POLITECNICO DI TORINO

Master's degree in Electronic Engineering

Master's Degree Thesis

A Flexible FPGA-based Neural Processing Unit Architecture

Supervisors
Prof. Mario Roberto Casu
Prof. Diana Göhringer
Dr. Lester Kalms

Candidate
Arianna Palermo

Advisors
Dr. Ahmed Kamal
Dr.Veronia Iskandar

October, 2024

A Mamma e Papà,
perchè ogni mio successo è frutto del
vostro amore e costante sostegno.

Contents

Abstract

Deep Neural Networks (DNNs) exhibit varying performance requirements and energy constraints depending on the applications in which they are employed. Neural Processing Units (NPUs) are designed and optimized to execute network functions and applications efficiently. Today, the same application might need to operate with different data bit-widths to meet varying task requirements. A fixed bit-width accelerator would have limited advantages in accommodating these diverse bit-width needs. Therefore, it is beneficial to develop NPUs that, with the same internal structure, can support different types of quantized data. Recently, various precision-scalable MAC (Multiply-Accumulate) architectures optimized for neural networks have been proposed.

This work presents a review of state-of-the-art precision-scalable MAC architectures and proposes a new solution that can function both as a MAC and as a standard parallel multiplier or adder. For each operation, it supports various precisions: 16x16, 8x8, and 4x4. The synthesis and implementation of the design indicate a maximum operating frequency of 200 MHz. Tests for each precision are conducted, with detailed analyses and reports on timing, power consumption, energy efficiency, and operations per second.

List of Figures

List of Tables

List of Abbreviations

AI Artificial Intelligence
ML Maching Learning
NN Neural Network
NPU Neural Processing Unit
PE Processing Element
PP Partial Product
CU Control Unit
PS Present State

Chapter 1

Introduction

The term Artificial Intelligence (AI) is related to machines that simulate the behavior of the human mind. It is based on computational algorithms that allow to perform human tasks such as learning, problem-solving also perceiving sensory information from the surrounding environment. One of the sub-fields of AI is Machine learning (ML), which aims to make predictions and decisions based on data, exploiting algorithms and statistical models. ML has various application domains such as computer vision (object recognition, object detection, object processing), prediction (medical diagnosis, network intrusion detection, predicting denial of service attacks), semantic analysis, natural language processing, and information retrieval [1]. Figure 1.1 summarizes all machine learning applications.

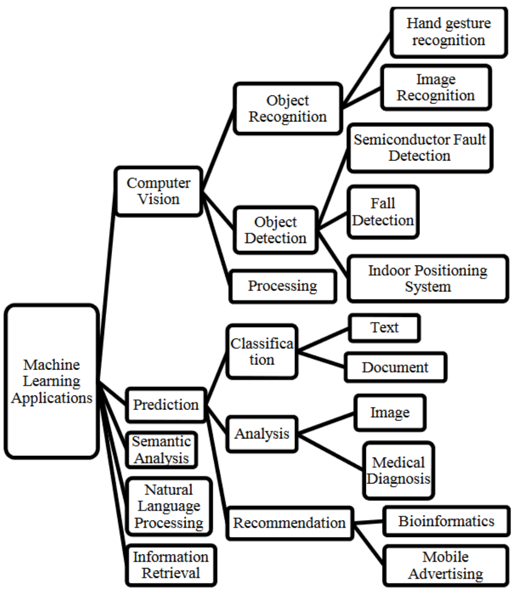


Figure 1.1: Machine Learning Applications [1]

The main model on which ML is based is the neural network. The latter is made of different layers, each of which consists of a fixed number of input and output neurons. When the analysis involves an extraction process for a large data collection, neural networks with more than one hidden layer are required, and they are called deep neural networks.[2]

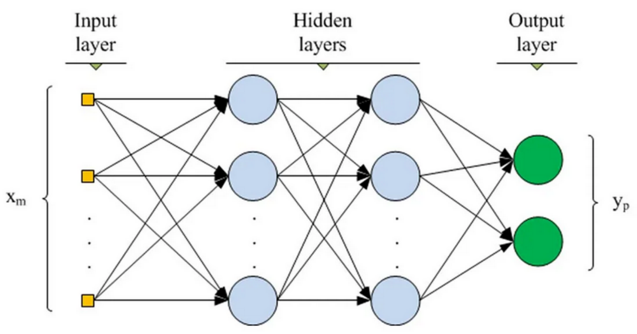


Figure 1.2: Deep Neural Network [3]

More and more devices today require real-time responses, always-on features, and privacy preservation. This means that the demand for on-device machine learning technology is increasing. However, many devices have limited hardware resources. At the same time, deep neural networks require significant computation and communication capabilities.
Recently, a so-called Neural Processing Unit (NPU), which is a dedicated hardware accelerator for neural networks, has been significantly studied for its efficient execution of neural network operations. NPUs are used to efficiently perform the tasks required for deep learning model training and inference. These are Application-Specific Integrated Circuits (ASICs) specialized and optimized to realize various network functionalities. NPUs offer several advantages over other implementations, including high performance, flexibility, fast time-to-market, and low power consumption. [2]
Typical NPUs are implemented with a fixed size and a specific bit-width processing element array. Unfortunately, in the same device, two or more different types of applications might exist, requiring different data bit widths or levels of precision. Voice recognition applications, such as Siri or Google Assistant, require high precision (e.g., 16 or 32 bits) to ensure high accuracy in word recognition and context understanding. However, real-time camera filters can tolerate lower precision (e.g., 8 bits) because processing speed is more critical than absolute accuracy. A slight loss of image quality is acceptable in exchange for real-time performance. [4]
For this reason it is useful to have a flexible NPU that with the same internal structure could support several different quantized data types: a versatile NPU facilitates the integration and efficient communication between all the different applications.

1.1 Motivation and Objective

The objective of this work is to design a flexible Neural Processing Unit that has to support different workloads reusing the same internal architecture. The accelerator could support several quantized data types and various bit-width configurations. The structure of the NPU is modular and can support various NN types, multi-precision convolution operations at runtime, and configurable Processing Element (PE) array size. The PE is configurable to support a multi-precision MAC unit, accommodating INT4, INT8, and INT16 operations. The workflow of this design is as follows.
To comprehend the current architectures and determine a set of design decisions for implementation, state-of-the-art domain research is carried out. The multiplier and

the adder are implemented and then joined together to build the MAC structure. It is realized the re-configurable PE array 4*4 and then the control and operation scheduler units. This system should manage and configure the precision of MAC units, as well as handle task scheduling during matrix multiplication and convolution operations. Finally the accelerator is evaluated in terms of resource utilization, power, and maximum operable frequency.

1.2 Overview of the report

The rest of this work is structured as follows: Chapter 2 provides a theoretical background, reviews the relevant literature, and examines different architectures of MAC structures. Chapter 3 discusses the implementation of the NPU architecture, with in-depth descriptions of all the individual components. Chapter 4 evaluates the performance of the NPU and compares it with the state of the art. Finally, Chapter 5 offers concluding remarks and discusses potential future work.

Chapter 2

Theoretical Background and Related Work

Deep learning neural networks have been successful in solving a wide range of machine learning problems. For this reason, hardware accelerators with specialized functions have been proposed to speed up the execution of DNN algorithms to achieve higher energy efficiency, particularly in their multiplication-accumulation (MAC) operations. This chapter provides an introduction to the deep neural network and then an analysis of the state-of-the-art related to configurable and multi-precision NPU is conducted. At the end are presented the design decisions regarding this project.

2.1 Introduction to the Neural Network and Deep Neural Network

Neural Network is an attempt to build a machine that plays brain activities and can learn. A basic neural network is composed of three layers: input, hidden, and output layers. Nodes from the input layer are connected to nodes in the hidden layer, and each layer may have a certain number of nodes. The nodes in the hidden layer are connected to nodes in the output layer. Weights between nodes are represented by these connections. The raw data entered into the network is represented by the input layer, and the values in this section of the network never change. All network inputs are replicated and sent to the nodes in the hidden layer. The hidden layer receives data from the input layer, applies weight values to modify them, and sends the new values to the output layer. The output layer then adjusts these values using weight values derived from the hidden-to-output layer links. After processing data from the hidden layer, the output layer generates an output, which is then processed by the activation function. [5]

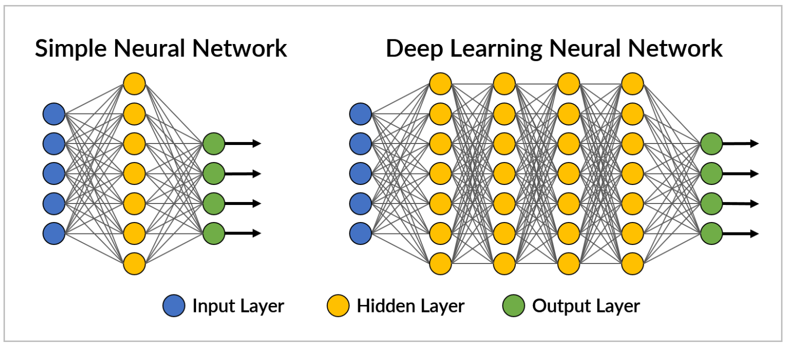


Figure 2.1: Simple and Deep Neural Network [6]

Initial weights are usually set at some random numbers and then they are adjusted during NN training. Weights are adjusted during this process following iterations. The iteration continues with the new weight values maintained if the outcomes of NN after the weight updates exceed the old set of weights. [2]
The biggest difference between the NN and the DNN is the number of hidden layers present between the input and the output. These additional layers allow the network to learn more complex representations of data and this is especially useful for tasks such as image recognition, voice recognition, machine translation, etc. They require much more computing power and memory due to a large number of computational parameters and operations, necessitating the use of hardware accelerators.

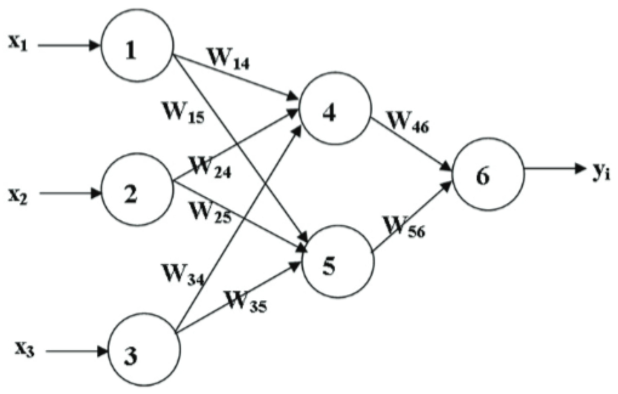


Figure 2.2: Simple and Deep Neural Network [7]

As is shown in Figure 2.2, the output of a node in an artificial neural network is computed by the equation given below:

$$y_i = \sum_{i=1}^{\infty} w_{ij} x_i + \theta_j$$

where y_j is the value that will be passed to the next layer from node j, n is the number of incoming edges to node j, x_i's are the inputs coming from previous layer to node j and θ_j is the bias for node j [7].
From this equation it is possible to notice that for deep learning are needed billions of MultiplyAccumulate (MAC) operations per inference. To significantly improve the

energy efficiency and throughput of DNN accelerators, reduced precision of MAC operations is necessary.

2.2 Booth's encoding technique and Booth's multiplication

Multiplier circuits are the basis of MAC operations and are essential elements in digital signal processing, such as in convolutions. Therefore, the speed at which these multiplications must be executed is of great importance. Fast circuits require small dimensions to minimize wire delay effects. Small dimensions imply a single-chip implementation to further reduce wire delays and make it possible to integrate these fast circuits as part of a larger single-chip system, thus minimizing input/output delays. Typically, multiplication is performed as shown in Figure 2.3, by realizing and summing partial products:

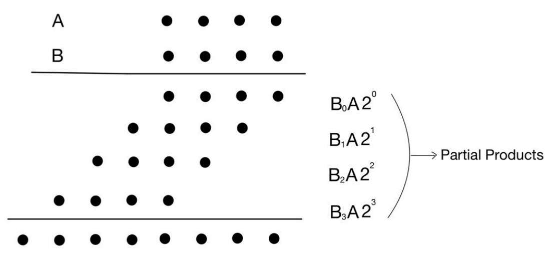
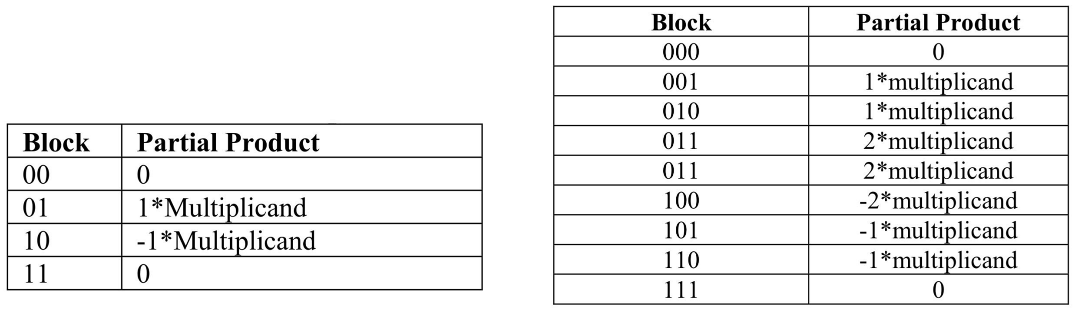


Figure 2.3: Simple multiplication

Booth multiplication allows for smaller, quicker multiplication circuits via encoding the signed numbers to 2's complement, which is likewise a preferred approach used in chip design, and affords large upgrades by decreasing the quantity of partial product to 1/2 of over "long multiplication" techniques [8].
Booth's encoding involves representing values with an extended digit set consisting of three values instead of just two: the number is not represented only by 0 and 1, but also by -1.

Figure 2.4 shows two type of Booth's encoding: Radix-2 and Radix-4.



Block	Partial Product
00	0
01	1*Multiplicand
10	-1*Multiplicand
11	0

Block	Partial Product
000	0
001	1*multiplicand
010	1*multiplicand
011	2*multiplicand
011	2*multiplicand
100	-2*multiplicand
101	-1*multiplicand
110	-1*multiplicand
111	0

Figure 2.4: Radix-2 and Radix-4 Booth Encoding Table [8]

The usage of this encoding technique with the Radix-2 doesn't give a lot of advantages, but with the Radix-4 it is possible to reduce the number of iterations. Normally higher the radix, the lower the number of partial products to add.

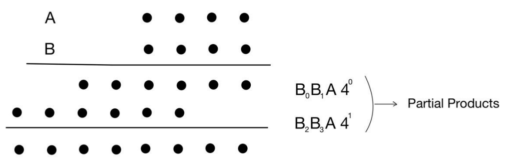


Figure 2.5: Multiplication Radix-4

2.3 State of art of precision-scalable MAC architecture

Different architectures for a scalable MAC exist, each of which aims to try to improve the energy efficiency and throughput of the DNN accelerators.

As [4] explains, the run-time precision scalability is tightly involved with neural-network data flow considerations. It's necessary to introduce two dataflow scalability options: Sum Apart (SA) and Sum Together (ST). These two concepts were introduced to qualify two opposite ways of accumulating subword-parallel computations: the Sum Apart keeps the parallel-generated products separately while the Sum Together sums the parallel-generated products together to form one single output result. It's possible to mix these two typologies and obtain a 2D PE array that can be categorized into three types, SA-SA, SA-ST, or ST-ST. When talking about precision-scalable MACs, when computational precision is scaled down to a fraction of the full-precision operation, the MACs subdivide the processing element (PE) into several parallel lower-resolution PEs. Thus, each nominal PE becomes an array of PEs in itself, offering more spatial loop-unrolling opportunities along the SA or ST dimensions. In full-precision mode, SA and ST MACs behave identically: only one result is generated per clock cycle. In scaled-precision mode, several low-precision results are generated in parallel.

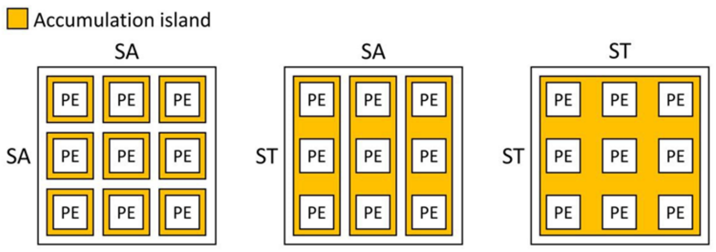


Figure 2.6: SA and ST models [4]

Precision-configurable MAC units can also be divided into two categories: time-based and space-based. Time-based structures use an iterative series of steps to increase precision, adding greater resolution at each stage. Space-based approaches use one of two methods to implement variable computing precision: either by rearranging and sub-mapping the architecture to change signal flow patterns and activating specific circuit components when the bit width changes, or by aggregating simple multiplier units and connecting them through a network of adders and shifters. Examples of temporal types are the Bit Serial or Multibit Serial and LOOM design.
The spacial categories are divided in Divide-and-Conquer (D&C) which is a bottom-up design methodology where the multiplier logic is formed by combining several individual and identical sub-blocks, which are always active. Their configurability is in the interconnection and shift-add logic. Examples are BitFusion and BitBlade.
The other category is the subword-parallel (SWP) which is a top-down design philosophy: single-block full-precision multiplier is selectively gated to reuse existing arithmetic cells in reduced-precision modes. An example is the reconfigurable multiplier.

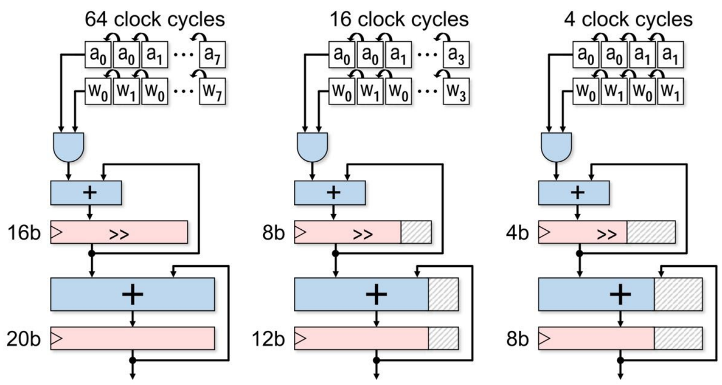


Figure 2.7: Bit Serial [4]

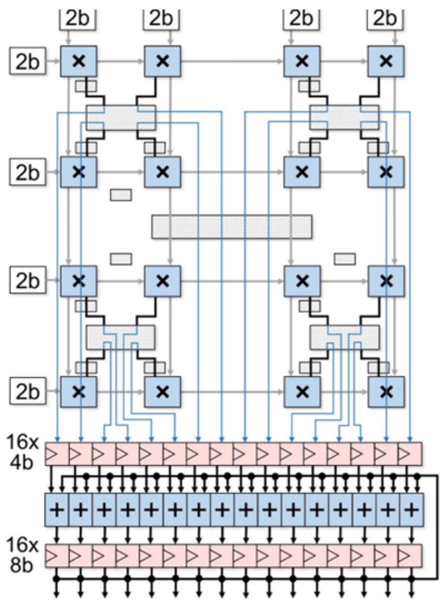


Figure 2.8: D&C [4]

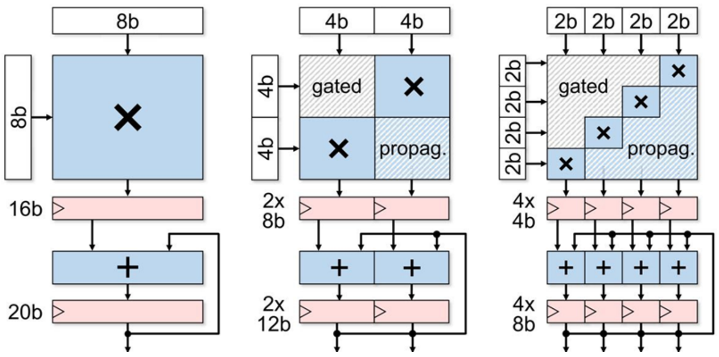


Figure 2.9: SWP [4]

In the [9] is presented the structure of the Bit Fusion architecture. Here, the bit-level flexibility in the architecture allows for minimizing computation and communication at the finest granularity possible without any loss of precision.

This is an ST architecture and it is based on a 2-bit multiplier, called BitBriks, that logically builds Fused Processing Engines (Fused-PE) by dynamically composing, to perform DNN operations with the necessary bit width. Fused-PEs, in particular, offer bit-level flexibility for multiply-add operations which are the most common operations across all DNN kinds. Each BitBrick in a Fusion Unit can perform individual binary (0, +1) and ternary (-1, 0, +1) multiply-add operations. The product is produced by the BitBricks in a Fusion Unit by multiplying an incoming variable-bitwidth input (input forward) by a variable-bitwidth weight. Then, in order to create an outgoing partial sum, the Fusion Unit adds the product to an incoming partial sum.

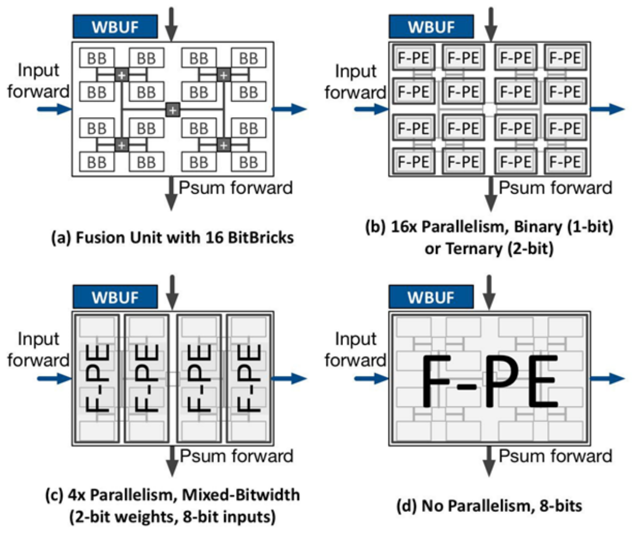

WBUF
Input forward
BB
Psum forward
(a) Fusion Unit with 16 BitBricks
F-PE
(b) 16x Parallelism, Binary (1-bit) or Ternary (2-bit)
(c) 4x Parallelism, Mixed-Bitwidth (2-bit weights, 8-bit inputs)
(d) No Parallelism, 8-bits

Figure 2.10: Dynamic composition of BBs in a Fusion Unit [9]

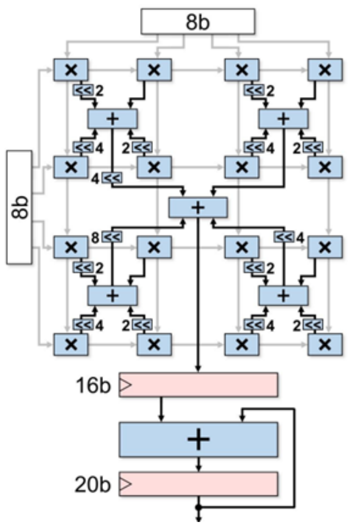


Figure 2.11: Bit Fusion MAC configured for 8b x 8b [4]

As shown in Figure 2.12, the microarchitecture of a single BitBrick is realized by half adder and full adder, and it works with two-bit operands and their corresponding sign-bits.

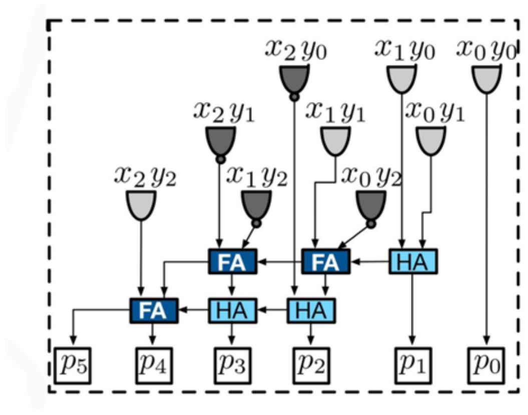


Figure 2.12: BitBrik's microarchitecture [9]

To understand how a BitBrick works, Figure 2.13 can help. It takes a 4-bit multiplication as an example. This operation is decomposed into four 2-bit multiplications that can be executed using BitBricks to generate the decomposed products, which require shifting before being correctly combined.

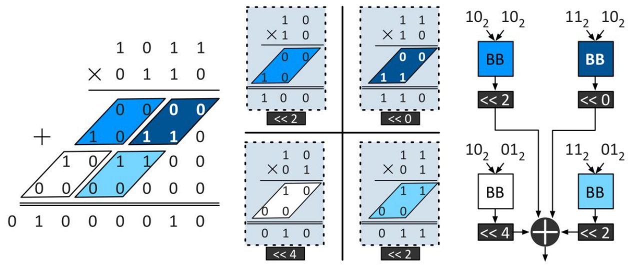


Figure 2.13: 4 bits multiplications [9]

A structure that features a much smaller overhead for shift-add logic compared to Bit Fusion is presented in [10]: this architecture is called BitBlade. The idea is that in BitFusion, the BitBriks in the same position on different PEs have the same parameter for the variable shift operations.
In the BitBlade the idea is to place BitBriks that have the same shift in the same PE. In this way is possible to reduce the number of shifts needed and the complexity, and is also possible to work with two different types of adders: intra-processing element, which can perform the addition operation with low bit-width precision and small area, and inter-processing element, that is the same used in the BitBrik's architecture.

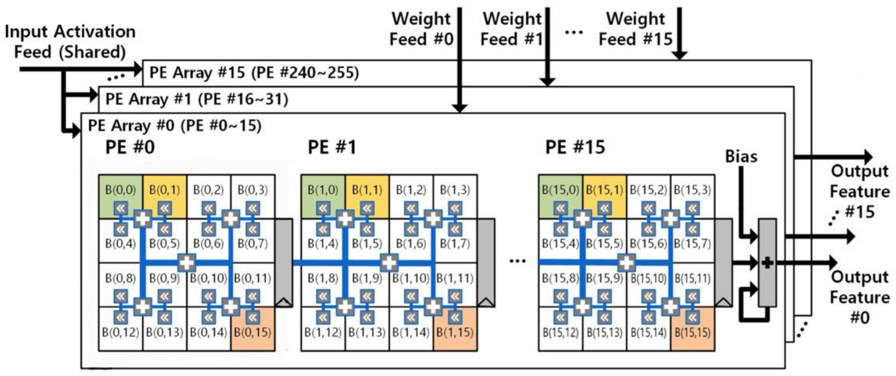


Figure 2.14: BitBricks in Bit Fusion architecture with different shift-left parameters within each PE [10]

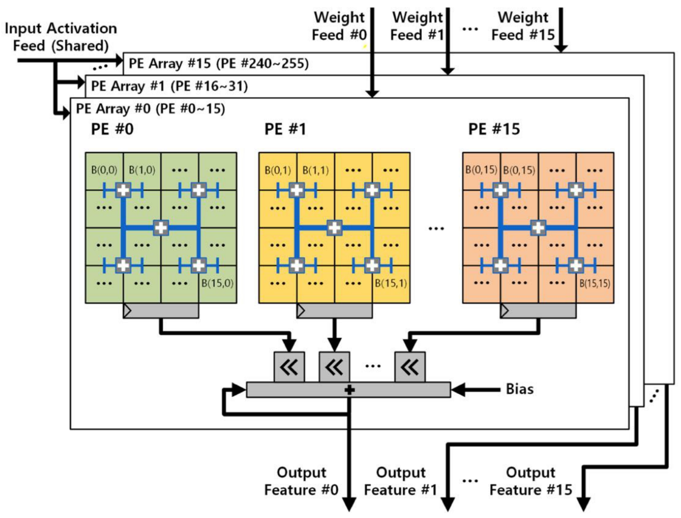
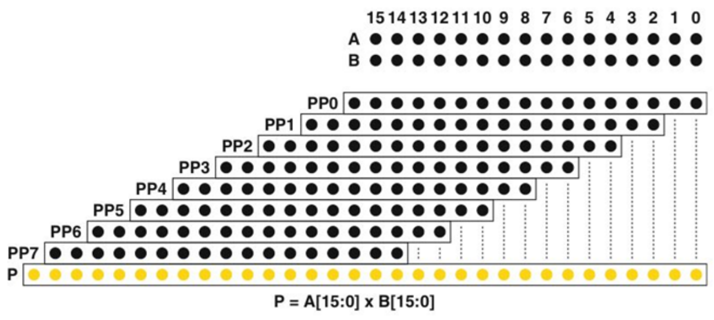


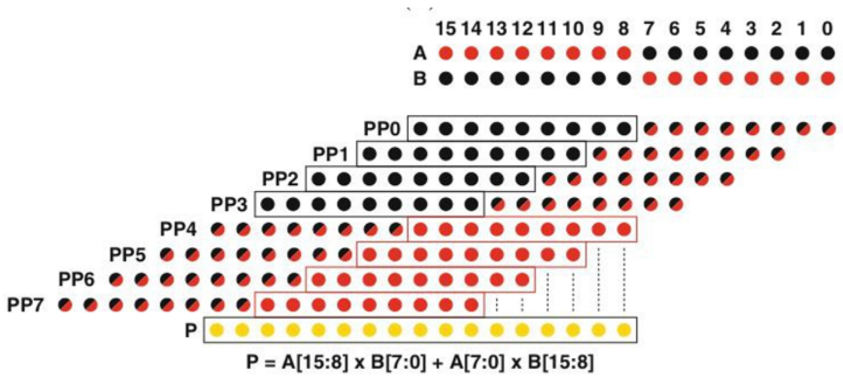
Figure 2.15: BitBricks in Bit Blalde architecture with the same shift-left parameters within each PE [10]

In[11] is shown instead a scalable multiplier, called Reconfigurable Multiplier. This architecture is realized by a modified Radix-4 Booth signed with an ST mode, and it can perform both multiplications at full precision and a dot-product at lower precision.



(a)

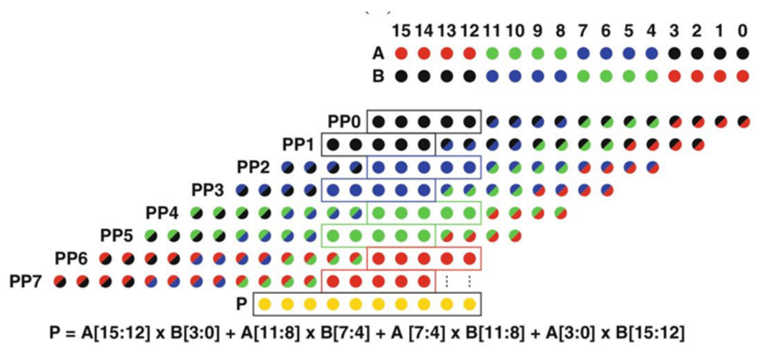
Figure 2.16: Alignment of partial products for configuration 16x16 [11]



(b)

Figure 2.17: Alignment of partial products for configuration 8x8 [11]

As can be seen in Figures 2.16, 2.17, and 2.18, the idea behind this architecture is that a dedicated external adder is not needed to sum up the low-precision products during dot-product operations. Instead, it exploits the natural alignment of partial products in a standard multiplier. When this structure operates with maximum parallelism, all aligned bits are summed together; in other configurations, only the fully colored dots in the diagram are summed, as the partially colored ones are zero. When the chosen configuration is eight bits, two numbers are packed into the same input, or four if the configuration is four bits.


15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
A
B
PP0
PP1
PP2
PP3
PP4
PP5
PP6
PP7
P
P = A[15:12] x B[3:0] + A[11:8] x B[7:4] + A [7:4] x B[11:8] + A[3:0] x B[15:12]

(c)

Figure 2.18: Alignment of partial products for configuration 4x4 [11]

Figure 2.19 shows the structure of this multiplier that consists of two blocks (green) responsible for precision reconfiguration, a Booth encoder that performs the encoding of one of the two inputs and then generates the partial products using the Booth selector. To obtain the correct result, there is a compression tree and a final adder that produces a single output to accommodate the summation configurations.

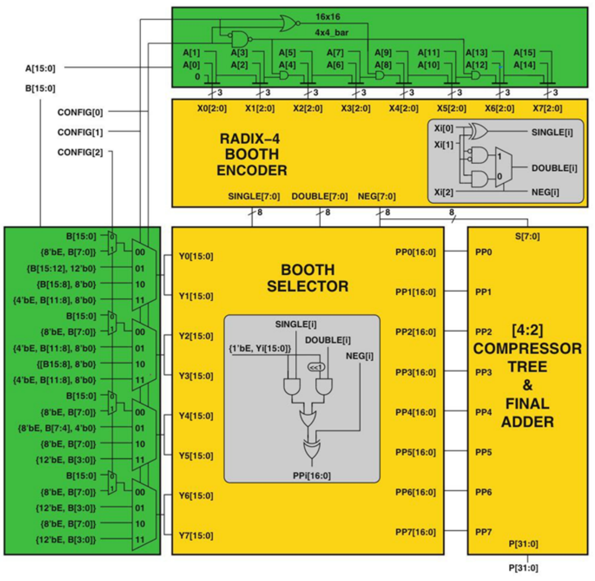


Figure 2.19: Reconfigurable Multiplier [11]

2.4 Comparison between the state of the art

Architecture	SA/ST	Precision (bit)	Sub-blocks always active/Single-block gated	Multiplier with implicit addition/ Multiplier and separate adder
2D D&C [4]	SA/ST	2,4,8	Sub-blocks always active	Multiplier and separate adder
BitFusion [9]	ST	2,4,8	Sub-blocks always active	Multiplier and separate adder
BitBlade [10]	ST	2,4,8	Sub-blocks always active	Multiplier and separate adder
Reconfigurable Multiplier [11]	ST	4,8,16	Single-block gated	Multiplier with implicit addition

Table 2.1: Comparison state of the art

2.5 Design Decision

This section explains the design choices made for this implementation, which were influenced by the literature review.
At the base of this work there is the implementation of a MAC unit, that takes inspiration from the literature survey for the design of the multiplier. It is realized by combining the idea of the reconfigurable multiplier [11] and the 2D Divide-and-Conquer Sum Apart approach [4]. The implemented MAC structure can be configured to work with data of different bit widths: 16x16, 8x8, 4x4. From the [11] is taken the structure until the production of the partial products, and then the approach of [4] is used to sum together the different partial products (PPs) in the right way to obtain the right parallelism.

Chapter 3

Microarchitecture Design and Implementation

This chapter presented a description of the design and implementation of the datapath and the control unit (CU) for a flexible Neural Processing Unit.
At the base of the datapath there is the processing element unit, composed by a multiplier and an adder.
The CU can control three types of simple operations: only parallel multiplication with different precision, only parallel sum with different precision, or parallel MAC operations. It is able also to develop other two cases, parallel matrix multiplication and convolution operations, that were used for the evaluation part.
The VHSIC Hardware Description Language (VHDL) is used for this design, targeting Xilinx FPGAs.

3.1 Data precision and data input format

The processing element can perform operations with three different precisions: 16x16, 8x8, and 4x4. Analyzing the data format for the multiplier, when the logic is working with the highest precision, there is only one output, and the last bit of the input is the sign bit. When the 8x8 configuration is chosen, each input contains two 8-bit values, where the 8th and 16th bits are the sign bits, resulting in two outputs. In the 4x4 configuration, there are four 4-bit inputs per operand, each with its sign bit, producing four outputs.
When the logic functions as an adder, the data format is different. For example, if we want to sum two 32-bit numbers, the two input ranges are the first from bit 0 to 31, and the second input ranges from bit 32 to 63. As shown in Figure 3.1, the first 4 bits of the first input are summed with the first 4 bits of the second input, and the same applies to the other bits. If we are working with a different parallelism, such as with a 4-bit number, the concept remains the same: bits from 0 to 31 are inserted, but they are always summed with the corresponding 4 bits from 32 to 63.

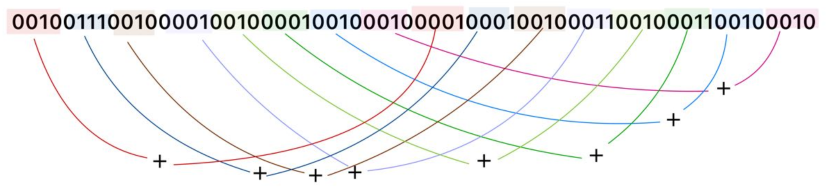


Figure 3.1: Input format for external adder's input

When the adder is working with :

- 4 bits : it gives 8 outputs of 5 bits
- 8 bits : it gives 4 outputs of 9 bits
- 16 bits : it gives 2 outputs of 17 bits
- 32 bits : it gives 1 output of 33 bits or 1 output of 32 bits if it is working like a MAC unit

3.2 Multiplier's design

The multiplier's block is composed of two main blocks: booth multiplier and multiplier.

3.2.1 Booth multiplier

Figure 3.2 shows the structure of the booth multiplier. The booth multiplier implemented in this thesis is based on the structure presented in [11] until the formation of partial products. It exploits the concept of Booth's encoding presented in section 2.2.

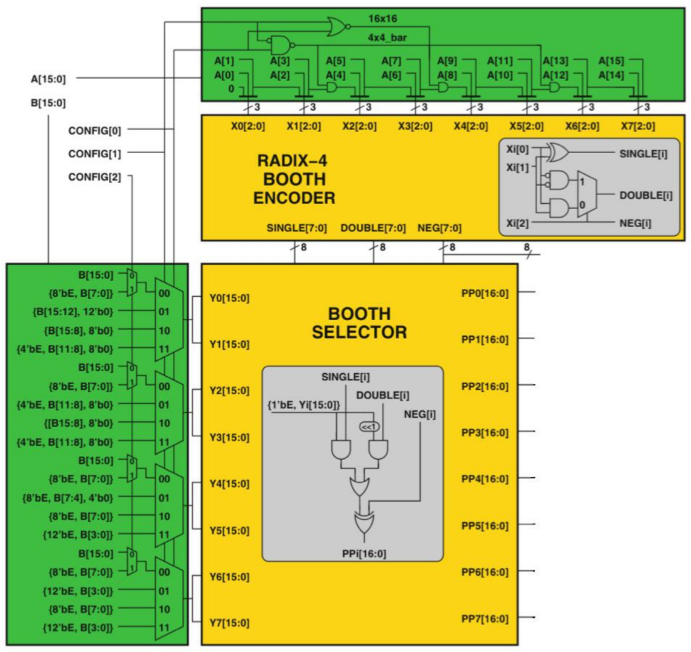


Figure 3.2: Booth Multiplier [11]

The green blocks are controlled with three bits, the configuration bits, and are responsible for the correct precision configuration of the input operands. Table 3.1 represents the three configurations that are implemented :

Bit CONFIG	Precision Configuration	Operand Organization
000	16x16	A[15:0] x B [15:0]
010	8x8	A[15:8] x B [7:0] + A[7:0] x B [15:8]
001	4x4	A[15:12] x B [3:0] + A[11:8] x B [7:4] + A[7:4] x B [11:8] + A[3:0] x B [15:12]

Table 3.1: Precision configuration

The Radix-4 Booth Encoder is the logic responsible for the encoding part. The data with the Radix-4 booth encoding technique are encoded according to Table 3.2. According to the wanted precision configuration, the 'CONFIG' bit needs to be set correctly.

Input bits	Encoded output
000	000
001	1*multiplicand
010	1*multiplicand
011	2*multiplicand
100	-2*multiplicand
101	-1*multiplicand
110	-1*multiplicand
111	000

Table 3.2: Radix-4 Booth encoding

The Booth Selector block can combine the two operands to obtain the eight partial products.
Figure 3.3 reports the Vivado implementation of the Booth multiplier.

To obtain the correct partial product, it is necessary to add some '1s in the positions shown in Figure 3.4. In the implementation, this block is called sign_correction, and it is located between the two parts of the total multiplier.

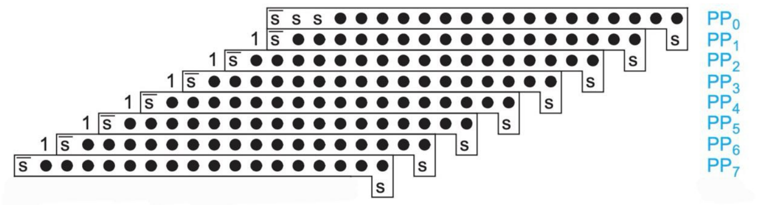


Figure 3.4: Sign Correction [11]

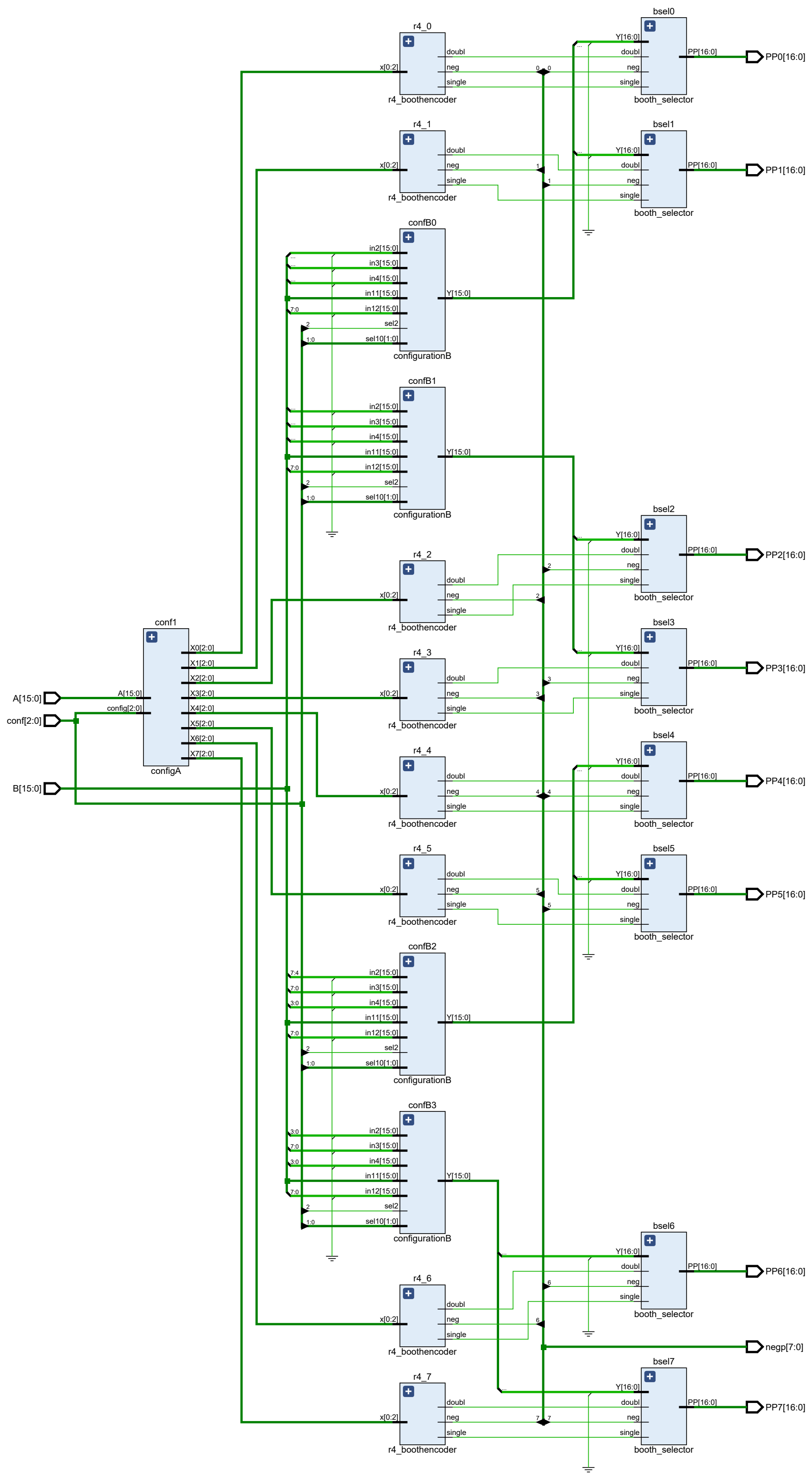


Figure 3.3: Booth multiplier design

3.2.2 Multiplier

The second part of the final multiplier is the one that is responsible for summing together the partial products. Its structure is inspired by the approach of [4]. The idea is to start by summing all the pairs needed to obtain a 4x4 multiplication. To sum them up correctly, a shift on one of the operands is necessary. If the chosen bit width is 4, the output is given immediately in a register. If the chosen bit width is 8, the previous results are combined in pairs, with a right shift, to produce two 16-bit outputs. If the chosen bit width is 16, then all the partial results are added together to obtain a single 32-bit result at the output. Figure 3.5 shows the implementation of the multiplier. The blocks indicated as PP0, PP1, PP2, PP3, PP4, PP5, PP6 and PP7 are the partial product that arrives from the sign_correction block.

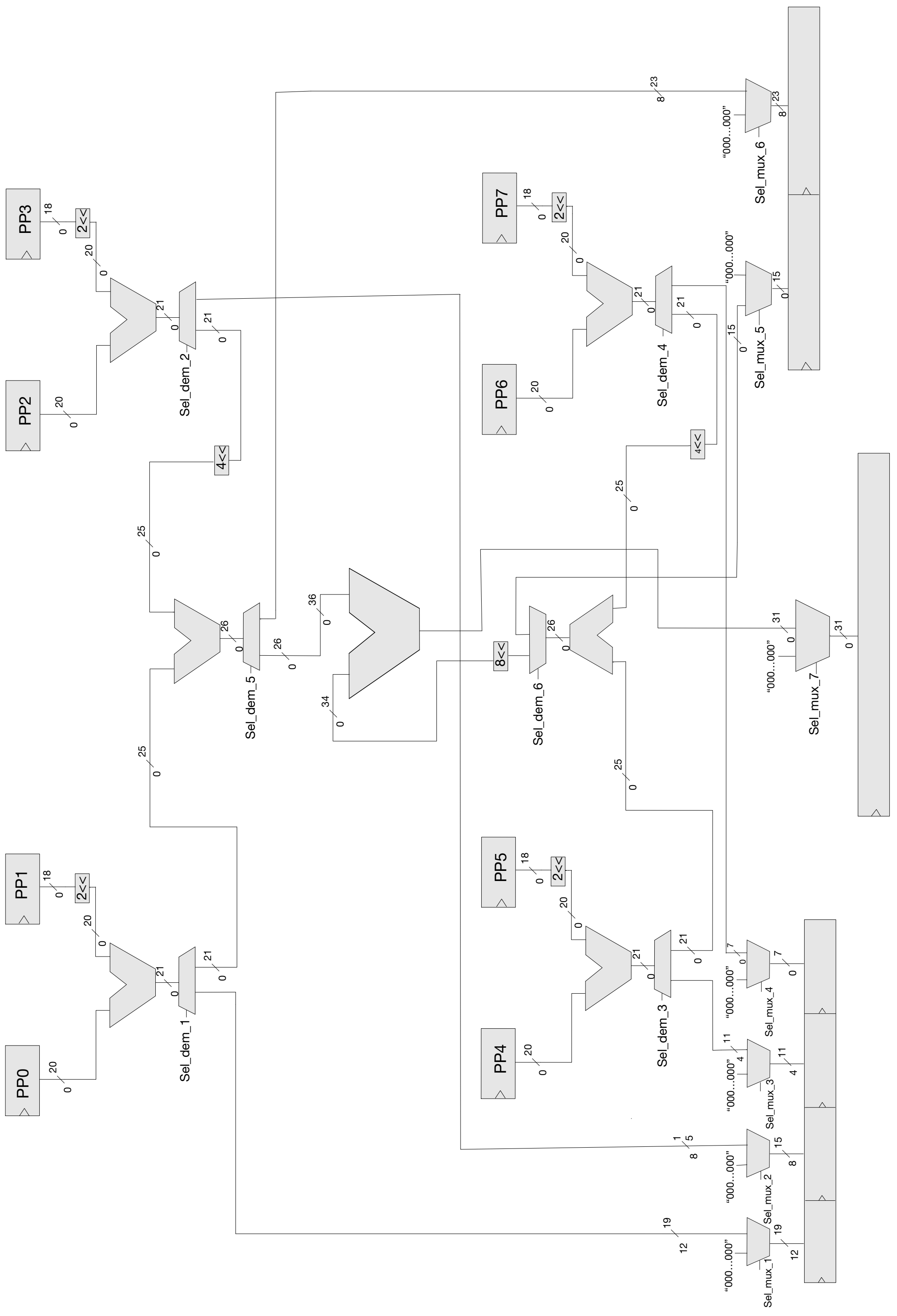


Figure 3.5: Multiplier's design

3.2.3 Final Multiplier

Here the implementation of the final multiplier is reported in Figure 3.6.

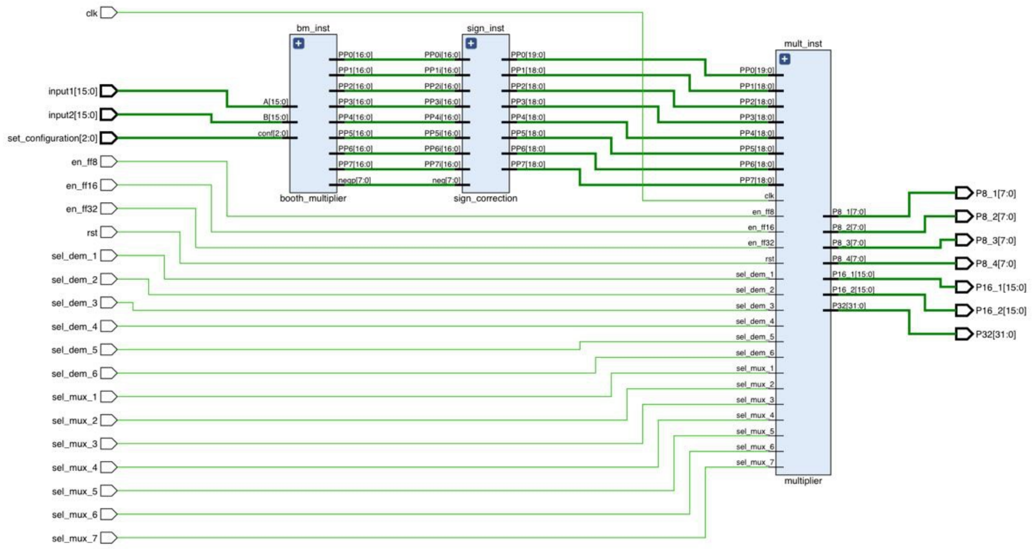


Figure 3.6: Final Multiplier

3.3 Adder

The total adder is realized by combining eight four-bit adders. In Figure 3.7, a single four-bit adder is shown.
Inputs can come from:

- The different registers of the multiplier, according to the chosen bit-width. In Figure 3.7, these are the three inputs in the first multiplexer.
- External inputs, if the structure is used only as an adder. In this case, a register is placed before to correctly synchronize when the input data changes.
- The MAC register, if it is used like a MAC.

The idea is that each adder can work independently, or multiple adders can communicate and work together through carry propagation, controlled by a specific multiplexer. For instance, if the chosen precision is 8 bits, the adders will work together in pairs, and if the precision is 16 bits, four adders will work together.
The multiplexer which the output is called 'validx' in figure 3.7, ensures that when the adders need to work together, each adder only starts working when the previous one has finished; otherwise, the adder cannot work with the correct numbers in the case of carry propagation.

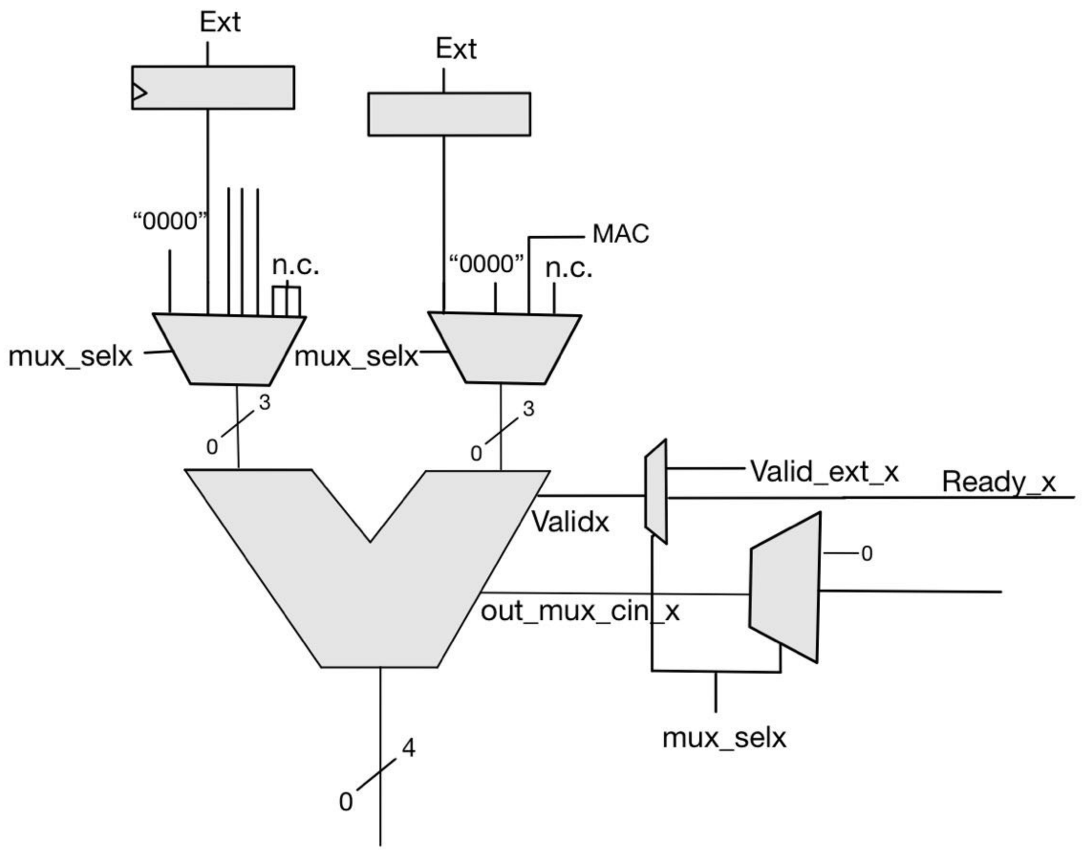


Figure 3.7: Simple adder 4 bits with the multiplexer that select the carry in and the valid signal

Outputs are saved in registers that are different according to the chosen bit width. The parallelism of the register could be 5 bits, 9 bits, 17 bits, 33 bits or 32 for the MAC one. Figure 3.8 shows the structure of the adder.

3.4 Processing Element: MAC UNIT

Figure 3.9 shows a simplified version of the complete design of the PE element, the MAC unit.

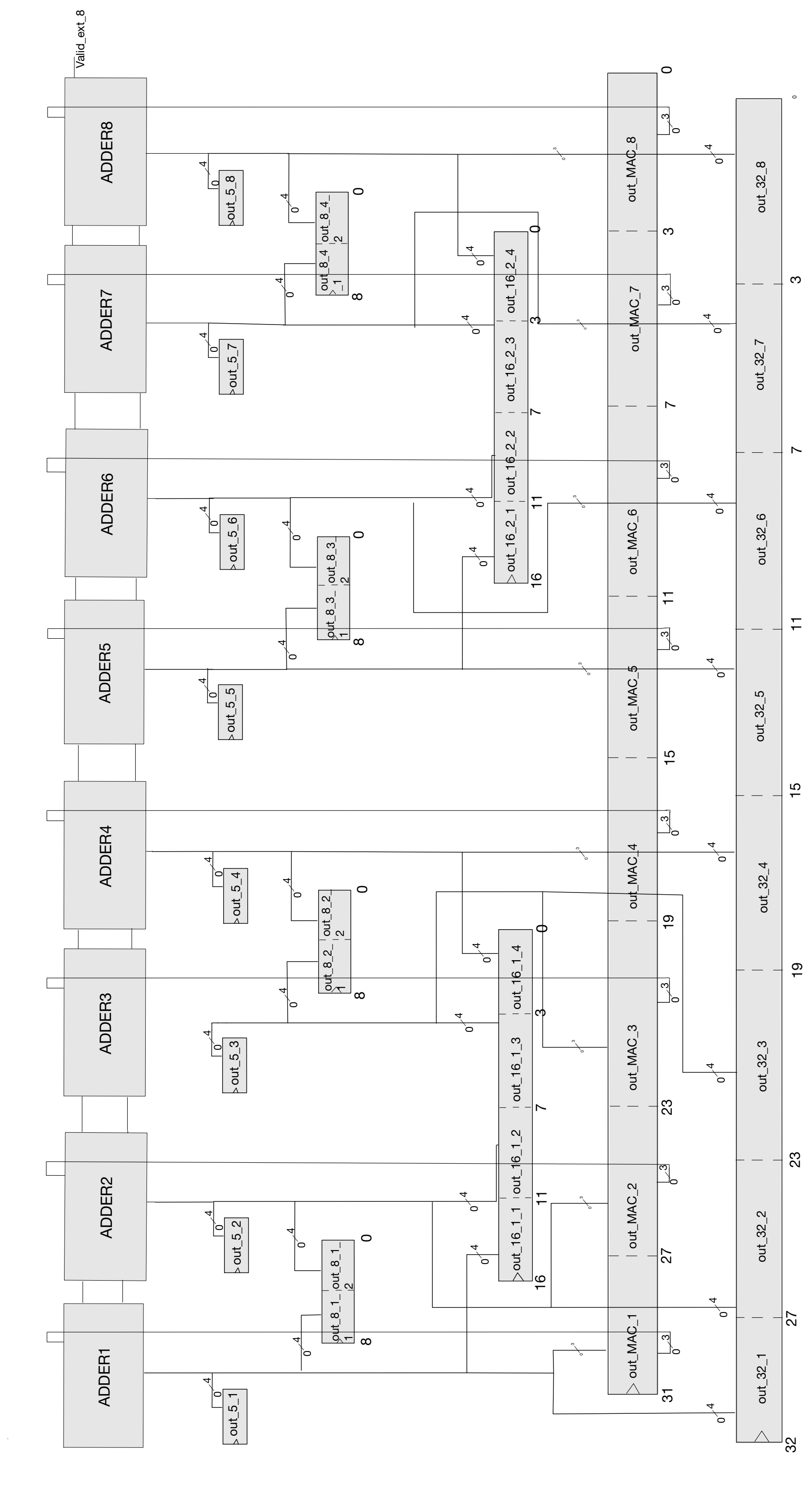


Figure 3.8: Adder's design

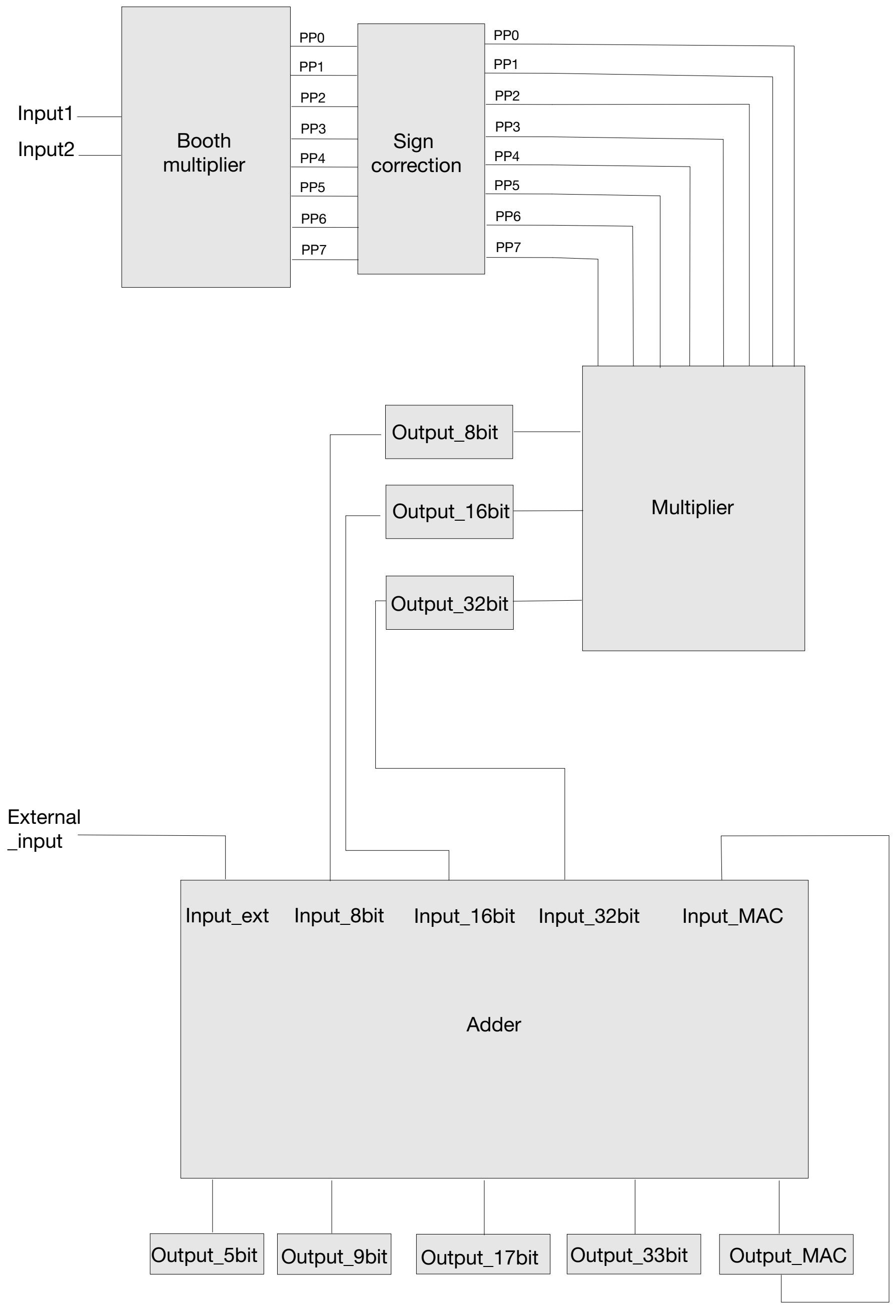


Figure 3.9: MAC design

3.5 Datapath

The total datapath is represented in Figure 3.10. It is composed of the 4x4 PE array that receives inputs from two matrices: the input matrix and the weight matrix, which is set by a 10-to-2 multiplexer. It can also receive inputs directly from the user when the desired operation is a simple one: only addition, only multiplication, or MAC.
The implemented design is also capable of performing matrix multiplication and convolution operations. To realize these types of operations, additional blocks are needed. Below is an explanation of the logic behind how these operations are performed, which will clarify the composition of the different blocks.

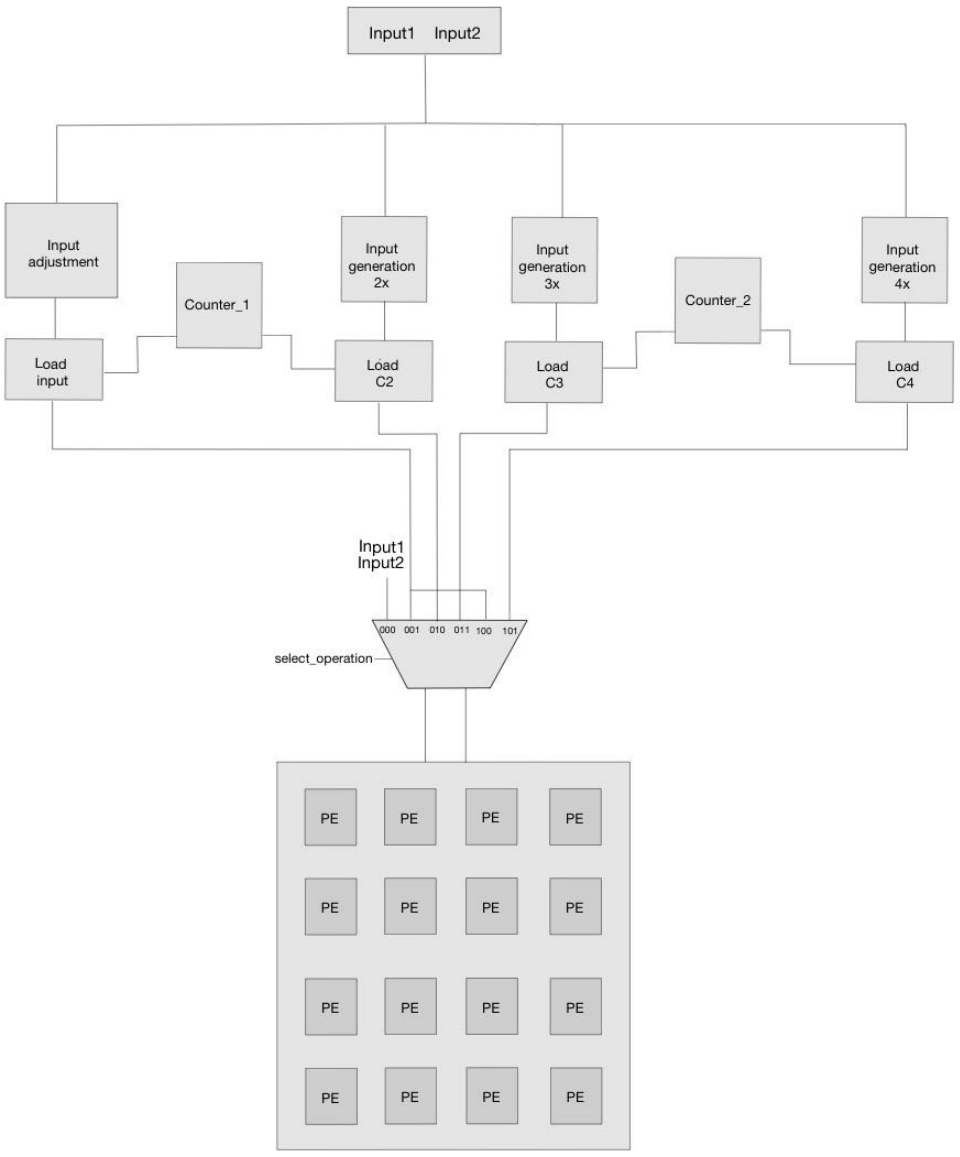


Figure 3.10: Datapath

3.5.1 Matrix multiplication

When a matrix multiplication is performed, typically each row of one matrix is multiplied by a column of the other matrix, and all these partial results are summed together.
To achieve this in this implementation, each PE of the array manages the multiplication of an entire row by an iterated column. For instance, to multiply two 4x4 matrices, the first row of the PE array provides the result of the first row multiplied by all the columns, as shown in Figure 3.11. The idea is that in each clock cycle, each PE receives a different pair of values to be multiplied and added to the previous result: each PE performs a MAC operation each time.

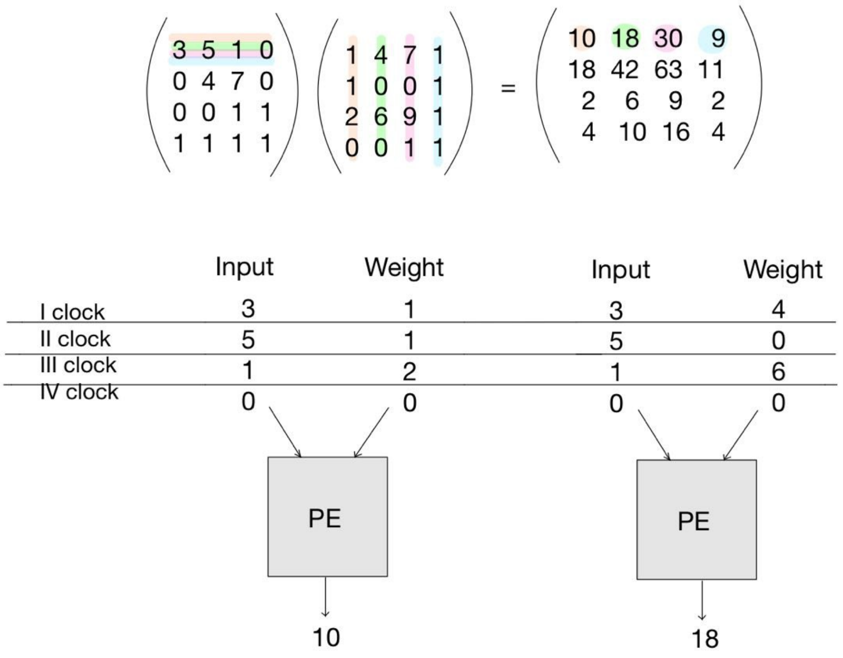


Figure 3.11: Example of matrix multiplication

To achieve this behavior, three blocks were created in the datapath: the input adjustment, the load input, and the counter. The idea is that the user can insert the two matrices to be multiplied, and these will be processed by the input adjustment block. This block outputs a matrix with four rows and two columns. Each row contains the 'input' and 'weight' matrices needed by the array for each clock cycle. Figure 3.12 shows an example of the matrix sent in the first two clock cycles of the previous example.

I clock
$$\begin{pmatrix} 3 & 3 & 3 & 3 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 4 & 7 & 1 \\ 1 & 4 & 7 & 1 \\ 1 & 4 & 7 & 1 \\ 1 & 4 & 7 & 1 \end{pmatrix}$$

II clock
$$\begin{pmatrix} 5 & 5 & 5 & 5 \\ 4 & 4 & 4 & 4 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$

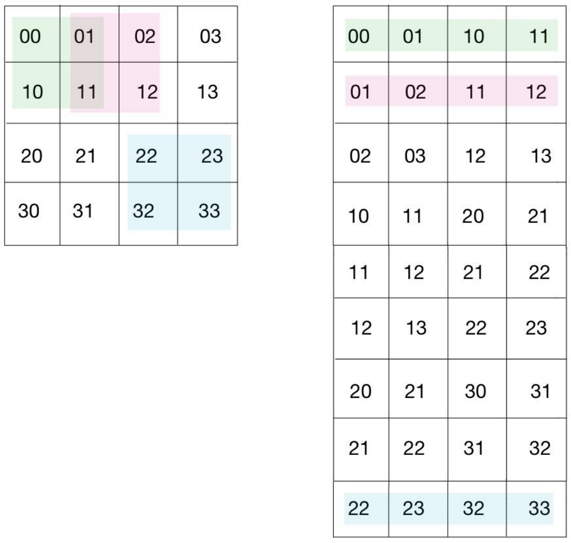
III clock
$$\vdots$$

Figure 3.12: Example of convolution array inputs

The load input block then takes the correct input to send to the array from the matrix. To select the correct one, it uses a counter that determines which row will be sent as output. In the example of Figure 3.12, the counter counts until 4. If the size of the matrix that the user wants to multiply is smaller than the 4x4 case, the remaining positions must be filled with zero values.

3.5.2 Convolution

The convolution operation can be of three different types: the input matrix is always a 4x4 matrix, but the kernel can be 2x2, 3x3, or 4x4.
The idea and structure are similar to the matrix multiplication case. In this case, there are different blocks called 'input generation 2x', 'input generation 3x', and 'input generation 4x', and the respective '2x load input', '3x load input', and '4x load input'. Take the 2x2 case as a case study. The idea is that each PE will handle one complete multiplication between the kernel and 4 matrix values. To achieve this, a matrix is constructed that will contain in each row all the values that each PE needs to take as input in different clock cycles. In the case of a 2x2 kernel, 9 PEs work together. In the case of a 3x3 kernel, just 4 PEs work together, and in the case of a 4x4 kernel, only one PE is working.



00	01	02	03
10	11	12	13
20	21	22	23
30	31	32	33

00	01	10	11
01	02	11	12
02	03	12	13
10	11	20	21
11	12	21	22
12	13	22	23
20	21	30	31
21	22	31	32
22	23	32	33

Figure 3.13: Example of array inputs

3.6 Control Unit

This section describes the structure of the Control Unit (CU) that manages and sets all the commands required to perform the requested operations.
When the user wants to operate, this one has to set different commands:

- "START": is the signal that has to be set at '1' when the user wants to start an operation
- "select_operation": This is a 3-bit signal that selects the operation the user wants to perform: normal operation (only parallel multiplication, only parallel addition, only parallel MAC), matrix multiplication, or different types of convolutions
- "config_MAC_mult_adder": This signal selects the bit width the user wants to work with. It is a 7-bit signal. Let's analyze it in detail:
  - Bits 1 and 0 determine the MAC operation: they decide whether the input of the adder comes from the multiplier or is external.
  - Bits 4 to 2 specify the multiplication:
    - "000" selects the 16x16 bit width operation.
    - "010" selects the 8x8 bit width operation.
    - "001" selects the 4x4 bit width operation.
  - Bits 7 to 5 specify the adder:
    - "000" selects the sum of two 4-bit numbers, with a 5-bit result. This configuration will produce 8 outputs.

   * "001" selects the sum of two 8-bit numbers, with a 9-bit result. This configuration will produce 4 outputs.
   * "011" selects the sum of two 16-bit numbers, with a 17-bit result. This configuration will produce 2 outputs.
   * "010" selects the sum of two 32-bit numbers, with a 32-bit result. This configuration will produce 1 output. This is the MAC case.
   * "110" selects the sum of two 32-bit numbers, with a 33-bit result. This configuration will produce 1 output.

Figure 3.14 and Figure 3.15 illustrate the sequence of states based on the selected command.
First, here is a summary of all the different states present in the control unit:

- IDLE:
  If the RESET signal is activated or some commands are given incorrectly, the system enters the IDLE state, where everything is reset.
- MULT_16 :
  All commands to perform a simple 16x16 multiplication are set.
- MULT_8 :
  All commands to perform a simple 8x8 multiplication are set.
- MULT_4 :
  All commands to perform a simple 4x4 multiplication are set.
- RES_SUM_5:
  All commands to perform a simple addition of 4+4 are set.
- RES_SUM_8:
  All commands to perform a simple addition 8+8 are set.
- RES_SUM_16:
  All commands to perform a simple addition 16+16 are set.
- RES_SUM_32:
  All commands to perform a simple addition 32+32 are set.
- SINGLE_MAC16:
  All commands to perform a simple MAC operation 16x16 are set.
- SINGLE_MAC8:
  All commands to perform a simple MAC operation 8x8 are set.
- SINGLE_MAC4:
  All commands to perform a simple MAC operation 4x4 are set.
- INPUT_GENERATION:
  The block that creates the correct matrix to be sent to the PE array for matrix multiplication is activated.
- LOAD_MM:
  The counter is activated and the right input is sent to the array.

- MAC_16:
  Commands for a 16x16 MAC operation are set, running in a loop for matrix multiplication or convolution.
- MAC_8:
  Commands for an 8x8 MAC operation are set, running in a loop for matrix multiplication or convolution.
- MAC_4:
  Commands for a 4x4 MAC operation are set, running in a loop for matrix multiplication or convolution.
- WAIT & WAIT2 & WAIT_MULT1 & WAIT_ADDER:
  These states are used to give the array time to complete the result.
- EN_REG:
  The adder is enabled to sum the correct value.
- DIS_REG:
  The adder is disabled.
- INC_CNT1 & INC_CNT2:
  The counter is incremented.
- INPUT_GENERATION_2X:
  The block that creates the correct matrix for the 2x2 convolution is activated.
- LOAD_C2:
  The counter is activated, and the correct inputs are sent to the array.
- INPUT_GENERATION_3X:
  The block that creates the correct matrix for the 3x3 convolution is activated.
- LOAD_C3:
  The counter is activated, and the correct input is sent to the array.
- INPUT_GENERATION_4X:
  The block that creates the correct matrix for the 4x4 convolution is activated.
- LOAD_C4:
  The counter is activated, and the correct inputs are sent to the array.
- DONE:
  If the START signal is not activated again, this state is activated, and the operation is finished.

Figures 3.14 and 3.15 illustrate the structure of the CU. The figures are divided into two parts for better visibility: the first figure shows the cases of simple multiplication, simple addition, MAC, and matrix multiplication, while the second figure shows the convolutions.

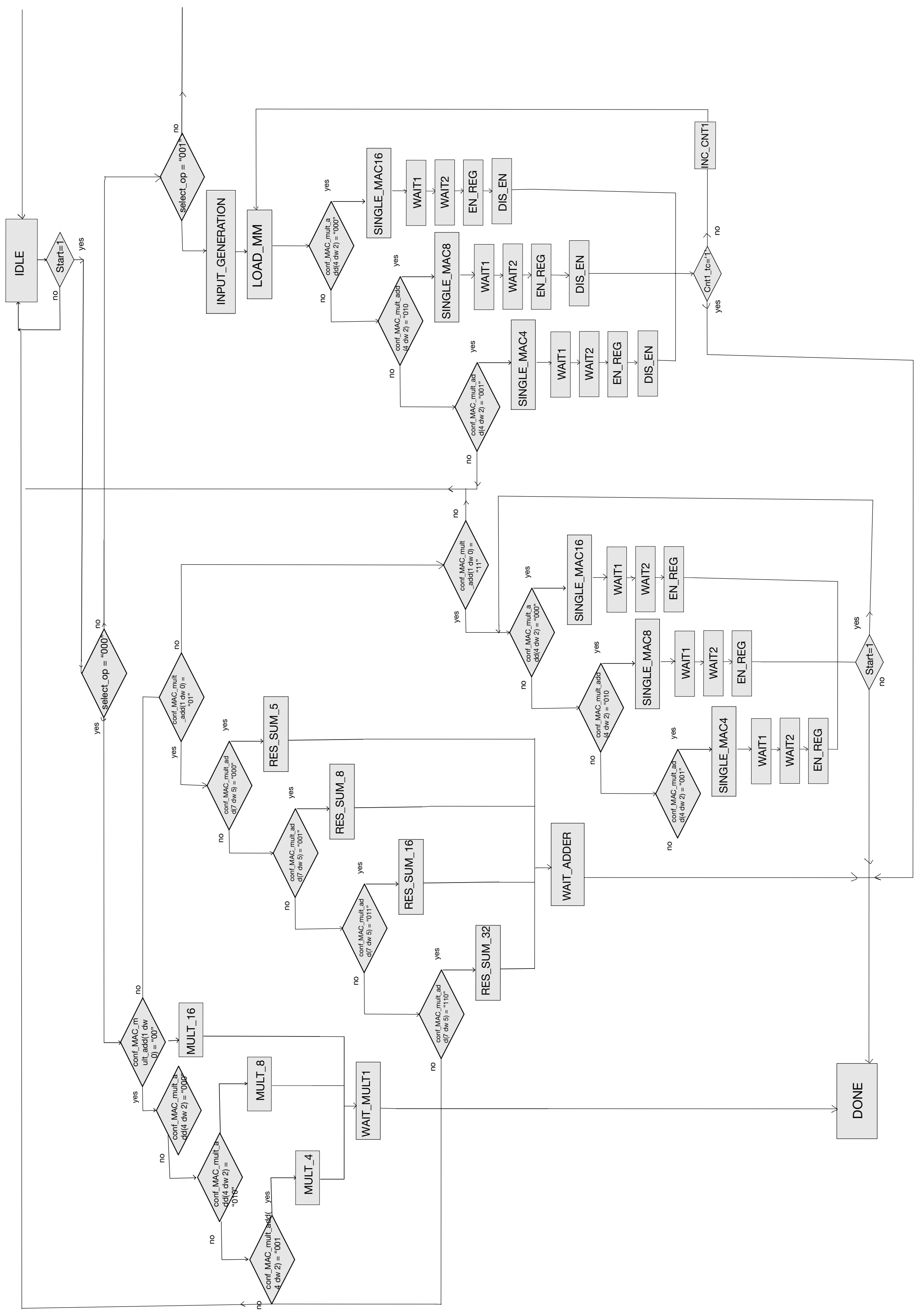

IDLE
Start=1
yes
no
select_op = "001"
INPUT_GENERATION
LOAD_MM
conf_MAC_mult_add(4 dw 2) = "000"
SINGLE_MAC16
WAIT1
WAIT2
EN_REG
DIS_EN
conf_MAC_mult_add(4 dw 2) = "010"
SINGLE_MAC8
conf_MAC_mult_add(4 dw 2) = "001"
SINGLE_MAC4
INC_CNT1
Cnt1_tc='1'
select_op = "000"
conf_MAC_mult_add(1 dw 0) = "11"
conf_MAC_mult_add(1 dw 0) = "01"
conf_MAC_mult_add(7 dw 5) = "000"
RES_SUM_5
conf_MAC_mult_add(7 dw 5) = "001"
RES_SUM_8
conf_MAC_mult_add(7 dw 5) = "011"
RES_SUM_16
conf_MAC_mult_add(7 dw 5) = "110"
RES_SUM_32
WAIT_ADDER
conf_MAC_mult_add(1 dw 0) = "00"
MULT_16
MULT_8
MULT_4
WAIT_MULT1
DONE

Figure 3.14: CU first part

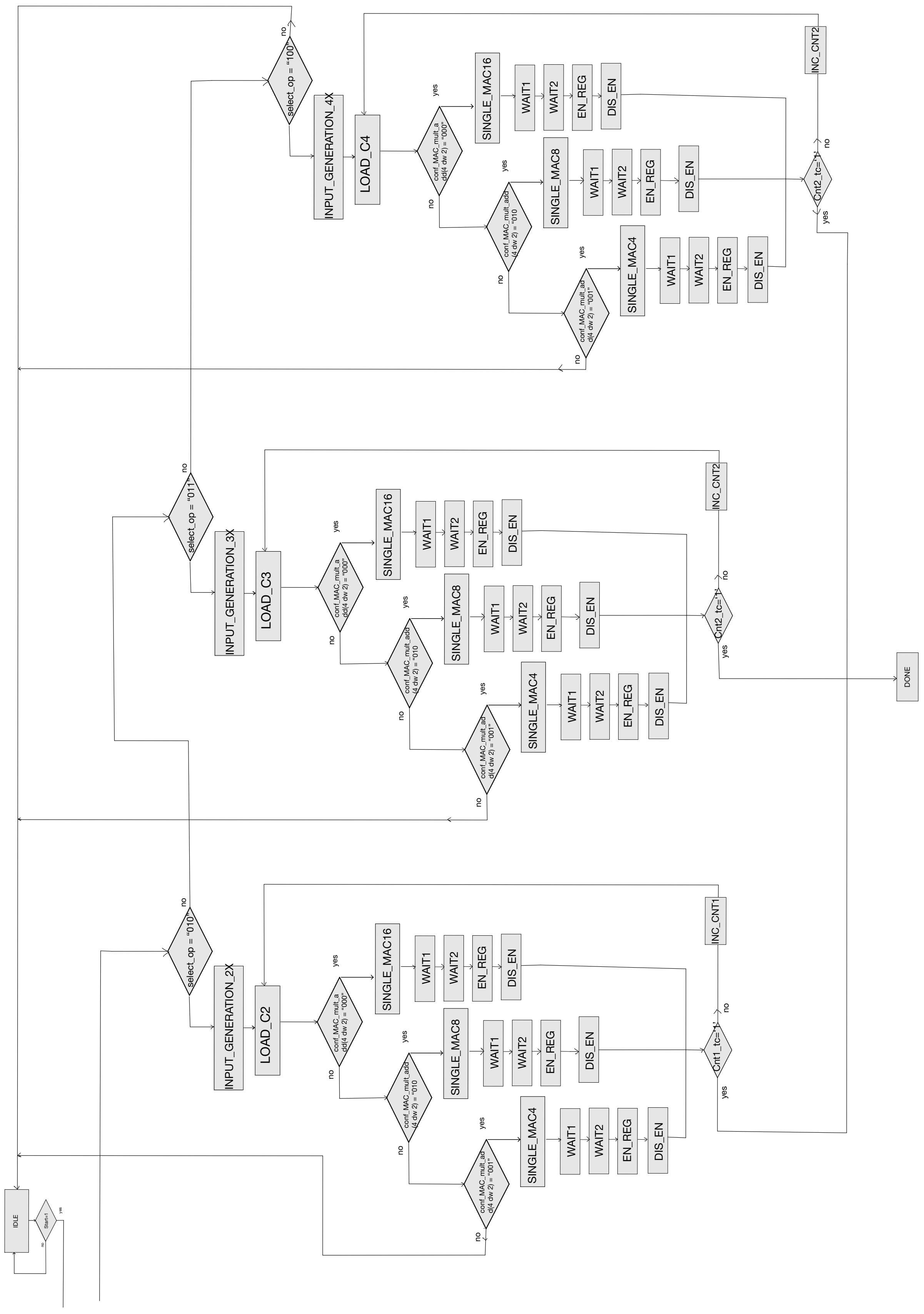


Figure 3.15: CU second part

Chapter 4

Results and Discussions

This chapter contains the evaluation of the NPU and a discussion of the results. This includes the tests of all operation's typologies and a comparison with the other existent architectures.

4.1 Implementation results

This section focuses on the synthesis and implementation of the current design. The following are the results of the synthesis and implementation of NPU design. The target device for synthesis is a Xilinx FPGA on the Pynq-z2 Evaluation Board. Based on the result, the maximum operable frequency of the design is 200 MHz.
The table 4.1 shows the utilization of FPGA resources in the design .

Implementation	LUTs	FFs
NPU	7487	9118
CU	1695	55
Datapath	5791	9063

Table 4.1: Resource utilization

Table 4.2 shows the total power consumption for each different precision. To obtain the different consumption for each case generated from the simulator the different SAIF files were used.

OPERATION	STATIC POWER [W]	DYNAMIC POWER [W]	TOTAL POWER [W]
Multiplication 16*16	0.106	0.142	0.248
Multiplication 8*8	0.106	0.091	0.197
Multiplication 4*4	0.106	0.086	0.192
Sum 32+32	0.110	0.394	0.504
Sum 16+16	0.110	0.343	0.453
Sum 8+8	0.109	0.330	0.439
Sum 4+4	0.109	0.277	0.386
MAC 16*16	0.105	0.066	0.171
MAC 8*8	0.105	0.067	0.172
MAC 4*4	0.105	0.065	0.170
MM 16*16	0.107	0.153	0.258
MM 8*8	0.107	0.146	0.253
MM 4*4	0.106	0.129	0.235
Convolution 2X 16*16	0.107	0.196	0.303
Convolution 2X 8*8	0.107	0.187	0.294
Convolution 2X 4*4	0.107	0.154	0.261
Convolution 3X 16*16	0.107	0.195	0.302
Convolution 3X 8*8	0.107	0.186	0.293
Convolution 3X 4*4	0.107	0.153	0.260
Convolution 4X 16*16	0.107	0.198	0.305
Convolution 4X 8*8	0.107	0.192	0.299
Convolution 4X 4*4	0.107	0.162	0.269

Table 4.2: Power analysis for the different operations at different precisions

As can be seen from the table 4.2, the static power remains more or less constant, while the dynamic power varies greatly among the different operations. This is mainly because, depending on the selected operation, some components are activated or not. For example, to perform a simple operation, the inputs are passed directly to the array, while for matrix multiplication, the counter enable, input adjustment, and load input are activated, and thus they work each time the counter is incremented. The same applies to convolutions.

The power consumption in various operations decreases as the precision decreases. This is due to the reduced switching activity during the operations. For example, in the multiplier, when the precision is 4 bits, only zeros will reach the two innermost adders, resulting in lower switching activity and, consequently, lower power consumption.
The results indicate that there is higher power consumption when using the adder-only mode with external inputs. This could be due to the greater switching activity compared to using internal inputs derived from the multiplier. External inputs may have higher load capacitance than internal inputs, as the power consumption of the I/Os can be higher than that of internally generated signals within the chip. Additionally, the presence of registers to synchronize the arrival of the external inputs introduces a greater capacitive load that needs to be charged and discharged during each clock cycle.
It can be seen that the biggest consumption is due to the dynamic power.

4.2 Simulation results

In this section are shown all the simulation results for each case treated by the implemented design.
The simulations are realized working with the maximum operable frequency of 200 MHz, with a 5 ns clock cycle period. The data radix in the figures for the input and the output matrices is hexadecimal. The reported time to execute each operation doesn't take into account the time of the IDLE state. In the figures is possible to notice also the present state (PS) of the CU for each operation.

- For the parallel multiplication operations the result is available after 10 ns from the beginning of the operation, after two clock cycles.
    - Case 16x16:

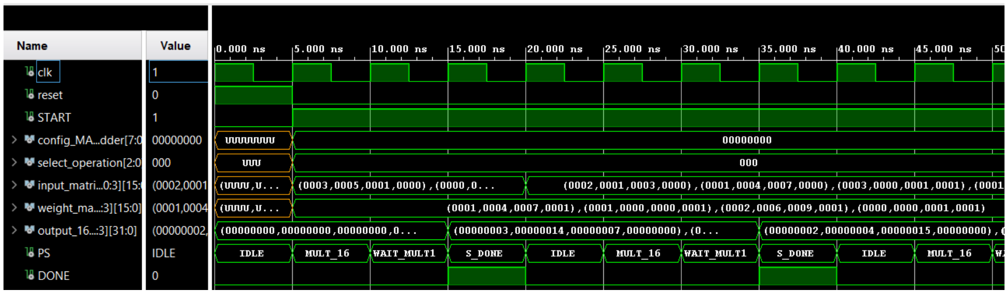


Figure 4.1: Simulation of parallel multiplications 16x16

    - Case 8x8 :

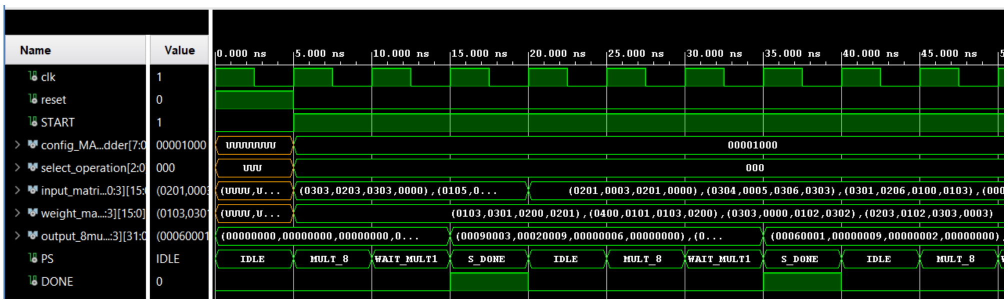


Figure 4.2: Simulation of parallel multiplications 8x8

– Case 4x4:

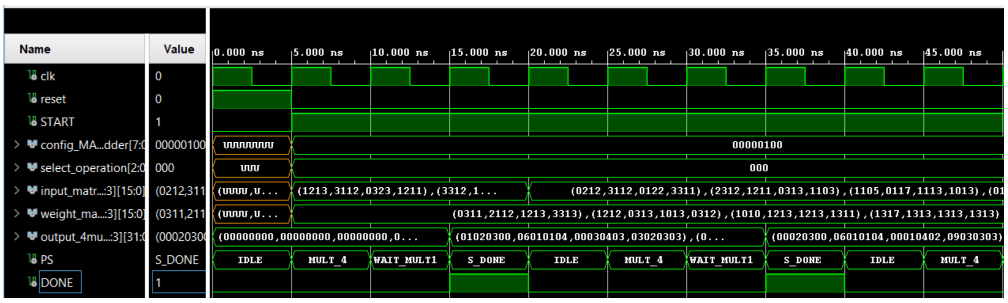


Figure 4.3: Simulation of parallel multiplications 4x4

- For the parallel addition operations the result is available after 10 ns from the beginning of the operation, after two clock cycles.

  – Case 32+32:

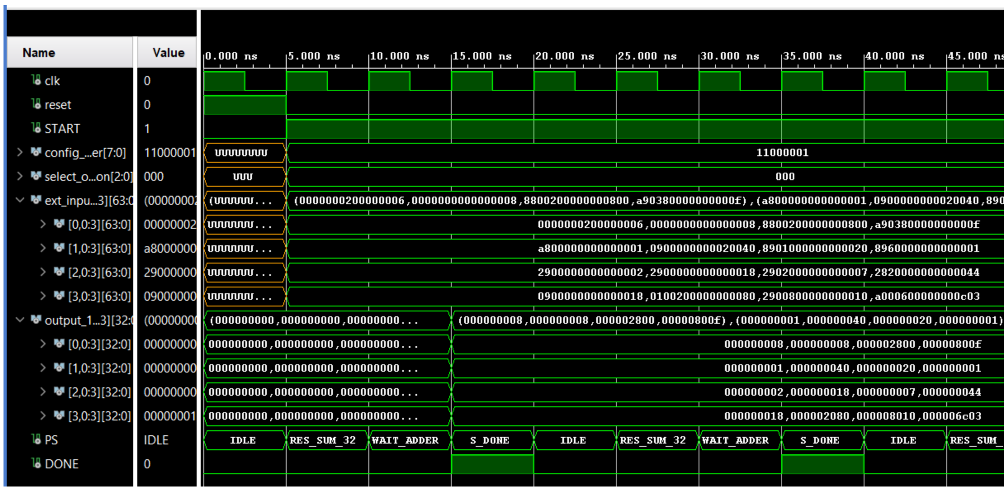
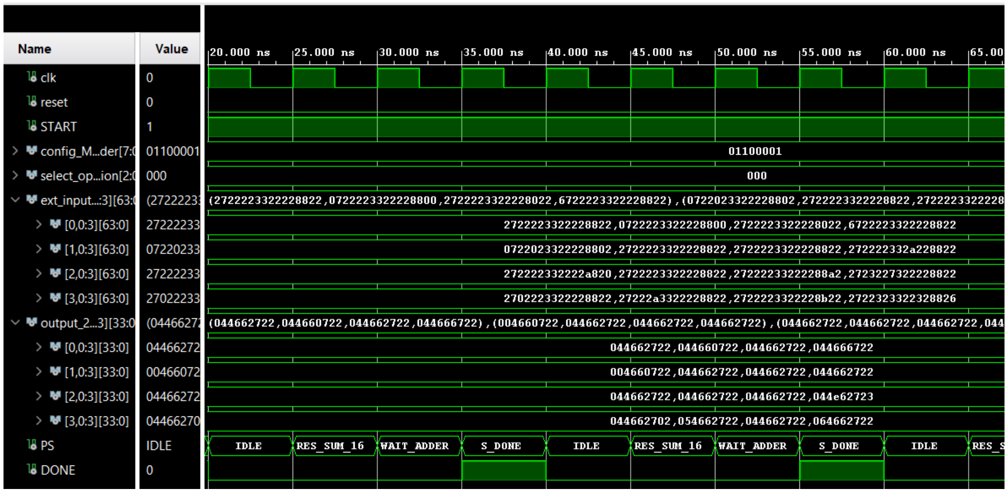

Name
Value
clk 0
reset 0
START 1
config_...er[7:0] 11000001
select_o...on[2:0] 000
ext_inpu...3][63:0]
[0,0:3][63:0]
[1,0:3][63:0]
[2,0:3][63:0]
[3,0:3][63:0]
output_1...3][32:0]
[0,0:3][32:0]
[1,0:3][32:0]
[2,0:3][32:0]
[3,0:3][32:0]
PS IDLE
DONE 0
11000001
000
0000000200000006,0000000000000008,8800200000000800,a90380000000000f
a800000000000001,0900000000020040,8901000000000020,8960000000000001
2900000000000002,2900000000000018,2902000000000007,2820000000000044
0900000000000018,0100200000000080,2900800000000010,a000600000000c03
000000008,000000008,000002800,00000800f
000000001,000000040,000000020,000000001
000000002,000000018,000000007,000000044
000000018,000002080,000008010,000006c03
IDLE RES_SUM_32 WAIT_ADDER S_DONE IDLE RES_SUM_32 WAIT_ADDER S_DONE IDLE RES_SUM

Figure 4.4: Simulation of parallel additions 32+32

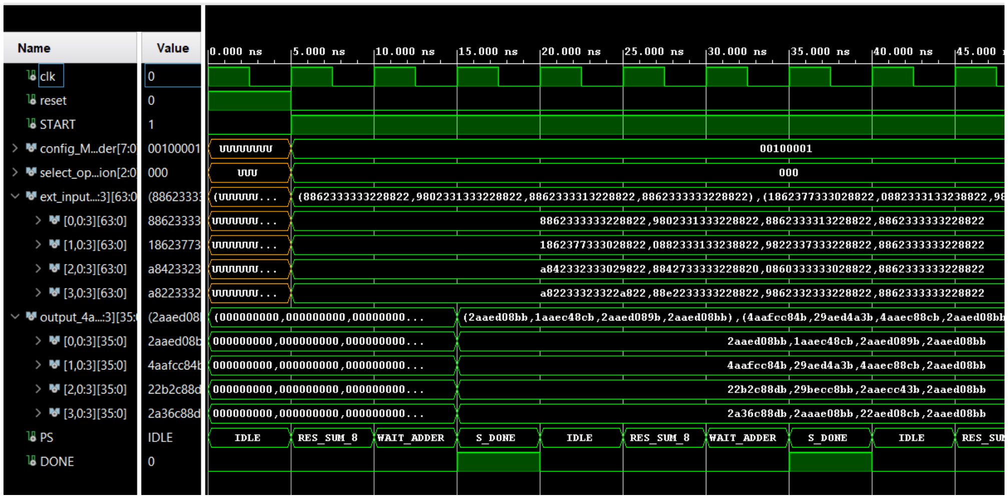
– Case 16+16:



```
Name                 Value       20.000 ns  25.000 ns  30.000 ns  35.000 ns  40.000 ns  45.000 ns  50.000 ns  55.000 ns  60.000 ns  65.00
clk                  0
reset                0
START                1
config_M...der[7:0   01100001    01100001
select_op...ion[2:0  000         000
ext_input...3][63:0  (27222233   (2722223322228822,0722223322228800,2722223322228022,6722223322228822),(0722023322228802,2722223322228822,2722223322228
  [0,0:3][63:0]      27222233    2722223322228822,0722223322228800,2722223322228022,6722223322228822
  [1,0:3][63:0]      07220233    0722023322228802,2722223322228822,2722223322228822,272222332a228822
  [2,0:3][63:0]      27222233    272222332222a820,2722223322228822,27222233222288a2,2723227322228822
  [3,0:3][63:0]      27022233    2702223322228822,27222a3322228822,2722223322228b22,2722323322328826
output_2...3][33:0   (04466272   (044662722,044660722,044662722,044666722),(004660722,044662722,044662722,044662722),(044662722,044662722,044662722,044
  [0,0:3][33:0]      04466272    044662722,044660722,044662722,044666722
  [1,0:3][33:0]      00466072    004660722,044662722,044662722,044662722
  [2,0:3][33:0]      04466272    044662722,044662722,044662722,044e62723
  [3,0:3][33:0]      04466270    044662702,054662722,044662722,064662722
PS                   IDLE        IDLE  RES_SUM_16  WAIT_ADDER  S_DONE  IDLE  RES_SUM_16  WAIT_ADDER  S_DONE  IDLE  RES_S
DONE                 0
```

Figure 4.5: Simulation of parallel additions 16+16

– Case : 8+8 :



```
Name                 Value       0.000 ns  5.000 ns  10.000 ns  15.000 ns  20.000 ns  25.000 ns  30.000 ns  35.000 ns  40.000 ns  45.000
clk                  0
reset                0
START                1
config_M...der[7:0   00100001    UUUUUUUU  00100001
select_op...ion[2:0  000         UUU       000
ext_input...3][63:0  (88623333   (UUUUUU...  (8862333333228822,9802331333228822,8862333313228822,8862333333228822),(1862377333028822,0882333133238822,98
  [0,0:3][63:0]      88623333    UUUUUUU...  8862333333228822,9802331333228822,8862333313228822,8862333333228822
  [1,0:3][63:0]      18623773    UUUUUUU...  1862377333028822,0882333133238822,9822337333228822,8862333333228822
  [2,0:3][63:0]      a8423323    UUUUUUU...  a842332333029822,8842733333228820,0860333333028822,8862333333228822
  [3,0:3][63:0]      a8223332    UUUUUUU...  a82233323322a822,88e2233333228822,9862332333228822,8862333333228822
output_4a...3][35:0  (2aaed08b   (000000000,000000000,00000000...  (2aaed08bb,1aaec48cb,2aaed089b,2aaed08bb),(4aafcc84b,29aed4a3b,4aaec88cb,2aaed08bb
  [0,0:3][35:0]      2aaed08b    000000000,000000000,000000000...  2aaed08bb,1aaec48cb,2aaed089b,2aaed08bb
  [1,0:3][35:0]      4aafcc84b   000000000,000000000,000000000...  4aafcc84b,29aed4a3b,4aaec88cb,2aaed08bb
  [2,0:3][35:0]      22b2c88d    000000000,000000000,000000000...  22b2c88db,29becc8bb,2aaecc43b,2aaed08bb
  [3,0:3][35:0]      2a36c88d    000000000,000000000,000000000...  2a36c88db,2aaae08bb,22aed08cb,2aaed08bb
PS                   IDLE        IDLE  RES_SUM_8  WAIT_ADDER  S_DONE  IDLE  RES_SUM_8  WAIT_ADDER  S_DONE  IDLE  RES_SUM
DONE                 0
```

Figure 4.6: Simulation of parallel additions 8+8

– Case 4+4:

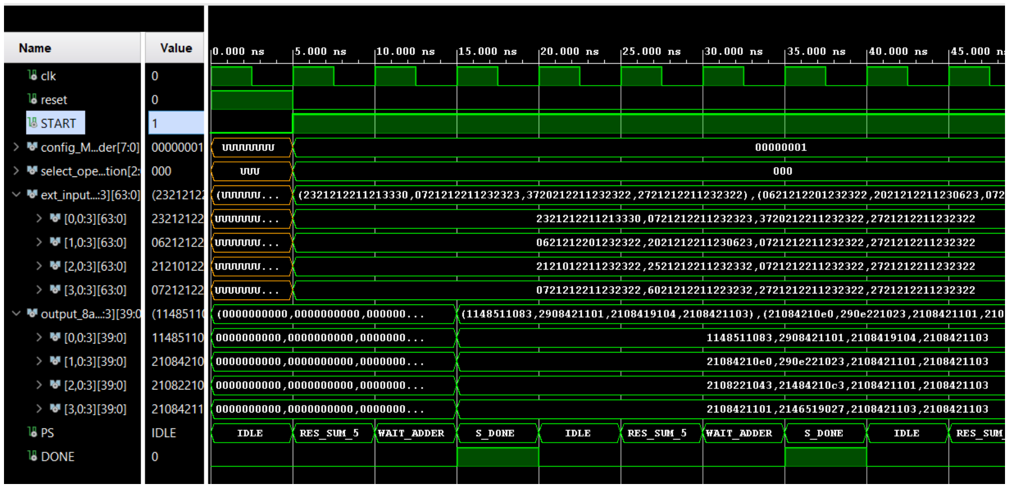


Figure 4.7: Simulation of parallel additions 4+4

- Case MAC operation: to show the implementation of this operation, two consecutive inputs are given to the PE array. For one single operation, the needed time from the start of the operation is 20 ns (four clock cycle, without the idle time), and for two consecutive operations, to test the idea of MAC, are needed 40 ns.

  - Case 16x16:

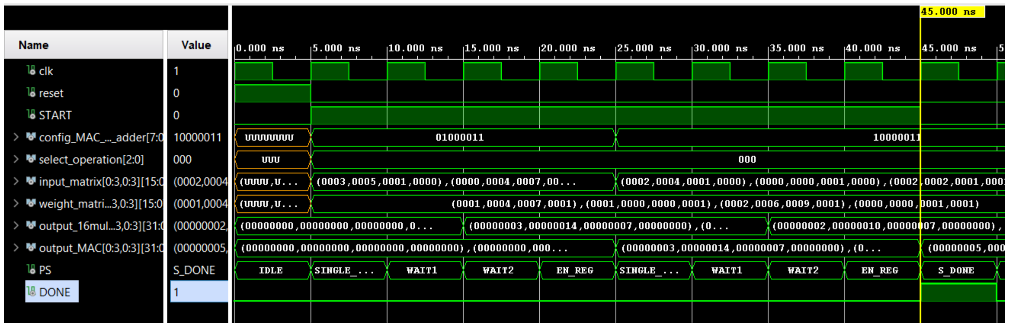


Figure 4.8: Simulation of MAC operation 16x16

  - Case 8x8:

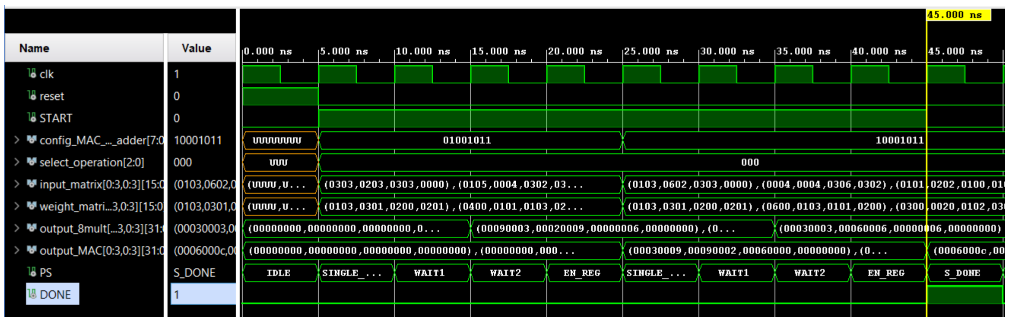


Figure 4.9: Simulation of MAC operation 8x8

– Case 4x4:

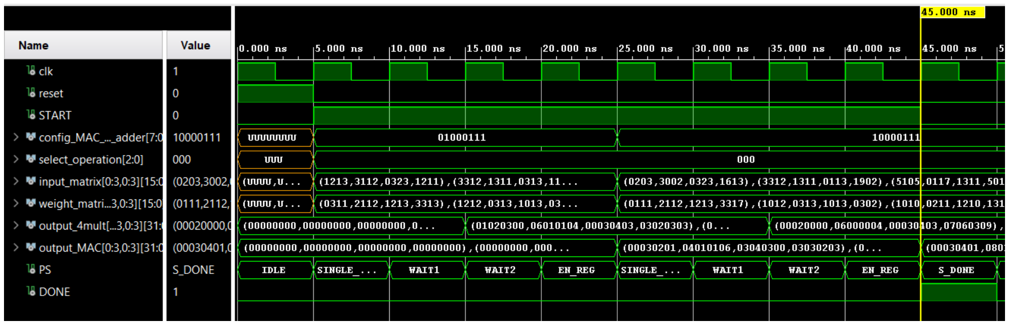

Name
Value
clk 1
reset 0
START 0
config_MAC_..._adder[7:0] 10000111
select_operation[2:0] 000
input_matrix[0:3,0:3][15:0]
weight_matri...3,0:3][15:0]
output_4mult[...3,0:3][31:0]
output_MAC[0:3,0:3][31:0]
PS S_DONE
DONE 1
45.000 ns
IDLE
SINGLE_...
WAIT1
WAIT2
EN_REG
S_DONE

Figure 4.10: Simulation of MAC operation 4x4

- For the Matrix Multiplication (MM) operations, the output is available (without taking into account the idle time) after 135 ns, in the 'disable reg' state. However, this state is necessary for a hypothetical consecutive execution of the operation, so with the done state all the operation counts 140 ns.

  – Case 16x16:

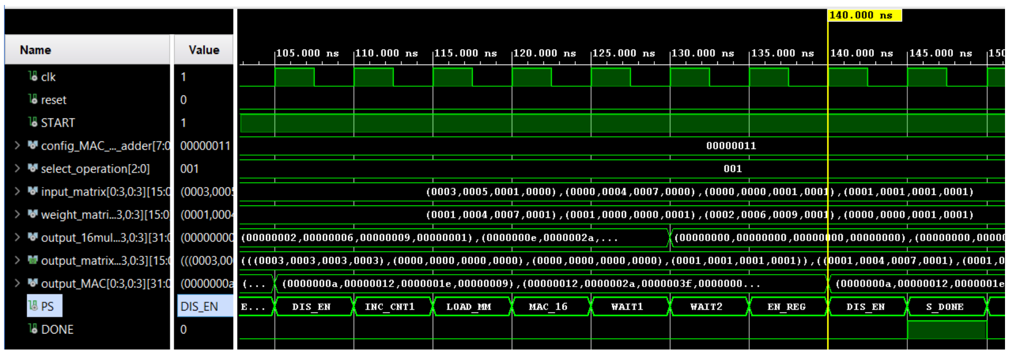

Name
Value
clk 1
reset 0
START 1
config_MAC_..._adder[7:0] 00000011
select_operation[2:0] 001
input_matrix[0:3,0:3][15:0]
weight_matri...3,0:3][15:0]
output_16mul...3,0:3][31:0]
output_matrix...3,0:3][15:0]
output_MAC[0:3,0:3][31:0]
PS DIS_EN
DONE 0
140.000 ns
DIS_EN
INC_CNT1
LOAD_MM
MAC_16
WAIT1
WAIT2
EN_REG
S_DONE

Figure 4.11: Simulation of MM operation 16x16

  – Case 8x8:

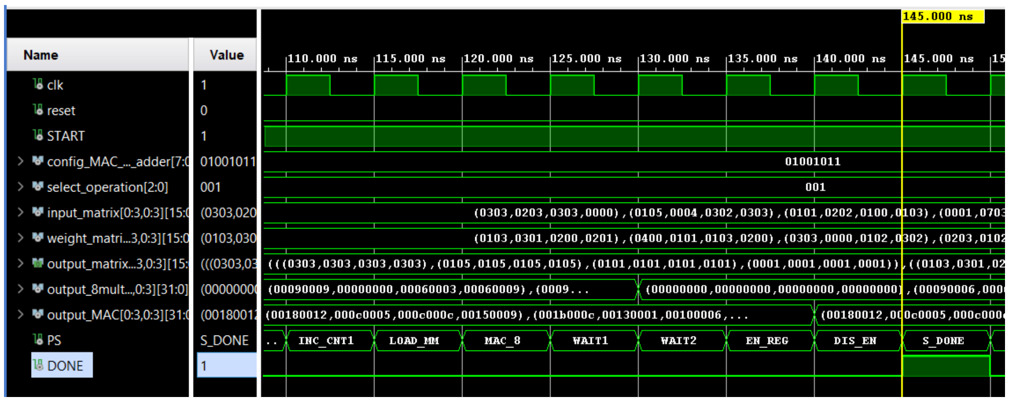

Name
Value
clk 1
reset 0
START 1
config_MAC_..._adder[7:0] 01001011
select_operation[2:0] 001
input_matrix[0:3,0:3][15:0]
weight_matri...3,0:3][15:0]
output_matrix...3,0:3][15:0]
output_8mult...0:3][31:0]
output_MAC[0:3,0:3][31:0]
PS S_DONE
DONE 1
145.000 ns
INC_CNT1
LOAD_MM
MAC_8
WAIT1
WAIT2
EN_REG
DIS_EN
S_DONE

Figure 4.12: Simulation of MM operation 8x8

– Case 4x4:

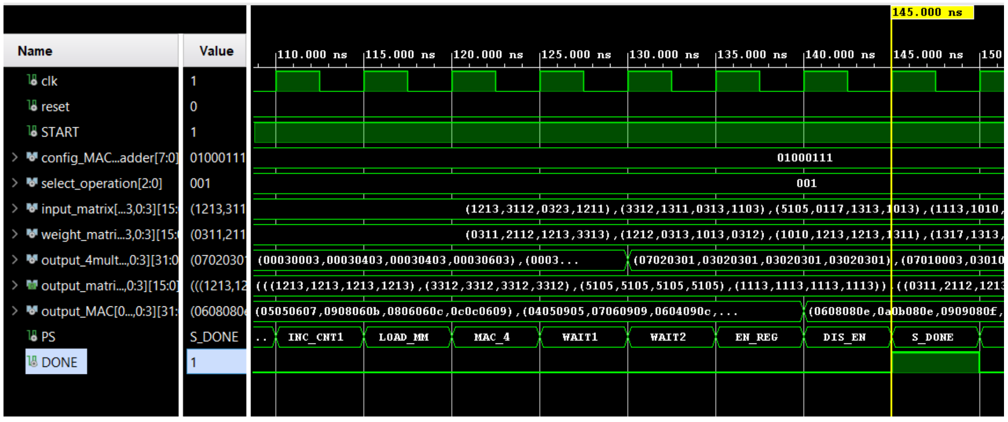


Figure 4.13: Simulation of MM operation 4x4

- For the 2X convolution operation the output is available (without taking into account the idle time) after 140 ns, in the ’disable reg’ state. However, this state is necessary for a hypothetical consecutive execution of the operation, so with the done state all the operation counts 145 ns.

– Case 16x16:

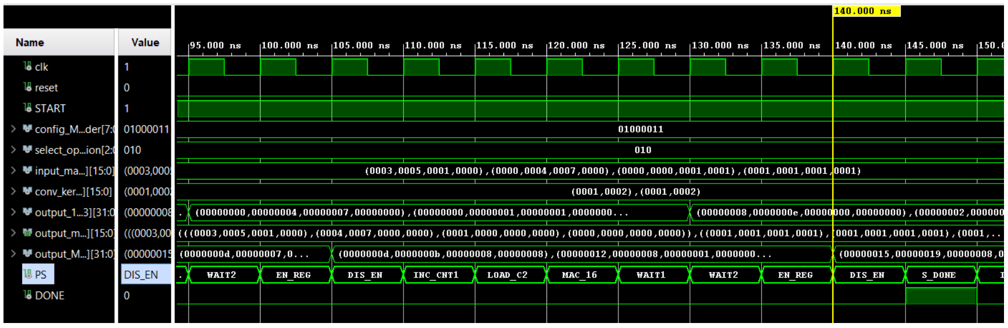


Figure 4.14: Simulation of 2X convolution operation 16x16

– Case 8x8:

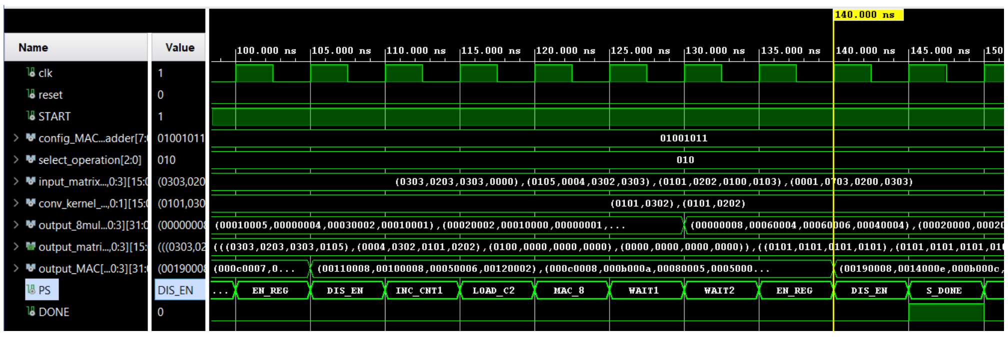


Figure 4.15: Simulation of 2X convolution operation 8x8

– Case 4x4:

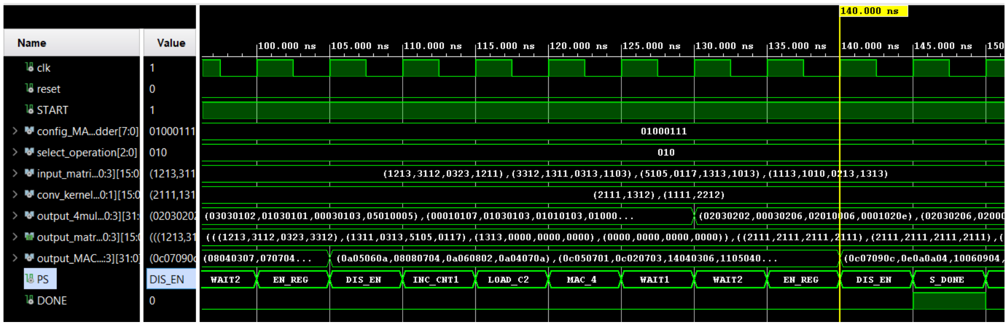


Figure 4.16: Simulation of 2X convolution operation 4x4

- For the 3X convolution operation the output is available (without taking into account the idle time) after 310 ns, in the 'disable reg' state. However, this state is necessary for a hypothetical consecutive execution of the operation, so with the done state all the operation counts 315 ns.

– Case 16x16:

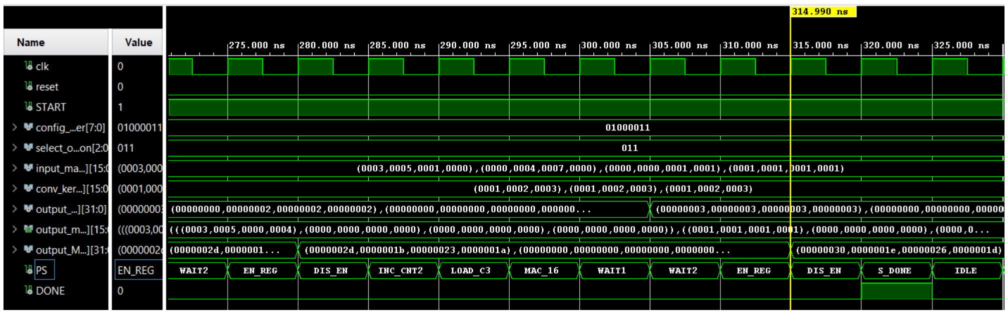


Figure 4.17: Simulation of 3X convolution operation 16x16

– Case 8x8:

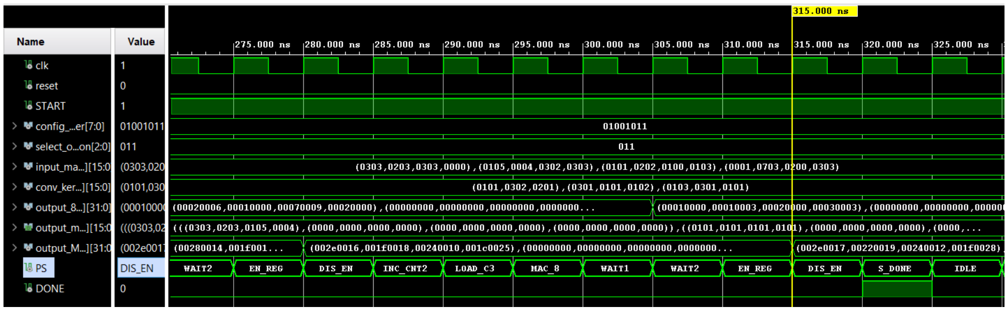


Figure 4.18: Simulation of 3X convolution operation 8x8

– Case 4x4:

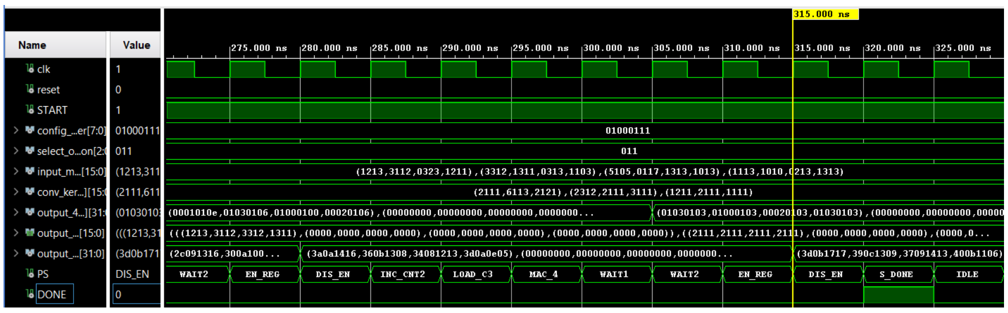


Figure 4.19: Simulation of 3X convolution operation 4x4

- For the 4X convolution operation the output is available (without taking into account the idle time) after 555 ns, in the 'disable reg' state. However, this state is necessary for a hypothetical consecutive execution of the operation, so with the done state all the operation counts 560 ns.

  – Case 16x16:

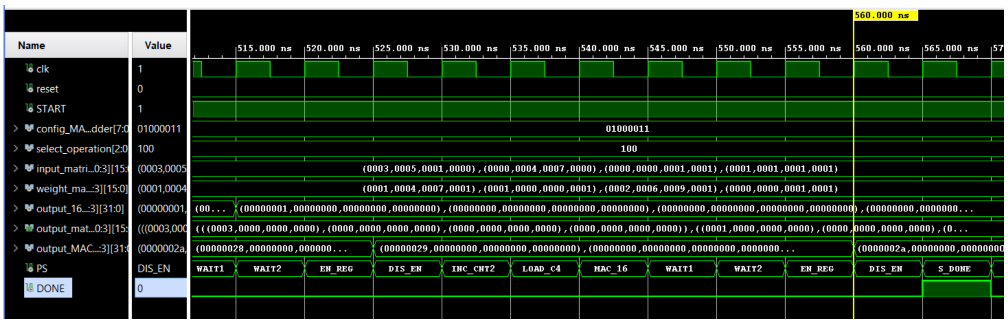


Figure 4.20: Simulation of 4X convolution operation 16x16

  – Case 8x8:

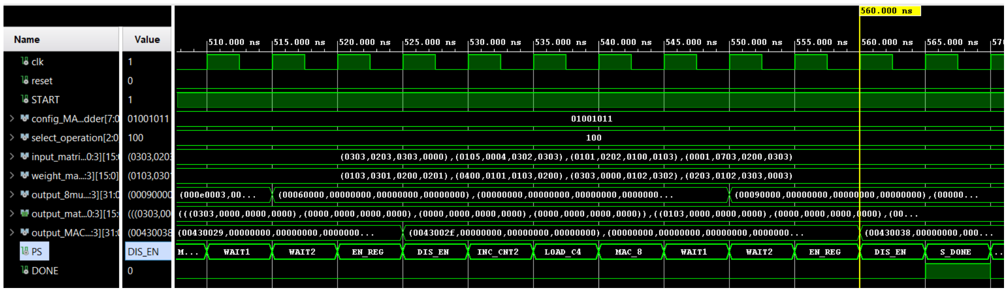


Figure 4.21: Simulation of 4X convolution operation 8x8

  – Case 4x4:

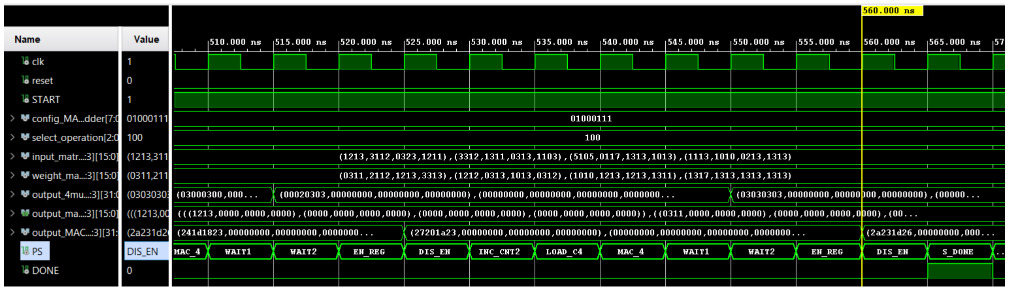


Figure 4.22: Simulation of 4X convolution operation 4x4

It is now evaluated the op/s for each type of operation.
To evaluate this one are taking into account:

- The total operation realized from each processing element in the array.
- The total operation for all the arrays.
- The total number of clock cycles to complete the operation, considering also the state (clock cycle) in which the result is given. For instance, for multiplication, the total clock cycle considered is three because the result is available after the third one.
- The frequency at which the operation is executed.

In general:

$$\frac{op}{s} = \frac{OperationforPE \cdot NumberofPE}{Executiontime}$$

Operation	Op/s [MOp/s]
Multiplication	1066
Sum	1066
MAC	640
Matrix Multiplication	457
Convolution 2X	514
Convolution 3X	228
Convolution 4X	57

Table 4.3: Op/s for the different operations

As expected this number is larger as the operation is simple.
Known the op/s and the power consumed in each operation, it is possible also to evaluate the energy needed from each operation.

$$E = \frac{Op/s}{Power}$$

Operation	Power [W]	Op/s [MOp/s]	E [Gops/s/W]
Multiplication 16*16	0.248	1066	4.298
Multiplication 8*8	0.197	1066	5.411
Multiplication 4*4	0.192	1066	5.552
Sum 32+32	0.504	1066	2.115
Sum 16+16	0.453	1066	2.353
Sum 8+8	0.439	1066	2.428
Sum 4+4	0.386	1066	2.761
MAC 16*16	0.171	640	3.743
MAC 8*8	0.172	640	3.721
MAC 4*4	0.170	640	3.764
MM 16*16	0.258	457	1.771
MM 8*8	0.253	457	1.806
MM 4*4	0.235	457	1.944
Convolution 2X 16*16	0.303	514	1.696
Convolution 2X 8*8	0.294	514	1.748
Convolution 2X 4*4	0.261	514	1.969
Convolution 3X 16*16	0.302	228	0.7549
Convolution 3X 8*8	0.293	228	0.7781
Convolution 3X 4*4	0.260	228	0.8769
Convolution 4X 16*16	0.305	57	0.1868
Convolution 4X 8*8	0.299	57	0.1906
Convolution 4X 4*4	0.269	57	0.2118

Table 4.4: Power, Op/s and Energy for the different operations

As can be observed in Table 4.4, the value of

$$\frac{Op/s}{Power}$$

is lower for more complex operations due to increased latency, and it is also lower as the precision increases. This is due to the higher switching activity.

Chapter 5

Conclusion and Future Scope

This chapter presents the conclusion of this thesis, summarizing the main design aspects and results. In the second section then are presented some possible future works to get through some limitations of this implemented design.

5.1 Conclusion

This work presents the design of an NPU. This proposed design focuses on the realization of a processing element that has to work like a multiplier, like an adder, and like a MAC structure with different data bit-width. In particular, it can receive input data in 4,8 or 16-bit format.
According to the selected bit-width, the number of generated outputs changes: if the data input format is 4-bit, it is possible to work with four inputs simultaneously, and consequently obtain four outputs. If the input format is 8-bit, there are two inputs and two outputs. If the bit-width is 16-bit, it is possible to work with only one input and one output. To realize the PE, a re-configurable multiplier and an adder composed of eight adders at four bits, that can also take external inputs. A 4x4 matrix of this PE was created, which is capable of carrying out parallel operations. In particular, it can perform 16 multiplications, additions, or MAC operations simultaneously. To allow this structure to perform operations like matrix multiplication and convolution, this matrix is inserted in a datapath that sends different inputs to the matrix according to the wanted operation. All this selection is managed by a CU that needs only four commands to execute the chosen operation. The control unit can handle 22 different cases involving various operations and precisions. The operations that require a minimal number of clock cycles are parallel multiplication and parallel addition, each taking 3 clock cycles, whereas the most demanding operation is the 4X convolution, which requires 112 clock cycles. Consequently, the required energy is lower for operations with shorter latencies.
The implementation of the results demonstrates that the maximum operable frequency is 200 MHz. The results show also that the power and the energy required are smaller when the precision is lower, due to the lower switching activity of the component during the evaluation of the result of the operation.

5.2 Future Scope

Although this architecture can realize different types of operations with different bit-widths, it is possible to think about some future works that can improve this architecture.
First of all this architecture can perform operations 4x4, 8x8, and 16x16, which are all symmetric cases. It would be interesting to try to optimize this structure so that it can also deal with non-symmetrical cases such as 16x8, and 8x4.
Another interesting improvement regarding matrix multiplication is that it is always treated as a 4x4 matrix structure. When the requested matrix multiplication has lower parallelism, the 4x4 matrix is filled with zeros. An idea could be to make this structure more efficient for this type of operation without using the zero-filling.
This architecture could also be interesting for developing power-saving techniques to reduce the structure's power consumption. An interesting idea is to implement the power gating technique. With this technique, it will be possible to reduce consumption because the unused logic will not receive the clock signal, thereby reducing switching activity.

Bibliography

[1] Pramila P. Shinde and Seema Shah. “A Review of Machine Learning and Deep Learning Applications”. In: *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)* (Aug. 2018).

[2] Esam Khan et al. “Network Processors for Communication Security: A Review”. In: *2003 IEEE Pacific Rim Conference on Communications Computers and Signal Processing (PACRIM 2003) (Cat. No.03CH37490)* (Aug. 2003), pp. 173–176.

[3] Will Koehrsen. “Deep Neural Network Classifier”. In: *Medium* (July 2017).

[4] Vincent Camus et al. “Review and Benchmarking of Precision-Scalable Multiply-Accumulate Unit Architectures for Embedded Neural-Network Processing”. In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9.4 (Oct. 2019), pp. 697–711. ISSN: 2156-3365.

[5] Mirza Cilimkovic. “Neural Networks and Back Propagation Algorithm”. In: *Institute of Technology Blanchardstown, Blanchardstown Road North Dublin 15.1* (2015).

[6] Ray Bernard. “Deep Learning to the Rescue”. In: *securityinfowatch* (Mar. 2019).

[7] Md Islam and Md Rafiqul Islam. “Modeling Spammer Behavior: Artificial Neural Network vs. Naive Bayesian Classifier”. In: *Artificial Neural Networks - Application* (Apr. 2011).

[8] K. Babulu and H. Parasuram. “FPGA Realization of Radix-4 Booth Multiplication Algorithm for High Speed Arithmetic Logics”. In: *International Journal of Computer Science and Information Technologies* 2.5 (2011), pp. 2–3. ISSN: 0975-9646.

[9] Hardik Sharma et al. “Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Networks”. In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)* (June 2018), pp. 764–775.

[10] Sungju Ryu et al. “BitBlade: Area and Energy-Efficient Precision-Scalable Neural Network Accelerator with Bitwise Summation”. In: *Proceedings of the 56th Annual Design Automation Conference 2019* (June 2019), pp. 697–711.

[11] Luca Urbinati and Mario R. Casu. “A Reconfigurable Multiplier/Dot-Product Unit for Precision-Scalable Deep Learning Applications”. In: *Proceedings of SIE 2022* (2023), pp. 9–14.