BARVINN: Arbitrary Precision DNN Accelerator Controlled by a RISC-V CPU

MohammadHossein AskariHemmat[1], Sean Wagner[2], Olexa Bilaniuk[3],
Yassine Hariri[4], Yvon Savaria[1], Jean-Pierre David[1]
[1]Ecole Polytechnique Montreal, Canada [2]IBM, Toronto, Canada,
[3]Mila, Montreal, Canada, [4]CMC Microsystems, Kingston, Canada

28th Asia and South Pacific Design Automation
Conference, ASP-DAC 2023
Tokyo, Japan
Jan 16-19 2023

Mila

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

IBM

CMC MICROSYSTEMS

Outline

1. Motivation and Background.
2. BARVINN Overall Architecture:
    a. MVU Array and Architecture.
    b. PITO: Multi-Threaded RISC-V Controller.
3. BARVINN programming model and software stack.
4. Experiments and Results.
5. Conclusion and Future Work

1. Motivation and Background

- Quantized Deep Neural Networks (QDNNs) rely on floating-point computations.

1. Motivation and Background

- Quantized Deep Neural Networks (QDNNs) rely on floating-point computations.
- Compared to fixed-point and integer operations, floating-point computations are slow and costly in terms of power consumption and silicon area.

1. Motivation and Background

- Quantized Deep Neural Networks (QDNNs) rely on floating-point computations.
- Compared to fixed-point and integer operations, floating-point computations are slow and costly in terms of power consumption and silicon area.
- On the other hand, it has been shown that quantized models can achieve near floating-point precisions in vision tasks.

Task	Dataset	Model	Precision A/W	Acc/ MAP	Size (MB)
Classification	CIFAR 100	ResNet18	LSQ(2/2)	76.81	2.889
			LSQ(4/4)	76.92	5.559
			LSQ(8/8)	78.45	10.87
			FP32	76.82	42.8
Object Detection	VOC-2007	SSD300-ResNet18	LSQ(2/2)	0.61	10.34
			LSQ(4/4)	0.60	11.81
			LSQ(8/8)	0.68	14.77
			FP32	0.59	32.49

1. Motivation and Background

- Quantized Deep Neural Networks (QDNNs) rely on floating-point computations.
- Compared to fixed-point and integer operations, floating-point computations are slow and costly in terms of power consumption and silicon area.
- On the other hand, it has been shown that quantized models can achieve near floating-point precisions in vision tasks.
- However, there are no commercially available general processors (CPU or GPU) that can efficiently process data in arbitrary precision.

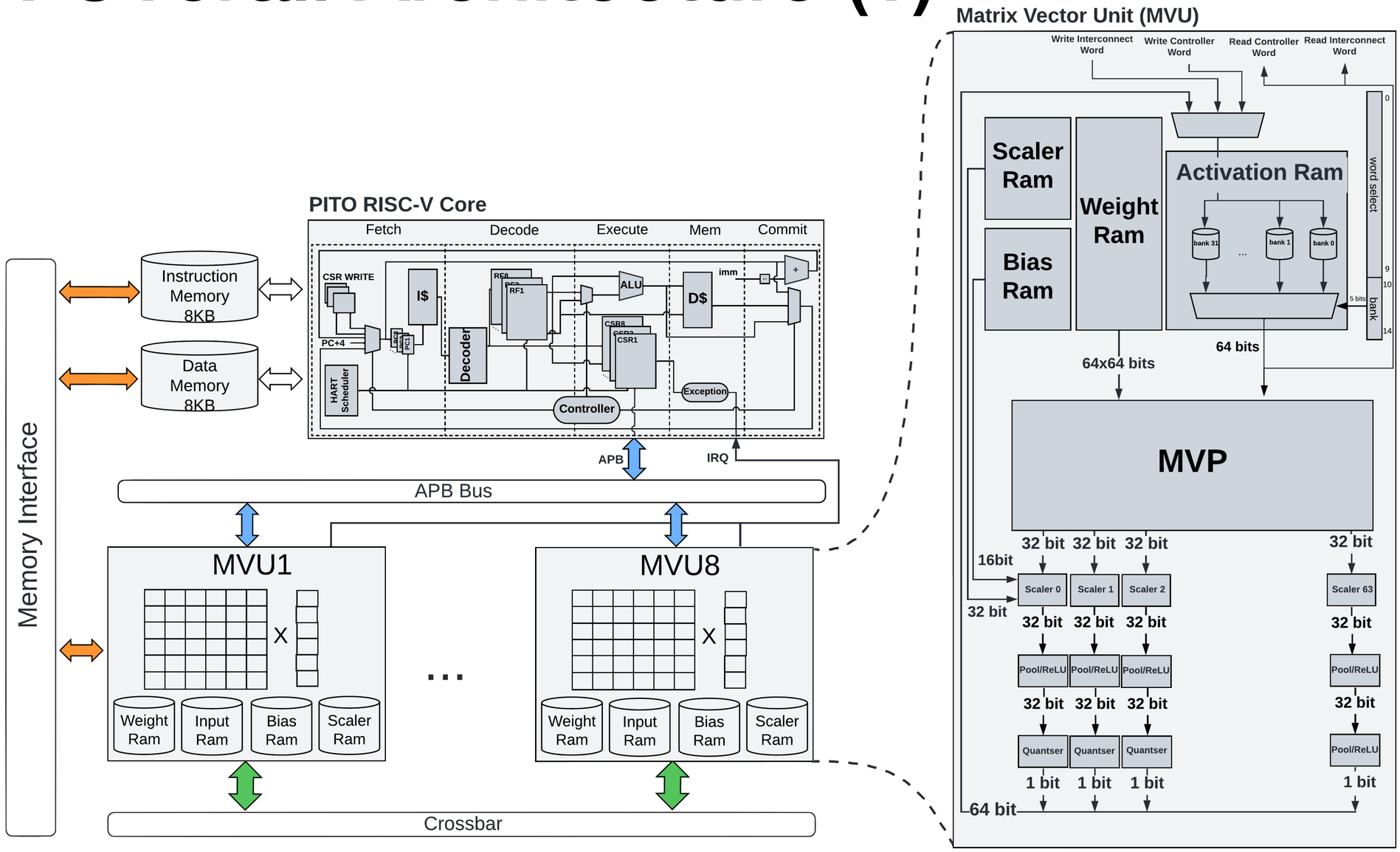
Introducing ...

BARVINN

1. Motivation and Background

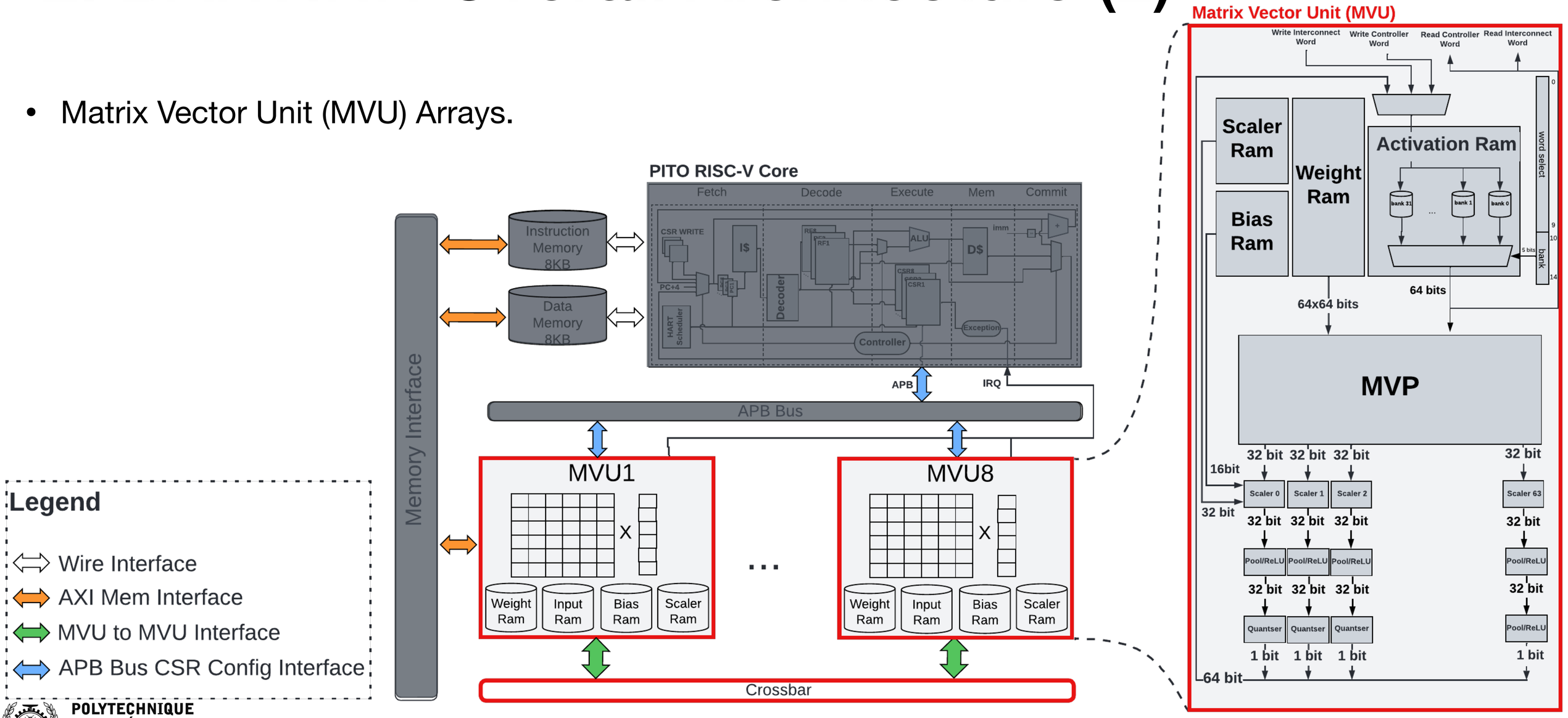
- Introducing BARVINN!
- BARVINN is a DNN for running arbitrary precision quantized models.
- It has 8 processing elements (MVUs) that are controlled by a RISC-V controller.
- It has an overall 8.2 TMACs of computational power (binary ops).
- It has been implemented on Alveo U250 FPGA platform.

2. BARVINN Overall Architecture (1)



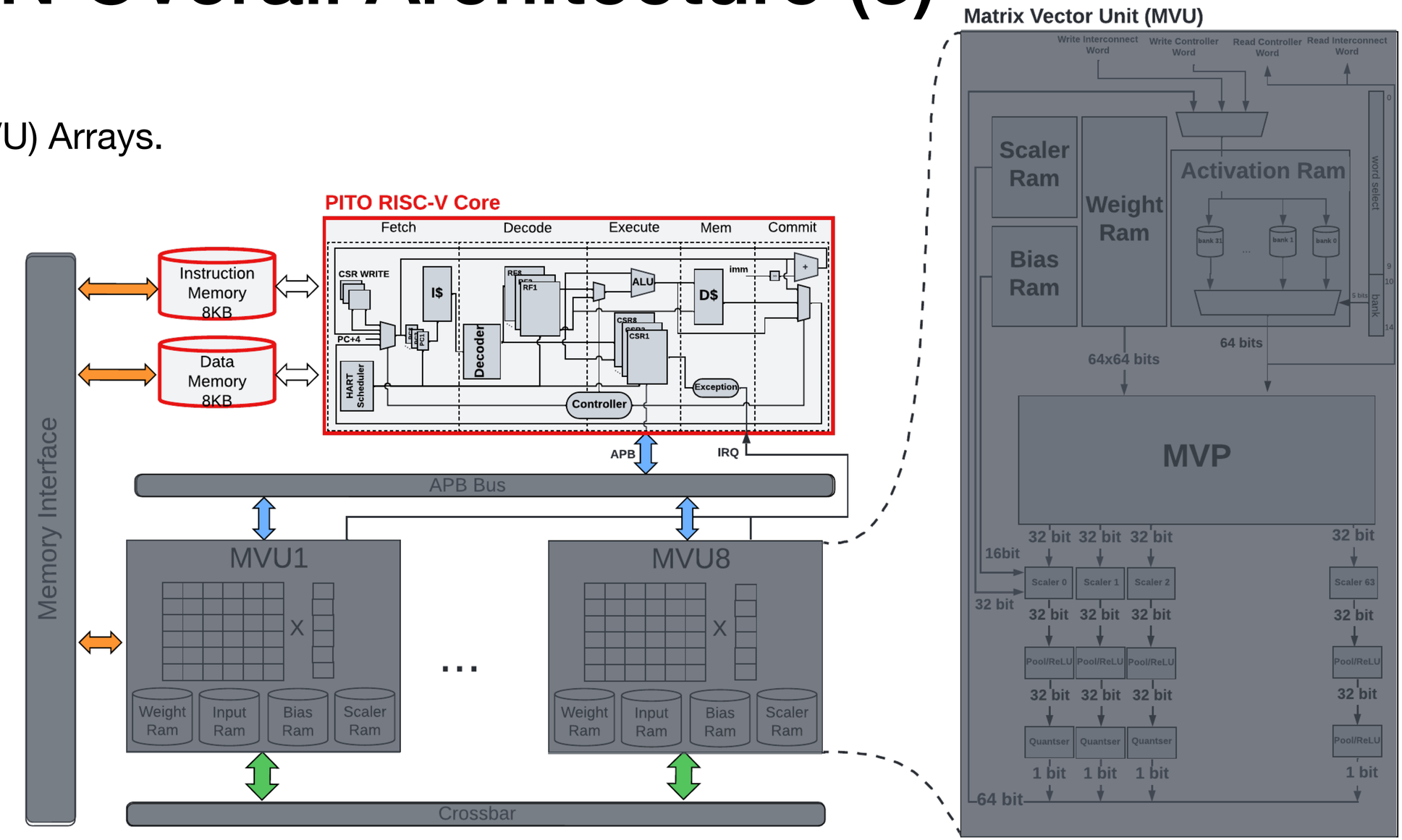
2. BARVINN Overall Architecture (2)

- Matrix Vector Unit (MVU) Arrays.



2. BARVINN Overall Architecture (3)

- Matrix Vector Unit (MVU) Arrays.
- RISC-V Controller.



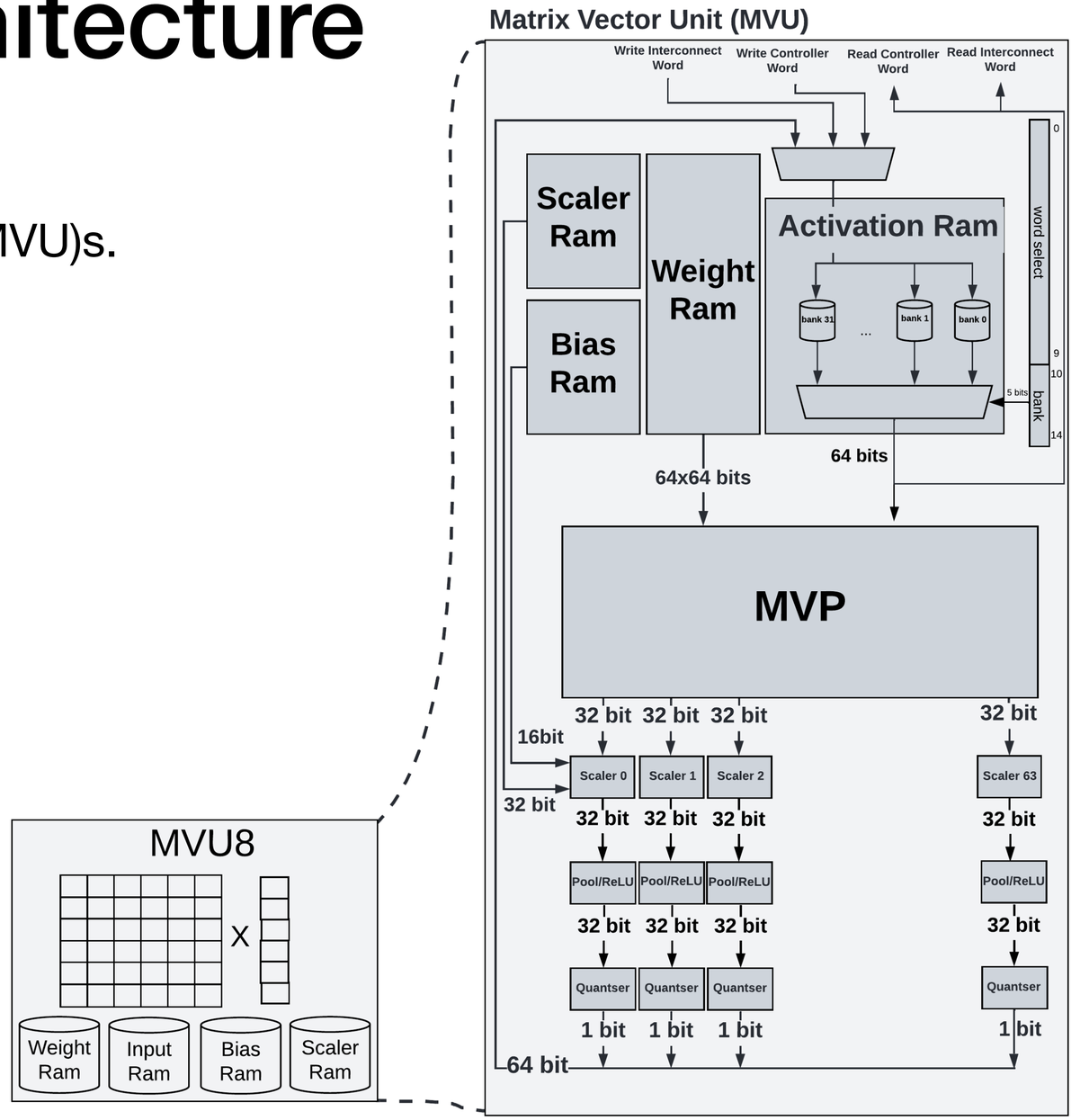
Legend

- Wire Interface
- AXI Mem Interface
- MVU to MVU Interface
- APB Bus CSR Config Interface

Matrix Vector Units: BARVINN processing elements.

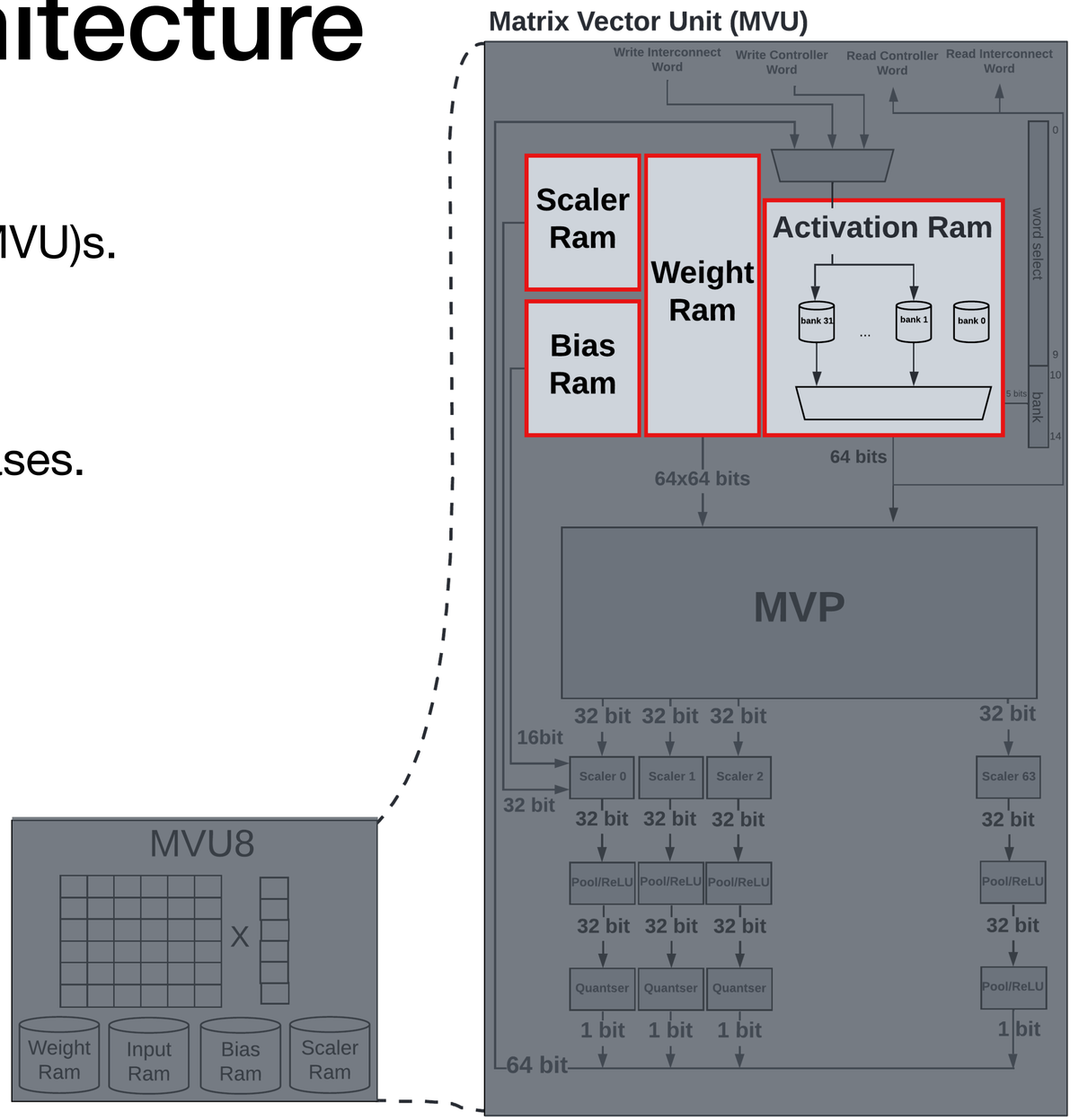
2.1. MVU Array and Architecture

- BARVINN has an array of 8 Matrix Vector Unit (MVU)s.
- Each MVU is consist of:



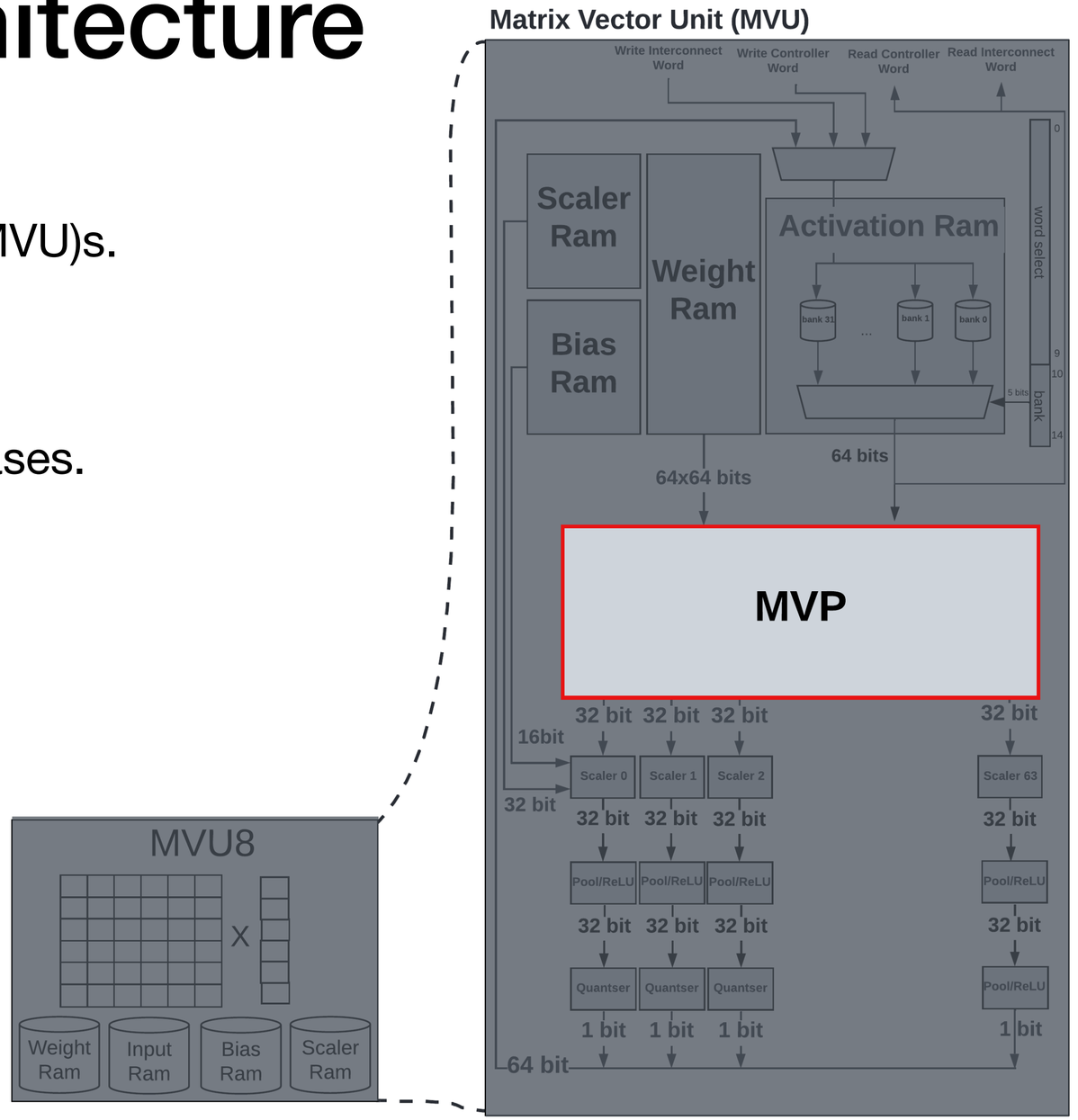
2.1. MVU Array and Architecture

- BARVINN has an array of 8 Matrix Vector Unit (MVU)s.
- Each MVU is consist of:
  - RAMs for activation, Weights, Scalers and Biases.



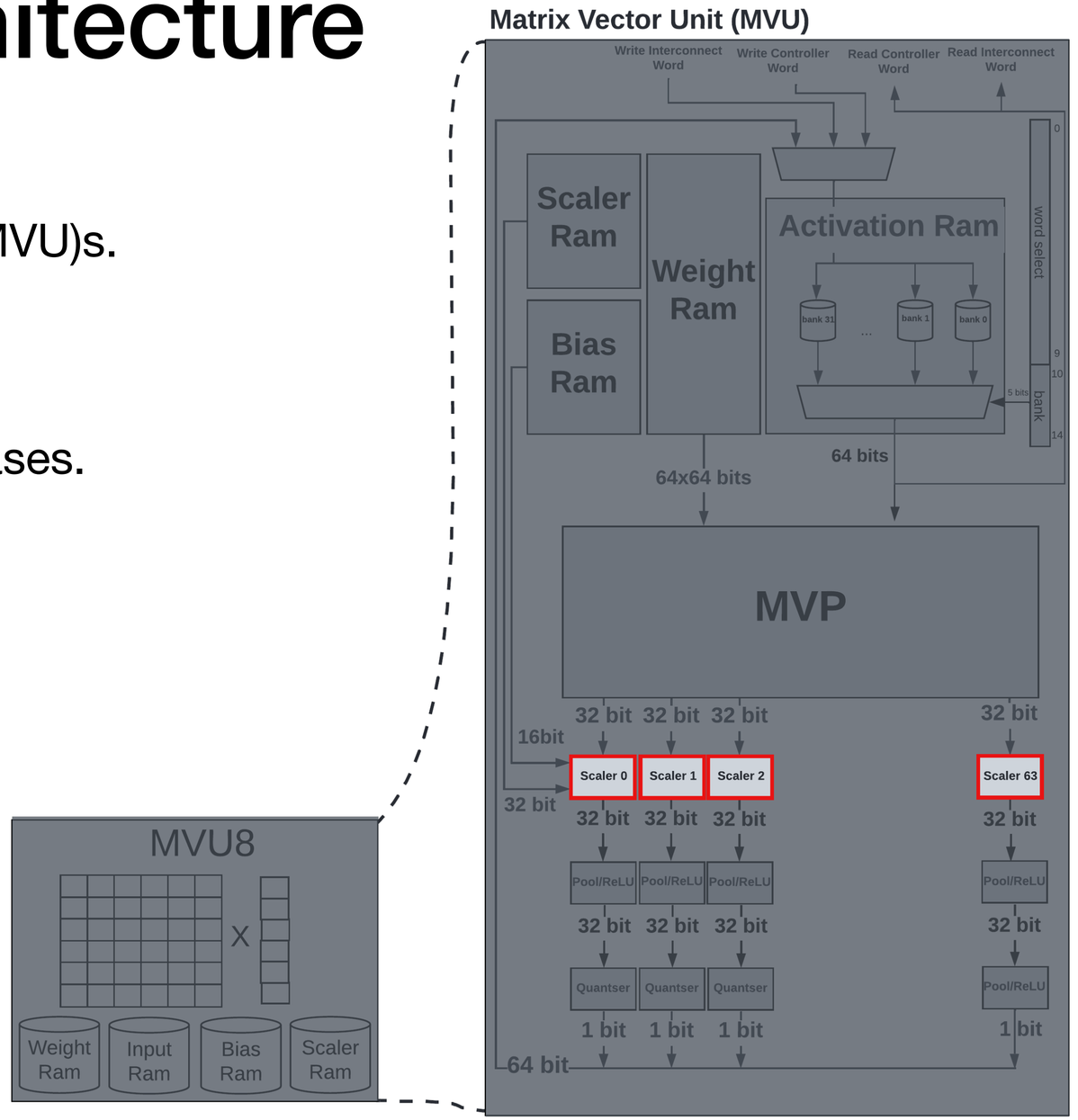
2.1. MVU Array and Architecture

- BARVINN has an array of 8 Matrix Vector Unit (MVU)s.
- Each MVU is consist of:
  - RAMs for activation, Weights, Scalers and Biases.
  - Matrix Vector Product unit (MVP).



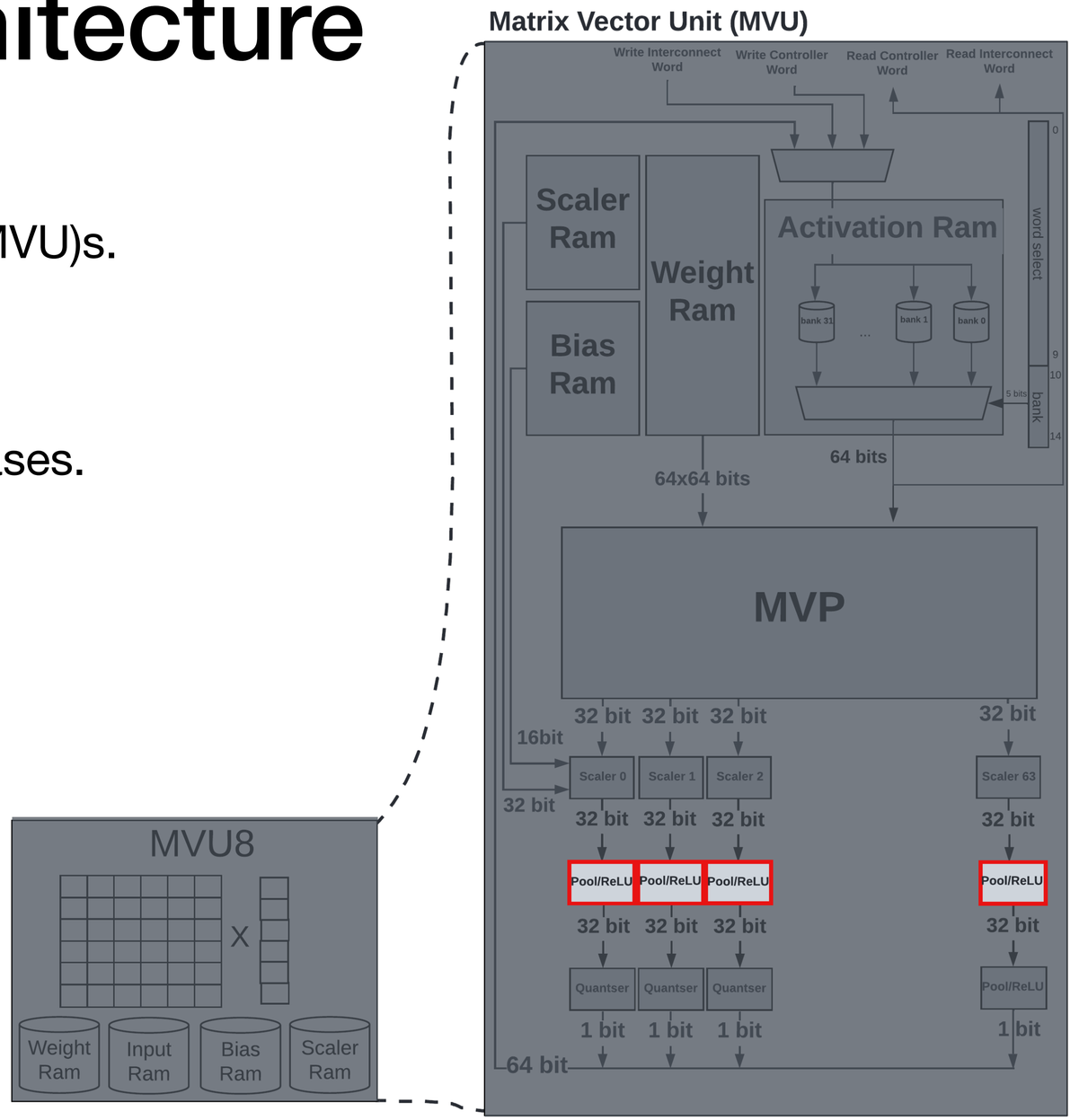
2.1. MVU Array and Architecture

- BARVINN has an array of 8 Matrix Vector Unit (MVU)s.
- Each MVU is consist of:
  - RAMs for activation, Weights, Scalers and Biases.
  - Matrix Vector Product unit (MVP).
  - Pooling and Activation units.



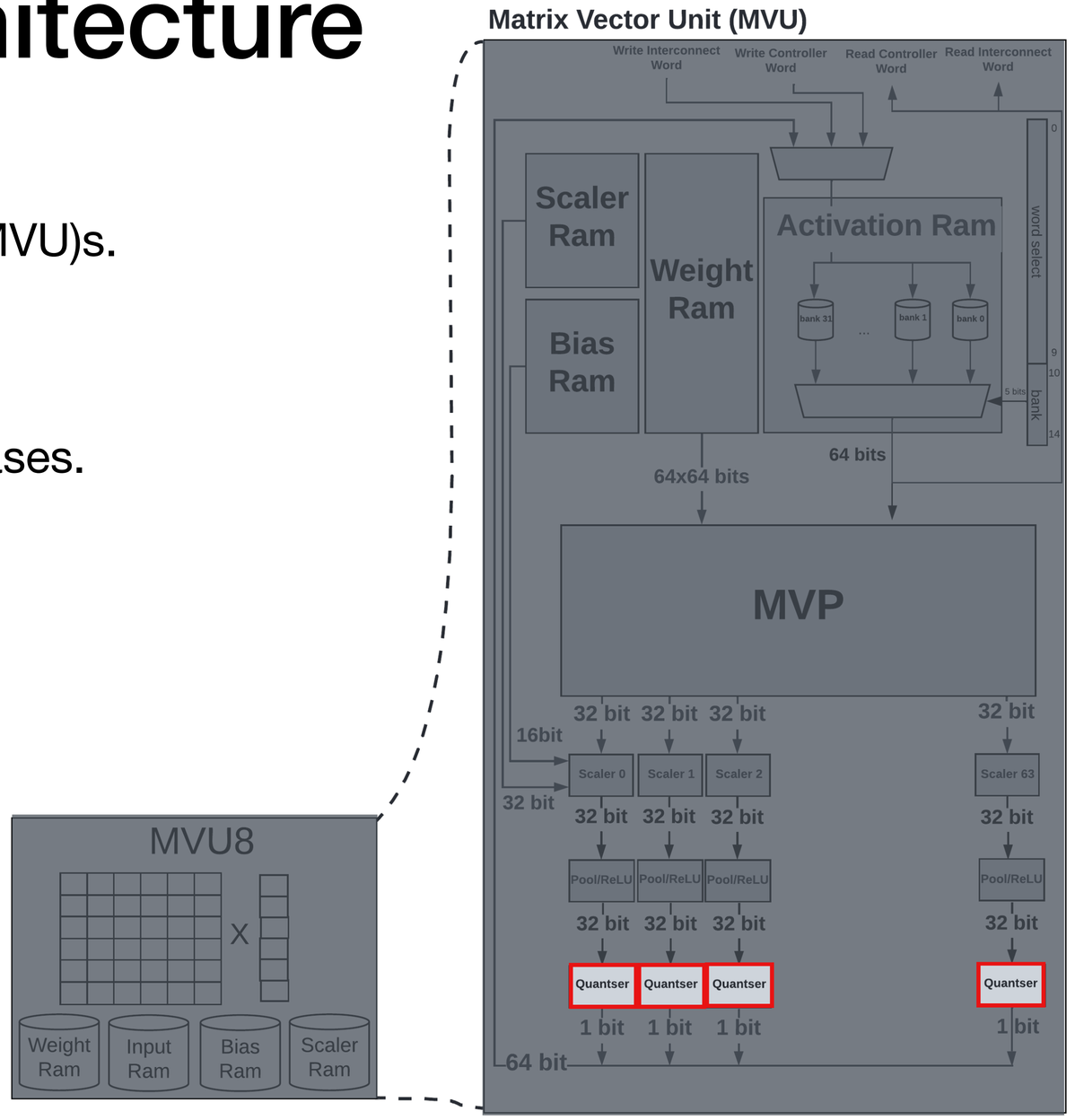
2.1. MVU Array and Architecture

- BARVINN has an array of 8 Matrix Vector Unit (MVU)s.
- Each MVU is consist of:
  - RAMs for activation, Weights, Scalers and Biases.
  - Matrix Vector Product unit (MVP).
  - Pooling and Activation units.
  - Scaler Unit.



2.1. MVU Array and Architecture

- BARVINN has an array of 8 Matrix Vector Unit (MVU)s.
- Each MVU is consist of:
  - RAMs for activation, Weights, Scalers and Biases.
  - Matrix Vector Product unit (MVP).
  - Pooling and Activation units.
  - Scaler unit.
  - Quantizer unit.

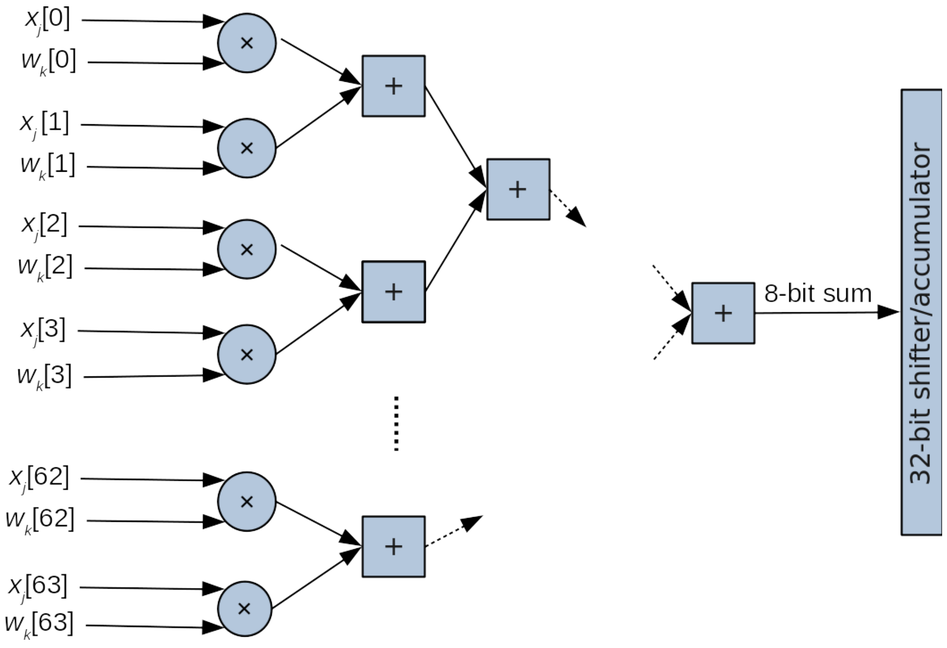


2.1. MVU Array and Architecture

- BARVINN has an array of 8 Matrix Vector Unit (MVU)s.
- Each MVU is consist of: 1- RAMs for activation, 2- Weights, Scalers and Biases. 3- Matrix Vector Product unit (MVP). 4- Pooling and Activation units. Scaler unit. 5- Quantizer unit.
- Using 64 input element data and 64 x64 element matrix from Weight RAM, each MVU compute 64 output vector elements per each clock cycle.

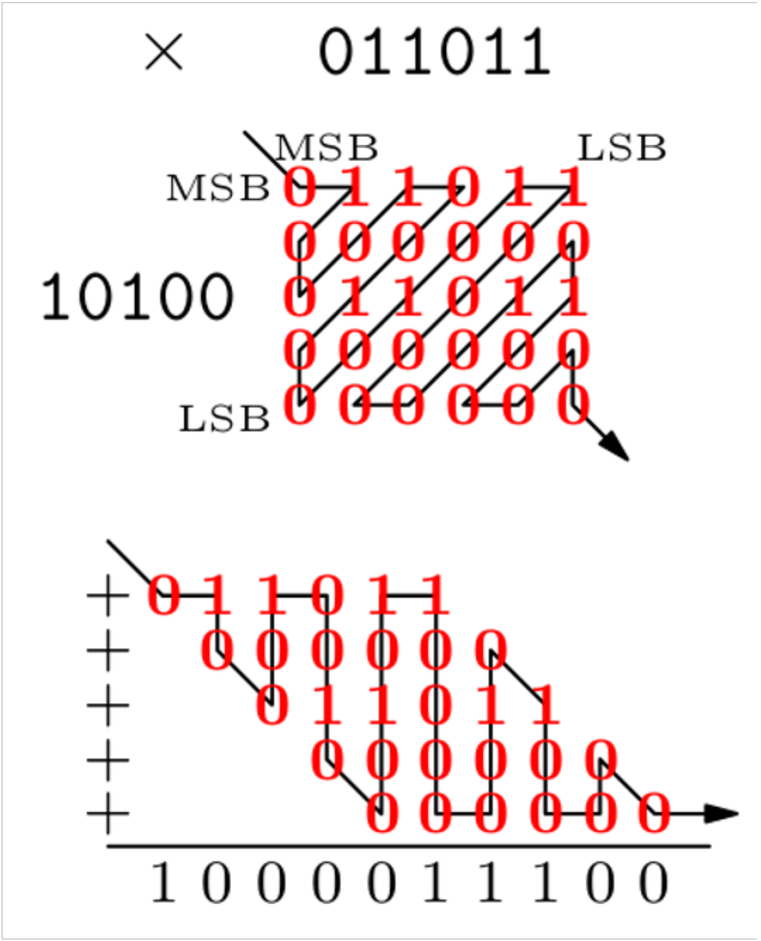
2.1. MVU Array and Architecture

- Matrix Vector Products (MVP)s:
  - Compute fixed-point arbitrary precision operands 1- to 16-bit.
  - Each MVP has 64 Vector-Vector Product (VVP).
  - Each cycle, 64 bits from activation RAM is broadcasted to each of the 64 VVPs, and a 64x64 matrix tile is loaded from the weight ram and loaded to separate VVPs.
  - The VVPs compute a 64-element dot product on 1-bit operands (as displayed in the adder tree).



2.1. MVU Array and Architecture

- We use **bit-serial** math to support arbitrary precision.
- Example: **`A=0b011011, B=0b10100, C= A x B`**

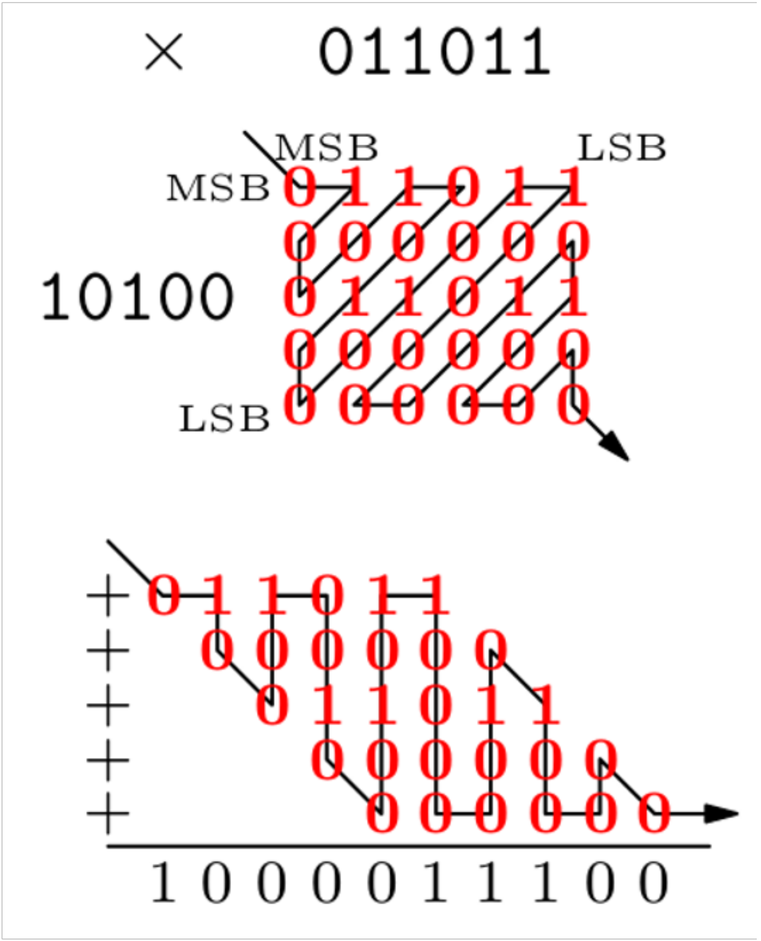


2.1. MVU Array and Architecture

- We use bit-serial math to support arbitrary precision.
- Example: **A=0b011011, B=0b10100, C= A x B**

Start cycle time

```
Step 1 (t=0)
A:      011011
B:       10100
+              0   ← Partial sums
C: 0000000000
```



2.1. MVU Array and Architecture

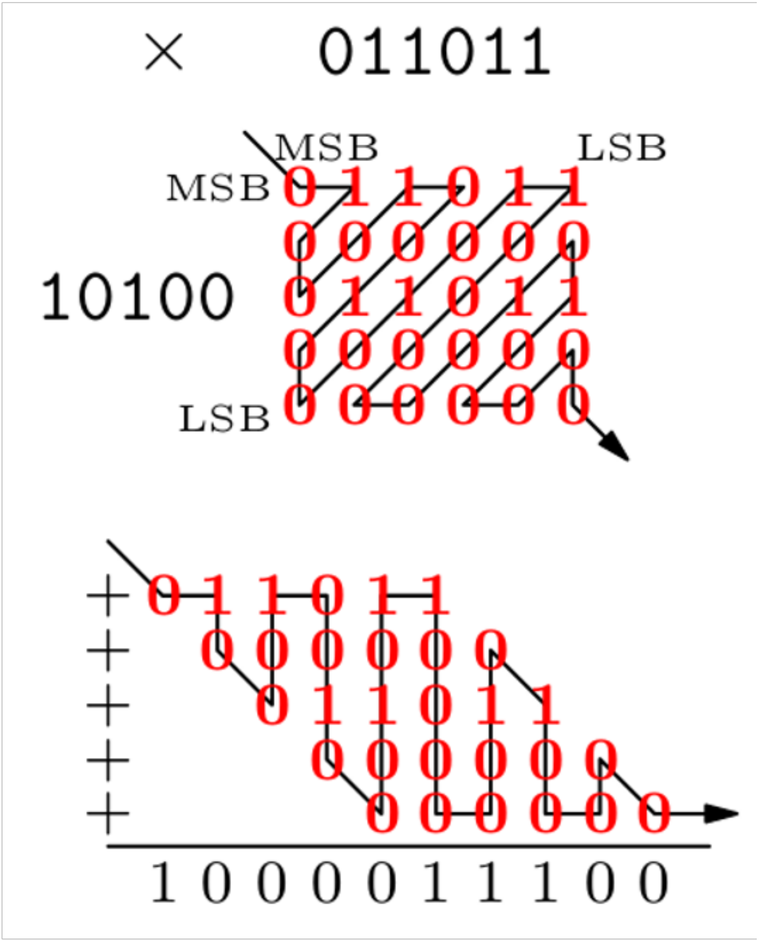
- We use bit-serial math to support arbitrary precision.
- Example: **A=0b011011, B=0b10100, C= A x B**

Step 1 (t=0)

```
A:      011011
B:       10100
+            0
C: 0000000000
```

Step 2 (t=1)

```
A:      011011
B:       10100
+            1
C: 0000000001
           ←
```



2.1. MVU Array and Architecture

- We use bit-serial math to support arbitrary precision.
- Example: **A=0b011011, B=0b10100, C= A x B**

Step 1 (t=0)

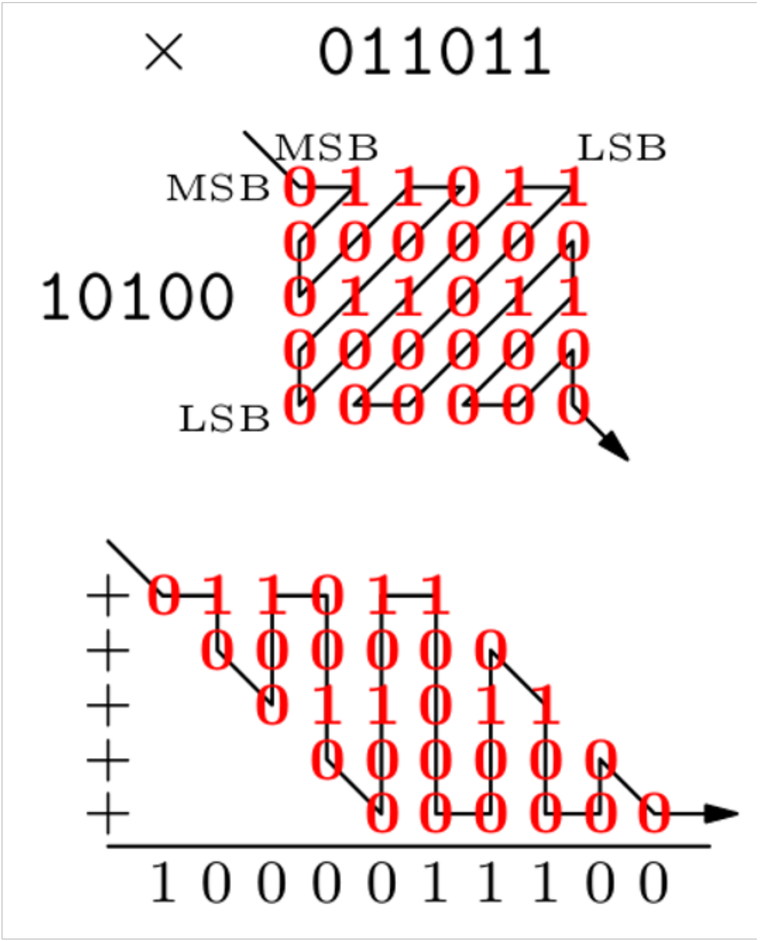
```
A:      011011
B:       10100
+            0
C: 0000000000
```

Step 2 (t=1)

```
A:      011011
B:       10100
+            1
C: 0000000001
          ←
```

Step 3 (t=3)

```
A:      011011
B:       10100
+            1
C: 0000000011
          ←
```



2.1. MVU Array and Architecture

- We use bit-serial math to support arbitrary precision.
- Example: **A=0b011011, B=0b10100, C= A x B**

Step 1 (t=0)
```
A:      011011
B:       10100
+            0
C: 0000000000
```

Step 2 (t=1)
```
A:      011011
B:       10100
+            1
C: 0000000001
```
←

Step 3 (t=3)
```
A:      011011
B:       10100
+            1
C: 0000000011
```
←

Step 4 (t=6)
```
A:      011011
B:       10100
+            1
C: 0000000111
```
←

Step 5 (t=10)
```
A:      011011
B:       10100
+           10
C: 0000010000
```
←

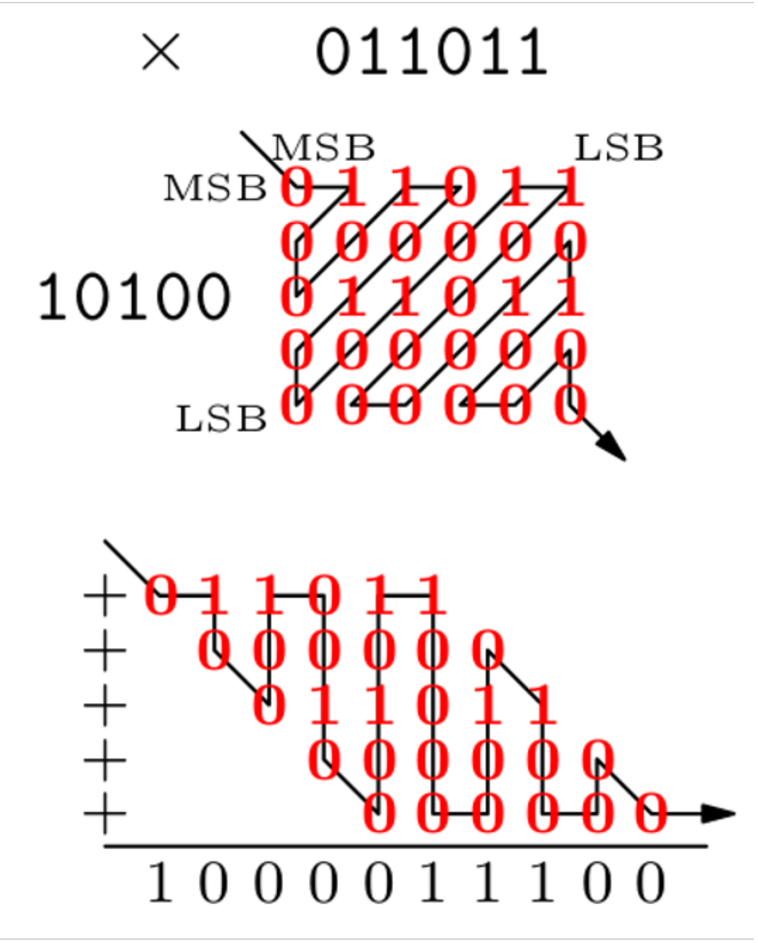
Step 6 (t=15)
```
A:      011011
B:       10100
+            1
C: 0000100001
```
←

Step 7 (t=20)
```
A:      011011
B:       10100
+            1
C: 0001000011
```
←

Step 8 (t=24)
```
A:      011011
B:       10100
+            1
C: 0010000111
```
←

Step 9 (t=27)
```
A:      011011
B:       10100
+            0
C: 0100001110
```
←

Step 10 (t=29)
```
A:      011011
B:       10100
+            0
C: 1000011100
```



Total time:
30 cycles

Pito: A Simple RISC-V Procesor to Control MVU arrays.

2.2. PITO: Multithreaded RISC-V Controller

- PITO is a RISC-V processor that supports RV32I instruction set.

2.2. PITO: Multithreaded RISC-V Controller

- PITO is a RISC-V processor that supports RV32I instruction set.
- PITO has 8KB of instruction and 8KB of data RAM.

2.2. PITO: Multithreaded RISC-V Controller

- PITO is a RISC-V processor that supports RV32I instruction set.
- PITO has 8KB of instruction and 8KB of data RAM.
- It supports privilege mode to read and write from and to CSRs.

2.2. PITO: Multithreaded RISC-V Controller

- PITO is a RISC-V processor that supports RV32I instruction set.
- PITO has 8KB of instruction and 8KB of data RAM.
- It supports privilege mode to read and write from and to CSRs.
- It uses 75 extra RISC-V CSRs per MVU to control different MVU promoters.

2.2. PITO: Multithreaded RISC-V Controller

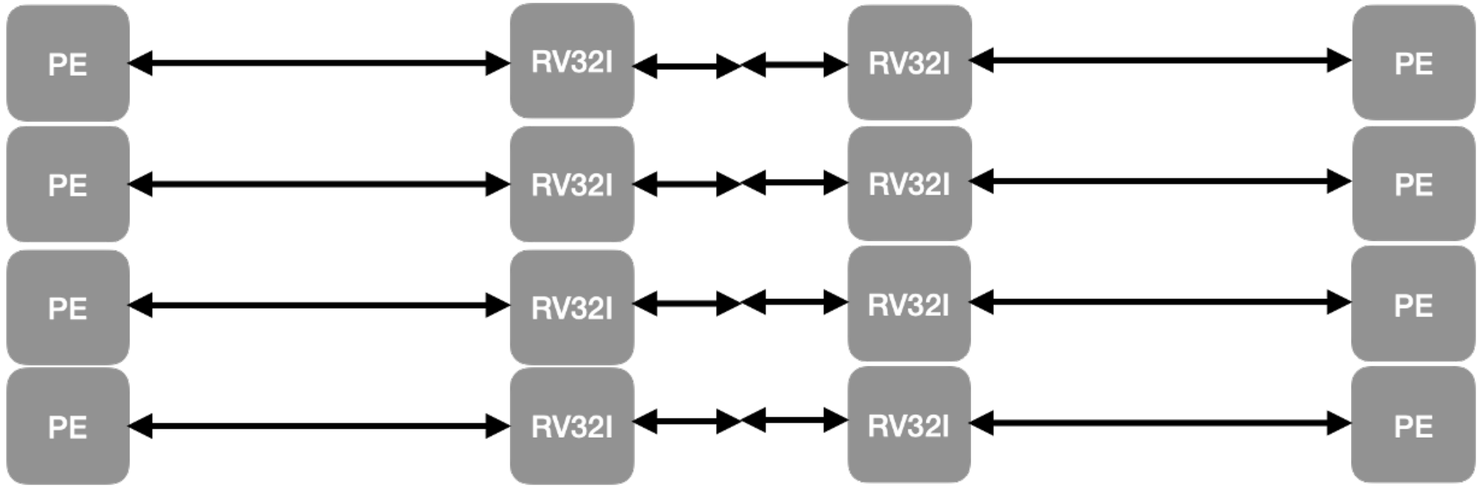
- PITO is a RISC-V processor that supports RV32I instruction set.
- PITO has 8KB of instruction and 8KB of data RAM.
- It supports privilege mode to read and write from and to CSRs.
- It uses 75 extra RISC-V CSRs per MVU to control different MVU promoters.
- It has a custom C runtime for controlling different MVUs in a thread safe environment, reducing the need for a custom RTOS.

How to control multiple processing elements?

2.2. PITO: Multithreaded RISC-V Controller (1)

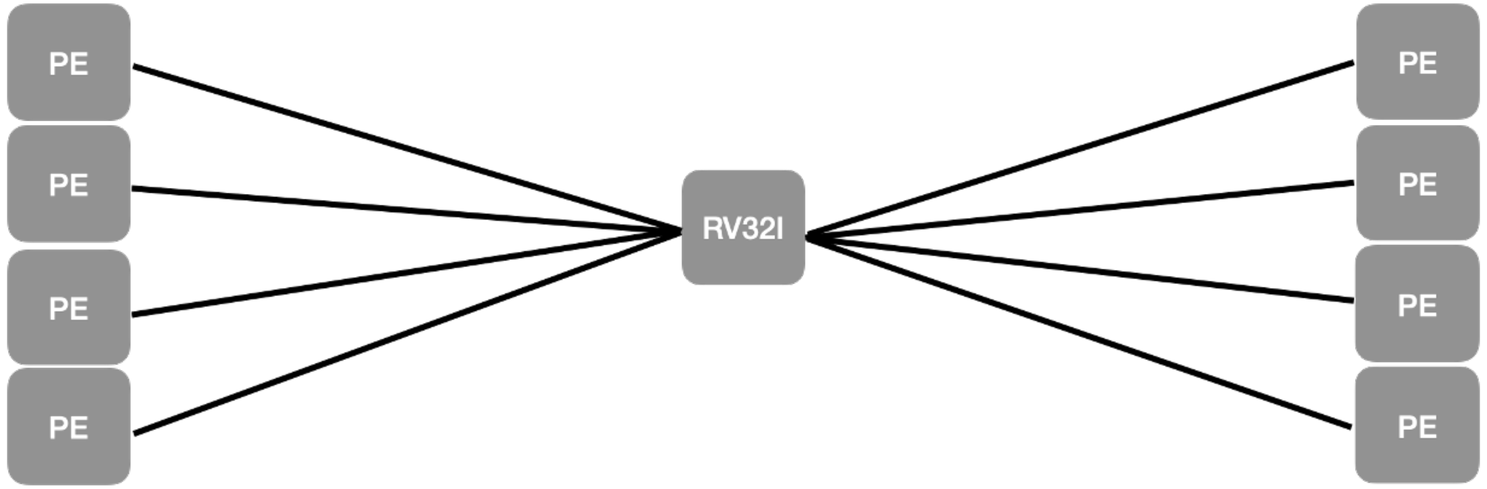
- How to control multiple processing elements?
  - Solution1: Use a separate controller for each PE.
    - High throughput.
    - Fine control over each PE.
    - High resource utilization.

.



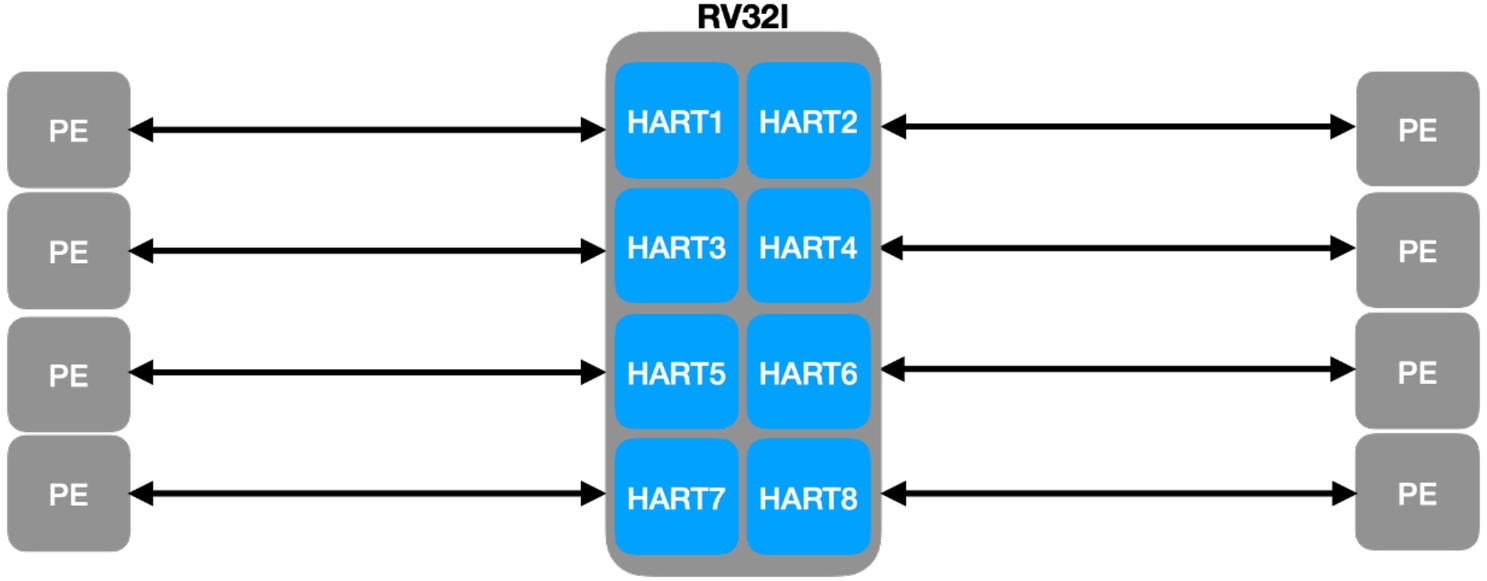
2.2. PITO: Multithreaded RISC-V Controller (2)

- How to control multiple processing elements?
  - Solution1: Use a separate controller for each PE.
  - Solution2: Use a shared controller for all PEs.
    - Low resource utilization.
    - Lower throughput.

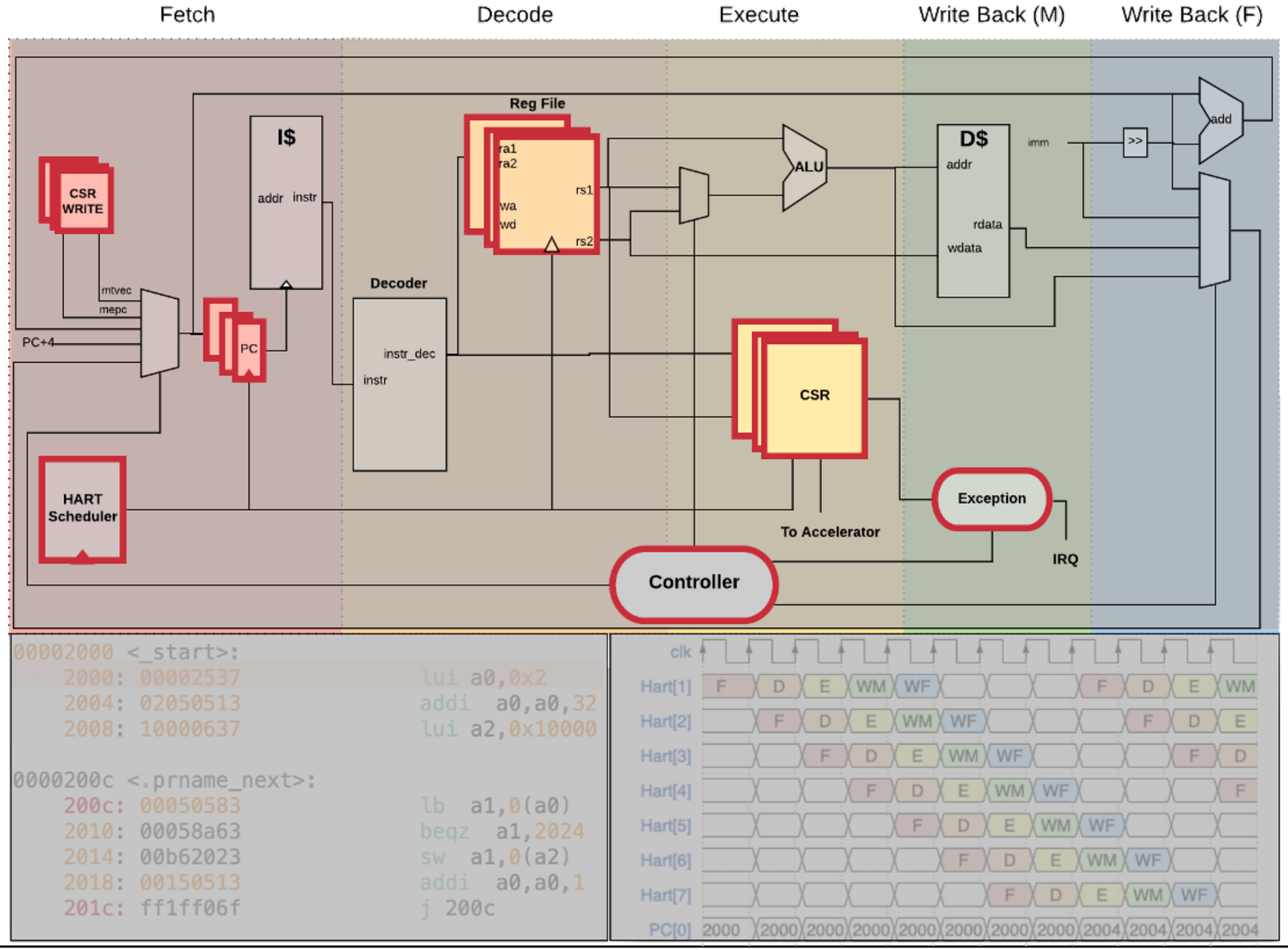


2.2. PITO: Multithreaded RISC-V Controller (3)

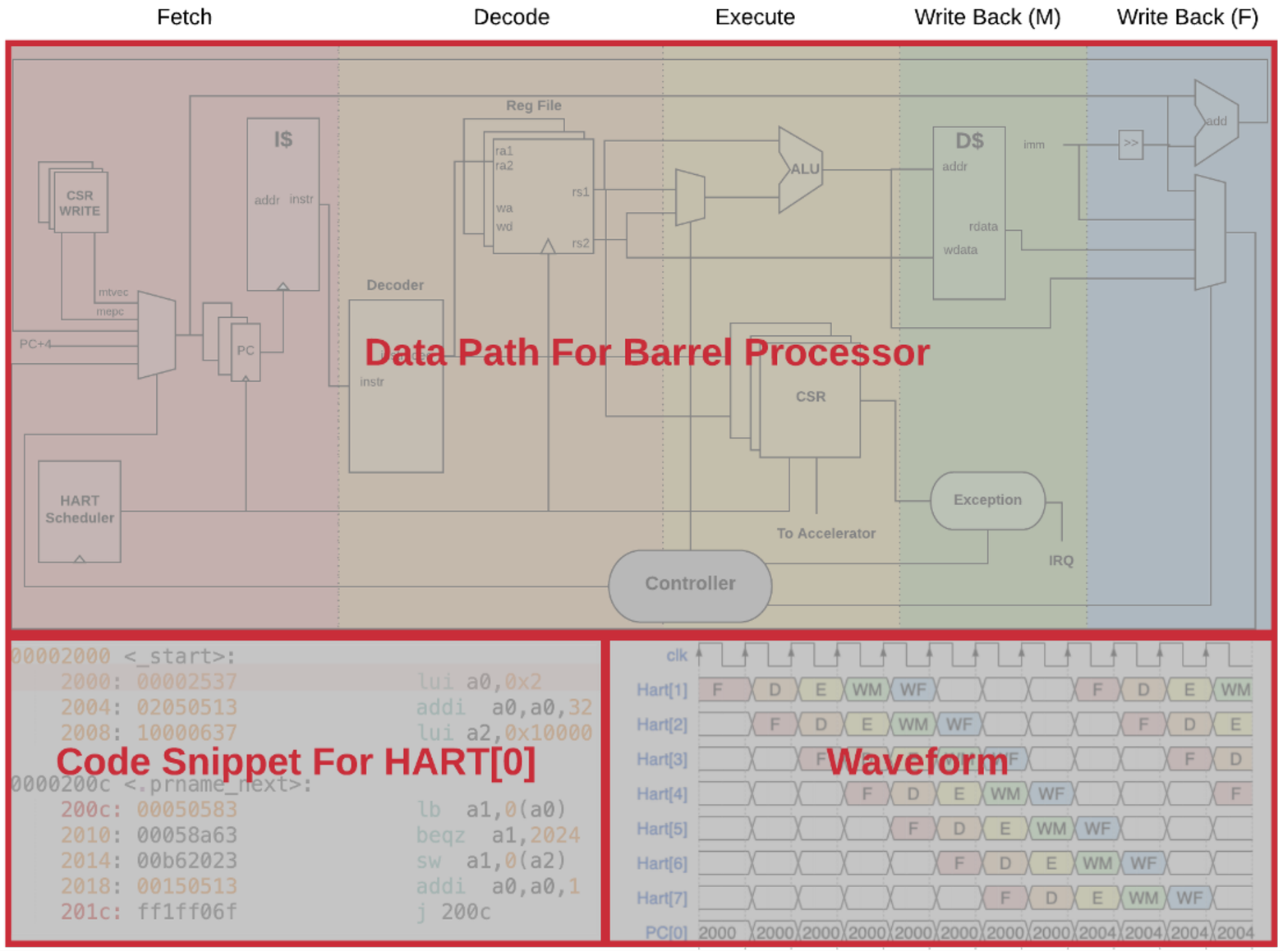
- How to control multiple processing elements?
  - Solution1: Use a separate controller for each PE.
  - Solution2: Use a shared controller for all PEs.
  - Our Solution: Barrel Processing: Share data path with multiple hardware threads (HARTs).
    - High throughput, fine grain control, low resource usage, no need for data or control hazard logic, etc



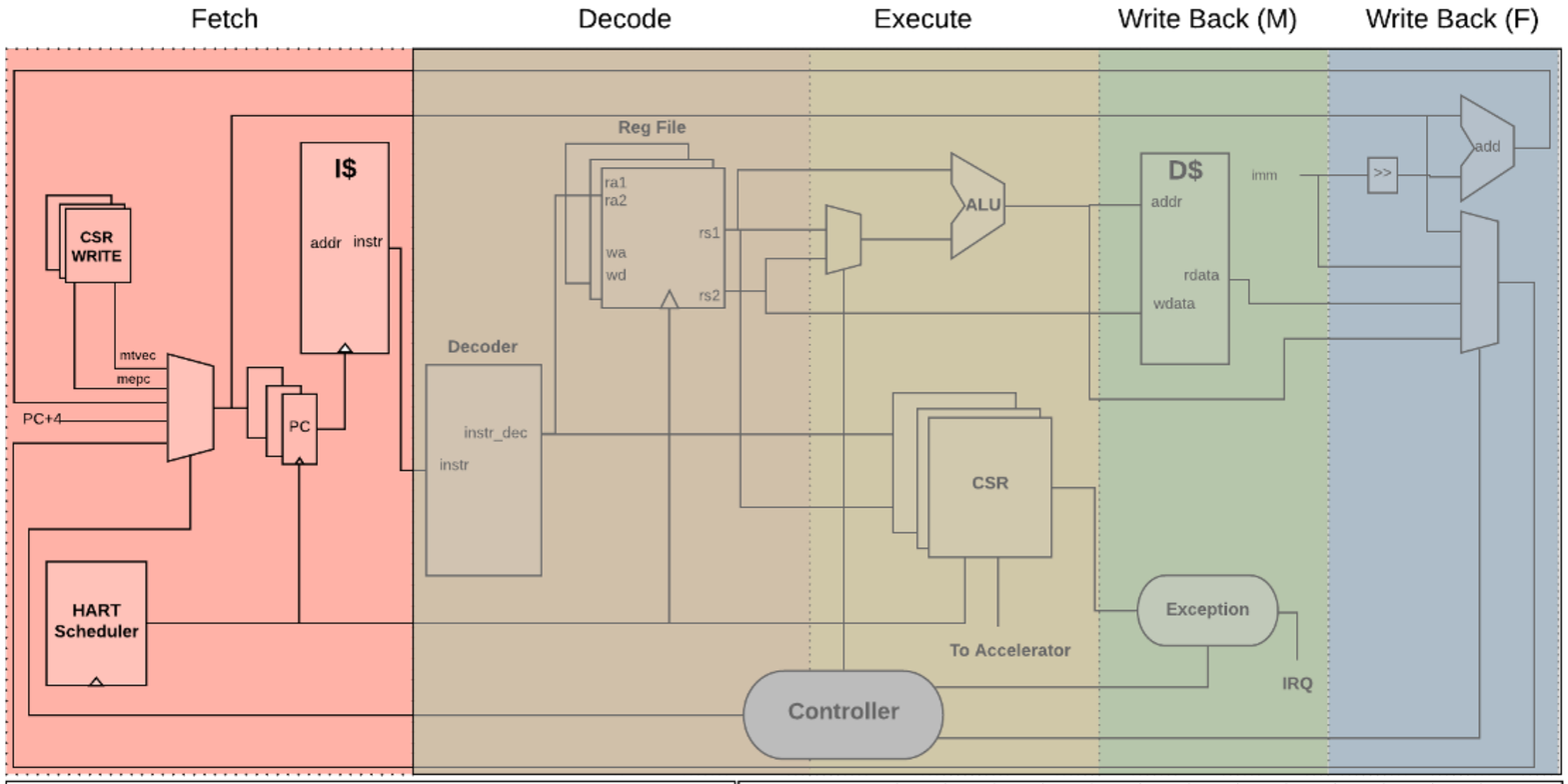
2.2. PITO: Multithreaded RISC-V Controller (4)



2.2. PITO: Multithreaded RISC-V Controller (4)

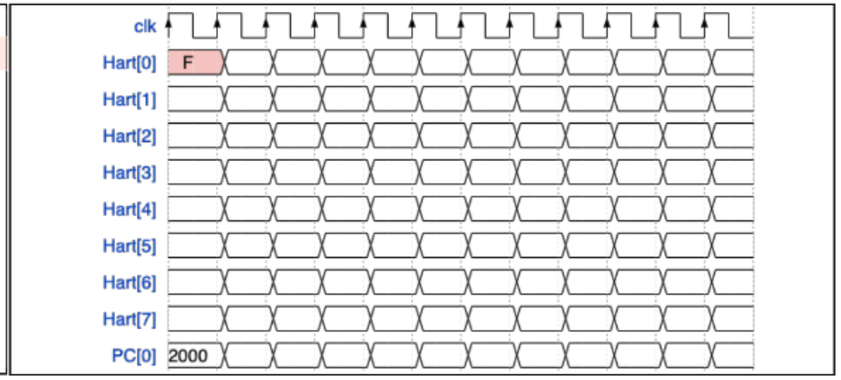


2.2. PITO: Multithreaded RISC-V Controller (4)

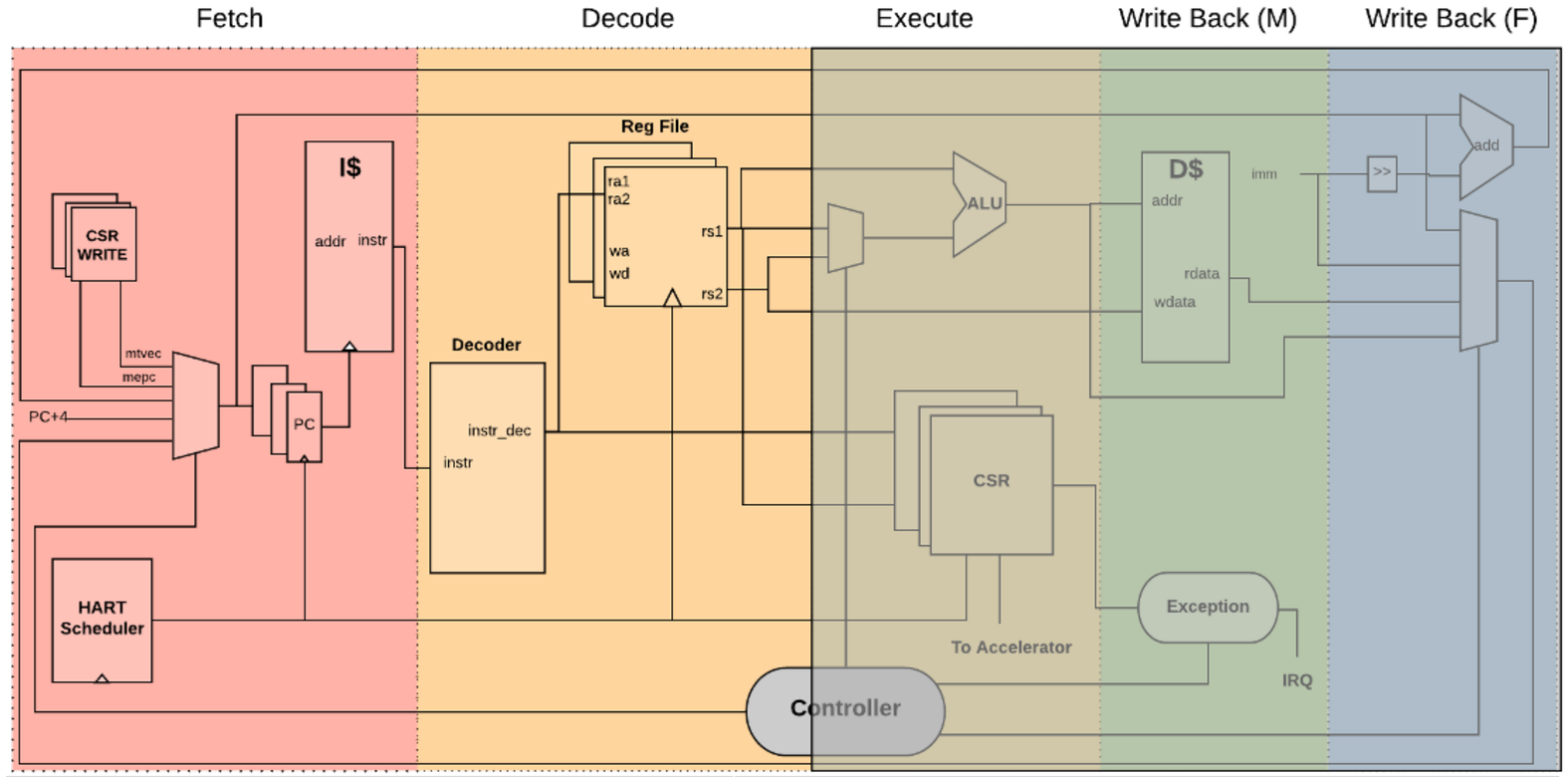



```
00002000 <_start>:
    2000: 00002537          lui a0,0x2
    2004: 02050513          addi  a0,a0,32
    2008: 10000637          lui a2,0x10000

0000200c <.prname_next>:
    200c: 00050583          lb  a1,0(a0)
    2010: 00058a63          beqz  a1,2024
    2014: 00b62023          sw  a1,0(a2)
    2018: 00150513          addi  a0,a0,1
    201c: ff1ff06f          j 200c
```

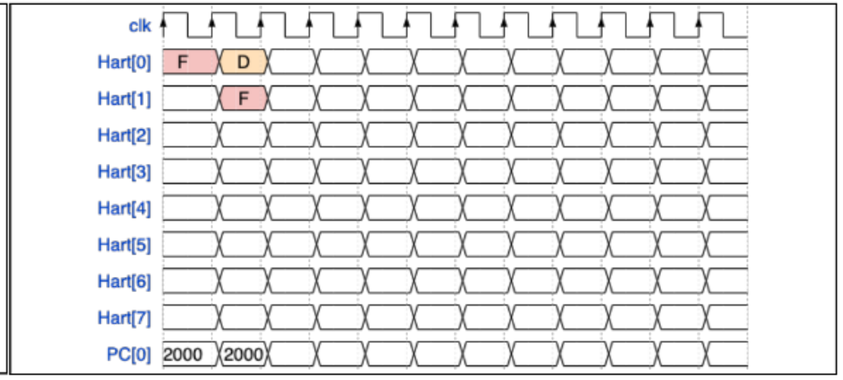


2.2. PITO: Multithreaded RISC-V Controller (4)

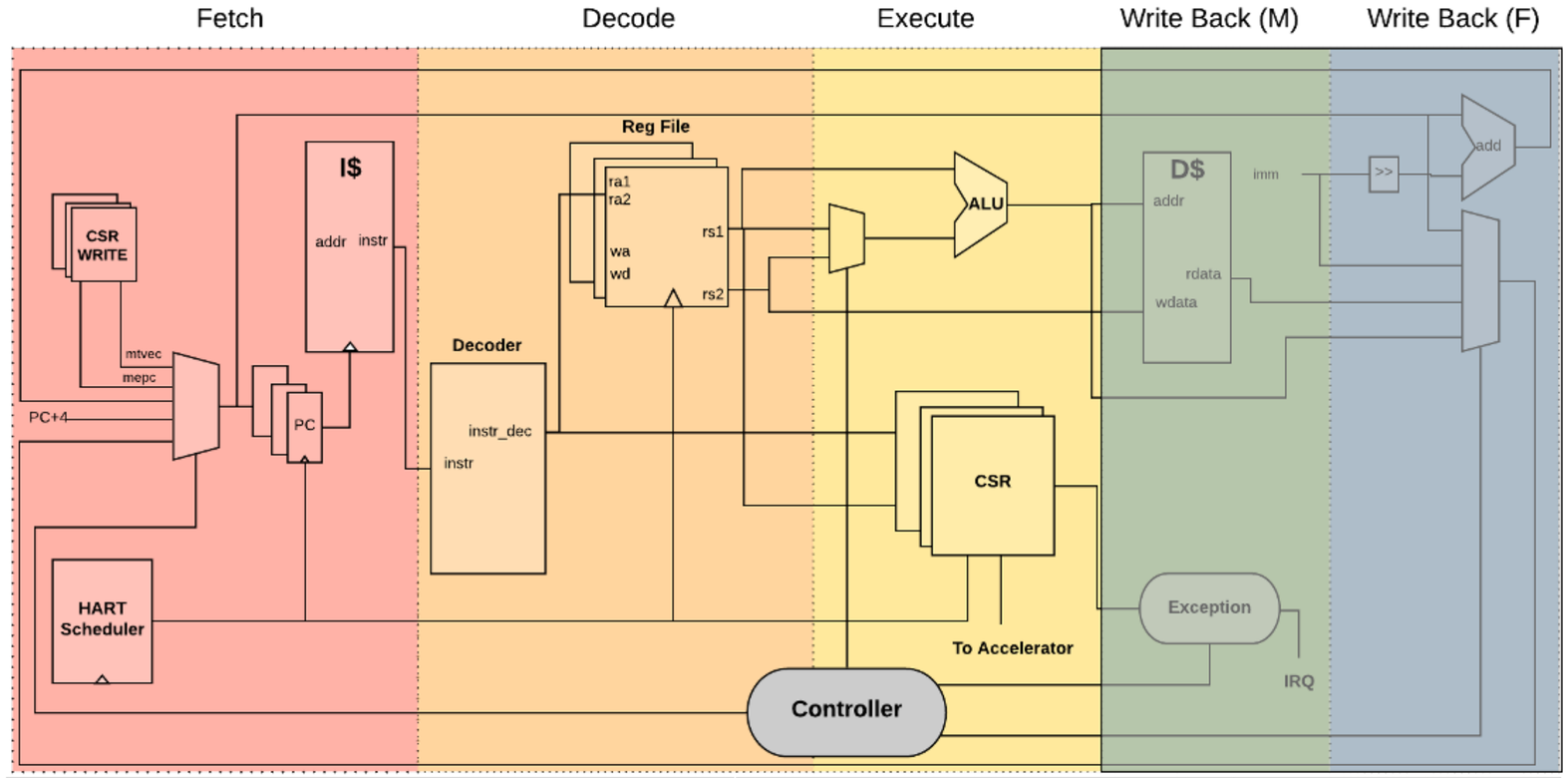



```
00002000 <_start>:
    2000: 00002537          lui a0,0x2
    2004: 02050513          addi  a0,a0,32
    2008: 10000637          lui a2,0x10000

0000200c <.prname_next>:
    200c: 00050583          lb  a1,0(a0)
    2010: 00058a63          beqz  a1,2024
    2014: 00b62023          sw  a1,0(a2)
    2018: 00150513          addi  a0,a0,1
    201c: ff1ff06f          j 200c
```

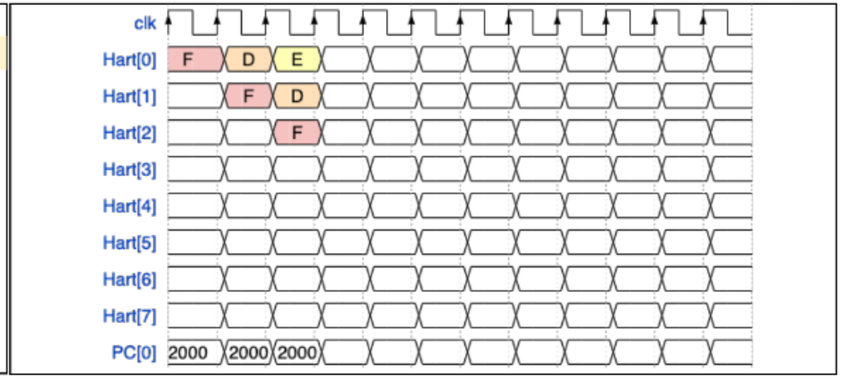


2.2. PITO: Multithreaded RISC-V Controller (4)

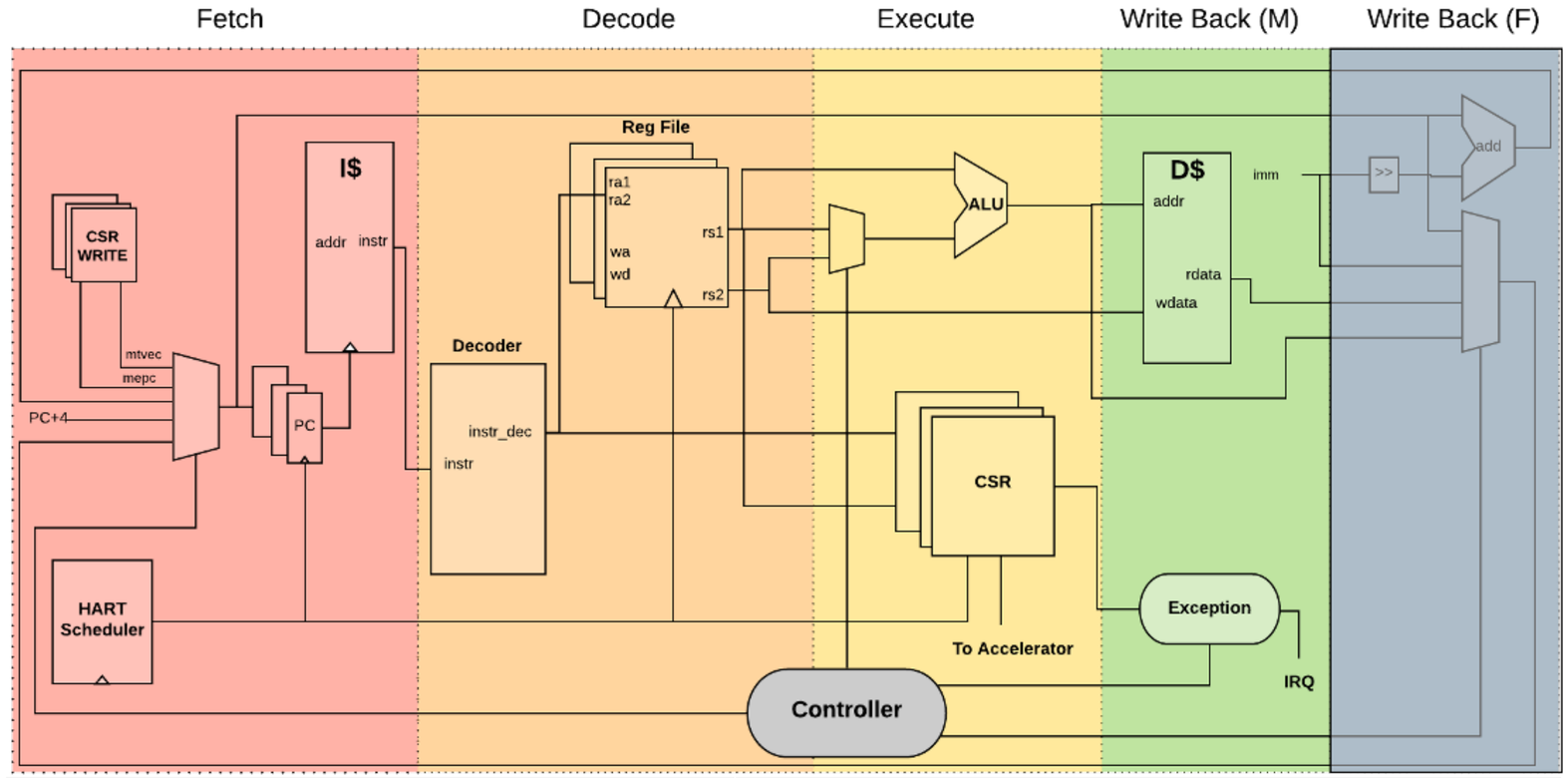



```
00002000 <_start>:
    2000: 00002537          lui a0,0x2
    2004: 02050513          addi  a0,a0,32
    2008: 10000637          lui a2,0x10000

0000200c <.prname_next>:
    200c: 00050583          lb  a1,0(a0)
    2010: 00058a63          beqz  a1,2024
    2014: 00b62023          sw  a1,0(a2)
    2018: 00150513          addi  a0,a0,1
    201c: ff1ff06f          j 200c
```

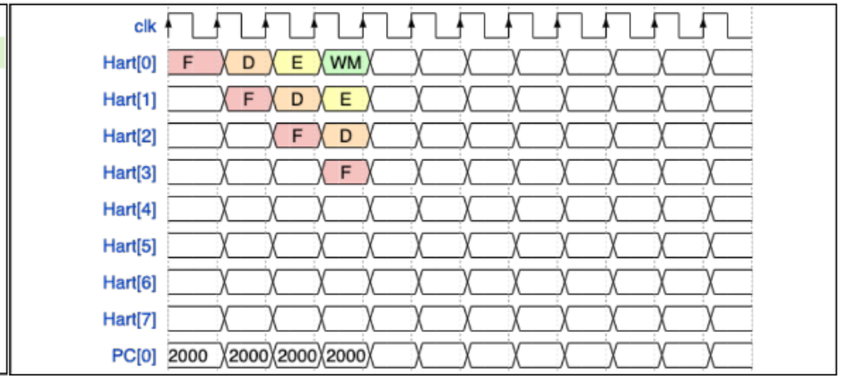


2.2. PITO: Multithreaded RISC-V Controller (4)

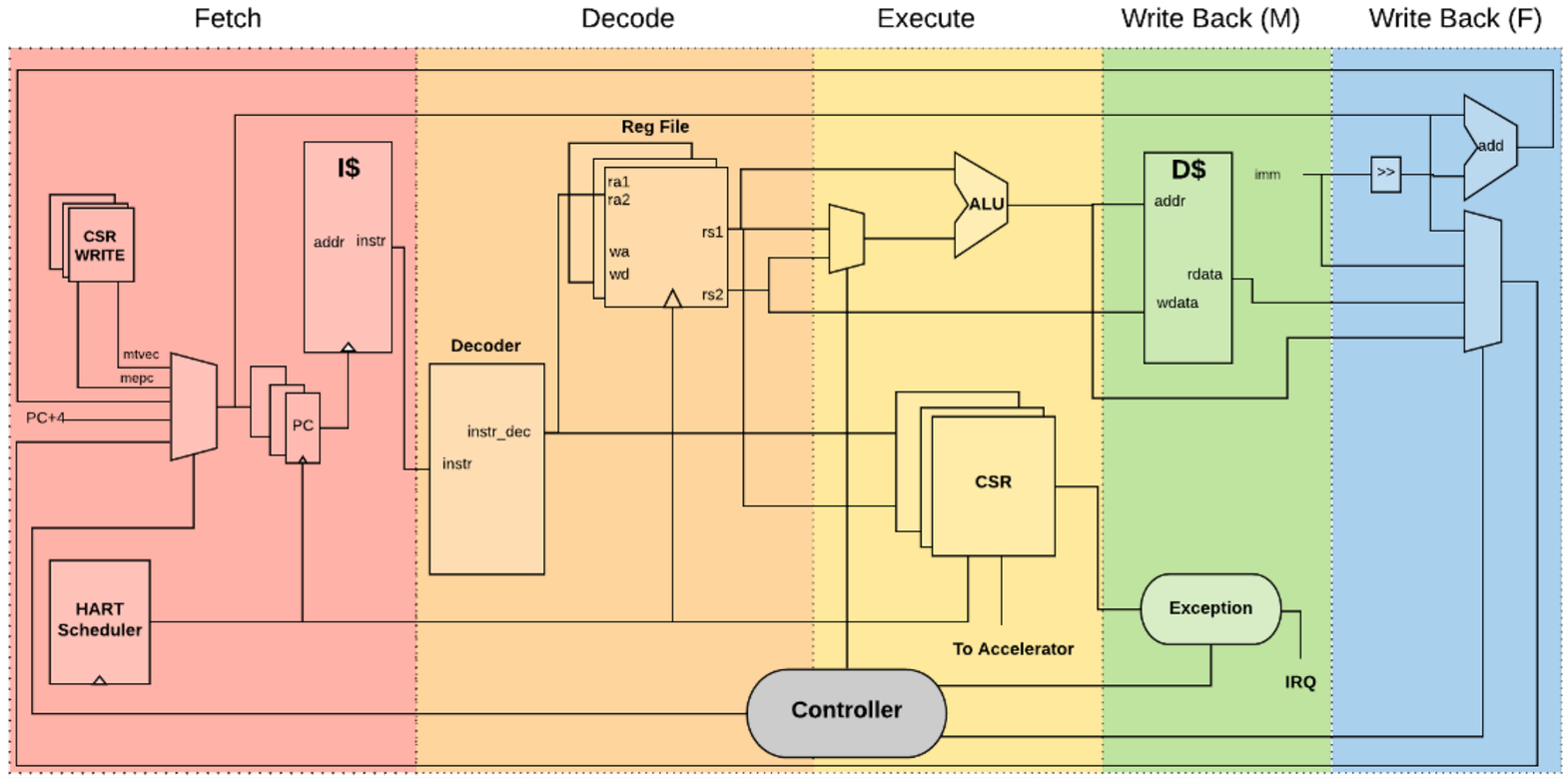



```
00002000 <_start>:
    2000: 00002537          lui a0,0x2
    2004: 02050513          addi  a0,a0,32
    2008: 10000637          lui a2,0x10000

0000200c <.prname_next>:
    200c: 00050583          lb  a1,0(a0)
    2010: 00058a63          beqz  a1,2024
    2014: 00b62023          sw  a1,0(a2)
    2018: 00150513          addi  a0,a0,1
    201c: ff1ff06f          j 200c
```

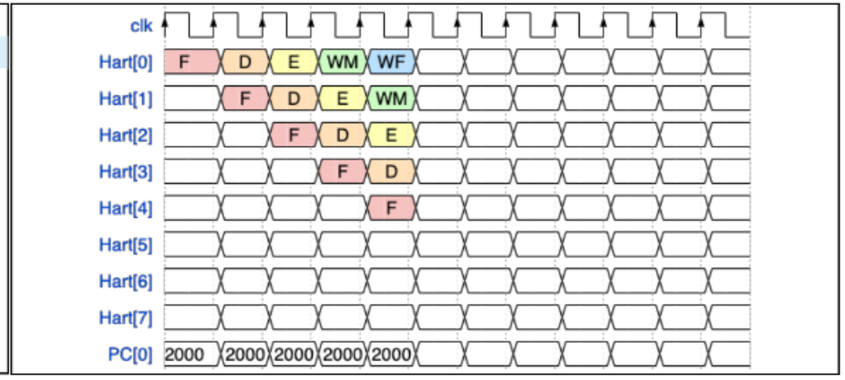


2.2. PITO: Multithreaded RISC-V Controller (4)

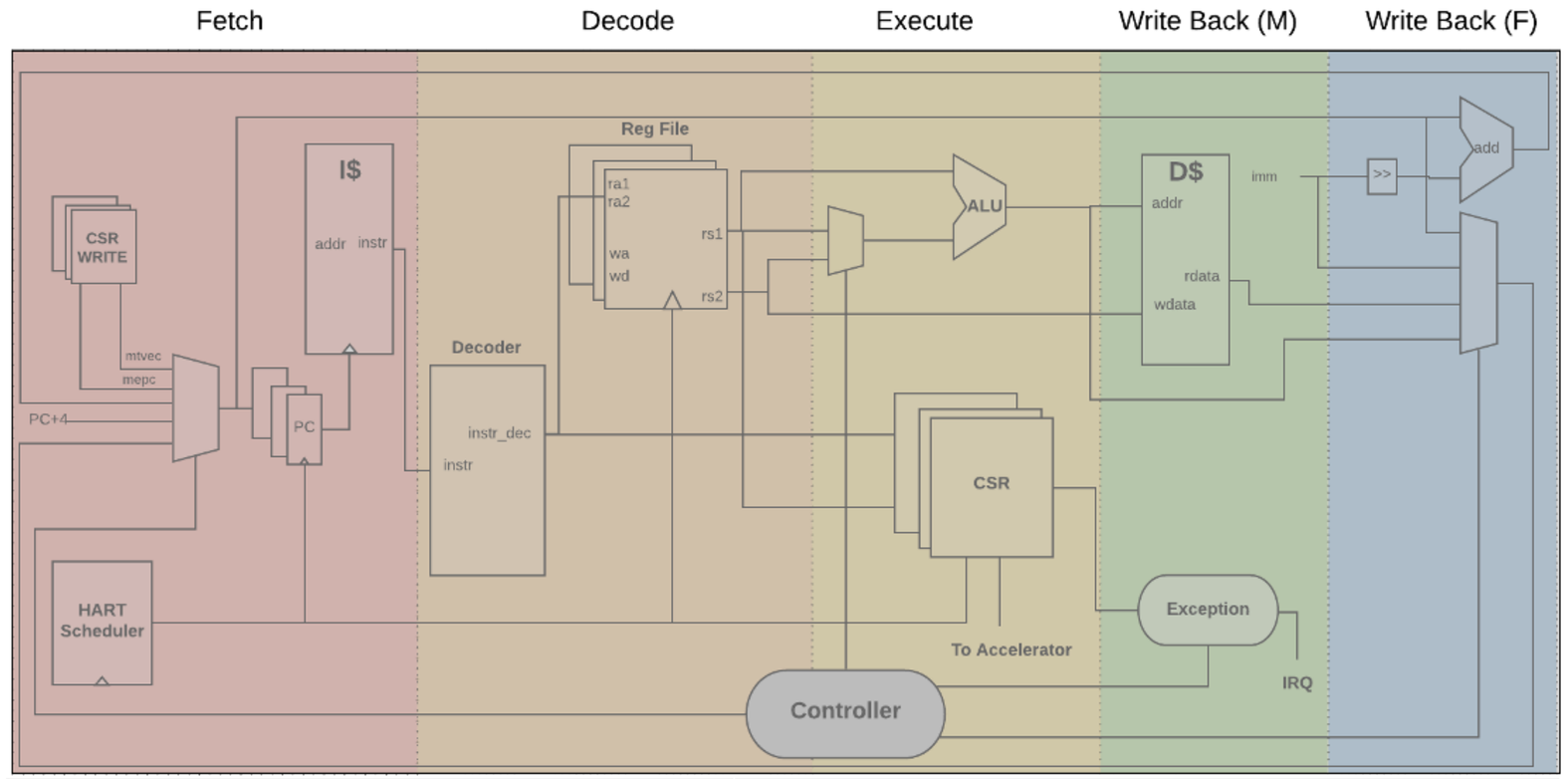



```
00002000 <_start>:
    2000: 00002537          lui a0,0x2
    2004: 02050513          addi  a0,a0,32
    2008: 10000637          lui a2,0x10000

0000200c <.prname_next>:
    200c: 00050583          lb  a1,0(a0)
    2010: 00058a63          beqz  a1,2024
    2014: 00b62023          sw  a1,0(a2)
    2018: 00150513          addi  a0,a0,1
    201c: ff1ff06f          j 200c
```

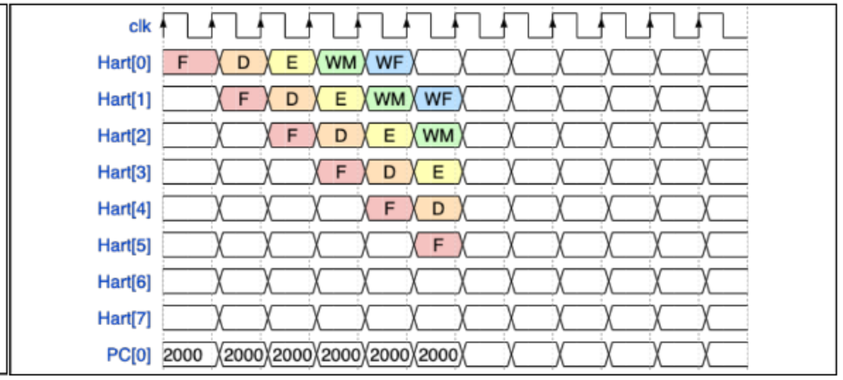


2.2. PITO: Multithreaded RISC-V Controller (4)

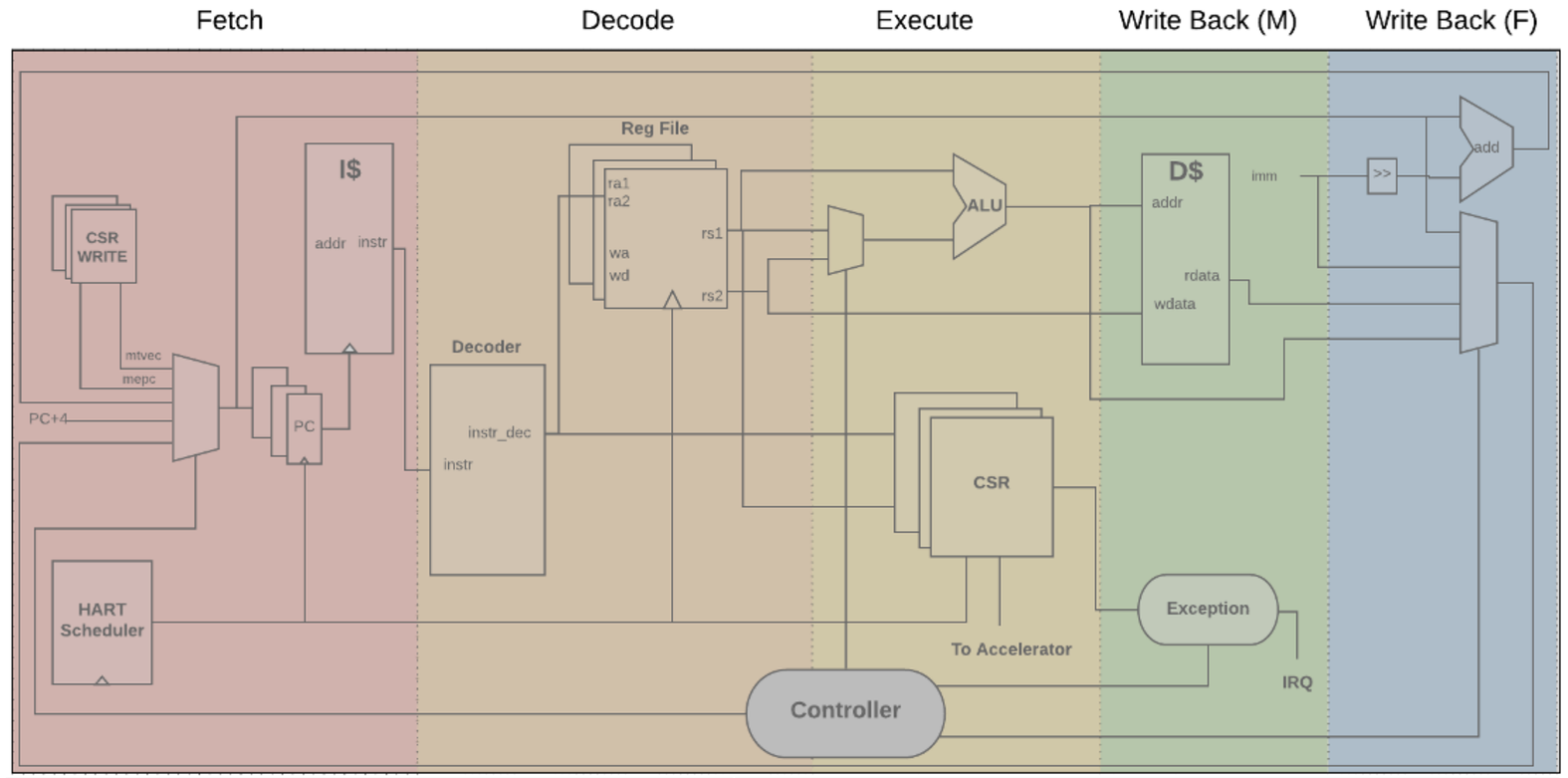



```
00002000 <_start>:
    2000: 00002537          lui a0,0x2
    2004: 02050513          addi  a0,a0,32
    2008: 10000637          lui a2,0x10000

0000200c <.prname_next>:
    200c: 00050583          lb  a1,0(a0)
    2010: 00058a63          beqz  a1,2024
    2014: 00b62023          sw  a1,0(a2)
    2018: 00150513          addi  a0,a0,1
    201c: ff1ff06f          j 200c
```

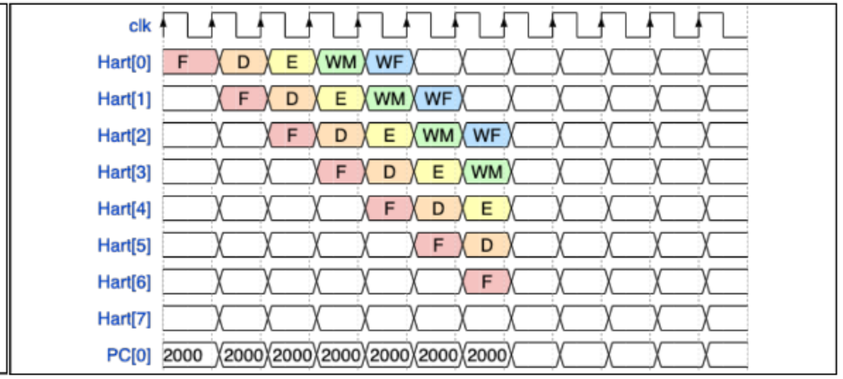


2.2. PITO: Multithreaded RISC-V Controller (4)

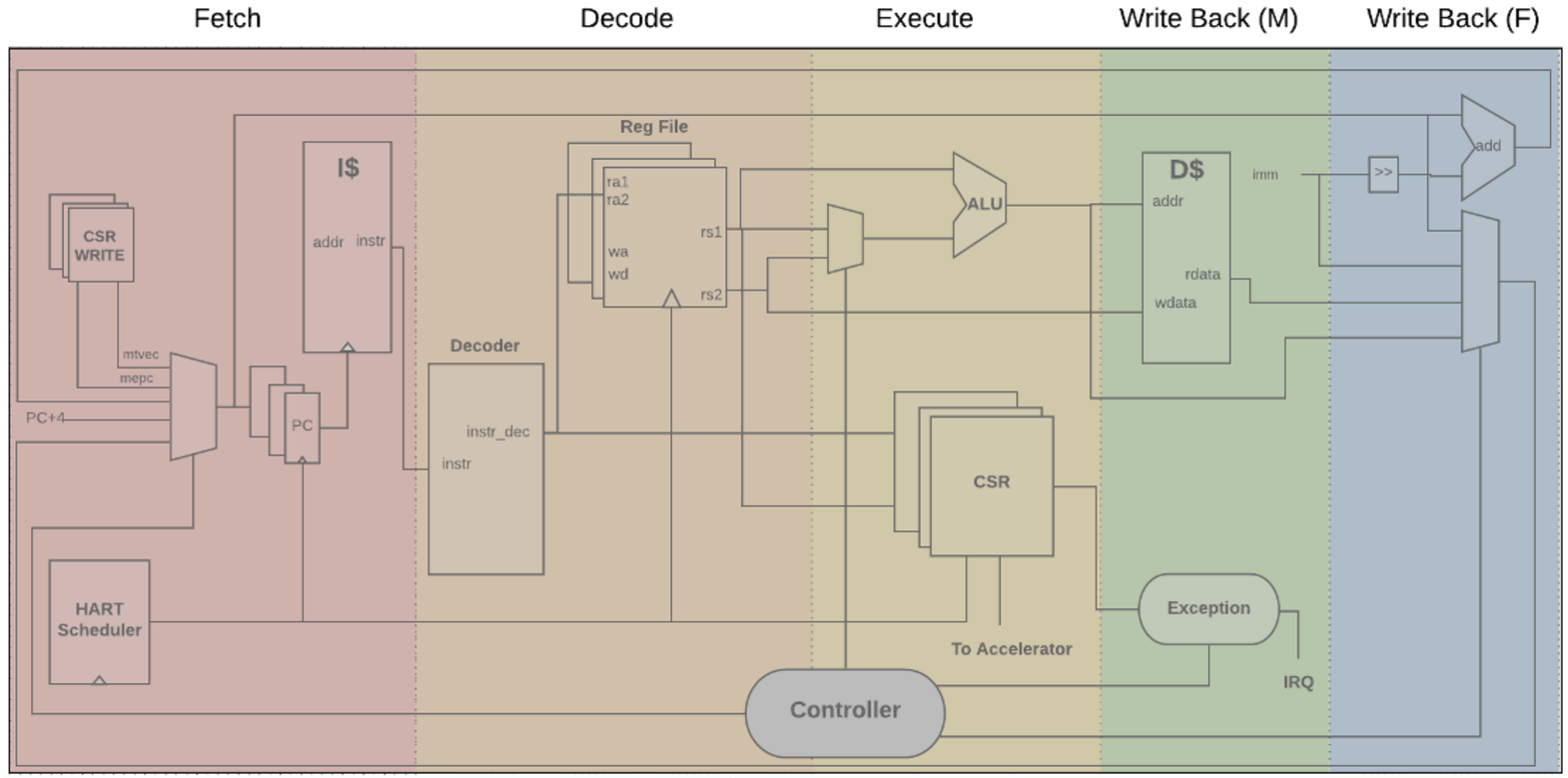



```
00002000 <_start>:
    2000: 00002537          lui a0,0x2
    2004: 02050513          addi  a0,a0,32
    2008: 10000637          lui a2,0x10000

0000200c <.prname_next>:
    200c: 00050583          lb  a1,0(a0)
    2010: 00058a63          beqz  a1,2024
    2014: 00b62023          sw  a1,0(a2)
    2018: 00150513          addi  a0,a0,1
    201c: ff1ff06f          j 200c
```

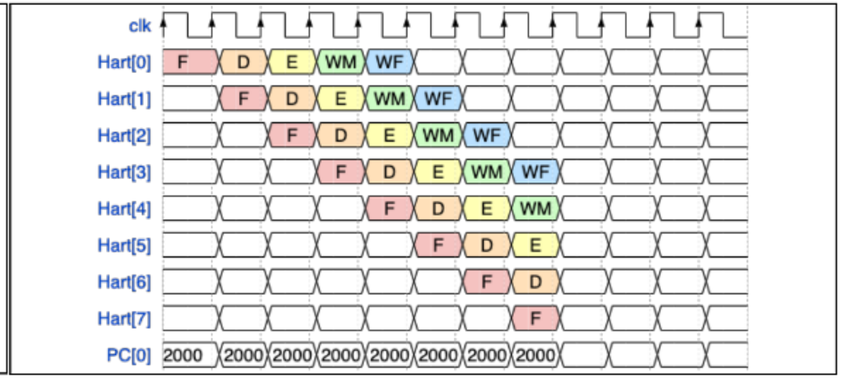


2.2. PITO: Multithreaded RISC-V Controller (4)

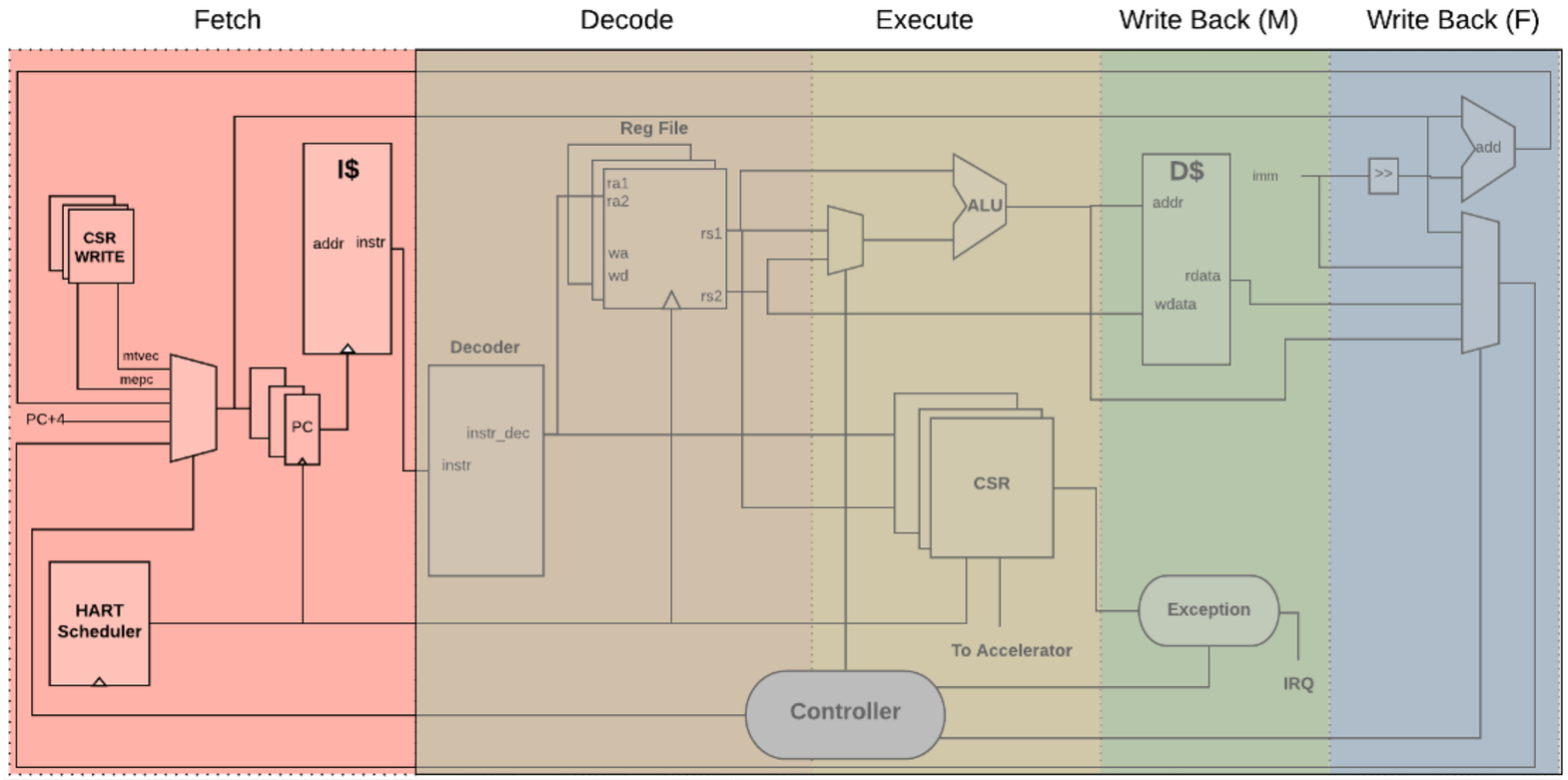



```
00002000 <_start>:
    2000: 00002537          lui a0,0x2
    2004: 02050513          addi  a0,a0,32
    2008: 10000637          lui a2,0x10000

0000200c <.prname_next>:
    200c: 00050583          lb  a1,0(a0)
    2010: 00058a63          beqz  a1,2024
    2014: 00b62023          sw  a1,0(a2)
    2018: 00150513          addi  a0,a0,1
    201c: ff1ff06f          j 200c
```

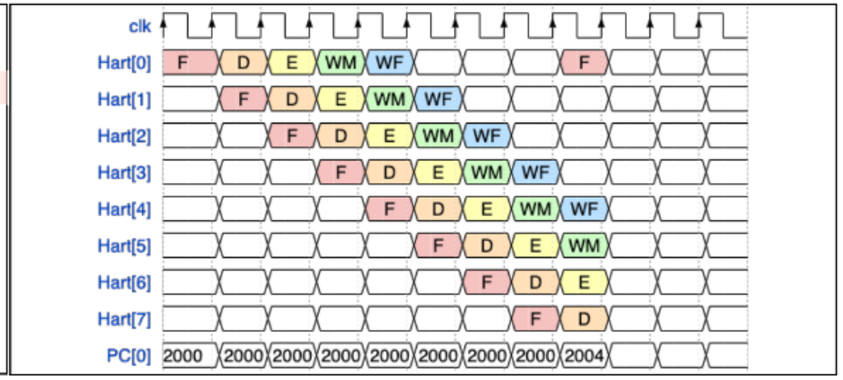


2.2. PITO: Multithreaded RISC-V Controller (4)

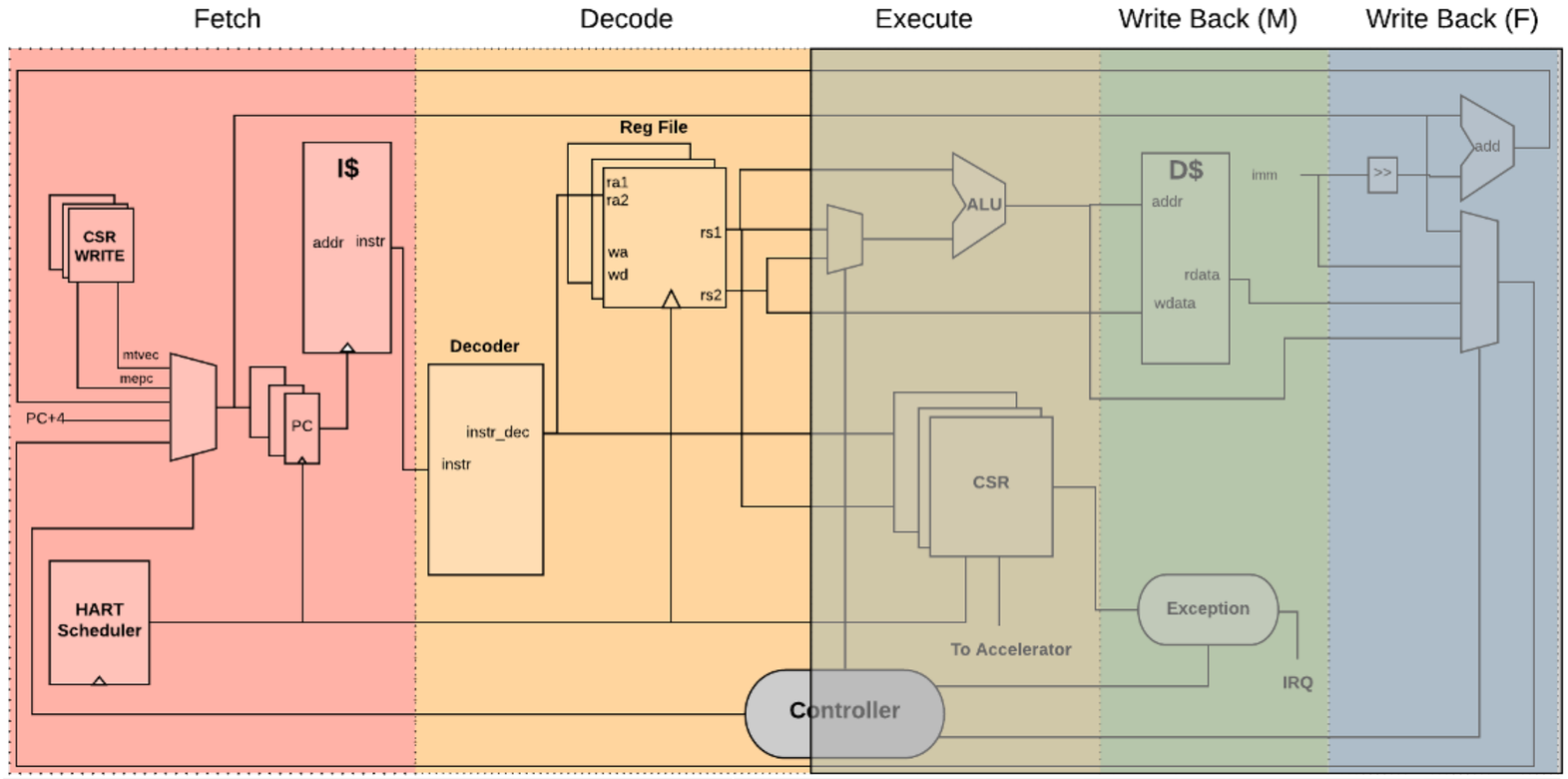



```
00002000 <_start>:
    2000: 00002537          lui a0,0x2
    2004: 02050513          addi  a0,a0,32
    2008: 10000637          lui a2,0x10000

0000200c <.prname_next>:
    200c: 00050583          lb  a1,0(a0)
    2010: 00058a63          beqz  a1,2024
    2014: 00b62023          sw  a1,0(a2)
    2018: 00150513          addi  a0,a0,1
    201c: ff1ff06f          j 200c
```

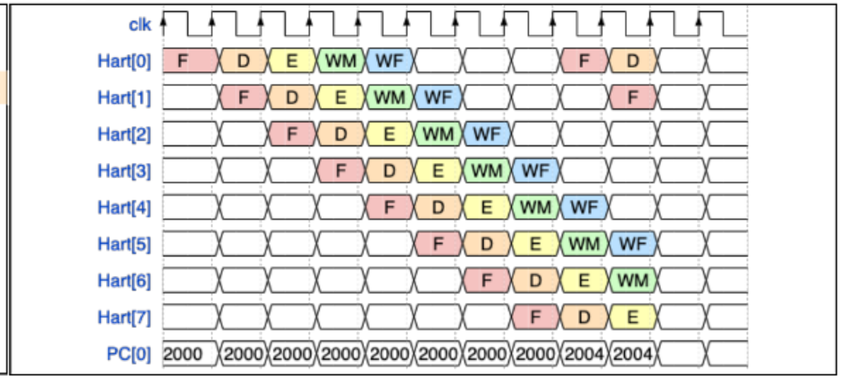


2.2. PITO: Multithreaded RISC-V Controller (4)



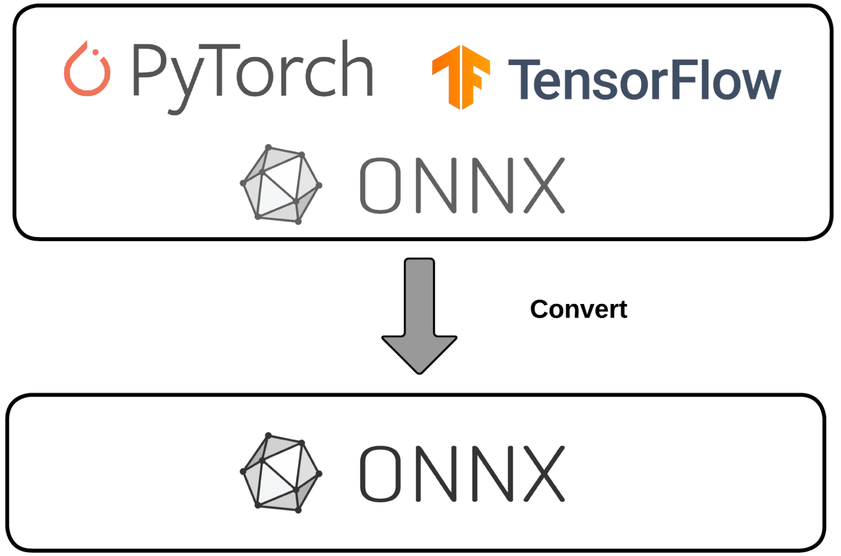

```
00002000 <_start>:
    2000: 00002537          lui a0,0x2
    2004: 02050513          addi  a0,a0,32
    2008: 10000637          lui a2,0x10000

0000200c <.prname_next>:
    200c: 00050583          lb  a1,0(a0)
    2010: 00058a63          beqz  a1,2024
    2014: 00b62023          sw  a1,0(a2)
    2018: 00150513          addi  a0,a0,1
    201c: ff1ff06f          j 200c
```



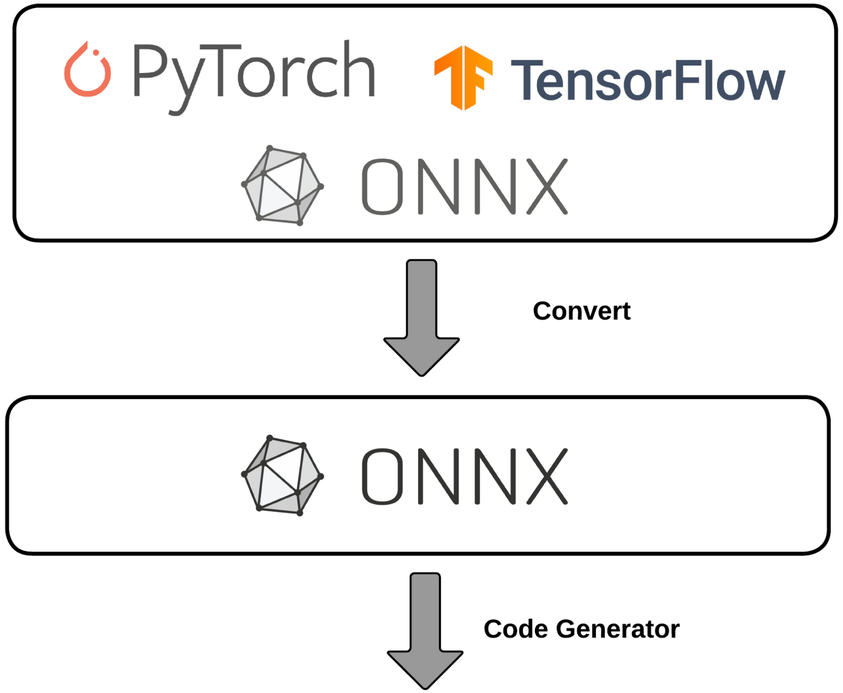
3. BARVINN programming model and software stack

- BARVINN takes in a model in onnx format.



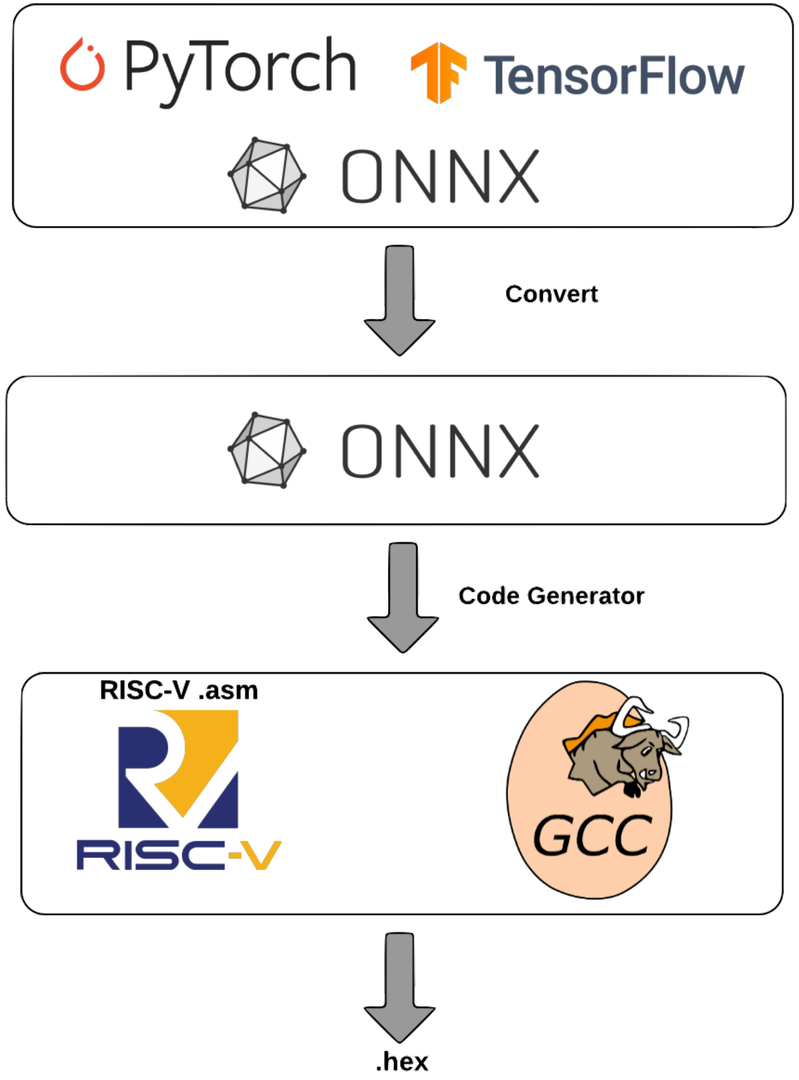
3. BARVINN programming model and software stack

- BARVINN takes in a model in onnx format.
- Our code generator traverse the computation graph and based on the node type, it generates the appropriate jobs and assigns them to different MVUs.
- The code generator then produces C code for each node.



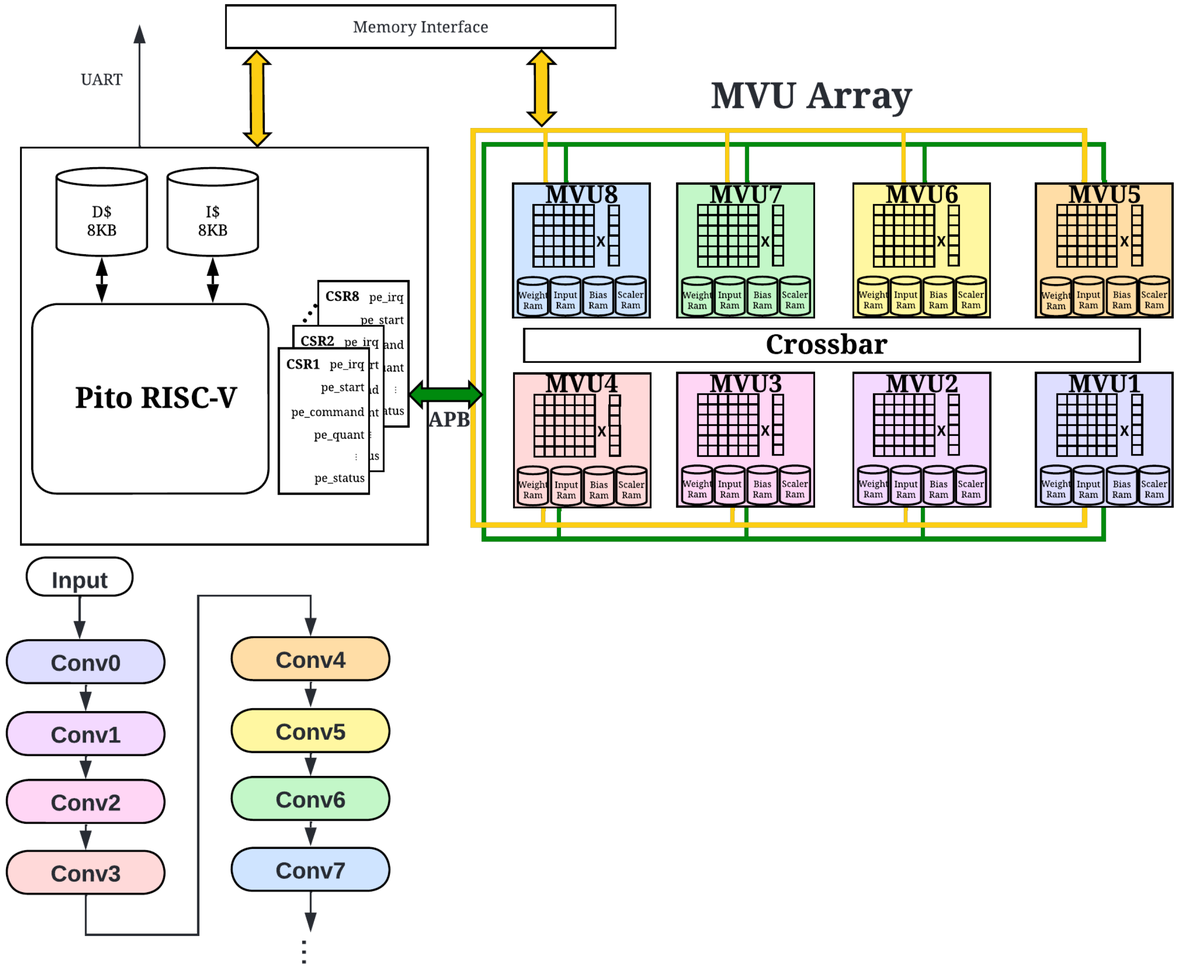
3. BARVINN programming model and software stack

- BARVINN takes in a model in onnx format.
- Our code generator traverse the computation graph and based on the node type, it generates the appropriate jobs and assigns them to different MVUs.
- The code generator then produces C code for each node.
- Finally, using RISC-V toolchain, our C-Runtime and memory map, a binary is generated.



3. BARVINN programming model and software stack

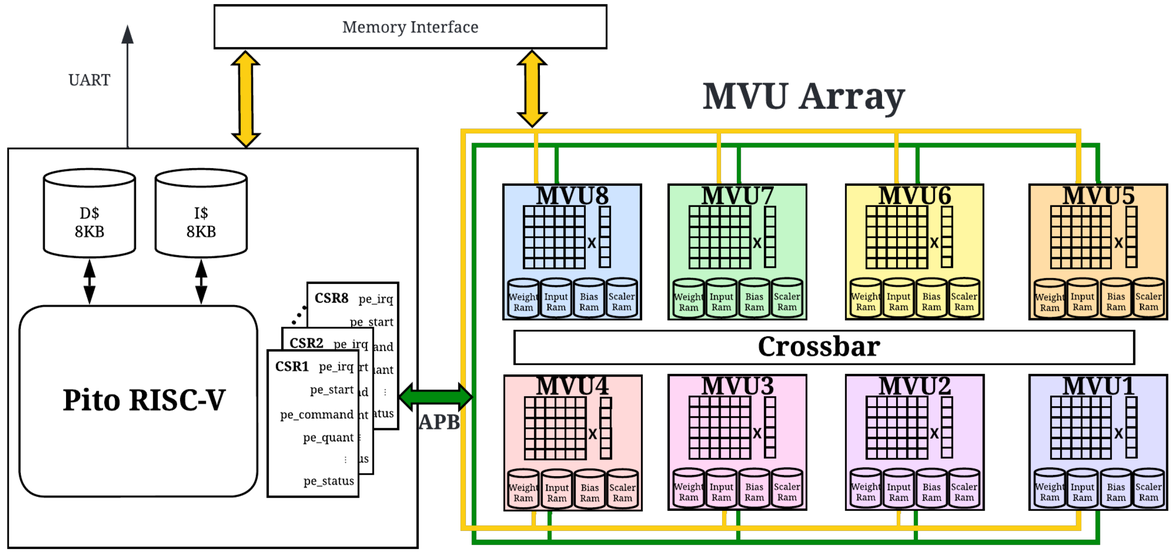
- BARVINN can be programmed for two computation modes:
  - Pipelined mode: Each computation node is assigned to a separate MVU.

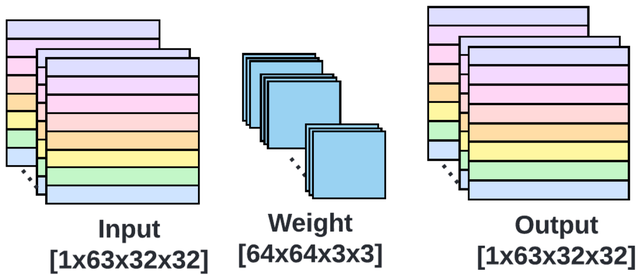


Pipelined Mode

3. BARVINN programming model and software stack

- BARVINN can be programmed for two computation modes:
  - Pipelined mode: Each computation node is assigned to a separate MVU.
  - Distributed mode: Computation of a single layer is distributed among multiple MVUs.

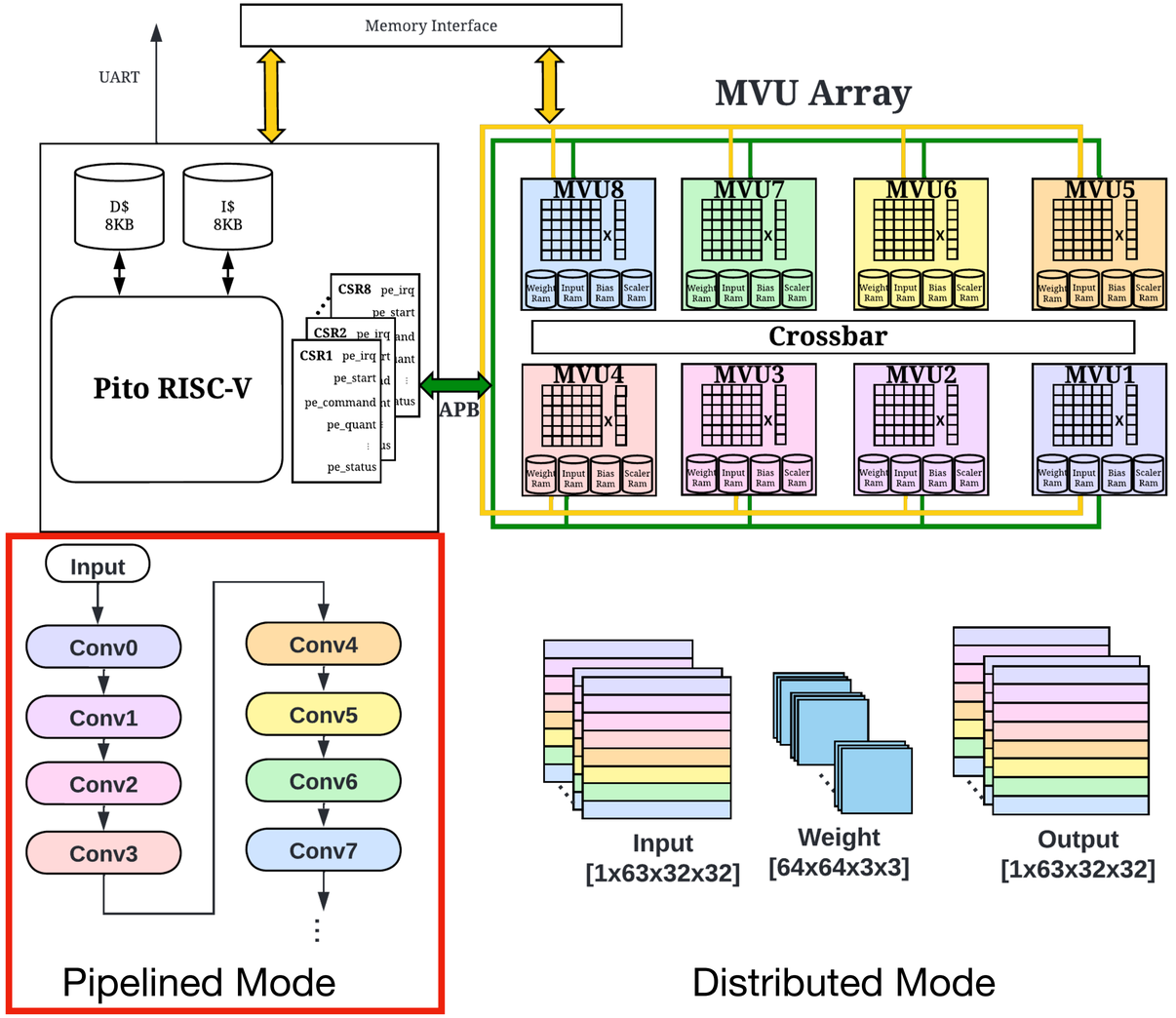




Distributed Mode

3. BARVINN programming model and software stack

- BARVINN can be programmed for two computation modes:
  - Pipelined mode: Each computation node is assigned to a separate MVU.
  - Distributed mode: Computation of a single layer is distributed among multiple MVUs.
  - For now, our code generator only supports pipelined mode code generation.



3. Use case: Matrix Multiply

- Example, Matrix Multiply:
  - Weight Matrix: 128 x 128
  - Input Vector: 8 x 128
  - Weight precision: 2 bits
  - Input precision: 2 bits
  - Activation precision: 2 bits

For each hart

```
#**************************************************************************
# matmul.S
# Kernel code for Matrix Multiply
#--------------------------------------------------------------------------
#include "pito_def.h"

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 mvu_quant     , 10    // set quant_msbidx to 10
csrwi mvu_wbaseaddr , 0     // set weight address to 0
csrwi mvu_ibaseaddr , 0     // set input address to 0
addi  x1, x0, 1
slli  x1, x1, 10            // set output address to 0x400
csrw mvu_obaseaddr , x1
csrwi mvu_wstride_0 , 30    // 1 tile back move x 2 bits
csrwi mvu_wstride_1 ,  2    // 1 tile ahead move x 2 bits
csrwi mvu_wstride_2 ,  0
csrwi mvu_wstride_3 ,  0
csrwi mvu_istride_0 , 30    // 1 tile back move x 2 bits
csrwi mvu_istride_1 ,  0
csrwi mvu_istride_2 ,  0
csrwi mvu_istride_3 , 30
csrwi mvu_ostride_0 ,  0
csrwi mvu_ostride_1 ,  0
csrwi mvu_ostride_2 ,  0
csrwi mvu_ostride_3 ,  0
csrwi mvu_wlength_0 ,  1    // 2 tiles in width
csrwi mvu_wlength_1 ,  3    // number bit combinations i.e. 2x2 bits
csrwi mvu_wlength_2 ,  1    // 2 tiles in height
csrwi mvu_wlength_3 ,  0
csrwi mvu_ilength_0 ,  1    // 2 tiles in height
csrwi mvu_ilength_1 ,  0    // number bit combinations
csrwi mvu_ilength_2 ,  0    // 2 tiles in width of matrix operand
csrwi mvu_ilength_3 ,  0
csrwi mvu_olength_0 ,  1
csrwi mvu_olength_1 ,  ś
csrwi mvu_olength_2 ,  0
csrwi mvu_olength_3 ,  0
addi x1, x0, 1
slli x1, x1, 30             // mul mode 01
addi x1, x1, 16
csrw mvu_command, x1        // Kick start MVU, 2 tiles x 2 tiles x 2bit x 2bits
ebreak
```

Setting input, weight and output precision

```
#*****************************************************************************
# matmul.S
# Kernel code for Matrix Multiply
#-----------------------------------------------------------------------------
#include "pito_def.h"

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 mvu_quant     , 10    // set quant_msbidx to 10
csrwi mvu_wbaseaddr , 0     // set weight address to 0
csrwi mvu_ibaseaddr , 0     // set input address to 0
addi  x1, x0, 1
slli  x1, x1, 10            // set output address to 0x400
csrw mvu_obaseaddr , x1
csrwi mvu_wstride_0 , 30    // 1 tile back move x 2 bits
csrwi mvu_wstride_1 ,  2    // 1 tile ahead move x 2 bits
csrwi mvu_wstride_2 ,  0
csrwi mvu_wstride_3 ,  0
csrwi mvu_istride_0 , 30    // 1 tile back move x 2 bits
csrwi mvu_istride_1 ,  0
csrwi mvu_istride_2 ,  0
csrwi mvu_istride_3 , 30
csrwi mvu_ostride_0 ,  0
csrwi mvu_ostride_1 ,  0
csrwi mvu_ostride_2 ,  0
csrwi mvu_ostride_3 ,  0
csrwi mvu_wlength_0 ,  1    // 2 tiles in width
csrwi mvu_wlength_1 ,  3    // number bit combinations i.e. 2x2 bits
csrwi mvu_wlength_2 ,  1    // 2 tiles in height
csrwi mvu_wlength_3 ,  0
csrwi mvu_ilength_0 ,  1    // 2 tiles in height
csrwi mvu_ilength_1 ,  0    // number bit combinations
csrwi mvu_ilength_2 ,  0    // 2 tiles in width of matrix operand
csrwi mvu_ilength_3 ,  0
csrwi mvu_olength_0 ,  1
csrwi mvu_olength_1 ,  ś
csrwi mvu_olength_2 ,  0
csrwi mvu_olength_3 ,  0
addi x1, x0, 1
slli x1, x1, 30             // mul mode 01
addi x1, x1, 16
csrw mvu_command, x1        // Kick start MVU, 2 tiles x 2 tiles x 2bit x 2bits
ebreak
```

Setting input, weight and output precision

Setting input, weight and output address

```
#*****************************************************************************
# matmul.S
# Kernel code for Matrix Multiply
#-----------------------------------------------------------------------------
#include "pito_def.h"

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 mvu_quant     , 10    // set quant_msbidx to 10
csrwi mvu_wbaseaddr , 0     // set weight address to 0
csrwi mvu_ibaseaddr , 0     // set input address to 0
addi  x1, x0, 1
slli  x1, x1, 10            // set output address to 0x400
csrw mvu_obaseaddr , x1
csrwi mvu_wstride_0 , 30    // 1 tile back move x 2 bits
csrwi mvu_wstride_1 ,  2    // 1 tile ahead move x 2 bits
csrwi mvu_wstride_2 ,  0
csrwi mvu_wstride_3 ,  0
csrwi mvu_istride_0 , 30    // 1 tile back move x 2 bits
csrwi mvu_istride_1 ,  0
csrwi mvu_istride_2 ,  0
csrwi mvu_istride_3 , 30
csrwi mvu_ostride_0 ,  0
csrwi mvu_ostride_1 ,  0
csrwi mvu_ostride_2 ,  0
csrwi mvu_ostride_3 ,  0
csrwi mvu_wlength_0 ,  1    // 2 tiles in width
csrwi mvu_wlength_1 ,  3    // number bit combinations i.e. 2x2 bits
csrwi mvu_wlength_2 ,  1    // 2 tiles in height
csrwi mvu_wlength_3 ,  0
csrwi mvu_ilength_0 ,  1    // 2 tiles in height
csrwi mvu_ilength_1 ,  0    // number bit combinations
csrwi mvu_ilength_2 ,  0    // 2 tiles in width of matrix operand
csrwi mvu_ilength_3 ,  0
csrwi mvu_olength_0 ,  1
csrwi mvu_olength_1 ,  ś
csrwi mvu_olength_2 ,  0
csrwi mvu_olength_3 ,  0
addi x1, x0, 1
slli x1, x1, 30             // mul mode 01
addi x1, x1, 16
csrw mvu_command, x1        // Kick start MVU, 2 tiles x 2 tiles x 2bit x 2bits
ebreak
```

Setting input, weight and output precision

Setting input, weight and output address

Setting input, weight and output memory access pattern variables

```
#*****************************************************************************
# matmul.S
# Kernel code for Matrix Multiply
#-----------------------------------------------------------------------------
#include "pito_def.h"

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 mvu_quant     , 10    // set quant_msbidx to 10
csrwi mvu_wbaseaddr , 0     // set weight address to 0
csrwi mvu_ibaseaddr , 0     // set input address to 0
addi  x1, x0, 1
slli  x1, x1, 10            // set output address to 0x400
csrw mvu_obaseaddr , x1
csrwi mvu_wstride_0 , 30    // 1 tile back move x 2 bits
csrwi mvu_wstride_1 ,  2    // 1 tile ahead move x 2 bits
csrwi mvu_wstride_2 ,  0
csrwi mvu_wstride_3 ,  0
csrwi mvu_istride_0 , 30    // 1 tile back move x 2 bits
csrwi mvu_istride_1 ,  0
csrwi mvu_istride_2 ,  0
csrwi mvu_istride_3 , 30
csrwi mvu_ostride_0 ,  0
csrwi mvu_ostride_1 ,  0
csrwi mvu_ostride_2 ,  0
csrwi mvu_ostride_3 ,  0
csrwi mvu_wlength_0 ,  1    // 2 tiles in width
csrwi mvu_wlength_1 ,  3    // number bit combinations i.e. 2x2 bits
csrwi mvu_wlength_2 ,  1    // 2 tiles in height
csrwi mvu_wlength_3 ,  0
csrwi mvu_ilength_0 ,  1    // 2 tiles in height
csrwi mvu_ilength_1 ,  0    // number bit combinations
csrwi mvu_ilength_2 ,  0    // 2 tiles in width of matrix operand
csrwi mvu_ilength_3 ,  0
csrwi mvu_olength_0 ,  1
csrwi mvu_olength_1 ,  ś
csrwi mvu_olength_2 ,  0
csrwi mvu_olength_3 ,  0
addi x1, x0, 1
slli x1, x1, 30             // mul mode 01
addi x1, x1, 16
csrw mvu_command, x1        // Kick start MVU, 2 tiles x 2 tiles x 2bit x 2bits
ebreak
```

Setting input, weight and output precision

Setting input, weight and output address

Setting input, weight and output memory access pattern variables

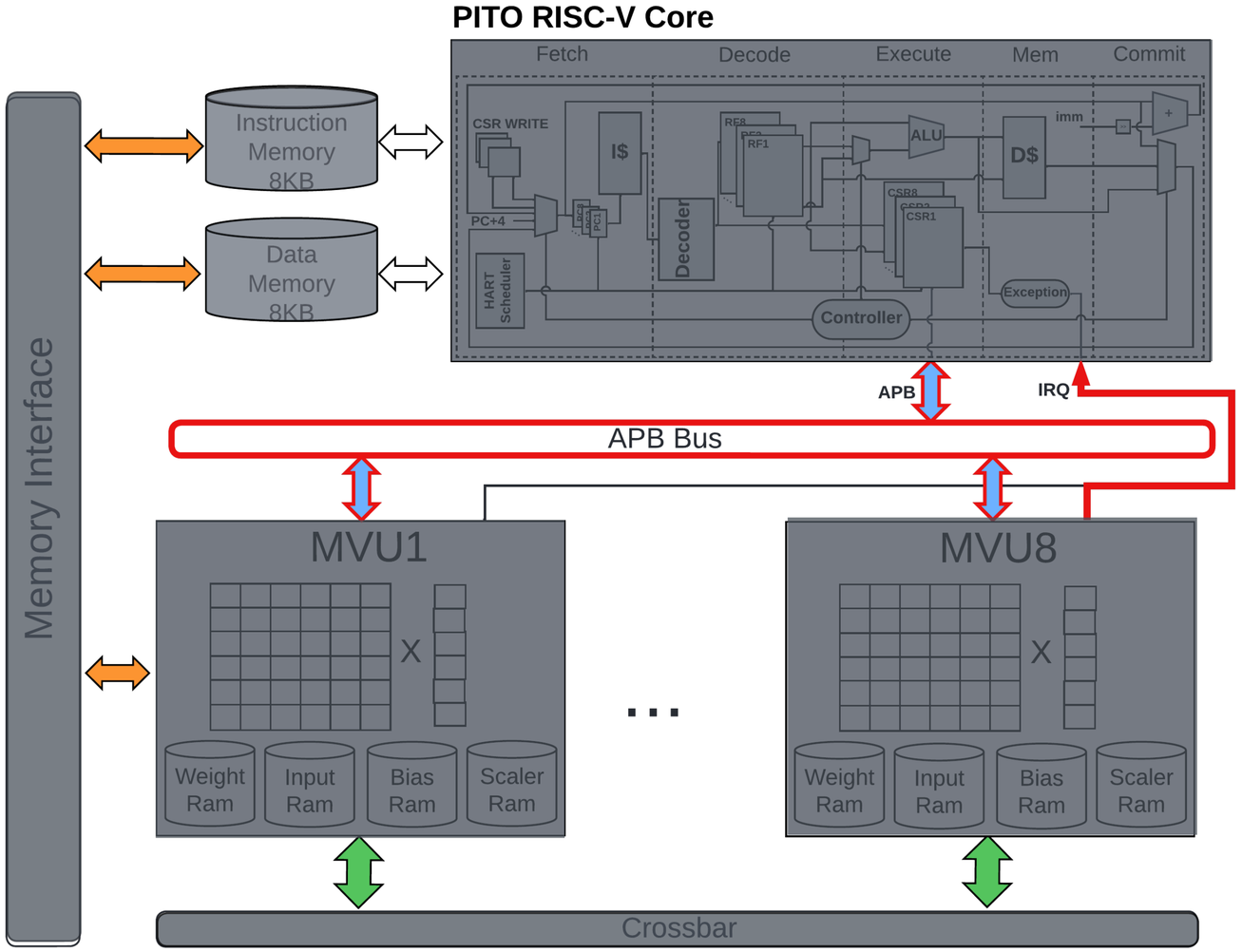
Kick start the accelerator

```
#*****************************************************************************
# matmul.S
# Kernel code for Matrix Multiply
#-----------------------------------------------------------------------------
#include "pito_def.h"

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 mvu_quant     , 10    // set quant_msbidx to 10
csrwi mvu_wbaseaddr , 0     // set weight address to 0
csrwi mvu_ibaseaddr , 0     // set input address to 0
addi  x1, x0, 1
slli  x1, x1, 10            // set output address to 0x400
csrw mvu_obaseaddr , x1
csrwi mvu_wstride_0 , 30    // 1 tile back move x 2 bits
csrwi mvu_wstride_1 ,  2    // 1 tile ahead move x 2 bits
csrwi mvu_wstride_2 ,  0
csrwi mvu_wstride_3 ,  0
csrwi mvu_istride_0 , 30    // 1 tile back move x 2 bits
csrwi mvu_istride_1 ,  0
csrwi mvu_istride_2 ,  0
csrwi mvu_istride_3 , 30
csrwi mvu_ostride_0 ,  0
csrwi mvu_ostride_1 ,  0
csrwi mvu_ostride_2 ,  0
csrwi mvu_ostride_3 ,  0
csrwi mvu_wlength_0 ,  1    // 2 tiles in width
csrwi mvu_wlength_1 ,  3    // number bit combinations i.e. 2x2 bits
csrwi mvu_wlength_2 ,  1    // 2 tiles in height
csrwi mvu_wlength_3 ,  0
csrwi mvu_ilength_0 ,  1    // 2 tiles in height
csrwi mvu_ilength_1 ,  0    // number bit combinations
csrwi mvu_ilength_2 ,  0    // 2 tiles in width of matrix operand
csrwi mvu_ilength_3 ,  0
csrwi mvu_olength_0 ,  1
csrwi mvu_olength_1 ,  ś
csrwi mvu_olength_2 ,  0
csrwi mvu_olength_3 ,  0
addi x1, x0, 1
slli x1, x1, 30             // mul mode 01
addi x1, x1, 16
csrw mvu_command, x1        // Kick start MVU, 2 tiles x 2 tiles x 2bit x 2bits
ebreak
```

3. Use case: Matrix Multiply

- The configurations are written to each MVU through APB bus.
- Once the MVU is done computing, it will send and interrupt to the corresponding HART.



4. Experiments and Results

- We synthesized BARVINN for ALVEO U250 FPGA from AMD.
- We used Vivado 2021.4 for synthesis.
- We used 8 MVUs with 64KB weight and Data RAMs and 2KB Scaler and 4KB Bias Rams.
- For PITO, we used 8KB instruction and data caches.

Resource	Pito RISC-V	MVU Array	Overall
LUT	10454	190625	201079
BRAM	15	1312	1327
DSP	0	512	512
Dynamic Power	0.410 W	21.066 W	21.504 W
Frequency	250 MHz	250 MHz	250 MHz

4. Experiments and Results

- We compared our platform against FINN and FILM-QNN.
- We used models provided by FINN repository.

4. Experiments and Results

- We compared our platform against FINN.
- We used models provided by FINN repository.
- For CNV model on CIFAR10, over different bit precisions:
  - We achieve better FPS/kLUTs
  - FINN uses less LUTs.

	Bits (W/A)	kLUT	BRAM	DSP	FPS	FPS/ kLUT
Ours	1/1	201.1 (15.0%)	1327	512	61035	303.5
	1/2	201.1 (15.0%)	1327	512	30517	151.7
	2/2	201.1 (15.0%)	1327	512	15258	75.8
FINN	1/1	28.2 (2.1%)	150	0	7716	273.6
	1/2	19.8(1.47%)	103	0	2170	109.6
	2/2	24.3(1.81%)	202	0	2170	89.3

4. Experiments and Results

- We compared our platform against FINN.
- We used models provided by FINN repository.
- For CNV model on CIFAR10, over different bit precisions, we achieve better FPS/kLUT.
- For Resnet50 model, we achieve better FPS/Watt compared to FINN and FILM-QNN.

	Bits (W/A)	Clock Freq.	FPS	FPS/Watt
Ours	1/2	250 MHz	2296	106.8
FINN-R [1][6]	1/2	178 MHz	2873	41.0
FILM-QNN [20]	4(8)/5	150 MHz	109	8.4

BARVINN is open source!

Documentation:

https://barvinn.readthedocs.io/en/latest/

Source Code:

https://github.com/hossein1387/BARVINN

5. Conclusion and Future Work

- In this work, we presented BARVINN, an arbitrary precision DNN accelerator controlled by a RISC-V processor.
- We presented the architecture of different components.
- We presented synthesis results.
- We used BARVINN to run inference on different models with different bit precisions.
- We are preparing BARVINN for ASIC implementation (GF 22 or GF12nm)
- We are planning to use TVM to improve code generation and optimization.

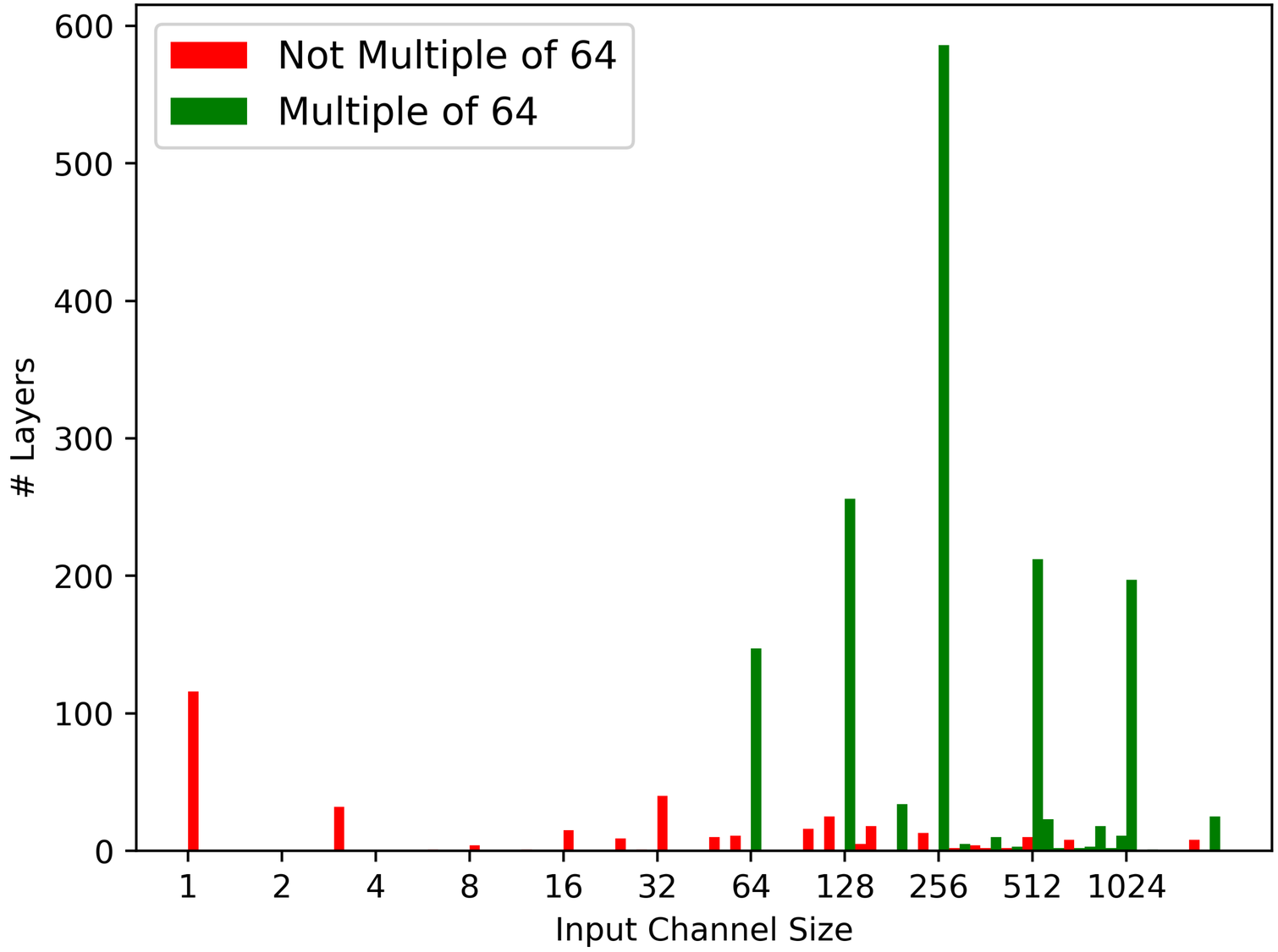
5. Acknowledgement:

- This project is made possible with the help and support from:
  - CMC Microsystems
  - IBM
  - MILA
  - Mitacs
  - FRQNT

Thank You!

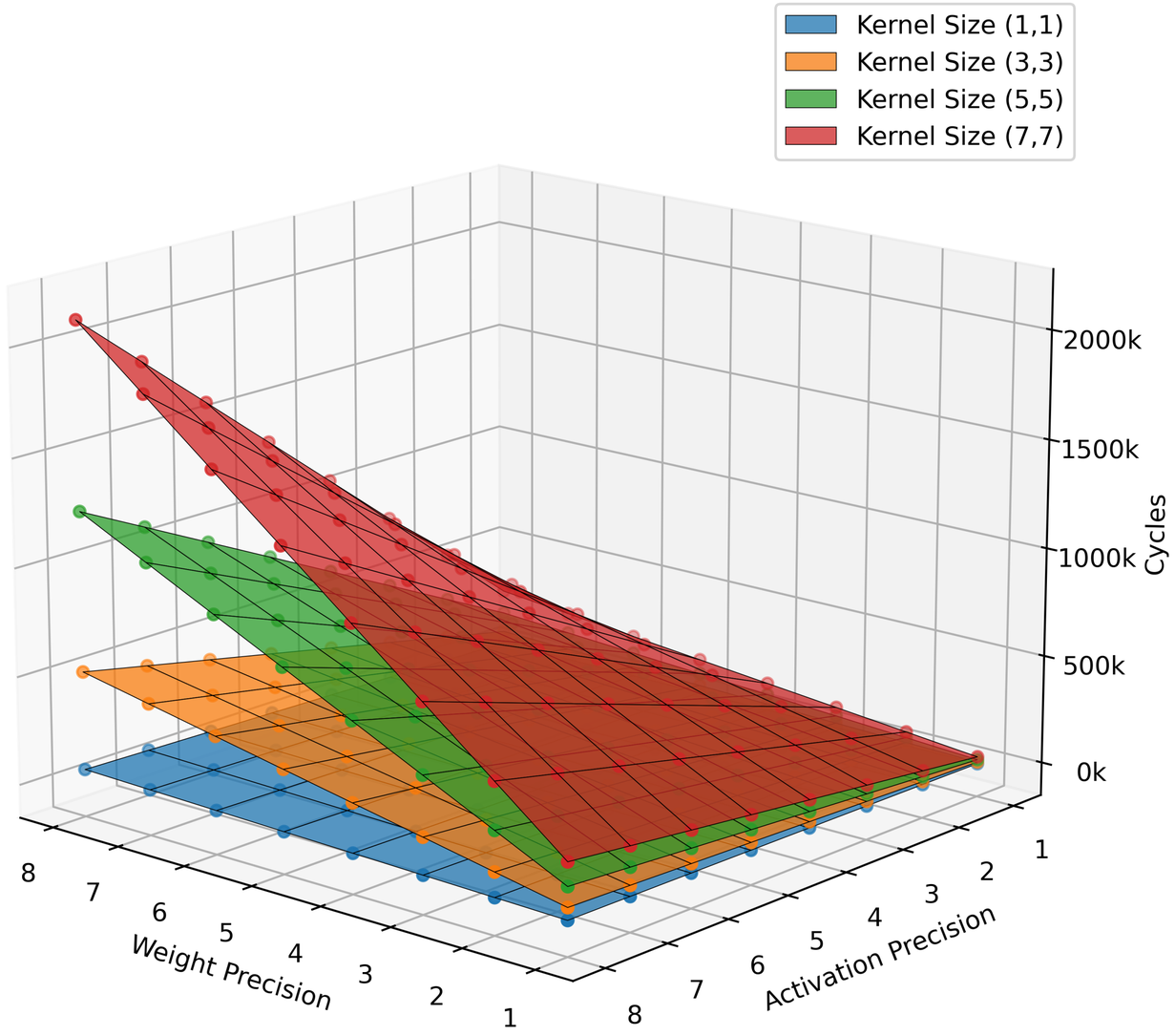
A. Auxiliary Slides:

- We analyzed over classification 50 models from ONNX model zoo.
- Around 79% of these models use convolution with input channel sizes that are multiples of 64.



A. Auxiliary Slides:

- Computation complexity diagram.



A. Auxiliary Slides:

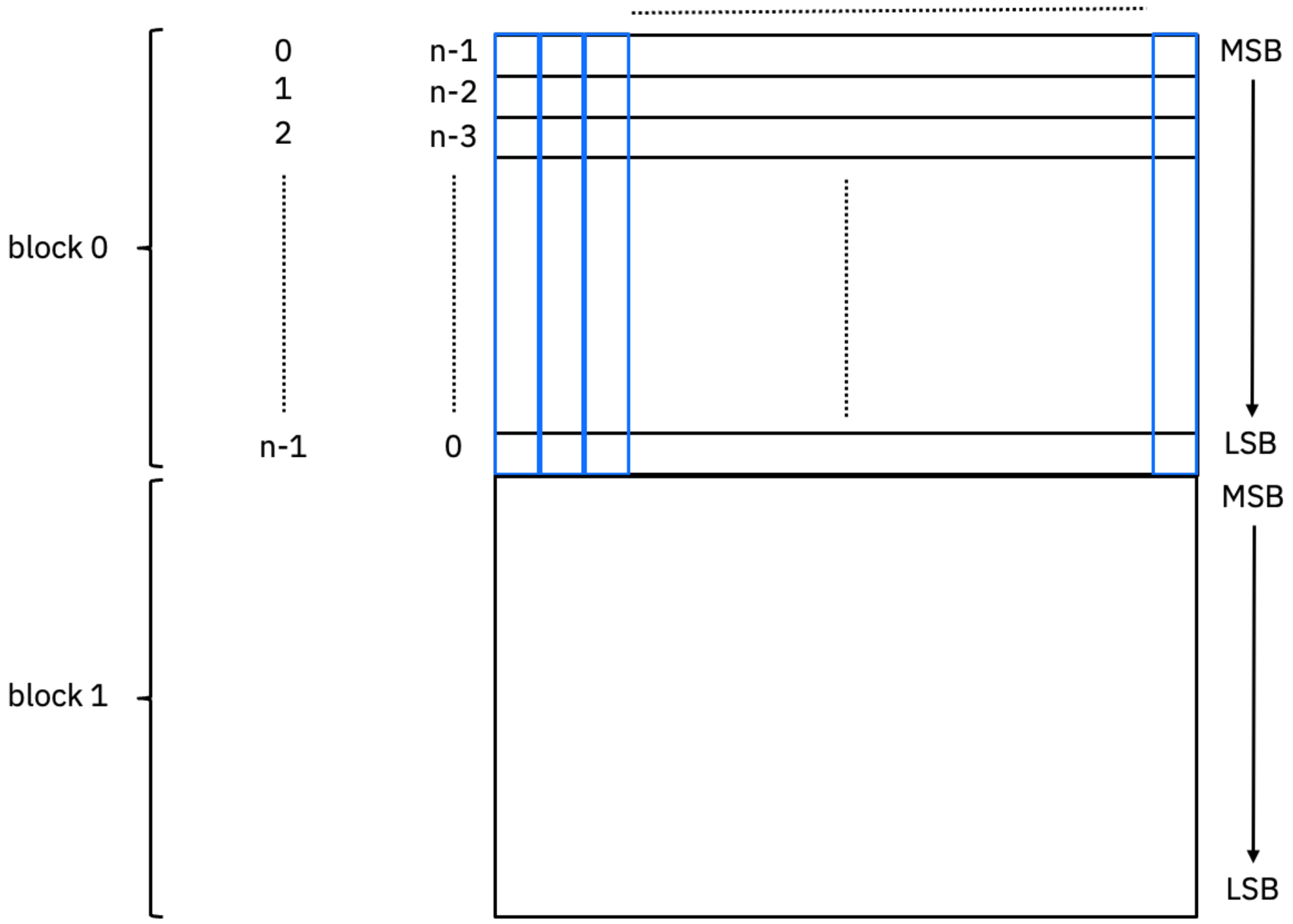
- MVU Data Storage Format

  Blocks of 64 n-bit numbers

  Organized in bit-sliced order

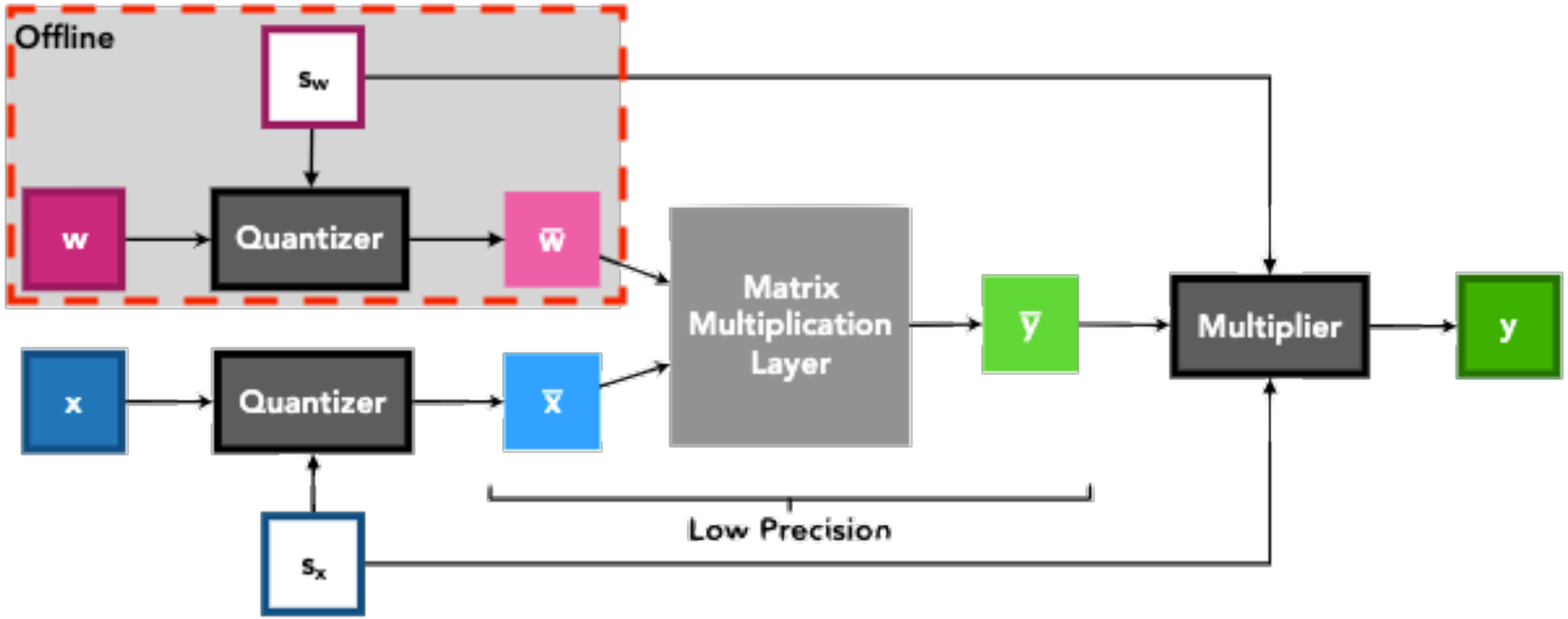
  Each value is a “column”

  Each row are bits from each value with same bit position



A. Auxiliary Slides:

- Low precision computation pipeline.



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