RISC-V Matrix Specification

Xin Ouyang, Zhiqiang Liu

Version 0.2, 9/2022: This document is in development.

Table of Contents

Preamble

This document is in the Development state

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

Copyright and license information

Contributors

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

- Kening Zhang
- Kun Hu
- Hui Yao
- **Your name here**

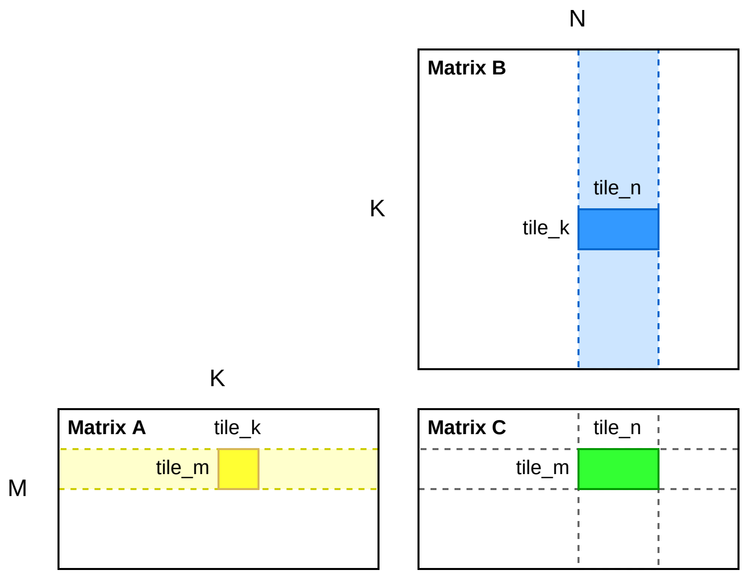
We will be very grateful to the huge number of other people who will have helped to improve this specification through their comments, reviews, feedback and questions.

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 tile_m × tile_n, the tile size of A is tile_m × tile_k and the tile size of B is tile_k × tile_n.



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, $\mathrm{ELEN} \geq 8$, which must be a power of 2.
2. The number of bits in a single matrix tile register, 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, which must be a power of 2, and must be no greater than 2^{16}.
4. ELEN < RLEN < MLEN, this supports matrix tile size from 2×2 to $2^{16} \times 2^{16}$

Chapter 3. Programmer’s Model

The matrix extension adds 10 matrix tile registers, and 5 unprivileged CSRs (tile_m, tile_n, tile_k, mtype, mlenb) to the base scalar RISC-V ISA.

Table 1. New matrix CSRs

Address	Privilege	Name	Description
0xC40	URO	tile_m	Tile length in m direction
0xC41	URO	tile_n	Tile length in n direction
0xC42	URO	tile_k	Tile length in k direction
0xC43	URO	mtype	Matrix tile data type register
0xC44	URO	mlenb	MLEN/8 (matrix tile register length in bytes)

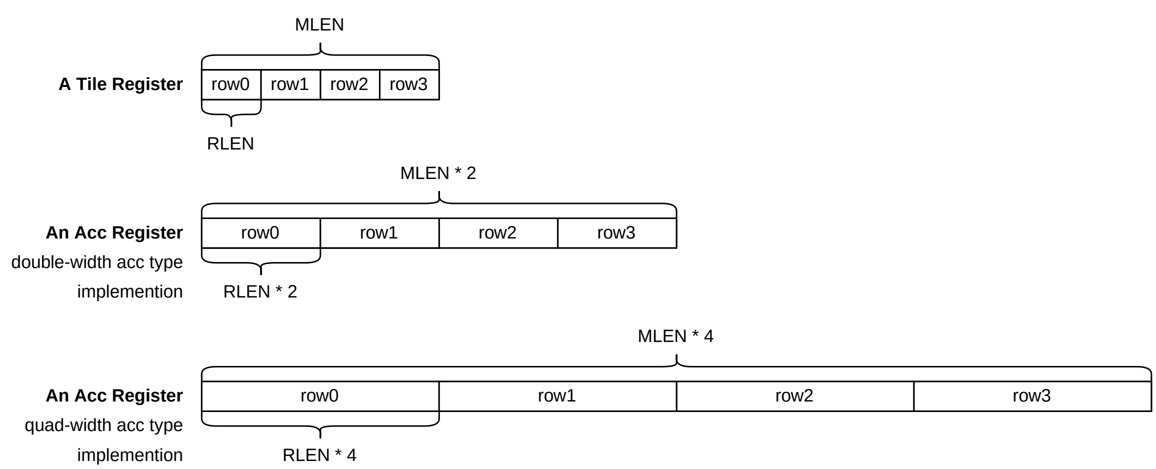
3.1. Matrix Registers

The matrix extension adds 10 architectural matrix registers, 8 **Tile Registers**(tr0-tr7) for tiles of input matrices, and 2 **Accumulator Registers**(acc0-acc1) for tiles of output matrices.

A **Tile Register** has a fixed MLEN bits of state.

If a implementation supports using a accumulator data type which is quad-width of the input data type, each **Accumulator Register** would have a MLEN*4 bits of state.

Otherwise, each **Accumulator Register** has a MLEN*2 bits of state, to support a double-width accumulator data type.



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}` instructions. The matrix type determines the organization of elements in each matrix register.

> Allowing updates only via the `msettype{i}` instructions simplifies maintenance of the `mtype` register state.

The `mtype` register has three fields, `mill`, `maccq`, and `msew[2:0]`. Bits `mtype[XLEN-2:4]` should be written with zero, and non-zero values in this field are reserved.

Table 2. `mtype` register layout

Bits	Name	Description
XLEN-1	mill	Illegal value if set
XLEN-2:4	0	Reserved if non-zero
3	maccq	Support quad-width accumulator element
2:0	msew[2:0]	Selected element width (SEW) setting

3.3. Matrix tile configure registers, tile_m/tile_k/tile_n

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

Chapter 4. Instructions

4.1. Instruction Formats

The instructions in the matrix extension use a new major opcode (1110111, which inst[6:5]=11, inst[4:2]=101 is reserved in RISC-V opcode map).

This instruction formats are list below.

Configuration instructions, funct3 = 111

31 28	27 20	19 15	14 12	11 7	6 0
funct4	imm13		funct3	rd	1110111
funct4	00000000	rs1	funct3	rd	1110111

Load & Store instructions, eew = 000 - 011

31 26	25	24 20	19 15	14 12	11 7	6 0
funct6	ls	rs2	rs1	eew	md	1110111

Arithmetic & Type-Convert instructions, funct3 = 110

31 26	25	24 20	19 15	14 12	11 7	6 0
funct6	fp	ms2	ms1	funct3	md	1110111

Data Move instructions, funct3 = 101

31 26	25	24 20	19 15	14 12	11 7	6 0
funct6	di	rs2	ms1	funct3	md	1110111

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 `tile_m/tile_k/tile_n`. Based on microarchitecture implementation and mtype setting, hardware returns a new `tile_m/tile_k/tile_n` value via a general purpose register (usually smaller), also stored in `tile_m/tile_k/tile_n` 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 tile_* 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, mtypei                # rd = new mtype, mtypei = new mtype setting
msettype rd, rs1                    # rd = new mtype, rs1 = new mtype value
```

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

```
msettilemi rd, mleni                # rd = new tile_m, mleni = ATM
msettilem rd, rs1                   # rd = new tile_m, rs1 = ATM
msettileki rd, mleni                # rd = new tile_k, mleni = ATN
msettilek rd, rs1                   # rd = new tile_k, rs1 = ATN
msettileni rd, mleni                # rd = new tile_n, mleni = ATK
msettilen rd, rs1                   # rd = new tile_n, rs1 = ATK
```

4.2.1. `mtype` encoding

The `mtype` register has three fields, `mill`, `maccq`, and `msew[2:0]`. Bits `mtype[XLEN-2:4]` should be written with zero, and non-zero values in this field are reserved.

Table 3. `mtype` register layout

Bits	Name	Description
XLEN-1	mill	Illegal value if set
XLEN-2:4	0	Reserved if non-zero
3	maccq	Support quad-width accumulator element
2:0	msew[2:0]	Selected element width (SEW) setting

The new `mtype` value is encoded in the immediate fields of msettypei, and in the rs1 register for msettype.

```
Suggested assembler names used for msettypei mtypei immediate

    e8    # SEW = 8b
    e16   # SEW = 16b
    e32   # SEW = 32b
    e64   # SEW = 64b

    accq  # support 32-bit accumulator element

Examples:

    msettypei t0, e8                # SEW = 8
    msettypei t0, e32               # SEW = 32

    msettypei t0, e8, accq          # SEW = 8, support 32-bit accumulator element
```

4.2.2. `ATM/ATK/ATN` encoding

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

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

For examples, if MLEN=256, RLEN=64, `TMMAX`, `TKMAX`, `TNMAX` values are shown below.

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

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

rd	rs1	`ATM/ATK/ATN` value	Effect on `tile_m/tile_k/tile_n`
-	!x0	Value in `x[rs1]`	Normal tiling
!x0	x0	~0	Set `tile_m/tile_k/tile_n` to `TMMAX/TKMAX/TNMAX`

x0	x0	Value in `tile_m/tile_k/tile_n`	Keep existing `tile_m/tile_k/tile_n` if less than `TMMAX/TKMAX/TNMAX`

For the `msettile[m|k|n]i` instructions, the `ATM/ATK/ATN` is encoded as a 13-bit zero-extended immediate in the rs1.

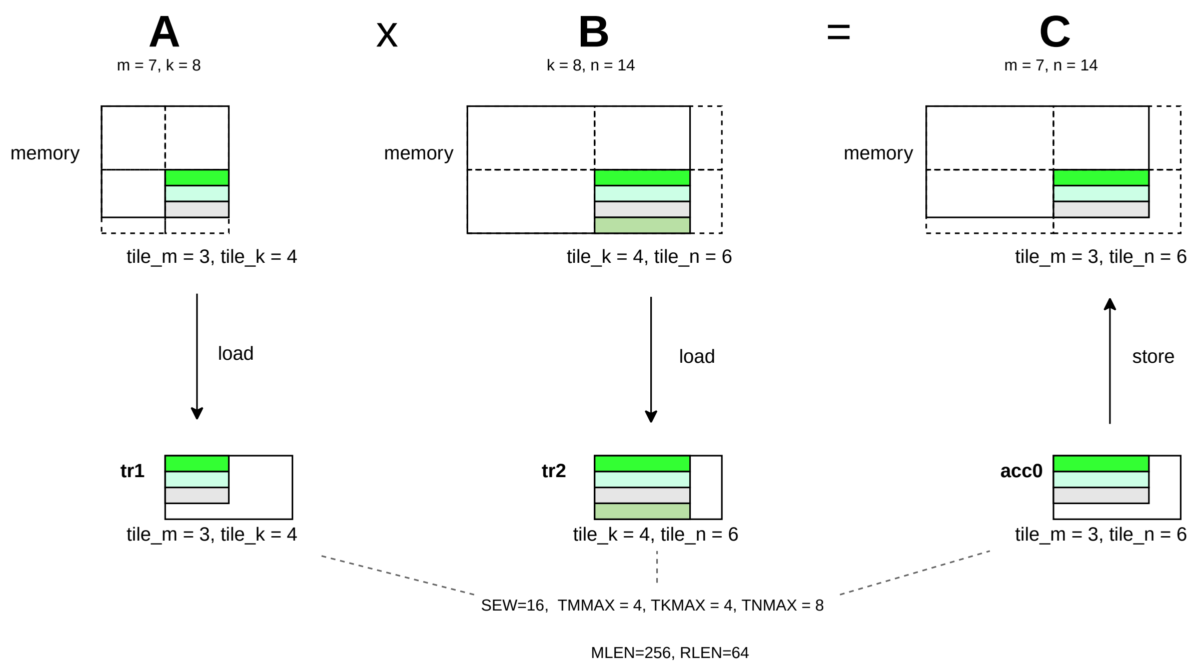
4.2.3. Constraints on Setting `tile_m/tile_k/tile_n`

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

1. `tile_m = ATM` if `ATM <= TMMAX`
2. `ceil(ATM / 2) <= tile_m <= TMMAX` if `ATM < (2 * TMMAX)`
3. `tile_m = 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. `tile_m = 0` if `ATM = 0`
    b. `tile_m > 0` if `ATM > 0`
    c. `tile_m <= TMMAX`
    d. `tile_m <= ATM`
    e. a value read from `tile_m` when used as the ATM argument to `msettile[m|k|n]{i}` results in the same value in `tile_m`, provided the resultant TMMAX equals the value of TMMAX at the time that `tile_m` was read.

Continue to use MLEN=256, RLEN=64 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 `tile_m/tile_k/tile_n` 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.

For `mla*`, `msa*`, `mlb*` or `msb*` instructions, source and destination operands are **tile registers**(`tr0`-`tr7`). And for `mlc*`, `msc*` instructions, source and destination operands are **accumulator registers**(`acc0`-`acc1`)

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

# for left matrix, a
# tile size = tile_m * tile_k
mlae8.m  md, (rs1), rs2 #  8-bit tile load
mlae16.m md, (rs1), rs2 # 16-bit tile load
mlae32.m md, (rs1), rs2 # 32-bit tile load
mlae64.m md, (rs1), rs2 # 64-bit tile load

# for right matrix, b
# tile size = tile_k * tile_n
mlbe8.m  md, (rs1), rs2 #  8-bit tile load
mlbe16.m md, (rs1), rs2 # 16-bit tile load
mlbe32.m md, (rs1), rs2 # 32-bit tile load
mlbe64.m md, (rs1), rs2 # 64-bit tile load
```

```
# for output matrix, c
# tile size = tile_m * tile_n
mlce8.m  md, (rs1), rs2 #  8-bit acc load
mlce16.m md, (rs1), rs2 # 16-bit acc load
mlce32.m md, (rs1), rs2 # 32-bit acc load
mlce64.m md, (rs1), rs2 # 64-bit acc load
```

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

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

# for left matrix, a
# tile size = tile_k * tile_m
mlate8.m  md, (rs1), rs2 #  8-bit tile load
mlate16.m md, (rs1), rs2 # 16-bit tile load
mlate32.m md, (rs1), rs2 # 32-bit tile load
mlate64.m md, (rs1), rs2 # 64-bit tile load

# for right matrix, b
# tile size = tile_n * tile_k
mlbte8.m  md, (rs1), rs2 #  8-bit tile load
mlbte16.m md, (rs1), rs2 # 16-bit tile load
mlbte32.m md, (rs1), rs2 # 32-bit tile load
mlbte64.m md, (rs1), rs2 # 64-bit tile load

# for output matrix, c
# tile size = tile_n * tile_m
mlcte8.m  md, (rs1), rs2 #  8-bit acc load
mlcte16.m md, (rs1), rs2 # 16-bit acc load
mlcte32.m md, (rs1), rs2 # 32-bit acc load
mlcte64.m md, (rs1), rs2 # 64-bit acc load
```

4.3.2. Store instructions

Store a matrix tile to memory.

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

# for left matrix, a
# tile size = tile_m * tile_k
msae8.m  ms3, (rs1), rs2 #  8-bit tile store
msae16.m ms3, (rs1), rs2 # 16-bit tile store
msae32.m ms3, (rs1), rs2 # 32-bit tile store
msae64.m ms3, (rs1), rs2 # 64-bit tile store

# for right matrix, b
```

```
# tile size = tile_k * tile_n
msbe8.m  ms3, (rs1), rs2 #  8-bit tile store
msbe16.m ms3, (rs1), rs2 # 16-bit tile store
msbe32.m ms3, (rs1), rs2 # 32-bit tile store
msbe64.m ms3, (rs1), rs2 # 64-bit tile store

# for output matrix, c
# tile size = tile_m * tile_n
msce8.m  ms3, (rs1), rs2 #  8-bit acc store
msce16.m ms3, (rs1), rs2 # 16-bit acc store
msce32.m ms3, (rs1), rs2 # 32-bit acc store
msce64.m ms3, (rs1), rs2 # 64-bit acc store
```

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

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

# for left matrix, a
# tile size = tile_k * tile_m
msate8.m  ms3, (rs1), rs2 #  8-bit tile store
msate16.m ms3, (rs1), rs2 # 16-bit tile store
msate32.m ms3, (rs1), rs2 # 32-bit tile store
msate64.m ms3, (rs1), rs2 # 64-bit tile store

# for right matrix, b
# tile size = tile_n * tile_k
msbte8.m  ms3, (rs1), rs2 #  8-bit tile store
msbte16.m ms3, (rs1), rs2 # 16-bit tile store
msbte32.m ms3, (rs1), rs2 # 32-bit tile store
msbte64.m ms3, (rs1), rs2 # 64-bit tile store

# for output matrix, c
# tile size = tile_n * tile_m
mscte8.m  ms3, (rs1), rs2 #  8-bit acc store
mscte16.m ms3, (rs1), rs2 # 16-bit acc store
mscte32.m ms3, (rs1), rs2 # 32-bit acc store
mscte64.m ms3, (rs1), rs2 # 64-bit acc store
```

4.4. Arithmetic Instructions

4.4.1. Matrix Multiplication Instructions

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

```
# int matrix multiplication and add, md = md + ms1 * ms2
mma.mm md, ms1, ms2
mwma.mm md, ms1, ms2    # output double widen
mqma.mm md, ms1, ms2    # output quadruple widen

# float matrix multiplication and add, md = md + ms1 * ms2
mfma.mm md, ms1, ms2
mfwma.mm md, ms1, ms2   # output double widen
```

4.4.2. Element-Wise Add/Sub/Multiply Instructions

Matrix element-wise add/sub/multiply instructions.

```
# int matrix element-wise add, md[i,j] = md[i,j] + ms1[i,j]
maddc.mm md, ms1
mwaddc.mm md, ms1           # output double widen
mqaddc.mm md, ms1           # output quadruple widen

# int matrix element-wise subtract, md[i,j] = md[i,j] - ms1[i,j]
msubc.mm md, ms1
mwsubc.mm md, ms1           # output double widen
mqsubc.mm md, ms1           # output quadruple widen

# int matrix element-wise reverse subtract, md[i,j] = ms1[i,j] - md[i,j]
mrsubc.mm md, ms1
mwrsubc.mm md, ms1          # output double widen
mqrsubc.mm md, ms1          # output quadruple widen

# int matrix element-wise multiply with scalar int, md[i,j] = ms1[i,j] * rs2
memulc.mx md, ms1, rs2
mwemulc.mx md, ms1, rs2     # output double widen
mqemulc.mx md, ms1, rs2     # output quadruple widen

# int matrix element-wise multiply with int imm, md[i,j] = ms1[i,j] * imm
memulc.mi md, ms1, imm
mwemulc.mi md, ms1, imm     # output double widen
mqemulc.mi md, ms1, imm     # output quadruple widen


# float matrix element-wise add, md[i,j] = md[i,j] + ms1[i,j]
mfaddc.mm md, ms1
mfwaddc.mm md, ms1          # output double widen

# float matrix element-wise subtract, md[i,j] = md[i,j] - ms1[i,j]
mfsubc.mm md, ms1
mfwsubc.mm md, ms1          # output double widen
```

```
# float matrix element-wise reverse subtract, md[i,j] = ms1[i,j] - md[i,j]
mfrsubc.mm md, ms1
mfwrsubc.mm md, ms1         # output double widen

# float matrix element-wise multiply with scalar float, md[i,j] = ms1[i,j] * rs2
mfemulc.mf md, ms1, rs2
mfwemulc.mf md, ms1, rs2    # output double widen
```

4.4.3. Type-Convert Instructions

```
# convert float to float
mfncvtc.f.fw.m  md, ms1     # double-width float to single-width float
mfwcvtc.fw.f.m  md, ms1     # single-width float to double-width float

# convert int to float
mfcvtc.f.x.m md, ms1        # int to float

mfncvtc.f.xw.m md, ms1      # double-width int to float
mfncvtc.f.xq.m md, ms1      # quad-width int to float

mfwcvtc.fw.x.m md, ms1      # single-width int to double-width float
mfcvtc.fw.xw.m md, ms1      # double-width int to double-width float
mfncvtc.fw.xq.m md, ms1     # quad-width int to double-width float

# convert float to int
mfcvtc.x.f.m md, ms1        # float to int

mfwcvtc.xw.f.m md, ms1      # float to double-width int
mfwcvtc.xq.f.m md, ms1      # float to quad-width int

mfncvtc.x.fw.m md, ms1      # double-width float to single-width int
mfcvtc.xw.fw.m md, ms1      # double-width float to double-width int
mfwcvtc.xq.fw.m md, ms1     # double-width float to quad-width int
```

4.5. Instruction Listing

No.		**31 28**	27 20	19 15	14 12	11 7	6 0
Configuration		funct4		rs1	funct3	rd	opcode
1	msettypei	0000	mtypei[27:15]		111	rd	1110111
2	msettype	0001	00000000	rs1	111	rd	1110111
3	msettilemi	0010	mleni[27:15]		111	rd	1110111
4	msettilem	0011	00000000	rs1	111	rd	1110111

5	msettileki	0100	mleni[27:15]			111	rd	1110111
6	msettilek	0101	00000000		rs1	111	rd	1110111
7	msettileni	0110	mleni[27:15]			111	rd	1110111
8	msettilen	0111	00000000		rs1	111	rd	1110111
No.		**31 26**	25	24 20	19 15	14 12	11 7	6 0
Load		funct6	ls	rs2	rs1	eew	md	opcode
9	mlae8.m	000001	0	rs2	rs1	000	md	1110111
10	mlae16.m	000001	0	rs2	rs1	001	md	1110111
11	mlae32.m	000001	0	rs2	rs1	010	md	1110111
12	mlae64.m	000001	0	rs2	rs1	011	md	1110111
13	mlbe8.m	000010	0	rs2	rs1	000	md	1110111
14	mlbe16.m	000010	0	rs2	rs1	001	md	1110111
15	mlbe32.m	000010	0	rs2	rs1	010	md	1110111
16	mlbe64.m	000010	0	rs2	rs1	011	md	1110111
17	mlce8.m	000000	0	rs2	rs1	000	md	1110111
18	mlce16.m	000000	0	rs2	rs1	001	md	1110111
19	mlce32.m	000000	0	rs2	rs1	010	md	1110111
20	mlce64.m	000000	0	rs2	rs1	011	md	1110111
9	mlate8.m	000101	0	rs2	rs1	000	md	1110111
10	mlate16.m	000101	0	rs2	rs1	001	md	1110111
11	mlate32.m	000101	0	rs2	rs1	010	md	1110111
12	mlate64.m	000101	0	rs2	rs1	011	md	1110111
13	mlbte8.m	000110	0	rs2	rs1	000	md	1110111
14	mlbte16.m	000110	0	rs2	rs1	001	md	1110111
15	mlbte32.m	000110	0	rs2	rs1	010	md	1110111
16	mlbte64.m	000110	0	rs2	rs1	011	md	1110111
17	mlcte8.m	000100	0	rs2	rs1	000	md	1110111
18	mlcte16.m	000100	0	rs2	rs1	001	md	1110111
19	mlcte32.m	000100	0	rs2	rs1	010	md	1110111

20	mlcte64.m	000100	0	rs2	rs1	011	md	1110111
Store		funct6	ls	rs2	rs1	eew	ms3	opcode
21	msae8.m	000001	1	rs2	rs1	000	ms3	1110111
22	msae16.m	000001	1	rs2	rs1	001	ms3	1110111
23	msae32.m	000001	1	rs2	rs1	010	ms3	1110111
24	msae64.m	000001	1	rs2	rs1	011	ms3	1110111
25	msbe8.m	000010	1	rs2	rs1	000	ms3	1110111
26	msbe16.m	000010	1	rs2	rs1	001	ms3	1110111
27	msbe32.m	000010	1	rs2	rs1	010	ms3	1110111
28	msbe64.m	000010	1	rs2	rs1	011	ms3	1110111
29	msce8.m	000000	1	rs2	rs1	000	ms3	1110111
30	msce16.m	000000	1	rs2	rs1	001	ms3	1110111
31	msce32.m	000000	1	rs2	rs1	010	ms3	1110111
32	msce64.m	000000	1	rs2	rs1	011	ms3	1110111
21	msate8.m	000101	1	rs2	rs1	000	ms3	1110111
22	msate16.m	000101	1	rs2	rs1	001	ms3	1110111
23	msate32.m	000101	1	rs2	rs1	010	ms3	1110111
24	msate64.m	000101	1	rs2	rs1	011	ms3	1110111
25	msbte8.m	000110	1	rs2	rs1	000	ms3	1110111
26	msbte16.m	000110	1	rs2	rs1	001	ms3	1110111
27	msbte32.m	000110	1	rs2	rs1	010	ms3	1110111
28	msbte64.m	000110	1	rs2	rs1	011	ms3	1110111
29	mscte8.m	000100	1	rs2	rs1	000	ms3	1110111
30	mscte16.m	000100	1	rs2	rs1	001	ms3	1110111
31	mscte32.m	000100	1	rs2	rs1	010	ms3	1110111
32	mscte64.m	000100	1	rs2	rs1	011	ms3	1110111
Arithmetic		funct6	fp	ms2	ms1	funct3	md	opcode
33	mma.mm	000000	0	ms2	ms1	110	md	1110111
34	mfma.mm	000000	1	ms2	ms1	110	md	1110111

35	mwma.mm	000001	0	ms2	ms1	110	md	1110111
36	mfwma.mm	000001	1	ms2	ms1	110	md	1110111
37	mqma.mm	000010	0	ms2	ms1	110	md	1110111
38	maddc.mm	000100	0	00000	ms1	110	md	1110111
39	mfaddc.mm	000100	1	00000	ms1	110	md	1110111
40	mwaddc.mm	000101	0	00000	ms1	110	md	1110111
41	mfwaddc.mm	000101	1	00000	ms1	110	md	1110111
42	mqaddc.mm	000110	0	00000	ms1	110	md	1110111
43	msubc.mm	000111	0	00000	ms1	110	md	1110111
44	mfsubc.mm	000111	1	00000	ms1	110	md	1110111
45	mwsubc.mm	001000	0	00000	ms1	110	md	1110111
46	mfwsubc.mm	001000	1	00000	ms1	110	md	1110111
47	mqsubc.mm	001001	0	00000	ms1	110	md	1110111
48	mrsubc.mm	001010	0	00000	ms1	110	md	1110111
49	mfrsubc.mm	001010	1	00000	ms1	110	md	1110111
50	mwrsubc.mm	001011	0	00000	ms1	110	md	1110111
51	mfwrsubc.mm	001011	1	00000	ms1	110	md	1110111
52	mqrsubc.mm	001100	0	00000	ms1	110	md	1110111
53	memulc.mx	001101	0	rs2	ms1	110	md	1110111
54	mfemulc.mf	001101	1	rs2	ms1	110	md	1110111
55	mwemulc.mx	001110	0	rs2	ms1	110	md	1110111
56	mfwemulc.mf	001110	1	rs2	ms1	110	md	1110111
57	mqemulc.mx	001111	0	rs2	ms1	110	md	1110111
58	memulc.mi	010000	0	imm	ms1	110	md	1110111
59	mwemulc.mi	010001	0	imm	ms1	110	md	1110111
60	mqemulc.mi	010010	0	imm	ms1	110	md	1110111
Convert		funct6	fdst	ms2	ms1	funct3	md	opcode
61	mfncvtc.f.fw.m	011000	1	00000	ms1	110	md	1110111
62	mfwcvtc.fw.f.m	011000	0	00000	ms1	110	md	1110111

63	mfcvtc.f.x.m	011010	1	00000	ms1	110	md	1110111
64	mfcvtc.x.f.m	011010	0	00000	ms1	110	md	1110111
65	mfncvtc.f.xw.m	011011	1	00000	ms1	110	md	1110111
66	mfwcvtc.xw.f.m	011011	0	00000	ms1	110	md	1110111
67	mfncvtc.f.xq.m	011100	1	00000	ms1	110	md	1110111
68	mfwcvtc.xq.f.m	011100	0	00000	ms1	110	md	1110111
69	mfwcvtc.fw.x.m	011101	1	00000	ms1	110	md	1110111
70	mfncvtc.x.fw.m	011101	0	00000	ms1	110	md	1110111
71	mfcvtc.fw.xw.m	011110	1	00000	ms1	110	md	1110111
72	mfcvtc.xw.fw.m	011110	0	00000	ms1	110	md	1110111
73	mfncvtc.fw.xq.m	011111	1	00000	ms1	110	md	1110111
74	mfwcvtc.xq.fw.m	011111	0	00000	ms1	110	md	1110111

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+=tile_m) {                  // loop at dim m with tiling
        tile_m = msettile_m(m-i);
        for (j=0; j<n; j+=tile_n) {              // loop at dim n with tiling
            tile_n = msettile_n(n-j);

            acc = mfemul_mf(acc, 0.f)            // clear acc reg
            for (s=0; s<k; s+=tile_k) {          // loop at dim k with tiling
                tile_k = msettile_k(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
                acc = mfwma_mm(tr1, tr2);        // tiled matrix multiply,
                                                 // double widen output acc
            }

            acc = mfncvt_f_fw_m(acc);            // convert widen result
            msce16_m(acc, &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+=tile_m) {                  // loop at dim m with tiling
        tile_m = msettile_m(m-i);
        for (j=0; j<n; j+=tile_n) {              // loop at dim n with tiling
            tile_n = msettile_n(n-j);

            acc = mfemul_mf(acc, 0.f)            // clear acc reg
            for (s=0; s<k; s+=tile_k) {          // loop at dim k with tiling
                tile_k = msettile_k(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
                acc = mfwma_mm(tr1, tr2);        // tiled matrix multiply,
                                                 // double widen output acc
            }
```

```
            acc = mfncvt_f_fw_m(acc);           // convert widen result
            msce16_m(acc, &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+=tile_m) {                  // loop at dim m with tiling
        tile_m = msettile_m(h-i);
        for (j=0; j<w; j+=tile_k) {              // loop at dim k with tiling
            tile_k = msettile_k(w-j);

            tr1 = mlae16_m(&in[i][j], w*2);    // load input matrix
            msate16_m(tr1, &out[j][i], h*2);  // store output matrix
        }
    }
}
```

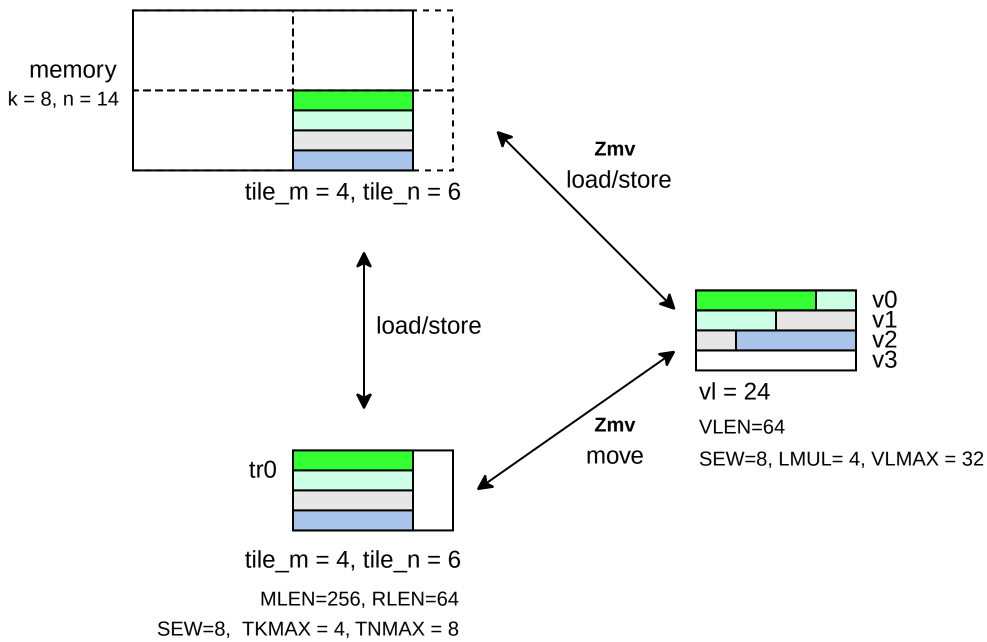
Chapter 6. Standard Matrix Extensions

6.1. 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. Element-wise multiply between a matrix register and a vector register(broadcast to a matrix) is also supported.

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



6.1.1. 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/acc slices load to vregs
mlae16.v   vd, (rs1), rs2 # 16-bit tile/acc slices load to vregs
mlae32.v   vd, (rs1), rs2 # 32-bit tile/acc slices load to vregs
mlae64.v   vd, (rs1), rs2 # 64-bit tile/acc slices load to vregs

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

```
mlbe64.v   vd, (rs1), rs2 # 64-bit tile/acc slices load to vregs

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

6.1.2. 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/acc slices store from vregs
msae16.v   vs3, (rs1), rs2 # 16-bit tile/acc slices store from vregs
msae32.v   vs3, (rs1), rs2 # 32-bit tile/acc slices store from vregs
msae64.v   vs3, (rs1), rs2 # 64-bit tile/acc slices store from vregs

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

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

6.1.3. Data Move Instructions

Normal data move, rows or columns = lmul, eew = sew.

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

# vd[(i - rs2) * tile_n + j] = md[i, j], i = rs2 .. rs2 + lmul - 1
mmvar.v.m  vd, ms1, rs2
mmvbr.v.m  vd, ms1, rs2
mmvcr.v.m  vd, ms1, rs2

# md[i, j] = vd[(i - rs2) * tile_n + j], i = rs2 .. rs2 + lmul - 1
mmvar.m.v  md, vs1, rs2
mmvbr.m.v  md, vs1, rs2
mmvcr.m.v  md, vs1, rs2
```

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

# vd[(j - rs2) * tile_m + i] = md[i, j], j = rs2 .. rs2 + lmul - 1
mmvac.v.m  vd, ms1, rs2
mmvbc.v.m  vd, ms1, rs2
mmvcc.v.m  vd, ms1, rs2

# md[i, j] = vd[(j - rs2) * tile_m + i], j = rs2 .. rs2 + lmul - 1
mmvac.m.v  md, vs1, rs2
mmvbc.m.v  md, vs1, rs2
mmvcc.m.v  md, vs1, rs2
```

Double widen data move, rows or columns = lmul / 2, eew = sew * 2.

Only support accumulator registers as source and destination registers.

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

# vd[(i - rs2) * tile_n + j] = md[i, j], i = rs2 .. rs2 + lmul/2 - 1
mwmvcr.v.m  vd, ms1, rs2

# md[i, j] = vd[(i - rs2) * tile_n + j], i = rs2 .. rs2 + lmul/2 - 1
mwmvcr.m.v  md, vs1, rs2


# Data move between matrix register columns and vector registers.

# vd[(j - rs2) * tile_m + i] = md[i, j], j = rs2 .. rs2 + lmul/2 - 1
mwmvcc.v.m  vd, ms1, rs2

# md[i, j] = vd[(j - rs2) * tile_m + i], j = rs2 .. rs2 + lmul/2 - 1
mwmvcc.m.v  md, vs1, rs2
```

quadruple widen data move, rows or columns = lmul / 4, eew = sew * 4.

Only support accumulator registers as source and destination registers.

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

# vd[(i - rs2) * tile_n + j] = md[i, j], i = rs2 .. rs2 + lmul/4 - 1
mqmvcr.v.m  vd, ms1, rs2  # vl = lmul * tile_n

# md[i, j] = vd[(i - rs2) * tile_n + j], i = rs2 .. rs2 + lmul/4 - 1
mqmvcr.m.v  md, vs1, rs2  # vl = lmul * tile_n
```

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

# vd[(j - rs2) * tile_m + i] = md[i, j], j = rs2 .. rs2 + lmul/4 - 1
mqmvcc.v.m  vd, ms1, rs2  # vl = lmul * tile_m

# md[i, j] = vd[(j - rs2) * tile_m + i], j = rs2 .. rs2 + lmul/4 - 1
mqmvcc.m.v  md, vs1, rs2  # vl = lmul * tile_m
```

6.1.4. Matrix element-wise multiply

Matrix element-wise multiply instructions, the vector operand will broadcast to a matrix.

For m*emulcr.mv instructions, the vector operand will broadcast from a row to a c matrix.

```
# int matrix element-wise multiply with a row of vector int
# md[i,j] = ms1[i,j] * vs2[j]
memulcr.mv md, ms1, vs2
mwemulcr.mv md, ms1, vs2  # output double widen
mqemulcr.mv md, ms1, vs2  # output quadruple widen

# float matrix element-wise multiply with a row of vector float
# md[i,j] = ms1[i,j] * vs2[j]
mfemulcr.mv md, ms1, vs2
mfwemulcr.mv md, ms1, vs2  # output double widen
```

For m*emulcc.mv instructions, the vector operand will broadcast from a column to a c matrix.

```
# int matrix element-wise multiply with a column of vector int,
# md[i,j] = ms1[i,j] * vs2[i]
memulcc.mv md, ms1, vs2
mwemulcc.mv md, ms1, vs2  # output double widen
mqemulcc.mv md, ms1, vs2  # output quadruple widen

# float matrix element-wise multiply with a column of vector float,
# md[i,j] = ms1[i,j] * vs2[i]
mfemulcc.mv md, ms1, vs2
mfwemulcc.mv md, ms1, vs2  # output double widen
```

6.1.5. Matrix element-wise add

Matrix element-wise add instructions, the vector operand will broadcast to a matrix.

For m*addcr.mv instructions, the vector operand will broadcast from a row to a c matrix.

```
# int matrix element-wise add with a row of vector int
```

```
# md[i,j] = ms1[i,j] + vs2[j]
maddcr.mv md, ms1, vs2
mwaddcr.mv md, ms1, vs2  # output double widen
mqaddcr.mv md, ms1, vs2  # output quadruple widen

# float matrix element-wise add with a row of vector float
# md[i,j] = ms1[i,j] + vs2[j]
mfaddcr.mv md, ms1, vs2
mfwaddcr.mv md, ms1, vs2  # output double widen
```

For m*addcc.mv instructions, the vector operand will broadcast from a column to a c matrix.

```
# int matrix element-wise add with a column of vector int,
# md[i,j] = ms1[i,j] + vs2[i]
maddcc.mv md, ms1, vs2
mwaddcc.mv md, ms1, vs2  # output double widen
mqaddcc.mv md, ms1, vs2  # output quadruple widen

# float matrix element-wise add with a column of vector float,
# md[i,j] = ms1[i,j] + vs2[i]
mfaddcc.mv md, ms1, vs2
mfwaddcc.mv md, ms1, vs2  # output double widen
```

6.1.6. Matrix element-wise fused multiply-accumulate

Matrix element-wise fused multiply-add instructions, the vector operand will broadcast to a matrix.

For m*macccr.mv instructions, the vector operand will broadcast from a row to a c matrix.

```
# int matrix element-wise multiply-accumulate with a row of vector int
# md[i,j] = vs1[j] * md[i,j] + vs2[j]
mmacccr.mv md, vs1, vs2
mwmacccr.mv md, vs1, vs2  # output double widen
mqmacccr.mv md, vs1, vs2  # output quadruple widen

# float matrix element-wise multiply-accumulate with a row of vector float
# md[i,j] = vs1[j] * md[i,j] + vs2[j]
mfmacccr.mv md, vs1, vs2
mfwmacccr.mv md, vs1, vs2  # output double widen
```

For m*macccc.mv instructions, the vector operand will broadcast from a column to a c matrix.

```
# int matrix element-wise multiply-accumulate with a column of vector int,
# md[i,j] = vs1[i] * md[i,j] + vs2[i]
mmacccc.mv md, vs1, vs2
```

```
mwmacccc.mv md, vs1, vs2  # output double widen
mqmacccc.mv md, vs1, vs2  # output quadruple widen

# float matrix element-wise add with a column of vector float,
# md[i,j] = vs1[i] * md[i,j] + vs2[i]
mfmacccc.mv md, vs1, vs2
mfwmacccc.mv md, vs1, vs2  # output double widen
```

6.1.7. Instruction Listing

No.		31 26	25	24 20	19 15	14 12	11 7	6 0
Load		funct6	ls	rs2	rs1	eew	md	opcode
1	mlae8.v	100001	0	rs2	rs1	000	md	1110111
2	mlae16.v	100001	0	rs2	rs1	001	md	1110111
3	mlae32.v	100001	0	rs2	rs1	010	md	1110111
4	mlae64.v	100001	0	rs2	rs1	011	md	1110111
5	mlbe8.v	100010	0	rs2	rs1	000	md	1110111
6	mlbe16.v	100010	0	rs2	rs1	001	md	1110111
7	mlbe32.v	100010	0	rs2	rs1	010	md	1110111
8	mlbe64.v	100010	0	rs2	rs1	011	md	1110111
9	mlce8.v	100000	0	rs2	rs1	000	md	1110111
10	mlce16.v	100000	0	rs2	rs1	001	md	1110111
11	mlce32.v	100000	0	rs2	rs1	010	md	1110111
12	mlce64.v	100000	0	rs2	rs1	011	md	1110111
Store		funct6	ls	rs2	rs1	eew	ms3	opcode
13	msae8.v	100001	1	rs2	rs1	000	ms3	1110111
14	msae16.v	100001	1	rs2	rs1	001	ms3	1110111
15	msae32.v	100001	1	rs2	rs1	010	ms3	1110111
16	msae64.v	100001	1	rs2	rs1	011	ms3	1110111
17	msbe8.v	100010	1	rs2	rs1	000	ms3	1110111
18	msbe16.v	100010	1	rs2	rs1	001	ms3	1110111
19	msbe32.v	100010	1	rs2	rs1	010	ms3	1110111

No.		31 26	25	24 20	19 15	14 12	11 7	6 0
20	msbe64.v	100010	1	rs2	rs1	011	ms3	1110111
21	msce8.v	100000	1	rs2	rs1	000	ms3	1110111
22	msce16.v	100000	1	rs2	rs1	001	ms3	1110111
23	msce32.v	100000	1	rs2	rs1	010	ms3	1110111
24	msce64.v	100000	1	rs2	rs1	011	ms3	1110111
Data Move		funct6	v2m	rs2	*s1	funct3	*d	opcode
25	mmvar.v.m	000001	0	rs2	ms1	101	vd	1110111
26	mmvar.m.v	000001	1	rs2	vs1	101	md	1110111
27	mmvbr.v.m	000010	0	rs2	ms1	101	vd	1110111
28	mmvbr.m.v	000010	1	rs2	vs1	101	md	1110111
29	mmvcr.v.m	000000	0	rs2	ms1	101	vd	1110111
30	mmvcr.m.v	000000	1	rs2	vs1	101	md	1110111
31	mmvac.v.m	000101	0	rs2	ms1	101	vd	1110111
32	mmvac.m.v	000101	1	rs2	vs1	101	md	1110111
33	mmvbc.v.m	000110	0	rs2	ms1	101	vd	1110111
34	mmvbc.m.v	000110	1	rs2	vs1	101	md	1110111
35	mmvcc.v.m	000100	0	rs2	ms1	101	vd	1110111
36	mmvcc.m.v	000100	1	rs2	vs1	101	md	1110111
37	mwmvcr.v.m	010000	0	rs2	ms1	101	vd	1110111
38	mwmvcr.m.v	010000	1	rs2	vs1	101	md	1110111
39	mwmvcc.v.m	010100	0	rs2	ms1	101	vd	1110111
40	mwmvcc.m.v	010100	1	rs2	vs1	101	md	1110111
41	mqmvcr.v.m	100000	0	rs2	ms1	101	vd	1110111
42	mqmvcr.m.v	100000	1	rs2	vs1	101	md	1110111
43	mqmvcc.v.m	100100	0	rs2	ms1	101	vd	1110111
44	mqmvcc.m.v	100100	1	rs2	vs1	101	md	1110111
Arithmetic		funct6	fp	vs2	ms1	funct3	md	opcode
45	memulcr.mv	100001	0	vs2	ms1	110	md	1110111

No.		31 26	25	24 20	19 15	14 12	11 7	6 0
46	mfemulcr.mv	100001	1	vs2	ms1	110	md	1110111
47	mwemulcr.mv	100010	0	vs2	ms1	110	md	1110111
48	mfwemulcr.mv	100010	1	vs2	ms1	110	md	1110111
49	mqemulcr.mv	100011	0	vs2	ms1	110	md	1110111
50	memulcc.mv	100100	0	vs2	ms1	110	md	1110111
51	mfemulcc.mv	100100	1	vs2	ms1	110	md	1110111
52	mwemulcc.mv	100101	0	vs2	ms1	110	md	1110111
53	mfwemulcc.mv	100101	1	vs2	ms1	110	md	1110111
54	mqemulcc.mv	100110	0	vs2	ms1	110	md	1110111
55	maddcr.mv	100111	0	vs2	ms1	110	md	1110111
56	mfaddcr.mv	100111	1	vs2	ms1	110	md	1110111
57	mwaddcr.mv	101000	0	vs2	ms1	110	md	1110111
58	mfwaddcr.mv	101000	1	vs2	ms1	110	md	1110111
59	mqaddcr.mv	101001	0	vs2	ms1	110	md	1110111
60	maddcc.mv	101010	0	vs2	ms1	110	md	1110111
61	mfaddcc.mv	101010	1	vs2	ms1	110	md	1110111
62	mwaddcc.mv	101011	0	vs2	ms1	110	md	1110111
63	mfwaddcc.mv	101011	1	vs2	ms1	110	md	1110111
64	mqaddcc.mv	101100	0	vs2	ms1	110	md	1110111
55	mmacccr.mv	101101	0	vs2	vs1	110	md	1110111
56	mfmacccr.mv	101101	1	vs2	vs1	110	md	1110111
57	mwmacccr.mv	101110	0	vs2	vs1	110	md	1110111
58	mfwmacccr.mv	101110	1	vs2	vs1	110	md	1110111
59	mqmacccr.mv	101111	0	vs2	vs1	110	md	1110111
60	mmacccc.mv	110000	0	vs2	vs1	110	md	1110111
61	mfmacccc.mv	110000	1	vs2	vs1	110	md	1110111
62	mwmacccc.mv	110001	0	vs2	vs1	110	md	1110111
63	mfwmacccc.mv	110001	1	vs2	vs1	110	md	1110111

No.		31 26	25	24 20	19 15	14 12	11 7	6 0
64	mqmacccc.mv	110010	0	vs2	vs1	110	md	1110111

6.1.8. Intrinsic Examples: 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 = msettile_m(m-i);
        for (j=0; j<n; j+=tile_n) {             // loop at dim n with tiling
            tile_n = msettile_n(n-j);

            acc = mfemul_mf(acc, 0.f)           // clear acc reg
            for (s=0; s<k; s+=tile_k) {         // loop at dim k with tiling
                tile_k = msettile_k(k-s);

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

            acc = mfncvt_f_fw_m(acc);           // convert widen result to single


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

                v1 = mmvcr_v_m(acc, s);         // move acc 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
            }
        }
    }
}
```

6.2. Zmbf16: Matrix Bfloat16(BF16) Extension

The Zmbf16 extension allows to use BF16 format as the data type of input matrix elements.

The Zmbf16 extension add a bit `mtype[4]` in `mtype` register.

Table 4. `mtype` register layout

Bits	Name	Description
XLEN-1	mill	Illegal value if set
XLEN-2:5	0	Reserved if non-zero
4	**mbf16**	**Use BF16 input format**
3	maccq	Support quad-width accumulator element
2:0	msew[2:0]	Selected element width (SEW) setting

The new `mtype` value is encoded in the immediate fields of msettypei, and in the rs1 register for msettype.

```
Suggested bf16 assembler name used for msettypei mtypei immediate

    bf16  # Use BF16 format

Examples:

    msettypei t0, e16, bf16         # SEW = 16, use BF16 as input matrix element
```

For implemention not support Bfloat16 format, `mtype.mill` will be set.

`bf16` should be always used with `e16`(SEW=16), otherwise `mtype.mill` will be set.

6.3. Zmtf32: Matrix TensorFloat-32(TF32) Extension

The Zmtf32 extension allows to use TF32 FMA for matrix multiplication.

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

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

The Zmtf32 extension add a bit `mtype[5]` in `mtype` register.

Table 5. `mtype` register layout

Bits	Name	Description
XLEN-1	mill	Illegal value if set
XLEN-2:6	0	Reserved if non-zero

Bits	Name	Description
5	**mtf32**	**Enable TF32 FMA for matrix multiplication**
4	mbf16	Use bfloat16 input format
3	maccq	Support quad-width accumulator element
2:0	msew[2:0]	Selected element width (SEW) setting

The new `mtype` value is encoded in the immediate fields of msettypei, and in the rs1 register for msettype.

```
Suggested tf32 assembler name used for msettypei mtypei immediate

    tf32  # enable TF32 FMA

Examples:

    msettypei t0, e32, tf32         # SEW = 32, enable TF32 FMA
```

For implemention not support TF32 format, `mtype.mill` will be set.

`tf32` should be always used with `e32`(SEW=32), otherwise `mtype.mill` will be set.

6.4. Zmic: Im2col Matrix Multiplication Extension

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

The Zmic extension allows to perform the im2col operation on-the-fly, by the new load instructions.

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

6.4.1. CSRs

The matrix extension adds 5 unprivileged CSRs (mkrsh, mfdsh, mpad, mstdi, msk) to the base scalar RISC-V ISA.

Table 6. New matrix CSRs

Address	Privilege	Name	Description
0xC45	URO	moutsh	Fold/unfold output shape

Address	Privilege	Name	Description
0xC46	URO	minsh	Fold/unfold input shape
0xC47	URO	mpad	Fold/unfold padding parameters
0xC48	URO	mstdi	Fold/unfold sliding strides and dilations
0xC49	URO	minsk	Fold/unfold sliding kernel position of input
0xC50	URO	moutsk	Fold/unfold sliding kernel position of output

Table 7. `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 8. `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 9. `mstdi` *register layout*

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

Table 10. `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

6.4.2. 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)
```

6.4.3. Load Unfold Instructions

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

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

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

6.4.4. Store Fold Instructions

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

```
# ms3 destination, rs1 base address, rs2 row byte stride

# for left matrix, a
msfdae8.m    ms3, (rs1), rs2
msfdae16.m   ms3, (rs1), rs2
msfdae32.m   ms3, (rs1), rs2
msfdae64.m   ms3, (rs1), rs2
```

6.4.5. Instruction Listing

No.		**31 28**	27 25	24 20	19 15	14 12	11 7	6 0
Configuration		funct4	000	rs2	rs1	funct3	rd	opcode
2	msetoutsh	1000	000	rs2	rs1	111	rd	1110111
4	mseindsh	1001	000	rs2	rs1	111	rd	1110111
6	msetsk	1010	00000000		rs1	111	rd	1110111
No.		**31 26**	25	24 20	19 15	14 12	11 7	6 0
Load		funct6	ls	rs2	rs1	eew	md	opcode
1	mlufae8.m	110001	0	rs2	rs1	000	md	1110111
2	mlufae16.m	110001	0	rs2	rs1	001	md	1110111
3	mlufae32.m	110001	0	rs2	rs1	010	md	1110111
4	mlufae64.m	110001	0	rs2	rs1	011	md	1110111
5	mlufbe8.m	110010	0	rs2	rs1	000	md	1110111
6	mlufbe16.m	110010	0	rs2	rs1	001	md	1110111
7	mlufbe32.m	110010	0	rs2	rs1	010	md	1110111
8	mlufbe64.m	110010	0	rs2	rs1	011	md	1110111
9	mlufce8.m	110000	0	rs2	rs1	000	md	1110111
10	mlufce16.m	110000	0	rs2	rs1	001	md	1110111
11	mlufce32.m	110000	0	rs2	rs1	010	md	1110111
12	mlufce64.m	110000	0	rs2	rs1	011	md	1110111
Store		funct6	ls	rs2	rs1	eew	ms3	opcode
13	msfdae8.m	110001	1	rs2	rs1	000	ms3	1110111
14	msfdae16.m	110001	1	rs2	rs1	001	ms3	1110111
15	msfdae32.m	110001	1	rs2	rs1	010	ms3	1110111
16	msfdae64.m	110001	1	rs2	rs1	011	ms3	1110111
17	msfdbe8.m	110010	1	rs2	rs1	000	ms3	1110111
18	msfdbe16.m	110010	1	rs2	rs1	001	ms3	1110111
19	msfdbe32.m	110010	1	rs2	rs1	010	ms3	1110111
20	msfdbe64.m	110010	1	rs2	rs1	011	ms3	1110111

21	msfdce8.m	110000	1	rs2	rs1	000	ms3	1110111
22	msfdce16.m	110000	1	rs2	rs1	001	ms3	1110111
23	msfdce32.m	110000	1	rs2	rs1	010	ms3	1110111
24	msfdce64.m	110000	1	rs2	rs1	011	ms3	1110111

6.4.6. Intrinsic Examples: 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 = msettile_m(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 = msettile_n(n-j);

            acc = mfemul_mf(acc, 0.f)           // clear acc 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

                    for (skc=0; skc<inc; skc+=tile_k) { // loop for kernel channels
                        tile_k = msettile_k(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
                        acc = mfwma_mm(tr1, tr2);   // tiled matrix multiply,
                                                    // double widen output acc
```

```
                    }
                }
            }

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

6.4.7. Intrinsic Examples: 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 = msettile_m(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 = msettile_n(n-j);

            acc = mfemul_mf(acc, 0.f)           // clear acc 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 = msettile_k(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
                              acc = mfwma_mm(tr1, tr2); // tiled matrix multiply,
                                                       // double widen output acc
                          }
                      }
                  }
              }

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

6.4.8. Intrinsic Examples: 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 = msettile_m(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 = msettile_n(n-j);

            acc = mfemul_mf(acc, -inf)          // clear acc 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
```

```
                    acc1 = mlufce16_m(&in[inh_pos][inw_pos][j]);
                    acc = mfmaxc_mm(acc, acc1);
                }
            }

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

6.4.9. Intrinsic Examples: 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);

    for (i=0; i<m; i+=tile_m) {                 // loop at dim m with tiling
        tile_m = msettile_m(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 = msettile_n(n-j);

            acc = mfemul_mf(acc, -inf)          // clear acc 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
                    acc1 = mlufce16_m(&in[inh_pos][inw_pos][j]);
                    acc = mfaddc_mm(acc, acc1);
                }
            }
```

```
            acc = mfdivc_mf(acc, kh*kw);
            msce16_m(acc, &out[i][j], n*2);     // store to matrix c
        }
    }
}
```

Bibliography