The OPU ISA Specification

Version 0.1-draft, July 2020

Edited by Louis Delhez, Lei He, Yunxuan Yu, and Tiandong Zhao.
opu-admin@googlegroups.com

Contents

Chapter 1

Introduction

Academic research of CPU has been facilitated greatly by open source toolchain and resources such as instruction set (ISA), compiler, and instruction/microarchitecture level simulation. Examples include SimpleScalar, GEM5, and more recently, RISC-V toolset. Yet, such open-source and complete eco-system does not exist for general machine learning algorithms. OPU, short for *open processing unit*, is meant to be an open-source and complete eco-system for machine learning hardware research, including: ISA with executable specifications, compiler with formal verification, instruction level (functional) and microarchitecture level (cycle-accurate) simulation, parametrized modules in RTL and Chisel, and FPGA emulation and development boards.

This documents describes the OPU Instruction Set Architecture.

1.1 Data types

The OPU ISA is designed to be polymorphic: different implementations operating on different data types can execute the same binary. Implementations are characterized by the four data types that they use for different kinds of values:

- `ITYPE`: type of feature values in memory
- `BTYPE`: type of bias values in memory
- `KTYPE`: type of kernel weights in memory
- `OTYPE`: type of partial sums in the processor

Chapter 3 defines how the various combinations of datatypes should be handled by the implementations.

1.2 Instruction Fundamentals

An OPU *program* consists of a sequence of *instructions* stored contiguously in memory. Instructions are always encoded on 32 bits, following the little-endian convention in memory. Instructions are executed one after the other in program order. The execution stops when an `end` instruction is executed. The address of the first instruction of a program must be 64-byte aligned. There is no such restriction regarding the last instruction of the program.

1.3 Memory

OPU has an address space of 2^{32} bytes. The little-endian convention is always assumed by OPU when reading and writing multi-bytes values in memory. Instructions and data share a same unique address space.

Memory accesses are always performed in program order. A load operation always sees the effect of all previous store operations.

Chapter 2

OPU ISA

2.1 Registers

Contrary to conventional ISAs that define registers for storing temporary values on which operations can be performed, the registers of the OPU ISA are exclusively utilized for configuration of subsequent operations.

The registers of the OPU ISA are listed in Table 2.1.

Registers are set by *configuration instructions*, whose names start with the symbol `@`.

Name	*Description*	*Possible values*
`ifm_h`	first dimension of the `ifm` buffer	$\{1, \ldots, 2^7 - 1\}$
`ifm_w`	second dimension of the `ifm` buffer	$\{1, \ldots, 2^7 - 1\}$
`ifm_c`	third dimension of the `ifm` buffer	$2^i, i \in \{4, \ldots, 6\}$
`ofm_h`	first dimension of the `ofm` buffer	$\{1, \ldots, 2^7 - 1\}$
`ofm_w`	second dimension of the `ofm` buffer	$\{1, \ldots, 2^7 - 1\}$
`ofm_c`	third dimension of the `ofm` buffer	$2^i, i \in \{1, \ldots, 6\}$
`ker_n`	first dimension of the `ker` buffer	$\{1, \ldots, 36\}$
`stride_h`	stride along first dimension for `conv` instructions	$\{1, \ldots, 7\}$
`stride_w`	stride along second dimension for `conv` instructions	$\{1, \ldots, 7\}$
`ifm_addr`	address offset for `ld.ifm` instructions	$a \times 2^{28}, a \in \{0, \ldots, 2^4 - 1\}$
`ker_addr`	address offset for `ld.ker` instructions	$a \times 2^{28}, a \in \{0, \ldots, 2^4 - 1\}$
`bias_addr`	address offset for `ld.bias` instructions	$a \times 2^{28}, a \in \{0, \ldots, 2^4 - 1\}$
`ifm_mem_w`	second dimension of array for `ld.ifm` instructions	$\{1, \ldots, 2^{10} - 1\}$
`ofm_mem_w`	second dimension of array for `ld.ofm` instructions	$\{1, \ldots, 2^{10} - 1\}$
`ofm_mem_h`	first dimension of array for `pad` instructions	$\{1, \ldots, 2^{10} - 1\}$
`ifm_shift`	scale factor of the `ifm` buffer	$\{-2^7, \ldots, 2^7 - 1\}$
`bias_shift`	scale factor of the `bias` buffer	$\{-2^7, \ldots, 2^7 - 1\}$
`post_order`	order of operations for `store` instructions	$\{0, 1, 2\}$
`add_ifm`	switch for `ifm` addition for `store` instructions	$\{0, 1\}$
`act`	activation function to use for `store` instructions	$\{0, 1, 2\}$
`pool_h`	first dimension of the pooling filter	$\{1, \ldots, 2^4 - 1\}$
`pool_w`	second dimension of the pooling filter	$\{1, \ldots, 2^4 - 1\}$
`pool_h_stride`	stride along first dimension for the pooling filter	$\{1, \ldots, 2^3 - 1\}$
`pool_w_stride`	stride along first dimension for the pooling filter	$\{1, \ldots, 2^3 - 1\}$

Table 2.1: Registers of the OPU ISA.

2.2 Buffers

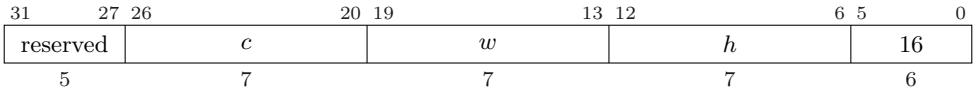
In OPU, the role of traditional processor registers is fulfilled by large flexible multi-dimensional memories called buffers. Four different buffers are defined: `ifm`, `ker`, `bias`, and `ofm`. Table 2.2 details their characteristics. While the dimensions of these buffers can be modified at runtime by setting the appropriate registers, the types of the elements of the buffers are constant for a given implementation.

Buffer	Dimensions	Data type
`ifm`	`ifm_h` × `ifm_w` × `ifm_c`	`ITYPE`
`ofm`	`ofm_h` × `ofm_w` × `ofm_c`	`OTYPE`
`ker`	`ker_n` × `ofm_c` × `ifm_c`	`KTYPE`
`bias`	`ofm_c`	`BTYPE`

2.3 Instructions

2.3.1 Configuration Instructions

`@shape.ifm`



The `@shape.ifm` instruction sets the size of the `ifm` buffer as

$$\begin{aligned} &\texttt{ifm\_h} \leftarrow h \\ &\texttt{ifm\_w} \leftarrow w \qquad \text{where} \qquad \gamma = 2^c. \\ &\texttt{ifm\_c} \leftarrow \gamma \end{aligned}$$

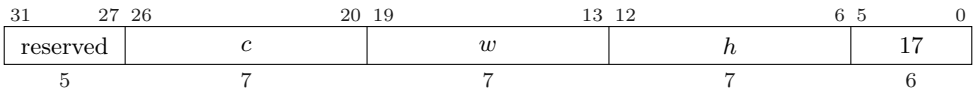
When this instruction is executed, the content of the `ifm` and `ker` buffers is invalidated. The instruction is only valid if the following condition is satisfied:

$$1 \leqslant h \times w \leqslant 2048 \wedge 16 \leqslant \gamma \leqslant 64.$$

In assembly, this instruction is represented by

`@shape.ifm [`h`,`w`,`γ`]`

`@shape.ofm`



The `@shape.ofm` instruction sets the size of the `ofm` buffer as

$$\begin{aligned} &\texttt{ofm\_h} \leftarrow h \\ &\texttt{ofm\_w} \leftarrow w \qquad \text{where} \qquad \gamma = 2^c. \\ &\texttt{ofm\_c} \leftarrow \gamma \end{aligned}$$

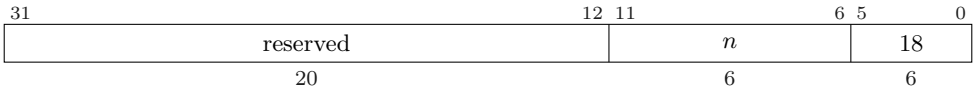
When this instruction is executed, the content of the `ofm` and `ker` buffers is invalidated. The instruction is only valid if the following condition is satisfied:

$$1 \leqslant h \times w \leqslant 2048 \wedge 2 \leqslant \gamma \leqslant 64.$$

In assembly, this instruction is represented by

`@shape.ofm` [h,w,γ]

`@shape.ker`



The `@shape.ker` instruction sets the first dimension of the `ker` buffer as

$$\texttt{ker\_n} \leftarrow n.$$

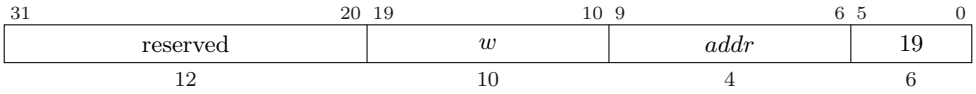
When this instruction is executed, the content of the `ker` buffer is invalidated. The instruction is only valid if the following condition is satisfied:

$$1 \leqslant n \leqslant 36.$$

In assembly, this instruction is represented by

`@shape.ker` n

`@mem.ifm`



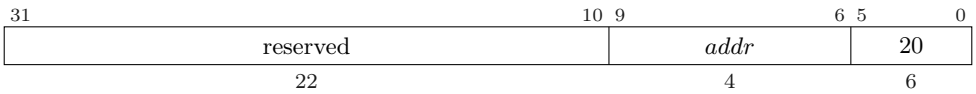
The `@mem.ifm` instruction specifies the location of the data to be loaded by the subsequent `ld.ifm` instructions as

$$\texttt{ifm\_addr} \leftarrow addr \ll 28$$
$$\texttt{ifm\_mem\_w} \leftarrow w.$$

In assembly, this instruction is represented by

`@mem.ifm` $addr$, w

`@mem.ker`



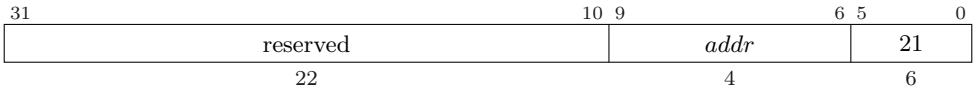
The `@mem.ker` instruction specifies the location of the data to be loaded by the subsequent `ld.ker` instructions as

$$\texttt{ker\_addr} \leftarrow addr \ll 28.$$

In assembly, this instruction is represented by

`@mem.ker` $addr$

`@mem.bias`



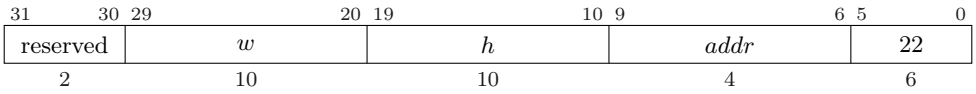
The `@mem.bias` instruction specifies the location of the data to be loaded by the subsequent `ld.bias` instructions as

$$\texttt{bias\_addr} \leftarrow addr \ll 28.$$

In assembly, this instruction is represented by

`@mem.bias` *addr*

`@mem.ofm`



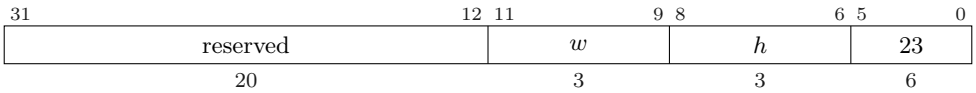
The `@mem.ofm` instruction specifies the location where data will be stored by the subsequent `store` instructions as

$$\texttt{ofm\_addr} \leftarrow addr \ll 28$$
$$\texttt{ofm\_mem\_h} \leftarrow h$$
$$\texttt{ofm\_mem\_w} \leftarrow w.$$

In assembly, this instruction is represented by

`@mem.ofm` *addr*`, [`*h*`,`*w*`]`

`@stride`



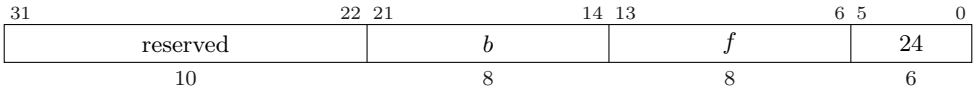
The `@stride` instruction sets the strides for the sebsequent `conv` instructions with

$$\texttt{stride\_h} \leftarrow h$$
$$\texttt{stride\_w} \leftarrow w.$$

In assembly, this instruction is represented by

`@stride [`*h*`,` *w*`]`

`@shift`



The `@shift` sets the scale factors for the values of the `ifm` and `bias` buffers:

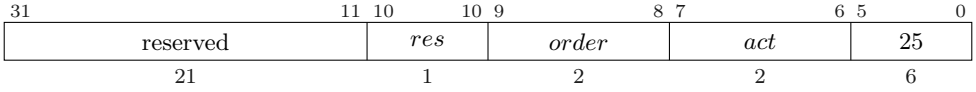
$$\texttt{ifm\_shift} \leftarrow f$$
$$\texttt{bias\_shift} \leftarrow b$$

The fields f and b are signed integers encoded using two's complement representation.

In assembly, this instruction is represented by

`@shift` *f*`,` *b*

@post



The post instruction specifies what operation must be performed during subsequent store as

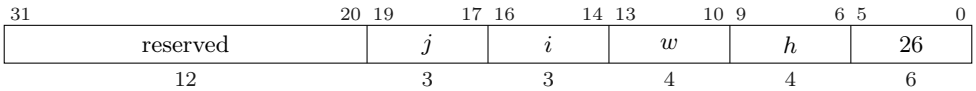
$$\begin{aligned} \texttt{act} &\leftarrow act \\ \texttt{post\_order} &\leftarrow order \\ \texttt{add\_ifm} &\leftarrow res \end{aligned}$$

This instruction is only valid for the values listed in Table 2.2. As shown in this table, the representation of this instruction in assembly depends on the values of *res*, *order*, and *act*.

order	*act*	*res*	Assembly
0	0	0	`@post pool`
0	0	1	`@post res, pool`
0	1	0	`@post act.relu, pool`
0	1	1	`@post act.relu, res, pool`
0	2	0	`@post act.leaky, pool`
0	2	1	`@post act.leaky, res, pool`
1	1	1	`@post res, act.relu, pool`
1	2	1	`@post res, act.leaky, pool`
2	0	1	`@post pool, res`
2	1	1	`@post act.relu, pool, res`
2	2	1	`@post act.leaky, pool, res`

Table 2.2: Assembly representation of `@post`.

@pool



The @pool instruction characterizes the pooling filter applied in store instructions:

$$\begin{aligned} \texttt{pool\_h} &\leftarrow h \\ \texttt{post\_w} &\leftarrow w \\ \texttt{pool\_h\_stride} &\leftarrow i \\ \texttt{pool\_w\_stride} &\leftarrow j \end{aligned}$$

The instruction is only valid if the following condition is satisfied:

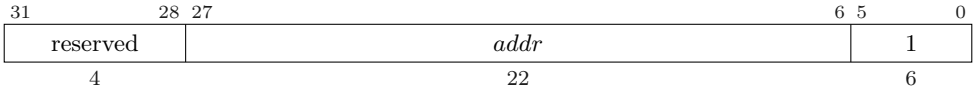
$$(1 \leqslant h) \wedge (1 \leqslant w) \wedge (1 \leqslant i) \wedge (1 \leqslant j).$$

In assembly, this instruction is represented by

`@pool [`h`,`w`], [`i`,`j`]`

2.3.2 Imperative Instructions

`ld.ifm`



The `ld.ifm` instruction loads a 3-dimensional array of type `ITYPE` from memory into the `ifm` buffer.

The shape of the loaded array is $\texttt{ifm\_h} \times \texttt{ifm\_w} \times \texttt{ifm\_c}$ (i.e. the `ifm` buffer is filled entirely). The first element of this array is assumed to be stored at address

$$\texttt{ifm\_addr} + addr \times 64.$$

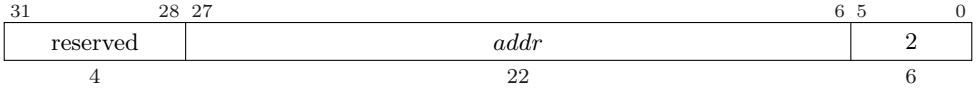
The access stride along the three dimensions of the array are respectively

$$\texttt{ifm\_mem\_w} \times 64, 64, \text{ and } 1.$$

In assembly, this instruction is represented by

$$\texttt{ld.ifm}\ addr$$

`ld.ker`



The `ld.ker` instruction loads a 3-dimensional array of type `KTYPE` from memory into the `ker` buffer.

The shape of the loaded array is $\texttt{ker\_n} \times \texttt{ofm\_c} \times \texttt{ifm\_c}$ (i.e. the `ker` buffer is filled entirely). The first element of this array is assumed to be stored at address

$$\texttt{ker\_addr} + addr \times 64.$$

The array is assumed to be stored contiguously in memory in row-major order, i.e. the access strides along the three dimensions of the array are respectively

$$\texttt{ofm\_c} \times \texttt{ifm\_c}, \texttt{ofm\_c}, \text{ and } 1.$$

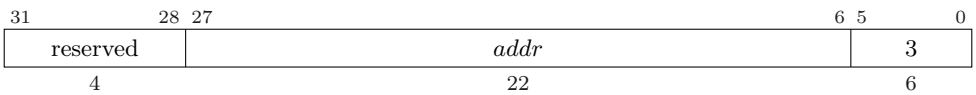
The `ld.ker` instruction is only valid if

$$n \times \max\left\{\frac{\texttt{ifm\_c} \times \texttt{ofm\_c}}{1024}, 1\right\} \leqslant 36.$$

In assembly, this instruction is represented by

$$\texttt{ld.ker}\ addr$$

`ld.bias`



The `ld.bias` instruction loads a 1-dimensional array of type `BTYPE` from memory into the `bias` buffer.

The length of the loaded array is equal to `ofm_c` (i.e. the `bias` buffer is filled entirely). The first element of this array is assumed to be stored at address

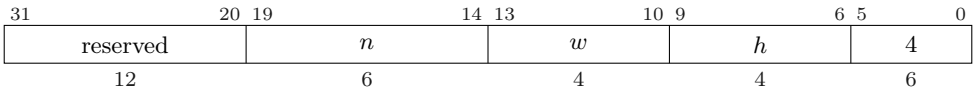
$$\texttt{bias\_addr} + addr \times 64.$$

The array is assumed to be stored contiguously in memory.

In assembly, this instruction is represented by

`ld.bias` *addr*

conv



The `conv` instruction performs a pointwise convolution between the `ifm` buffer and one slice of the `ker` buffer. The result of this operation is written to the `ofm` buffer:

$$\forall (i,j,k) \in \{0,\dots,\texttt{ofm\_h}-1\} \times \{0,\dots,\texttt{ofm\_w}-1\} \times \{0,\dots,\texttt{ofm\_c}-1\}:$$
$$\texttt{ofm}[i,j,k] := \kappa_{\texttt{OTYPE}}\left(2^{\texttt{ifm\_shift}} \times \sum_{\ell=0}^{\texttt{ifm\_c}-1} A[i,j,k,\ell]\right)$$

where

$$A[i,j,k,\ell] = \rho_{\texttt{KTYPE}}\left(\texttt{ker}[\texttt{ker\_n},k,\ell]\right) \times \rho_{\texttt{ITYPE}}\left(\texttt{ifm}[h+\texttt{stride\_h}\times i, w+\texttt{stride\_w}\times j,\ell]\right).$$

The type casting functions ρ and κ are defined in Chapter 3.

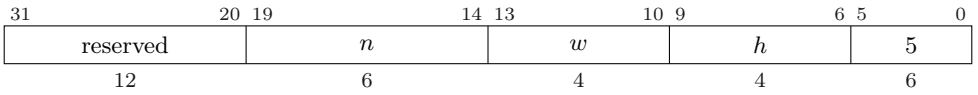
The instruction is only valid if the following condition is satisfied:

$$h + \texttt{ofm\_h} \times (\texttt{stride\_h} - 1) < \texttt{ifm\_h} \wedge w + \texttt{ofm\_w} \times (\texttt{stride\_w} - 1) < \texttt{ifm\_w}.$$

In assembly, this instruction is represented by

`conv ifm:[`h`,`w`], ker:`n

conv.bias



Similarly to `conv`, the `conv.bias` instruction performs a pointwise convolution between the `ifm` buffer and one slice of the `ker` buffer. In addition, the content of the `bias` buffer is added to the result. The result of this operation is written to the `ofm` buffer:

$$\forall (i,j,k) \in \{0,\dots,\texttt{ofm\_h}-1\} \times \{0,\dots,\texttt{ofm\_w}-1\} \times \{0,\dots,\texttt{ofm\_c}-1\}:$$
$$\texttt{ofm}[i,j,k] := \kappa_{\texttt{OTYPE}}\left(2^{\texttt{bias\_shift}} \times \rho_{\texttt{BTYPE}}(\texttt{bias}[k]) + 2^{\texttt{ifm\_shift}} \times \sum_{\ell=0}^{\texttt{ifm\_c}-1} A[i,j,k,\ell]\right)$$

where

$$A[i,j,k,\ell] = \rho_{\texttt{KTYPE}}\left(\texttt{ker}[\texttt{ker\_n}, k, \ell]\right) \times \rho_{\texttt{ITYPE}}\left(\texttt{ifm}[h + \texttt{stride\_h} \times i, w + \texttt{stride\_w} \times j, \ell]\right).$$

The type casting functions ρ and κ are defined in Chapter 3.

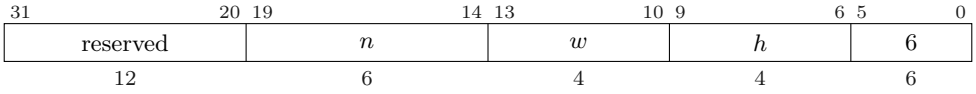
The instruction is only valid if the following condition is satisfied:

$$h + \texttt{ofm\_h} \times (\texttt{stride\_h} - 1) < \texttt{ifm\_h} \wedge h + \texttt{ofm\_w} \times (\texttt{stride\_w} - 1) < \texttt{ifm\_w}.$$

In assembly, this instruction is represented by

`conv.bias ifm:[`h`,`w`], ker:`n

`conv.acc`



Similarly to `conv`, the `conv.acc` instruction performs a pointwise convolution between the `ifm` buffer and one slice of the `ker` buffer. In addition, the content of the `ofm` buffer is added to the result:

$$\forall (i,j,k) \in \{0,\ldots,\texttt{ofm\_h}-1\} \times \{0,\ldots,\texttt{ofm\_w}-1\} \times \{0,\ldots,\texttt{ofm\_c}-1\}:$$
$$\texttt{ofm}[i,j,k] := \kappa_{\texttt{OTYPE}}\left(\rho_{\texttt{OTYPE}}(\texttt{ofm}[i,j,k]) + 2^{\texttt{ifm\_shift}} \times \sum_{\ell=0}^{\texttt{ifm\_c}-1} A[i,j,k,\ell]\right)$$

where

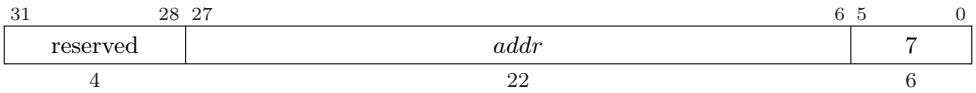
$$A[i,j,k,\ell] = \rho_{\texttt{KTYPE}}\left(\texttt{ker}[\texttt{ker\_n}, k, \ell]\right) \times \rho_{\texttt{ITYPE}}\left(\texttt{ifm}[h + \texttt{stride\_h} \times i, w + \texttt{stride\_w} \times j, \ell]\right).$$

The type casting functions ρ and κ are defined in Chapter 3.

In assembly, this instruction is represented by

`conv.acc ifm:[`h`,`w`], ker:`n

`store`



Let A be the 3-dimensional array of shape $(\texttt{ofm\_h}, \texttt{ofm\_w}, \texttt{ofm\_c})$ obtained by casting the elements of the `ofm` buffer to the type `ITYPE` as follows:

$$A[i,j,k] := \kappa_{\texttt{ITYPE}}\left(2^{\delta_{\texttt{ITYPE}} - \delta_{\texttt{OTYPE}}} \times \rho_{\texttt{OTYPE}}(\texttt{ofm}[i,j,k])\right)$$

where δ, κ, and ρ are defined in Chapter 3.

Let *activation* be a function that accepts as argument a 3-dimensional array X of shape $(\texttt{ofm\_h}, \texttt{ofm\_w}, \texttt{ofm\_c})$ and type `ITYPE` and that returns an array of same shape and type according to

$$activation(X)[i,j,k] := \begin{cases} X[i,k,k] & \text{if } \texttt{act} = 0 \\ \kappa_{\texttt{ITYPE}}(\max\{\rho_{\texttt{ITYPE}}(X[i,j,k]), 0\}) & \text{if } \texttt{act} = 1 \\ \kappa_{\texttt{ITYPE}}(\max\{\rho_{\texttt{ITYPE}}(X[i,j,k]), \rho_{\texttt{ITYPE}}(X[i,j,k])/8\}) & \text{if } \texttt{act} = 2 \end{cases}$$

Let *residual* be a function that accepts as argument a 3-dimensional array X of type `ITYPE` and returns an array of same shape and type according to

$$residual(X)[i,j,k] := \kappa_{\texttt{ITYPE}}\Big(\rho_{\texttt{ITYPE}}(X[i,j,k]) + \rho_{\texttt{ITYPE}}(\texttt{ifm}[i,j,k])\Big).$$

Let *pooling* be a function that accepts as argument a 3-dimensional array X of shape $(\texttt{ofm\_h}, \texttt{ofm\_w}, \texttt{ofm\_c})$ and type `ITYPE` and returns an array of same type and shape

$$\left(\left\lfloor \frac{\texttt{ofm\_h} - \texttt{pool\_h}}{\texttt{pool\_h\_stride}} \right\rfloor + 1, \left\lfloor \frac{\texttt{ofm\_w} - \texttt{pool\_w}}{\texttt{pool\_w\_stride}} \right\rfloor + 1, \texttt{ofm\_c}\right)$$

according to

$$pooling(X)[i,j,k] := \kappa_{\texttt{ITYPE}}\left(\max_{(a,b)\in\mathcal{A}\times\mathcal{B}} \rho_{\texttt{ITYPE}}\left(X[i \times \texttt{pool\_h\_stride} + a, j \times \texttt{pool\_w\_stride} + b, k]\right)\right)$$

where where $\mathcal{A} = \{0, 1, \ldots, \texttt{pool\_h} - 1\}$ and $\mathcal{B} = \{0, 1, \ldots, \texttt{pool\_w} - 1\}$.

The `store` instruction produces a 3-dimensional array B of type `ITYPE` as given by

$$B := \begin{cases} pooling(residual(activation(A))) & \text{if } \texttt{pos\_order} = 0 \\ pooling(activation(residual(A))) & \text{if } \texttt{pos\_order} = 1 \\ residual(pooling(activation(A))) & \text{if } \texttt{pos\_order} = 2 \end{cases}$$

The array B is then stored to memory starting at address

$$\texttt{ofm\_addr} + addr \times 64.$$

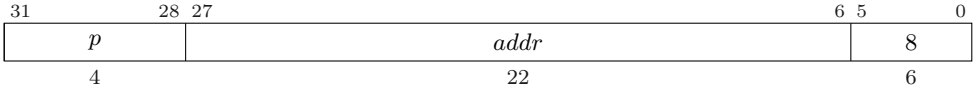
The strides along the three dimensions are respectively

$$\texttt{ofm\_mem\_w} \times 64, 64, \text{ and } 1.$$

In assembly, this instruction is represented by

`store` *addr*

pad



The `pad` instruction pads a 3-dimensional array in memory along its first two dimension. More specifically, assuming that *rows* and *colums* refer respectively to the first and second dimensions of the array, this instruction fills with zeros (i.e. $\kappa_{\texttt{ITYPE}}(0)$) the first p rows, first p colums, last p rows, and last p columns.

The shape of the considered array is

$$[\texttt{ofm\_mem\_h}, \texttt{ofm\_mem\_w}, 64].$$

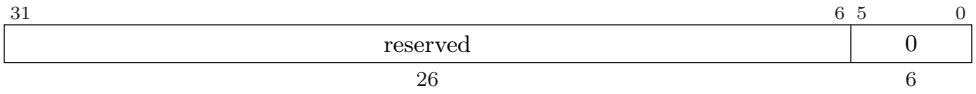
This array is assumed to be stored contiguously in memory in row-major order, i.e. its strides are respectively

$$\texttt{ofm\_mem\_w}, 64, \text{ and } 1.$$

In assembly, this instruction is represented by

$$\texttt{pad}\ \ addr\texttt{,}\ \ p$$

end



The end instruction terminates the execution of the current program. All programs must end with an end instruction.

In assembly, this instruction is simply represented by

end

Chapter 3

Arithmetic rules

In OPU, a data type T is defined by

- a set $\mathcal{S}_T$ of all valid representations
- a function $\rho_T : \mathcal{S}_T \to \mathbb{R}$ that maps a represented number to its actual value

3.1 Representable set

The set $\mathcal{R}_T \subset \mathbb{R}$ of all representable values using a data type T is defined as

$$\mathcal{R}_T \equiv \{x \in \mathbb{R} \mid (\exists y \in \mathcal{S}_T)[\rho_T(y) = x]\}\,.$$

3.2 Effective width

The effective width δ_T of a data type T is defined as

$$\delta_T \equiv \left\lceil \log_2 \left(\max_{x \in \mathcal{R}_T} |x| \right) \right\rceil .$$

3.3 Conversion from $\mathbb{R}$

The function $\kappa_T : \mathbb{R} \to \mathcal{S}_T$ maps any real number to its nearest representation. For all $x \in \mathbb{R}$, this function must satisfy

$$\rho_T(\kappa_T(x)) = \max_{a \in \mathcal{A}}(\rho_T(a))$$

where

$$\mathcal{A} = \arg\min_{y \in \mathcal{S}_T} |\rho_T(y) - x|.$$

Appendix A

Instruction Listing