RISC-V Matrix Specification

Fujie Fan, Xin Ouyang, Zhiqiang Liu

Version 0.3.0, 9/2023: 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:

- Hui Yao
- Kening Zhang
- Kun Hu
- Xin Yang
- Zhiyong Zhang
- **Your name here**

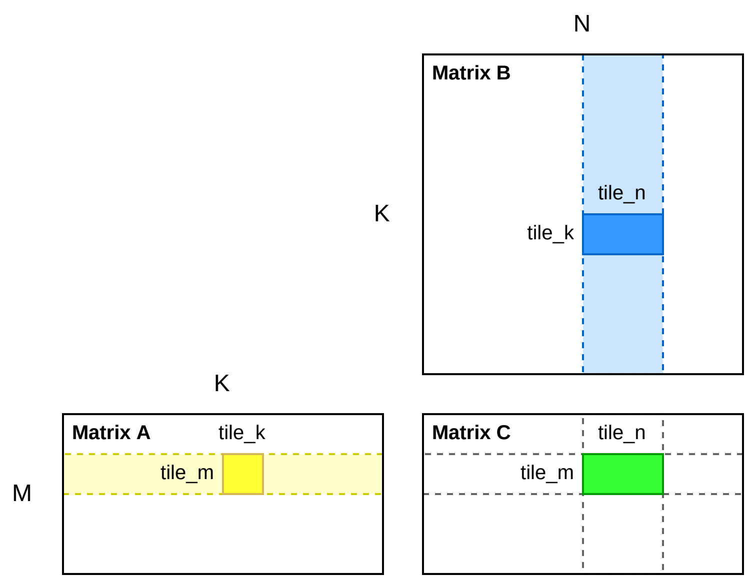
We will be very grateful to the huge number of other people who will have helped to improve this specification through their comments, reviews, feedback and questions.

Chapter 1. Introduction

This document describes the matrix extension for RISC-V.

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

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



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

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

Chapter 2. Implementation-defined Constant Parameters

Each hart supporting a matrix extension defines three parameters:

1. The maximum size in bits of a matrix element that any operation can produce or consume, $\text{ELEN} \geq 8$, 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 < 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 8 matrix registers, and 8 unprivileged CSRs to the base scalar RISC-V ISA.

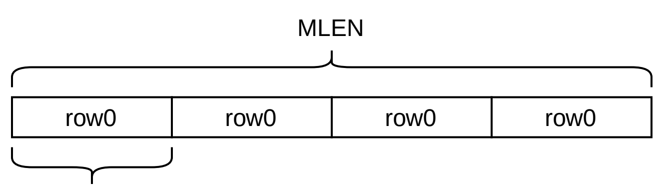
Table 1. Matrix CSRs

Address	Privilege	Name	Description
0xXXX	URO	mtype	Matrix tile data type register.
0xXXX	URO	mlenb	MLEN/8 (matrix tile register length in bytes).
0xXXX	URO	mrlenb	RLEN/8 (matrix tile register row length in bytes).
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	URW	mstart	Start element index.
0xXXX	URW	mcsr	Matrix control and status register.

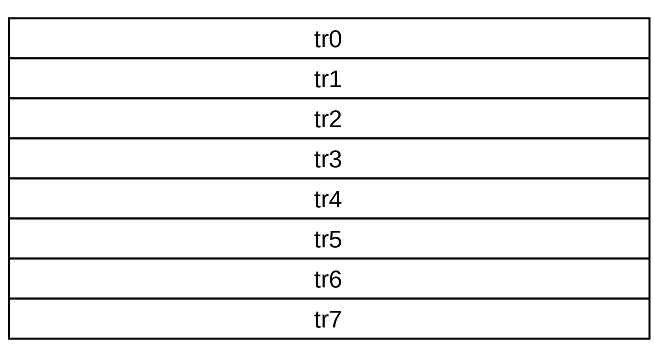
3.1. Matrix Registers

The matrix extension adds 8 architectural **Tile Registers** (tr0-tr7) 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.



Register Structure.



Register File.

An input matrix of matrix multipliaction 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 and quad-width output. 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/tr6). Odd-indexed `td` is reserved. A quad-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). Other-indexed `td` is reserved.

> Although there may be multiple output registers, the result can only be accessed by the first index.

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 5 fields, `mill`, `mfp64`, `mba`, `msew[2:0]` and `mlmul[1:0]`. Bits `mtype[XLEN-2:7]` 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-2:7	0	Reserved if non-zero.
6	mfp64	Support 64-bit float point.
5	mba	Matrix out of bound agnostic.
4:2	msew[2:0]	Selected element width (SEW) setting.
1:0	mlmul[1:0]	Register group multiplier (LMUL) setting.

The `mfp64` field is set to support 64-bit float point. To support FP64 format, the implementation should support "D" extension at the same time. If the implementation does not support FP64 but `mfp64` is set to 1, `mtype.mill` will be set to indicate such an exception.

The `mlmul` field sets the group size of input register. Load/store and element-wise instructions support grouped input and output, so that a single instruction can operate on multiple tile registers. Different from matrix multiplication instructions, load/store and element-wise instructions can be pipelined in element or row level. There is no order limit across elements and rows. So grouped operation can provide great efficiency with acceptable complexity.

> For widen-output instructions, the group size of output cannot be larger than 4.

The `mba` bit 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.

Table 3. `mlmul` field of `mtype`

mlmul[1:0]	Group Size
2'b00	1
2'b01	2
2'b10	4
2'b11	Reserved.

The LMUL can also be specified by the `lmul` field of an instruction, with higher priority than `mtype` setting.

Table 4. `lmul` field of instruction

lmul[1:0]	Group Size
2'b00	1
2'b01	2
2'b10	4
2'b11	Use `mtype.mlmul`.

> The `lmul` field of instruction provides a more efficient way for grouped operation, avoiding extra configuration instructions and possible misprediction of speculative CSR settings.

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

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

3.4. Matrix Start Index Register, mstart

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

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

3.5. Matrix Control and Status Register, mcsr

The `mcsr` register only has 1 field, and other bits with non-zero value are reserved.

Table 5. `mcsr` register layout

Bits	Name	Description
XLEN-1:1	0	Reserved if non-zero.
0	mxsat	Integer arithmetic instruction accrued saturation flag.

3.6. Matrix Context Status in mstatus and sstatus

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

Chapter 4. Instructions

4.1. Instruction Formats

The instructions in the matrix extension use 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 listed 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

Data Move instructions (funct3 = 101), where `di` field indicates the moving direction.

31 26	25	24 20	19 15	14 12	11 7	6 0
funct6	di	rs2/funct5	ts1/rs1	funct3	td/rd	1110111

Load & Store instructions (ew = 000 - 011), where `ls` field indicates the type (load or store) and `lmul` field indicates the group size.

31 26	25	24 20	19 15	14 12	11 10	9 7	6 0
funct6	ls	rs2	rs1	eew	lmul	td	1110111

Arithmetic & Type-Convert instructions (funct3 = 110), where `fp` field indicates the type (float or integer), `sn` field indicates the sign extension rule and `sa` field indicates saturated or unsaturated operation.

31 26	25	24 20	19	18	17 15	14 12	11 10	9 7	6 0
funct6	fp	ts2/rs2	sn	sa	ts1	funct3	lmul	td	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 `mtilem/mtilek/mtilen`. Based on microarchitecture implementation and mtype 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 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 mtilem/mtilek/mtilen CSRs based on their arguments, and write the new value into rd.

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

The `msettile` instruction sets the mtilem/mtilek/mtilen CSRs simultaneously, where the values are combined to rs1, and write the combined new values into rd.

```
msettile rd, rs1                    # rd[7:0] = new mtilem, rd[15:8] = new mtilen,
                                    # rd[XLEN-1:0] = new mtilek;
                                    # rs1[7:0] = ATM, rs1[15:8] = ATN,
                                    # rs1[XLEN-1:0] = ATK
```

To use the combined set instruction, the values of ATM and ATN must be held by 8 bits, and the value of ATK must be held by `XLEN-16` bits.

4.2.1. `mtype` Encoding

The `mtype` register has 5 fields, `mill`, `mfp64`, `mba`, `msew[2:0]` and `mlmul[1:0]`. Bits `mtype[XLEN-2:7]` should be written with zero, and non-zero values of this field are reserved.

Table 6. `mtype` register layout

Bits	Name	Description
XLEN-1	mill	Illegal value if set.
XLEN-2:7	0	Reserved if non-zero.
6	mfp64	Support 64-bit float point.
5	mba	Matrix out of bound agnostic.
4:2	msew[2:0]	Selected element width (SEW) setting.
1:0	mlmul[1:0]	Register group multiplier (LMUL) 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

    fp64  # support 64-bit float point

Examples:

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

    msettypei t0, e64, fp64          # SEW = 8, support 64-bit float point
```

4.2.2. `ATM/ATK/ATN` Encoding

There are three values, `TMMAX`, `TKMAX` and `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.

The values of `TMMAX`, `TKMAX` and `TNMAX` are related to MLEN, RLEN and the micro-architecture of implementation.

For examples, with MLEN=256 and RLEN=64, possible `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}` is encoded in the rs1 and rd fields.

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

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

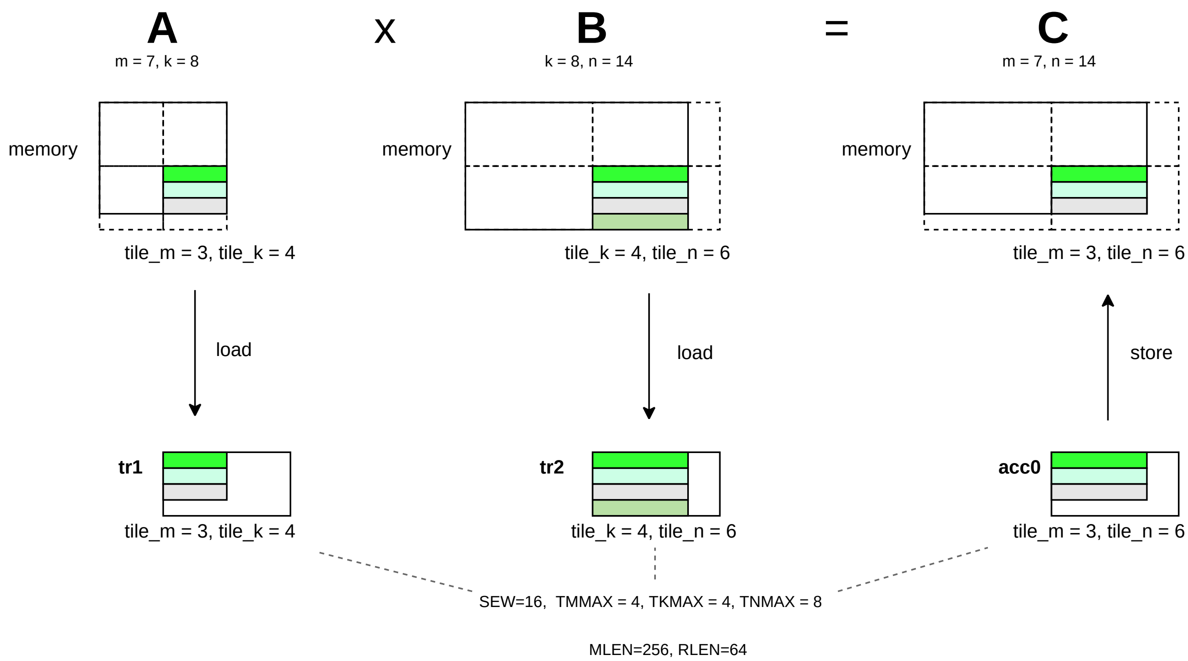
4.2.3. Constraints on Setting `mtilem/mtilek/mtilen`

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

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

Continue to use MLEN=256, RLEN=64 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, lmul   #  8-bit left tile load
mlae16.m td, (rs1), rs2, lmul   # 16-bit left tile load
mlae32.m td, (rs1), rs2, lmul   # 32-bit left tile load
mlae64.m td, (rs1), rs2, lmul   # 64-bit left tile load

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

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

Load a matrix tile from memory, 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, lmul  #  8-bit left tile load
mlate16.m td, (rs1), rs2, lmul  # 16-bit left tile load
mlate32.m td, (rs1), rs2, lmul  # 32-bit left tile load
mlate64.m td, (rs1), rs2, lmul  # 64-bit left tile load

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

# For output matrix, C
# tile size = mtilen * mtilem
mlcte8.m  td, (rs1), rs2, lmul  #  8-bit output tile load
mlcte16.m td, (rs1), rs2, lmul  # 16-bit output tile load
mlcte32.m td, (rs1), rs2, lmul  # 32-bit output tile load
mlcte64.m td, (rs1), rs2, lmul  # 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 * mtilek
msae8.m  ts3, (rs1), rs2, lmul  #  8-bit left tile store
msae16.m ts3, (rs1), rs2, lmul  # 16-bit left tile store
msae32.m ts3, (rs1), rs2, lmul  # 32-bit left tile store
msae64.m ts3, (rs1), rs2, lmul  # 64-bit left tile store
```

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

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

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

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

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

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

# For output matrix, C
# tile size = mtilen * mtilem
mscte8.m  ts3, (rs1), rs2, lmul #  8-bit output tile store
mscte16.m ts3, (rs1), rs2, lmul # 16-bit output tile store
mscte32.m ts3, (rs1), rs2, lmul # 32-bit output tile store
mscte64.m ts3, (rs1), rs2, lmul # 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, lmul   #  8-bit whole matrix load
mlre16.m td, (rs1), rs2, lmul   # 16-bit whole matrix load
mlre32.m td, (rs1), rs2, lmul   # 32-bit whole matrix load
```

```
mlre64.m td, (rs1), rs2, lmul   # 64-bit whole matrix load
```

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

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

Whole matrix load and store instructions are usually used for context saving and restoring, and no tranposed version is provided.

4.4. Data Move Instructions

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]
mmv.x.s rd, ts1, rs2

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

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

The `mmv.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 valud is sign-extended to SEW bits. The other elements of the tile register are treated as out-of-bound elements, using the setting of `mtype.mba`.

Float point data move instructions are similar.

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

# td[i, j] = f[rs1], i = rs2[15:0], j = rs2[XLEN-1:16]
mfmv.s.f td, rs1, rs2
```

The pseudo-instruction `mmv.m.m td, ts1` to move a whole matrix between two tile registers can be implemented as `memul.mi td, ts1, 1, m1`.

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.
mbcac.m td, ts1
mbcbc.m td, ts1
mbccc.m td, ts1

# Broadcast the first element to fill the whole matrix.
mbcae.m td, ts1
mbcbe.m td, ts1
mbcce.m td, ts1
```

4.5. Arithmetic 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`.

```
# Unigned integer matrix multiplication and add, td = td + ts1 * ts2
mmau.mm   td, ts1, ts2
mwmau.mm  td, ts1, ts2  # output double widen
mqmau.mm  td, ts1, ts2  # output quadruple widen

msmau.mm  td, ts1, ts2  # saturated
mswmau.mm td, ts1, ts2  # saturated and output double widen
msqmau.mm td, ts1, ts2  # saturated and output quadruple widen

# Signed integer matrix multiplication and add, td = td + ts1 * ts2
mma.mm    td, ts1, ts2
mwma.mm   td, ts1, ts2  # output double widen
mqma.mm   td, ts1, ts2  # output quadruple widen

msma.mm   td, ts1, ts2  # saturated
mswma.mm  td, ts1, ts2  # saturated and output double widen
msqma.mm  td, ts1, ts2  # saturated and output quadruple widen

# Float matrix multiplication and add, td = td + ts1 * ts2
mfma.mm   td, ts1, ts2
```

```
mfwma.mm  td, ts1, ts2  # output double widen
```

4.5.2. Element-Wise Add/Sub/Multiply Instructions

Matrix element-wise add/sub/multiply instructions. The input and output matrices are with size `mtilem x mtilen`.

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

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

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

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

# Integer matrix element-wise minimum.
# td[i,j] = min{ts1[i,j], ts2[i,j]}
mminu.mm   td, ts1, ts2, lmul
mmin.mm    td, ts1, ts2, lmul

# Integer matrix element-wise maximum.
# td[i,j] = max{ts1[i,j], ts2[i,j]}
mmaxu.mm   td, ts1, ts2, lmul
mmax.mm    td, ts1, ts2, lmul

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

```
# Saturated integer matrix element-wise multiply.
msmul.mm   td, ts1, ts2, lmul   # signed
msmulu.mm  td, ts1, ts2, lmul   # unsigned
msmulsu.mm td, ts1, ts2, lmul   # signed-unsigned

# Widening integer matrix element-wise multiply.
mwmul.mm   td, ts1, ts2, lmul   # signed
mwmulu.mm  td, ts1, ts2, lmul   # unsigned
mwmulsu.mm td, ts1, ts2, lmul   # signed-unsigned

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

# Float matrix element-wise subtract.
# td[i,j] = ts1[i,j] - ts2[i,j]
mfsub.mm   td, ts1, ts2, lmul
mfwsub.mm  td, ts1, ts2, lmul   # output double widen

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

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

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

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

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

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

4.5.3. Type-Convert Instructions

```
# Convert float to float
mfncvt.f.fw.m  td, ts1, ts2, lmul   # double-width float to single-width float
mfwcvt.fw.f.m  td, ts1, ts2, lmul   # single-width float to double-width float

# Convert integer to float
mfcvt.f.x.m    td, ts1, ts2, lmul   # integer to float

mfncvt.f.xw.m  td, ts1, ts2, lmul   # double-width integer to float
mfncvt.f.xq.m  td, ts1, ts2, lmul   # quad-width integer to float

mfwcvt.fw.x.m  td, ts1, ts2, lmul   # single-width integer to double-width float
mfcvt.fw.xw.m  td, ts1, ts2, lmul   # double-width integer to double-width float
mfncvt.fw.xq.m td, ts1, ts2, lmul   # quad-width integer to double-width float

# Convert float to integer
mfcvt.x.f.m    td, ts1, ts2, lmul   # float to integer

mfwcvt.xw.f.m  td, ts1, ts2, lmul   # float to double-width integer
mfwcvt.xq.f.m  td, ts1, ts2, lmul   # float to quad-width integer

mfncvt.x.fw.m  td, ts1, ts2, lmul   # double-width float to single-width integer
mfcvt.xw.fw.m  td, ts1, ts2, lmul   # double-width float to double-width integer
mfwcvt.xq.fw.m td, ts1, ts2, lmul   # double-width float to quad-width integer
```

4.6. 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
9	msettile	1000	00000000	rs1	111	rd	1110111

No.		**31 26**	25	24 20	19 15	14 12	11 7	6 0
Data Move		funct6	di	rs2	*s1	funct3	*d	opcode
1	mmv.x.s	000000	0	rs2	ts1	101	rd	1110111
2	mmv.s.x	000000	1	rs2	rs1	101	td	1110111
3	mfmv.f.s	000001	0	rs2	ts1	101	rd	1110111
4	mfmv.s.f	000001	1	rs2	rs1	101	td	1110111
5	mbcar.m	000010	0	00001	ts1	101	td	1110111
6	mbcbr.m	000010	0	00010	ts1	101	td	1110111
7	mbccr.m	000010	0	00000	ts1	101	td	1110111
8	mbcac.m	000010	0	00101	ts1	101	td	1110111
9	mbcbc.m	000010	0	00110	ts1	101	td	1110111
10	mbccc.m	000010	0	00100	ts1	101	td	1110111
11	mbcae.m	000010	0	01001	ts1	101	td	1110111
12	mbcbe.m	000010	0	01010	ts1	101	td	1110111
13	mbcce.m	000010	0	01000	ts1	101	td	1110111

No.		**31 26**	25	24 20	19 15	14 12	11 10	9 7	6 0
Load		funct6	ls	rs2	rs1	eew	lmul	td	opcode
1	mlae8.m	000001	0	rs2	rs1	000	lmul	td	1110111
2	mlae16.m	000001	0	rs2	rs1	001	lmul	td	1110111
3	mlae32.m	000001	0	rs2	rs1	010	lmul	td	1110111
4	mlae64.m	000001	0	rs2	rs1	011	lmul	td	1110111
5	mlbe8.m	000010	0	rs2	rs1	000	lmul	td	1110111
6	mlbe16.m	000010	0	rs2	rs1	001	lmul	td	1110111
7	mlbe32.m	000010	0	rs2	rs1	010	lmul	td	1110111
8	mlbe64.m	000010	0	rs2	rs1	011	lmul	td	1110111
9	mlce8.m	000000	0	rs2	rs1	000	lmul	td	1110111
10	mlce16.m	000000	0	rs2	rs1	001	lmul	td	1110111
11	mlce32.m	000000	0	rs2	rs1	010	lmul	td	1110111
12	mlce64.m	000000	0	rs2	rs1	011	lmul	td	1110111

13	mlre8.m	000011	0	rs2	rs1	000	lmul	td	1110111
14	mlre16.m	000011	0	rs2	rs1	001	lmul	td	1110111
15	mlre32.m	000011	0	rs2	rs1	010	lmul	td	1110111
16	mlre64.m	000011	0	rs2	rs1	011	lmul	td	1110111
17	mlate8.m	000101	0	rs2	rs1	000	lmul	td	1110111
18	mlate16.m	000101	0	rs2	rs1	001	lmul	td	1110111
19	mlate32.m	000101	0	rs2	rs1	010	lmul	td	1110111
20	mlate64.m	000101	0	rs2	rs1	011	lmul	td	1110111
21	mlbte8.m	000110	0	rs2	rs1	000	lmul	td	1110111
22	mlbte16.m	000110	0	rs2	rs1	001	lmul	td	1110111
23	mlbte32.m	000110	0	rs2	rs1	010	lmul	td	1110111
24	mlbte64.m	000110	0	rs2	rs1	011	lmul	td	1110111
25	mlcte8.m	000100	0	rs2	rs1	000	lmul	td	1110111
26	mlcte16.m	000100	0	rs2	rs1	001	lmul	td	1110111
27	mlcte32.m	000100	0	rs2	rs1	010	lmul	td	1110111
28	mlcte64.m	000100	0	rs2	rs1	011	lmul	td	1110111
Store		funct6	ls	rs2	rs1	eew	lmul	ts3	opcode
1	msae8.m	000001	1	rs2	rs1	000	lmul	ts3	1110111
2	msae16.m	000001	1	rs2	rs1	001	lmul	ts3	1110111
3	msae32.m	000001	1	rs2	rs1	010	lmul	ts3	1110111
4	msae64.m	000001	1	rs2	rs1	011	lmul	ts3	1110111
5	msbe8.m	000010	1	rs2	rs1	000	lmul	ts3	1110111
6	msbe16.m	000010	1	rs2	rs1	001	lmul	ts3	1110111
7	msbe32.m	000010	1	rs2	rs1	010	lmul	ts3	1110111
8	msbe64.m	000010	1	rs2	rs1	011	lmul	ts3	1110111
9	msce8.m	000000	1	rs2	rs1	000	lmul	ts3	1110111
10	msce16.m	000000	1	rs2	rs1	001	lmul	ts3	1110111
11	msce32.m	000000	1	rs2	rs1	010	lmul	ts3	1110111
12	msce64.m	000000	1	rs2	rs1	011	lmul	ts3	1110111

13	msre8.m	000011	1	rs2	rs1	000	lmul	ts3	1110111
14	msre16.m	000011	1	rs2	rs1	001	lmul	ts3	1110111
15	msre32.m	000011	1	rs2	rs1	010	lmul	ts3	1110111
16	msre64.m	000011	1	rs2	rs1	011	lmul	ts3	1110111
17	msate8.m	000101	1	rs2	rs1	000	lmul	ts3	1110111
18	msate16.m	000101	1	rs2	rs1	001	lmul	ts3	1110111
19	msate32.m	000101	1	rs2	rs1	010	lmul	ts3	1110111
20	msate64.m	000101	1	rs2	rs1	011	lmul	ts3	1110111
21	msbte8.m	000110	1	rs2	rs1	000	lmul	ts3	1110111
22	msbte16.m	000110	1	rs2	rs1	001	lmul	ts3	1110111
23	msbte32.m	000110	1	rs2	rs1	010	lmul	ts3	1110111
24	msbte64.m	000110	1	rs2	rs1	011	lmul	ts3	1110111
25	mscte8.m	000100	1	rs2	rs1	000	lmul	ts3	1110111
26	mscte16.m	000100	1	rs2	rs1	001	lmul	ts3	1110111
27	mscte32.m	000100	1	rs2	rs1	010	lmul	ts3	1110111
28	mscte64.m	000100	1	rs2	rs1	011	lmul	ts3	1110111

No.		**31 26**	25	24 20	19	18	17 15	14 12	11 10	9 7	6 0
Arithmetic		funct6	fp	*s2	sn	sa	ts1	funct3	lmul	td	opcode
1	mmau.mm	000000	0	ts2	0	0	ts1	110	00	td	1110111
2	msmau.mm	000000	0	ts2	0	1	ts1	110	00	td	1110111
3	mma.mm	000000	0	ts2	1	0	ts1	110	00	td	1110111
4	msma.mm	000000	0	ts2	1	1	ts1	110	00	td	1110111
5	mfma.mm	000000	1	ts2	0	0	ts1	110	00	td	1110111
6	mwmau.mm	000001	0	ts2	0	0	ts1	110	00	td	1110111
7	mswmau.mm	000001	0	ts2	0	1	ts1	110	00	td	1110111
8	mwma.mm	000001	0	ts2	1	0	ts1	110	00	td	1110111
9	mswma.mm	000001	0	ts2	1	1	ts1	110	00	td	1110111

10	mfwma.mm	000001	1	ts2	0	0	ts1	110	00	td	1110111
11	mqmau.mm	000010	0	ts2	0	0	ts1	110	00	td	1110111
12	msqmau.mm	000010	0	ts2	0	1	ts1	110	00	td	1110111
13	mqma.mm	000010	0	ts2	1	0	ts1	110	00	td	1110111
14	msqma.mm	000010	0	ts2	1	1	ts1	110	00	td	1110111
15	maddu.mm	000100	0	ts2	0	0	ts1	110	lmul	td	1110111
16	msaddu.mm	000100	0	ts2	0	1	ts1	110	lmul	td	1110111
17	madd.mm	000100	0	ts2	1	0	ts1	110	lmul	td	1110111
18	msadd.mm	000100	0	ts2	1	1	ts1	110	lmul	td	1110111
19	mfadd.mm	000100	1	ts2	0	0	ts1	110	lmul	td	1110111
20	mwaddu.mm	000101	0	ts2	0	0	ts1	110	lmul	td	1110111
21	mwadd.mm	000101	0	ts2	1	0	ts1	110	lmul	td	1110111
22	mfwadd.mm	000101	1	ts2	0	0	ts1	110	lmul	td	1110111
23	msubu.mm	000110	0	ts2	0	0	ts1	110	lmul	td	1110111
24	mssubu.mm	000110	0	ts2	0	1	ts1	110	lmul	td	1110111
25	msub.mm	000110	0	ts2	1	0	ts1	110	lmul	td	1110111
26	mssub.mm	000110	0	ts2	1	1	ts1	110	lmul	td	1110111
27	mfsub.mm	000110	1	ts2	0	0	ts1	110	lmul	td	1110111
28	mwsubu.mm	000111	0	ts2	0	0	ts1	110	lmul	td	1110111
29	mwsub.mm	000111	0	ts2	1	0	ts1	110	lmul	td	1110111
30	mfwsub.mm	000111	1	ts2	0	0	ts1	110	lmul	td	1110111
31	mminu.mm	001000	0	ts2	0	0	ts1	110	lmul	td	1110111
32	mmin.mm	001000	0	ts2	1	0	ts1	110	lmul	td	1110111
33	mfmin.mm	001000	1	ts2	0	0	ts1	110	lmul	td	1110111
34	mmaxu.mm	001001	0	ts2	0	0	ts1	110	lmul	td	1110111
35	mmax.mm	001001	0	ts2	1	0	ts1	110	lmul	td	1110111
36	mfmax.mm	001001	1	ts2	0	0	ts1	110	lmul	td	1110111
38	msmulu.mm	001010	0	rs2	0	1	ts1	110	lmul	td	1110111
37	mmul.mm	001010	0	rs2	1	0	ts1	110	lmul	td	1110111

38	msmul.mm	001010	0	rs2	1	1	ts1	110	lmul	td	1110111
39	mfmul.mm	001010	1	rs2	0	0	ts1	110	lmul	td	1110111
40	mmulhu.mm	001011	0	rs2	0	0	ts1	110	lmul	td	1110111
41	mmulh.mm	001011	0	rs2	1	0	ts1	110	lmul	td	1110111
42	mmulhsu.mm	001100	0	rs2	0	0	ts1	110	lmul	td	1110111
42	msmulsu.mm	001100	0	rs2	0	1	ts1	110	lmul	td	1110111
43	mwmulu.mm	001101	0	rs2	0	0	ts1	110	lmul	td	1110111
44	mwmul.mm	001101	0	rs2	1	0	ts1	110	lmul	td	1110111
45	mwmulsu.mm	001101	0	rs2	1	1	ts1	110	lmul	td	1110111
46	mfwmul.mm	001101	1	rs2	0	0	ts1	110	lmul	td	1110111
47	mfdiv.mm	001110	1	rs2	0	0	ts1	110	lmul	td	1110111
48	mfsqrt.mm	001111	1	0..	0	0	ts1	110	lmul	td	1110111

No.		**31 26**	25	24 20	19 15	14 12	11 10	9 7	6 0
Convert		funct6	fdst	ts2	ts1	funct3	lmul	td	opcode
1	mfncvtc.f.fw.m	010000	1	ts2	ts1	110	lmul	td	1110111
2	mfwcvtc.fw.f.m	010000	0	ts2	ts1	110	lmul	td	1110111
3	mfcvtc.f.x.m	010010	1	ts2	ts1	110	lmul	td	1110111
4	mfcvtc.x.f.m	010010	0	ts2	ts1	110	lmul	td	1110111
5	mfncvtc.f.xw.m	010011	1	ts2	ts1	110	lmul	td	1110111
6	mfwcvtc.xw.f.m	010011	0	ts2	ts1	110	lmul	td	1110111
7	mfncvtc.f.xq.m	010100	1	ts2	ts1	110	lmul	td	1110111
8	mfwcvtc.xq.f.m	010100	0	ts2	ts1	110	lmul	td	1110111
9	mfwcvtc.fw.x.m	010101	1	ts2	ts1	110	lmul	td	1110111
10	mfncvtc.x.fw.m	010101	0	ts2	ts1	110	lmul	td	1110111
11	mfcvtc.fw.xw.m	010110	1	ts2	ts1	110	lmul	td	1110111
12	mfcvtc.xw.fw.m	010110	0	ts2	ts1	110	lmul	td	1110111
13	mfncvtc.fw.xq.m	010111	1	ts2	ts1	110	lmul	td	1110111
14	mfwcvtc.xq.fw.m	010111	0	ts2	ts1	110	lmul	td	1110111

Chapter 5. Intrinsic Examples

5.1. Matrix multiplication

```
void matmul_float16(c, a, b, m, k, n) {
    msettype(e16, m1);                          // use 16bit input matrix element
    for (i = 0; i < m; i += mtilem) {           // loop at dim m with tiling
        mtilem = msettile_m(m-i);
        for (j = 0; j < n; j += mtilen) {       // loop at dim n with tiling
            mtilen = msettile_n(n-j);

            out = mwsub_mm(out, out, m1)        // clear output reg
            for (s = 0; s < k; s += mtilek) {   // loop at dim k with tiling
                mtilek = 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
                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
        }
    }
}
```

5.2. Matrix multiplication with left matrix transposed

```
void matmul_a_tr_float16(c, a, b, m, k, n) {
    msettype(e16, m1);                          // use 16bit input matrix element
    for (i = 0; i < m; i += mtilem) {           // loop at dim m with tiling
        mtilem = msettile_m(m-i);
        for (j = 0; j < n; j += mtilen) {       // loop at dim n with tiling
            mtilen = msettile_n(n-j);

            out = mwsub_mm(out, out, m1)        // clear output reg
            for (s = 0; s < k; s += mtilek) {   // loop at dim k with tiling
                mtilek = 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
                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, m1);                          // use 16bit input matrix element

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

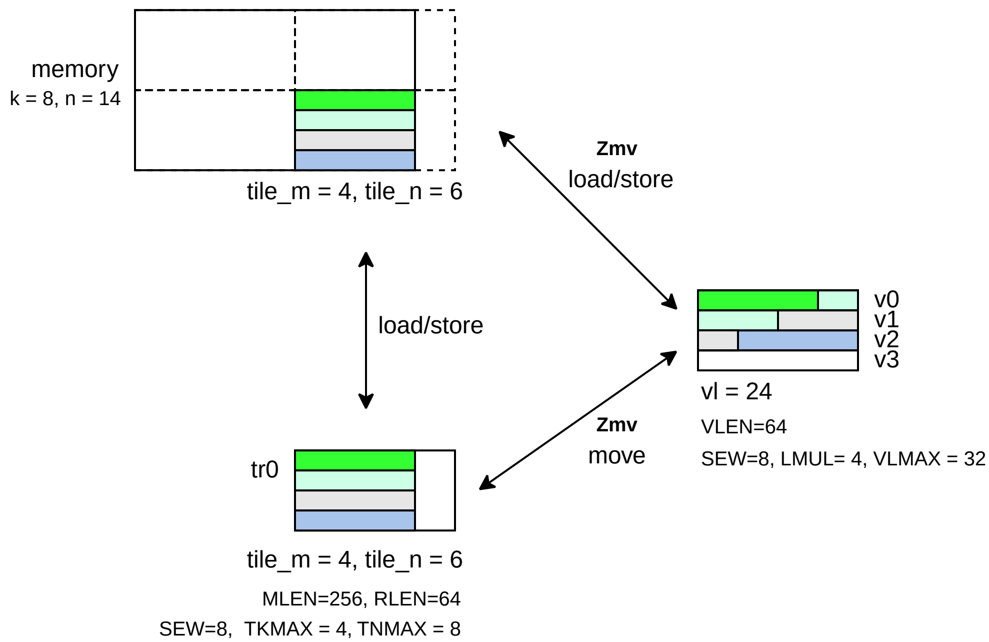
Chapter 6. Standard Matrix Extensions

6.1. 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.



6.1.1. 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 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
```

6.1.2. 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 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
```

6.1.3. 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] = td[i, j], i = rs2 .. rs2 + lmul - 1
mmvar.v.m  vd, ts1, rs2
mmvbr.v.m  vd, ts1, rs2
mmvcr.v.m  vd, ts1, rs2

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

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

# vd[(j - rs2) * tile_m + i] = td[i, j], j = rs2 .. rs2 + lmul - 1
mmvac.v.m  vd, ts1, rs2
mmvbc.v.m  vd, ts1, rs2
mmvcc.v.m  vd, ts1, rs2

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

6.1.4. Instruction Listing

No.		31 26	25	24 20	19 15	14 12	11 7	6 0
Load		funct6	ls	rs2	rs1	eew	td	opcode
1	mlae8.v	100001	0	rs2	rs1	000	td	1110111
2	mlae16.v	100001	0	rs2	rs1	001	td	1110111
3	mlae32.v	100001	0	rs2	rs1	010	td	1110111
4	mlae64.v	100001	0	rs2	rs1	011	td	1110111
5	mlbe8.v	100010	0	rs2	rs1	000	td	1110111
6	mlbe16.v	100010	0	rs2	rs1	001	td	1110111
7	mlbe32.v	100010	0	rs2	rs1	010	td	1110111
8	mlbe64.v	100010	0	rs2	rs1	011	td	1110111
9	mlce8.v	100000	0	rs2	rs1	000	td	1110111
10	mlce16.v	100000	0	rs2	rs1	001	td	1110111
11	mlce32.v	100000	0	rs2	rs1	010	td	1110111
12	mlce64.v	100000	0	rs2	rs1	011	td	1110111
Store		funct6	ls	rs2	rs1	eew	ts3	opcode
13	msae8.v	100001	1	rs2	rs1	000	ts3	1110111
14	msae16.v	100001	1	rs2	rs1	001	ts3	1110111
15	msae32.v	100001	1	rs2	rs1	010	ts3	1110111
16	msae64.v	100001	1	rs2	rs1	011	ts3	1110111
17	msbe8.v	100010	1	rs2	rs1	000	ts3	1110111

No.		31 26	25	24 20	19 15	14 12	11 7	6 0
18	msbe16.v	100010	1	rs2	rs1	001	ts3	1110111
19	msbe32.v	100010	1	rs2	rs1	010	ts3	1110111
20	msbe64.v	100010	1	rs2	rs1	011	ts3	1110111
21	msce8.v	100000	1	rs2	rs1	000	ts3	1110111
22	msce16.v	100000	1	rs2	rs1	001	ts3	1110111
23	msce32.v	100000	1	rs2	rs1	010	ts3	1110111
24	msce64.v	100000	1	rs2	rs1	011	ts3	1110111
Data Move		funct6	v2m	rs2	*s1	funct3	*d	opcode
25	mmvar.v.m	000101	0	rs2	ts1	101	vd	1110111
26	mmvar.m.v	000101	1	rs2	vs1	101	td	1110111
27	mmvbr.v.m	000110	0	rs2	ts1	101	vd	1110111
28	mmvbr.m.v	000110	1	rs2	vs1	101	td	1110111
29	mmvcr.v.m	000100	0	rs2	ts1	101	vd	1110111
30	mmvcr.m.v	000100	1	rs2	vs1	101	td	1110111
31	mmvac.v.m	001001	0	rs2	ts1	101	vd	1110111
32	mmvac.m.v	001001	1	rs2	vs1	101	td	1110111
33	mmvbc.v.m	001010	0	rs2	ts1	101	vd	1110111
34	mmvbc.m.v	001010	1	rs2	vs1	101	td	1110111
35	mmvcc.v.m	001000	0	rs2	ts1	101	vd	1110111
36	mmvcc.m.v	001000	1	rs2	vs1	101	td	1110111

6.1.5. Intrinsic Examples: Matrix multiplication fused with element-wise vector operation

```
void fused_matmul_relu_float16(c, a, b, m, k, n) {
    msettype(e16, m1);                          // 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);

            out = mfemul_mf(out, 0.f, m2)       // 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
                out = mfwma_mm(tr1, tr2);       // tiled matrix multiply,
                                                // double widen output
            }

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


            for (s = 0; s < tile_m; s += rows) {
                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(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
            }
        }
    }
}
```

6.2. Zmbf16: Matrix Bfloat16(BF16) Extension

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

The Zmbf16 extension adds a bit `mtype[7]` in `mtype` register.

Table 7. `mtype` register layout

Bits	Name	Description
XLEN-1	mill	Illegal value if set.
XLEN-2:8	0	Reserved if non-zero.
7	**mbf16**	Support BFloat16 format.
6	mfp64	Support 64-bit float point.
5	mba	Matrix out of bound agnostic.
4:2	msew[2:0]	Selected element width (SEW) setting.
1:0	mlmul[1:0]	Register group multiplier (LMUL) 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 adds a bit `mtype[8]` in `mtype` register.

Table 8. `mtype` register layout

Bits	Name	Description
XLEN-1	mill	Illegal value if set.
XLEN-2:9	0	Reserved if non-zero.
8	**mtf32**	Support TensorFloat32 format.
7	mbf16	Support BFloat16 format.
6	mfp64	Support 64-bit float point.
5	mba	Matrix out of bound agnostic.
4:2	msew[2:0]	Selected element width (SEW) setting.
1:0	mlmul[1:0]	Register group multiplier (LMUL) 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. Zmfp8: Matrix 8-bit Float Point Extension

The Zmfp8 extension allows to use 8-bit float point format as the data type of input matrix elements.

The Zmfp8 extension adds a bit `mtype[9]` in `mtype` register.

Table 9. `mtype` register layout

Bits	Name	Description
XLEN-1	mill	Illegal value if set.
XLEN-2:10	0	Reserved if non-zero.
9	**mfp8**	Support 8-bit float point format.
8	mtf32	Support TensorFloat32 format.
7	mbf16	Support BFloat16 format.
6	mfp64	Support 64-bit float point.
5	mba	Matrix out of bound agnostic.
4:2	msew[2:0]	Selected element width (SEW) setting.
1:0	mlmul[1:0]	Register group multiplier (LMUL) setting.

The Zmfp8 extension adds another unprivileged CSR (mfcsr) to the base scalar RISC-V ISA.

Table 10. New matrix CSR

Address	Privilege	Name	Description
0xXXX	URW	mfcsr	Matrix float point control and status.

Table 11. `mfcsr` register layout

Bits	Name	Description
XLEN-1:4	0	Reserved if non-zero.
3:1	mftype[2:0]	The types of ts1, ts2 and td.
0	mfsat	If to saturate the FP8 result.

The CSR `mfcsr` is valid when `mtype.mfp8=1`.

The bits mftype[0], mftype[1] and mftype[2] specify the type of ts1, ts2 and td, respectively.

Table 12. `mftype` bit

Value	Type
0	E4M3, with 4-bit exponent and 3-bit mantissa.
1	E5M2, with 5-bit exponent and 2-bit mantissa.

If `mfsat=1`, the result of an FP8 operation will be saturated to the maximum value of the corresponding type. Otherwise, the result will be set to NaN and Infinity for E4M3 and E5M2, respectively.

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

```
Suggested fp8 assembler name used for msettypei mtypei immediate

    fp8   # Use FP8 format

Examples:

    msettypei t0, e8, fp8           # SEW = 8, use FP8 as input matrix element
```

For implemention not support FP8 format, `mtype.mill` will be set.

`fp8` should be always used with `e8`(SEW=8), otherwise `mtype.mill` will be set.

The float-point matrix multiplication and add instructions, `mfma.mm` and `mfwma.mm`, are reused for FP8 format. A quad-widen instruction is added for quad-widening matrix multiplication and add for FP8 format.

```
# Float matrix multiplication and add, td = td + ts1 * ts2
mfma.mm   td, ts1, ts2
mfwma.mm  td, ts1, ts2  # output double widen
mfqma.mm  td, ts1, ts2  # output quadruple widen
```

The element-wise instructions are not available under FP8 format.

6.4.1. Instruction Listing

No.		**31 26**	25	24 20	19	18	17 15	14 12	11 10	9 7	6 0
Arithmetic		funct6	fp	*s2	sn	sa	ts1	func t3	lmul	td	opcode
1	mfqma.mm	000010	1	ts2	0	0	ts1	110	00	td	1110111

6.5. Zmi4: Matrix 4-bit Integer (INT4) 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[10]` in `mtype` register.

Table 13. `mtype` register layout

Bits	Name	Description
XLEN-1	mill	Illegal value if set.
XLEN-2:11	0	Reserved if non-zero.
10	**mint4**	Support 4-bit integer.
9	mfp8	Support 8-bit float point format.
8	mtf32	Support TensorFloat32 format.
7	mbf16	Support BFloat16 format.
6	mfp64	Support 64-bit float point.
5	mba	Matrix out of bound agnostic.
4:2	msew[2:0]	Selected element width (SEW) setting.
1:0	mlmul[1:0]	Register group multiplier (LMUL) setting.

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

```
Suggested int4 assembler name used for msettypei mtypei immediate

    int4   # Use INT4 format

Examples:
```

```
msettypei t0, e8, int4          # SEW = 8, use INT4 as input matrix element
```

For implemention not support INT4 format, `mtype.mill` will be set.

`int4` should be always used with `e8`(SEW=8), otherwise `mtype.mill` will be set. Two 4-bit values are combined to a 8-bit element in tile register. So the size of a row must be even.

The integer matrix multiplication and add instructions, both unsigned one and signed one, are reused for INT4 format.

The element-wise instructions are not available under INT4 format.

The Zmbf16, Zmtf32, Zmfp8, Zmi4 extensions can be implemented with any combination.

6.6. 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`.

6.6.1. CSRs

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

Table 14. 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.
0xXXX	URO	moutsk	Fold/unfold sliding kernel position of output.
0xXXX	URO	mpadval	Fold/unfold padding value, default to zero.

Table 15. `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 16. `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 17. `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 18. `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

6.6.2. 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
```

6.6.3. 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`.

Unfolded load and folded store only support LMUL=1. Other LMUL settings will be ignored.

```
# 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
```

6.6.4. 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`.

```
# 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
msfdbe8.m    ts3, (rs1), rs2
```

```
msfdbe16.m   ts3, (rs1), rs2
msfdbe32.m   ts3, (rs1), rs2
msfdbe64.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
```

6.6.5. 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
1	msetoutsh	1000	000	rs2	rs1	111	rd	1110111
2	mseindsh	1001	000	rs2	rs1	111	rd	1110111
3	msetsk	1010	000	rs2	rs1	111	rd	1110111
4	msetpadval	1011	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	td	opcode
1	mlufae8.m	110001	0	rs2	rs1	000	td	1110111
2	mlufae16.m	110001	0	rs2	rs1	001	td	1110111
3	mlufae32.m	110001	0	rs2	rs1	010	td	1110111
4	mlufae64.m	110001	0	rs2	rs1	011	td	1110111
5	mlufbe8.m	110010	0	rs2	rs1	000	td	1110111
6	mlufbe16.m	110010	0	rs2	rs1	001	td	1110111
7	mlufbe32.m	110010	0	rs2	rs1	010	td	1110111
8	mlufbe64.m	110010	0	rs2	rs1	011	td	1110111
9	mlufce8.m	110000	0	rs2	rs1	000	td	1110111
10	mlufce16.m	110000	0	rs2	rs1	001	td	1110111
11	mlufce32.m	110000	0	rs2	rs1	010	td	1110111
12	mlufce64.m	110000	0	rs2	rs1	011	td	1110111
Store		funct6	ls	rs2	rs1	eew	ts3	opcode

13	msfdae8.m	110001	1	rs2	rs1	000	ts3	1110111
14	msfdae16.m	110001	1	rs2	rs1	001	ts3	1110111
15	msfdae32.m	110001	1	rs2	rs1	010	ts3	1110111
16	msfdae64.m	110001	1	rs2	rs1	011	ts3	1110111
17	msfdbe8.m	110010	1	rs2	rs1	000	ts3	1110111
18	msfdbe16.m	110010	1	rs2	rs1	001	ts3	1110111
19	msfdbe32.m	110010	1	rs2	rs1	010	ts3	1110111
20	msfdbe64.m	110010	1	rs2	rs1	011	ts3	1110111
21	msfdce8.m	110000	1	rs2	rs1	000	ts3	1110111
22	msfdce16.m	110000	1	rs2	rs1	001	ts3	1110111
23	msfdce32.m	110000	1	rs2	rs1	010	ts3	1110111
24	msfdce64.m	110000	1	rs2	rs1	011	ts3	1110111

6.6.6. 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, m1);      // 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);

            out = mwsub_mm(out, out, m1)            // 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 = 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
                        out = mfwma_mm(tr1, tr2);   // tiled matrix multiply,
                                                    // double widen output
                    }
                }
            }

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

6.6.7. 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, m1);      // 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);

```

```
            out = mwsub_mm(out, out, m1)            // 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 = 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
                            out = mfwma_mm(tr1, tr2);   // tiled matrix multiply,
                                                        // double widen output
                        }
                    }
                }
            }

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

6.6.8. 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, m1);      // 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);

            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
        }
    }
}
```

6.6.9. 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, m1);      // 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 = 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);

            tr_out = mwsub_mm(tr_out, tr_out, m1)   // 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 = mfdiv_mm(tr_out, tr_div);
            msce16_m(tr_out, &out[i][j], n*2);      // store to matrix c
        }
    }
}
```

6.7. Zmsp: Matrix Sparsity Extension

Work in progress.

Bibliography