VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Logic Design Project (CO3091)

FPGA-Based Implementation of a 2D Convolutional Layer

Advisors:	Assoc. Prof. Trần Ngọc Thịnh
	B.Sc. Huỳnh Phúc Nghị
Students:	Đặng Hoàng Gia - 2153312
	Nguyễn Hữu Hào - 2153327

HO CHI MINH CITY, December 2023

Contents

1 Acknowledgements **2**

2 Member list & Workload **2**

3 Introduction **3**

4 Theory **4**
- 4.1 Convolutional Neural Network 4
- 4.2 Field-Programmable Gate Array 6
- 4.3 2D convolutional layer 6

5 Related Work **8**

6 Implementation **10**
- 6.1 Architecture 10
  - 6.1.1 Overview 10
  - 6.1.2 Floating point representation 15
  - 6.1.3 Floating Point Multiplier 15
  - 6.1.4 Floating Point Adder 17
  - 6.1.5 Multiplier-Accumulator 19
  - 6.1.6 Checker 20
  - 6.1.7 Convolver 20
  - 6.1.8 Controller 20
- 6.2 System Performance Evaluation 23
  - 6.2.1 Throughput 23
  - 6.2.2 Fault-tolerance 25
  - 6.2.3 Pipelining 27
  - 6.2.4 Resource Utilization 28
  - 6.2.5 Real-time Performance 30
  - 6.2.6 Future Work 30

7 Demonstration & Testing **31**
- 7.1 Int8 31
- 7.2 Float32 31
- 7.3 Parallelism 31
- 7.4 Partial Reset 32

8 Conclusion **33**

1 Acknowledgements

To our families and friends who are always supportive of us.
To Assoc. Prof. Trần Ngọc Thịnh and B.Sc.Huỳnh Phúc Nghị who devoted their time to helping us complete this project.
Thank you all.

2 Member list & Workload

No.	**Fullname**	**Student ID**	**Tasks**	**Percentage of work**
1	Nguyễn Hữu Hào	2153327	Research, Design, Implement, Test, Report	60%
2	Đặng Hoàng Gia	2153312	Research, Report	40%
Total				100%

3 Introduction

In the realm of Convolutional Neural Networks (CNNs), the design of acceleration blocks aims to mitigate the memory and computational demands while optimizing the network's inference accuracy. This research delves into the implementation of two-dimensional Convolutional Layers (2D Convolutional layers) on Field-Programmable Gate Arrays (FPGAs). The primary objective is to simulate a convolutional block on an FPGA, subsequently comparing its performance with that on a Personal Computer (PC). Through this investigation, the study seeks to assess the efficiency of FPGA implementations in contrast to PC counterparts when executing convolutional operations, providing valuable insights into the potential benefits of FPGA acceleration in the context of 2D Convolutional layers.

4 Theory

4.1 Convolutional Neural Network

Convolutional Neural Networks (CNNs) constitute a cornerstone in deep learning theory, specifically designed for processing grid-like data like images and video. The canonical architecture encompasses convolutional layers, where learnable filters convolve across input data, extracting hierarchical and spatial features. Subsequent pooling layers facilitate downsampling, enhancing translation invariance. These are often followed by fully connected layers and non-linear activation functions to model complex relationships. The complete suite of layers typically includes input layers, convolutional layers, pooling layers, fully connected layers, and output layers, collectively orchestrating the network's ability to learn intricate patterns and representations for diverse tasks such as image recognition and object detection.

The inception of Convolutional Neural Networks (CNNs) marked a significant milestone in neural network architecture, with the Neocognitron standing as a pioneering precursor to this revolutionary development. The Neocognitron paper introduced groundbreaking concepts such as feature extraction, pooling layers, and the integration of convolution within a neural network for recognition or classification purposes. Inspired by the visual nervous system of vertebrates, the Neocognitron structured its network with alternating layers of S-cells (simple cells or lower-order hypercomplex cells) and C-cells (complex cells or higher-order hypercomplex cells). This arrangement facilitated the repeated process of feature extraction by S-cells and positional shift tolerance by C-cells(see Figure 1). The network's overarching design, resembling the visual nervous system, enabled the gradual integration of local features into more global representations. Utilized for tasks like handwritten (Japanese) character recognition, the Neocognitron laid the foundation for subsequent Convolutional Neural Networks, influencing their application in diverse pattern recognition tasks.

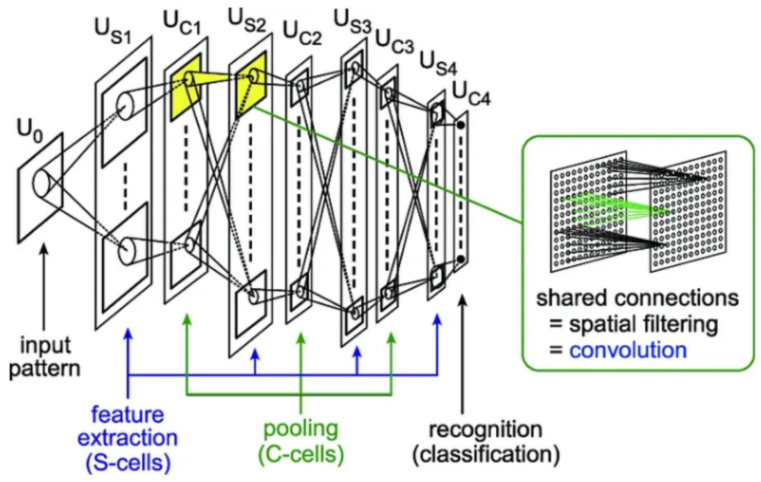

U0
US1
UC1
US2
UC2
US3
UC3
US4
UC4
input pattern
feature extraction (S-cells)
pooling (C-cells)
recognition (classification)
shared connections = spatial filtering = convolution

Figure 1: Neocognitron Architecture

The idea of the convolution operation in a Conv layer might derive from the signal processing study. From two signals (or two functions), denoted by f and g, we create the third function $(f * g)$. This derived function intricately expresses the transformation of one function's shape under the influence of the other, emphasizing the interplay and modification of their respective mathematical structures (Figure 2).

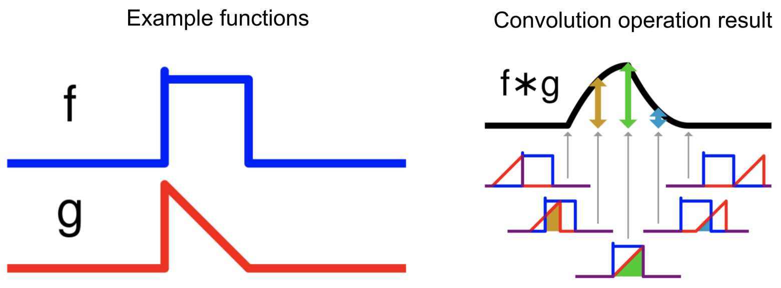


Figure 2: The functions f and g represent signal pulses, and their outcomes are determined through the convolution operation.

4.2 Field-Programmable Gate Array

Field-Programmable Gate Arrays (FPGAs) are integral to hardware accelerator fields, offering unique advantages for diverse computational applications. Their inherent support for parallel processing, coupled with reconfigurability, allows dynamic adaptation to specific tasks, enhancing performance and throughput. FPGAs excel in low-latency scenarios crucial for real-time data processing, while their energy efficiency addresses power consumption concerns. Customization for specific algorithms and adaptability to evolving standards make FPGAs well-suited for dynamic accelerator environments. They achieve high throughput by parallelizing tasks, and their integration with CPUs and GPUs in heterogeneous computing platforms provides a synergistic approach to enhance overall system performance. In essence, FPGAs play a pivotal role in accelerating computations, providing a flexible and efficient solution for a wide range of applications in modern computing architectures.

The parallel computing capabilities intrinsic to FPGAs align with the computational attributes of convolutional neural networks (CNNs). Concurrently, the reprogrammable nature of FPGAs proves conducive to accommodating the variable network structures inherent in neural networks. Consequently, the development of CNNs on FPGA platforms has garnered considerable scientifically interest.

4.3 2D convolutional layer

As mentioned above, convolution is a mathematical operation involving two functions with a real-valued variable. In essence, the operation * is essentially the integration of the product of two functions. However, in some computational-related fields, the formula can be represented as discrete function, as others called discrete convolution:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a)$$

In this term, the variable x represents the **input**, while the variable w represents the **kernel** or **filter**. The output s is often called the **output**.

In most applications, the input is a multidimensional array of data, and the kernel is a multidimensional array of parameters. For instance, in image processing tasks, we use a 128×128 image as the input. To perform convolutional calculations, we typically create kernels with parameters. Additionally, we assume that the outer space of these multidimensional arrays is set to zero to ensure consistent calculation results.

Hence, the 2D convolutional layer can be represented as the convolutions between two-dimentional image I as our input and two-dimentional kernel K with parameters:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m,j-n)$$

However, in machine learning aspects, we often applied the commutative property of convolution and also the equivalent between cross-correlation and convolution in order to simplified the calculation. Therefore, we deploy the formula (without flipping the kernel):

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m,j+n)K(m,n)$$

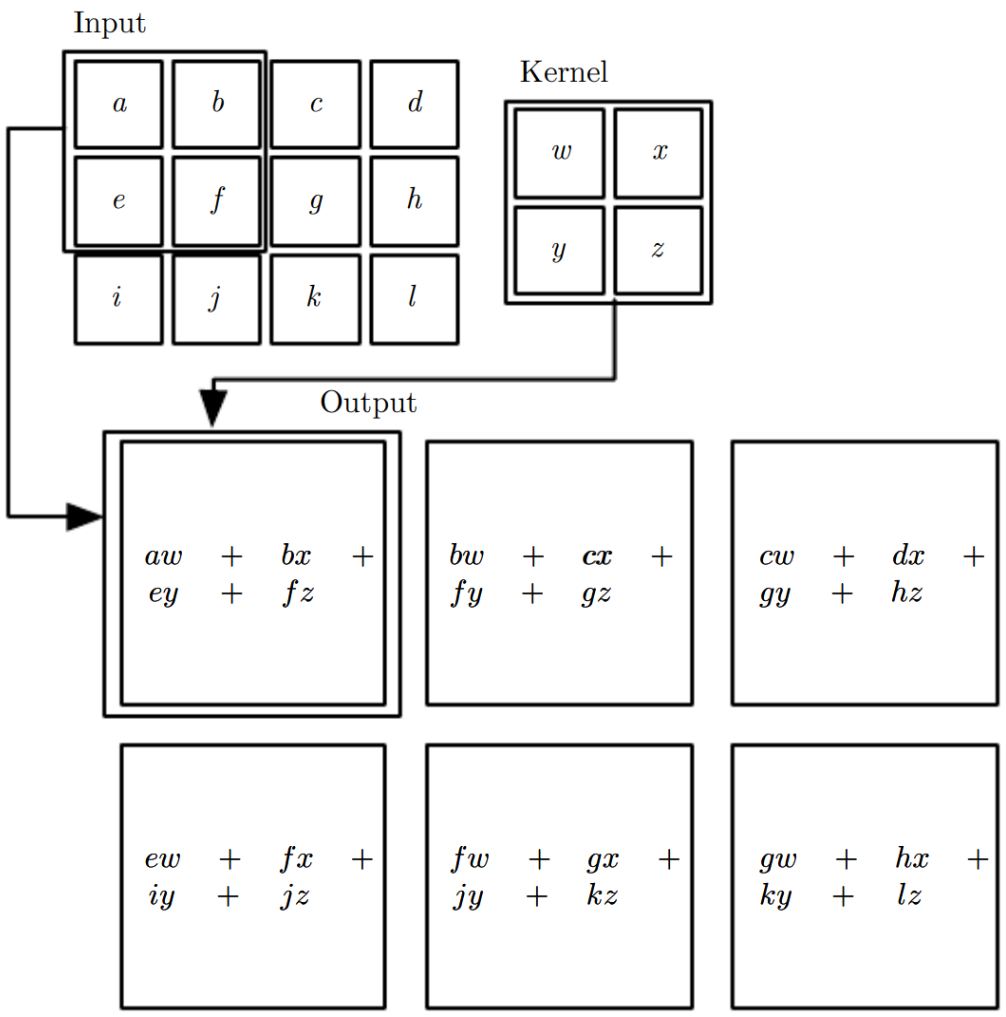


Figure 3: Example of 2D-convolution without kernel flipping

5 Related Work

In the past few years, a considerable number of academic publications have presented proposals focused on implementing Convolutional Neural Networks (CNN) and accelerating the computational processes through the utilization of Field-Programmable Gate Arrays (FPGA). Through several referenced studies, the authors have successfully refined convolutional computations, with some extending them into a complete

CNN and enhancing architectures such as AlexNet and LeNet-5.

Regarding the aspect of implementing the convolutional operation, the majority of the articles we have reviewed ([1] [2] [3]) utilize the modules Multiply-Accumulate (MAC) multiplier for calculations, following the principles of convolutional operations. However, [4] adopts a different design by incorporating a distinct architecture with the use of the modules named Single Instruction, Multiple Data (SIMD) multiplier. Furthermore, most studies utilize fixed-point numbers (8-bit or 16-bit) for computations to facilitate ease (compared to 32-bit floating point) on FPGA, as 32-bit floating-point operations can consume substantial resources. (Sheping Zhai et al, 2019)

In each referenced article, the authors constructed modules with various differences, as briefly described above. The input parameters also vary significantly, including image sizes. We endeavored to collect commonalities among the inputs and performed computations using a 3x3 kernel to compare the results, as shown in Table 1.

Research	**LUT**	**FF**	**Time (ms)**	**Power (W)**
[1] Lenet-5	25436 (47.8%)	NA	6.8	4.35
[2] 224x224 Conv. Layer	5027 (9.45%)	4959 (4.66%)	14.08	NA
[3] 32x32 Conv. Layer	1463(2.75%)	486 (0.46%)	33	NA
[4] 1024x1024 Conv. Layer	NA	NA	4.6	$\approx$ 2.92

Table 1: Results of the resources and power utilization of selected research studies.

6 Implementation

6.1 Architecture

6.1.1 Overview

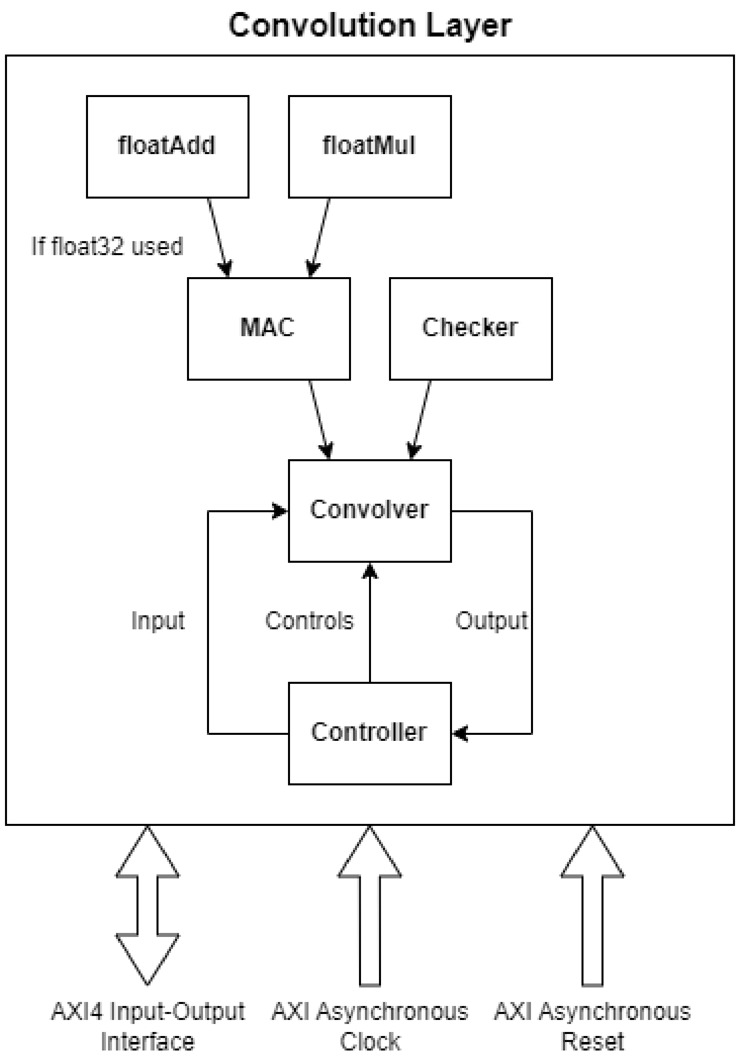


Figure 4: Overview of the architecture

The 2D convolutional layer is implemented upon 4 main block, with the upper most being the controller, which is used to direct the input, output and calculations inside the system. The controller controls the convolver, which is the block responsible for all calculations. The convolver is made up of 2 smaller blocks, namely the multiplier-accumulator (MAC) and the checker. The system supports 2 number format: float32 (single-precision floating-point format) and int8 (8-bit integers). If the user wishes to use float32, MAC is further made up of 2 block, floating point multiplier and floating point adder.

With the architecture designed as an AXI4 peripheral IP core, it has an input-output interface along with 2 ports for asynchronous clock and reset. The address it is assigned is 0x43C00000.

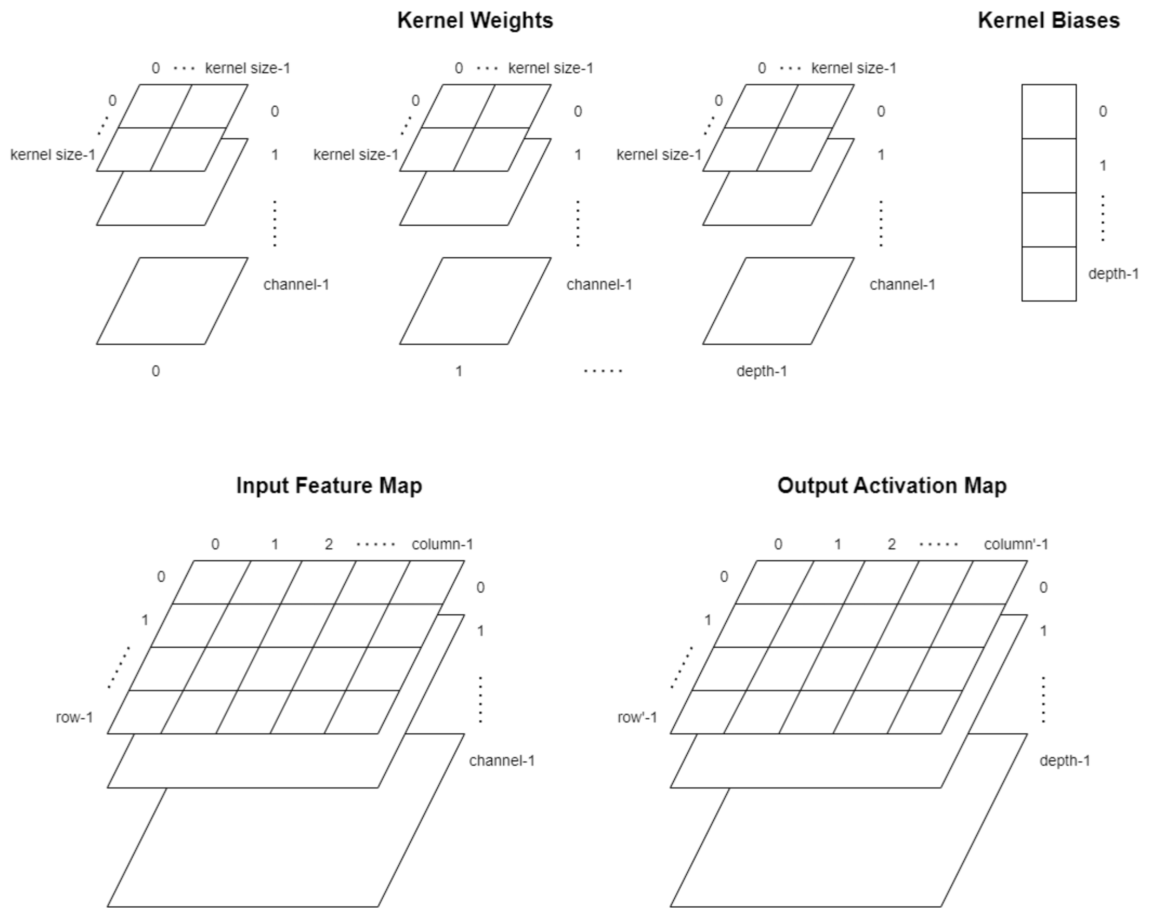


Figure 5: High level view of the 4 data blocks

The processes revolves around 4 data blocks: the kernel weights map, the kernel biases vector, the input feature map and the output activation map. The kernel weights map is a 4-dimensional map with size of *column* columns, *row* rows, *channel* channels and a depth of *depth*. The kernel biases vector is a vector of *depth* elements. The input feature map is a 3-dimensional map with size of *column* columns, *row* rows, *channel* channels. The output activation map is a 3-dimensional map with size of *column'* columns, *row'* rows and *depth* channels.

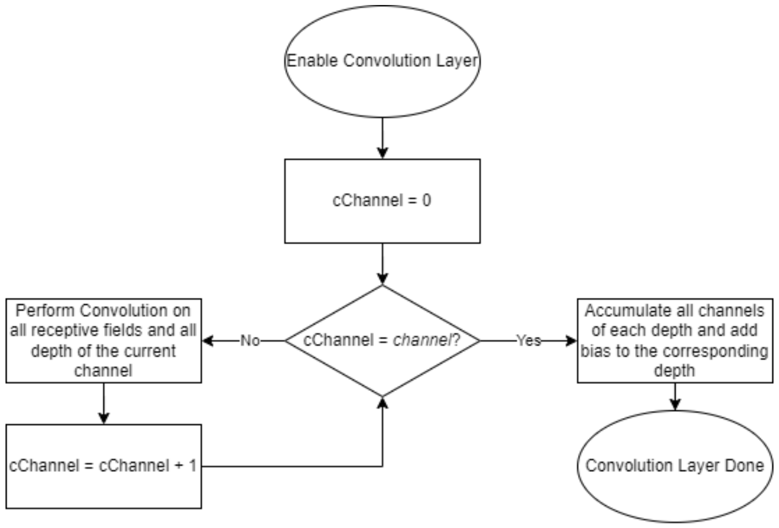

Enable Convolution Layer
cChannel = 0
cChannel = *channel*?
No
Yes
Perform Convolution on all receptive fields and all depth of the current channel
cChannel = cChannel + 1
Accumulate all channels of each depth and add bias to the corresponding depth
Convolution Layer Done

Figure 6: Flowchart of the convolution operation

The convolution operation performed by the convolution layer is a 4-step process:

- Step 1: Initialize, currentChannel = 0.
- Step 2: If currentChannel = *channel*, jump to step 4.
- Step 3: Perform convolution on all receptive fields and on all depths of the current channel, increment currentChannel by 1. Go back to step 2.
- Step 4: Based on *depth*, accumulate all temporary output channels and bias of the corresponding depth. Signal that convolution layer is done.

An example of the described process is represented in the 3 figures above. We first have data input from controller into convolver, then when enable signal is set to 1, for each clock signal, the convolver calculates convolution on all receptive fields of a channel, moving to the next adjacent channel after finishing the current. Once all channels are calculated, for each clock cycle and each depth of the output activation map, values of corresponding receptive fields of the channels of each depth are

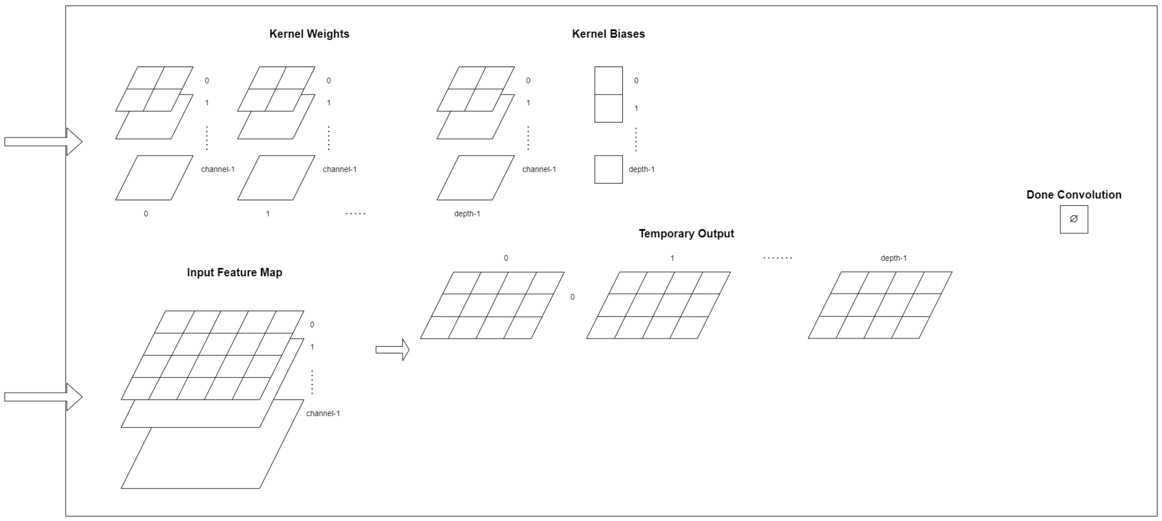


Figure 7: Input done, enable set, convolution on all receptive fields of first channel

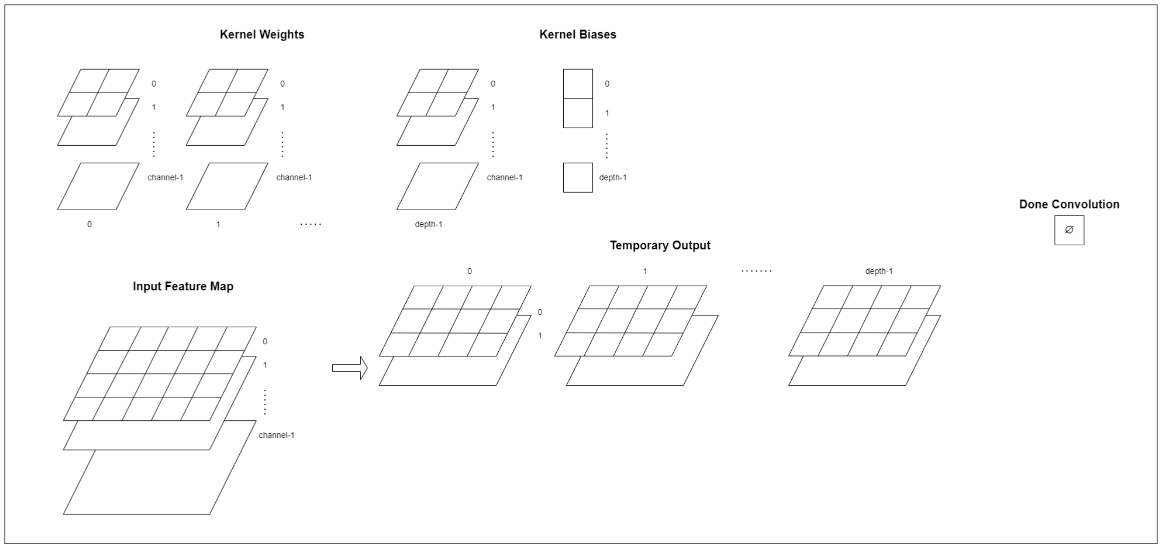


Figure 8: Subsequently move to convolution on all receptive fields of second channel

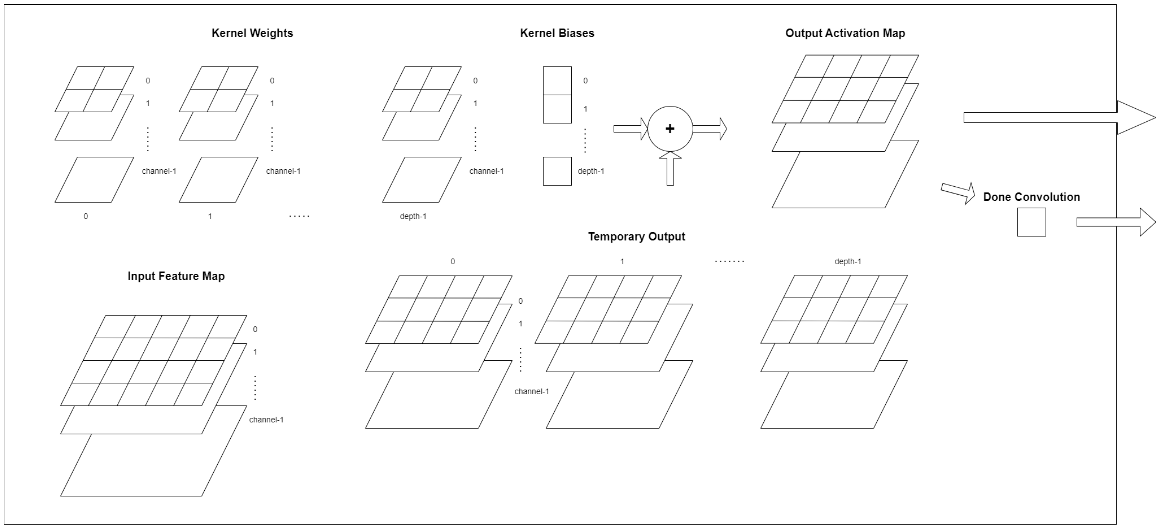


Figure 9: All channels is done convolution, each depth accumulate its channels, when done, set done convolution to 1

accumulated along with the bias of each depth. When all channels are accumulated, done convolution bit is set to 1.

6.1.2 Floating point representation

It is clear that float numbers are difficult to represent in a computer, and also consume a lot of utilization, we try to illustrate the floating-point arithmetic by representing a 32-bit number: 1 bit for the sign, 8 bits for the exponent, 23 bits for the mantissa. For example, 12.56 can be represented in format:

Type	Bit size	Value
Sign bit	1	0
Mantissa (Decimal)		0.56
Mantissa (Binary)	23	$10001111010111000010 10_2$
Exponent	8	10000010_2
IEEE-754 Mantissa	23	$10010001111010111000010_2$
Single Precision float (HEX)	32	0x4148F5C2

Table 2: Explanation of the use mantissa and exponent in representing float numbers

This floating-point representation is widely used in most cases, however, it may cause some drawbacks in resource utilization when computing.

6.1.3 Floating Point Multiplier

Suppose we have 2 input number A, B and an output number Y, the floating point multiplication process can be describe as:

1. Addition of A, B exponent and product of A, B mantissa. If mantissa product $>= 2_{10}$, meaning it is in the form $1x.xxxx..._2$, add 1 to exponent sum.
2. If exponent sum overflow, set exponent and mantissa of Y to maximum value, sign Y is XOR of sign of A and B.
3. If exponent sum underflow, set Y to 0.
4. If exponent sum not overflow and not underflow and mantissa product >= 2:
   - Sign Y = (sign A) XOR (sign B)
   - Exponent Y = exponent sum
   - Mantissa Y = upper 23 bits of product mantissa shifted left by 1 + rounding if the 24th bit is set to 1 and upper 23 bits all not already set to 1
5. If exponent sum not overflow and not underflow and mantissa product < 2:

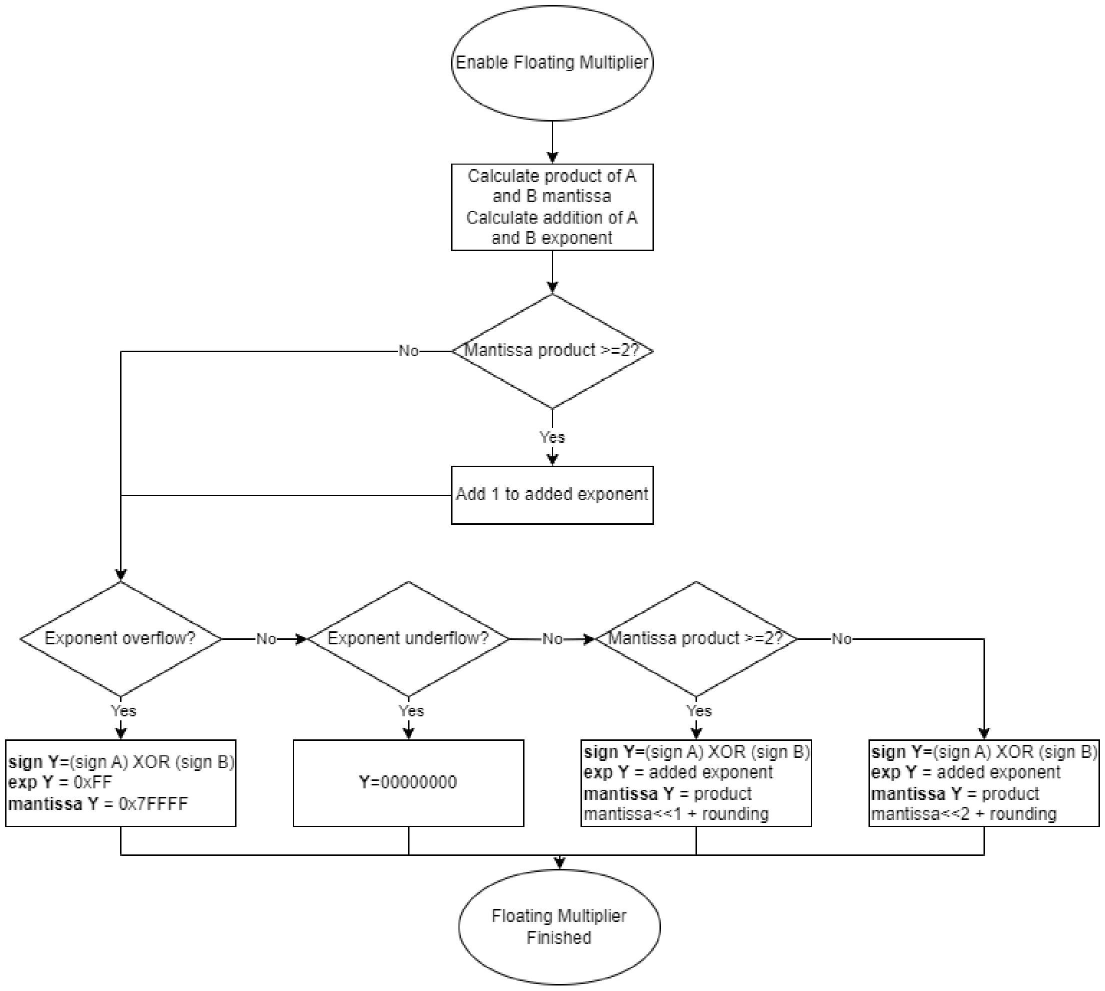

Enable Floating Multiplier
Calculate product of A and B mantissa
Calculate addition of A and B exponent
Mantissa product >=2?
No
Yes
Add 1 to added exponent
Exponent overflow?
No
Yes
Exponent underflow?
No
Yes
Mantissa product >=2?
No
Yes
sign Y=(sign A) XOR (sign B)
exp Y = 0xFF
mantissa Y = 0x7FFFF
Y=00000000
sign Y=(sign A) XOR (sign B)
exp Y = added exponent
mantissa Y = product mantissa<<1 + rounding
sign Y=(sign A) XOR (sign B)
exp Y = added exponent
mantissa Y = product mantissa<<2 + rounding
Floating Multiplier Finished

Figure 10: Flowchart of the floating point multiplication

- Sign Y = (sign A) XOR (sign B)
- Exponent Y = exponent sum
- Mantissa Y = upper 23 bits of product mantissa shifted left by 2 (to skip $1._2$) + rounding if the 24th bit is set to 1 and upper 23 bits all not already set to 1

6.1.4 Floating Point Adder

Suppose we have 2 input number A, B and an output number Y, the floating point addition process can be describe as:

1. Align binary point:
   - If exponent A > exponent B: shift mantissa B by the difference of 2 exponent, exponent and sign is set to A's.
   - If exponent B > exponent A: shift mantissa A by the difference of 2 exponent, exponent and sign is set to B's.
   - If exponent A = exponent B and mantissa A > mantissa B: exponent and sign is set to A's.
   - If exponent A = exponent B and mantissa B >= mantissa A: exponent and sign is set to B's.
2. Addition: Append sign A, B into mantissa A, B accordingly then perform signed addition of the two. If the sum is overflow, exponent is added 1, else if the sum is underflow, exponent is subtracted 1.
3. Check and normalize:
   - If the exponent is overflow, set exponent and mantissa of Y to maximum value, sign Y is the determined sign.
   - If the exponent is underflow, set Y to 0.
   - If the exponent is neither overflow nor underflow, check if sum is overflow but sign is 1, if yes then:
     - sign Y = sign
     - exponent Y = exponent
     - mantissa Y = mantissa sum shifted right by 2, to skip both sign and the $1._2$

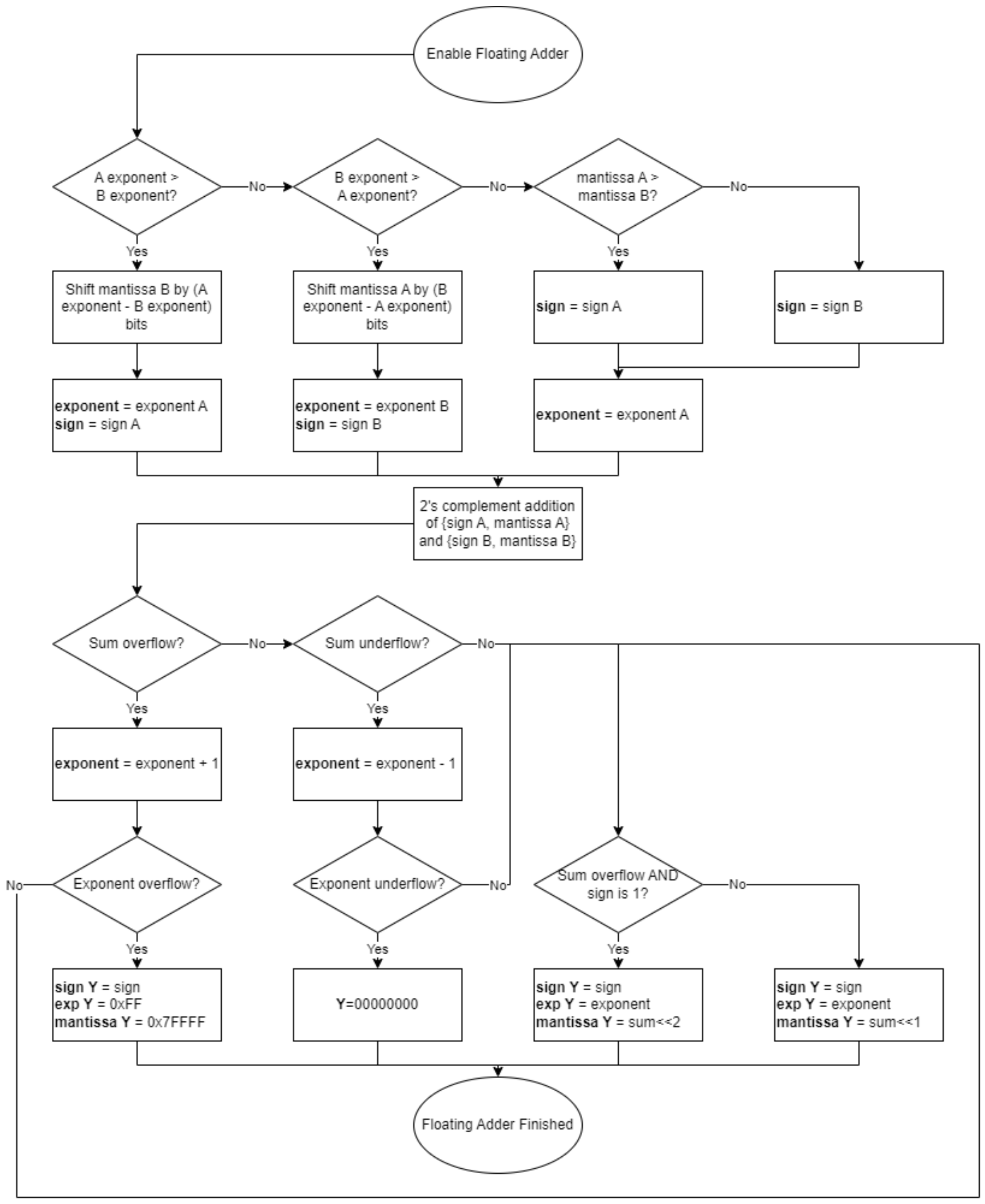


Figure 11: Flowchart of the floating point addition

else:

- sign Y = sign
- exponent Y = exponent
- mantissa Y = mantissa sum shifted right by 1, to skip the $1._2$

6.1.5 Multiplier-Accumulator

Multiplier-accumulator (MAC) operation is a mathematics operation which multiplies 2 input together and accumulates the result with a bias, usually x itself and output into x. Similarly, in this architecture, MAC units are designed to multiply 2 input matrices and accumulate with an input bias to output the feature value that has gone through a filter.

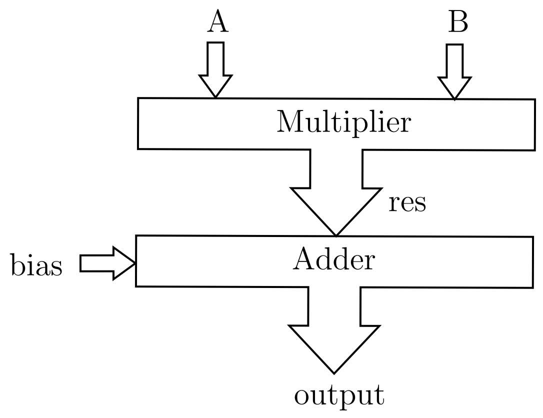


Figure 12: Multiplier-Accumulate (MAC) Structure

6.1.6 Checker

In the context of this architecture, checker unit applies logical AND operation on a given input and output the result of the operation.

6.1.7 Convolver

The convolution unit has 3 primary functions:

1. Padding (if specified): The convolution unit can apply zero padding upon an input feature map, the padding size depends on the parameter set inside it.
2. Convolve feature map with kernel: As in its name, the convolution unit will apply an input kernel upon an input feature map, or the padded input feature map if padding is specified. Depending on the parameters set, the output feature map will have different shapes and values.
3. Signaling when done: Whenever the convolution unit finishes calculation, checker unit will output a HIGH logical signal.

6.1.8 Controller

Top-level module of this architecture, controller unit's mission is to control the input, output and calculating process of the architecture. This architecture follows Advanced Microcontroller Bus Architecture (AMBA), which has data input and output size to be of power of 2, beginning from 32. We define the upper 8 bit to be control bits, while the remaining are data bits. By choosing the minimum size of data transfer in accordance with AMBA, which means input and output into the architecture are logic vectors of 32 bits, with 8 being control bits and 24 being data bits. Each control bit, for both input and output, has its meaning explained in the table below.

The functionalities according to 8 control bits can be described as follow:

- Convolve: On the input side, the reset convolution bit is used to reset the convolution output, the enable convolution bit is used to enable convolution on the input feature map. When the enable convolution bit is set to 1, the input feature map and kernel is copied to a new memory and is used to calculate the convolution, then when the convolution is done, the done convolution bit on the output is set to 1.

Bit Index	Input	Output
0	Reset convolution	Done convolution
1	Enable convolution	Done input
2	Reset input	Done output
3	Enable input	Reserved
4	Reset output	Reserved
5	Enable output	Reserved
6	Input comparator	Input comparator
7	Output comparator	Output comparator

Table 3: Meaning of input and output control bits

- Input: On the input side, the reset input bit is used to reset all input, including the feature map and kernel, the enable input is used to signal that the input process is ongoing and the data bits are data needed to be input as the input feature map or kernel. Furthermore, if both the reset input and enable input bits are set to 1, then the system only reset the input feature map while the kernel is kept, when the input process begins, only the feature map is to be input. When the input process is done, whether it is both the feature map and kernel or just the feature map, the done input bit is set to 1.

- Output: On the input side, the reset output bit is used to reset the output process and copy the output from convolution calculations to output buffer, the enable output bit is used to output the data from the aforementioned output buffer and when the output process is done, the done output bit is set to 1. Moreover, the system also provides an option to check whether the data that have just been input are the same as the data in memory by setting both reset and enable output bit to 1, now the data bits on the output side will mirror the data bits on the input side.

For the architecture to function properly, the procedure is as follow:

1. When turned on, the 3 reset bits need to be asserted first to properly clear data inside the block and then set them to 0.

2. Enable input bit for the whole duration of input, with data on the lower bits. Once input is done, the done input bit should be set to 1.

3. Input is done and convolution can be enabled at any time.

4. Once enable convolution bit is set to 1 and done convolution bit is set to 1, output can be enabled.

5. Like during input, enable output is needed to be set to 1 for the whole duration until done output is set to 1. In the meanwhile, data is continuously output to be read.

There are 7 parameters that can be used to adjust the functionality of the architecture.

Parameter	Meaning	Size
col	The number of columns that a channel of the input feature map have	Larger than 0
row	The number of rows that a channel of the input feature map have	Larger than 0
channel	The number of channels of the input feature map	Larger than 0
depth	The number of channels of the output activation map, and the depth of the kernel map	Larger than 0
k_size	The number of rows and columns of each channel of the kernel	Larger than 0
stride	The step size used when moving the kernel across the input data	Larger than 0
pad_size	Padding size of the input map, if set to 0, then there is no padding	Is not negative
mem_size	Size of the memory that stores input, output, and data from the convolution layer	Larger or equal to size of input and output map, ideally, the input and output process should not have segmentation fault due to small memory

Table 4: Parameters in controller

6.2 System Performance Evaluation

For the purpose of evaluating the convolution layer, a simple block design is implemented. The convolution layer acted as a slave directly communicate with PS (the Processing System), through the AXI protocol.

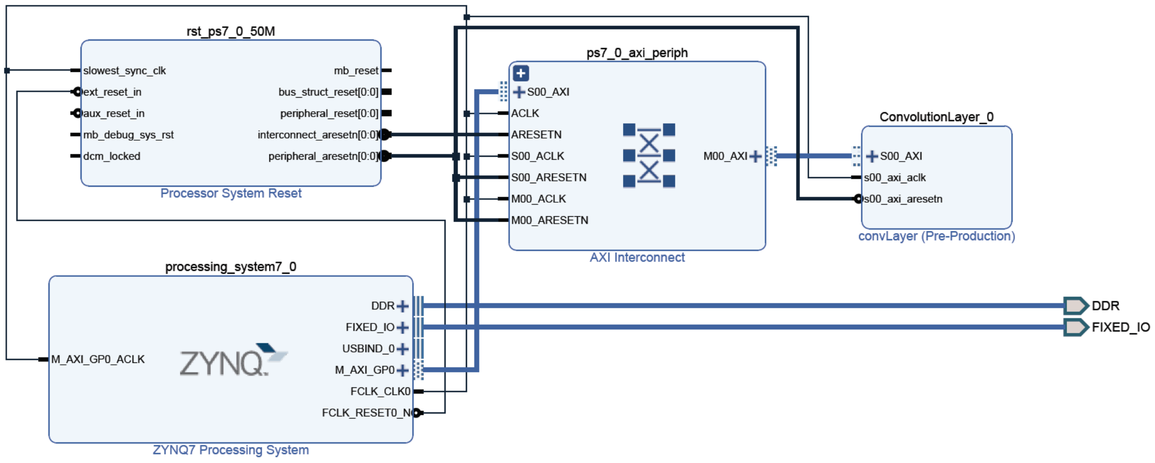


Figure 13: Simple block design

6.2.1 Throughput

A process of performing convolution upon an input feature map involves 3 sub-processes, which is input, can be divided further into input of a kernel and feature map, calculation and output. For each sub-process, we have a formula for the time it took to finish it.

For input of kernel, we have the kernel weights matrix and the kernel biases vector, we have the formula:

$$C_{input\ kernel} = \left\lceil \frac{K^2 N_{channel} D + D}{\frac{Data\ width - 8}{Number\ format\ bit}} \right\rceil \tag{1}$$

With:

- $C_{input\ kernel}$: number of clock cycles for input kernel.
- $N_{channel}$: number of channels on the input feature map.

- K: size of kernel, both vertically and horizontally since we consider square kernels.
- D: depth of the output feature map.
- *Data width*: data width for input and output that we choose.
- *Number format bit*: 8 for integer or 32 for floating point number.

Secondly is the input of feature map, we have the formula:

$$C_{input\ feature\ map} = \left\lceil \frac{N_{row}N_{column}N_{channel}}{\frac{Data\ width - 8}{Number\ format\ bit}} \right\rceil \tag{2}$$

With:

- $C_{input\ feature\ map}$: number of clock cycles for input of a feature map.
- N_{row}: number of rows on the input feature map.
- N_{column}: number of columns on the input feature map.
- $N_{channel}$: number of channels on the input feature map.
- *Data width*: data width for input and output that we choose.
- *Number format bit*: 8 for integer or 32 for floating point number.

For convolution, we have the formula:

$$\text{If int8:} C_{convolution} = 6 + K^2 + N_{channel} \text{If float32:} C_{convolution} = 6 \tag{3}$$

With:

- $C_{convolution}$: number of clock cycles for input.
- $N_{channel}$: number of channels on the input feature map.
- K: size of kernel, both vertically and horizontally since we consider square kernels.

For output, we have just 1 output feature map. For that, we have the formula:

$$C_{output} = \left\lceil \frac{(N_{row} - K + 2P)(N_{column} - K + 2P) D}{S^2 \left(\frac{Data\ width - 8}{Number\ format\ bit} \right)} \right\rceil \tag{4}$$

With:

- C_{output}: number of clock cycles for output.
- N_{row}: number of rows on the input feature map.
- K: size of kernel, both vertically and horizontally since we consider square kernels.
- P: size of padding.
- N_{column}: number of columns on the input feature map.
- D: depth of the output feature map.
- S: stride value.
- *Data width*: data width for input and output that we choose.
- *Number format bit*: 8 for integer or 32 for floating point number.

In general, the number of clock cycles required for the processing of an input feature map is:

$$C = C_{input\ kernel} + C_{input\ feature\ map} + C_{convolution} + C_{output} \tag{5}$$

6.2.2 Fault-tolerance

As mentioned, the convolution layer is controlled by a controller module, which has upper 8 bits as control bits. Input and output is guarded by 2 bits each, the enable bit and comparator bit. The comparator bits aim to check whether the input or output is properly set, then signal to the user that it is indeed set and needed new data, which minimize the problem of unintended input or output is not caught yet while the enable bit aim to check that if the user is indeed wanted to use the functionality.

Effectively, this forms a two-step check upon the architecture for both input and output, reduces the chance that a error may harm the system. If an error occur in the enable bit of a system, the comparator bit will block the invalid action. If an error occur in the comparator bit, then without the enable bit, the system can never perform the corresponding action. The system can always differentiate 2 different, sequential output by the comparator bit, so data redundancy will not occur.

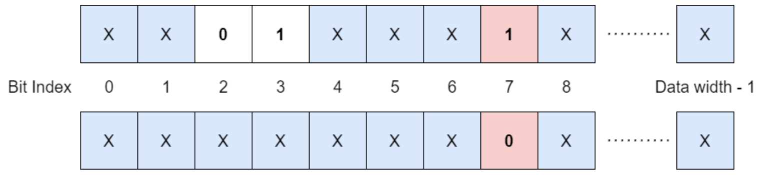

X X 0 1 X X X 1 X X
Bit Index 0 1 2 3 4 5 6 7 8 Data width - 1
X X X X X X X 0 X X

Figure 14: Input action invalid owing to difference in comparator bits

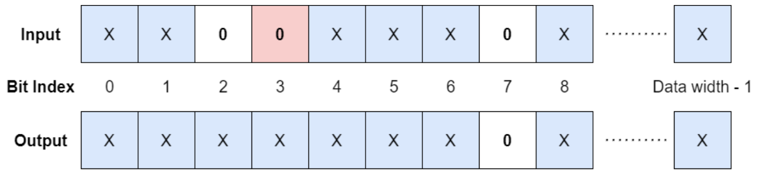

Input X X 0 0 X X X 0 X X
Bit Index 0 1 2 3 4 5 6 7 8 Data width - 1
Output X X X X X X X 0 X X

Figure 15: Input action invalid owing to enable input bit not set

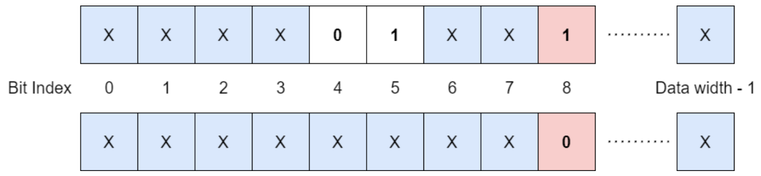

X X X X 0 1 X X 1 X
Bit Index 0 1 2 3 4 5 6 7 8 Data width - 1
X X X X X X X X 0 X

Figure 16: Output action invalid owing to difference in comparator bits

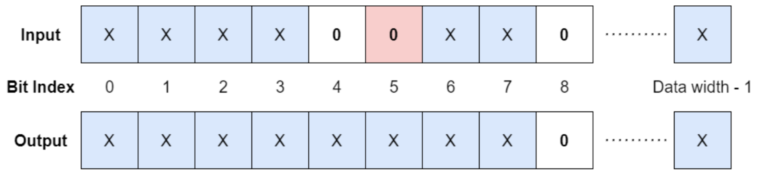

Input X X X X 0 0 X X 0 X
Bit Index 0 1 2 3 4 5 6 7 8 Data width - 1
Output X X X X X X X X 0 X

Figure 17: Output action invalid owing to enable output bit not set

6.2.3 Pipelining

Another notable feature for the architecture is that it enables the 3 independent aforementioned sub-processes to process in parallel to each other, which resembles instruction parallelism because the architecture uses the pipelining technique to perform parallelism.

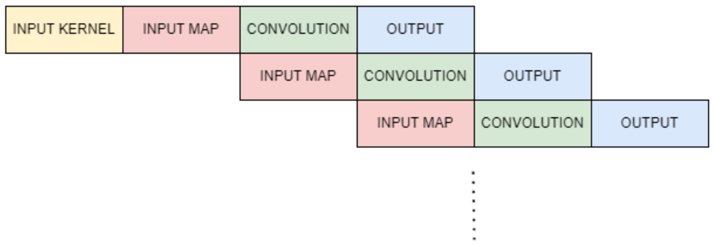


Figure 18: Parallelism using pipelining

With pipelining, the time required for the system in the long run can belong to 1 of the 3 following cases:

- If input is the longest process:

$$C = nC_{input} + Max\left(C_{convolution}, C_{output}\right) + C_{output} \tag{6}$$

  With:

  - C: number of clock cycles for processing all processes.
  - C_{input}: number of clock cycles for input.
  - $C_{convolution}$: number of clock cycles for convolution.
  - C_{output}: number of clock cycles for output.
  - n: number of convolution processes.

- If convolution calculation is the longest process:

$$C = C_{input} + nC_{convolution} + C_{output} \tag{7}$$

  With:

  - C: number of clock cycles for processing all processes.
  - C_{input}: number of clock cycles for input.

  - $C_{convolution}$: number of clock cycles for convolution.
  - C_{output}: number of clock cycles for output.
  - n: number of convolution processes.

- If output is the longest process:

$$C = C_{input} + Max\left(C_{input}, C_{convolution}\right) + nC_{output} \tag{8}$$

  With:

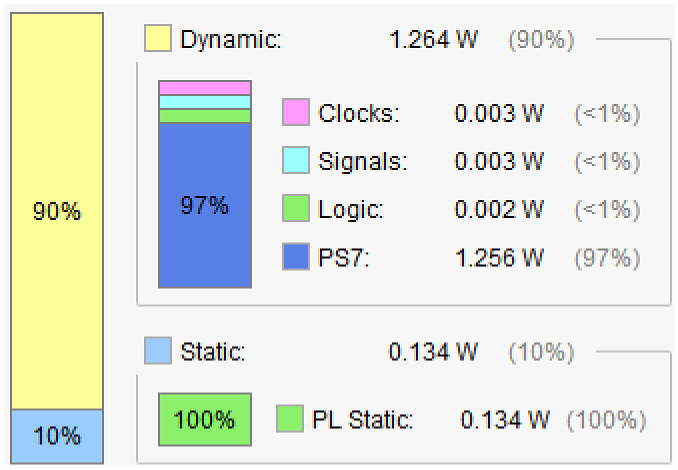
  - C: number of clock cycles for processing all processes.
  - C_{input}: number of clock cycles for input.
  - $C_{convolution}$: number of clock cycles for convolution.
  - C_{output}: number of clock cycles for output.
  - n: number of convolution processes.

6.2.4 Resource Utilization

With the configuration of 2 columns, 1 rows, 2 channels, 1 depths, 1 padding size, kernel size of 2x2, the resource utilization on the board Arty Z7-20 is as follows:

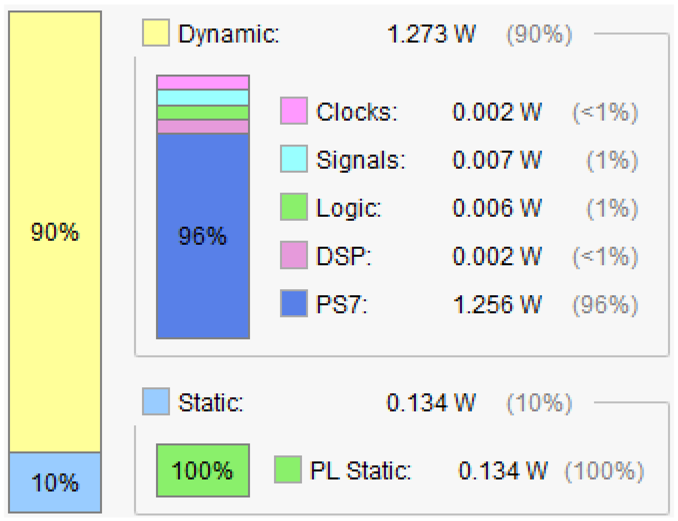
	LUT	FF	BRAM	URAM	DSP	Total Power (W)
Int8	2728 (5%)	1972 (2%)	0	0	0	1.398
Float32	16294 (31%)	2413 (2%)	0	0	88 (40%)	1.407

Table 5: Resource utilization



🟨	Dynamic:	1.264 W	(90%)
🟪	Clocks:	0.003 W	(<1%)
🟦	Signals:	0.003 W	(<1%)
🟩	Logic:	0.002 W	(<1%)
🟦	PS7:	1.256 W	(97%)
🟦	Static:	0.134 W	(10%)
🟩	PL Static:	0.134 W	(100%)

Figure 19: Power usage breakdown of the system with Int8



🟨	Dynamic:	1.273 W	(90%)
🟪	Clocks:	0.002 W	(<1%)
🟦	Signals:	0.007 W	(1%)
🟩	Logic:	0.006 W	(1%)
🟪	DSP:	0.002 W	(<1%)
🟦	PS7:	1.256 W	(96%)
🟦	Static:	0.134 W	(10%)
🟩	PL Static:	0.134 W	(100%)

Figure 20: Power usage breakdown of the system with Float32

6.2.5 Real-time Performance

With the setup above and a data width of 32, the minimal time for each subprocess is as follows:

Type	Clock speed (MHz)	Number of clock periods for input and output	Number of clock periods for convolution	Total time (µs)
Int8	50	7	13	0.4
Float32	13.8889	26	6	2.304

Table 6: Time taken by each numerical type to process a map

6.2.6 Future Work

With the discussion of the system's implementation and performance above, there are some ways that the system can be improved:

- **Optimize calculation pathway:** The current design's calculation pathway requires the system to perform many calculations on each single clock tick. This leads to the problem of inconsistency between parts that don't calculate much, leaving room for inefficiency. Notable example can be said of the MAC, both in int8 and float32.
- **Increase clock frequency:** The problem that clock frequency can not be increased further is because of the limitation on each calculation mentioned above. With better optimization, clock frequency can be increased, leading to better performance.
- **Make use of BRAMs and URAMs:** The architecture currently does not make use of BRAMs and URAMs to store data but instead use local register. With the use of BRAMs and URAMs, performance can be increased owing to specialized hardware used for storage.

Furthermore, the system can be expanded into a whole Convolutional Neural Network with the ability to train and deploy the network.

7 Demonstration & Testing

7.1 Int8

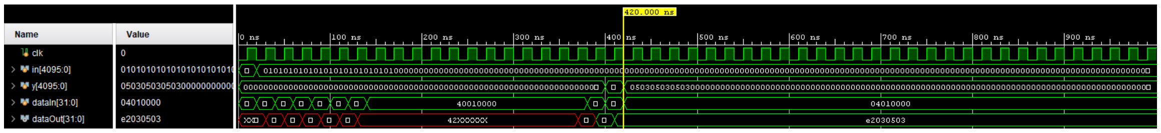


Figure 21: Testing with parameters define in 6.2.4 and with Int8

With the addition of a period clock dedicated for reset and another for waiting when convolution calculation is done to signal beginning of output, we have an exact number of total time it took and the number of clock periods as in theoretical analysis.

7.2 Float32

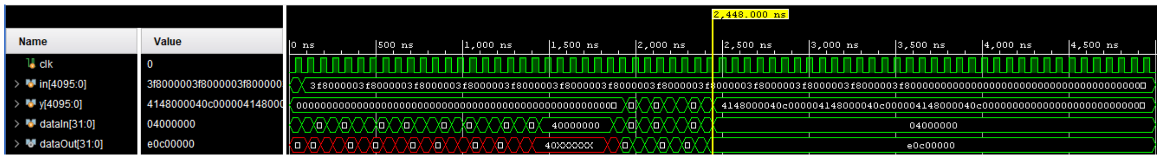


Figure 22: Testing with parameters define in 6.2.4 and with Float32

This situation is exactly like what the one in the section above, with 2 additional clock periods longer, otherwise everything is the same.

7.3 Parallelism

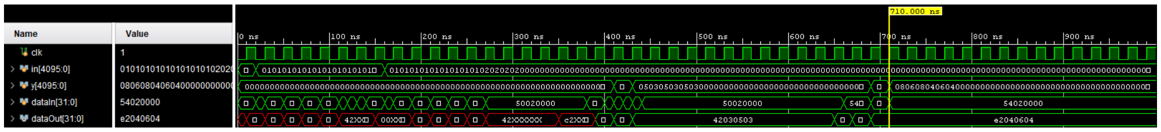


Figure 23: Testing with parameters define in 6.2.4, with Int8 in terms of parallel processing

With this test, we test the parallel processing capability, thanks to pipelining, of our architecture as mentioned in 6.2.3. The result is that 2 whole processes combined into 1 pipelined process requires 0.71 µs, but if they execute 1 after another, will require 0.76 µs, which is 7% faster.

7.4 Partial Reset

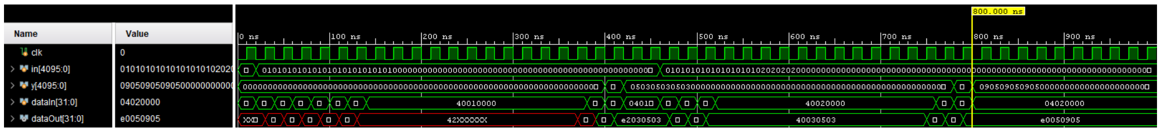


Figure 24: Testing with parameters define in 6.2.4, with Int8 in terms of reset just activation map

With partial reset, we just reset the input activation map by using a combination of input reset and enable input. Notice that the number of clock periods required in the second input is less than the first input.

8 Conclusion

In summary, our FPGA-based implementation of a 2D convolutional layer represents a fundamental stride in accelerating neural network computations. Utilizing the FPGA's logic blocks, we achieved notable improvements in computation speed and resource efficiency compared to conventional software approaches. This foundational work not only establishes the effectiveness of FPGA technology in neural network acceleration but also sets the stage for essential advancements, playing a pivotal role in shaping the future of hardware design for artificial intelligence.

Acknowledging the inherent limitations in our current design, the significance of this research lies in its potential to reshape the landscape of hardware acceleration in AI. This study lays the groundwork for further exploration and optimization of FPGA-based architectures, offering a crucial pathway for real-time processing in applications like image recognition and edge computing.

References

[1] Sheping Zhai et al. (2019). Design of Convolutional Neural Network Based on FPGA. Journal of Physics: Conference Series, 1168(6), 062016. DOI: 10.1088/1742-6596/1168/6/062016.

[2] Trung Pham-Dinh et al. (2022). An FPGA-based Solution for Convolution Operation Acceleration. arXiv preprint arXiv:2206.04520 [cs.AR]. DOI: https://doi.org/10.48550/arXiv.2206.04520

[3] Huynh, T. V. (2022). FPGA-based Acceleration for Convolutional Neural Networks on PYNQ-Z2. International Journal of Computing and Digital Systems, 11(1), Article 136. https://dx.doi.org/10.12785/ijcds/110136

[4] Perri, S., Lanuzza, M., Corsonello, P., & Cocorullo, G. (2005). "A high-performance fully reconfigurable FPGA-based 2D convolution processor." Microprocessors and Microsystems, 29(11), 381-391. DOI: 10.1016/j.micpro.2004.10.004