TOSA 1.0.1 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 and nothing in this License shall restrict you from further disseminating this Specification. You shall provide a copy of this License upon disseminating the Arm Specification to a third party.

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 Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this Specification may be the trademarks of their respective owners. No licence, express, implied or otherwise, is granted to Licensee under this Licence, to use the Arm trade marks in connection with the Specification or any products based thereon. Visit Arm's website at https://www.arm.com/company/policies/trademarks for more information about Arm's trademarks.

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.

Changes from TOSA 1.0.0

Revision	Fixes
1.0.1	Fixes for POW conformance and clarifies NaN propagation when doing the dot product conformance check.

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 their numerical behavior in the case of precision, saturation, scaling, and range as required by quantized datatypes.
- Independent of any single high-level framework, compiler backend stack or particular implementation.
- 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 a combination of XML, AsciiDoc mark-up, and pseudocode files. The content is managed through a git repository here: https://git.mlplatform.org/tosa/specification.git/. The specification is developed and versioned much like software. The pseudocode (.tosac files) is written in a style similar to C++, however it is not guaranteed to be valid or compile as it exists. While the AsciiDoc content 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. The build process will also create the tables in the specification from the XML. 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 usable as a component out of which more than one type of complex operation 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 complexity.
P3	Numerical definition of common sub-operations should be consistent between operators (for example: value scaling).	Consistent sub-operation definition reduces the operator implementation complexity.
P4	The valid input and output ranges for all arguments 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. Versioning

TOSA follows a semantic versioning policy with a major.minor.patch.draft scheme. See below for the TOSA definition of backward compatibility.

- Major version changes may break backwards compatibility.
- Minor numbers may add functionality in a backwards compatible way.
- Patch versions are for bug fixes, clarifications, or trivial changes.
- The draft flag notes whether the version referenced is finalized.

Major, minor, and patch numbers are limited to eight bits. Draft is a single bit flag. If stored in a 32-bit value, the remaining bits are reserved for future use.

1.5.1. Backwards Compatibility

TOSA graphs created with previous minor versions within a major version must continue to work.

The following portions of the specification and implementation will not change within a major version:

- Operator Names
- Arguments including ordering, input/attribute/output, name, rank
- ERROR_IF statements
- Functionality of the pseudocode for each operator
- Level definitions and checks
- Removal of rows in the Supported Data Type tables
- Enumerated types and values

Changes to the following do not break compatibility:

- Machine readable specification format (currently XML)
- Machine readable specification schema
- Order of operation definitions within the XML specification
- Operator section names
- Descriptive text that does not affect functionality
- Non-functional changes to pseudocode (for example: cleanup, variable name changes)
- Additions of new rows to the Supported Data Type tables

Minor versions are allowed to add new operators or other functionality as long as the above guarantees hold.

In this version of TOSA, the precision requirements are subject to change while additional implementations are tested. When multiple implementations are shown to meet the precision requirements, those requirements will be guaranteed not to change within a major version.

In addition, new extensions may be added to the specification between TOSA releases. They may not change anything that would break backward compatibility according to the above definitions.

1.6. Profiles

TOSA profiles enable efficient implementation on different classes of device. Each profile is an independent set of operations and data type combinations.

TOSA profile extensions define optional operation and data type combinations.

Each operator's Supported Data Types table defines which profile or extension includes that operator with different data types. An operator / data type combination may be part of multiple profiles or extensions. If so, each profile and extension will be listed in the Supported Data Types

table. In addition, a table listing all operations for each profile can be found in Appendix B.

The following are required for compliant TOSA implementations:

- A TOSA implementation must implement at least one profile.
- A TOSA implementation may choose to implement any extensions.
- If a TOSA implementation chooses to implement an extension, it must implement the complete extension.
- If an operator / data type combination requires multiple extensions, the combination is only required to be implemented if all extensions are implemented
  - For example, a CAST from bf16 to fp8 is only required if both extensions are implemented.

Table 2. Profiles

Profile	Name	Description	Specification Status
Integer	PRO-INT	Integer operations, primarily 8- and 32-bit values	Complete
Floating-Point	PRO-FP	FP16 and FP32 operations	Complete

Table 3. Profile Extensions

Name	Description	Allowed profiles	Specification Status
EXT-INT16	16-bit integer operations	PRO-INT	Complete
EXT-INT4	4-bit integer weights	PRO-INT	Complete
EXT-BF16	BFloat16 operations	PRO-FP	Experimental
EXT-FP8E4M3	8-bit floating-point operations E4M3	PRO-FP	Experimental
EXT-FP8E5M2	8-bit floating-point operations E5M2	PRO-FP	Experimental
EXT-FFT	Fast Fourier Transform operations	PRO-FP	Complete
EXT-VARIABLE	Stateful variable operations	PRO-INT,PRO-FP	Experimental
EXT-CONTROLFLOW	Control Flow operations	PRO-INT,PRO-FP	Experimental
EXT-DYNAMIC	Removes all Compile Time Constant state for CTC inputs	PRO-INT,PRO-FP	Experimental

Name	Description	Allowed profiles	Specification Status
EXT-DOUBLEROUND	Adds double rounding support to the RESCALE operator	PRO-INT	Complete
EXT-INEXACTROUND	Adds inexact rounding support to the RESCALE operator	PRO-INT	Experimental

1.7. Levels

A TOSA level defines operator argument ranges that an implementation shall support. This is distinct from a profile that defines the operations and data-types supported. One level must apply to all profiles and extensions supported by an implementation.

This version of the specification defines two TOSA levels:

- No level : allows the full range of arguments specified by the operations according to the operation data types.
- Level 8K : ranges are expected to be sufficient for applications with frame sizes up to 8K.

Later versions of the specification may define additional levels. The following table defines the value ranges for each level. These ranges are checked using the LEVEL_CHECK() function with the operator descriptions.

Table 4. Level maxima

tosa_level_t	tosa_level_none	tosa_level_8K
Description	No level	Level 8K
MAX_RANK	32	6
MAX_KERNEL	2147483647	8192
MAX_STRIDE	2147483647	8192
MAX_SCALE	2048	256
MAX_LOG2_SIZE	63	31
MAX_NESTING	256	6
MAX_TENSOR_LIST_SIZE	256	64

1.8. Status

This specification is the release candidate for TOSA 1.0.

The specific status of each profile and extension is contained in the tables in Profiles. Possible values for status are:

Status	Description
Complete	All operators are specified, conformance tests are provided, no changes are expected. Backward compatibility is guaranteed.
Experimental	Operators are subject to change, backwards compatibility is not guaranteed for experimental extensions. If an implementation implements an experimental extension, it must pass the conformance tests for that extension.
Deprecated	Operators retained for compatibility, but may be removed in a future major release of TOSA.

1.9. 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. A TOSA implementation must support the number formats listed in the supported data types for operators contained in that profile. Number formats not required for any operators in a profile do not need to be implemented.

Table 5. Number formats

Format	Minimum	Maximum	Description
bool_t	-	-	Boolean value that is either `true` or `false`. Size is 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`.
i4_t	-	-	Signless 4-bit integer type. Will be interpreted as int4_t by all operators
int4_t	-7	+7	Signed 4-bit two's-complement value. Excludes -8 to maintain a symmetric about zero range for weights.
i8_t	-	-	Signless 8-bit integer value. Will be interpreted as int8_t unless otherwise specified by an operator.
int8_t	-128	+127	Signed 8-bit two's-complement value.
uint8_t	0	255	Unsigned 8-bit integer value.
i16_t	-	-	Signless 16-bit integer type. Will be interpreted as int16_t unless otherwise specified by an operator.
int16_t	-32768	+32767	Signed 16-bit two's-complement value.
uint16_t	0	65535	Unsigned 16-bit value.
i32_t	-	-	Signless 32-bit integer value. Will be interpreted as int32_t by all operators.
int32_t	-(1<<31)	(1<<31)-1	Signed 32-bit two's-complement value.
i48_t	-	-	Signless 48-bit integer value. Will be interpreted as int48_t by all operators.

Format	Minimum	Maximum	Description
int48_t	-(1<<47)	(1<<47)-1	Signed 48-bit two's-complement value.
fp8e4m3_t	-448	448	8-bit floating-point defined by OCP-OFP8 with four bits of exponent and three bits of mantissa. Normal values must be supported. Subnormal values must be supported. NaN encodings must be supported. Signed zero must be supported. This format has no encoding for infinities. The range is extended by using a mantissa-exponent bit pattern to encode NaN instead of sacrificing an exponent value.
fp8e5m2_t	-infinity	+infinity	8-bit floating-point defined by OCP-OFP8 with five bits of exponent and two bits of mantissa. Normal values must be supported. Subnormal values must be supported. Positive and negative infinity must be supported. NaN encodings must be supported. Signed zero must be supported.
fp16_t	-infinity	+infinity	16-bit half-precision floating-point defined by IEEE-754. Normal values must be supported. Subnormal values must either be supported or flushed to sign-preserved zero. Positive and negative infinity must be supported. At least one NaN encoding must be supported. Signed zero must be supported.
bf16_t	-infinity	+infinity	16-bit brain floating-point defined as bits [31:16] of the fp32_t format. Normal values must be supported. Subnormal values must either be supported or flushed to sign-preserved zero. Positive and negative infinity must be supported. At least one NaN encoding must be supported. Signed zero must be supported.
fp32_t	-infinity	+infinity	32-bit single-precision floating-point defined by IEEE-754. Normal values must be supported. Subnormal values must either be supported or flushed to sign-preserved zero. Positive and negative infinity must be supported. At least one NaN encoding must be supported. Signed zero must be supported.

Format	Minimum	Maximum	Description
fp64_t	-infinity	+ infinity	64-bit double-precision floating-point defined by IEEE-754. Normal values must be supported. Subnormal values must either be supported or flushed to sign-preserved zero. Positive and negative infinity must be supported. At least one NaN encoding must be supported. Signed zero must be supported.

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 are given in the preceding 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, the result of a convolution with low precision data must equal that of running the convolution at 8 bits and then clipping the result to the permitted output range. This ensures that an Integer profile TOSA implementation can calculate the same result.

1.10. Compliance

This section defines when a TOSA implementation is compliant to a given TOSA specification profile and level. To be compliant an implementation must achieve the results and accuracy defined by this specification. TOSA also defines a set of conformance tests. A compliant implementation must pass the conformance tests. The conformance tests are not exhaustive, so an implementation that passes the conformance tests may not be compliant if there is a non-compliance that is undetected by the tests.

1.10.1. TOSA Graph 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 pseudocode 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 must behave as follows given the above tosa_graph result values:

- 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 pseudocode, if **graph** is a TOSA graph consisting of TOSA operators and **input_list** is a list of input tensors then the following test must pass.

```
// Global result status value
// Will be updated by REQUIRE and ERROR_IF statements when evaluating the TOSA graph
tosa_result_t tosa_graph_result;
// Tracks the nesting depth of TOSA operators to allow a limit on nesting depth to be
checked.
int32_t tosa_nesting_depth;

bool_t tosa_test_compliance(tosa_graph_t graph, tensor_list_t input_list, tosa_level_t
level) {
    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_nesting_depth = 0;            // if/while nesting level
    tosa_execute_graph(graph, input_list, output_list_spec, level);
    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;
}
```

1.10.2. Integer Profile Compliance

An Integer profile compliant implementation must satisfy the following:

- The implementation must support all operator and data type combinations listed in Integer.
  - The operations must meet the Integer Precision Requirements.
- The implementation must follow the TOSA Graph Compliance behavior.

Integer Precision Requirements

In a compliant implementation, individual integer operations within the graph must match exactly.

1.10.3. Floating-Point Profile Compliance

A Floating-Point profile compliant implementation must satisfy the following:

- The implementation must support all operator and data type combinations listed in Floating-Point.
  - The operations must meet the Floating-Point Precision Requirements.
  - Note: These requirements allow fp16_t operations to be implemented using the fp32_t datatype.
- The implementation must follow the TOSA Graph Compliance behavior.

Floating-Point Precision Requirements

When evaluating floating-point precision, *ulp* means unit of the last place.

In a compliant implementation, individual integer operations must match exactly. To check exact matching, the tosa_reference_check_fp function can be used with num_ulp set to 0. In a compliant implementation, individual floating-point operations within the graph must meet the accuracy bounds defined in each operator definition.

When the operator precision definitions refer to a result "calculated by fp64_t arithmetic", the following rules must be followed:

- The operation order must follow that specified in the operator pseudocode.
- Each fp64_t operation result must be round-to-nearest-even of the infinite precision result as defined by IEEE-754 round to nearest even rounding.

The function tosa_reference_check_fp() defines the error range permitted by a given number of units of last place in this specification. For data types that allow subnormal values to be flushed to zero, either all values must be flushed to sign-preserved zero, or none of them.

Operator sequence precision requirement

Precision criteria are specified for a single operator.

An implementation M of a sequence of n TOSA operators, A[0] to A[n-1] is said to be compliant if M gives the same result as a sequence of implementations M[0] to M[n-1] such that:

- Each M[k] implements A[k] with same or higher precision datatypes.
- Each M[k] meets the accuracy defined in this specification for A[k] where the M[k] output is converted to A[k] output precision using round to nearest.

Dot product accuracy requirements

This section assumes an operation acting on tensors named 'input', 'weight' and optionally 'bias'. Each output tensor element can be expressed as a dot product of elements between the 'input' and 'weight' tensors with optional bias addition. The dot product has length KS, the kernel size. If the operation does not specify a bias then 'bias' is taken to be zero in this section. Note: KS is defined for each relevant operator in the appendix section Floating-Point Operator Test Data.

In other words, each output element `out` can be expressed as a dot product between input elements `in[k]`, weight elements `w[k]`, bias `b`:

```
out = in[0] * w[0] + in[1] * w[1] + ⋯ + in[KS-1] * w[KS-1] + b
```

The positions of `in[k]`, `w[k]`, `b` in the input, weight and bias tensors depends on the operation being performed. This may be, for example, a convolution.

This section defines the accuracy required for these operations. In this section:

- "fp64 arithmetic" refers to double-precision floating-point arithmetic defined by IEEE-754.
- `reference_fp64()` is the result of the operation calculated using fp64_t arithmetic.
- `implementation()` is the implementation under test.
- `local_bound` is defined as follows:
  - For operations with a local_bound attribute it is the value of the optional attribute, with default value of false.
  - For operations that do not have a local_bound attribute the value is true.

For the checks described in the following code:

- Data sets defined for the operation in Appendix A Floating-Point Operator Test Data must pass.

The following tables describe some of the constraints applied during the dot product conformance check.

ABS_BOUND is the maximum allowed absolute error when NaN or overflow is not present.

Condition	ABS_BOUND	Notes
all cases	`2 * ksb`	Allow one rounding error for each accumulation. The 2 factor allows for different rounding modes.

VARIANCE_ERROR_BOUND is the maximum allowed variance across the entire output tensor. The squared error for each result is summed, and the result must be less than the VARIANCE_ERROR_BOUND.

Condition	VARIANCE_ERROR_BOUND	Notes
all cases	`4 * 0.4 * ksb`	The 0.4 factor is derived from the uniform [-1,1] distribution variance of 1/3 by rounding up. The 4 factor is the square of the 2 factor in the absolute bound to allow for different rounding modes.

```
bool_t tosa_reference_check_dotproduct<OP, in_t, weight_t, out_t, acc_t>(
```

```
    i32_t S,             // test set number
    T<in_t> input,       // input tensor
    T<weight_t> weight,  // weight tensor
    T<out_t> bias        // bias tensor
) {
    T<in_t>     input_abs  = apply_max_s<in_t>(abs(input),      normal_min<in_t>());
// Element-wise
    T<weight_t> weight_abs = apply_max_s<weight_t>(abs(weight), normal_min<in_t>());
// Element-wise
    T<out_t>    bias_abs   = apply_max_s<out_t>(abs(bias),      normal_min<in_t>());
// Element-wise
    if (!local_bound) {
        in_t input_abs_max = max_value(input_abs);  // maximum over all elements
        for_each_data_position(index in shape(input_abs)) {
            input_abs[index] = input_abs_max;  // set all entries to global maximum
        }
    }
    // reference_fp64(OP) calculates the results of operation OP using fp64_t
arithmetic
    // implementation(OP) runs operation OP using the implementation under test
    T<out_t>  output_imp = implementation(OP, input, weight, bias,
tosa_extra_multiplies = IMPLEMENTATION_DEFINED);
    T<fp64_t> output_ref = reference_fp64(OP, input, weight, bias,
tosa_extra_multiplies = false);
    T<fp64_t> output_bnd = reference_fp64(OP, input_abs, weight_abs, bias_abs,
tosa_extra_multiplies = true);

    tensor_size_t T = tensor_size(output_shape);  // number dot product results
    tensor_size_t ksb = ceil(KS / pow(2, (normal_frac<acc_t>() -
normal_frac<out_t>())/2)) + ((max_value(bias_abs) > 0) ? 1 : 0);
    fp64_t out_err_sum = 0.0;
    fp64_t out_err_sumsq = 0.0;
    for_each_data_position(index in output_shape) {
        fp64_t out_bnd_el = tensor_read<fp64_t>(output_bnd, output_shape, index);
        fp64_t out_ref_el = tensor_read<fp64_t>(output_ref, output_shape, index);
        acc_t  out_imp_el = tensor_read<acc_t> (output_imp, output_shape, index);
        fp64_t out_err;

        if (isNaN(out_ref_el)) {
            // Reference is a NaN on non-padded data, the implementation must match
            if (!isNaN(out_imp_el)) {
                return false;
            }
            out_err = 0.0;
        } else if (isNaN(out_bnd_el)) {
            // No further accuracy requirements for a NaN bound
            out_err = 0.0;
        } else if.(static_cast<out_t>(out_bnd_el * (1 + ABS_BOUND * exp2(-1-
normal_frac<out_t>()))) == infinity) {
            // dot product can overflow within error bound and there is no accuracy
limit
```

```
            out_err = 0.0;
        } else if (out_bnd_el == 0.0) {
            // All products in the dot product are zero
            if (out_ref_el != 0.0 || out_imp_el != 0.0) {
                return false;
            }
            out_err = 0.0;
        } else {  // 0.0 < out_bnd < infinity
            fp64_t out_err_bnd = max(out_bnd_el * exp2(-1-normal_frac<out_t>()),
normal_min<out_t>());
            out_err = (static_cast<fp64_t>(out_imp_el) - out_ref_el) / out_err_bnd;
            // Check the absolute error. See the table for the definition of
ABS_BOUND.
            if (abs(out_err) > ABS_BOUND) {
                return false;
            }
        }
        out_err_sum   += out_err;
        out_err_sumsq += out_err * out_err;
    }
    // Only check this for input and weights for test data sets 3-5
    if ( S >= 3 && S <= 5) {
        // check output error bias magnitude for data sets S which are not positive
biased
        // The factor 10 allows for up to a 4 sigma difference of the error sum around
the
        // expected error sum assuming errors are normally distributed.
        if (abs(out_err_sum) > sqrt(10 * VARIANCE_ERROR_BOUND * T)) {
            return false;
        }
    }
    // check output error variance magnitude
    // See the table for the definition of VARIANCE_ERROR_BOUND
    if (out_err_sumsq > VARIANCE_ERROR_BOUND * T) {
        return false;
    }
    return true;
}
```

1.11. Tensor Definitions

1.11.1. 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. 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.

The shape of a tensor is a special type `shape_t`. `shape_t` is a one-dimensional list with the size equal to the rank of the original tensor. The components of a `shape_t` are of type `tensor_size_t`. `tensor_size_t` is a signed integer as it may be used for negative offsets. This type must be able to hold integers in the range [`-(1 << MAX_LOG2_SIZE)` .. `(1 << MAX_LOG2_SIZE) - 1`] where `MAX_LOG2_SIZE` is defined in Levels. The `shape_t` for a zero-dimensional tensor is the empty list.

For each tensor, the number of tensor elements multiplied by the element size in bytes (which is taken to be 1 for elements smaller than a 8-bit) must be representable as a `tensor_size_t`.

In this version of the specification, `shape_t` values must be resolvable to constants at backend compile time.

1.11.2. 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 6. 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

1.11.3. 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 Helpers.

1.12. Integer Behavior

TOSA integer inputs and outputs are specified by signless values with the given number of bits. Unless otherwise specified, these values will be interpreted as signed two's-complement. The pseudocode will use int*_t to indicate use as a signed value and uint*_t to indicate use as an unsigned value. 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.

1.12.1. 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 to belong to the same domain. The scale arguments for RESCALE can be calculated to ensure that the resulting tensors belong to the same domain. 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.

1.12.2. Precision Scaling

TOSA uses the RESCALE operation to scale between values with differing precision.

1.12.3. 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.

1.12.4. 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.

1.12.5. 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_s(int16_t *table, int32_t value)
{
    int16_t clipped_value = static_cast<int16_t>(apply_clip_s<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] = static_cast<int16_t>(apply_clip_s<int32_t>(value, -32768,
+32767));
    }
}
```

1.13. Reporting Errors

If you find any errors in this document, please feel free to report them by sending an errata entry. To propose new features or text in the specification, please refer to the TOSA contribution process, including the contribution agreement: Specification Contributions To submit an erratum, kindly

include the following information:

1. Section or Location: The section, chapter, or specific location where the error appears.
2. Description of the Error: A brief explanation of the issue you found (e.g., typo, formatting problem, inaccuracy, unclear wording).
3. Corrected Text: The corrected or suggested clarified text (if applicable).

Please send this information via email to: tosa.errata@arm.com.

1.14. Other Publications

The following publications are referred to in this specification, or provide more information:

1. IEEE Std 754-2008, *IEEE Standard for Floating-point Arithmetic*, August 2008.
2. Open Compute Project OCP 8-bit Floating Point Specification (OFP8) Revision 1.0

2. Operators

2.1. Operator Arguments

Operators process input arguments to produce output arguments. Their behavior can be configured using attribute arguments. Arguments may have one of the following types:

- `tensor_t<element_type>`, abbreviated `T<element_type>`, represents a tensor whose elements are of type `element_type` where `element_type` can be any of the data types supported in TOSA.
- `tensor_list_t` represents a list of tensors. When lists are homogeneous, containing tensors of the same type, their type is further qualified as follows: `tensor_list_t<T<element_type>>`.
  The maximum number of elements in a tensor list is set by the MAX_TENSOR_LIST_SIZE level parameter.
- `tosa_graph_t` represents a TOSA graph (see Operator Graphs).

Arguments belong to one of three categories: Input, Output, or Attribute. The category to which an argument belongs further constrains its type:

- An Input argument must be a tensor or a list of tensors used to provide the data read by the operation.
- An Attribute argument is constant, its value is always known at compilation time.
- An Output argument must be a tensor or a list of tensors into which the data produced by the operation is written.

Profiles may restrict a set of inputs and/or outputs to be known (constant) at compile time. Compile time is defined as the time at which TOSA operators are converted to implementation specific commands prior to execution. This type of input or output is called a Compile Time Constant (CTC).

- A TOSA profile may define some input and/or output arguments to be CTC.

- A TOSA extension may change the state of a CTC back to a non-constant input or output.
- A TOSA extension is not allowed to change a non-constant input or output to a CTC.
- When a profile defines a CTC for an operator a **Compile Time Constant Status** table will be defined for that operator.
  For any arguments in the table, the profiles under which it must be constant are listed as well as the extensions which change this back to a variable input or output.
- For a TOSA conformant graph a CTC input must be connected to a CTC output, such as the output of a CONST or CONST_SHAPE operator.

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.
- The inputs to a graph are a list of tensors, and may not include any elements of type shape_t.
- The outputs of a graph are a list of tensors, and may not include any elements of type shape_t.

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(tensor_list_t tensor_list);
```

The following function denotes the execution of a TOSA graph within a TOSA context, on an input tensor list to produce an output tensor list. A TOSA context, represented by `tosa_context_t` provides the environment in which a TOSA graph is executed. Any side-effects that result from the execution of a graph within a context are not observable by graphs executing in a different context. Operators are executed in an implementation-defined order that must be a topological ordering of the TOSA graph.

```
tosa_execute_graph(tosa_context_t context, tosa_graph_t graph, tensor_list_t
```

```
input_list, tensor_list_t output_list, tosa_level_t level) {
    ERROR_IF(tensor_list_shape(input_list) != tosa_input_shape(graph));
    ERROR_IF(tensor_list_shape(output_list) != tosa_output_shape(graph));

    // Declare the global list for storing persistent variable tensors across multiple
graphs
    if (!variable_tensors) {
        variable_tensors = list<tensor_t>();
    } else { // Clear the "seen flag"
        for (tensor_t var_tensor in variable_tensors) {
            var_tensor.seen = false;
        }
    }

    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 the operator as defined by the operation function pseduo-code
        tosa_execute_operator(context, operator, level);
    }
}
```

2.3. Tensor Operators

2.3.1. ARGMAX

This returns the index with the largest value across the given axis of the input tensor. If multiple locations have equal values, returns the first match along the search axis.

Precision Requirements

Integer results must be exact.

NaN propagation mode only affects floating-point types. It indicates either propagating or ignoring NaN.

For floating-point input, the following rules apply:

- In the NaN propagating mode, NaN values always compare as greater than non-NaN values.
- In the NaN ignoring mode, NaN values always compare as less than non-NaN values.
- The sign of zero is ignored when comparing values.
- Infinities of the same sign compare as equal.

- All NaN values compare as equal to other NaN values.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_t>	input	shape1	1 to MAX_RANK	Input tensor
Attribute	i32_t	axis	-		Axis in range from 0 to rank(shape1) - 1
Attribute	nan_propagation_mode_t	nan_mode	-		PROPAGATE or IGNORE. Set to PROPAGATE by default. This attribute affects the floating-point NaN propagation approach. This attribute is ignored by non floating-point types.
Output	T<out_t>	output	shape	0 to MAX_RANK - 1	Output tensor, with rank = rank(shape1) - 1

Supported Data Types:

Profile/Extension	Mode	in_t	out_t
PRO-FP	fp16	fp16_t	i32_t
PRO-FP	fp32	fp32_t	i32_t
PRO-INT	signed 8	i8_t	i32_t
EXT-BF16	bf16	bf16_t	i32_t
EXT-FP8E4M3	fp8e4m3	fp8e4m3_t	i32_t
EXT-FP8E5M2	fp8e5m2	fp8e5m2_t	i32_t
EXT-INT16	signed 16	i16_t	i32_t

Operation Function:

```
LEVEL_CHECK(rank(shape1) <= MAX_RANK);
```

```
ERROR_IF(axis < 0 || axis >= rank(shape1));
shape_t left_shape, right_shape;
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_data_position(left_index in left_shape) {
    for_each_data_position(right_index in right_shape) {
        in_t max_value = (is_floating_point<in_t>() && nan_mode == IGNORE)
                             ? nan<in_t>()
                             : minimum_s<in_t>();
        out_t max_index = 0;
        for (tensor_size_t i = 0; i < shape1[axis]; i++) {
            shape_t index = flatten(left_index, [i], right_index);
            in_t value = tensor_read<in_t>(input, shape1, index);
            in_t result = apply_max_s<in_t>(value, max_value, nan_mode);
            if (result != max_value) {
                if (!(isNaN(result) && isNaN(max_value))) {
                    max_value = result;
                    max_index = i;
                }
            }
        }
        shape_t index = flatten(left_index, right_index);
        tensor_write<out_t>(output, shape, index, max_index);
    }
}
```

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.

Precision Requirements

Integer results must be exact.

For floating-point values, the following rules apply:

- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero before calculation.
- Outputs can be expressed as a dot product of an input vector with a vector with elements 1/KS where KS is the kernel size.
- This dot product must meet the Dot product accuracy requirements.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input	[N,IH,IW,C]	4	Input tensor

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input_zp	[1]	1	Input tensor zero point. Must be zero for non-int8 types.
Input	T<in_out_t>	output_zp	[1]	1	Output tensor zero point. Must be zero for non-int8 types.
Attribute	T<i32_t>	kernel	[2]	1	[kernel_y, kernel_x]
Attribute	T<i32_t>	stride	[2]	1	[stride_y, stride_x]
Attribute	T<i32_t>	pad	[4]	1	[pad_top, pad_bottom, pad_left, pad_right]
Attribute	acc_type_t	acc_type	-		Enumerated type, must be one of INT32, FP16, FP32 matching the type of acc_t in the Supported Data Types table for this operation
Output	T<in_out_t>	output	[N,OH,OW,C]	4	Output tensor 4D

Compile Time Constant Status:

Argument	CTC enabled profile(s)	CTC disabled extension(s)
input_zp	PRO-INT, PRO-FP	EXT-DYNAMIC
output_zp	PRO-INT, PRO-FP	EXT-DYNAMIC

Supported Data Types:

Profile/Extension	Mode	in_out_t	acc_t
PRO-FP	fp16 with fp16 accumulate	fp16_t	fp16_t
PRO-FP	fp16 with fp32 accumulate	fp16_t	fp32_t
PRO-FP	fp32 with fp32 accumulate	fp32_t	fp32_t
PRO-INT	signed 8 with int32 accumulate	i8_t	i32_t
EXT-BF16	bf16 with fp32 accumulate	bf16_t	fp32_t
EXT-FP8E4M3	fp8e4m3 with fp16 accumulate	fp8e4m3_t	fp16_t
EXT-FP8E5M2	fp8e5m2 with fp16 accumulate	fp8e5m2_t	fp16_t
EXT-INT16	signed 16 with int32 accumulate	i16_t	i32_t

Operation Function:

```
LEVEL_CHECK(kernel_y <= MAX_KERNEL);
LEVEL_CHECK(kernel_x <= MAX_KERNEL);
LEVEL_CHECK(stride_y <= MAX_STRIDE);
LEVEL_CHECK(stride_x <= MAX_STRIDE);
LEVEL_CHECK(pad_top <= MAX_KERNEL);
LEVEL_CHECK(pad_bottom <= MAX_KERNEL);
LEVEL_CHECK(pad_left <= MAX_KERNEL);
LEVEL_CHECK(pad_right <= MAX_KERNEL);
```

```
ERROR_IF(!is_same<in_out_t,i8_t>() && input_zp != 0); // Zero point only for int8_t
ERROR_IF(!is_same<in_out_t,i8_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;
    tensor_size_t iy = oy * stride_y - pad_top;
    tensor_size_t ix = ox * stride_x - pad_left;
    for_each(0 <= ky < kernel_y, 0 <= kx < kernel_x) {
        tensor_size_t y = iy + ky;
        tensor_size_t x = ix + kx;
        // Only values from the input tensor are used to calculate the
        // average, padding does not count
        if (0 <= y < IH && 0 <= x < IW) {
            count++;
            acc_t value = sign_extend<acc_t>(tensor_read<in_out_t>(input, [N,IH,IW,C],
[n,y,x,c]));
            value = apply_sub_s<acc_t>(value, sign_extend<acc_t>(input_zp));
            acc = apply_add_s<acc_t>(acc, value);
        }
    }
    if (is_floating_point<in_out_t>()) {
        output_val = acc / static_cast<in_out_t>(count);
    } else {
        scale_t scale = reciprocal_scale(count);
        acc = apply_scale_32(acc, scale.multiplier, scale.shift, false);
        acc = apply_add_s<acc_t>(acc, sign_extend<acc_t>(output_zp));
        acc = apply_clip_s<acc_t>(acc, minimum_s<in_out_t>(), maximum_s<in_out_t>());
```

```
        output_val = static_cast<in_out_t>(acc);
    }
    tensor_write<in_out_t>(output, [N,OH,OW,C], [n,oy,ox,c], output_val);
}
```

2.3.3. CONV2D

Performs a 2D convolution over the given tensor input, using the weight tensor. Implementations may choose to skip calculation of multiplies in the padding area.

Precision Requirements

Integer results must be exact.

For floating-point values, the following rules apply:

- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero before calculation.
- Each output can be expressed as a dot product of two input vectors.
- The dot product must meet the Dot product accuracy requirements.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_t>	input	[N,IH,IW,IC]	4	Input tensor
Input	T<weight_t>	weight	[OC,KH,KW,IC]	4	Weight kernel size KH x KW
Input	T<out_t>	bias	[BC]	1	Per output channel bias data. Bias data will be broadcast if BC == 1.
Input	T<in_t>	input_zp	[1]	1	Input tensor zero point. Must be zero for non-int8 types.
Input	T<weight_t>	weight_zp	[1]	1	Weight zero point. Must be zero for non-int8 types.
Attribute	T<i32_t>	pad	[4]	1	[pad_top, pad_bottom, pad_left, pad_right]
Attribute	T<i32_t>	stride	[2]	1	[stride_y, stride_x]
Attribute	T<i32_t>	dilation	[2]	1	[dilation_y, dilation_x]
Attribute	acc_type_t	acc_type	-		Enumerated type, must be one of INT32, INT48, FP16, FP32 matching the type of acc_t in the Supported Data Types table for this operation

Argument	Type	Name	Shape	Rank	Description
Attribute	bool_t	local_bound	-		This optional attribute affects the floating-point compliance error bound. The default of false allows for direct and transform based, fast convolution algorithms. Only set to true if direct dot-product calculation precision is required.
Output	T<out_t>	output	[N,OH,OW,OC]	4	Output tensor

Compile Time Constant Status:

Argument	CTC enabled profile(s)	CTC disabled extension(s)
input_zp	PRO-INT, PRO-FP	EXT-DYNAMIC
weight_zp	PRO-INT, PRO-FP	EXT-DYNAMIC

Supported Data Types:

Profile/Extension	Mode	in_t	weight_t	out_t	acc_t
PRO-FP	fp16 with fp16 accumulate	fp16_t	fp16_t	fp16_t	fp16_t
PRO-FP	fp16 with fp32 accumulate	fp16_t	fp16_t	fp16_t	fp32_t
PRO-FP	fp32 with fp32 accumulate	fp32_t	fp32_t	fp32_t	fp32_t
PRO-INT	signed 8x8 with int32 accumulate	i8_t	i8_t	i32_t	i32_t
EXT-BF16	bf16 with fp32 accumulate	bf16_t	bf16_t	bf16_t	fp32_t
EXT-FP8E4M3	fp8e4m3 with fp16 accumulate	fp8e4m3_t	fp8e4m3_t	fp16_t	fp16_t
EXT-FP8E5M2	fp8e5m2 with fp16 accumulate	fp8e5m2_t	fp8e5m2_t	fp16_t	fp16_t
EXT-INT16	signed 16x8 with int48 accumulate	i16_t	i8_t	i48_t	i48_t

Profile/Extension	Mode	in_t	weight_t	out_t	acc_t
EXT-INT4	signed 8x4 with int32 accumulate	i8_t	i4_t	i32_t	i32_t

Operation Function:

```
LEVEL_CHECK(dilation_y * KH <= MAX_KERNEL);
LEVEL_CHECK(dilation_x * KW <= MAX_KERNEL);
LEVEL_CHECK(pad_top <= MAX_KERNEL);
LEVEL_CHECK(pad_bottom <= MAX_KERNEL);
LEVEL_CHECK(pad_left <= MAX_KERNEL);
LEVEL_CHECK(pad_right <= MAX_KERNEL);
LEVEL_CHECK(stride_y <= MAX_STRIDE);
LEVEL_CHECK(stride_x <= MAX_STRIDE);
```

```
ERROR_IF(!is_same<in_t,i8_t>() && input_zp != 0); // Zero point only for int8_t
ERROR_IF(!is_same<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);
ERROR_IF(BC != OC && BC != 1);

for_each(0 <= n < N, 0 <= oy < OH, 0 <= ox < OW, 0 <= oc < OC) {
    acc_t acc = 0;
    tensor_size_t iy = oy * stride_y - pad_top;
    tensor_size_t ix = ox * stride_x - pad_left;
    for_each(0 <= ky < KH, 0 <= kx < KW, 0 <= ic < IC) {
        tensor_size_t y = iy + ky * dilation_y;
        tensor_size_t x = ix + kx * dilation_x;
        acc_t value = 0;
        if (0 <= y < IH && 0 <= x < IW) {
            value  = static_cast<acc_t>(tensor_read<in_t>(input,
                                                          [N,IH,IW,IC],
                                                          [n,y,x,ic]));
            value  = apply_sub_s<acc_t>(value, static_cast<acc_t>(input_zp));
        }
        if ((0 <= y < IH && 0 <= x < IW) || tosa_extra_multiplies) {
            acc_t weight_el = static_cast<acc_t>(tensor_read<weight_t>(weight,
                                                                       [OC,KH,KW,IC],

[oc,ky,kx,ic]));
            weight_el = apply_sub_s<acc_t>(weight_el, static_cast<acc_t>(weight_zp));
```

```
            acc = apply_add_s<acc_t>(acc, apply_mul_s<acc_t>(value, weight_el));
        }
    }
    out_t out = static_cast<out_t>(acc);
    out = apply_add_s<out_t>(out, bias[(BC == 1) ? 0 : oc]);
    tensor_write<out_t>(output, [N,OH,OW,OC], [n,oy,ox,oc], out);
}
```

2.3.4. CONV3D

Performs a 3D convolution over the given input tensor. Implementations may choose to skip calculation of multiplies in the padding area.

Precision Requirements

Integer results must be exact.

For floating-point values, the following rules apply:

- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero before calculation.
- Each output can be expressed as a dot product of two input vectors.
- The dot product must meet the Dot product accuracy requirements.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_t>	input	[N,ID,IH,IW,IC]	5	Input tensor
Input	T<weight_t>	weight	[OC,KD,KH,KW,IC]	5	Weight kernel size KDxKHxKW
Input	T<out_t>	bias	[BC]	1	Per output channel bias data. Bias data will be broadcast if BC == 1.
Input	T<in_t>	input_zp	[1]	1	Input tensor zero point. Must be zero for non-int8 types.
Input	T<weight_t>	weight_zp	[1]	1	Weight zero point. Must be zero for non-int8 types.
Attribute	T<i32_t>	pad	[6]	1	[pad_d0, pad_d1, pad_top, pad_bottom, pad_left, pad_right]
Attribute	T<i32_t>	stride	[3]	1	[stride_d, stride_y, stride_x]
Attribute	T<i32_t>	dilation	[3]	1	[dilation_d, dilation_y, dilation_x]

Argument	Type	Name	Shape	Rank	Description
Attribute	acc_type_t	acc_type	-		Enumerated type, must be one of INT32, INT48, FP16, FP32 matching the type of acc_t in the Supported Data Types table for this operation
Attribute	bool_t	local_bound	-		This optional attribute affects the floating-point compliance error bound. The default of false allows for direct and transform based, fast convolution algorithms. Only set to true if direct dot-product calculation precision is required.
Output	T<out_t>	output	[N,OD,OH,OW,OC]	5	Output tensor

Compile Time Constant Status:

Argument	CTC enabled profile(s)	CTC disabled extension(s)
input_zp	PRO-INT, PRO-FP	EXT-DYNAMIC
weight_zp	PRO-INT, PRO-FP	EXT-DYNAMIC

Supported Data Types:

Profile/Extension	Mode	in_t	weight_t	out_t	acc_t
PRO-FP	fp16 with fp16 accumulate	fp16_t	fp16_t	fp16_t	fp16_t
PRO-FP	fp16 with fp32 accumulate	fp16_t	fp16_t	fp16_t	fp32_t
PRO-FP	fp32 with fp32 accumulate	fp32_t	fp32_t	fp32_t	fp32_t
PRO-INT	signed 8x8 with int32 accumulate	i8_t	i8_t	i32_t	i32_t
EXT-BF16	bf16 with fp32 accumulate	bf16_t	bf16_t	bf16_t	fp32_t
EXT-FP8E4M3	fp8e4m3 with fp16 accumulate	fp8e4m3_t	fp8e4m3_t	fp16_t	fp16_t
EXT-FP8E5M2	fp8e5m2 with fp16 accumulate	fp8e5m2_t	fp8e5m2_t	fp16_t	fp16_t

Profile/Extension	Mode	in_t	weight_t	out_t	acc_t
EXT-INT16	signed 16x8 with int48 accumulate	i16_t	i8_t	i48_t	i48_t
EXT-INT4	signed 8x4 with int32 accumulate	i8_t	i4_t	i32_t	i32_t

Operation Function:

```
LEVEL_CHECK(dilation_d * KD <= MAX_KERNEL);
LEVEL_CHECK(dilation_y * KH <= MAX_KERNEL);
LEVEL_CHECK(dilation_x * KW <= MAX_KERNEL);
LEVEL_CHECK(pad_d0 <= MAX_KERNEL);
LEVEL_CHECK(pad_d1 <= MAX_KERNEL);
LEVEL_CHECK(pad_top <= MAX_KERNEL);
LEVEL_CHECK(pad_bottom <= MAX_KERNEL);
LEVEL_CHECK(pad_left <= MAX_KERNEL);
LEVEL_CHECK(pad_right <= MAX_KERNEL);
LEVEL_CHECK(stride_y <= MAX_STRIDE);
LEVEL_CHECK(stride_x <= MAX_STRIDE);
LEVEL_CHECK(stride_d <= MAX_STRIDE);
```

```
ERROR_IF(!is_same<in_t,i8_t>() && input_zp != 0); // Zero point only for int8_t
ERROR_IF(!is_same<weight_t,i8_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);
ERROR_IF(BC != OC && BC != 1);

for_each(0 <= n < N, 0 <= od < OD, 0 <= oy < OH, 0 <= ox < OW, 0 <= oc < OC) {
    acc_t acc = 0;
    tensor_size_t id = od * stride_d - pad_d0;
    tensor_size_t iy = oy * stride_y - pad_top;
    tensor_size_t ix = ox * stride_x - pad_left;
    for_each(0 <= kd < KD, 0 <= ky < KH, 0 <= kx < KW, 0 <= ic < IC) {
        tensor_size_t d = id + kd * dilation_d;
        tensor_size_t y = iy + ky * dilation_y;
        tensor_size_t x = ix + kx * dilation_x;
```

```
        acc_t value = 0;
        if (0 <= x < IW && 0 <= y < IH && 0 <= d < ID) {
            value  = static_cast<acc_t>(tensor_read<in_t>(input,
                                                          [N,ID,IH,IW,IC],
                                                          [n,d,y,x,ic]));
            value  = apply_sub_s<acc_t>(value, static_cast<acc_t>(input_zp));
        }
        if ((0 <= x < IW && 0 <= y < IH && 0 <= d < ID) || tosa_extra_multiplies) {
            acc_t weight_el = static_cast<acc_t>(tensor_read<weight_t>(weight,

[OC,KD,KH,KW,IC],

[oc,kd,ky,kx,ic]));
            weight_el = apply_sub_s<acc_t>(weight_el, static_cast<acc_t>(weight_zp));
            acc = apply_add_s<acc_t>(acc, apply_mul_s<acc_t>(value, weight_el));
        }
    }
    out_t out = static_cast<out_t>(acc);
    out = apply_add_s<out_t>(out, bias[(BC == 1) ? 0 : oc]);
    tensor_write<out_t>(output, [N,OD,OH,OW,OC], [n,od,oy,ox,oc], out);
}
```

2.3.5. DEPTHWISE_CONV2D

Performs 2D convolutions separately over each channel of the given tensor input, using the weight tensor. Implementations may choose to skip calculation of multiplies in the padding area.

Precision Requirements

Integer results must be exact.

For floating-point values, the following rules apply:

- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero before calculation.
- Each output can be expressed as a dot product of two input vectors.
- The dot product must meet the Dot product accuracy requirements.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_t>	input	[N,IH,IW,C]	4	Input tensor
Input	T<weight_t>	weight	[KH,KW,C,M]	4	Weight kernel size KH x KW
Input	T<out_t>	bias	[BC]	1	Per output channel bias data. Bias data will be broadcast if BC == 1.

Argument	Type	Name	Shape	Rank	Description
Input	T<in_t>	input_zp	[1]	1	Input tensor zero point. Must be zero for non-int8 types.
Input	T<weight_t>	weight_zp	[1]	1	Weight zero point. Must be zero for non-int8 types.
Attribute	T<i32_t>	pad	[4]	1	[pad_top, pad_bottom, pad_left, pad_right]
Attribute	T<i32_t>	stride	[2]	1	[stride_y, stride_x]
Attribute	T<i32_t>	dilation	[2]	1	[dilation_y, dilation_x]
Attribute	acc_type_t	acc_type	-		Enumerated type, must be one of INT32, INT48, FP16, FP32 matching the type of acc_t in the Supported Data Types table for this operation
Attribute	bool_t	local_bound	-		This optional attribute affects the floating-point compliance error bound. The default of false allows for direct and transform based, fast convolution algorithms. Only set to true if direct dot-product calculation precision is required.
Output	T<out_t>	output	[N,OH,OW,C*M]	4	Output tensor

Compile Time Constant Status:

Argument	CTC enabled profile(s)	CTC disabled extension(s)
input_zp	PRO-INT, PRO-FP	EXT-DYNAMIC
weight_zp	PRO-INT, PRO-FP	EXT-DYNAMIC

Supported Data Types:

Profile/Extension	Mode	in_t	weight_t	out_t	acc_t
PRO-FP	fp16 with fp16 accumulate	fp16_t	fp16_t	fp16_t	fp16_t
PRO-FP	fp16 with fp32 accumulate	fp16_t	fp16_t	fp16_t	fp32_t
PRO-FP	fp32 with fp32 accumulate	fp32_t	fp32_t	fp32_t	fp32_t

Profile/Extension	Mode	in_t	weight_t	out_t	acc_t
PRO-INT	signed 8x8 with int32 accumulate	i8_t	i8_t	i32_t	i32_t
EXT-BF16	bf16 with fp32 accumulate	bf16_t	bf16_t	bf16_t	fp32_t
EXT-FP8E4M3	fp8e4m3 with fp16 accumulate	fp8e4m3_t	fp8e4m3_t	fp16_t	fp16_t
EXT-FP8E5M2	fp8e5m2 with fp16 accumulate	fp8e5m2_t	fp8e5m2_t	fp16_t	fp16_t
EXT-INT16	signed 16x8 with int48 accumulate	i16_t	i8_t	i48_t	i48_t
EXT-INT4	signed 8x4 with int32 accumulate	i8_t	i4_t	i32_t	i32_t

Operation Function:

```
LEVEL_CHECK(dilation_y * KH <= MAX_KERNEL);
LEVEL_CHECK(dilation_x * KW <= MAX_KERNEL);
LEVEL_CHECK(pad_top <= MAX_KERNEL);
LEVEL_CHECK(pad_bottom <= MAX_KERNEL);
LEVEL_CHECK(pad_left <= MAX_KERNEL);
LEVEL_CHECK(pad_right <= MAX_KERNEL);
LEVEL_CHECK(stride_y <= MAX_STRIDE);
LEVEL_CHECK(stride_x <= MAX_STRIDE);
```

```
ERROR_IF(!is_same<in_t,i8_t>() && input_zp != 0); // Zero point only for int8_t
ERROR_IF(!is_same<weight_t,i8_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);
ERROR_IF(BC != C*M && BC != 1);

for_each(0 <= n < N, 0 <= oy < OH, 0 <= ox < OW, 0 <= c < C, 0 <= m < M) {
    acc_t acc = 0;
    tensor_size_t iy = oy * stride_y - pad_top;
```

```
    tensor_size_t ix = ox * stride_x - pad_left;
    for_each(0 <= ky < KH, 0 <= kx < KW) {
        tensor_size_t y = iy + ky * dilation_y;
        tensor_size_t x = ix + kx * dilation_x;
        acc_t value = 0;
        if (0 <= y < IH && 0 <= x < IW) {
            value  = static_cast<acc_t>(tensor_read<in_t>(input,
                                                          [N,IH,IW,C],
                                                          [n,y,x,c]));
            value  = apply_sub_s<acc_t>(value, static_cast<acc_t>(input_zp));
        }
        if ((0 <= y < IH && 0 <= x < IW) || tosa_extra_multiplies) {
            acc_t weight_el = static_cast<acc_t>(tensor_read<weight_t>(weight,
                                                                       [KH,KW,C,M],
                                                                       [ky,kx,c,m]));
            weight_el = apply_sub_s<acc_t>(weight_el, static_cast<acc_t>(weight_zp));
            acc = apply_add_s<acc_t>(acc, apply_mul_s<acc_t>(value, weight_el));
        }
    }
    out_t out = static_cast<out_t>(acc);
    out = apply_add_s<out_t>(out, bias[(BC == 1) ? 0 : (c * M) + m]);
    tensor_write<out_t>(output, [N,OH,OW,C * M], [n,oy,ox,c * M + m], out);
}
```

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)

Precision Requirements

- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero before calculation.
- Each output can be expressed as a dot product of an input vector with a constant coefficient vector.
- The following may be used to validate the output
  - Let `input_real` be the real input tensor.
  - Let `input_imag` be the imaginary input tensor.

- Let `weight_real` be the coefficient vector tensor of real values.
- Let `weight_imag` be the coefficient vector tensor of imaginary values.
- Let `input` be an interleaved tensor of real values from input tensors `input_real` and `input_imag` such that `input = [input_real[0], input_imag[0], input_real[1], input_imag[1], ⋯]` and `shape(input) = [N,H,W*2]`.
- Let `weight` be an interleaved tensor of real values from weight tensors `weight_real` and `weight_imag` such that `weight = [weight_real[0], weight_imag[0], weight_real[1], weight_imag[1], ⋯]` and `shape(weight) = [N,H,W*2]`.
- Let `FFT_Real` be the operation that calculates the real output (`output_real`).
- Let `FFT_Imag` be the operation that calculates the imaginary output (`output_imag`).
- For all `S` in the defined data sets in Appendix A Floating-Point Operator Test Data.
  - `tosa_reference_check_dotproduct<FFT_Real, in_out_t, in_out_t, in_out_t, in_out_t>(S, input, weight, [])` must be true.
  - `tosa_reference_check_dotproduct<FFT_Imag, in_out_t, in_out_t, in_out_t, in_out_t>(S, input, weight, [])` must be true.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input_real	[N,H,W]	3	Real part of the complex input. H,W must be powers of two.
Input	T<in_out_t>	input_imag	[N,H,W]	3	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
Attribute	bool_t	local_bound	-		This optional attribute affects the floating-point compliance error bound. The default of false allows for direct and transform based, fast convolution algorithms. Only set to true if direct dot-product calculation precision is required.
Output	T<in_out_t>	output_real	[N,H,W]	3	Real part of the complex output.
Output	T<in_out_t>	output_imag	[N,H,W]	3	Imaginary part of the complex output.

Supported Data Types:

Profile/Extension	Mode	in_out_t
EXT-FFT	fp32	fp32_t

Operation Function:

```
LEVEL_CHECK(H <= MAX_KERNEL);
LEVEL_CHECK(W <= MAX_KERNEL);
```

```
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]);
        tensor_size_t ay = (static_cast<tensor_size_t>(iy) *
static_cast<tensor_size_t>(oy)) % static_cast<tensor_size_t>(H);
        tensor_size_t ax = (static_cast<tensor_size_t>(ix) *
static_cast<tensor_size_t>(ox)) % static_cast<tensor_size_t>(W);
        in_out_t a = sign_val * 2 * pi() * (static_cast<in_out_t>(ay) / H +
static_cast<in_out_t>(ax) / 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);
}
```

2.3.7. MATMUL

Performs two dimensional matrix multiplications.

Precision Requirements

Integer results must be exact.

For floating-point values, the following rules apply:

- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero before calculation.
- Each output can be expressed as a dot product of two input vectors.
- The dot product must meet the Dot product accuracy requirements.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_t>	A	[N,H,C]	3	Input tensor A, N matrices of size HxC
Input	T<in_t>	B	[N,C,W]	3	Input tensor B, N matrices of size CxW
Input	T<in_t>	A_zp	[1]	1	Input tensor A zero point. Must be zero for non-int8 types.
Input	T<in_t>	B_zp	[1]	1	Input tensor B zero point. Must be zero for non-int8 types.
Output	T<out_t>	output	[N,H,W]	3	Output tensor, N matrices of size HxW

Compile Time Constant Status:

Argument	CTC enabled profile(s)	CTC disabled extension(s)
A_zp	PRO-INT, PRO-FP	EXT-DYNAMIC
B_zp	PRO-INT, PRO-FP	EXT-DYNAMIC

Supported Data Types:

Profile/Extension	Mode	in_t	out_t
PRO-FP	fp16 with fp16 accumulate	fp16_t	fp16_t
PRO-FP	fp16 with fp32 accumulate	fp16_t	fp32_t
PRO-FP	fp32 with fp32 accumulate	fp32_t	fp32_t
PRO-INT	signed 8x8 with int32 accumulate	i8_t	i32_t
EXT-BF16	bf16 with fp32 accumulate	bf16_t	fp32_t
EXT-FP8E4M3	fp8e4m3 with fp16 accumulate	fp8e4m3_t	fp16_t
EXT-FP8E5M2	fp8e5m2 with fp16 accumulate	fp8e5m2_t	fp16_t
EXT-INT16	signed 16x16 with int48 accumulate	i16_t	i48_t

Operation Function:

```
ERROR_IF(is_same<in_t,i8_t> && (A_zp != 0 || B_zp != 0)); // Zero point only for i8_t
for_each(0 <= n < N, 0 <= h < H, 0 <= w < W) {
```

```
    out_t acc = 0;
    for_each(0 <= c < C) {
        out_t value1 = static_cast<out_t>(tensor_read<in_t>(A, [N,H,C], [n,h,c]));
        out_t value2 = static_cast<out_t>(tensor_read<in_t>(B, [N,C,W], [n,c,w]));
        value1 = apply_sub_s<out_t>(value1, static_cast<out_t>(A_zp));
        value2 = apply_sub_s<out_t>(value2, static_cast<out_t>(B_zp));
        acc = apply_add_s<out_t>(acc, apply_mul_s<out_t>(value1 * value2));
    }
    tensor_write<out_t>(output, [N,H,W], [n,h,w], acc);
}
```

2.3.8. 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.

Precision Requirements

Integer results must be exact.

NaN propagation mode only affects floating-point types. It indicates either propagating or ignoring NaN.

The following rules apply to floating-point inputs:

- Comparison rules:
  - The sign of a zero is ignored.
  - Infinities of the same sign compare as equal.
  - In the NaN propagating mode, if any input in the window is a NaN then the result must be NaN.
  - In the NaN ignoring mode, if all inputs in the window are NaN, the result is NaN. Otherwise the result is the maximum non-NaN value in the window.
- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero before calculation.
- If a floating-point result is zero, then the result must be either +0.0 or -0.0 but either sign is permitted.
- If the result is a subnormal value for bf16_t, fp16_t, or fp32_t, the result may be a zero of either sign.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input	[N,IH,IW,C]	4	Input tensor 4D
Attribute	T<i32_t>	kernel	[2]	1	[kernel_y, kernel_x]
Attribute	T<i32_t>	stride	[2]	1	[stride_y, stride_x]

Argument	Type	Name	Shape	Rank	Description
Attribute	T<i32_t>	pad	[4]	1	[pad_top, pad_bottom, pad_left, pad_right]
Attribute	nan_propagation_mode_t	nan_mode	-		PROPAGATE or IGNORE. Set to PROPAGATE by default. This attribute affects the floating-point NaN propagation approach. This attribute is ignored by non floating-point types.
Output	T<in_out_t>	output	[N,OH,OW,C]	4	Output tensor 4D

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-FP	fp16	fp16_t
PRO-FP	fp32	fp32_t
PRO-INT	signed 8	i8_t
EXT-BF16	bf16	bf16_t
EXT-FP8E4M3	fp8e4m3	fp8e4m3_t
EXT-FP8E5M2	fp8e5m2	fp8e5m2_t
EXT-INT16	signed 16	i16_t

Operation Function:

```
LEVEL_CHECK(kernel_y <= MAX_KERNEL);
LEVEL_CHECK(kernel_x <= MAX_KERNEL);
LEVEL_CHECK(stride_y <= MAX_STRIDE);
LEVEL_CHECK(stride_x <= MAX_STRIDE);
LEVEL_CHECK(pad_top <= MAX_KERNEL);
LEVEL_CHECK(pad_bottom <= MAX_KERNEL);
LEVEL_CHECK(pad_left <= MAX_KERNEL);
LEVEL_CHECK(pad_right <= MAX_KERNEL);
```

```
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 < OH, 0 <= ox < OW, 0 <= c < C ) {
    in_out_t acc = (is_floating_point<in_out_t>() && nan_mode == IGNORE)
                       ? nan<in_out_t>()
                       : minimum_s<in_out_t>();
    tensor_size_t iy = oy * stride_y - pad_top;
    tensor_size_t ix = ox * stride_x - pad_left;
    for_each( 0 <= ky < kernel_y, 0 <= kx < kernel_x ) {
        tensor_size_t y = iy + ky;
        tensor_size_t 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_s<in_out_t>(acc, value, nan_mode);
        }
    }
    tensor_write<in_out_t>(output, [N,OH,OW,C], [n,oy,ox,c], acc);
}
```

2.3.9. 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 final output axis. Implementations may choose to skip calculation of the imaginary values at (0,0), (0,W/2), (H/2,0), and (H/2, W/2). If the calculation is skipped, the result at that location must be zero.

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

Precision Requirements

- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero before calculation.
- Each output can be expressed as a dot product of an input vector with a constant coefficient vector.
- The following may be used to validate the output
  - Let `RFFT_Real` be the operation that calculates the real output (`output_real`).
  - Let `RFFT_Imag` be the operation that calculates the imaginary output (`output_imag`).
  - Let `input` be the input tensor.
  - Let `weight_real` be the coefficient vector tensor of real values.
  - Let `weight_imag` be the coefficient vector tensor of imaginary values.
  - For all `S` in the defined data sets in Appendix A Floating-Point Operator Test Data.
    - `tosa_reference_check_dotproduct<RFFT_Real, in_out_t, in_out_t, in_out_t, in_out_t>(S, input, weight_real, [])` must be true.
    - `tosa_reference_check_dotproduct<RFFT_Imag, in_out_t, in_out_t, in_out_t,`

`in_out_t>(S, input, weight_imag, [])` must be true.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input_real	[N,H,W]	3	Real input. H,W must be powers of two.
Attribute	bool_t	local_bound	-		This optional attribute affects the floating-point compliance error bound. The default of false allows for direct and transform based, fast convolution algorithms. Only set to true if direct dot-product calculation precision is required.
Output	T<in_out_t>	output_real	[N,H,W/2 + 1]	3	Real part of the complex output
Output	T<in_out_t>	output_imag	[N,H,W/2 + 1]	3	Imaginary part of the complex output.

Supported Data Types:

Profile/Extension	Mode	in_out_t
EXT-FFT	fp32	fp32_t

Operation Function:

```
LEVEL_CHECK(H <= MAX_KERNEL);
LEVEL_CHECK(W <= MAX_KERNEL);
```

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

for_each(0 <= n < N, 0 <= oy < H, 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]);
        tensor_size_t ay = (static_cast<tensor_size_t>(iy) *
static_cast<tensor_size_t>(oy)) % static_cast<tensor_size_t>(H);
        tensor_size_t ax = (static_cast<tensor_size_t>(ix) *
static_cast<tensor_size_t>(ox)) % static_cast<tensor_size_t>(W);
        in_out_t a = 2 * pi() * (static_cast<in_out_t>(ay) / H +
static_cast<in_out_t>(ax) / W);
        sum_real += val_real * cos(a);
        if ((H > 1 && (ay % (H/2)) > 0) || (W > 1 && (ax % (W/2)) > 0)) {
            sum_imag += -val_real * sin(a);
```

```
        } else if (tosa_extra_multiplies) {
            sum_imag += -val_real * 0.0;
        }
    }
    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);
}
```

2.3.10. TRANSPOSE_CONV2D

Performs a 2D transposed convolution over the given tensor input, using the weights tensor. Implementations may choose to skip calculation of multiplies by zero at fractional input positions.

Precision Requirements

Integer results must be exact.

The following rules apply to floating-point inputs:

- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero before calculation.
- Each output can be expressed as a dot product of two input vectors.
- The dot product must meet the Dot product accuracy requirements.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_t>	input	[N,IH,IW,IC]	4	Input tensor
Input	T<weight_t>	weight	[OC,KH,KW,IC]	4	Weight kernel size KH x KW
Input	T<out_t>	bias	[BC]	1	Per output channel bias data. Bias data will be broadcast if BC == 1.
Input	T<in_t>	input_zp	[1]	1	Input tensor zero point. Must be zero for non-int8 types.
Input	T<weight_t>	weight_zp	[1]	1	Weight zero point. Must be zero for non-int8 types.
Attribute	T<i32_t>	out_pad	[4]	1	[out_pad_top, out_pad_bottom, out_pad_left, out_pad_right]
Attribute	T<i32_t>	stride	[2]	1	[stride_y, stride_x]
Attribute	acc_type_t	acc_type	-		Enumerated type, must be one of INT32, INT48, FP16, FP32 matching the type of acc_t in the Supported Data Types table for this operation

Argument	Type	Name	Shape	Rank	Description
Attribute	bool_t	local_bound	-		This optional attribute affects the floating-point compliance error bound. The default of false allows for direct and transform based, fast convolution algorithms. Only set to true if direct dot-product calculation precision is required.
Output	T<out_t>	output	[N,OH,OW,OC]	4	Output tensor

Compile Time Constant Status:

Argument	CTC enabled profile(s)	CTC disabled extension(s)
input_zp	PRO-INT, PRO-FP	EXT-DYNAMIC
weight_zp	PRO-INT, PRO-FP	EXT-DYNAMIC

Supported Data Types:

Profile/Extension	Mode	in_t	weight_t	out_t	acc_t
PRO-FP	fp16 with fp16 accumulate	fp16_t	fp16_t	fp16_t	fp16_t
PRO-FP	fp16 with fp32 accumulate	fp16_t	fp16_t	fp16_t	fp32_t
PRO-FP	fp32 with fp32 accumulate	fp32_t	fp32_t	fp32_t	fp32_t
PRO-INT	signed 8x8 with int32 accumulate	i8_t	i8_t	i32_t	i32_t
EXT-BF16	bf16 with fp32 accumulate	bf16_t	bf16_t	bf16_t	fp32_t
EXT-FP8E4M3	fp8e4m3 with fp16 accumulate	fp8e4m3_t	fp8e4m3_t	fp16_t	fp16_t
EXT-FP8E5M2	fp8e5m2 with fp16 accumulate	fp8e5m2_t	fp8e5m2_t	fp16_t	fp16_t
EXT-INT16	signed 16x8 with int48 accumulate	i16_t	i8_t	i48_t	i48_t

Profile/Extension	Mode	in_t	weight_t	out_t	acc_t
EXT-INT4	signed 8x4 with int32 accumulate	i8_t	i4_t	i32_t	i32_t

Operation Function:

```
LEVEL_CHECK(KH <= MAX_KERNEL);
LEVEL_CHECK(KW <= MAX_KERNEL);
LEVEL_CHECK(out_pad_top <= MAX_KERNEL);
LEVEL_CHECK(out_pad_bottom <= MAX_KERNEL);
LEVEL_CHECK(out_pad_left <= MAX_KERNEL);
LEVEL_CHECK(out_pad_right <= MAX_KERNEL);
LEVEL_CHECK(stride_y <= MAX_STRIDE);
LEVEL_CHECK(stride_x <= MAX_STRIDE);
```

```
ERROR_IF(!is_same<in_t,i8_t>() && input_zp != 0); // Zero point only allowed for
int8_t
ERROR_IF(!is_same<weight_t,i8_t>() && weight_zp != 0);
ERROR_IF(out_pad_top <= -KH || out_pad_bottom <= -KH);
ERROR_IF(out_pad_left <= -KW || out_pad_right <= -KW);
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);
ERROR_IF(BC != OC && BC != 1);

for_each(0 <= n < N, 0 <= oy < OH, 0 <= ox < OW, 0 <= oc < OC) {
    acc_t acc = 0;
    tensor_size_t iy = oy - out_pad_top;
    tensor_size_t ix = ox - out_pad_left;
    for_each(0 <= ky < KH, 0 <= kx < KW, 0 <= ic < IC) {
        tensor_size_t y = iy - ky;
        tensor_size_t x = ix - kx;
        acc_t value = 0;
        if (0 <= y < IH * stride_y && 0 <= x < IW * stride_x && (y % stride_y)==0 &&
(x % stride_x)==0) {
            value  = static_cast<acc_t>(tensor_read<in_t>(input,
                                                          [N,IH,IW,IC],

[n,y/stride_y,x/stride_x,ic]));
            value  = apply_sub_s<acc_t>(value, static_cast<acc_t>(input_zp));
        }
        if ((0 <= y < IH * stride_y && 0 <= x < IW * stride_x && (y % stride_y)==0 &&
(x % stride_x)==0) || tosa_extra_multiplies) {
            acc_t weight_el = static_cast<acc_t>(tensor_read<weight_t>(weight,
                                                                       [OC,KH,KW,IC],
                                                                       [oc,ky,kx,ic]));
```

```
            weight_el = apply_sub_s<acc_t>(weight_el, static_cast<acc_t>(weight_zp));
            acc = apply_add_s<acc_t>(acc, apply_mul_s<acc_t>(value, weight_el));
        }
    }
    out_t out = static_cast<out_t>(acc);
    out = apply_add_s<out_t>(out, bias[(BC == 1) ? 0 : oc]);
    tensor_write<out_t>(output, [N,OH,OW,OC], [n,oy,ox,oc], out);
}
```

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.

Precision Requirements

Integer results must be exact.

NaN propagation mode only affects floating-point types. It indicates either propagating or ignoring NaN.

The following rules apply to floating-point inputs:

- Comparison rules:
  - The sign of a zero is ignored.
  - Infinities of the same sign compare as equal.
  - In the NaN propagating mode, if the input is a NaN, the output must be a NaN.
  - In the NaN ignoring mode, if the input is a NaN, the output is the specified minimum value.
- bf16_t, fp16_t, and fp32_t subnormal values may be flushed to zero before computation.
- If a floating-point result is zero, then the result must be either +0.0 or -0.0 but either sign is permitted.
- If the result is a subnormal value for bf16_t, fp16_t, or fp32_t, the result may be a zero of either sign.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input	shape	0 to MAX_RANK	Input tensor
Attribute	in_out_t	min_val	-		Minimum clip value

Argument	Type	Name	Shape	Rank	Description
Attribute	in_out_t	max_val	-		Maximum clip value
Attribute	nan_propagation_mode_t	nan_mode	-		PROPAGATE or IGNORE. Set to PROPAGATE by default. This attribute affects the floating-point NaN propagation approach. This attribute is ignored by non floating-point types.
Output	T<in_out_t>	output	shape	0 to MAX_RANK	Output tensor of same type and shape as input

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-FP	fp16	fp16_t
PRO-FP	fp32	fp32_t
PRO-INT	signed 8	i8_t
EXT-BF16	bf16	bf16_t
EXT-INT16	signed 16	i16_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

```
ERROR_IF(max_val < min_val);
ERROR_IF(isNaN(min_val) || isNaN(max_val));
for_each_data_position(index in shape) {
    in_out_t value = tensor_read<in_out_t>(input, shape, index);
    value = apply_clip_s<in_out_t>(value, min_val, max_val, nan_mode);
    tensor_write<in_out_t>(output, shape, index, value);
}
```

2.4.2. ERF

Error function:

$$erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

Figure 3. Calculation for the error function

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

The ERF table has 513 entries each of 16-bit precision and covering the input range -4.0 to +4.0 in steps of 1/64.

```
int16_t erf_reference(int16_t x) { // input x range is -256 to + 256 inclusive
    fp64_t v = static_cast<fp64_t>(x) / static_cast<fp64_t>(64);
    v = erf(v);
    return round_to_nearest_int(32768.0 * v);
}

generate_lookup_table(&erf_table, &erf_reference);
```

Precision Requirements

- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero before calculation.
- The following may be used to validate the result:
  - Let `out_imp` be the implementation output.
  - Let `out_ref` be the result calculated using fp64_t arithmetic.
  - Then `tosa_reference_check_fp<in_t>(out_imp, out_ref, 5)` must be true.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input	shape	0 to MAX_RANK	Input tensor
Output	T<in_out_t>	output	shape	0 to MAX_RANK	Output tensor of same type and shape as input

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-FP	fp16	fp16_t
PRO-FP	fp32	fp32_t
EXT-BF16	bf16	bf16_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

```
for_each_data_position(index in shape) {
    in_out_t value1 = tensor_read<in_out_t>(input, shape, index);
    in_out_t value = erf<in_out_t>(value1);
    tensor_write<in_out_t>(output, shape, index, value);
}
```

2.4.3. SIGMOID

Applies the sigmoid logistic function to each element of the input tensor.

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$

Figure 4. Calculation for the sigmoid function

For quantized integer data types, the TABLE operator should be used instead. Each implementation may choose an appropriate TABLE given the scale and zero point of the input data. Eight or sixteen bit precision tables may be used based on the input tensor to the sigmoid function. Below we give an example table generation for 16-bit sigmoid. This 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.

Code for generating 16-bit sigmoid table

```
int16_t sigmoid_reference(int16_t x) { // input x range is -256 to + 256 inclusive
    fp64_t v = static_cast<fp64_t>(x) / static_cast<fp64_t>(16);
    v = 1.0/(1.0 + exp(-v));
    return round_to_nearest_int(32768.0 * v);
}

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

Precision Requirements

- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero before calculation.
- Infinity, NaN, and Zero behavior as defined in the following table.
- Otherwise the following may be used to validate the result:
  - Let `x` be an input element.
  - Let `out_imp` be the implementation output.
  - Let `out_ref` be the result calculated using fp64_t arithmetic.
  - Let `err_bnd = calcAbsErrorBound<in_out_t>(out_ref, 2 * (1+abs(x)), 0, 1)`.
  - Then `tosa_reference_check_fp_bnd<in_out_t>(out_imp, out_ref, err_bnd)` must be true.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input	shape	0 to MAX_RANK	Input tensor
Output	T<in_out_t>	output	shape	0 to MAX_RANK	Output tensor of same type and shape as input

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-FP	fp16	fp16_t
PRO-FP	fp32	fp32_t
EXT-BF16	bf16	bf16_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

Floating-point behavior:

Input	-infinity	+infinity	-0	+0	NaN
Output	0	1	0.5	0.5	any NaN

```
for_each_data_position(index in shape) {
    in_out_t value1 = tensor_read<in_out_t>(input, shape, index);
    in_out_t value = sigmoid<in_out_t>(value1);
    tensor_write<in_out_t>(output, shape, index, value);
}
```

2.4.4. TANH

Parameterized hyperbolic tangent.

$$tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

Figure 5. Calculation for the tanh function

For quantized integer data types, the TABLE operator should be used instead. Each implementation may choose an appropriate TABLE given the scale and zero point of the input data. Eight or sixteen bit precision tables may be used based on the input tensor to the tanh function. Below we give an example table generation for 16-bit hyperbolic tangent. This 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.

Calculation of an example 16-bit tanh table

```
int16_t tanh_reference(int16_t x) {  // input x range is -256 to +256 inclusive
    fp64_t v = static_cast<fp64_t>(x) / static_cast<fp64_t>(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);
```

Precision Requirements

- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero before calculation.
- Infinity, NaN, and Zero behavior as defined in the following table.
- Otherwise the following may be used to validate the result:
  - Let `x` be an input element.
  - Let `out_imp` be the implementation output.
  - Let `out_ref` be the result calculated using fp64_t arithmetic.
    - Note (informational): The error bound is derived from the error bound for the exp operator based on implementation of TANH as `(exp(x) - exp(-x)) / (exp(x) + exp(-x))`.
    - The absolute lower bound of 0.5 ulp relative to 1.0 is due to `exp(x) - exp(-x)` having an error of this magnitude when x is close to 0.
  - Let `exp_err_base = is_same<in_out_t, fp32_t>() ? 3 : 1`.
  - Let `err_bnd = calcAbsErrorBound<in_out_t>(out_ref, 4 * (exp_err_base + 2 * abs(x)), 0.5, 1)`.
  - Then `tosa_reference_check_fp_bnd<in_out_t>(out_imp, out_ref, err_bnd)` must be true.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input	shape	0 to MAX_RANK	Input tensor
Output	T<in_out_t>	output	shape	0 to MAX_RANK	Output tensor of same type and shape as input

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-FP	fp16	fp16_t
PRO-FP	fp32	fp32_t
EXT-BF16	bf16	bf16_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

Floating-point behavior:

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

```
for_each_data_position(index in shape) {
```

```
    in_out_t value1 = tensor_read<in_out_t>(input, shape, index);
    in_out_t value = tanh<in_out_t>(value1);
    tensor_write<in_out_t>(output, shape, index, value);
}
```

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.

Precision Requirements

Integer results must be exact.

The following rules apply to floating-point inputs:

- If any input is a NaN, the result must be a NaN.
- Addition of infinities of different signs must produce a NaN.
- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero before calculation.
- Subnormal bf16_t, fp16_t, and fp32_t result values may be flushed to zero of the appropriate sign after the calculation.
- The following may be used to validate the result:
  - Let `out_imp` be the implementation output.
  - Let `out_ref` be the result calculated using fp64_t arithmetic.
  - Then `tosa_reference_check_fp<in_t>(out_imp, out_ref, 0.5)` must be true.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input1	shape1	0 to MAX_RANK	Input tensor
Input	T<in_out_t>	input2	shape2	0 to MAX_RANK	Input tensor with the same rank as input1
Output	T<in_out_t>	output	shape	0 to MAX_RANK	Output tensor

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-FP	fp16	fp16_t
PRO-FP	fp32	fp32_t
PRO-INT or PRO-FP	signed 32	i32_t
EXT-BF16	bf16	bf16_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

```
ERROR_IF(shape != broadcast_shape(shape1, shape2));
for_each_data_position(index in shape) {
    shape_t index1 = apply_broadcast(shape, shape1, index);
    shape_t 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_s<in_out_t>(value1, value2);
    tensor_write<in_out_t>(output, shape, index, result);
}
```

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.

Precision Requirements

Results must be exact.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input1	shape1	0 to MAX_RANK	Input tensor
Input	T<in_out_t>	input2	shape2	0 to MAX_RANK	Input tensor with the same rank as input1
Attribute	bool_t	round	-		If true then the shift is rounded
Output	T<in_out_t>	output	shape	0 to MAX_RANK	Output tensor

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-INT	signed 16	i16_t
PRO-INT	signed 32	i32_t
PRO-INT	signed 8	i8_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

```
ERROR_IF(shape != broadcast_shape(shape1, shape2));
for_each_data_position(index in shape) {
    shape_t index1 = apply_broadcast(shape, shape1, index);
    shape_t 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((is_same<in_out_t,i32_t>() && 0 <= value2 && value2 <= 31) ||
            (is_same<in_out_t,i16_t>() && 0 <= value2 && value2 <= 15) ||
            (is_same<in_out_t,i8_t>() && 0 <= value2 && value2 <= 7));

    in_out_t result = apply_arith_rshift<in_out_t>(value1, value2);
    if (round == true && static_cast<int32_t>(value2) > 0 &&
        (apply_arith_rshift<in_out_t>(value1, apply_sub_s<in_out_t>(value2, 1)) & 1 !=
0)) {
        result = result + 1;
    }
    tensor_write<in_out_t>(output, shape, index, result);
}
```

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.

Precision Requirements

Results must be exact.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input1	shape1	0 to MAX_RANK	Input tensor
Input	T<in_out_t>	input2	shape2	0 to MAX_RANK	Input tensor with the same rank as input1
Output	T<in_out_t>	output	shape	0 to MAX_RANK	Output tensor

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-INT	signed 16	i16_t
PRO-INT	signed 32	i32_t
PRO-INT	signed 8	i8_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

```
ERROR_IF(shape != broadcast_shape(shape1, shape2));
for_each_data_position(index in shape) {
    shape_t index1 = apply_broadcast(shape, shape1, index);
    shape_t 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);
}
```

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.

Precision Requirements

Results must be exact.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input1	shape1	0 to MAX_RANK	Input tensor
Input	T<in_out_t>	input2	shape2	0 to MAX_RANK	Input tensor with the same rank as input1
Output	T<in_out_t>	output	shape	0 to MAX_RANK	Output tensor

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-INT	signed 16	i16_t
PRO-INT	signed 32	i32_t
PRO-INT	signed 8	i8_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

```
ERROR_IF(shape != broadcast_shape(shape1, shape2));
for_each_data_position(index in shape) {
    shape_t index1 = apply_broadcast(shape, shape1, index);
```

```
    shape_t 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);
}
```

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.

Precision Requirements

Results must be exact.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input1	shape1	0 to MAX_RANK	Input tensor
Input	T<in_out_t>	input2	shape2	0 to MAX_RANK	Input tensor with the same rank as input1
Output	T<in_out_t>	output	shape	0 to MAX_RANK	Output tensor

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-INT	signed 16	i16_t
PRO-INT	signed 32	i32_t
PRO-INT	signed 8	i8_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

```
ERROR_IF(shape != broadcast_shape(shape1, shape2));
for_each_data_position(index in shape) {
    shape_t index1 = apply_broadcast(shape, shape1, index);
    shape_t 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);
}
```

2.5.6. INTDIV

Elementwise integer divide of input1 by input2. Axis of size 1 will be broadcast as necessary. Rank of input tensors must match. 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.

Precision Requirements

Results must be exact.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input1	shape1	0 to MAX_RANK	Input tensor
Input	T<in_out_t>	input2	shape2	0 to MAX_RANK	Input tensor with the same rank as input1
Output	T<in_out_t>	output	shape	0 to MAX_RANK	Output tensor

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-INT or PRO-FP	signed 32	i32_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

```
ERROR_IF(shape != broadcast_shape(shape1, shape2));
for_each_data_position(index in shape) {
    shape_t index1 = apply_broadcast(shape, shape1, index);
    shape_t 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);
    in_out_t result = apply_intdiv_s<in_out_t>(value1, value2);
    tensor_write<in_out_t>(output, shape, index, result);
}
```

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.

Precision Requirements

Results must be exact.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input1	shape1	0 to MAX_RANK	Input tensor
Input	T<in_out_t>	input2	shape2	0 to MAX_RANK	Input tensor with the same rank as input1
Output	T<in_out_t>	output	shape	0 to MAX_RANK	Output tensor

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-INT or PRO-FP	Boolean	bool_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

```
ERROR_IF(shape != broadcast_shape(shape1, shape2));
for_each_data_position(index in shape) {
    shape_t index1 = apply_broadcast(shape, shape1, index);
    shape_t 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);
}
```

2.5.8. LOGICAL_LEFT_SHIFT

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

Precision Requirements

Results must be exact.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input1	shape1	0 to MAX_RANK	Input tensor
Input	T<in_out_t>	input2	shape2	0 to MAX_RANK	Input tensor with the same rank as input1

Argument	Type	Name	Shape	Rank	Description
Output	T<in_out_t>	output	shape	0 to MAX_RANK	Output tensor

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-INT or PRO-FP	signed 16	i16_t
PRO-INT or PRO-FP	signed 32	i32_t
PRO-INT or PRO-FP	signed 8	i8_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

```
ERROR_IF(shape != broadcast_shape(shape1, shape2));
for_each_data_position(index in shape) {
    shape_t index1 = apply_broadcast(shape, shape1, index);
    shape_t 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((is_same<in_out_t,i32_t>() && 0 <= value2 && value2 <= 31) ||
            (is_same<in_out_t,i16_t>() && 0 <= value2 && value2 <= 15) ||
            (is_same<in_out_t,i8_t>() && 0 <= value2 && value2 <= 7));
    in_out_t result = value1 << value2;
    tensor_write<in_out_t>(output, shape, index, result);
}
```

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.

Precision Requirements

Results must be exact.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input1	shape1	0 to MAX_RANK	Input tensor
Input	T<in_out_t>	input2	shape2	0 to MAX_RANK	Input tensor with the same rank as input1
Output	T<in_out_t>	output	shape	0 to MAX_RANK	Output tensor

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-INT or PRO-FP	signed 16	i16_t
PRO-INT or PRO-FP	signed 32	i32_t
PRO-INT or PRO-FP	signed 8	i8_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

```
ERROR_IF(shape != broadcast_shape(shape1, shape2));
for_each_data_position(index in shape) {
    shape_t index1 = apply_broadcast(shape, shape1, index);
    shape_t 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((is_same<in_out_t,i32_t>() && 0 <= value2 && value2 <= 31) ||
            (is_same<in_out_t,i16_t>() && 0 <= value2 && value2 <= 15) ||
            (is_same<in_out_t,i8_t>() && 0 <= value2 && value2 <= 7));

    // Logical shifts happen as unsigned types internally
    in_out_t result = apply_logical_rshift<in_out_t>(value1, value2);
    tensor_write<in_out_t>(output, shape, index, result);
}
```

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.

Precision Requirements

Results must be exact.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input1	shape1	0 to MAX_RANK	Input tensor
Input	T<in_out_t>	input2	shape2	0 to MAX_RANK	Input tensor with the same rank as input1
Output	T<in_out_t>	output	shape	0 to MAX_RANK	Output tensor

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-INT or PRO-FP	Boolean	bool_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

```
ERROR_IF(shape != broadcast_shape(shape1, shape2));
for_each_data_position(index in shape) {
    shape_t index1 = apply_broadcast(shape, shape1, index);
    shape_t 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);
}
```

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.

Precision Requirements

Results must be exact.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input1	shape1	0 to MAX_RANK	Input tensor
Input	T<in_out_t>	input2	shape2	0 to MAX_RANK	Input tensor with the same rank as input1
Output	T<in_out_t>	output	shape	0 to MAX_RANK	Output tensor

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-INT or PRO-FP	Boolean	bool_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

```
ERROR_IF(shape != broadcast_shape(shape1, shape2));
for_each_data_position(index in shape) {
    shape_t index1 = apply_broadcast(shape, shape1, index);
    shape_t 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);
}
```

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.

Precision Requirements

Integer results must be exact.

NaN propagation mode only affects floating-point types. It indicates either propagating or ignoring NaN.

The following rules apply to floating-point inputs:

- Comparison rules:
  - The sign of a zero is ignored.
  - Infinities of the same sign compare as equal.
  - In the NaN propagating mode, if either input value is a NaN, the result is NaN.
  - In the NaN ignoring mode, if either input value is a NaN, the result is the non-NaN element.
  - If both values are NaN, the result is NaN.
- bf16_t, fp16_t, and fp32_t subnormal values may be flushed to zero before computation.
- If a floating-point result is zero, then the result must be either +0.0 or -0.0 but either sign is permitted.
- If the result is a subnormal value for bf16_t, fp16_t, or fp32_t, the result may be a zero of either sign.
- If none of the above conditions apply, the floating-point result must be exact.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input1	shape1	0 to MAX_RANK	Input tensor
Input	T<in_out_t>	input2	shape2	0 to MAX_RANK	Input tensor with the same rank as input1

Argument	Type	Name	Shape	Rank	Description
Attribute	nan_propagation_mode_t	nan_mode	-		PROPAGATE or IGNORE. Set to PROPAGATE by default. This attribute affects the floating-point NaN propagation approach. This attribute is ignored by non floating-point types.
Output	T<in_out_t>	output	shape	0 to MAX_RANK	Output tensor

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-FP	fp16	fp16_t
PRO-FP	fp32	fp32_t
PRO-INT	signed 32	i32_t
EXT-BF16	bf16	bf16_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

```
ERROR_IF(shape != broadcast_shape(shape1, shape2));
for_each_data_position(index in shape) {
    shape_t index1 = apply_broadcast(shape, shape1, index);
    shape_t 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_s<in_out_t>(value1, value2, nan_mode);
    tensor_write<in_out_t>(output, shape, index, result);
}
```

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.

Precision Requirements

Integer results must be exact.

NaN propagation mode only affects floating-point types. It indicates either propagating or ignoring NaN.

The following rules apply to floating-point inputs:

- Comparison rules:
  - The sign of a zero is ignored.
  - Infinities of the same sign compare as equal.
  - In the NaN propagating mode, if either input value is a NaN, the result is NaN.
  - In the NaN ignoring mode, if either input value is a NaN, the result is the non-NaN element.
  - If both values are NaN, the result is NaN.
- bf16_t, fp16_t, and fp32_t subnormal values may be flushed to zero before computation.
- If a floating-point result is zero, then the result must be either +0.0 or -0.0 but either sign is permitted.
- If the result is a subnormal value for bf16_t, fp16_t, or fp32_t, the result may be a zero of either sign.
- If none of the above conditions apply, the floating-point result must be exact.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input1	shape1	0 to MAX_RANK	Input tensor
Input	T<in_out_t>	input2	shape2	0 to MAX_RANK	Input tensor with the same rank as input1
Attribute	nan_propagation_mode_t	nan_mode	-		PROPAGATE or IGNORE. Set to PROPAGATE by default. This attribute affects the floating-point NaN propagation approach. This attribute is ignored by non floating-point types.
Output	T<in_out_t>	output	shape	0 to MAX_RANK	Output tensor

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-FP	fp16	fp16_t
PRO-FP	fp32	fp32_t
PRO-INT	signed 32	i32_t
EXT-BF16	bf16	bf16_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

```
ERROR_IF(shape != broadcast_shape(shape1, shape2));
```

```
for_each_data_position(index in shape) {
    shape_t index1 = apply_broadcast(shape, shape1, index);
    shape_t 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_s(value1, value2, nan_mode);
    tensor_write<in_out_t>(output, shape, index, result);
}
```

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.

Precision Requirements

Integer results must be exact.

The following rules apply to floating-point inputs:

- If any input is a NaN, the result must be a NaN.
- Multiplication of an infinity by a zero must produce a NaN.
- Multiplication of two infinities must produce an infinity of the correct sign.
- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero before calculation.
- Subnormal bf16_t, fp16_t, and fp32_t result values may be flushed to zero of the appropriate sign after the calculation.
- The following may be used to validate the result:
  - Let `out_imp` be the implementation output.
  - Let `out_ref` be the result calculated using fp64_t arithmetic.
  - Then `tosa_reference_check_fp<in_t>(out_imp, out_ref, 0.5)` must be true.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_t>	input1	shape1	0 to MAX_RANK	Input tensor
Input	T<in_t>	input2	shape2	0 to MAX_RANK	Input tensor with the same rank as input1
Input	T<i8_t>	shift	[1]	1	Result right shift (used only when in_t is i32_t)
Output	T<out_t>	output	shape	0 to MAX_RANK	Output tensor

Compile Time Constant Status:

Argument	CTC enabled profile(s)	CTC disabled extension(s)
shift	PRO-INT, PRO-FP	EXT-DYNAMIC

Supported Data Types:

Profile/Extension	Mode	in_t	out_t
PRO-FP	fp16	fp16_t	fp16_t
PRO-FP	fp32	fp32_t	fp32_t
PRO-INT	signed 16	i16_t	i32_t
PRO-INT or PRO-FP	signed 32	i32_t	i32_t
PRO-INT	signed 8	i8_t	i32_t
EXT-BF16	bf16	bf16_t	bf16_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

```
REQUIRE(0 <= shift && shift <= 63);
REQUIRE(is_same<in_t,int32_t>() || shift == 0);
ERROR_IF(shape != broadcast_shape(shape1, shape2));
for_each_data_position(index in shape) {
    shape_t index1 = apply_broadcast(shape, shape1, index);
    shape_t 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 (is_same<in_t,i32_t>() && shift > 0) {
        int64_t product = sign_extend<int64_t>(value1) * sign_extend<int64_t>(value2);
        int64_t round   = static_cast<int64_t>(1) << (shift - 1);
        product = (product + round) >> shift;
        REQUIRE(product >= minimum_s<i32_t>() && product <= maximum_s<i32_t>());
        result = static_cast<out_t>(product);
    } else {
        result = apply_mul_s<out_t>(static_cast<out_t>(value1),
static_cast<out_t>(value2));  // low 32-bits of result for i32_t
    }
    tensor_write<out_t>(output, shape, index, result);
}
```

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.

Precision Requirements

- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero before calculation.
- Otherwise the following may be used to validate the result.
  - Let `x`, `y` be input elements from `input1` and `input2` respectively.
  - Let `out_imp` be the implementation output.
  - If x or y is an infinity, the result is undefined.
  - If x or y is a NaN, the result is undefined.
  - If x < 0, the result is undefined.
  - If x == 0 and y ⇐ 0, the result is undefined.
  - If x == 0 and y > 0 then the result is 0.
  - If x > 0 and y == 0 then the result is 1.
  - Otherwise:
    - Let `out_ref` be the result calculated using `fp64_t` arithmetic
    - Note (informational): the error bound is derived from the error bound for the `EXP`, `MUL` and `LOG` operators to allow implementation of `POW` as `exp(y * log(x))` for positive values of `x`.
    - Let `err_base = is_same<T, fp32_t> ? 3 : 1`.
    - Let `mul_err = calcAbsErrorBound<in_out_t>(y * log(x), 5.5, abs(0.55 * y), 1)`.
    - Let `exp_err = calcAbsErrorBound<in_out_t>(out_ref, err_base + 2 * abs(y * log(x) + mul_err), 0, 1)`.
    - Let `err_bnd = out_ref * expm1(mul_err) + exp_err`.
    - Then `tosa_reference_check_fp_bnd<in_out_t>(out_imp, out_ref, err_bnd)` must be true.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input1	shape1	0 to MAX_RANK	Input tensor
Input	T<in_out_t>	input2	shape2	0 to MAX_RANK	Input tensor with the same rank as input1
Output	T<in_out_t>	output	shape	0 to MAX_RANK	Output tensor

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-FP	fp16	fp16_t
PRO-FP	fp32	fp32_t
EXT-BF16	bf16	bf16_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

```
ERROR_IF(shape != broadcast_shape(shape1, shape2));

for_each_data_position(index in shape) {
    shape_t index1 = apply_broadcast(shape, shape1, index);
    shape_t 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(value1 >= 0);
    REQUIRE(value1 > 0 || value2 > 0);
    REQUIRE(!isNaN(value1) && !isNaN(value2));
    REQUIRE(is_finite(value1) && is_finite(value2));
    in_out_t result = apply_pow<in_out_t>(value1, value2);
    tensor_write<in_out_t>(output, shape, index, result);
}
```

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.

Precision Requirements

Integer results must be exact.

The following rules apply to floating-point inputs:

- If any input is a NaN, the result must be a NaN.
- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero before calculation.
- Subnormal bf16_t, fp16_t, and fp32_t result values may be flushed to zero of the appropriate sign after the calculation.
- The following may be used to validate the result:
  - Let `out_imp` the implementation output.
  - Let `out_ref` be the result calculated using fp64_t arithmetic.
  - Then `tosa_reference_check_fp<in_t>(out_imp, out_ref, 0.5)` must be true.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input1	shape1	0 to MAX_RANK	Input tensor

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input2	shape2	0 to MAX_RANK	Input tensor with the same rank as input1
Output	T<in_out_t>	output	shape	0 to MAX_RANK	Output tensor

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-FP	fp16	fp16_t
PRO-FP	fp32	fp32_t
PRO-INT or PRO-FP	signed 32	i32_t
EXT-BF16	bf16	bf16_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

```
ERROR_IF(shape != broadcast_shape(shape1, shape2));
for_each_data_position(index in shape) {
    shape_t index1 = apply_broadcast(shape, shape1, index);
    shape_t 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_s<in_out_t>(value1, value2);
    tensor_write<in_out_t>(output, shape, index, result);
}
```

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)

Precision Requirements

Results must be exact.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_t>	input1	shape	0 to MAX_RANK	Input tensor
Input	T<table_t>	table	[TABLE_SIZE]	1	Lookup table tensor
Output	T<out_t>	output	shape	0 to MAX_RANK	Output tensor

Compile Time Constant Status:

Argument	CTC enabled profile(s)	CTC disabled extension(s)
table	PRO-INT	EXT-DYNAMIC

Supported Data Types:

Profile/Extension	Mode	in_t	table_t	out_t	TABLE_SIZE
PRO-INT	signed 8	i8_t	i8_t	i8_t	256
EXT-INT16	signed 16	i16_t	i16_t	i32_t	513

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

```
REQUIRE(length(table) == TABLE_SIZE);
for_each_data_position(index in shape) {
    in_t value = tensor_read<in_t>(input1, shape, index);
    out_t result;
    if (is_same<in_t,i8_t>()) {
        // value is a signed int, convert to a 0 based index
        result = table[static_cast<int16_t>(value) + 128];
    } else {
        result = apply_lookup_s(static_cast<int16_t>(table),
static_cast<int16_t>(value));
    }
    tensor_write<out_t>(output, shape, index, result);
}
```

2.6. Elementwise Unary Operators

2.6.1. ABS

Elementwise absolute value operation.

Precision Requirements

Integer results must be exact.

For floating-point values, in addition to the Floating-point behavior table, the following rules apply:

- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero before calculation.
- Otherwise floating-point results must be exact.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input1	shape	0 to MAX_RANK	Input tensor
Output	T<in_out_t>	output	shape	0 to MAX_RANK	Output tensor of same type, size as the input tensor

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-FP	fp16	fp16_t
PRO-FP	fp32	fp32_t
PRO-INT	signed 32	i32_t
EXT-BF16	bf16	bf16_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

Floating-point behavior:

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

```
for_each_data_position(index in shape) {
    in_out_t value1 = tensor_read<in_out_t>(input1, shape, index);
    if (is_floating_point<in_out_t>() && value1 == -0.0) {
        value1 = 0.0;
    }
    if (static_cast<int32_t>(value1) < 0.0) {
        value1 = apply_sub_s<in_out_t>(0, value1);
    }
```

```
    tensor_write<in_out_t>(output, shape, index, value1);
}
```

2.6.2. BITWISE_NOT

Elementwise bitwise NOT of input tensor.

Precision Requirements

Results must be exact.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input1	shape	0 to MAX_RANK	Input tensor
Output	T<in_out_t>	output	shape	0 to MAX_RANK	Output tensor of same type, size as the input tensor

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-INT	signed 16	i16_t
PRO-INT	signed 32	i32_t
PRO-INT	signed 8	i8_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

```
for_each_data_position(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);
}
```

2.6.3. CEIL

Elementwise ceiling operation

Precision Requirements

- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero before calculation.
- Infinity, NaN, and Zero behavior as defined in the following table.
- The following may be used to validate the result:

- Let `out_imp` the implementation output.
- Let `out_ref` be the result calculated using fp64_t arithmetic.
- Then `tosa_reference_check_fp<in_t>(out_imp, out_ref, 0.5)` must be true.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input1	shape	0 to MAX_RANK	Input tensor
Output	T<in_out_t>	output	shape	0 to MAX_RANK	Output tensor of same type, size as the input tensor

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-FP	fp16	fp16_t
PRO-FP	fp32	fp32_t
EXT-BF16	bf16	bf16_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

Floating-point behavior:

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

```
for_each_data_position(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);
}
```

2.6.4. CLZ

Elementwise count leading zeros operation

Precision Requirements

Results must be exact.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input1	shape	0 to MAX_RANK	Input tensor
Output	T<in_out_t>	output	shape	0 to MAX_RANK	Output tensor of same type, size as the input tensor

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-INT	signed 32	i32_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

```
for_each_data_position(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);
}
```

2.6.5. COS

Elementwise cosine operation for values given in radians.

Precision Requirements

- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero before calculation.
- Infinity, NaN, and Zero behavior as defined in the following table.
- The following may be used to validate the result:
  - Let `x` be an input element.
  - Let `out_imp` the implementation output.
  - Let `out_ref` be the result calculated using fp64_t arithmetic.
  - Let `err_bnd = calcAbsErrorBound<in_out_t>(1+abs(x), 1, 0, 2)`.
  - Then `tosa_reference_check_fp_bnd<in_out_t>(out_imp, out_ref, err_bnd)` must be true.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input1	shape	0 to MAX_RANK	Input tensor
Output	T<in_out_t>	output	shape	0 to MAX_RANK	Output tensor of same type and shape as input

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-FP	fp16	fp16_t
PRO-FP	fp32	fp32_t
EXT-BF16	bf16	bf16_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

Floating-point behavior:

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

```
for_each_data_position(index in shape) {
    in_out_t value1 = tensor_read<in_out_t>(input1, shape, index);
    in_out_t value = cos<in_out_t>(value1);
    tensor_write<in_out_t>(output, shape, index, value);
}
```

2.6.6. EXP

Elementwise e to the x operation

Precision Requirements

- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero before calculation.
- Infinity, NaN, and Zero behavior as defined in the following table.
- The following may be used to validate the result:
  - Let `x` be an input element.
  - Let `out_imp` the implementation output.
  - Let `out_ref` be the result calculated using fp64_t arithmetic.
  - Let `err_base` be `is_same<in_out_t, fp32_t> ? 3 : 1`.
  - Let `err_bnd = calcAbsErrorBound<in_out_t>(out_ref, (err_base+2*abs(x)), 0, 1)`.
  - Then `tosa_reference_check_fp_bnd<in_out_t>(out_imp, out_ref, err_bnd)` must be true.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input1	shape	0 to MAX_RANK	Input tensor

Argument	Type	Name	Shape	Rank	Description
Output	T<in_out_t>	output	shape	0 to MAX_RANK	Output tensor of same type, size as the input tensor

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-FP	fp16	fp16_t
PRO-FP	fp32	fp32_t
EXT-BF16	bf16	bf16_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

Floating-point behavior:

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

```
for_each_data_position(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);
}
```

2.6.7. FLOOR

Elementwise floor operation.

Precision Requirements

- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero before calculation.
- Infinity, NaN, and Zero behavior as defined in the following table.
- The following may be used to validate the result:
  - Let `out_imp` the implementation output.
  - Let `out_ref` be the result calculated using fp64_t arithmetic.
  - Then `tosa_reference_check_fp<in_t>(out_imp, out_ref, 0.5)` must be true.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input1	shape	0 to MAX_RANK	Input tensor
Output	T<in_out_t>	output	shape	0 to MAX_RANK	Output tensor of same type, size as the input tensor

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-FP	fp16	fp16_t
PRO-FP	fp32	fp32_t
EXT-BF16	bf16	bf16_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

Floating-point behavior:

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

```
for_each_data_position(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);
}
```

2.6.8. LOG

Elementwise natural logarithm operation

Precision Requirements

- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero before calculation.
- Infinity, NaN, and Zero behavior as defined in the following table.
- If the input to LOG is less than zero, then the result must be a NaN.
- The following may be used to validate the result:
  - Let `out_imp` the implementation output.
  - Let `out_ref` be the result calculated using fp64_t arithmetic.
  - Let `err_bnd = calcAbsErrorBound<in_out_t>(out_ref, 5, 0.5, 1)`.
  - Then `tosa_reference_check_fp_bnd<in_out_t>(out_imp, out_ref, err_bnd)` must be true.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input1	shape	0 to MAX_RANK	Input tensor
Output	T<in_out_t>	output	shape	0 to MAX_RANK	Output tensor of same type, size as the input tensor

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-FP	fp16	fp16_t
PRO-FP	fp32	fp32_t
EXT-BF16	bf16	bf16_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

Floating-point behavior:

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

```
for_each_data_position(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);
}
```

2.6.9. LOGICAL_NOT

Elementwise logical NOT of input.

Precision Requirements

Results must be exact.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input1	shape	0 to MAX_RANK	Input tensor
Output	T<in_out_t>	output	shape	0 to MAX_RANK	Output tensor of same type, size as the input tensor

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-INT or PRO-FP	Boolean	bool_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

```
for_each_data_position(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);
}
```

2.6.10. NEGATE

Elementwise negation operation.

Precision Requirements

Integer Results must be exact.

For floating-point values, the following rules apply:

- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero before calculation.
- Infinity, NaN, and Zero behavior as defined in the following table.
- Otherwise floating-point results must be exact.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input1	shape	0 to MAX_RANK	Input tensor
Input	T<in_out_t>	input1_zp	[1]	1	Input 1 zero point. Must be zero for non-int8 types.
Input	T<in_out_t>	output_zp	[1]	1	Output zero point. Must be zero for non-int8 types.
Output	T<in_out_t>	output	shape	0 to MAX_RANK	Output tensor of same type, size as the input tensor

Compile Time Constant Status:

Argument	CTC enabled profile(s)	CTC disabled extension(s)
input1_zp	PRO-INT, PRO-FP	EXT-DYNAMIC

Argument	CTC enabled profile(s)	CTC disabled extension(s)
output_zp	PRO-INT, PRO-FP	EXT-DYNAMIC

Supported Data Types:

Profile/Extension	Mode	in_out_t	acc_t
PRO-FP	fp16	fp16_t	fp16_t
PRO-FP	fp32	fp32_t	fp32_t
PRO-INT	signed 16	i16_t	i32_t
PRO-INT	signed 32	i32_t	i32_t
PRO-INT	signed 8	i8_t	i32_t
EXT-BF16	bf16	bf16_t	bf16_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

Floating-point behavior:

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

```
ERROR_IF(!is_same<in_out_t,i8_t>() && input1_zp != 0); // Zero point only for int8_t
ERROR_IF(!is_same<in_out_t,i8_t>() && output_zp != 0); // Zero point only for int8_t
for_each_data_position(index in shape) {
    in_out_t value1 = tensor_read<in_out_t>(input1, shape, index);
    acc_t value = apply_sub_s<acc_t>(sign_extend<acc_t>(value1),
                                     sign_extend<acc_t>(input1_zp));
    value = apply_sub_s<acc_t>(0, value);
    value = apply_add_s<acc_t>(value, sign_extend<acc_t>(output_zp));
    in_out_t result = truncate<in_out_t>(apply_clip_s<acc_t>(value,
                                                             minimum_s<in_out_t>(),
                                                             maximum_s<in_out_t>()));
    tensor_write<in_out_t>(output, shape, index, result);
}
```

2.6.11. RECIPROCAL

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

Precision Requirements

- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero before calculation.

- Infinity, NaN, and Zero behave as defined in the following table.
- Otherwise, the following may be used to validate the result:
  - Let `out_imp` the implementation output.
  - Let `out_ref` be the result calculated using fp64_t arithmetic.
  - Then `tosa_reference_check_fp<in_t>(out_imp, out_ref, 1)` must be true.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input1	shape	0 to MAX_RANK	Input tensor
Output	T<in_out_t>	output	shape	0 to MAX_RANK	Output tensor of same type, size as the input tensor

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-FP	fp16	fp16_t
PRO-FP	fp32	fp32_t
EXT-BF16	bf16	bf16_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

Floating-point behavior:

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

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

2.6.12. RSQRT

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

Precision Requirements

- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero before calculation.

- Infinity, NaN, and Zero behave as defined in the following table.
- Otherwise, the following may be used to validate the result:
  - Let `out_imp` the implementation output.
  - Let `out_ref` be the result calculated using fp64_t arithmetic.
  - Then `tosa_reference_check_fp<in_t>(out_imp, out_ref, 2)` must be true.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input1	shape	0 to MAX_RANK	Input tensor
Output	T<in_out_t>	output	shape	0 to MAX_RANK	Output tensor of same type, size as the input tensor

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-FP	fp16	fp16_t
PRO-FP	fp32	fp32_t
EXT-BF16	bf16	bf16_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

Floating-point behavior:

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

```
for_each_data_position(index in shape) {
    in_out_t value1 = tensor_read<in_out_t>(input1, shape, 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);
}
```

2.6.13. SIN

Elementwise sine operation for values given in radians.

Precision Requirements

- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero before calculation.
- Infinity, NaN, and Zero behavior as defined in the Floating-point behavior table.
- Otherwise, the following may be used to validate the result:
  - Let `x` be an input element.
  - Let `out_imp` the implementation output.
  - Let `out_ref` be the result calculated using fp64_t arithmetic.
  - Let `err_bnd = max(calcAbsErrorBound<in_out_t>(x, 2, 0, 2), pi*normal_min<in_out_t>())`.
  - Then `tosa_reference_check_fp_bnd<in_out_t>(out_imp, out_ref, err_bnd)` must be true.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input1	shape	0 to MAX_RANK	Input tensor
Output	T<in_out_t>	output	shape	0 to MAX_RANK	Output tensor of same type and shape as input

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-FP	fp16	fp16_t
PRO-FP	fp32	fp32_t
EXT-BF16	bf16	bf16_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

Floating-point behavior:

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

```
for_each_data_position(index in shape) {
    in_out_t value1 = tensor_read<in_out_t>(input1, shape, index);
    in_out_t value = sin<in_out_t>(value1);
    tensor_write<in_out_t>(output, shape, index, value);
```

```
}
```

2.7. Elementwise Ternary Operators

2.7.1. SELECT

Elementwise select of the output based on a condition.

Precision Requirements

Integer results must be exact.

For floating-point values in input2 and input3, the following rules apply:

- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero before calculation.
- If an output is a NaN, any NaN is permitted.
- Otherwise floating-point results must be exact.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<bool_t>	input1	shape1	0 to MAX_RANK	Input selector tensor
Input	T<in_out_t>	input2	shape2	0 to MAX_RANK	Input value tensor if input1 is True
Input	T<in_out_t>	input3	shape3	0 to MAX_RANK	Input value tensor if input1 is False
Output	T<in_out_t>	output	shape	0 to MAX_RANK	Output tensor of same type as input2 and input3

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-FP	fp16	fp16_t
PRO-FP	fp32	fp32_t
PRO-INT or PRO-FP	Boolean	bool_t
PRO-INT	signed 16	i16_t
PRO-INT	signed 32	i32_t
PRO-INT	signed 8	i8_t
EXT-BF16	bf16	bf16_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

```
ERROR_IF(shape != broadcast_shape(broadcast_shape(shape1, shape2), shape3));
for_each_data_position(index in shape) {
    shape_t index1 = apply_broadcast(shape, shape1, index);
    shape_t index2 = apply_broadcast(shape, shape2, index);
    shape_t index3 = apply_broadcast(shape, shape3, index);
    bool_t value1 = tensor_read<bool_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);
}
```

2.8. Comparison Operators

2.8.1. EQUAL

Elementwise comparison operation

Precision Requirements

Integer results must be exact.

The following rules apply to floating-point inputs:

- Comparison rules:
  - The sign of a zero is ignored.
  - Infinities of the same sign compare as equal.
  - If either or both input values are a NaN, the result is false.
- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero before calculation.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_t>	input1	shape1	0 to MAX_RANK	Input tensor
Input	T<in_t>	input2	shape2	0 to MAX_RANK	Input tensor with the same rank as input1
Output	T<out_t>	output	shape	0 to MAX_RANK	Output tensor

Supported Data Types:

Profile/Extension	Mode	in_t	out_t
PRO-FP	fp16	fp16_t	bool_t
PRO-FP	fp32	fp32_t	bool_t
PRO-INT	signed 32	i32_t	bool_t
EXT-BF16	bf16	bf16_t	bool_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

```
ERROR_IF(shape != broadcast_shape(shape1, shape2));
for_each_data_position(index in shape) {
    shape_t index1 = apply_broadcast(shape, shape1, index);
    shape_t 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);
}
```

2.8.2. GREATER

Elementwise greater than comparison operation

Precision Requirements

Integer results must be exact.

The following rules apply to floating-point inputs:

- Comparison rules:
  - The sign of a zero is ignored.
  - Infinities of the same sign compare as equal.
  - If either or both input values are a NaN, the result is false.
- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero before calculation.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_t>	input1	shape1	0 to MAX_RANK	Input tensor

Argument	Type	Name	Shape	Rank	Description
Input	T<in_t>	input2	shape2	0 to MAX_RANK	Input tensor with the same rank as input1
Output	T<out_t>	output	shape	0 to MAX_RANK	Output tensor

Supported Data Types:

Profile/Extension	Mode	in_t	out_t
PRO-FP	fp16	fp16_t	bool_t
PRO-FP	fp32	fp32_t	bool_t
PRO-INT	signed 32	i32_t	bool_t
EXT-BF16	bf16	bf16_t	bool_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

```
ERROR_IF(shape != broadcast_shape(shape1, shape2));
for_each_data_position(index in shape) {
    shape_t index1 = apply_broadcast(shape, shape1, index);
    shape_t 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);
}
```

2.8.3. GREATER_EQUAL

Elementwise comparison operation

Precision Requirements

Integer results must be exact.

The following rules apply to floating-point inputs:

- Comparison rules:
  - The sign of a zero is ignored.
  - Infinities of the same sign compare as equal.

    - If either or both input values are a NaN, the result is false.
- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero before calculation.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_t>	input1	shape1	0 to MAX_RANK	Input tensor
Input	T<in_t>	input2	shape2	0 to MAX_RANK	Input tensor with the same rank as input1
Output	T<out_t>	output	shape	0 to MAX_RANK	Output tensor

Supported Data Types:

Profile/Extension	Mode	in_t	out_t
PRO-FP	fp16	fp16_t	bool_t
PRO-FP	fp32	fp32_t	bool_t
PRO-INT	signed 32	i32_t	bool_t
EXT-BF16	bf16	bf16_t	bool_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

```
ERROR_IF(shape != broadcast_shape(shape1, shape2));
for_each_data_position(index in shape) {
    shape_t index1 = apply_broadcast(shape, shape1, index);
    shape_t 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);
}
```

2.9. Reduction Operators

2.9.1. REDUCE_ALL

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

Precision Requirements

Results must be exact.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input	shape1	1 to MAX_RANK	Input tensor
Attribute	i32_t	axis	-		Axis to reduce, in range from 0 to rank(shape1)-1
Output	T<in_out_t>	output	shape	1 to MAX_RANK	Output tensor. Same rank as the input tensor.

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-INT or PRO-FP	Boolean	bool_t

Operation Function:

```
ERROR_IF(axis < 0  || axis >= rank(shape1));
ERROR_IF(shape[axis] != 1);
shape_t left_shape  = (axis > 1) ? shape[0:axis-1] : [];
shape_t right_shape = (axis < rank(shape)-1) ? shape[axis+1:rank(shape)-1] : [];
for_each_data_position(left_index in left_shape) {
    for_each_data_position(right_index in right_shape) {
        in_out_t acc = true;
        for (tensor_size_t i = 0; i < shape1[axis]; i++) {
            shape_t index = flatten(left_index, [i], right_index);
            in_out_t value = tensor_read<in_out_t>(input, shape1, index);
            acc = acc && value;
        }
        shape_t out_index = flatten(left_index, [0], right_index);
        tensor_write<in_out_t>(output, shape, out_index, acc);
    }
}
```

2.9.2. REDUCE_ANY

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

Precision Requirements

Results must be exact.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input	shape1	1 to MAX_RANK	Input tensor
Attribute	i32_t	axis	-		Axis to reduce, in range from 0 to rank(shape1)-1
Output	T<in_out_t>	output	shape	1 to MAX_RANK	Output tensor. Same rank as the input tensor.

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-INT or PRO-FP	Boolean	bool_t

Operation Function:

```
ERROR_IF(axis < 0  || axis >= rank(shape1));
ERROR_IF(shape[axis] != 1);
shape_t left_shape  = (axis > 1) ? shape[0:axis-1] : [];
shape_t right_shape = (axis < rank(shape)-1) ? shape[axis+1:rank(shape)-1] : [];
for_each_data_position(left_index in left_shape) {
    for_each_data_position(right_index in right_shape) {
        in_out_t acc = false;
        for (tensor_size_t i = 0; i < shape1[axis]; i++) {
            shape_t index = flatten(left_index, [i], right_index);
            in_out_t value = tensor_read<in_out_t>(input, shape1, index);
            acc = acc || value;
        }
        shape_t out_index = flatten(left_index, [0], right_index);
        tensor_write<in_out_t>(output, shape, out_index, acc);
    }
}
```

2.9.3. REDUCE_MAX

Reduce a tensor along the given axis with a maximum operation

Precision Requirements

Integer results must be exact.

NaN propagation mode only affects floating-point types. It indicates either propagating or ignoring NaN.

The following rules apply to floating-point inputs:

- Comparison rules:
  - The sign of a zero is ignored.

    - Infinities of the same sign compare as equal.
    - In the NaN propagating mode, if any input value along the reduction axis is a NaN, the result is NaN.
    - In the NaN ignoring mode, if all input values along the reduction axis are NaN, the result is NaN. Otherwise the result is the maximum non-NaN value.
- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero before calculation.
- If a floating-point result is zero, then the result must be either +0.0 or -0.0 but either sign is permitted.
- If the result is a subnormal value for bf16_t, fp16_t, or fp32_t, the result may be a zero of either sign.
- Otherwise floating-point results must be exact.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input	shape1	1 to MAX_RANK	Input tensor
Attribute	i32_t	axis	-		Axis to reduce, in range from 0 to rank(shape1)-1
Attribute	nan_propagation_mode_t	nan_mode	-		PROPAGATE or IGNORE. Set to PROPAGATE by default. This attribute affects the floating-point NaN propagation approach. This attribute is ignored by non floating-point types.
Output	T<in_out_t>	output	shape	1 to MAX_RANK	Output tensor. Same rank as the input tensor.

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-FP	fp16	fp16_t
PRO-FP	fp32	fp32_t
PRO-INT	signed 16	i16_t
PRO-INT	signed 32	i32_t
PRO-INT	signed 8	i8_t
EXT-BF16	bf16	bf16_t

Operation Function:

```
ERROR_IF(axis < 0  || axis >= rank(shape1));
ERROR_IF(shape[axis] != 1);
shape_t left_shape  = (axis > 1) ? shape[0:axis-1] : [];
```

```
shape_t right_shape = (axis < rank(shape)-1) ? shape[axis+1:rank(shape)-1] : [];
for_each_data_position(left_index in left_shape) {
    for_each_data_position(right_index in right_shape) {
        in_out_t acc = (is_floating_point<in_out_t>() && nan_mode == IGNORE)
                           ? nan<in_out_t>()
                           : minimum_s<in_out_t>();
        for (tensor_size_t i = 0; i < shape1[axis]; i++) {
            shape_t index = flatten(left_index, [i], right_index);
            in_out_t value = tensor_read<in_out_t>(input, shape1, index);
            acc = apply_max_s<in_out_t>(acc, value, nan_mode);
        }
        shape_t out_index = flatten(left_index, [0], right_index);
        tensor_write<in_out_t>(output, shape, out_index, acc);
    }
}
```

2.9.4. REDUCE_MIN

Reduce a tensor along the given axis with a minimum operation

Precision Requirements

Integer results must be exact.

NaN propagation mode only affects floating-point types. It indicates either propagating or ignoring NaN.

The following rules apply to floating-point inputs:

- Comparison rules:
  - The sign of a zero is ignored.
  - Infinities of the same sign compare as equal.
  - In the NaN propagating mode, if any input value along the reduction axis is a NaN, the result is NaN.
  - In the NaN ignoring mode, if all input values along the reduction axis are NaN, the result is NaN. Otherwise the result is the minimum non-NaN value.
- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero before calculation.
- If a floating-point result is zero, then the result must be either +0.0 or -0.0 but either sign is permitted.
- If the result is a subnormal value for bf16_t, fp16_t, or fp32_t, the result may be a zero of either sign.
- Otherwise floating-point results must be exact.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input	shape1	1 to MAX_RANK	Input tensor
Attribute	i32_t	axis	-		Axis to reduce, in range from 0 to rank(shape1)-1
Attribute	nan_propagation_mode_t	nan_mode	-		PROPAGATE or IGNORE. Set to PROPAGATE by default. This attribute affects the floating-point NaN propagation approach. This attribute is ignored by non floating-point types.
Output	T<in_out_t>	output	shape	1 to MAX_RANK	Output tensor. Same rank as the input tensor.

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-FP	fp16	fp16_t
PRO-FP	fp32	fp32_t
PRO-INT	signed 16	i16_t
PRO-INT	signed 32	i32_t
PRO-INT	signed 8	i8_t
EXT-BF16	bf16	bf16_t

Operation Function:

```
ERROR_IF(axis < 0  || axis >= rank(shape1));
ERROR_IF(shape[axis] != 1);
shape_t left_shape  = (axis > 1) ? shape[0:axis-1] : [];
shape_t right_shape = (axis < rank(shape)-1) ? shape[axis+1:rank(shape)-1] : [];
for_each_data_position(left_index in left_shape) {
    for_each_data_position(right_index in right_shape) {
        in_out_t acc = (is_floating_point<in_out_t>() && nan_mode == IGNORE)
                           ? nan<in_out_t>()
                           : maximum_s<in_out_t>();
        for (tensor_size_t i = 0; i < shape1[axis]; i++) {
            shape_t index = flatten(left_index, [i], right_index);
            in_out_t value = tensor_read<in_out_t>(input, shape1, index);
            acc = apply_min_s<in_out_t>(acc, value, nan_mode);
        }
        shape_t out_index = flatten(left_index, [0], right_index);
        tensor_write<in_out_t>(output, shape, out_index, acc);
    }
}
```

2.9.5. REDUCE_PRODUCT

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

Precision Requirements

- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero before calculation.
- If the input is a NaN, the output must be a NaN.
  - Otherwise the following may be used to validate the result:
  - Let `n` be number of elements in the product.
  - Let `out_imp` be the implementation result.
  - Let `out_ref` be the result calculated using fp64_t arithmetic.
  - Let `err_bnd = max(abs(out_ref), normal_min<in_out_t>()) * (pow(1 + pow(2, -normal_frac<in_out_t>() - 1), n) - 1)`.
  - Then `tosa_reference_check_fp_bnd<in_out_t>(out_imp, out_ref, err_bnd)` must be true.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input	shape1	1 to MAX_RANK	Input tensor
Attribute	i32_t	axis	-		Axis to reduce, in range from 0 to rank(shape1)-1
Output	T<in_out_t>	output	shape	1 to MAX_RANK	Output tensor. Same rank as the input tensor.

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-FP	fp16	fp16_t
PRO-FP	fp32	fp32_t
EXT-BF16	bf16	bf16_t

Operation Function:

```
ERROR_IF(axis < 0  || axis >= rank(shape1));
ERROR_IF(shape[axis] != 1);
shape_t left_shape  = (axis > 1) ? shape[0:axis-1] : [];
shape_t right_shape = (axis < rank(shape)-1) ? shape[axis+1:rank(shape)-1] : [];
for_each_data_position(left_index in left_shape) {
    for_each_data_position(right_index in right_shape) {
        in_out_t acc = 1.0;
        for (tensor_size_t i = 0; i < shape1[axis]; i++) {
            shape_t index = flatten(left_index, [i], right_index);
            in_out_t value = tensor_read<in_out_t>(input, shape1, index);
```

```
            acc = apply_mul_s<in_out_t>(acc, value);
        }
        shape_t out_index = flatten(left_index, [0], right_index);
        tensor_write<in_out_t>(output, shape, out_index, acc);
    }
}
```

2.9.6. REDUCE_SUM

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

Precision Requirements

Integer results must be exact.

Floating-point outputs can be expressed as a dot product of an input vector with a vector of ones. This dot product must meet the Dot product accuracy requirements.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input	shape1	1 to MAX_RANK	Input tensor
Attribute	i32_t	axis	-		Axis to reduce, in range from 0 to rank(shape1)-1
Output	T<in_out_t>	output	shape	1 to MAX_RANK	Output tensor. Same rank as the input tensor.

Supported Data Types:

Profile/Extension	Mode	in_out_t	acc_t
PRO-FP	fp16	fp16_t	fp16_t
PRO-FP	fp32	fp32_t	fp32_t
PRO-INT	signed 32	i32_t	i32_t
EXT-BF16	bf16	bf16_t	fp32_t

Operation Function:

```
ERROR_IF(axis < 0  || axis >= rank(shape1));
ERROR_IF(shape[axis] != 1);
shape_t left_shape  = (axis > 1) ? shape[0:axis-1] : [];
shape_t right_shape = (axis < rank(shape)-1) ? shape[axis+1:rank(shape)-1] : [];
for_each_data_position(left_index in left_shape) {
    for_each_data_position(right_index in right_shape) {
        acc_t acc = 0;
        for (tensor_size_t i = 0; i < shape1[axis]; i++) {
            shape_t index = flatten(left_index, [i], right_index);
```

```
            acc_t value = tensor_read<in_out_t>(input, shape1, index);
            acc = apply_add_s<acc_t>(acc, value);
        }
        shape_t out_index = flatten(left_index, [0], right_index);
        in_out_t result = static_cast<in_out_t>(acc);
        tensor_write<in_out_t>(output, shape, out_index, result);
    }
}
```

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.

Precision Requirements

Results must be exact.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	tensor_list_t <T<in_out_t >>	input1	shapes1	1 to MAX_RANK	List of input tensors. All inputs must have the same rank and data type
Attribute	i32_t	axis	-		Axis along which concatenation is to occur, in range from 0 to rank(shape)-1
Output	T<in_out_t>	output	shape	1 to MAX_RANK	Output tensor

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-FP	fp16	fp16_t
PRO-FP	fp32	fp32_t
PRO-INT or PRO-FP	Boolean	bool_t
PRO-INT	signed 32	i32_t
PRO-INT	signed 8	i8_t
EXT-BF16	bf16	bf16_t
EXT-FP8E4M3	fp8e4m3	fp8e4m3_t
EXT-FP8E5M2	fp8e5m2	fp8e5m2_t
EXT-INT16	signed 16	i16_t

Operation Function:

```
LEVEL_CHECK(tensor_list_shape(input1) <= MAX_TENSOR_LIST_SIZE);
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

```
ERROR_IF(input1 == []); // There must be at least one input in the input list
ERROR_IF(axis < 0 || axis >= max(1,rank(shapes1[0])));

// The following checks ensure all inputs are compatible for concatenation
// Iterate over each shape and dimension
// All shapes must have the same rank
// If the dimension is the axis dimension, sum the size of this dimension to check
// that the size of the output equals the size of the concatenated shapes
// For all other dimensions, the size must match for all inputs

tensor_size_t axis_sum = 0;
for (int32_t shape_index = 0; shape_index < length(shapes1); shape_index++) {
    ERROR_IF(rank(shapes1[shape_index]) != rank(shapes1[0]));
    for (int32_t axis_index = 0; axis_index < length(shapes1[0]); axis_index++) {
        if (axis_index == axis) {
            axis_sum += shapes1[shape_index][axis_index];
        }
        else {
            ERROR_IF(shapes1[shape_index][axis_index] != shapes1[0][axis_index]);
        }
    }
}
ERROR_IF(axis_sum != shape[axis]);

for_each_data_position(index1 in shape) {
    shape_t index2 = index1;
    for (int32_t 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] < shape_dim(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] - shape_dim(shapes1[t], axis);
    }
}
```

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.

Precision Requirements

Results must be exact.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input1	shape1	1 to MAX_RANK	Input tensor
Input	shape_t<[2*rank(shape1)]>	padding	[2*rank(shape1)]	1	Number of pad elements at the start and end of each dimension. The values in padding are interpreted as start, end of each dimension. As an example for rank 2, the values would be interpreted as [start_dim0, end_dim0, start_dim1, end_dim1].
Input	T<in_out_t>	pad_const	[1]	1	The value to be used as padding.
Output	T<in_out_t>	output	shape	1 to MAX_RANK	Output tensor of same type as the input tensor

Compile Time Constant Status:

Argument	CTC enabled profile(s)	CTC disabled extension(s)
padding	PRO-INT, PRO-FP	EXT-DYNAMIC
pad_const	PRO-INT, PRO-FP	EXT-DYNAMIC

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-FP	fp16	fp16_t
PRO-FP	fp32	fp32_t
PRO-INT or PRO-FP	Boolean	bool_t
PRO-INT	signed 16	i16_t
PRO-INT	signed 32	i32_t
PRO-INT	signed 8	i8_t
EXT-BF16	bf16	bf16_t
EXT-FP8E4M3	fp8e4m3	fp8e4m3_t
EXT-FP8E5M2	fp8e5m2	fp8e5m2_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

```
// Check output shape matches the padded input shape
ERROR_IF(rank(shape) != rank(shape1));
for (int32_t i = 0; i < rank(shape); i++) {
    ERROR_IF(padding[i * 2] < 0 || padding[(i * 2) + 1] < 0);
    ERROR_IF(shape[i] != padding[i * 2] + shape1[i] + padding[(i * 2) + 1]);
}
for_each_data_position(index in shape) {
    shape_t index1 = index;
    bool_t is_pad = false;
    for(int32_t i = 0; i < rank(shape); i++) {
        index1[i] = index1[i] - padding[i * 2];
        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);
}
```

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.

Precision Requirements

Results must be exact.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input1	shape1	0 to MAX_RANK	Input tensor
Input	shape_t<[rank(shape)]>	shape	[rank(shape)]	1	shape_t giving the new shape.
Output	T<in_out_t>	output	shape	0 to MAX_RANK	Output tensor of same type, size as the input tensor

Compile Time Constant Status:

Argument	CTC enabled profile(s)	CTC disabled extension(s)
shape	PRO-INT, PRO-FP	EXT-DYNAMIC

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-FP	fp16	fp16_t
PRO-FP	fp32	fp32_t
PRO-INT or PRO-FP	Boolean	bool_t
PRO-INT	signed 16	i16_t
PRO-INT	signed 32	i32_t
PRO-INT	signed 8	i8_t
EXT-BF16	bf16	bf16_t
EXT-FP8E4M3	fp8e4m3	fp8e4m3_t
EXT-FP8E5M2	fp8e5m2	fp8e5m2_t

Operation Function:

```
LEVEL_CHECK(rank(shape1) <= MAX_RANK);
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

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

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

    // Now read/write the value
    in_out_t val = tensor_read<in_out_t>(input1, shape1, tmp_index);
    tensor_write<in_out_t>(output, shape, index, val);
}
```

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.

Precision Requirements

Results must be exact.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input1	shape	1 to MAX_RANK	Input tensor
Attribute	i32_t	axis	-		Axis to reverse, in range from 0 to rank(shape)-1
Output	T<in_out_t>	output	shape	1 to MAX_RANK	Output tensor. Same shape as input tensor

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-FP	fp16	fp16_t
PRO-FP	fp32	fp32_t
PRO-INT or PRO-FP	Boolean	bool_t
PRO-INT	signed 16	i16_t
PRO-INT	signed 32	i32_t
PRO-INT	signed 8	i8_t
EXT-BF16	bf16	bf16_t
EXT-FP8E4M3	fp8e4m3	fp8e4m3_t
EXT-FP8E5M2	fp8e5m2	fp8e5m2_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

```
ERROR_IF(axis < 0 || axis >= rank(shape));
for_each_data_position(index in shape) {
    shape_t tmp_index = index;
    tmp_index[axis] = shape[axis] - 1 - index[axis];
    in_out_t value = tensor_read<in_out_t>(input1, shape, tmp_index);
    tensor_write<in_out_t>(output, shape, index, value);
}
```

2.10.5. SLICE

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

Precision Requirements

Results must be exact.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input1	shape1	1 to MAX_RANK	Input tensor
Input	shape_t<[rank(shape1)]>	start	[rank(shape1)]	1	List of integer coordinates, of length equal to the rank of input1. Start coordinate for slicing.
Input	shape_t<[rank(shape1)]>	size	[rank(shape1)]	1	List of integer size values, of length equal to the rank of input1. Size of the input to be used.
Output	T<in_out_t>	output	shape	1 to MAX_RANK	Output tensor of same type as the input tensor

Compile Time Constant Status:

Argument	CTC enabled profile(s)	CTC disabled extension(s)
start	PRO-INT, PRO-FP	EXT-DYNAMIC
size	PRO-INT, PRO-FP	EXT-DYNAMIC

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-FP	fp16	fp16_t
PRO-FP	fp32	fp32_t
PRO-INT or PRO-FP	Boolean	bool_t
PRO-INT	signed 16	i16_t
PRO-INT	signed 32	i32_t
PRO-INT	signed 8	i8_t
EXT-BF16	bf16	bf16_t
EXT-FP8E4M3	fp8e4m3	fp8e4m3_t
EXT-FP8E5M2	fp8e5m2	fp8e5m2_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

```
ERROR_IF(rank(shape1) != length(start) || rank(shape1) != length(size));
ERROR_IF(rank(shape1) != rank(shape));
// Sanity check the given coordinates, ensure start and end are
// within tensor bounds
for_each(0 <= index < rank(shape1)) {
    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_data_position(index in shape) {
    shape_t tmp_index = index;
    for(int32_t i = 0; i < rank(shape); i++) {
       tmp_index[i] = index[i] + start[i];
    }
    in_out_t value = tensor_read<in_out_t>(input1, shape1, tmp_index);
    tensor_write<in_out_t>(output, shape, index, value);
}
```

2.10.6. TILE

Replicates input1 multiples times along each dimension.

Precision Requirements

Results must be exact.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input1	shape1	1 to MAX_RANK	Input tensor
Input	shape_t<[rank(shape1)]>	multiples	[rank(shape1)]	1	Number of times to replicate input1 in each dimension
Output	T<in_out_t>	output	shape	1 to MAX_RANK	Output tensor of same type, rank as the input tensor

Compile Time Constant Status:

Argument	CTC enabled profile(s)	CTC disabled extension(s)
multiples	PRO-INT, PRO-FP	EXT-DYNAMIC

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-FP	fp16	fp16_t
PRO-FP	fp32	fp32_t
PRO-INT or PRO-FP	Boolean	bool_t
PRO-INT	signed 16	i16_t
PRO-INT	signed 32	i32_t
PRO-INT	signed 8	i8_t

Profile/Extension	Mode	in_out_t
EXT-BF16	bf16	bf16_t
EXT-FP8E4M3	fp8e4m3	fp8e4m3_t
EXT-FP8E5M2	fp8e5m2	fp8e5m2_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

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

for_each_data_position(index in shape) {
    shape_t tmp_index = index;
    for(int32_t 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>(input1, shape1, tmp_index);
    tensor_write<in_out_t>(output, shape, index, value);
}
```

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.

Precision Requirements

Results must be exact.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input1	shape1	1 to MAX_RANK	Input tensor
Attribute	T<i32_t>	perms	[rank(shape1)]	1	List of integers of length equal to the rank of input1. Values must be valid dimensions within shape1, and may not be repeated.
Output	T<in_out_t>	output	shape	1 to MAX_RANK	Output tensor of same type, rank as the input tensor

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-FP	fp16	fp16_t
PRO-FP	fp32	fp32_t
PRO-INT or PRO-FP	Boolean	bool_t
PRO-INT	signed 16	i16_t
PRO-INT	signed 32	i32_t
PRO-INT	signed 8	i8_t
EXT-BF16	bf16	bf16_t
EXT-FP8E4M3	fp8e4m3	fp8e4m3_t
EXT-FP8E5M2	fp8e5m2	fp8e5m2_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

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

bool_t indexes_used[rank(shape1)];
for_each_data_position(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(int32_t i = 0; i < rank(shape); i++) {
    ERROR_IF(shape1[perms[i]] != shape[i]);
}

for_each_data_position(index in shape) {
    shape_t tmp_index = index;
    for(int32_t i = 0; i < rank(shape); i++) {
        tmp_index[perms[i]] = index[i];
    }
    in_out_t value = tensor_read<in_out_t>(input1, shape1, tmp_index);
    tensor_write<in_out_t>(output, shape, index, value);
}
```

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.

Precision Requirements

Results must be exact.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	values	[N,K,C]	3	3D value tensor
Input	T<index_t>	indices	[N,W]	2	2D index tensor
Output	T<in_out_t>	output	[N,W,C]	3	3D output tensor

Supported Data Types:

Profile/Extension	Mode	index_t	in_out_t
PRO-FP	fp16	i32_t	fp16_t
PRO-FP	fp32	i32_t	fp32_t
PRO-INT	signed 16	i32_t	i16_t
PRO-INT	signed 32	i32_t	i32_t
PRO-INT	signed 8	i32_t	i8_t
EXT-BF16	bf16	i32_t	bf16_t
EXT-FP8E4M3	fp8e4m3	i32_t	fp8e4m3_t
EXT-FP8E5M2	fp8e5m2	i32_t	fp8e5m2_t

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);
    in_out_t value = tensor_read<in_out_t>(values, [N,K,C], [n,k,c]);
    tensor_write<in_out_t>(output, [N,W,C], [n,w,c], value);
}
```

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. It follows that K >= W. In use cases that require multiple updates to the same output position, these must be decomposed into multiple SCATTER operations.

Precision Requirements

Results must be exact.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	values_in	[N,K,C]	3	3D values in tensor
Input	T<index_t>	indices	[N,W]	2	2D index tensor
Input	T<in_out_t>	input	[N,W,C]	3	3D input tensor
Output	T<in_out_t>	values_out	[N,K,C]	3	3D output tensor

Supported Data Types:

Profile/Extension	Mode	index_t	in_out_t
PRO-FP	fp16	i32_t	fp16_t
PRO-FP	fp32	i32_t	fp32_t
PRO-INT	signed 16	i32_t	i16_t
PRO-INT	signed 32	i32_t	i32_t
PRO-INT	signed 8	i32_t	i8_t
EXT-BF16	bf16	i32_t	bf16_t
EXT-FP8E4M3	fp8e4m3	i32_t	fp8e4m3_t
EXT-FP8E5M2	fp8e5m2	i32_t	fp8e5m2_t

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) {
    in_out_t value = tensor_read<in_out_t>(values_in, [N,K,C], [n,k,c]);
    tensor_write<in_out_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);
    in_out_t value = tensor_read<in_out_t>(input, [N,W,C], [n,w,c]);
    tensor_write<in_out_t>(values_out, [N,K,C], [n, k, c], value);
    output_modified[n,k,c] = true;
}
```

2.12. Image Operators

2.12.1. RESIZE

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

The height dimension is scaled by factor (scale_y_n/scale_y_d). The width dimension is scaled by factor (scale_x_n/scale_x_d).

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 must be scaled by 1/(scale_y_n * scale_x_n) in a following operation to complete the interpolation (for example with a RESCALE operator).

The following examples show practical uses of the parameters:

- For approximate uniform input sampling between (0, 0) and (IH - 1, IW - 1) set
  - scale_y_n/scale_y_d = (OH - 1)/(IH - 1) as integer ratios
  - scale_x_n/scale_x_d = (OW - 1)/(IW - 1) as integer ratios
  - offset_x = 0, offset_y = 0, border_x = 0, border_y = 0
- For power of two upscale [OH - 1,OW - 1] = (1 << k) * [IH - 1, IW - 1], sampling between (0,0) and (IH - 1,IW - 1), set:
  - scale_y_n = (1 << k), scale_y_d = 1, offset_y = 0, border_y = 0
  - scale_x_n = (1 << k), scale_x_d = 1, offset_x = 0, border_x = 0
- For power of two upscale [OH,OW] = (1 << k) * [IH,IW], sampling range approximately (-0.5, -0.5) to (IH - 0.5, IW - 0.5), set:
  - scale_y_n = 2 << k, scale_y_d = 2, offset_y = -(1 << k) + 1, border_y = (1 << k) - 1

- scale_x_n = 2 << k, scale_x_d = 2, offset_x = -(1 << k) + 1, border_x = (1 << k) - 1

The output dimensions can be derived from the input dimensions by inverting the scale as described in the pseudocode. The [border_y, border_x] values adjust the output size to allow fractional sampling beyond integer input position (IH - 1,IW - 1).

The limit MAX_SCALE is applied to each scale ratio after reduction of the ratio. Individual scale numerator and denominator values are allowed to be larger than MAX_SCALE.

Precision Requirements

- The result corresponds to a sequence of floating-point calculations.
- The allowable error bound for the result of a resize is based on the maximum value of an element in the input tensor.
- Let `out_imp` be the implementation output.
- Let `out_ref` be the result calculated using fp64_t arithmetic.
- Let `ulp_bnd = calcAbsErrorBound<in_out_t>(max(abs(input)), 20.0, 0, 1)`.
- Let `relative_bnd = max(abs(input)) * 0.006`.
- Let `err_bnd = max(ulp_bnd, relative_bnd)`.
- Then `tosa_reference_check_fp_bnd<out_t>(out_imp, out_ref, err_bnd)` must be true.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_t>	input	[N,IH,IW,C]	4	Input tensor
Input	shape_t<[4]>	scale	[4]	1	[scale_y_n, scale_y_d, scale_x_n, scale_x_d]
Input	shape_t<[2]>	offset	[2]	1	[offset_y, offset_x]
Input	shape_t<[2]>	border	[2]	1	[border_y, border_x]
Attribute	resize_mode_t	mode	-		BILINEAR or NEAREST
Output	T<out_t>	output	[N,OH,OW,C]	4	Output tensor

Compile Time Constant Status:

Argument	CTC enabled profile(s)	CTC disabled extension(s)
scale	PRO-INT, PRO-FP	EXT-DYNAMIC
offset	PRO-INT, PRO-FP	EXT-DYNAMIC
border	PRO-INT, PRO-FP	EXT-DYNAMIC

Supported Data Types:

Profile/Extension	Mode	resize_t	in_t	out_t
PRO-FP	fp16	fp16_t	fp16_t	fp16_t
PRO-FP	fp32	fp32_t	fp32_t	fp32_t
PRO-INT	signed 8, bilinear	i16_t	i8_t	i32_t
PRO-INT	signed 8, nearest	i16_t	i8_t	i8_t
EXT-BF16	bf16	bf16_t	bf16_t	bf16_t
EXT-INT16	signed 16, bilinear	i16_t	i16_t	i48_t
EXT-INT16	signed 16, nearest	i16_t	i16_t	i16_t

Operation Function:

```
LEVEL_CHECK(scale_y_n/scale_y_d <= MAX_SCALE);
LEVEL_CHECK(scale_x_n/scale_x_d <= MAX_SCALE);
```

Resize Modes:

Mode	Description
NEAREST	Nearest Neighbor
BILINEAR	Bilinear interpolation

```
// 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(scale_y_n <= 0 || scale_y_d <= 0 || scale_x_n <= 0 || scale_x_d <= 0);
// if in_t=int8_t ensure that an int32_t accumulator can be used
ERROR_IF(scale_y_n > (1 << 11) || scale_x_n > (1 << 11));
// set a consistent lower limit of 1/16 downscale to simplify implementations
ERROR_IF(scale_y_d >= 16 * scale_y_n || scale_x_d >= 16 * scale_x_n);
ERROR_IF(offset_y < -scale_y_n || offset_y >= 16 * scale_y_n);
ERROR_IF(offset_x < -scale_x_n || offset_x >= 16 * scale_x_n);
ERROR_IF(border_y < -16 * scale_y_n || border_y >= scale_y_n);
ERROR_IF(border_x < -16 * scale_x_n || border_x >= scale_x_n);
ERROR_IF(OH != idiv_check((IH - 1) * scale_y_n - offset_y + border_y, scale_y_d) + 1);
ERROR_IF(OW != idiv_check((IW - 1) * scale_x_n - offset_x + border_x, scale_x_d) + 1);
for_each(0 <= n < N, 0 <= oy < OH, 0 <= ox < OW, 0 <= c < C) {
    out_t acc;
    resize_t dx, dy;
    resize_t unit_x, unit_y;

    unit_x = (is_floating_point<resize_t>()) ? 1.0 : scale_x_n;
    unit_y = (is_floating_point<resize_t>()) ? 1.0 : scale_y_n;

    int32_t y = oy * scale_y_d + offset_y;
    int32_t x = ox * scale_x_d + offset_x;
```

```
    int16_t iy = idiv_floor(y, scale_y_n);
    int16_t ix = idiv_floor(x, scale_x_n);
    int16_t ry = y - iy * scale_y_n;  // (y % scale_y_n)
    int16_t rx = x - ix * scale_x_n;  // (x % scale_x_n)

    if (is_floating_point<resize_t>()) {
        dy = static_cast<resize_t>(ry) / static_cast<resize_t>(scale_y_n);
        dx = static_cast<resize_t>(rx) / static_cast<resize_t>(scale_x_n);
    } else {
        dy = ry;
        dx = rx;
    }
    // Note that -1 <= iy < IH and -1 <= ix < IW
    int16_t iy0 = apply_max_s(iy, 0);
    int16_t iy1 = apply_min_s(iy + 1, IH - 1);
    int16_t ix0 = apply_max_s(ix, 0);
    int16_t ix1 = apply_min_s(ix + 1, IW - 1);
    if (mode==BILINEAR) {
        using in_s_t = make_signed(in_t); // Use signed calculations for i8/i16
        in_s_t v00 = static_cast<in_s_t>(tensor_read<in_t>(input, [N,IH,IW,C],
[n,iy0,ix0,c]));
        in_s_t v01 = static_cast<in_s_t>(tensor_read<in_t>(input, [N,IH,IW,C],
[n,iy0,ix1,c]));
        in_s_t v10 = static_cast<in_s_t>(tensor_read<in_t>(input, [N,IH,IW,C],
[n,iy1,ix0,c]));
        in_s_t v11 = static_cast<in_s_t>(tensor_read<in_t>(input, [N,IH,IW,C],
[n,iy1,ix1,c]));
        acc  = v00 * (unit_y - dy) * (unit_x - dx);
        acc += v01 * (unit_y - dy) * dx;
        acc += v10 * dy * (unit_x - dx);
        acc += v11 * dy * dx;
        tensor_write<out_t>(output, [N,OH,OW,C], [n,oy,ox,c], acc);
    } else if (mode==NEAREST) {
        int32_t iy, ix;
        if (is_floating_point<resize_t>()) {
            iy = (dy >= 0.5) ? iy1 : iy0;
            ix = (dx >= 0.5) ? ix1 : ix0;
        } else {
            iy = (2 * dy >= scale_y_n) ? iy1 : iy0;
            ix = (2 * dx >= scale_x_n) ? ix1 : ix0;
        }
        in_t 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);
    }
}
```

2.13. Type Conversion

2.13.1. CAST

Casts a tensor from one data type to another.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_t>	input	shape	0 to MAX_RANK	Input tensor
Output	T<out_t>	output	shape	0 to MAX_RANK	Output tensor

Supported Data Types:

Profile/Extension	Mode	in_t	out_t
PRO-FP	fp16 to fp32	fp16_t	fp32_t
PRO-FP	fp16 to signed 16	fp16_t	i16_t
PRO-FP	fp16 to signed 32	fp16_t	i32_t
PRO-FP	fp16 to signed 8	fp16_t	i8_t
PRO-FP	fp32 to fp16	fp32_t	fp16_t
PRO-FP	fp32 to signed 16	fp32_t	i16_t
PRO-FP	fp32 to signed 32	fp32_t	i32_t
PRO-FP	fp32 to signed 8	fp32_t	i8_t
PRO-FP	signed 16 to fp16	i16_t	fp16_t
PRO-FP	signed 16 to fp32	i16_t	fp32_t
PRO-FP	signed 32 to fp16	i32_t	fp16_t
PRO-FP	signed 32 to fp32	i32_t	fp32_t
PRO-FP	signed 8 to fp16	i8_t	fp16_t
PRO-FP	signed 8 to fp32	i8_t	fp32_t
PRO-INT	bool to signed 16	bool_t	i16_t
PRO-INT	bool to signed 32	bool_t	i32_t
PRO-INT	bool to signed 8	bool_t	i8_t
PRO-INT	signed 16 to bool	i16_t	bool_t
PRO-INT	signed 16 to signed 32	i16_t	i32_t
PRO-INT	signed 16 to signed 8	i16_t	i8_t
PRO-INT	signed 32 to bool	i32_t	bool_t
PRO-INT	signed 32 to signed 16	i32_t	i16_t
PRO-INT	signed 32 to signed 8	i32_t	i8_t
PRO-INT	signed 8 to bool	i8_t	bool_t
PRO-INT	signed 8 to signed 16	i8_t	i16_t

Profile/Extension	Mode	in_t	out_t
PRO-INT	signed 8 to signed 32	i8_t	i32_t
EXT-BF16	bf16 to fp32	bf16_t	fp32_t
EXT-BF16	bf16 to signed 16	bf16_t	i16_t
EXT-BF16	bf16 to signed 32	bf16_t	i32_t
EXT-BF16	bf16 to signed 8	bf16_t	i8_t
EXT-BF16	fp32 to bf16	fp32_t	bf16_t
EXT-BF16	signed 16 to bf16	i16_t	bf16_t
EXT-BF16	signed 32 to bf16	i32_t	bf16_t
EXT-BF16	signed 8 to bf16	i8_t	bf16_t
EXT-BF16 and EXT-FP8E4M3	bf16 to fp8e4m3	bf16_t	fp8e4m3_t
EXT-BF16 and EXT-FP8E4M3	fp8e4m3 to bf16	fp8e4m3_t	bf16_t
EXT-BF16 and EXT-FP8E5M2	bf16 to fp8e5m2	bf16_t	fp8e5m2_t
EXT-BF16 and EXT-FP8E5M2	fp8e5m2 to bf16	fp8e5m2_t	bf16_t
EXT-FP8E4M3	fp16 to fp8e4m3	fp16_t	fp8e4m3_t
EXT-FP8E4M3	fp32 to fp8e4m3	fp32_t	fp8e4m3_t
EXT-FP8E4M3	fp8e4m3 to fp16	fp8e4m3_t	fp16_t
EXT-FP8E4M3	fp8e4m3 to fp32	fp8e4m3_t	fp32_t
EXT-FP8E5M2	fp16 to fp8e5m2	fp16_t	fp8e5m2_t
EXT-FP8E5M2	fp32 to fp8e5m2	fp32_t	fp8e5m2_t
EXT-FP8E5M2	fp8e5m2 to fp16	fp8e5m2_t	fp16_t
EXT-FP8E5M2	fp8e5m2 to fp32	fp8e5m2_t	fp32_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

Precision Requirements

When casting between integer types, the results must be exact.

Rules when casting between floating-point types:

- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero before calculation.
- If the input value is a NaN, a NaN of the output type must be returned.

- The following sequence describes conversion between floating-point types:
  - The mantissa of the input value is converted to the mantissa size of the output format, with rounding if needed.
  - If the resulting value is below the minimum subnormal number magnitude for the target format, a zero of the appropriate sign must be returned.
  - For bf16_t, fp16_t, and fp32_t, if the resulting value is below the minimum normal number magnitude for the target format, either the corresponding subnormal value may be returned, or a zero of the appropriate sign may be returned.
  - If the output type is fp8e4m3_t or fp8e5m2_t the result must use the non-saturating conversion mode defined in OCP-OFP8.
  - Otherwise if the resulting value is outside of the output representable range, an infinity of the appropriate sign must be returned.
- If the input value is inside the output representable range then the following accuracy must be met:
  - Let `x` be an input element.
  - Let `out_imp` be the implementation output.
  - Let `out_ref` be the result calculated using fp64_t arithmetic.
  - Let `ulp = tosa_reference_ulp<out_t>(out_ref)` define one ulp of error.
  - Then `abs(out_ref) - ulp <= abs(x) <= abs(out_ref) + 0.5*ulp` must be true and `x` must have the correct sign.
  - This allows for both round to zero and round to nearest rounding modes.

Rules when casting between different types:

- Casting from floating-point to integer:
  - Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero before calculation.
  - If any input is a NaN, the result is unpredictable.
  - Result overflows must be saturated.
  - Conversion must use the round to nearest, ties to even rounding mode.
- Casting from integer to floating-point:
  - Let `out_imp` be the implementation output.
  - Let `out_ref` be the result calculated using fp64_t arithmetic.
  - Then `tosa_reference_check_fp<out_t>(out_imp, out_ref, 0.5)` must be true.
- Casting to boolean type:
  - The result must be exact.

```
for_each_data_position(index in shape) {
    in_t in = tensor_read<in_t>(input, shape, index);
    out_t out;
    if (is_same<out_t,bool_t>()) {
```

```
        out = (in != 0) ? true : false;
    } else if (is_floating_point<out_t>()) {
        // Conversion to float cases
        if (is_same<in_t,bool_t>()) {
            out = (in) ? 1.0 : 0.0;
        }
        out = round_to_nearest_float(in);
    } else {
        // Conversion to integer cases
        if (is_same<in_t,bool_t>()) {
            out = (in) ? 1 : 0;
        } else if (is_floating_point<in_t>()) {
            out = truncate<out_t>(apply_clip_s<i32_t>(round_to_nearest_int(in),
minimum_s<out_t>(), maximum_s<out_t>()));
        } else if (sizeof<out_t>() >= sizeof<in_t>()) {
            out = sign_extend<out_t>(in);
        } else {
            out = truncate<out_t>(in);
        }
    }
    tensor_write<out_t>(output, shape, index, out);
}
```

2.13.2. RESCALE

RESCALE is defined using an integer multiply, add, and shift.

Rescale 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 result fits in 32 bits. In particular, a 48-bit value cannot be scaled with the 32-bit multiplier because the accumulator would need to have 80 bits.

The apply_scale_* functions provide a scaling of approximately (multiplier / 2^{shift}).

The shift and value range are 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.

For apply_scale_32, the value must be between `(-1 << (shift - 1)) <= value < (1 << (shift - 1))`. This allows for 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.

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.

The right shift of result is an arithmetic shift.

For implementation details of the apply_scale_functions, see Scaling Helpers.

Precision Requirements

If rounding_mode is SINGLE_ROUND or DOUBLE_ROUND, results must be exact.

If rounding_mode is INEXACT_ROUND, the following must be true:

- Let `x` be the input value to be rescaled.
- Let `m` be the scaling multiplier.
- Let `out_ref = (x - input_zp) * m * exp2(-shift)` be the result calculated using fp64_t arithmetic.
- Note (informational): The error bound accounts for 3 rounding errors from casting the activation input, multiplier input, and floating-point multiply, plus 1 rounding error from adding `output_zp` and 0.5 absolute error from the final output rounding.
  A SINGLE_ROUND exact implementation will automatically meet this error bound since its rounding error is at most 0.5.
- Let `err_bnd = 0.5 + (3 * abs(out_ref) + abs(out_ref + output_zp)) * exp2(-normal_frac<fp32>() - 1)`.
- Let `out_ref_min = apply_clip_s<fp64_t>(apply_ceil<fp64_t>(out_ref - err_bnd + output_zp), minimum_s<out_t>, maximum_s<out_t>)`.
- Let `out_ref_max = apply_clip_s<fp64_t>(apply_floor<fp64_t>(out_ref + err_bnd + output_zp), minimum_s<out_t>, maximum_s<out_t>)`.
- Let `out_ref_min_int = static_cast<out_t>(out_ref_min)`.
- Let `out_ref_max_int = static_cast<out_t>(out_ref_max)`.
- Let `out_imp` be the implementation output.
- Then `out_ref_min_int <= out_imp <= out_ref_max_int` must be true.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_t>	input	shape	0 to MAX_RANK	Input tensor
Input	T<mul_t>	multiplier	[NC]	1	Scaling multiplier array
Input	T<i8_t>	shift	[NC]	1	Scaling shift array

Argument	Type	Name	Shape	Rank	Description
Input	T<in_t>	input_zp	[1]	1	Input tensor zero point. int8/uint8 can have zero point within their valid range. uint16 zero point must be either 0 or 32768. All other types must have zero point equal to 0.
Input	T<out_t>	output_zp	[1]	1	Output tensor zero point.int8/uint8 can have zero point within their valid range. uint16 zero point must be either 0 or 32768. All other types must have zero point equal to 0.
Attribute	bool_t	scale32	-		if (scale32) mul_t=i32_t else mul_t=i16_t
Attribute	rounding_mode_t	rounding_mode	-		Select rounding mode
Attribute	bool_t	per_channel	-		if (per_channel) NC=shape[rank(shape)-1] else NC=1
Attribute	bool_t	input_unsigned	-		If True, treat the input values as unsigned.
Attribute	bool_t	output_unsigned	-		If True, treat the output values as unsigned.
Output	T<out_t>	output	shape	0 to MAX_RANK	Output tensor with the same shape as input

Compile Time Constant Status:

Argument	CTC enabled profile(s)	CTC disabled extension(s)
multiplier	PRO-INT	EXT-DYNAMIC
shift	PRO-INT	EXT-DYNAMIC
input_zp	PRO-INT	EXT-DYNAMIC
output_zp	PRO-INT	EXT-DYNAMIC

Supported Data Types:

Profile/Extension	Mode	in_t	out_t
PRO-INT	16-bit to 16-bit	i16_t	i16_t
PRO-INT	16-bit to 32-bit	i16_t	i32_t

Profile/Extension	Mode	in_t	out_t
PRO-INT	16-bit to 8-bit	i16_t	i8_t
PRO-INT	32-bit to 16-bit	i32_t	i16_t
PRO-INT	32-bit to 32-bit	i32_t	i32_t
PRO-INT	32-bit to 8-bit	i32_t	i8_t
PRO-INT	8-bit to 16-bit	i8_t	i16_t
PRO-INT	8-bit to 32-bit	i8_t	i32_t
PRO-INT	8-bit to 8-bit	i8_t	i8_t
EXT-INT16	48-bit to 16-bit	i48_t	i16_t
EXT-INT16	48-bit to 32-bit	i48_t	i32_t
EXT-INT16	48-bit to 8-bit	i48_t	i8_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

```
for_each_data_position(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(!is_same<in_t,i8_t>() &&
             (!is_same<in_t,i16_t>() || input_unsigned == false) && input_zp != 0);
    ERROR_IF(!is_same<out_t,i8_t>() &&
             (!is_same<out_t,i16_t>() || output_unsigned == false) && output_zp != 0);
    ERROR_IF(is_same<in_t,i16_t>() && input_unsigned == true && input_zp != 0 &&
input_zp != 32768);
    ERROR_IF(is_same<out_t,i16_t>() && output_unsigned == true && output_zp != 0 &&
output_zp != 32768);
    ERROR_IF(scale32 && is_same<in_t,i48_t>());
    ERROR_IF(!scale32 && (rounding_mode == DOUBLE_ROUND));
    ERROR_IF(input_unsigned && output_unsigned);
    ERROR_IF(is_same<out_t,i32_t>() && input_unsigned);
    ERROR_IF(is_same<in_t,i32_t>() && output_unsigned);
    ERROR_IF(is_same<in_t,i48_t>() && output_unsigned);
    ERROR_IF(is_same<in_t, i48_t> && input_unsigned); // No support for u48 inputs
    ERROR_IF(is_same<in_t, i32_t> && input_unsigned); // No support for u32 inputs
    ERROR_IF(is_same<out_t, i32_t> && output_unsigned); // No support for u32 outputs
    ERROR_IF(per_channel && rank(input) < 1);

    in_t in_value = tensor_read<in_t>(input, shape, index);

    int48_t value, extended_in_zp;
    if (input_unsigned) {
        value = zero_extend<int48_t>(in_value);
```

```
        extended_in_zp = zero_extend<int48_t>(input_zp);
    }
    else {
        value = sign_extend<int48_t>(value);
        extended_in_zp = sign_extend<int48_t>(input_zp);
    }

    value = value - extended_in_zp;
    int c = (per_channel) ? index[rank(input) - 1] : 0;
    int32_t result = (scale32) ?
        apply_scale_32(value, multiplier[c], shift[c], rounding_mode == DOUBLE_ROUND)
:
        apply_scale_16(value, multiplier[c], shift[c]);

    out_t out;
    if (output_unsigned) {
        int32_t extended_out_zp = zero_extend<int32_t>(output_zp);
        result = apply_add_s<int32_t>(result, extended_out_zp);
        out = static_cast<out_t>(apply_clip_u<i32_t>(result,
                                                     minimum_u<out_t>(),
                                                     maximum_u<out_t>()));
    }
    else {
        int32_t extended_out_zp = sign_extend<int32_t>(output_zp);
        result = apply_add_s<int32_t>(result, extended_out_zp);
        out = static_cast<out_t>(apply_clip_s<i32_t>(result,
                                                     minimum_s<out_t>(),
                                                     maximum_s<out_t>()));
    }
    tensor_write<out_t>(output, shape, index, out);
}
```

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.

Precision Requirements

Integer results must be exact.

For floating-point values, the following rules apply:

- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Attribute	T<out_t>	values	shape	0 to MAX_RANK	Constant values
Output	T<out_t>	output	shape	0 to MAX_RANK	Output tensor

Compile Time Constant Status:

Argument	CTC enabled profile(s)	CTC disabled extension(s)
output	PRO-INT, PRO-FP	

Supported Data Types:

Profile/Extension	Mode	out_t
PRO-FP	fp16	fp16_t
PRO-FP	fp32	fp32_t
PRO-INT or PRO-FP	16-bit	i16_t
PRO-INT or PRO-FP	32-bit	i32_t
PRO-INT or PRO-FP	8-bit	i8_t
PRO-INT or PRO-FP	Boolean	bool_t
EXT-BF16	bf16	bf16_t
EXT-FP8E4M3	fp8e4m3	fp8e4m3_t
EXT-FP8E5M2	fp8e5m2	fp8e5m2_t
EXT-INT16	48-bit	i48_t
EXT-INT4	4-bit	i4_t

Operation Function:

```
output = values;
```

2.14.2. IDENTITY

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

Precision Requirements

Integer results must be exact.

For floating-point values, the following rules apply:

- Subnormal bf16_t, fp16_t, and fp32_t input values may be flushed to zero.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_out_t>	input1	shape	0 to MAX_RANK	Input tensor
Output	T<in_out_t>	output	shape	0 to MAX_RANK	Output tensor of the same type, size as the input tensor

Supported Data Types:

Profile/Extension	Mode	in_out_t
PRO-FP	fp16	fp16_t
PRO-FP	fp32	fp32_t
PRO-INT or PRO-FP	16-bit	i16_t
PRO-INT or PRO-FP	32-bit	i32_t
PRO-INT or PRO-FP	8-bit	i8_t
PRO-INT or PRO-FP	Boolean	bool_t
EXT-BF16	bf16	bf16_t
EXT-FP8E4M3	fp8e4m3	fp8e4m3_t
EXT-FP8E5M2	fp8e5m2	fp8e5m2_t
EXT-INT16	48-bit	i48_t
EXT-INT4	4-bit	i4_t

Operation Function:

```
output = input1;
```

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

Runs an implementation defined custom operator. CUSTOM operators are not tested in the conformance suite as results will be implementation defined. The `domain_name` attribute should be unique to each implementation. To achieve this, using a domain name as the `domain_name` attribute is recommended. No conformance testing is done for CUSTOM operators as they are implementation dependent.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	tensor_list_t	input_list	-		List of input tensors
Attribute	String	operator_name	-		String which tells the backend which custom operator is being called
Attribute	String	domain_name	-		String identifier which can help avoid name collisions on the operator field. Different implementations of a given operator would be in different domains. Implementations can choose which domains they want to support.
Attribute	String	implementation_attrs	-		String value containing implementation specific attributes which apply to the operation
Output	tensor_list_t	output_list	-		List of output tensors

Supported Data Types:

Profile/Extension	Mode
PRO-INT or PRO-FP	All

Operation Function:

```
LEVEL_CHECK(tensor_list_shape(input_list) <= MAX_TENSOR_LIST_SIZE);
LEVEL_CHECK(tensor_list_shape(output_list) <= MAX_TENSOR_LIST_SIZE);
```

```
// Implementation defined behavior
```

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	Shape	Rank	Description
Input	T<bool_t>	condition	shape	0 to MAX_RANK	Input condition as a size 1 tensor
Input	tensor_list_t	input_list	-		List of input tensors
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

Supported Data Types:

Profile/Extension	Mode
EXT-CONTROLFLOW	All

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
LEVEL_CHECK(tensor_list_shape(input_list) <= MAX_TENSOR_LIST_SIZE);
LEVEL_CHECK(tensor_list_shape(output_list) <= MAX_TENSOR_LIST_SIZE);
```

```
ERROR_IF(tosa_nesting_depth >= MAX_NESTING);
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));
ERROR_IF(tensor_size(shape) != 1);

tosa_nesting_depth++;
if (condition[0]) {
    tosa_execute_graph(then_graph, input_list, output_list);
} else {
    tosa_execute_graph(else_graph, input_list, output_list);
}
tosa_nesting_depth--;
```

2.16.2. WHILE_LOOP

Generates and evaluates a Boolean 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	Shape	Rank	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

Supported Data Types:

Profile/Extension	Mode
EXT-CONTROLFLOW	All

Operation Function:

```
LEVEL_CHECK(tensor_list_shape(input_list) <= MAX_TENSOR_LIST_SIZE);
LEVEL_CHECK(tensor_list_shape(output_list) <= MAX_TENSOR_LIST_SIZE);
```

```
ERROR_IF(tosa_nesting_depth >= MAX_NESTING);
ERROR_IF(tensor_list_shape(input_list) != tensor_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));
// Condition graph output must be a single element tensor with a single bool value
ERROR_IF(tensor_size(tosa_output_shape(cond_graph)) != 1);
ERROR_IF(tosa_output_type(cond_graph) != 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
tensor_list_t list[];    // array of tensor lists indexed by iteration
bool_t *condition[];     // array of condition tensors indexed by iteration
list[i] = input_list;    // copy input data as list[0]
tosa_nesting_depth++;
tosa_execute_graph(cond_graph, list[i], [ condition[i] ]);   // initial condition
while (condition[i][0]) {
    tosa_execute_graph(body_graph, list[i], list[i+1]);
    i = i+1;
    tosa_execute_graph(cond_graph, list[i], [ condition[i] ]);
}
tosa_nesting_depth--;
```

```
output_list = list[i];
```

2.17. Variable Operators

TOSA implements three variable operators for expressing persistent mutable values across multiple TOSA graph invocations.

2.17.1. VARIABLE

Defines a new TOSA variable. This is a persistent mutable value across multiple TOSA graph invocations. Modifications are expressed using read/write semantics.

Precision Requirements

Results must be exact.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Attribute	String	name	-		Globally unique identifier for the declared variable tensor.
Attribute	T<tensor_size_t>	var_shape	var_shape	1	The variable tensor shape
Attribute	var_t	type	-		Type of the tensor variable elements.
Attribute	T<in_t>	initial_value	shape	0 to MAX_RANK	Initial value of the variable tensor. This argument is optional with default value NULL.

Supported Data Types:

Profile/Extension	Mode	in_t
EXT-VARIABLE and PRO-FP	fp16	fp16_t
EXT-VARIABLE and PRO-FP	fp32	fp32_t
EXT-VARIABLE and PRO-INT	signed 8	i8_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

```
tensor_t var_tensor = variable_tensor_lookup(name);

// Invocation for the first time
if (var_tensor == NULL) {
```

```
  // Allocate the persistent mutable memory for the variable tensor
  var_tensor = variable_tensor_allocate<var_t>(var_shape, name);

  if (initial_value != NULL) {
    REQUIRE(var_t == in_t);
    REQUIRE(var_shape == shape);
    for_each_data_position (index in shape) {
      // Copy data from initial_value to var_tensor
      in_t value = tensor_read<in_t>(initial_value, shape, index);
      tensor_write<in_t>(var_tensor.data, var_shape, index, value);
    }
    var_tensor.is_written = true;
  }
} else { // Variable tensor has already been declared
  // It's invalid to declare the second variable with the same name in a single graph
execution,
  REQUIRE(!var_tensor.seen);
}

var_tensor.seen = true;
```

2.17.2. VARIABLE_WRITE

Assigns a value to the pseudo-buffer resource holding a persistent mutable tensor.

Precision Requirements

Results must be exact.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Input	T<in_t>	input1	shape	0 to MAX_RANK	Input tensor
Attribute	String	name	-		Globally unique identifier of the variable tensor that is writing to

Supported Data Types:

Profile/Extension	Mode	in_t
EXT-VARIABLE and PRO-FP	fp16	fp16_t
EXT-VARIABLE and PRO-FP	fp32	fp32_t
EXT-VARIABLE and PRO-INT	signed 8	i8_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

```
tensor_t variable_tensor = variable_tensor_lookup(uid);
// Check this variable tensor has been declared
REQUIRE(variable_tensor);
// The tensor has to be seen before to be written to
// The seen variable is cleared before each graph execution and set in declaration
REQUIRE(variable_tensor.seen);
//  Input tensor's shape and variable_tensor's shape have to match
REQUIRE(variable_tensor.shape == shape);
//  Input tensor's shape and variable_tensor's type have to match
REQUIRE(is_same<variable_tensor.type,in_t>());

for_each_data_position (index in shape) {
    // Write data from the input to the pseudo-buffer resource
    in_t value = tensor_read<in_t>(input1, shape, index);
    tensor_write<tensor_t>(variable_tensor.data, variable_tensor.shape, index, value);
}

variable_tensor.is_written = true;
```

2.17.3. VARIABLE_READ

Reads the value from a pseudo-buffer resource holding a persistent mutable tensor.

Precision Requirements

Results must be exact.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Attribute	String	name	-		Globally unique identifier of the variable tensor that is reading from
Output	T<out_t>	output1	shape	0 to MAX_RANK	Output tensor

Supported Data Types:

Profile/Extension	Mode	out_t
EXT-VARIABLE and PRO-FP	fp16	fp16_t
EXT-VARIABLE and PRO-FP	fp32	fp32_t
EXT-VARIABLE and PRO-INT	signed 8	i8_t

Operation Function:

```
LEVEL_CHECK(rank(shape) <= MAX_RANK);
```

```
tensor_t variable_tensor = variable_tensor_lookup(name);
// Check this variable tensor has been decalred
REQUIRE(variable_tensor != NULL);
// Check this variable tensor has been written
REQUIRE(variable_tensor.is_written);
// Output tensor's shape and variable_tensor's shape have to match
REQUIRE(variable_tensor.shape == shape);
//  Output tensor's shape and variable_tensor's type have to match
REQUIRE(is_same<variable_tensor.type,out_t>());

for_each_data_position (index in shape) {
    // Read data from pseudo-buffer resource to the output
    out_t value = tensor_read<tensor_t>(variable_tensor.data, variable_tensor.shape,
index);
    tensor_write<out_t>(output1, shape, index, value);
}
```

2.18. Shape Operators

The shape operators are operators which describe the shapes of parameters and the corresponding transformations.

Having separate shape operations allows easier tracking of shape propagation than would be possible by using the existing TOSA operators.

2.18.1. CONST_SHAPE

A node containing a constant shape.

Arguments:

Argument	Type	Name	Shape	Rank	Description
Attribute	shape_t<>	values	-		Constant shape
Output	shape_t<>	output	-		Output shape

Compile Time Constant Status:

Argument	CTC enabled profile(s)	CTC disabled extension(s)
output	PRO-INT, PRO-FP	

Supported Data Types:

Profile/Extension	Mode
PRO-INT or PRO-FP	shape

Operation Function:

```
output = values;
```

3. Enumerations

Where enumerated types are specified for an operator, the provided value must be a valid enumerant for that type. The included tables provide reference values for the enumerations. Implementations do not need to use these values, they may substitute other values as long as they are functionally equivalent. If no entry is listed in 'Required Extension' then the enumeration is always available.

3.1. resize_mode_t

Valid resize types

Name	Value	Description	Required Extension
NEAREST_NEIGHBOR	1	Nearest neighbor resize	
BILINEAR	2	Bilinear resize	

3.2. acc_type_t

Allowed accumulator types

Name	Value	Description	Required Extension
INT32	1	32-bit integer	
FP16	2	16-bit floating-point	
FP32	3	32-bit floating-point	
INT48	4	48-bit integer	

3.3. var_t

Variable tensor data type

Name	Value	Description	Required Extension
BOOLEAN	1	Boolean	
INT8	2	8-bit integer	
INT16	3	16-bit integer	
INT32	4	32-bit integer	
FP16	5	16-bit floating-point	

Name	Value	Description	Required Extension
BF16	6	16-bit brain floating-point	
FP32	7	32-bit floating-point	

3.4. nan_propagation_mode_t

NaN propagation policy

Name	Value	Description	Required Extension
PROPAGATE	1	NaN is returned when the operation has a NaN	
IGNORE	2	NaN is ignored when the operation has a NaN. NaN is produced if and only if all operands are NaN	

3.5. rounding_mode_t

Rounding mode

Name	Value	Description	Required Extension
SINGLE_ROUND	1	Perform single rounding.	
INEXACT_ROUND	2	Allow rounding results to be inexact.	EXT-INEXACTROUND
DOUBLE_ROUND	3	Perform double rounding.	EXT-DOUBLEROUND

4. 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.

4.1. for_each

The TOSA pseudocode uses the `for_each` loop to describe iterating over a multidimensional range. `for_each` is specified as:

```
for_each(Amin <= a < Amax, Bmin <= b < Bmax,  ...) {
    // body statements
}
```

The body of the `for_each` is executed for every possible combination of the iteration variables in the condition. The variables `a` and `b` are defined within the scope of the `for_each` body.

4.2. for_each_data_position

The TOSA pseudocode uses the `for_each_data_position` to execute a body over each data position in a shape. If the shape is empty, the body is executed a single time with the index being the empty list

```
// shape must be of type shape_t
for_each_data_position(index in shape) {
    // body statements
}
```

is executed as

```
if (shape == []) {
    // Execute body statements with index == []
}
else {
    // Execute body statements with index iterating over all locations in the shape
}
```

4.3. 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 satisfies 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;
    }
}

void LEVEL_CHECK(condition) {
    // If a level is specified and the level condition fails then
    // the result is unpredictable.
    REQUIRE(condition);
}
```

4.4. Tensor Access Helpers

4.4.1. Tensor Utilities

```
// Convert tensor index coordinates to an element offset
tensor_size_t tensor_index_to_offset(shape_t shape, shape_t index) {
    tensor_size_t size = tensor_size(shape);  // check tensor shape is valid
    tensor_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;
}

// Convert an element offset to tensor index coordinates
shape_t tensor_offset_to_index(shape_t shape, tensor_size_t offset) {
    tensor_size_t size = tensor_size(shape);  // check tensor shape is valid
    REQUIRE(offset < size);
```

```
    REQUIRE(offset >= 0);
    shape_t index(rank(shape));    // index has rank(shape) indicies
    for(int32_t i = rank(shape) - 1; i >= 0; i--) {
        index[i] = offset % shape[i];
        offset /= shape[i];
    }
    return index;
}

// Check the tensor shape is valid and return the tensor size in elements
tensor_size_t tensor_size(shape_t shape) {
    tensor_size_t size = 1;
    for (int32_t i = 0; i < rank(shape); i++) {
        REQUIRE(1 <= shape[i] && shape[i] <= maximum<tensor_size_t> / size);
        size *= shape[i];
    }
    return size;
}

// Return the size of the tensor in the given axis
// For a rank=0 tensor, returns 1 for all axes
tensor_size_t shape_dim(shape_t shape, int axis) {
    return (axis >= rank(shape)) ? 1 : shape[axis];
}
```

4.4.2. 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, shape_t shape, shape_t index) {
    tensor_size_t offset = tensor_index_to_offset(shape, index);
    return address[offset];
}
```

4.4.3. 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, shape_t shape, shape_t index, <type> value) {
    tensor_size_t offset = tensor_index_to_offset(shape, index);
    address[offset] = value;
}
```

4.4.4. Variable Tensor Allocate

variable_tensor_allocate allocates the mutable persistent memory block for storing variable tensors. The shape argument contains the shape of the allocated memory block for the variable_tensor. The 'name' argument is a globally unique identifier for variable tensors.

```
tensor_t* variable_tensor_allocate<in_t>(shape_t shape, String name) {
    tensor_size_t size = tensor_size(shape);
    tensor_t *allocated_tensor = new tensor_t;
    allocated_tensor->data = new in_t[size];
    allocated_tensor->name = name;
    allocated_tensor->is_written = false;
    allocated_tensor->shape = shape;
    allocated_tensor->type = in_t;
    return allocated_tensor;
}
```

4.4.5. Variable Tensor Lookup

variable_tensor_lookup checks whether a variable tensor has been allocated or not. The 'name' argument is a globally unique identifier for variable tensors.

```
tensor_t variable_tensor_lookup(String name) {
    // The global all_allocated_variable_tensors was instantiated at the first
    // time of executing the tosa graph
    for_each(tensor_t allocated_tensor in all_allocated_variable_tensors) {
        if (allocated_tensor.name == name) {
            return allocated_tensor;
        }
    }
    return NULL;
}
```

4.4.6. Broadcast Helpers

The following function derives the broadcast output shape from the input shapes.

```
shape_t broadcast_shape(shape_t shape1, shape_t shape2) {
    ERROR_IF(rank(shape1) != rank(shape2));
    shape_t shape = shape1;
    for (int32_t i = 0; i < rank(shape); i++) {
        if (shape[i] == 1) {
            shape[i] = shape2[i];
        } else {
            ERROR_IF(shape2[i] != 1 && shape2[i] != shape[i]);
        }
    }
```

```
    return shape;
}
```

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.

shape_t apply_broadcast(shape_t out_shape, shape_t in_shape, shape_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;
}
```

4.5. General Pseudocode Helpers

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

4.5.1. Arithmetic Helpers

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

```
in_t apply_add_s<in_t>(in_t a, in_t b) {
    if (is_floating_point<in_t>()) return a + b;
    int64_t c = sign_extend<int64_t>(a) + sign_extend<int64_t>(b);
    REQUIRE(c >= minimum_s<in_t>() && c <= maximum_s<in_t>());
    return static_cast<in_t>(c);
}

in_t apply_add_u<in_t>(in_t a, in_t b) {
    if (is_floating_point<in_t>()) return a + b;
    uint64_t c = zero_extend<uint64_t>(a) + zero_extend<uint64_t>(b);
    REQUIRE(c >= minimum_u<in_t>() && c <= maximum_u<in_t>());
    return truncate<in_t>(c);
}

in_t apply_arith_rshift<in_t>(in_t a, in_t b) {
    int32_t c = sign_extend<int32_t>(a) >> sign_extend<int32_t>(b);
    return static_cast<in_t>(c);
}
```

```
in_t apply_intdiv_s<in_t>(in_t a, in_t b) {
    int64_t c = sign_extend<int64_t>(a) / sign_extend<int64_t>(b);
    REQUIRE(c >= minimum_s<in_t>() && c <= maximum_s<in_t>());
    return static_cast<in_t>(c);
}

// return input value rounded up to nearest integer
in_t apply_ceil<in_t>(in_t input);

// return e to the power input
in_t apply_exp<in_t>(in_t input);

// return input value rounded down to nearest integer
in_t apply_floor<in_t>(in_t input);

// return the natural logarithm of input
in_t apply_log_positive_input<in_t>(in_t input);

in_t apply_log<in_t>(in_t input) {
    if (input == 0) {
        return -INFINITY;
    }
    else if (input < 0) {
        return NaN;
    }
    return apply_log_positive_input(input);
}

in_t apply_logical_rshift<in_t>(in_t a, in_t b) {
    uint64_t c = zero_extend<uint32_t>(a) >> zero_extend<uint32_t>(b);
    return static_cast<in_t>(c);
}

in_t compare_nan<in_t>(in_t a, in_t b, nan_propagation_t nan_mode) {
    REQUIRE(isNaN(a) || isNaN(b));

    if (nan_mode == PROPAGATE) {
          return NaN;
    }

    // Non NaN Propagation
    return isNaN(a) ? b : a;
}

in_t apply_max_s<in_t>(in_t a, in_t b, nan_propagation_t nan_mode=PROPAGATE) {
    if (is_floating_point<in_t>()) {
        if (isNaN(a) || isNaN(b)) {
            return compare_nan(a, b, nan_mode);
        }
        if (a >= b) return a; else return b;
```

```
    }
    // Integer version
    if (sign_extend<int64_t>(a) >= sign_extend<int64_t>(b)) return a; else return b;
}

in_t apply_max_u<in_t>(in_t a, in_t b) {
    if (zero_extend<uint64_t>(a) >= zero_extend<int64_t>(b)) return a; else return b;
}

in_t apply_min_s<in_t>(in_t a, in_t b, nan_propagation_t nan_mode=PROPAGATE) {
    if (is_floating_point<in_t>()) {
        if (isNaN(a) || isNaN(b)) {
            return compare_nan(a, b, nan_mode);
        }
        if (a < b) return a; else return b;
    }
    // Integer version
    if (sign_extend<int64_t>(a) < sign_extend<int64_t>(b)) return a; else return b;
}

in_t apply_min_u<in_t>(in_t a, in_t b) {
    if (zero_extend<int64_t>(a) < zero_extend<int64_t>(b)) return a; else return b;
}

in_t apply_clip_s<in_t>(in_t value, in_t min_val, in_t max_val, nan_propagation_t
nan_mode=PROPAGATE) {
    if (is_floating_point<in_t>()) {
        REQUIRE(min_val <= max_val);
        REQUIRE(!isNaN(min_val) && !isNaN(max_val));
    }
    else {
        REQUIRE(sign_extend<int64_t>(min_val) <= sign_extend<int64_t>(max_val));
    }
    value = apply_max_s<in_t>(value, min_val, nan_mode);
    value = apply_min_s<in_t>(value, max_val, nan_mode);
    return value;
}

in_t apply_clip_u<in_t>(in_t value, in_t min_val, in_t max_val) {
    REQUIRE(zero_extend<int64_t>(min_val) <= zero_extend<int64_t>(max_val));
    value = apply_max_u<in_t>(value, min_val);
    value = apply_min_u<in_t>(value, max_val);
    return value;
}

in_t apply_mul_s<in_t>(in_t a, in_t b) {
    if (is_floating_point<in_t>()) return a * b;
    int64_t c = sign_extend<int64_t>(a) * sign_extend<int64_t>(b);
    return static_cast<in_t>(c);
}
```

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

// return the square root of input
in_t apply_sqrt<in_t>(in_t input);

in_t apply_sub_s<in_t>(in_t a, in_t b) {
    if (is_floating_point<in_t>()) return a - b;
    int64_t c = sign_extend<int64_t>(a) - sign_extend<int64_t>(b);
    REQUIRE(c >= minimum_s<in_t>() && c <= maximum_s<in_t>());
    return static_cast<in_t>(c);
}

in_t apply_sub_u<in_t>(in_t a, in_t b) {
    uint64_t c = zero_extend<uint64_t>(a) - zero_extend<uint64_t>(b);
    REQUIRE(c >= minimum_u<in_t>() && c <= maximum_u<in_t>());
    return truncate<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;
}
```

4.5.2. Type Conversion Helpers

The following definitions indicate the type to be used when the given parameters are provided.

```
// Returns a signed version of the given type
// A no-op for floating-point types
Type make_signed(Type in_t)
{
    if (is_floating_point<in_t>()) {
        return in_t;
    }
    if (is_same<in_t,bool_t>()) {
        return bool_t;
    } else if (is_same<in_t,i8_t>()) {
        return int8_t;
    } else if (is_same<in_t,i16_t>()) {
```

```
        return int16_t;
    } else if (is_same<in_t,i32_t>()) {
        return int32_t;
    } else if (is_same<in_t,i48_t>()) {
        return int48_t;
    }
}

// Returns the usigned type of the given type
// Error to call this with anything but i8_t or i16_t

Type make_unsigned(Type in_t)
{
    ERROR_IF(!is_same<in_t,i8_t>() && !is_same<in_t,i16_t>());
    if (is_same<in_t,i8_t>()) {
        return uint8_t;
    } else if (is_same<in_t,i16_t>()) {
        return uint16_t;
    }
}

out_t static_cast<out_t>(in_t value)
{
    // Operates similar to the c++ standard static_cast
    // Limited to simple numeric conversion for TOSA.
    // Sign extends signed integer input types if needed
    // Zero extends unsigned integer input types if needed
    // Truncates when converting to a smaller width data type
    // Conversion from integer to floating-point is exact if possible
    // If converting between signless integer types, treated as signed integer
}

out_t bitcast<out_t>(in_t value)
{
    // Treats the bits of value as if they were of type out_t
    // Only supported for integer types of the same bit width
}
```

4.5.3. Numeric Accuracy Helpers

For a floating point number of type in_t a normal value is of the form (1.x * 2^e). The fractional part 'x' has a number of fractional or mantissa bits depending on the type. The exponent 'e' has a normal range depending on the type. The functions below return the ranges according to type.

```
fp64_t exp2(int n) {
    if (n < -1075) {
        return 0.0; // smaller than smallest denormal
    }
    REQUIRE(n <= 1023);
```

```
    fp64_t v = 1.0;
    while (n > 0) { v = v*2.0; n--; }
    while (n < 0) { v = v/2.0; n++; }
    return v;
}

int ilog2(fp64_t v) {
    REQUIRE(0 < v && v < infinity);
    int n = 0;
    while (v >= 2.0) { v = v/2.0; n++; }
    while (v <  1.0) { v = v*2.0; n--; }
    return n;
}

fp64_t normal_min<in_t>() {
  if (is_same<in_t,fp32_t>()) {
    return exp2(-126);
  } else if (is_same<in_t,bf16_t>()) {
    return exp2(-126);
  } else if (is_same<in_t,fp16_t>()) {
    return exp2( -14);
  } else if (is_same<in_t,fp8e4m3_t>()) {
    return exp2(-6);
  } else if (is_same<in_t,fp8e5m2_t>()) {
    return exp2(-14);
  }
}

fp64_t normal_max<in_t>() {
  if (is_same<in_t,fp32_t>()) {
    return exp2(128) - exp2(127-23);
  } else if (is_same<in_t,bf16_t>()) {
    return exp2(128) - exp2(127- 7);
  } else if (is_same<in_t,fp16_t>()) {
    return exp2( 16) - exp2( 15-10);
  } else if (is_same<in_t,fp8e4m3_t>()) {
    return exp2( 9) - exp2( 8-2);
  } else if (is_same<in_t,fp8e5m2_t>()) {
    return exp2( 16) - exp2( 15-2);
  }
}

// Number of fractional (mantissa bits)
int normal_frac<in_t> () {
  if (is_same<in_t,fp32_t>()) {
    return 23;
  } else if (is_same<in_t,bf16_t>()) {
    return  7;
  } else if (is_same<in_t,fp16_t>()) {
    return 10;
  } else if (is_same<in_t,fp8e4m3_t>()) {
```

```
    return 3;
  } else if (is_same<in_t,fp8e5m2_t>()) {
    return 2;
  }
}

fp64_t calcAbsErrorBound<in_t>(fp64_t bound_magnitude, fp64_t bounds_value,
                               fp64_t lower_bound, fp64_t normal_divisor) {
    fp64_t error_bound = 0.0;
    // Avoid cases where we generate an error_bound of NaN by multiplying inf * 0
    if (is_finite(bounds_value) || abs(bound_magnitude) != 0.0) {
      fp64_t value_bound = max(abs(bound_magnitude), normal_min<in_t>());
      if (lower_bound > 0) {
        value_bound = max(lower_bound / bounds_value, value_bound);
      }
      error_bound = exp2(-normal_frac<in_t>() / normal_divisor) * value_bound;
      error_bound = error_bound * bounds_value;
    }
    return error_bound;
}
```

The following functions check if a test value in floating-point format in_t is within an error range compared to a reference value. The functions assume that subnormal values for bf16, fp16, and fp32 may be flushed to zero. For the first function, the permitted error range is specified as num_ulp which is converted to an error bound as specified by the code. For the second function, the permitted error range is specified as an absolute error bound.

```
bool_t tosa_reference_check_fp_bnd<in_t>(in_t test_value, fp64_t ref_value, fp64_t
err_bnd) {
  if (isNaN(ref_value)) {
    // If the reference value is a NaN, the test value must also be any NaN.
    return isNaN(test_value);
  }
  if (!is_finite(err_bnd)) {
    return true;
  }
  REQUIRE(err_bnd >= 0.0);
  if (ref_value < 0) {
    ref_value  = -ref_value;
    test_value = -test_value;
  }

  fp64_t ref_max = ref_value + err_bnd;
  fp64_t ref_min = ref_value - err_bnd;

  if (ref_max > normal_max<in_t>()) ref_max = infinity;
  if (ref_min > normal_max<in_t>()) ref_min = infinity;
  if (ref_min < -normal_max<in_t>()) ref_min = -infinity;

  if (is_same<in_t,bf16_t>() || is_same<in_t,fp16_t>() || is_same<in_t,fp32_t>()) {
```

```
    // Allow subnormal values to be flushed to zero for non-fp8
    if (test_value == 0) return (ref_min < normal_min());
  }

  if (is_same<in_t,fp8e4m3_t>() && isNaN(test_value)) {
    // The case where ref is NaN is handled at the beginning of the function
    // The following check is enough because `abs(ref_max) >= abs(ref_min)` and
    // `ref_max >= 0`.
    return ref_max == infinity;
  }

  // Overflow/subnormals have been handled, can do a standard check
  // at this point.
  return (static_cast<fp64_t>(test_value) >= ref_min &&
          static_cast<fp64_t>(test_value) <= ref_max);
}

fp64_t tosa_reference_ulp<in_t>(fp64_t ref_value) {
  if (is_normal_fp64(ref_value) && abs(ref_value) != 0) {
    int ref_exp = ilog2(abs(ref_value));
    fp64_t ref_pow2 = max(exp2(ref_exp), normal_min<in_t>());
    return ref_pow2 * exp2(-normal_frac<in_t>());
  }
  return 0.0;
}

bool_t tosa_reference_check_fp<in_t>(in_t test_value, fp64_t ref_value, fp64_t
num_ulp) {
  fp64_t err_bnd = tosa_reference_ulp<in_t>(ref_value) * num_ulp;
  return tosa_reference_check_fp_bnd<in_t>(test_value, ref_value, err_bnd);
}
```

4.5.4. 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.

```
// Converts the floating-point value to an integer value using round to nearest
rounding.
int round_to_nearest_int(float_t f);

// Converts the input value into floating-point, using round to nearest rounding.
// Values that are not NaN outside of the representable range of the destination type
must be set to infinity of the correct sign.
// If the destination floating point type does not have an infinity representation,
values outside of the representable range must be set to NaN.
float_t round_to_nearest_float(in_t f);
```

```
// Floating point values are unchanged.
// For two's complement integer values where out_t has more bits than in_t, replicate
the top bit of input for all bits between the top bit of input and the top bit of
output.
out_t sign_extend<out_t>(in_t input);

// Floating point values are unchanged.
// For two's complement integer values where out_t has more bits than in_t, insert
zero values for all bits between the top bit of input and the top bit of output.
out_t zero_extend<out_t>(in_t input);

// output is the sizeof(out_t) least significant bits in input.
// Nop for floating-point types
out_t truncate(in_t input);
```

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

```
shape_t flatten(shape_t shapes[]) {
    shape_t output = [];
    for_each(shape in shapes) {
        for_each(element in shapes) {
            output.append(element);
        }
    }
}
```

Generic helper functions used to keep the pseudocode concise.

```
bool_t is_floating_point<type>() {
    if (is_same<type,fp16_t>() || is_same<type,fp32_t>() || is_same<type,bf16_t>() ||
is_same<type,fp8e4m3_t>() || is_same<type,fp8e5m2_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
}

// perform an integer division with rounding towards minus infinity
```

```
int32_t idiv_floor(int32_t input1, int32_t input2) {
    int32_t rval = input1 / input2;
    if (rval * input2 > input1) {
        rval--;
    }
    return rval;
}

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

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

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

// returns value of pi
float_t pi();

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

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

// return true if value is a power of two, false otherwise
bool_t power_of_two(int32_t value);

// return the maximum value when interpreting type in_out_t as a signed value as
returned by the make_signed helper.
in_out_t maximum_s<in_out_t>();

// return the minimum value when interpreting type in_out_t as a signed value as
returned by the make_signed helper.
in_out_t minimum_s<in_out_t>();

// return the maximum value when interpreting type in_out_t as an unsigned value as
returned by the make_unsigned helper.
in_out_t maximum_u<in_out_t>();

// return the minimum value when interpreting type in_out_t as an unsigned value as
returned by the make_unsigned helper.
in_out_t minimum_u<in_out_t>();

// return true if the given value is a NaN. Only valid for floating-point types
bool_t isNaN(in_t value);

// return true if the given value is an Infinity. Only valid for floating-point types
bool_t isInf(in_t input);
```

```
// return true if value is a normal fp64 value (Not zero, subnormal, infinite or NaN)
bool_t is_normal_fp64(fp64_t value);
```

4.5.5. Scaling Helpers

Helper functions used to scale between different integer domains

```
int32_t apply_scale_32(int32_t value, int32_t multiplier, int8_t shift, bool_t
double_round=false) {
    REQUIRE(multiplier >= 0);
    REQUIRE(2 <= shift && shift <= 62);
    REQUIRE(value >= (-1 << (shift - 1)) && value < (1 << (shift - 1)));
    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 = (static_cast<int64_t>(value) * multiplier) + round;
    result >>= shift;
    // result will fit a 32-bit range due to the REQUIRE on value
    return static_cast<int32_t>(result);
}

int32_t apply_scale_16(int48_t value, int16_t multiplier, int8_t shift) {
    REQUIRE(multiplier >= 0);
    REQUIRE(2 <= shift && shift <= 62);
    int64_t round = 1 << (shift - 1);
    int64_t result = (static_cast<int64_t>(value) * multiplier) + round;
    result >>= shift;
    REQUIRE(result >= minimum<int32_t> && result <= maximum<int32_t>);
    return static_cast<int32_t>(result);
}

// Struct which describes the scale factors
typedef struct {
    int32_t multiplier;
    int8_t shift;
} scale_t;

// Calculate an appropriate scale factor to use when a divide is required
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;
```

```
}
```

5. Appendix A

5.1. Random Data Generation

The following function generates a pseudo-random floating-point value in the range -1.0 to +1.0 for use as test data. It uses a modulo (1<<32) recurrent sequence with multiplier derived from "TOSASETS" and the set number.

```
float set_data(uint32_t set, uint32_t index)
{
    uint32_t m = (8*set + 1) * 0x705A5E75;   // mod (1<<32) calculation
    uint32_t r = m + 1;                      // mod (1<<32) calculation
    for (uint32_t i = 0; i < index; i++) {
        r = r * m + 1;                       // mod (1<<32) calculation
    }
    float  sign = (r>>31)==0 ? +1 : -1;
    return sign * (float)(r & 0x7FFFFFFF) / (float)(0x7FFFFFFF);
}
```

5.2. Floating-Point Test Data Generator

This section describes the function tosa_pro_fp_data(S, KS, p, k, i) that generates test data for floating-point profile compliance. This function takes the following arguments:

- S is the test set number which identifies which generator is used
- KS is the kernel size
- p is the parameter number of:
  - 0 for the first input (usually data)
  - 1 for the second input (usually weights)
  - 2 for the third input if present (usually bias)
- k is the index within the kernel in the range 0 <= k < KS
- i is the index within the tensor to write

Some test data values are scaled by the bound parameter B which is defined in the table below. B is set to be the largest value that is both representable by the input type and such that B*B does not overflow the output precision. In the case of mixed input types, B is the largest value that is representable by both input types such that B*B does not overflow the output precision.

inputs type	output type	B value
fp8e4m3	fp16	(1<<8) - (1<<4) = 240

fp8e4m3	fp32	448
fp8e5m2	fp16	(1<<8) - (1<<5) = 224
fp8e5m2	fp32	57344
fp8e4m3 and fp8e5m2	fp16	224
fp8e4m3 and fp8e5m2	fp32	448
fp8e4m3	fp8e4m3	20
fp8e5m2	fp8e5m2	224
fp16	fp16	(1<<8) - (1/8) = 255.875
fp16	fp32	(1<<16) - (1<<5) = 65504
bf16	bf16	(1<<64) - (1<<56)
bf16	fp32	(1<<64) - (1<<56)
fp32	fp32	(1<<64) - (1<<40)

5.2.1. Test Set S=0 Generator

The aim of this generator is to check that sum of products with zero gives zero result.

p	tosa_pro_fp_data(S, KS, p, k, i) =
0	set_data(3*S, i) < 0 ? 0.0 : set_data(3*S+1, i)
1	set_data(3*S, i) < 0 ? set_data(3*S+1, i) : 0.0
2	0.0

5.2.2. Test Set S=1

The aim of this test set is to check values with large exponents.

p	tosa_pro_fp_data(S, KS, p, k, i) =
0	(B/sqrt(KS+1))*((set_data(3*S+0, i*2) < 0 ? -0.75 : 0.75) + 0.25*set_data(3*S+0, 2*i+1))
1	(B/sqrt(KS+1))*((set_data(3*S+1, i*2) < 0 ? -0.75 : 0.75) + 0.25*set_data(3*S+1, 2*i+1))
2	(B*B/(KS+1))*((set_data(3*S+2, i*2) < 0 ? -0.75 : 0.75) + 0.25*set_data(3*S+2, 2*i+1))

5.2.3. Test Set S=2

The aim of this test set is to check rounding error when accumulating small values onto a large value. In this case the small values are of similar magnitude. If the implementation changes the order of the sum, then the test data must also be reordered so that the largest values occur first in the sum.

p	tosa_pro_fp_data(S, KS, p, k, i) =
0	(k==0) ? 1.0 : set_data(3*S+0, i)/sqrt(KS)

1	(k==0) ? 1.0 : set_data(3*S+1, i)/sqrt(KS)
2	0.0

5.2.4. Test Set S=3

The aim of this test set is to check rounding error when accumulating small values onto a large value. In this case the small values are of varying magnitude. If the implementation changes the order of the sum, then the test data must also be reordered so that the largest values occur first in the sum.

p	tosa_pro_fp_data(S, KS, p, k, i) =
0	(k==0) ? ((set_data(3*S+0, 2*i+0) < 0) ? -16.0 : 16.0) : exp(2*set_data(3*S+0, 2*i+0)) * set_data(3*S+0, 2*i+1)
1	(k==0) ? ((set_data(3*S+1, 2*i+0) < 0) ? -16.0 : 16.0) : exp(2*set_data(3*S+1, 2*i+0)) * set_data(3*S+1, 2*i+1)
2	0.0

5.2.5. Test Set S=4

The aim of this test set is to check a mixture of zero and non-zero products.

p	tosa_pro_fp_data(S, KS, p, k, i) =
0	(k==KS/2) ? (set_data(3*S, i) < 0 ? -0.5 : +0.5) : (set_data(3*S, i) < 0 ? 0.0 : (B/sqrt(KS))*set_data(3*S+1, i))
1	(k==KS/2) ? (set_data(3*S, i) < 0 ? +0.5 : -0.5) : (set_data(3*S, i) < 0 ? (B/sqrt(KS))*set_data(3*S+1, i) : 0.0)
2	0.0

5.2.6. Test Set S=5

The aim of this test set is to check signed inputs of large range.

p	tosa_pro_fp_data(S, KS, p, k, i) =
0	(B/sqrt(KS))*set_data(3*S+0, i)
1	(B/sqrt(KS))*set_data(3*S+1, i)
2	0.0

5.3. Floating-Point Operator Test Data

For each operator, this section defines how to generate test data for test set S. For the results to be statistically significant the operation must calculate at least MIN_DOT_PRODUCTS dot products. For most operations this means that the output tensor must have at least MIN_DOT_PRODUCTS output values. For most operations batch size can be increased if necessary so that this holds. For this

version of the specification, MIN_DOT_PRODUCTS is set to 1000.

5.3.1. CONV2D

The following generates input test data for test set S. For compliant implementation, the test must pass whenever the attributes satisfy: `N*OH*OW*OC >= MIN_DOT_PRODUCTS`

```
KS = KW*KH*IC;
for (0 <= n < N, 0 <= iy < IH, 0 <= ix < IW, 0 <= ic < IC) {
  input [ n, iy, ix, ic] = tosa_pro_fp_data(S, KS, 0, ((iy % KH)*KW+(ix % KW))*IC+ic,
((n*IH+iy)*IW+ix)*IC+ic);
}
for (0 <= oc < OC, 0 <= ky < KH, 0 <= kx < KW, 0 <= ic < IC) {
  weight[oc, ky, kx, ic] = tosa_pro_fp_data(S, KS, 1, (ky*KW+kx)*IC+ic,
((oc*KH+ky)*KW+kx)*IC+ic);
}
for (0 <= oc < BC) {
  bias[oc] = tosa_pro_fp_data(S, KS, 2, oc)
}
```

5.3.2. CONV3D

The following generates input test data for test set S. For compliant implementation, the test must pass whenever the attributes satisfy: `N*OD*OH*OW*OC >= MIN_DOT_PRODUCTS`

```
KS = KD*KW*KH*IC;
for (0 <= n < N, 0 <= id < UD, 0 <= iy < IH, 0 <= ix < IW, 0 <= ic < IC) {
  input [ n, id, iy, ix, ic] = tosa_pro_fp_data(S, KS, 0, (((id % KD)*KH+(iy %
KH))*KW+(ix % KW))*IC+ic, (((n*ID+id)*IH+iy)*IW+ix)*IC+ic);
}
for (0 <= oc < OC, 0 <= kd < KD, 0 <= ky < KH, 0 <= kx < KW, 0 <= ic < IC) {
  weight[oc, kd, ky, kx, ic] = tosa_pro_fp_data(S, KS, 1, ((kd*KH+ky)*KW+kx)*IC+ic,
(((oc*KD+kd)*KH+ky)*KW+kx)*IC+ic);
}
for (0 <= oc < BC) {
  bias[oc] = tosa_pro_fp_data(S, KS, 2, oc)
}
```

5.3.3. DEPTHWISE_CONV2D

The following generates input test data for test set S. For compliant implementation, the test must pass whenever the attributes satisfy: `N*OH*OW*C*M >= MIN_DOT_PRODUCTS`

```
KS = KW*KH;
for (0 <= n < N, 0 <= iy < IH, 0 <= ix < IW, 0 <= c < C) {
  input [ n, iy, ix, c] = tosa_pro_fp_data(S, KS, 0, (iy % KH)*KW+(ix % KW),
((n*IH+iy)*IW+ix)*C+c);
```

```
}
for (0 <= ky < KH, 0 <= kx < KW, 0 <= c < C, 0 <= m < M) {
  weight[ky, kx,  c, m] = tosa_pro_fp_data(S, KS, 1, (ky*KW+kx),
((ky*KW+kx)*C+c)*M+m);
}
for (0 <= oc < C*M) {
  bias[oc] = tosa_pro_fp_data(S, KS, 2, oc)
}
```

5.3.4. MATMUL

The following generates input test data for test set S. For compliant implementation, the test must pass whenever the attributes satisfy: `N*H*W >= MIN_DOT_PRODUCTS`

```
KS = C;
for (0 <= n < N, 0 <= y < H, 0 <= c < C) {
  A[n, y, c] = tosa_pro_fp_data(S, KS, 0, c, (n*H+y)*C+c);
}
for (0 <= n < N, 0 <= c < C, 0 <= x < W) {
  B[n, c, x] = tosa_pro_fp_data(S, KS, 1, c, (n*C+c)*W+x);
}
```

5.3.5. TRANSPOSE_CONV2D

The following generates input test data for test set S. For compliant implementation, the test must pass whenever the attributes satisfy: `N*OH*OW*OC >= MIN_DOT_PRODUCTS`

```
KS = KW*KH*IC;
for (0 <= n < N, 0 <= iy < IH, 0 <= ix < IW, 0 <= ic < IC) {
  input [ n, iy, ix, ic] = tosa_pro_fp_data(S, KS, 0, ((iy % KH)*KW+(ix % KW))*IC+ic,
((n*IH+iy)*IW+ix)*IC+ic);
}
for (0 <= oc < OC, 0 <= ky < KH, 0 <= kx < KW, 0 <= ic < IC) {
  weight[oc, ky, kx, ic] = tosa_pro_fp_data(S, KS, 1, (ky*KW+kx)*IC+ic,
((oc*KH+ky)*KW+kx)*IC+ic);
}
for (0 <= oc < BC) {
  bias[oc] = tosa_pro_fp_data(S, KS, 2, oc)
}
```

5.3.6. FFT2D

The following generates input test data for test set S. For compliant implementation, the test must pass whenever the attributes satisfy: `N*H*W >= MIN_DOT_PRODUCTS`

```
KS = 2*H*W;
```

```
for (0 <= n < N, 0 <= y < H, 0 <= x < W) {
  input_real[n, y, x] = tosa_pro_fp_data(S, KS, 0, y*W+x, ((0*N+n)*H+y)*IW+x);
  input_imag[n, y, x] = tosa_pro_fp_data(S, KS, 0, y*W+x, ((1*N+n)*H+y)*IW+x);
}
for (0 <= y < H, 0 <= x < W, 0 <= m < H, 0 <= n < W) {
  weight_real[y, x, m, n] = real(exp(2*pi*i*((m*h/H) + (n*w/W))));
  weight_imag[y, x, m, n] = imag(exp(2*pi*i*((m*h/H) + (n*w/W))));
}
```

5.3.7. RFFT2D

The following generates input test data for test set S. For compliant implementation, the test must pass whenever the attributes satisfy: `N*H*W >= MIN_DOT_PRODUCTS`

```
KS = H*W;
for (0 <= n < N, 0 <= y < H, 0 <= x < W) {
  input_real[n, y, x] = tosa_pro_fp_data(S, KS, 0, y*W+x, ((0*N+n)*H+y)*IW+x);
}
for (0 <= y < H, 0 <= x < W, 0 <= m < H, 0 <= n < W) {
  weight_real[y, x, m, n] = real(exp(2*pi*i*((m*h/H) + (n*w/W))));
  weight_imag[y, x, m, n] = imag(exp(2*pi*i*((m*h/H) + (n*w/W))));
}
```

5.3.8. REDUCE_SUM

The following generates input test data for test set S. For compliant implementation, the test must pass whenever the attributes satisfy: `tensor_size(shape) >= MIN_DOT_PRODUCTS`

```
KS = shape1[axis];
for (index in shape1) {
  input[index] = tosa_pro_fp_data(S, KS, 0, index[axis],
tensor_index_to_offset(index));
}
for (0 <= c < KS) {
  weight[c] = 1;
}
```

5.3.9. AVG_POOL2D

The following generates input test data for test set S. For compliant implementation, the test must pass whenever the attributes satisfy: `N*OH*OW*C >= MIN_DOT_PRODUCTS`

```
KX = kernel_x;
KY = kernel_y;
KS = KX*KY;
for (0 <= n < N, 0 <= iy < IH, 0 <= ix < IW, 0 <= c < C) {
```

```
  input [ n, iy, ix, c] = tosa_pro_fp_data(S, KS, 0, ((iy % KY)*KX+(ix % KX))*C+c,
((n*IH+iy)*IW+ix)*C+c);
}
for (0 <= ky < KY, 0 <= kx < KX, 0 <= c < C, 0 <= m < M) {
  weight[ky, kx] = 1/KS;
}
```

6. Appendix B - Profile operator tables

6.1. Profiles

6.1.1. Integer

Integer operations, primarily 8- and 32-bit values

Status: Complete

Operator	Mode	Version Added
ABS	signed 32	1.0
ADD	signed 32	1.0
ARGMAX	signed 8	1.0
ARITHMETIC_RIGHT_SHIFT	signed 8	1.0
ARITHMETIC_RIGHT_SHIFT	signed 16	1.0
ARITHMETIC_RIGHT_SHIFT	signed 32	1.0
AVG_POOL2D	signed 8 with int32 accumulate	1.0
BITWISE_AND	signed 8	1.0
BITWISE_AND	signed 16	1.0
BITWISE_AND	signed 32	1.0
BITWISE_NOT	signed 8	1.0
BITWISE_NOT	signed 16	1.0
BITWISE_NOT	signed 32	1.0
BITWISE_OR	signed 8	1.0
BITWISE_OR	signed 16	1.0
BITWISE_OR	signed 32	1.0
BITWISE_XOR	signed 8	1.0
BITWISE_XOR	signed 16	1.0
BITWISE_XOR	signed 32	1.0
CAST	bool to signed 8	1.0

Operator	Mode	Version Added
CAST	bool to signed 16	1.0
CAST	bool to signed 32	1.0
CAST	signed 8 to bool	1.0
CAST	signed 8 to signed 16	1.0
CAST	signed 8 to signed 32	1.0
CAST	signed 16 to bool	1.0
CAST	signed 16 to signed 8	1.0
CAST	signed 16 to signed 32	1.0
CAST	signed 32 to bool	1.0
CAST	signed 32 to signed 8	1.0
CAST	signed 32 to signed 16	1.0
CLAMP	signed 8	1.0
CLZ	signed 32	1.0
CONCAT	Boolean	1.0
CONCAT	signed 8	1.0
CONCAT	signed 32	1.0
CONST	Boolean	1.0
CONST	8-bit	1.0
CONST	16-bit	1.0
CONST	32-bit	1.0
CONST_SHAPE	shape	1.0
CONV2D	signed 8x8 with int32 accumulate	1.0
CONV3D	signed 8x8 with int32 accumulate	1.0
CUSTOM	All	1.0
DEPTHWISE_CONV2D	signed 8x8 with int32 accumulate	1.0
EQUAL	signed 32	1.0
GATHER	signed 8	1.0
GATHER	signed 16	1.0
GATHER	signed 32	1.0
GREATER	signed 32	1.0
GREATER_EQUAL	signed 32	1.0

Operator	Mode	Version Added
IDENTITY	Boolean	1.0
IDENTITY	8-bit	1.0
IDENTITY	16-bit	1.0
IDENTITY	32-bit	1.0
INTDIV	signed 32	1.0
LOGICAL_AND	Boolean	1.0
LOGICAL_LEFT_SHIFT	signed 8	1.0
LOGICAL_LEFT_SHIFT	signed 16	1.0
LOGICAL_LEFT_SHIFT	signed 32	1.0
LOGICAL_NOT	Boolean	1.0
LOGICAL_OR	Boolean	1.0
LOGICAL_RIGHT_SHIFT	signed 8	1.0
LOGICAL_RIGHT_SHIFT	signed 16	1.0
LOGICAL_RIGHT_SHIFT	signed 32	1.0
LOGICAL_XOR	Boolean	1.0
MATMUL	signed 8x8 with int32 accumulate	1.0
MAXIMUM	signed 32	1.0
MAX_POOL2D	signed 8	1.0
MINIMUM	signed 32	1.0
MUL	signed 8	1.0
MUL	signed 16	1.0
MUL	signed 32	1.0
NEGATE	signed 8	1.0
NEGATE	signed 16	1.0
NEGATE	signed 32	1.0
PAD	Boolean	1.0
PAD	signed 8	1.0
PAD	signed 16	1.0
PAD	signed 32	1.0
REDUCE_ALL	Boolean	1.0
REDUCE_ANY	Boolean	1.0
REDUCE_MAX	signed 8	1.0
REDUCE_MAX	signed 16	1.0

Operator	Mode	Version Added
REDUCE_MAX	signed 32	1.0
REDUCE_MIN	signed 8	1.0
REDUCE_MIN	signed 16	1.0
REDUCE_MIN	signed 32	1.0
REDUCE_SUM	signed 32	1.0
RESCALE	8-bit to 8-bit	1.0
RESCALE	8-bit to 16-bit	1.0
RESCALE	8-bit to 32-bit	1.0
RESCALE	16-bit to 8-bit	1.0
RESCALE	16-bit to 16-bit	1.0
RESCALE	16-bit to 32-bit	1.0
RESCALE	32-bit to 8-bit	1.0
RESCALE	32-bit to 16-bit	1.0
RESCALE	32-bit to 32-bit	1.0
RESHAPE	Boolean	1.0
RESHAPE	signed 8	1.0
RESHAPE	signed 16	1.0
RESHAPE	signed 32	1.0
RESIZE	signed 8, bilinear	1.0
RESIZE	signed 8, nearest	1.0
REVERSE	Boolean	1.0
REVERSE	signed 8	1.0
REVERSE	signed 16	1.0
REVERSE	signed 32	1.0
SCATTER	signed 8	1.0
SCATTER	signed 16	1.0
SCATTER	signed 32	1.0
SELECT	Boolean	1.0
SELECT	signed 8	1.0
SELECT	signed 16	1.0
SELECT	signed 32	1.0
SLICE	Boolean	1.0
SLICE	signed 8	1.0

Operator	Mode	Version Added
SLICE	signed 16	1.0
SLICE	signed 32	1.0
SUB	signed 32	1.0
TABLE	signed 8	1.0
TILE	Boolean	1.0
TILE	signed 8	1.0
TILE	signed 16	1.0
TILE	signed 32	1.0
TRANSPOSE	Boolean	1.0
TRANSPOSE	signed 8	1.0
TRANSPOSE	signed 16	1.0
TRANSPOSE	signed 32	1.0
TRANSPOSE_CONV2D	signed 8x8 with int32 accumulate	1.0

6.1.2. Floating-Point

FP16 and FP32 operations

Status: Complete

Operator	Mode	Version Added
ABS	fp16	1.0
ABS	fp32	1.0
ADD	signed 32	1.0
ADD	fp16	1.0
ADD	fp32	1.0
ARGMAX	fp16	1.0
ARGMAX	fp32	1.0
AVG_POOL2D	fp16 with fp16 accumulate	1.0
AVG_POOL2D	fp16 with fp32 accumulate	1.0
AVG_POOL2D	fp32 with fp32 accumulate	1.0
CAST	signed 8 to fp16	1.0
CAST	signed 8 to fp32	1.0
CAST	signed 16 to fp16	1.0
CAST	signed 16 to fp32	1.0

Operator	Mode	Version Added
CAST	signed 32 to fp16	1.0
CAST	signed 32 to fp32	1.0
CAST	fp16 to signed 8	1.0
CAST	fp16 to signed 16	1.0
CAST	fp16 to signed 32	1.0
CAST	fp16 to fp32	1.0
CAST	fp32 to signed 8	1.0
CAST	fp32 to signed 16	1.0
CAST	fp32 to signed 32	1.0
CAST	fp32 to fp16	1.0
CEIL	fp16	1.0
CEIL	fp32	1.0
CLAMP	fp16	1.0
CLAMP	fp32	1.0
CONCAT	Boolean	1.0
CONCAT	fp16	1.0
CONCAT	fp32	1.0
CONST	Boolean	1.0
CONST	8-bit	1.0
CONST	16-bit	1.0
CONST	32-bit	1.0
CONST	fp16	1.0
CONST	fp32	1.0
CONST_SHAPE	shape	1.0
CONV2D	fp16 with fp16 accumulate	1.0
CONV2D	fp16 with fp32 accumulate	1.0
CONV2D	fp32 with fp32 accumulate	1.0
CONV3D	fp16 with fp16 accumulate	1.0
CONV3D	fp16 with fp32 accumulate	1.0
CONV3D	fp32 with fp32 accumulate	1.0
COS	fp16	1.0
COS	fp32	1.0
CUSTOM	All	1.0

Operator	Mode	Version Added
DEPTHWISE_CONV2D	fp16 with fp16 accumulate	1.0
DEPTHWISE_CONV2D	fp16 with fp32 accumulate	1.0
DEPTHWISE_CONV2D	fp32 with fp32 accumulate	1.0
EQUAL	fp16	1.0
EQUAL	fp32	1.0
ERF	fp16	1.0
ERF	fp32	1.0
EXP	fp16	1.0
EXP	fp32	1.0
FLOOR	fp16	1.0
FLOOR	fp32	1.0
GATHER	fp16	1.0
GATHER	fp32	1.0
GREATER	fp16	1.0
GREATER	fp32	1.0
GREATER_EQUAL	fp16	1.0
GREATER_EQUAL	fp32	1.0
IDENTITY	Boolean	1.0
IDENTITY	8-bit	1.0
IDENTITY	16-bit	1.0
IDENTITY	32-bit	1.0
IDENTITY	fp16	1.0
IDENTITY	fp32	1.0
INTDIV	signed 32	1.0
LOG	fp16	1.0
LOG	fp32	1.0
LOGICAL_AND	Boolean	1.0
LOGICAL_LEFT_SHIFT	signed 8	1.0
LOGICAL_LEFT_SHIFT	signed 16	1.0
LOGICAL_LEFT_SHIFT	signed 32	1.0
LOGICAL_NOT	Boolean	1.0
LOGICAL_OR	Boolean	1.0
LOGICAL_RIGHT_SHIFT	signed 8	1.0

Operator	Mode	Version Added
LOGICAL_RIGHT_SHIFT	signed 16	1.0
LOGICAL_RIGHT_SHIFT	signed 32	1.0
LOGICAL_XOR	Boolean	1.0
MATMUL	fp16 with fp16 accumulate	1.0
MATMUL	fp16 with fp32 accumulate	1.0
MATMUL	fp32 with fp32 accumulate	1.0
MAXIMUM	fp16	1.0
MAXIMUM	fp32	1.0
MAX_POOL2D	fp16	1.0
MAX_POOL2D	fp32	1.0
MINIMUM	fp16	1.0
MINIMUM	fp32	1.0
MUL	signed 32	1.0
MUL	fp16	1.0
MUL	fp32	1.0
NEGATE	fp16	1.0
NEGATE	fp32	1.0
PAD	Boolean	1.0
PAD	fp16	1.0
PAD	fp32	1.0
POW	fp16	1.0
POW	fp32	1.0
RECIPROCAL	fp16	1.0
RECIPROCAL	fp32	1.0
REDUCE_ALL	Boolean	1.0
REDUCE_ANY	Boolean	1.0
REDUCE_MAX	fp16	1.0
REDUCE_MAX	fp32	1.0
REDUCE_MIN	fp16	1.0
REDUCE_MIN	fp32	1.0
REDUCE_PRODUCT	fp16	1.0
REDUCE_PRODUCT	fp32	1.0
REDUCE_SUM	fp16	1.0

Operator	Mode	Version Added
REDUCE_SUM	fp32	1.0
RESHAPE	Boolean	1.0
RESHAPE	fp16	1.0
RESHAPE	fp32	1.0
RESIZE	fp16	1.0
RESIZE	fp32	1.0
REVERSE	Boolean	1.0
REVERSE	fp16	1.0
REVERSE	fp32	1.0
RSQRT	fp16	1.0
RSQRT	fp32	1.0
SCATTER	fp16	1.0
SCATTER	fp32	1.0
SELECT	Boolean	1.0
SELECT	fp16	1.0
SELECT	fp32	1.0
SIGMOID	fp16	1.0
SIGMOID	fp32	1.0
SIN	fp16	1.0
SIN	fp32	1.0
SLICE	Boolean	1.0
SLICE	fp16	1.0
SLICE	fp32	1.0
SUB	signed 32	1.0
SUB	fp16	1.0
SUB	fp32	1.0
TANH	fp16	1.0
TANH	fp32	1.0
TILE	Boolean	1.0
TILE	fp16	1.0
TILE	fp32	1.0
TRANSPOSE	Boolean	1.0
TRANSPOSE	fp16	1.0

Operator	Mode	Version Added
TRANSPOSE	fp32	1.0
TRANSPOSE_CONV2D	fp16 with fp16 accumulate	1.0
TRANSPOSE_CONV2D	fp16 with fp32 accumulate	1.0
TRANSPOSE_CONV2D	fp32 with fp32 accumulate	1.0

6.2. Profile Extensions

6.2.1. EXT-INT16 extension

16-bit integer operations

Status: Complete

Compatible profiles: PRO-INT

Operator Change Table

Operator	Mode	Version Added	Note
ARGMAX	signed 16	1.0	
AVG_POOL2D	signed 16 with int32 accumulate	1.0	
CLAMP	signed 16	1.0	
CONCAT	signed 16	1.0	
CONST	48-bit	1.0	
CONV2D	signed 16x8 with int48 accumulate	1.0	
CONV3D	signed 16x8 with int48 accumulate	1.0	
DEPTHWISE_CONV2D	signed 16x8 with int48 accumulate	1.0	
IDENTITY	48-bit	1.0	
MATMUL	signed 16x16 with int48 accumulate	1.0	
MAX_POOL2D	signed 16	1.0	
RESCALE	48-bit to 8-bit	1.0	
RESCALE	48-bit to 16-bit	1.0	
RESCALE	48-bit to 32-bit	1.0	
RESIZE	signed 16, bilinear	1.0	
RESIZE	signed 16, nearest	1.0	

Operator	Mode	Version Added	Note
TABLE	signed 16	1.0	
TRANSPOSE_CONV2D	signed 16x8 with int48 accumulate	1.0	

6.2.2. EXT-INT4 extension

4-bit integer weights

Status: Complete

Compatible profiles: PRO-INT

Operator Change Table

Operator	Mode	Version Added	Note
CONST	4-bit	1.0	
CONV2D	signed 8x4 with int32 accumulate	1.0	
CONV3D	signed 8x4 with int32 accumulate	1.0	
DEPTHWISE_CONV2D	signed 8x4 with int32 accumulate	1.0	
IDENTITY	4-bit	1.0	
TRANSPOSE_CONV2D	signed 8x4 with int32 accumulate	1.0	

6.2.3. EXT-BF16 extension

BFloat16 operations

Status: Experimental

Compatible profiles: PRO-FP

Operator Change Table

Operator	Mode	Version Added	Note
ABS	bf16	1.0	
ADD	bf16	1.0	
ARGMAX	bf16	1.0	
AVG_POOL2D	bf16 with fp32 accumulate	1.0	
CAST	signed 8 to bf16	1.0	

Operator	Mode	Version Added	Note
CAST	signed 16 to bf16	1.0	
CAST	signed 32 to bf16	1.0	
CAST	bf16 to signed 8	1.0	
CAST	bf16 to signed 16	1.0	
CAST	bf16 to signed 32	1.0	
CAST	bf16 to fp8e4m3	1.0	If EXT-FP8E4M3 is also supported
CAST	bf16 to fp8e5m2	1.0	If EXT-FP8E5M2 is also supported
CAST	bf16 to fp32	1.0	
CAST	fp8e4m3 to bf16	1.0	If EXT-FP8E4M3 is also supported
CAST	fp8e5m2 to bf16	1.0	If EXT-FP8E5M2 is also supported
CAST	fp32 to bf16	1.0	
CEIL	bf16	1.0	
CLAMP	bf16	1.0	
CONCAT	bf16	1.0	
CONST	bf16	1.0	
CONV2D	bf16 with fp32 accumulate	1.0	
CONV3D	bf16 with fp32 accumulate	1.0	
COS	bf16	1.0	
DEPTHWISE_CONV2D	bf16 with fp32 accumulate	1.0	
EQUAL	bf16	1.0	
ERF	bf16	1.0	
EXP	bf16	1.0	
FLOOR	bf16	1.0	
GATHER	bf16	1.0	
GREATER	bf16	1.0	
GREATER_EQUAL	bf16	1.0	
IDENTITY	bf16	1.0	
LOG	bf16	1.0	

Operator	Mode	Version Added	Note
MATMUL	bf16 with fp32 accumulate	1.0	
MAXIMUM	bf16	1.0	
MAX_POOL2D	bf16	1.0	
MINIMUM	bf16	1.0	
MUL	bf16	1.0	
NEGATE	bf16	1.0	
PAD	bf16	1.0	
POW	bf16	1.0	
RECIPROCAL	bf16	1.0	
REDUCE_MAX	bf16	1.0	
REDUCE_MIN	bf16	1.0	
REDUCE_PRODUCT	bf16	1.0	
REDUCE_SUM	bf16	1.0	
RESHAPE	bf16	1.0	
RESIZE	bf16	1.0	
REVERSE	bf16	1.0	
RSQRT	bf16	1.0	
SCATTER	bf16	1.0	
SELECT	bf16	1.0	
SIGMOID	bf16	1.0	
SIN	bf16	1.0	
SLICE	bf16	1.0	
SUB	bf16	1.0	
TANH	bf16	1.0	
TILE	bf16	1.0	
TRANSPOSE	bf16	1.0	
TRANSPOSE_CONV2D	bf16 with fp32 accumulate	1.0	

6.2.4. EXT-FP8E4M3 extension

8-bit floating-point operations E4M3

Status: Experimental

Compatible profiles: PRO-FP

Operator Change Table

Operator	Mode	Version Added	Note
ARGMAX	fp8e4m3	1.0	
AVG_POOL2D	fp8e4m3 with fp16 accumulate	1.0	
CAST	bf16 to fp8e4m3	1.0	If EXT-BF16 is also supported
CAST	fp8e4m3 to fp16	1.0	
CAST	fp8e4m3 to bf16	1.0	If EXT-BF16 is also supported
CAST	fp8e4m3 to fp32	1.0	
CAST	fp16 to fp8e4m3	1.0	
CAST	fp32 to fp8e4m3	1.0	
CONCAT	fp8e4m3	1.0	
CONST	fp8e4m3	1.0	
CONV2D	fp8e4m3 with fp16 accumulate	1.0	
CONV3D	fp8e4m3 with fp16 accumulate	1.0	
DEPTHWISE_CONV2D	fp8e4m3 with fp16 accumulate	1.0	
GATHER	fp8e4m3	1.0	
IDENTITY	fp8e4m3	1.0	
MATMUL	fp8e4m3 with fp16 accumulate	1.0	
MAX_POOL2D	fp8e4m3	1.0	
PAD	fp8e4m3	1.0	
RESHAPE	fp8e4m3	1.0	
REVERSE	fp8e4m3	1.0	
SCATTER	fp8e4m3	1.0	
SLICE	fp8e4m3	1.0	
TILE	fp8e4m3	1.0	
TRANSPOSE	fp8e4m3	1.0	
TRANSPOSE_CONV2D	fp8e4m3 with fp16 accumulate	1.0	

6.2.5. EXT-FP8E5M2 extension

8-bit floating-point operations E5M2

Status: Experimental

Compatible profiles: PRO-FP

Operator Change Table

Operator	Mode	Version Added	Note
ARGMAX	fp8e5m2	1.0	
AVG_POOL2D	fp8e5m2 with fp16 accumulate	1.0	
CAST	bf16 to fp8e5m2	1.0	If EXT-BF16 is also supported
CAST	fp8e5m2 to fp16	1.0	
CAST	fp8e5m2 to bf16	1.0	If EXT-BF16 is also supported
CAST	fp8e5m2 to fp32	1.0	
CAST	fp16 to fp8e5m2	1.0	
CAST	fp32 to fp8e5m2	1.0	
CONCAT	fp8e5m2	1.0	
CONST	fp8e5m2	1.0	
CONV2D	fp8e5m2 with fp16 accumulate	1.0	
CONV3D	fp8e5m2 with fp16 accumulate	1.0	
DEPTHWISE_CONV2D	fp8e5m2 with fp16 accumulate	1.0	
GATHER	fp8e5m2	1.0	
IDENTITY	fp8e5m2	1.0	
MATMUL	fp8e5m2 with fp16 accumulate	1.0	
MAX_POOL2D	fp8e5m2	1.0	
PAD	fp8e5m2	1.0	
RESHAPE	fp8e5m2	1.0	
REVERSE	fp8e5m2	1.0	
SCATTER	fp8e5m2	1.0	
SLICE	fp8e5m2	1.0	

Operator	Mode	Version Added	Note
TILE	fp8e5m2	1.0	
TRANSPOSE	fp8e5m2	1.0	
TRANSPOSE_CONV2D	fp8e5m2 with fp16 accumulate	1.0	

6.2.6. EXT-FFT extension

Fast Fourier Transform operations

Status: Complete

Compatible profiles: PRO-FP

Operator Change Table

Operator	Mode	Version Added	Note
FFT2D	fp32	1.0	
RFFT2D	fp32	1.0	

6.2.7. EXT-VARIABLE extension

Stateful variable operations

Status: Experimental

Compatible profiles: PRO-INT, PRO-FP

Operator Change Table

Operator	Mode	Version Added	Note
VARIABLE	signed 8	1.0	If PRO-INT is also supported
VARIABLE	fp16	1.0	If PRO-FP is also supported
VARIABLE	fp32	1.0	If PRO-FP is also supported
VARIABLE_READ	signed 8	1.0	If PRO-INT is also supported
VARIABLE_READ	fp16	1.0	If PRO-FP is also supported
VARIABLE_READ	fp32	1.0	If PRO-FP is also supported

Operator	Mode	Version Added	Note
VARIABLE_WRITE	signed 8	1.0	If PRO-INT is also supported
VARIABLE_WRITE	fp16	1.0	If PRO-FP is also supported
VARIABLE_WRITE	fp32	1.0	If PRO-FP is also supported

6.2.8. EXT-CONTROLFLOW extension

Control Flow operations

Status: Experimental

Compatible profiles: PRO-INT, PRO-FP

Operator Change Table

Operator	Mode	Version Added	Note
COND_IF	All	1.0	
WHILE_LOOP	All	1.0	

6.2.9. EXT-DYNAMIC extension

Removes all Compile Time Constant state for CTC inputs

Status: Experimental

Compatible profiles: PRO-INT, PRO-FP

Operator Change Table

Operator	Mode	Version Added	Note
AVG_POOL2D	all		Remove CTC from input_zp
AVG_POOL2D	all		Remove CTC from output_zp
CONV2D	all		Remove CTC from input_zp
CONV2D	all		Remove CTC from weight_zp
CONV3D	all		Remove CTC from input_zp

Operator	Mode	Version Added	Note
CONV3D	all		Remove CTC from weight_zp
DEPTHWISE_CONV2D	all		Remove CTC from input_zp
DEPTHWISE_CONV2D	all		Remove CTC from weight_zp
MATMUL	all		Remove CTC from A_zp
MATMUL	all		Remove CTC from B_zp
MUL	all		Remove CTC from shift
NEGATE	all		Remove CTC from input1_zp
NEGATE	all		Remove CTC from output_zp
PAD	all		Remove CTC from padding
PAD	all		Remove CTC from pad_const
RESCALE	all		Remove CTC from multiplier
RESCALE	all		Remove CTC from shift
RESCALE	all		Remove CTC from input_zp
RESCALE	all		Remove CTC from output_zp
RESHAPE	all		Remove CTC from shape
RESIZE	all		Remove CTC from scale
RESIZE	all		Remove CTC from offset
RESIZE	all		Remove CTC from border
SLICE	all		Remove CTC from start
SLICE	all		Remove CTC from size
TABLE	all		Remove CTC from table
TILE	all		Remove CTC from multiples

Operator	Mode	Version Added	Note
TRANSPOSE_CONV2D	all		Remove CTC from input_zp
TRANSPOSE_CONV2D	all		Remove CTC from weight_zp

6.2.10. EXT-DOUBLEROUND extension

Adds double rounding support to the RESCALE operator

Status: Complete

Compatible profiles: PRO-INT

Operator Change Table

Operator	Mode	Version Added	Note
No changes			

Enum Changes

Enum	Value	Note
rounding_mode_t	DOUBLE_ROUND	New Value

6.2.11. EXT-INEXACTROUND extension

Adds inexact rounding support to the RESCALE operator

Status: Experimental

Compatible profiles: PRO-INT

Operator Change Table

Operator	Mode	Version Added	Note
No changes			

Enum Changes

Enum	Value	Note
rounding_mode_t	INEXACT_ROUND	New Value

7. Appendix C - Rationale

This appendix documents the rationale behind decisions made while creating the TOSA specification. Explanations and definitions contained in this appendix are non-normative.

7.1. FP8

The operators that perform calculations on FP8 data types are limited. Fewer mantissa bits in FP8 make it inappropriate for use in most elementwise operations such as ADD. Support was also added to the data layout and movement operations on the understanding that no calculations are performed. Two extensions for the FP8 types were created in order to cover both formats defined by OCP-OFP8.

7.2. Transcendental Functions

In the TOSA specification, a limited number of transcendental operations are supported. The operators supported are sufficient for common networks while minimizing the number of operations an implementation must support. Originally, SIGMOID and TANH were added as the common functions used for activations. ERF was added to support GELU style activation functions. SIN and COS were added to provide a base level of trigonometric functionality as well as support for Rotary Position Embedding.

7.3. Removed Operators

In version 0.90, a set of shape operators were introduced to attempt to allow dynamically shaped network to be expressed completely with TOSA. There are gaps in this implementation, and as such the shape operators have been removed. This removes the requirement on future implementations retain compatibility with these operators. The shape_t type remains, and the CONST_SHAPE operator allows creating instances of shape_t type.

FULLY_CONNECTED has been removed from TOSA. FULLY_CONNECTED functionality can be achieved by using the CONV2D operator. Using CONV2D allows the bias add to be included in the operator, where MATMUL does not include bias support.