Appendix

A RISC-V Tensor Extension Instruction Set

The tensor extension instructions introduce a set of specialized instructions to accelerate tensor-based computations, commonly used in AI workloads. These instructions expand the basic RISC-V instruction set by incorporating 32-bit and 64-bit formats that map to the Custom-2 instruction format, enabling efficient handling of tensor operations. The instructions include various types, such as configuration instructions, memory operations, computational instructions, and synchronization primitives. By integrating these tensor-specific operations into the instruction pipeline, significant performance gains are achieved in AI computation tasks such as matrix multiplication, convolution, and pooling. The structure and encoding of these instructions are carefully designed to balance flexibility with efficiency, ensuring that both standard scalar and vector operations as well as advanced tensor manipulations can be executed with minimal overhead. In the following sections, we detail the instruction formats and functions.

A.0.1 Instruction Formats. The tensor extension instructions encompass both 32-bit and 64-bit formats, which are mapped to the Custom-2 instruction format in the RISC-V architecture, as shown in Fig. 21. These specialized formats enable efficient execution of tensor operations by extending the basic instruction set. The 32-bit format is used for simpler tensor operations, while the 64-bit format supports more complex operations, allowing for higher precision and larger data throughput. Using the Custom-2 format, these instructions are seamlessly integrated into the RISC-V pipeline, maintaining compatibility with existing scalar and vector operations while enabling enhanced tensor processing capabilities. This design facilitates the efficient handling of AI-specific computations, such as matrix multiplication, convolution, and pooling, directly within the instruction stream.

A.0.2 CSR, CR, TVR, and GVR Configuration Instructions. These instructions are responsible for configuring CSRs, CRs, TVRs, and GVRs as shown in Figs. 22-23. Due to limitations in the bit width of the extended instructions, many details required by LS, MOVE, and computation instructions are pre-loaded into these registers via configuration instructions. During instruction decoding, the IFE reads the information stored in these registers and encapsulates it along with the decoded instruction. The encapsulated instruction is then sent to the execution unit for processing. This approach ensures that the necessary configuration data are readily available for tensor operations, optimizing the overall performance of the system by minimizing delays associated with instruction execution and register setup.

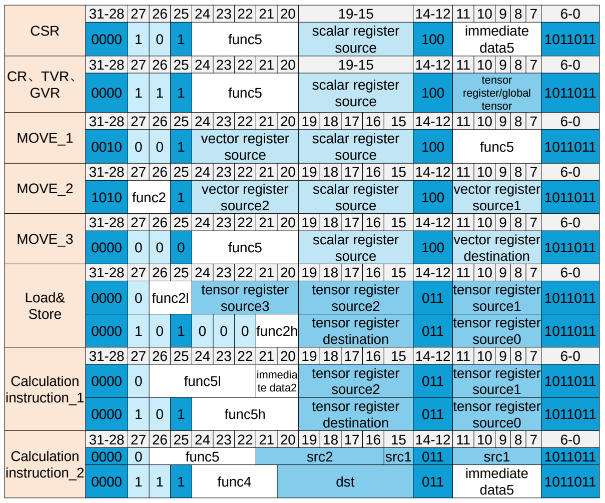
A.0.3 MOVE Instructions. The MOVE instructions facilitate data transfer between the Vector Register File (VRF) and LMEM as shown in Fig. 24. They support the transfer of one or two sets of VRF data, which can be either contiguous or scattered across LMEM's address space. Due to the inherent difference in the mapping of physical storage between LMEM and TR, the MOVE instructions indirectly enable data exchange between the VRF and TR.

A.0.4 LOAD & STORE Instructions. The LOAD & STORE (LS) instructions are responsible for managing the data transfer between TR and DRAM as shown in Figs. 25-26. These instructions support efficient tensor data movement with enhanced capabilities, such as performing transpose and broadcasting along the C dimension during data transfer. In addition, they facilitate data replication between TR and GT or between different TR regions, with support for simultaneous transpose, dimension reversal, and C-dimension broadcasting. The instructions also include advanced features such as Mask Select and Non-Zero Index, allowing for selective data retrieval based on specific conditions. Furthermore, they enable Scatter and Gather operations along the H dimension, offering flexible data manipulation.

A.0.5 COMPUTE Instructions. The Compute instructions are designed to perform tensor computations within the TR as shown in Fig. 27-29. Building upon basic arithmetic operations such as integer and floating-point arithmetic, data-type conversion, and specialized operations, we introduce several AI-specific instructions to accelerate deep learning workloads. These include convolution and CUBE matrix multiplication instructions, which are essential for many AI algorithms, as well as pooling instructions that support Max Pooling, Average Pooling, Region of Interest (RoI) Pooling, and Depth-wise Convolution. Additionally, the Re-Quantization (RQ) and Dequantization (DQ) instructions cater to AI algorithms' precision adjustments, while the fully connected matrix multiplication instructions enable efficient processing of the Full Connect layer in neural networks.

These specialized Compute instructions significantly reduce the instruction density during the execution of the AI model, thus optimizing both the computation and the memory access efficiency. Furthermore, the Compute instructions include powerful data manipulation features for TR registers, such as broadcasting, transposition, Scatter and Gather operations across one or two dimensions, as well as Mask Select and Non-Zero Index functionalities. These features not only improve computation efficiency but also minimize TR register read/write overhead, enabling faster data processing for AI algorithms.

A.0.6 Synchronization Instructions. Synchronization instructions are used to coordinate the execution of tensor extension instructions both within a single RTPU and across multiple RTPUs as shown in Fig. 30. These instructions ensure that operations are executed in the correct sequence, enabling efficient parallelism and scalability.



	31-28	27	26	25	24	23	22	21	20	19	18	17	16	15	14-12	11	10	9	8	7	6-0
CSR	0000	1	0	1	func5					scalar register source					100	immediate data5					1011011
CR、TVR、GVR	0000	1	1	1	func5					scalar register source					100	tensor register/global tensor					1011011
MOVE_1	0010	0	0	1	vector register source					scalar register source					100	func5					1011011
MOVE_2	1010	func2		1	vector register source2					scalar register source					100	vector register source1					1011011
MOVE_3	0000	0	0	0	func5					scalar register source					100	vector register destination					1011011
Load& Store	0000	0	func2l		tensor register source3					tensor register source2					011	tensor register source1					1011011
	0000	1	0	1	0	0	0	func2h		tensor register destination					011	tensor register source0					1011011
Calculation instruction_1	0000	0	func5l					immediate data2		tensor register source2					011	tensor register source1					1011011
	0000	1	0	1	func5h					tensor register destination					011	tensor register source0					1011011
Calculation instruction_2	0000	0	func5					src2						src1	011	src1					1011011
	0000	1	1	1	func4				dst						011	immediate data5					1011011

Figure 21: Tensor Extension Instruction Encoding Format.

```
cfg.satu       rs
//Configure whether the results of integer calculation
      instructions are saturated and whether symmetric
      saturation is adopted.
cfg.round_mode rs
// Configure the rounding mode during floating-point
      precision conversion or integer right shift
cfg.rsqrt_iter rs
// Configure the number of Newton iterations for rsqrt
      and fdiv instructions
cfg.quant       ts
// Configure the quantization method, supporting per-
      channel quantization and per-tensor quantization
cfg.pad         rs
// Configure the top, bottom, left, and right padding of
      the feature map for convolution and pooling
      instructions
cfg.insrt       rs
// Configure the interpolation of the feature map or
      kernel for convolution and pooling instructions
cfg.stencil     rs
// Configure the sliding window, stride, whether the
      kernel is rotated, and whether the result is passed
      through ReLU for convolution and pooling
      instructions
cfg.kzp         ts
// Configure the zero point of the kernel for convolution
       and matrix multiplication instructions
cfg.dmaidx      rs
// Configure parameters such as the initial value of the
      index or the default write-back value for
      instructions like ls.hscatter, ls.hgather, and ls.
      nzidx
```

Figure 22: CSR Configuration Instructions

```
cfgcr          ca, rs
// Configure the data type and constant value of CR
cfgtr          ta, rs
// Configure the data type and offset address of TR
cfgtr.shape     ta, rs
// Configure TR as N, C, H, W
cfgtr.hwstride  ta, rs
// Configure the H stride and W stride of TR
cfgtr.ncstride  ta, rs
// Configure the N stride and C stride of TR
cfggt           ga, rs
// Configure the data type and address of GT
cfggt.shape     ga, rs
// Configure GT as N, C, H, W
cfggt.hwstride   ga, rs
// Configure the H stride and W stride of GT
cfggt.ncstride   ga, rs
// Configure the N stride and C stride of GT
```

Figure 23: CR, TVR, GVR Configuration Instructions

```
mov.t.v          vs, (rs)
// Load a set of VRF data into LMEM; rs specifies the
      starting relative address in LMEM
mov.dist.v        vs, (rs)
// Distribute a set of VRF data evenly to all LMEMs; rs
      specifies the starting relative address in LMEM
mov.t.vv          vs2, vs1, (rs)
// Load two sets of VRF data into LMEM; rs specifies the
      starting relative address in LMEM
mov.dist.vv      vs2, vs1, (rs)
// Distribute two sets of VRF data evenly to all LMEMs;
     rs specifies the starting relative address in LMEM
mov.v.t            vd, (rs)
// Store data from LMEM into a set of VRF; rs specifies
      the starting relative address in LMEM
mov.v.coll       vd, (rs)
// Collect data from the same relative offset in each
      LMEM and store it into a set of VRF; rs specifies
      the starting relative address in LMEM
```

Figure 24: MOVE Instructions

```
////Data Copy
ls.cp           dst, src
// Copy the source GT or TR to the destination GT or TR
ls.cpbc         dst, src
// Copy the source GT or TR(n, 1, h, w) to the
      destination TR(n, c, h, w)
ls.cpt          dst, src
// Copy the source GT or TR(n, c, h, w) to the
      destination GT or TR(c, n, h, w)
ls.cpr          dst, src, _dim
// Copy the source GT or TR(n, 1, h, w) to the
      destination GT or TR(n, c, h, w) while reversing the
       elements along a specified dimension
```

Figure 25: LOAD & STORE Instructions I

```
////Tensor Load and Store Instructions
ls.ld        td, gs
// Load the source GT into the destination TR. The shapes
     of the source and destination tensors may differ,
     but the total number of elements must remain the
     same
ls.ldt       td, gs
// Load the source GT(c, n, h, w) into the destination TR
     (n, c, h, w), transposing the N and C dimensions
     during loading
ls.ldbc      td, gs
// Load the source GT(n, 1, h, w) into the destination TR
     (n, c, h, w), broadcasting in the C dimension during
      loading
ls.st        gd, ts
// Store the source TR into the destination GT. The
     shapes of the source and destination tensors may
     differ, but the total number of elements must remain
      the same
ls.stt       gd, ts
// Store the source TR(n, c, h, w) into the destination
     GT(c, n, h, w), transposing the N and C dimensions
     during storing
////Matrix Load and Store Instructions
ls.mld       td, gs
// Load the source GT(1, 1, m, n) into the destination TR
     (m, ROUNDUP(n/CU_NUM), 1, w)
ls.mst       gd, ts
// Store the source TR(m, ROUNDUP(n/CU_NUM), 1, w) into
     the destination GT(1, 1, m, n)
//// Mask & Index Instructions
ls.masksel    dst, src, mask, _bigmask
// Select elements from the source tensor at positions
     where the mask tensor has a value of 1, and output
     them to the destination GT. The mask and source
     tensor can be either GT or TR; the mask tensor is of
      integer type
ls.fmasksel   dst, src, mask, _bigmask
// Select elements from the source tensor at positions
     where the mask tensor has a value of 1, and output
     them to the destination GT. The mask and source
     tensor can be either GT or TR; the mask tensor is of
      floating-point type
ls.nzidx      dst, src
// Treat the source tensor as a one-dimensional vector,
     generate indices of non-zero elements, and output
     them to the destination GT; the elements of the
     source tensor are integers
ls.fnzidx     dst, src
// Treat the source tensor as a one-dimensional vector,
     generate indices of non-zero elements, and output
     them to the destination GT; the elements of the
     source tensor are floating-point numbers
//// Gather & Scatter Instructions
ls.hgather    dst, src, idx, _bigh
// Using elements in the index tensor(1,1,h,1) or (1,c,h
     ,1) as coordinates, gather data from the source
     tensor(1,1,ih,1) or (1,c,ih,1) into the destination
     tensor(1,1,h,1) or (1,c,h,1)
ls.hscatter   dst, src, idx, _bigh, _accu
// Sequentially read data from the source tensor(1,1,ih
     ,1) or (1,c,ih,1), and scatter it to the destination
      tensor(1,1,h,1) or (1,c,h,1) using elements in the
     index tensor(1,1,ih,1) or (1,c,ih,1) as coordinates,
      supporting accumulation into an integer destination
      tensor
ls.fhscatter  dst, src, idx, _bigh, _accu
// Sequentially read data from the source tensor(1,1,ih
     ,1) or (1,c,ih,1), and scatter it to the destination
      tensor(1,1,h,1) or (1,c,h,1) using elements in the
     index tensor(1,1,ih,1) or (1,c,ih,1) as coordinates,
      supporting accumulation into a floating-point
     destination tensor
```

Figure 26: LOAD & STORE Instructions II

```
////Integer Arithmetic Instructions
add      out, a, b, shift, _satu
// Integer addition, supporting shifting and saturation
sub      out, a, b, shift, _satu
// Integer subtraction, supporting shifting and
     saturation
mul      out, a, b, shift, _satu
// Integer multiplication, supporting shifting and
     saturation
abs      out, a
// Integer absolute value
mac      out, a, b, shift
// Integer multiply-accumulate, supporting shifting and
     saturation
max      out, a, b
// Integer maximum value
min      out, a, b
// Integer minimum value
selgt    out, a, b, c
// If integer a > b, then out = c; otherwise, 0
seleq    out, a, b, c
// If integer a == b, then out = c; otherwise, 0
sellt    out, a, b, c
// If integer a < b, then out = c; otherwise, 0
cmplt    out, a, b, c, d
// Integer: if a < b then c, else d
cmpeq    out, a, b, c, d
// Integer: if a == b then c, else d
cmpgt    out, a, b, c, d
// Integer: if a > b then c, else d
and      out, a, b
// Logical AND
xor      out, a, b
// Logical XOR
or       out, a, b
// Logical OR
not      out, a
// Logical NOT
lshr     out, a, shift
// Logical right shift
ashr     out, a, shift
// Arithmetic right shift
rshr     out, a, shift
// Rotate right shift
clz      out, a
// Count the number of leading zeros
clo      out, a
// Count the number of leading ones
//// Floating-Point Arithmetic Instructions
fadd     out, a, b
// Floating-point addition with optional saturation
fsub     out, a, b
// Floating-point subtraction with optional saturation
fmul     out, a, b
// Floating-point multiplication with optional saturation
fdiv     out, a, b
// Floating-point division with optional saturation
fabs     out, a
// Floating-point absolute value
fmac     out, a, b
// Floating-point multiply-accumulate
fmax     out, a, b
// Floating-point maximum value
fmin     out, a, b
// Floating-point minimum value
fselgt   out, a, b, c
// If floating-point a > b then out = c, otherwise 0
fseleq   out, a, b, c
// If floating-point a == b then out = c, otherwise 0
fsellt   out, a, b, c
// If floating-point a < b then out = c, otherwise 0
fcmplt   out, a, b, c, d
// Floating-point: if a < b then out = c, else out = d
fcmpeq   out, a, b, c, d
// Floating-point: if a == b then out = c, else out = d
fcmpgt   out, a, b, c, d
// Floating-point: if a > b then out = c, else out = d
```

Figure 27: Compute Instructions I

```
//// Data Type Conversion Instructions
cvt.i2i         out, a
// Integer precision conversion
cvt.i2f         out, a
// Convert integer to float
cvt.f2i         out, a
// Convert float to integer
cvt.f2f         out, a
// Floating-point precision conversion
//// SFU Instructions
sfu.norm        out, a
// Extract exponent part of float and convert to integer
sfu.taylor      out, a, coeff
// Polynomial evaluation
sfu.rsqrt       out, a
// Reciprocal square root
//// Quantization/Dequantization Instructions
dq0             out, a, scale
// The first dequantization: scale and convert to
     floating-point
rq0             out, a, scale
// The first quantization: scale and convert to low-
     precision integer
dq1             out, a, scale
// The second dequantization: scale and convert to high-
     precision integer
rq1             out, a, scale
// The second quantization: scale and convert to low-
     precision integer
dq2             out, a, scale, _gsize
// The third dequantization: scale and convert to half-
     precision floating-point
//// Pooling Instructions
pool.avg        out, x, w, _rq
// Integer 2D average pooling
pool.max        out, x
// Integer 2D max pooling
pool.min        out, x
// Integer 2D min pooling
pool.favg       out, x, w
// Floating-point 2D average pooling
pool.fmax       out, x
// Floating-point 2D max pooling
pool.fmin       out, x
// Floating-point 2D min pooling
roipool.avg     out, x, w, roi, _rq
// Integer ROI average pooling
roipool.favg    out, x, w, roi
// Floating-point ROI average pooling
roipool.max     out, x, roi
// Integer ROI max pooling
roipool.fmax    out, x, roi
// Floating-point ROI max pooling
roipool.min     out, x, roi
// Integer ROI min pooling
roipool.fmin    out, x, roi
// Floating-point ROI min pooling
dwconv          out, x, w, bias, _rq
// Integer depthwise convolution
fdwconv         out, x, w, bias
//Floating-point depthwise convolution
//// Convolution Instructions
conv            out, x, w, bias, _rq
// Integer convolution
conva           out, x, w, bias
// Integer convolution accumulate
fconv           out, x, w, bias
// Floating-point convolution
fconva          out, x, w, bias
// Floating-point convolution accumulate
//// CUBE Matrix Multiplication Instructions
mm.<nn|nt|tt>   out, x, w, bias, _rq, _relu
// Integer matrix multiplication
mma.<nn|nt|tt>  out, x, w, bias, _relu
// Integer matrix multiplication accumulate
fmm.<nn|nt|tt>  out, x, w, bias, _relu
// Floating-point matrix multiplication
fmma.<nn|nt|tt> out, x, w, bias, _relu
//Floating-point matrix multiplication accumulate
```

Figure 28: Compute Instructions II

```
//// Fully Connected Matrix Multiplication Instructions
fcmm.<nn|tn>   out, x, w, bias, _rq, _relu
// Integer matrix multiplication for fully connected
     layers
fcmma.<nn|tn>  out, x, w, bias, _relu
// Accumulated integer matrix multiplication for fully
     connected layers
ffcmm.<nn|tn>  out, x, w, bias, _relu
// Floating-point matrix multiplication for fully
     connected layers
ffcmma.<nn|tn> out, x, w, bias, _relu
// Accumulated floating-point matrix multiplication for
     fully connected layers
////Cross Comparison Instructions
fvcmax         out, a, b
// Element-wise cross comparison of two floating-point
     vectors to select the maximum value
fvcmin         out, a, b
// Element-wise cross comparison of two floating-point
     vectors to select the minimum value
vcmax          out, a, b
// Element-wise cross comparison of two integer vectors
     to select the maximum value
vcmin          out, a, b
// Element-wise cross comparison of two integer vectors
     to select the minimum value
////Data Copy and Reordering Instructions
cp             out, a
// Tensor copy instruction.
bc             out, a
// Broadcast: replicate a(n, 1, h, w) along the C
     dimension to form out(n, c, h, w)
cwtrans        out, a
// CW-dimension transpose instruction. Transposes the C
     and W dimensions of the input floating-point Tensor
     A and outputs the result to Tensor out.
wctrans        out, a
// WC-dimension transpose instruction. Transpose the W
     and C dimensions of the input floating-point Tensor
     A and outputs the result to Tensor out.
gather.pc      out, a, idx, cs, _bdlimit
// Per-channel gather instruction. Gather data from the W
      dimension of Tensor A into Tensor out based on the
     channel-shared index.
scatter.pc     out, a, idx
// Per-channel scatter instruction. Scatter data from the
      W dimension of Tensor A into Tensor out based on
     the channel-shared index.
gather2d.pc    out, a, idx, cs
// 2D gather instruction. Gather data from the H and W
     dimensions of Tensor A into Tensor out based on 2D
     coordinates shared per channel.
scatter2d.pc   out, a, idx
// 2D scatter instruction. Scatter data from the H and W
     dimensions of Tensor A into Tensor out based on 2D
     coordinates shared per channel.
gather         out, a, idx, cs
// Gather instruction. Gather data from the W dimension
     of Tensor A into Tensor out based on the provided
     index.
scatter        out, a, idx
// Scatter instruction. Scatter data from the W dimension
      of Tensor A into Tensor out based on the provided
     index.
hgatter        out, a, idx, cs
// H-dimension gather instruction. Gather data from the H
      dimension of Tensor A into Tensor out based on the
     provided index.
hscatter       out, a, idx
// H-dimension scatter instruction. Scatter data from the
      H dimension of Tensor A into Tensor out based on
     the provided index.
masksel        out_cnt, out, a, mask
// Mask selection instruction. Write elements from the W
     dimension of Tensor A to Tensor out at positions
     where the mask has a value of 1.
nzidx          out_cnt, out_idx, a
// Non-zero index generation instruction. Write the
     indices of non-zero elements from the W dimension of
      Tensor A into Tensor out.
```

Figure 29: Compute Instructions III

```
sync.i          rs, _engine
// rs holds the tag. After executing this instruction,
     the tag is written into CSR.sync_tag.The CPU can
     read this register to determine whether the
     instruction has finished executing, and once read,
     the register is automatically cleared.
msgsend         rs
// Send the information in rs to the message queue.
msgwait         rs
// Block instruction issue until a message is received
     from the message queue, then store the message in rs
     .
```

Figure 30: Synchronization Instructions