.3.3. Whole Matrix Load & Store Instructions

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

```

mlre8.m   td, (rs1), rs2      # 8-bit whole matrix load
mlre16.m  td, (rs1), rs2     # 16-bit whole matrix load
mlre32.m  td, (rs1), rs2     # 32-bit whole matrix load
mlre64.m  td, (rs1), rs2     # 64-bit whole matrix load

```

Load a whole matrix from memory without considering the tile size, where the matrix on memory is transposed.

```

mlrte8.m  td, (rs1), rs2     # 8-bit whole matrix load
mlrte16.m td, (rs1), rs2     # 16-bit whole matrix load
mlrte32.m td, (rs1), rs2     # 32-bit whole matrix load
mlrte64.m td, (rs1), rs2     # 64-bit whole matrix load

```

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

```

msre8.m   ts3, (rs1), rs2     # 8-bit whole matrix store
msre16.m  ts3, (rs1), rs2     # 16-bit whole matrix store
msre32.m  ts3, (rs1), rs2     # 32-bit whole matrix store
msre64.m  ts3, (rs1), rs2     # 64-bit whole matrix store

```

Store a whole matrix to memory without considering the tile size, where the matrix on memory is transposed.

```

msrte8.m  ts3, (rs1), rs2     # 8-bit whole matrix store
msrte16.m ts3, (rs1), rs2     # 16-bit whole matrix store
msrte32.m ts3, (rs1), rs2     # 32-bit whole matrix store

```

```
msrte64.m ts3, (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 and Integer

The basic data move instructions 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] = ts1[i, j], i = rs2[15:0], j = rs2[XLEN-1:16]
mmve8.x.s rd, ts1, rs2
mmve16.x.s rd, ts1, rs2
mmve32.x.s rd, ts1, rs2
mmve64.x.s rd, ts1, rs2

# td[i, j] = x[rs1], i = rs2[15:0], j = rs2[XLEN-1:16]
mmve8.s.x td, rs1, rs2
mmve16.s.x td, rs1, rs2
mmve32.s.x td, rs1, rs2
mmve64.s.x td, rs1, rs2
```

The **mmve*.x.s** instruction copies a single SEW-wide element of the tile register to an integer register, where the element coordinates are specified by rs2. If $SEW > XLEN$, the least-significant XLEN bits are transferred. If $SEW < XLEN$, the value is sign-extended to XLEN bits.

The **mmve*.s.x** instruction copies an integer register to an element of the destination tile 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 value 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**.



The pseudo-instruction **mmve*.m.m td, ts1** to move a whole matrix between two tile registers can be implemented as **mmve*.s.x td, x0, x0; mbce*.m td, td; madd*.mm td, ts1, td**.

4.4.2. Data Move Instructions between Matrix and Float-point

Float point data move instructions are similar with integer.

```
# f[rd] = ts1[i, j], i = rs2[15:0], j = rs2[XLEN-1:16]
```



```

mfmve8.f.s rd, ts1, rs2
mfmve16.f.s rd, ts1, rs2
mfmve32.f.s rd, ts1, rs2
mfmve64.f.s rd, ts1, rs2

# td[i, j] = f[rs1], i = rs2[15:0], j = rs2[XLEN-1:16]
mfmve8.s.f td, rs1, rs2
mfmve16.s.f td, rs1, rs2
mfmve32.s.f td, rs1, rs2
mfmve64.s.f td, rs1, rs2

```

4.4.3. Data Broadcast Instructions

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

```

# Broadcast the first row to fill the whole matrix.
mbcar.m td, ts1
mbcbr.m td, ts1
mbccr.m td, ts1

# Broadcast the first column to fill the whole matrix.
mbcace8.m td, ts1
mbcace16.m td, ts1
mbcace32.m td, ts1
mbcace64.m td, ts1

mbcbce8.m td, ts1
mbcbce16.m td, ts1
mbcbce32.m td, ts1
mbcbce64.m td, ts1

mbccce8.m td, ts1
mbccce16.m td, ts1
mbccce32.m td, ts1
mbccce64.m td, ts1

# Broadcast the first element to fill the whole matrix.
mbcaee8.m td, ts1
mbcaee16.m td, ts1
mbcaee32.m td, ts1
mbcaee64.m td, ts1

mbcbee8.m td, ts1
mbcbee16.m td, ts1
mbcbee32.m td, ts1
mbcbee64.m td, ts1

```

```
mbccee8.m  td, ts1
mbccee16.m td, ts1
mbccee32.m td, ts1
mbccee64.m td, ts1
```

4.4.4. Matrix Transpose Instructions

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

```
# Transpose square matrix.
mtae8.m  td, ts1
mtae16.m td, ts1
mtae32.m td, ts1
mtae64.m td, ts1

mtbe8.m  td, ts1
mtbe16.m td, ts1
mtbe32.m td, ts1
mtbe64.m td, ts1

mtce8.m  td, ts1
mtce16.m td, ts1
mtce32.m td, ts1
mtce64.m td, ts1
```

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 **ts1** and **ts2** respectively, and the output matrix tile is a matrix **tile register** specified by **td** or a group matrix **tile registers** started from **td**.

```
# Unsigned integer matrix multiplication and add, td = td + ts1 * ts2.
mmau.[dw].mm  td, ts1, ts2      # unsigned int64, output no-widen
mmau.[w].mm    td, ts1, ts2      # unsigned int32, output no-widen
mmau.[h].mm    td, ts1, ts2      # unsigned int16, output no-widen
mqmau.[b].mm   td, ts1, ts2      # unsigned int8, output quad-widen
momau.[hb].mm  td, ts1, ts2      # unsigned int4, output oct-widen

msmau.[dw].mm  td, ts1, ts2      # unsigned int64, output no-widen and saturated
msmau.[w].mm   td, ts1, ts2      # unsigned int32, output no-widen and saturated
msmau.[h].mm   td, ts1, ts2      # unsigned int16, output no-widen and saturated
```

```

msqmau.[b].mm  td, ts1, ts2      # unsigned int8, output quad-widen and saturated
msomau.[hb].mm td, ts1, ts2      # unsigned int4, output oct-widen and saturated

# Signed integer matrix multiplication and add, td = td + ts1 * ts2.
mma.[dw].mm    td, ts1, ts2      # signed int64, output no-widen
mma.[w].mm     td, ts1, ts2      # signed int32, output no-widen
mma.[h].mm     td, ts1, ts2      # signed int16, output no-widen
mqma.[b].mm    td, ts1, ts2      # signed int8, output quad-widen
moma.[hb].mm   td, ts1, ts2      # signed int4, output oct-widen

msma.[dw].mm   td, ts1, ts2      # signed int64, output no-widen and saturated
msma.[w].mm    td, ts1, ts2      # signed int32, output no-widen and saturated
msma.[h].mm    td, ts1, ts2      # signed int16, output no-widen and saturated
msqma.[b].mm   td, ts1, ts2      # signed int8, output quad-widen and saturated
msoma.[hb].mm  td, ts1, ts2      # signed int4, output oct-widen and saturated

# Float point matrix multiplication and add, td = td + ts1 * ts2.
mfma.[d].mm    td, ts1, ts2      # 64-bit float point
mfma.[f].mm    td, ts1, ts2      # 32-bit float point
mfma.[hf].mm   td, ts1, ts2      # 16-bit float point

mfwma.[f].mm   td, ts1, ts2      # 32-bit float point, output double-widen
mfwma.[hf].mm  td, ts1, ts2      # 16-bit float point, output double-widen
mfwma.[cf].mm  td, ts1, ts2      # 8-bit float point, output double-widen
mfqma.[cf].mm  td, ts1, ts2      # 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 always with size `mtilem x mtilen`.

```
# Unsigned integer matrix element-wise add.
# td[i,j] = ts1[i,j] + ts2[i,j]
maddu.[b|h|w|dw].mm    td, ts1, ts2
msaddu.[b|h|w|dw].mm    td, ts1, ts2 # output saturated
mwaddu.[b|h|w].mm       td, ts1, ts2 # output double widen

# td[i,j] = ts1[i,j] + unsigned_ext(imm)
maddu.[b|h|w|dw].mi     td, ts1, imm
msaddu.[b|h|w|dw].mi     td, ts1, imm # output saturated
mwaddu.[b|h|w].mi       td, ts1, imm # output double widen

# Signed integer matrix element-wise add.
# td[i,j] = ts1[i,j] + ts2[i,j]
madd.[b|h|w|dw].mm      td, ts1, ts2
msadd.[b|h|w|dw].mm      td, ts1, ts2 # output saturated
mwadd.[b|h|w].mm        td, ts1, ts2 # output double widen

# td[i,j] = ts1[i,j] + signed_ext(imm)
madd.[b|h|w|dw].mi      td, ts1, imm
msadd.[b|h|w|dw].mi      td, ts1, imm # output saturated
mwadd.[b|h|w].mi        td, ts1, imm # output double widen

# Unsigned integer matrix element-wise subtract.
# td[i,j] = ts1[i,j] - ts2[i,j]
msubu.[b|h|w|dw].mm     td, ts1, ts2
mssubu.[b|h|w|dw].mm     td, ts1, ts2 # output saturated
mwsubu.[b|h|w].mm       td, ts1, ts2 # output double widen

# td[i,j] = ts1[i,j] - unsigned_ext(imm)
msubu.[b|h|w|dw].mi     td, ts1, imm
mssubu.[b|h|w|dw].mi     td, ts1, imm # output saturated
mwsubu.[b|h|w].mi       td, ts1, imm # output double widen

# Signed integer matrix element-wise subtract.
# td[i,j] = ts1[i,j] - ts2[i,j]
msub.[b|h|w|dw].mm      td, ts1, ts2
mssub.[b|h|w|dw].mm      td, ts1, ts2 # output saturated
mwsub.[b|h|w].mm        td, ts1, ts2 # output double widen

# td[i,j] = ts1[i,j] - signed_ext(imm)
msub.[b|h|w|dw].mi      td, ts1, imm
mssub.[b|h|w|dw].mi      td, ts1, imm # output saturated
mwsub.[b|h|w].mi        td, ts1, imm # output double widen
```

```

# td[i,j] = unsigned_ext(imm) - ts1[i,j]
mrsubu.[b|h|w|dw].mi  td, ts1, imm
msrsubu.[b|h|w|dw].mi td, ts1, imm # output saturated
mwrsubu.[b|h|w].mi    td, ts1, imm # output double widen

# td[i,j] = signed_ext(imm) - ts1[i,j]
mrsub.[b|h|w|dw].mi  td, ts1, imm
msrsub.[b|h|w|dw].mi td, ts1, imm # output saturated
mwrsub.[b|h|w].mi    td, ts1, imm # output double widen

# Integer matrix element-wise minimum.
# td[i,j] = min{ts1[i,j], ts2[i,j]}
mminu.[b|h|w|dw].mm  td, ts1, ts2
mmin.[b|h|w|dw].mm   td, ts1, ts2

# Integer matrix element-wise maximum.
# td[i,j] = max{ts1[i,j], ts2[i,j]}
mmaxu.[b|h|w|dw].mm  td, ts1, ts2
mmax.[b|h|w|dw].mm   td, ts1, ts2

# Integer matrix bit-wise logic.
mand.mm              td, ts1, ts2
mor.mm               td, ts1, ts2
mxor.mm              td, ts1, ts2

# Integer matrix element-wise shift.
msll.[b|h|w|dw].mm   td, ts1, ts2
msrl.[b|h|w|dw].mm   td, ts1, ts2
msra.[b|h|w|dw].mm   td, ts1, ts2

msll.[b|h|w|dw].mi    td, ts1, imm
msrl.[b|h|w|dw].mi    td, ts1, imm
msra.[b|h|w|dw].mi    td, ts1, imm

# Integer matrix element-wise multiply.
# td[i,j] = ts1[i,j] * ts2[i,j]
mmul.[b|h|w|dw].mm    td, ts1, ts2 # signed, returning low bits of product
mmulh.[b|h|w|dw].mm    td, ts1, ts2 # signed, returning high bits of product
mmulhu.[b|h|w|dw].mm   td, ts1, ts2 # unsigned, returning high bits of product
mmulhsu.[b|h|w|dw].mm td, ts1, ts2 # signed-unsigned, returning high bits of
product

# td[i,j] = ts1[i,j] * imm
mmul.[b|h|w|dw].mi    td, ts1, imm # signed_ext(imm)
mmulh.[b|h|w|dw].mi    td, ts1, imm # signed_ext(imm)
mmulhu.[b|h|w|dw].mi   td, ts1, imm # unsigned_ext(imm)
mmulhsu.[b|h|w|dw].mi td, ts1, imm # unsigned_ext(imm)

```

```

# Saturated integer matrix element-wise multiply.
msmul.[b|h|w|dw].mm    td, ts1, ts2 # signed
msmulu.[b|h|w|dw].mm   td, ts1, ts2 # unsigned
msmulu.[b|h|w|dw].mm   td, ts1, ts2 # signed-unsigned

msmul.[b|h|w|dw].mi    td, ts1, imm # signed
msmulu.[b|h|w|dw].mi   td, ts1, imm # unsigned
msmulu.[b|h|w|dw].mi   td, ts1, imm # signed-unsigned

# Widening integer matrix element-wise multiply.
mwmul.[b|h|w].mm       td, ts1, ts2 # signed
mwmulu.[b|h|w].mm      td, ts1, ts2 # unsigned
mwmulu.[b|h|w].mm      td, ts1, ts2 # signed-unsigned

mwmul.[b|h|w].mi       td, ts1, imm # signed
mwmulu.[b|h|w].mi      td, ts1, imm # unsigned
mwmulu.[b|h|w].mi      td, ts1, imm # signed-unsigned

# Float matrix element-wise add.
# td[i,j] = ts1[i,j] + ts2[i,j]
mfadd.[hf|f|d].mm      td, ts1, ts2
mfwadd.[hf|f].mm        td, ts1, ts2 # output double widen

# Float matrix element-wise subtract.
# td[i,j] = ts1[i,j] - ts2[i,j]
mfsub.[hf|f|d].mm      td, ts1, ts2
mfwsb.[hf|f].mm        td, ts1, ts2 # output double widen

# Float matrix element-wise minimum.
# td[i,j] = min{ts1[i,j], ts2[i,j]}
mfmin.[hf|f|d].mm      td, ts1, ts2

# Float matrix element-wise maximum.
# td[i,j] = max{ts1[i,j], ts2[i,j]}
mfmax.[hf|f|d].mm      td, ts1, ts2

# Float matrix element-wise multiply.
# td[i,j] = ts1[i,j] * ts2[i,j]
mfmul.[hf|f|d].mm      td, ts1, ts2
mfwmul.[hf|f].mm       td, ts1, ts2 # output double widen

# Float matrix element-wise divide.
# td[i,j] = ts1[i,j] / ts2[i,j]
mfdiv.[hf|f|d].mm      td, ts1, ts2

# Float matrix element-wise square root.
# td[i,j] = ts1[i,j] ^ (1/2)
mfsqrt.[hf|f|d].m      td, ts1

```



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

```
# Convert integer to integer
mwcvtu.b.hb.m    td, ts1      # uint4 to uint8
mwcvt.b.hb.m     td, ts1      # int4 to int8

mncvtu.hb.b.m    td, ts1, lmul # uint8 to uint4
mncvt.hb.b.m     td, ts1, lmul # int8 to int4
msncvtu.hb.b.m   td, ts1, lmul # uint8 to uint4, saturated
msncvt.hb.b.m    td, ts1, lmul # int8 to int4, saturated

# Convert float to float
mfcvt.bf.hf.m    td, ts1      # fp16 to bf16
mfcvt.hf.bf.m    td, ts1      # bf16 to fp16

mfwcvt.fw.f.m    td, ts1      # single-width float to double-width float
mfwcvt.hf.cf.m   td, ts1      # fp8 to fp16
mfwcvt.f.hf.m    td, ts1      # fp16 to fp32
mfwcvt.d.f.m     td, ts1      # fp32 to fp64

mfncvt.f.fw.m    td, ts1, lmul # double-width float to single-width float
mfncvt.cf.hf.m   td, ts1, lmul # fp16 to fp8
mfncvt.hf.f.m    td, ts1, lmul # fp32 to fp16
mfncvt.f.d.m     td, ts1, lmul # fp64 to fp32

# Convert integer to float
mfcvtu.f.x.m     td, ts1      # uint to float
mfcvtu.hf.h.m    td, ts1      # uint16 to fp16
mfcvtu.f.w.m     td, ts1      # uint32 to fp32
mfcvtu.d.dw.m    td, ts1      # uint64 to fp64

mfcvt.f.x.m      td, ts1      # int to float
mfcvt.hf.h.m     td, ts1      # int16 to fp16
mfcvt.f.w.m      td, ts1      # int32 to fp32
mfcvt.d.dw.m     td, ts1      # int64 to fp64

mfwcvtu.fw.x.m   td, ts1      # single-width uint to double-width float
mfwcvtu.fq.x.m   td, ts1      # single-width uint to quad-width float
mfwcvtu.hf.b.m   td, ts1      # uint8 to fp16
mfwcvtu.f.b.m    td, ts1      # uint8 to fp32
mfwcvtu.f.h.m    td, ts1      # uint16 to fp32
mfwcvtu.d.w.m    td, ts1      # uint32 to fp64
```

mfwevt.fw.x.m	td, ts1	# single-width int to double-width float
mfwevt.fq.x.m	td, ts1	# single-width int to quad-width float
mfwevt.hf.b.m	td, ts1	# int8 to fp16
mfwevt.f.b.m	td, ts1	# int8 to fp32
mfwevt.f.h.m	td, ts1	# int16 to fp32
mfwevt.d.w.m	td, ts1	# int32 to fp64
mfncvtu.f.xw.m	td, ts1, lmul	# double-width uint to float
mfncvtu.hf.w.m	td, ts1, lmul	# uint32 to fp16
mfncvtu.f.dw.m	td, ts1, lmul	# uint64 to fp32
mfncvt.f.xw.m	td, ts1, lmul	# double-width int to float
mfncvt.hf.w.m	td, ts1, lmul	# int32 to fp16
mfncvt.f.dw.m	td, ts1, lmul	# int64 to fp32
# Convert float to integer		
mfcvtu.x.f.m	td, ts1	# float to uint
mfcvtu.h.hf.m	td, ts1	# fp16 to uint16
mfcvtu.w.f.m	td, ts1	# fp32 to uint32
mfcvtu.dw.d.m	td, ts1	# fp64 to uint64
mfcvt.x.f.m	td, ts1	# float to int
mfcvt.h.hf.m	td, ts1	# fp16 to int16
mfcvt.w.f.m	td, ts1	# fp32 to int32
mfcvt.dw.d.m	td, ts1	# fp64 to int64
mfwevtu.xw.f.m	td, ts1	# single-width float to double-width uint
mfwevtu.w.hf.m	td, ts1	# fp16 to uint32
mfwevtu.dw.f.m	td, ts1	# fp32 to uint64
mfwevt.xw.f.m	td, ts1	# single-width float to double-width int
mfwevt.w.hf.m	td, ts1	# fp16 to int32
mfwevt.dw.f.m	td, ts1	# fp32 to int64
mfncvtu.x.fw.m	td, ts1, lmul	# double-width float to single-width uint
mfncvtu.x.fq.m	td, ts1, lmul	# quad-width float to single-width uint
mfncvtu.b.hf.m	td, ts1, lmul	# fp16 to uint8
mfncvtu.b.f.m	td, ts1, lmul	# fp32 to uint8
mfncvtu.h.f.m	td, ts1, lmul	# fp32 to uint16
mfncvtu.w.d.m	td, ts1, lmul	# fp64 to uint32
mfncvt.x.fw.m	td, ts1, lmul	# double-width float to single-width int
mfncvt.x.fq.m	td, ts1, lmul	# quad-width float to single-width int
mfncvt.b.hf.m	td, ts1, lmul	# fp16 to int8
mfncvt.b.f.m	td, ts1, lmul	# fp32 to int8
mfncvt.h.f.m	td, ts1, lmul	# fp32 to int16
mfncvt.w.d.m	td, ts1, lmul	# 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 = mwsb_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 = mfnvvt_f_fw_m(out, m2); // 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 = mwsb_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, m2);          // 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. 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 instructions are not available for int4 format.

The type-convert instructions with suffix .hb are used for int4 format.

Four octuple-widen instructions are added to support int4 matrix multiplication. So the output type is always 32-bit integer.

```

momau.[hb].mm  td, ts1, ts2    # unsigned int4, output oct-widen
msomau.[hb].mm td, ts1, ts2    # unsigned int4, output oct-widen and saturated

moma.[hb].mm   td, ts1, ts2    # signed int4, output oct-widen
msoma.[hb].mm  td, ts1, ts2    # signed int4, output oct-widen and saturated

```

6.2. 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.
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 used for int8 format.

Four quadruple-widen instructions are added to support int8 matrix multiplication. So the output type is always 32-bit integer.

```
mqmau.[b].mm td, ts1, ts2  # unsigned int8, output quad-widen
msqmau.[b].mm td, ts1, ts2  # unsigned int8, output quad-widen and saturated

mqma.[b].mm   td, ts1, ts2  # signed int8, output quad-widen
msqma.[b].mm  td, ts1, ts2  # signed int8, output quad-widen and saturated
```

6.3. 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.
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.

Bits	Name	Description
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**.

```

msetypei rd, imm      # rd = new mtype, imm = new mtype setting.
msetype  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 used for int16 format.

Four no-widen instructions are added to support int16 matrix multiplication. So the output type is always 16-bit integer.

```

mmau.[h].mm td, ts1, ts2      # unsigned int16, output no-widen
msmau.[h].mm td, ts1, ts2     # unsigned int16, output no-widen and saturated

mma.[h].mm   td, ts1, ts2     # signed int16, output no-widen
msma.[h].mm  td, ts1, ts2     # signed int16, output no-widen and saturated

```

6.4. 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.
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 used for int32 format.

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

```

mmau.[w].mm  td, ts1, ts2      # unsigned int32, output no-widen
msmau.[w].mm td, ts1, ts2      # unsigned int32, output no-widen and saturated
mma.[w].mm   td, ts1, ts2      # signed int32, output no-widen
```

```
msma.[w].mm  td, ts1, ts2      # signed int32, output no-widen and saturated
```

6.5. 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 used for `int64` format.

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

<code>mmau.[dw].mm</code>	<code>td, ts1, ts2</code>	# unsigned <code>int64</code> , output no-widen
<code>msmau.[dw].mm</code>	<code>td, ts1, ts2</code>	# unsigned <code>int64</code> , output no-widen and saturated
<code>mma.[dw].mm</code>	<code>td, ts1, ts2</code>	# signed <code>int64</code> , output no-widen
<code>msma.[dw].mm</code>	<code>td, ts1, ts2</code>	# signed <code>int64</code> , output no-widen and saturated

6.6. 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.
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 used 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.

```
mfwma.[cf].mm td, ts1, ts2    # 8-bit float point, output double-widen
mfqma.[cf].mm td, ts1, ts2    # 8-bit float point, output quad-widen
```

6.7. 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.

Bits	Name	Description
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 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 used 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.

```

mfwma.[cf].mm td, ts1, ts2    # 8-bit float point, output double-widen
mfqma.[cf].mm td, ts1, ts2    # 8-bit float point, output quad-widen
```

6.8. 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.

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 **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 used 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.

```

mfwma.[cf].mm td, ts1, ts2  # 8-bit float point, output double-widen
mfqma.[cf].mm td, ts1, ts2  # 8-bit float point, output quad-widen
```

6.9. 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 **mf16** 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	mf16	64-bit float point enabling.
13:12	mf32[1:0]	32-bit float point enabling.
11:10	mf16[1:0]	16-bit float point enabling.
9:8	mf8[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 **mf16** 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 **mf16** 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, fp16     # rd = new mtype, set mf16 = 01 to enable FP16 type.
munsetfp  rd, fp16     # rd = new mtype, set mf16 = 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 used 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.

<code>mfma.[hf].mm</code>	<code>td, ts1, ts2</code>	# 16-bit float point, output no-widen
<code>mfwma.[hf].mm</code>	<code>td, ts1, ts2</code>	# 16-bit float point, output double-widen

6.10. 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.
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[i]` or set independently by `msetfp` or `munsetfp`.

```

msetypei rd, imm      # rd = new mtype, imm = new mtype setting.
msetype  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 used 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.

```

mfma.[hf].mm td, ts1, ts2    # 16-bit float point, output no-widen
mfwma.[hf].mm td, ts1, ts2   # 16-bit float point, output double-widen

```

6.11. 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.
3	mint4	4-bit integer enabling.

Bits	Name	Description
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[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, 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 used for FP32 format.

A no-widen instruction is added to support FP32 matrix multiplication. So the output type is 32-bit float point.

```
mfma.[f].mm  td, ts1, ts2      # 32-bit float point, output no-widen
```

6.12. 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.

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 **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[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, 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 implementations 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 with **.f** suffix are used 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).

```
mfma.[f].mm  td, ts1, ts2      # 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).

6.13. 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[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, 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 used for FP64 format.

A no-widen instruction is added to support FP64 matrix multiplication. And a double-widen instruction is added to support FP32 widening matrix multiplication. The output type is always 64-bit float point (FP64).

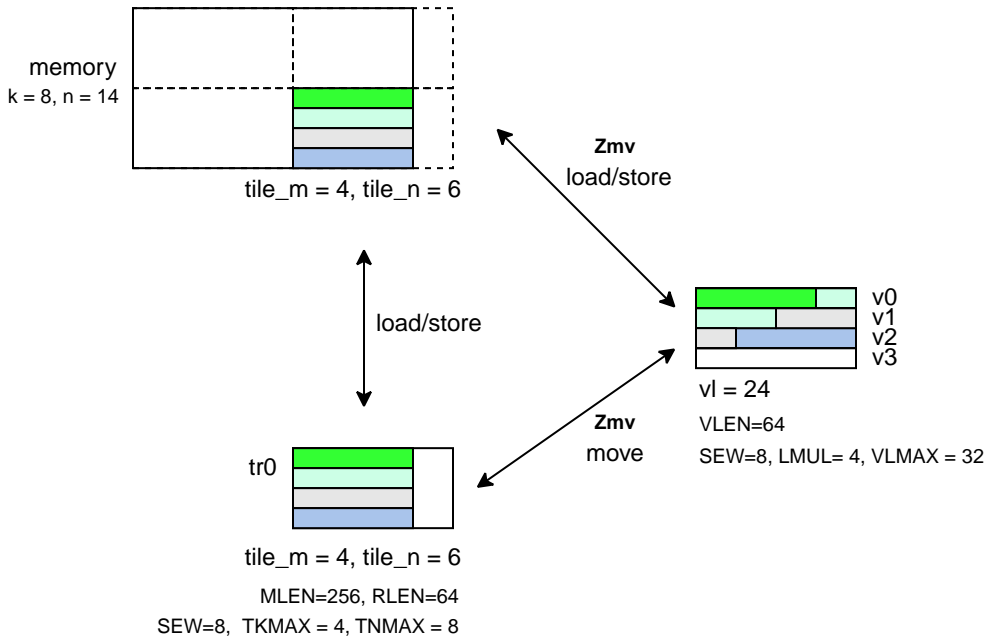
```
mfma.[d].mm td, ts1, ts2      # 64-bit float point, output no-widen
mfwma.[f].mm td, ts1, ts2     # 32-bit float point, output double-widen
```

6.14. 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.



6.14.1. 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
```

6.14.2. 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
```

6.14.3. 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 $\min(\text{VLEN}/\text{SEW} * \text{VLMUL}, \text{matrix_size})$.

- For matrix A, $\text{matrix_size} = \text{mtilem} * \text{mtilek}$.
- For matrix B, $\text{matrix_size} = \text{mtilek} * \text{mtilen}$.

- For matrix C, $\text{matrix_size} = \text{mtilem} * \text{mtilen}$.

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

# vd[(i - rs2) * mtilek + j] = td[i, j], i = rs2 .. rs2 + mtilem - 1
mmvare8.v.m   vd, ts1, rs2
mmvare16.v.m  vd, ts1, rs2
mmvare32.v.m  vd, ts1, rs2
mmvare64.v.m  vd, ts1, rs2

# vd[(i - rs2) * mtilen + j] = td[i, j], i = rs2 .. rs2 + mtilek - 1
mmvbre8.v.m   vd, ts1, rs2
mmvbre16.v.m  vd, ts1, rs2
mmvbre32.v.m  vd, ts1, rs2
mmvbre64.v.m  vd, ts1, rs2

# vd[(i - rs2) * mtilen + j] = td[i, j], i = rs2 .. rs2 + mtilem - 1
mmvcre8.v.m   vd, ts1, rs2
mmvcre16.v.m  vd, ts1, rs2
mmvcre32.v.m  vd, ts1, rs2
mmvcre64.v.m  vd, ts1, rs2

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

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

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

# Data move between matrix register columns and vector registers.

# vd[(j - rs2) * mtilem + i] = td[i, j], j = rs2 .. rs2 + mtilek - 1
mmvace8.v.m   vd, ts1, rs2
mmvace16.v.m  vd, ts1, rs2
mmvace32.v.m  vd, ts1, rs2
mmvace64.v.m  vd, ts1, rs2
```

```

# vd[(j - rs2) * mtilek + i] = td[i, j], j = rs2 .. rs2 + mtilek - 1
mmvbce8.v.m   vd, ts1, rs2
mmvbce16.v.m  vd, ts1, rs2
mmvbce32.v.m  vd, ts1, rs2
mmvbce64.v.m  vd, ts1, rs2

# vd[(j - rs2) * mtilem + i] = td[i, j], j = rs2 .. rs2 + mtilem - 1
mmvcce8.v.m   vd, ts1, rs2
mmvcce16.v.m  vd, ts1, rs2
mmvcce32.v.m  vd, ts1, rs2
mmvcce64.v.m  vd, ts1, rs2

# td[i, j] = vd[(j - rs2) * mtilem + i], j = rs2 .. rs2 + mtilek - 1
mmvace8.m.v   td, vs1, rs2
mmvace16.m.v  td, vs1, rs2
mmvace32.m.v  td, vs1, rs2
mmvace64.m.v  td, vs1, rs2

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

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

```

6.14.4. 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 = mfnvvt_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
}
}
}
}

```

6.15. 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.

6.15.1. 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
0xXXX	URO	moutsh	Fold/unfold output shape.
0xXXX	URO	minsh	Fold/unfold input shape.
0xXXX	URO	mpad	Fold/unfold padding parameters.
0xXXX	URO	mstdi	Fold/unfold sliding strides and dilations.
0xXXX	URO	minsk	Fold/unfold sliding kernel position of input.

Address	Privilege	Name	Description
0xXXX	URO	moutsk	Fold/unfold sliding kernel position of output.
0xXXX	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	tdil_h	Height spacing of the kernel elements
23:16	tdil_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

6.15.2. 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
```

6.15.3. 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**.

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

# for left matrix, a
mlufae8.m   td, (rs1), rs2
mlufae16.m  td, (rs1), rs2
mlufae32.m  td, (rs1), rs2
mlufae64.m  td, (rs1), rs2

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

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

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

```

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

```

6.15.6. 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);

        tr_out = mfmv_s_f(tr_out, -inf)    // move -inf to output reg
        tr_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
                tr_in = mlufce16_m(&in[inh_pos][inw_pos][j]);
                tr_out = mfmax_mm(tr_out, tr_in);
            }
        }

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

```

6.15.7. 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
tr_div = mfmv_s_f(tr_div, kh*kw)
tr_div = mbcce_m (tr_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);

        tr_out = mwsb_mm(tr_out, tr_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
                tr_in = mlufce16_m(&in[inh_pos][inw_pos][j]);
                tr_out = mfadd_mm(tr_out, tr_in);
            }
        }

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

```

6.16. Zmc2i: Col2im Extension

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

6.16.1. CSRs

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

6.16.2. Configuration Instructions

The Zmc2i extension reuses all configuration instructions of Zmi2c.

6.16.3. Store Fold Instructions

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

```
# ts3 destination, rs1 base address, rs2 row byte stride

# for left matrix, a
msfdae8.m    ts3, (rs1), rs2
msfdae16.m   ts3, (rs1), rs2
msfdae32.m   ts3, (rs1), rs2
msfdae64.m   ts3, (rs1), rs2

# for left matrix, b
msfdb8.m     ts3, (rs1), rs2
msfdb16.m    ts3, (rs1), rs2
msfdb32.m    ts3, (rs1), rs2
msfdb64.m    ts3, (rs1), rs2

# for left matrix, c
msfdce8.m    ts3, (rs1), rs2
msfdce16.m   ts3, (rs1), rs2
msfdce32.m   ts3, (rs1), rs2
msfdce64.m   ts3, (rs1), rs2
```

6.17. Zmsp: Matrix Sparsity Extension

The Zmsp extension allows to perform 4:2 sparsing for left or right matrix.

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

```
# Unigned integer sparsing matrix multiplication and add, td = td + ts1 * ts2.
mmau.spa.[dw].mm  td, ts1, ts2, tsp    # left matrix is sparsing
mmau.spb.[dw].mm  td, ts1, ts2, tsp    # right matrix is sparsing
```

```

mmau.spa.[w].mm    td, ts1, ts2, tsp    # left matrix is sparsing
mmau.spb.[w].mm    td, ts1, ts2, tsp    # right matrix is sparsing
mmau.spa.[h].mm    td, ts1, ts2, tsp    # left matrix is sparsing
mmau.spb.[h].mm    td, ts1, ts2, tsp    # right matrix is sparsing
mqmau.spa.[b].mm   td, ts1, ts2, tsp    # left matrix is sparsing
mqmau.spb.[b].mm   td, ts1, ts2, tsp    # right matrix is sparsing
momau.spa.[hb].mm  td, ts1, ts2, tsp    # left matrix is sparsing
momau.spb.[hb].mm  td, ts1, ts2, tsp    # right matrix is sparsing

msmau.spa.[dw].mm  td, ts1, ts2, tsp    # left matrix is sparsing
msmau.spb.[dw].mm  td, ts1, ts2, tsp    # right matrix is sparsing
msmau.spa.[w].mm   td, ts1, ts2, tsp    # left matrix is sparsing
msmau.spb.[w].mm   td, ts1, ts2, tsp    # right matrix is sparsing
msmau.spa.[h].mm   td, ts1, ts2, tsp    # left matrix is sparsing
msmau.spb.[h].mm   td, ts1, ts2, tsp    # right matrix is sparsing
msqmau.spa.[b].mm  td, ts1, ts2, tsp    # left matrix is sparsing
msqmau.spb.[b].mm  td, ts1, ts2, tsp    # right matrix is sparsing
msoma.spa.[hb].mm  td, ts1, ts2, tsp    # left matrix is sparsing
msoma.spb.[hb].mm  td, ts1, ts2, tsp    # right matrix is sparsing

# Signed integer sparsing matrix multiplication and add, td = td + ts1 * ts2.
mma.spa.[dw].mm    td, ts1, ts2, tsp    # left matrix is sparsing
mma.spb.[dw].mm    td, ts1, ts2, tsp    # right matrix is sparsing
mma.spa.[w].mm     td, ts1, ts2, tsp    # left matrix is sparsing
mma.spb.[w].mm     td, ts1, ts2, tsp    # right matrix is sparsing
mma.spa.[h].mm     td, ts1, ts2, tsp    # left matrix is sparsing
mma.spb.[h].mm     td, ts1, ts2, tsp    # right matrix is sparsing
mqma.spa.[b].mm    td, ts1, ts2, tsp    # left matrix is sparsing
mqma.spb.[b].mm    td, ts1, ts2, tsp    # right matrix is sparsing
moma.spa.[hb].mm   td, ts1, ts2, tsp    # left matrix is sparsing
moma.spb.[hb].mm   td, ts1, ts2, tsp    # right matrix is sparsing

msma.spa.[dw].mm   td, ts1, ts2, tsp    # left matrix is sparsing
msma.spb.[dw].mm   td, ts1, ts2, tsp    # right matrix is sparsing
msma.spa.[w].mm    td, ts1, ts2, tsp    # left matrix is sparsing
msma.spb.[w].mm    td, ts1, ts2, tsp    # right matrix is sparsing
msma.spa.[h].mm    td, ts1, ts2, tsp    # left matrix is sparsing
msma.spb.[h].mm    td, ts1, ts2, tsp    # right matrix is sparsing
msqma.spa.[b].mm   td, ts1, ts2, tsp    # left matrix is sparsing
msqma.spb.[b].mm   td, ts1, ts2, tsp    # right matrix is sparsing
msoma.spa.[hb].mm  td, ts1, ts2, tsp    # left matrix is sparsing
msoma.spb.[hb].mm  td, ts1, ts2, tsp    # right matrix is sparsing

# Float point sparsing matrix multiplication and add, td = td + ts1 * ts2.
mfma.spa.[d].mm    td, ts1, ts2, tsp    # left matrix is sparsing
mfma.spb.[d].mm    td, ts1, ts2, tsp    # right matrix is sparsing
mfma.spa.[f].mm    td, ts1, ts2, tsp    # left matrix is sparsing
mfma.spb.[f].mm    td, ts1, ts2, tsp    # right matrix is sparsing
mfma.spa.[hf].mm   td, ts1, ts2, tsp    # left matrix is sparsing

```

mfma.spb.[hf].mm	td, ts1, ts2, tsp	# right matrix is sparsing
mfwma.spa.[f].mm	td, ts1, ts2, tsp	# left matrix is sparsing
mfwma.spb.[f].mm	td, ts1, ts2, tsp	# right matrix is sparsing
mfwma.spa.[hf].mm	td, ts1, ts2, tsp	# left matrix is sparsing
mfwma.spb.[hf].mm	td, ts1, ts2, tsp	# right matrix is sparsing
mfwma.spa.[cf].mm	td, ts1, ts2, tsp	# left matrix is sparsing
mfwma.spb.[cf].mm	td, ts1, ts2, tsp	# right matrix is sparsing
mfqma.spa.[cf].mm	td, ts1, ts2, tsp	# left matrix is sparsing
mfqma.spb.[cf].mm	td, ts1, ts2, tsp	# right matrix is sparsing

Chapter 7. Matrix Instruction Listing

Table 24. Configuration Instructions

Format	63 44				43 39	38 32
	imm[31:12]				funct5	funct7
	31 25	24 20	19 17	16 12	11 7	6 0
	imm[11:5]	rs1	ew	rd	opcode	suffix
msetypei	mtypei[31:12]				00000	0000000
	mtypei[11:0]		100	rd	OP-M64	0111111
msetsew	setval				00001	0000000
	field		100	rd	OP-M64	0111111
msetint	setval				00001	0000000
	field		100	rd	OP-M64	0111111
munsetint	0...000				00001	0000000
	field		100	rd	OP-M64	0111111
msetfp	setval				00001	0000000
	field		100	rd	OP-M64	0111111
munsetfp	0...000				00001	0000000
	field		100	rd	OP-M64	0111111
msetba	setval				00001	0000000
	field		100	rd	OP-M64	0111111
msettype	0...000				00010	0000000
	0000000	rs1	100	rd	OP-M64	0111111
msettilemi	mtilemi[31:12]				00100	0000000
	mtilemi[11:0]		100	rd	OP-M64	0111111
msettilem	0...000				00101	0000000
	0000000	rs1	100	rd	OP-M64	0111111
msettileki	mtileki[31:12]				00110	0000000
	mtileki[11:0]		100	rd	OP-M64	0111111

msettilek	0...000				00111	0000000
	0000000	rs1	100	rd	OP-M64	0111111
msettileni	mtileni[31:12]				01000	0000000
	mtileni[11:0]		100	rd	OP-M64	0111111
msettilen	0...000				01001	0000000
	0000000	rs1	100	rd	OP-M64	0111111
msetoutsh	0...000				10000	0000000
	0000000	rs1	100	rd	OP-M64	0111111
msetinsh	0...000				10001	0000000
	0000000	rs1	100	rd	OP-M64	0111111
msetsk	0...000				10010	0000000
	0000000	rs1	100	rd	OP-M64	0111111
msetpadval	0...000				10011	0000000
	0000000	00000	100	rd	OP-M64	0111111

Table 25. Load/Store Instructions

Format	63 44						43 39	38 32
	resv						funct5	funct7
	31	30	29 25	24 20	19 17	16 12	11 7	6 0
	ls	tr	rs2	rs1	eew	ts3/td	opcode	suffix
mlae*.m	0.000						00001	0000001
	0	0	rs2	rs1	eew	td	OP-M64	0111111
mlbe*.m	0.000						00010	0000001
	0	0	rs2	rs1	eew	td	OP-M64	0111111
mlce*.m	0.000						00000	0000001
	0	0	rs2	rs1	eew	td	OP-M64	0111111
mlre*.m	0.000						00011	0000001
	0	0	rs2	rs1	eew	td	OP-M64	0111111
mlate*.m	0.000						00001	0000001
	0	1	rs2	rs1	eew	td	OP-M64	0111111

mlbte*.m	0.000						00010	0000001
	0	1	rs2	rs1	eev	td	OP-M64	0111111
mlcte*.m	0.000						00000	0000001
	0	1	rs2	rs1	eev	td	OP-M64	0111111
mlrte*.m	0.000						00011	0000001
	0	1	rs2	rs1	eev	td	OP-M64	0111111
msae*.m	0.000						00001	0000001
	1	0	rs2	rs1	eev	ts3	OP-M64	0111111
msbe*.m	0.000						00010	0000001
	1	0	rs2	rs1	eev	ts3	OP-M64	0111111
msce*.m	0.000						00000	0000001
	1	0	rs2	rs1	eev	ts3	OP-M64	0111111
msre*.m	0.000						00011	0000001
	1	0	rs2	rs1	eev	ts3	OP-M64	0111111
msate*.m	0.000						00001	0000001
	1	1	rs2	rs1	eev	ts3	OP-M64	0111111
msbte*.m	0.000						00010	0000001
	1	1	rs2	rs1	eev	ts3	OP-M64	0111111
mscte*.m	0.000						00000	0000001
	1	1	rs2	rs1	eev	ts3	OP-M64	0111111
msrte*.m	0.000						00011	0000001
	1	1	rs2	rs1	eev	ts3	OP-M64	0111111
mlae*.v	0.000						00101	0000001
	0	0	rs2	rs1	eev	vd	OP-M64	0111111
mlbe*.v	0.000						00110	0000001
	0	0	rs2	rs1	eev	vd	OP-M64	0111111
mlce*.v	0.000						00100	0000001
	0	0	rs2	rs1	eev	vd	OP-M64	0111111

msae*.v	0.000						00101	0000001
	1	0	rs2	rs1	ew	vs3	OP-M64	0111111
msbe*.v	0.000						00110	0000001
	1	0	rs2	rs1	ew	vs3	OP-M64	0111111
msce*.v	0.000						00100	0000001
	1	0	rs2	rs1	ew	vs3	OP-M64	0111111
mlufae*.m	0.000						01001	0000001
	0	0	rs2	rs1	ew	td	OP-M64	0111111
mlufbe*.m	0.000						01010	0000001
	0	0	rs2	rs1	ew	td	OP-M64	0111111
mlufce*.m	0.000						01000	0000001
	0	0	rs2	rs1	ew	td	OP-M64	0111111
mlfdae*.m	0.000						01001	0000001
	1	0	rs2	rs1	ew	td	OP-M64	0111111
mlfdbe*.m	0.000						01010	0000001
	1	0	rs2	rs1	ew	td	OP-M64	0111111
mlfdce*.m	0.000						01000	0000001
	1	0	rs2	rs1	ew	td	OP-M64	0111111

Table 26. Data Move Instructions

Format	63 44						43 39	38 32
	resv						funct5	funct7
	31	30	29 25	24 20	19 17	16 12	11 7	6 0
	di	rc	rs2	rs1	ew	rd/td	opcode	suffix
mmve*.x.s	0.000						00000	0000010
	0	0	rs2	ts1	ew	rd	OP-M64	0111111
mmve*.s.x	0.000						00000	0000010
	1	0	rs2	rs1	ew	td	OP-M64	0111111
mfmve*.f.s	0.000						00001	0000010
	0	0	rs2	ts1	ew	rd	OP-M64	0111111

mfmve*.f.x	0.000						00001	0000010
	1	0	rs2	rs1	ew	td	OP-M64	0111111
mbcar.m	0.000						00101	0000010
	0	0	00000	ts1	000	td	OP-M64	0111111
mbcbr.m	0.000						00110	0000010
	0	0	00000	ts1	000	td	OP-M64	0111111
mbccr.m	0.000						00100	0000010
	0	0	00000	ts1	000	td	OP-M64	0111111
mbcace*.m	0.000						00101	0000010
	0	1	00000	ts1	ew	td	OP-M64	0111111
mbcbce*.m	0.000						00110	0000010
	0	1	00000	ts1	ew	td	OP-M64	0111111
mbccce*.m	0.000						00100	0000010
	0	1	00000	ts1	ew	td	OP-M64	0111111
mbcaee*.m	0.000						00101	0000010
	1	0	00000	ts1	ew	td	OP-M64	0111111
mbcbee*.m	0.000						00110	0000010
	1	0	00000	ts1	ew	td	OP-M64	0111111
mbccee*.m	0.000						00100	0000010
	1	0	00000	ts1	ew	td	OP-M64	0111111
mtae*.m	0.000						00101	0000010
	1	1	00000	ts1	ew	td	OP-M64	0111111
mtbe*.m	0.000						00110	0000010
	1	1	00000	ts1	ew	td	OP-M64	0111111
mtce*.m	0.000						00100	0000010
	1	1	00000	ts1	ew	td	OP-M64	0111111
mmvare*.v.m	0.000						01001	0000010
	0	0	rs2	ts1	ew	rd	OP-M64	0111111

mmvbre*.v.m	0.000						01010	0000010
	0	0	rs2	ts1	eew	rd	OP-M64	0111111
mmvcre*.v.m	0.000						01000	0000010
	0	0	rs2	ts1	eew	rd	OP-M64	0111111
mmvare*.m.v	0.000						01001	0000010
	1	0	rs2	ts1	eew	rd	OP-M64	0111111
mmvbre*.m.v	0.000						01010	0000010
	1	0	rs2	ts1	eew	rd	OP-M64	0111111
mmvcre*.m.v	0.000						01000	0000010
	1	0	rs2	ts1	eew	rd	OP-M64	0111111
mmvace*.v.m	0.000						01001	0000010
	0	1	rs2	ts1	eew	rd	OP-M64	0111111
mmvbce*.v.m	0.000						01010	0000010
	0	1	rs2	ts1	eew	rd	OP-M64	0111111
mmvcce*.v.m	0.000						01000	0000010
	0	1	rs2	ts1	eew	rd	OP-M64	0111111
mmvace*.m.v	0.000						01001	0000010
	1	1	rs2	ts1	eew	rd	OP-M64	0111111
mmvbce*.m.v	0.000						01010	0000010
	1	1	rs2	ts1	eew	rd	OP-M64	0111111
mmvcce*.m.v	0.000						01000	0000010
	1	1	rs2	ts1	eew	rd	OP-M64	0111111

Table 27. Matrix Multiplication Instructions

Format	63 57			56 52	51 49	48 46	45 44	43 39	38 32
	resv			tsp	enw	frm	spm	funct5	funct7
	31	30	29 25	24 20	19 17	16 12		11 7	6 0
	fp	sn	ts2	ts1	eew	td		opcode	suffix
mmau.mm	0000000			00000	000	000	00	00000	0000100
	0	0	ts2	ts1	100	td		OP-M64	0111111

mmau.h.mm	0000000			00000	000	000	00	00000	0000100
	0	0	ts2	ts1	001	td		OP-M64	0111111
mmau.w.mm	0000000			00000	000	000	00	00000	0000100
	0	0	ts2	ts1	010	td		OP-M64	0111111
mmau.dw.mm	0000000			00000	000	000	00	00000	0000100
	0	0	ts2	ts1	011	td		OP-M64	0111111
mqmau.mm	0000000			00000	010	000	00	00000	0000100
	0	0	ts2	ts1	100	td		OP-M64	0111111
mqmau.b.mm	0000000			00000	010	000	00	00000	0000100
	0	0	ts2	ts1	000	td		OP-M64	0111111
momau.mm	0000000			00000	011	000	00	00000	0000100
	0	0	ts2	ts1	100	td		OP-M64	0111111
momau.hb.mm	0000000			00000	011	000	00	00000	0000100
	0	0	ts2	ts1	111	td		OP-M64	0111111
msmau.mm	0000000			00000	000	111	00	00000	0000100
	0	0	ts2	ts1	100	td		OP-M64	0111111
msmau.h.mm	0000000			00000	000	111	00	00000	0000100
	0	0	ts2	ts1	001	td		OP-M64	0111111
msmau.w.mm	0000000			00000	000	111	00	00000	0000100
	0	0	ts2	ts1	010	td		OP-M64	0111111
msmau.dw.mm	0000000			00000	000	111	00	00000	0000100
	0	0	ts2	ts1	011	td		OP-M64	0111111
msqmau.mm	0000000			00000	010	111	00	00000	0000100
	0	0	ts2	ts1	100	td		OP-M64	0111111
msqmau.b.mm	0000000			00000	010	111	00	00000	0000100
	0	0	ts2	ts1	000	td		OP-M64	0111111
msomau.mm	0000000			00000	011	111	00	00000	0000100
	0	0	ts2	ts1	100	td		OP-M64	0111111

msomau.hb.mm	0000000			00000	011	111	00	00000	0000100
	0	0	ts2	ts1	111	td		OP-M64	0111111
mma.mm	0000000			00000	000	000	00	00000	0000100
	0	1	ts2	ts1	100	td		OP-M64	0111111
mma.h.mm	0000000			00000	000	000	00	00000	0000100
	0	1	ts2	ts1	001	td		OP-M64	0111111
mma.w.mm	0000000			00000	000	000	00	00000	0000100
	0	1	ts2	ts1	010	td		OP-M64	0111111
mma.dw.mm	0000000			00000	000	000	00	00000	0000100
	0	1	ts2	ts1	011	td		OP-M64	0111111
mqma.mm	0000000			00000	010	000	00	00000	0000100
	0	1	ts2	ts1	100	td		OP-M64	0111111
mqma.b.mm	0000000			00000	010	000	00	00000	0000100
	0	1	ts2	ts1	000	td		OP-M64	0111111
moma.mm	0000000			00000	011	000	00	00000	0000100
	0	1	ts2	ts1	100	td		OP-M64	0111111
moma.hb.mm	0000000			00000	011	000	00	00000	0000100
	0	1	ts2	ts1	111	td		OP-M64	0111111
msma.mm	0000000			00000	000	111	00	00000	0000100
	0	1	ts2	ts1	100	td		OP-M64	0111111
msma.h.mm	0000000			00000	000	111	00	00000	0000100
	0	1	ts2	ts1	001	td		OP-M64	0111111
msma.w.mm	0000000			00000	000	111	00	00000	0000100
	0	1	ts2	ts1	010	td		OP-M64	0111111
msma.dw.mm	0000000			00000	000	111	00	00000	0000100
	0	1	ts2	ts1	011	td		OP-M64	0111111
msqma.mm	0000000			00000	010	111	00	00000	0000100
	0	1	ts2	ts1	100	td		OP-M64	0111111

msqma.b.mm	0000000			00000	010	111	00	00000	0000100
	0	1	ts2	ts1	000	td		OP-M64	0111111
msoma.mm	0000000			00000	011	111	00	00000	0000100
	0	1	ts2	ts1	100	td		OP-M64	0111111
msoma.hb.mm	0000000			00000	011	111	00	00000	0000100
	0	1	ts2	ts1	111	td		OP-M64	0111111
mfma.mm	0000000			00000	000	frm	00	00000	0000100
	1	0	ts2	ts1	100	td		OP-M64	0111111
mfma.hf.mm	0000000			00000	000	frm	00	00000	0000100
	1	0	ts2	ts1	001	td		OP-M64	0111111
mfma.f.mm	0000000			00000	000	frm	00	00000	0000100
	1	0	ts2	ts1	010	td		OP-M64	0111111
mfma.d.mm	0000000			00000	000	frm	00	00000	0000100
	1	0	ts2	ts1	011	td		OP-M64	0111111
mfwma.mm	0000000			00000	001	frm	00	00000	0000100
	1	0	ts2	ts1	100	td		OP-M64	0111111
mfwma.cf.mm	0000000			00000	001	frm	00	00000	0000100
	1	0	ts2	ts1	000	td		OP-M64	0111111
mfwma.hf.mm	0000000			00000	001	frm	00	00000	0000100
	1	0	ts2	ts1	001	td		OP-M64	0111111
mfwma.f.mm	0000000			00000	001	frm	00	00000	0000100
	1	0	ts2	ts1	010	td		OP-M64	0111111
mfqma.mm	0000000			00000	010	frm	00	00000	0000100
	1	0	ts2	ts1	100	td		OP-M64	0111111
mfqma.cf.mm	0000000			00000	010	frm	00	00000	0000100
	1	0	ts2	ts1	000	td		OP-M64	0111111

Table 28. Sparsing Matrix Multiplication Instructions

Format	63 57			56 52	51 49	48 46	45 44	43 39	38 32
	resv			tsp	enw	frm	spm	funct5	funct7
	31	30	29 25	24 20	19 17	16 12		11 7	6 0
	fp	sn	ts2	ts1	eev	td		opcode	suffix
mmau.spa.mm	0000000			tsp	000	000	01	00000	0000100
	0	0	ts2	ts1	100	td		OP-M64	0111111
mmau.spb.mm	0000000			tsp	000	000	10	00000	0000100
	0	0	ts2	ts1	100	td		OP-M64	0111111
mmau.spa.h.m m	0000000			tsp	000	000	01	00000	0000100
	0	0	ts2	ts1	001	td		OP-M64	0111111
mmau.spb.h.m m	0000000			tsp	000	000	10	00000	0000100
	0	0	ts2	ts1	001	td		OP-M64	0111111
mmau.spa.w.m m	0000000			tsp	000	000	01	00000	0000100
	0	0	ts2	ts1	010	td		OP-M64	0111111
mmau.spb.w.m m	0000000			tsp	000	000	10	00000	0000100
	0	0	ts2	ts1	010	td		OP-M64	0111111
mmau.spa.dw. mm	0000000			tsp	000	000	01	00000	0000100
	0	0	ts2	ts1	011	td		OP-M64	0111111
mmau.spb.dw. mm	0000000			tsp	000	000	10	00000	0000100
	0	0	ts2	ts1	011	td		OP-M64	0111111
mqmau.spa.mm	0000000			tsp	010	000	01	00000	0000100
	0	0	ts2	ts1	100	td		OP-M64	0111111
mqmau.spb.mm	0000000			tsp	010	000	10	00000	0000100
	0	0	ts2	ts1	100	td		OP-M64	0111111
mqmau.spa.b.m m	0000000			tsp	010	000	01	00000	0000100
	0	0	ts2	ts1	000	td		OP-M64	0111111
mqmau.spb.b.m m	0000000			tsp	010	000	10	00000	0000100
	0	0	ts2	ts1	000	td		OP-M64	0111111

momau.spa.mm	0000000			tsp	011	000	01	00000	0000100
	0	0	ts2	ts1	100	td		OP-M64	0111111
momau.spb.mm	0000000			tsp	011	000	10	00000	0000100
	0	0	ts2	ts1	100	td		OP-M64	0111111
momau.spa.hb. mm	0000000			tsp	011	000	01	00000	0000100
	0	0	ts2	ts1	111	td		OP-M64	0111111
momau.spb.hb. mm	0000000			tsp	011	000	10	00000	0000100
	0	0	ts2	ts1	111	td		OP-M64	0111111
msmau.spa.mm	0000000			tsp	000	111	01	00000	0000100
	0	0	ts2	ts1	100	td		OP-M64	0111111
msmau.spb.mm	0000000			tsp	000	111	10	00000	0000100
	0	0	ts2	ts1	100	td		OP-M64	0111111
msmau.spa.h.m m	0000000			tsp	000	111	01	00000	0000100
	0	0	ts2	ts1	001	td		OP-M64	0111111
msmau.spb.h.m m	0000000			tsp	000	111	10	00000	0000100
	0	0	ts2	ts1	001	td		OP-M64	0111111
msmau.spa.w.m m	0000000			tsp	000	111	01	00000	0000100
	0	0	ts2	ts1	010	td		OP-M64	0111111
msmau.spb.w.m m	0000000			tsp	000	111	10	00000	0000100
	0	0	ts2	ts1	010	td		OP-M64	0111111
msmau.spa.dw. mm	0000000			tsp	000	111	01	00000	0000100
	0	0	ts2	ts1	011	td		OP-M64	0111111
msmau.spb.dw. mm	0000000			tsp	000	111	10	00000	0000100
	0	0	ts2	ts1	011	td		OP-M64	0111111
msqmau.spa.m m	0000000			tsp	010	111	01	00000	0000100
	0	0	ts2	ts1	100	td		OP-M64	0111111
msqmau.spb.m m	0000000			tsp	010	111	10	00000	0000100
	0	0	ts2	ts1	100	td		OP-M64	0111111

msqmau.spa.b. mm	0000000			tsp	010	111	01	00000	0000100
	0	0	ts2	ts1	000	td		OP-M64	0111111
msqmau.spb.b. mm	0000000			tsp	010	111	10	00000	0000100
	0	0	ts2	ts1	000	td		OP-M64	0111111
msomau.spa.m m	0000000			tsp	011	111	01	00000	0000100
	0	0	ts2	ts1	100	td		OP-M64	0111111
msomau.spb.m m	0000000			tsp	011	111	10	00000	0000100
	0	0	ts2	ts1	100	td		OP-M64	0111111
msomau.spa.hb. mm	0000000			tsp	011	111	01	00000	0000100
	0	0	ts2	ts1	111	td		OP-M64	0111111
msomau.spb.hb. mm	0000000			tsp	011	111	10	00000	0000100
	0	0	ts2	ts1	111	td		OP-M64	0111111
mma.spa.mm	0000000			tsp	000	000	01	00000	0000100
	0	1	ts2	ts1	100	td		OP-M64	0111111
mma.spb.mm	0000000			tsp	000	000	10	00000	0000100
	0	1	ts2	ts1	100	td		OP-M64	0111111
mma.spa.h.mm	0000000			tsp	000	000	01	00000	0000100
	0	1	ts2	ts1	001	td		OP-M64	0111111
mma.spb.h.mm	0000000			tsp	000	000	10	00000	0000100
	0	1	ts2	ts1	001	td		OP-M64	0111111
mma.spa.w.mm	0000000			tsp	000	000	01	00000	0000100
	0	1	ts2	ts1	010	td		OP-M64	0111111
mma.spb.w.mm	0000000			tsp	000	000	10	00000	0000100
	0	1	ts2	ts1	010	td		OP-M64	0111111
mma.spa.dw.m m	0000000			tsp	000	000	01	00000	0000100
	0	1	ts2	ts1	011	td		OP-M64	0111111
mma.spb.dw.m m	0000000			tsp	000	000	10	00000	0000100
	0	1	ts2	ts1	011	td		OP-M64	0111111

mqma.spa.mm	0000000			tsp	010	000	01	00000	0000100
	0	1	ts2	ts1	100	td		OP-M64	0111111
mqma.spb.mm	0000000			tsp	010	000	10	00000	0000100
	0	1	ts2	ts1	100	td		OP-M64	0111111
mqma.spa.b.m m	0000000			tsp	010	000	01	00000	0000100
	0	1	ts2	ts1	000	td		OP-M64	0111111
mqma.spb.b.m m	0000000			tsp	010	000	10	00000	0000100
	0	1	ts2	ts1	000	td		OP-M64	0111111
moma.spa.mm	0000000			tsp	011	000	01	00000	0000100
	0	1	ts2	ts1	100	td		OP-M64	0111111
moma.spb.mm	0000000			tsp	011	000	10	00000	0000100
	0	1	ts2	ts1	100	td		OP-M64	0111111
moma.spa.hb.m m	0000000			tsp	011	000	01	00000	0000100
	0	1	ts2	ts1	111	td		OP-M64	0111111
moma.spb.hb.m m	0000000			tsp	011	000	10	00000	0000100
	0	1	ts2	ts1	111	td		OP-M64	0111111
msma.spa.mm	0000000			tsp	000	111	01	00000	0000100
	0	1	ts2	ts1	100	td		OP-M64	0111111
msma.spb.mm	0000000			tsp	000	111	10	00000	0000100
	0	1	ts2	ts1	100	td		OP-M64	0111111
msma.spa.h.mm	0000000			tsp	000	111	01	00000	0000100
	0	1	ts2	ts1	001	td		OP-M64	0111111
msma.spb.h.m m	0000000			tsp	000	111	10	00000	0000100
	0	1	ts2	ts1	001	td		OP-M64	0111111
msma.spa.w.m m	0000000			tsp	000	111	01	00000	0000100
	0	1	ts2	ts1	010	td		OP-M64	0111111
msma.spb.w.m m	0000000			tsp	000	111	10	00000	0000100
	0	1	ts2	ts1	010	td		OP-M64	0111111

msma.spa.dw.m m	0000000			tsp	000	111	01	00000	0000100
	0	1	ts2	ts1	011	td		OP-M64	0111111
msma.spb.dw.m m	0000000			tsp	000	111	10	00000	0000100
	0	1	ts2	ts1	011	td		OP-M64	0111111
msqma.spa.mm	0000000			tsp	010	111	01	00000	0000100
	0	1	ts2	ts1	100	td		OP-M64	0111111
msqma.spb.mm	0000000			tsp	010	111	10	00000	0000100
	0	1	ts2	ts1	100	td		OP-M64	0111111
msqma.spa.b.m m	0000000			tsp	010	111	01	00000	0000100
	0	1	ts2	ts1	000	td		OP-M64	0111111
msqma.spb.b.m m	0000000			tsp	010	111	10	00000	0000100
	0	1	ts2	ts1	000	td		OP-M64	0111111
msoma.spa.mm	0000000			tsp	011	111	01	00000	0000100
	0	1	ts2	ts1	100	td		OP-M64	0111111
msoma.spb.mm	0000000			tsp	011	111	10	00000	0000100
	0	1	ts2	ts1	100	td		OP-M64	0111111
msoma.spa.hb. mm	0000000			tsp	011	111	01	00000	0000100
	0	1	ts2	ts1	111	td		OP-M64	0111111
msoma.spb.hb. mm	0000000			tsp	011	111	10	00000	0000100
	0	1	ts2	ts1	111	td		OP-M64	0111111
mfma.spa.mm	0000000			tsp	000	frm	01	00000	0000100
	1	0	ts2	ts1	100	td		OP-M64	0111111
mfma.spb.mm	0000000			tsp	000	frm	10	00000	0000100
	1	0	ts2	ts1	100	td		OP-M64	0111111
mfma.spa.hf.m m	0000000			tsp	000	frm	01	00000	0000100
	1	0	ts2	ts1	001	td		OP-M64	0111111
mfma.spb.hf.m m	0000000			tsp	000	frm	10	00000	0000100
	1	0	ts2	ts1	001	td		OP-M64	0111111

mfma.spa.hf.m m	0000000			tsp	000	frm	01	00000	0000100
	1	0	ts2	ts1	001	td		OP-M64	0111111
mfma.spb.hf.m m	0000000			tsp	000	frm	10	00000	0000100
	1	0	ts2	ts1	001	td		OP-M64	0111111
mfma.spa.f.mm	0000000			tsp	000	frm	01	00000	0000100
	1	0	ts2	ts1	010	td		OP-M64	0111111
mfma.spb.f.mm	0000000			tsp	000	frm	10	00000	0000100
	1	0	ts2	ts1	010	td		OP-M64	0111111
mfma.spa.d.mm	0000000			tsp	000	frm	01	00000	0000100
	1	0	ts2	ts1	011	td		OP-M64	0111111
mfma.spb.d.mm	0000000			tsp	000	frm	10	00000	0000100
	1	0	ts2	ts1	011	td		OP-M64	0111111
mfma.spa.f.mm	0000000			tsp	000	frm	01	00000	0000100
	1	0	ts2	ts1	001	td		OP-M64	0111111
mfma.spb.hf.m m	0000000			tsp	000	frm	10	00000	0000100
	1	0	ts2	ts1	001	td		OP-M64	0111111
mfwma.spa.mm	0000000			tsp	001	frm	01	00000	0000100
	1	0	ts2	ts1	100	td		OP-M64	0111111
mfwma.spb.mm	0000000			tsp	001	frm	10	00000	0000100
	1	0	ts2	ts1	100	td		OP-M64	0111111
mfwma.spa.cf. mm	0000000			tsp	001	frm	01	00000	0000100
	1	0	ts2	ts1	000	td		OP-M64	0111111
mfwma.spb.cf. mm	0000000			tsp	001	frm	10	00000	0000100
	1	0	ts2	ts1	000	td		OP-M64	0111111
mfwma.spa.hf. mm	0000000			tsp	001	frm	01	00000	0000100
	1	0	ts2	ts1	001	td		OP-M64	0111111
mfwma.spb.hf. mm	0000000			tsp	001	frm	10	00000	0000100
	1	0	ts2	ts1	001	td		OP-M64	0111111

mfwma.spa.f.m m	0000000			tsp	001	frm	01	00000	0000100
	1	0	ts2	ts1	010	td		OP-M64	0111111
mfwma.spb.f.m m	0000000			tsp	001	frm	10	00000	0000100
	1	0	ts2	ts1	010	td		OP-M64	0111111
mfqma.spa.mm	0000000			tsp	010	frm	01	00000	0000100
	1	0	ts2	ts1	100	td		OP-M64	0111111
mfqma.spb.mm	0000000			tsp	010	frm	10	00000	0000100
	1	0	ts2	ts1	100	td		OP-M64	0111111
mfqma.spa.cf.m m	0000000			tsp	010	frm	01	00000	0000100
	1	0	ts2	ts1	000	td		OP-M64	0111111
mfqma.spb.cf.m m	0000000			tsp	010	frm	10	00000	0000100
	1	0	ts2	ts1	000	td		OP-M64	0111111

Table 29. Element-wise Arithmetic & Logic Instructions

Format	63 52				51 49	48 46	45 44	43 39	38 32
	imm[16:5]				enw	frm	opm	funct5	funct7
	31	30	29 25	24 20	19 17	16 12		11 7	6 0
	fp	sn	ts2	ts1	eew	td		opcode	suffix
maddu.*.mm	0.000				000	000	00	00000	0000101
	0	0	ts2	ts1	eew	td		OP-M64	0111111
maddu.*.mi	imm[16:5]				000	000	11	00000	0000101
	0	0	imm[4:0]	ts1	eew	td		OP-M64	0111111
msaddu.*.m m	0.000				000	111	00	00000	0000101
	0	0	ts2	ts1	eew	td		OP-M64	0111111
msaddu.*.mi	imm[16:5]				000	111	11	00000	0000101
	0	0	imm[4:0]	ts1	eew	td		OP-M64	0111111
mwaddu.*.m m	0.000				001	000	00	00000	0000101
	0	0	ts2	ts1	eew	td		OP-M64	0111111
mwaddu.*.m i	imm[16:5]				001	000	11	00000	0000101
	0	0	imm[4:0]	ts1	eew	td		OP-M64	0111111

madd.*.mm	0.000				000	000	00	00000	0000101
	0	1	ts2	ts1	eeew	td		OP-M64	0111111
madd.*.mi	imm[16:5]				000	000	11	00000	0000101
	0	1	imm[4:0]	ts1	eeew	td		OP-M64	0111111
msadd.*.mm	0.000				000	111	00	00000	0000101
	0	1	ts2	ts1	eeew	td		OP-M64	0111111
msadd.*.mi	imm[16:5]				000	111	11	00000	0000101
	0	1	imm[4:0]	ts1	eeew	td		OP-M64	0111111
mwadd.*.m m	0.000				001	000	00	00000	0000101
	0	1	ts2	ts1	eeew	td		OP-M64	0111111
mwadd.*.mi	imm[16:5]				001	000	11	00000	0000101
	0	1	imm[4:0]	ts1	eeew	td		OP-M64	0111111
msubu.*.mm	0.000				000	000	00	00001	0000101
	0	0	ts2	ts1	eeew	td		OP-M64	0111111
msubu.*.mi	imm[16:5]				000	000	11	00001	0000101
	0	0	imm[4:0]	ts1	eeew	td		OP-M64	0111111
mssubu.*.m m	0.000				000	111	00	00001	0000101
	0	0	ts2	ts1	eeew	td		OP-M64	0111111
mssubu.*.mi	imm[16:5]				000	111	11	00001	0000101
	0	0	imm[4:0]	ts1	eeew	td		OP-M64	0111111
mwsubu.*.m m	0.000				001	000	00	00001	0000101
	0	0	ts2	ts1	eeew	td		OP-M64	0111111
mwsubu.*.mi	imm[16:5]				001	000	11	00001	0000101
	0	0	imm[4:0]	ts1	eeew	td		OP-M64	0111111
msub.*.mm	0.000				000	000	00	00001	0000101
	0	1	ts2	ts1	eeew	td		OP-M64	0111111
msub.*.mi	imm[16:5]				000	000	11	00001	0000101
	0	1	imm[4:0]	ts1	eeew	td		OP-M64	0111111

mssub.*.mm	0.000				000	111	00	00001	0000101
	0	1	ts2	ts1	eew	td		OP-M64	0111111
mssub.*.mi	imm[16:5]				000	111	11	00001	0000101
	0	1	imm[4:0]	ts1	eew	td		OP-M64	0111111
mwsb.*.mm	0.000				001	000	00	00001	0000101
	0	1	ts2	ts1	eew	td		OP-M64	0111111
mwsb.*.mi	imm[16:5]				001	000	11	00001	0000101
	0	1	imm[4:0]	ts1	eew	td		OP-M64	0111111
mrsub.*.mi	imm[16:5]				000	000	11	00010	0000101
	0	0	imm[4:0]	ts1	eew	td		OP-M64	0111111
msrsub.*.mi	imm[16:5]				000	111	11	00010	0000101
	0	0	imm[4:0]	ts1	eew	td		OP-M64	0111111
mwrsub.*.mi	imm[16:5]				001	000	11	00010	0000101
	0	0	imm[4:0]	ts1	eew	td		OP-M64	0111111
mrsub.*.mi	imm[16:5]				000	000	11	00010	0000101
	0	1	imm[4:0]	ts1	eew	td		OP-M64	0111111
msrsub.*.mi	imm[16:5]				000	111	11	00010	0000101
	0	1	imm[4:0]	ts1	eew	td		OP-M64	0111111
mwrsub.*.mi	imm[16:5]				001	000	11	00010	0000101
	0	1	imm[4:0]	ts1	eew	td		OP-M64	0111111
mminu.*.mm	0.000				000	000	00	00100	0000101
	0	0	ts2	ts1	eew	td		OP-M64	0111111
mmin.*.mm	0.000				000	000	00	00100	0000101
	0	1	ts2	ts1	eew	td		OP-M64	0111111
mmaxu.*.m	0.000				000	000	00	00101	0000101
	0	0	ts2	ts1	eew	td		OP-M64	0111111
mmax.*.mm	0.000				000	000	00	00101	0000101
	0	1	ts2	ts1	eew	td		OP-M64	0111111

mand.mm	0.000				000	000	00	00110	0000101
	0	0	ts2	ts1	000	td		OP-M64	0111111
mor.mm	0.000				000	000	00	00111	0000101
	0	0	ts2	ts1	000	td		OP-M64	0111111
mxor.mm	0.000				000	000	00	00011	0000101
	0	0	ts2	ts1	000	td		OP-M64	0111111
msll.*.mm	0.000				000	000	00	01000	0000101
	0	0	ts2	ts1	eeew	td		OP-M64	0111111
msll.*.mi	imm[16:5]				000	000	11	01000	0000101
	0	0	imm[4:0]	ts1	eeew	td		OP-M64	0111111
msrl.*.mm	0.000				000	000	00	01001	0000101
	0	0	ts2	ts1	eeew	td		OP-M64	0111111
msrl.*.mi	imm[16:5]				000	000	11	01001	0000101
	0	0	imm[4:0]	ts1	eeew	td		OP-M64	0111111
msra.*.mm	0.000				000	000	00	01001	0000101
	0	1	ts2	ts1	eeew	td		OP-M64	0111111
msra.*.mi	imm[16:5]				000	000	11	01001	0000101
	0	1	imm[4:0]	ts1	eeew	td		OP-M64	0111111
mmul.*.mm	0.000				000	000	00	01010	0000101
	0	1	ts2	ts1	eeew	td		OP-M64	0111111
mmul.*.mi	imm[16:5]				000	000	11	01010	0000101
	0	1	imm[4:0]	ts1	eeew	td		OP-M64	0111111
msmul.*.mm	0.000				000	111	00	01010	0000101
	0	1	ts2	ts1	eeew	td		OP-M64	0111111
msmul.*.mi	imm[16:5]				000	111	11	01010	0000101
	0	1	imm[4:0]	ts1	eeew	td		OP-M64	0111111
mwmul.*.m m	0.000				001	000	00	01010	0000101
	0	1	ts2	ts1	eeew	td		OP-M64	0111111

mwmul.*.mi	imm[16:5]				001	000	11	01010	0000101
	0	1	imm[4:0]	ts1	eeew	td		OP-M64	0111111
msmulu.*.m m	0.000				000	111	00	01010	0000101
	0	0	ts2	ts1	eeew	td		OP-M64	0111111
msmulu.*.mi	imm[16:5]				000	111	11	01010	0000101
	0	0	imm[4:0]	ts1	eeew	td		OP-M64	0111111
mwmulu.*.m m	0.000				001	000	00	01010	0000101
	0	0	ts2	ts1	eeew	td		OP-M64	0111111
mwmulu.*.m i	imm[16:5]				001	000	11	01010	0000101
	0	0	imm[4:0]	ts1	eeew	td		OP-M64	0111111
mmulh.*.mm	0.000				000	000	00	01011	0000101
	0	1	ts2	ts1	eeew	td		OP-M64	0111111
mmulh.*.mi	imm[16:5]				000	000	11	01011	0000101
	0	1	imm[4:0]	ts1	eeew	td		OP-M64	0111111
mmulhu.*.m m	0.000				000	000	00	01011	0000101
	0	0	ts2	ts1	eeew	td		OP-M64	0111111
mmulhu.*.mi	imm[16:5]				000	000	11	01011	0000101
	0	0	imm[4:0]	ts1	eeew	td		OP-M64	0111111
mmulhsu.*. mm	0.000				000	000	00	01100	0000101
	0	1	ts2	ts1	eeew	td		OP-M64	0111111
mmulhsu.*. mi	imm[16:5]				000	000	11	01100	0000101
	0	1	imm[4:0]	ts1	eeew	td		OP-M64	0111111
msmulu.*.m m	0.000				000	111	00	01100	0000101
	0	1	ts2	ts1	eeew	td		OP-M64	0111111
msmulu.*.m i	imm[16:5]				000	111	11	01100	0000101
	0	1	imm[4:0]	ts1	eeew	td		OP-M64	0111111
mwmulsu.*. mm	0.000				001	000	00	01100	0000101
	0	1	ts2	ts1	eeew	td		OP-M64	0111111

mwmulsu.*. mi	imm[16:5]				001	000	11	01100	0000101
	0	1	imm[4:0]	ts1	eew	td		OP-M64	0111111
mfadd.*.mm	0.000				000	frm	00	00000	0000101
	1	0	ts2	ts1	eew	td		OP-M64	0111111
mfwadd.*.m m	0.000				001	frm	00	00000	0000101
	1	0	ts2	ts1	eew	td		OP-M64	0111111
mfsub.*.mm	0.000				000	frm	00	00001	0000101
	1	0	ts2	ts1	eew	td		OP-M64	0111111
mfwsb.*.m m	0.000				001	frm	00	00001	0000101
	1	0	ts2	ts1	eew	td		OP-M64	0111111
mfmin.*.mm	0.000				000	000	00	00100	0000101
	1	0	ts2	ts1	eew	td		OP-M64	0111111
mfmax.*.mm	0.000				000	000	00	00101	0000101
	1	0	ts2	ts1	eew	td		OP-M64	0111111
mfmul.*.mm	0.000				000	frm	00	01010	0000101
	1	0	ts2	ts1	eew	td		OP-M64	0111111
mfwmul.*.m m	0.000				001	frm	00	01010	0000101
	1	0	ts2	ts1	eew	td		OP-M64	0111111
mfdiv.*.mm	0.000				000	frm	00	01011	0000101
	1	0	ts2	ts1	eew	td		OP-M64	0111111
mfsqrt.*.m	0.000				000	frm	01	01100	0000101
	1	0	00000	ts1	eew	td		OP-M64	0111111

Table 30. Type Convert Instructions

Format	63 52				51 49	48 46	45 44	43 39	38 32
	resv				enw	frm	lmul	funct5	funct7
	31	30	29 25	24 20	19 17	16 12		11 7	6 0
	fd	sn	zero	ts1	eew	td		opcode	suffix
mwcvtu.b.hb.m	0.000				001	000	00	00000	0000110
	0	0	00000	ts1	111	td		OP-M64	0111111

mwcvb.b.hb.m	0.000				001	000	00	00000	0000110
	0	1	00000	ts1	111	td		OP-M64	0111111
mncvtu.hb.b.m	0.000				111	000	lmul	00000	0000110
	0	0	00000	ts1	000	td		OP-M64	0111111
mncvt.hb.b.m	0.000				111	000	lmul	00000	0000110
	0	1	00000	ts1	000	td		OP-M64	0111111
msncvtu.hb.b.m	0.000				111	111	lmul	00000	0000110
	0	0	00000	ts1	000	td		OP-M64	0111111
msncvt.hb.b.m	0.000				111	111	lmul	00000	0000110
	0	1	00000	ts1	000	td		OP-M64	0111111
mfcvb.bf.hf.m	0.000				000	frm	00	00000	0000110
	1	0	00000	ts1	100	td		OP-M64	0111111
mfcvb.hf.bf.m	0.000				000	frm	00	00000	0000110
	1	1	00000	ts1	100	td		OP-M64	0111111
mfwcvt.fw.f.m	0.000				001	frm	00	00001	0000110
	1	0	00000	ts1	100	td		OP-M64	0111111
mfwcvt.hf.cf.m	0.000				001	frm	00	00001	0000110
	1	0	00000	ts1	000	td		OP-M64	0111111
mfwcvt.f.hf.m	0.000				001	frm	00	00001	0000110
	1	0	00000	ts1	001	td		OP-M64	0111111
mfwcvt.d.f.m	0.000				001	frm	00	00001	0000110
	1	0	00000	ts1	010	td		OP-M64	0111111
mfncvt.f.fw.m	0.000				111	frm	lmul	00001	0000110
	1	0	00000	ts1	100	td		OP-M64	0111111
mfncvt.cf.hf.m	0.000				111	frm	lmul	00001	0000110
	1	0	00000	ts1	001	td		OP-M64	0111111
mfncvt.hf.f.m	0.000				111	frm	lmul	00001	0000110
	1	0	00000	ts1	010	td		OP-M64	0111111

mfncvt.f.d.m	0.000				111	frm	lmul	00001	0000110
	1	0	00000	ts1	011	td		OP-M64	0111111
mfcvtu.x.f.m	0.000				000	frm	00	00010	0000110
	0	0	00000	ts1	100	td		OP-M64	0111111
mfcvtu.h.hf.m	0.000				000	frm	00	00010	0000110
	0	0	00000	ts1	001	td		OP-M64	0111111
mfcvtu.w.f.m	0.000				000	frm	00	00010	0000110
	0	0	00000	ts1	010	td		OP-M64	0111111
mfcvtu.dw.d.m	0.000				000	frm	00	00010	0000110
	0	0	00000	ts1	011	td		OP-M64	0111111
mfwcvtu.xw.f.m	0.000				001	frm	00	00010	0000110
	0	0	00000	ts1	100	td		OP-M64	0111111
mfwcvtu.w.hf.m	0.000				001	frm	00	00010	0000110
	0	0	00000	ts1	001	td		OP-M64	0111111
mfwcvtu.dw.f.m	0.000				001	frm	00	00010	0000110
	0	0	00000	ts1	010	td		OP-M64	0111111
mfncvtu.x.fw.m	0.000				111	frm	00	00010	0000110
	0	0	00000	ts1	100	td		OP-M64	0111111
mfncvtu.b.hf.m	0.000				111	frm	00	00010	0000110
	0	0	00000	ts1	001	td		OP-M64	0111111
mfncvtu.h.f.m	0.000				111	frm	00	00010	0000110
	0	0	00000	ts1	010	td		OP-M64	0111111
mfncvtu.w.d.m	0.000				111	frm	00	00010	0000110
	0	0	00000	ts1	011	td		OP-M64	0111111
mfncvtu.x.fq.m	0.000				110	frm	00	00010	0000110
	0	0	00000	ts1	100	td		OP-M64	0111111
mfncvtu.b.f.m	0.000				110	frm	00	00010	0000110
	0	0	00000	ts1	010	td		OP-M64	0111111

mfcvt.x.f.m	0.000				000	frm	00	00010	0000110
	0	1	00000	ts1	100	td		OP-M64	0111111
mfcvt.h.hf.m	0.000				000	frm	00	00010	0000110
	0	1	00000	ts1	001	td		OP-M64	0111111
mfcvt.w.f.m	0.000				000	frm	00	00010	0000110
	0	1	00000	ts1	010	td		OP-M64	0111111
mfcvt.dw.d.m	0.000				000	frm	00	00010	0000110
	0	1	00000	ts1	011	td		OP-M64	0111111
mfwcvt.xw.f.m	0.000				001	frm	00	00010	0000110
	0	1	00000	ts1	100	td		OP-M64	0111111
mfwcvt.w.hf.m	0.000				001	frm	00	00010	0000110
	0	1	00000	ts1	001	td		OP-M64	0111111
mfwcvt.dw.f.m	0.000				001	frm	00	00010	0000110
	0	1	00000	ts1	010	td		OP-M64	0111111
mfncvt.x.fw.m	0.000				111	frm	00	00010	0000110
	0	1	00000	ts1	100	td		OP-M64	0111111
mfncvt.b.hf.m	0.000				111	frm	00	00010	0000110
	0	1	00000	ts1	001	td		OP-M64	0111111
mfncvt.h.f.m	0.000				111	frm	00	00010	0000110
	0	1	00000	ts1	010	td		OP-M64	0111111
mfncvt.w.d.m	0.000				111	frm	00	00010	0000110
	0	1	00000	ts1	011	td		OP-M64	0111111
mfncvt.x.fq.m	0.000				110	frm	00	00010	0000110
	0	1	00000	ts1	100	td		OP-M64	0111111
mfncvt.b.f.m	0.000				110	frm	00	00010	0000110
	0	1	00000	ts1	010	td		OP-M64	0111111
mfcvtu.f.x.m	0.000				000	frm	00	00010	0000110
	1	0	00000	ts1	100	td		OP-M64	0111111

mfcvtu.hf.h.m	0.000				000	frm	00	00010	0000110
	1	0	00000	ts1	001	td		OP-M64	0111111
mfcvtu.f.w.m	0.000				000	frm	00	00010	0000110
	1	0	00000	ts1	010	td		OP-M64	0111111
mfcvtu.d.dw.m	0.000				000	frm	00	00010	0000110
	1	0	00000	ts1	011	td		OP-M64	0111111
mfwcvtu.fw.x.m	0.000				001	frm	00	00010	0000110
	1	0	00000	ts1	100	td		OP-M64	0111111
mfwcvtu.hf.b.m	0.000				001	frm	00	00010	0000110
	1	0	00000	ts1	000	td		OP-M64	0111111
mfwcvtu.f.h.m	0.000				001	frm	00	00010	0000110
	1	0	00000	ts1	001	td		OP-M64	0111111
mfwcvtu.d.w.m	0.000				001	frm	00	00010	0000110
	1	0	00000	ts1	010	td		OP-M64	0111111
mfwcvtu.fq.x.m	0.000				010	frm	00	00010	0000110
	1	0	00000	ts1	100	td		OP-M64	0111111
mfwcvtu.f.b.m	0.000				010	frm	00	00010	0000110
	1	0	00000	ts1	000	td		OP-M64	0111111
mfncvtu.f.xw.m	0.000				111	frm	lmul	00010	0000110
	1	0	00000	ts1	100	td		OP-M64	0111111
mfncvtu.hf.w.m	0.000				111	frm	lmul	00010	0000110
	1	0	00000	ts1	010	td		OP-M64	0111111
mfncvtu.f.dw.m	0.000				111	frm	lmul	00010	0000110
	1	0	00000	ts1	011	td		OP-M64	0111111
mfcvtt.f.x.m	0.000				000	frm	00	00010	0000110
	1	1	00000	ts1	100	td		OP-M64	0111111
mfcvtt.hf.h.m	0.000				000	frm	00	00010	0000110
	1	1	00000	ts1	001	td		OP-M64	0111111

mfcvt.f.w.m	0.000				000	frm	00	00010	0000110
	1	1	00000	ts1	010	td		OP-M64	0111111
mfcvt.d.dw.m	0.000				000	frm	00	00010	0000110
	1	1	00000	ts1	011	td		OP-M64	0111111
mfwcvt.fw.x.m	0.000				001	frm	00	00010	0000110
	1	1	00000	ts1	100	td		OP-M64	0111111
mfwcvt.hf.b.m	0.000				001	frm	00	00010	0000110
	1	1	00000	ts1	000	td		OP-M64	0111111
mfwcvt.f.h.m	0.000				001	frm	00	00010	0000110
	1	1	00000	ts1	001	td		OP-M64	0111111
mfwcvt.d.w.m	0.000				001	frm	00	00010	0000110
	1	1	00000	ts1	010	td		OP-M64	0111111
mfwcvt.fq.x.m	0.000				010	frm	00	00010	0000110
	1	1	00000	ts1	100	td		OP-M64	0111111
mfwcvt.f.b.m	0.000				010	frm	00	00010	0000110
	1	1	00000	ts1	000	td		OP-M64	0111111
mfncvt.f.xw.m	0.000				111	frm	lmul	00010	0000110
	1	1	00000	ts1	100	td		OP-M64	0111111
mfncvt.hf.w.m	0.000				111	frm	lmul	00010	0000110
	1	1	00000	ts1	010	td		OP-M64	0111111
mfncvt.f.dw.m	0.000				111	frm	lmul	00010	0000110
	1	1	00000	ts1	011	td		OP-M64	0111111