TOSA 0.30.0 specification

Table of Contents

TOSA Specification License ("License")

This Licence is a legal agreement between you and Arm Limited (“Arm”) for the use of Arm’s intellectual property (including, without limitation, any copyright) embodied in the relevant TOSA Specification accompanying this Licence (“Specification”). Arm licenses its intellectual property in the Specification to you on condition that you agree to the terms of this Licence. By using or copying the Specification you indicate that you agree to be bound by the terms of this Licence.

“Subsidiary” means any company the majority of whose voting shares is now or hereafter owner or controlled, directly or indirectly, by you. A company shall be a Subsidiary only for the period during which such control exists.

This Specification is NON-CONFIDENTIAL and any use by you and your Subsidiaries (“Licensee”) is subject to the terms of this Licence between you and Arm.

Subject to the terms and conditions of this Licence, Arm hereby grants to Licensee under the intellectual property in the Specification owned or controlled by Arm, a perpetual, a non-exclusive, non-transferable, non-sub-licensable, royalty-free, worldwide licence to:

i. use and copy the Specification solely for the purpose of designing and having designed products that fully complies with the Specification;

ii. manufacture and have manufactured products which have been created under the licence granted in (i) above; and

iii. sell, supply and distribute products which have been created under the licence granted in (i) above.

Licensee hereby agrees that the licenses granted above are conditional on implementing the Specification in products in its entirety and shall not extend to any portion or function of a product that is not itself fully compliant with the Specification.

Except as expressly licensed above, Licensee acquires no right, title or interest in any Arm technology or any intellectual property embodied therein.

Your access to the information in the Specification is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THE SPECIFICATION IS PROVIDED “AS IS”. ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE SPECIFICATION. Arm may make changes to the Specification at any time and without notice. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, third party patents, copyrights, trade secrets, or other rights.

NOTWITHSTANDING ANYTHING TO THE CONTRARY CONTAINED IN THIS LICENCE, TO THE FULLEST EXTENT PERMITTED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, IN CONTRACT, TORT OR OTHERWISE, IN CONNECTION WITH THE SUBJECT MATTER OF THIS LICENCE (INCLUDING WITHOUT LIMITATION: (I) LICENSEE’S USE OF THE SPECIFICATION; AND (II)

THE IMPLEMENTATION OF THE SPECIFICATION IN ANY PRODUCT CREATED BY LICENSEE UNDER THIS LICENCE). THE EXISTENCE OF MORE THAN ONE CLAIM OR SUIT WILL NOT ENLARGE OR EXTEND THE LIMIT. LICENSEE RELEASES ARM FROM ALL OBLIGATIONS, LIABILITY, CLAIMS OR DEMANDS IN EXCESS OF THIS LIMITATION.

This Licence shall remain in force until terminated by Licensee or by Arm. Without prejudice to any of its other rights, if Licensee is in breach of any of the terms and conditions of this Licence then Arm may terminate this Licence immediately upon giving written notice to Licensee. Licensee may terminate this Licence at any time. Upon termination of this Licence by Licensee or by Arm, Licensee shall stop using the Specification and destroy all copies of the Specification in its possession. Upon termination of this Licence, all terms shall survive except for the licence grants.

Any breach of this Licence by a Subsidiary shall entitle Arm to terminate this Licence as if you were the party in breach. Any termination of this Licence shall be effective in respect of all Subsidiaries. Any rights granted to any Subsidiary hereunder shall automatically terminate upon such Subsidiary ceasing to be a Subsidiary.

The Specification consists solely of commercial items. Licensee shall be responsible for ensuring that any use, duplication or disclosure of the Specification complies fully with any relevant export laws and regulations to assure that the Specification or any portion thereof is not exported, directly or indirectly, in violation of such export laws.

This Licence may be translated into other languages for convenience, and Licensee agrees that if there is any conflict between the English version of this Licence and any translation, the terms of the English version of this Licence shall prevail.

The validity, construction and performance of this Licence shall be governed by English Law.

Arm Limited. Company 02557590 registered in England. 110 Fulbourn Road, Cambridge, England CB1 9NJ.

1. Introduction

1.1. Overview

Tensor Operator Set Architecture (TOSA) provides a set of whole-tensor operations commonly employed by Deep Neural Networks. The intent is to enable a variety of implementations running on a diverse range of processors, with the results at the TOSA level consistent across those implementations. Applications or frameworks which target TOSA can therefore be deployed on a wide range of different processors, such as SIMD CPUs, GPUs and custom hardware such as NPUs/TPUs, with defined accuracy and compatibility constraints. Most operators from the common ML frameworks (TensorFlow, PyTorch, etc.) should be expressible in TOSA. It is expected that there will be tools to lower from ML frameworks into TOSA.

1.2. Goals

The goals of TOSA include the following:

- A minimal and stable set of tensor-level operators to which machine learning framework operators can be reduced.
- Full support for both quantized integer and floating-point content.
- Precise functional description of the behavior of every operator, including the treatment of their numerical behavior in the case of precision, saturation, scaling, and range as required by quantized datatypes.
- Agnostic to any single high-level framework, compiler backend stack or particular target.
- The detailed functional and numerical description enables precise code construction for a diverse range of targets – SIMD CPUs, GPUs and custom hardware such as NPUs/TPUs.

1.3. Specification

The TOSA Specification is written as AsciiDoc mark-up and developed in its raw mark-up form, managed through a git repository here: https://git.mlplatform.org/tosa/specification.git/. The specification is developed and versioned much like software. While the mark-up is legible and can be read fairly easily in its raw form, it is recommended to build or “render” the mark-up into PDF or HTML. To do this, please follow the instructions in the README.md in the root of the specification repository.

1.4. Operator Selection Principles

TOSA defines a set of primitive operators to which higher level operators can be lowered in a consistent way. To remain effective and efficient to implement, the set of operators must be constrained to a reasonably small set of primitive operations out of which others can be constructed. The following principles govern the selection of operators within TOSA.

Table 1. Principles

ID	Principle	Reason for this
P0	An operator shall be a primitive operation or building block that cannot be decomposed into simpler whole tensor operations.	If the operator can be broken down, then we should look at the component operators.
P1	An operator shall be a usable as a component out of which more complex operations can be constructed.	Single use operators have a high architectural cost and a more reusable version should be considered instead.
P2	Precision should be appropriate for the input and output data types.	Precision higher than that needed to calculate the result leads to extra implementation cost.
P3	Numerical definition of common sub-operations should be consistent between operators (for example: value scaling).	Consistent sub-operation definition reduces the operator implementation cost.
P4	The valid input and output ranges for all operands shall be specified.	Ranges are required to make consistent (numerically agreeing) implementations possible.
P5	Integer operators shall be implementable in a bit-exact form with good efficiency on CPU, GPU and hardware targets.	Reduces implementation cost and gives consistent inference results.

1.5. Profiles

TOSA supports three profiles that enable efficient implementation on different classes of device. The Base Inference profile is intended for embedded integer/fixed-point designs performing inference only. The Main Inference profile is intended for general inference functionality including integer and floating-point data types. The Main Training profile adds training operators in addition to inference operators. This version of the specification covers the Base Inference and Main Inference profiles. Main Training profile is expected in a later version of the specification. The following table summarizes the three profiles:

Table 2. Profiles

Profile	Name	Integer Inference	Floating-point Inference	Training
Base Inference	TOSA-BI	Yes	No	No
Main Inference	TOSA-MI	Yes	Yes	No
Main Training	TOSA-MT	Yes	Yes	Yes

1.6. Status

The TOSA specification is a work in progress.

- The Base Inference profile should be considered to be near release quality, with conformance

tests available.

- The Main Inference profile has most of the expected operators in place, but is still subject to change.
- The reference model and conformance tests do not yet support all of the floating point types that have been defined.
- There is not currently a conformance test suite available for Main Inference.
- Main Training profile is pre-alpha, significant work still needs to be done for the profile, and no conformance tests are available.

1.7. Compliance

This section defines when a TOSA implementation is compliant to a given TOSA specification profile. The term conformant will mean the same as compliant.

1.7.1. Baseline Inference Profile Compliance

The Operator Graphs section of this specification defines a TOSA graph and the behavior defined for a TOSA graph. This behavior is captured in the pseudo-code function tosa_execute_graph(). For a given input graph (with attributes) and input tensors there are three possible tosa_graph_result values after executing the graph:

- tosa_unpredictable: The result of the graph on the given inputs cannot be relied upon.
- tosa_error: The graph does not meet the specification and is recognised as an illegal graph.
- tosa_valid: The result is defined and predictable and the list of output tensors defines the result.

An implementation is compliant to the TOSA Baseline Inference Profile if it matches the above results as follows:

- For tosa_unpredictable, the implementation can return whatever result it chooses (including error)
- For tosa_error, the implementation must return an error result (and there is no requirement on how much of the graph is executed, if any)
- For tosa_valid, the implementation must execute the entire graph without error and return the result defined by this specification.

In terms of psuedo-code, if **graph** is a TOSA graph consisting of Baseline Inference Profile operators and **input_list** is a list of input tensors then the following test must pass.

```
bool tosa_test_compliance(tosa_graph_t graph, tosa_list_t input_list) {
    shape_list_t output_list_spec = tosa_allocate_list(tosa_output_shape(graph));
    shape_list_t output_list_test = tosa_allocate_list(tosa_output_shape(graph));
    tosa_graph_result = tosa_valid    // result starts as valid
    tosa_execute_graph(graph, input_list, output_list_spec);
    if (tosa_graph_result == tosa_unpredictable) {
        return true;    // No requirement to match an unpredictable result
    }
    result_test = execute_implementation_under_test(graph, input_list,
output_list_test);
    if (tosa_graph_result == tosa_error) {
        return result_test == tosa_error;   // result must be an error
    }
    if (exact_tensor_match(output_list_spec, output_list_test)) {
       // Predictable bit-exact value match required
       return true;
    }
    return false;
}
```

1.7.2. Main Inference and Main Training Profile

An implementation is compliant to the Main Inference or Main Training profiles if the following both hold for that respective profile:

- For a graph returning tosa_error the implementation must also return an error
- For a graph returning tosa_valid the implementation must execute the entire graph without error
- For a graph returning tosa_valid and consisting only of integer operators the results must match exactly
- The implementation must report the maximum relative error on a set of standard graphs that contain floating point operators. These graphs will be provided as a future appendix to this specification.

Note that for graphs containing floating point there is no strict precision requirement that must be met, but that the precision achieved must be reported.

1.8. Tensor Definitions

1.8.1. Tensors

Tensors are multidimensional arrays of data. Tensors have metadata associated with them that describe characteristics of the tensor, including:

- Data Type
- Shape

The number of dimensions in a shape is called the rank. A tensor with rank equal to zero is permitted. In that case, the tensor has a single entry. A tensor shape is an array of integers of size equal to the rank of the tensor. Each element in the tensor shape describes the number of elements in the dimension. The tensor shape in each dimension must be greater than or equal to 1. For tensor access information, see Tensor Access Helpers. Tensor dimensions are given in the pseudocode as type dim_t. dim_t is a vector of int32_t values, with the length of the vector defining the rank of the tensor. Tensor elements are addressed using dim_t values, where each element of the vector indicates the offset of the requested element within the corresponding dimension.

1.8.2. Tensor size limit

Tensor size is limited by the data type size_t. In this version of the specification, size_t is defined as (1<<32) - 1, and can be represented with an unsigned 32-bit integer.

1.8.3. Data Layouts

The following data layouts are supported in TOSA. TOSA operations are defined in terms of a linear packed tensor layout. In a linear packed layout a rank r tensor has elements of dimension (r-1) consecutive. The next to increment is dimension (r-2) and so on. For a specification of this layout see the tensor read and write functions in section Tensor Access Helpers.

An implementation of TOSA can choose a different tensor memory layout provided that the operation behavior is maintained.

Table 3. Data Layouts

Name	Description of dimensions	Usage
NHWC	Batch, Height, Width, Channels	Feature maps
NDHWC	Batch, Depth, Height, Width, Channels	Feature maps for 3D convolution
OHWI	Output channels, Filter Height, Filter Width, Input channels	Weights
HWIM	Filter Height, Filter Width, Input channels, Channel Multiplier	Weights for depthwise convolutions
DOHWI	Depth, Output Channels, Filter Height, Filter Width, Input Channels	Weights for 3D convolution

1.8.4. Broadcasting

In operations where broadcasting is supported, an input shape dimension can be broadcast to an output shape dimension if the input shape dimension is 1. TOSA broadcast requires the rank of both tensors to be the same. A RESHAPE can be done to create a compatible tensor with appropriate dimensions of size 1. To map indexes in an output tensor to that of an input tensor, see Broadcast Helper.

1.8.5. Supported Number Formats

The following number formats are defined in TOSA. The number formats supported by a given

operator are listed in its table of supported types.

Table 4. Number formats

Format	Minimum	Maximum	Description
bool_t	-	-	Boolean value. Size implementation defined. The TOSA reference model implements this as int8_t with 0 for false and 1 for true. All non-zero values are accepted on input as true.
int4_t	-7	+7	Signed 4-bit two's-complement value. Excludes -8 to maintain a symmetric about zero range for weights.
int8_t	-128	+127	Signed 8-bit two's-complement value.
uint8_t	0	255	Unsigned 8-bit value.
int16_t	-32768	+32767	Signed 16-bit two's-complement value.
uint16_t	0	65535	Unsigned 16-bit value.
int32_t	-(1<<31)	(1<<31)-1	Signed 32-bit two's-complement value.
int48_t	-(1<<47)	(1<<47)-1	Signed 48-bit two's-complement value.
fp16_t	-infinity	+infinity	16-bit floating-point value.
bf16_t	-infinity	+infinity	16-bit brain float value.
fp32_t	-infinity	+infinity	32-bit floating-point value.

Note: In this specification minimum<type> and maximum<type> will denote the minimum and maximum values of the data as stored in memory (ignoring the zero point). The minimum and maximum values for each type is given in the preceeding table.

Note: Integer number formats smaller than 8 bits may be used provided that the numerical result is the same as using a sequence of 8-bit TOSA operations. For example, a convolution with low precision data must equal that of running the convolution at 8 bits and then clipping the result to the peritted output range. This ensures that a Base Inference profile TOSA implementation can calculate the same result.

1.9. Integer Behavior

Integer calculations must be standard two's-complement or unsigned calculations. If overflow occurs doing integer calculation, the result is unpredictable, as indicated by the REQUIRE checks in the pseudocode for the operators.

Unsigned 8 and 16-bit values are only allowed in the RESCALE operation, to allow for compatibility with networks which expect unsigned 8-bit or 16-bit tensors for input and output.

1.9.1. Quantization

Machine Learning frameworks may represent tensors with a quantized implementation, using integer values to represent the original floating-point numbers. TOSA integer operations do not perform any implicit scaling to represent quantized values. Required zero point values are passed

to the operator as necessary, and will be processed according to the pseudocode for each operator.

To convert a network containing quantized tensors to TOSA, generate explicit RESCALE operators for any change of quantization scaling. This reduces quantized operations to purely integer operations.

As an example, an ADD between two quantized tensors requires the integer values represent the same range. The scale parameters for RESCALE can be calculated to ensure that the resulting tensors represent the same range. Then the ADD is performed, and a RESCALE can be used to ensure that the result is scaled properly.

RESCALE provides support for per-tensor and per-channel scaling values to ensure compatibility with a range of possible quantization implementations.

1.9.2. Precision scaling

TOSA uses the RESCALE operation to scale between values with differing precision. The RESCALE operator is defined using an integer multiply, add, and shift. This guarantees that all TOSA implementations will return the same result for a RESCALE, including those with no support for floating-point numbers.

This TOSA specification supports two precisions of multiplier: 16-bit and 32-bit. The 32-bit multiplier version supports two rounding modes to enable simpler lowering of existing frameworks that use two stage rounding. All arithmetic is designed so that it does not overflow a 64-bit accumulator and that the final result fits in 32 bits. In particular a 48-bit value can only be scaled with the 16-bit multiplier.

The apply_scale functions provide a scaling of approximately (multiplier * 2^{-shift}). The shift and value range is limited to allow a variety of implementations. The limit of 62 on shift allows the shift to be decomposed as two right shifts of 31. The limit on value allows implementations that left shift the value before the multiply in the case of shifts of 32 or less. For example, in the case shift=30 an implementation of the form ((value<<2) * multiplier + round)>>32 can be used. A scaling range of 2^{+12} down to 2^{-32} is supported for both functions with a normalized multiplier.

For example, in typical usage a scaling of m*2^{-n} where m is a fraction in the range 1.0 <= m < 2.0 can be represented using multiplier=(1<<30)*m, shift=(30+n) for apply_scale_32() and multiplier=(1<<14)*m, shift=(14+n) for apply_scale_16(). The values to achieve a scaling of 1.0 are shift=30, multiplier=1<<30 for apply_scale_32 and shift=14, multiplier=1<<14 for apply_scale_16.

```
int32_t apply_scale_32(int32_t value, int32_t multipler, uint6_t shift, bool_t
double_round=false) {
    REQUIRE(multiplier >= 0);
    REQUIRE(2 <= shift && shift <= 62);
    REQUIRE(value >= (-1<<(shift-2)) && value < (1<<(shift-2));
    int64_t round = 1 << (shift - 1);
    if (double_round) {
        if (shift > 31 && value >= 0) round += 1<<30;
        if (shift > 31 && value < 0)  round -= 1<<30;
    }
    int64_t result = (int64_t)value * multiplier + round;
    result = result >> shift;
    // result will fit a 32-bit range due to the REQUIRE on value
    return (int32_t)result;
}

int32_t apply_scale_16(int48_t value, int16_t multipler, uint6_t shift) {
    REQUIRE(multiplier >= 0);
    REQUIRE(2 <= shift && shift <= 62);
    int64_t round = (1 << (shift - 1));
    int64_t result = (int64_t)value * multiplier + round;
    result = result >> shift;
    REQUIRE(result >= minimum<int32_t> && result <= maximum<int32_t>);
    return (int32_t)result;
}
```

In some functions, the multiplier and shift are combined into a scale_t structure:

```
typedef struct {
    int32_t multiplier;
    uint6_t shift;
} scale_t;
```

In places where a divide is required, we also use the function below to calculate an appropriate scaling value.

```
scale_t reciprocal_scale(uint32_t value) {
    REQUIRE(value > 0);
    scale_t scale;
    int32_t k = 32 - count_leading_zeros(value - 1); // (1 << k) / 2 < value <= (1 <<
k)
    int64_t numerator = ((1 << 30) + 1) << k;
    scale.multiplier = numerator / value; // (1 << 30) <= multiplier < (1 << 31)
    scale.shift = 30 + k;
    return scale;
}
```

1.9.3. Integer Convolutions

For the convolution operators, the input is not required to be scaled. The integer versions of the convolution operators will subtract the zero point from the integer values as defined for each operator. The convolution produces an accumulator output of type int32_t or int48_t. This accumulator output is then scaled to the final output range using the RESCALE operator. The scale applied in the RESCALE operator should be set to multiplier and shift values such that: multiplier * 2^{-shift} = (input scale * weight scale) / output_scale. Here, input_scale, weight_scale and output_scale are the conversion factors from integer to floating-point for the input, weight and output tensor values respectively. If per-channel scaling is needed then the per-channel option of the RESCALE operation should be used.

1.9.4. Integer Elementwise Operators

When two quantized tensors are used in an operation, they must represent the same numeric range for the result to be valid. In this case, TOSA expects that RESCALE operators will be used as necessary to generate 32-bit integer values in a common range. There are many valid choices for scale factors and options for the common range. TOSA does not impose a requirement on which scale factors and range should be used. Compilers generating TOSA sequences should choose a range that allows the operation to be computed without overflow, while allowing the highest possible accuracy of the output.

1.9.5. General Unary Functions

General unary functions such as sigmoid(), tanh(), exp() for integer inputs are expressed using a lookup table and interpolation to enable efficient implementation. This also allows for other operations with the addition of user-supplied tables (the TABLE operation). All table lookups are based on the following reference lookup function that takes as input a table of 513 entries of 16 bits each.

```
int32_t apply_lookup(int16_t *table, int32_t value)
{
    int16_t clipped_value = (int16_t)apply_clip<int32_t>(value, -32768, +32767);
    int32_t index = (clipped_value + 32768) >> 7;
    int32_t fraction = clipped_value & 0x7f;
    int16_t base = table[index];
    int16_t next = table[index+1];
    int32_t slope = next - base;
    REQUIRE(slope >= minimum<int16_t> && slope <= maximum<int16_t>)
    int32_t return_value = (base << 7) + slope * fraction;
    return return_value;    // return interpolated value of 16 + 7 = 23 bits
}
```

Note that although the table lookup defined here has 16-bit precision, for 8-bit only operations an 8-bit table can be derived by applying the reference function to each of the possible 256 input values. The following code constructs a 513-entry table based on a reference function.

```
void generate_lookup_table(int16_t *table, int32_t (*reference)(int32_t))
{
    for (int i = -256; i <= 256; i++) {
        int32_t value = (*reference)(i);
        table[i + 256] = (int16_t)apply_clip<int32_t>(value, -32768, +32767)
    }
}
```

1.10. Floating-point

Floating-point support is included in the main inference profile. TOSA does not define bit-exact behavior of the floating-point type, since floating-point operation results can vary according to operation order (floating-point addition is not associative in general) and rounding behavior. If a bit-exact answer is required then integer operations should be used. TOSA does define that the floating-point type must support the following list of features. These features ensure that detection of overflow and other exceptional conditions can be handled consistently.

- The floating-point type must have at least 16 total bits including the sign bit
- The floating-point type must support positive and negative infinity values
- The floating-point type must support at least one Not-a-Number encoding (NaN)
- The floating-point type must support signed zero
- The floating-point type must support handling of infinities, NaNs, zeros as in the following table

Table 5. floating-point behavior

Case	**Result**
Operators other than explicitly mentioned by other rules: Any input operand is a NaN	a NaN
Comparisons (EQUAL, GREATER, GREATER_EQUAL), where either or both operands is NaN	False
Comparisons ignore the sign of 0	
RSQRT (reciprocal square root) of negative numbers	a NaN
(± 0) × (± infinity), (± infinity) × (± 0)	a NaN
LOG of negative numbers	a NaN
nonzero numbers / (± 0)	(± infinity)
(± 0) / (± 0), (± infinity) / (± infinity)	a NaN
(± infinity) * 0	a NaN
(+infinity) - (+infinity), (+infinity) + (-infinity)	a NaN
Any positive overflow	+ infinity

Case	Result
Any negative overflow	- infinity
Any positive underflow	+ 0
Any negative underflow	- 0

2. Operators

2.1. Operator Parameters

An operator processes input operands to produce output operands. An operator can have three types of parameters:

- An input operand. This must be a tensor or a list of tensors and data is read by the operation.
- An output operand. This must be a tensor or a list of tensors and data is written by the operation.
- An attribute. This is a parameter that is constant for a particular instance of the operator. It may have any data type supported by TOSA. It is expected to be set at compile time.

2.2. Operator Graphs

A TOSA graph is a collection of TOSA operators where:

- The output of an operator in the graph may be connected to one or more inputs of other operators in the graph
- When an output is connected to an input the tensor list shapes must match
- The attributes of the operators are defined and considered part of the graph
- The attributes must be in the valid range permitted for the operator
- The tensor dimensions must be in the valid range permitted for the operator

Some operators, such as control flow operators, take a graph of other operators as an attribute. The type tosa_graph_t will denote a graph of operators and the following functions define the tensor shape list for the graph input and outputs:

```
shape_list_t tosa_input_shape(tosa_graph_t graph);
shape_list_t tosa_output_shape(tosa_graph_t graph);
```

Similarly the type tensor_list_t will be used for a list of tensors and the following function returns the shape of a tensor list:

```
shape_list_t tensor_list_shape(tosa_list_t tensor_list);
```

The following function denotes the execution of a TOSA graph, on an input tensor list to produce an output tensor list.

```
tosa_execute_graph(tosa_graph_t graph, tosa_list_t input_list, tosa_list_t
output_list) {
    ERROR_IF(tensor_list_shape(input_list) != tosa_input_shape(graph));
    ERROR_IF(tensor_list_shape(output_list) != tosa_output_shape(graph));
    for_each(operator in graph order) {
        ERROR_IF(operator input tensors do not meet requirement of operator Arguments
inputs)
        ERROR_IF(operator attributes do not meet requirement of operator Arguments
attributes)
        ERROR_IF(operator output tensors do not meet requirement of operator Arguments
outputs)
        ERROR_IF(operator data types do not meet requirement of operator Supported
Data Types)
        <Execute operator as defined by the Operation Function pseduo-code>
    }
}
```

2.3. Tensor Operators

2.3.1. ARGMAX

This returns the index with the largest value across the given axis of the input tensor.

Arguments

Argument	Type	Name	Shape	Description
Input	in_t*	input	shape1	Input tensor with rank from 1 to 4
Attribute	int32_t	axis	-	Axis in range from 0 to rank(shape1)-1
Output	out_t*	output	shape	Output tensor, with rank = rank(shape1)-1

Operation Function:

```
ERROR_IF(axis < 0 || axis >= rank(shape1) || rank(shape1) > 4);
if (axis == 0) {
    left_shape = [];
} else {
    left_shape = shape1[0:axis - 1];
}
if (axis == rank(shape1)-1) {
    right_shape = [];
} else {
    right_shape = shape1[axis+1:rank(shape1) - 1];
}
ERROR_IF(flatten(left_shape, right_shape) != shape);
for_each(left_index in left_shape) {
    for_each(right_index in right_shape) {
        in_t max_value = minimum_value<in_t>;
        out_t max_index = 0;
        for (i = 0; i < shape[axis]; i++) {
            index = flatten(left_index, [i], right_index);
            in_t value = tensor_read<in_t>(input, shape1, index);
            if (value > max_value) { max_value = value; max_index = i; }
        }
        index = flatten(left_index, right_index);
        tensor_write<out_t>(output, shape, index, max_index);
    }
}
```

Supported Data Types:

Profile	Mode	in_t	out_t
Any	signed 8	int8_t	int32_t
Any	signed 16	int16_t	int32_t
MI, MT	fp16	fp16_t	int32_t
MI, MT	bf16	bf16_t	int32_t
MI, MT	fp32	fp32_t	int32_t

2.3.2. AVG_POOL2D

This performs an average pooling over the given input tensor. A sliding window of size given by <kernel size> is passed over the input tensor, with the mean value being placed in the output tensor. When calculating the average, only the number of valid input tensor values, but not padding, are used to calculate the divisor. **Arguments:**

Argument	Type	Name	Shape	Description
Input	in_out_t*	input	[N,IH,IW,C]	Input tensor 4D

Argument	Type	Name	Shape	Description
Attribute	int32_t*	kernel	[2]	[kernel_y, kernel_x]
Attribute	int32_t*	stride	[2]	[stride_y, stride_x]
Attribute	int32_t*	pad	[4]	[pad_top, pad_bottom, pad_left, pad_right]
Attribute	in_out_t	input_zp	-	Input tensor zero point. Must be zero for non-int8 types.
Attribute	in_out_t	output_zp	-	Output tensor zero point. Must be zero for non-int8 types.
Output	in_out_t*	output	[N,OH,OW,C]	Output tensor 4D

Operation Function:

```
ERROR_IF(in_out_t != int8_t && input_zp != 0); // Zero point only for int8_t
ERROR_IF(in_out_t != int8_t && output_zp != 0); // Zero point only for int8_t
ERROR_IF(kernel_y < 1 || kernel_x < 1); // kernel size must be >= 1
ERROR_IF(stride_y < 1 || stride_x < 1);
ERROR_IF(pad_top < 0 || pad_bottom < 0 || pad_left < 0 || pad_right < 0);
// Padding must be less than kernel size to avoid
// a divide-by-zero.
ERROR_IF(pad_right >= kernel_x || pad_left >= kernel_x);
ERROR_IF(pad_top >= kernel_y || pad_bottom >= kernel_y);
ERROR_IF(OH != idiv_check(IH + pad_top + pad_bottom - kernel_y, stride_y) + 1);
ERROR_IF(OW != idiv_check(IW + pad_left + pad_right - kernel_x, stride_x) + 1);

for_each(0 <= n < N, 0 <= oy < OH, 0 <= ox < OW, 0 <= c < C ) {
    in_out_t output_val;
    acc_t acc = 0;
    int count = 0;
    iy = oy * stride_y - pad_top;
    ix = ox * stride_x - pad_left;
    for_each(0 <= ky < kernel_y, 0 <= kx < kernel_x) {
        y = iy + ky;
        x = ix + kx;
        // Only values from the input tensor are used to calculate the
        // average, padding does not count
        if (0 <= y < IH and 0 <= x < IW) {
            count++;
            acc_t value = tensor_read<in_out_t>(input, [N,IH,IW,C], [n,y,x,c]);
            value = value - input_zp;
            acc = apply_add<acc_t>(acc, value);
        }
    }
    if (is_float(in_out_t)) {
        output_val = acc / (float)count;
    } else {
        scale_t scale = reciprocal_scale(count);
        acc = apply_scale_32(acc, scale.multiplier, scale.shift, false);
        output_val = (in_out_t)apply_clip<acc_t>(acc + output_zp, minimum<in_out_t>,
maximum<in_out_t>)
    }
    tensor_write<in_out_t>(output, [N,OH,OW,C], [n,oy,ox,c], output_val);
}
```

Supported Data Types:

Profile	Mode	in_out_t	acc_t
Any	signed 8	int8_t	int32_t
Any	signed 16	int16_t	int32_t
MI, MT	fp16 with fp16 accumulate	fp16_t	fp16_t

Profile	Mode	in_out_t	acc_t
MI, MT	fp16 with fp32 accumulate	fp16_t	fp32_t
MI, MT	bf16 with fp32 accumulate	bf16_t	fp32_t
MI, MT	fp32	fp32_t	fp32_t

2.3.3. CONV2D

Performs a 2D convolution over the given tensor input, using the weight tensor.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_t*	input	[N,IH,IW,IC]	Input tensor
Input (MT profile) Attribute (BI/MI profiles)	weight_t*	weight	[OC,KH,KW,IC]	Weight kernel size KH x KW
Input (MT profile) Attribute (BI/MI profiles)	out_t*	bias	[OC]	Per output channel bias data.
Attribute	int32_t*	pad	[4]	[pad_top, pad_bottom, pad_left, pad_right]
Attribute	int32_t*	stride	[2]	[stride_y, stride_x]
Attribute	int32_t*	dilation	[2]	[dilation_y, dilation_x]
Attribute	in_t	input_zp	-	Input tensor zero point. Must be zero for non-int8 types.
Attribute	weight_t	weight_zp	-	Weight zero point. Must be zero for non-int8 types.
Output	out_t*	output	[N,OH,OW,OC]	Output tensor

Operation Function

```
ERROR_IF(in_t != int8_t && input_zp != 0); // Zero point only for int8_t
ERROR_IF(weight_t != int8_t && weight_zp != 0);
ERROR_IF(pad_top < 0 || pad_bottom < 0 || pad_left < 0 || pad_right < 0);
ERROR_IF(stride_y < 1 || stride_x < 1);
ERROR_IF(dilation_y < 1 || dilation_x < 1);
ERROR_IF(OH != idiv_check(IH - 1 + pad_top + pad_bottom - (KH - 1) * dilation_y,
stride_y) + 1);
ERROR_IF(OW != idiv_check(IW - 1 + pad_left + pad_right - (KW - 1) * dilation_x,
stride_x) + 1);

pad = flatten([0,0], pad, [0,0]);
for_each(0 <= n < N, 0 <= oy < OH, 0 <= ox < OW; 0 <= oc < OC) {
    out_t acc = 0;
    iy = oy * stride_y - pad_top;
    ix = ox * stride_x - pad_left;
    for_each(0 <= ky < KH, 0 <= kx < KW, 0 <= ic < IC) {
        y = iy + ky * dilation_y;
        x = ix + kx * dilation_x;
        if (0 <= y < IH && 0 <= x < IW) {
            out_t value  = tensor_read<in_t>(input, [N,IH,IW,IC], [n,y,x,ic]);
            out_t weight = tensor_read<weight_t>(weight, [OC,KH,KW,IC],
[oc,ky,kx,ic]);
            value  = value - input_zp;
            weight = weight - weight_zp;
            acc = apply_add<out_t>(acc, value * weight);
        }
    }
    acc = apply_add<out_t>(acc, bias[oc]);
    tensor_write<out_t>(output, [N,OH,OW,OC], [n,oy,ox,oc], acc);
}
```

Supported Data Types:

Profile	Mode	in_t	weight_t	out_t
Any	signed 8x8	int8_t	int8_t	int32_t
Any	signed 8x4	int8_t	int4_t	int32_t
Any	signed 16x8	int16_t	int8_t	int48_t
MI, MT	fp16 with fp16 accumulate	fp16_t	fp16_t	fp16_t
MI, MT	fp16 with fp32 accumulate	fp16_t	fp16_t	fp32_t
MI, MT	bf16 with fp32 accumulate	bf16_t	bf16_t	fp32_t
MI, MT	fp32	fp32_t	fp32_t	fp32_t

2.3.4. CONV3D

Performs a 3D convolution over the given input tensor.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_t*	input	[N,ID,IH,IW,IC]	Input tensor
Input (MT profile) Attribute (BI/MI profiles)	weight_t*	weight	[OC,KD,KH,KW,IC]	Weight kernel size KDxKHxKW
Input (MT profile) Attribute (BI/MI profiles)	out_t*	bias	[OC]	Per output channel bias data.
Attribute	int32_t*	pad	[6]	[pad_d0, pad_d1, pad_top, pad_bottom, pad_left, pad_right]
Attribute	int32_t*	stride	[3]	[stride_d, stride_y, stride_x]
Attribute	int32_t*	dilation	[3]	[dilation_d, dilation_y, dilation_x]
Attribute	in_t	input_zp	-	Input tensor zero point. Must be zero for non-int8 types.
Attribute	weight_t	weight_zp	-	Weight zero point. Must be zero for non-int8 types.
Output	out_t*	output	[N,OD,OH,OW,OC]	Output tensor

Operation Function

```
ERROR_IF(in_t != int8_t && input_zp != 0); // Zero point only for int8_t
ERROR_IF(weight_t != int8_t && weight_zp != 0);
ERROR_IF(pad_d0 < 0 || pad_d1 < 0 || pad_top < 0 || pad_bottom < 0 || pad_left < 0 ||
pad_right < 0);
ERROR_IF(stride_d < 1 || stride_y < 1 || stride_x < 1);
ERROR_IF(dilation_d < 1 || dilation_y < 1 || dilation_x < 1);
ERROR_IF(OD != idiv_check(ID - 1 + pad_d0 + pad_d1      - (KD - 1) * dilation_d,
stride_d) + 1);
ERROR_IF(OH != idiv_check(IH - 1 + pad_top + pad_bottom - (KH - 1) * dilation_y,
stride_y) + 1);
ERROR_IF(OW != idiv_check(IW - 1 + pad_left + pad_right - (KW - 1) * dilation_x,
stride_x) + 1);

pad = flatten([0,0], pad, [0,0]);
for_each(0 <= n < N, 0 <= od < OD, 0 <= oy < OH, 0 <= ox < OW; 0 <= oc < OC) {
    out_t acc = 0;
    id = od * stride_d - pad_d0;
    iy = oy * stride_y - pad_top;
    ix = ox * stride_x - pad_left;
    for_each(0 <= kd < KD, 0 <= ky < KH, 0 <= kx < KW, 0 <= ic < IC) {
        d = id + kd * dilation_d;
        y = iy + ky * dilation_y;
        x = ix + kx * dilation_x;
        if (0 <= x < IW && 0 <= y < IH && 0 <= d < ID) {
            out_t value  = tensor_read<in_t>(input, [N,ID,IH,IW,IC], [n,d,y,x,ic]);
            out_t weight =
tensor_read<weight_t>(weight,[OC,KD,KH,KW,IC],[oc,kd,ky,kx,ic]);
            value  = value - input_zp;
            weight = weight - weight_zp;
            acc = apply_add<out_t>(acc, value * weight);
        }
    }
    acc = apply_add<out_t>(acc, bias[oc]);
    tensor_write<out_t>(output, [N,OD,OH,OW,OC], [n,od,oy,ox,oc], acc);
}
```

Supported Data Types:

Profile	Mode	in_t	weight_t	out_t
Any	signed 8x8	int8_t	int8_t	int32_t
Any	signed 8x4	int8_t	int4_t	int32_t
Any	signed 16x8	int16_t	int8_t	int48_t
MI, MT	fp16 with fp16 accumulate	fp16_t	fp16_t	fp16_t
MI, MT	fp16 with fp32 accumulate	fp16_t	fp16_t	fp32_t

Profile	Mode	in_t	weight_t	out_t
MI, MT	bf16 with fp32 accumulate	bf16_t	bf16_t	fp32_t
MI, MT	fp32	fp32_t	fp32_t	fp32_t

2.3.5. DEPTHWISE_CONV2D

Performs 2D convolutions separately over each channel of the given tensor input, using the weight tensor.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_t*	input	[N,H,W,C]	Input tensor
Input (MT profile) Attribute (BI/MI profiles)	weight_t*	weight	[KH,KW,C,M]	Weight kernel size KH x KW
Input (MT profile) Attribute (BI/MI profiles)	out_t*	bias	[C*M]	Per output channel bias data.
Attribute	int32_t*	pad	[4]	[pad_top, pad_bottom, pad_left, pad_right]
Attribute	int32_t*	stride	[2]	[stride_y, stride_x]
Attribute	int32_t*	dilation	[2]	[dilation_y, dilation_x]
Attribute	in_t	input_zp	-	Input tensor zero point. Must be zero for non-int8 types.
Attribute	weight_t	weight_zp	-	Weight zero point. Must be zero for non-int8 types.
Output	out_t*	output	[N,OH,OW,C*M]	Output tensor

Operation Function

```
ERROR_IF(in_t != int8_t && input_zp != 0); // Zero point only for int8_t
ERROR_IF(weight_t != int8_t && weight_zp != 0);
ERROR_IF(pad_top < 0 || pad_bottom < 0 || pad_left < 0 || pad_right < 0);
ERROR_IF(stride_y < 1 || stride_x < 1);
ERROR_IF(dilation_y < 1 || dilation_x < 1);
ERROR_IF(OH != idiv_check(IH - 1 + pad_top + pad_bottom - (KH - 1) * dilation_y,
stride_y) + 1);
ERROR_IF(OW != idiv_check(IW - 1 + pad_left + pad_right - (KW - 1) * dilation_x,
stride_x) + 1);

pad = flatten([0,0], pad, [0,0]);
for_each(0 <= n < N, 0 <= oy < OH, 0 <= ox < OW; 0 <= c < C, 0 <= m < M) {
    out_t acc = 0;
    iy = oy * stride_y - pad_top;
    ix = ox * stride_x - pad_left;
    for_each(0 <= ky < KH, 0 <= kx < KW) {
        y = iy + ky * dilation_y;
        x = ix + kx * dilation_x;
        if (0 <= y < IH && 0 <= x < IW) {
            out_t value  = tensor_read<in_t>(input, [N,IH,IW,C], [n,y,x,c]);
            out_t weight = tensor_read<weight_t>(weight, [KH,KW,C,M], [ky,kx,c,m]);
            value  = value - input_zp;
            weight = weight - weight_zp;
            acc = apply_add<out_t>(acc, value * weight);
        }
    }
    acc = apply_add<out_t>(acc, bias[(c * M) + m]);
    tensor_write<out_t>(output, [N,OH,OW,C * M], [n,oy,ox,c * M + m], acc);
}
```

Supported Data Types:

Profile	Mode	in_t	weight_t	out_t
Any	signed 8x8	int8_t	int8_t	int32_t
Any	signed 8x4	int8_t	int4_t	int32_t
Any	signed 16x8	int16_t	int8_t	int48_t
MI, MT	fp16 with fp16 accumulate	fp16_t	fp16_t	fp16_t
MI, MT	fp16 with fp32 accumulate	fp16_t	fp16_t	fp32_t
MI, MT	bf16 with fp32 accumulate	bf16_t	bf16_t	fp32_t
MI, MT	fp32	fp32_t	fp32_t	fp32_t

2.3.6. FFT2D

Performs a batched complex 2D Fast Fourier Transform over the input. The complex input values are constructed from the corresponding values in the input_real and input_imag tensors. The resulting values in the output are split into the output_real and output_imag tensors. No normalization is applied on either the forward or inverse versions of the operation.

$$output[h][w] = \sum_{m=0}^{H-1} \sum_{n=0}^{W-1} input[m][n] * \exp\left(-2\pi i \left(\frac{mh}{H} + \frac{nw}{W}\right)\right)$$

Figure 1. Calculation for the forward FFT2D calculation (inverse=false)

$$output[h][w] = \sum_{m=0}^{H-1} \sum_{n=0}^{W-1} input[m][n] * \exp\left(2\pi i \left(\frac{mh}{H} + \frac{nw}{W}\right)\right)$$

Figure 2. Calculation for the inverse FFT2D calculation (inverse=true)

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input_real	[N,H,W]	Real part of the complex input. H,W must be powers of two.
Input	in_out_t*	input_imag	[N,H,W]	Imaginary part of the complex input. H,W must be powers of two.
Attribute	bool_t	inverse	-	false for forward FFT, true for inverse FFT
Output	in_out_t*	output_real	[N,H,W]	Real part of the complex output
Output	in_out_t*	output_imag	[N,H,W]	Imaginary part of the complex output.

Operation Function

```
ERROR_IF(!power_of_two(H));
ERROR_IF(!power_of_two(W));

float sign_val = 1.0;

if (inverse) {
    sign_val = -1.0;
}

for_each(0 <= n < N, 0 <= oy < H, 0 <= ox < W) {
    in_out_t sum_real = 0.0;
    in_out_t sum_imag = 0.0;
    for_each(0 <= iy < H, 0 <= ix < W) {
        in_out_t val_real = tensor_read<in_out_t>(input_real, [N,H,W], [n,iy,ix]);
        in_out_t val_imag = tensor_read<in_out_t>(input_imag, [N,H,W], [n,iy,ix]);
        float_t a = sign_val * 2 * pi() * ((iy * oy) / H + (ix * ox) / W);
        sum_real += val_real * cos(a) + val_imag * sin(a);
        sum_imag += -val_real * sin(a) + val_imag * cos(a);
    }
    tensor_write<in_out_t>(output_real, [N,H,W], [n,oy,ox], sum_real);
    tensor_write<in_out_t>(output_imag, [N,H,W], [n,oy,ox], sum_imag);
}
```

Supported Data Types:

Profile	Mode	in_out_t
MI,MT	fp32_t	fp32_t

2.3.7. FULLY_CONNECTED

Performs a fully connected network.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_t*	input	[N,IC]	Input tensor
Attribute	weight_t*	weight	[OC,IC]	Weights
Attribute	out_t*	bias	[OC]	Per output channel bias data.
Attribute	in_t	input_zp	-	Input tensor zero point. Must be zero for non-int8 types.
Attribute	weight_t	weight_zp	-	Weight zero point. Must be zero for non-int8 types.

Argument	Type	Name	Shape	Description
Output	out_t*	output	[N,OC]	Output tensor

Operation Function

```
ERROR_IF(in_t != int8_t && input_zp != 0); // Zero point only for int8_t
ERROR_IF(weight_t != int8_t && weight_zp != 0);
for_each(0 <= n < N, 0 <= oc < OC) {
    out_t acc = 0;
    for_each(0 <= ic < IC) {
        out_t value  = tensor_read<in_t>(input, [N,IC], [n,ic]);
        out_t weight = tensor_read<weight_t>(weight, [OC,IC], [oc,ic]);
        value  = value - input_zp;
        weight = weight - weight_zp;
        acc = apply_add<out_t>(acc, value * weight);
    }
    acc = apply_add<out_t>(acc, bias[oc]);
    tensor_write<out_t>(output, [N,OC], [n,oc], acc);
}
```

Supported Data Types:

Profile	Mode	in_t	weight_t	out_t
Any	signed 8x8	int8_t	int8_t	int32_t
Any	signed 8x4	int8_t	int4_t	int32_t
Any	signed 16x8	int16_t	int8_t	int48_t
MI, MT	fp16 with fp16 accumulate	fp16_t	fp16_t	fp16_t
MI, MT	fp16 with fp32 accumulate	fp16_t	fp16_t	fp32_t
MI, MT	bf16 with fp32 accumulate	bf16_t	bf16_t	fp32_t
MI, MT	fp32	fp32_t	fp32_t	fp32_t

2.3.8. MATMUL

Performs two dimensional matrix multiplications. This allows both inputs to be activations, rather than reserving weights as an attribute in the FULLY_CONNECTED operator.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_t*	A	[N,H,C]	Input tensor A, N matrices of size HxC
Input	in_t*	B	[N,C,W]	Input tensor B, N matrices of size CxW
Attribute	in_t	A_zp	-	Input tensor A zero point. Must be zero for non-int8 types.
Attribute	in_t	B_zp	-	Input tensor B zero point. Must be zero for non-int8 types.
Output	out_t*	output	[N,H,W]	Output tensor, N matrices of size HxW

Operation Function

```
ERROR_IF(in_t != int8_t && (A_zp != 0 || B_zp != 0)); // Zero point only for int8_t
for_each(0 <= n < N, 0 <= h < H, 0 <= w < W) {
    out_t acc = 0;
    for_each(0 <= c < C) {
        out_t value1 = tensor_read<in_t>(A, [N,H,C], [n,h,c]);
        out_t value2 = tensor_read<in_t>(B, [N,C,W], [n,c,w]);
        value1 = value1 - A_zp;
        value2 = value2 - B_zp;
        acc = apply_add<out_t>(acc, value1 * value2);
    }
    tensor_write<out_t>(output, [N,H,W], [n,h,w], acc);
}
```

Supported Data Types:

Profile	Mode	in_t	out_t
Any	signed 8x8	int8_t	int32_t
Any	signed 16x16	int16_t	int48_t
MI, MT	fp16 with fp16 accumulate	fp16_t	fp16_t
MI, MT	fp16 with fp32 accumulate	fp16_t	fp32_t

Profile	Mode	in_t	out_t
MI, MT	bf16 with fp32 accumulate	bf16_t	fp32_t
MI, MT	fp32	fp32_t	fp32_t

2.3.9. MAX_POOL2D

This performs a max pooling over the given input tensor. A sliding window of size given by <kernel size> is passed over the input tensor, with the maximum value being placed in the output tensor.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input	[N,IH,IW,C]	Input tensor 4D
Attribute	int32_t*	kernel	[2]	[kernel_y, kernel_x]
Attribute	int32_t*	stride	[2]	[stride_y, stride_x]
Attribute	int32_t*	pad	[4]	[pad_top, pad_bottom, pad_left, pad_right]
Output	in_out_t*	output	[N,OH,OW,C]	Output tensor 4D

Operation Function:

```
ERROR_IF(kernel_y < 1 || kernel_x < 1); // kernel size must be >= 1
ERROR_IF(stride_y < 1 || stride_x < 1);
ERROR_IF(pad_top < 0 || pad_bottom < 0 || pad_left < 0 || pad_right < 0);
// Padding must be less than kernel size, otherwise no
// input values will be used.
ERROR_IF(pad_right >= kernel_x || pad_left >= kernel_x);
ERROR_IF(pad_top >= kernel_y || pad_bottom >= kernel_y);
ERROR_IF(OH != idiv_check(IH + pad_top + pad_bottom - kernel_y, stride_y) + 1);
ERROR_IF(OW != idiv_check(IW + pad_left + pad_right - kernel_x, stride_x) + 1);

for_each(0 <= n < N, 0 <= oy < H, 0 <= ox < W, 0 <= c < C ) {
    in_out_t acc = minimum_value<in_out_t>;
    iy = oy * stride_y - pad_top;
    ix = ox * stride_x - pad_left;
    for_each( 0 <= ky < kernel_y, 0 <= kx < kernel_x ) {
        y = iy + ky;
        x = ix + kx;
        if (y >= 0 && y < IH && x >= 0 && x < IW) {
            in_out_t value = tensor_read<in_out_t>(input, [N,IH,IW,C], [n,y,x,c]);
            acc = apply_max(acc, value);
        }
    }
    tensor_write<in_out_t>(output, [N,OH,OW,C], [n,oy,ox,c], acc);
}
```

Supported Data Types:

Profile	Mode	in_out_t
Any	signed 8	int8_t
Any	16-bit	int16_t
MI, MT	fp16	fp16_t
MI, MT	bf16	bf16_t
MI, MT	fp32	fp32_t

2.3.10. RFFT2D

Performs a batched 2D real-valued Fast Fourier Transform over the input where the input tensor consists of real values producing complex valued output. The complex output values will be split into the output_real and output_imag tensor arguments. RFFT2D takes advantage of Hermitian symmetry to only calculate the first half of the output. Imaginary values with locations h=0,H/2 or w=0,W/2 are zero.

$$output[h][w] = \sum_{m=0}^{H-1} \sum_{n=0}^{W-1} input[m][n] * \exp\left(-2\pi i \left(\frac{mh}{H} + \frac{nw}{W}\right)\right)$$

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input	[N,H,W]	Real input. H,W must be powers of two.
Output	in_out_t*	output_real	[N,H/2 + 1,W/2 + 1]	Real part of the complex output
Output	in_out_t*	output_imag	[N,H/2 + 1,W/2 + 1]	Imaginary part of the complex output.

Operation Function

```
ERROR_IF(!power_of_two(H));
ERROR_IF(!power_of_two(W));

for_each(0 <= n < N, 0 <= oy < H/2 + 1, 0 <= ox < W/2 + 1) {
    in_out_t sum_real = 0.0;
    in_out_t sum_imag = 0.0;
    for_each(0 <= iy < H, 0 <= ix < W) {
        in_out_t val_real = tensor_read<in_out_t>(input_real, [N,H,W], [n,iy,ix]);
        float_t a = 2 * pi() * ((iy * oy) / H + (ix * ox) / W);
        sum_real += val_real * cos(a);
        sum_imag += -val_real * sin(a);
    }
    tensor_write<in_out_t>(output_real, [N,H,W], [n,oy,ox], sum_real);
    tensor_write<in_out_t>(output_imag, [N,H,W], [n,oy,ox], sum_imag);
}
```

Supported Data Types:

Profile	Mode	in_out_t
MI,MT	fp32_t	fp32_t

2.3.11. TRANSPOSE_CONV2D

Performs a 2D transposed convolution over the given tensor input, using the weights tensor.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_t*	input	[N,IH,IW,IC]	Input tensor
Input (MT profile) Attribute (BI/MI profiles)	weight_t*	weight	[OC,KH,KW,IC]	Weight kernel size KH x KW

Argument	Type	Name	Shape	Description
Input (MT profile) Attribute (BI/MI profiles)	out_t*	bias	[OC]	Per output channel bias data.
Attribute	int32_t*	out_pad	[4]	[out_pad_top, out_pad_bottom, out_pad_left, out_pad_right]
Attribute	int32_t*	stride	[2]	[stride_y, stride_x]
Attribute	int32_t*	out_shape	[4]	[N,OH,OW,OC]
Attribute	in_t	input_zp	-	Input tensor zero point. Must be zero for non-int8 types.
Attribute	weight_t	weight_zp	-	Weight zero point. Must be zero for non-int8 types.
Output	out_t*	output	[N,OH,OW,OC]	Output tensor

Operation Function

```
ERROR_IF(in_t != int8_t  && input_zp != 0); // Zero point only allowed for int8_t
ERROR_IF(weight_t != int8_t && weight_zp != 0);
ERROR_IF(out_pad_top < 0 || out_pad_bottom < 0);
ERROR_IF(out_pad_left < 0 || out_pad_right < 0);
ERROR_IF(stride_y < 1 || stride_x < 1);
ERROR_IF(OH != (IH - 1) * stride_y - out_pad_top - out_pad_bottom + KH);
ERROR_IF(OW != (IW - 1) * stride_x - out_pad_left - out_pad_right + KW);

for_each(index in out_shape) {
    tensor_write<out_t>(output, [N,OH,OW,OC], index, bias[index[3]])
}
for_each(0 <= n < N, 0 <= iy < IH, 0 <= ix < IW, 0 <= oc < OC,
          0 <= ic < IC, 0 <= ky < KH,  0 <= kx < KW) {
    oy = iy * stride_y - out_pad_top  + ky;
    ox = ix * stride_x - out_pad_left + kx;
    if (oy >= 0 && oy < OH && ox >= 0 && ox < OW) {
        out_t acc = tensor_read<out_t>(output, [N,OH,OW,OC], [n,oy,ox,oc]);
        out_t value = tensor_read<in_t>(input, [N,IH,IW,IC], [n,iy,ix,ic]);
        out_t weight = tensor_read<weight_t>(weight, [OC,KH,KW,IC], [oc,ky,kx,ic]);
        value = value - input_zp;
        weight = weight - weight_zp;
        acc = apply_add<out_t>(acc, value * weight);
        tensor_write<out_t>(output, [N,OH,OW,OC], [n,oy,ox,oc], acc);
    }
}
```

Supported Data Types:

Profile	Mode	in_t	weight_t	out_t
Any	signed 8x8	int8_t	int8_t	int32_t
Any	signed 8x4	int8_t	int4_t	int32_t
Any	signed 16x8	int16_t	int8_t	int48_t
MI, MT	fp16 with fp16 accumulate	fp16_t	fp16_t	fp16_t
MI, MT	fp16 with fp32 accumulate	fp16_t	fp16_t	fp32_t
MI, MT	bf16 with fp32 accumulate	bf16_t	bf16_t	fp32_t
MI, MT	fp32	fp32_t	fp32_t	fp32_t

2.4. Activation Functions

2.4.1. CLAMP

Clamp to an arbitrary minimum and maximum value. Maximum and minimum values are

specified as values in the range of the input type. No zero point subtraction is done to the values, thus to clamp to the zero point value, the zero point itself should be supplied as the minimum value.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	Input	shape	Input tensor
Attribute	in_out_t	min_val	-	minimum clip value
Attribute	in_out_t	max_val	-	maximum clip value
Output	in_out_t*	Output	shape	Output tensor of same type and shape as input

Operation Function:

```
ERROR_IF(max_val < min_val);
for_each(index in shape) {
    in_out_t value = tensor_read<in_out_t>(input, shape, index);
    value = apply_clip<in_out_t>(value, min_val, max_val);
    tensor_write<in_out_t>(output, shape, index, value);
}
```

Supported Data Types:

Profile	Mode	in_out_t
Any	signed 8	int8_t
Any	signed 16	int16_t
MI, MT	fp16	fp16_t
MI, MT	bf16	bf16_t
MI, MT	fp32	fp32_t

2.4.2. SIGMOID

Sigmoid function: output = 1 / (1 + exp(-input))

For quantized integer data types, the TABLE operator should be used instead with the following definition.

The sigmoid table has 513 entries each of 16-bit precision and covering the input range -16.0 to +16.0 in steps of 1/16.

```
int16_t sigmoid_reference(int16_t x) { // input x range is -256 to + 256 inclusive
    F64 v = (double)x / (double)16;
    v = 1.0/(1.0 + exp(-v));
    return round_to_nearest_int(32768.0 * v);
}

generate_lookup_table(&sigmoid_table, &sigmoid_reference);
```

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	Input	shape	Input tensor
Output	in_out_t*	Output	shape	Output tensor of same type and shape as input

Supported Data Types:

Profile	Mode	in_out_t
MI, MT	fp16	fp16_t
MI, MT	bf16	bf16_t
MI, MT	fp32	fp32_t

2.4.3. TANH

Parameterized hyperbolic tangent.

For quantized integer data types, the TABLE operator should be used instead with the following definition.

The tanh_table has 513 entries each of 16-bit precision and covering the input range -8.0 to +8.0 in steps of 1/32. The table is specified by:

```
int16_t tanh_reference(int16_t x) {  // input x range is -256 to +256 inclusive
    F64 v = (double)x/(double)32;
    v = exp(-2.0*v);
    v = (1.0-v)/(1.0+v);
    return round_to_nearest_int(32768.0 * v);
}

generate_lookup_table(&tanh_table, &tanh_reference);
```

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	Input	shape	Input tensor
Output	in_out_t*	Output	shape	Output tensor of same type and shape as input

Supported Data Types:

Profile	Mode	in_out_t
MI, MT	fp16	fp16_t
MI, MT	bf16	bf16_t
MI, MT	fp32	fp32_t

2.5. Elementwise Binary Operators

2.5.1. ADD

Elementwise addition of input1 and input2. Axis of size 1 will be broadcast, as necessary. Rank of input tensors must match.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input1	shape1	Input tensor
Input	in_out_t*	input2	shape2	Input tensor with the same rank as input1
Output	in_out_t*	output	shape	Output tensor with broadcast shape if necessary

Operation Function:

```
for_each(index in shape) {
    index1 = apply_broadcast(shape, shape1, index);
    index2 = apply_broadcast(shape, shape2, index);
    in_out_t value1 = tensor_read<in_out_t>(input1, shape1, index1);
    in_out_t value2 = tensor_read<in_out_t>(input2, shape2, index2);
    in_out_t result = apply_add<in_out_t>(value1, value2);
    tensor_write<in_out_t>(output, shape, index, result);
}
```

Supported Data Types:

Profile	Mode	in_out_t
Any	signed 32	int32_t
MI, MT	fp16	fp16_t
MI, MT	bf16	bf16_t
MI, MT	fp32	fp32_t

2.5.2. ARITHMETIC_RIGHT_SHIFT

Elementwise arithmetic right shift of input1 by the amount specified in input2. Axis of size 1 will be broadcast, as necessary. Rank of input tensors must match.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input1	shape1	Input tensor
Input	in_out_t*	input2	shape2	Input tensor with the same rank as input1
Attribute	bool_t	round	-	If true then the shift is rounded
Output	in_out_t*	output	shape	Output tensor with broadcast shape if necessary

Operation Function:

```
for_each(index in shape) {
    index1 = apply_broadcast(shape, shape1, index);
    index2 = apply_broadcast(shape, shape2, index);
    in_out_t value1 = tensor_read<in_out_t>(input1, shape1, index1);
    in_out_t value2 = tensor_read<in_out_t>(input2, shape2, index2);

    // Ensure that shift amount is appropriate for the data type
    REQUIRE((in_out_t == int32_t && 0 <= value2 && value2 <= 31) ||
            (in_out_t == int16_t && 0 <= value2 && value2 <= 15) ||
            (in_out_t == int8_t && 0 <= value2 && value2 <= 7));

    in_out_t result = value1 >> value2;
    if (round == true && value2 > 0 && (value1 >> (value2 - 1)) & 1 != 0) {
        result = result + 1;
    }
    result = apply_clip<in_out_t>(result, minimum<in_out_t>, maximum<in_out_t>);
    tensor_write<in_out_t>(output, shape, index, result);
}
```

Supported Data Types:

Profile	Mode	in_out_t
Any	signed 8	int8_t
Any	signed 16	int16_t
Any	signed 32	int32_t

2.5.3. BITWISE_AND

Elementwise bitwise AND of input1 and input2. Axis of size 1 will be broadcast as necessary. Rank of input tensors must match.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input1	shape1	Input tensor
Input	in_out_t*	input2	shape2	Input tensor with the same rank as input1
Output	in_out_t*	output	shape	Output tensor of same type as the input tensors, with broadcast shape if necessary

Operation Function:

```
for_each(index in shape) {
    index1 = apply_broadcast(shape, shape1, index);
    index2 = apply_broadcast(shape, shape2, index);
    in_out_t value1 = tensor_read<in_out_t>(input1, shape1, index1);
    in_out_t value2 = tensor_read<in_out_t>(input2, shape2, index2);
    in_out_t result = value1 & value2;
    tensor_write<in_out_t>(output, shape, index, result);
}
```

Supported Data Types:

Profile	Mode	in_out_t
Any	signed 8	int8_t
Any	signed 16	int16_t
Any	signed 32	int32_t

2.5.4. BITWISE_OR

Elementwise bitwise OR of input1 and input2. Axis of size 1 will be broadcast as necessary. Rank of input tensors must match.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input1	shape1	Input tensor
Input	in_out_t*	input2	shape2	Input tensor with the same rank as input1
Output	in_out_t*	output	shape	Output tensor with broadcast shape if necessary

Operation Function:

```
for_each(index in shape) {
    index1 = apply_broadcast(shape, shape1, index);
    index2 = apply_broadcast(shape, shape2, index);
    in_out_t value1 = tensor_read<in_out_t>(input1, shape1, index1);
    in_out_t value2 = tensor_read<in_out_t>(input2, shape2, index2);
    in_out_t result = value1 | value2;
    tensor_write<in_out_t>(output, shape, index, result);
}
```

Supported Data Types:

Profile	Mode	in_out_t
Any	signed 8	int8_t
Any	signed 16	int16_t
Any	signed 32	int32_t

2.5.5. BITWISE_XOR

Elementwise bitwise XOR of input1 and input2. Axis of size 1 will be broadcast as necessary. Rank of input tensors must match.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input1	shape1	Input tensor

Argument	Type	Name	Shape	Description
Input	in_out_t*	input2	shape2	Input tensor with the same rank as input1
Output	in_out_t*	output	shape	Output tensor with broadcast shape if necessary

Operation Function:

```
for_each(index in shape) {
    index1 = apply_broadcast(shape, shape1, index);
    index2 = apply_broadcast(shape, shape2, index);
    in_out_t value1 = tensor_read<in_out_t>(input1, shape1, index1);
    in_out_t value2 = tensor_read<in_out_t>(input2, shape2, index2);
    in_out_t result = value1 ^ value2;
    tensor_write<in_out_t>(output, shape, index, result);
}
```

Supported Data Types:

Profile	Mode	in_out_t
Any	signed 8	int8_t
Any	signed 16	int16_t
Any	signed 32	int32_t

2.5.6. INTDIV

Elementwise integer divide of input1 by input2. The result of the divide is truncated towards zero. Expected use is for operations on non-scaled integers. Floating point divide should use RECIPROCAL and MUL. Quantized integer divide should use TABLE (for 1/x) and MUL.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input1	shape1	Input tensor
Input	in_out_t*	input2	shape2	Input tensor with the same rank as input1
Output	in_out_t*	output	shape	Output tensor with broadcast shape if necessary

Operation Function:

```
for_each(index in shape) {
    index1 = apply_broadcast(shape, shape1, index);
    index2 = apply_broadcast(shape, shape2, index);
    in_out_t value1 = tensor_read<in_out_t>(input1, shape1, index1);
    in_out_t value2 = tensor_read<in_out_t>(input2, shape2, index2);
    REQUIRE(value2 != 0);
    // This catches the case where we divide minimum<in_out_t> by -1
    // which is not representable in two's complement
    REQUIRE((int64_t)value1 / value2 <= maximum<in_out_t>);
    in_out_t result = value1 / value2;
    tensor_write<in_out_t>(output, shape, index, result);
}
```

Supported Data Types:

Profile	Mode	in_out_t
Any	signed 32	int32_t

2.5.7. LOGICAL_AND

Elementwise logical AND of input1 and input2. Axis of size 1 will be broadcast, as necessary. Rank of input tensors must match.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input1	shape1	Input tensor
Input	in_out_t*	input2	shape2	Input tensor with the same rank as input1
Output	in_out_t*	output	shape	Output tensor with broadcast shape if necessary

Operation Function:

```
for_each(index in shape) {
    index1 = apply_broadcast(shape, shape1, index);
    index2 = apply_broadcast(shape, shape2, index);
    in_out_t value1 = tensor_read<in_out_t>(input1, shape1, index1);
    in_out_t value2 = tensor_read<in_out_t>(input2, shape2, index2);
    in_out_t result = value1 && value2;
    tensor_write<in_out_t>(output, shape, index, result);
}
```

Supported Data Types:

Profile	Mode	in_out_t
Any	Bool	bool_t

2.5.8. LOGICAL_LEFT_SHIFT

Elementwise left shift of input1 and input2. Axis of size 1 will be broadcast, as necessary. Rank of input tensors must match.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input1	shape1	Input tensor
Input	in_out_t*	input2	shape2	Input tensor with the same rank as input1
Output	in_out_t*	output	shape	Output tensor with broadcast shape if necessary

Operation Function:

```
for_each(index in shape) {
    index1 = apply_broadcast(shape, shape1, index);
    index2 = apply_broadcast(shape, shape2, index);
    in_out_t value1 = tensor_read<in_out_t>(input1, shape1, index1);
    in_out_t value2 = tensor_read<in_out_t>(input2, shape2, index2);
    REQUIRE(0 <= value2 && value2 <= 31);
    in_out_t result = value1 << value2;
    tensor_write<in_out_t>(output, shape, index, result);
}
```

Supported Data Types:

Profile	Mode	in_out_t
Any	signed 8	int8_t
Any	signed 16	int16_t
Any	signed 32	int32_t

2.5.9. LOGICAL_RIGHT_SHIFT

Elementwise logical right shift of input1 by the amount specified in input2. Axis of size 1 will be broadcast, as necessary. Rank of input tensors must match.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input1	shape1	Input tensor
Input	in_out_t*	input2	shape2	Input tensor with the same rank as input1
Output	in_out_t*	output	shape	Output tensor with broadcast shape if necessary

Operation Function:

```
for_each(index in shape) {
    index1 = apply_broadcast(shape, shape1, index);
    index2 = apply_broadcast(shape, shape2, index);
    in_out_t value1 = tensor_read<in_out_t>(input1, shape1, index1);
    in_out_t value2 = tensor_read<in_out_t>(input2, shape2, index2);
    REQUIRE(0 <= value2 && value2 <= 31);
    in_out_t result = (in_out_t)((unsigned in_out_t)value1 >> value2);
    tensor_write<in_out_t>(output, shape, index, result);
}
```

Supported Data Types:

Profile	Mode	in_out_t
Any	signed 8	int8_t
Any	signed 16	int16_t
Any	signed 32	int32_t

2.5.10. LOGICAL_OR

Elementwise logical OR of input1 and input2. Axis of size 1 will be broadcast as necessary. Rank of input tensors must match.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input1	shape1	Input tensor
Input	in_out_t*	input2	shape2	Input tensor with the same rank as input1
Output	in_out_t*	output	shape	Output tensor with broadcast shape if necessary

Operation Function:

```
for_each(index in shape) {
    index1 = apply_broadcast(shape, shape1, index);
    index2 = apply_broadcast(shape, shape2, index);
    in_out_t value1 = tensor_read<in_out_t>(input1, shape1, index1);
    in_out_t value2 = tensor_read<in_out_t>(input2, shape2, index2);
    in_out_t result = value1 || value2;
    tensor_write<in_out_t>(output, shape, index, result);
}
```

Supported Data Types:

Profile	Mode	in_out_t
Any	Bool	bool_t

2.5.11. LOGICAL_XOR

Elementwise logical XOR of input1 and input2. Axis of size 1 will be broadcast as necessary. Rank of input tensors must match.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input1	shape1	Input tensor
Input	in_out_t*	input2	shape2	Input tensor with the same rank as input1
Output	in_out_t*	output	shape	Output tensor of same type as the input tensors, with broadcast shape if necessary

Operation Function:

```
for_each(index in shape) {
    index1 = apply_broadcast(shape, shape1, index);
    index2 = apply_broadcast(shape, shape2, index);
    in_out_t value1 = tensor_read<in_out_t>(input1, shape1, index1);
    in_out_t value2 = tensor_read<in_out_t>(input2, shape2, index2);
    in_out_t result = value1 != value2;
    tensor_write<in_out_t>(output, shape, index, result);
}
```

Supported Data Types:

Profile	Mode	in_out_t
Any	Bool	bool_t

2.5.12. MAXIMUM

Elementwise max of input1 and input2. Axis of size 1 will be broadcast, as necessary. Rank of input tensors must match.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input1	shape1	Input tensor
Input	in_out_t*	input2	shape2	Input tensor with the same rank as input1
Output	in_out_t*	output	shape	Output tensor with broadcast shape if necessary

Operation Function:

```
for_each(index in shape) {
    index1 = apply_broadcast(shape, shape1, index);
    index2 = apply_broadcast(shape, shape2, index);
    in_out_t value1 = tensor_read<in_out_t>(input1, shape1, index1);
    in_out_t value2 = tensor_read<in_out_t>(input2, shape2, index2);
    in_out_t result = apply_max(value1, value2);
    tensor_write<in_out_t>(output, shape, index, result);
}
```

Supported Data Types:

Profile	Mode	in_out_t
Any	signed 32	int32_t
MI, MT	fp16	fp16_t
MI, MT	bf16	bf16_t
MI, MT	fp32	fp32_t

2.5.13. MINIMUM

Elementwise minimum of input1 and input2. Axis of size 1 will be broadcast, as necessary. Rank of input tensors must match.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input1	shape1	Input tensor
Input	in_out_t*	input2	shape2	Input tensor with the same rank as input1
Output	in_out_t*	output	shape	Output tensor with broadcast shape if necessary

Operation Function:

```
for_each(index in shape) {
    index1 = apply_broadcast(shape, shape1, index);
    index2 = apply_broadcast(shape, shape2, index);
    in_out_t value1 = tensor_read<in_out_t>(input1, shape1, index1);
    in_out_t value2 = tensor_read<in_out_t>(input2, shape2, index2);
    in_out_t result = apply_min(value1, value2);
    tensor_write<in_out_t>(output, shape, index, result);
}
```

Supported Data Types:

Profile	Mode	in_out_t
Any	signed 32	int32_t
MI, MT	fp16	fp16_t
MI, MT	bf16	bf16_t
MI, MT	fp32	fp32_t

2.5.14. MUL

Elementwise multiplication (Hadamard product) of input1 and input2. Axis of size 1 will be broadcast, as necessary. Rank of input tensors must match.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_t*	input1	shape1	Input tensor
Input	in_t*	input2	shape2	Input tensor with the same rank as input1
Input (MT profile) Attribute (BI/MI profiles)	uint6_t	shift	-	Result right shift (int32_t data type only)

Argument	Type	Name	Shape	Description
Output	out_t*	output	shape	Output tensor with broadcast shape if necessary

Operation Function:

```
ERROR_IF(in_t != int32_t && shift > 0);
for_each(index in shape) {
    index1 = apply_broadcast(shape, shape1, index);
    index2 = apply_broadcast(shape, shape2, index);
    in_t value1 = tensor_read<in_t>(input1, shape1, index1);
    in_t value2 = tensor_read<in_t>(input2, shape2, index2);
    out_t result;
    if (in_t == int32_t && shift > 0) {
        int64_t product = (int64_t)value1 * (int64_t)value2;
        int64_t round   = (int64_t)1 << (shift-1);
        product = (product + round) >> shift;
        REQUIRE(product >= minimum<int32_t> && product <= maximum<int32_t>)
        result = product;
    } else {
        result = value1 * value2;  // low 32-bits of result for int32_t
    }
    tensor_write<out_t>(output, shape, index, result);
}
```

Supported Data Types:

Profile	Mode	in_t	out_t
Any	signed 8	int8_t	int32_t
Any	signed 16	int16_t	int32_t
Any	signed 32	int32_t	int32_t
MI, MT	fp16	fp16_t	fp16_t
MI, MT	bf16	bf16_t	bf16_t
MI, MT	fp32	fp32_t	fp32_t

2.5.15. POW

Elementwise input1 value raised to the power of input2. Axis of size 1 will be broadcast, as necessary. Rank of input tensors must match.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input1	shape1	Input tensor from 1 to 4 dims
Input	in_out_t*	input2	shape2	Input tensor with the same rank as input1
Output	in_out_t*	output	shape	Output tensor of same type as the input tensors, with broadcast shape if necessary

Operation Function:

```
for_each(index in shape) {
    index1 = apply_broadcast(shape, shape1, index);
    index2 = apply_broadcast(shape, shape2, index);
    in_out_t value1 = tensor_read<in_out_t>(input1, shape1, index1);
    in_out_t value2 = tensor_read<in_out_t>(input2, shape2, index2);
    in_out_t result = apply_pow<in_out_t>(value1, value2);
    tensor_write<in_out_t>(output, shape, index, result);
}
```

Supported Data Types:

Profile	Mode	in_out_t
MI, MT	fp16	fp16_t
MI, MT	bf16	bf16_t
MI, MT	fp32	fp32_t

2.5.16. SUB

Elementwise subtraction of input1 and input2. Axis of size 1 will be broadcast as necessary. Rank of input tensors must match.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input1	shape1	Input tensor
Input	in_out_t*	input2	shape2	Input tensor with the same rank as input1

Argument	Type	Name	Shape	Description
Output	in_out_t*	output	shape	Output tensor with broadcast shape if necessary

Operation Function:

```
for_each(index in shape) {
    index1 = apply_broadcast(shape, shape1, index);
    index2 = apply_broadcast(shape, shape2, index);
    in_out_t value1 = tensor_read<in_out_t>(input1, shape1, index1);
    in_out_t value2 = tensor_read<in_out_t>(input2, shape2, index2);
    in_out_t result = apply_sub<in_out_t>(value1, value2);
    tensor_write<in_out_t>(output, shape, index, result);
}
```

Supported Data Types:

Profile	Mode	in_out_t
Any	signed 32	int32_t
MI, MT	fp16	fp16_t
MI, MT	bf16	bf16_t
MI, MT	fp32	fp32_t

2.5.17. TABLE

Table lookup operation. For int8_t TABLE operation, perform a 256 entry table lookup returning an int8_t value. For int16_t tables, the int16_t input is treated as a fixed-point 9.7 value. The most significant 9 bits are used to index into the table. The fractional 7 bits are used to interpolate based on table[index] and table[index+1]. For int16_t inputs, the TABLE operator returns a 16.7 interpolated value in an int32_t. This value can then be input to the RESCALE operator to scale to the required output data type. Note that int16_t table has 513 values to handle table[index+1] when index=511.

An int16_t to int16_t table lookup can be constructed in TOSA as follows:

- Use the TABLE operator to produce a fixed point 16.7 interpolated result
- Use RESCALE (in_t=int32_t, out_t=int16_t, scale=1<<14, shift=21) to scale the output to int16_t range (or alternate scale as required)

Arguments:

Argument	Type	Name	Shape	Description
Input	in_t*	Input	shape	Input tensor

Argument	Type	Name	Shape	Description
Input (MT profile) Attribute (BI/MI profiles)	table_t*	table	[TABLE_SIZE]	Lookup table tensor
Output	out_t*	output	shape	Output tensor

Operation Function:

```
REQUIRE(length(table) == TABLE_SIZE);
for_each(index in shape) {
    in_t value = tensor_read<in_t>(input, shape, index);
    out_t result;
    if (in_t == int8_t) {
        // value is a signed int, convert to a 0 based index
        result = table[value + 128];
    } else {
        result = apply_lookup(table, value);
    }
    tensor_write<out_t>(output, shape, index, result);
}
```

Supported Data Types:

Profile	Mode	in_t	table_t	TABLE_SIZE	out_t
Any	signed 8	int8_t	int8_t	256	int8_t
Any	signed 16	int16_t	int16_t	513	int32_t

2.6. Elementwise Unary Operators

2.6.1. ABS

Elementwise absolute value operation

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input1	shape	Input tensor
Output	in_out_t*	output	shape	Output tensor of same type, size as the input tensor

Floating-point behavior:

Input	-infinity	+infinity	-0	+0	NaN
Output	+infinity	+infinity	+0	+0	NaN

Operation Function:

```
for_each(index in shape) {
    in_out_t value1 = tensor_read<in_out_t>(input1, shape, index);
    if (in_out_t == float_t && value1 == -0.0) {
        value1 = 0.0;
    }
    if (value1 < 0.0)
        value1 = apply_sub<in_out_t>(0, value1);
    tensor_write<in_out_t>(output, shape, index, value1);
}
```

Supported Data Types:

Profile	Mode	in_out_t
Any	signed 32	int32_t
MI, MT	floating-point	float_t

2.6.2. BITWISE_NOT

Elementwise bitwise NOT of input tensor.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input1	shape	Input tensor
Output	in_out_t*	output	shape	Output tensor of same type, size as the input tensor

Operation Function:

```
for_each(index in shape) {
    in_out_t value1 = tensor_read<in_out_t>(input1, shape, index);
    in_out_t result = ~value1;
    tensor_write<in_out_t>(output, shape, index, result);
}
```

Supported Data Types:

Profile	Mode	in_out_t
Any	signed 8	int8_t
Any	signed 16	int16_t
Any	signed 32	int32_t

2.6.3. CEIL

Elementwise ceiling operation

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input1	shape	Input tensor
Output	in_out_t*	output	shape	Output tensor of same type, size as the input tensor

Floating-point behavior:

Input	-infinity	+infinity	-0	+0	NaN
Output	-infinity	+infinity	-0	+0	NaN

Operation Function:

```
for_each(index in shape) {
    in_out_t value1 = tensor_read<in_out_t>(input1, shape, index);
    in_out_t result = apply_ceil<in_out_t>(value1);
    tensor_write<in_out_t>(output, shape, index, result);
}
```

Supported Data Types:

Profile	Mode	in_out_t
MI, MT	floating-point	float_t

2.6.4. CLZ

Elementwise count leading zeros operation

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input1	shape	Input tensor

Argument	Type	Name	Shape	Description
Output	in_out_t*	output	shape	Output tensor of same type, size as the input tensor

Operation Function:

```
for_each(index in shape) {
    in_out_t value1 = tensor_read<in_out_t>(input1, shape, index);
    in_out_t result = count_leading_zeros(value1);
    tensor_write<in_out_t>(output, shape, index, result);
}
```

Supported Data Types:

Profile	Mode	in_out_t
Any	signed 32	int32_t

2.6.5. EXP

Elementwise e to the x operation

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input1	shape	Input tensor
Output	in_out_t*	output	shape	Output tensor of same type, size as the input tensor

Floating-point behavior:

Input	-infinity	+infinity	-0	+0	NaN
Output	+0	+infinity	1	1	NaN

Operation Function:

```
for_each(index in shape) {
    in_out_t value1 = tensor_read<in_out_t>(input1, shape, index);
    in_out_t result = apply_exp<in_out_t>(value1);
    tensor_write<in_out_t>(output, shape, index, result);
}
```

Supported Data Types:

Profile	Mode	in_out_t
MI, MT	fp16	fp16_t
MI, MT	bf16	bf16_t
MI, MT	fp32	fp32_t

2.6.6. FLOOR

Elementwise floor operation

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input1	shape	Input tensor
Output	in_out_t*	output	shape	Output tensor of same type, size as the input tensor

Floating-point behavior:

Input	-infinity	+infinity	-0	+0	NaN
Output	-infinity	+infinity	-0	+0	NaN

Operation Function:

```
for_each(index in shape) {
    in_out_t value1 = tensor_read<in_out_t>(input1, shape, index);
    in_out_t result = apply_floor<in_out_t>(value1);
    tensor_write<in_out_t>(output, shape, index, result);
}
```

Supported Data Types:

Profile	Mode	in_out_t
MI, MT	fp16	fp16_t
MI, MT	bf16	bf16_t
MI, MT	fp32	fp32_t

2.6.7. LOG

Elementwise natural logarithm operation

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input1	shape	Input tensor
Output	in_out_t*	output	shape	Output tensor of same type, size as the input tensor

Floating-point behavior:

Input	-infinity	+infinity	-0	+0	NaN
Output	NaN	+infinity	-infinity	-infinity	NaN

Operation Function:

```
for_each(index in shape) {
    in_out_t value1 = tensor_read<in_out_t>(input1, shape, index);
    in_out_t result = apply_log<in_out_t>(value1);
    tensor_write<in_out_t>(output, shape, index, result);
}
```

Supported Data Types:

Profile	Mode	in_out_t
MI, MT	fp16	fp16_t
MI, MT	bf16	bf16_t
MI, MT	fp32	fp32_t

2.6.8. LOGICAL_NOT

Elementwise logical NOT of input.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input1	shape	Input tensor
Output	in_out_t*	output	shape	Output tensor of same type, size as the input tensor

Operation Function:

```
for_each(index in shape) {
    in_out_t value1 = tensor_read<in_out_t>(input1, shape1, index);
    in_out_t result = !value1;
    tensor_write<in_out_t>(output, shape, index, result);
}
```

Supported Data Types:

Profile	Mode	in_out_t
Any	bool	bool_t

2.6.9. NEGATE

Elementwise negation operation

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input1	shape	Input tensor
Attribute	in_out_t	input1_zp	-	Input 1 zero point. Must be zero for non-int8 types.
Attribute	in_out_t	output_zp	-	Output zero point. Must be zero for non-int8 types.
Output	in_out_t*	output	shape	Output tensor of same type, size as the input tensor

Floating-point behavior:

Input	-infinity	+infinity	-0	+0	NaN
Output	+infinity	-infinity	+0	-0	NaN

Operation Function:

```
ERROR_IF(in_out_t != int8_t && input1_zp != 0) // Zero point only for int8_t
ERROR_IF(in_out_t != int8_t && output_zp != 0) // Zero point only for int8_t
for_each(index in shape) {
    in_out_t value1 = tensor_read<in_out_t>(input1, shape, index);
    acc_t value = (acc_t)value1 - input1_zp;
    value = apply_sub<acc_t>(0, value);
    in_out_t result = (in_out_t)apply_clip<acc_t>(value + output_zp,
minimum<in_out_t>, maximum<in_out_t>);
    tensor_write<in_out_t>(output, shape, index, result);
}
```

Supported Data Types:

Profile	Mode	in_out_t	acc_t
Any	signed 8	int8_t	int32_t
Any	signed 16	int16_t	int32_t
Any	signed 32	int32_t	int32_t
MI, MT	fp16	fp16_t	fp16_t
MI, MT	bf16	bf16_t	bf16_t
MI, MT	fp32	fp32_t	fp32_t

2.6.10. RECIPROCAL

Elementwise reciprocal operation. For integer operation, a TABLE should be used with the appropriate ranges.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input1	shape	Input tensor
Output	in_out_t*	output	shape	Output tensor of same type, size as the input tensor

Floating-point behavior:

Input	-infinity	+infinity	-0	+0	NaN
Output	-0	+0	-infinity	+infinity	NaN

Operation Function:

```
for_each(index in shape) {
    in_out_t value1 = tensor_read<in_out_t>(input1, shape1, index);
    in_out_t result = 1.0 / value1;
    tensor_write<in_out_t>(output, shape, index, result);
}
```

Supported Data Types:

Profile	Mode	in_out_t
MI, MT	fp16	fp16_t
MI, MT	bf16	bf16_t
MI, MT	fp32	fp32_t

2.6.11. RSQRT

Elementwise reciprocal square root operation. For integer operation, a TABLE should be used with the appropriate ranges.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input1	shape	Input tensor
Output	in_out_t*	output	shape	Output tensor of same type, size as the input tensor

Floating-point behavior:

Input	-infinity	+infinity	-0	+0	NaN
Output	NaN	+0	-infinity	+infinity	NaN

Operation Function:

```
for_each(index in shape) {
    in_out_t value1 = tensor_read<in_out_t>(input1, shape1, index);
    in_out_t result;
    if (value1 < 0) {
        result = NaN;
    }
    else {
        result = 1.0 / apply_sqrt<in_out_t>(value1);
    }
    tensor_write<in_out_t>(output, shape, index, result);
}
```

Supported Data Types:

Profile	Mode	in_out_t
MI, MT	fp16	fp16_t
MI, MT	bf16	bf16_t
MI, MT	fp32	fp32_t

2.7. Elementwise Ternary Operators

2.7.1. SELECT

Elementwise select of the output based on a condition.

Arguments:

Argument	Type	Name	Shape	Description
Input	cmp_t	input1	shape1	Input selector tensor
Input	in_out_t*	input2	shape2	Input value tensor if input1 is True
Input	in_out_t*	input3	shape3	Input value tensor if input1 is False
Output	in_out_t*	output	shape	Output tensor of same type as input2 and input3, with broadcast shape if necessary

Operation Function:

```
for_each(index in shape) {
    index1 = apply_broadcast(shape, shape1, index);
    index2 = apply_broadcast(shape, shape2, index);
    index3 = apply_broadcast(shape, shape3, index);
    cmp_t value1 = tensor_read<cmp_t>(input1, shape1, index1);
    in_out_t value2 = tensor_read<in_out_t>(input2, shape2, index2);
    in_out_t value3 = tensor_read<in_out_t>(input3, shape3, index3);
    in_out_t result;
    if (value1) {
         result = value2;
    } else {
         result = value3;
    }
    tensor_write<in_out_t>(output, shape, index, result);
}
```

Supported Data Types:

Profile	Mode	cmp_t	in_out_t
Any	Boolean	bool_t	bool_t
Any	signed 8	bool_t	int8_t
Any	signed 16	bool_t	int16_t
Any	signed 32	bool_t	int32_t
MI, MT	bool_t	fp16	fp16_t
MI, MT	bool_t	bf16	bf16_t
MI, MT	bool_t	fp32	fp32_t

2.8. Comparison Operators

2.8.1. EQUAL

Elementwise comparison operation

Arguments:

Argument	Type	Name	Shape	Description
Input	in_t*	input1	shape1	Input tensor
Input	in_t*	input2	shape2	Input tensor with the same rank as input1
Output	out_t*	output	shape	Output tensor with broadcast shape if necessary

Operation Function:

```
for_each(index in shape) {
    index1 = apply_broadcast(shape, shape1, index);
    index2 = apply_broadcast(shape, shape2, index);
    in_t value1 = tensor_read<in_t>(input1, shape1, index1);
    in_t value2 = tensor_read<in_t>(input2, shape2, index2);
    out_t result;
    if (isNaN(value1) || isNaN(value2))
        result = False;
    else
        result = (value1 == value2) ? True : False;
    tensor_write<out_t>(output, shape, index, result);
}
```

Supported Data Types:

Profile	Mode	in_t	out_t
Any	signed 32	int32_t	bool_t
MI, MT	fp16	fp16_t	bool_t
MI, MT	bf16	bf16_t	bool_t
MI, MT	fp32	fp32_t	bool_t

2.8.2. GREATER

Elementwise greater than comparison operation

Arguments:

Argument	Type	Name	Shape	Description
Input	in_t*	input1	shape1	Input tensor
Input	in_t*	input2	shape2	Input tensor with the same rank as input1
Output	out_t*	output	shape	Output tensor with broadcast shape if necessary

Operation Function:

```
for_each(index in shape) {
    index1 = apply_broadcast(shape, shape1, index);
    index2 = apply_broadcast(shape, shape2, index);
    in_t value1 = tensor_read<in_t>(input1, shape1, index1);
    in_t value2 = tensor_read<in_t>(input2, shape2, index2);
    out_t result;
    if (isNaN(value1) || isNaN(value2))
        result = False;
    else
        result = (value1 > value2) ? True : False;
    tensor_write<out_t>(output, shape, index, result);
}
```

Supported Data Types:

Profile	Mode	in_t	out_t
Any	signed 32	int32_t	bool_t
MI, MT	fp16	fp16_t	bool_t
MI, MT	bf16	bf16_t	bool_t

Profile	Mode	in_t	out_t
MI, MT	fp32	fp32_t	bool_t

2.8.3. GREATER_EQUAL

Elementwise comparison operation

Arguments:

Argument	Type	Name	Shape	Description
Input	in_t*	input1	shape1	Input tensor
Input	in_t*	input2	shape2	Input tensor with the same rank as input1
Output	out_t*	output	shape	Output tensor with broadcast shape if necessary

Operation Function:

```
for_each(index in shape) {
    index1 = apply_broadcast(shape, shape1, index);
    index2 = apply_broadcast(shape, shape2, index);
    in_t value1 = tensor_read<in_t>(input1, shape1, index1);
    in_t value2 = tensor_read<in_t>(input2, shape2, index2);
    out_t result;
    if (isNaN(value1) || isNaN(value2))
        result = False;
    else
        result = (value1 >= value2) ? True : False;
    tensor_write<out_t>(output, shape, index, result);
}
```

Supported Data Types:

Profile	Mode	in_t	out_t
Any	signed 32	int32_t	bool_t
MI, MT	fp16	fp16_t	bool_t
MI, MT	bf16	bf16_t	bool_t
MI, MT	fp32	fp32_t	bool_t

2.9. Reduction Operators

2.9.1. REDUCE_ALL

Reduce a tensor along the given axis with a logical AND operation

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input	shape1	Input tensor with rank from 1 to 4
Attribute	int32_t	axis	-	Axis to reduce, in range from 0 to rank(shape1)-1
Output	in_out_t*	output	shape	Output tensor. Same rank as the input tensor.

Operation Function:

```
ERROR_IF(axis < 0  || axis >= rank(shape1));
ERROR_IF(shape[axis] != 1);

// Initialize output state to true
for_each(index in shape) {
    tensor_write<in_out_t>(output, shape, index, true);
}
for_each(index in shape1) {
    out_index = index;
    out_index[axis] = 0;
    in_out_t value = tensor_read<in_out_t>(input, shape1, index);
    in_out_t state = tensor_read<in_out_t>(output, shape, out_index);
    state      = state && value;
    tensor_write<in_out_t>(output, shape, out_index, state);
}
```

Supported Data Types:

Profile	Mode	in_out_t
Any	Boolean	bool_t

2.9.2. REDUCE_ANY

Reduce a tensor along the given axis with a logical OR operation

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input	shape1	Input tensor with rank from 1 to 4
Attribute	int32_t	axis	-	Axis to reduce, in range from 0 to rank(shape1)-1
Output	in_out_t*	output	shape	Output tensor. Same rank as the input tensor.

Operation Function:

```
ERROR_IF(axis < 0  || axis >= rank(shape1));
ERROR_IF(shape[axis] != 1);

// Initialize output state to false
for_each(index in shape) {
    tensor_write<in_out_t>(output, shape, index, false);
}
for_each(index in shape1) {
    out_index = index;
    out_index[axis] = 0;
    in_out_t value = tensor_read<in_out_t>(input, shape1, index);
    in_out_t state = tensor_read<in_out_t>(output, shape, out_index);
    state      = state || value;
    tensor_write<in_out_t>(output, shape, out_index, state);
}
```

Supported Data Types:

Profile	Mode	in_out_t
Any	Boolean	bool_t

2.9.3. REDUCE_MAX

Reduce a tensor along the given axis with a maximum operation

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input	shape1	Input tensor with rank from 1 to 4
Attribute	int32_t	axis	-	Axis to reduce, in range from 0 to rank(shape1)-1

Argument	Type	Name	Shape	Description
Output	in_out_t*	output	shape	Output tensor. Same rank as the input tensor.

Operation Function:

```
ERROR_IF(axis < 0  || axis >= rank(shape1));
ERROR_IF(shape[axis] != 1);
for_each(index in shape) {
    tensor_write<in_out_t>(output, shape, index, minimum<in_out_t>);
}
for_each(index in shape1) {
    out_index = index;
    out_index[axis] = 0;
    in_out_t value = tensor_read<in_out_t>(input, shape1, index);
    in_out_t state = tensor_read<in_out_t>(output, shape, out_index);
    state      = apply_max<in_out_t>(state, value);
    tensor_write<in_out_t>(output, shape, out_index, state);
}
```

Supported Data Types:

Profile	Mode	in_out_t
Any	signed 8	int8_t
Any	signed 16	int16_t
Any	signed 32	int32_t
MI, MT	fp16	fp16_t
MI, MT	bf16	bf16_t
MI, MT	fp32	fp32_t

2.9.4. REDUCE_MIN

Reduce a tensor along the given axis with a minimum operation

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input	shape1	Input tensor with rank from 1 to 4
Attribute	int32_t	axis	-	Axis to reduce, in range from 0 to rank(shape1)-1

Argument	Type	Name	Shape	Description
Output	in_out_t*	output	shape	Output tensor. Same rank as the input tensor.

Operation Function:

```
ERROR_IF(axis < 0  || axis >= rank(shape1));
ERROR_IF(shape[axis] != 1);
for_each(index in shape) {
    tensor_write<in_out_t>(output, shape, index, maximum<in_out_t>);
}
for_each(index in shape1) {
    out_index = index;
    out_index[axis] = 0;
    in_out_t value = tensor_read<in_out_t>(input, shape1, index);
    in_out_t state = tensor_read<in_out_t>(output, shape, out_index);
    state      = apply_min<in_out_t>(state, value);
    tensor_write<in_out_t>(output, shape, out_index, state);
}
```

Supported Data Types:

Profile	Mode	in_out_t
Any	signed 8	int8_t
Any	signed 16	int16_t
Any	signed 32	int32_t
MI, MT	fp16	fp16_t
MI, MT	bf16	bf16_t
MI, MT	fp32	fp32_t

2.9.5. REDUCE_PRODUCT

Reduce a tensor along the given axis by computing the product of the axis.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input	shape1	Input tensor with rank from 1 to 4
Attribute	int32_t	axis	-	Axis to reduce, in range from 0 to rank(shape1)-1

Argument	Type	Name	Shape	Description
Output	in_out_t*	output	shape	Output tensor. Same rank as the input tensor.

Operation Function:

```
ERROR_IF(axis < 0  || axis >= rank(shape1));
ERROR_IF(shape[axis] != 1);
for_each(index in shape) {
    tensor_write<in_out_t>(output, shape, index, 1.0);
}
for_each(index in shape1) {
    out_index = index;
    out_index[axis] = 0;
    in_out_t value = tensor_read<in_out_t>(input, shape1, index);
    in_out_t state = tensor_read<in_out_t>(output, shape, out_index);
    state      = state * value;
    tensor_write<in_out_t>(output, shape, out_index, state);
}
```

Supported Data Types:

Profile	Mode	in_out_t
MI, MT	fp16	fp16_t
MI, MT	bf16	bf16_t
MI, MT	fp32	fp32_t

2.9.6. REDUCE_SUM

Reduce a tensor along the given axis by computing the sum of the axis.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input	shape1	Input tensor with rank from 1 to 4
Attribute	int32_t	axis	-	Axis to reduce, in range from 0 to rank(shape1)-1
Output	in_out_t*	output	shape	Output tensor. Same rank as the input tensor.

Operation Function:

```
ERROR_IF(axis < 0  || axis >= rank(shape1));
ERROR_IF(shape[axis] != 1);
for_each(index in shape) {
    tensor_write<in_out_t>(output, shape, index, 0);
}
for_each(index in shape1) {
    out_index = index;
    out_index[axis] = 0;
    in_out_t value = tensor_read<in_out_t>(input, shape1, index);
    in_out_t state = tensor_read<in_out_t>(output, shape, out_index);
    state      = apply_add<in_out_t>(state, value);
    tensor_write<in_out_t>(output, shape, out_index, state);
}
```

Supported Data Types:

Profile	Mode	in_out_t
Any	signed 32	int32_t
MI, MT	fp16	fp16_t
MI, MT	bf16	bf16_t
MI, MT	fp32	fp32_t

2.10. Data Layout

2.10.1. CONCAT

Concatenate a list of tensors along a given axis. No data conversion happens during a concat operation.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input1	shapes1[]	List of input tensors. All inputs must have the same rank and data type
Attribute	int32_t	axis	-	Axis along which concatenation is to occur, in range from 0 to rank(shape)-1
Output	in_out_t*	output	shape	Output tensor

Operation Function:

```
ERROR_IF(axis < 0 || axis >= rank(shapes1[0]));
ERROR_IF(shape[axis] != sum(shape1[k][axis] for all k))
// The following checks ensure all inputs are compatible for concatenation
for_each(input_shape in shapes1) {
    ERROR_IF(rank(input_shape) != rank(shapes1[0]));
    for_each(index in input_shape) {
        ERROR_IF(input_shape[index] != shapes1[0][index] && index != axis);
    }
}
for_each(index1 in shape) {
    index2 = index1;
    for (tensor t = 0; t < length(input1); t++) {
        // Continue to concatenate along axis from each tensor
        // For each output location, we are looking for the
        // appropriate input tensor
        if (index2[axis] >= 0 && index2[axis] < shapes1[t][axis]) {
            in_out_t value = tensor_read<in_out_t>(input1[t], shapes1[t], index2);
            tensor_write<in_out_t>(output, shape, index1, value);
        }
        index2[axis] = index2[axis] - shapes1[t][axis];
    }
}
```

Supported Data Types:

Profile	Mode	in_out_t
Any	Boolean	bool_t
Any	signed 8	int8_t
Any	signed 16	int16_t
Any	signed 32	int32_t
MI, MT	fp16	fp16_t
MI, MT	bf16	bf16_t
MI, MT	fp32	fp32_t

2.10.2. PAD

Pads a tensor along the borders of each dimension with a supplied value. Returns a new tensor with the padding included. The pad_const value includes the zero point if the tensor uses a zero point.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input1	shape1	Input tensor
Attribute	int32_t	padding	[rank(input1),2]	Amount of padding to be done
Attribute	in_out_t	pad_const	-	Constant value to be used as padding
Output	in_out_t*	output	shape	Output tensor of same type as the input tensor

Operation Function:

```
// Padding sizes must be >= 0.
for_each(pad_size in padding) {
    ERROR_IF(pad_size < 0);
}
for_each(index in shape) {
    index1 = index;
    bool_t is_pad = false;
    for(i = 0; i < rank(shape); i++) {
        index1[i] = index1[i] - padding[i,0];
        if (index1[i] < 0 || index[i] >= length(shape[i])) {
            is_pad = true;
        }
    }
    in_out_t value = is_pad ? pad_const : tensor_read<in_out_t>(input1, shape1,
index1);
    tensor_write<in_out_t>(output, shape, index, value);
}
```

Supported Data Types:

Profile	Mode	in_out_t
Any	Boolean	bool_t
Any	signed 8	int8_t
Any	signed 16	int16_t
Any	signed 32	int32_t
MI, MT	fp16	fp16_t
MI, MT	bf16	bf16_t
MI, MT	fp32	fp32_t

2.10.3. RESHAPE

Returns a tensor with the same type/values as the input, with a new shape specified by the shape argument. Reshape may operate on tensors of any rank. No data conversion happens during a reshape operation.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input1	shape1	Input tensor
Attribute	int32_t	new_shape	[rank(shape)]	List of values, with each element giving the size of the result tensor for the given dimension.
Output	in_out_t*	output	shape	Output tensor of same type, size as the input tensor

Operation Function:

```
ERROR_IF(tensor_size(shape1) != tensor_size(shape));

for_each(index in shape) {
    // Calculate flattened index for the output location (index)
    size_t offset = tensor_index_to_offset(shape, index);
    // Now convert to the location in the input
    dim_t tmp_index = tensor_offset_to_index(shape1, offset);

    // Now read/write the value
    in_out_t val = tensor_read<in_out_t>(input, shape1, tmp_index);
    tensor_write<in_out_t>(output, shape, index, val);
}
```

Supported Data Types:

Profile	Mode	in_out_t
Any	Boolean	bool_t
Any	signed 8	int8_t
Any	signed 16	int16_t
Any	signed 32	int32_t
MI, MT	fp16	fp16_t
MI, MT	bf16	bf16_t

Profile	Mode	in_out_t
MI, MT	fp32	fp32_t

2.10.4. REVERSE

Returns a tensor with the same type/values as the input, with the data reversed along the given axis. No data conversion happens during a reverse operation.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input	shape	Input tensor from 1 to 4 dims
Attribute	int32_t	axis	-	Axis to reverse, in range from 0 to rank(shape)-1
Output	in_out_t*	output	shape	Output tensor. Same shape as input tensor.

Operation Function:

```
ERROR_IF(axis < 0 || axis >= rank(shape));
for_each(index in shape) {
    tmp_index = index;
    tmp_index[axis] = shape[axis] - 1 - index[axis];
    in_out_t value = tensor_read<in_out_t>(input, shape, tmp_index);
    tensor_write<in_out_t>(output, shape, index, value);
}
```

Supported Data Types:

Profile	Mode	in_out_t
Any	Boolean	bool_t
Any	signed 8	int8_t
Any	signed 16	int16_t
Any	signed 32	int32_t
MI, MT	fp16	fp16_t
MI, MT	bf16	bf16_t
MI, MT	fp32	fp32_t

2.10.5. SLICE

Extracts a slice of the input1 on the given axis, beginning at the start coordinates, and extending for size elements in each direction. No data conversion happens during a slice operation.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input1	shape1	Input tensor with rank from 1 to 4
Attribute	int32_t	start	[rank(input1)]	List of integer coordinates, of length equal to the rank of input1. Start coordinate for slicing.
Attribute	int32_t	size	[rank(input1)]	List of integer size values, of length equal to the rank of input1. Size of the input to be used.
Output	in_out_t*	output	shape	Output tensor of same type as the input tensor

Operation Function:

```
ERROR_IF(rank(input1) != length(start) || rank(input1) != length(size));
ERROR_IF(rank(input1) != rank(output))
// Sanity check the given coordinates, ensure start and end are
// within tensor bounds
for_each(index in rank(input1)) {
    ERROR_IF(start[index] < 0);
    ERROR_IF(size[index] <= 0); //Output must be positive size
    ERROR_IF(start[index] + size[index] > shape1[index]);
    ERROR_IF(shape[index] != size[index]);
}

for_each(index in shape) {
    tmp_index = index;
    for(i = 0; i < rank(shape); i++) {
       tmp_index[i] = index[i] + start[i];
    }
    in_out_t value = tensor_read<in_out_t>(input, shape1, tmp_index);
    tensor_write<in_out_t>(output, shape, index, value);
}
```

Supported Data Types:

Profile	Mode	in_out_t
Any	Boolean	bool_t
Any	signed 8	int8_t
Any	signed 16	int16_t
Any	signed 32	int32_t
MI, MT	fp16	fp16_t
MI, MT	bf16	bf16_t
MI, MT	fp32	fp32_t

2.10.6. TILE

Replicates input1 multiples times along each dimension.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input1	shape1	Input tensor with rank from 1 to 4
Attribute	int32_t	multiples	[rank(shape1)]	Number of times to replicate input1 in each dimension
Output	in_out_t*	output	shape	Output tensor of same type, rank as the input tensor

Operation Function:

```
for_each(index in shape) {
    tmp_index = index;
    for(i = 0; i < rank(shape); i++) {
        ERROR_IF(shape1[i] * multiples[i] != shape[i]);
        tmp_index[i] = index[i] % shape1[i];
    }
    in_out_t value = tensor_read<in_out_t>(input, shape1, tmp_index);
    tensor_write<in_out_t>(output, shape, index, value);
}
```

Supported Data Types:

Profile	Mode	in_out_t
Any	Boolean	bool_t

Profile	Mode	in_out_t
Any	signed 8	int8_t
Any	signed 16	int16_t
Any	signed 32	int32_t
MI, MT	fp16	fp16_t
MI, MT	bf16	bf16_t
MI, MT	fp32	fp32_t

2.10.7. TRANSPOSE

Permutes the dimensions of the input tensor input1 based on the perms argument. Each value in the perms list must be a valid dimension of the input tensor and may not be repeated.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input1	shape1	Input tensor with minimum rank of one.
Attribute	int32_t	perms	[rank(input1)]	List of integers of length equal to the rank of input1. Values must be valid dimensions within shape1, and may not be repeated.
Output	in_out_t*	output	shape	Output tensor of same type, rank as the input tensor

Operation Function:

```
for_each(index in perms) {
    // Ensure each perms value is a valid value
    ERROR_IF(index >= rank(shape1));
    ERROR_IF(index < 0);
    // Ensure ranks aren't repeated
    ERROR_IF(indexes_used[index] == true);
    indexes_used[index] = true;
}

// Ensure that the output shapes have the properly
// permuted shapes
for(i = 0; i < rank(shape); i++) {
    ERROR_IF(shape1[perms[i]] != shape[i])
}

for_each(index in shape) {
    tmp_index = index;
    for(i = 0; i < rank(shape); i++) {
        tmp_index[perms[i]] = index[i]
    }
    in_out_t value = tensor_read<in_out_t>(input, shape1, tmp_index);
    tensor_write<in_out_t>(output, shape, index, value);
}
```

Supported Data Types:

Profile	Mode	in_out_t
Any	Boolean	bool_t
Any	signed 8	int8_t
Any	signed 16	int16_t
Any	signed 32	int32_t
MI, MT	fp16	fp16_t
MI, MT	bf16	bf16_t
MI, MT	fp32	fp32_t

2.11. Scatter/Gather Operators

2.11.1. GATHER

Generate a tensor for which each element in the output is a subtensor of the values tensor based on the indices. N is the number of batches, W the number of indices in each batch, K the range of each index and C the number data channels for each index.

Arguments:

Argument	Type	Name	Shape	Description
Input	value_t*	values	[N,K,C]	3D value tensor
Input	index_t*	indices	[N,W]	2D index tensor
Output	value_t*	output	[N,W,C]	3D output tensor

Operation Function:

```
for_each(0 <= n < N, 0 <= w < W, 0 <= c < C) {
    index_t k = tensor_read<index_t>(indices, [N,W], [n,w]);
    REQUIRE(0 <= k && k < K);
    value_t value = tensor_read<value_t>(values, [N,K,C], [n,k,c]);
    tensor_write<value_t>(output, [N,W,C], [n,w,c], value);
}
```

Supported Data Types:

Profile	Mode	index_t	value_t
Any	signed 8	int32_t	int8_t
Any	signed 16	int32_t	int16_t
Any	signed 32	int32_t	int32_t
MI,MT	float	int32_t	float

2.11.2. SCATTER

The values_out tensor is set to the values_in tensor with data modified as follows: data from the input tensor is inserted at the positions specified by the indices tensor. N is the number of batches, W the number of indices in each batch, K the range of each index and C the number data channels for each index. It is not permitted to repeat the same output index within a single SCATTER operation and so each output index occurs at most once. In use cases that require multiple updates to the same output position, these must be decomposed into multiple SCATTER operations.

Arguments:

Argument	Type	Name	Shape	Description
Input	value_t*	values_in	[N,K,C]	3D values in tensor
Input	index_t*	indices	[N,W]	2D index tensor
Input	value_t*	input	[N,W,C]	3D input tensor
Output	value_t*	values_out	[N,K,C]	3D values out tensor

Quantization Parameters:

None

Operation Function:

```
// The following array is used to check compliance that an output position
// is modified at most once.
bool_t output_modified[N,K,C];

// Copy the values_in tensor to the values_out tensor.
// Values not written by the scatter operation are unchanged in the output.
for_each(0 <= n < N, 0 <= k < K, 0 <= c < C) {
    value_t value = tensor_read<value_t>(values_in, [N,K,C], [n,k,c]);
    tensor_write<value_t>(values_out, [N,K,C], [n, k, c], value);
    output_modified[n,k,c]=false;
}

// Now perform the SCATTER operation, modifying the positions from the indices tensor
for_each(0 <= n < N, 0 <= w < W, 0 <= c < C) {
    index_t k = tensor_read<index_t>(indices, [N,W], [n,w]);
    REQUIRE(0 <= k && k < K);
    REQUIRE(output_modified[n,k,c] == false);
    value_t value = tensor_read<value_t>(input, [N,W,C], [n,w,c]);
    tensor_write<value_t>(values_out, [N,K,C], [n, k, c], value);
    output_modified[n,k,c] = true;
}
```

Supported Data Types:

Profile	Mode	index_t	value_t
Any	signed 8	int32_t	int8_t
Any	signed 16	int32_t	int16_t
Any	signed 32	int32_t	int32_t
MI,MT	fp16	int32_t	fp16_t
MI,MT	bf16	int32_t	bf16_t
MI,MT	fp32	int32_t	fp32_t

2.12. Image Operators

2.12.1. RESIZE

Resizes a tensor. Resize is only allowed in the H and W dimensions.

The NEAREST_NEIGHBOR mode returns the value of the input tensor closest to the calculated sample position for both floating-point and integer data formats.

Floating-point BILINEAR mode returns a bilinearly interpolated output value based on the four

closest input sample positions.

For integer BILINEAR interpolation mode, the output value is calculated by using the shift value along with the other parameters to create a fixed point scaling factor for each input. These values are then summed to create the value for output, which has 2 * shift fractional bits. To convert back to the original integer size, the output value must be rescaled.

The following examples show practical uses of the parameters:

- For approximate uniform input sampling between (0, 0) and (IH-1, IW-1) set stride_y = ((IH-1) * (1<<shift)) / (OH-1), stride_x = ((IW-1) * (1<<shift)) / (OW-1), offset_x=0, offset_y=0, border_x=0, border_y=0.
- For power of two upscale by factor (1<<k) the following parameters can be used for fixed point upscales:
  - For upscale [OH-1,OW-1] = (1<<k) * [IH-1, IW-1] set shift=k, stride_y=1, stride_x=1, offset_x=0, offset_y=0, border_x=0, border_y=0.
  - For upscale [OH,OW] = (1<<k) * [IH,IW] set shift=(k+1), stride_y=2, stride_x=2, offset_x=-(1<<k)+1, offset_y=-(1<<k)+1, border_x=1<<(k-1), border_y=1<<(k-1). This samples approximately the input area (-0.5, -0.5) to (IH-0.5, IW-0.5).

Arguments:

Argument	Type	Name	Shape	Description
Input	in_t*	input	[N,IH,IW,C]	Input tensor
Attribute	int32_t*	output_size	[2]	[OH,OW]
Attribute	resize_t*	stride	[2]	[stride_y, stride_x]
Attribute	resize_t*	offset	[2]	[offset_y, offset_x]
Attribute	int32_t*	border	[2]	[border_y, border_x]
Attribute	int32_t	shift	-	Shift value (must be zero if resize_t is float)
Attribute	mode_t	mode	-	BILINEAR or NEAREST
Output	out_t*	output	[N,OH,OW,C]	Output tensor

Operation Function

```
// Derive the output dimensions from the input dimensions
OH = idiv((IH-1)*(1<<shift) - offset_y, stride_y) + 1 + border_y;
OW = idiv((IW-1)*(1<<shift) - offset_x, stride_x) + 1 + border_x;
// Ensure the image size is supported by GPU APIs and that for integer
// implementations, position * stride does not overflow int32_t.
ERROR_IF(max(OH,OW,IH,IW) >= 16384);
ERROR_IF(stride_x <= 0 || stride_y <= 0);
```

```
if (is_floating_point(resize_t)) {
    // The shift attribute is not used for floating point
    ERROR_IF(shift != 0);
    ERROR_IF(stride_x > IW || stride_y > IH);
} else {
    // if in_t=int8_t ensure that an int32_t accumulator can be used
    ERROR_IF(shift < 1 || shift > 11);
    // set a consistent lower limit of 1/16 downscale
    // independent of the shift value to simplify implementations
    ERROR_IF(stride_x >= (16 << shift));
    ERROR_IF(stride_y >= (16 << shift));
    // offset range is similarly limited to maximum 16 pixels irrespective
    // of shift. Both stride and offset fit in int16_t when shift=11.
    ERROR_IF(offset_x <= (-16 << shift) || offset_x >= (16 << shift));
    ERROR_IF(offset_y <= (-16 << shift) || offset_y >= (16 << shift));
}
for_each(0 <= n < N, 0 <= oy < OH, 0 <= ox < OW; 0 <= c < C) {
    unit = (is_floating_point(resize_t)) ? 1.0 : (1 << shift);
    y = oy * stride_y + offset_y;
    x = ox * stride_x + offset_x;
    if (is_floating_point(resize_t)) {
        iy = (int32_t)apply_floor(y); dy = y - (resize_t)iy;
        ix = (int32_t)apply_floor(x); dx = x - (resize_t)ix;
    } else {
        iy = y >> shift; dy = y - (iy<<shift);
        ix = x >> shift; dx = x - (ix<<shift);
    }
    iy0 = apply_max(iy, 0);
    iy1 = apply_min(iy+1, IH-1);
    ix0 = apply_max(ix, 0);
    ix1 = apply_min(ix+1, IW-1);
    REQUIRE(ix0 <= ix1 && iy0 <= iy1);
    if (mode==BILINEAR) {
        v00 = tensor_read<in_t>(input, [N,IH,IW,C], [n,iy0,ix0,c]);
        v01 = tensor_read<in_t>(input, [N,IH,IW,C], [n,iy0,ix1,c]);
        v10 = tensor_read<in_t>(input, [N,IH,IW,C], [n,iy1,ix0,c]);
        v11 = tensor_read<in_t>(input, [N,IH,IW,C], [n,iy1,ix1,c]);
        out_t acc = v00 * (unit - dy) * (unit - dx) + v01 * (unit - dy) * dx;
        acc = acc + v10 * dy * (unit-dx) + v11 * dy * dx;
        tensor_write<out_t>(output, [N,OH,OW,C], [n,oy,ox,c], acc);
    } else if (mode==NEAREST) {
        iy = (dy >= unit/2) ? iy1 : iy0;
        ix = (dx >= unit/2) ? ix1 : ix0;
        v = tensor_read<in_t>(input, [N,IH,IW,C], [n,iy,ix,c]);
        tensor_write<out_t>(output, [N,OH,OW,C], [n,oy,ox,c], v);
    }
}
```

Supported Data Types:

Profile	Mode	resize_t	in_t	out_t
Any	signed 8, bilinear	int16_t	int8_t	int32_t
Any	signed 8, nearest	int16_t	int8_t	int8_t
Any	signed 16, bilinear	int16_t	int16_t	int48_t
Any	signed 16, nearest	int16_t	int16_t	int16_t
MI,MT	fp16	fp32_t	fp16_t	fp16_t
MI,MT	bf16	fp32_t	bf16_t	bf16_t
MI,MT	fp32	fp32_t	fp32_t	fp32_t

Resize Modes:

Mode	Description
NEAREST	Nearest Neighbor
BILINEAR	Bilinear interpoloation

2.13. Type Conversion

2.13.1. CAST

Casts a tensor from one data type to another.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_t*	input	shape	Input tensor
Output	out_t*	output	shape	Output tensor

Operation Function:

```
for_each(index in shape) {
    in_t in = tensor_read<in_t>(input, shape, index);
    out_t out;
    if (out_t == bool_t) {
        out = (in != 0) ? true : false;
    } else if (in_t == bool_t) {
        out = (in) ? 1 : 0;
    } else if (out_t == fp16_t || out_t == bf16_t || out_t == fp32_t) {
        out = round_to_nearest_float(in);
    } else if (in_t == fp16_t || in_t == bf16_t || in_t == fp32_t) {
        out = apply_clip<out_t>(round_to_nearest_int(in), minimum<out_t>,
maximum<out_t>);
    } else if (sizeof(out_t) >= sizeof(in_t)) {
        out = sign_extend(in);
    } else {
        out = truncate(in);
    }
    tensor_write<out_t>(output, shape, index, out)
}
```

Supported Data Types:

Profile	Mode	in_t	out_t
Any	bool to signed 8	bool_t	int8_t
Any	bool to signed 16	bool_t	int16_t
Any	bool to signed 32	bool_t	int32_t
Any	signed 8 to bool	int8_t	bool_t
Any	signed 8 to signed 16	int8_t	int16_t
Any	signed 8 to signed 32	int8_t	int32_t
MI, MT	signed 8 to fp16	int8_t	fp16_t
MI, MT	signed 8 to bf16	int8_t	bf16_t
MI, MT	signed 8 to fp32	int8_t	fp32_t
Any	signed 16 to bool	int16_t	bool_t
Any	signed 16 to signed 8	int16_t	int8_t
Any	signed 16 to signed 32	int16_t	int32_t
MI, MT	signed 16 to fp16	int16_t	fp16_t
MI, MT	signed 16 to bf16	int16_t	bf16_t
MI, MT	signed 16 to fp32	int16_t	fp32_t
Any	signed 32 to bool	int32_t	bool_t
Any	signed 32 to signed 8	int32_t	int8_t
Any	signed 32 to signed 16	int32_t	int16_t

Profile	Mode	in_t	out_t
MI, MT	signed 32 to fp16	int32_t	fp16_t
MI, MT	signed 32 to bf16	int32_t	bf16_t
MI, MT	signed 32 to fp32	int32_t	fp32_t
MI, MT	fp16 to signed 8	fp16_t	int8_t
MI, MT	fp16 to signed 16	fp16_t	int16_t
MI, MT	fp16 to signed 32	fp16_t	int32_t
MI, MT	bf16 to signed 8	bf16_t	int8_t
MI, MT	bf16 to signed 16	bf16_t	int16_t
MI, MT	bf16 to signed 32	bf16_t	int32_t
MI, MT	fp32 to signed 8	fp32_t	int8_t
MI, MT	fp32 to signed 16	fp32_t	int16_t
MI, MT	fp32 to signed 32	fp32_t	int32_t

2.13.2. RESCALE

Rescale quantized values into a new domain. This function scales by factor: multiplier * 2^{-shift}.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_t*	input	shape	Input tensor from 1 to 4 dims
Output	out_t*	output	shape	Output tensor with the same shape as input
Attribute	in_t	input_zp	-	Input tensor zero point. Must be zero for non-int8 types.
Attribute	out_t	output_zp	-	Output tensor zero point. Must be zero for non-int8 types.
Input (MT profile) Attribute (BI/MI profiles)	mul_t*	multiplier	[NC]	Scaling multiplier array
Input (MT profile) Attribute (BI/MI profiles)	uint6_t*	shift	[NC]	Scaling shift array

Argument	Type	Name	Shape	Description
Attribute	bool_t	scale32	-	if (scale32) mul_t=int32_t else mul_t=int16_t
Attribute	bool_t	double_round	-	Select double round mode
Attribute	bool_t	per_channel	-	if (per_channel) NC=shape[dims-1] else NC=1

Operation Function:

```
for_each(index in shape) {
    // uint16 values can have zero_point 0 or 32768
    // int8/uint8 can have zero point within their valid range
    // No other types can have zero point != 0
    ERROR_IF(in_t != int8_t &&
             in_t != uint8_t &&
             in_t != uint16_t && input_zp != 0);
    ERROR_IF(out_t != int8_t &&
             out_t != uint8_t &&
             out_t != uint16_t && output_zp != 0);
    ERROR_IF(in_t == uint16_t && (input_zp != 0 || input_zp != 32768));
    ERROR_IF(out_t == uint16_t && (output_zp != 0 || output_zp != 32768));
    ERROR_IF(scale32 && in_t == int48_t);
    ERROR_IF(!scale32 && double_round);
    int48_t value = tensor_read<in_t>(input, shape, index);
    value = value - input_zp;
    int c = (per_channel) ? index[dims-1] : 0;
    int32_t result = (scale32) ?
        apply_scale_32(value, multiplier[c], shift[c], double_round) :
        apply_scale_16(value, multiplier[c], shift[c]);
    result = (out_t)apply_clip<int32_t>(result + output_zp, minimum<out_t>,
maximum<out_t>);
    tensor_write<out_t>(output, shape, index, result);
}
```

Supported Data Types:

Profile	Mode	in_t	out_t
Any	signed 8 to signed 8	int8_t	int8_t
Any	signed 8 to signed 16	int8_t	int16_t
Any	signed 8 to signed 32	int8_t	int32_t
Any	signed 8 to unsigned 8	int8_t	uint8_t
Any	signed 16 to signed 8	int16_t	int8_t

Profile	Mode	in_t	out_t
Any	signed 16 to signed 16	int16_t	int16_t
Any	signed 16 to signed 32	int16_t	int32_t
Any	signed 16 to unsigned 8	int16_t	uint8_t
Any	signed 16 to unsigned 16	int16_t	uint16_t
Any	signed 32 to signed 8	int32_t	int8_t
Any	signed 32 to signed 16	int32_t	int16_t
Any	signed 32 to signed 32	int32_t	int32_t
Any	signed 48 to signed 8	int48_t	int8_t
Any	signed 48 to signed 16	int48_t	int16_t
Any	signed 48 to signed 32	int48_t	int32_t
Any	unsigned 8 to signed 8	uint8_t	int8_t
Any	unsigned 8 to signed 16	uint8_t	int16_t
Any	unsigned 16 to signed 16	uint16_t	int16_t

2.14. Data Nodes

2.14.1. CONST

A node containing constant data for use as the input to an operation. May hold data in any of the supported data formats.

Arguments:

Argument	Type	Name	Shape	Description
Attribute	out_t*	values	shape	Constant values
Output	out_t*	output	shape	Output tensor of same type, size as the input tensor

Supported Data Types:

Profile	Mode	out_t
Any	Boolean	bool_t
Any	signed 8	int8_t
Any	signed 16	int16_t
Any	signed 32	int32_t

Profile	Mode	out_t
MI, MT	fp16	fp16_t
MI, MT	bf16	bf16_t
MI, MT	fp32	fp32_t

2.14.2. IDENTITY

Returns a tensor with the same shape, type, and contents as the input.

Arguments:

Argument	Type	Name	Shape	Description
Input	in_out_t*	input1	shape	Input tensor
Output	in_out_t*	output	shape	Output tensor of same type, size as the input tensor

Supported Data Types:

Profile	Mode	in_out_t
Any	Boolean	bool_t
Any	signed 8	int8_t
Any	signed 16	int16_t
Any	signed 32	int32_t
MI, MT	fp16	fp16_t
MI, MT	bf16	bf16_t
MI, MT	fp32	fp32_t

2.15. Custom Operators

Hardware implementing TOSA may choose to add additional custom operators that are not expressed in the existing TOSA operations. These operators are not expected to be portable across TOSA implementations. The input and output signatures must be expressed in the corresponding TOSA node.

2.15.1. CUSTOM

Input Operands:

- Num input operands – Scalar number of input operands
- Num output operands – Scalar number of output operands
- Operator code – untyped data consisting of the operator data

- Affine transform description for each tensor

2.16. Control Flow Operators

TOSA implements two control flow operators, for conditional branching and loop based control. Both have attributes that are TOSA sub-graphs.

2.16.1. COND_IF

Evaluates a Boolean condition and then takes one of two distinct execution paths. This implements the semantic if-then-else structure.

Arguments:

Argument	Type	Name	Description
Input	tensor_list_t	input_list	List of input tensors
Input	bool_t	condition	Input condition as rank-0 tensor
Attribute	tosa_graph_t	then_graph	TOSA graph to execute if condition is true
Attribute	tosa_graph_t	else_graph	TOSA graph to execute if condition is false
Output	tensor_list_t	output_list	List of output tensors

Operation Function:

```
ERROR_IF(tensor_list_shape(input_list) != tosa_input_shape(then_graph));
ERROR_IF(tensor_list_shape(input_list) != tosa_input_shape(else_graph));
ERROR_IF(tensor_list_shape(output_list) != tosa_output_shape(then_graph));
ERROR_IF(tensor_list_shape(output_list) != tosa_output_shape(else_graph));

if (condition) {
    tosa_execute_graph(then_graph, input_list, output_list);
} else {
    tosa_execute_graph(else_graph, input_list, output_list);
}
```

2.16.2. WHILE_LOOP

Generates and evaluates a Bool condition and either executes a loop body or exits the loop. This action is performed repeatedly after updating and re-evaluating the Boolean condition every iteration. This implements the semantic foreach or while iterative loop structure.

Arguments:

Argument	Type	Name	Description
Input	tensor_list_t	input_list	List of input tensors
Attribute	tosa_graph_t	cond_graph	TOSA graph to evaluate the condition
Attribute	tosa_graph_t	body_graph	TOSA graph to execute the loop body
Output	tensor_list_t	output_list	List of output tensors

Operation Function:

```
ERROR_IF(tensor_list_shape(input_list) != tosa_list_shape(output_list));
ERROR_IF(tensor_list_shape(input_list) != tosa_input_shape(cond_graph));
ERROR_IF(tensor_list_shape(input_list) != tosa_input_shape(body_graph));
ERROR_IF(tensor_list_shape(input_list) != tosa_output_shape(body_graph));
ERROR_IF(tosa_output_shape(cond_graph) != tosa_list_shape([bool_t]));

// The iteration number 'i' is included to give unique names to variables
// in each iteration of the loop and is not required by implementations
int32_t i=0;             // iteration number
list[i] = input_list;    // copy input data as list[0]
tosa_execute_graph(cond_graph, list[i], [condition[i]]);   // initial condition
while (condition[i]) {
    tosa_execute_graph(body_graph, list[i], list[i+1]);
    i = i+1;
    tosa_execute_graph(cond_graph, list[i], [condition[i]]);
}
output_list = list[i];
```

3. TOSA Pseudocode

The TOSA pseudocode provides precise descriptions of TOSA operations. Each operator contains pseudocode describing the operator's functionality. This section contains pseudocode functions shared across multiple operators in the specification.

3.1. Operator Validation Helpers

The following functions are used to define the valid conditions for TOSA operators.

The REQUIRE function defines the conditions required by the TOSA operator. If the conditions are not met then the result of the TOSA graph is marked as unpredictable. Once the tosa_graph_result is set to tosa_unpredictable, the whole graph is considered unpredictable.

The ERROR_IF function defines a condition that must set an error if the condition holds and the graph is not unpredictable. Note that if a graph contains both unpredictable and error statements then result of tosa_execute_graph() is tosa_unpredictable. This condition is captured in the

ERROR_IF function.

Implementation Notes

- An implementation is not required to detect unpredictable behavior. If tosa_execute_graph() returns tosa_unpredictable then the tosa_test_compliance() function does not require any specific output from an implementation.
- An implementation is required to detect errors in a graph that does not have unpredictable behavior (see tosa_test_compliance).
- An acceptable implementation is to stop and report an error on the first ERROR_IF condition that occurs. This satifies tosa_test_compliance() even if the tosa_execute_graph() was tosa_unpredictable.
- If the tosa_execute_graphs() result is tosa_unpredictable or tosa_error, then there is no requirement on the implementation to execute any portion of the TOSA graph.

```
void REQUIRE(condition) {
    // Unpredictable overrides any previous result
    if (!(condition)) {
        tosa_graph_result = tosa_unpredictable;
    }
}

void ERROR_IF(condition) {
    // Error encodes a predictable error state and so is not registered
    // if the graph is marked as unpredictable.
    if (tosa_graph_result != tosa_unpredictable && condition) {
        tosa_graph_result = tosa_error;
    }
}
```

3.2. Tensor Access Helpers

3.2.1. Tensor Utilities

```
size_t tensor_index_to_offset(dim_t shape, dim_t index) {
    // Ensure this is a proper tensor with each dimension having size >= 1
    for_each(dimension_size in shape) {
        REQUIRE(dimension_size >= 1);
    }
    size_t offset = 0;
    for (int32_t i = 0; i < rank(shape); i++) {
        REQUIRE(index[i] >= 0 && index[i] < shape[i]);
        offset = offset * shape[i] + index[i];
    }
    return offset;
}

dim_t tensor_offset_to_index(dim_t shape, size_t offset) {
    // Index is a dim_t with rank equal to the rank of shape
    dim_t index(rank(shape));
    for(int32_t r = rank(shape1) - 1; r >= 0; r--) {
        index[r] = offset % shape1[r];
        calculated_index /= shape[r];
    }
    return index;
}

// Input is the shape of the given tensor
size_t tensor_size(dim_t shape) {
    size_t size = 1;
    for (int32_t i=0; i < rank(shape); i++) {
        size *= input[i];
    }
    return size;
}
```

3.2.2. Tensor Read

tensor_read reads a single data value out of the given tensor. The shape argument contains the shape of the tensor. Index is the coordinates within the tensor of the value to be read.

```
in_t tensor_read<in_t>(in_t *address, dim_t shape, dim_t index) {
    size_t offset = tensor_index_to_offset(shape, index);
    return address[offset];
}
```

3.2.3. Tensor Write

tensor_write writes a single data value into the given tensor. The shape argument contains the shape of the tensor. Index is the coordinates within the tensor of the value to be written. value is the value to be written to the given coordinate.

```
void tensor_write<type>(<type> *address, dim_t shape, dim_t index, <type> value) {
    size_t offset = tensor_index_to_offset(shape, index);
    address[offset] = value;
}
```

3.2.4. Broadcast Helper

The following function maps an index in the output tensor to an index in the input tensor.

```
// The index argument should be a valid location within out_shape.
// The function returns the location within in_shape that contributes
// to the output based on broadcasting rules.

dim_t apply_broadcast(dim_t out_shape, dim_t in_shape, dim_t index) {
    ERROR_IF(rank(out_shape) != rank(in_shape));
    ERROR_IF(rank(out_shape) != rank(index));
    for (int32_t i = 0; i < rank(out_shape); i++) {
        if (out_shape[i] != in_shape[i]) {
            ERROR_IF(in_shape[i] != 1);
            index[i] = 0;
        }
    }
    return index;
}
```

3.3. General Pseudocode Helpers

This section contains general pseudocode utility functions used throughout the specification.

3.3.1. Arithmetic Helpers

The following functions provide arithmetic while defining requirements such that values stay in the valid range.

```
in_t apply_add<in_t>(in_t a, in_t b) {
    if (is_floating_point(in_t)) return a + b;
    int64_t c = (int64_t)a + (int64_t)b;
    REQUIRE(c >= minimum<in_t> && c <= maximum<in_t>);
    return (in_t)c;
}

in_t apply_ceil<in_t>(in_t input) {
    return input value rounded up to nearest integer
}

in_t apply_clip<in_t>(in_t value, in_t min_val, in_t max_val) {
    REQUIRE(min_val <= max_val);
```

```
    value = apply_max(value, min_val);
    value = apply_min(value, max_val);
    return value;
}

in_t apply_exp<in_t>(in_t input) {
    return e to the power input
}

in_t apply_floor<in_t>(in_t input) {
    return input value rounded down to nearest integer
}

in_t apply_log<in_t>(in_t input) {
    if (input == 0) {
        return -INFINITY
    }
    else if (input < 0) {
        return NaN;
    }
    return the natural logarithm of input
}

in_t apply_max<in_t>(in_t a, in_t b) {
    if (is_floating_point(in_t)) {
        if (isNaN(a) || isNaN(b)) {
            return NaN;
        }
    }
    if (a >= b) return a; else return b;
}

in_t apply_min<in_t>(in_t a, in_t b) {
    if (is_floating_point(in_t)) {
        if (isNaN(a) || isNaN(b)) {
            return NaN;
        }
    }
    if (a < b) return a; else return b;
}

in_t apply_pow<in_t>(in_t a, in_t b) {
    return a ** b; // a raised to the power b
}

in_t apply_sqrt<in_t>(in_t input) {
    return the square root of input
}

in_t apply_sub<in_t>(in_t a, in_t b) {
    if (is_floating_point(in_t)) return a - b;
```

```
    int64_t c = (int64_t)a - (int64_t)b;
    REQUIRE(c >= minimum<in_t> && c <= maximum<in_t>);
    return (in_t)c;
}

int32_t count_leading_zeros(int32_t a) {
    int32_t acc = 32;
    if (a != 0) {
        uint32_t mask;
        mask = 1 << (32 - 1); // width of int32_t - 1
        acc = 0;
        while ((mask & a) == 0) {
            mask = mask >> 1;
            acc = acc + 1;
        }
    }
    return acc;
}
```

3.3.2. Numeric Conversion Helpers

The following definitions are used in pseudocode to do numeric conversions. Where the **float_t** type is used, it represents all of the floating-point data types supported by the given profile. See [Number formats] for details on the floating-point formats.

```
int round_to_nearest_int(float_t f)
  Converts the floating-point value to f, with rounding to the nearest integer value.

float_t round_to_nearest_float(in_t f)
  Converts the input value into floating-point, rounding to the nearest representable
value.
  The behavior for ties is implementation dependent.

out_t sign_extend(in_t input)
  Only valid for two's complement integer values where out_t has more bits than in_t.
  Output = input
  Replicate the top bit of input for all bits between the top bit of input and the top
bit of output.

out_t truncate(in_t input)
  output is the sizeof(out_t) least significant bits in input.
```

The following definition is used to flatten a list of lists into a single list.

```
in_t* flatten(in_t lists[]) {
    in_t output = [];
    for_each(list in lists) {
        for_each(element in list) {
            output.append(element);
        }
    }
}
```

Generic helper functions used to keep the pseudocode concise.

```
bool_t is_floating_point(type) {
    if (type == fp16_t || type == fp32_t || type == bf16_t)
        return true;
    return false;
}

int32_t idiv(int32_t input1, int32_t input2) {
    return input1 / input2; // Integer divide that truncates towards zero
}

// Integer division that checks input1 is a multiple of input2

int32_t idiv_check(int32_t input1, int32_t input2) {
    ERROR_IF(input1 % input2 != 0); // input1 must be a multiple of input2
    return input1 / input2;         // exact quotient without rounding
}

int32_t length(in_t input)
    return number of elements in input list

int32_t rank(in_t input)
    return rank of an input tensor

int32_t sum(in_t input[])
    return the sum of values of an input list

bool isNaN(float input)
    return True if floating-point input value is NaN

float_t pi()
    returns value of pi

float_t sin(angle)
    return sine of angle given in radians

float_t cos(angle)
    return cosine of angle given in radians

bool power_of_two(int32_t value)
    return true if value is a power of two, false otherwise
```