BARVINN

MohammadHossein AskariHemmat, Olexa Bilaniuk, Sean Wagner

Mar 10, 2023

CONTENTS:

Editor: **MohammadHossein AskariHemmat** m.h.askari.hemmat@gmail.com

CHAPTER

ONE

INTRODUCTION

Deep neural networks (DNNs) have been the focus of much research and development in the last few years as the uptake of artificial intelligence in a number of application areas has grown rapidly. Although a lot of research has been carried out in developing new deep learning models and techniques, making deep learning models computationally affordable and accessible is still a challenge.

One way to accelerate computation in a deep neural network is to use less precision for computation. This is called quantization (Hubara et al., 2018). In deep learning, quantization is a technique to reduce memory consumption as well as the computation time of deep neural networks. In contrast, floating-point operations are slower and more costly (in terms of power consumption and the required area in a silicon chip) compared to fixed-point and integer operations. For instance, in a 45nm process, 32-bit integer multiplication and addition take 3.1 pJ (pico joules) and 0.1pJ, respectively (Horowitz, 2014). However, to do the same operation with floating-point values, it requires 3.7 pJ for multiplication and 0.9 pj for addition. On the other hand, using integer operands make the computation process faster. As an example in Intel Core i7 4770 running at 3.4GHz multiplication is more than 3 times faster for fixed-point data types compared to floating-point datatypes (LIMARE, LIMARE).

To benefit from quantization in a neural network, one must use a hardware that supports low precision computation. At the time of writing this documentation, there are no commercially available general processors (CPU or GPU) that can efficiently store and load sub-8-bit parameters of a neural network. Also, the general processors are not equipped with customized hardware to perform arbitrary precision computation. Hence, to fully benefit from quantization, one should consider designing custom ASICS. This document provides technical details for *BARVINN: a Barrel RISC-V Neural Network Accelerator Engine*. The main purpose of designing BARVINN was to fill the need for arbitrary precision computation in neural networks.

This documentation tries to help users and developers use BARVINN in their projects or improve it depending on their custom computation needs.

CHAPTER

TWO

DESIGN

2.1 BARVINN

BARVINN is a Barrel RISC-V Neural Network Accelerator. The main purpose of designing BARVINN is to fill the need for arbitrary precision neural network acceleration. The overall architecture of BARVINN is illustrated below. BARVINN is implemented in an FPGA. Fig. 2.1 illustrates the overall system design for BARVINN. It is consist of the following components:

- Array of Matrix Vector Units
- RISC-V Controller Core
- Host Machine

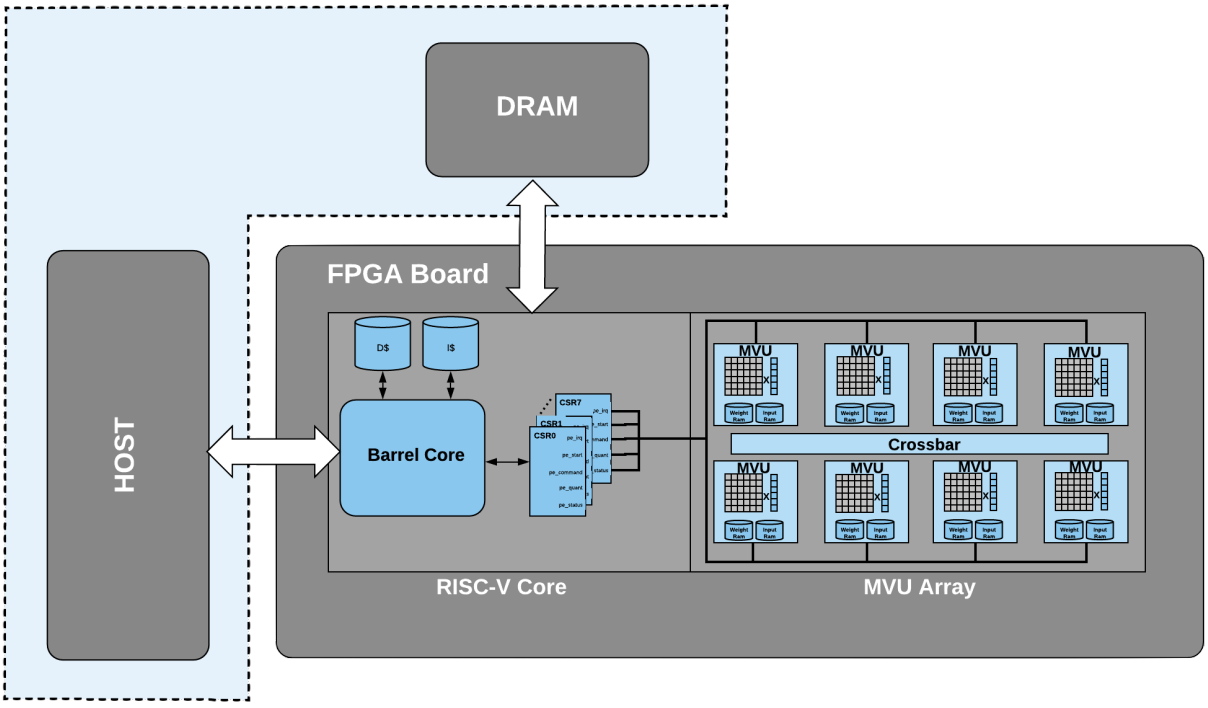


Fig. 2.1: BARVINN overall architecture.

In the following sections, we will review each part in details.

2.2 Matrix Vector Unit (MVU) Array

In the base configuration, BARVINN uses 8 MVUs. At every clock cycle, each MVU is capable of performing a binary matrix-vector product of the following size:

- Input Vector of 1 x 64 with 1 bit precision
- Weight Matrix of 64 x 64 with 1 bit precision

Each MVU has a local memory to store activation and weights. The MVUs are connected through a crossbar. The crossbar allows MVUs to send part of their local memory (activations) among themselves. This allows MVUs to work on different jobs with different configurations or to work together to compute a single task.

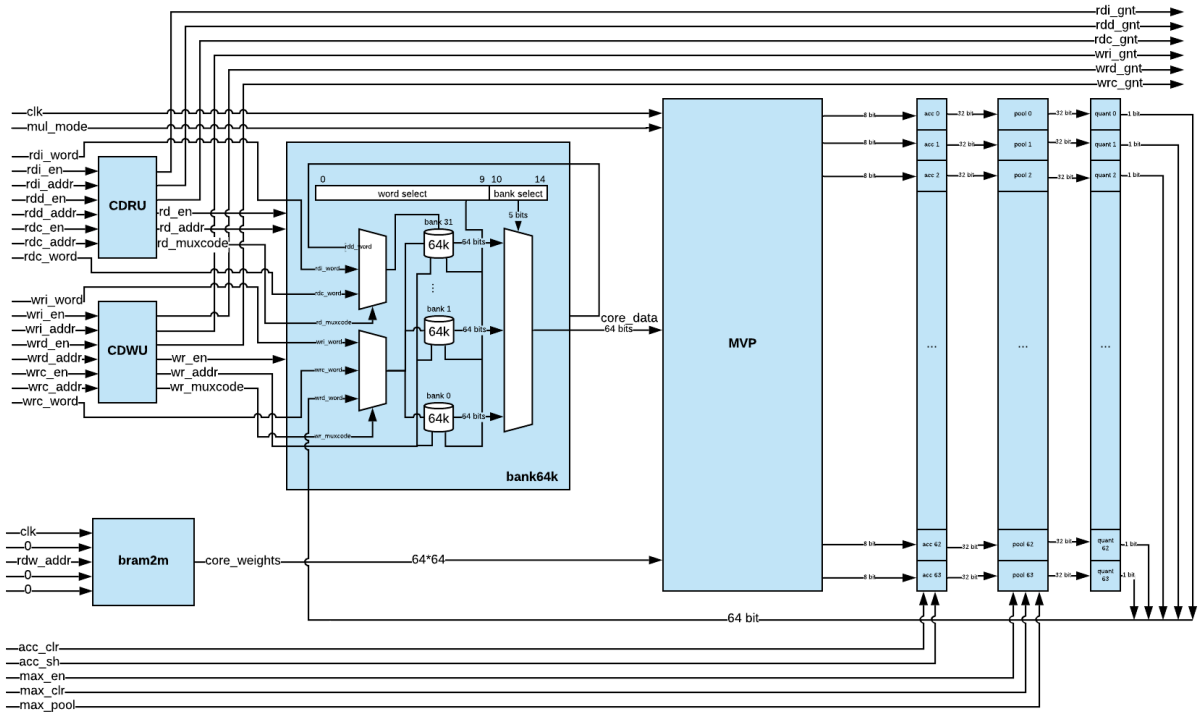


Fig. 2.2: This figure illustrates an MVU block diagram.

Fig. 2.2 illustrates the block diagram of an MVU. Each MVU is consist of a Matrix Vector Product unit (MVP), Collision Detection Read Unit (CDRU), Collision Detection Write Unit (CDWU), activation RAM, weight RAM and a set of machine learning specific blocks such as quantizers, scaler units and pooling unit that can be switched on or off (technically, data will pass through all of these blocks and the user should provide proper configuration to bypass the functionality. For instance for *scaler* unit, if there is no need to scale the output, the user should write *1s* in scaler RAMs) depending on the job configuration. As it can be seen in Fig. 2.2, at each clock cycle, an MVU word (64 bits) is read from the activation RAM. At the same time, a long word of 4096 bits (64 by 64) is read from weight RAM. This is then fed into MVP unit which can perform one binary matrix-vector product each clock cycle. Depending on the precision configuration register (take a look at *MVU_CSR_REG_TABLE* for detailed register configuration for each MVU), multiple words will be read from weight and data memory to perform bit-serial multiplication.

Fig. 2.3 illustrates bit-serial operation in MVU. As it can be seen, an MVU data word of size 64 bit is read from data RAM. This will be fed into 64 bit-serial multiplication blocks. Each of these blocks performs a dot product between the two vectors. Fig. 2.3 shows only one bit-slice operation in the MVU, however, in reality, there are 64 modules that perform the same task on input data but with different weight vectors. For more information on MVU bit-serial

operation, please refer to "Bit-Slicing FPGA Accelerator for Quantized Neural Networks" by O. Bilaniu et al.

As we mentioned before, the MVU is capable of performing computation with different bit precision. The way we achieve this task is by storing values in MSB transposed format in memory. This format of saving data in memory allows MVU to read-only as many words as the operand precision specifies. Since all the computations are happening in this format, the user should not worry about memory layout except when it wants to read results or write inputs (such as input image) into MVU RAMs. To solve this issue, there is a data transposer module that transposes the data to the correct format. Data transposer's job is to write input data (that is stored in a processor RAM in linear format) into MVU RAM in a transposed format. The input word can be packed with 2, 4, 8 or 16 bits of data. Given the input data precision (prec) the transposer will unpack, transpose and store them in the correct format. Once the MVU word is prepared, data tranposer will go into *BUSY* state in which it will ignore any incoming new input data. At this point, the transposed data will be written into MVU word. Once complete, it will go back into *IDLE* state and it will wait for a new posedge on start signal to start the process all over again.

2.2.1 MVU Job Configuration

MVUs are programmed to perform a single job. A job is started by the controller by raising the *start* signal. Once the job is finished, the MVU will generate an interrupt, informing the controller that the requested job is finished and the results are ready to be sent back to the host or to other MVUs. Once MVU is busy with a job, the *busy* signal is raised. During this time, MVU can be programmed for the next job and raising the *start* signal will not initiate any new job.

Fig. 2.5 shows the timing diagram for sending a job to MVU. For sake of breavity, all config parameters are represented by *configs* signal. In the following sections, we will review what parameters can be set in the MVU.

2.2.2 Feature map memory access

Fig. 2.6 illustrates the memory layout for feature maps. MVU expects a NHWC layout for feature map features. Each element should be stored in a MSB transposed format. Fig. 2.6 shows that each word is 64 bit. As a result, accessing memory at location *0* will return a 64-bit word, where each bit, belongs to the MSB bit of the first 64 channels of the feature map. Elements of these 64 channels are concatenated (in MSB transposed format) together to form a channel block. The next memory address i.e *1* will return the *MSB-1* bits of the first 64 channels. This pattern continues until we reach the configured input precision i.e. *iprecision*.

Elements of each channel are written into feature map memory with an offset of *iprecision*. In case there are more than 64 channels in the feature map, we will store the first 64 channels in the first block, the second 64 channels into the second block and so on. As an example, an input tensor of *[N=1, H=8, W=8, C=256]* with 2-bit precision, will have 4 channel blocks, each block will have 64 rows of 2 by 64-bit elements.

2.2.3 Weight Memory Access

Weight memory layout is very similar to feature map memory layout. Fig. 2.7 illustrates the weight memory layout. Same as Fig. 2.6, MVU expects a NHWC layout for weight tensor. However, in weight memory, we have input and output channels. By default, weight memory words are 4096 bit long. Allowing to concatenate a single MSB bit of *64x64* channels per row of weight memory. In deep neural network models, weight tensors are usually consist of a set of filters. The weight memory layout in MVU allows concatenating 64 input channels into 64 set of filters i.e. output channels. Like feature map memory layout, in case we have more than 64 input channels, we will write them into the next input channel blocks. Instead of *iprecision*, here we use *wprecision* to specify how many bits are required to represent any weight element.

Like feature map memory layout, channel blocks are grouped together to form width columns and then height rows. Finally, we can group multiple height rows together to form output channels i.e. filters.

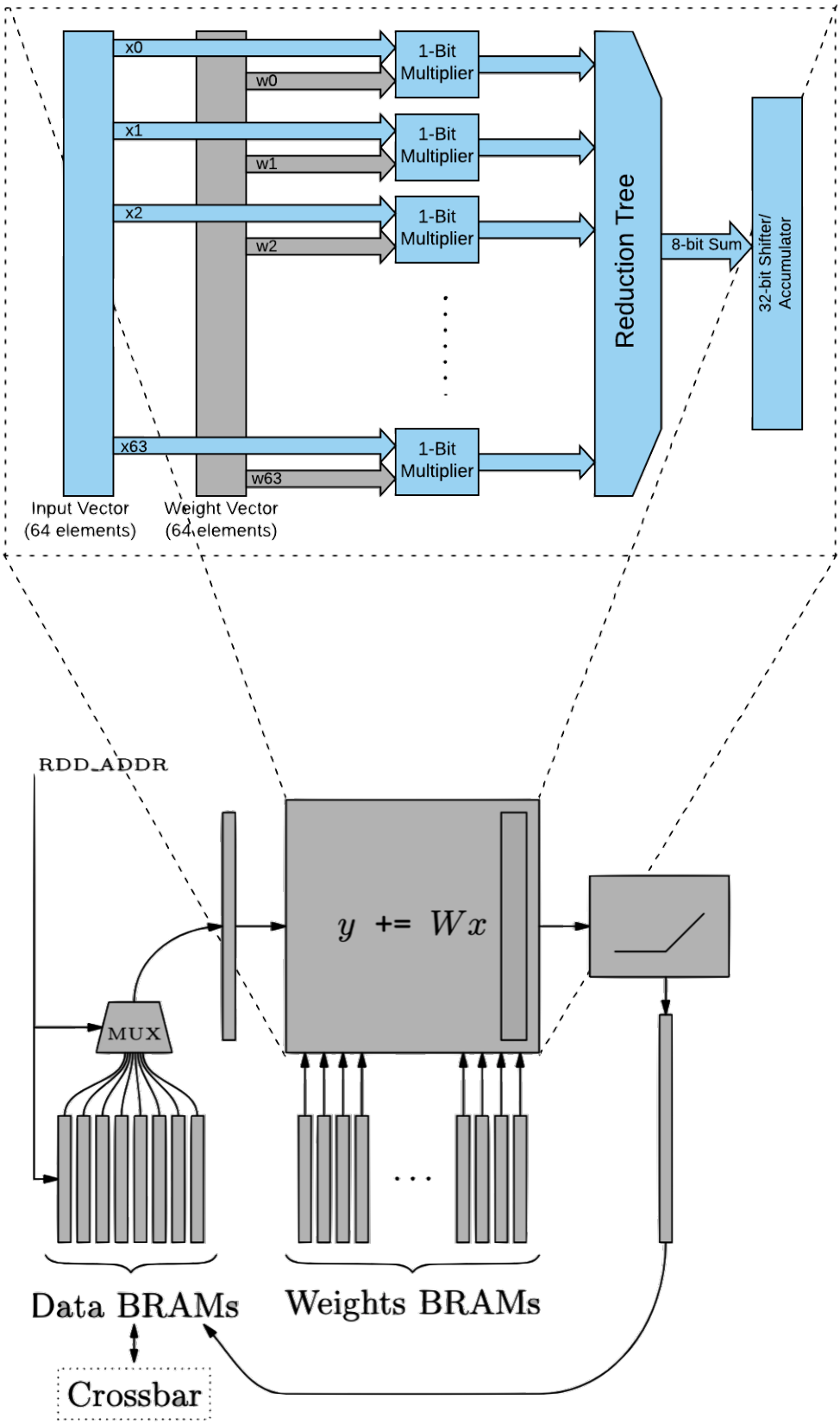


Fig. 2.3: Bit serial operation in MVU.

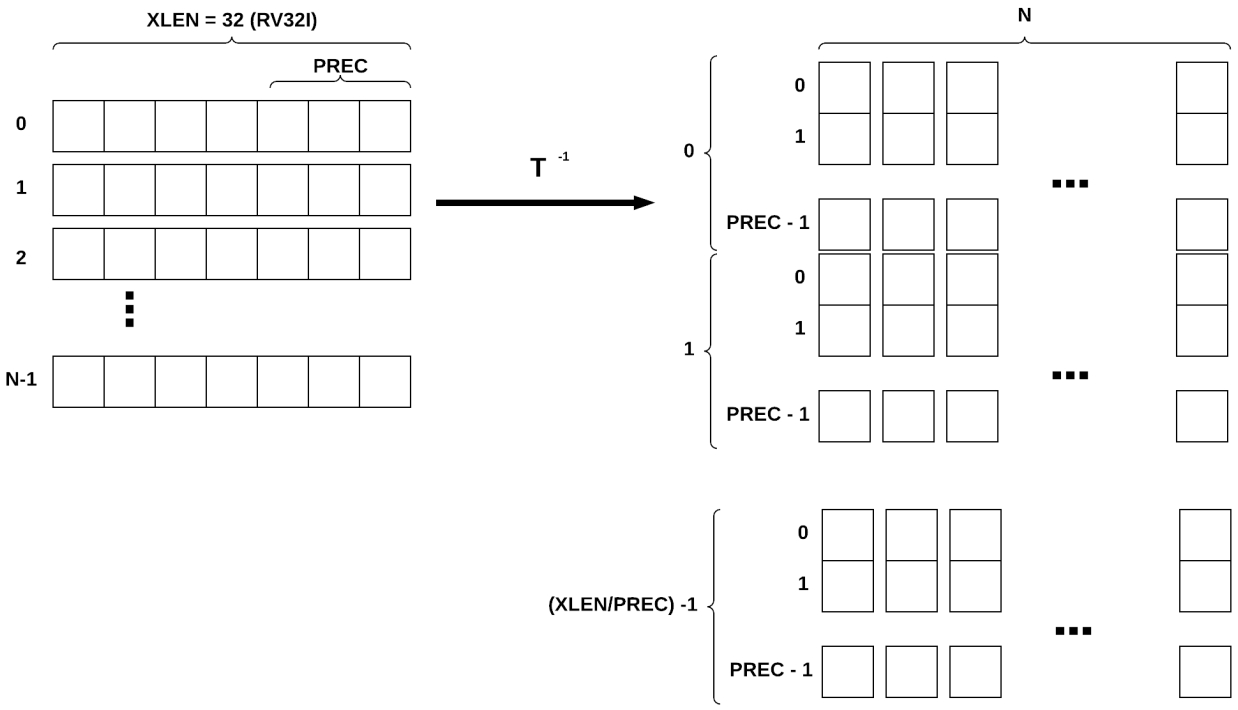


Fig. 2.4: Data transposer modlue, this module will pack vectors of size *XLEN* in MSB first transposed format.

Fig. 2.5: Timing diagram for configuring an MVU job.

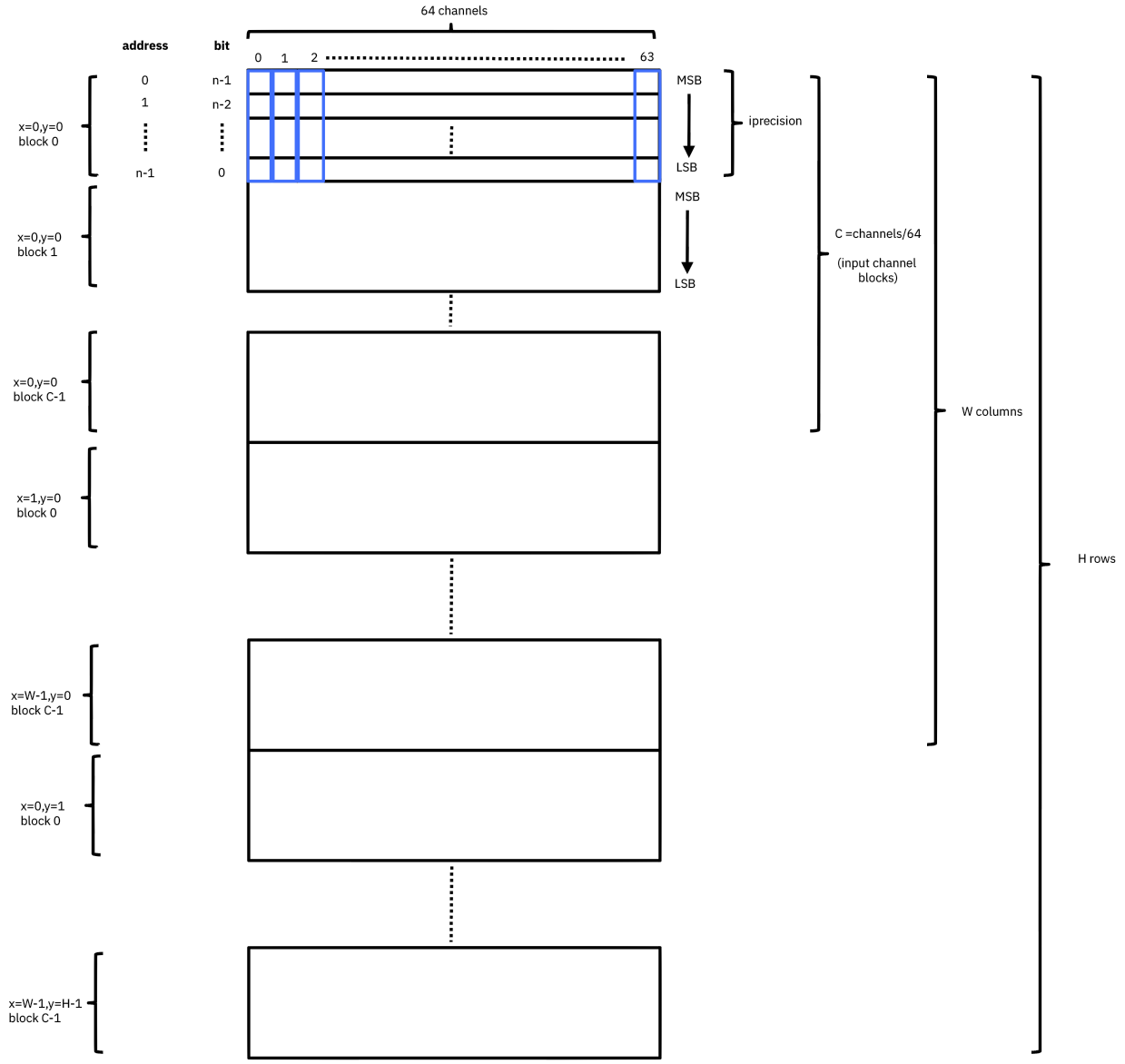


Fig. 2.6: Input feature map memory layout.

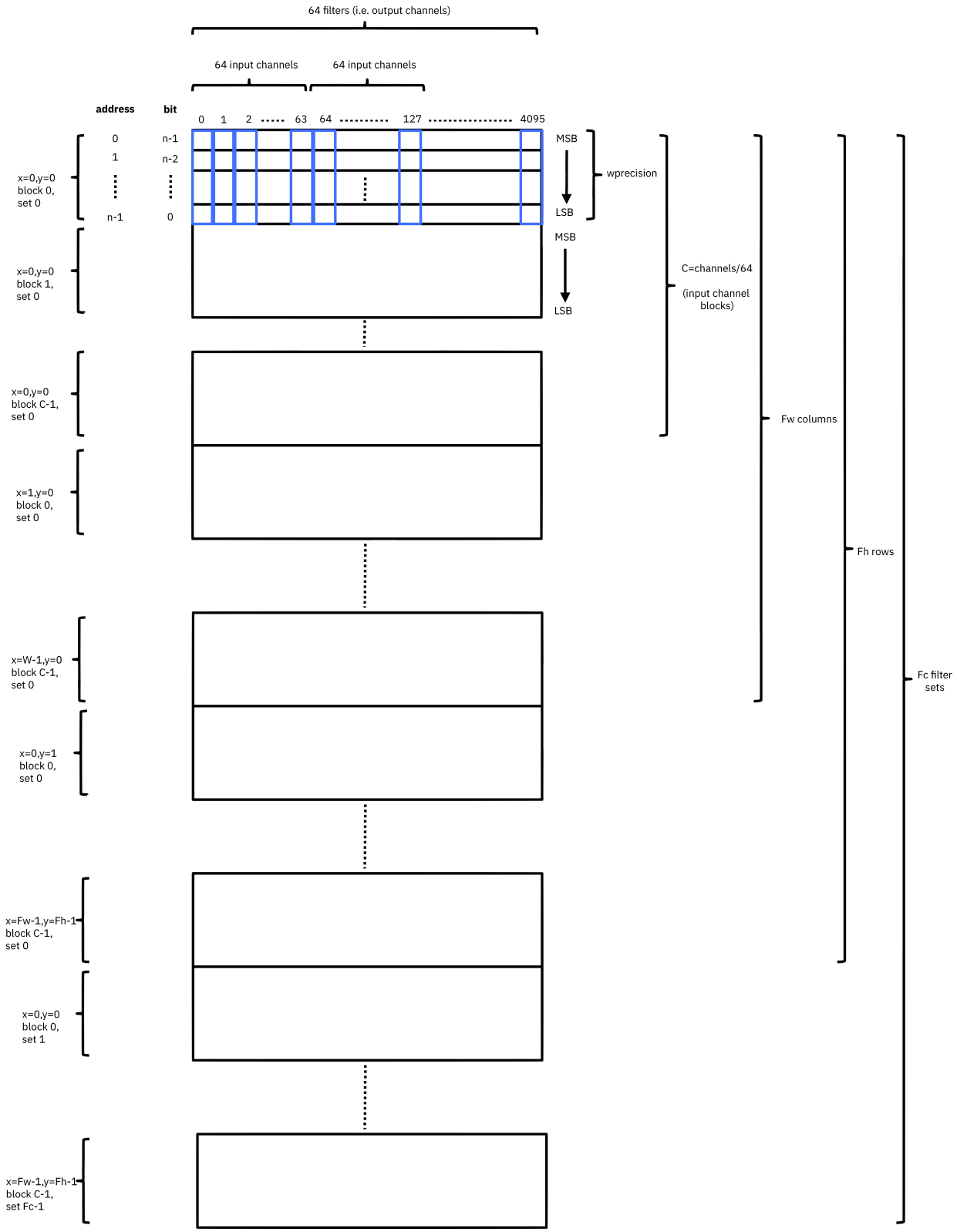


Fig. 2.7: Weight memory layout.

2.2.4 Jump Schedules

The memory layout described in previous sections allows MVU to efficiently compute matrix multiplication between input vectors and the weight matrices. However, a convolutional neural network, many matrix multiplies should be performed. One of the most common ways to perform convolution is to slide the weight tensor over input. Fig. 2.8 illustrates this operation.

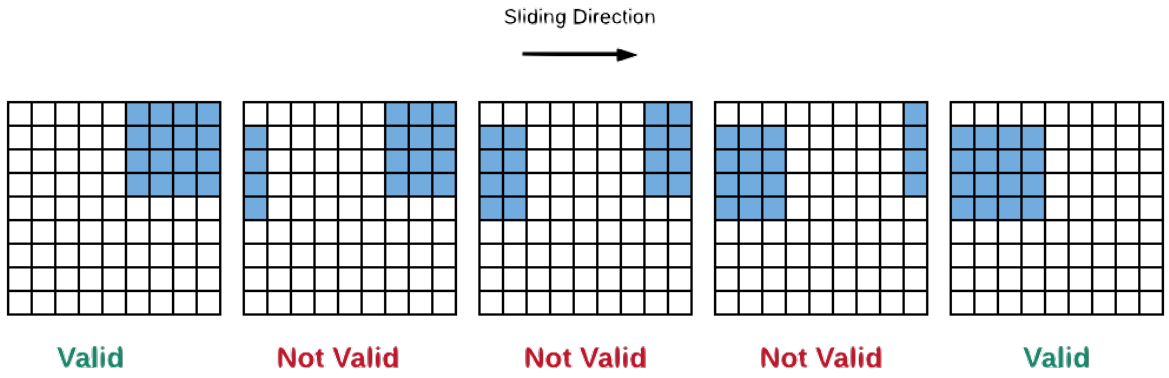


Fig. 2.8: Sliding window operation to perform Convolution.

As you can see in Fig. 2.8, if we just slide the weight tensor over input, not all dot products are valid. Luckily, for a given stride, padding and weight shape, we can pre-compute the pattern of memory accesses by the MVU to compute an operation such as GEMV or convolution. Each MVU includes address generators that can be programmed to implement a series of nested loops that can be used to move across the input data and weight tensors. Address generators have a set of *length* parameters that set the bounds of each nested loop, and a set of associated address *jump* (*jX*) parameters that are used to compute the next memory address to move to in a given loop. This is illustrated in the following pseudocode:

```
while (1) {
  for (i1 = length1; i1 > 0; i1--)
  {
    for (i2 = length2; i2 > 0; i2--)
    {
      for (i3 = length3; i3 > 0; i3--)
      {
        for (i4 = length4; i4 > 0; i4--)
        {
          addr_out += j4;
        }
        addr_out += j3;
      }
      addr_out += j2;
    }
    addr_out += j1;
  }
  addr_out += j0;
}
```

For a 2D convolution operation, Fig. 2.9 and Fig. 2.10 illustrates what each jump configuration is:

For inputs we have the following configurable *jump* variables:

- *j3*: jump over precision length for input data (i.e. set to *iprec*).
- *j2*: Specifies if we have reached window width, if so, move to the next row in the window.
- *j1*: Specifies if we have reached window height and width, if so, move back to window start for next precision combo or next filter set (i.e. for same output (x,y), start computing next output channel block).
- *j0*: Specifies if we have finished all filter sets in the window and done output (x,y). Slide window by horizontal stride. Start output (x+1, y). Note that the diagram shows a horizontal stride of 1.
- *j4*: not applicable.
- *j3*: jump over precison length for weights (i.e. set to *wprecision*).
- *j2*: Specifies if we have reached window width and height, if so, move back to filter start for next precision combo.
- *j1*: Specifies if we have finished all bit combos for the current filter set and channel block for output (x,y) and if so, move to the next filter set and compute the next channel block for output (x,y).
- *j0*: Specifies if we have finished all filter sets and channel blocks for output (x,y) and if so, move back to the start of the first filter set for the next window and output (x+1, y).
- *j4*: not applicable.
- 16-bit fixed point values
- Standard bit ordering, i.e. non-bit-sliced, little-endian
- Each channel block is 64 channels
- n channel blocks in a layer; would be same as Fc in conv or bn following conv
- 32-bit fixed point values
- Only lower 27-bits are used in addition (due to FPGA DSP structure)
- Standard bit ordering (i.e. non-bit-sliced), little-endian
- Each channel block is 64 channels
- n channel blocks in a layer; would be same as Fc in conv or bn following conv

In general, each MVU has 44 configurable registers that can be used in the software. Section *Control Status Registers (MVU)* provides details of each register.

2.3 PITO: A Barrel RISC-V Processor

To make use of MVUs for neural networks, some form of the control unit is required. It is not possible to foresee and provide for all possible neural networks that may crop up in the literature in the future. Therefore, the high-level sequencing of tensor operations should be provided for in software, possibly assisted by *glue* logic to help drive the MVUs' control signals.

PITO is a Barrel RISC-V processor, designed to control the 8 MVUs in *Bilaniuk et al. (2019)* using separate but communicating hardware threads (harts) that each manages their respective MVUs. Neural network layers can then be executed either in parallel or in a pipelined fashion depending on whether the neural network software is compiled to maximize throughput or minimize latency. This design also allows MVUs to complete tensor operations independently of each other. However, the drawback is that, at least nominally, this requires 8 microprocessors to execute the 8 programs, putting serious pressure on the remaining logic of the host FPGA. We instead amortized the fixed costs of the processor by adopting an old idea: *the barrel processor*. By making the barrel processor 8-way threaded, we may

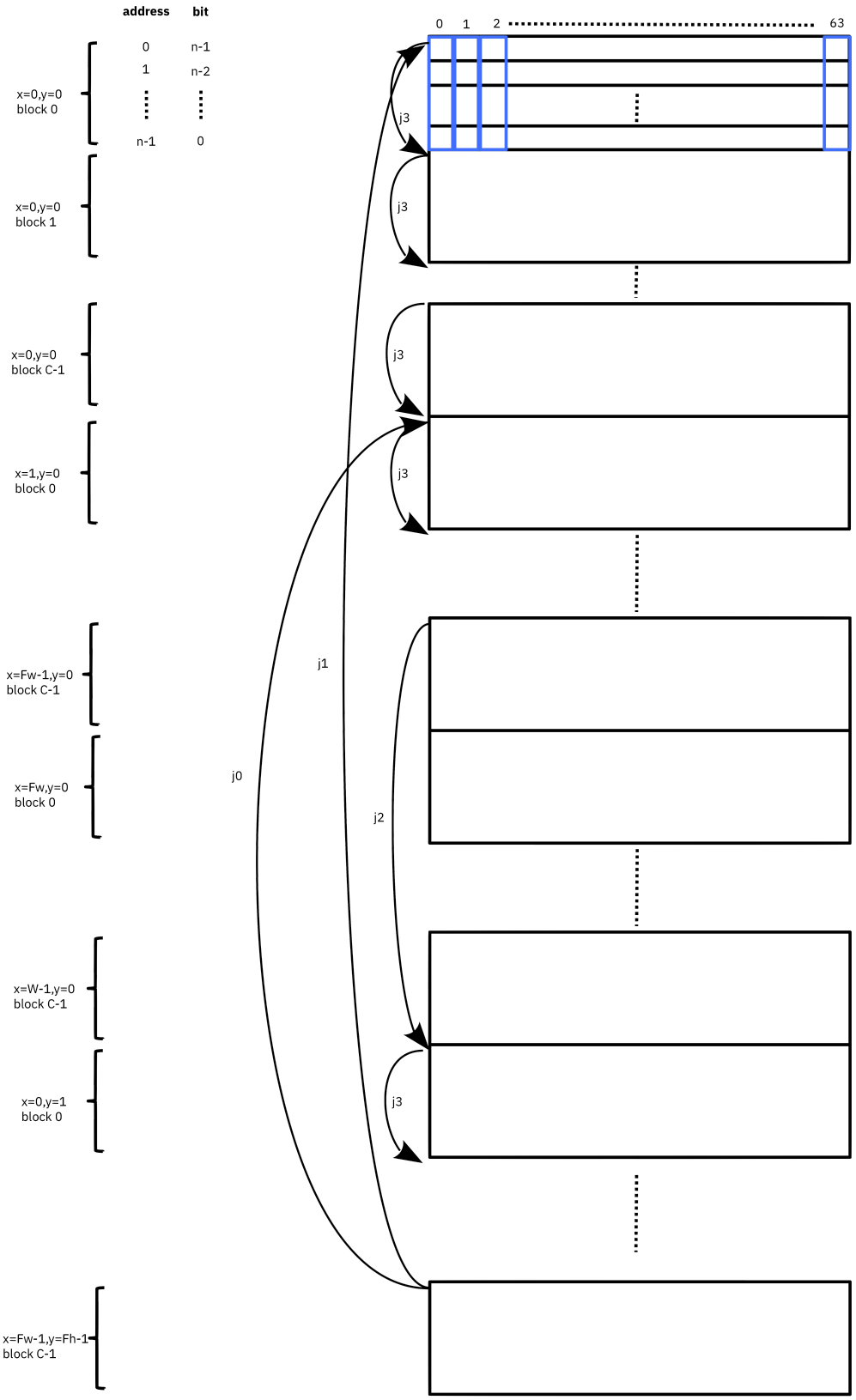


Fig. 2.9: Input feature jump schedule.

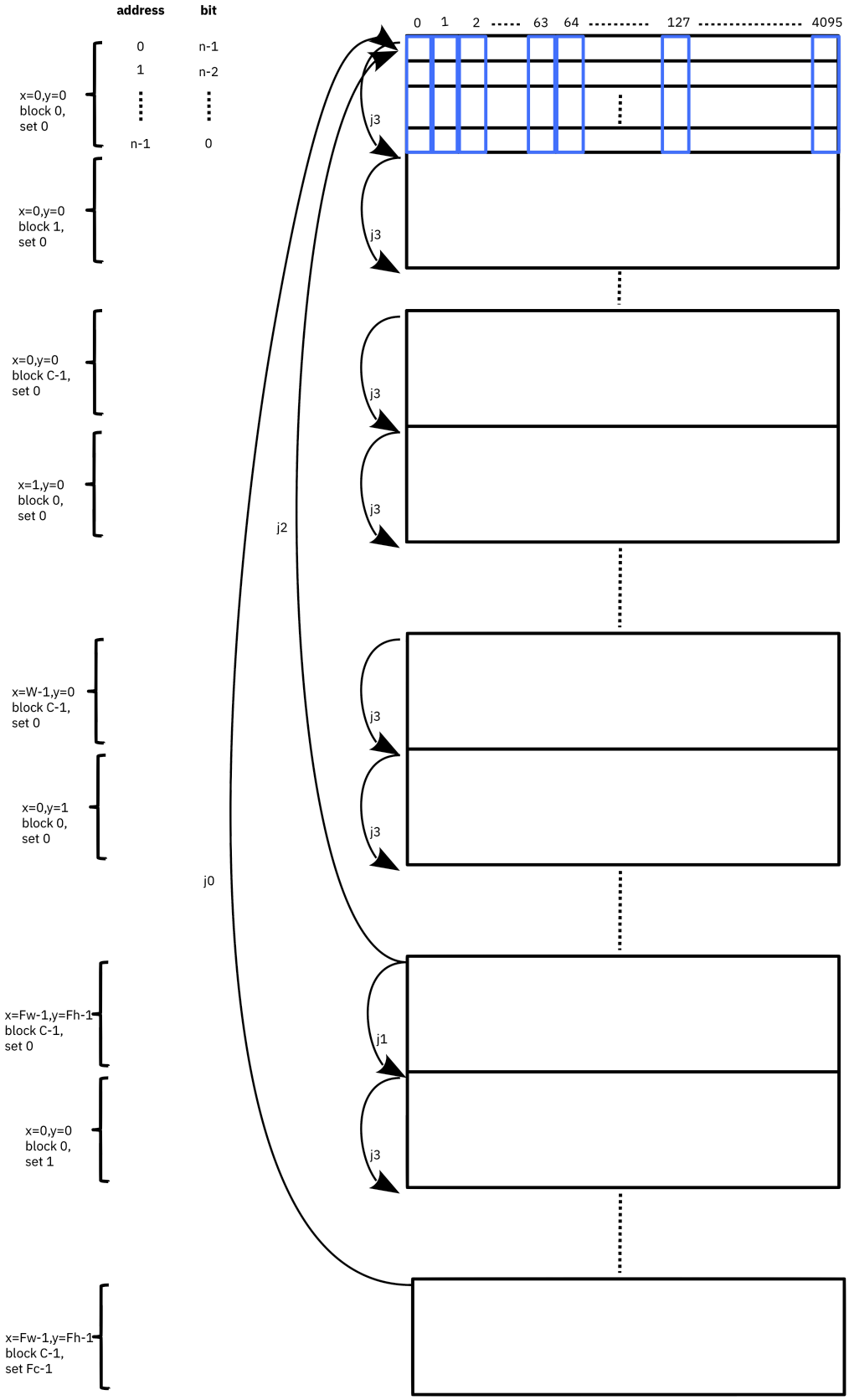


Fig. 2.10: Weight jump schedule.

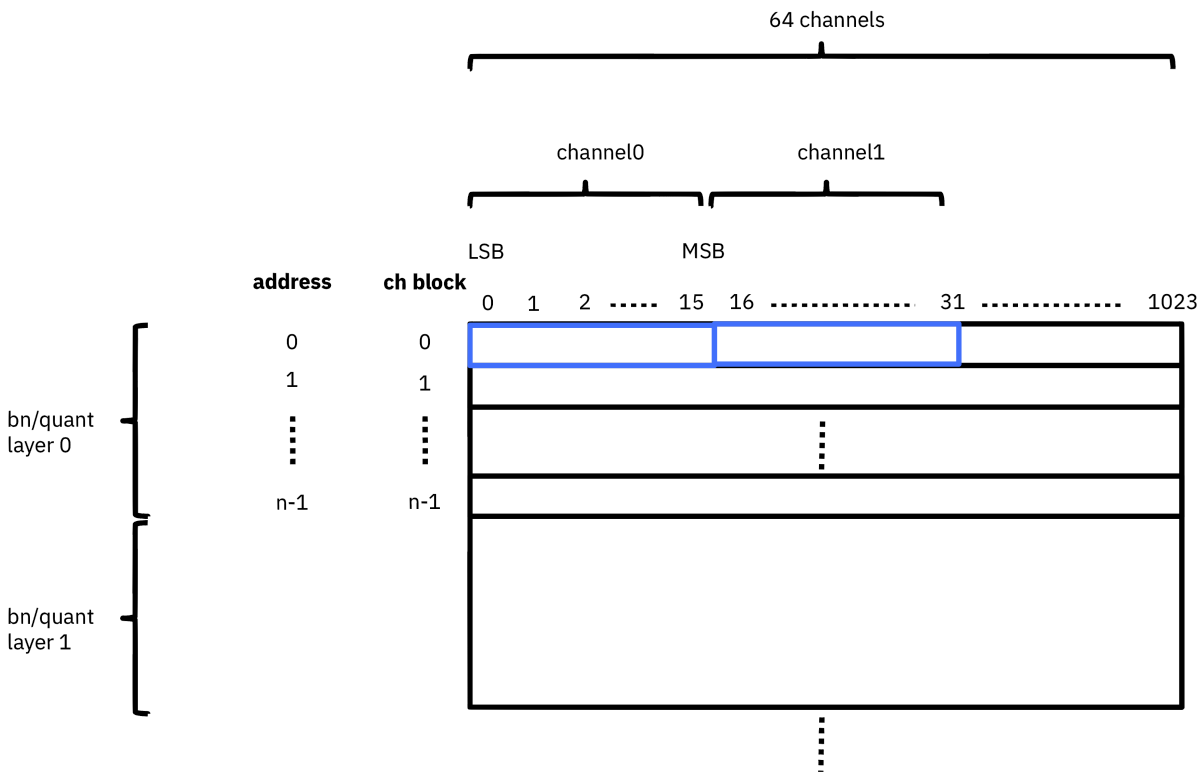


Fig. 2.11: Quantizer/BN weights

assign one thread to control each of the MVUs, while amortizing the fixed costs of each microprocessor over the 8 threads. Because every thread comes up for execution only every 8 clock cycles, up to 8 pipeline stages including instruction fetch, decode, execution and data read & writes can be completely hidden. Branch prediction units are also made unnecessary. Because even modest tensor operations can require hundreds of matrix-vector products (and therefore clock cycles) to execute on an MVU, the barrel processor has the opportunity to fully turn over dozens of times in the interim, allowing each thread to issue the next command to its MVU in a few instructions.

A barrel processor is a form of a fine-grain multithreading processor that exploits thread-level parallelism by switching between different threads on each clock cycle (Hennessey and Patterson,2011). The aim is to maximize the overall utilization of the processor's resources, and instruction throughput. This is similar to the technique of simultaneous multi-threading (SMT) that is used in modern superscalar processors. However, unlike SMT superscalar processors, barrel processors do not issue more than one instruction per clock cycle. Instead, a single execution pipeline is shared by all threads. Fig. 2.13 illustrates the data path of *PITO*, a 5 stage 8 hart, barrel processor compatible with RV32I RISC-V ISA.

We adopted a Harvard architecture and divided the instruction and data cache. In our design, we used 32KB BRAM for each cache. This gives a 1K word space to store data and instructions to control each MVU. The processor is an in-order CPU and instructions are executed following compilation order and without any further scheduling. However, a hart scheduler is needed to give access to the required resources for the hart at each stage. In the fetch stage, each hart needs to fetch instructions from the instruction cache. As explained earlier, we used 32KB of instruction cache which is shared between all harts. However, the program counter (PC) for each hart is different. To keep track of this, we used 8 registers for PCs and the hart scheduler indicates which register should be accessed at any given time. In the Decode stage, the fetched instruction needs to be decoded, and source registers (rs1 and rs2) or an immediate (imm) operand needs to be loaded. Each hart has its own register file and in the Decode stage, the hart scheduler gives access to the scheduled hart's register file.

The hart scheduler itself uses a strict round-robin algorithm. No preemption or priority is implemented and every hart

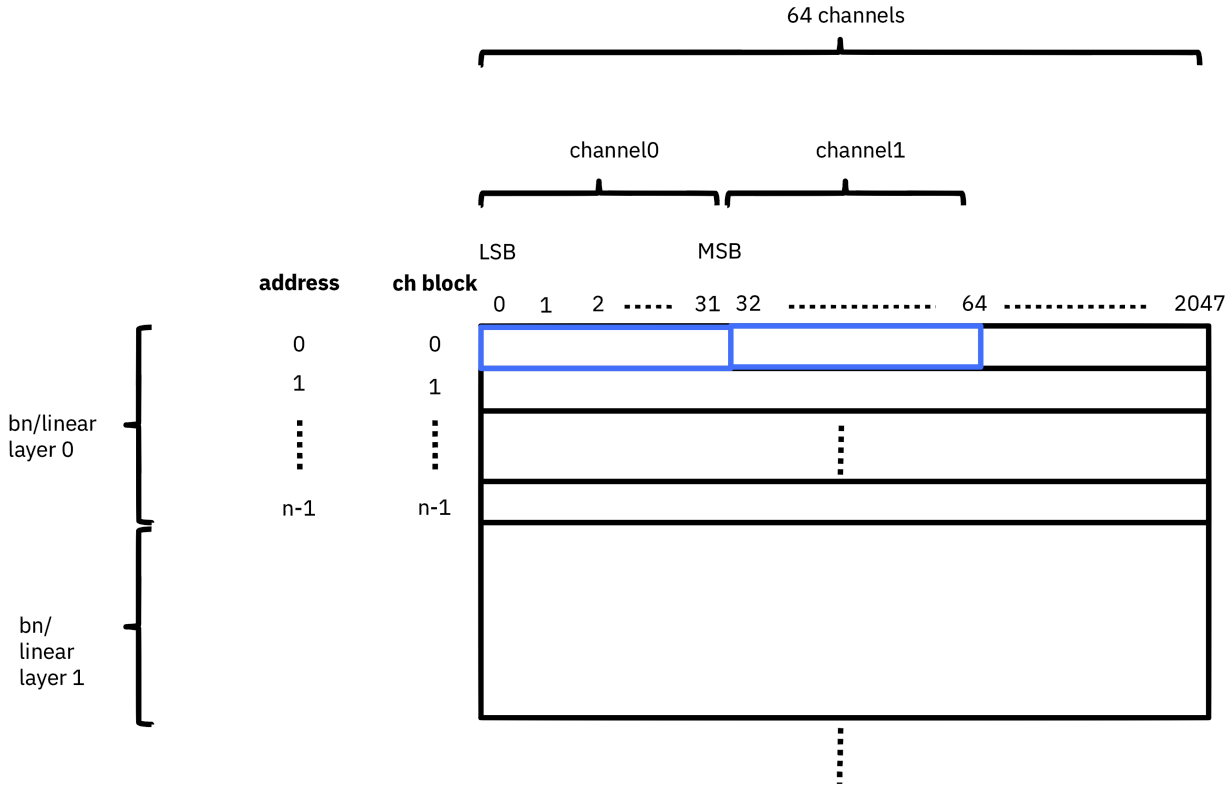


Fig. 2.12: BN/linear biases

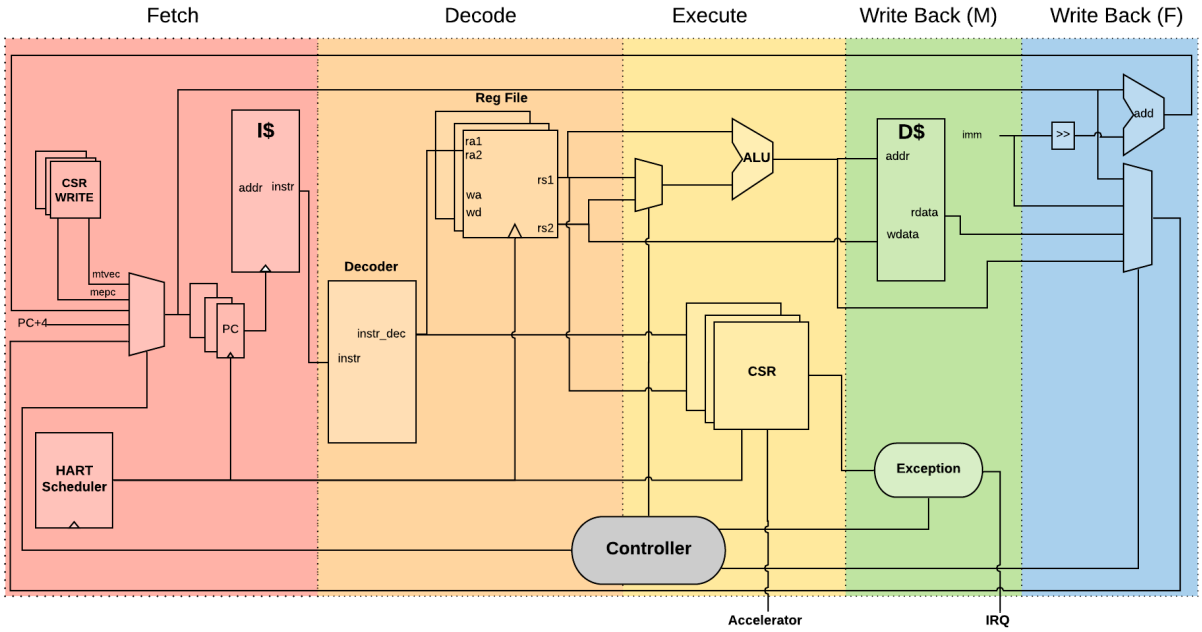


Fig. 2.13: PITO Datapath, a 5 stage 8 hart, barrel processor.

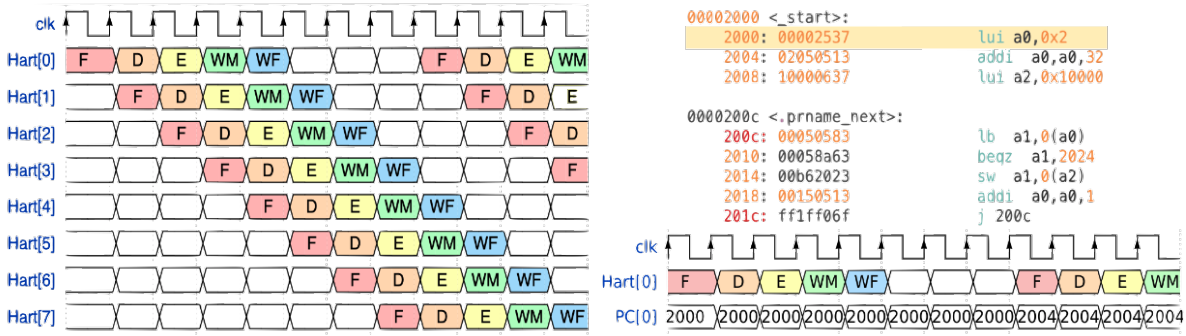


Fig. 2.14: This figure shows 8 harts running in the barrel processor that has 5-stage pipeline. The figure on the right shows every 8 clock cycles, the program counter of the associated hart increments, which allows this pipeline to be implemented without any data or control hazard circuitry.

is given a fixed amount of time slots for execution. Figure 4.3a shows how harts are scheduled for execution in our design. Considering the execution for Hart[0], it takes 5 clock cycles for an instruction to be completed. After the 5th clock tick, no more processing associated with Hart[0] is performed. The next three slots are given to Hart[5], Hart[6] and Hart[7]. Thus each hart executes an instruction every 8th cycle of the main clock. Hence the CPI of 8. From the perspective of the main CPU, the throughput is one instruction per clock cycle. From the perspective of each hart, we are running at an 8th of the main clock speed with a CPI of 1.

PITO is compatible with RV32I RISC-V ISA. In fact, PITO passes all the RISC-V tests, confirming that it is compliant with the RV32I ISA. In addition to base CSRs (refer to *Control Status Registers (RISC-V)* for details) and to specialize PITO for our accelerator, we have added 44 MVU specific CSRs. In Section *Examples*, we have provided example codes to program these CSRs to submit a job to MVU.

2.3.1 Interrupts

In BARVINN, MVUs can send interrupts to their associated hart. These interrupts are added to RISC-V custom interrupts *mie* field. To reduce complexity, there are no supports for nested interrupts or interrupt priorities. However, we followed RISC-V's interrupt operation flow. Fig. 2.15 illustrates servicing interrupt flow in software and hardware.

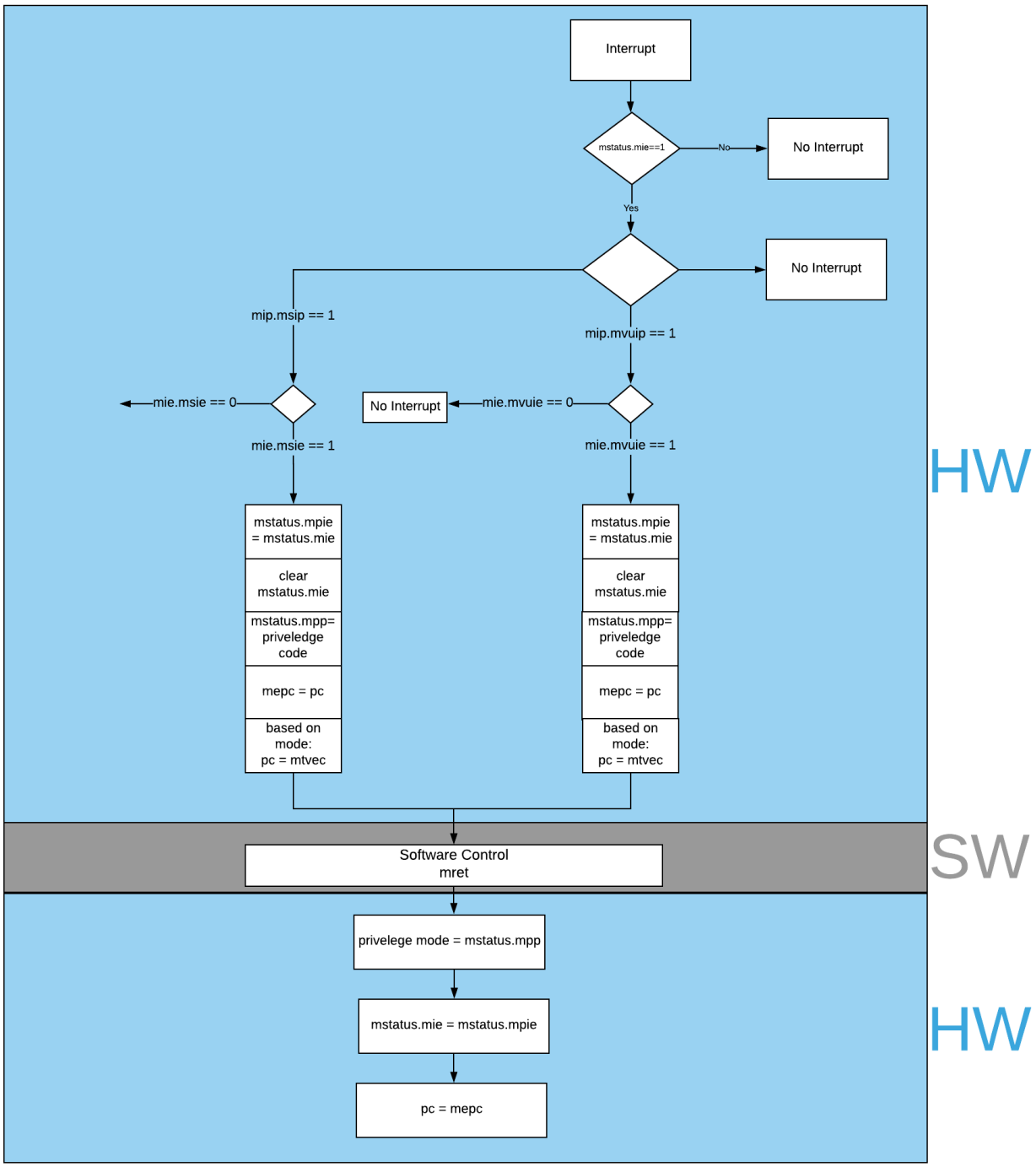


Fig. 2.15: Interrupt service routine in hardware and software

2.3.2 Control Status Registers (RISC-V)

ADR	CSR	RO/RW	Description
0x301	misa	RO	A constant, but MSB = 0 for open-source implementation..
0xF11	mvendorid	RO/Zero	Identification. Can be zero.
0xF12	marchid	RO/Zero	Identification. Can be zero.
0xF13	mimpid	RO/Zero	Identification. Can be zero.
0xF14	mhartid	RO, cycle counter % 8	Shared with cycle counter.
0x300	mstatus	RW, per-thread	Critically-important bits like Global Interrupt Enables
0x305	mtvec	RO or RW if wanted	Interrupt vector, or interrupt vector table base address. Register is RW if we want to be able to choose between these two modes, or change the address.
0x344	mip	RO, per-thread	Pending interrupts bitfield
0x304	mie	RW, per-thread	Enabled interrupts bitfield
0xB00	mcycle	RW per-thread	Cycles counter, low 32 bits
0xB80	mcycleh	RW per-thread	Cycles counter, high 32 bits
0xB02	minstret	RW per-thread	Instructions retired counter, low 32 bits
0xB82	minstreth	RW per-thread	Instructions retired counter, high 32 bits
0xxxx	mhpm*	RO/Zero	High-performance counter control registers, not supported
0xxxx	mcountinhibit	RO/Zero	High-performance counter inhibit, not supported
0x340	mscratch	RW, per-thread	Scratch register, necessary to support interrupts
0x341	mepc	RW, per-thread	Exception program counter
0x342	mcause	RW, per-thread	Interrupt cause
0x343	mtval	RW, per-thread	Stores either faulting address, or contains illegal instruction

2.3.3 Control Status Registers (MVU)

CSR	RO/RW	Description
mvuwbaseptr	RW	Base address for weight memory
mvuibaseptr	RW	Base address for input memory
mvusbaseptr	RW	Base address for scaler memory (6-bit)
mvubbaseptr	RW	Base address for bias memory (6-bit)
mvuobaseptr	RW	Output base address:
		0-23: address
		31-24: destination MVUs (bit 24 -> MVU 0)
mvuwjump[0-4]	RW	Weight address jumps in loops 0-4

continues on next page

Table 2.1 – continued from previous page

CSR	RO/RW	Description
mvuijump[0-4]	RW	Input data address jumps in loops 0-4
mvusjump[0-1]	RW	Scaler memory address jumps (6-bit)
mvubjump[0-1]	RW	Bias memory address jumps (6-bit)
mvuojump[0-4]	RW	Output data address jumps in loops 0-4
mvuwlength[1-4]	RW	Weight length in loops 1-4
mvuilength[1-4]	RW	Input data length in loops 1-4
mvuslength[1]	RW	Scaler tensor lengths(15-bit)
mvublength[1]	RW	Bias tensor lengths (15-bit)
mvuolength[1-4]	RW	Output data length in loops 1-4
mvuprecision	RW	Precision in bits for all tensors:
		0-5: weights precision
		6-11: input data precision
		12-17: output data precision
		24: weights signed (0: unsigned, 1: signed)
		25: input data signed (0: unsigned, 1: signed)
mvustatus	RO	Status of MVU:
		0: busy
		1: done
mvucommand	RW	Kick to send command:
		30-31: MulMode (00:{0,0} 01:{0,+1} 10:{-1,+1} 11:{0, -1})
		29: MaxPool enable
		0-28: Clock cycle countdown
mvuquant	RW	MVU Quantization Configs:
		6-11: MSB index position
		12-31: reserved (possibly for activation params)
mvuscaler	RW	0-15: fixed point operand for multiplicative scaling
mvuconfig1	RW	MVU General Configurations
		0-7: Shift/accumulator load on jump select (only 0-4 valid)
		8-16: Pool/Activation clear on jump select (only 0-4 valid)

2.3.4 mvuwbaseptr

2.3.5 mvuibaseptr

2.3.6 mvusbaseptr

2.3.7 mvubbaseptr

2.3.8 mvuobaseptr

mvuobaseptr output address, results of each operation will be written into this address. Destination MVU, results can be sent to other MVUs by setting the appropriate MVU (0 to7) field. The result can be broadcasted to any number of MVUs in the system.

2.3.9 mvuwjump

mvuwjump is the weight address jumps in loops 0-4. Hence, there are 5 registers all start with *mvuwjump_* but then to access a specific loop, you need to append the loop number at the end (refer to *Jump Schedules* section for details on loop count). For instance, for *loop1* one can use *mvuwjump_1*.

2.3.10 mvuijump

Same as *mvuwjump*, there are 5 loops that can be used to address input data. These loops can be accessed as *mvuijump_0* to *mvuijump_4*.

2.3.11 mvusjump

For scaler memory, we have only two jumps and they can be accessed as *mvusjump_0* and *mvusjump_1*.

2.3.12 mvubjump

For bias memory, we have only two jumps and they can be accessed as *mvubjump_0* and *mvubjump_1*.

2.3.13 mvuojump

Same as *mvuwjump*, there are 5 loops that can be used to address output memory. These loops can be accessed as *mvuojump_0* to *mvuojump_4*.

2.3.14 mvuwlength

There are 4 registers to specify weight length loops and can be accessed as *mvuwlength_1* to *mvuwlength_4*. Note, *mvuwlength_0* is intentionally not used.

2.3.15 mvuilength

There are 4 registers to specify input data length loops and can be accessed as *mvuilength_1* to *mvuilength_4*. Note, *mvuilength_0* is intentionally not used.

2.3.16 mvuslength

There is only one register to specify scaler tensor length and it can be accessed as *mvuslength_1*. Note, *mvuslength_0* is intentionally not used.

2.3.17 mvublength

There is only one register to specify scaler tensor length and it can be accessed as *mvublength_1*. Note, *mvublength_0* is intentionally not used.

2.3.18 mvuolength

There are 4 registers to specify input data length loops and can be accessed as *mvuolength_1* to *mvuolength_4*. Note, *mvuolength_0* is intentionally not used.

2.3.19 mvuprecision

weight precision, *input precision* and *output precision* indicates the computation precision accordingly. *isign* and *wsign* can be used to set if the data is signed *1* or not *0*.

2.3.20 mvustatus

Specifies MVU status which is either *busy* (0) or *done* (1).

2.3.21 mvucommand

Setting any value to this register will send a kick start signal to MVU to start the configured job. The register fields are described in *Control Status Registers (MVU)*.

2.3.22 mvuquant

In the case we need to quantize results, *msbidx* can be used. This field indicates that where does the *msb* position start.

2.3.23 mvuscaler

A fixed point multiplier value that can be used to rescale a quantized value.

2.3.24 mvuconfig1

CHAPTER

THREE

VERIFICATION

To verify the functionality of our design, we have created a verification environment. We used Vivado's support for Systemverilog. Although UVM verification was a much better choice to start with, however, when we started the project, there was no support for UVM in Vivado. Currently, Vivado 2020 supports UVM based verification but our verification is still based on a simple class-based verification. In the following sections, we provide an overview of what has been implemented. Also, we will review how new tests can be added and how to run simulations.

3.1 Verification Environement

Fig. 3.1 illustrates the overall architecture of our verification environment. For simplicity, we will review the verification environment in PITO. However, both MVU and BARVINN follow the same verification structure. There are four main verification components in our verification design:

- Testbench Monitor
- Testbench Predictor
- Testbench Base
- Testbench Top

3.2 Testbench Monitor

In our design, a *Testbench Monitor* is a testbench module that monitors transactions inside the DUT. In PITO the *Testbench Monitor* module is named *pito_monitor*. An important task of *pito_monitor* is to sync with the DUT. This is a crucial step since our predictor module and DUT must be in the exact same state to allow the predictor module to correctly predict the next state. *pito_monitor* syncs to DUT by checking if the first instruction in the firmware is correctly executed by the DUT. Once the monitor found such an instruction, it will move to the sync state. Otherwise, it will wait for a predetermined wait period *NUM_WAIT_CYCELS* until it times out and halts the simulation.

Once the sync period is done, *pito_monitor* samples data for the predictor module. In PITO verification environment, *pito_monitor* class instantiates a *RV32IPredictor* module and uses hdl path and *pito_interface* to monitor transactions within the DUT. On every clock cycle, *pito_monitor* samples the DUT's CSR, register file and memory as well as the executed instruction. It then passes all these samples to the *RV32IPredictor* module.

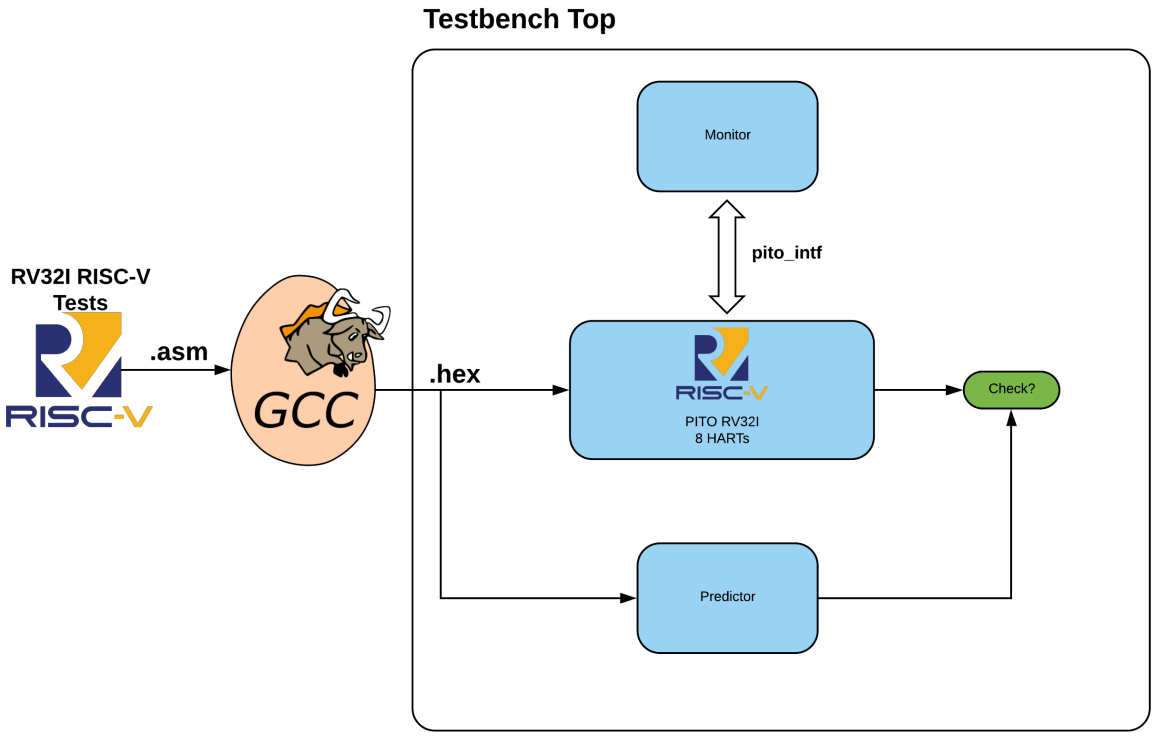


Fig. 3.1: Verification Environment for PITO, showcasing how RISC-V tests are verified in our design.

3.3 Testbench Predictor

A testbench predictor module is responsible to predict the state of each hardware block in the DUT. There are many industry proven predictors. As an example, Spike is a RISC-V simulator that provides functional model for different RISC-V ISA. It can also be integrated with Systemverilog through SyetemVerilog DPI. However, we decided to write our own functional model of PITO since Spike does not support a Barrel design. As an example, in PITO, there is no logic to prevent data or control hazards. On the other hand, although Spike supports multiple harts, a normal data path with data and control hazard has been implemented. Hence, Spike in its default format is unable to correctly predict the DUT behavior.

RV32IPredictor module is a functional model of PITO written in SystemVerilog. It supports all base RV32I instructions. For every instruction that is executed in the DUT, our predictor can predict the expected results. *RV32IPredictor* is designed to support as many harts as is required. It also contains the base RISC-V CSRs plus the custom CSRs that we added for configuring the MVU.

The *RV32IPredictor* module has no direct connection to the DUT. All the transactions are sampled by the monitor module and then they are provided to the predictor module. Hence, as mentioned before, it is crucial for the monitor module to sync correctly with the DUT. Once the predictor module receives a sample from the monitor, it will process the instruction and it will update the *test_stat* variable to be used by the testbench.

3.4 Testbench Base

Testbench base is a SystemVerilog class that contains testbench predictor and testbench monitor class. Figure Fig. 3.2 illustrates the class structure for our verification environment. As it can be seen, the testbench base class (*pito_testbench_base* in PITO verification environment) should be used as the base class for all other test classes. Each test has three phases, testbench setup phase, testbench run phase and testbench report phase. All these phases are virtual tasks that allow the user to override them.

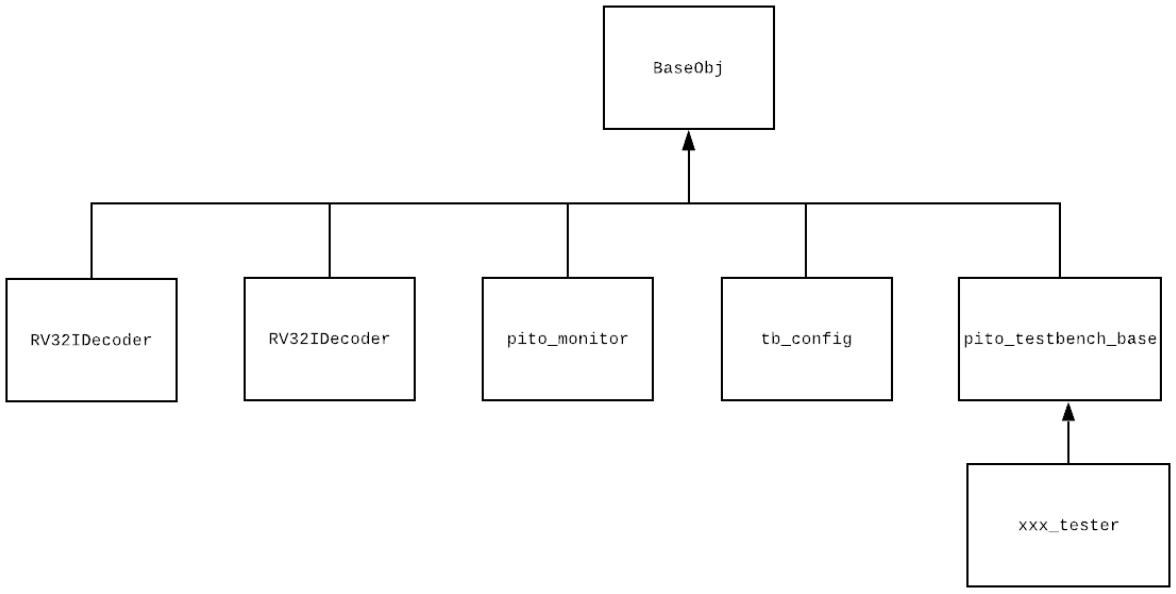


Fig. 3.2: Class structure of PITO verification

In the testbench setup phase, we usually put the DUT into reset mode and we will provide reset configurations. At

this stage, we should load any firmware into the instruction ram and load data ram with the data generated by the compiler. We should also initialize the register files and prepare the start sequence for the processor. However, this can be overwritten by the test in case there are other things that need to be added. In the testbench run phase, we usually run the firmware on the DUT. However, before that, we should kick start the monitor to sync with the DUT. Finally, in the report phase, we report the result of the test. The testbench base class has a *test_stat* variable that is passed to the monitor class. In the report phase, we will use this data structure to report the result of the test.

3.5 Testbench Top

Unlike the previous testbench components, the testbench top is a SystemVerilog module. As it can be seen in Fig. 3.1, the testbench top module instantiate all the other components (DUT, tests, interface). It also connects the DUT to the testbench through the interface. Another important task of this module is to call the three phase of the testbench that was described earlier. The testbench top module also provides the clock signal for the entire system.

3.6 Running a Test in PITO Verification Environment

Our design supports FuseSoC. In order to run any of the tests provided, you will first need to make make sure that the Vivado is available in the system. We currently support Vivado 2019.1:

```
source /opt/Xilinx/Vivado/2019.1/settings64.sh
```

Then, make sure you have fusesoc installed:

```
python3 -m pip install fusesoc
```

Then add pito to your fusesoc libraries:

```
git clone https://github.com/hossein1387/pito_riscv.git
cd pito_riscv
fusesoc library add pito .
```

Then run simulation (No GUI):

```
fusesoc run --target=sim pito
```

For synthesis:

```
fusesoc run --target=synth pito
```

To open sim in GUI mode:

```
cd build/pito_0/sim-vivado/
make run-gui
```

And for synthesis:

```
cd build/pito_0/synth-vivado/
make build-gui
```

This should open the project for you. Make sure you have run simulation or synthesis at least once, otherwise FuseSoC would not create a project file for you.

CHAPTER

FOUR

SOFTWARE STACK

As mentioned earlier, PITO is compliant with RV32I RISC-V ISA. Hence, all the toolchains developed for RV32I can be used. However, there is still a huge gap for running a high-level neural network model described in Pytorch, Tensorflow, or ONNX on a Neural Network accelerator such as BARVINN. Fig. 4.1 shows how we try to close this gap.

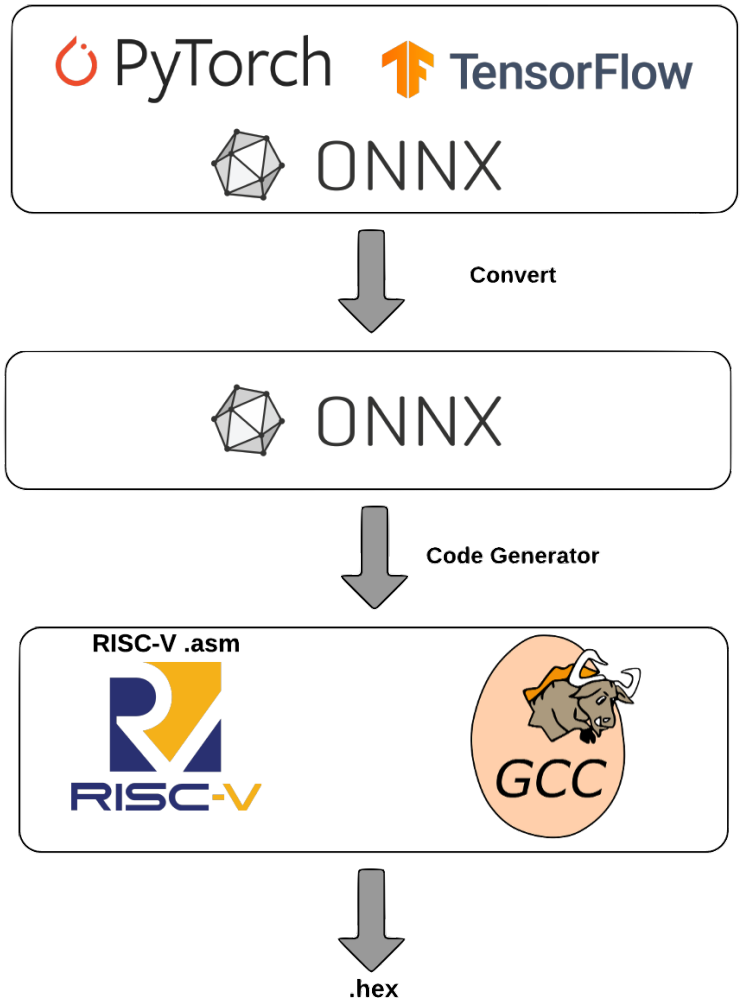


Fig. 4.1: Software stack used in BARVINN.

Given a model trained in Pytorch, Tensorflow or any other machine learning framework, we first need to convert it to an ONNX model. ONNX is an open format for representing machine learning models. Using ONNX models allows us to write use a single code generator module for all types of machine learning models described in any machine learning framework.

Before using the code generator, we should first quantize the model. Currently, model quantization is a hot research topic. The main goal of quantization is to reduce calculation precision while maintaining accuracy. Quantization can be applied to a model after or while training. Quantization after training (post-training quantization) can be done statically or dynamically. In post-training static quantization, weights are quantized ahead of time and during a calibration process on the validation set, a scale and bias is computed for the activations.

In post-training dynamic quantization, much like post-training static quantization, the weights are quantized ahead of time but the activations are dynamically quantized at inference. Dynamic quantization is useful for models where model execution time is dominated by the time it takes to load weights for the model e.g LSTM.

Quantization can also be learned by the network. In quantization-aware training, the quantization parameters are learned while other parameters in the network are learned.

There are many quantization methods proposed in the literature. However, although they are very different in training, at inference, these methods usually use a scaling factor and clip function to quantize the value. As an example, in Learned Step Size Quantization SK Esser, et.al (2020), the authors provided the following Fig. 4.2 computational graph:

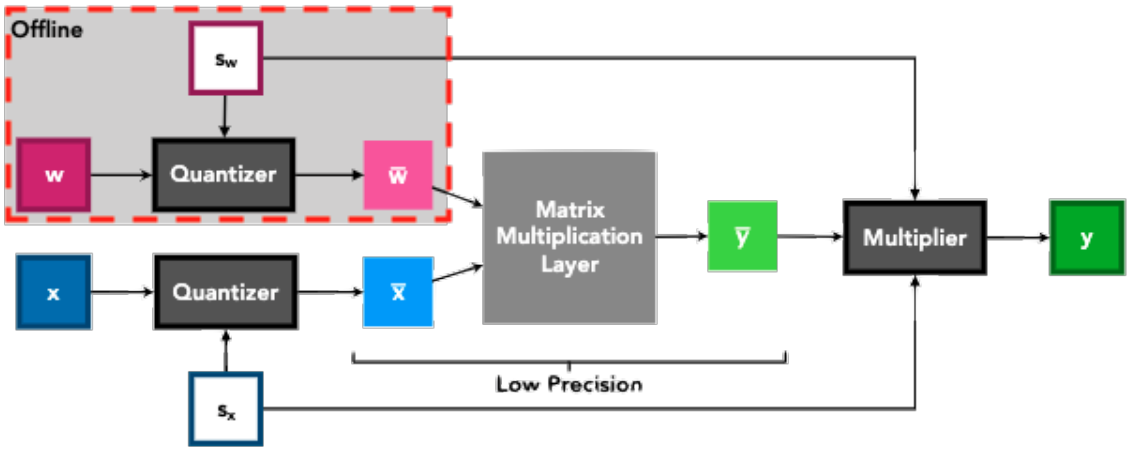


Fig. 4.2: Low precision computation in LSQ, this image was taken from LSQ paper SK Esser, et.al (2020).

As it can be seen, at training time, *S_w* and *S_x* are used to first quantize both activation and weights. These quantized values are fed into a low precision matrix multiply block. Finally, *S_w* and *S_x* are used to rescale the result. However, at inference time, the weight quantization can be performed offline and only activation quantization is necessary. In BARVINN, we added support for such quantization methods. There are scaling factor rams in each MVU that can be programmed to hold the scaling factor.

Warning: Currently, we only support plain CNN models without any residual connections. You can refer to "Residual Distillation: Towards Portable Deep Neural Networks without Shortcuts" NeurIPS 2020 paper to learn how to train a resnet-like model and convert it into a plain CNN model.

4.1 Code Generator

Once model training and quantization are done, we can export the model to ONNX format. We have provided a python library to take the ONNX model and generate MVU configuration code. There are two components to map an ONNX model to configuration code for MVU. We first need to parse an ONNX model and depending on the operation, break it down into matrix multiply operations. Then we need to generate configuration code for each matrix multiply. Since MVU expects the weights to be in the transposed MSB first format, we then need to reformat the weights. In BARVINN, we have provided a python library to help users map their ONNX model into a format that can be used to be executed on BARVINN. One then can use the following code to map and ONNX model into configuration code:

```
import logging
import argparse
from OnnxParser import OnnxParser
from Generator import Generator
import utils

def parse_args():
    parser = argparse.ArgumentParser()
    parser.add_argument('-x', '--onnx_model', help='input onnx model', required=True)
    parser.add_argument('--aprec', help='Activation precision', required=False,␣
↪default=8, type=int)
    parser.add_argument('--wprec', help='Weight precision', required=False, default=8,
↪ type=int)
    parser.add_argument('--oprec', help='Output precision', required=False, default=8,
↪ type=int)
    parser.add_argument('--input_shape', help='input shape for ',  nargs='*',␣
↪required=False, default=[3,32,32], type=int)
    args = parser.parse_args()
    return vars(args)

if __name__ == '__main__':
    args = parse_args()
    model_path = args['onnx_model']
    precision = [args['aprec'], args['wprec'], args['oprec']]
    input_shape = args['input_shape']
    model = OnnxParser(model_path)

    # model.print_onnx_graph()
    # model.print_onnx_model()
    if len(args['input_shape'])>3:
        print("Expecting an input array of shape: [channels, height, lenghth]")
        import sys
        sys.exit()
    generator = Generator(model, precision, input_shape)
    generator.generate_mvu_configs()
    generator.export_weigths()
    utils.gen_test_vecs(model_path, precision, input_shape)
```

As an example, we have used the quantized *distilled_resnet18.onnx* (available in BARVINN repo) with the sample code above to generate MVU configuration code. The following is the output of the code generator:

```
Generated MVU configuration:
+-------------+-----------+-------------------+------------------------+-----------------
```

(continues on next page)

(continued from previous page)

```
↪-+-------------------------+-----------+-----------------------+
| iShape       | fShape     | ilength              | ijump                    | wlength         ␣
↪ | wjump                   | countdown | total layer countdown |
+--------------+------------+--------------------+-------------------------+----------------
↪-+-------------------------+-----------+-----------------------+
| [1, 32, 32] | [1, 3, 3] | [0, 3, 2, 2, 0]    | [-132, -132, 60, 2, 0] | [0, 0, 3, 8, 0]␣
↪ | [-16, 2, -16, 2, 0]     | 1080      | 36720                 |
+--------------+------------+--------------------+-------------------------+----------------
↪-+-------------------------+-----------+-----------------------+
| [1, 32, 32] | [1, 3, 3] | [0, 3, 2, 2, 0]    | [-132, -132, 60, 2, 0] | [0, 0, 3, 8, 0]␣
↪ | [-16, 2, -16, 2, 0]     | 1080      | 36720                 |
+--------------+------------+--------------------+-------------------------+----------------
↪-+-------------------------+-----------+-----------------------+
| [1, 32, 32] | [1, 3, 3] | [0, 3, 2, 2, 0]    | [-132, -132, 60, 2, 0] | [0, 0, 3, 8, 0]␣
↪ | [-16, 2, -16, 2, 0]     | 1080      | 36720                 |
+--------------+------------+--------------------+-------------------------+----------------
↪-+-------------------------+-----------+-----------------------+
| [1, 32, 32] | [1, 3, 3] | [0, 3, 2, 2, 0]    | [-132, -132, 60, 2, 0] | [0, 0, 3, 8, 0]␣
↪ | [-16, 2, -16, 2, 0]     | 1080      | 36720                 |
+--------------+------------+--------------------+-------------------------+----------------
↪-+-------------------------+-----------+-----------------------+
| [1, 32, 32] | [1, 3, 3] | [0, 3, 2, 2, 0]    | [-132, -132, 60, 2, 0] | [0, 0, 3, 8, 0]␣
↪ | [-16, 2, -16, 2, 0]     | 1080      | 36720                 |
+--------------+------------+--------------------+-------------------------+----------------
↪-+-------------------------+-----------+-----------------------+
| [1, 32, 32] | [2, 3, 3] | [0, 7, 2, 2, 0]    | [-130, -132, 60, 2, 0] | [0, 1, 3, 8, 0]␣
↪ | [-34, 2, -16, 2, 0]     | 1080      | 18360                 |
+--------------+------------+--------------------+-------------------------+----------------
↪-+-------------------------+-----------+-----------------------+
| [2, 16, 16] | [2, 3, 3] | [0, 7, 2, 5, 0]    | [-134, -138, 54, 2, 0] | [0, 1, 3, 17,␣
↪0] | [-70, 2, -34, 2, 0]     | 2016      | 36288                 |
+--------------+------------+--------------------+-------------------------+----------------
↪-+-------------------------+-----------+-----------------------+
| [2, 16, 16] | [2, 3, 3] | [0, 7, 2, 5, 0]    | [-134, -138, 54, 2, 0] | [0, 1, 3, 17,␣
↪0] | [-70, 2, -34, 2, 0]     | 2016      | 36288                 |
+--------------+------------+--------------------+-------------------------+----------------
↪-+-------------------------+-----------+-----------------------+
| [2, 16, 16] | [2, 3, 3] | [0, 7, 2, 5, 0]    | [-134, -138, 54, 2, 0] | [0, 1, 3, 17,␣
↪0] | [-70, 2, -34, 2, 0]     | 2016      | 36288                 |
+--------------+------------+--------------------+-------------------------+----------------
↪-+-------------------------+-----------+-----------------------+
| [2, 16, 16] | [4, 3, 3] | [0, 15, 2, 5, 0]   | [-130, -138, 54, 2, 0] | [0, 3, 3, 17,␣
↪0] | [-142, 2, -34, 2, 0]    | 2016      | 18144                 |
+--------------+------------+--------------------+-------------------------+----------------
↪-+-------------------------+-----------+-----------------------+
| [4, 8, 8]   | [4, 3, 3] | [0, 15, 2, 11, 0] | [-138, -150, 42, 2, 0] | [0, 3, 3, 35,␣
↪0] | [-286, 2, -70, 2, 0]    | 3456      | 34560                 |
+--------------+------------+--------------------+-------------------------+----------------
↪-+-------------------------+-----------+-----------------------+
| [4, 8, 8]   | [4, 3, 3] | [0, 15, 2, 11, 0] | [-138, -150, 42, 2, 0] | [0, 3, 3, 35,␣
↪0] | [-286, 2, -70, 2, 0]    | 3456      | 34560                 |
+--------------+------------+--------------------+-------------------------+----------------
```

(continues on next page)

(continued from previous page)

```
↪-+------------------------+----------+----------------------+
| [4, 8, 8]    | [4, 3, 3] | [0, 15, 2, 11, 0] | [-138, -150, 42, 2, 0] | [0, 3, 3, 35,␣
↪0] | [-286, 2, -70, 2, 0]    | 3456       | 34560                  |
+-------------+----------+-----------------+------------------------+----------------
↪-+------------------------+----------+----------------------+
| [4, 8, 8]    | [8, 3, 3] | [0, 31, 2, 11, 0] | [-130, -150, 42, 2, 0] | [0, 7, 3, 35,␣
↪0] | [-574, 2, -70, 2, 0]    | 3456       | 17280                  |
+-------------+----------+-----------------+------------------------+----------------
↪-+------------------------+----------+----------------------+
| [8, 4, 4]    | [8, 3, 3] | [0, 31, 2, 23, 0] | [-146, -174, 18, 2, 0] | [0, 7, 3, 71,␣
↪0] | [-1150, 2, -142, 2, 0] | 4608       | 27648                  |
+-------------+----------+-----------------+------------------------+----------------
↪-+------------------------+----------+----------------------+
| [8, 4, 4]    | [8, 3, 3] | [0, 31, 2, 23, 0] | [-146, -174, 18, 2, 0] | [0, 7, 3, 71,␣
↪0] | [-1150, 2, -142, 2, 0] | 4608       | 27648                  |
+-------------+----------+-----------------+------------------------+----------------
↪-+------------------------+----------+----------------------+
| [8, 4, 4]    | [8, 3, 3] | [0, 31, 2, 23, 0] | [-146, -174, 18, 2, 0] | [0, 7, 3, 71,␣
↪0] | [-1150, 2, -142, 2, 0] | 4608       | 27648                  |
+-------------+----------+-----------------+------------------------+----------------
↪-+------------------------+----------+----------------------+
Total countdown: 532872
Exporting conv1.0.weight to conv1.0.weight.hex
Exporting conv2_x.0.residual_function.0.weight to conv2_x.0.residual_function.0.weight.
↪hex
Exporting conv2_x.0.residual_function.3.weight to conv2_x.0.residual_function.3.weight.
↪hex
Exporting conv2_x.1.residual_function.0.weight to conv2_x.1.residual_function.0.weight.
↪hex
Exporting conv2_x.1.residual_function.3.weight to conv2_x.1.residual_function.3.weight.
↪hex
Exporting conv3_x.0.residual_function.0.weight to conv3_x.0.residual_function.0.weight.
↪hex
Exporting conv3_x.0.residual_function.3.weight to conv3_x.0.residual_function.3.weight.
↪hex
Exporting conv3_x.1.residual_function.0.weight to conv3_x.1.residual_function.0.weight.
↪hex
Exporting conv3_x.1.residual_function.3.weight to conv3_x.1.residual_function.3.weight.
↪hex
Exporting conv4_x.0.residual_function.0.weight to conv4_x.0.residual_function.0.weight.
↪hex
Exporting conv4_x.0.residual_function.3.weight to conv4_x.0.residual_function.3.weight.
↪hex
Exporting conv4_x.1.residual_function.0.weight to conv4_x.1.residual_function.0.weight.
↪hex
Exporting conv4_x.1.residual_function.3.weight to conv4_x.1.residual_function.3.weight.
↪hex
Exporting conv5_x.0.residual_function.0.weight to conv5_x.0.residual_function.0.weight.
↪hex
Exporting conv5_x.0.residual_function.3.weight to conv5_x.0.residual_function.3.weight.
↪hex
Exporting conv5_x.1.residual_function.0.weight to conv5_x.1.residual_function.0.weight.
```

(continues on next page)

(continued from previous page)

```
↪hex
Exporting conv5_x.1.residual_function.3.weight to conv5_x.1.residual_function.3.weight.
↪hex
Inference finised in 0.0082 seconds
Exporting output to output.hex
Exporting input to input.hex
```

As you can see, the code generator provides a configuration for each layer of the input model. These values can be directly used in C/assembly code to program the MVU. The code generator also generates a weight hex file for each layer that can be used by the simulator to program the MVU rams. Finally, the code generator used the input ONNX model with OnnxRuntime engine to generate and expected results given a random input vector, both of which are also saved the generator code so that they can be used for verification purposes.

CHAPTER

FIVE

FPGA PROTOTYPING

TBD

CHAPTER

SIX

EXAMPLES

6.1 Environment Setup

To run a neural network model using BARVINN, you will need to use 4 different repositories:

- BARVINN. : The top module repo that re-uses pito and MVU project.
- PITO_RISCV. : The barrel RISC-V processor.
- MVU. : The Matrix Vector Unit accelerator.
- MVU_Code_Gen. : A repository that contains python libraries to generate configuration code for MVU.

We have added the last three repositories as a gitmodule to BARVINN repository. Hence, you only need to clone BARVINN repository as below:

```
git clone https://github.com/hossein1387/BARVINN
cd BARVINN
git submodule update --init --recursive
```

As mentioned earlier, BARVINN requires RISC-V GCC that supports RV32I. You can either install RISC-V GCC with your favorite OS package manager, or you can follow picorv32. project to build a pure RV32I toolchain. The following are some of the examples that you can run on BARVINN.

6.2 Matrix Multiplication

In this example code, we want to program *MVU[0]* to perform a matrix multiplication. Note that we do not include code for transferring data into MVU's feature map and weight memory. Here we are simply assuming that the data is in the correct format and layout. The following code performs a matrix multiplication between input feature map vector of size *[1x1x1x64]* at 2-bit precision with a weight matrix of size *[1x64x64x16]* at 2-bit precision. The output result is written to *0x400* with 2-bit precision. As we mentioned in the *design* section, the controller (*pito*) configures a job by setting the appropriate CSR registers and then kick starts the job by writing into *mvucommand* CSR register. Although one can monitor the job status by polling the *mvustatus* register, MVU will send an interrupt once the job is done and ready to be read. In the following code block, we first enable global and MVU specific irq (in *enable_mvu_irq* function). We then set the address for the MVU irq handler to service the interrupt (in *__startup_code__*). We then program a matrix multiply job in *mat_mul* function. At this point, we can start to prepare and configure the next job, or we can just wait for an interrupt. For this simple example, we wait for an interrupt from *MVU*. Finally, if everything works as expected, we should see *OKn* in register *a1*, *a2* and *a3* and in memory address *0x1000*.

```
#include "pito_def.h"
```

(continues on next page)

(continued from previous page)

```
jal sp, enable_mvu_irq
jal sp, __startup_code__
jal sp, mat_mul
jal t3, wait_for_mvu_irq
jal sp, prog_end


// in startup code, we need to set the following:
//   -> mtvec addresses
//
__startup_code__:
    // addi x1, x0, pito_mtvec_mask
    // creating mtvec mask
    lui  a0, %hi(mvu_irq_handler)
    addi a0, a0, %lo(mvu_irq_handler )
    csrw mtvec, a0
    addi ra, sp, 0
    ret

wait_for_mvu_irq:
    csrr t0, mcause
    srli t0, t0, 31
    addi t1, x0, 1
    // wait for mcause[31] interrupt to go high
    bne t0, t1, wait_for_mvu_irq
    addi ra, t3, 0
    ret

mvu_irq_handler:
    // make sure global interrupt is disabled
    csrwi mstatus, 0x0
    // first things first, clear mvu intterupts pending bit while processing current irq.
    addi t1, x0, 1
    slli t1, t1, 16
    csrc mip, t1
    // do whatever to make MVU happy
    addi x0, x0, 0
    // we can now start processing incoming interrupts
    addi gp, sp, 0
    jal sp, enable_mvu_irq
    addi ra, gp, 0
    mret

enable_mvu_irq:
    // make sure global interrupt is enabled
    csrwi mstatus, 0x8
    // set MVU specific MIE bit aka mie[16]
    addi t0, x0, 1
    slli t0, t0, 16
    csrw mie, t0
    addi ra, sp, 0
    ret
```

(continues on next page)

(continued from previous page)

```
disable_mvu_irq:
    // clear MVU specific MIE bit
    addi t0, x0, 1
    slli t0, t0, 16
    not t0, t0
    csrw mie, t0
    addi ra, sp, 0
    ret

clear_mvu_pending_irq:
    csrrci x0, mip, 0
    ret

mat_mul:
    addi  t1, x0, 0
    addi  t2, x0, 2
    add   t1, t1, t2              // set weight precision to 2
    slli  t3, t2, 6               // set input precision to 2
    add   t1, t1, t3
    slli  t3, t2, 12               // set output precision to 2
    add   t1, t1, t3
    csrw  mvuprecision,  t1

    csrwi mvuquant     , 10       // set quant_msbidx to 10
    csrwi mvuwbaseptr  , 0        // set weight address to 0
    csrwi mvuibaseptr  , 0        // set input address to 0

    addi  t1, x0, 1
    slli  t1, t1, 10               // set output address to 0x400
    csrw mvuobaseptr , t1

    csrwi mvuwjump_0, 30           // 1 tile back move x 2 bits
    csrwi mvuwjump_1, 2            // 1 tile ahead move x 2 bits
    csrwi mvuwjump_2, 0
    csrwi mvuwjump_3, 0
    csrwi mvuwjump_4, 0
    csrwi mvuijump_0, 30           // 1 tile back move x 2 bits
    csrwi mvuijump_1, 0
    csrwi mvuijump_2, 0
    csrwi mvuijump_3, 0
    csrwi mvuijump_4, 0
    csrwi mvusjump_0, 0
    csrwi mvusjump_1, 0
    csrwi mvubjump_0, 0
    csrwi mvubjump_1, 0
    csrwi mvuojump_0, 0
    csrwi mvuojump_1, 0
    csrwi mvuojump_2, 0
    csrwi mvuojump_3, 0
    csrwi mvuojump_4, 0
    csrwi mvuwlength_1 ,  1       // 2 tiles in width
```

(continues on next page)

(continued from previous page)

```
    csrwi mvuwlength_2 ,  3       // number bit combinations i.e. 2x2 bits
    csrwi mvuwlength_3 ,  1       // 2 tiles in height
    csrwi mvuwlength_4 ,  0
    csrwi mvuilength_1 ,  1       // 2 tiles in height
    csrwi mvuilength_2 ,  0       // number bit combinations
    csrwi mvuilength_3 ,  0       // 2 tiles in width of matrix operand
    csrwi mvuilength_4 ,  0
    csrwi mvuolength_1 ,  1
    csrwi mvuolength_2 ,  0
    csrwi mvuolength_3 ,  0
    csrwi mvuolength_4 ,  0

    addi t1, x0, 1
    slli t1, t1, 30               // mul mode 01
    addi t1, t1, 16
    csrw mvucommand, t1           // Kick start MVU, 2 tiles x 2 tiles x 2bit x 2bits
    addi ra, sp, 0
    ret

// Done with our awesome program!
prog_end:
    lui a0,0x1000>>12
    addi  a1,zero,'O'
    addi  a2,zero,'K'
    addi  a3,zero,'\n'
    sw  a1,0(a0)
    sw  a2,0(a0)
    sw  a3,0(a0)
    ebreak
```

To run the code on BARVINN, we will first need to compile the above code. This source code is provided in BARVINN's csrc directory. You can compile the code using the following instructions:

```
cd matmul
make matmul.hex
```

This will generate a hex file that should be loaded into BARVINN. Now to run th program on BARVINN, you should follow these steps:

First make sure Vivado is in the PATH:

```
source /opt/Xilinx/Vivado/2019.1/settings64.sh
```

Then, assuming FuseSoC is already instlled, if not done already, we need to let FuseSoC know where to find PITO and MVU repos:

```
cd BARVINN/MVU
fusesoc library add mvu .
cd ..
cd BARVINN/pito_riscv
fusesoc library add pito .
cd ..
fusesoc library add barvinn .
```

The commands above need to be executed once so that FuseSoC registers the BARVINN, PITO and MVU project correctly. Now that FuseSoC is configured properly, we can run a FuseSoC target for BARVINN (assuming *matmul.hex* is in the current directory):

```
cd ..
fusesoc library add barvinn .
fusesoc run --target=sim barvinn --firmware=matmul.hex
```

By default, we have set *verification/tests/core/core_tester.sv* to run. However, one can change this by modifying barvinn core file . Also, you by default, there are initial simulation values in MVU's weight and input rams. You can modify that by using different input and weight files.

6.3 Convolution

In this example code, we want to program *MVU[0]* to perform a Convolution operation. We will first start with an ONNX model. Fig. 6.1 shows that the second layer of *resnet18* on cifar100 performs a convolution with input size of *[1x64x32x32]* with a weight tensor of size *[64x64x3x3]*. The convolution parameters are illustrated by Netron in Fig. 6.1.

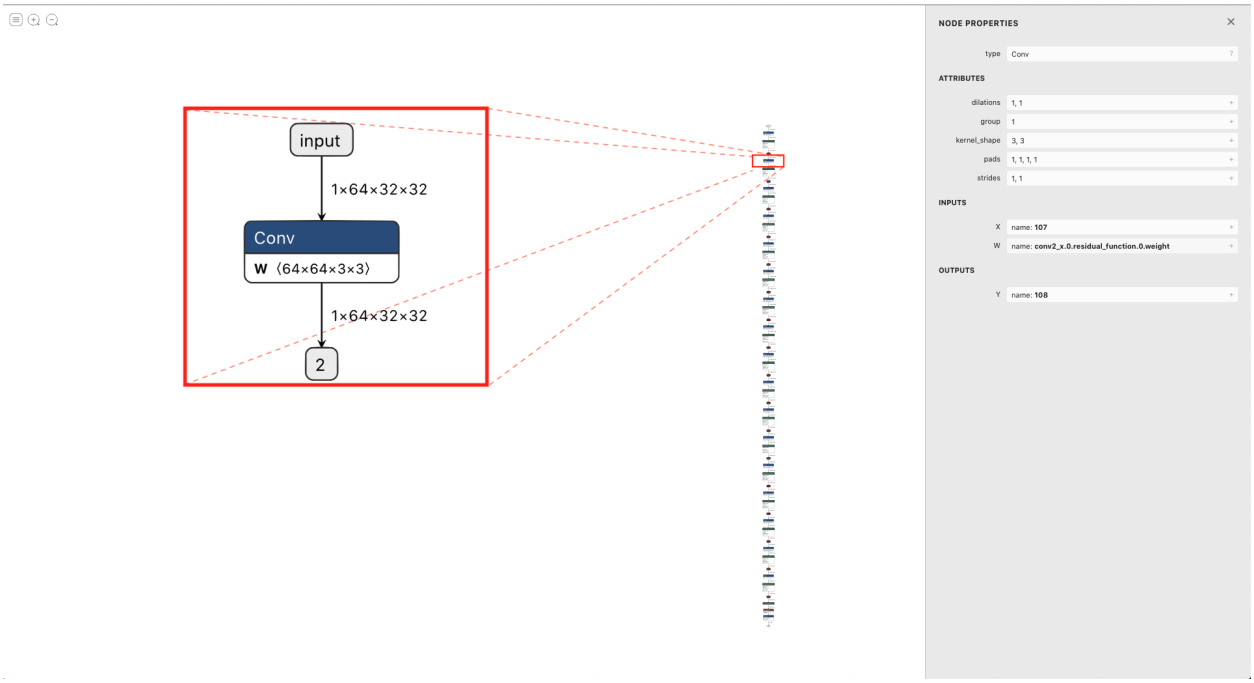


Fig. 6.1: Model used for Convolution example. This image shows that we are using the second conv layer of resnet18 on Cifar100. ONNX model is illustrated using Netron.

The model in ONNX format is not suitable for MVU. As we discussed in previous sections, we have written a code generator software to take an ONNX model and then provide the user with the proper MVU configuration settings. For this example, assuming we have saved this simple one layer convolution block as *SimpleConv.onnx*, we can use the code generator as below:

```
1 import logging
2 import argparse
3 from OnnxParser import OnnxParser
```

(continues on next page)

(continued from previous page)

```
from Generator import Generator
import utils

def parse_args():
    parser = argparse.ArgumentParser()
    parser.add_argument('-x', '--onnx_model', help='input onnx model', required=True)
    parser.add_argument('--aprec', help='Activation precision', required=False, default=8, type=int)
    parser.add_argument('--wprec', help='Weight precision', required=False, default=8, type=int)
    parser.add_argument('--oprec', help='Output precision', required=False, default=8, type=int)
    parser.add_argument('--input_shape', help='input shape for ',  nargs='*', required=False, default=[3,32,32], type=int)
    args = parser.parse_args()
    return vars(args)

if __name__ == '__main__':
    args = parse_args()
    model_path = args['onnx_model']
    precision = [args['aprec'], args['wprec'], args['oprec']]
    input_shape = args['input_shape']
    model = OnnxParser(model_path)

    # model.print_onnx_graph()
    # model.print_onnx_model()
    if len(args['input_shape'])>3:
        print("Expecting an input array of shape: [channels, height, lenghth]")
        import sys
        sys.exit()
    generator = Generator(model, precision, input_shape)
    generator.generate_mvu_configs()
    generator.export_weigths()
    utils.gen_test_vecs(model_path, precision, input_shape)
```

And then execute the script above as below:

```
python sample_mvu_code_generator.py -x SimpleConv.onnx  --aprec 8 --wprec 8 --oprec 8 --input_shape 64 32 32
```

In the command above, we are specifying a 2 bit precision for weights, activation and output result. We are also specifying the input shape of the model. Here is the output for the command above:

```
Generated MVU configuration:
+------------+----------+-----------------+-------------------------+-----------------+---------------------+----------+-----------------------+
| iShape     | fShape   | ilength         | ijump                   | wlength         | wjump               | countdown | total layer countdown |
+------------+----------+-----------------+-------------------------+-----------------+---------------------+----------+-----------------------+
| [1, 32, 32] | [1, 3, 3] | [0, 63, 2, 2, 0] | [-528, -528, 240, 8, 0] | [0, 0, 63, 8, 0] | [-64, 8, -64, 8, 0] | 17280     | 587520                |
```

(continues on next page)

(continued from previous page)

```
+------------+----------+-----------------+-------------------------+-----------------
↪-+--------------------+----------+----------------------+
Total countdown: 587520
Exporting conv1.weight to conv1.weight.hex
Inference finised in 0.0030 seconds
Exporting output to output.hex
Exporting input to input.hex
```

Here is what we generated after executing the command above:

- A Generated MVU configuration table.
- A weight hex file in MSB transposed format *conv1.weight.hex*
- An input hex file *input.hex*
- An output hex file *output.hex*

The generated MVU configurations can be used to write a program to configure MVU csrs. The weight hex file can be directly used in simulation using *$readmemh* to write into MVU weight rams. For verification and testing the correctness of our design, we run the model through *OnnxRuntime* engine to capture the execution time and output results. However, since *OnnxRuntime* supports only 8-bit operation, the MVU results might not be the same as *OnnxRuntime* so for now we use 8 bit precision on MVU.

```
#include "pito_def.h"

jal sp, enable_mvu_irq
jal sp, __startup_code__
jal sp, mat_mul
jal t3, wait_for_mvu_irq
jal sp, prog_end


// in startup code, we need to set the following:
//   -> mtvec addresses
//
__startup_code__:
    // addi x1, x0, pito_mtvec_mask
    // creating mtvec mask
    lui  a0, %hi(mvu_irq_handler)
    addi a0, a0, %lo(mvu_irq_handler )
    csrw mtvec, a0
    addi ra, sp, 0
    ret

wait_for_mvu_irq:
    csrr t0, mcause
    srli t0, t0, 31
    addi t1, x0, 1
    // wait for mcause[31] interrupt to go high
    bne t0, t1, wait_for_mvu_irq
    addi ra, t3, 0
    ret

mvu_irq_handler:
```

(continues on next page)

(continued from previous page)

```
    // make sure global interrupt is disabled
    csrwi mstatus, 0x0
    // first things first, clear mvu intterupts pending bit while processing current irq.
    addi x1, x0, 1
    slli x1, x1, 16
    csrc mip, x1
    // do whatever to make MVU happy
    addi x0, x0, 0
    // we can now start processing incoming interrupts
    addi gp, sp, 0
    jal sp, enable_mvu_irq
    addi ra, gp, 0
    mret

enable_mvu_irq:
    // make sure global interrupt is enabled
    csrwi mstatus, 0x8
    // set MVU specific MIE bit aka mie[16]
    addi t0, x0, 1
    slli t0, t0, 16
    csrw mie, t0
    addi ra, sp, 0
    ret

disable_mvu_irq:
    // clear MVU specific MIE bit
    addi t0, x0, 1
    slli t0, t0, 16
    not t0, t0
    csrw mie, t0
    addi ra, sp, 0
    ret

clear_mvu_pending_irq:
    csrrci x0, mip, 0
    ret

mat_mul:
    addi  x1, x0, 0
    addi  x2, x0, 2
    add   x1, x1, x2              // set weight precision to 2
    slli  x3, x2, 6               // set input precision to 2
    add   x1, x1, x3
    slli  x3, x2, 12               // set output precision to 2
    add   x1, x1, x3
    csrw  mvu_precision,  x1

    csrwi mvuquant     , 10       // set quant_msbidx to 10
    csrwi mvuwbaseptr  , 0        // set weight address to 0
    csrwi mvuibaseptr  , 0        // set input address to 0

    addi  x1, x0, 1
```

(continues on next page)

(continued from previous page)

```
    slli  x1, x1, 10                // set output address to 0x400
    csrw mvuobaseptr , x1

    csrwi mvuwjump_0, 30            // 1 tile back move x 2 bits
    csrwi mvuwjump_1, 2             // 1 tile ahead move x 2 bits
    csrwi mvuwjump_2, 0
    csrwi mvuwjump_3, 0
    csrwi mvuwjump_4, 0
    csrwi mvuijump_0, 30            // 1 tile back move x 2 bits
    csrwi mvuijump_1, 0
    csrwi mvuijump_2, 0
    csrwi mvuijump_3, 0
    csrwi mvuijump_4, 0
    csrwi mvusjump_0, 0
    csrwi mvusjump_1, 0
    csrwi mvubjump_0, 0
    csrwi mvubjump_1, 0
    csrwi mvuojump_0, 0
    csrwi mvuojump_1, 0
    csrwi mvuojump_2, 0
    csrwi mvuojump_3, 0
    csrwi mvuojump_4, 0
    csrwi mvuwlength_0 ,  1         // 2 tiles in width
    csrwi mvuwlength_1 ,  3         // number bit combinations i.e. 2x2 bits
    csrwi mvuwlength_2 ,  1         // 2 tiles in height
    csrwi mvuwlength_3 ,  0
    csrwi mvuilength_0 ,  1         // 2 tiles in height
    csrwi mvuilength_1 ,  0         // number bit combinations
    csrwi mvuilength_2 ,  0         // 2 tiles in width of matrix operand
    csrwi mvuilength_3 ,  0
    csrwi mvuolength_0 ,  1
    csrwi mvuolength_1 ,  0
    csrwi mvuolength_2 ,  0
    csrwi mvuolength_3 ,  0

    addi x1, x0, 1
    slli x1, x1, 30                 // mul mode 01
    addi x1, x1, 16
    csrw mvucommand, x1             // Kick start MVU, 2 tiles x 2 tiles x 2bit x 2bits

    ret

// Done with our awesome program!
prog_end:
    lui a0,0x10000000>>12
    addi  a1,zero,'O'
    addi  a2,zero,'K'
    addi  a3,zero,'\n'
    sw  a1,0(a0)
    sw  a2,0(a0)
    sw  a3,0(a0)
    ebreak
```

6.4 Classification

6.5 Segmentation

CHAPTER

SEVEN

CREDITS AND PUBLICATIONS

The main contributors of this project are as follow:

- Hossein Askari
- Sean Wagner
- Olexa Bilaniuk

If you found this project interesting and useful, please consider citing our papers:

Our paper on designing BARVINN:

```
@Article{barvinn_aspdac,
    author={AskariHemmat, MohammadHossein and Bilaniuk, Olexa and Wagner, Sean and␣
↪Hariri, Yassine and Savaria, Yvon and David, Jean-Pierre},
    journal= {28th Asia and South Pacific Design Automation Conference ASP-DAC 2023},
    title  = {BARVINN: Arbitrary Precision DNN Accelerator Controlled by a RISC-V CPU},
    year   = {2023},
    doi    = {10.1145/3566097.3567872}
}
```

Our paper on designing PITO:

```
@INPROCEEDINGS{9114581,
    author={AskariHemmat, MohammadHossein and Bilaniuk, Olexa and Wagner, Sean and␣
↪Savaria, Yvon and David, Jean-Pierre},
    booktitle={2020 IEEE 28th Annual International Symposium on Field-Programmable␣
↪Custom Computing Machines (FCCM)},
    title={RISC-V Barrel Processor for Accelerator Control},
    year={2020},
    volume={},
    number={},
    pages={212-212},
    doi={10.1109/FCCM48280.2020.00063}
}

@INPROCEEDINGS{9401617,
    author={AskariHemmat, MohammadHossein and Bilaniuk, Olexa and Wagner, Sean and␣
↪Savaria, Yvon and David, Jean-Pierre},
    booktitle={2021 IEEE International Symposium on Circuits and Systems (ISCAS)},
    title={RISC-V Barrel Processor for Deep Neural Network Acceleration},
    year={2021},
    volume={},
```

(continues on next page)

(continued from previous page)

```
    number={},
    pages={1-5},
    doi={10.1109/ISCAS51556.2021.9401617}
}
```

Our paper on designing MVU:

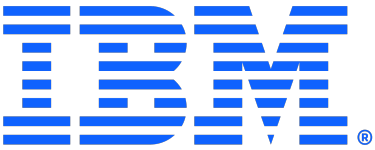
```
@INPROCEEDINGS{8702332,
    author={Bilaniuk, Olexa and Wagner, Sean and Savaria, Yvon and David, Jean-Pierre},
    booktitle={2019 IEEE International Symposium on Circuits and Systems (ISCAS)},
    title={Bit-Slicing FPGA Accelerator for Quantized Neural Networks},
    year={2019},
    volume={},
    number={},
    pages={1-5},
    doi={10.1109/ISCAS.2019.8702332}
}
```

CHAPTER

EIGHT

ACKNOWLEDGEMENT

This project was supported by the following organizations:




CMC
MICROSYSTEMS


Fonds de recherche
Nature et
technologies
Québec


Mitacs