Cairo University
Faculty of Engineering
Electronics & Electrical Communication
Engineering Department

Convolutional Neural Network Accelerated Coprocessor Based on RISC-V Instruction Set

Graduation Project Report

Submitted in partial fulfillment of the requirements of the
degree of Bachelor of Science in Electronics & Electrical Communication Engineering

Submitted by

Ahmed Adel Emara
Omar Mohamed Abdulsalam
Mohamed Samir Monir
Heba Mahmoud Ahmed
Yasmin Behairy Abdelsalam
Yomna Gamal Fawzy

Supervised by

Prof.Dr. Hassan Mostafa
Technical support: Eng. Ahmed Allam

Sponsored by
ICpedia and ONELAB

Acknowledgments

First and foremost, praise is to Allah, the Almighty, the greatest of all, on whom ultimately, we depend for sustenance and guidance. We would like to thank Almighty Allah for giving us opportunity, determination and strength to achieve this work. His continuous grace and mercy were with us throughout our life and even more during the tenure of this work.

Second, we want to thank our families for their support, tolerance and love during this year especially during the hard times they were always there having faith in what we do. We are grateful to our families, colleagues and friends for always motivating us, without them we wouldn't have come so far. We want to thank our major advisor Dr. Hassan Mostafa and Eng. Ahmed Allam for their caring about following up each stage in the project and his suggestions to solve some problems we faced during the project work.

And a special thanks to Eng. Ahmed Allam, it has been a great pleasure and honor being our supervisor. You were continuously encouraging us, even before we decided to work on this project.

Finally, though only our names appear on the cover of this thesis, but many have - knowingly or unknowingly – contributed to its production, and for that we are extremely thankful,

Ahmed, Mohamed, Omar, Heba, Yasmin and Yomna

Abstract

As a typical artificial intelligence algorithm, the convolutional neural network (CNN) is widely used in the Internet of Things (IoT) system. In order to improve the computing ability of an IoT CPU, we design a reconfigurable CNN-accelerated coprocessor based on the RISC-V instruction set. The interconnection structure of the acceleration chain designed by the predecessors is optimized, and the accelerator is connected to the RISC-V CPU core in the form of a coprocessor. the coprocessor instructions are called, coprocessor acceleration library functions are established, and common algorithms in the IoT system are implemented on the coprocessor.

Increased design complexity has resulted in the need for efficient verification. The verification process is crucial for discovering and fixing bugs prior to fabrication and system integration. However, as designs increase in complexity, the use of traditional verification techniques with VHDL and Verilog may fall short to provide a proper toolset.

This thesis explores the use of the universal verification methodology (UVM) to verify the Processing element component like convolution, Max Pool, RELU and ADD using complete environment for the chain.

To verify the operation of the custom co-processor, a RISC-V core and interface is needed. Several opensource options for the main CPU core are available, from which the CVA6 core developed by PULP platform was deemed as the most appropriate for our project.

An interface is needed to integrate the core and the custom co-processor. An interface named Core-V X-Interface developed by PULP platform that serves as a generic interconnection between PULP cores and custom accelerators was used. Deep modifications in the core pipeline are necessary to inform the core of the existence of the custom co-processor.

List of Contents

List of Contents 3
List of Figures 5
List of Tables 7
List of Abbreviation 8
Chapter 1: Introduction....9
Chapter 2: Coprocessor 11
2.1. Coprocessor design.... 11
2.2. The Processing Element 13
2.2.1. Convolution Unit.... 13
2.2.2. Pool Unit 24
2.2.3. Add Unit.... 27
2.2.4. Relu Unit 28
2.2.4. Crossbar Unit 30
2.3. The Memory File.... 32
2.3.1. Memory File.... 32
2.3.2. RAM Design 33
2.3.3. Operations Logic.... 35
2.4. Custom Instructions.... 41
2.5. Reconfigurable Control Unit 48
2.5.1. Control Unit Design 48
2.6. Generalization of Idea 53
Chapter 3: Software path.... 54
3.1. Machine Learning vs. Deep Learning: 54
3.2. Different types of Neural Networks in Deep Learning 55
3.2.1. Artificial Neural Network (ANN): 55
3.2.2. Recurrent Neural Network (RNN):.... 57

3.2.3. Convolution Neural Network (CNN): ... 59
3.3. Most popular Convolutional Neural Network architectures (CNN): ... 60
3.3.1. LeNet-5 [17] ... 61
3.4. General layers of Convolutional Neural Network architectures: ... 67
3.4.1. Convolution ... 67
3.4.2. Max pooling ... 68
3.4.3. Average pooling ... 69
3.4.4. Activation functions: ... 70
3.5 Software Algorithm ... 75
Chapter 4: Verification ... 77
UVM overview ... 80
Chapter 5: RISCV ... 103
5.1. CVA6 (Ariane): ... 106
5.1.1. Architecture: ... 106
5.1.2. PC Gen stage: ... 107
5.1.3. Instruction Fetch stage: ... 108
5.1.4. Instruction decode stage: ... 109
5.1.5. Instruction issue: ... 111
5.1.6. Get ready to work with CVA6: ... 112
5.2. Interface: ... 123
APPENDIX ... 130
Conclusion ... 136
Future Work ... 137
References ... 139

List of Figures

Figure 1 : coprocessor design architecture 11
Figure 2: Convolution of 6x6 matrix with 3x3 kernel 13
Figure 3: Data has to be updated each transition 13
Figure 4: Convolution Datapath Block Diagram 14
Figure 5: Detailed Design of the Convolution Datapath 15
Figure 6: Convolution Interface with Reconfigurable Controller 17
Figure 7: ROM Control Circuit 18
Figure 8: IF buffer Load signals generator 19
Figure 9: Control sub circuit for separated ROM design 20
Figure 10: 4 inputs 4 outputs ROM 22
Figure 11: Multiplier with Enable 23
Figure 12: Multiply enable and local reset generator 23
Figure 13: Zero Detector Design 24
Figure 14 Example of Pooling circuit of 4x4 matrix with 2x2 filter size 24
Figure 15: Max Pool design architecture 25
Figure 16: input serial elements for op 1 & op 2 26
Figure 17: Add unit design architecture 27
Figure 18: Relu unit design archiecture 29
Figure 19: Crossbar design architecture 30
Figure 20: Memory File Design 33
Figure 21: RAM Design 33
Figure 22: RAM Design with Arbiter 35
Figure 23: Operation 1 Logic 36
Figure 24: Operation 3 Logic 37
Figure 25: Convolution result matrix 37
Figure 26: Operation 2 logic 38
Figure 27: Operation 4 Logic 39
Figure 28: Arbiter Opcode selection 40
Figure 29: Controller Finite State Machine 50
Figure 30: Execution of Combo1 51
Figure 31: Executing Store instruction 52

Figure 32: Area vs performance....53
Figure 33: Comparison between Machine Learning & Deep Learning....55
Figure 34: ANN....55
Figure 35: ANN Image classification....56
Figure 36: Backward Propagation....56
Figure 37: Difference between an RNN and an ANN....58
Figure 38: Many2Many Seq2Seq model....59
Figure 39: Detect patterns edges, corners, objects....59
Figure 40: CNN....60
Figure 41: LeNet-5....61
Figure 42: first layer convolution layer(C1)....61
Figure 43: second layer pooling layer (S2)....62
Figure 44: third layer convolution layer (C3)....63
Figure 45: fourth layer pooling layer (S4)....63
Figure 46: fully connected layer (F5)....64
Figure 47: fully connected layer (F6)....65
Figure 48: output layer....66
Figure 49: LeNet-5 table....66
Figure 50: 2D convolution example....67
Figure 51: Max pool sample2....68
Figure 52: Max pool real life example....69
Figure 53: average pooling example....70
Figure 54: RelU in process....71
Figure 55: ReLU Activation function....71
Figure 56: sigmoid function....73
Figure 57: Softmax function....73
Figure 58: Tanh vs. sigmoid function....74
Figure 59: Divide source matrix when its size is multiple from co-size....75
Figure 60: Divide src matrix when its size not multiple from co-size....76
Figure 61: comparison between verilog and systemverilog as test bench language....78
Figure 62: Typical UVM testbench....80
Figure 63: UVM classes , connections between testbench and DUT....83
Figure 64: Transaction level modeling VS Analysis port export....84
Figure 65: DUT connected to test-bench through irtual interface....85
Figure 66: UVM phases....88
Figure 67: UVM Tansactions approach envirnoment connectionss....90
Figure 68: Design of test bench module....91

Figure 69: Snippets from class of command transaction 91
Figure 70: do_copy function 92
Figure 71: cone_me function 92
Figure 72: do_compare function 93
Figure 73: conv2string sunction..93
Figure 74: Snippet from driver class 94
Figure 75: Snippet from driver class "run_phase" 95
Figure 76: snippet from command monitor 95
Figure 77: snippet from command monitor class 96
Figure 78: result_monitor class...96
Figure 79: Snippet from scoreboard class 97
Figure 80: Snippet from scoreboard class "write" function 98
Figure 81: Top module code 98
Figure 82: env class code 99
Figure 83: snippet from interface class 100
Figure 84: snippet from interface class "reset_conv" function 100
Figure 85: Snippet from interface class 101
Figure 86: behavior of test bench module...102
Figure 87 Area vs performance...137

List of Tables

Table 1 Coprocessor performance summary. 12
Table 2 the possible Combos can express any CNN architectures....................................43
Table 3 the custom instructions with Opcodes. 44

List of Abbreviation

AI	artificial intelligence
AIOT	Artificial Intelligence of Things
IOT	Internet of Things
GPU	graphics processing unit
TPU	Tensor Processing Unit
CNN	Convolution neural network
SCNN	sparse Convolution neural network
ANN	Artificial Neural Network
RNN	Recurrent Neural Network
FFT	Fast Fourier transform
MAC	Multiply and accumulate
Relu	rectified linear unit
RISC-V	reduced instruction set computer
SOC	System on chip
CPU	Central processing unit
FPGA	field programmable gate array
KW	Kernel weights
IF	Input file
PE	Processing element
1D	One dimensional
2D	Two dimensional
LSB	Least significant bit
MSB	Most significant bit
LUT	Look up table
CI	Convolution layer
UVM	Universal Verification Methodology
TLM	Transaction Level Modeling

Chapter 1: Introduction

With the rapid development of artificial intelligence (AI) technology, more and more AI applications are beginning to be developed and deployed on internet of things (IoT) node devices. The intelligent internet of things (AI + IoT, AIoT), which integrates the advantages of AI and IoT technology, has gradually become a research hotspot in IoT-related fields. Traditional cloud computing is not suitable for AIoT, because of its high delay and poor mobility. For this reason, a new computing paradigm, edge computing, is proposed. In order to reduce the computing delay and network congestion, the computing is migrated from the cloud server to the device. Edge computing brings innovation to the IoT system, but also challenges the AI computing performance of the IoT node processor. It is necessary to improve its AI computing performance under the condition of meeting the power consumption and area limitation of IoT node devices. In order to improve the AI computing power of IoT node processors, some IoT chip manufacturers provide some artificial intelligence acceleration libraries for their IoT node processors, but only from the software level optimization and tailoring algorithm, just a stopgap. It is necessary to design the AI algorithm calculator suitable for the IoT node processor from the hardware level.

Among all kinds of AI algorithms, the convolutional neural network (CNN) algorithm is widely used in various IoT systems with image scenes due to its excellent performance in image recognition. Compared to traditional signal processing algorithms, it has a higher recognition accuracy, can avoid complex and tedious data feature extraction, and has stronger adaptability to different image scenes. The common hardware of CNN acceleration is a GPU or TPU, but this kind of hardware accelerator is mainly used for high-performance servers, which are not suitable for the use of IoT node devices. In Reference [2, 1], most of the constituent units in the CNN network are implemented, which need to consume more hardware resources while obtaining higher computing acceleration performance, and cannot meet the resource-limited needs of nodes in the IoT systems. In Reference [3], the matrix multiplication in the CNN algorithm is reduced by the FFT, but the FFT itself consumes more hardware resources. The sparse CNN (SCNN) accelerator proposed in Reference [4] uses the data sparsity of neural network pruning operation to design a special MAC computing unit, but it has a highly customized structure and a single application scenario. In Reference [5], an acceleration structure based on in-memory computing is proposed. By shortening the distance between computing and storage, the access to memory is reduced and the

computing efficiency is improved. However, the cost of the chip based on in-memory computing is high, so it is not suitable for large-scale application in the IoT systems. The work in Reference [6] proposes several alternative reconfiguration schemes that significantly reduce the complexity of sum-of-products operations but do not explicitly propose a CNN architecture. In Reference [7], an FPGA implementation of a CNN for addressing portability and power efficiency is designed, but the proposed architecture is not reconfigurable. In Reference [8], a CNN accelerator on the Xilinx ZYNQ 7100 (Xilinx, San Jose, CA, USA) hardware platform that accelerates both standard convolution and depth wise separable convolution is implemented, but the designed accelerator cannot be configured for other algorithms in the IoT system. Several researches work [9, 10] use the RISC-V ecosystem to design an accelerator-centric SoC or multicore processor, but do not configure the design accelerator in the form of a coprocessor and design the corresponding custom coprocessor instructions to speed up the algorithm processing speed.

Based on Reference [11], this document further optimizes the structure of the CNN accelerator. The basic operation modules in the acceleration chain designed by Reference [11] are interconnected through a crossbar, which makes the flow direction of input data more diversified, thus enriching the function of the acceleration unit and forming a reconfigurable CNN accelerator.

Chapter 2: Coprocessor

2.1. Coprocessor design

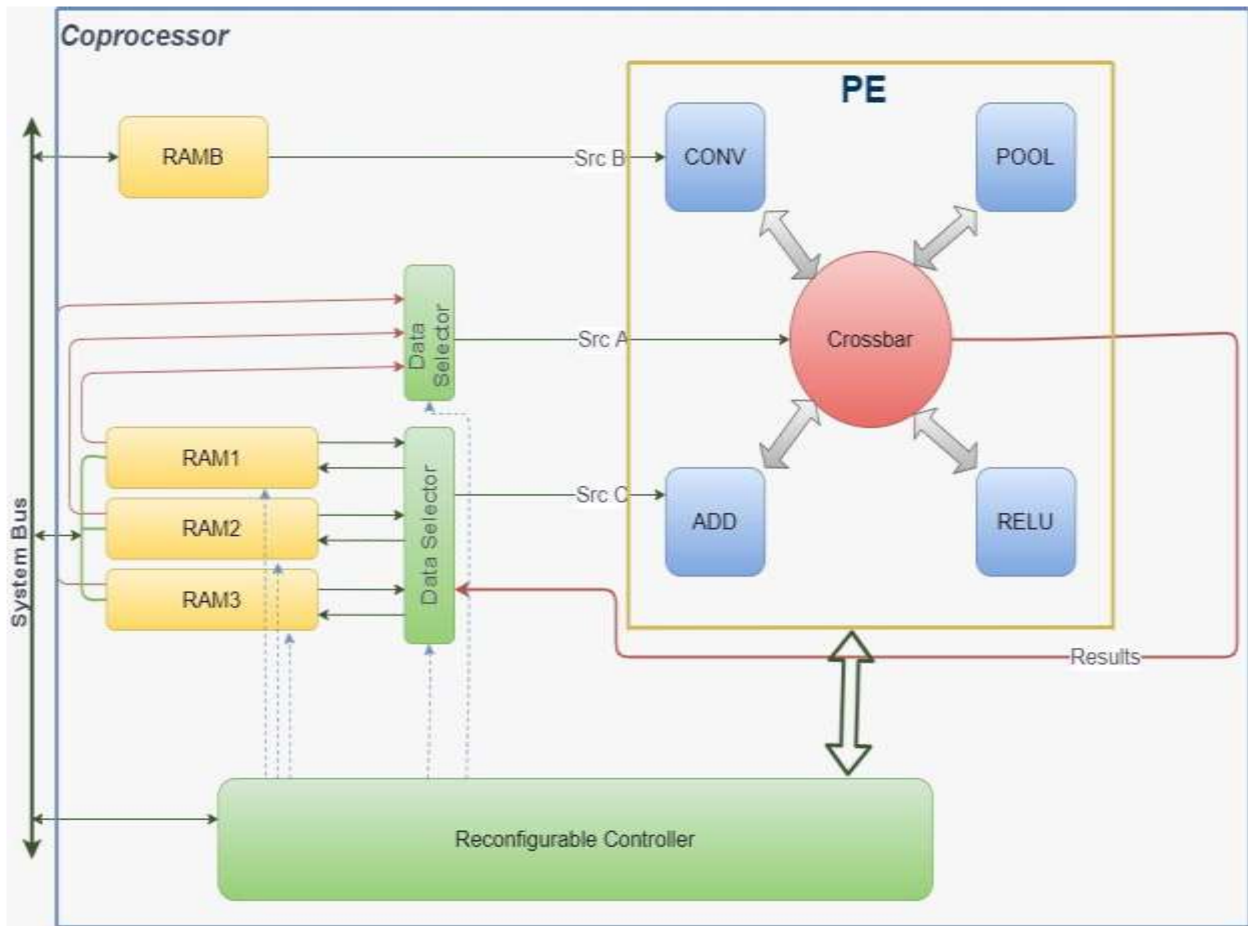


Figure 1 : coprocessor design architecture

How the Coprocessor works with CPU?
Coprocessor function is to accelerate the critical operations by HW. There are some other operations executed in CPU by SW in parallel.

- **Component**

The coprocessor design as shown in Figure 1 : coprocessor design architecture mainly consists of three building blocks which are the processing element PE, Reconfigurable controller and the Memory File.

- The **Processing Element PE** is the main block of coprocessor as it implements the algorithm and takes the most computing power, as shown there are four basic operation modules, convolution, pooling, ReLU, and matrix plus, are interconnected by a crossbar.

- While RAM1, RAM2, RAM3, RAMB and Data Selector constitute the **Memory File** which is considered as the interface between main memory and processing element and used to load or store processing element by data from path interface. The data stored or loaded inside PEs may be **Src_A** which contains the image matrix, **Src_B** which contains the kernel matrix and **Src_C** which contain Bias matrix.

- The **Reconfigurable controller** is used to send all the control signals needed by the PE and memory file for appropriate operation.

Table 1 Coprocessor performance summary.

Parameter	Description
precision	10-bit fixed-point
Feature map size	(1~256)* (1~256)
Kernel size	(1~8)* (1~8)
Pool type	Max pool
Pool size	(1~8)* (1~8)
Function (arbitrary combination)	2D Convolution Data add (matrix or bias) Pooling ReLU activation

- Both Feature map matrix and Kernel matrix designed to be square matrices

2.2. The Processing Element

2.2.1. Convolution Unit

2.2.1.1. Convolution Procedure

CNN algorithm basically depends on 2-D Convolution as it’s the basic block needed to be accelerated and optimized as according to [1], 2-D convolution consumes more than 90% of the total computational time. Thereby, 2-D convolution is always the focus of many CNN accelerators’ optimization. Furthermore, convolution operation contains a large amount of data loading-storage, multiplications and additions, but in fact, the data, which has to be stored and loaded to be processed, has a lot of repetition [2], for example if a matrix 6x6 is convolved with kernel 3x3 as shown in Figure 2 and Figure 3:

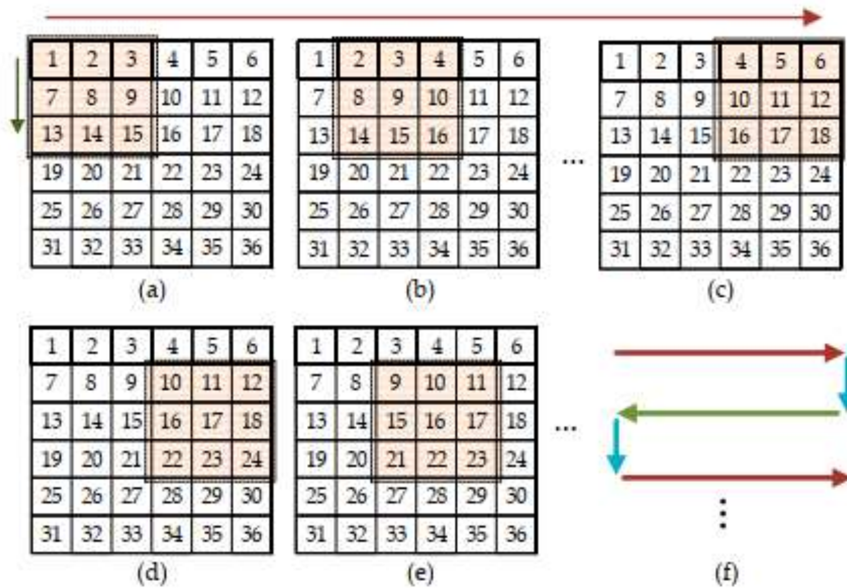


Figure 2: Convolution of 6x6 matrix with 3x3 kernel

1	7	13	2	8	14	3	9	15
4	10	16	2	8	14	3	9	15
4	10	16	5	11	17	3	9	15
4	10	16	5	11	17	6	12	18
24	10	16	22	11	17	23	12	18
21	10	16	23	11	17	24	9	15

Figure 3: Data has to be updated each transition

The valuable note here is that at the transition from submatrix to another one, there are common elements between them, and only the updated elements are 3 elements which is the value of kernel dimension and the rest of elements remains as it is, the idea comes here, as instead of loading to convolution each time the whole submatrix, it's sufficient to load only the updated elements and depend on the memory of the convolution unit itself to reuse the previously updated elements and hence the data transfer bandwidth between convolution unit and serving memory is reduced and get higher performance.

2.2.1.2. Convolution Datapath

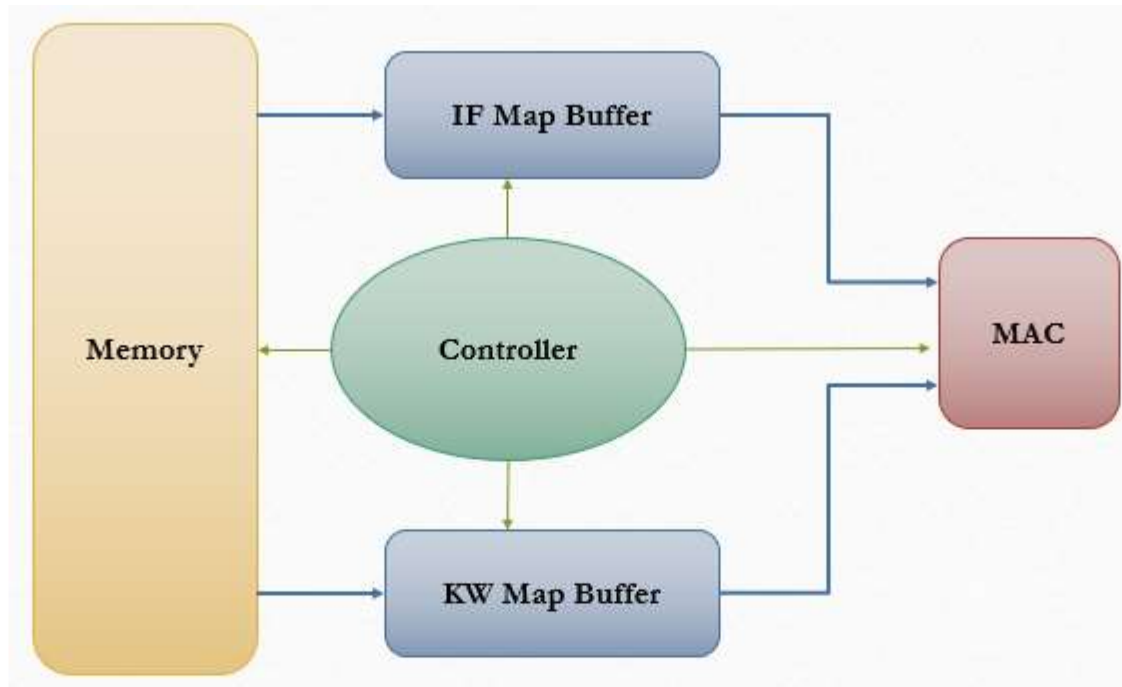


Figure 4: Convolution Datapath Block Diagram

As shown in Figure 4, the convolution data-path contains significantly Buffers to remain the elements to be reused while transitions occur, and Memory feeds with the updated elements every time, there are two buffers, one for Input Features (IF Map Buffer) and another for Kernel weights (KW Map Buffer), and then each element in IF buffer will be multiplied and accumulate to the corresponding one in KW buffer and hence get the final result element of each transition.

The controller has to determine the Memory location of the updated elements and also select the positions which these elements are updated in the buffers and finally destinates the MAC to operate on the elements processed which is the challenge in that procedure as will be discussed later on.

And as in Figure 5, the detailed design of the convolution data-path

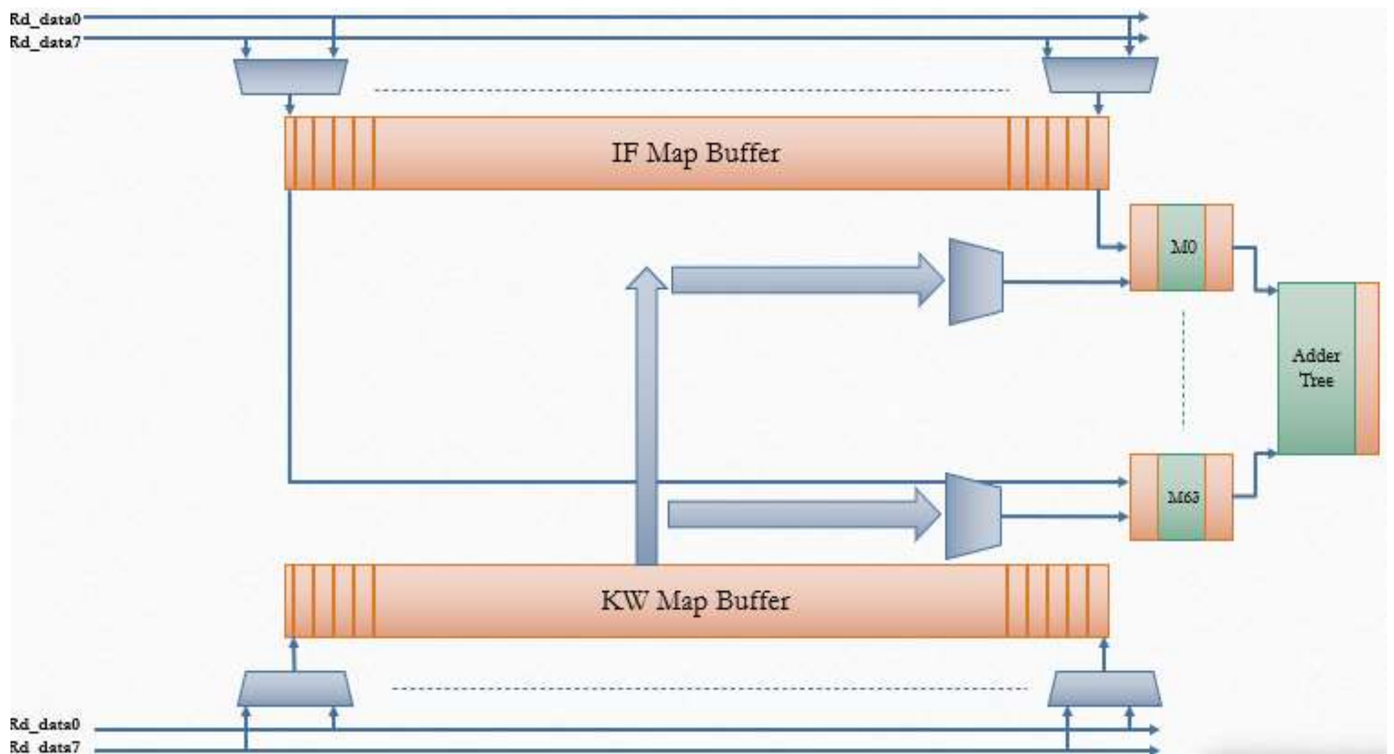


Figure 5: Detailed Design of the Convolution Datapath

That data-path is designed for worst case which occurs when kernel size = 8, and hence the submatrix elements are 64 one and therefore the number of registers are 64 register in the IF map buffer and KW as well, and the updated elements between transitions equal to kernel size which is 8 read data buses, these buffers naturally operates for any other kernel size below 8, as the number of used registers are square number of kernel, the input multiplexers to the buffers are used to route read data buses to select which line is fed to which registers and dramatically load signals of these registers has to be synchronized to enable the input lines get to be stored in registers, at the beginning of operation when the registers have no data, the first submatrix has to be fed on maximum 8 cycles, and that for all to unite the setup time for all kernel sizes, on the other hand the KW map buffer is fed only one time to have the kernel used for that convolution.

As the submatrix elements stored in IF map buffer has changeable inconstant positions each transition as it’s shown in the procedure, and as the kernel stored in KW map buffer is fed one time and not change, then the KW map buffer output multiplexers are needed to route the kernel elements to the corresponding correct elements in IF map buffer, and finally they are concatenated and fed to 64 multipliers which is chosen for

best performance at worst case, and then all multipliers outputs are fed to add tree which contains 5 levels of addition, 1st has 32 adders works in parallel and accept 64 input to get 32 output which is fed to 2nd one which has 16 adder and then to 3rd which has 8 adders and then 4th which has 4 adders and then the 5th has 2 adders and finally the 6th one is registered output adder to get the final result of the transition, and so on each transition is updated by elements and go through the convolution data-path and so on.

The overflow occurrence possibility is managed by determining the overflow in all multipliers and adders used in MAC and if the overflow occurs in any module, then the convolution overflow is raised high.

The convolution Datapath is pipelined as shown above into 4 stages pipelining, 1st one for updating new elements and 2nd for getting ready to be multiplied, 3rd to be multiplied and 4th for accumulation, and as the convolution consumes 8 cycles at the beginning for the setup, then the total setup time for the convolution to get the first element out is after 11 clock cycles, and consecutively it outs one element each cycle.

Convolution throughput can be determined as follows:

$$Setup\ time = 11\ cycles$$

$$number\ of\ elements = (N - F + 1)^2$$

$$: N = matrix\ size, F = kernel\ size$$

$$\therefore Total\ Clock\ cycles = (N - F + 1)^2 + 11$$

$$\boldsymbol{Throughput = \frac{Total\ clock\ cycles}{(N - F + 1)^2} \approx 1\ element/cycle}$$

And that under the assumption of $(N - F + 1)^2 \gg 11$

Which is usually valid assumption as $N = 1\sim256$ and $F = 1\sim8$, although the input matrix has small size and convolved with high size kernel which is rare scenario to occur.

2.2.1.3. Convolution Controller

It’s very important to determine the control and status signals of data-path to be evaluated by the controller:

Control Signals:

1. IF and KW map buffers Input multiplexers selection lines which are 3-bit for each 8x1 multiplexer and their number is 64 multiplexers for IF buffer and same for KW buffer.
2. Load signals of registers in IF and KW map buffers which is 64-bit for each.
3. KW map buffers output multiplexers selection lines which are 6-bit for each 64x1 multiplexer and their number is 64 multiplexers as well.
4. Multipliers enables as most of cases, the whole multipliers are not fully used when the kernel size is not 8 and also if one of the inputs is zero.
5. Local Resets to all registers in the data-path as the unused registers has to be reset.

Status Signal is only the overflow, as when it occurs, controller has to reset operation.

Hence, the interface between the convolution controller and data-path is as in Figure 6:

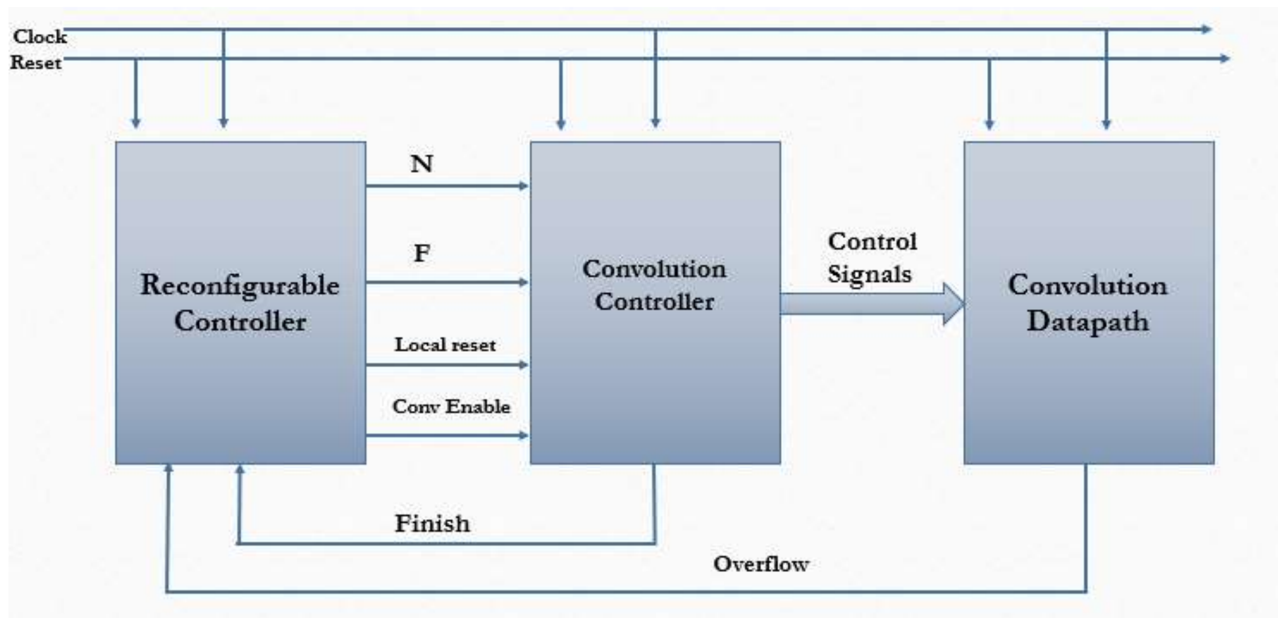


Figure 6: Convolution Interface with Reconfigurable Controller

The reconfigurable controller feeds the convolution controller with the matrix and kernel sizes to be convolved with convolution operation enable raised high for one cycle at the start of convolution, and also local reset to reset all internal registers before the operation and the convolution controller has to count on until the whole input matrix is convolved totally with the kernel and raise the finish signal high to inform that the operation is done.

Due to the multitude of the cases that input matrix and kernel can have as N = 1~256 and F = 1~8 then the number of possibilities is evaluated to 2048 cases minus the forbidden cases which happens if the kernel size is greater than the input matrix size and finally is evaluated to 2020 cases, and as each input matrix with each kernel have their specific sequence of control signals as each line in the sequence is corresponding to a transition occurs and when the sequence is ended up the whole transitions through the whole matrix is done, the LUTs or ROMs are used here, as these cases are modelled by MATLAB and get the sequence of control signals specially the routing multiplexers and load signals of IF and KW map buffers.

The controller takes the input matrix size and kernel size and go through the sequence appropriate for that case and raise the finish signal high when the chosen sequence is ended up, that is managed by sub-circuit that is attached to the ROM and its final target is to get the appropriate address that has the control word suitable for the transition in the chosen case.

ROMs have the codewords of the sequences in order, for example IF input multiplexers selection lines ROM is ordered from F = 1 and N = 1, passing F = 1 and N = 256 through F = 8 and N = 256.

The following Figure 7 is design of the ROM sub-circuit for example if that ROM holds the sequences of IF map input loads or input multiplexers control signals:

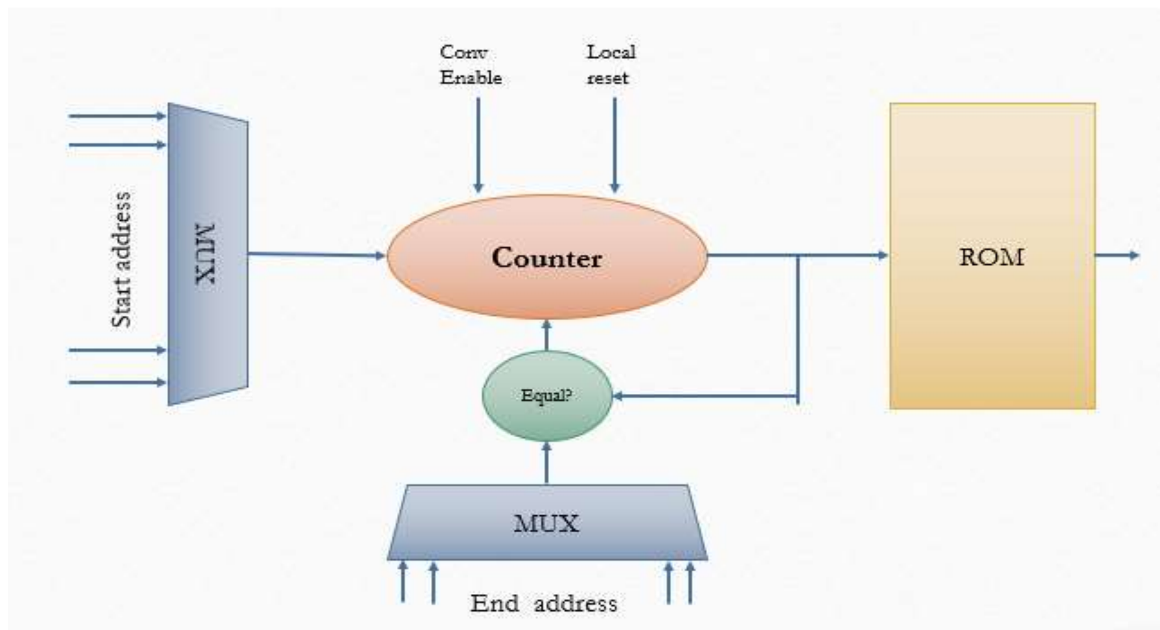


Figure 7: ROM Control Circuit

In the above design, the multiplexer which determines Start address of the sequence based on case of matrix and kernel sizes, and as convolution enables comes, it enable the counter to load start address and that address comes out as address to ROM to point on the first control word in the sequence and then counts on and go through the sequence with each clock cycle until it reach to end address which is also determined by the matrix and filter size case, and compare them, if they are equal, the comparator raises the finish signal high and disables the counter to point to the last address until controller reset it at the end of operation.

A valuable note here is that when the kernel size is below 8, most of the codeword is previously known, for example, if kernel size is 3, that means that the 9 least significant registers are used and the rest is zeros and that means that load enable of idle registers is zeros all the time and their input multiplexers selection lines are "don't cares", same for KW output multiplexers selection lines which are also don't cares for the corresponding idle registers, that for KW map buffer also, and hence the number of used registers is square of kernel size only and that note can be used to reduce the size of ROMs and the it has to separate the ROMs as each one can store the sequence of the cases related to each kernel size as shown in Figure 8.

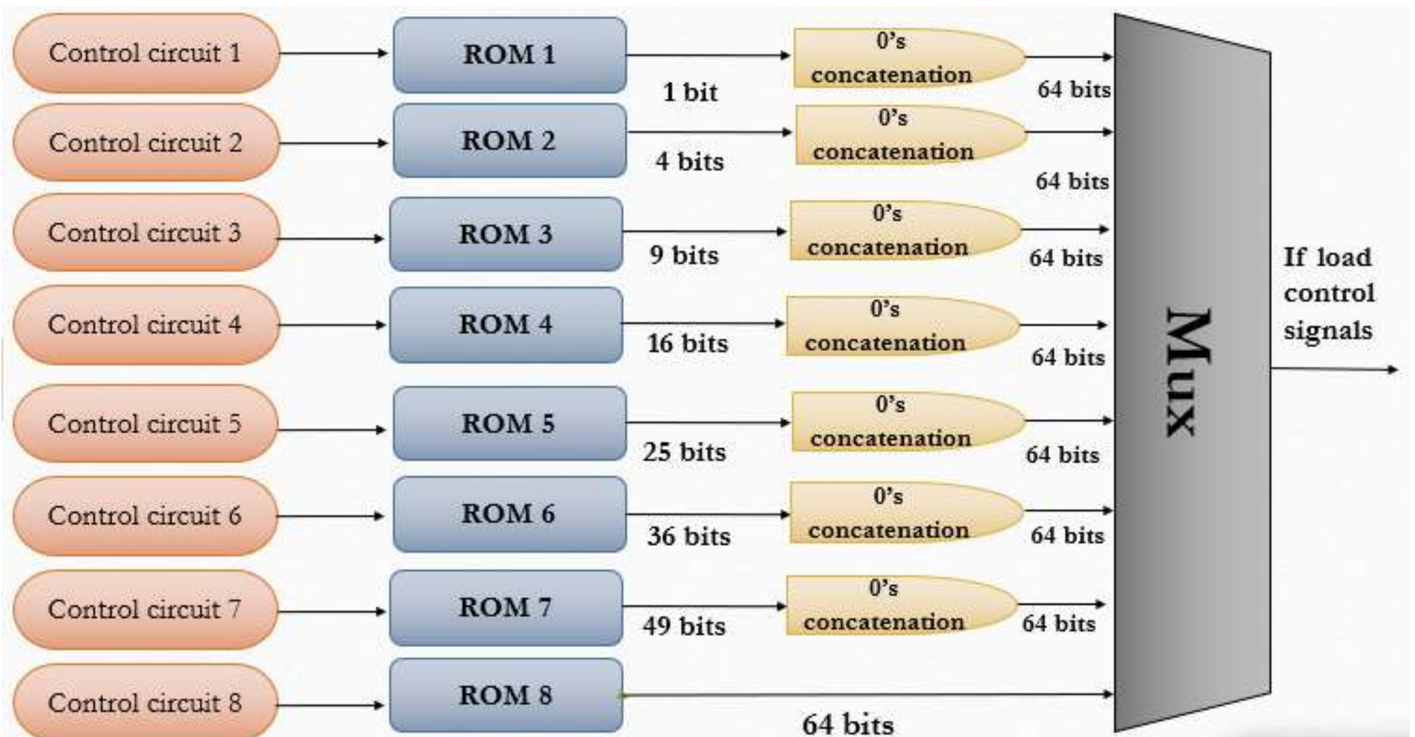


Figure 8: IF buffer Load signals generator

As the ROMs is separated to 8 ROMs, each for specific kernel size, then control sub-circuit is required for each one as it determines the start address based only on matrix input as shown in Figure 9, and then take them to be zeros concatenated and produce the whole code word for the IF buffer loads and select which one outs based on the kernel size, for multiplexer selection, it's don't cares concatenated.

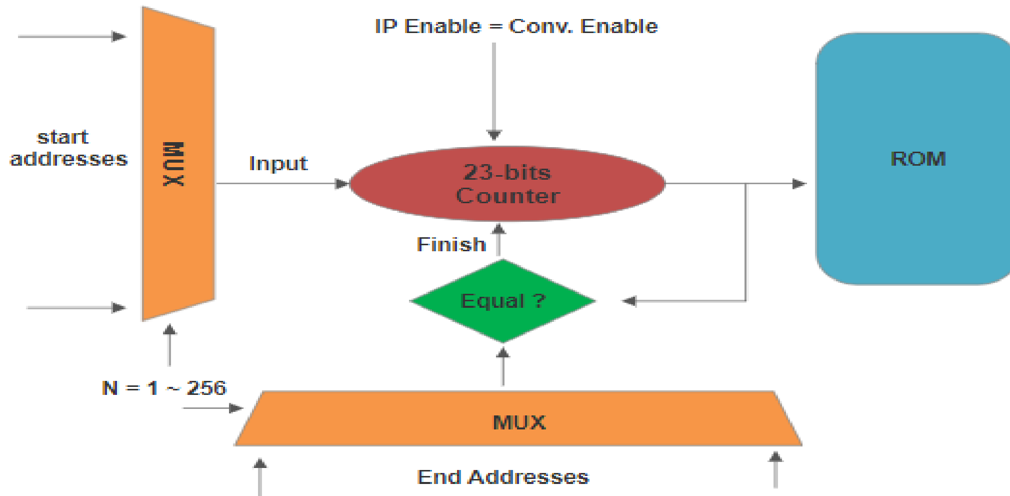


Figure 9: Control sub circuit for separated ROM design

These ROMs sizes have to be evaluated as it's extremely important metric to evaluate the area of the module, these ROM sizes can be evaluated by determining the number of transitions for each case, and that can be evaluated as follows:

For IF buffer load signals, taking into consider setup time for first submatrix:

$$at\ F = 1, \#\ of\ addresses = \sum_{N=2}^{256} (N - F + 1)^2 - 1 + 8 = 5{,}627{,}000 * 1\ bit$$

$$at\ F = 2, \#\ of\ addresses = \sum_{N=3}^{256} (N - F + 1)^2 - 1 + 8 = 5{,}561{,}457 * 4\ bits$$

$$at\ F = 3, \#\ of\ addresses = \sum_{N=4}^{256} (N - F + 1)^2 - 1 + 8 = 5{,}496{,}425 * 9\ bits$$

$$at\ F = 4, \#\ of\ addresses = \sum_{N=5}^{256} (N - F + 1)^2 - 1 + 8 = 5{,}431{,}902 * 16\ bits$$

$$at\ F = 5, \#\ of\ addresses = \sum_{N=6}^{256} (N - F + 1)^2 - 1 + 8 = 5{,}367{,}886 * 25\ bits$$

$$at\ F = 6, \#\ of\ addresses = \sum_{N=7}^{256} (N - F + 1)^2 - 1 + 8 = 5{,}304{,}375 * 36\ bits$$

$$at\ F = 7, \#\ of\ addresses = \sum_{N=8}^{256} (N - F + 1)^2 - 1 + 8 = 5{,}241{,}367 * 49\ bits$$

$$at\ F = 8, \#\ of\ addresses = \sum_{N=9}^{256} (N - F + 1)^2 - 1 + 8 = 5{,}178{,}860 * 64\ bits$$

$$Total\ size|_{IF\ buffer\ Load} = \sum_{i=1}^{8} ROM_i = 1{,}077{,}679{,}758\ \approx 128\ MB$$

Same calculations for IF input selection lines but with considerations of that they are 64 x 3 bits not only 64 elements, then it’s evaluated as:

$$Total\ Size|_{IF\ buffer\ selection}\ \approx 128 * 3 \approx 384\ MB$$

Same calculations for KW output selection line with consideration that there are 64 x 6 bits and there’s no need to customize the setup time in calculations, then it’s evaluated as:

$$Total\ Size|_{KW\ buffer\ selection}\ \approx 772.2\ MB$$

Then the total ROM size used in the convolution can be evaluated as:

$$Total\ ROM\ Size = 772 + 384 + 128 \approx 1.2\ GB!!$$

That ROM size is extremely huge Area and also bad for performance as the more ROM size is big, the worst delay it has, but the problem of performance can be resolved by dividing ROMs into small ones until it has the required performance and each one has its control circuit as it’s shown above, the good news about ROMs is that it has repeated

sequences which can be partially encoded by a lossless compression technique and take the compressed value expressed certain sequence and decode it by hardware decoding circuit to get the original sequence, that has to be modelled and iterated by GPU to discover these repeated sequences and their positions in ROMs and hence storing the compressed data only in ROMs, that will mostly reduce the ROM size powerfully to have suitable one for the application.

But the question is what's good of using ROMs rather than using straight forward convolution technique? The good aspect besides that the data transfer bandwidth between convolution and memory is extremely small compared to straight forward one, but it also good that ROMs can serve lots of convolution modules with implementing multi-input multi-output ROMs as each one can serve on one Convolution module as shown in Figure 10.

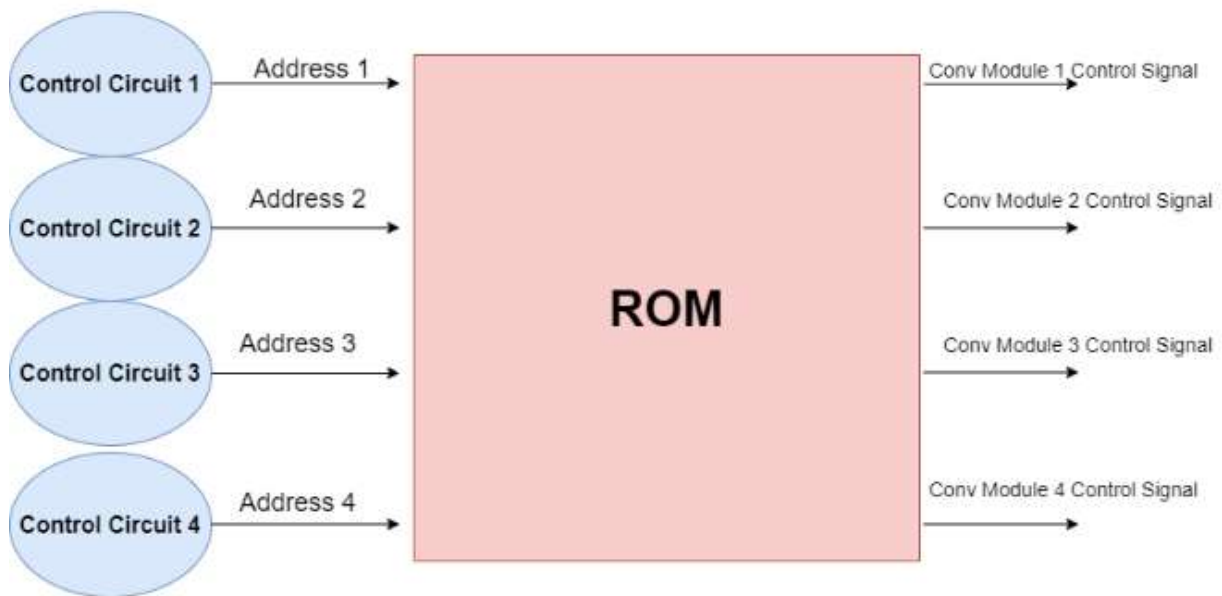


Figure 10: 4 inputs 4 outputs ROM

Final control signals have to be fed by the controller is Multiply enable to the used multipliers and to locally reset the unused registers, the multipliers are power hungry computing modules so it's better to turn them off when it's previously known that its inputs are zeros and their output is zero also, so the multiply enable is considered in

multiplier circuit as shown in Figure 11, when the multiply enable is 0, the inputs are zeros and output register is reset to zero also

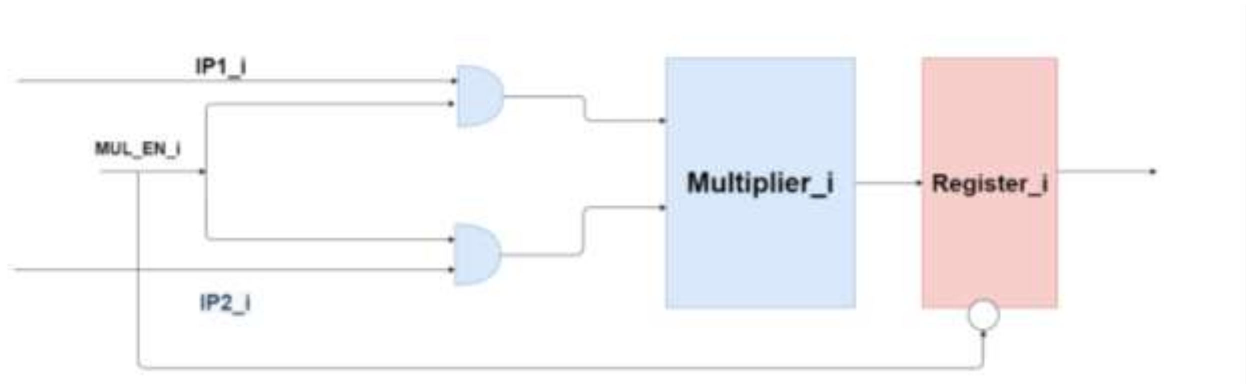


Figure 11: Multiplier with Enable

As it's known in CMOS circuits, the gates don't consume power as well as the inputs doesn't change and for unused register, they are zeros for long time so the corresponding multipliers don't consume power, so the determination of multipliers enables depends on the kernel size as well as the local reset of the unused registers is 1 as shown in Figure 12.

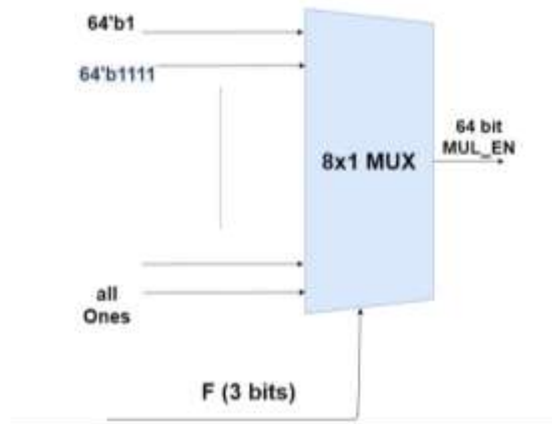


Figure 12: Multiply enable and local reset generator

An additional optimalization is to use zero detector for KW or IF buffers elements as zeros are often occurs in kernels, that helps in disable multiplier, the design of zero detector is shown in Figure 13.

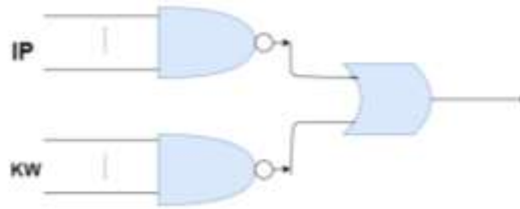


Figure 13: Zero Detector Design

2.2.2. Pool Unit

2.2.2.1. Pool Procedure

The pooling circuit in CNN is used to down sample the convolution result that reduces the input data size of the subsequent network and accelerates the calculation of the neural network. The commonly used pooling methods are average pooling and max-pooling, in which average pooling includes accumulation and division calculation, so it is not suitable for hardware implementation. Therefore, we select max-pooling to act as our pooling method. According to the principle of max-pooling, combined with the characteristics of data serial input.

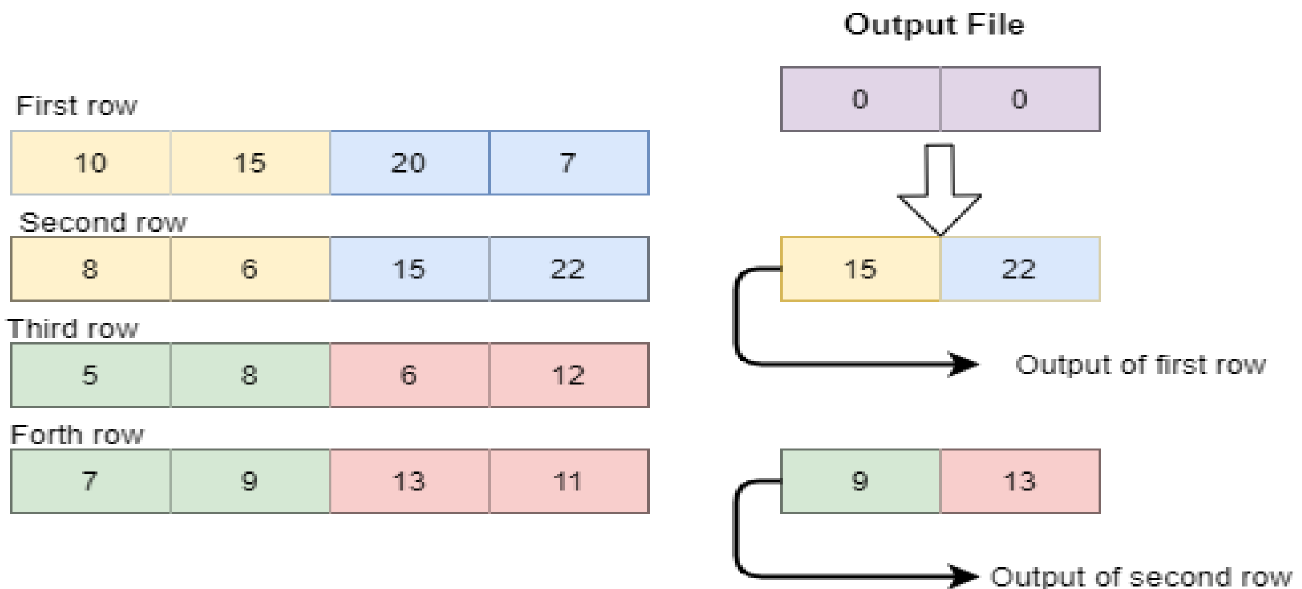


Figure 14 Example of Pooling circuit of 4x4 matrix with 2x2 filter size

For example shown in Figure 14, output of first row will be ready after all elements of first two rows enters in pool circuit, and the second one will be ready second two rows enters in pool circuit. Each output row will display element by element to be able to enter in next computing layer depending on the algorithm used.

2.2.2.2. Pool Data path

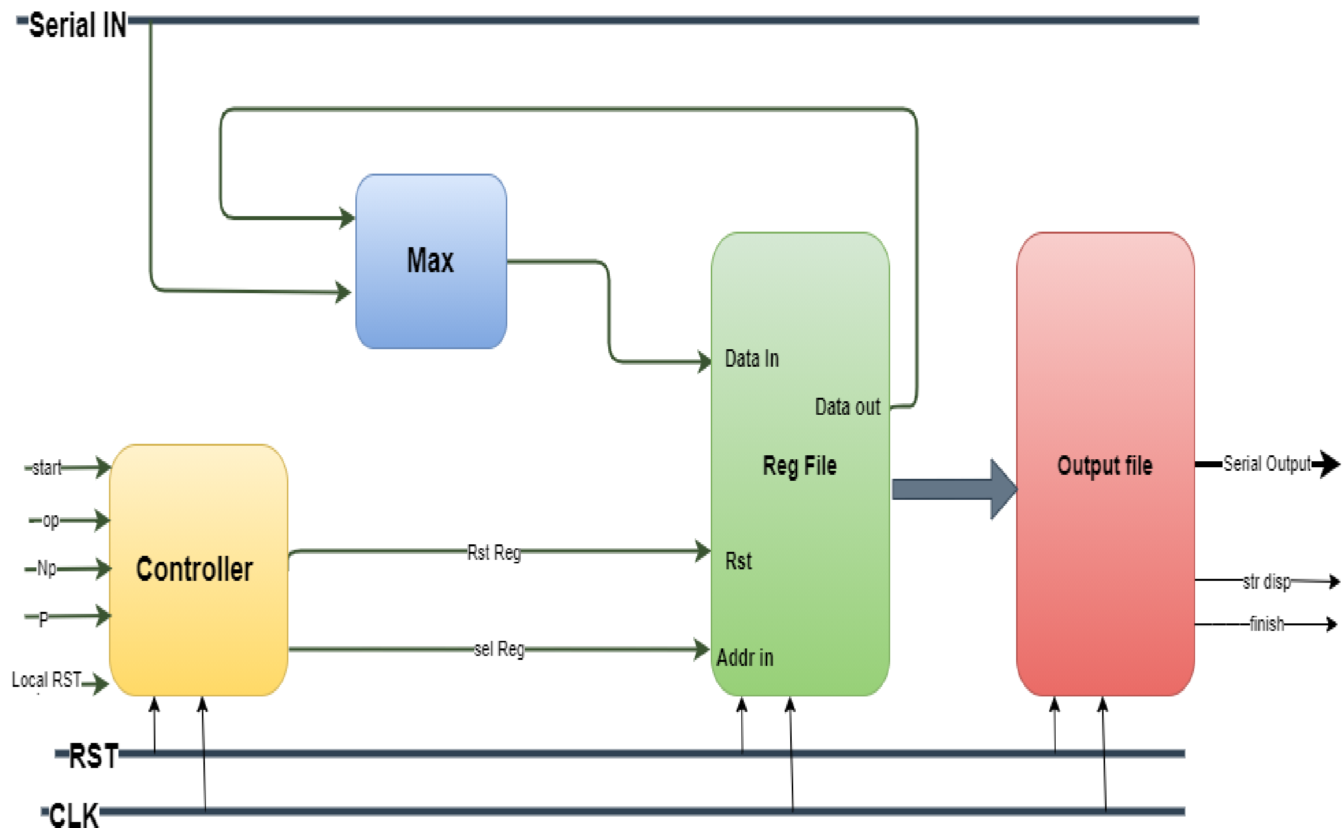


Figure 15: Max Pool design architecture

The designed serial pooling circuit consists of maximum comparator, pool controller, a previous result buffer and an Output File. The circuit structure is shown in Figure 15.

Input and output of Pool:

It consist of Control signals as:

- CLK &RST are the global signals.
- **Start** signal sent to Pool to start the operation and to be ready to receive square matrix input in series.
- **Np:** consist of 8 bits it is the range of input square matrix [1:255].
- **P:** consist of 3 bits it is the range of filter size of Max Pool [2:8].
- **Local_RST:** it is a pulse raise before each start of new operation to ensure all registers have zero values.

- **sel_Reg:** it is address select which data in Reg_File has to be compared with the input date.
- **OP** signal, if it's value equal '0' this mean that the data serial input enter the module in order(rows ordered always from left to right) as shown OP 1 in Figure 16, but if it's value equal '1' this mean that the data serial input enter the module **not** in order(rows ordered in even rows from left to right and in odd rows from left to right) to match with Conv output as shown OP 2 in figure 16, depending on this value the controller out control signals (sel_Reg, RST_Reg)

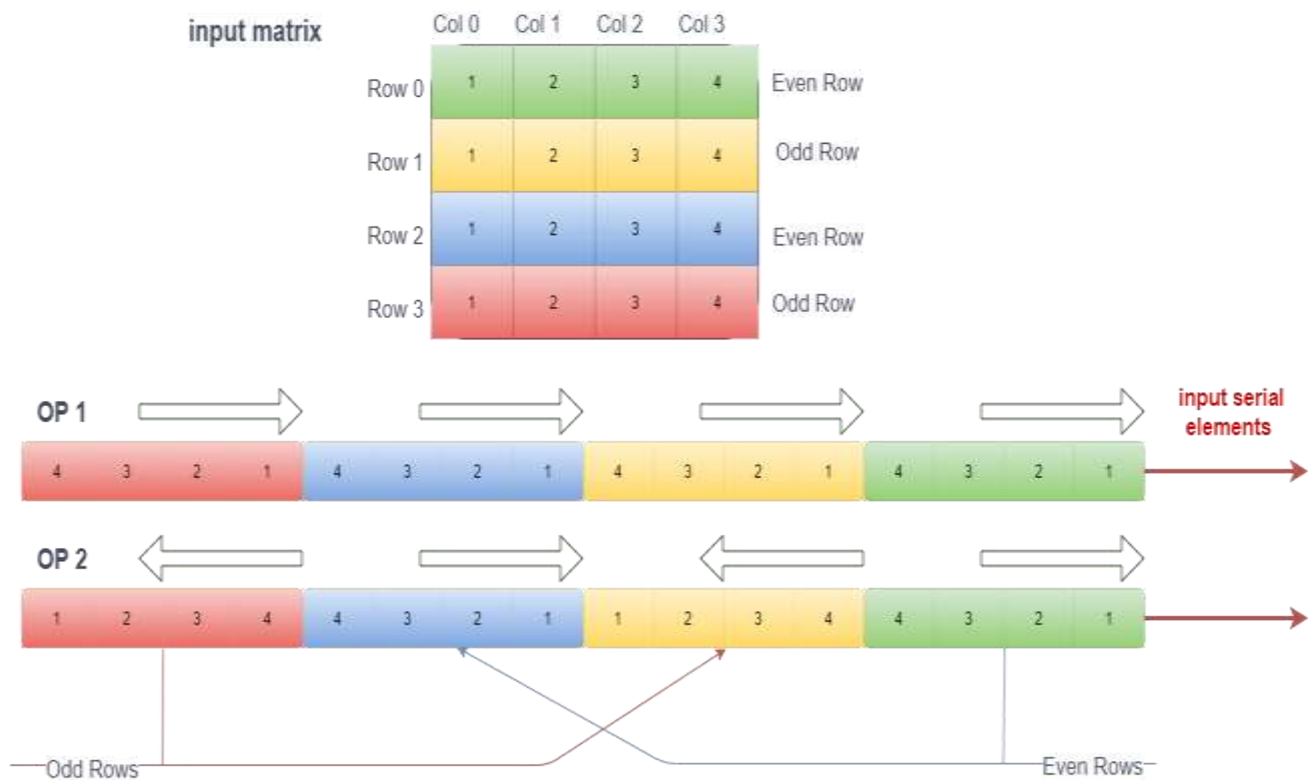


Figure 16: input serial elements for op 1 & op 2

- **RST_Reg:** it is a pulse raise when data in Reg_File has the correct values of output row and it is ready to display, then then all data in Reg_File move to Output_file then,
  - Reg_File will reset all its registers and be ready to get next elements to get next row.
  - Output_file raise Str_disp signal to start display this row element by element when it out all row elements, it lower Str_disp signal, if it is the final output row of the matrix, the finish signal will raise too for only one clock cycle.

2.2.3. Add Unit

Bias is an additional parameter in the Neural Network which is used to adjust the output along with the weighted sum of the inputs to the neuron. Therefore Bias is a constant which helps the model in a way that it can fit best for the given data.

So, bias nodes are added to increase the flexibility of the model to fit the data. Specifically, it allows the network to fit the data when all input features are equal to 0, and very likely decreases the bias of the fitted values elsewhere in the data space.

For that the addition layer is required to add the output of convolution layer to Src_c (bias matrix).

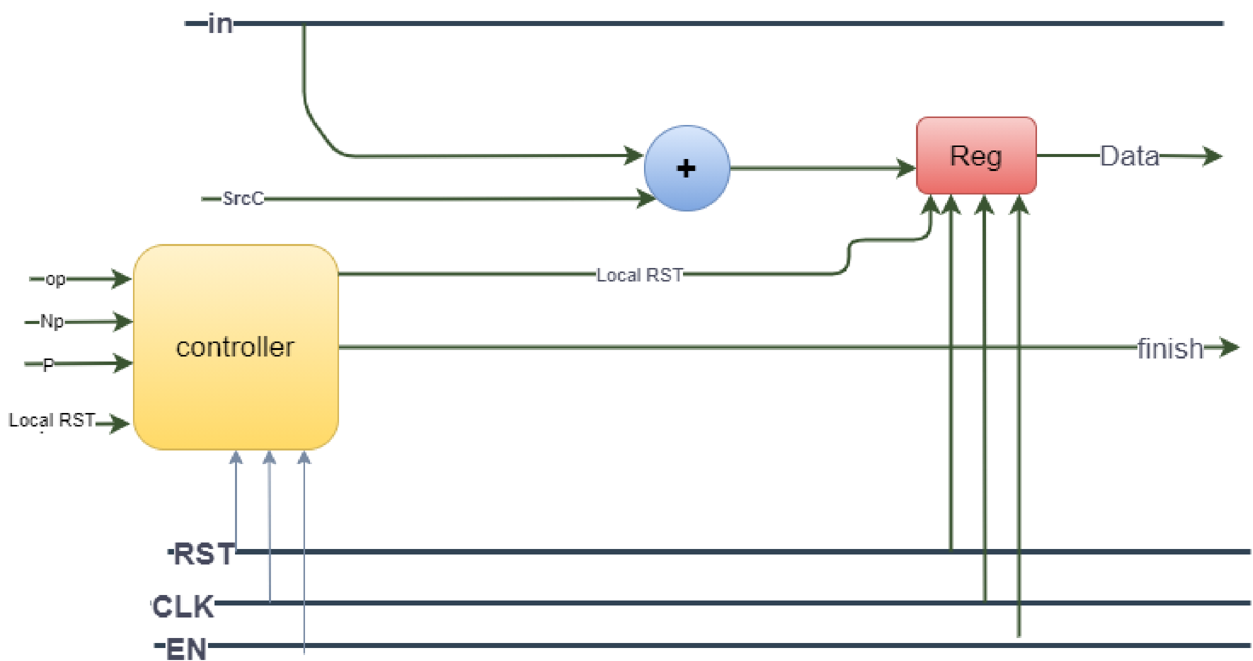


Figure 17: Add unit design architecture

As shown Figure 17 the "Add_Module" takes input_1 (in1) and input_2 (in2) and some control signals like Enable , Local_reset and the size of image (N) as inputs and gives back output (out) , Finish signal Overflow_Add as outputs then start to check the signs of both inputs in1 and in2 to determine which case must be applied.

First of all, if reset signal or Local_reset is activated then the output is Zero.

- **Output**

There are three cases may happen if Enable signal is activated:

First case: If they both are positive or both are negative, then output will be the addition of them, then the output takes the sign of any one of them.

Second case: if in1 is positive while in2 is negative, then output will be the subtraction of both in1 and in2 (out=in1-in2), then starts to check which one is greater, if (in1>in2) then out=in1-in2 and the sign of the output must be positive, while if (in2> in1) then out=in2-in1 and the sign of the output must be negative.

Third case: if in2 is positive while in1 is negative, then output will be the subtraction of both in1 and in2 (out=in2-in1), then starts to check which one is greater, if (in1>in2) then out=in1-in2 and the sign of the output must be negative, while if (in2> in1) then out=in2-in1 and the sign of the output must be positive.

- **Overflow_Add**

To check if there is an **overflow** happen we check the sign of MSB of output if '1' this means an overflow happen and then activate this signal and send it as feedback to the controller to take the appropriate action.

- **Finish**

To determine the **Finish** signal when must be activated, we need a Counter (Counter_out) that counts until reach to a pre-defined number (End_address) which is determined by N*N, If (Counter_out = End_address) then Finish signal is activated to '1'.

2.2.4. Relu Unit

The ReLU function is another non-linear activation function that has gained popularity in the deep learning domain. ReLU stands for **Rectified Linear Unit**. The main advantage of using the ReLU function over other activation functions is that it does not activate all the neurons at the same time.

In Activation functions, Relu is the Simplest, faster and suitable for HW implementation.

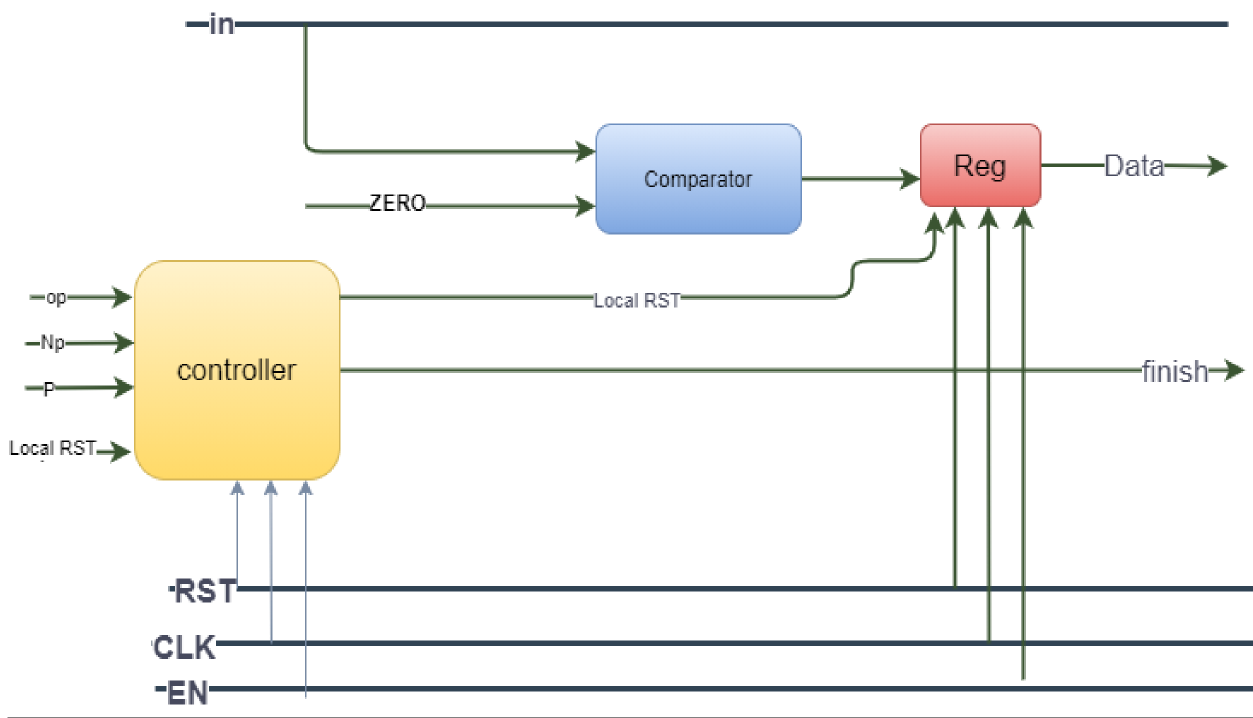


Figure 18: Relu unit design archiecture

As shown in Figure 18 the “Relu” module takes input (in) and some control signals like Enable, Local_reset and the size of image (N) as inputs and gives back output (out), Finish signal as outputs.

➢ **Output**

The main idea of Relu is to compare the input with zero, we find that we have two cases:

First case: if input is positive number then the output will be equal the input

Second case: if input is negative number then the output will be Zero.

➢ **Finish**

To determine the **Finish** signal when must be activated, we need a Counter (Counter_out) that counts until reach to a pre-defined number (End_address) which is determined by N*N, If (Counter_out = End_address) then Finish signal is activated to ‘1’.

2.2.4. Crossbar Unit

The configuration of the crossbar allows data to flow to different computing components, which can speed up different algorithms. The entire accelerator can be parameterized and data accessed through an external bus.

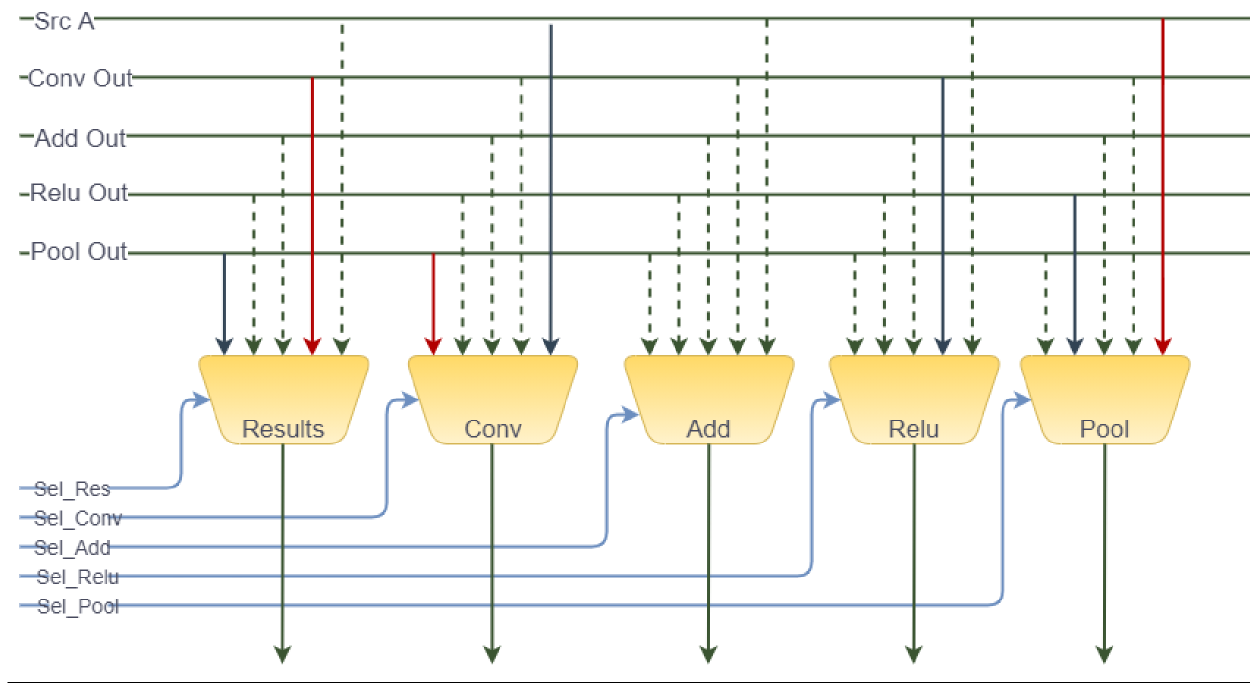


Figure 19: Crossbar design architecture

➢ **Main Idea**

The crossbar mainly consists of f an input buffer (FIFO), a configuration register group (Cfg Regs), and five multiplexers (MUX), for simplicity we didn't use the "FIFO" and adjust the design to consist of five MUXES only. The five MUX select the data path to be opened according to the configuration information of the Cfg Regs. Thus, the data stream passes through different calculation modules in different orders. For example, the edge detection operation in image processing usually extracts the input image and then convolves it with the edge detection operator.

In the convolutional neural network algorithm, it is usually necessary to perform convolution, ReLU, and pooling operations on the source matrix. You need to configure the MUX to apply this order of operation.

The main benefit from the crossbar module is to have flexibility to change the order of operation or algorithm implemented on the data of the same architecture instead of having only one fixed sequence of operations, Using the five MUX which select the data path to be opened according to the configuration information of the Reconfigurable controller. Thus, the data stream passes through different calculation modules in different orders. For example, the edge detection operation in image processing usually extracts the input image and then convolves it with the edge detection operator. It is necessary to configure the MUX as the path in the red part of Figure 19. In the convolutional neural network algorithm, it is usually necessary to perform convolution, ReLU, and pooling operations on the source matrix. You need to configure the MUX as the path in the blue part of Figure 19Figure 18.

- **Input and output port list**

<u>Inputs of crossbar</u>:

It mainly consist of Control signals of MUX (sel_Res, sel_Pool, sel_Conv, sel_Add, sel_Relu), SrcA and modules outputs as (CONV, ADD, RELU and POOL).

<u>Outputs of crossbar</u>:

It mainly consist of modules inputs (CONV, ADD, RELU and POOL) and the Results of PE.

<u>The mechanism:</u>

The mechanism of crossbar is to recieve the selection signal of the mux and choose the right output to be passed.

- For analogous, if we have the image matrix and we need to apply 2d-conv then Relu then Max pooling layers in that sequence then get the result, all we must do to put the selection first by “sel_Conv” to out src_A as an input to convolution module then we put selection to be “sel_Relu” to enter the output of convolution as an input to Relu as before we put selection by “sel_Pool” then finally we put it by ”sel_Res” to get the final result as Max_pool result.

  There are all component modules in PE, now we will talk in details about The Memory File

2.3. The Memory File

2.3.1. Memory File

Processing elements have to attached to RAMs to serve their loading input data or storing result processed data, the PE has three inputs which are source A, and that to feed it with input data to any computing module and that's routed by crossbar and source B which feed the convolution with kernel matrix and finally source C which is responsible to feed the add module with the accumulating input matrix, so the memory file as it's shown in Figure 20 has four RAMs, one is called RAMB and it's specific to feed source B, that RAM stores data from system and load it to convolution module, but the other three RAMs are specific for loading source A, C, and PE result, the multiplexers attached to them is for Memory Interleaving, as any RAM of them can work as source A,C, or even result RAM, that helps in sequence of layers, for example, if RAM1 works as result RAM and hence it's required to load the result and work on in another layer, so it can work as source A RAM and so on, that's better than make each RAM specific for a certain target and between successive layers, it has to make memory transfer, which is consumption of time and hence bad for performance, so the choice is always performance over area.

RAMs interface with Main RAM is also important as the data can not flow to the PE computing modules unless they are stored in Memory file first, as RAMs in Memory file are aware of the nature of computing modules inside the PE and particularly convolution module, as it has 8 input read data bus and they have to be fed in particular way as discussed above which is overhead on CPU to work on, so the RAMs here are processing RAMs more than storage ones.

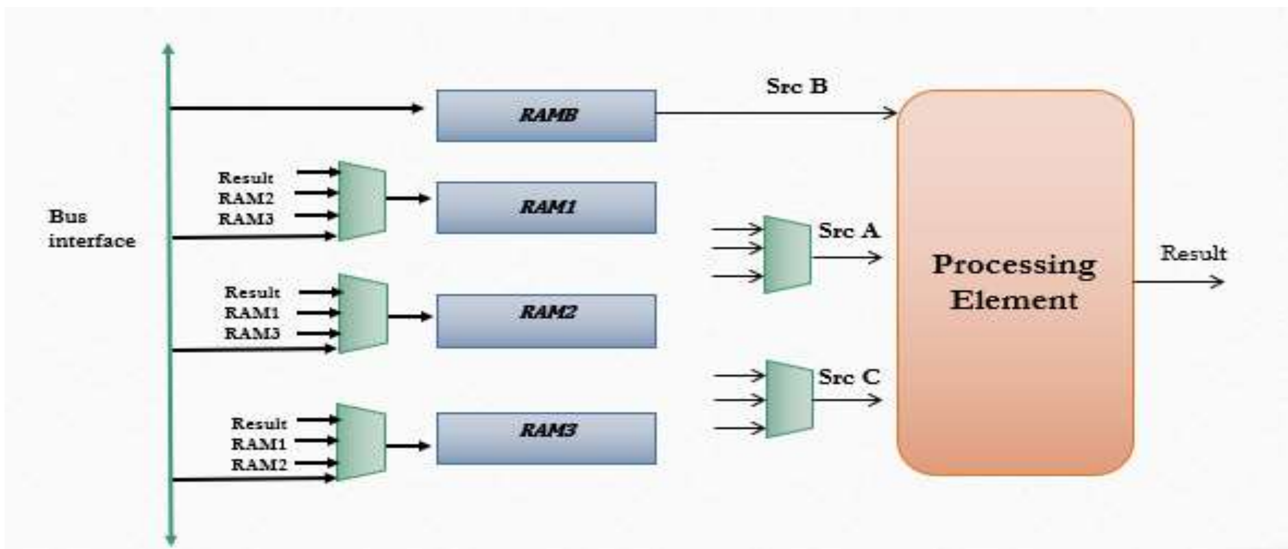


Figure 20: Memory File Design

2.3.2. RAM Design

The RAM specifications and design as shown in Figure 21 has to be designed for worst case that stored matrix inside has the maximum value which is 256, and as it's square matrix, then number of positions have to be available inside the RAM is 65536 positions each is 16 bits that is implemented as 1D RAM and has 8 output data buses if it feeds the convolution with 8 read data enables and 8 read address buses each is 16 bits and only one write bus with enable and write address, but the RAM feeds the computing modules except convolution with only one data bus which is the least significant one.

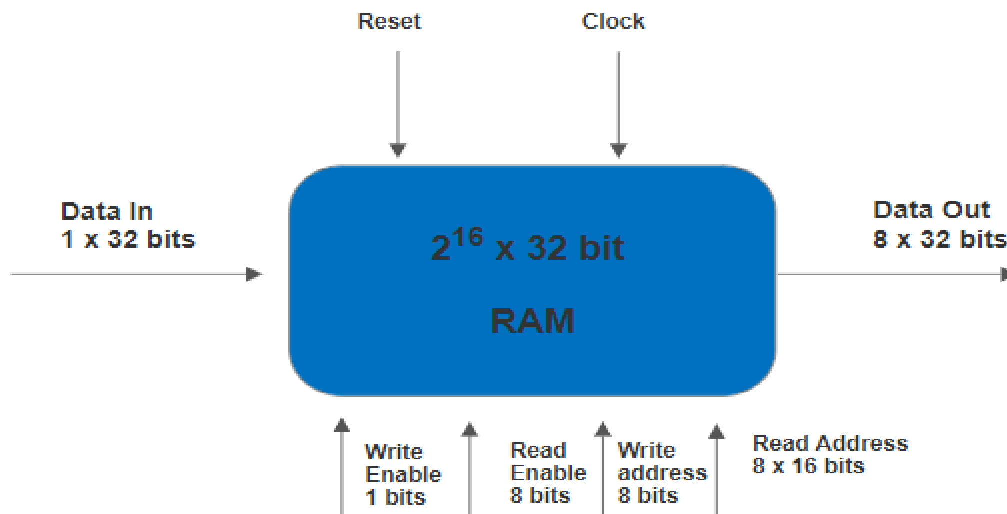


Figure 21: RAM Design

By specifying the Operations which the RAMs can operate on, it's obvious that convolution has different nature of loading and storing compared to loading and storing on other computing modules like Pooling, ReLU, Add, or even another RAM or bs interface, that's because convolution provides data in zigzag way, the odd rows comes in order and even ones comes in opposite order, same for loading which has to determine the addresses of the updated elements provided to convolution on each transition, and hence the operations can be specified as follows:

1) Operation 1: Storing data into RAM from bus interface, another RAM, or from computing modules that has no convolution
2) Operation 2: Storing data into RAM from computing modules that has convolution, and the aim here is to store the unordered result in order for another operation.
3) Operation 3: Loading data from RAM into bus interface, another RAM, or into computing modules that has no convolution
4) Operation 4: Loading data from RAM into computing modules that has convolution, and as the convolution is always is the first module, then the way of updated elements has to be declared here.
5) Operation 5: Loading data from RAM into add module when it's directly after convolution as RAM has to feed the add here with zigzag way to accumulate each element resulted from convolution to the corresponding element in source C matrix.

These operations aims to get the correct addresses of reading or writing with correct enables which makes the RAM operates in a certain operation from one of the above and that is the functionality of Arbiter attached to the RAM to direct it in the right manner as shown in Figure 22.

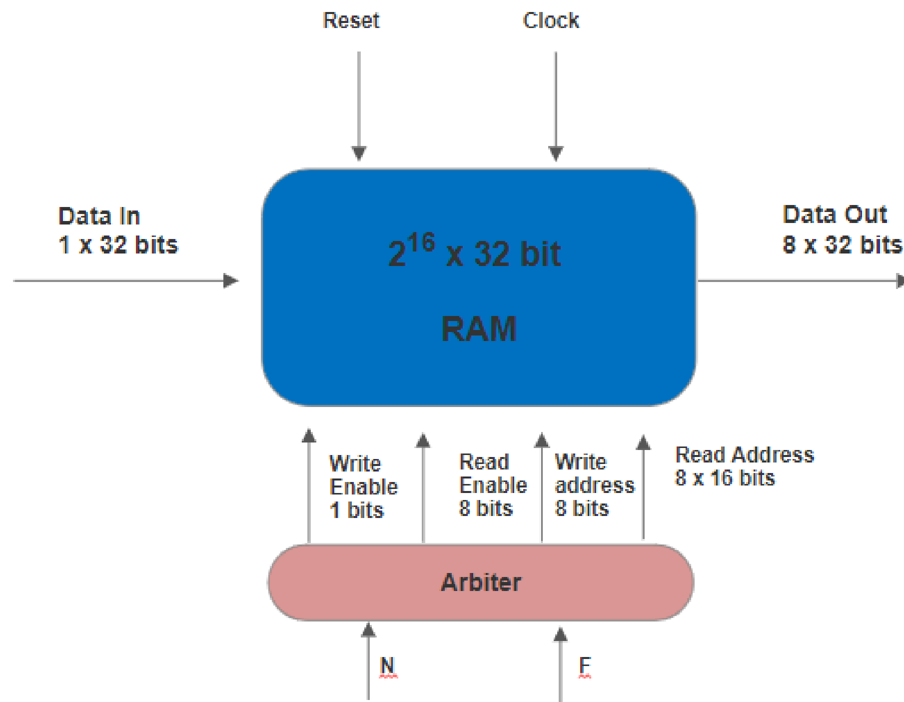


Figure 22: RAM Design with Arbiter

The Arbiter takes matrix and kernel sizes with other operation control signals from the reconfigurable controller which guides the arbiter to direct RAM to work on which operation according to memory interleaving positions discussed before, as all RAMs can be interleaved has to able to operate in all above operations as it's possible for any one of them to be any source or destination due to user preference, so the whole operations logic has to be included to all of them as it will be discussed next.

2.3.3. Operations Logic

Starting with operation 1 design as shown in Figure 23, the aim here is normal storing input matrix in Row-wise, so the counter needed here as the 16 bits counter has to points from first address in RAM and count on until it reaches the address of matrix size square and that the functionality of below multiplexer which has inputs from 1 to 65536 and selects which one according to input matrix size and the end address here is the square of input matrix size.

When it reaches the end address the counter is disabled and informs end of operation with finish signal raises high until it has local reset from the reconfigurable controller after it has finished the operation, the output of that counter is the write

address, while writing the arbiter raise write enable up and disable read enables for the RAM.

The note here is that the reconfigurable controller has to have control on the count enable of the controller, as when RAM is storing from Pooling which takes time until it processes the elements and then decide what's maximum and hence outs results, so it takes time to decide and then outs data as discussed before and hence the controller that is aware of that sequence has to guide the arbiter when to write and when to stop.

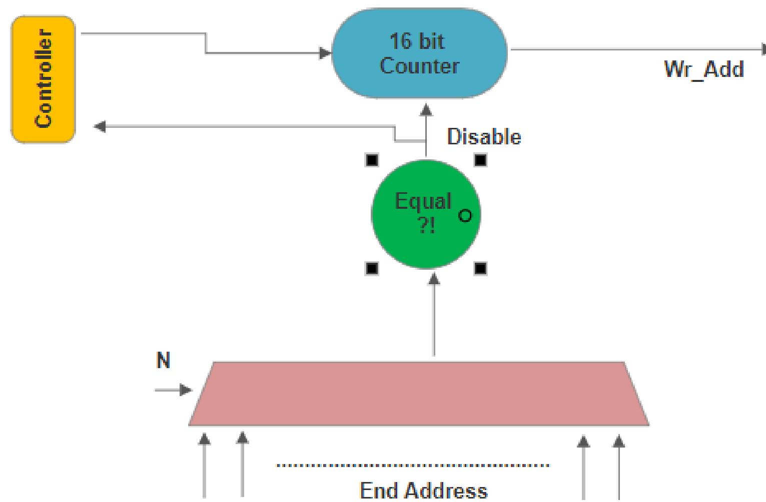


Figure 23: Operation 1 Logic

In operation 3 as shown in Figure 24, same logic happens but the output is the address of least significant read data bus.

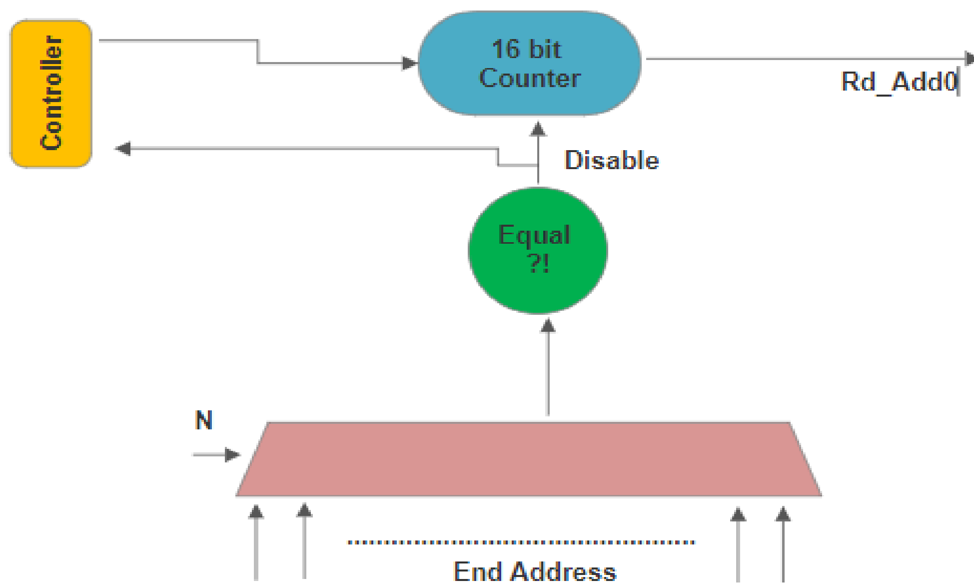


Figure 24: Operation 3 Logic

In operation 2 as shown in Figure 26, when the storing is from convolution module in zigzag is as follows, the 2D dealing with RAM here is required as the convolution outs the result of odd rows in order and even ones out of order as shown in Figure 23

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Mapping ? ←

1	2	3	4
8	7	6	5
9	10	11	12
16	15	14	13

Figure 25: Convolution result matrix

As shown here the convolution result gets in the order of right hand side, and the target of operation 3 to map it to the order of left hand side, so the design depends counter y which count on rows and by its LSB, that determination of even or odd row can be easily obtained by checking if it's LSB = 0, it's odd row, if it's 1, then it's even one, and that attached to up/down counter x, which counts on columns, if it's odd row, counter x count up from 0 to matrix size and vice versa if it's even row, it's disabled by sub circuit which informs it to stop at N if it counts up and stop at 0 if it count down according to LSB of counter y, that besides the count enable signal that under the control of reconfigurable controller as it's discussed before.

Counter y is enabled to count based on that if counter x has finishes the row or not and that's also based on LSB of counter y, and disable the counter y when it reaches the matrix size.

Finally, the conversion from 2D to 1D is done by the equation of

$$1D\ RAM\ Address = y * (N + 1) + x$$

$$: x = column\ number, y = row\ number, N = 0 \sim 255$$

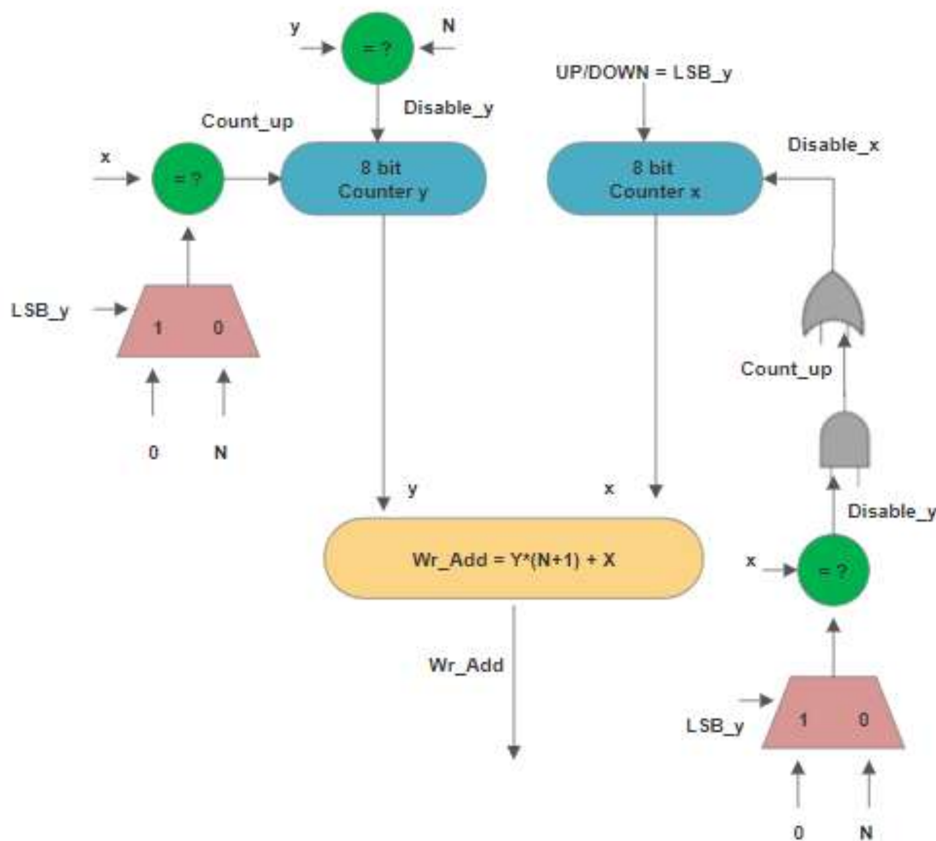


Figure 26: Operation 2 logic

That feedback circuit is responsible for getting writing address of zigzag mode to be stored in RAM in normal mode, as if the result of convolution has to be fed to another layer, it has to be ordered in right manner to be fed correctly

Operation 5 is same logic but produces the read address of least significant bus instead of write address.

Finally, operation 4 as shown in Figure 27 is the most difficult operation to be executed, as when the RAM feeds the convolution data it has to feed updating element for each transition and these elements positions are not having addresses which can be deduced from certain equations but these positions in RAMs depends on the input matrix and kernel sizes.

So, the ROMs or LUTs approach is considered here also as shown in Figure 26 with the same designs for control sub-circuits and also the ROMs are divided into 8 ROMs each one for specific kernel size, and also the control circuits have their enables and local resets feeding from the reconfigurable controller and the operating one feedback controller with finish signa; at the end of operation.

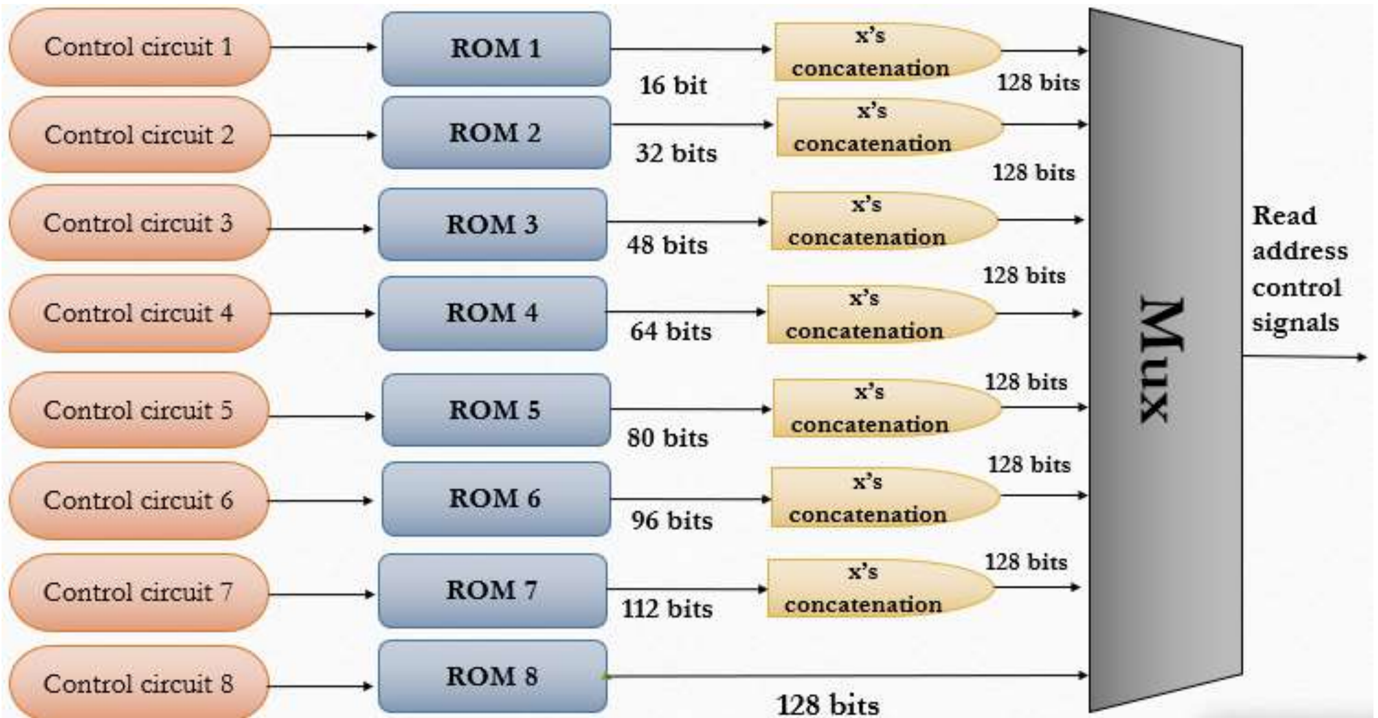


Figure 27: Operation 4 Logic

Here's an extra calculation can be added to the ROM sizes used in the accelerator data-path, which can be evaluated according to following equations:

$$ROM\ Size = \sum_{i=1}^{8} ROM\ Size_i$$

$$ROM\ Size_i = \sum_{N=i+1}^{256} ((N - F + 1)^2 - 1 + 8)\ *\ no\ of\ bits\ in\ codeword$$

Then $Total\ ROM\ Size|_{Read\ address} = 3{,}068{,}046{,}816 \approx 360\ MB$

$$Then, Total\ ROM\ Size|_{Datapath} = 1.6\ GB\ !!$$

That huge ROM size has to be optimized as it's mention before.

As three RAMs which is used in the accelerator data-path has to be able to operate in operation 4 due to memory interleaving, these ROMs here for operation 4 has to be 3 input 3 output ROMs to serve the operation 4 in the three RAMs.

It's also worth to mention that the operating sub circuit control has to be enabled by an operation enable to make it go through the sequence of read addresses from the first address loaded until it reaches the end address and then feedbacks with the finish signal as it's discussed before in Convolution module.

Finally, the Arbiter takes a 3 bits Opcode from the reconfigurable controller which directs it to the enable the needed operation and outs its own control signals to the RAM as shown in Figure 28, as the multiplexer shown selects which operation control signal can out to the RAM by the obtained opcode.

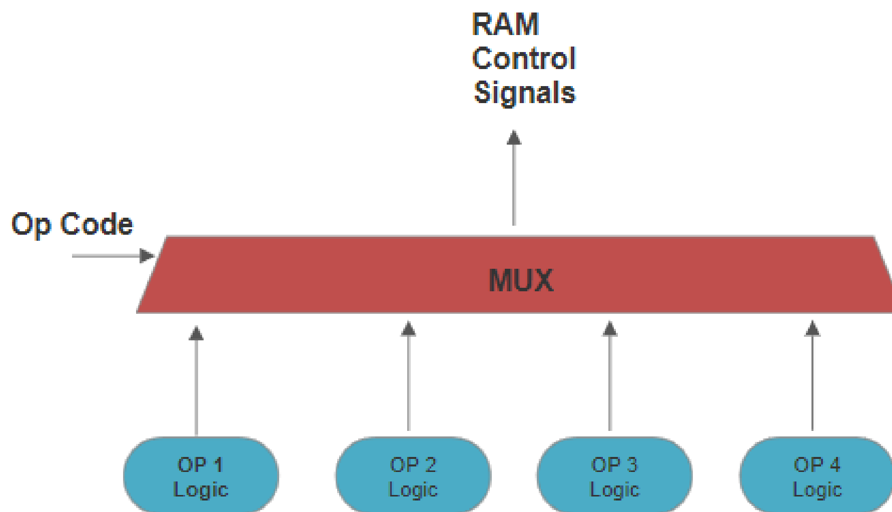


Figure 28: Arbiter Opcode selection

For RAMB which is considered as a special case of other RAMs, that has size of 64 element each is 16 bits for worst case of storing kernel size of 8, storing with one write address and load to convolution module by 8 read data bus.

It's only storing from the main RAM through bus interface and loading the kernel to convolution for the first time in known order, these are the two operations has to be executed by RAMB

That's all about the Memory File and the whole Accelerator Data-path

2.4. Custom Instructions

The Custom instructions are those ones which can be executed by the accelerator not the CPU, as when one instruction of them comes, the CPU passes them to the accelerator to work on them, so the reconfigurable controller decodes them and provides the control signals to the whole data-path to direct it where and when to go, the determination of custom instruction is the most important step in determination of the reconfigurable controller functionality.

So, the custom instructions which is 32-bit instruction according to the RISCV CPU instruction-based, which uses R-type instructions and specify 7-bit for the determination of the operation and these are Opcode filed, the Operands related to the instruction can be stored in specific register R_s,which can hold them.

The custom instructions can be characterized for four types:

1) Reset Instruction:

It's responsible for resetting all registers inside the data-path locally to zero, and that for any reason, maybe because overflow occurrence or Main RAM loaded wrong data, and that instruction is determined by the Opcode only as shown below

$Op - Code\ (7bits)$

2) Load and Store Instructions:
It's responsible to inform the reconfigurable controller to Store data from Main RAM into specific RAM in Data-path through bus interface or to Load data from specific RAM to Main RAM, and that is defined by Opcode only as the destination or source memory can be implicitly defined inside as it will be discussed later on

3) Memory Transfer Instructions:
   It's responsible to transfer data from one memory to another if not to use memory interleaving or for any advance coming optimization, it's also can be determined by the Opcode.

4) Execution Instructions:
   It's responsible for executing the operations inside the PE, whether convolution, Pooling, ReLU, or even addition, but as it's shown above the computing modules inside the PE processes on matrices which is sequence of elements, and no module of them stores it's result and then concatenated to feed it into another computing modules and if RAMs in Memory file are used in transfer between one computing element to another, that hurts the performance much because the data transfer between computing modules and memory increases, and as it's usually the computing modules contain layer which means it's usually needed to execute combination of instructions, combinations approach is used here, which means the instruction can be ordered combination for example, takes the data in RAM1 and make convolution on it with RAMB, then ReLU on convolution data out, then accumulate ReLU out with RAM2 data in add module, then pooling the output of add, and finally store pooling result into RAM3, that makes the whole layer to be executed without resorting to RAMs except in loading to first module and storing from last one.

The instruction format of that is as follows:

$Op\ Code$ $(7\ bits)$	N $(8\ bits)$	F $3\ bits$	P $3\ bits$	$N_c = N - f + 1$ $(8\ bits)$	$N_p = \frac{N - f + 1}{p}$ $(8\ bits)$	$Instruction\ \#$

The operands needed for that type of instructions, besides the determination of Opcode which selects the operations and will be discussed in more details later on, is to determine the input matrix size which is between 1~256 and can be expressed by 8 bits and convolution kernel size which is between 1~8 and can be expressed by 3 bits and also the Pool filter size which is also 3 bits and other auxiliary operands which helps it to determine the output matrix size of convolution and pooling which is very important for the data flow determination of the matrix inside the layer or combination, and finally the

Instruction number which is additional operand for future works and will be explained later.

Therefore, by characterizing the combinations can be executed by the PE, the possibilities are much, but based on some practical or algorithmic aspects related to CNN, like that convolution always comes the first of the layer, the combinations (Combos) are selected like below

Table 2 the possible Combos can express any CNN architectures.

Combo 1	CONV	POOL	RELU	ADD
Combo 2	CONV	POOL	ADD	RELU
Combo 3	CONV	RELU	POOL	ADD
Combo 4	CONV	RELU	ADD	POOL
Combo 5	CONV	ADD	POOL	RELU
Combo 6	CONV	ADD	RELU	POOL
Combo 7	CONV	POOL	RELU	---------
Combo 8	CONV	POOL	ADD	---------
Combo 9	CONV	RELU	POOL	---------
Combo 10	CONV	RELU	ADD	---------
Combo 11	CONV	ADD	POOL	---------
Combo 12	CONV	ADD	RELU	---------
Combo 13	CONV	POOL	---------	---------
Combo 14	CONV	RELU	---------	---------
Combo 15	CONV	ADD	---------	---------
Combo 16	CONV	---------	---------	---------
Combo 17	POOL	ADD	RELU	-------
Combo 18	POOL	RELU	ADD	-------
Combo 19	POOL	ADD	-------	-------

Combo 20	POOL	RELU	-------	-------
Combo 21	RELU	ADD	POOL	-------
Combo 22	RELU	POOL	ADD	-------
Combo 23	RELU	POOL	-------	-------
Combo 24	RELU	ADD	-------	-------
Combo 25	ADD	POOL	RELU	-------
Combo 26	ADD	RELU	POOL	-------
Combo 27	ADD	POOL	-------	-------
Combo 28	ADD	RELU	-------	-------
Combo 29	POOL	-------	-------	-------
Combo 30	RELU	-------	-------	-------
Combo 31	ADD	-------	-------	-------
Combo 32	NOP	NOP	NOP	NOP

These Combos can express any CNN architectures, even if the Combo needed is not declared which not usually happen, the needed Combo can be executed by concatenating more than one of the above Combos.

As it's shown in the table??, the number of possible Combos is 32 which can be expressed by 5 LSBs in the Opcode and the 2 MSBs are for Memory inter-leaving, as if they are 01, then RAM1 is the SrcA, RAM2 is SrcC, and RAM3 is the Result, and if they are 10, then RAM1 is the Result, RAM2 is SrcA, and RAM3 is SrcC, and finally if they are 11, then RAM1 is the SrcC, RAM2 is the Result, and RAM3 is SrcA.

By that, the custom instructions with Opcodes can be totally customized with the table??:

Table 3 the custom instructions with Opcodes.

Op Code	Instruction Function	Source and Destinations	Operands needed
"00 000 00"	Reset	--------------------------------------	-------------------------------
"00 001 00"	Store in MB	$Source = Main\ RAM, Destination = MB$	$Start\ Address, f, Inst\#$

“00 001 01”	Store in M1	$Source = Main\ RAM, Destination = M1$	$Start\ Address, N, Inst\#$
“00 001 10”	Store in M2	$Source = Main\ RAM, Destination = M2$	$Start\ Address, N, Inst\#$
“00 001 11”	Store in M3	$Source = Main\ RAM, Destination = M3$	$Start\ Address, N, Inst\#$
“00 010 01”	Load from M1	$Source = M1, Destination = Main\ RAM$	$Start\ Address, N, Inst\#$
“00 010 10”	Load from M2	$Source = M1, Destination = Main\ RAM$	$Start\ Address, N, Inst\#$
“00 010 11”	Load from M3	$Source = M1, Destination = Main\ RAM$	$Start\ Address, N, Inst\#$
“00 10 000”	Mov M1 to M2	$Source = M1, Destination = M2$	$N, Inst\ \#$
“00 10 001”	Mov M1 to M3	$Source = M1, Destination = M3$	$N, Inst\ \#$
“00 10 010”	Mov M2 to M1	$Source = M2, Destination = M1$	$N, Inst\ \#$
“00 10 011”	Mov M3 to M1	$Source = M3, Destination = M1$	$N, Inst\ \#$
“00 10 100”	Mov M2 to M3	$Source = M2, Destination = M3$	$N, Inst\ \#$
“00 10 101”	Mov M3 to M2	$Source = M3, Destination = M2$	$N, Inst\ \#$
-------------	-------------------	--	-----------------------------------
“01 00000”	Combo1	$SrcA = M1, SrcC = M2, Result = M3$	$N, f, p, N_c, N_p, Inst\#$
“01 00001”	Combo2	$SrcA = M1, SrcC = M2, Result = M3$	$N, f, p, N_c, N_p, Inst\#$
“01 00010”	Combo3	$SrcA = M1, SrcC = M2, Result = M3$	$N, f, p, N_c, N_p, Inst\#$
“01 00011”	Combo4	$SrcA = M1, SrcC = M2, Result = M3$	$N, f, p, N_c, N_p, Inst\#$
“01 00100”	Combo5	$SrcA = M1, SrcC = M2, Result = M3$	$N, f, p, N_c, N_p, Inst\#$
“01 00101”	Combo6	$SrcA = M1, SrcC = M2, Result = M3$	$N, f, p, N_c, N_p, Inst\#$
“01 00110”	Combo7	$SrcA = M1, SrcC = M2, Result = M3$	$N, f, p, N_c, N_p, Inst\#$
“01 00111”	Combo8	$SrcA = M1, SrcC = M2, Result = M3$	$N, f, p, N_c, N_p, Inst\#$
“01 01000”	Combo9	$SrcA = M1, SrcC = M2, Result = M3$	$N, f, p, N_c, N_p, Inst\#$
“01 01001”	Combo10	$SrcA = M1, SrcC = M2, Result = M3$	$N, f, p, N_c, N_p, Inst\#$
“01 01010”	Combo11	$SrcA = M1, SrcC = M2, Result = M3$	$N, f, p, N_c, N_p, Inst\#$
“01 01011”	Combo12	$SrcA = M1, SrcC = M2, Result = M3$	$N, f, p, N_c, N_p, Inst\#$
“01 01100”	Combo13	$SrcA = M1, SrcC = M2, Result = M3$	$N, f, p, N_c, N_p, Inst\#$
“01 01101”	Combo14	$SrcA = M1, SrcC = M2, Result = M3$	$N, f, p, N_c, N_p, Inst\#$
“01 01110”	Combo15	$SrcA = M1, SrcC = M2, Result = M3$	$N, f, p, N_c, N_p, Inst\#$
“01 01111”	Combo16	$SrcA = M1, SrcC = M2, Result = M3$	$N, f, p, N_c, N_p, Inst\#$
“01 10000”	Combo17	$SrcA = M1, SrcC = M2, Result = M3$	$N, f, p, N_c, N_p, Inst\#$
“01 10001”	Combo18	$SrcA = M1, SrcC = M2, Result = M3$	$N, f, p, N_c, N_p, Inst\#$
“01 10010”	Combo19	$SrcA = M1, SrcC = M2, Result = M3$	$N, f, p, N_c, N_p, Inst\#$
“01 10011”	Combo20	$SrcA = M1, SrcC = M2, Result = M3$	$N, f, p, N_c, N_p, Inst\#$
“01 10100”	Combo21	$SrcA = M1, SrcC = M2, Result = M3$	$N, f, p, N_c, N_p, Inst\#$
“01 10101”	Combo22	$SrcA = M1, SrcC = M2, Result = M3$	$N, f, p, N_c, N_p, Inst\#$
“01 10110”	Combo23	$SrcA = M1, SrcC = M2, Result = M3$	$N, f, p, N_c, N_p, Inst\#$

“01 10111”	Combo24	$SrcA = M1, SrcC = M2, Result = M3$	$N, f, p, N_c, N_p, Inst\#$
“01 11000”	Combo25	$SrcA = M1, SrcC = M2, Result = M3$	$N, f, p, N_c, N_p, Inst\#$
“01 11001”	Combo26	$SrcA = M1, SrcC = M2, Result = M3$	$N, f, p, N_c, N_p, Inst\#$
“01 11010”	Combo27	$SrcA = M1, SrcC = M2, Result = M3$	$N, f, p, N_c, N_p, Inst\#$
“01 11011”	Combo28	$SrcA = M1, SrcC = M2, Result = M3$	$N, f, p, N_c, N_p, Inst\#$
“01 11100”	Combo29	$SrcA = M1, SrcC = M2, Result = M3$	$N, f, p, N_c, N_p, Inst\#$
“01 11101”	Combo30	$SrcA = M1, SrcC = M2, Result = M3$	$N, f, p, N_c, N_p, Inst\#$
“01 11110”	Combo31	$SrcA = M1, SrcC = M2, Result = M3$	$N, f, p, N_c, N_p, Inst\#$
“01 11111”	Combo32	$SrcA = M1, SrcC = M2, Result = M3$	$N, f, p, N_c, N_p, Inst\#$
------------	------------------	--	--
“10 00000”	Combo1	$SrcA = M2, SrcC = M3, Result = M1$	$N, f, p, N_c, N_p, Inst\#$
“10 00001”	Combo2	$SrcA = M2, SrcC = M3, Result = M1$	$N, f, p, N_c, N_p, Inst\#$
“10 00010”	Combo3	$SrcA = M2, SrcC = M3, Result = M1$	$N, f, p, N_c, N_p, Inst\#$
“10 00011”	Combo4	$SrcA = M2, SrcC = M3, Result = M1$	$N, f, p, N_c, N_p, Inst\#$
“10 00100”	Combo5	$SrcA = M2, SrcC = M3, Result = M1$	$N, f, p, N_c, N_p, Inst\#$
“10 00101”	Combo6	$SrcA = M2, SrcC = M3, Result = M1$	$N, f, p, N_c, N_p, Inst\#$
“10 00110”	Combo7	$SrcA = M2, SrcC = M3, Result = M1$	$N, f, p, N_c, N_p, Inst\#$
“10 00111”	Combo8	$SrcA = M2, SrcC = M3, Result = M1$	$N, f, p, N_c, N_p, Inst\#$
“10 01000”	Combo9	$SrcA = M2, SrcC = M3, Result = M1$	$N, f, p, N_c, N_p, Inst\#$
“10 01001”	Combo10	$SrcA = M2, SrcC = M3, Result = M1$	$N, f, p, N_c, N_p, Inst\#$
“10 01010”	Combo11	$SrcA = M2, SrcC = M3, Result = M1$	$N, f, p, N_c, N_p, Inst\#$
“10 01011”	Combo12	$SrcA = M2, SrcC = M3, Result = M1$	$N, f, p, N_c, N_p, Inst\#$
“10 01100”	Combo13	$SrcA = M2, SrcC = M3, Result = M1$	$N, f, p, N_c, N_p, Inst\#$
“10 01101”	Combo14	$SrcA = M2, SrcC = M3, Result = M1$	$N, f, p, N_c, N_p, Inst\#$
“10 01110”	Combo15	$SrcA = M2, SrcC = M3, Result = M1$	$N, f, p, N_c, N_p, Inst\#$
“10 01111”	Combo16	$SrcA = M2, SrcC = M3, Result = M1$	$N, f, p, N_c, N_p, Inst\#$
“10 10000”	Combo17	$SrcA = M2, SrcC = M3, Result = M1$	$N, f, p, N_c, N_p, Inst\#$
“10 10001”	Combo18	$SrcA = M2, SrcC = M3, Result = M1$	$N, f, p, N_c, N_p, Inst\#$
“10 10010”	Combo19	$SrcA = M2, SrcC = M3, Result = M1$	$N, f, p, N_c, N_p, Inst\#$
“10 10011”	Combo20	$SrcA = M2, SrcC = M3, Result = M1$	$N, f, p, N_c, N_p, Inst\#$
“10 10100”	Combo21	$SrcA = M2, SrcC = M3, Result = M1$	$N, f, p, N_c, N_p, Inst\#$
“10 10101”	Combo22	$SrcA = M2, SrcC = M3, Result = M1$	$N, f, p, N_c, N_p, Inst\#$
“10 10110”	Combo23	$SrcA = M2, SrcC = M3, Result = M1$	$N, f, p, N_c, N_p, Inst\#$
“10 10111”	Combo24	$SrcA = M2, SrcC = M3, Result = M1$	$N, f, p, N_c, N_p, Inst\#$
“10 11000”	Combo25	$SrcA = M2, SrcC = M3, Result = M1$	$N, f, p, N_c, N_p, Inst\#$

“10 11001”	Combo26	$SrcA = M2, SrcC = M3, Result = M1$	$N, f, p, N_c, N_p, Inst\#$
“10 11010”	Combo27	$SrcA = M2, SrcC = M3, Result = M1$	$N, f, p, N_c, N_p, Inst\#$
“10 11011”	Combo28	$SrcA = M2, SrcC = M3, Result = M1$	$N, f, p, N_c, N_p, Inst\#$
“10 11100”	Combo29	$SrcA = M2, SrcC = M3, Result = M1$	$N, f, p, N_c, N_p, Inst\#$
“10 11101”	Combo30	$SrcA = M2, SrcC = M3, Result = M1$	$N, f, p, N_c, N_p, Inst\#$
“10 11110”	Combo31	$SrcA = M2, SrcC = M3, Result = M1$	$N, f, p, N_c, N_p, Inst\#$
“10 11111”	Combo32	$SrcA = M2, SrcC = M3, Result = M1$	$N, f, p, N_c, N_p, Inst\#$
-------------	-------------------	--	$N, f, p, N_c, N_p, Inst\#$
“11 00000”	Combo1	$SrcA = M3, SrcC = M1, Result = M2$	$N, f, p, N_c, N_p, Inst\#$
“11 00001”	Combo2	$SrcA = M3, SrcC = M1, Result = M2$	$N, f, p, N_c, N_p, Inst\#$
“11 00010”	Combo3	$SrcA = M3, SrcC = M1, Result = M2$	$N, f, p, N_c, N_p, Inst\#$
“11 00011”	Combo4	$SrcA = M3, SrcC = M1, Result = M2$	$N, f, p, N_c, N_p, Inst\#$
“11 00100”	Combo5	$SrcA = M3, SrcC = M1, Result = M2$	$N, f, p, N_c, N_p, Inst\#$
“11 00101”	Combo6	$SrcA = M3, SrcC = M1, Result = M2$	$N, f, p, N_c, N_p, Inst\#$
“11 00110”	Combo7	$SrcA = M3, SrcC = M1, Result = M2$	$N, f, p, N_c, N_p, Inst\#$
“11 00111”	Combo8	$SrcA = M3, SrcC = M1, Result = M2$	$N, f, p, N_c, N_p, Inst\#$
“11 01000”	Combo9	$SrcA = M3, SrcC = M1, Result = M2$	$N, f, p, N_c, N_p, Inst\#$
“11 01001”	Combo10	$SrcA = M3, SrcC = M1, Result = M2$	$N, f, p, N_c, N_p, Inst\#$
“11 01010”	Combo11	$SrcA = M3, SrcC = M1, Result = M2$	$N, f, p, N_c, N_p, Inst\#$
“11 01011”	Combo12	$SrcA = M3, SrcC = M1, Result = M2$	$N, f, p, N_c, N_p, Inst\#$
“11 01100”	Combo13	$SrcA = M3, SrcC = M1, Result = M2$	$N, f, p, N_c, N_p, Inst\#$
“11 01101”	Combo14	$SrcA = M3, SrcC = M1, Result = M2$	$N, f, p, N_c, N_p, Inst\#$
“11 01110”	Combo15	$SrcA = M3, SrcC = M1, Result = M2$	$N, f, p, N_c, N_p, Inst\#$
“11 01111”	Combo16	$SrcA = M3, SrcC = M1, Result = M2$	$N, f, p, N_c, N_p, Inst\#$
“11 10000”	Combo17	$SrcA = M3, SrcC = M1, Result = M2$	$N, f, p, N_c, N_p, Inst\#$
“11 10001”	Combo18	$SrcA = M3, SrcC = M1, Result = M2$	$N, f, p, N_c, N_p, Inst\#$
“11 10010”	Combo19	$SrcA = M3, SrcC = M1, Result = M2$	$N, f, p, N_c, N_p, Inst\#$
“11 10011”	Combo20	$SrcA = M3, SrcC = M1, Result = M2$	$N, f, p, N_c, N_p, Inst\#$
“11 10100”	Combo21	$SrcA = M3, SrcC = M1, Result = M2$	$N, f, p, N_c, N_p, Inst\#$
“11 10101”	Combo22	$SrcA = M3, SrcC = M1, Result = M2$	$N, f, p, N_c, N_p, Inst\#$
“11 10110”	Combo23	$SrcA = M3, SrcC = M1, Result = M2$	$N, f, p, N_c, N_p, Inst\#$
“11 10111”	Combo24	$SrcA = M3, SrcC = M1, Result = M2$	$N, f, p, N_c, N_p, Inst\#$
“11 11000”	Combo25	$SrcA = M3, SrcC = M1, Result = M2$	$N, f, p, N_c, N_p, Inst\#$
“11 11001”	Combo26	$SrcA = M3, SrcC = M1, Result = M2$	$N, f, p, N_c, N_p, Inst\#$
“11 11010”	Combo27	$SrcA = M3, SrcC = M1, Result = M2$	$N, f, p, N_c, N_p, Inst\#$

“11 11011”	Combo28	$SrcA = M3, SrcC = M1, Result = M2$	$N, f, p, N_c, N_p, Inst\#$
“11 11100”	Combo29	$SrcA = M3, SrcC = M1, Result = M2$	$N, f, p, N_c, N_p, Inst\#$
“11 11101”	Combo30	$SrcA = M3, SrcC = M1, Result = M2$	$N, f, p, N_c, N_p, Inst\#$
“11 11110”	Combo31	$SrcA = M3, SrcC = M1, Result = M2$	$N, f, p, N_c, N_p, Inst\#$
“11 11111”	Combo32	$SrcA = M3, SrcC = M1, Result = M2$	$N, f, p, N_c, N_p, Inst\#$

The above Opcodes are chosen for minimum hardware obtained in the controller shown with their sources and destinations and required operands for each instruction.

That’s all about Custom Instructions to be executed.

2.5. Reconfigurable Control Unit

2.5.1. Control Unit Design

As shown before, the data-path control signals are declared and also the instructions can be executed by it, so the functionality of control unit is to direct the data-path to execute the operation according to input instruction, and then sends end of operation as well as the operation is done.

The interface between the controller and the CPU, is based on start signal that is raised high by the CPU parallel to the needed instruction to be executed, and when it’s done, the controller raised finish signal to inform the CPU that accelerator is free to accept another instruction.

The procedure followed to design controller is the Hardwired Finite State Machine, and that’s very appropriate to the nature of data flow inside the data-path, in another words, the data is not processed in one cycle but multiple ones, so the transition between states defines that flow.

The chosen states for the FSM are as follows:

1) Wait State:
It’s used to wait until the Start signals come to inform that there’s an instruction needed to be executed.

2) Reset State:
It’s used to locally reset the whole accelerator block even if computing modules or RAMs, also it’s used to reset the destination RAM before storing any data inside if

the instruction is Move or Store instructions, as it's required to flash the data to make the RAM store it in a right manner.

3) Load/Store State:
   It's used to control the RAMs interface between it and between the main RAM, if there's a store or load from or to internal RAM, it has to be enabled for the required operation.
   Notice that the RAM operation enables have to be raised up for one clock cycle, so this state and also the following states are two clock cycle states, one cycle is to enable the module to work and another to low it down again with enabling the operation to count on with the count enable.
   That state will not come back to the wait state until the whole matrix to be loaded or stored is completely done, and that's known by the finish signals comes from the RAMs, that's also followed procedure in move and store result states.

4) Move State:
   It's used for move instruction, also for enabling the destination RAM to store and source RAM to load according to the given Opcode.

5) Conv State:
   It's also a two clock cycle states, which enables the convolution and then lower it back with maintenance of count enable high, and as the convolution is always first of the Combo if it exists, so the enabling of loading RAM is required two, to be synchronically works well.

6) Pool State:
   It's used to go through Pool states which enables it for working on a matrix and also concerns about if Pool state comes first in some combos, so these combos have to be included to enable the loading matrix in that case.

7) ReLU State:
   It's used to enable ReLU module, but as it's shown before the ReLU module works as well as its enable is high, but it has to include the cases of that the ReLU module is first of the combos to also enable the loading RAM.

8) Add State:
   It's used to enable Add module, and it's as ReLU module, but the difference is that it's required to enable the SrcC RAM whatever it's due to Opcode.

9) Store Result State:
   It's used to store the results of the Combos and its functionality is to enable the result matrix for the appropriate store operation due to the opcode and it will not back to the wait state until the storing is done.

The transition of these States as shown in Figure 29, is totally according to the Opcode of the instruction needed.

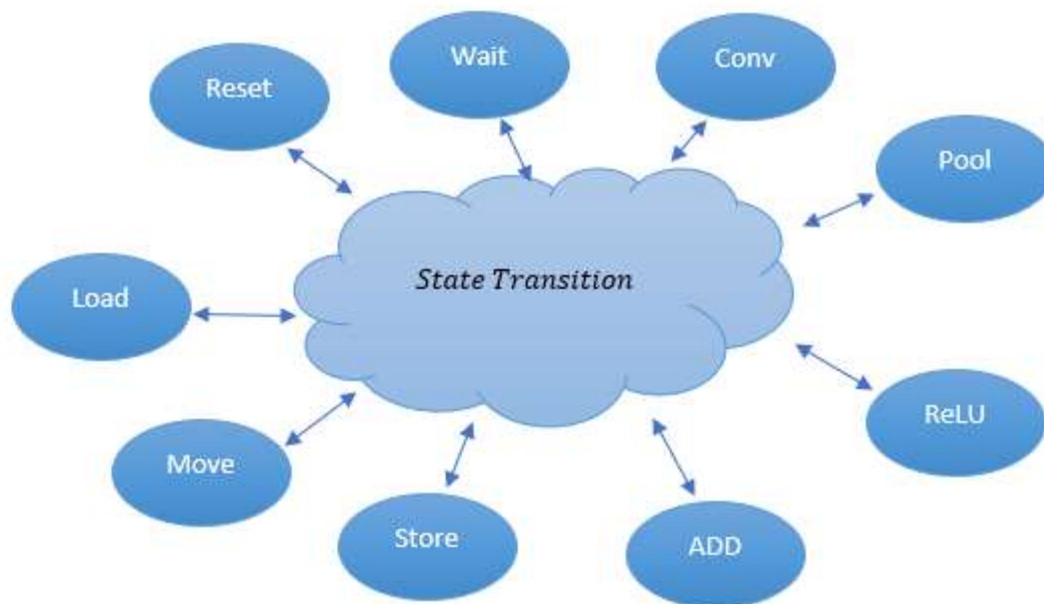


Figure 29: Controller Finite State Machine

It's required to consider that there're some modules have enables that can't be reset, for example if it's Pool state, it's not allowed to disable the ReLU module, as it may come before it, so before the determination of its enable, it has to check the Opcode.

Notice that states are two clock cycles which means that the modules don't stop its operation even if its state is over, in another words, the states are only enable the modules not also wait until there functionality is done and that's appropriate for the data flow occurrence.

Let's take examples of some instructions to see how the transition is done, for example, if Combo1 with Opcode = 0100000 is needed to be executed, the transition between states as shown in Figure 30, is that it in wait state till the start signal comes and then it goes to convolution state which enables the operation 4 in RAM1 and enables the convolution module itself and then lower enable down with maintenance of count enable, and hence go to the pool state to count it up after the setup time of the convolution to enable it to work on the first result element of the convolution and hence go to ReLU and then Add with RAM2 to work in Operation 3 to enable them when the Pool starts to display its output and disable if it stops, and finally enable the RAM3 which is the result matrix and all modules are still enabled until the finish signal of last stage comes to go to the wait state and locally reset the computing modules, that happens with setting the path between RAMs and PE through data selectors of Memory file and setting path between computing modules through the selectors of crossbar.

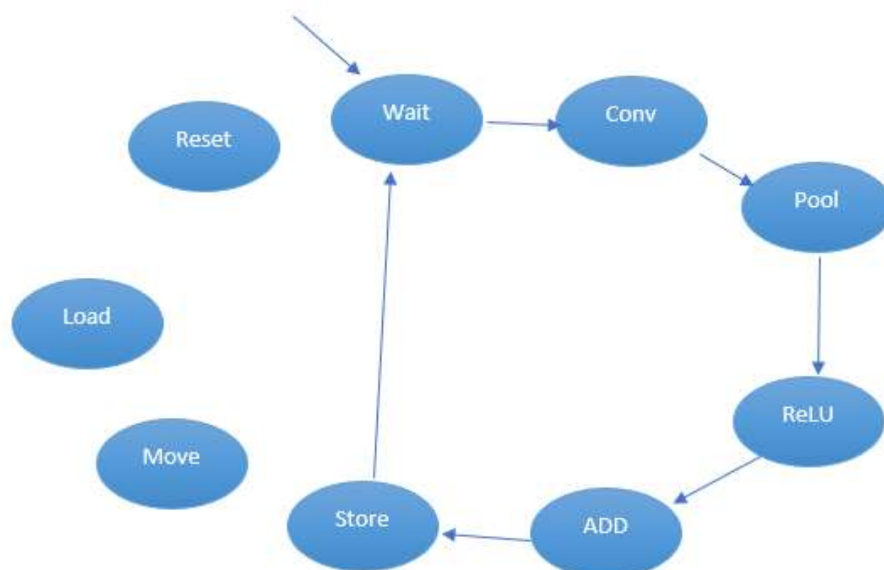


Figure 30: Execution of Combo1

Another example is in store instruction for example with Opcode = 0000101, is to store in RAM1, the transition between states is as shown in Figure 30 after moving from wait state, it goes to reset state, to reset the destination RAM, before storing in it, and hence, goes to Load/Store State to enable normal store operation for RAM1 and after RAM1 finished by informing the state, it go back to wait state again, and wait state informs CPU with end of operation signal to inform that accelerator is back to free state.

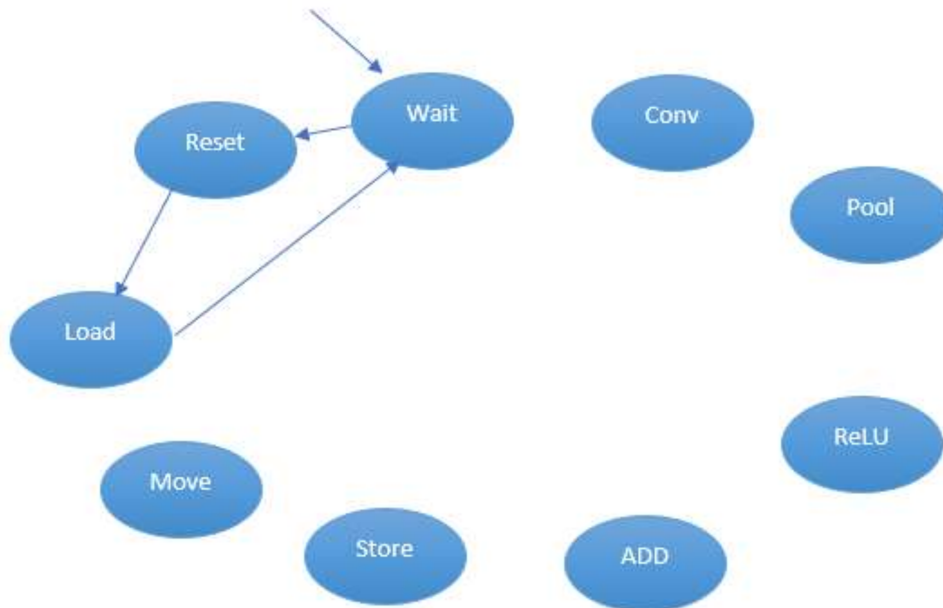


Figure 31: Executing Store instruction

And so on until the whole transition between states totally specified according to Opcode, one other thing is worth to be mentioned, for future work if there's another module needed to be hardware implemented such as sigmoid or other thing needed to CNN architecture and totally needed, it can be designed by same design procedure and add it to the states.

2.6. Generalization of Idea

Backing to the design, the most of area can be customized to used ROMs or LUTs area, regardless the Area optimization needed in ROM size reduction as mentioned above, but the good thing with ROMs is that it may be multi-input multi-output ROM which means it can serve for lots of PE modules, which encourages to use lots of data-paths served by same ROM.

As the ROM can be considered as Overhead in Area must tolerated and then the performance will increase as well as the number of data-paths increases as shown in Figure 32

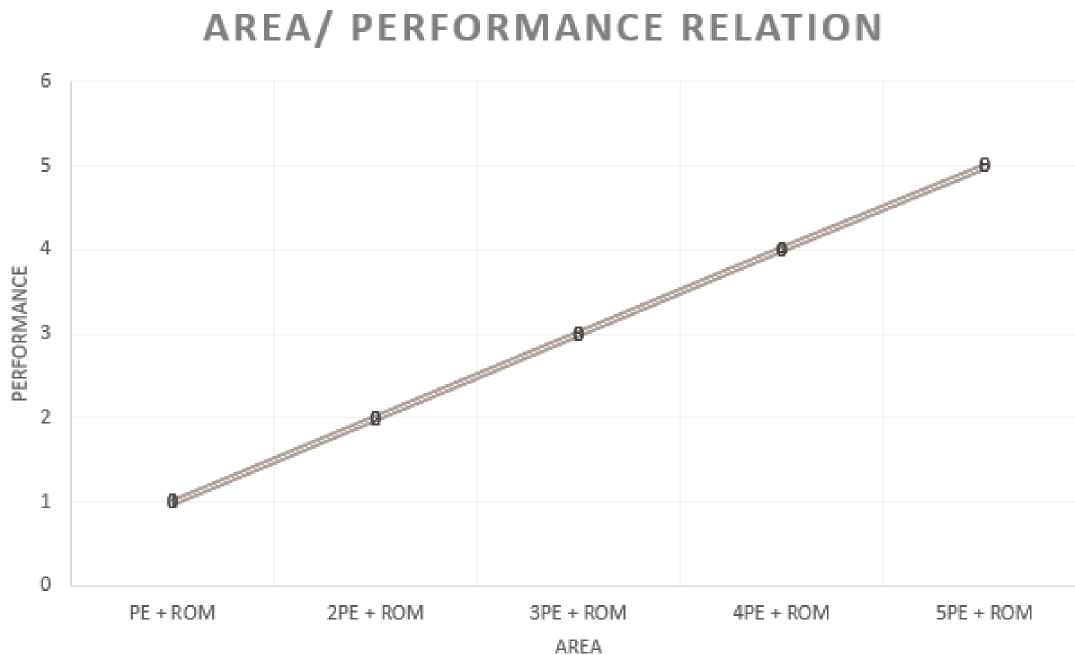


Figure 32: Area vs performance

And that is the importance of the field of instruction number, as each data-path is responsible to execute one instruction and can't be make another one although it ends the operation, the controller can be used for more than one but to handle more than signal in same time.

Chapter 3: Software path

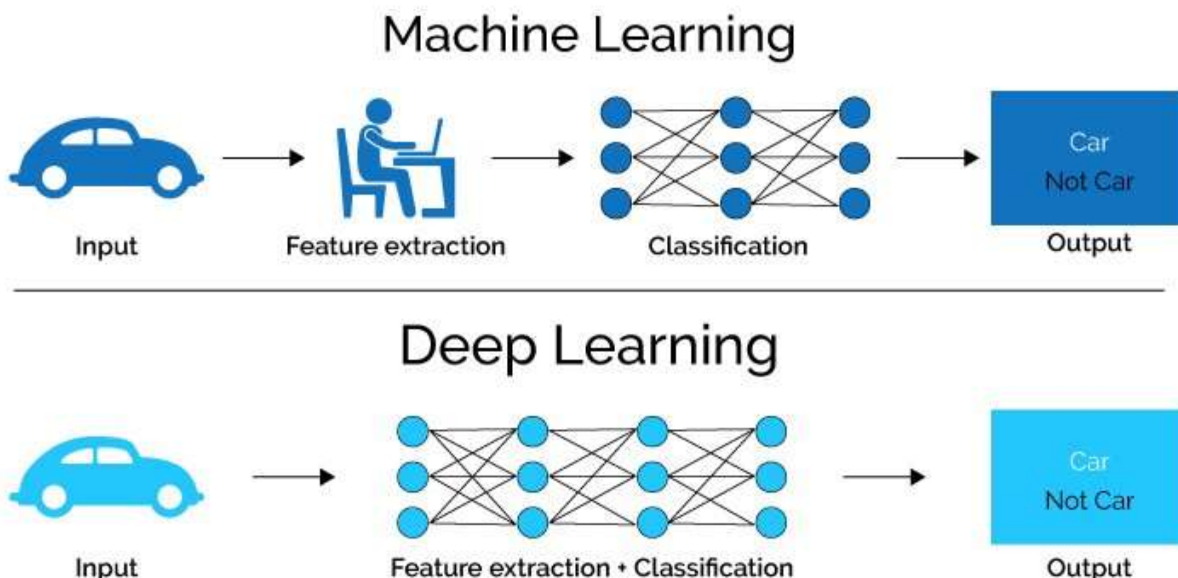
In this chapter we will talk about:

- Different between machine learning and Deep learning.
- Different types of Neural Networks in Deep Learning.
- Most popular Convolutional Neural Network architectures (CNN).
- General layers of Convolutional Neural Network architectures.
- Software Algorithm to prepare matrix before send it to coprocessor.

3.1. Machine Learning vs. Deep Learning:

Feature engineering is a key step in the model building process. It is a two-step process:

1. Feature extraction → Extract all the features required to report our problem
2. Feature selection → select the important features that improve the performance of our machine learning or deep learning model. In image classification problem. The manual extraction of features of an image requires strong subject and domain



knowledge.

Figure 33: Comparison between Machine Learning & Deep Learning

It is an extremely time-consuming process. With Deep Learning, the process of Feature Engineering can be automated.Figure 33 [15].

3.2. Different types of Neural Networks in Deep Learning

Three important types of neural networks that form the basis for most pre-trained models in deep learning:

- Artificial Neural Networks (ANN)
- Convolution Neural Networks (CNN)
- Recurrent Neural Networks (RNN)

3.2.1. Artificial Neural Network (ANN):

ANN, is a group of multiple neurons at each layer. ANN is also known as a **Feed-Forward Neural network** because inputs are processed only in the forward direction:

As shown in Figure 34 [14], ANN consists of 3 layers → Input, Hidden and Output.

- The input layer accepts the inputs.
- The hidden layer processes the inputs.
- The output layer produces the result.

ANN can be used to solve problems related to:

- Tabular data
- Image data
- Text data

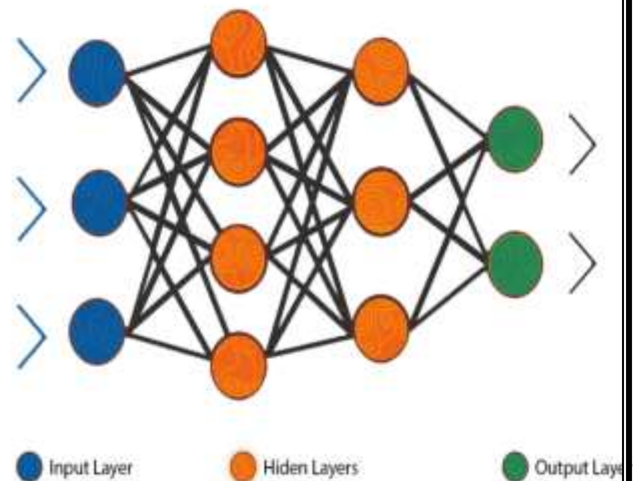


Figure 34: ANN

Challenges with Artificial Neural Network (ANN):

- While solving an image classification problem using ANN, the first step is to convert a 2-dimensional image into a 1-dimensional vector prior to training the model. Figure 35 [14]. There are two disadvantages:

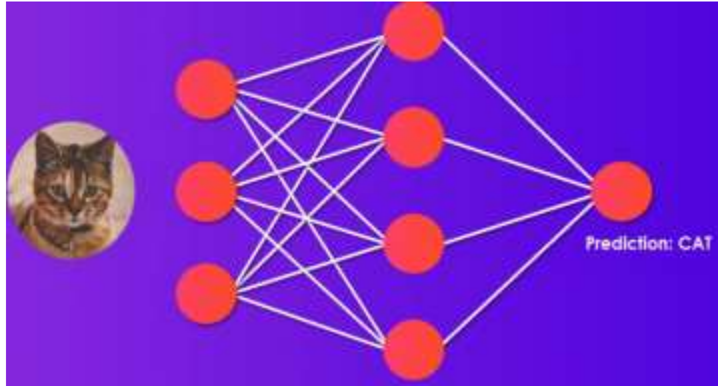


Figure 35: ANN Image classification

1. The number of trainable parameters increases drastically with an increase in the size of the image

   Ex: if the size of the image is 224*224, then the number of trainable parameters at the first hidden layer with just 4 neurons is 602,112. That's huge!

2. ANN loses the spatial features of an image. Spatial features refer to the arrangement of the pixels in an image.
   - One common problem in all these neural networks is the Vanishing and Exploding Gradient. This problem is associated with

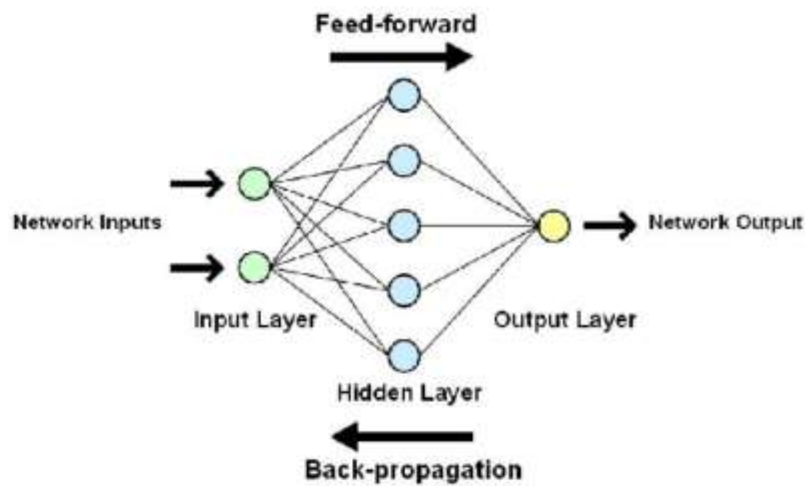


Figure 36: Backward Propagation

the backpropagation algorithm Figure 36.The weights of a neural network are updated through this backpropagation algorithm by finding the gradients:

So, in the case of a very deep neural network (network with a large number of hidden layers), the gradient vanishes or explodes as it propagates backward which leads to vanishing and exploding gradient.

- ANN cannot capture sequential information in the input data which is required for dealing with sequence data

Why not use regular ANNs for image tasks?

For small images, it might work, but for large images the number of pixels are so many leading to millions of connections between neurons, leading to intractable solutions.

3.2.2. Recurrent Neural Network (RNN):

Difference between an RNN and an ANN:

As shown in Figure 37**[12], RNN has a recurrent connection on the hidden state. This looping constraint ensures that sequential information is captured in the input data.**

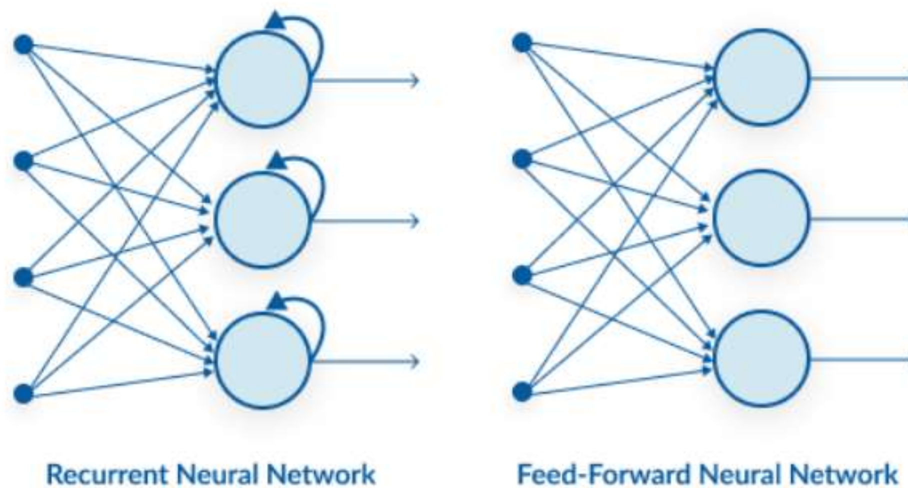


Figure 37: Difference between an RNN and an ANN

The applications of RNNs include:

- Speech recognition
- Time series prediction
- Music composition
- Machine translation

Advantages of Recurrent Neural Network (RNN)

- Sequence recognition: produce particular output pattern when a specific input sequence is seen→Application handwriting recognition.
- Temporal association: produce particular output sequence in response of a specific input sequence is seen → Application machine translation.

- RNNs share the parameters across different time steps. This is popularly known as **Parameter Sharing**. This results **in fewer parameters to train** and **decreases the computational cost** and make it **more generalized**

Disadvantage:

- Gradient vanishing and exploding problems.
- Training an RNN is a very difficult task.
- It cannot process very long sequences if using tanh or relu as an activation function.

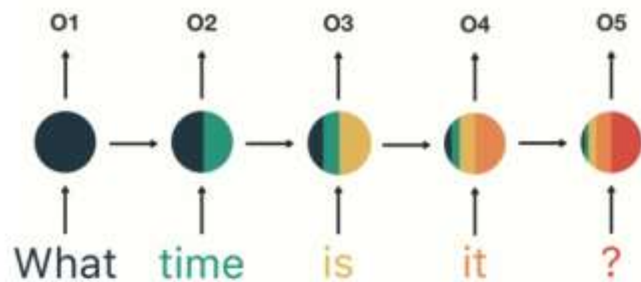


Figure 38: Many2Many Seq2Seq model

3.2.3. Convolution Neural Network (CNN):

A Convolutional Neural Network (CNN) is a type of neural network that **specializes in image recognition and computer vision tasks**

Features of CNN

- Convolution layers is the base of CNN.
- The convolution operation is one of the fundamental building blocks of a convolutional neural network
  - It used to detect patterns (edges, corners, objects….) Figure 39.

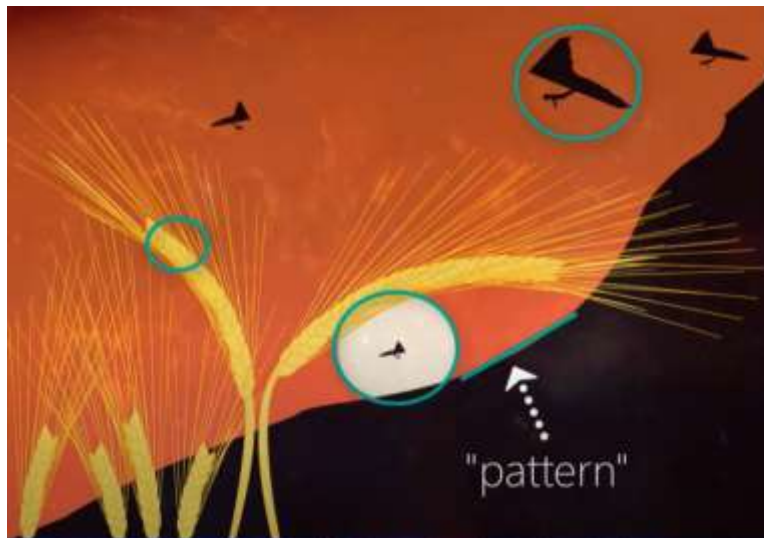


Figure 39: Detect patterns edges, corners, objects….

- We need to know how many filters and from any types we will Use in convolution layers.
- Due to this convolutional operation the network can be **much deeper** but with much **fewer parameters**.
- Due to this ability, convolutional neural networks **show very effective results in image and video recognition**, natural language processing, and recommender systems.
- Each neuron in the hidden layer is connected to a small region of the input neurons. Figure 40.

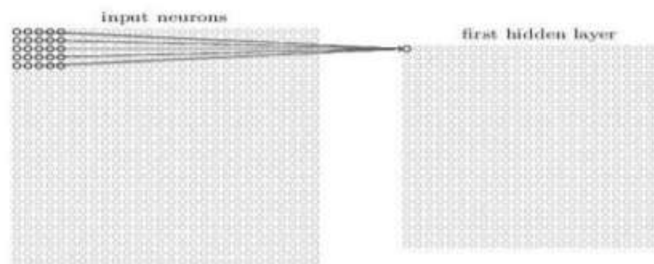


Figure 40: CNN

3.3. Most popular Convolutional Neural Network architectures (CNN):

- LeNet-5 (1998)
- AlexNet (2012)
- ZFNet (2013)
- GoogLeNet (2014)
- VGGNet (2014)
- ResNet (2015)

We choose LeNet-5 for:

- Simplicity
- The available open source codes.
- It was the lighter in the architecture among rest CNN arch.

3.3.1. LeNet-5 [17]

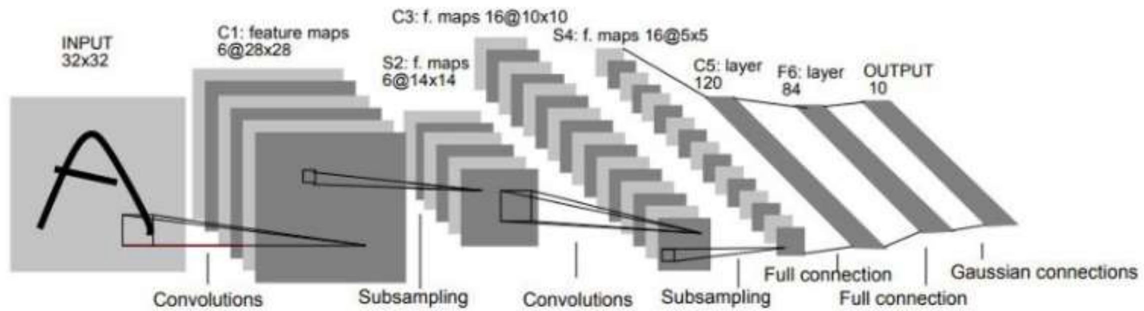


Figure 41: LeNet-5

- The dimensions of the image decreases as the number of channels increases.
- The LeNet-5 architecture: consists of two sets of convolutional and average pooling layers, followed by a flattening convolutional layer, then two fully-connected layers and finally a softmax classifier aas shown in Figure 41.

- **First Layer:**

The input for LeNet-5 is a 32×32 grayscale image which passes through the first convolutional layer (C1) Figure 42 with 6 feature maps or filters having size 5×5 and a stride of one. The image dimensions changes from 32x32x1 to 28x28x6 (32-5+1=28 and no. of filter = 6).Here, total no. of parameters are (6x (5x5+1)) = 156 parameters, where +1 indicates that a kernel has a bias. In this convolutional layer C1, there are total (156x28x28) = 122304 no. of connections.

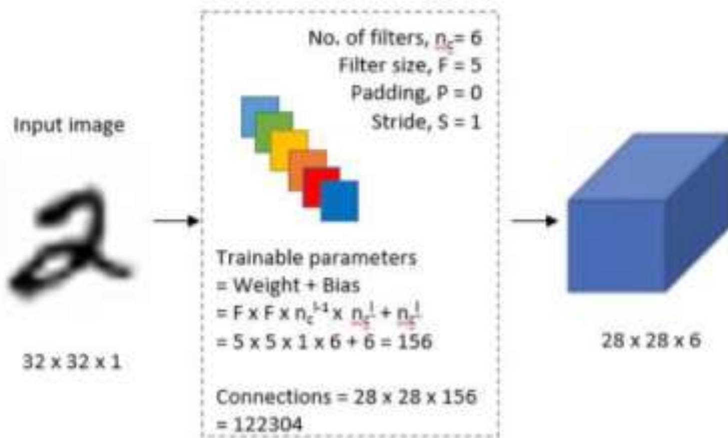


Figure 42: first layer convolution layer(C1)

- **Second Layer:**

Then the LeNet-5 applies average pooling layer or sub-sampling layer (S2) (**but now using Max pooling**) Figure 43 with a filter size 2×2 and a stride of two.

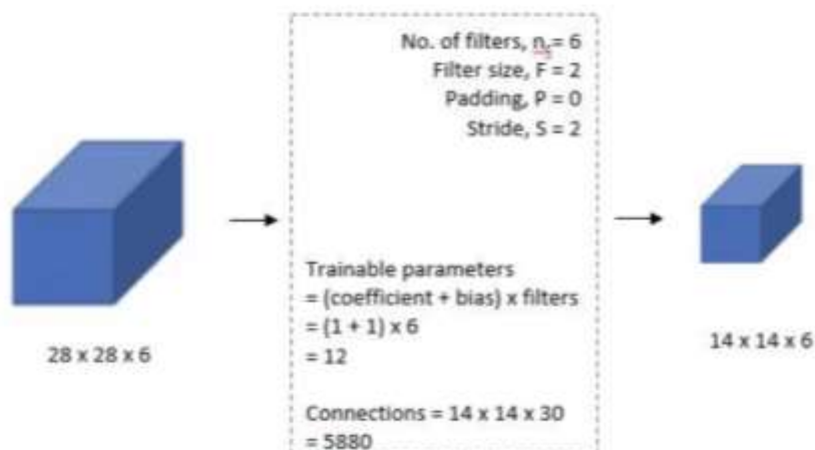


Figure 43: second layer pooling layer (S2)

The resulting image dimensions will be reduced to 14x14x6 (28/2=14). There are total **(1+1)x6** = 12 training parameters and **(5x14x14x6** **)** = 5880 connections.

- **Third Layer:**

In the third convolutional layer (C3) Figure 44, there are 16 filters having size 5x5 and stride is 1. Here image size will be 10x10x16 **[(((14-5)/1)+1=10)].** So, here total no. of parameters are **(5x5x6x10+16) =1516**. The image size is 10 x10, so there are 151600 connections

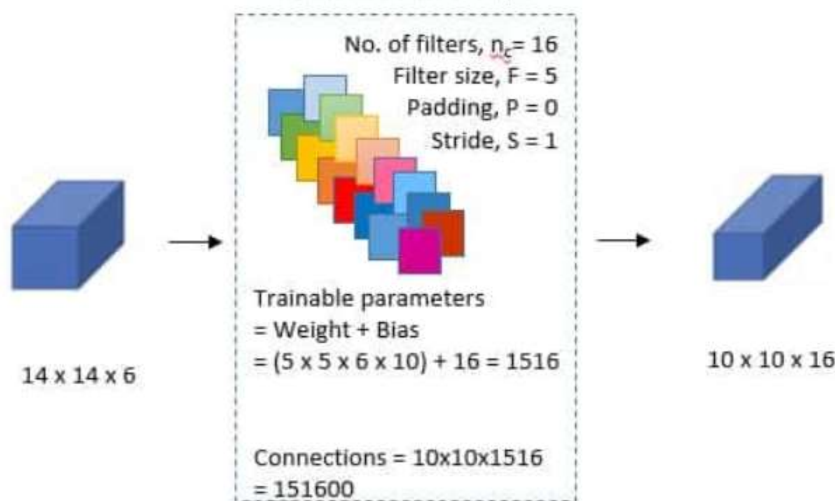


Figure 44: third layer convolution layer (C3)

- **Fourth Layer:**

In the fourth layer (S4) Figure 45 after applying avg. pooling (Now Max. pooling) we get the image size of 5x5x16 where filter and stride no. is 2. Here, we have 16 filters and 25 layers. Here no. of nodes will be 400 (5x5x16=400).This layer has a total of **(2x16)** 32 training parameters of, **(5x5x5x16)** = **2000** connections.

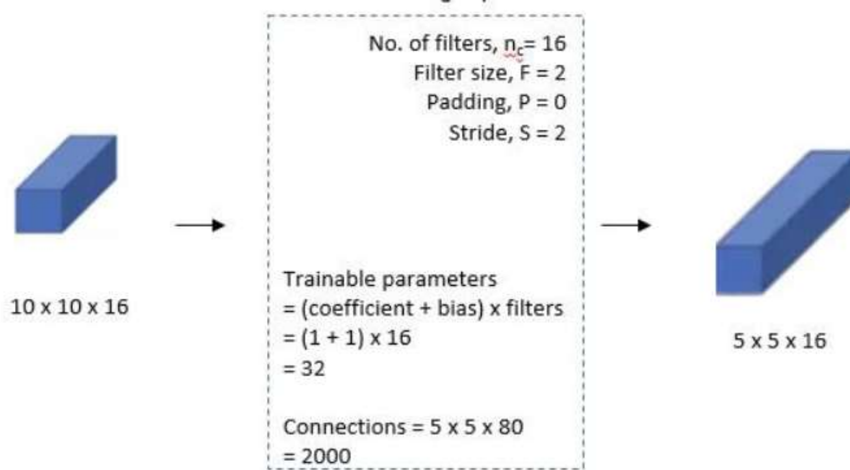


Figure 45: fourth layer pooling

- **Fifth Layer:**

In the fifth convolutional layer (F5) Figure 46 we are flattening the image with 120 feature maps. Each of the 120 units is connected to all the 400 nodes (5x5x16) of the previous layer .The size of the image formed after convolution is 1x1. So, here total no. of connections are **(5x5x16 + 1) x120 = 48120**.

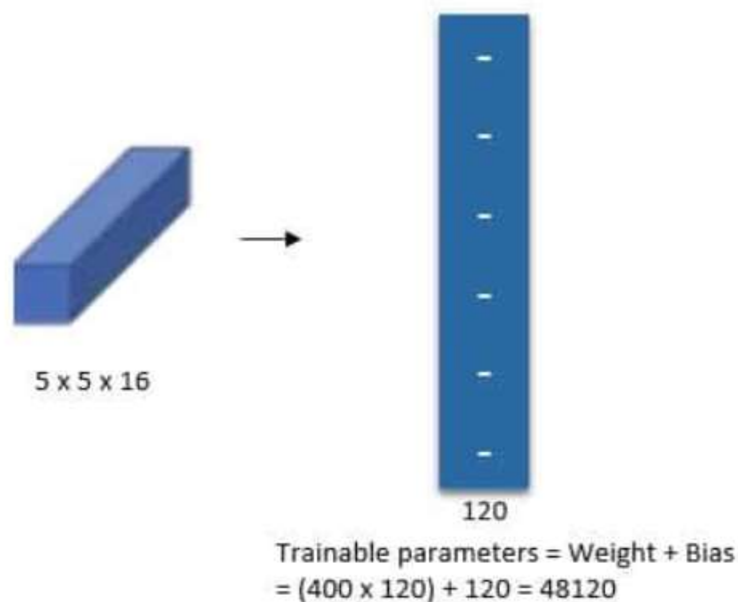


Figure 46: fully connected layer (F5)

- **Sixth Layer:**

The sixth layer is a fully connected layer (F6) Figure 47 with 84 units. Input: 120-dimensional vector. Calculate the dot product between the input vector and the weight vector, plus an offset, and the result is output through the sigmoid function. The training parameters and number of connections for this layer are (120 + 1) x84 = 10164.

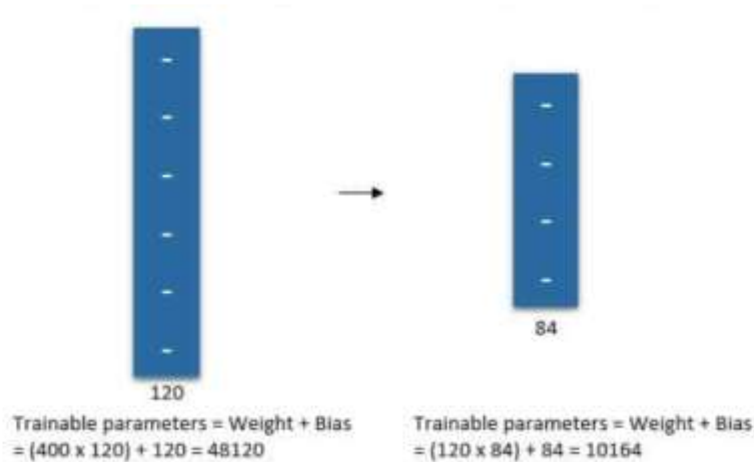


Figure 47: fully connected layer (F6)

The output layer (F6) is also a fully connected layer, with a total of 10 nodes, which respectively represent the numbers 0 to 9, and if the value of node i is 0, the result of network recognition is the number i. A radial basis function (RBF) network connection is used. This layer has 84x10 = 840 parameters and connections as shown in Figure 48.

F6: Fully Connected Layer

Output

Figure 48: output layer

- ***Summary of LeNet-5 Architecture***

Layer		Feature Map	Size	Kernel Size	Stride	Activation
Input	Image	1	32x32	-	-	-
1	Convolution	6	28x28	5x5	1	tanh
2	Average Pooling	6	14x14	2x2	2	tanh
3	Convolution	16	10x10	5x5	1	tanh
4	Average Pooling	16	5x5	2x2	2	tanh
5	Convolution	120	1x1	5x5	1	tanh
6	FC	-	84	-	-	tanh
Output	FC	-	10	-	-	softmax

Figure 49: LeNet-5 table

3.4. General layers of Convolutional Neural Network architectures:

In this section will talk about all layers that in CNN and show how we can implement all these layers in C++ to be suitable with coprocessor.[20]

3.4.1. Convolution

The first layer in a CNN is a Convolutional Layer. The convolutional layer receives N feature maps as input. Each input feature map is convolved by a shifting window with K x K kernel (filter) to generate one element in one output feature map. The stride of the shifting window is S, which is normally smaller than K. A total of M output feature maps will form the set of input feature maps for the next convolutional layer. By stacking a number of convolutional layers, the network hierarchically learns high-level features of the image as Convolution is very strong and effective to extract features from images.

- Figure 50, shows a 2D example where Yellow Square is filter weights and the green one is input [18].

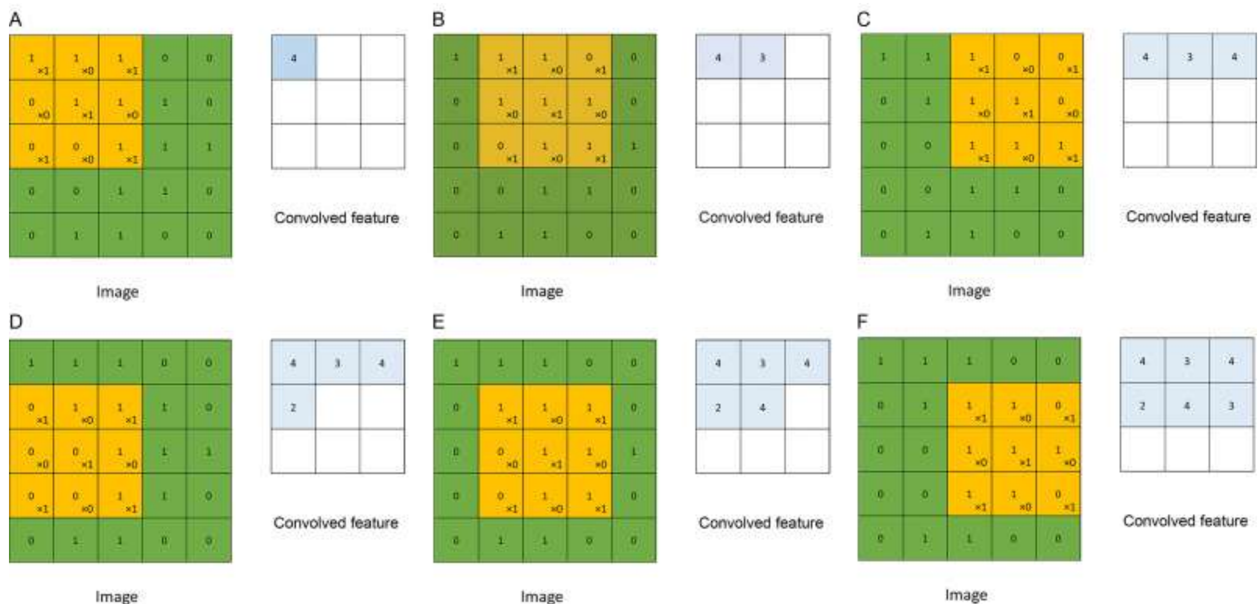


Figure 50: 2D convolution example

- Represent convolution layer using C++:

```
float getConvolution(float *image , float *kernel ,int i ,int j,int n , int f ){
    float sum = 0.0;
     for(int x = 0;x< f;x++){
        for(int y = 0;y< f;y++){
            sum += *(image + (x + i)*n + (y + j))*(*(kernel + x*f + y)) ;
        }
     }

    return sum ;
}
```

3.4.2. Max pooling

- **Max Pooling** is a pooling operation that calculates the maximum value for patches of a feature map, and uses it to create a downsampled (pooled) feature map. It is usually used after a convolutional layer. It adds a small amount of translation invariance - meaning translating the image by a small amount does not significantly affect the values of most pooled outputs.
- **EX:** Let's say we have a 4x4 matrix representing our initial input. Let's say, as well, that we have a 2x2 filter that we'll run over our input. We'll have a **stride** of 2 (meaning the (dx, dy) for stepping over our input will be (2, 2)) and won't overlap regions.

For each of the regions represented by the filter, we will take the **max** of that region and create a new, output matrix where each element is the max of a region in the original input as shown in Figure 51.[19]

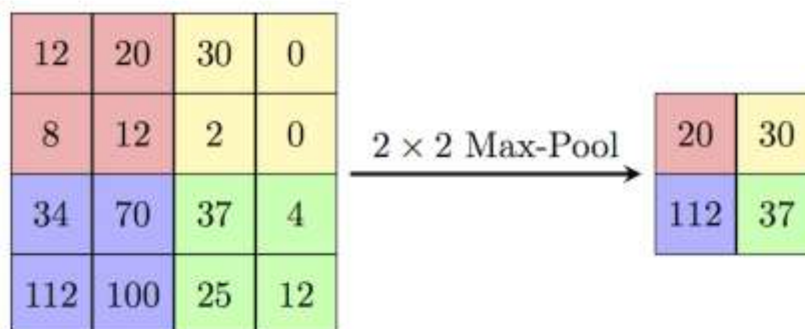


Figure 51: Max pool sample2

- Real life example: as shown in Figure 52

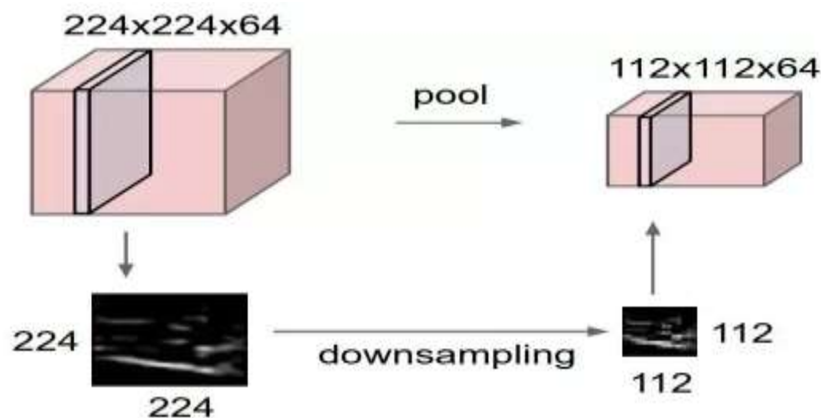


Figure 52: Max pool real life example

- Represent Max pooling layer using C++:

```
float max_pool(float *arr, int n, int f, int s, int i, int j){

    float max = -1.0;
    int a,index_i,index_j;
    a = i*s;

    for(int x=0; x<f; x++){
        for(int y=0;y<f; y++){
            index_i = i*s + x;
            index_j = j*s + y;
            if(index_i< n && index_j< n){
                if( *(arr + (a+x)*n + j*s + y) > max){
                    max = *(arr + (a+x)*n + j*s + y);
                }
            }
        }
    }

    return max;
}
```

3.4.3. Average pooling

- **Average Pooling** is a pooling operation that calculates the average value for patches of a feature map, and uses it to create a downsampled (pooled) feature map. It is usually used after a convolutional layer. It adds a small amount of translation invariance - meaning translating the image by a small amount does not significantly affect the values of most pooled outputs. It extracts features more smoothly than Max Pooling, whereas max pooling extracts more pronounced features like edges.

- **Ex:** Figure 53 shows an example of average pooling with a 2x2 pixel filter size from 4x4 pixel input. The value taken is the average value of the filter size.

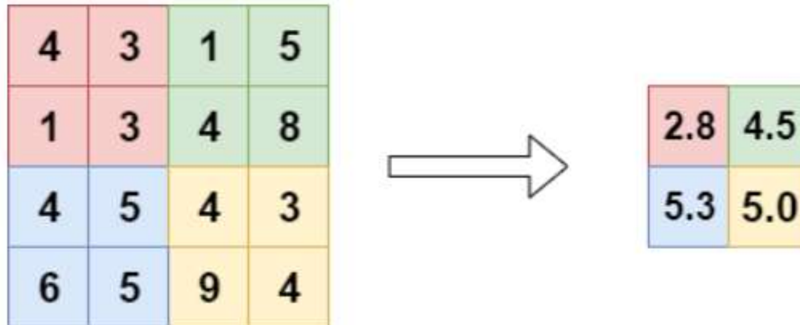


Figure 53: average pooling example

- Represent Average pooling layer using C++:

```
float avg_pool(float *arr, int n, int f, int s, int i, int j){

    float sum = 0.0,avg;
    int a,b,index_i,index_j;
    a = i*s;
    b = j*s;

    for(int x=0; x<f; x++){
        for(int y=0;y<f; y++){
            index_i = i*s + x;
            index_j = j*s + y;
            if(index_i< n && index_j< n){
                sum = sum + *(arr + (a+x)*n + j*s + y);
            }
        }
    }

    avg = sum/(f*f);
    return avg;
}
```

3.4.4. Activation functions:

After each convolutional layer, it is convention to apply a nonlinear layer (or activation layer) immediately afterward. The purpose of this layer is to introduce nonlinearity to a system that basically has just been computing linear operations during the convolutional layers (just element wise multiplications and summations).

2.4.4.1 Rectified Linear Unit (ReLU)

- ReLu has a linear relationship with the dependent variable. it avoids and rectifies vanishing gradient problem. Almost all deep learning Models use ReLu nowadays. Because the network is able to train a lot faster (because of the computational efficiency) without making a significant difference to the accuracy. But its limitation is that it should only be used within Hidden layers of a Neural Network Model.
- RELU layer applies the function $R(z) = \max(0, z)$ to all of the values in the input volume as shown in Figure 55. In basic terms, this layer just changes all the negative activations to zero. This layer increases the nonlinear properties of the model and the overall network without affecting the receptive fields of the convolutional layer.[22]
- Example: for the RELU function is shown in Figure 54 below

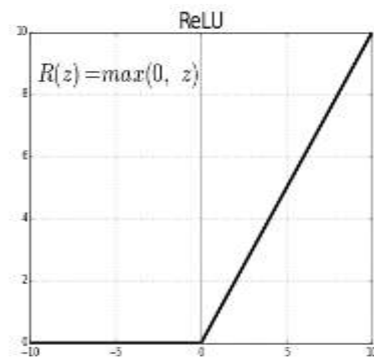


Figure 55: ReLU Activation function

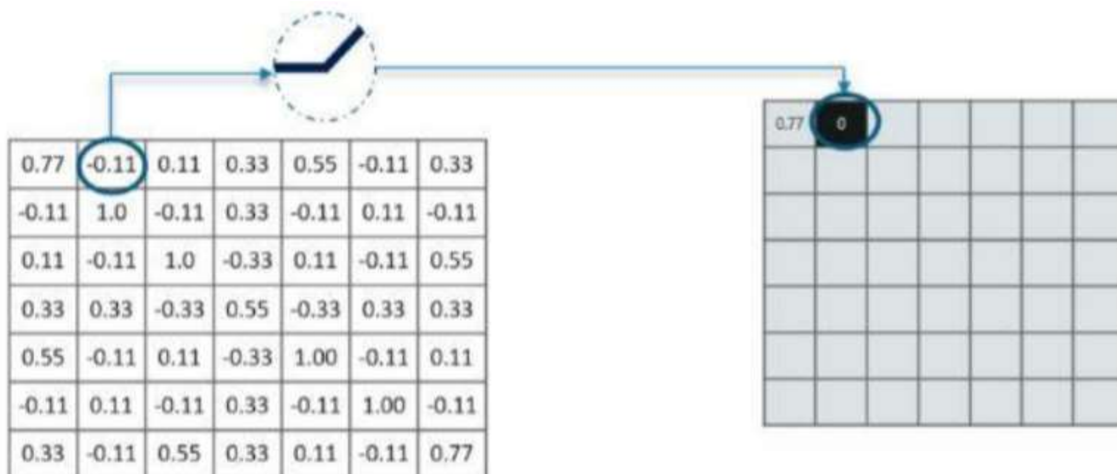


Figure 54: RelU in process

- Represent ReLU function using C++:

```
float* relu(float* arr,int n){

    float* output = (float*) malloc(sizeof(float)*n*n) ;
    float* temp = arr ;
    float entry ;
    for(int i = 0 ; i < n ; i++){

        for(int j = 0 ; j < n ; j++){
            entry = *(temp + i*n + j);
           if(entry > 0){
                *(output + i*n + j) = entry ;
           }
           else{
                *(output + i*n + j) = 0 ;
           }
        }
    }

    return output ;
}
```

2.4.4.2 Sigmoid function

- Sigmoid function is normally used to refer specifically to the logistic function, also called the logistic sigmoid function.[11]
- All sigmoid functions have the property that they map the entire number line into a small range such as between 0 and 1, or -1 and 1, so one use of a sigmoid function is to convert a real value into one that can be interpreted as a probability.
- One of the most widely used sigmoid functions is the logistic function, which maps any real value to the range (0, 1).
- It applies the function $R(z) = \frac{1}{1+e^{-z}}$ as shown in Figure 56.

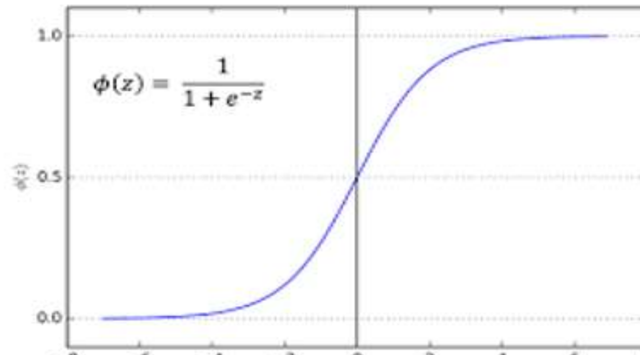


Figure 56: sigmoid function

- Represent sigmoid function using C++:

```
float sigmoid(float f){

    f = -1.0*f;
    f = exp(f);
    f = 1/(1+f);

    return f;
}
```

2.4.4.3 Softmax function

- The softmax function is a function that turns a vector of K real values into a vector of K real values that sum to 1.
- The input values can be positive, negative, zero, or greater than one, but the softmax transforms them into values between 0 and 1, so that they can be interpreted as probabilities. If one of the inputs is small or negative, the softmax turns it into a small probability, and if an input is large, then it turns it into a large probability, but it will always remain between 0 and 1 as shown in Figure 57.[22]

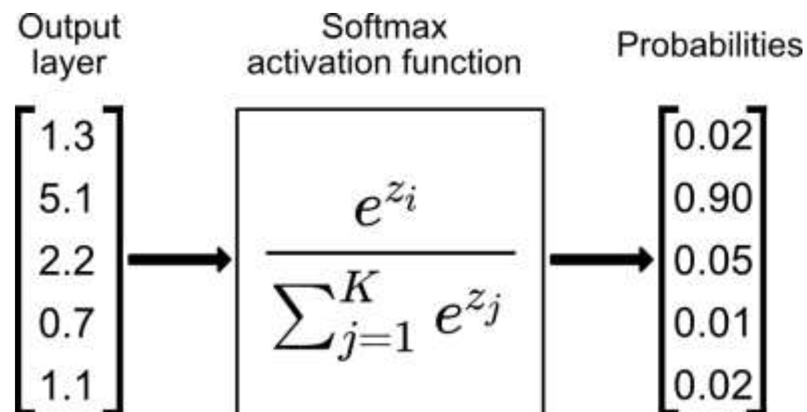


Figure 57: Softmax function

- Represent softmax function using C++:

```cpp
float softmax(float f, float sum){
    f = exp(f);
    f = f/sum;

    return f;
}
```

2.4.4.4 Tanh function

- Tanh is like logistic sigmoid but better. The range of the tanh function is from (-1 to 1). Tanh is also sigmoidal (s - shaped).
- The advantage is that the negative inputs will be mapped strongly negative and the zero inputs will be mapped near zero in the tanh graph as shown in Figure 58.[22]
- It applies the function $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x}-1}{e^{2x}+1}$

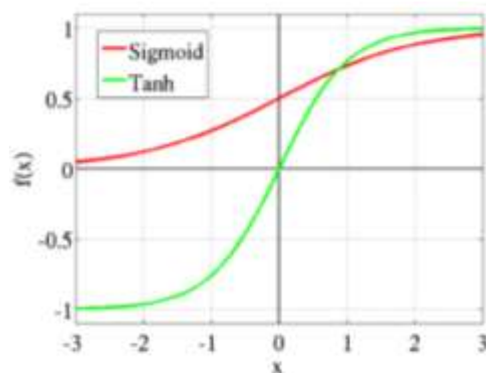


Figure 58: Tanh vs. sigmoid function

- Represent tanh function using C++:

```cpp
float* tanh(float* arr,int n){

    float* output = (float*) malloc(sizeof(float)*n*n);
    float* temp = arr;
    float entry,result;
    for(int i = 0;i< n;i++){
        for(int j = 0;j< n;j++){
            entry = *(temp + i*n + j);
            result = exp(2*entry);
            *(output + i*n + j) = (result-1)/(result + 1);
        }
    }

    return output;
}
```

3.5 Software Algorithm

After implement general layer of convolution neural network we use it to implement LeNet-5.Than testing a simple Lenet-5 on RISCV simulators: To know which layer take larger time and need to be implemented using hardware. From simulation we find out that the convolution layer is the most layer that takes time so it need to be implemented hardware. We need to prepare matrix before send it to coprocessor because coprocessor is fixed size can't deal with different sizes from matrices.

- Two cases to prepare matrix before send it to coprocessors:
    i) If Source matrix size multiple form Coprocessors matrix size.
    ii) If Source matrix size not multiple form Coprocessors matrix size.

- Depend on three things:
    1. Source matrix size
    2. Coprocessors matrix size
    3. Filter matrix size

i) Source matrix size multiple form Coprocessors matrix size:
In this case we can divide source matrix into matrices that have size equal to Coprocessor's matrix size and send them to coprocessor but still have matrices in between should be sent so we also take matrix in between as shown in Figure 59.

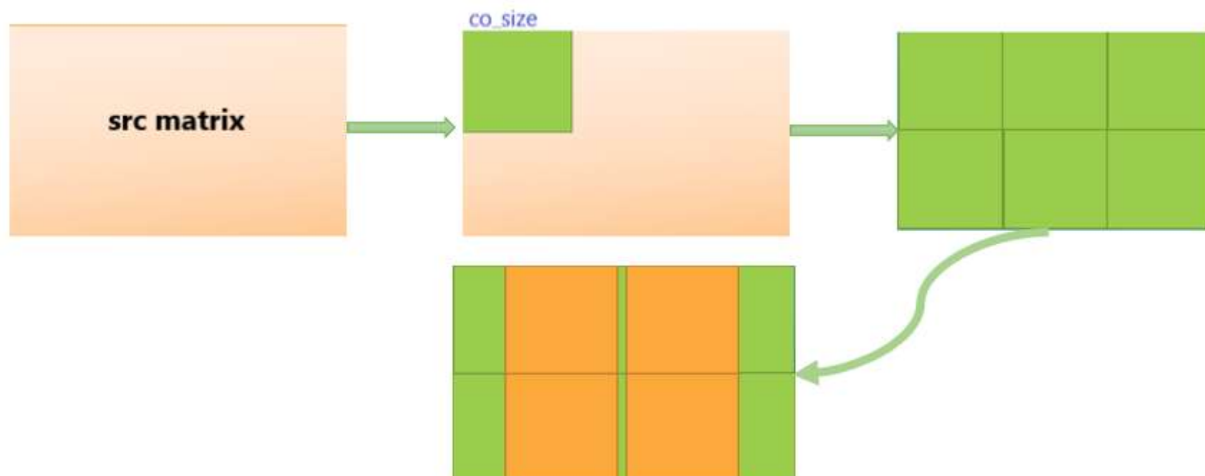


Figure 59: Divide source matrix when its size is multiple from co-size

ii) If Source matrix size not multiple form Coprocessors matrix size.

In this case we will divide src matrix just exact as first case but we have parts from src matrix not sending so we divide this parts and send them as shown in Figure 60

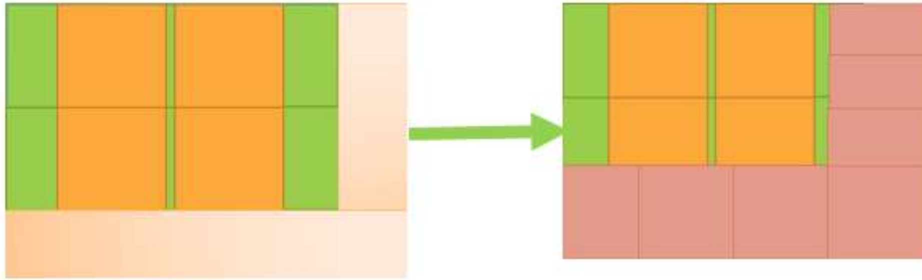


Figure 60: Divide src matrix when its size not multiple from co-size

Chapter 4: Verification

What is the advantage of UVM based testbench over System Verilog based testbench?

A simple analogy:

System Verilog based TB: You are asked to build a house from scratch, not much of tools are provided to you to build the house. You will build the tools first and then start building the house. You probably would have put in lot of effort to make the whole thing work. But you have the flexibility to use those tools to build other houses or develop new tools depending on your need. You will struggle to build larger houses though, not that it is impossible.

UVM based TB: You are again asked to build a similar house from scratch, and you are provided with not just the basic tools but some are quite sophisticated too.. You effortlessly build the house. You can build larger and more sophisticated houses but they shall lack the "human touch" that SV based TBs could offer.

Note that both the houses could be equally good in terms of quality and strength but the effort you put in for the latter house is very minimal. This, however, is subjective and many might still prefer SV based TB since they have more controllability over it and that they have the flexibility to develop necessary verification infrastructure/ libraries based on their taste and requirement. UVM based TB may make life easier but at the cost of not providing developers independence to implement their own solutions in some aspects. This is, however, debatable since UVM is open-source and developers are free to play with base class library.

In the following section, there is a comparison between both Verilog, Systemverilog and UVM as testbench languages [23].

Verilog	Systemverilog
• It's an incremental approach. • Produces immediate results since little infrastructure is needed. • When the design complexity doubles, it takes twice as long to complete or requires twice as many people. • Another methodology is needed to find bugs faster to reach the goal of 100% coverage	• Automatic functions which provide flexibility. • Huge scope of constrained randomization. • functional coverage hit more uncovered areas • Assertion increase observability of the DUT

Figure 61: comparison between verilog and systemverilog as test bench language

Here are Some of the benefits of using UVM are

1- Modularity and Reusability – The methodology is designed as modular components (Driver, Sequencer, Agents , env etc) which enables reusing components across unit level to multi-unit or chip level verification as well as across projects.

2- Separating Tests from Testbenches – Tests in terms of stimulus/sequencers are kept separate from the actual testbench hierarchy and hence there can be reuse of stimulus across different units or across projects.

3- Simulator independent – The base class library and the methodology is supported by all simulators and hence there is no dependence on any specific simulator.

4- Sequence methodology gives good control on stimulus generation. There are several ways in which sequences can be developed which includes randomization, layered sequences, virtual sequences etc which provides a good control and rich stimulus generation capability.

5- Config mechanisms simplify configuration of objects with deep hierarchy. The configuration mechanism helps in easily configuring different testbench

components based on which verification environment uses it and without worrying about how deep any component is in testbench hierarchy.

6- Factory mechanisms simplifies modification of components easily. Creating each components using factory enables them to be overridden in different tests or environments without changing underlying code base.

For all this reasons we choose to build our testbench using **UVM** methodology, before understanding UVM, we need to understand verification.

Right now, we have a DUT and we will have to interact with it in order to test its functionality, so we need to stimulate it. To achieve this, we will need a block that generates sequences of bits to be transmitted to the DUT, this block is going to be named sequencer.

Usually sequencers are unaware of the communication bus, they are responsible for generating generic sequences of data and they pass that data to another block that takes care of the communication with the DUT. This block will be the driver.

While the driver maintains activity with the DUT by feeding it data generated from the sequencers, it doesn't do any validation of the responses to the stimuli. We need another block that listens to the communication between the driver and the DUT and evaluates the responses from the DUT. This block is the monitor.

Monitors sample the inputs and the outputs of the DUT, they try to make a prediction of the expected result and send the prediction and result of the DUT to another block, the scoreboard, in order to be compared and evaluated.

All these blocks constitute a typical system used for verification and it's the same structure used for UVM testbenches.

You can find a representation of a similar environment in Figure 61.

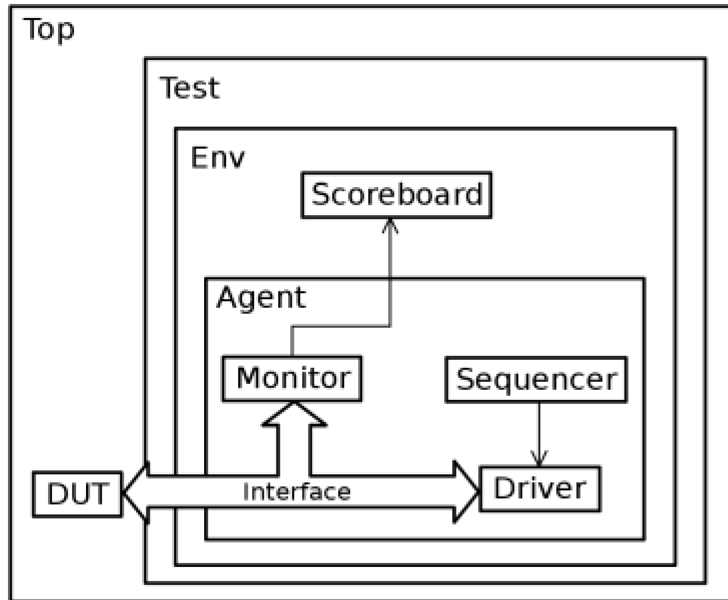


Figure 62: Typical UVM testbench

Usually, sequencers, drivers and monitors compose an agent. An agent and a scoreboard compose an environment. All these blocks are controlled by a greater block denominated of test. The test block controls all the blocks and sub blocks of the testbench. This means that just by changing a few lines of code, we could add, remove and override blocks in our testbench and build different environments without rewriting the whole test.

To illustrate the advantage of this feature, let's imagine a situation where we are testing another DUT that uses SPI for communication. If, by any chance, we want to test a similar DUT but with I2C instead, we would just need to add a monitor and a driver for I2C and override the existing SPI blocks, the sequencer and the scoreboard could reused just fine.

UVM **overview**

4.1 What is UVM?

UVM is a methodology for functional verification using SystemVerilog, complete with a supporting library of SystemVerilog code. The letters UVM stand for the Universal Verification Methodology. UVM was created by Accellera based on the OVM (Open Verification Methodology) version 2.1.1. The roots of these methodologies lie in the application of the languages IEEE 1800™ SystemVerilog, IEEE 1666™ SystemC, and IEEE 1647™ e.

UVM is a methodology for the functional verification of digital hardware, primarily using simulation. The hardware or system to be verified would typically be described using Verilog, SystemVerilog, VHDL or SystemC at any appropriate abstraction level. This could be behavioral, register transfer level, or gate level. UVM is explicitly simulation-oriented, but UVM can also be used alongside assertion-based verification, hardware acceleration or emulation.

If you currently run RTL simulations in Verilog or VHDL, you can think of UVM as replacing whatever framework and coding style you use for your testbenches. But UVM testbenches are more than traditional HDL testbenches, which might wiggle a few pins on the design-under-test (DUT) and rely on the designer to inspect a waveform diagram to verify correct operation. UVM testbenches are complete verification environments composed of reusable verification components, and used as part of an overarching methodology of constrained random, coverage-driven, verification.

4.1.1 System Verilog in UVM [40]

UVM is built with SystemVerilog's Object Oriented Programming constructs based on aggregation or composition and inheritance concepts. Aggregation or composition means that a class has a reference to another class, in other words an object container relationship and inheritance concept describes the relationship between base classes and extended ones you can say it shows the UVM hierarchy.

4.2 UVM Methodology

The verification methodology has many goals, First goal is reusability as UVM facilitates the construction of verification environments and tests, both by providing reusable machinery in the form of a library of SystemVerilog classes, and also by providing a set of guidelines for best practice when using SystemVerilog for verification.

Verification productivity can be enhanced by reusing verification components, and this is an important objective of UVM. Verification reuse is enabled by having a modular verification environment where each component has clearly defined responsibilities, by allowing flexibility in the way in which components are configured and used, by having a mechanism to allow imported components to be customized to the application at hand, and by having well-defined coding guidelines to ensure consistency.

The architecture of UVM has been designed to encourage modular and layered verification environments, where verification components at all layers can be reused in different environments. Low-level driver and monitor components can be reused across multiple designs-under-test. The whole verification environment can be reused by multiple tests and configured top-down by those tests. Finally, test scenarios can be reused from application to application. This degree of reuse is enabled by having UVM verification components able to be configured in a very flexible way without modification to their source code. This flexibility is built into the UVM class library.[37]

The second goal is interoperability needed for tools from multiple vendors and with multiple types of tools as well. For example an engineer develops a testbench with one simulator and wants to make sure that your code behaves correctly in other simulators as the system Verilog is very large and most of time many vendors contributes to implement the features. UVM allows each vendor to focus on a common subset of systemverilog feature so that a design simulates consistently, UVM also allows verification intellectual property (VIP) models in your testbench that's why In-house verification code of components that make up the design and commercial verification code for of the shelf components can be used. There is no need to write the code from scratch, moreover the UVM separates stimulus generation from delivering it to the DUT, In UVM, classes that describe the transaction are different from the classes that describe how components are connected together that allows several engineers to generate stimulus and develop the testbench in parallel, UVM code also is written in a maintainable manner so that it is easy to read or modify according to your needs in the project.

4.3 UVM Topology

Using UVM we are trying to build a verification environment that can be used over and over for many test, the main idea is to separate the stimulus from the test bench ,So that the stimulus will be responsible for defining what exactly will happen for this particular simulation run , while the testbench will be responsible for defining all components that are needed to interact with DUT. The test class is to build and configure the environment and to generate stimulus we can also determine how many times are we going to run a particular stimulus and what type of transactions are we going to generate. The environment class instantiate the components for driving transactions into the DUT, Monitoring values read for the DUT and checking the result as well. The DUT communicates with the testbench through a systemverilog interface that has different methods which you can call to drive transactions to the design and read them out of it , So this structure rarely changes. All of these of these components are constructed under a

base class called UVM root which constructs your testbench and start the simulation phases.

The DUT communicates with the test environment through a systemverilog interfacre , So every interface inside your design needs an agent that encapsulates everything needed for counicating with this interface , first drive transactions into DUT so the driver send transactions to the interface which then wiggles to DUT pins. A sequencer components connected to the driver sends the transactions to the driver then the driver sends the transactions to the DUT through the interface, To verify the result the monitor watches the pin wiggles through the interface coverts those pin wiggles into transactions and sends them to the scoreboard or coverage collector for checking the values and verify results, This is done an analysis port which is like a sysytemverilog mailbox. A configuration object is a class with configuration values like the virtual interface. See 63

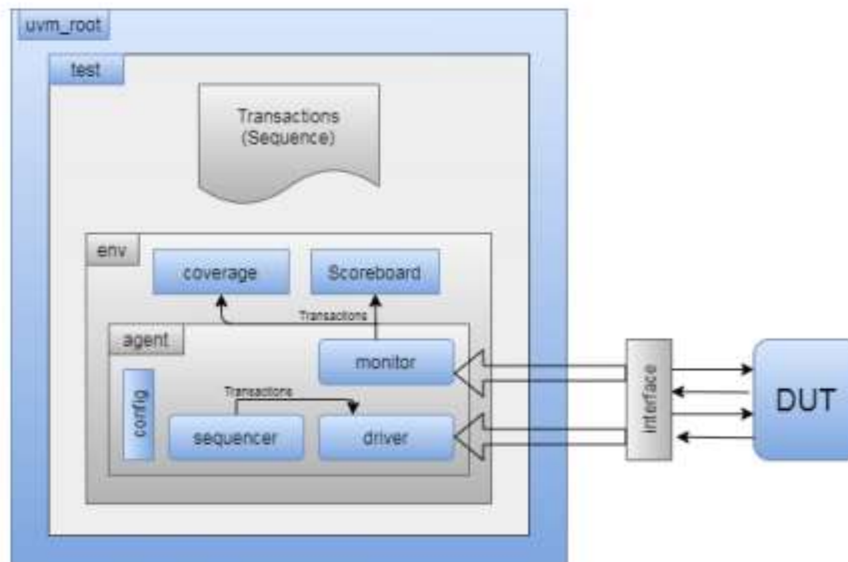


Figure 63: UVM classes , connections between testbench and DUT

4.3.1 UVM factory

Previously, creating a dynamically different type of objects required modifying source code which contradicts the reusability concept of UVM. UVM factory is a mechanism introduced by the UVM to improve the flexibility and scalability of the testbench by allowing the user to substitute an existing class object by any of its inherited child class objects. Factory is a critical aspect which is introduced in the UVM that builds everything in the UVM environment like dynamically adaptable testbenches, which are tests created and compiled at run time . Therefore, factory requires that all the classes to be registered with the factory with macros like ‘uvm component utils and ‘uvm object utils macros.

4.4 Class Hierarchy

The UVM package contains a class library that provides a set of base classes which can be extended by users as required , UVM object is the base class for all UVM data and hierarchical classes . Its role is to define a set of methods for common operation such as (create , copy , compare, print). there are two group class that are inherited from uvm object the first one is uvm component that is used to build the testbench topology also there classes are dynamically created they exist for the entire simulation , the classes has additional characteristics like being in fixed location inside uvm topology and having methods that are called in a fixed order to build and connect the testbench run the test and report the results , The second droup is transaction classes the stimulus is described into the design by extending uvm base classes a single transaction is a sequence item and multiple items form a sequence . Transactions are transient object they are created and destroyed during the simulation run and has no fixed location in the topology and created in the test ,flow into the driver or are created in the monitor and are sent into the scoreboard The complete diagram expansion in figure 64 may show other predefined component types that are driver from uvm component class , The class should be used as the base class for any user definend components so to create my agent class you should etend uvm agent base class and te same for driver, monitor . . . etc. The uvm class library provides all the building blocks you need to develop and easily constructed.

4.4.1 UVM TLM communication

The communication between components has 2 connections called a TLM that stands for transaction level modeling , The TLM connections has 2 pins the initiator that has an object called port and the target contains an object called export, As in the driver when it pulls transactions from sequencer , A port is a one to one connection to an export a less common type is when producer pushes transactions to a consumer. Another kind of

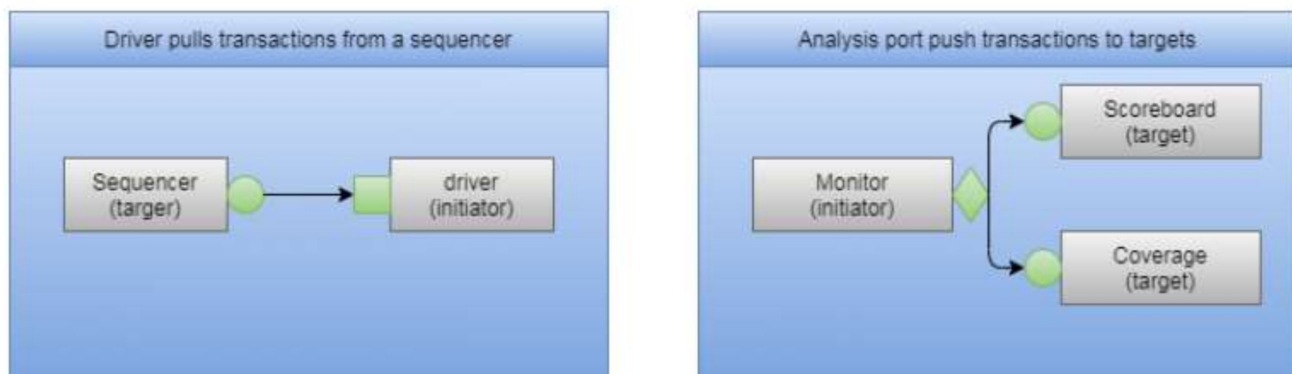


Figure 64: Transaction level modeling VS Analysis port export

connections called Analysis port export that is used to connect monitor to both scoreboard and coverage collectors so it's a one to many connection.see 64

4.4.2 DUT connections to testbench

The DUT's ports can not be connected directly to the testbench class objects so a different SystemVerilog means of communication, which is virtual interfaces is used.see 65 The DUT's ports are connected to an instance of an interface. The Testbench communicates with the DUT through the interface instance. Using a virtual interface as a reference or handle to the interface instance, the testbench can access the tasks, functions, ports, and internal variables of the SystemVerilog interface. As the interface instance is connected to the DUT pins, the testbench can monitor and control the DUT pins indirectly through the interface elements.

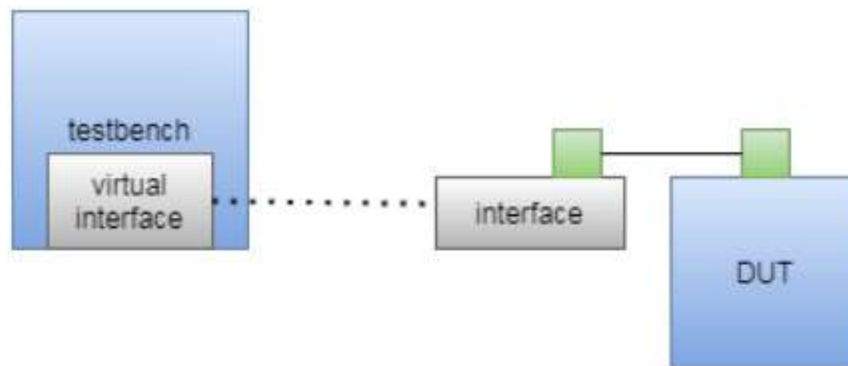


Figure 65: DUT connected to test-bench through irtual interface

4.4.3 How are UVM classes related?

The UVM library defines a set of base classes and utilities that facilitate the design of modular, scalable, reusable verification environments. All UVM classes are derived from uvm object and an individual transaction is a sequence item contains a transactions properties and methods, A series of generated sequence items are known as sequence to obtain a real stimulus. The transactions are sent to the sequencer that routes between multiple sequences, Sequencer and drivers are components which are permanent objects created at the start of simulation and remain for the entire simulation. UVM sequence class is derives from uvm sequence base class that contains a task body to generate one or more sequences, There are 4 steps should be done to generate transactions first one is to create a sequence item object, then wait for a driver to request a transaction through sequencer, The third step is to assign the transaction values where you have to check your

randomization and Lastly send the transaction to driver and wait for completion. Sequences can have randomized properties by allocating them as random variables then the sequences can behave differently each time when it is started, Complex sequences may get a feedback for the dut to choose between branches to complete the sequence.

4.5 UVM Phases

4.5.1 Why does UVM need phases?

Because UVM uses system Verilog OOP which enables reusing and editing classes and objects which can be created at different times, so it is possible to create a new object in the middle of the simulation , which could end by calling a component while it hasn't been initialized yet leading to wrong testbench outputs.

4.5.2 Why Verilog testbenches don't need phases?

Because it consists of static modules which have a set of ports to communicate with other test bench components Static modules means have their instances created at the beginning of the simulation, so there are no worries about any component being called without it being created

4.5.3 Hierarchy of UVM Phases:

UVM -phases can be grouped into three categories see [38]:

1. **Build time phases:**

Phases executed in the start of simulation in which the testbench components are constructed, configured and connected in zero time simulation since this phase methods are functions executed in zero time simulation.

Build phase Is done from the top to the bottom, They consist of:

- Build phase: function used to build test bench components and create their instance.
- Connect phase: function used to connect between different testbench components via TLM ports.
- End of elaboration phase: function used to display UVM topology and other functions required to be done after connection
- Start of simulation phase: function used to set initial run-time configuration and display topology.

2. Run time phases:

Actual simulation that consumes time happens in this UVM phase and runs parallel to other UVM run-time phases. Consists of:

- Pre-reset : the pre reset phase starts at the same time as the run phase. Its purpose is to take care of any activity that should occur before the reset, such as waiting for a power-good signal to go active.

- Reset: responsible for DUT reset.
- Post reset: responsible for any required actions after reset.

- Pre configure: This phase is intended for anything that is required to prepare for the DUT configuration process after the DUT is out of reset.
- Configure: configure phase is used to put the DUT into a known state before the stimulus could be applied to the DUT.

- Post configure: This phase is intended to wait for the effect of the configuration to propagate through the DUT.

- Pre main: pre main is used to ensure that all the components needed to generate the stimulus are ready to do so.

- Main: main phase is where the stimulus specified by the Test case is generated and applied to the DUT.

- Post main: Used for any final act after the main phase.
- Pre shutdown: This phase is acts like a buffer to apply any stimulus before the shutdown phase starts.

- Shutdown: The shutdown phase is to ensure that the effects of stimulus generated during the main phase has propagated through the DUT and that the resultant data has drained away.
- Post shutdown: post shutdown is intended for any final activity before exiting the run phase. After it UVM Testbench starts the cleanup phase.

3. Clean-Up phases:

It is the phase where the results of the testcase are collected and reported. Consists of:

- Extract : Used to retrieve and process information from scoreboards and functional coverage monitors.
- Check: Used to check that the DUT behaved correctly and to identify any errors that may have occurred during the execution of the test bench.
- Report: Used to display the results of the simulation or to write the results to file.
- Final: Used to complete any other outstanding actions that the test bench has not already completed.

To summarize the UVM phases, here in the next figure 66.

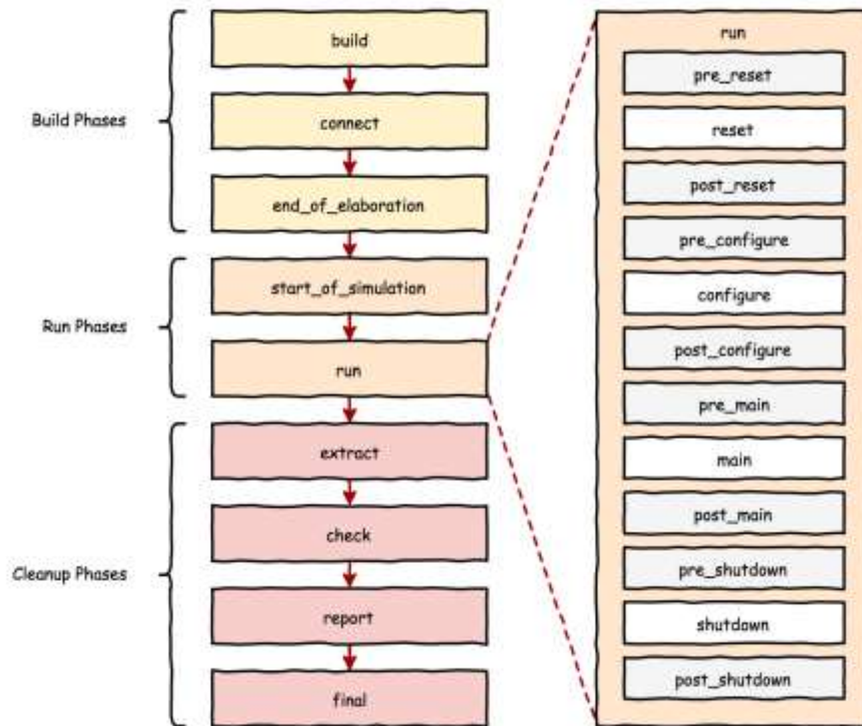


Figure 66: UVM phases

4.6 UVM environment approaches

4.6.1 UVM Transactions approach [39]

We've modularized our work by providing methods in each class that do the work of that class and by being careful to avoid situations where one class needs to know the internal workings of another . To maintain adaptability and reusability for passing data between classes (tester , driver and scoreboard) we exploit the advantages of classes , methods and OOP as the class and the objects we instantiate from that class have two useful points : classes have methods that interact with the data and hide details from users. We work with objects through handles, and we can pass these handles around our test- bench. Therefore several object scan easily share a piece of data. This is implemented these advantages using UVM class library transactions. Encapsulating all this data in the transaction makes the rest of the test-bench much simpler. For example, tester won't need to figure out legal values to drive the test-bench. It will simply let the transaction randomize itself. transactions classes definition We define transactions by extending the uvm transaction base class and writing the methods (convert2string , do copy , do compare) .Transactions encapsulate both data and all the operations we can do to that data. Data fields can be randomized using System Verilog's built-in randomize method. The uvm transaction class extends uvm object, not uvm component, there is a result transaction class to hold results and another transaction class that extends command transaction to generate different stimulus without changing the tester object this class will use same way as the command transaction but under dedicated constrains. The result transaction class is just like the command transaction class , The scoreboard will use the do compare() method to compare predicted results to actual results. Transaction-level simulation makes it easier to compare predicted and actual results Both the result monitor and the predictor create result transaction objects. The result monitor passes us an actual result Then we get the corresponding command from the command monitor and use the predict result() method to create a predicted result transaction. We use compare() to see if we got the right result. The scoreboard is now much simpler. We override the command transaction with the class that generates stimulus, The override causes the tester to create a constrained transaction from the class mentioned rather than a command transaction, without modifying the tester. This transaction approach focuses on data. Classes and objects are created to easily create, compare, and transport data. Although in this approach the data classes are separated from the structure classes, the data stimulus is not separated from structure. Good test benches separate the order of the transactions (the test stimulus) from the test bench structure. See 67

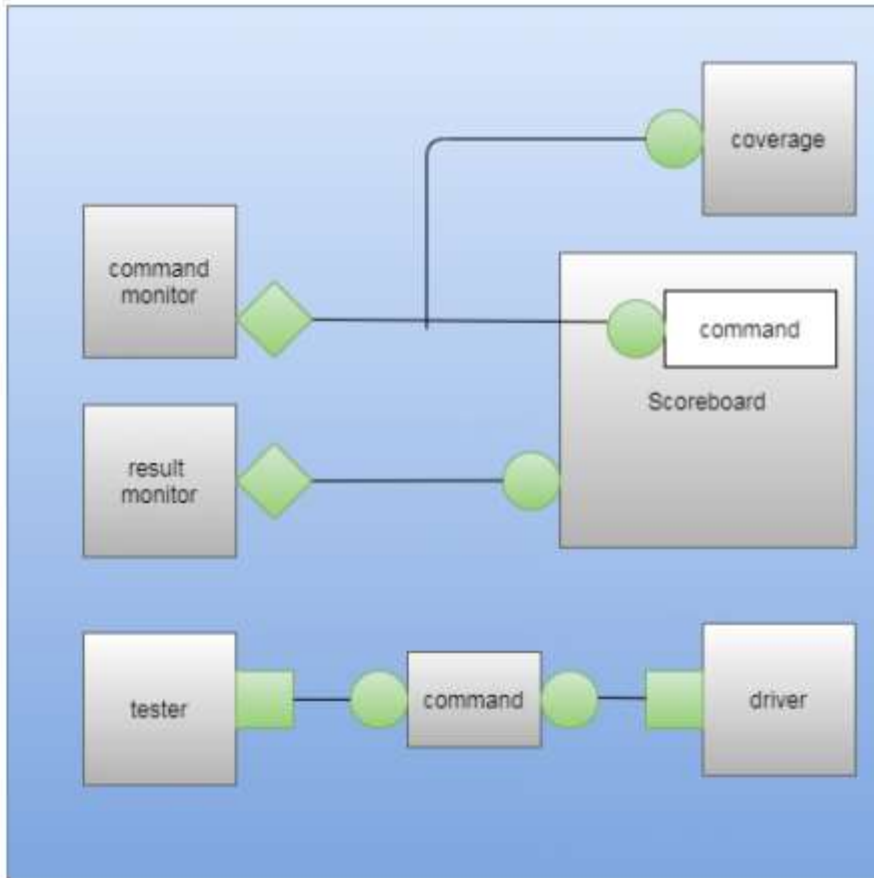


Figure 67: UVM Tansactions approach envirnoment connectionss

In the following sections there are detailed explanations on the design and mechanism of TOP module of convolution testbench.

The design of **UVM test bench** can be described in the following figure 68

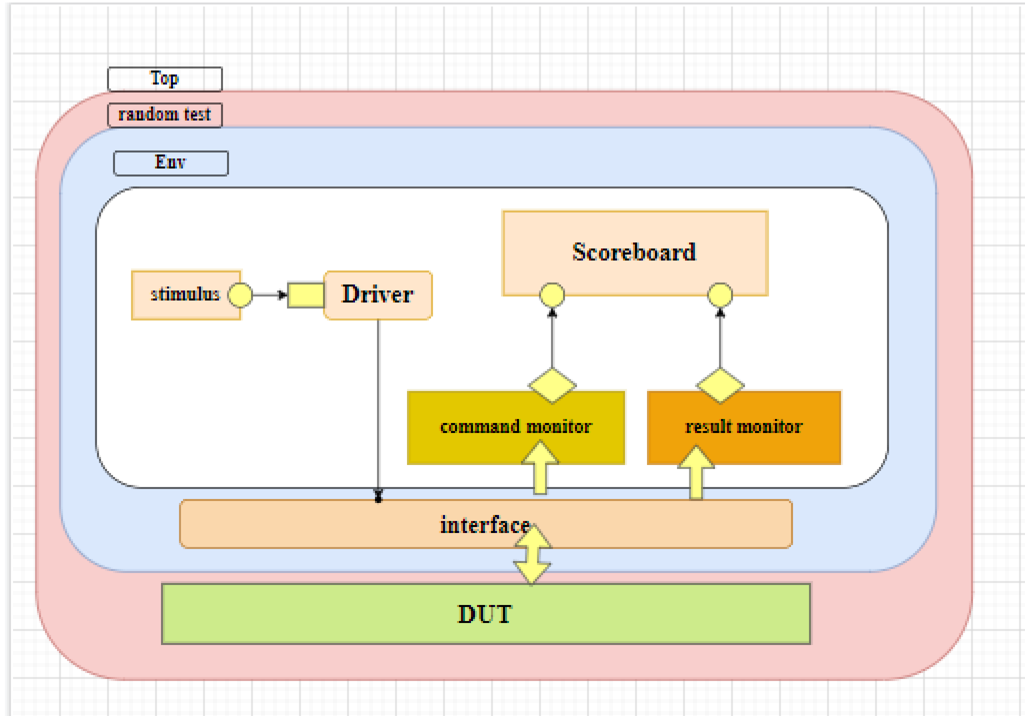


Figure 68: Design of test bench module

In the following sections there are more details about each part of the testbench of convolution function:

1- Command /input Transaction Class

Which mainly define the randomized input variables of the module F, N, image matrix and filter matrix also put constraints to control relations between the size of both image and filter matrix like in figure 69.

```
rand [2:0]    unsigned  F;
rand [7:0]    unsigned  N;
rand [N:0]    unsigned  src_mat    [N:0];
rand [F:0]    unsigned  filter_mat [F:0]; //i must pass the whole matr

constraint c   { (N+1)%(F+1) == 0;} // relation between N and F
constraint c_f {F inside {[1:7]};}
constraint c_N {N inside {[1:255]};}
```

Figure 69: Snippets from class of command transaction

Also this class contains functions like copy, clone, compare and convert to string.

1- "do_copy" function which create a new transaction and copy all elements of transactionas well as structure as shown in figure 70.

```
function void do_copy(uvm_object rhs);
  command_transaction copied_transaction_h;

  if(rhs == null)
    `uvm_fatal("COMMAND TRANSACTION", "Tried to copy from a null pointer")

  if(!$cast(copied_transaction_h,rhs))
    `uvm_fatal("COMMAND TRANSACTION", "Tried to copy wrong type.")

  super.do_copy(rhs); // copy all parent class data

  F = copied_transaction_h.F;
  N = copied_transaction_h.N;
  src_mat = copied_transaction_h.src_mat;
  filter_mat = copied_transaction_h.filter_mat;

endfunction : do_copy
```

Figure 70: do_copy function

2- "clone_me" function which create a new transaction but don't copy the data just copy structure only as shown in figure 71.

```
function command_transaction clone_me();
  command_transaction clone;
  uvm_object tmp;

  tmp = this.clone();
  $cast(clone, tmp);
  return clone;
endfunction : clone_me
```

Figure 71: cone_me function

3- "do_compare" function which define a bit "same" then compare the type of both transactions and then compare each element in the transaction if the two transaction are the same then [same='1'] if else then [same='0'] as shown in the figure 72.

```
function bit do_compare(uvm_object rhs, uvm_comparer comparer);
  command_transaction compared_transaction_h;
  bit   same;

  if (rhs==null) `uvm_fatal("RANDOM TRANSACTION",
                            "Tried to do comparison to a null pointer");

  if (!$cast(compared_transaction_h,rhs))
    same = 0;
  else
    same = super.do_compare(rhs, comparer) &&
           (compared_transaction_h.F == F) &&
           (compared_transaction_h.N == N) &&
           (compared_transaction_h.src_mat == src_mat) &&
           (compared_transaction_h.filter_mat == filter_mat);

  return same;
endfunction : do_compare
```

Figure 72: do_compare function

4- "Convert2string" function to print string S which contains the size of image and filter matrix and the corresponding randomized two matrices as shown in figure 73.

```
function string convert2string();

  string s,str_src,str_filter;
  s = $sformatf("F: %0d  N: %0d ",F, N);
  str_src = $sformatf("%p", src_mat);
  str_filter = $sformatf("%p", filter_mat);
  s={s,"src matrix contents:  ",str_src,"filter matrix contents:  ",str_filter};

  return s;
endfunction : convert2string
```

Figure 73: conv2string sunction

2- Result/output transaction Class

This class is same like command transaction class but specified for output of DUT only, we have three functions which can be applied on the output of convolution, copy, compare and convert to string as explained before.

3- Driver Class

In this class we define an instant from the interface "conv_bfm" class and then build the connection between driver class and interface class as shown in figure 74

```
class driver extends uvm_component;
  `uvm_component_utils(driver)

  virtual conv_bfm bfm;

  uvm_get_port #(command_transaction) command_port;

  function new (string name, uvm_component parent);
    super.new(name, parent);
  endfunction : new

  function void build_phase(uvm_phase phase);
    if(!uvm_config_db #(virtual conv_bfm)::get(null, "*","bfm", bfm))
      `uvm_fatal("DRIVER", "Failed to get BFM")
    command_port = new("command_port",this);
  endfunction : build_phase
```

Figure 74: Snippet from driver class

The main function of driver is that it get a new command/input transaction when I need to test a new case then pass it to interface which in turn the interface send it to both DUT unit and scoreboard to calculate both actual and predicted output.

Also in this class we have run task which send data from driver through interface to command monitor (through task "send_op") this done by defining the operation we will apply if store in ram0 (src matrix) or store in ramb (kernel matrix) this done through variable "op_set" then send data element by element from the two matrices then start in storing data to convolution unit as shown in the figure 75

```
task run_phase(uvm_phase phase);//**************************

  int  [31:0] Data;
  int             conv_finalproduct;
  bit   [2:0]     op_set;
  command_transaction    command;
  forever begin : command_loop
    command_port.get(command);
    for(int i=0;i<N*N+1;i++) //for loop to send src matrix
      op_set = 3'b001; //store in ram0 without conv
      bfm.send_op(command.src_mat[i] , op_set , command.F,command.N ,conv_finalproduct);
    end
    for(int i=0;i<F*F+1;i++) //for loop to send src matrix
      op_set = 3'b010; //store in ramb without conv
      bfm.send_op(command.filter_mat[i] , op_set , command.F,command.N ,conv_finalproduct);
    end
    //to make load to conv *****
     op_set= 3'b011 ;
     Data=32'b0;
     bfm.send_op(Data, op_set , command.F,command.N ,conv_finalproduct);
   //to store from conv
     op_set= 3'b100 ;
     Data=32'b0;
     bfm.send_op(Data, op_set , command.F,command.N ,conv_finalproduct);
  end : command_loop
endtask : run_phase
```

Figure 75: Snippet from driver class "run_phase"

4- Command monitor Class

Monitor class function is to monitor the DUT until the result become ready, In this class we define an instant from the interface “conv_bfm” class and then build the connection with monitor like in figure 76.

```
class command_monitor extends uvm_component;
   `uvm_component_utils(command_monitor);
    int i=0,j=0;
   virtual conv_bfm bfm;

   uvm_analysis_port #(command_transaction) ap;

   function new (string name, uvm_component parent);
      super.new(name,parent);
   endfunction

   function void build_phase(uvm_phase phase);
      if(!uvm_config_db #(virtual conv_bfm)::get(null, "*","bfm", bfm))
    `uvm_fatal("COMMAND MONITOR", "Failed to get BFM")
      bfm.command_monitor_h = this;
      ap  = new("ap",this);
   endfunction : build_phase
```

Figure 76: snippet from command monitor

Also this class contains function called “write_to_monitor” which enable us to send data (F, N, Src matrix, filter matrix) from DUT through interface to command monitor as shown in the figure 77, we send data of two matrices element by element and then from variable “iop” which define the operation applied at the present which may be one of (**store** data in the convolution unit or **load** data from convolution unit)

```
function void write_to_monitor(bit [31:0] Data_in ,  operation_t iop, bit [2:0]  F_size, bit [7:0]  N_size );
  command_transaction cmd;
  `uvm_info("COMMAND MONITOR",$sformatf("MONITOR: Data_in: %0d  F_size: %0d  N_size: %0d operation_type: %s",
            Data_in, F_size, N_size,op.name()), UVM_HIGH);
  cmd = new("cmd");
  cmd.F = F_size;
  cmd.N = N_size;
  case(iop)

     3'b001 : begin //store_in_ram0
               cmd.str_mat[i]=Data_in;
               i++;
             end
     3'b010 :  begin //store_in_ramb
               cmd.filter_mat[j]= Data_in;
               j++;
             end
     3'b011 : `uvm_info("COMMAND MONITOR",$sformatf("MONITOR: store data in convolution operation");
     3'b100 : `uvm_info("COMMAND MONITOR",$sformatf("MONITOR: load data from convolution operation ");
     default : $fatal("Illegal operation on op bus");
   endcase // case (op)
  ap.write(cmd);
endfunction : write_to_monitor
```

Figure 77: snippet from command monitor class

5- Result monitor Class

In this class we write the result of convolution unit in the monitor through interface as shown in the figure 78.

```
class result_monitor extends uvm_component;
   `uvm_component_utils(result_monitor);

   virtual conv_bfm bfm;
   uvm_analysis_port #(result_transaction) ap;

   function new (string name, uvm_component parent);
      super.new(name, parent);
   endfunction : new

   function void build_phase(uvm_phase phase);
    if(!uvm_config_db #(virtual conv_bfm)::get(null, "*","bfm", bfm))
        `uvm_fatal("RESULT MONITOR", "Failed to get BFM")

      bfm.result_monitor_h = this;
      ap  = new("ap",this);
   endfunction : build_phase

   function void write_to_monitor(int r); //*****
      result_transaction result_t;
      result_t = new("result_t");
      result_t.conv_finalproduct = r;
      ap.write(result_t);
   endfunction : write_to_monitor

endclass : result_monitor
```

Figure 78: result_monitor class

6- Scoreboard Class

```
class scoreboard extends uvm_subscriber #(result_transaction);
    `uvm_component_utils(scoreboard);


    uvm_tlm_analysis_fifo #(command_transaction) cmd_f; //to get data (transaction ) from fifo

    function new (string name, uvm_component parent);
        super.new(name, parent);
    endfunction : new

    function void build_phase(uvm_phase phase);
        cmd_f = new ("cmd_f", this);
    endfunction : build_phase

function result_transaction predict_result(command_transaction cmd);
    result_transaction predicted;

    predicted = new("predicted");

    predicted. conv_finalproduct =conv(cmd.str_mat,cmd.filter_mat,cmd.N,cmd.F); //put here external module cpp

    return predicted;

endfunction : predict_result
```

Figure 79: Snippet from scoreboard class

The scoreboard function is to compare the result output from the DUT unit with the reference model output to check the functionality of the DUT module, as shown from figure 79, first we connect the monitor class with scoreboard class through **“uvm_tlm_analysis_fifo”** after this compute the expected/predicted result from reference model this done by calling “conv” which is CPP function included in the scoreboard using **DPI** method which is referred to SystemVerilog “Direct Programming Interface” which will be explained very well in the next section, for “conv” function it has four arguments which are N, F, src_mat, filter_mat as shown in fugure 79.

After getting output from CPP function it started to compare the two results and print if they are matched or there is an error then print string containing both actual and predicted outputs as shown in the figure 80.

```
function void write(result_transaction t);
   string data_str;
   command_transaction cmd;
   result_transaction predicted;


     if (!cmd_f.try_get(cmd)) //try to get transaction from fifo
       $fatal(1, "Missing command in self checker");


   predicted = predict_result(cmd);

   data_str = {                    cmd.convert2string(),
                 " ==> Actual "  ,    t.convert2string(), //coming from dut
                 "/Predicted ",predicted.convert2string()};//coming from reference model

   if (!predicted.compare(t))
     `uvm_error("SELF CHECKER", {"FAIL: ",data_str})
   else
     `uvm_info ("SELF CHECKER", {"PASS: ", data_str}, UVM_HIGH)

endfunction : write
```

Figure 80: Snippet from scoreboard class "write" function

7- Top module

In top module we have instant from interface and DUT unit which has arguments of comprehensive convolution module (convolution unit with memory file) passed through interface, then it run test as shown in the figure 81

```
module top;
   import uvm_pkg::*;
   import   conv_pkg::*;
   `include "conv_macros.svh"
   `include "uvm_macros.svh"


   // code to regulare order of operation must be done


   conv_bfm       bfm(); //interface
   comprehensive_conv Dut ( .clk(bfm.clk), .reset(bfm.reset), .Local_Reset(bfm.Local_Reset), .F(bfm.F), .N(bfm.N),.Opcode0
                             .Op1_Enable0(bfm.Op1_Enable0), .Op1_Enable1(bfm.Op1_Enable1), .Op1_Enableb(bfm.Op1_Enableb),
                             .Op3_Enable0(bfm.Op3_Enable0), .Op3_Enable1(bfm.Op3_Enable1),.Op4_Enable0(bfm.Op4_Enable0), .
                             .start(bfm.start),.conv_finalproduct(bfm.conv_finalproduct), .Finish_RAM0(bfm.Finish_RAM0), .
                            );
initial begin
   uvm_config_db #(virtual conv_bfm)::set(null, "*", "bfm", bfm);
   run_test();
end

endmodule : top
```

Figure 81: Top module code

8- Env class

Env class function is to connect all the component in top module together, here env connect both tester with scoreboard, driver, command monitor, result monitor as shown in figure 82.

```
class env extends uvm_env;
  `uvm_component_utils(env);

  tester    tester_h;
  //coverage  coverage_h;
  scoreboard scoreboard_h;
  driver    driver_h;
  command_monitor command_monitor_h;
  result_monitor result_monitor_h;
  uvm_tlm_fifo #(command_transaction) command_f;

  function new (string name, uvm_component parent);
    super.new(name,parent);
  endfunction : new
function void build_phase(uvm_phase phase);
  command_f = new("command_f", this);
  tester_h  = tester::type_id::create("tester_h",this);
  driver_h  = driver::type_id::create("driver_h",this);
 // coverage_h  =  coverage::type_id::create ("coverage_h",this);
  scoreboard_h = scoreboard::type_id::create("scoreboard_h",this);
  command_monitor_h = command_monitor::type_id::create("command_monitor_h",this);
  result_monitor_h = result_monitor::type_id::create("result_monitor_h",this);
endfunction : build_phase

function void connect_phase(uvm_phase phase);
  driver_h.command_port.connect(command_f.get_export);
  tester_h.command_port.connect(command_f.put_export);
  //command_f.put_ap.connect(coverage_h.analysis_export);
  command_monitor_h.ap.connect(scoreboard_h.cmd_f.analysis_export);
  result_monitor_h.ap.connect(scoreboard_h.analysis_export);
endfunction : connect_phase

function void end_of_elaboration_phase(uvm_phase phase);
  scoreboard_h.set_report_verbosity_level_hier(UVM_HIGH);
endfunction : end_of_elaboration_phase
```

Figure 82: env class code

9- Interface class

“conv_bfm” is the interface between both monitor and DUT unit as shown in the figure 83 it defines all the inputs and outputs which passed as arguments to DUT unit.

```
interface conv_bfm; //interface
   import conv_pkg::*;

// declaration of inputs and outputs
logic   clk;
logic   reset;
//declare variables of conv module
logic  Local_Reset;
logic  conv_en; // Convolution Start
logic  [2:0] F;  // Filter Size 1~8
logic  [7:0] N; // Feature Input Size 1~32
logic [1:0] Opcode0, Opcode1;
logic       Opcodeb;
logic Op1_Enable0, Op1_Enable1, Op1_Enableb;  //Enable to start Op1
logic Op2_Enable0, Op2_Enable1, Op2_Enableb; // local Reset of counters in operation 2
logic Op3_Enable0, Op3_Enable1; //Enable to start Op3
logic Op4_Enable0, Op4_Enable1;//Enable to start Op4
logic [31:0] Bus_interface; //Data_in
operation_t   op; // to determine which operation
bit           start;

// Output
logic [31:0] conv_finalproduct;
logic Finish_RAM0, Finish_RAM1, Finish_RAMB, Finish_Convolution;
```

Figure 83: snippet from interface class

Also it contains reset function which reset conv unit whenever we need as shown in the figure 84 this done by setting the value of **Local_reset** by ‘1’ for one clock cycle then make it ‘0’ again.

```
//to reset conv
  task reset_conv();
     Local_Reset = 1'b1;
     @(posedge clk); //delay
     Local_Reset = 1'b0;
     start = 1'b0;
  endtask : reset_conv
```

Figure 84: snippet from interface class "reset_conv" function

Then we have “send_op” task which used by interface class to send data to command monitor as explained before, it has input arguments (F_size, N_size, Data_in, iop and convolution_product) in this task we handle the order of implementing the four operations we explained in the **Memory file** section which are:

1- Store without convolution in ram0 (image matrix) → (op =3’b001)
2- Store in ramb (kernel matrix) → (op =3’b010)
3- Store both kernel and image matrices from ramb and ram0 to convolution unit → (op =3’b011)

4- Load the result matrix from convolution unit in ram1 → (op =3’b100)

This done by case statement as shown in the figure **, then pass each of N, F, Data_in to bus_interface and set the “**start”** signal to ‘1’.

Then waiting until convolution ends its operation and “**Finish_Convolution”** became ‘1’ after that it pass the convolution output and reset start signal to ‘0’.

```
task send_op(input [31:0] Data_in , input operation_t iop,  input [2:0]  F_size, input [7:0]  N_size , output [31:0] Convolution_Product);
  if (iop == 000) begin //local reset reset
    @(posedge clk);
    reset <= 1'b1; //= ----- <=
    start <= 1'b0;
    @(posedge clk);
    #1;
    reset <= 1'b0;
  end else begin
    @(negedge clk);
    op <= iop;
    case (op)
      3'b001 : begin //store in ram0
                Opcode0 <=2'b00; //store without_conv
                Op1_Enable0 <=1'b1;
              end

      3'b010 :begin  //store in ramb
                Opcodeb <=1'b0;
                Op1_Enableb <=1'b1; //Store Matrix F x F from Bus interface without conv
              end

      3'b011 :begin //load to conv
               Opcode0 <=2'b11;//Load Matrix N x N to PE with Convolution
               Opcodeb <=1'b1;//Load Matrix F x F to PE with Convolution
               Op2_Enableb <=1'b1;
               Op4_Enable0 <=1'b1;
              end

      3'b100 : begin //to load from conv/PE to ram1
               Opcode1 <=2'b10;
               Op2_Enable1 <=1'b1;
               end
     default : $fatal("Illegal operation on op bus");
  endcase // case (op)
    Bus_interface <= Data_in;
    F             <= F_size;
    N             <= N_size;
    start         <= 1'b1;
       do
         @(negedge clk);
       while (Finish_Convolution == 0); //after conv ended
       start <= 1'b0;
       Convolution_Product <= conv_finalproduct;

  end // else: !if(iop == rst_op)
endtask : send_op
```

Figure 85: Snippet from interface class

Finally, we can summarize the behavior of the whole module from figure 86, at each new test driver access the command transaction for a new randomized input transaction then pass it to interface which in turn resend the input transaction to the DUT unit to get the actual output and resend both the input transaction to command monitor and actual output to result monitor then they both send to scoreboard which try to compare them and report the result of analysis.

Figure 86: behavior of test bench module

Chapter 5: RISCV

RISC-V (pronounced "risk-five") is an open standard instruction set architecture (ISA) based on established reduced instruction set computer (RISC) principles. Unlike most other ISA designs, the RISC-V ISA is provided under open source licenses that do not require fees to use. A number of companies are offering or have announced RISC-V hardware, open source operating systems with RISC-V support are available and the instruction set is supported in several popular software toolchains.

Notable features of the RISC-V ISA include a load–store architecture, bit patterns to simplify the multiplexers in a CPU, floating-point, a design that is architecturally neutral, and placing most-significant bits at a fixed location to speed sign extension. The instruction set is designed for a wide range of uses. The base instruction set has a fixed length of 32-bit naturally aligned instructions, and the ISA supports variable length extensions where each instruction could be an any number of 16-bit parcels in length. Subsets support small embedded systems, personal computers, supercomputers with vector processors, and warehouse-scale 19 inch rack-mounted parallel computers.

The instruction set specification defines 32-bit and 64-bit address space variants. The specification includes a description of 128-bit flat address space variant, as an extrapolation of 32 and 64 bit variants, but the 128-bit ISA remains "not frozen" intentionally, because there is yet so little practical experience with such large memory systems.

The project began in 2010 at the University of California, Berkeley, but now many current contributors are volunteers not affiliated with the university. Unlike other academic designs which are typically optimized only for simplicity of exposition, the designers intended that the RISC-V instruction set be useable for practical computers.

RISC-V is unusual not only because it is a recent ISA—born this decade when most alter- natives date from the 1970s or 1980s—but also because it is an open ISA. Unlike practically all prior architectures, its future is free from the fate or the whims of any single corporation, which has doomed many ISAs in the past. It belongs instead to an open, non-profit foundation. The goal of the RISC-V Foundation is to maintain the

stability of RISC-V, evolve it slowly and carefully, solely for technical reasons, and try to make it as popular for hardware as Linux is for operating systems. Extensions for example like:

Q	Standard Extension for Quad-Precision Floating-Point
L	Standard Extension for Decimal Floating-Point
C	Standard Extension for Compressed Instructions
B	Standard Extension for Bit Manipulation
J	Standard Extension for Dynamically Translated Languages
T	Standard Extension for Transactional Memory
P	Standard Extension for Packed-SIMD Instructions
V	Standard Extension for Vector Operations

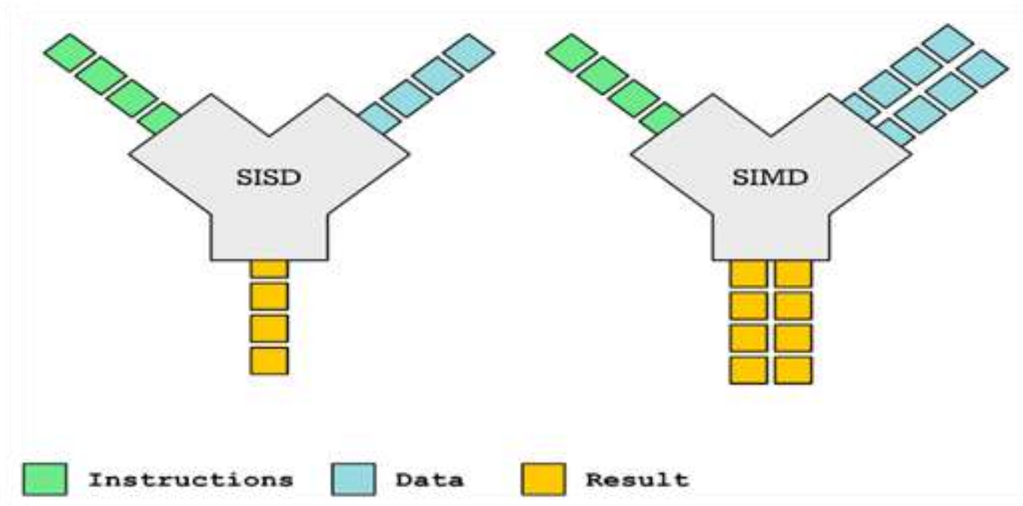
RISC-V has a modular design, consisting of alternative base parts, with added optional extensions. The ISA base and its extensions are developed in a collective effort between industry, the research community and educational institutions. The base specifies instructions (and their encoding), control flow, registers (and their sizes), memory and addressing, logic (i.e., integer) manipulation, and ancillaries. The base alone can implement a simplified general-purpose computer, with full software support, including a general-purpose compiler. The standard extensions are specified to work with all of the standard bases, and with each other without conflict. Many RISC-V computers might implement the compact extension to reduce power consumption, code size, and memory use. Together with a supervisor instruction set extension, S, an RVGC defines all instructions needed to conveniently support a general-purpose operating system.

There are many companies lead the revolution of RISCV in electronics industry can't forget sifive of course…

risc vs risc --> riscv5 vs arm so you can find many people care about to compare between them to work with the best….

Instruction Set Architecture (ISA) is basically the portion of the machine that is visible to the assembly level programmer or the compiler writer. ISA is where software meets hardware. ISA defines the commands/ instructions that can natively be understood by a machine and its micro-architecture, and it also defines how the instructions are to be stored, accessed, and implemented. We give instructions to the hardware of the computer using a language that a computer can understand. The computer language is made up of the words called instructions and the vocabulary is called an instruction set. Instruction sets tell us about the function of each instruction and how the instruction is represented in memory (encoding).

The term architecture describes the functional specification of a processor. It describes what functionality the software can rely on the hardware to provide. Architecture does not tell you how a processor is built. It tells you what a processor can do. Micro-architecture on the other hand describes how a processor is built and designed. Micro-architecture defines, the number and size of caches, cycle counts of instructions, pipeline length, and more.



And we chose to work with pulp cores as they are low power and low area relatively, specifically we chose Ariane core (CVA6)

5.1. CVA6 (Ariane):

CVA6 is an open source RISC V core that implements the 64 bit RISC V base instruction set. In addition to base instructions, CVA6 fully implements the multiplication (M), atomic (A), floating (F), double (D) as well as the compressed (C) extensions as specified by volume I: User-Level ISA V 2.3 as well as the draft privilege extension 1.10. It implements a machine, supervisor, and user privilege levels as well as a tightly integrated data and instruction caches to support Unix like operating system (1).

The primary goal of CVA6 was to design an application class core with reasonable speed to support applications based on open source operating systems like Linux. CVA6 achieves 1.7 GHz speed in 22nm FDSOI technology (2). Moreover, CVA6 has a configurable size option allowing it to operate as a 32 bit core for light weight operation.

CVA6 open source repository includes FPGA SoC prototype with DRAM and IO peripherals. Initially developed to work on Digilent Genesys II board, the SoC prototype allows the user to test the core and run a Linux terminal with simple applications on it. Other FPGA ports for the SoC like Virtex 707 and Kintex 705 are currently under development.

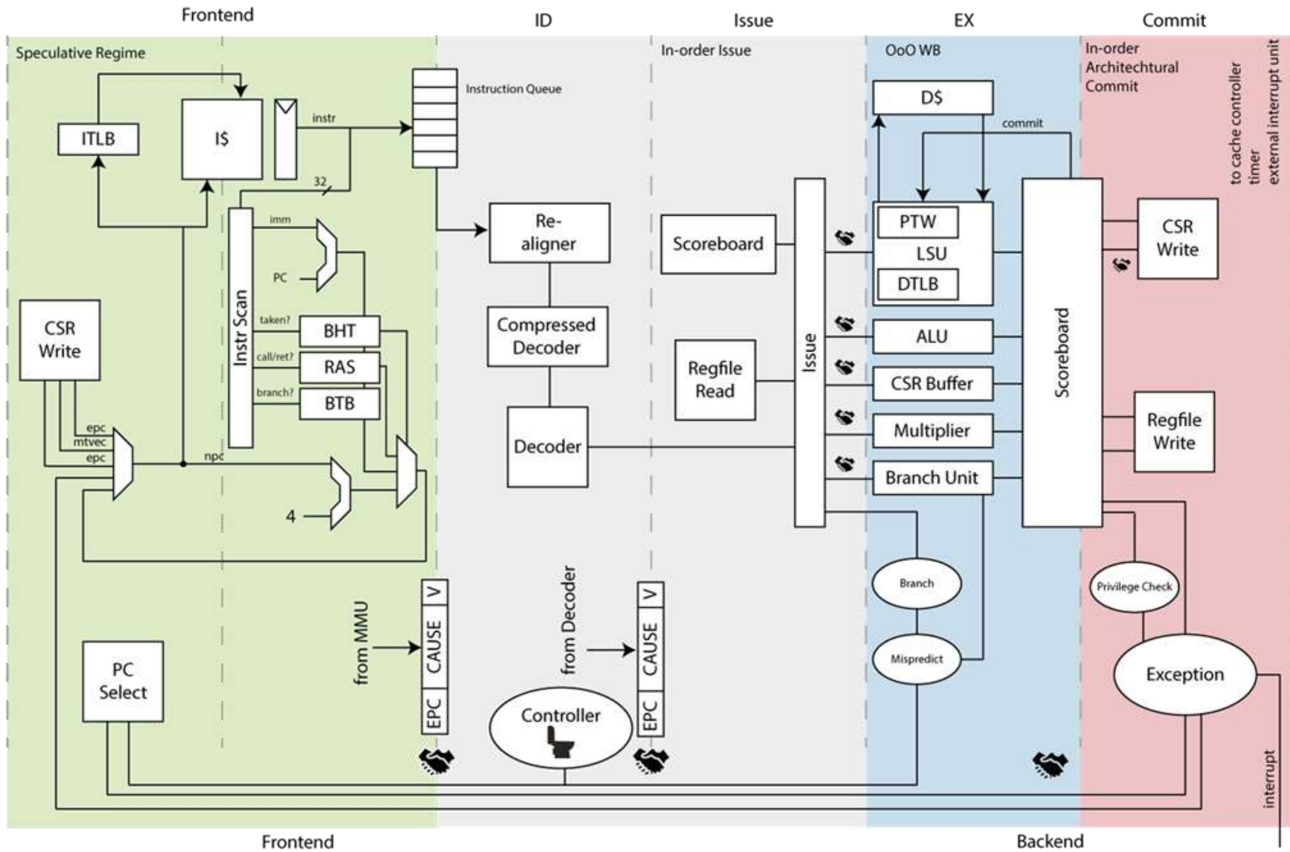
Initially labeled as project Ariane, CVA6 was developed by PULP platform, also responsible for developing other more lightweight RISC V cores including CV32E40P and Ibex. CVA6 is currently maintained by Open Hardware group, who provide active support for the project.

Architecture:

CVA6 consists of 6 stage single issue pipeline with out-of-order execution and in-order commit. The six pipeline stages include:

1- PC generation
2- Instruction fetch
3- Instruction decode
4- Instruction issuing
5- Instruction execution
6- Instruction commit

CVA6 implements dynamic branch prediction using branch history table and branch target buffer to reduce the frequency of control hazards inside the CPU pipeline. A scoreboard was implemented in the form of a FIFO to allow for the parallel execution in multiple functional units and in-order commit of multiple issues.



Block diagram of CVA6. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-ready 1.7GHz 64bit RISC-V Core in 22nm FDSOI Technology

PC Gen stage:

PC generation is responsible for generating the next program counter. The next PC can originate from the following sources:

1- Default assignment: Fetch PC + 4. Each cycle, 32 bit words are fetched. Compressed instructions are processed later in the pipeline

2- Branch predict: A branch history table (BHT) identifies the current instruction as a branch and decides whether the branch is to be taken or not. A Branch target buffer stores the target address of the branch. PC generation stage informs the instruction fetch stage that it performed a prediction on the PC using a user defined data structure **branchpredict_sbe_t** found in the Ariane package**.** The validity of the prediction is verified in the execution stage and a controller is used to flush the pipeline in case of mispredictions

3- Control flow change request: To correct mispredictions

4- Return from environment call: PC gen stage features a return address stack (RAS) that performs corrective action of the PC upon returning from environment calls

5- Exceptions/Interrupt:

Exception is used to refer to an unusual condition occurring at run time associated with an instruction in the current RISCV hardware thread.

Interrupt is used to refer to an external asynchronous event that may cause a RISCV hart to experience an unexpected transfer of control. Trap to refer to the transfer of control to a trap handler caused by either an exception or an interrupt (3).

Upon encountering an exception, PC Gen will generate the next PC as part of the trap vector base address.

6- Pipeline Flush: CSR side effects

7- Debug: Debug has the highest order of precedence as it can interrupt any control flow requests.

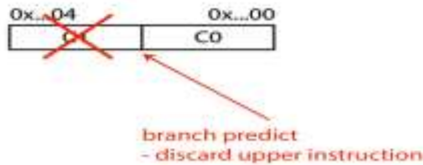
Instruction Fetch stage:

Instruction fetch stage receives the current PC from PC Gen stage and asks the MMU to do address translation on the requested PC and control the instruction memory interface.

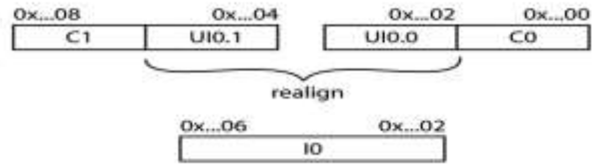
PC Gen and instruction fetch stage are collectively known as the front end of the CVA6. The instruction queue separates between the front end and the back end of the pipeline

The instruction re-aligner was placed in the instruction fetch stage as of release 4.2.0. Since the program counter fetches on word boundaries(32 bits), compressed instructions (16 bits) can cause the instructions to start at half word boundaries, instruction re-aligner keeps track of whether the previous instructions were unaligned or compressed correctly

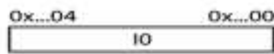
2 Compressed Instructions:



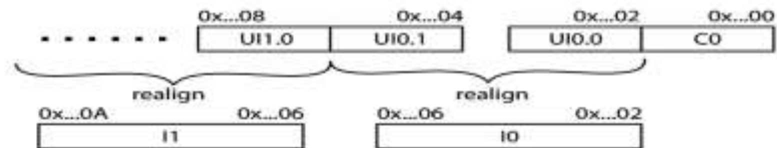
Unaligned Instruction:



Regular Instruction:



Unaligned Instructions:



CVA6 Instruction realigned

Instruction decode stage:

As of release 4.2.0, the instruction decode stage consists of 2 main blocks. The compressed instruction decoder and the instruction decoder. Compressed instructions come out of the front end as 32 bits with the compressed instruction occupying the least significant 16 bits. By checking the first 2 bits, the compressed decoder recognizes whether the the instruction is compressed or not. If the instruction is compressed, then it outputs a 32 bit equivalent of the 16 bit instruction and hands it the the decoder. If the instruction is not compressed then the compressed decoder passes the instruction without modification

The instruction decoder receives the 32 bit instruction and determines whether the instruction is valid or illegal. If the instruction was illegal then the instruction decoder raises an exception, thus an additional code is added to the instruction decoder to validate the acceptance of the co-processor specific instructions. This includes defining the co-processor as a valid functional unit in the **fu_t** structure in the Ariane package.

Additionally, the output of the instruction decode stage takes the form of a data structure named **scoreboard_entry_t** and is defined in the Ariane package. It contains information about the instruction that is useful in later stages. The different fields of the structure includes:

PC: Program counter of the instruction

TRANS ID: Used to index the scoreboard entry in later stages

FU: Functional unit used for the instruction

OP: Operation used

RS1: Address of source register 1

RS2: Address of source register 2

Rd: Address of destination register

RESULT: Used to store the immediate for unfinished instructions

VALID: Is the result valid and ready for commit. If there's an exception then the field is asserted

USE-IMM: Use the immediate as operand b

USE-ZIMM: Use zimm as operand a

USE_PC: Use PC as operand a

EX: Data structure that determines whether an exception has occurred and the cause of the exception.

BP: Data structures that determines the type of control flow and the target address of the prediction

IS_COMPRESSED: Signals a compressed instruction

The **EX** field also carries the Instruction word in a subfield named **tval** as it is usually needed in later stages in the pipeline or when dispatching the instruction to the co-processor for full decoding.

Instruction issue:

Instruction issue stage is concerned with dispatching instructions and their operands to their respective functional units.

By using a scoreboard implemented as a FIFO with 8 entries, the CPU core keeps track of multiple issued instruction. If the scoreboard is full, we can’t issue a new instruction. We can’t issue two predicted branch instructions simultaneously without resolving the first branch predict.

The scoreboard of CVA6 has 4 write-back ports connected to the execute stage of the CPU and 2 commit ports connected to the commit stage to allow up to 2 instructions to be committed simultaneously, this is possible through the register file of the CVA6 that has 2 write ports. The scoreboard also outputs the specific registers being currently used as destination registers by issued instructions in **rd_clobber_gpr_o.**

Instruction issue stage is also concerned with sending the correct operands to their respective functional unit. Reading the correct operands using the address obtained from the instruction decode stage can take place by accessing the register file for these operands.

However, in-case one of the operands is currently being used as a destination register by an already issued instruction, data hazard occurs. We can use forwarding logic to solve this problem by obtaining the operands directly from the result field in the scoreboard. In order for this to work properly, the result must be valid and ready to be committed.

Instruction execute stage:

Instruction execute stage holds the different functional units used for executing the instruction. This includes:

1- ALU: To perform simple arithmetic operations

2- Branch predict unit: To resolve decoded branch predictions and detect mispredictions.

3- Multiplier/Divider

4- CSR buffer: Stores the address of the register the instruction is going to read/write

5- Load Store Unit: Regulates the process of data loading and storing from and into the data cache.

Through its write back ports, instruction execute stage stores the result of the executed operations in an out-of-order manner.

The execute stage is where the custom co-processor instructions are dispatched to the co-processor

Instruction commit stage:

The rule of thumb of the commit stage is that no instruction should be able to modify the architectural state of the CPU including the register file and the control and status registers before committing the instruction. Instruction commit stage is responsible for committing instructions that were executed and written back out-of-order in an in-order manner.

Instruction commit stage is also responsible for committing possible exceptions and interrupts to the controller of the pipeline

Controller:

Pipeline controller is mainly responsible for flushing the pipeline in case of exceptions and mispredictions.

Get ready to work with CVA6:

To deal with CVA6 and test it on FPGA you have to get some files ready before you can burn the core on that FPGA as in the repository of openhwgroup they developed the CVA6 and made it support many of FPGA by making these FPGAs' files ...

CVA6 as we talked about before... it supports some FPGA boards like GENESIS-II and vc707 and Kc705...every board differ from the other in its fabric and how can you use it to test your code in our case in CVA6 we want to make debugging on chip with openocd after we burn the bitstream we generated on tool like vivado then we can do that debugging.

RISCV in general work with Linux foundation so you can find most of RISCV cores and codes are worked on linux OS so you have to get linux OS and we recommend UBUNTU-20 and set up vivado on it... if you will work with CVA6 you will need to get VIVADO 2018.2 version...that's really important to know the version of vivado that the codes of the core you will work with developed on which version cause of the IPs they

used and everything regarding to that version will differ from version to another so working with the same version is very important to take into consideration before you start.

before you go to work on FPGA and codes of the core you picked you have to setup some tools of RISCV on your linux to be able use them in build and compile any code of RISCV and other tools will help in debug and simulation.

we will stat with ariane-SDK it will be convenient to CVA6 you can get clone it with linux terminal but we recommend to clone every repository in the ariane-SDK alone first then add them into the ariane-SDK then do the steps in the repository of ariane-SDK step by step .most important thing in this repository is the makefile which we can use to setup every tool we want to set on our os like RISCV-GNU and PK and Spike(riscv-isa-sim)...you can use this makefile to setup every tool alone or use (make all) command to set up all of them before we go to another topic we had here problems with the makefile to set up all of them at once you will find a lot of errors on the terminal during the setting process so our recommendation to set up every tool alone independent on other tools (that will happen by cloning each tool's repository and set it step by step) or you can do something else if you want to use th makefile of ariane-SDK you can clone all files of ariane-SDK repository without repositories of tools in it like[riscv-fesvr,riscv-gnu-toolchain,riscv-isa-sim,riscv-pk] then clone every repository of tools alone into the folder of ariane-SDK then use make all it will work properly...small hint here that there are some very important tools not in this make file like riscv-openocd and riscv-debug-spec so in general we recommend to set every single tool all alone on your linux OS.

what is makefile? this is a file has a specific way or language to be written with and help you to work smart on linux terminal (and you can find it in everything on linux or OS using terminal) you can write on it many things which can help you in work with a program or anything want to do on linux demands many steps and don't want to repeat these steps every time you do that work like the makefile you can find in every repository want to set it up on your OS.

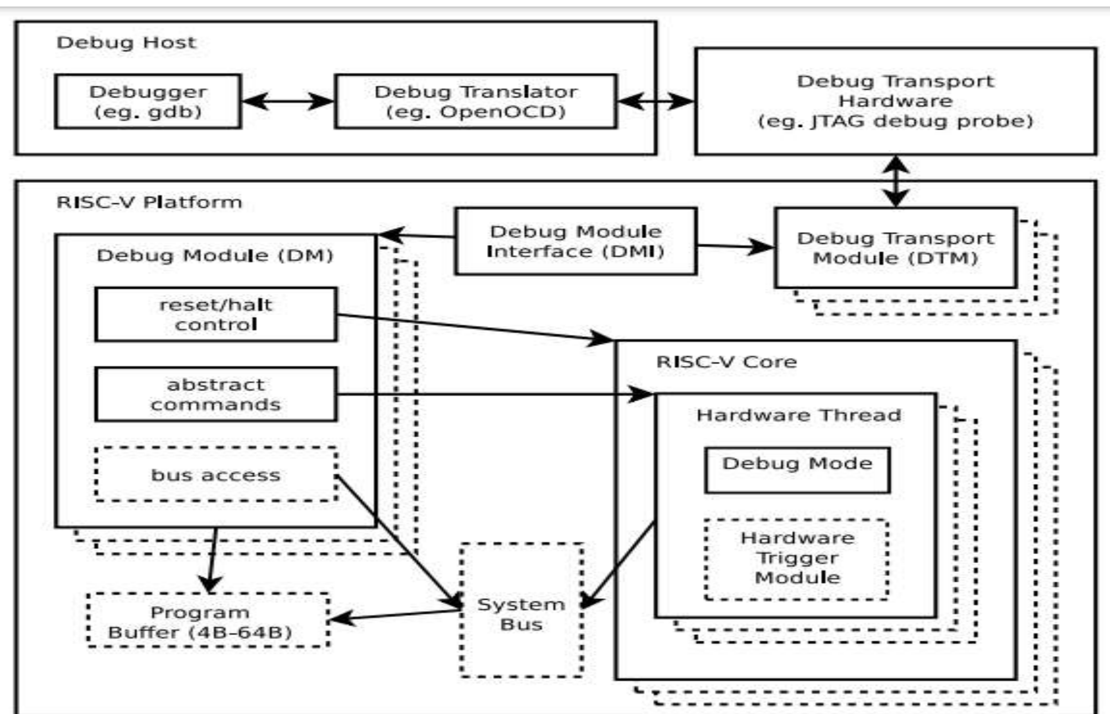
we will talk about some important tools which helped us in our work or we used it heavily:

1) riscv-gnu-toolchain: This is the RISC-V C and C++ cross-compiler. It supports two build modes: a generic ELF/Newlib toolchain and a more sophisticated Linux-ELF/glibc

toolchain. The GNU toolchain is a collection of programming tools...it includes the RISCV extensions and the compilers or RISCV we can use to compile codes in C and C++ by using compilers of riscv32-unknown-elf-gcc for C codes and riscv32-unknown-elf-g++ for C++ codes that compilers we can use them as commands on terminal to compiles files written in these languages in RISCV-ISA...it also include riscv32-unknown-elf-gdb which is very important in debugging codes on RISCV-ISA and other programs which can be useful in many applications with RISCV.

2) riscv-isa-sim(the RISC-V ISA Simulator): implements a functional model of one or more RISC-V harts. It is named after the golden spike used to celebrate the completion of the US transcontinental railway.it contain [Spike] it's a RISCV simulator and supports many RISCV ISA features like A,F,Q,D,V and so on... it's used for Compiling and Running codes in C or C++...

3)riscv-debug-spec(RISC-V Debug Specification):this is very important in our work and everyone wants to work with RISCV on FPGA and want to debug or to do the debugging process with RISCV in general so we will talk about the system overview then talk about this repository... system overview:



The user interacts with the Debug Host (e.g. laptop), which is running a debugger (e.g. gdb). The debugger communicates with a Debug Translator (e.g. OpenOCD, which may

include a hardware driver) to communicate with Debug Transport Hardware (e.g. Olimex USB-JTAG adapter). The Debug Transport Hardware connects the Debug Host to the hardware platform's Debug Transport Module (DTM). The DTM provides access to one or more Debug Modules (DMs) using the Debug Module Interface (DMI). Each hart in the hardware platform is controlled by exactly one DM. Harts may be heterogeneous. There is no further limit on the hart-DM mapping, but usually all harts in a single core are controlled by the same DM. In most hardware platforms there will only be one DM that controls all the harts in the hardware platform. DMs provide run control of their harts in the hardware platform. Abstract commands provide access to GPRs. Additional registers are accessible through abstract commands or by writing programs to the optional Program Buffer. The Program Buffer allows the debugger to execute arbitrary instructions on a hart. This mechanism can also be used to access memory. An optional system bus access block allows memory accesses without using a RISC-V hart to perform the access. Each RISC-V hart may implement a Trigger Module. When trigger conditions are met, harts will halt and inform the debug module that they have halted.

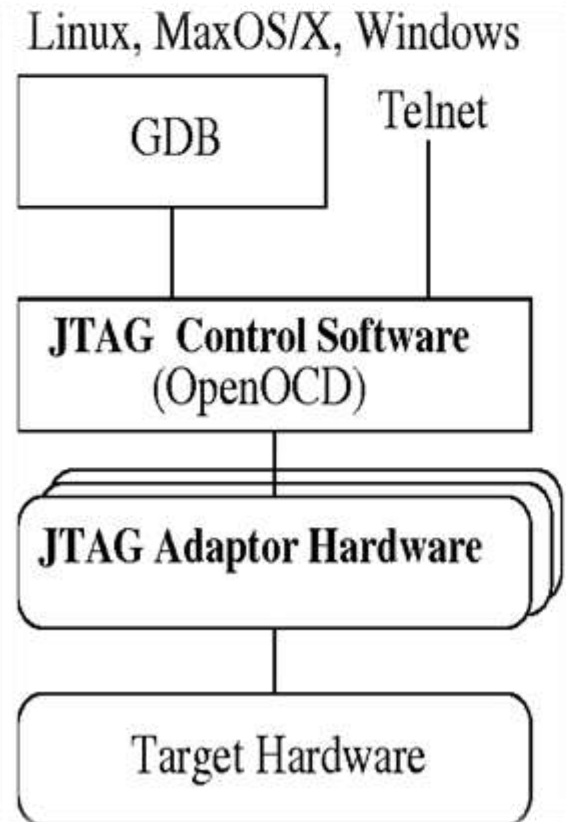
hint...hardware platform: A single system consisting of one or more components,
hart: a hardware thread in a RISC-V core.

The Debug Module implements a translation interface between abstract debug operations and their specific implementation. Debug Modules are slaves to a bus called the Debug Module Interface (DMI). The master of the bus is the Debug Transport Module(s). The Debug Module Interface can be a trivial bus with one master and one slave The DMI uses between 7 and 32 address bits. It supports read and write operations. The bottom of the address space is used for the first (and usually only) DM. Extra space can be used for custom debug devices, other cores, additional DMs, etc.

4)riscv-openocd:in the previous figure OpenOCD can be used as a translator between GDB and a RISC-V platform, as can be seen in the RISC-V

external debug specification...OpenOCD can connect to GDB in two ways: with a TCP/IP socket or with pipes (stdin/stdout),it supports many cores like ARM and RISCV to make the on-chip debugging...

What is OpenOCD?



The Open On-Chip Debugger (OpenOCD) aims to provide debugging, in-system programming and boundary-scan testing for embedded target devices.It does so with the assistance of a debug adapter, which is a small hardware module which helps provide the right kind of electrical signaling to the target being debugged. These are required since the debug host (on which OpenOCD runs) won't usually have native support for such signaling, or the connector needed to hook up to the target. Such debug adapters support one or more transport protocols, each of which involves different electrical signaling (and uses different messaging protocols on top of that signaling). There are many types of debug adapter, and little uniformity in what they are called. (There are also product naming differences.) These adapters are sometimes packaged as discrete dongles, which may generically be called hardware interface dongles. Some development boards also integrate them directly, which may let the development board connect directly to the debug host over USB (and sometimes also to power it over USB). For example, a JTAG Adapter supports JTAG signaling, and is used to communicate with JTAG compliant TAPs on your target board. A TAP is a "Test Access Port", a module which processes special instructions and data. TAPs are daisy-chained within and between chips and boards. JTAG supports debugging and boundary scan operations. There are also SWD Adapters that support Serial Wire Debug (SWD) signaling to communicate with some newer ARM cores, as well as debug adapters which support both JTAG and SWD transports. SWD supports only debugging, whereas JTAG also supports boundary scan operations.

to work with openocd you can install it on base of your linux OS like [etc] in linux UBUNTU or install it in seperate folder in any place on your hard-disc...the difference between the two ways is when you use it on terminal the command will differ you will need to call the whole build directory in the folder you installed it on the hard-disc or you can use [openocd] command if you installed it on linux base[etc]...

to use openocd you have to make the configuration file which will be used by openocd...A config file contains commands that OpenOCD will execute using its Jim-Tcl interpreter. it contain many things configs the openocd with them:1)The interface: configuration tells OpenOCD how to use the transport hardware.2) TAP declaration: A device with a JTAG interface means the device has a Test Access Port (TAP). You need to set up the active TAPs of the device by declaring them inside a configuration file.3)CPU configuration: This step gives information about the debug target.4)Starting the debug session:[init] end the configuration stages and enters the run stage,[halt] sends a halt request to the target. After these commands, the target is halted and OpenOCD is waiting for a GDB connection.5)some useful commands: Set up a maximum speed for the JTAG interface:[adapter_khz 1000],Enable error reports from GDB:

[gdb_report_data_abort enable

gdb_report_register_access_error enable].

6)RISCV-commands: there are riscv commands need to be added to the config file you can find them in the link (http://openocd.org/doc/html/Architecture-and-Core-Commands.html#RISC_002dV-Architecture)...

Simulating using RISCV simulators:

RISCV provides easy access 64 bit control and status registers to measure the program performance. One of these registers stores the clock cycle count. The cycle count registers are accessed through the RISCV instruction **csrrs**

csrrs rd, cycle, x0

Alternatively, we can use the pseudo instruction **rdcycle** to read the cycle count and store it in the destination register **rdcycle rd**

To measure the total number of cycles needed to execute a certain block in a C/C++ code we can use inline assembly to count the number of cycles using function read_cycles:

```
unsigned long read_cycles(void)
{
    unsigned long cycles;
    asm volatile ("rdcycle %0" : "=r" (cycles));
    return cycles;
}
```

Cycle count estimation. C++ function read_cycles

By inserting it before and after a program block, we can read the register content before and after the execution and compare the content:

```
#include <iostream>

unsigned long read_cycles(void)
{
        unsigned long cycles;
        asm volatile ("rdcycle %0" : "=r" (cycles));
        return cycles;
}

int main() {

        unsigned long start, end;

        start = read_cycles();

        // Code to be timed

        end = read_cycles();

        cout<<"Took "<<end-start<<" cycles to complete";

        return 0;
}
```

Cycle count estimation C++ sample test

Accurately monitoring the performance of a code block requires an FPGA or ASIC implementation of the core. Alternatively, the codes is run on a software simulator that emulates the behavior of a RISCV hardware. Some of these hardware emulators include:

1- QEMU

2- Spike

3- Verilator

1-QEMU:

By running a program on QEMU, the compiled binary of the guest architecture (RISCV) is translated to match that of the host machine (X86 for PC). By doing this, QEMU can generally produce functionally accurate results but not cycle accurate behavior.

The main advantage of QEMU is the emulation speed. QEMU can be used to boot Linux and test user space applications.

2-Spike:

Spike is the golden reference for RISCV ISA simulators. Spike produces instruction by instruction trace and models the system register for a general RISCV architecture.

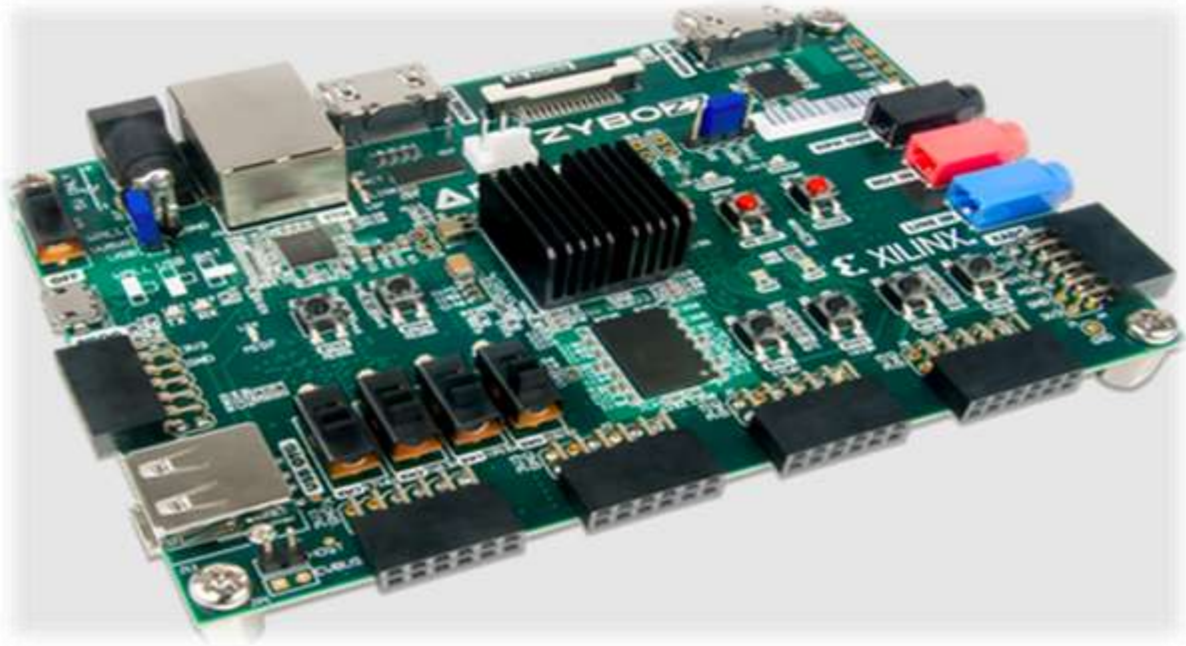
The main disadvantage of Spike is that it does not emulate a specific RISCV implementation like CVA6. Spike assumes that every instruction executes in one clock cycle whether it is an addition or a multiplication. This produces fast simulation results although generally, non-accurate cycle by cycle behavior.

3-Verilator:

Verilator works by converting the RTL of a specific RISCV implementation like CVA6 to a C++ model and produces a cyclic simulation of the core. By constructing the model, cycle accurate behavior can be achieved. CVA6 open-source repository contain a Verilator model. The main disadvantage of the verilator model is its very slow speed.

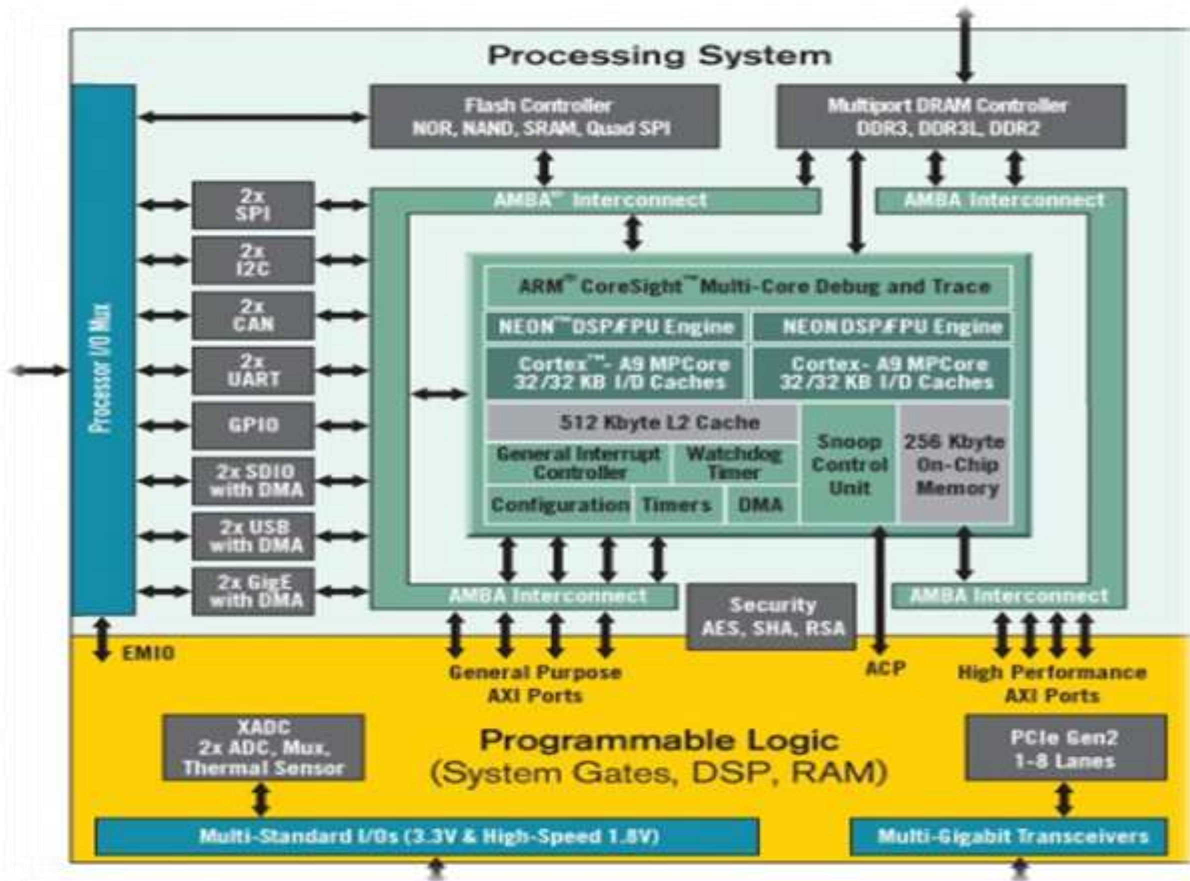
Spike is recommended for fast estimation of clock cycle count for a program block

As we want to burn it on FPGA we picked to work on ZYBO-7020



First, we will talk some little about ZYBO 7020 board to be able to understand what we did to make CVA6 applicable to it... this board's architecture is based on Zynq APSoC Architecture... The Zynq APSoC is divided into two distinct subsystems: The Processing System (PS) and the Programmable Logic (PL). The figure below shows an overview of the Zynq APSoC architecture, with the PS colored light green and the PL in yellow. Note that the PCIe Gen2 controller and multi-gigabit transceivers are not available on the Zynq-7020 ….

The PL is nearly identical to a Xilinx 7-series Artix FPGA, except that it contains several dedicated ports and buses that tightly couple it to the PS. The PL also does not contain the same configuration hardware as a typical 7-series FPGA, and it must be configured either directly by the processor or via the JTAG port. The PS consists of many components, including the Application Processing Unit (APU, which includes 2 Cortex-A9 processors), Advanced Microcontroller Bus Architecture (AMBA) Interconnect,



DDR3 Memory controller, and various peripheral controllers with their inputs and outputs multiplexed to 54 dedicated pins (called Multiplexed I/O, or MIO pins). Peripheral controllers that do not have their inputs and outputs connected to MIO pins can instead route their I/O through the PL, via the Extended-MIO (EMIO) interface. The peripheral controllers are connected to the processors as slaves via the AMBA interconnect, and contain readable/writable control registers that are addressable in the processors' memory space. The programmable logic is also connected to the interconnect as a slave, and designs can implement multiple cores in the FPGA fabric that each also contain addressable control registers. Furthermore, cores implemented in the PL can trigger interrupts to the processors and perform DMA accesses to DDR3 memory.

there are three paths we can take one "out of context" (ooc)mode, that means that the CVA6 architecture is synthetized in the FPGA fabric without consideration of the external IOs constraints, the other two paths do consider external IOs constraints.

so, we have two paths we can take to build CVA6 in ZYBO-7020 which are through Processing System (PS) or Programmable Logic (PL). that will happen if we choose to work with BRAM in PL and don't enter PS...

on the other hand, we can use DDR in the PS that path will save a lot of logic resources in th FPGA fabric actually the difference is about 400 LUTs if your work is small compared with CVA6 and the FPGA LUTs you can use BRAM and work in the PL only.

the files we worked on to make synthesis and generate the bit stream of course we have to set constrains of CVA6 on zybo-7020 and the zybo-7020 constrains like buttons and clocks ...

and the TCL files which we used on linux which lead the work on VIVado which we can work with on batch mode or with vivado GUI:

1) (Makefile): defined in it the platform we work on and the IPs we are using, and of course determine the mode we work on from three paths we talk about if ooc or the other two paths. that will lead us to second file →

2) (run_cva6_fpga): that is a TCL file which we can run it in the TCL console on VIVado GUI or in batch mode by make file that describes the process of making project and choosing the board and then call the Ips we work on from Xilinx and of course the src files of CVA6 and constrain files then determine which path you want to take from three paths ooc , PS or PL then make the synthesis and generate the utilization reports and other reports which we get from synthesis process and this file also generate the bitstream of the path you determined with makefile and make everything ready to run and program on FPGA.

To program the bitstream on the FPGA we can run with the makefile option of program after we pick our path in generating bitstream.

5.2. Interface:

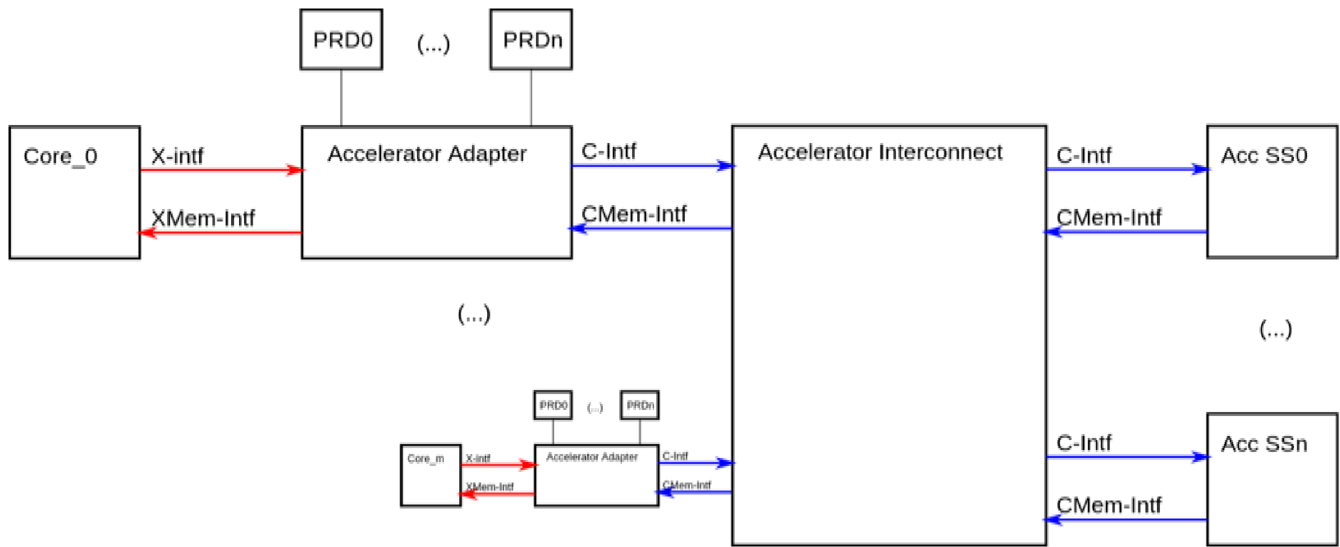
CORE-V X-Interface is a RISC-V extension interface that provides a generalized framework suitable to implement custom co-processors and ISA extensions for existing RISC-V CPU cores(4). The lightweight interface was developed to connect different PULP cores with external co-processors and provide a generalized infrastructure suitable to implement custom co-processors

The module comprises the following:

1-co-processor adapter: to communicate between core and co-processor units

2-co-processor predecoder: that provides limited decoding of instruction specific metadata necessary for correct operation

3-co-processor interconnect: in case we want to connect more than one CPU core with more than one co-processor



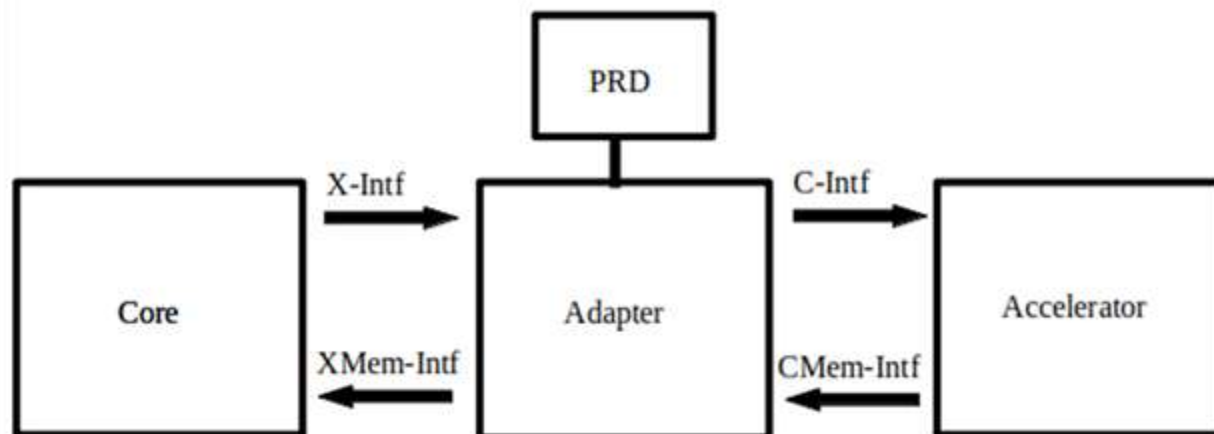
Interface infrastructure

The offloading core communicates with the co-processor adapter through the X-Intf and Xmem-Intf channels. The adapter module communicates with the co-processor through

the C-Interface and Cmem-Intf channels. Each of the 4 aforementioned channels comprises separate request and response subchannels

X-Intf	Request	Core → Adapter	Instruction offloading
	Response	Adapter → Core	Result write-back
C-Intf	Request	Adapter → co-processor	Forwarding pre-decoded instructions to co-processor
	Response	co-processor → Adapter	Response of co-processor
XMem-Intf	Request	Adapter → Core	Forwarding memory request to the core
	Response	Core → Adapter	Memory response
CMem-Intf	Request	co-processor → Adapter	Request memory operation
	Response	Adapter → co-processor	Memory response write-back

The Interconnect module is necessary in the cases where multiple cores and co-processors are communicating with each other. In our case, there's a single hardware thread communicating with our co-processor, we get rid of the interconnect module, and get rid of unnecessary signals like **hartId** necessary to route back responses in multi-core environments are discarded.



Interface between core and co-processor

Adapter:

Provides communication infrastructure between the offloading core and the target co-processor. On the core side, communication between the core and the adapter takes place through the X-Intf and XMem-Intf channels. On the co-processor side, communication takes place through the C-Intf and Cmem-Intf channels.

Pre-decoder:

The co-processor specific pre-decoder module connected to the adapter of the interface receives the off loadable instruction as input and performs simple decoding on it to extract useful metadata necessary for the offloading process. The pre-decoder module notifies the adapter on whether:

1- The instruction is valid and off loadable

2- The Instruction includes a write back to the destination register in core

3- The instruction uses source operands that needs to be sent to the co-processor

4- The instruction includes memory operations

A System Verilog struct **offload_instr_t** is defined and is used to carry the predefined pre-decoder response. The struct is then declared as a parameter in an independent file and is used to construct the pre-decoder

```
package acc_cnn_pkg;

parameter int unsigned NumInstr=110;
parameter acc_pkg::offload_instr_t Instr[110] = '{
  '{
    instr_data: 32'b 0000000_00000_00000_111_00000_0001011, // RST
    instr_mask: 32'b 1111111_00000_00000_111_00000_1111111,
    prd_rsp : '{
      p_accept : 1'b1,
      p_writeback : 2'b00,
      p_is_mem_op : 1'b0,
      p_use_rs : 3'b000
    }
  },
  '{
    instr_data: 32'b 0000100_00000_00000_111_00000_0001011, // Store in MB
    instr_mask: 32'b 1111111_00000_00000_111_00000_1111111,
    prd_rsp : '{
      p_accept : 1'b1,
      p_writeback : 2'b00,
      p_is_mem_op : 1'b1,
      p_use_rs : 3'b011
    }
  },

  //////////
  //Rest of the custom instructions
  //////////
  //////////
```

Predecoder predefined response

X-Intf:

Communication between the core and the adapter is regulated using a simple valid/ready handshake protocol.

Once an offloadable instruction is encountered within the core, the valid signal is asserted and the instruction data is examined by the pre-decoder. The pre-decoder verifies that the instruction is acceptable by the target co-processor. The pre-decoder asserts that it is ready to accept the instruction once the offloading core verifies a valid register content on operand channels, and that there is no writebacks that targets the destination register of the operation or currently pending memory operations in the core pipeline.

The X-Intf response channel carries the instruction response data alongside the destination register address back to the core. Additionally it informs the co-processor whether an error has occurred.

C-Intf:

The C-Intf request channel forwards the instruction data and the operands to the co-processor. Similarly, the response channel forwards the response data form the co-processor and back to the adapter.

Cmem-Intf:

Once an instruction has been decoded inside the co-processor and identified as a load/store operation, a request is sent to the adapter and the core to access the memory through the load store unit local to the core. The Cmem-Intf request channel forwards the target address, the write data, the request type (read/write).

The response channel forwards the response of the read operations and the whether a transaction can be considered as a success

Xmem-Intf:

The Xmem-Intf channel forwards the memory requests and responses between the core and the co-processor on the core side

Operating Principle:

The offloading core initiates the offloading process by signaling the validity of a co-processor custom instruction to the interface adapter. The instruction predecoder provides limited decoding capabilities and notifies the adapter and the offloading core on whether:

a) Instruction requires source operands

b) Instruction involves writeback to offloading core

c) Instruction involves a memory operation

The Core awaits until the source registers rs1 and rs2, addressed by instr_data[19:15] and instr_data[24:20] of the custom instruction are valid and no other preceding instruction can modify them.

If the offloaded instruction involves a write-back operation to a destination register addressed by instr_data[19:15], then we make sure there's no outstanding writeback operation to this register in the core pipeline to avoid write after write hazard. This done by asserting the signal x.q_rd_clean by the offloading core. When the source operands are valid and the destination register is clean, the adapter asserts its ready signal initiating the transaction. No register writes back in our defined instruction set.

If the instruction process involves a memory operation, the offloading core is not allowed to commit any new instruction until the status signal adapter_mem_pending is deasserted. Furthermore, no instruction offloading is further allowed while there's a pending memory operation. It is required to make sure that there's no preceding instruction in the core pipeline that can raise an exception that might require the flushing of the core pipeline. This is due to the fact that there's way to retract an offloaded instruction and roll back an architectural state local to the co-processor.

The interface defines two modes of memory operation, internal and external mode.

1- Internal mode: Control of the core load store unit is handled to the adapter to send requests and receive responses

2-External mode: Memory operations are performed through separately implemented co-processor private memory ports

Furthermore, there are two modes of memory accessing that vary according to the behavior upon encountering access faults:

1- Synchronous operations: Memory operations are performed in lockstep with the rest of the instrction stream. Access faults raise an exception and are trapped by the core.

2- Speculative synchronous operations: Memory operations are performed in lockstep with the rest of the instrction stream. CPU pipeline does not raise an exception upon encountering access fault

Speculative signaling is done through the co-processor by asserting the signal cmem.q_spec.

Our memory transactions will involve synchronous internal mode operations

Memory Transactions:

Once the offloaded instruction is marked as a memory operation by the instruction pre-decoder, the adapter_mem_pending status signal is asserted. No instructions are allowed to be committed in the core pipeline, no new offload requests are allowed

The custom co-processor receives the instruction and decodes it into a read/write operation and initiates a memory request through the Cmem-intf. Many requests transactions take place between the co-processor and the adapter, the last of which has its cmem.q_endoftransaction is asserted. Consequently, the adapter_mem_pending signal is deasserted and the core pipeline is unblocked allowing for new commits and custom instrction offloading

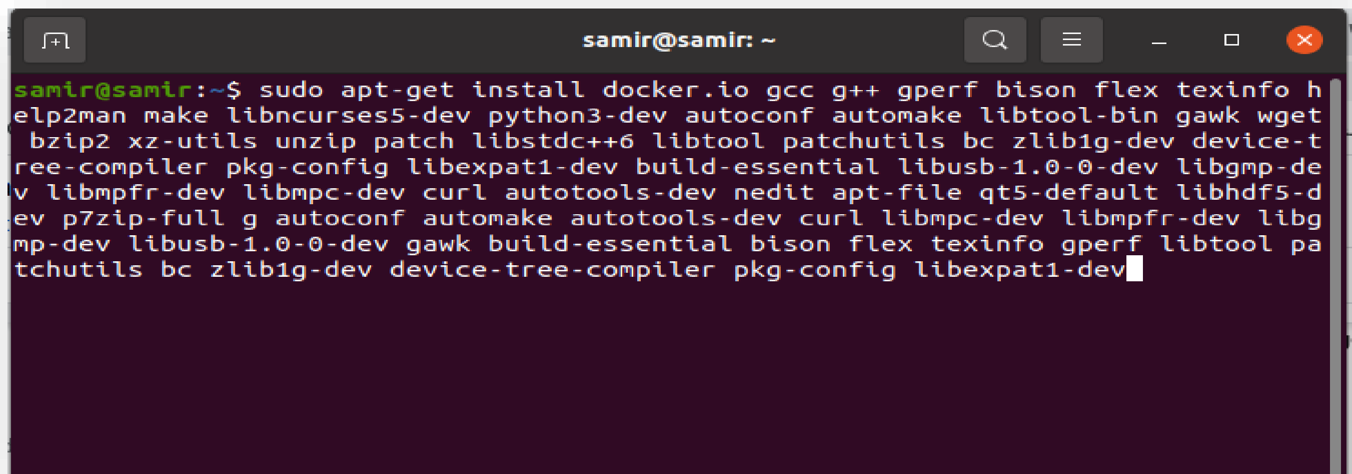
The control of the load/store unit of the core pipeline is handled to the interface adapter and memory requests are initiated through the Xmem-intf between the adapter and the load/store unit. If an access fault was encountered, it raises an exception in the core pipeline and the co-processor is notified through the memory response channel to roll-back the local architectural state of the co-processor.

APPENDIX

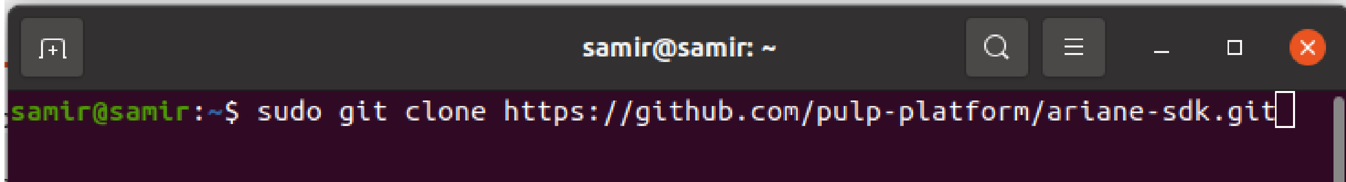
we worked on linux OS: **(64b) UBUNTU 20.04** and VIVADO **version 2018.2**
Hint:you can istall linux specially UBUNTU alongside with windows as windows is the main OS of the LABTOP.

first, there are some packages should be installed on UBUNTU before start dealing with RISCV you can install them by this command

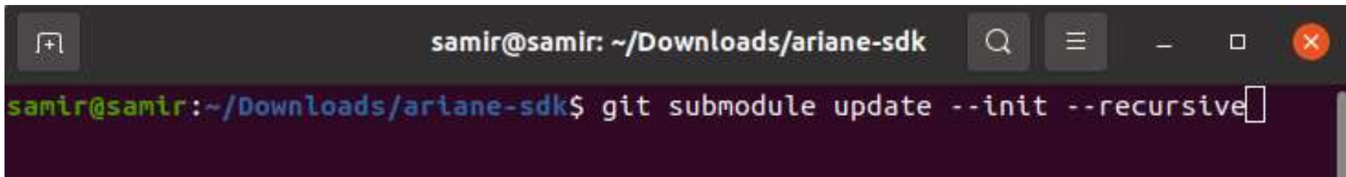
[sudo apt-get install docker.io gcc g++ gperf bison flex texinfo help2man make libncurses5-dev python3-dev autoconf automake libtool-bin gawk wget bzip2 xz-utils unzip patch libstdc++6 libtool patchutils bc zlib1g-dev device-tree-compiler pkg-config libexpat1-dev build-essential libusb-1.0-0-dev libgmp-dev libmpfr-dev libmpc-dev curl autotools-dev nedit apt-file qt5-default libhdf5-dev p7zip-full g autoconf automake autotools-dev curl libmpc-dev libmpfr-dev libgmp-dev libusb-1.0-0-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev device-tree-compiler pkg-config libexpat1-dev] of course without the []



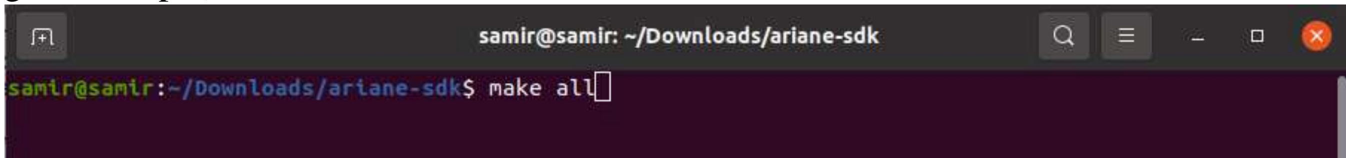
we installed all tools we talked about in the two ways by ariane-SDK and evry tool as standalone so as said about using ariane-SDK will clone it with this command (hint:clonning means that you download the whole repository by command on linux named [git clone])... [sudo git clone https://github.com/pulp-platform/ariane-sdk.git]



after finishing the cloning you can enter into the folder of repository of ariane-SDK and open the terminal into its folder which usually the cloning put it in [Downloads]foder of the UBUNTU... now to gurantee every thing is updated into the repository use this command



now we want to determine a specific place for the build of RISCV-tools to be the default on your linux environment you can use this command[export RISCV=RISCV=/home/samir/Downloads/ariane-sdk/install] (pic 3 export RISCV and give example) and then run the command



[make all] to begin the installation...if worked properly that's will install tools of [riscv-fesvr,riscv-gnu-toolchain,riscv-isa-sim,riscv-pk] which are exist in the makefile as we said... but if there is any error during the installing process that is because a problem in the cloning process so there are two solutions we tell them all and will recommend one ...

first solution that you can remove the [riscv-fesvr,riscv-gnu-toolchain,riscv-isa-sim,riscv-pk] folders from the ariane-SDK repository that you cloned before on your computer and clone every one at alone…while the terminal is opened from inside the ariane-SDK folder after cloning all of them again into the ariane-SDK after that you can rerun the first command.

the other solution is to install [riscv-fesvr,riscv-gnu-toolchain,riscv-isa-sim,riscv-pk] one by one with the same steps cloning and make.

we did the two sollution after many of tries to solve the errors and I deeply recommend the first solution.

After solve the problem if existed you can rerun the commands we said before after the clonning...

After all of that complested and get installed correctly without any errors...

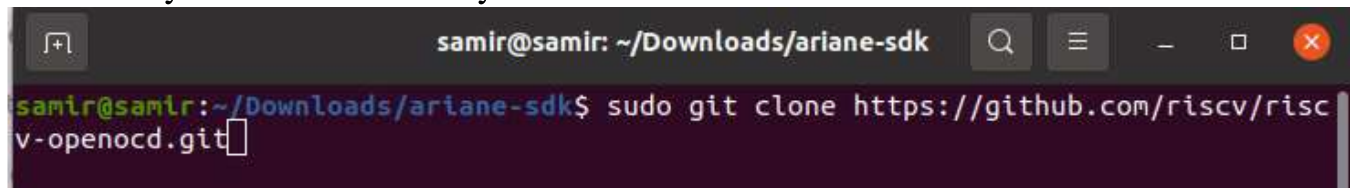
At the end of this precess you should edit a file named (.bashrc or .zshrc) to Add $RISCV/bin to your path in order to later make use of the installed tools and permanently export $RISCV. that will be happened go to folder [Home] in your linux and press (ctrl+h) in you keyboard that will show you the hidden files in you environment now you can find these files we want (.bashrc or .zshrc) then open it and go to the end of the file and write the

```
118
119 source /tools/Vivado/2018.2/settings64.sh
120 export RISCV=/home/samir/Downloads/ariane-sdk/install
121 export PATH=$PATH:$RISCV/bin
```

then save and close...this like set this path of setup of RISCV as default for every terminal will be opened in any place in the computer(hint: that happened when set up of vivado too to be able to open it from amy where)...

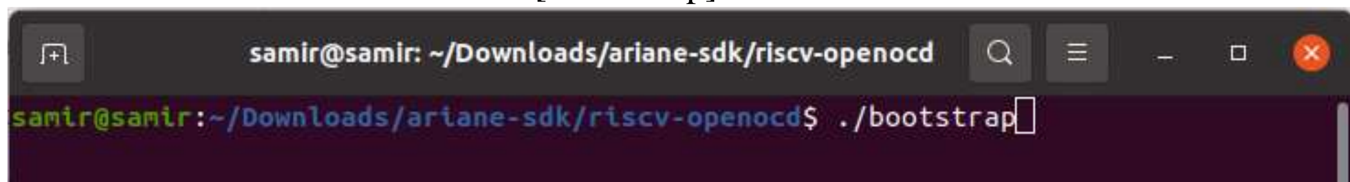
now we go to install openocd tool...you shold clone it too as it's a complete repository with configuration files and makefile wanted to be installed first to be able use its command as said before so:

first of all you should clone it by that



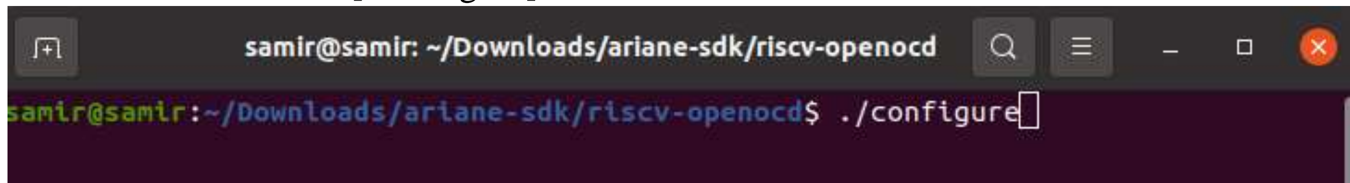
[sudo git clone https://github.com/riscv/riscv-openocd.git]

then open the folder of the repository after clonned on your computer and launch a terminal from it then run:command[./bootstrap]



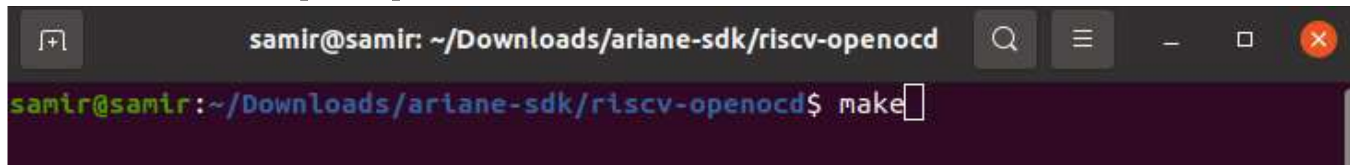
this will access the file named bootstrap in the repository you can open it and read its content.

then run this command[./configure]



this too will access the file named configure in the repository you can open it and read its content.

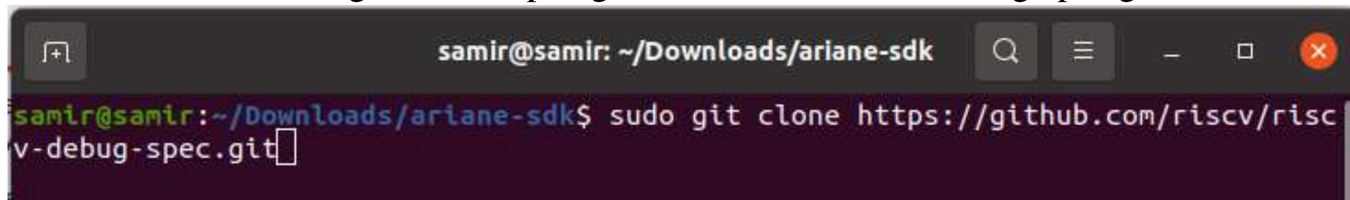
then run command [make]



then [sudo make install)]

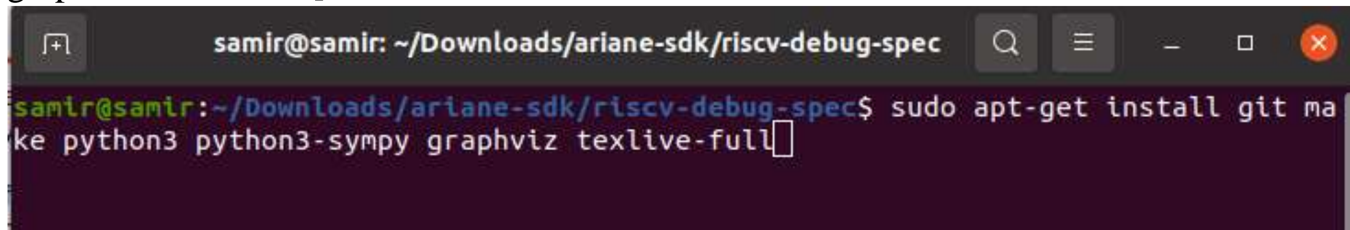
about the repository of riscv-debug-spec which is important in debugging if external or not...

clone it as usual [sudo git clone https://github.com/riscv/riscv-debug-spec.git]



and open the terminal from inside its folder.

then install these packages [sudo apt-get install git make python3 python3-sympy graphviz texlive-full]



then [make]

```
samir@samir: ~/Downloads/ariane-sdk/riscv-debug-spec
samir@samir:~/Downloads/ariane-sdk/riscv-debug-spec$ make
```

There are two other interesting make targets:

[make debug_defines.h] creates a C header file containing constants for addresses and fields of all the registers and abstract commands.

[make chisel] creates scala files for DM registers and abstract commands with the same information.

that was the main tools which were installed before working on RISCV. if you faced any error I don't mention it you shold google it to know how to solve it or revise any something was done wrong...

About the results of synthesis and implementation of the CVA6 on ZYBO-7020...

These are the results of synthesis and implementation in the path of PS (DDR):

```
Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.
-------------------------------------------------------------------------------
| Tool Version : Vivado v.2018.2 (lin64) Build 2258646 Thu Jun 14 20:02:38 MDT 2018
| Date         : Thu Apr  8 10:35:10 2021
| Host         : samir running 64-bit Ubuntu 20.04.1 LTS
| Command      : report_utilization -file cva6_zybo_z7_20_utilization_placed.rpt -pb cva6_zybo_z7_20_utilization_placed.pb
| Design       : cva6_zybo_z7_20
| Device       : 7z020clg400-1
| Design State : Fully Placed
-------------------------------------------------------------------------------

Utilization Design Information

Table of Contents
-----------------
1. Slice Logic
1.1 Summary of Registers by Type
2. Slice Logic Distribution
3. Memory
4. DSP
5. IO and GT Specific
6. Clocking
7. Specific Feature
8. Primitives
9. Black Boxes
10. Instantiated Netlists

1. Slice Logic
--------------

+----------------------------+-------+-------+-----------+-------+
|          Site Type         |  Used | Fixed | Available | Util% |
+----------------------------+-------+-------+-----------+-------+
| Slice LUTs                 | 26258 |     0 |     53200 | 49.36 |
|LUT as Logic                | 26181 |     0 |     53200 | 49.21 |
|LUT as Memory               |    77 |     0 |     17400 |  0.44 |
|LUT as Distributed RAM      |    48 |     0 |           |       |
|   LUT as Shift Register    |    29 |     0 |           |       |
| Slice Registers            | 22369 |     0 |    106400 | 21.02 |
|   Register as Flip Flop    | 22369 |     0 |    106400 | 21.02 |
|   Register as Latch        |     0 |     0 |    106400 |  0.00 |
| F7 Muxes                   |  1382 |     0 |     26600 |  5.20 |
| F8 Muxes                   |    73 |     0 |     13300 |  0.55 |
+----------------------------+-------+-------+-----------+-------+
```

```
Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.
-------------------------------------------------------------------------------
| Tool Version : Vivado v.2018.2 (lin64) Build 2258646 Thu Jun 14 20:02:38 MDT 2018
| Date         : Thu Apr  8 10:19:02 2021
| Host         : samir running 64-bit Ubuntu 20.04.1 LTS
| Command      : report_utilization -file cva6_zybo_z7_20_utilization_synth.rpt -pb cva6_zybo_z7_20_utilization_synth.pb
| Design       : cva6_zybo_z7_20
| Device       : 7z020clg400-1
| Design State : Synthesized
-------------------------------------------------------------------------------

Utilization Design Information

Table of Contents
-----------------
1. Slice Logic
1.1 Summary of Registers by Type
2. Memory
3. DSP
4. IO and GT Specific
5. Clocking
6. Specific Feature
7. Primitives
8. Black Boxes
9. Instantiated Netlists

1. Slice Logic
--------------

+-------------------------+-------+-------+-----------+-------+
|        Site Type        |  Used | Fixed | Available | Util% |
+-------------------------+-------+-------+-----------+-------+
| Slice LUTs*             | 26588 |     0 |     53200 | 49.98 |
|   LUT as Logic          | 26584 |     0 |     53200 | 49.97 |
|   LUT as Memory         |     4 |     0 |     17400 |  0.02 |
|     LUT as Distributed RAM |  0 |     0 |           |       |
|     LUT as Shift Register  |  4 |     0 |           |       |
| Slice Registers         | 22967 |     0 |    106400 | 21.59 |
|   Register as Flip Flop | 22967 |     0 |    106400 | 21.59 |
|   Register as Latch     |     0 |     0 |    106400 |  0.00 |
| F7 Muxes                |  1382 |     0 |     26600 |  5.20 |
| F8 Muxes                |    73 |     0 |     13300 |  0.55 |
+-------------------------+-------+-------+-----------+-------+
```

These are the results of synthesis and implementation in the path of PL (BRAM):

```
Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.
------------------------------------------------------------------------------------------
| Tool Version : Vivado v.2018.2 (lin64) Build 2258646 Thu Jun 14 20:02:38 MDT 2018
| Date         : Thu Apr  8 12:35:05 2021
| Host         : samir running 64-bit Ubuntu 20.04.1 LTS
| Command      : report_utilization -file cva6_zybo_z7_20_utilization_placed.rpt -pb cva6_zybo_z7_20_utilization_placed.pb
| Design       : cva6_zybo_z7_20
| Device       : 7z020clg400-1
| Design State : Fully Placed
------------------------------------------------------------------------------------------

Utilization Design Information

Table of Contents
-----------------
1. Slice Logic
1.1 Summary of Registers by Type
2. Slice Logic Distribution
3. Memory
4. DSP
5. IO and GT Specific
6. Clocking
7. Specific Feature
8. Primitives
9. Black Boxes
10. Instantiated Netlists

1. Slice Logic
--------------

+----------------------------+-------+-------+-----------+-------+
|          Site Type         |  Used | Fixed | Available | Util% |
+----------------------------+-------+-------+-----------+-------+
| Slice LUTs                 | 26696 |     0 |     53200 | 50.18 |
|   LUT as Logic             | 26616 |     0 |     53200 | 50.03 |
|   LUT as Memory            |    80 |     0 |     17400 |  0.46 |
|     LUT as Distributed RAM |    52 |     0 |           |       |
|     LUT as Shift Register  |    28 |     0 |           |       |
| Slice Registers            | 22452 |     0 |    106400 | 21.10 |
|   Register as Flip Flop    | 22452 |     0 |    106400 | 21.10 |
|   Register as Latch        |     0 |     0 |    106400 |  0.00 |
| F7 Muxes                   |  1448 |     0 |     26600 |  5.44 |
| F8 Muxes                   |    73 |     0 |     13300 |  0.55 |
+----------------------------+-------+-------+-----------+-------+
```

```
Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.
------------------------------------------------------------------------------------------
| Tool Version : Vivado v.2018.2 (lin64) Build 2258646 Thu Jun 14 20:02:38 MDT 2018
| Date         : Thu Apr  8 12:17:47 2021
| Host         : samir running 64-bit Ubuntu 20.04.1 LTS
| Command      : report_utilization -file cva6_zybo_z7_20_utilization_synth.rpt -pb cva6_zybo_z7_20_utilization_synth.pb
| Design       : cva6_zybo_z7_20
| Device       : 7z020clg400-1
| Design State : Synthesized
------------------------------------------------------------------------------------------

Utilization Design Information

Table of Contents
-----------------
1. Slice Logic
1.1 Summary of Registers by Type
2. Memory
3. DSP
4. IO and GT Specific
5. Clocking
6. Specific Feature
7. Primitives
8. Black Boxes
9. Instantiated Netlists

1. Slice Logic
--------------

+----------------------------+-------+-------+-----------+-------+
|          Site Type         |  Used | Fixed | Available | Util% |
+----------------------------+-------+-------+-----------+-------+
| Slice LUTs*                | 26590 |     0 |     53200 | 49.98 |
|   LUT as Logic             | 26588 |     0 |     53200 | 49.98 |
|   LUT as Memory            |     2 |     0 |     17400 |  0.01 |
|     LUT as Distributed RAM |     0 |     0 |           |       |
|     LUT as Shift Register  |     2 |     0 |           |       |
| Slice Registers            | 22866 |     0 |    106400 | 21.49 |
|   Register as Flip Flop    | 22866 |     0 |    106400 | 21.49 |
|   Register as Latch        |     0 |     0 |    106400 |  0.00 |
| F7 Muxes                   |  1382 |     0 |     26600 |  5.20 |
| F8 Muxes                   |    73 |     0 |     13300 |  0.55 |
+----------------------------+-------+-------+-----------+-------+
```

Conclusion

We have designed fully reconfigurable co-processor containing critical operations of CNN algorithms as computing module as convolution, pool, add and ReLU modules, They Implement the Algorithm And take the most Computing power.

Function verification is a very essential phase to verify any RTL to test its function correctly, is still one the most challenging activities in digital system development, the purpose of a testbench is to determine the correctness of the design under test (DUT).

Generating stimulus are most important step where this step generate the inputs which expresses a certain feature test, Random stimulus is crucial for exercising complex designs. A directed test finds the bugs you expect to be in the design, while a random test can find bugs you never anticipated.

Zybo 7020 was used for CVA6 core implementation.

Core V X-Interface was used connected to simple custom co-processor Modifications in core pipeline are necessary for the interface.

Future Work

Regarding the Co-processor, Backing to the design, the most of area can be customized to used ROMs or LUTs area, regardless the Area optimization needed in ROM size reduction as mentioned above, but the good thing with ROMs is that it may be multi-input multi-output ROM which means it can serve for lots of PE modules, which encourages to use lots of data-paths served by same ROM.

As the ROM can be considered as Overhead in Area must tolerated and then the performance will increase as well as the number of data-paths increases as shown in Figure 87

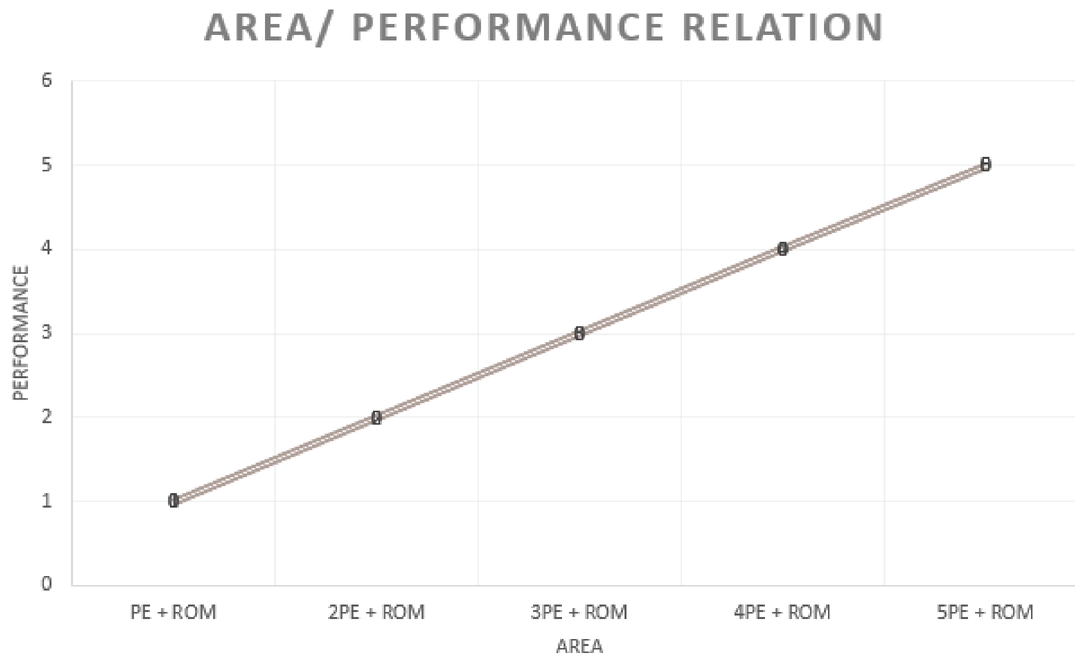


Figure 87 Area vs performance

And that is the importance of the field of instruction number, as each data-path is responsible to execute one instruction and can't be make another one although it ends the operation, the controller can be used for more than one but to handle more than signal in same time.

Regarding the Verification Path Until now all the components of the UVM testbench environment are coded except they need to be tested and get the expected results this will be our future work.

Regarding the RISC-V, Modifications in the core pipeline in ID, issue, and EX stages Connect the core with simple co-processor(done) for testing Modify the compiler to accept the new instructions and test simple programs Interface with the CNN accelerator…

References

[1] Sainath, T.N.; Kingsbury, B.; Saon, G.; Soltau, H.; Mohamed, A.R.; Dahl, G.; Ramabhadran, B. Deep Convolutional Neural Networks for Large-scale Speech Tasks. Neural Networks 2015, 64, 39–48. [CrossRef]

[2] Zhang, C.; Li, P.; Sun, G.; Guan, Y.; Xiao, B.; Cong, J. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 15 February 2015.

[3] Sze, V.; Chen, Y.H.; Yang, T.J.; Emer, J.S. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. Proc. IEEE 2017, 105, 2295–2329. [CrossRef]

[4] Parashar, A.; Rhu, M.; Mukkara, A.; Puglielli, A.; Venkatesan, R.; Khailany, B.; Emer, J.; Keckler, S.W.; Dally, W.J. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In Proceedings of the 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA), Toronto, ON, Canada, 24–28 June 2017.

[5] Chi, P.; Li, S.; Xu, C.; Zhang, T.; Zhao, J.; Liu, Y.; Wang, Y.; Xie, Y. PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory. In Proceedings of the 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), Seoul, Korea, 18–22 June 2016.

[6] Hardieck, M.; Kumm, M.; Möller, K.; Zipf, P. Reconfigurable Convolutional Kernels for Neural Networks on FPGAs. In Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, New York, NY, USA, 24–26 February 2019.

[7] Bettoni, M.; Urgese, G.; Kobayashi, Y.; Macii, E.; Acquaviva, A. A Convolutional Neural Network Fully Implemented on FPGA for Embedded Platforms. In Proceedings of the 2017 New Generation of CAS (NGCAS), Genova, Italy, 6–9 September 2017.

[8] Liu, B.; Zou, D.; Feng, L.; Feng, S.; Fu, P.; Li, J.B. An FPGA-Based CNN Accelerator Integrating Depthwise Separable Convolution. Electronics 2019, 8, 281. [CrossRef]

[9] Kurth, A.; Vogel, P.; Capotondi, A.; Marongiu, A.; Benini, L. HERO: Heterogeneous Embedded Research Platform for Exploring RISC-V Manycore Accelerators on FPGA. In Proceedings of the Workshop on Computer Architecture Research with RISC-V (CARRV), Boston, MA, USA, 14 October 2017.Electronics 2020, 9, 1005 19 of 19

[10] Matthews, E.; Shannon, L. Taiga: A Configurable RISC-V Soft-Processor Framework for Heterogeneous Computing Systems Research. In Proceedings of the Workshop on Computer Architecture Research with RISC-V (CARRV), Boston, MA, USA, 14 October 2017.

[11] Ge, F.; Wu, N.; Xiao, H.; Zhang, Y.; Zhou, F. Compact Convolutional Neural Network Accelerator for IoT Endpoint SoC. Electronics 2019, 8, 497. [CrossRef]

[12] “Difference between ANN, CNN and RNN.” GeeksforGeeks, 28 June 2020.

[13] Recurrent Neural Network (RNN). https://docs.paperspace.com/machine-learning/wiki/recurrent-neural-network-rnn.

[14] “ANN vs CNN vs RNN | Types of Neural Networks.” Analytics Vidhya, 17 Feb. 2020.

[15] Machine Learning versus Deep Learning - Studytonight.

[16] LeNet-5 - A Classic CNN Architecture. Muhammad Rizwan on October 16, 2018. datasciencecentral

[17] Paper: Gradient-Based Learning Applied to Document Recognition , published by Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner in 1998(Proceedings of the IEEE).

[18] Ehsan Fathi, Babak Maleki Shoja, in Handbook of Statistics, 2018.

[19] Max-Pooling / Pooling - Computer Science Wiki.

[20] https://github.com/csukuangfj/OpenCNN.git

[21] “What Are Max Pooling, Average Pooling, Global Max Pooling and Global Average Pooling?” MachineCurve, 30 Jan. 2020.

[22] “Activation Functions in Neural Networks.” I2tutorials, 28 Sept. 2020.

[23] https://www.quora.com/What-is-the-advantage-of-UVM-over-systemverilog

[24] https://github.com/openhwgroup/cva6

[25] The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-ready 1.7GHz 64bit RISC-V Core in 22nm FDSOI Technology, Florian Zaruba, Student Member, IEEE, and Luca Benini, Fellow, IEEE

[26] The RISC-V Instruction Set Manual Volume I: Unprivileged ISA Document Version 20191214-draft

[27] https://github.com/openhwgroup/core-v-xif/blob/main/doc/co-processor-predecoder.md

[28] https://reference.digilentinc.com/programmable-logic/zybo-z7/reference-manual

[29] https://github.com/anouar-saliheddine/cva6

[30] https://github.com/ThalesGroup/pulpino-compliant-debug/blob/pulpino-dbg/doc/riscv-debug-notes/pdfs/gnu-tools.pdf

[31] https://github.com/pulp-platform/ariane-sdk

[32] https://github.com/riscv/riscv-debug-spec/blob/master/riscv-debug-stable.pdf

[33] https://github.com/ThalesGroup/pulpino-compliant-debug/blob/pulpino-dbg/doc/riscv-debug-notes/pdfs/openocd.pdf

[34] Högl, H. and D. Rath. "Open On-Chip Debugger – OpenOCD –." (2006).

[35] http://openocd.org/doc/pdf/openocd.pdf

[36] http://openocd.org/doc/html/Architecture-and-Core-Commands.html#RISC_002dV-Architecture

[37] https://www.doulos.com/httpswwwdouloscomknowhow/systemverilog/uvm/uvm-verification-primer/

[38] https://accellera.org/images/downloads/standards/uvm/UVM_1.1_Class_Reference_Final_06062011.pdf

[39] https://github.com/raysalemi/uvmprimer

[40] Functional Verification of Narrow-band IOT Physical Layer Uplink Transmitter GP2020