POLITECNICO DI TORINO

Master's Degree Course in Electronic Engineering

Master's Degree Thesis

Optimizing FPGA Performance: Leveraging the Razor Technique in Digital Design

Supervisors
Prof. Luciano LAVAGNO
Dr. Filippo MINNELLA

Candidate
Lavinia COMERRO

April 2024

Summary

As silicon integration technology advances and clock frequencies increase, optimizing power and reaching high performances has become crucial in developing Embedded Systems and Systems on Chip (SoC). Field-Programmable Gate Arrays (FPGAs) are semiconductors that offer high flexibility as they can be reprogrammed after manufacturing, offering lower design cost and customization for specific applications. However, this flexibility comes at a price - FPGAs are less performance and energy efficient. Therefore, to achieve significant performance improvements, it is necessary to rely on architectural modifications and technology scaling of digital designs to operate beyond conventional safety limits. Typically, FPGA systems provide large timing guard bands during the design phase to guarantee safe operation across manufacturing and design. In order to achieve the demand power and performance levels, it is feasible to operate above these limits, acknowledging the possibility of errors occurring within the design and then implementing a technique for error detection and correction. This thesis aims to enhance the performance of FPGAs by implementing the Razor error detection and correction technique. The Razor technique is an innovative method that improves the performance of digital circuits, enabling faster clocking. The circuit is sampled twice using two different clocks, and any discrepancies between the two samples are detected, allowing the circuit to recover from timing errors and perform at higher clock speeds. Throughout this thesis, the intent has been to apply Razor to a Convolutional Neural Network (CNN) design for FPGAs using a custom design flow. Experimental results are obtained on the CIFAR-10 dataset using a two-layer NN on Xilinx FPGAs employing High-Level Synthesis (HLS) on both Ultra96-V2 and Kria KV260 boards. In order to implement Razor, two error recovery techniques are utilized. The first technique involves blocking the pipeline for several clock cycles to recover the error driving control signals, while the second approach uses clock gating. Regardless, it was found that applying Razor in FPGA with the proposed methods produced inconclusive results. The following chapters present a precise overview of the Razor technique and its implementation. The framework and design flow for generating NN designs are detailed, followed by an analysis of the methods and problems encountered.

Contents

List of Figures V

1 Introduction 1
1.1 Overview 1
1.2 Field Programmable Gate Arrays (FPGAs) 2
1.2.1 Design Flow 5
1.2.2 Operating Margins on FPGA 6
1.2.3 Error Detection on FPGA 7
1.3 Razor Methodology 8
1.4 Deep Neural Networks Framework 11
1.4.1 Convolutional Neural Networks (CNNs) 11
1.4.2 CNN Framework 12
1.5 Thesis structure 13

2 Related Work 14

3 Implementation 15
3.1 Error Detection 15
3.1.1 Block Design 17
3.1.2 Post-synthesis Netlist Modification 20
3.2 Error Recovery Methods 26
3.2.1 Razor architecture 26
3.2.2 Razor Architecture in HW 28
3.2.3 Driving control signals with Razor error 29
3.2.4 Clock Gating 30
3.3 Post Implementation Simulation 32

4 Frequency Dynamic Reconfiguration 33
4.1 RTL simulation accelerating one convolution block 33
4.1.1 Clock Domain Crossing 34
4.2 RTL simulation Results 37
4.3 Inference 39

4.3.1 Results expectation in HW 41
4.4 Results 43

5 Conclusions and Future Work 46

Bibliography 49

List of Figures

1.1 *Schematic diagram of a simple CLB* 3
1.2 *Block Diagram of an FPGA* 4
1.3 *An interconnection switch matrix of an FPGA* 5
1.4 *Programmable interconnect and switch matrix overview [1]* 5
1.5 *Design Flow, from RTL to bitstream* 7
1.6 *Razor circuit* 8
1.7 *Behavior of the Razor circuit* 9
1.8 *Short Paths Constraints* 10
1.9 *CNN Implementation Flow* 12

3.1 *Razor circuit with negated clock* 16
3.2 *Razor circuit Behavior* 16
3.3 *Block Design before Razor implementation* 17
3.4 *Modified Block Design* 18
3.5 *Two Layers Top* 19
3.6 *Internal Architecture of the two-layers NN* 20
3.7 *Convolution Block 0* 21
3.8 *Razor Architecture for Error Detection* 22
3.9 *Modified Design Flow* 23
3.10 *Razor Architecture* 27
3.11 *Implemented Razor Architecture* 28
3.12 *Driving control signal* 30
3.13 *Clock Gating Implementation* 31

4.1 *Clock Domain Crossing* 34
4.2 *Asynch FIFO Writing and Reading Operation* 35
4.3 *Modified blocks to change clock domains* 36
4.4 *RTL simulation with clk2 at 200 MHz* 38
4.5 *Expected Result for FPS* 41
4.6 *Expected Result for the accuracy* 42
4.7 *Neural Network Troughput* 44
4.8 *Neural Network Accuracy* 45

Chapter 1

Introduction

1.1 Overview

Advancements in silicon technology and increasing clock frequencies necessitate optimizing power and achieving high performance in Embedded Systems and Systems on Chip (SoC). Field-Programmable Gate Arrays (FPGAs), while offering flexibility and lower design costs, are less efficient in performance and energy. FPGA systems typically utilize static timing analysis (STA) to incorporate safety margins for operating and manufacturing during the design stage. FPGA designers also assume that circuit behavior is completely deterministic and highly reliable. As a result, the tools used for FPGA CAD are based on conservative timing models to ensure safe operation under all manufacturing, configuration, and operating conditions. However, operating beyond this conservative margin admitting errors within designs can significantly improve energy and performance. With the flexibility of the FPGA structure, reliability errors can be monitored and controlled with run-time instrumentation and adaptive techniques. [2]

FPGAs are made using Complementary Metal-Oxide-Semiconductor (CMOS) transistors, designed to operate under worst-case conditions to ensure the target performance is met. This means that when the operating conditions are less severe than the worst-case conditions, excess power is consumed. Two main categories of variation cause problems for CMOS circuits: static and dynamic. Static variations, mainly caused by the manufacturing process, affect the worst-case path of the circuit and remain constant over time. On the other hand, dynamic variations are caused by temperature, voltage, aging, and change over time. In-situ methods are required to prevent performance degradation caused by dynamic variations. [3] Micro-architectural techniques such as parallelism, pipelining, and clock gating are commonly used to improve the performance of CMOS circuits. Designers can also use circuit techniques such as low voltage operation, standby current reduction, and optimal gate sizing. However, most of these optimization techniques are static and must be applied during design. Thus, adaptive techniques can be employed to

achieve higher performance at run-time. Dynamic power management techniques have emerged for run-time monitoring of performance variability in the silicon. These techniques continuously adjust power consumption during the system's run-time by characterizing the voltage and frequency of the circuit. The most commonly used methods are Dynamic Power Switching (DPS), Dynamic Voltage and Frequency Scaling (DVS and DVFS) and Adaptive Voltage Scaling (AVS). [4] DVFS is a technique that uses pre-evaluated voltage and frequency operational points to scale power, energy, and performance. On the other hand, AVS is a run-time technique that monitors supply voltage, temperature, and other parameters changing on silicon and adjusts the frequency and voltage accordingly. [5] Another technique for finding the best operating point based on voltage and frequency changes is using Tunable Replica Circuits (TRCs). These circuits are designed to mimic the critical path, which means that the main and the replica paths are subject to the same process variations and aging effects. They are utilized to evaluate the extent to which the frequency can be increased or the voltage can be decreased based on the results derived from the critical path analysis. Nevertheless, the abovementioned methods allow us to find the optimal operating point when the design is error-free. Alternatively, another approach should be considered to eliminate the imposed operating margins. This approach finds possible faults in the design but controls their impact by integrating error detection with error recovery. Razor is an approach that pairs each flip-flop within the data path with a shadow flip-flop controlled by a delayed clock. The output of both blocks is XOR'd together after the data propagates through the shadow flip-flop. If the combinational logic meets the flip-flop setup time, the correct data is latched in both the data path flip-flop and the shadow flip-flop, and no error signal is set. If different values appear in the flip-flop and shadow flip-flop, it indicates an error in the result. When an error is detected, a logic-high signal is broadcast to an error recovery circuit. [3]

In the upcoming chapters, a detailed analysis of the implementation of the Razor methodology on FPGAs is presented. The aim is to provide a comprehensive understanding of the benefits and drawbacks associated with this approach with FPGAs. Furthermore, an overview of the key characteristics of the FPGAs is provided, along with an in-depth breakdown of the design employed.

1.2 Field Programmable Gate Arrays (FPGAs)

An FPGA, short for Field-Programmable Gate Array, is a highly versatile integrated circuit (IC) based on a matrix of *Logic Elements* (LE) or *Configurable Logic Block* (CLB) connected via programmable interconnects. FPGAs are reprogrammable semiconductor devices that offer flexibility, adaptability, and quick time-to-market at lower development costs than Application Specific Integrated Circuits (ASICs). Despite their power consumption and space occupancy, they

have various applications in various sectors, including automotive, consumer electronics, high-performance computing, industrial applications, medical equipment, security, video and image processing, and wireless communications. AMD Xilinx and Intel Altera are the two primary FPGA manufacturers and vendors, and both offer software (Vivado and Quartus) that supports their products.

FPGAs are an evolution of CPLDs with a more significant number of flip-flops available. They use small memories called Look-up Tables (LUT) to implement combinational logic and form Configurable Logic Blocks (CLBs). CLBs are then connected through programmable routing interconnections, forming a matrix of CLBs that can be configured to implement a wide range of digital circuits.

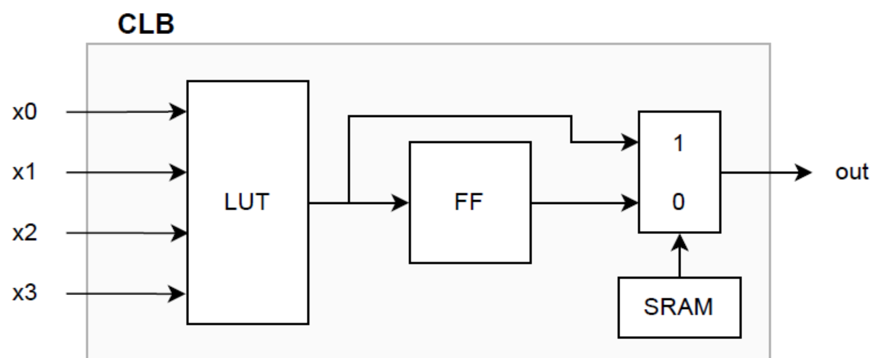

Figure 1.1: *Schematic diagram of a simple CLB*

FPGAs have SRAM-based logic blocks and come in various sizes with different amounts of programmable logic resources. The bigger the FPGA, the more resources it has, allowing for increased acceleration by implementing more parallel circuits. These devices enable designers to balance cost and performance to meet their needs.

Due to the limited inputs and outputs in Configurable Logic Blocks (CLBs), they require significant interconnection resources. Wires are organized into buses for various roles, such as local connections within CLBs, communication between blocks, and long-distance connections, as shown in Figure 1.2.

These interconnections are often established using a Switch Matrix. A Switch Matrix is a network of programmable switches that can be dynamically configured for different interconnections. It uses pass transistors driven by small 1-bit memories, where a value of 1 corresponds to a closed circuit, and 0 corresponds to an open circuit. This architecture is compact and efficient.

Finally, the FPGA's periphery houses input/output elements that can be configured as input, output, or bidirectional pins. These elements contain large buffers

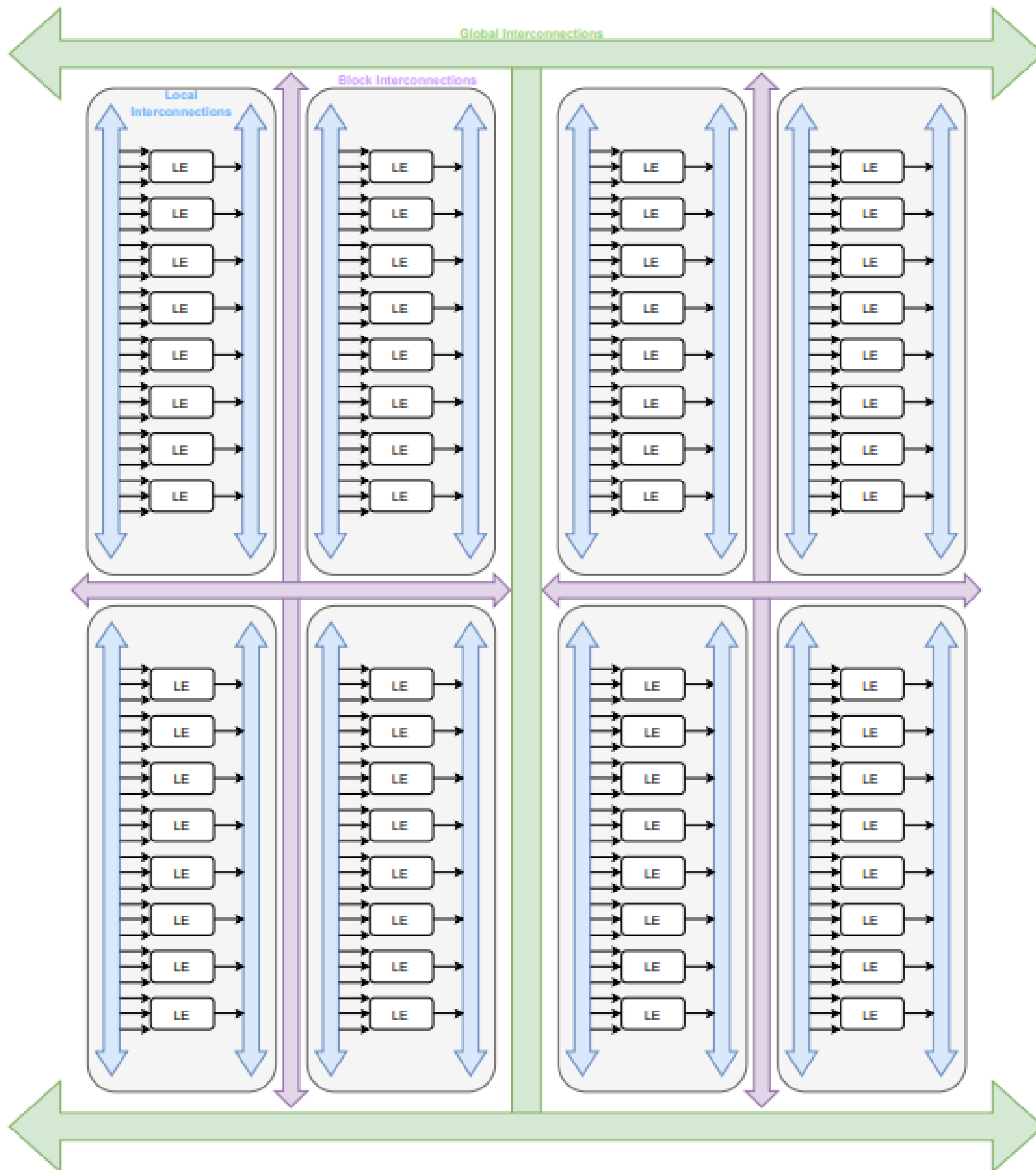

Figure 1.2: *Block Diagram of an FPGA*

with high current, designed to interface with pads with significant parasitic capacitance and external components.

Summing up, an FPGA has three central elements: logic blocks (CLBs), interconnections wires and switches, and I/O blocks, all programmable.

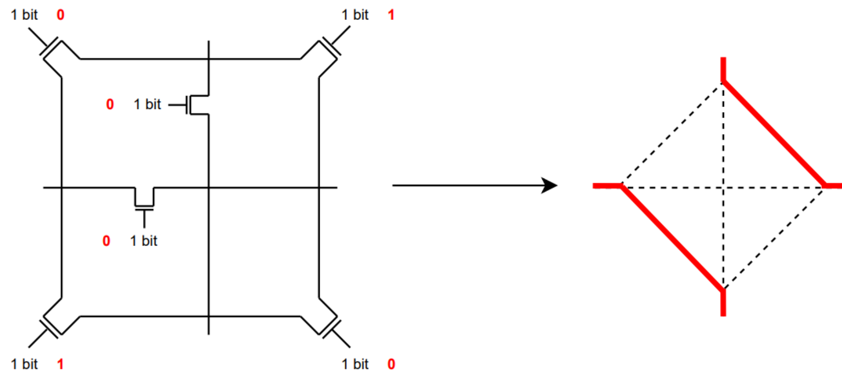

Figure 1.3: *An interconnection switch matrix of an FPGA*

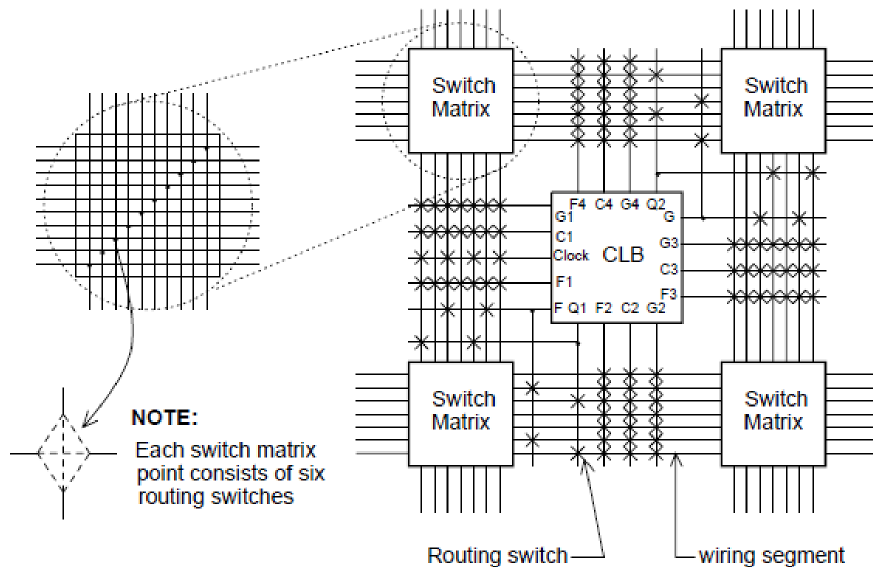

Figure 1.4: *Programmable interconnect and switch matrix overview [1]*

1.2.1 Design Flow

A typical design flow is shown in Figure 1.5. The first step in designing a digital circuit is to create a high-level representation. This is achieved using the Register Transfer Level (RTL) design abstraction, which models the circuit by indicating the flow of digital signals between hardware registers and the logical operations performed on those signals. To specify the design's characteristics, designers use a Hardware Description Language (HDL) like VHDL or Verilog. RTL source files can be used to describe a design or a Block Design that connects various Intellectual

Properties (IPs) via specialized interconnect protocols that can be created. IPs play a crucial role in today's FPGA and embedded system industry, allowing designers to choose from various pre-developed design blocks. A testbench can be used to test the initial high-level specification at a behavioral level to verify the algorithm's accuracy. This is done without considering timing at this stage because there is no information on the final implementation. The second step is the synthesis that converts high-level logic designs into gates. This process employs timing constraint files (XDC files) as input and provides a post-synthesis netlist containing the circuit's gate-level description as output. It also generates a set of timing delays that simulate the propagation of signals through the FPGA. The next step in the process is technology-dependent. First, the netlist generated earlier is optimized to eliminate any redundant or duplicated elements. Then, the netlist is mapped to the physical resources of the FPGA, which is done using the pinout description provided as input in the form of an XDC file. After optimization, the Place and Route step is accomplished. The optimized netlist is placed to the physical cells in the FPGA, and the interconnections between these cells are selected accordingly. At the end of the design process, a bitstream file (.bit) is produced as the output. This file describes the hardware logic, routing, and initial values for registers and on-chip memory. The bitstream file is used to program the board.

1.2.2 Operating Margins on FPGA

The performance of an FPGA circuit is subject to various external factors, such as environmental conditions (e.g., temperature and noise), manufacturing discrepancies, aging effects, and input data characteristics. These factors can significantly affect the circuit's stability and performance. Moreover, components on the same die or between different dice can exhibit varying timing and energy characteristics, which may change over time due to degradation. Stochastic effects like thermal noise, crosstalk, power supply ripple, and clock jitter can also cause timing variations. Additionally, the arrival time of data in a register depends on the transitions across all input nodes, and the critical path, which is the slowest, is typically considered when calculating delays. However, it may not be frequently utilized, and hence, the average delay time may be faster. The combination of various techniques, such as DVFS, AVS, and TRCs discussed earlier, can reduce the margins induced by these physical effects. Through the implementation of the Razor method, shadow flip-flops can be integrated into the design to identify any potential errors. In particular, timing-related errors can be addressed by incorporating a shadow register for a datapath register, with a clock phase shift that is appropriately adjusted. This enables a reliable comparison of the captured stabilized values with those of the datapath register. [6]

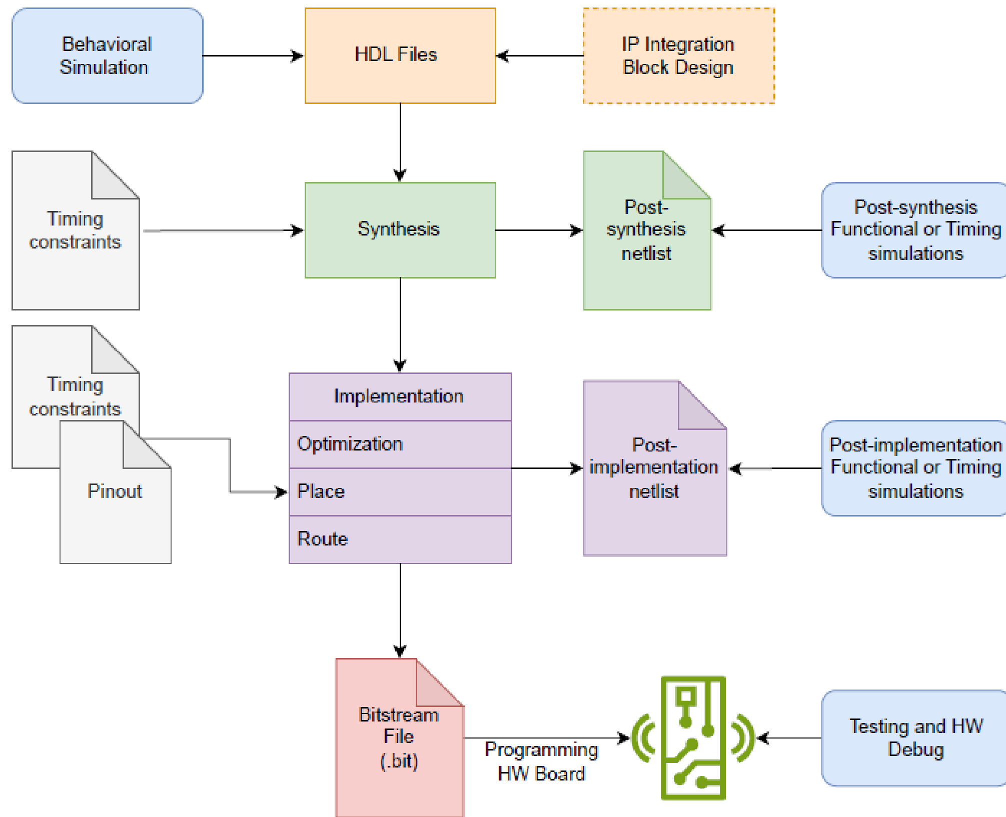

Figure 1.5: *Design Flow, from RTL to bitstream*

1.2.3 Error Detection on FPGA

When designing circuits using FPGAs, it can be beneficial to include error detection to improve circuit reliability and performance. However, it is essential to consider the added overhead of area, timing, and power that comes with operating beyond normal limits. To ensure success, the error detection circuit should be designed using the resources already available in the FPGA logic fabric. The designer must thoughtfully choose which paths to monitor and implement the error detection logic while preserving the timing of the original design. A strategic approach is necessary to ensure the error detection circuit does not interfere with the original circuit's functionality and provides reliable results. In real-world FPGA designs, there may be unused logic routing and clock resources. A Razor circuit can be incorporated into the circuit post-placement to monitor critical paths after compilation and make the most of these spare resources. Additionally, FPGAs offer a clock tree network that allows for adjustable frequencies and tunable phases, providing the flexibility to use different clocks and adjust their phases.

1.3 Razor Methodology

The Razor technique is a timing speculation technique that works by double-sampling the data at the setup endpoints of critical paths. This means that the original flip-flop of the datapath samples the input data first. Then, the same data is sampled again by another flip-flop called the shadow flip-flop. The shadow flip-flop has the same input data but uses a clock that is phase-shifted appropriately. This helps to detect errors due to any delay in the arrival of data. The main flip-flop speculates that the data has already completed the propagation along the preceding combinational logic, while the shadow flip-flop samples the data later, ensuring that the correct data has reached the end of the logic path. Actually, the Razor circuit consists of two flip-flops: a main flip-flop and a shadow flip-flop, controlled by a clock with a specific offset. The output data of both flip-flops is compared using an XOR gate, as shown in Figure 1.6. The main flip-flop may miss the input data if the combinational logic exceeds the data arrival delay. In such a case, the shadow flip-flop, which operates on a delayed clock, captures the data after a brief delay. To ensure the shadow flip-flop always captures the input data accurately, it is crucial to maintain the logic delay below the setup time of the shadow flip-flop.

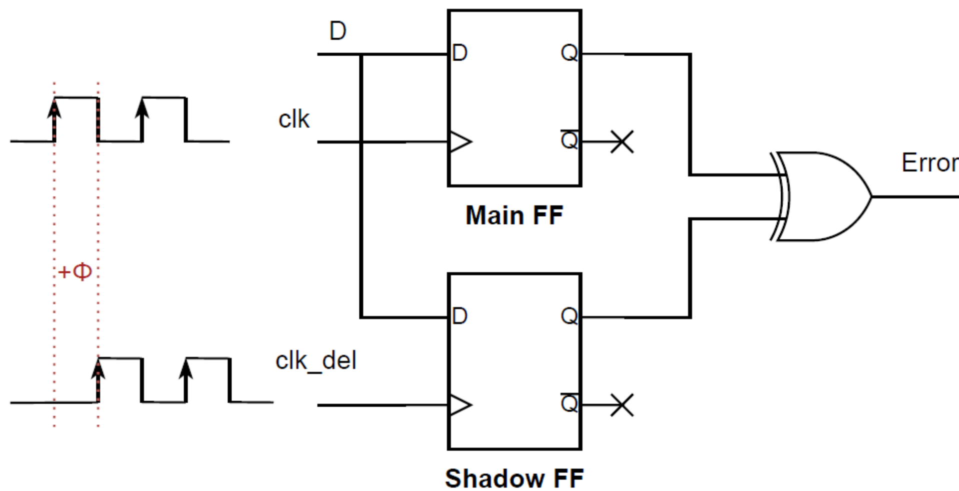

Figure 1.6: *Razor circuit*

Figure 1.7 analyzes the Razor circuit's timing behavior. During clock cycle 1, the input data is correctly sampled by the rising edge of the main clock within the setup time. Additionally, the shadow FF successfully samples the first instruction, and the error signal, which is the result of the XOR gate, remains low. During the second clock cycle, the combinational logic exceeds the intended delay, causing the main clock to sample incorrectly. However, since a delayed clock drives the shadow FF, it can sample the second instruction in cycle 3. An error is generated by comparing the valid data at the shadow FF's output with the main FF's output data. In clock cycle 4, the data is restored in the main FF and the error signal goes low again.

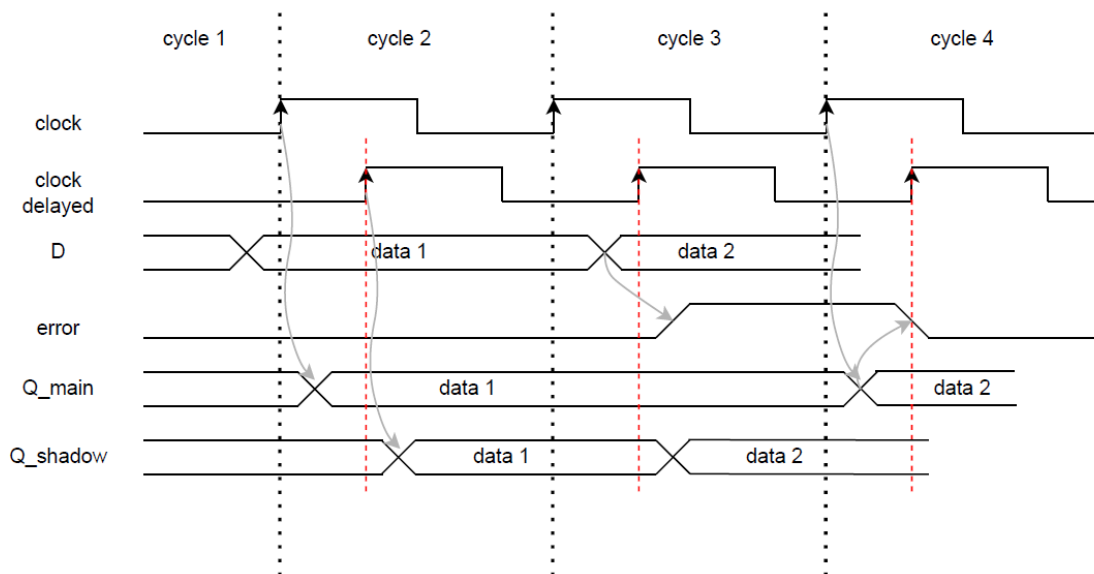

Figure 1.7: *Behavior of the Razor circuit*

This technique enables an increase in working frequency or a reduction in supply voltage until errors occur. If the frequency increase's benefits outweigh the decrease in performance due to error recovery, it may be acceptable to tolerate some errors. However, it is important to weigh the performance benefits against the potential overhead created by errors. This method eliminates the need for safe operating margins and can detect fast change errors in critical paths, making it more advantageous than adaptive techniques. However, some issues related to the circuit should be taken into account. Increasing frequency can cause metastability issues on flip-flops, while the clock delay on the shadow flip-flop can lead to timing violations. Using a delayed clock at the shadow flip-flop raises the possibility that a short path in the combinational logic will corrupt the data. Figure 1.8 demonstrates

how a short path allows data launched at the start of a cycle to be latched into the shadow flip-flop instead of the data launched from the previous cycle. The clock provided to the shadow flip-flop must be sufficiently delayed to ensure accurate data sampling at a higher working frequency. As a result, if the aim is to increase the frequency, the clock has to be further delayed to the flip-flop. Nonetheless, increasing the clock delay also means that data traveling along short paths that end in shadow flip-flops must not reach the flip-flop before the previously launched data has been correctly sampled in the previous clock cycle. [7]

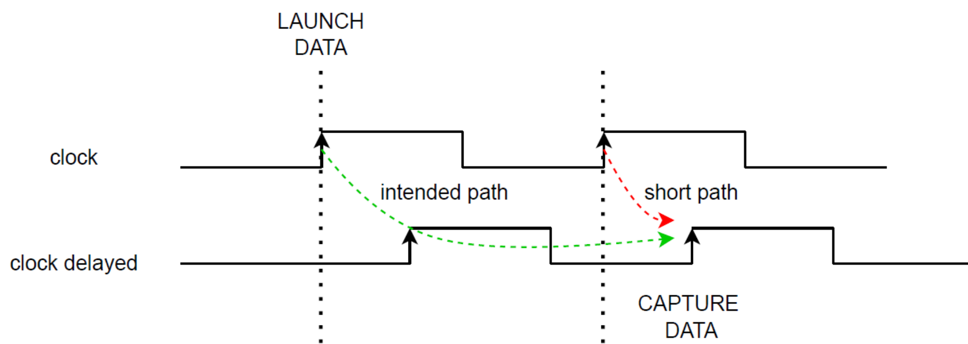

Figure 1.8: *Short Paths Constraints*

To prevent short paths, timing constraints can be added to the design. One such constraint is the Multicycle Paths constraint, which can be used to modify the setup and hold relationships. It delays the capturing edge by one clock cycle. The intended behavior is as follows: the first positive edge of the main clock launches the input data. One clock cycle later, this data is sampled and available on the flip-flop's output. After this, the data is captured on the second positive edge of the delayed clock.

1.4 Deep Neural Networks Framework

Deep Neural Networks (DNNs) are complex learning models that have proven incredibly useful in various fields. FPGAs, which offer a programmable platform for acceleration, are becoming an increasingly popular choice for speeding up DNNs. However, optimizing performance and energy efficiency with FPGAs is no easy feat.

In this thesis, Razor is integrated into a two-layer Convolutional Neural Network (CNN) for FPGAs. The benchmark is generated through an automated implementation flow utilizing a specific framework. The neural network is implemented using this framework, which produces quantized convolution neural network accelerators in C++ for AMD FPGAs.

1.4.1 Convolutional Neural Networks (CNNs)

Convolutional neural networks (CNNs) have consistently achieved state-of-the-art results in many tasks, including computer vision and speech recognition. [8] CNNs are similar to traditional Artificial Neural Networks (ANNs), consisting of self-optimizing neurons that learn through experience. Neural Networks are computational processing systems heavily inspired by how biological nervous systems (such as the human brain) operate. They are mainly comprised of many interconnected computational nodes (referred to as neurons), which work together in a distributed fashion to collectively learn from the input to optimize its final output. The main difference between CNNs and traditional ANNs is that CNNs are primarily used for image pattern recognition. [9] The choice of hardware can significantly influence the effectiveness of convolutional layers. While versatile, CPUs are not as efficient as GPUs, which are specifically designed to handle parallelism. On the other hand, ASICs and FPGAs offer varying tradeoffs between cost and flexibility. The area of FPGA-based acceleration for deep neural networks (DNNs) has received significant attention due to its potential to achieve high-performance and energy-efficient inference. In this field, tools such as Xilinx Vitis AI [10] and FINN [11, 12] are widely acknowledged as leading frameworks for deploying DNNs on FPGAs. However, a custom dataflow is used in the thesis design to improve resource management and achieve a better tradeoff between accuracy, throughput, and memory usage. [8]

1.4.2 CNN Framework

The framework utilized produces quantized convolution neural network accelerators in C++ for AMD FPGAs. To transform a quantized model into a hardware design, the specific flow shown in Figure 1.9 is followed. The first step is to generate C++ code from the quantized model, which will then be utilized to create RTL code using Vitis HLS. RTL code is a detailed description of the hardware design in a low-level language that can be synthesized into a hardware circuit. Once the RTL code is obtained, it can be imported as a block in the Vivado design tool, which is used to design, simulate, and synthesize digital circuits. The design flow can then be continued until the bitstream file is produced. This file includes the configuration data necessary to program the FPGA. First, Brevitas [13], a PyTorch library, is used for NN quantization and to extract the graph in QONNX format. PyTorch [14] is an end-to-end machine learning framework that supports the standard Open Neural Network Exchange (ONNX) format. The Quantized ONNX (QONNX) format defines the network by layer type, input/output quantization, and layer connections. From the network description, the C++ code of the top function that instantiates all network layers is generated. Then, the C++ code is synthesized into RTL code using AMD's High-level synthesis tool Vitis HLS. Once the RTL code is ready, it is imported into an IP block in the Vivado design and integrated into the Block Design. Finally, the bitstream file is generated. The framework used is working on a small dataset called CIFAR10 and the two supported boards are ULTRA96v2 and KRIA KV260.

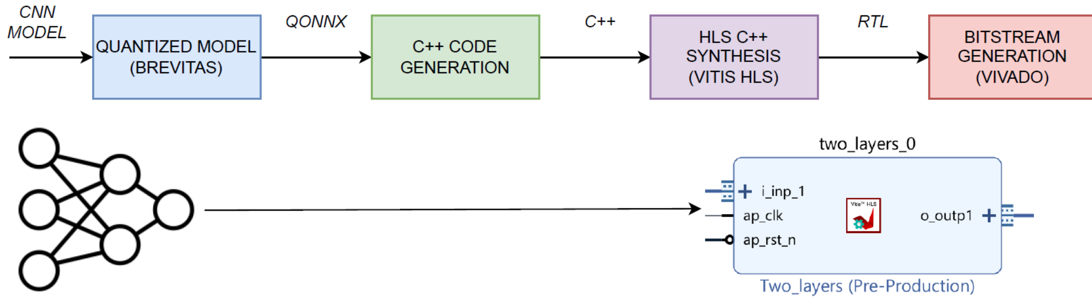

Figure 1.9: *CNN Implementation Flow*

1.5 Thesis structure

The dissertation of this thesis is divided into the following chapters:

- Chapter 2: Related Work
- Chapter 3: Implementation
- Chapter 4: Frequency Dynamic Reconfiguration
- Chapter 5: Conclusions and Future Work

Chapter 2

Related Work

The Razor approach has been extensively researched and documented in academic papers [7, 15, 16]. Error detection circuitry has been integrated into various circuit designs due to its effectiveness in managing errors while maintaining performance standards. Researchers have conducted studies to assess the practicality and usefulness of this technology in different applications. For example, one study evaluated the effectiveness of timing error detection on a single-precision multiplier benchmark [6]. Another study explored the application of timing fault detection to the Rocket Core, an out-of-order scalar processor [17]. These studies provide valuable insights into the practicality and applicability of this technology in real-world scenarios.

The key feature of this method relies on re-executing an instruction through the following stage if it fails at a certain point in the pipeline, incurring a one-cycle penalty. This ensures that the instruction can continue to move forward, preventing it from getting stuck indefinitely at a single stage. The data must be invalidated if an error occurs in a pipeline stage. The preceding pipeline stages will also need to be paused to prevent further errors. Additionally, the shadow latch data should be restored into the main flip-flops. Different methods can be used to recover from errors, such as clock gating and propagating the error signal through the pipeline.

The thesis employs a modified version of the method presented in [7], which involves using global clock gating to recover errors. The basic idea is that when an error is detected in any pipeline stage, the pipeline is stalled for one cycle by gating the next global clock edge. This extra clock cycle allows each stage to recompute its result using the Razor shadow latch as input. Consequently, any previously forwarded erroneous values are replaced with the correct value from the Razor shadow latch. Another valid approach is presented in [18]. Upon detecting an error, the system propagates the error signal to the neighboring blocks, temporarily halting the data pipeline. This pause lasts for a single clock cycle, during which the data is restored and retrieved in the main flip-flops.

Chapter 3

Implementation

The Razor circuit implementation consists of two steps: error detection circuitry insertion and error correction method management. The first step involves inserting the Razor circuit onto the most crucial paths of the design via a TCL script. The design flow is further refined in a second step to enable error correction.

3.1 Error Detection

Two different methods for implementing the Razor circuit have been evaluated. The first approach involves adding a second clock to the design, offset by half a period (90°) from the main system clock. This second clock drives the shadow flip-flops to sample data with a delay, as shown earlier in Figures 1.6 and 1.7. In the second approach, as illustrated in Figure 3.1, the shadow flip-flops are controlled by a clock with the same frequency as the system clock but data are captured on the falling edge. This alternative implementation has been decided as the definitive solution to address the synchronization and phase issues between the two clocks and facilitate timing closure. In this scenario, the datapath flip-flop assumes when the incoming data will arrive and captures it at the rising edge of the clock. Meanwhile, the shadow flip-flop allows the data to continue flowing for another half of the clock period before capturing it on the falling edge of the clock, as illustrated in Figure 3.2. Having the same clock period and phase offers several benefits, including reducing setup and hold timing violations and improved timing closure.

In both cases, the Clocking Wizard IP is used to modify the design's clock tree. It allows the addition and configuration of clocks, adjustment of their period and phase, and managing various parameters. By providing an interface for dynamic reconfiguration of the clock, the clocking wizard enables dynamic frequency variation of the design without the need to regenerate the bitstream file every time.

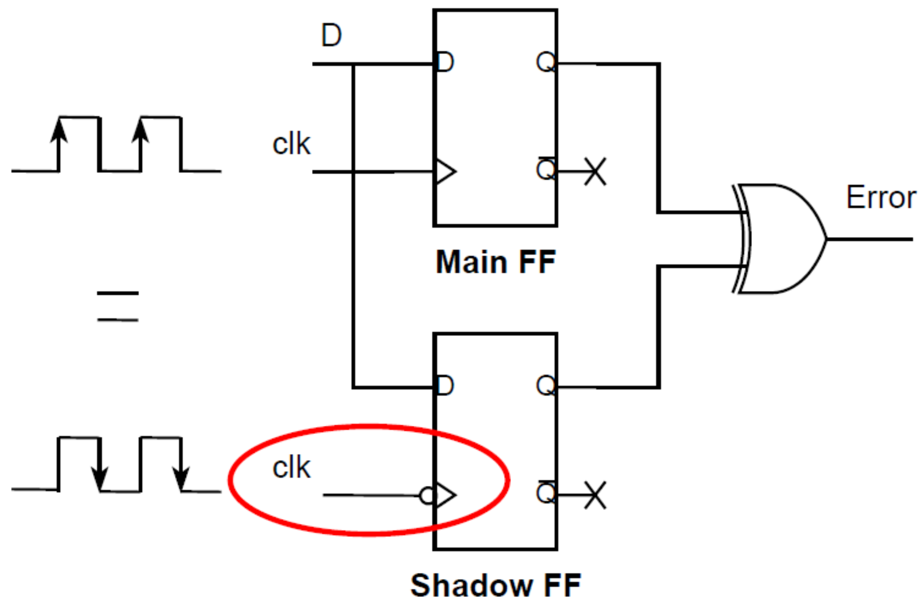

Figure 3.1: *Razor circuit with negated clock*

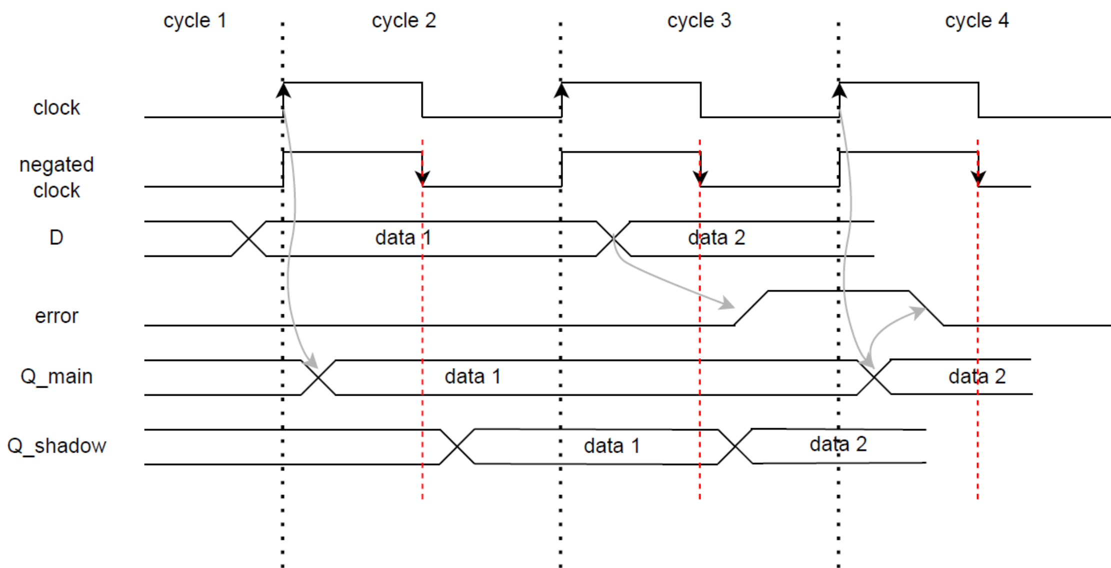

Figure 3.2: *Razor circuit Behavior*

3.1.1 Block Design

Following the Implementation Flow, as presented in Figure 1.9, the initial Block Design is generated. The design is then modified by adding a secondary clock. The system design is implemented using Vivado, a design software for AMD SoCs and FPGAs. Vivado enables the visualization of the entire design in a graphical representation. A Block Design is a graphical representation of a digital system created using Vivado's IP Integrator tool. It allows designers to construct complex digital systems visually by integrating various intellectual property (IP) cores, including processors, interfaces, memory controllers, and custom logic blocks. As shown in the implementation flow outlined in the Introduction, the RTL code generated using Vitis HLS is imported as an IP into the Vivado Block Design reported in Figure 3.3.

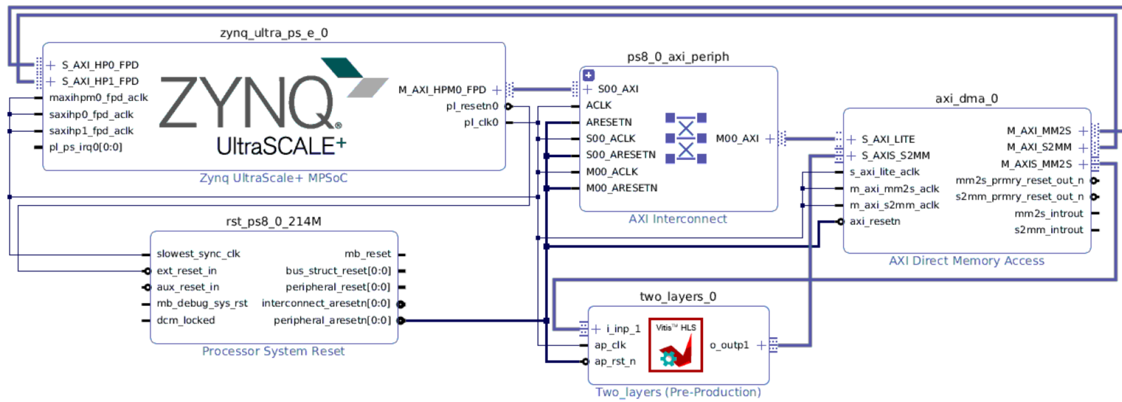

Figure 3.3: *Block Design before Razor implementation*

The IP name corresponding to the two-layer CNN is *two_layers_0*. It includes both input and output streams and is driven by the system clock *ap_clk* and features a reset *ap_rst_n*. To incorporate the CNN IP into the design, the processing system infrastructure and memory must be managed. To achieve this, the Zynq UltraScale+ MPSoC Processing System IP from AMD is used. The Zynq UltraScale+ Processing System core is based on the AMD system-on-chip architecture and is a logical connection between the Processing System (PS) and the Programmable Logic (PL). In addition, the AXI Direct Memory Access (AXI DMA) IP is employed to manage the input and output streams of the neural network with high-bandwidth direct memory access. Finally, the AXI Interconnect IP connects one or more AXI memory-mapped Master devices to one or more memory-mapped Slave devices and the Processing System Reset IP provides a mechanism to handle the reset conditions for a given system.

Clock Tree Network

The IP Clocking Wizard is a tool that creates HDL source code to set up a clock circuit, including parameters like period and phase. The wizard provides two clocking primitives, MMCM and PLL, and adjusts buffering, feedback, and timing parameters to configure the desired clocking network. As depicted in Figure 3.4, the clock wizard takes the system clock created by the processing system as input and produces two output clocks that can be dynamically reconfigured in hardware once the bitstream is generated. This functionality is essential to assess the feasibility of the Razor circuit with varying frequencies and determine whether the tradeoff between errors and performance gain is worthwhile.

The Clocking Wizard IP generates two output clocks. The first clock, named *clk1*, is the primary driver for the entire design instead of relying on the Zynq system clock. This approach eliminates the need to modify the IP and re-run the whole design flow each time the system clock frequency needs to be changed. Moreover, the clock generated by the Clocking Wizard can be dynamically reconfigured after the bitstream is downloaded onto the hardware board by manipulating specific registers. The second clock is necessary for the subsequent implementation step, which involves error detection with clock gating. This clock drives the convolution block in which the Razor cells are inserted and is gated using a clock buffer primitive. This clocking scheme provides a way to test the design by varying the entire design's frequency and evaluating the Razor technique's effectiveness in improving performance. An additional IP, the AXI4-Stream Clock Converter, keeps the two clocks asynchronous to avoid any issues with the phase relationship. This IP also transmits the CNN IP's input and output streams to the DMA using the AXI protocol. Furthermore, each clock has its reset IP. Figure 3.4 shows the new Block Design with the additional clock.

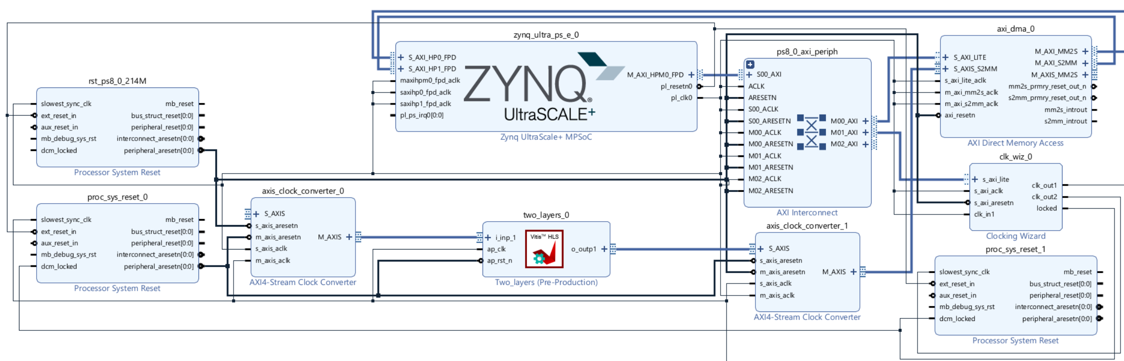

Figure 3.4: *Modified Block Design*

Neural Network Architecture

The CNN IP has a comprehensive two-layer architecture that uses input and output data streams. Direct Memory Access (DMA) blocks manage the input and output tensors, which read and write data streams. The frames are processed sequentially and continuously, ensuring each task's smooth functioning. As soon as the input data is available, the tasks start operating. The inference begins when the DMA is connected to the input port of the top-level interface and downloads the input images. The tasks are pipelined, where the first stage reads the input stream and the rest of the stages process data and write output streams.

Figure 3.5 illustrates the architecture overview. The network comprises two independent convolution blocks, with the first block being the bottleneck of the whole system. Therefore, the thesis focuses on implementing the Razor circuit in the first convolution block. This approach improves the processing efficiency of the first convolution block, enhancing the overall performance of the CNN IP.

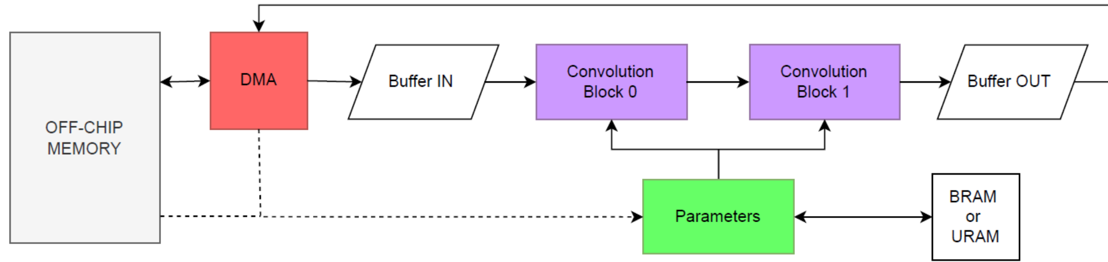

Figure 3.5: *Two Layers Top*

Convolution blocks are a crucial part of the computation pipeline in the QONNX network graph. These blocks take a window of input activations from input buffers, and their dimensions are based on the input and output channel sizes. The convolution blocks, also known as convolution computation blocks, are the most computationally intensive among the blocks. The slowest convolution block determines the overall accelerator throughput. Each convolution block performs multiply and accumulation (MAC) operations the DSP units execute. The accelerator performs MAC operations on simultaneously loaded input activations. After all input channels are processed, partial results move through the pipeline until the output data is written. For each convolution layer, parameters are required to feed the computation pipeline with data from the on-chip memory. The storage location of these parameters depends on the targeted FPGA. For instance, the Ultra96 board uses BRAM as it does not have URAM, whereas the KRIA KV260 board uses URAM. BRAMs have a storage capacity of 4KB and are initialized directly in the

bitstream. In contrast, URAMs have more storage capacity (32KB) but require dedicated hardware for initialization. When using URAMs, the DMA transfers the parameters as a stream during power-up. Figure 3.6 illustrates the internal architecture of the two-layer neural network IP. The diagram comprehensively illustrates the individual components of the design.

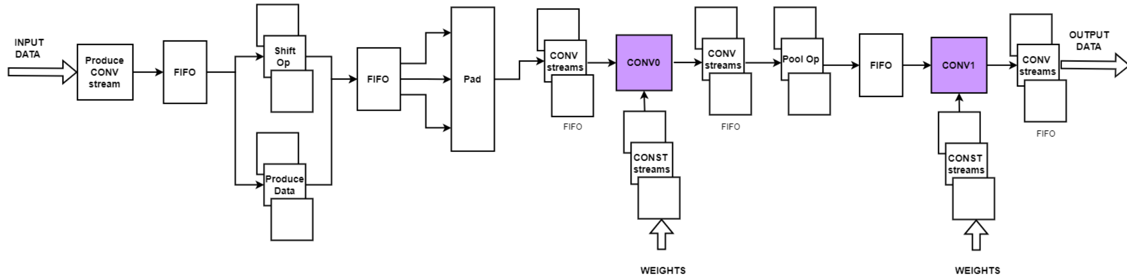

Figure 3.6: *Internal Architecture of the two-layers NN*

The schematic above highlights two convolution blocks. The first block, *conv0*, has been identified as the slowest task of the design and is, therefore, the focus of interest for testing the Razor technique. It has been observed that this convolution block is linked to the most critical paths. Consequently, to simplify the thesis evaluation, the design section being investigated is limited to the *conv0* block, shown in Figure 3.7.

3.1.2 Post-synthesis Netlist Modification

Once the Block Design has been suitably modified, it is synthesized, resulting in a post-synthesis netlist. This netlist enables the execution of a timing analysis that provides the most critical paths of the design, along with their respective slacks. The Razor technique requires selecting a percentage of these critical paths, allowing errors only in the most problematic paths of the circuit, which may increase performance in the entire design. It is worth noting that only the paths of the first convolution block, which is slowing down the entire system, are considered. To achieve this, the TCL script used to generate the complete project and block design is modified, adding the algorithm to include the Razor cells. After adding the Razor flip-flops and introducing the new dynamically reconfigurable clocks, a section for error correction is also incorporated (refer to paragraph 3.2).

Razor Architecture

The Razor cells are integrated into critical endpoints in the netlist to enable error detection (see Figure 3.8). These specialized cells comprise a shadow FF that

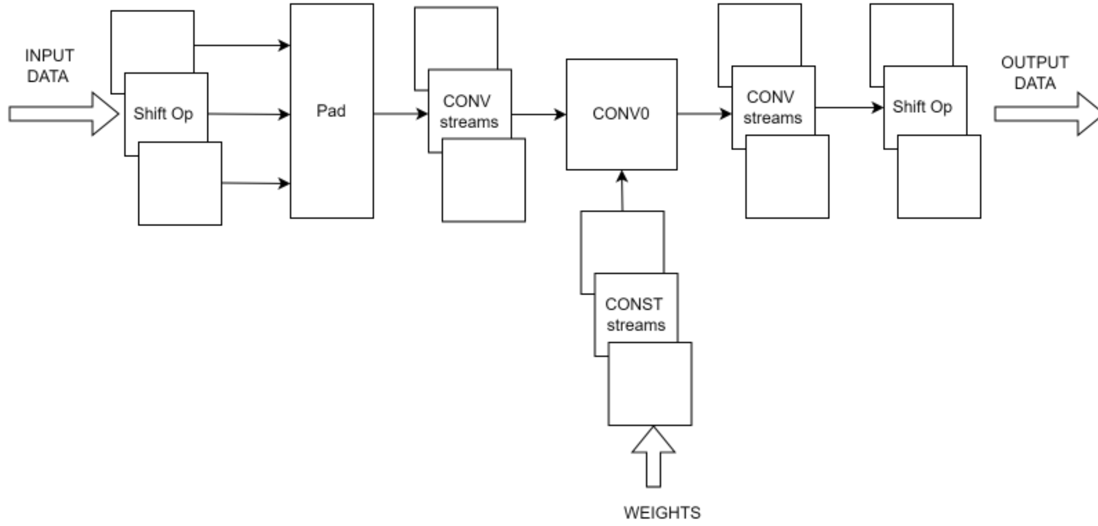

Figure 3.7: *Convolution Block 0*

shares the input net with the datapath FF. The output net of the shadow FF is then compared with that of the primary FF using an XOR gate. In the event that multiple shadow FFs are added, the error signals of all Razor cells are merged using a tree of OR gates. This allows for the regrouping of all errors generated by the Razor cell chain. The final OR output represents the error signal which is then utilized in the two methods analyzed to stall the pipeline and recover the error.

Modified Design Flow

After modifying the Block design presented in the previous section, the RTL code undergoes synthesis, resulting in a post-synthesis netlist. In Vivado, it is possible to modify the synthesized netlist using specific TCL commands integrated into the tool. With these commands, it is possible to create a cell or a primitive, connect or remove nets, and even create an algorithm to modify the synthesized design using the TCL scripting language. TCL commands are used to select critical paths of a design by applying filters such as name, cell function, slack value, etc. These commands are highly flexible and can be used to limit the search for critical paths in specific blocks of the design, such as the first convolution block, which allowed for the inclusion of the Razor circuit for error detection. One of the paragraphs below explains the algorithm for inserting the Razor circuit and the TCL commands used. Once the bitstream is generated, it needs to be downloaded onto the board in order to configure the FPGA fabric with the additional Razor circuitry. An overlay, or hardware library, is used to download the bitstream file onto the board. Overlays are configurable FPGA designs that extend the user application from the Processing

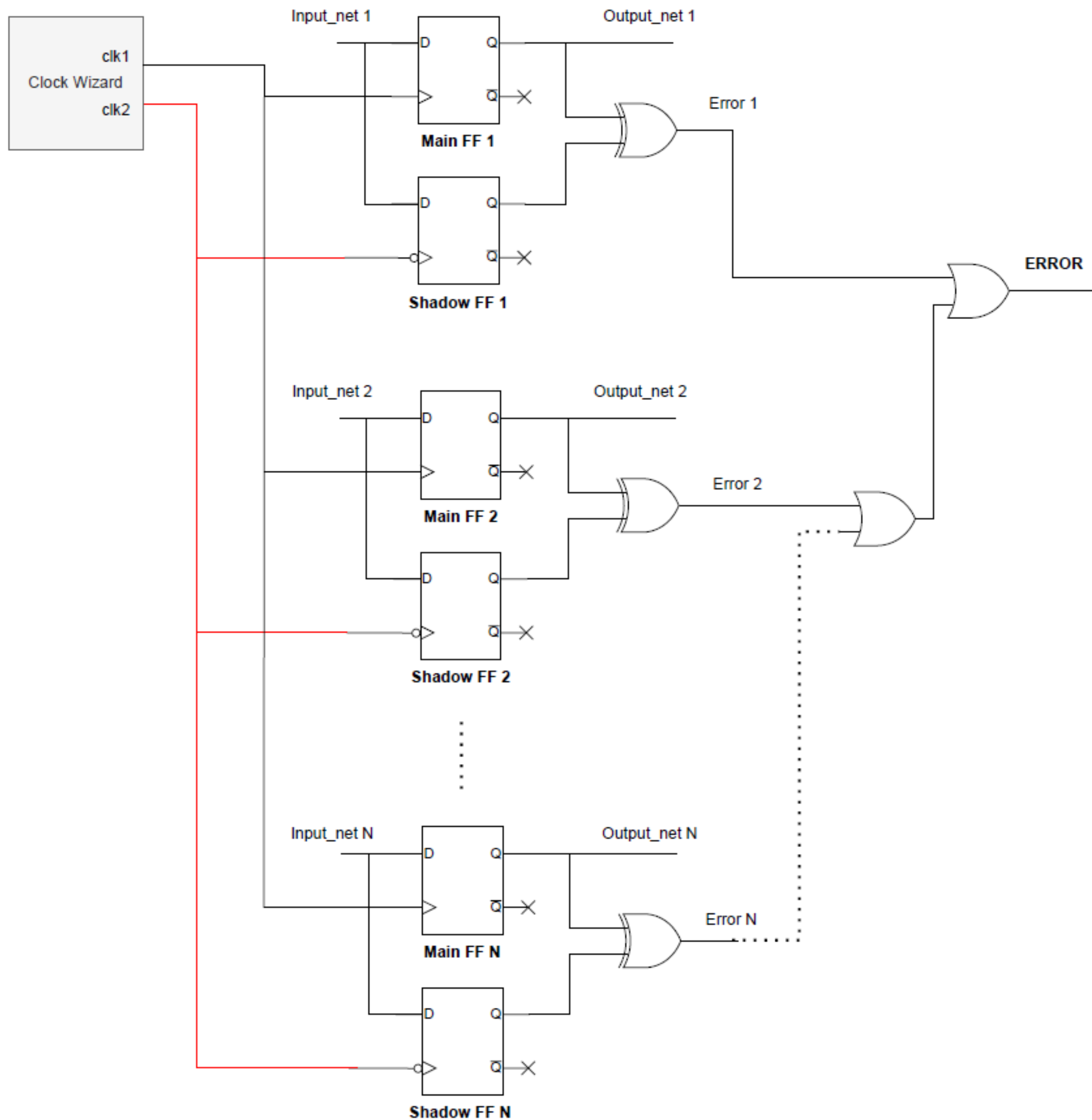

Figure 3.8: *Razor Architecture for Error Detection*

System of the Zynq into the Programmable Logic. These overlays can be loaded to the FPGA dynamically and used as software libraries to program functions and write registers without the need to generate the bitstream every time. To achieve this, an AMD open-source project called PYNQ [19] is used. It provides a Python interface that allows overlays in the PL to be controlled from Python running in the PS. The inference of the design and the Python scripts utilized are proposed in Chapter 4. The design process's final flow has been illustrated in Figure 3.9.

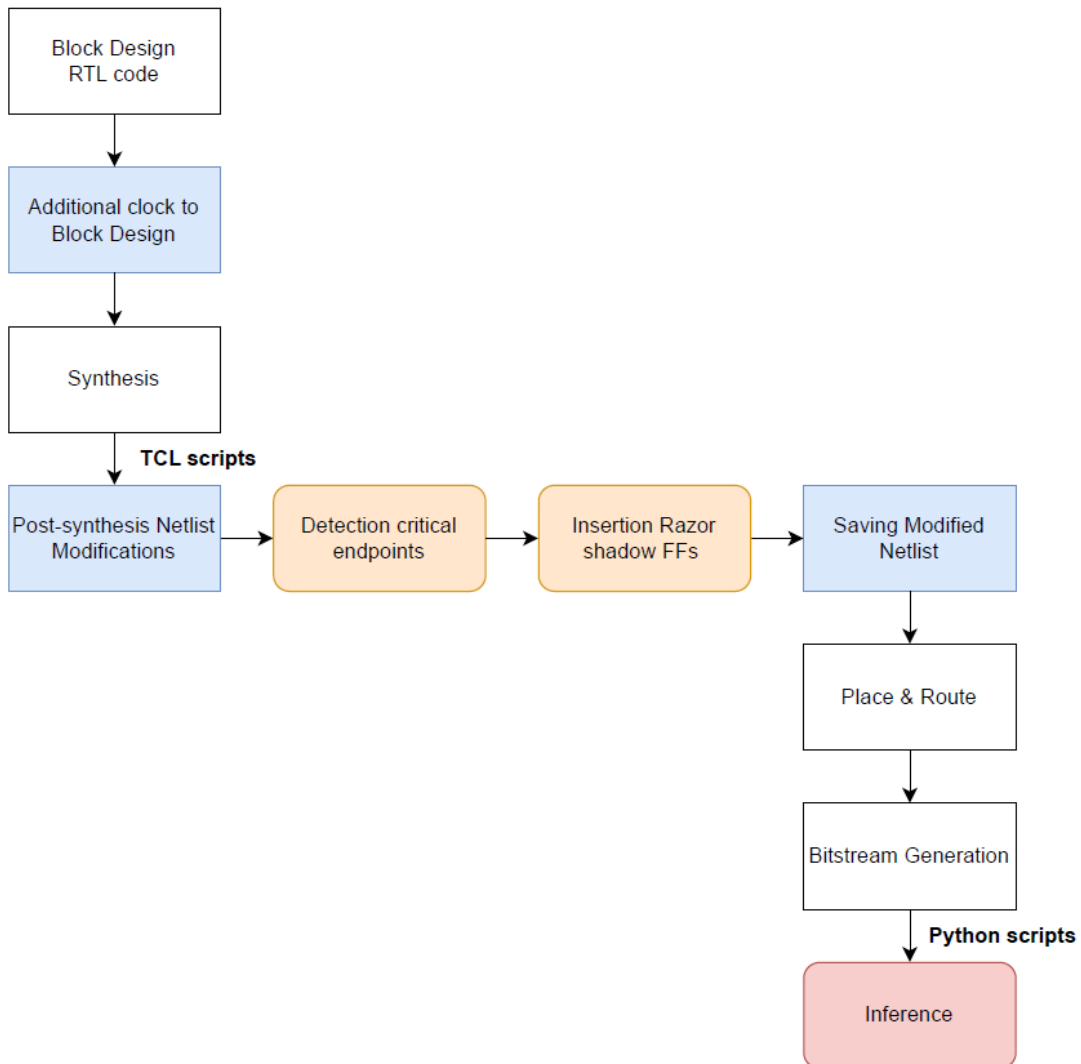

Figure 3.9: *Modified Design Flow*

Algorithm for Inserting Razor

The algorithm for inserting the Razor cells starts by initializing essential parameters such as the clock period and the desired percentage. It detects all the endpoints of combinational paths with a setup slack that is less than the threshold specified by the user. These found flip-flops are more susceptible to capturing a wrong value in case of slow signal propagation in the datapath. Therefore, these flip-flop inputs are double-sampled by inserting a shadow flip-flop to ensure error detection. The targeted block is convolution block 0, which is controlled by a Finite State Machine (FSM). This FSM can start, enable, block, and stop the convolution task through

control signals. The objective is to incorporate the Razor circuit for error detection on the FSM's critical paths. This will allow for the detection and correction of errors in the input stream before any computation takes place. The computation process involves the use of DSPs, multipliers, and accumulators.

The algorithm used to insert Razor cells in the design is outlined in the following pseudo-code. The Razor architecture has two parts. The first part is the Razor cells, which detect errors. The second part involves adding 2-to-1 multiplexers to the logic. These multiplexers enable either the datapath data or the shadow data, depending on the error being used as a selector. The purpose of this second part is to recover from the error by selecting the correct data after the pipeline is blocked. Recovery error methods are presented in the following paragraphs. The TCL script initializes key parameters such as clock characteristics and criteria for identifying specific endpoints in the design. For each path, it creates a 2-to-1 MUX and a shadow flip-flop. The script disconnects the datapath input net of the endpoint's associated cell and reroutes it to the MUX's input. The MUX's output is connected to both the cell's original input and the Shadow FF's input. The Shadow FF is clocked with the negated clock. To detect errors, the script employs an XOR gate to compare the output of the Shadow FF with the main output. If there is any discrepancy between the two, the script generates an error signal, indicative of a fault in the original data path. This error signal is then used to control the MUX's selection input, which enables error correction by switching between redundant and original data paths. In situations where multiple endpoints are identified, the script uses additional logic to manage them collectively. Depending on the number of endpoints, the script incorporates multiple OR gates to aggregate error signals from the XORs of the individual endpoints into a collective error signal. This collective error signal is then used to control the MUX selection inputs, facilitating coordinated error correction across multiple data paths. Once the error detection logic is in place, the next step is to select an appropriate error recovery method from two alternatives. The first approach involves connecting the error signal directly to the relevant control signals within the architecture, while the second involves using clock gating.

The next section presents the complete Razor architecture.

Algorithm 1 Algorithm for the addition of the Razor architecture

```
period ← clk1
percentage ← 30%

        ▷ This example only selects critical paths with setup slack <30% of period

for all Cells in the CONV_0 block except for computation elements do
    if Cell is SEQUENTIAL and SLACK is <30% of period then
        endpoint ← cell_i
    end if
    for all Setup endpoints of critical FFs found do
        Add a 2-to-1 multiplexer
        Disconnect input net of main FF
        Connect input net to the first input of mux
        Connect output mux to the main FF
        Add shadow FF
        Connect output mux to the shadow FF
        Create XOR gate between main FF and shadow FF outputs
        Error_i ← XOR_out
        if Razor cells are more than one then
            Create OR chain to merge all errors in a final one
            ...
            ERROR ← OR_final_out
        end if
        Connect ERROR to the select of mux
        Connect output of the shadow FF to the input 1 of mux
    end for
end for

if Method 1 then
    Management of error signal for pipeline stalling
end if
if Method 2 then
    Management of error with clock gating
end if
```

3.2 Error Recovery Methods

The Razor technique has a dual function of detecting and correcting errors. If an error occurs, the error signal becomes high, and the convolution block needs to be stalled for one clock cycle. This gives time to the shadow FF to sample the correct data that arrived late. The NN architecture consists of multiple logic blocks that can work simultaneously on different sets of data, as shown previously. Two methods are evaluated to stall the first convolution block and retrieve the correct data. The first method involves stopping the task for one clock by forcing the FIFO's reading and writing control signals in both input and output to the convolution block. The second method involves halting the entire convolution block for one clock cycle by using the error signal as an enable signal for the clock buffer, which implements the clock gating technique.

3.2.1 Razor architecture

Figure 3.10 shows the complete Razor architecture, which includes the basic error detection cell used in the previous circuit and the multiplexers' additional logic. Indeed, to recover data when an error occurs, it is necessary to determine whether to sample a new input or maintain the last stored value in order to recover data. The Razor circuit achieves this by enabling a 2-to-1 multiplexer that selects which signal to sample at the next clock edge directly with the error signal. One input is connected to the input data line of the main flip-flop, while the second input is the feedback from the output of the shadow flip-flop. This structure should stall and restart the pipeline after one clock cycle. This allows for error correction by switching between the shadow and original data paths.

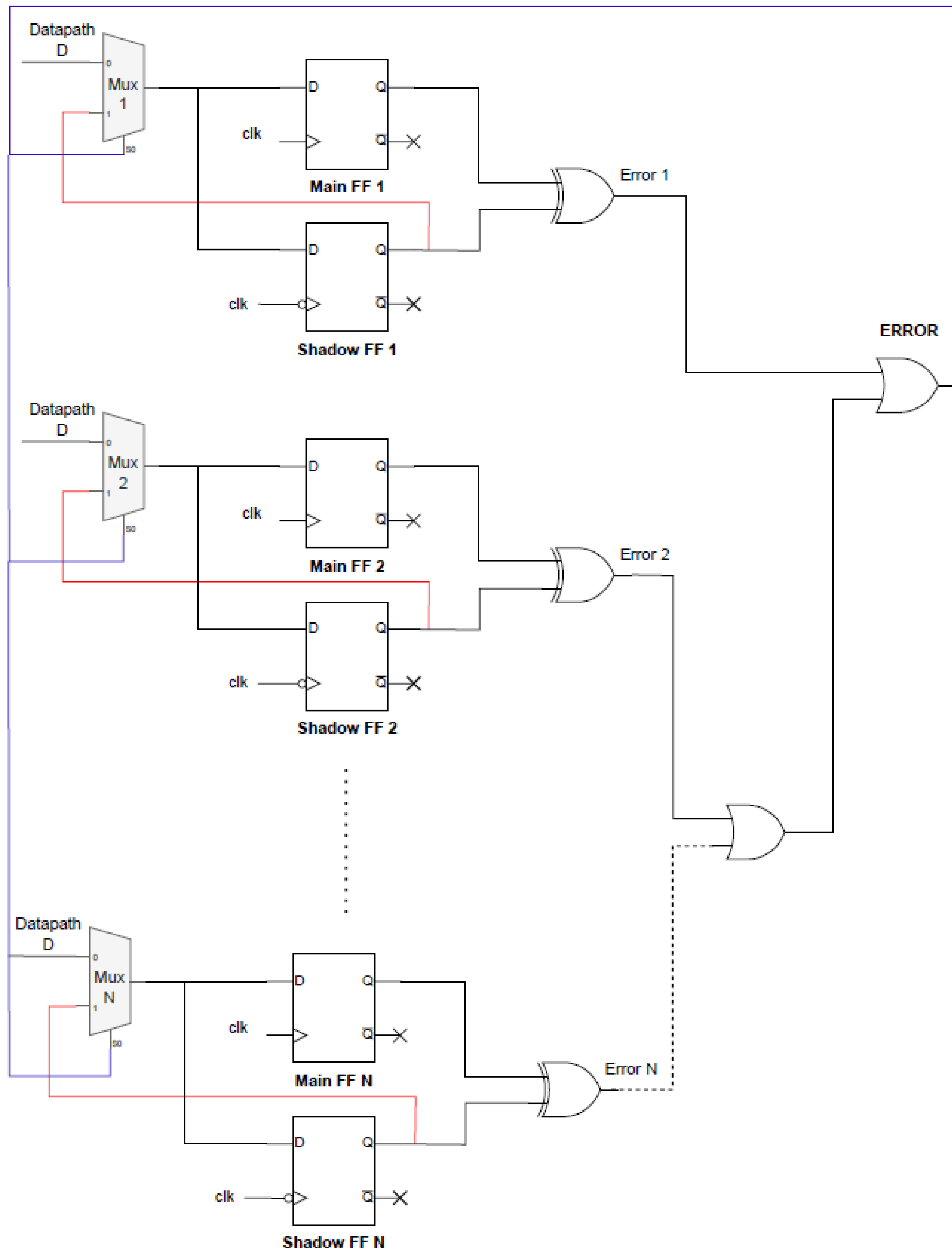

Figure 3.10: *Razor Architecture*

3.2.2 Razor Architecture in HW

The Razor architecture, whose implementation is shown in Figure 3.11, is fully integrated into the design using the entire implementation process. During this phase, the two-layer IP is modified by adding an output port to monitor the error signal exiting from the Razor logic. The error signal is also connected to GPIO, which allows it to be printed on the console after the design's hardware inference is completed.

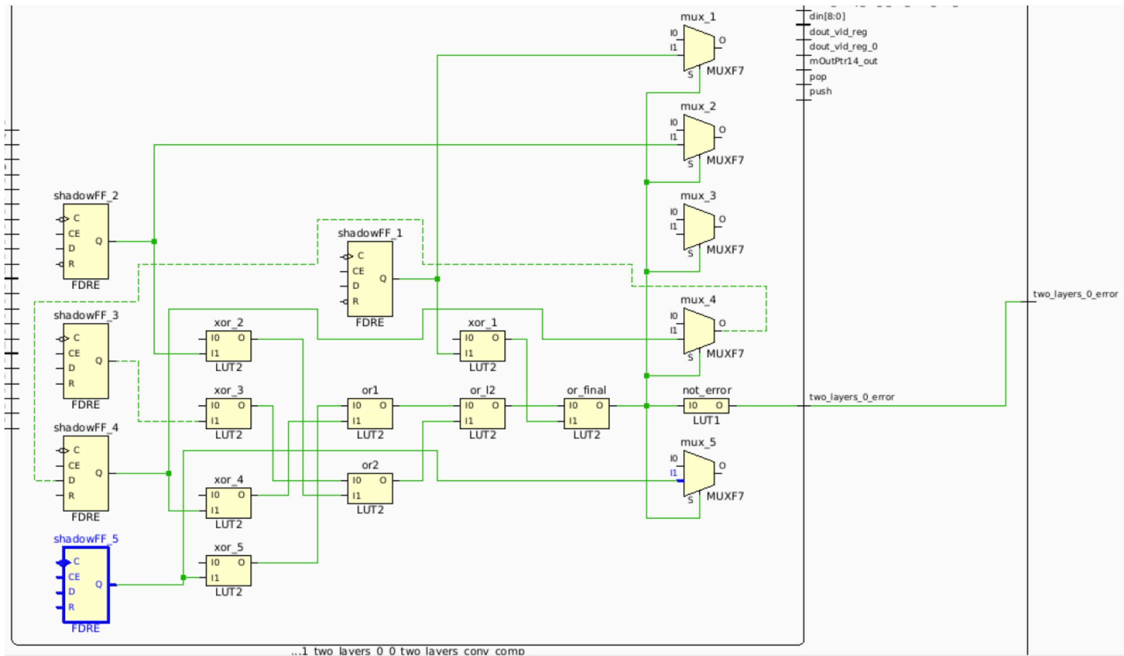

Figure 3.11: *Implemented Razor Architecture*

The Ultra96-V2 is used to test the error detection logic in hardware. After the bitstream is downloaded, the clock is dynamically reconfigured to evaluate the circuit at different frequencies. The table below shows the results of the circuit with the Razor logic at 100 MHz.

Total time	9.836675643920898 seconds
Total power	0.15625 W
Total energy	1.5369805693626404 J
Batch size	2000
images nums	10000
fps	1016.6036130488898
Accuracy	0.20630000531673431

3.2.3 Driving control signals with Razor error

The FSM controls the computational elements by using control signals to initiate, enable, block, and stop them. These control signals are directly affected by the empty and full signals of the FIFO in both input and output to the block. In case of an error, the global error signal is set, which halts the pipeline and forces the empty signals controlling the FIFOs in both input and output to the conv 0 for one clock cycle. If any of the empty signals sent to the convolution block are set to 0, the FSM block signal is triggered. This allows the data to be recovered by choosing the output of the shadow FF. In order to implement this method, one of the empty signals in the input block needs to be disconnected. The error signal is then negated, and an AND gate is created. This AND gate takes the negated error and the empty signal as inputs. The behavior of these two signals is summarized in the truth table provided below.

Truth table to drive empty signal		
empty	not error	out
0	1	STOP
0	0	STOP
1	1	RUN
1	0	STOP

To implement this first method (Figure 3.12), the necessary logic is integrated into a post-synthesis netlist using a TCL script. Once this is done, static timing analysis is performed to identify the most critical paths and any potential timing violations. During this stage, setup timing violations arise in paths involving the shadow flip-flops. Setup timing violations occur when the input data of the datapath flip-flop crosses the 2-to-1 multiplexer and is then sampled by the main flip-flop. After the comparison between the two flip-flops, the data crosses all the combinational logic. Then the error signal is propagated back as the selector of the multiplexer to enable the correct data in case of error. The input data to the shadow FF after the recovery arrives too late. When the arrival time is higher than the required time and the skew is negative, the result is a setup timing violation. To address this issue, the netlist is optimized, and automatic placement and routing are performed using the tool. At the design's operational frequency, the tool is capable of resolving the violations after implementation. However, when the frequency is increased, the timing becomes more sensitive, and the violations cannot be resolved anymore. Attempts have been made to resolve these timing issues by inserting delays as timing constraints to balance the arrival times, but without success. Delay elements, such as buffers, have also been tried to adjust the timing, but the delay couldn't be

controlled, and the clock skew increased. Furthermore, even when there are no timing violations and at low frequency, the design does not work in hardware. The inference process is blocked, and the DMA is not starting.

After multiple attempts, a new approach is being considered. The objective of this new approach is to disable the entire convolution block, which contains the control FSM and the computational part (DSPs), by utilizing the clock gating technique and using the error generated by the Razor circuit as an enable signal.

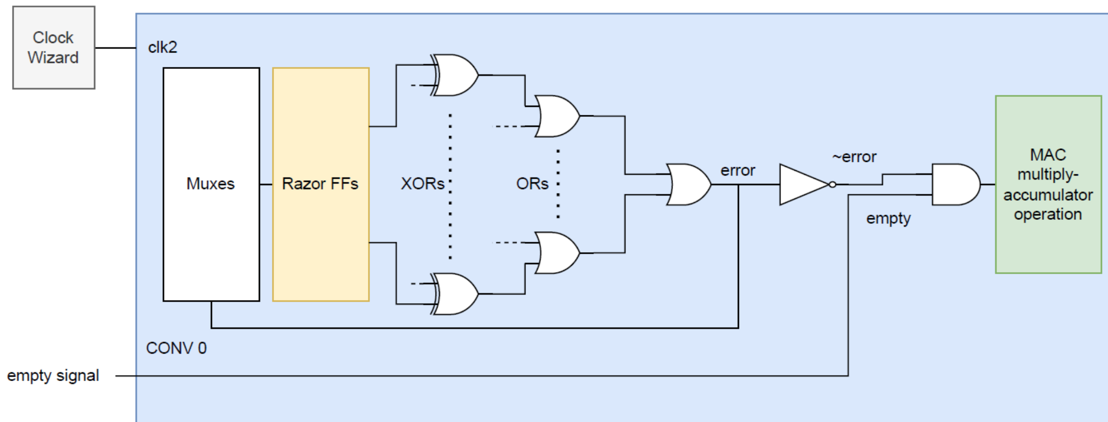

Figure 3.12: *Driving control signal*

3.2.4 Clock Gating

The thesis evaluated a second method for detecting errors (Figure 3.13), which involves applying clock gating to the clock that drives the convolution block. To implement clock gating in an FPGA, the primitive called BUFGCE is used, which is a clock buffer with a single gated input. Its behavior is simple: when the clock enable (CE) is low (inactive), the buffer output (O) is 0, but when CE is high, the input (I) is transferred to the output (O).

The clock buffer's enable pin is linked to the error signal resulting from the combinational logic after the Razor circuit. When an error is detected in any of the Razor cells, the error signal goes high, and its negated value is transmitted to the clock buffer. As a result, the block is disabled for one clock cycle. When the negated error signal goes high again, the correct value (the one sampled by the shadow FFs) is selected, and computation can resume. Several attempts have been made to implement clock gating and determine the best method in this design. Initially, a BUFGCE primitive was created and connected using a TCL command in the design flow process. However, this implementation caused problems with the clock tree managed by Vivado. The tool was unable to manage the timing and clock

tree optimization, which resulted in high timing violations. After further research, the error signal was connected to the clock wizard directly, always modifying the post-synthesis netlist. It was later discovered that the Clocking Wizard IP allows for clock gating with the ability to create directly an additional gated clock with its enable signal. Nonetheless, this modification did not yield any significant differences in comparison to manually adding the error enable signal using TCL.

The method works at low frequencies but with higher frequencies, timing violations occur, causing the design to stop working correctly. When a clock signal passes through a gate, it adds skew to the clock signal. ASICs can correct setup and hold time violations that occur due to such gating because they are rectified in the design layout process. FPGAs, on the other hand, lack such flexibility to correct the problems. Although one can calculate and predict such violations, they cannot be rectified using buffers, unlike in ASICs. Even if a buffer is introduced, the delay, location, and number of buffers are not predictable. Therefore, while normal gating can be done in an FPGA, highly constrained gating, where timing is crucial, can produce undesirable results.

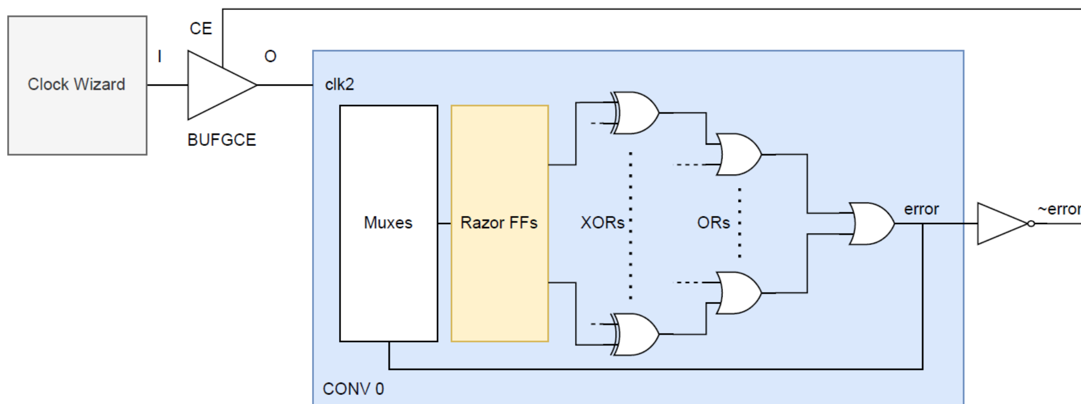

Figure 3.13: *Clock Gating Implementation*

3.3 Post Implementation Simulation

The circuit complete of error detection logic and error correction circuitry is then simulated to understand the source of the timing violations. To run a simulation of a circuit, the post-implementation netlist is used. This is done by running a batch mode simulation with Vivado commands. To simplify this process, a bash script is created that is designed to compile Verilog files and run the simulation using Xilinx tools. The script starts by compiling the necessary source files and testbench files. It then adds the input simulation files using the command *xvlog*. After compilation, the script proceeds to the elaboration stage using the *xelab* command, which links the compiled modules and elaborates the design hierarchy. Finally, the simulation is launched using the *xsim* command with the specified top-level module. Additionally, in the top-level simulation testbench, the Standard Delay Format (SDF) file is annotated to simulate the timing delays for all cells in the design. Although the simulation is used, there are still timing problems, and the circuit's expected behavior is not achieved.

After testing both in simulation and on hardware, the error detection circuit, clock modification, and management of multiplexers are functioning as expected. However, the results from the error recovery methods are inconclusive.

To better understand the limitations of the NN design on FPGA, Chapter 4 evaluates a new approach to determining the design's maximum operating point.

Chapter 4

Frequency Dynamic Reconfiguration

In this second part of the thesis, the focus is on determining the maximum frequency at which the design fails. The objective is to speed up the neural network's slowest task until the results are incorrect while ensuring that the other parts of the circuit continue to operate at their original frequency. This involves making some modifications to the design and writing an algorithm to perform a frequency sweep and identify the breaking point frequency. The modified RTL code is first simulated to ensure that it is functioning properly. Once the simulation is successful, the bitstream is downloaded onto the Ultra96-V2 board, and a Python algorithm is written to dynamically reconfigure the frequency.

4.1 RTL simulation accelerating one convolution block

In order to evaluate the functionality of the design in simulation, a new RTL project is generated, comprising Verilog source files, testbench files, and input files. Vitis HLS tool automatically generates a testbench, which is subsequently modified to optimize design testing. Specifically, a second clock is introduced to accelerate the convolution 0 and the input and output blocks. To accelerate only one block, two clock domain crossings are necessary. To facilitate clock domain conversion, the FIFOs utilized in the generated design are replaced with asynchronous FIFOs, which operate at different writing and reading clock frequencies. Subsequently, the clock is modified in the top-level testbench, and the results are compared with the golden simulation to confirm the correct operation of the design under two different clock scenarios, one of which is faster than the other. The RTL simulation results are reported and discussed in Section 4.2 below.

4.1.1 Clock Domain Crossing

As previously mentioned, in order to speed up the convolution block, two clock domains crossing are required. The blocks involved are the input FIFO, which provides the input stream, and the output FIFO, which reads the results generated by the convolution. Furthermore, the parameter blocks that provide the weights are sampled using clk2, which is the same clock used to sample conv0. Figure 4.1 provides a clearer representation of the clock domains used in the design.

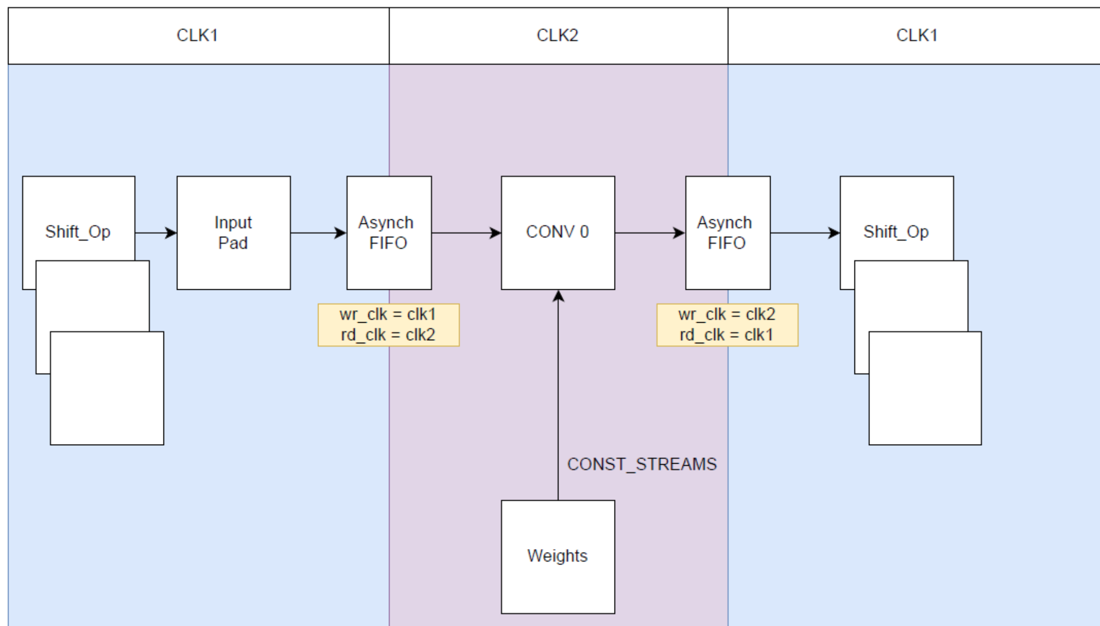

Figure 4.1: *Clock Domain Crossing*

To enable communication between two different clock domains, an asynchronous FIFO with separate reading and writing clocks is utilized, as depicted above. This component is implemented using a macro from AMD. The purpose of employing this particular FIFO is to establish two different clocks and ensure the proper reading and writing of data.

Asynch FIFO

To implement a replacement for the single-clock FIFO, a parameterized macro called XPM_FIFO_ASYNC is used, which is accessible in Vivado. This macro allows modification of various parameters such as FIFO width, read latency, and clocks. To incorporate this solution into a design and alter the clock domain, the standard Verilog instantiation template provided in AMD's guide is adjusted accordingly. In the case of an asynchronous FIFO, the write operation is executed

when the FIFO is not full, and the write enable signal is asserted on every writing clock cycle. Conversely, the read operation is performed when the FIFO is not empty, and the read enable signal is asserted on each reading clock cycle. To implement the FIFO functionality, the First-Word-Fall-Through (FWFT) read mode has been chosen. The behavior for read and write operations is shown in Figure 4.2 [20].

As a part of the RTL project, asynchronous FIFOs have been added to simulate the modified circuit's behavior. Subsequently, the previously generated two-layer IP has also been modified, and a new bitstream has been created. The bitstream is generated and downloaded onto the hardware board. To test the results on the hardware, the Ultra96-V2 board is used. A Python script and the Pynq environment are used to vary the frequency once the reference design is loaded on the board and the results are printed.

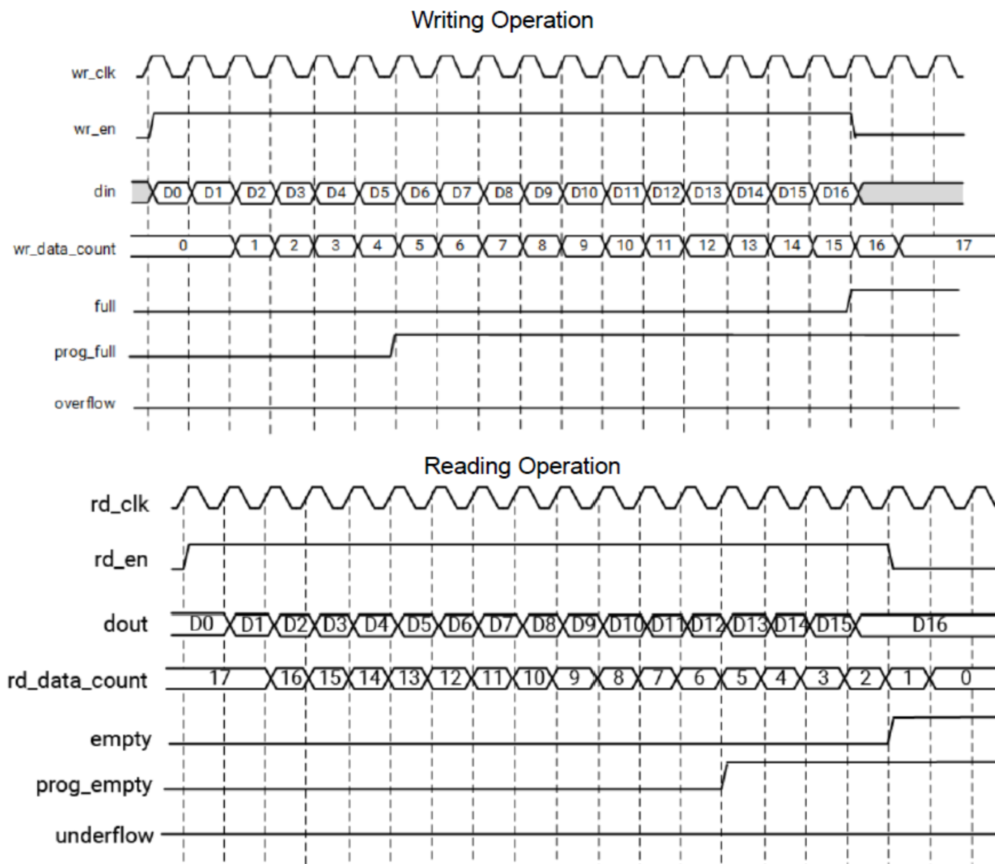

Figure 4.2: *Asynch FIFO Writing and Reading Operation*

During this phase, various blocks of the schematic presented in Figure 3.7 have been modified. Specifically, the FIFOs have been adjusted, and clk2 has been connected to the blocks to enable the clock domain to be changed. These alterations are underlined in Figure 4.3 in contrast to the previous version of the diagram.

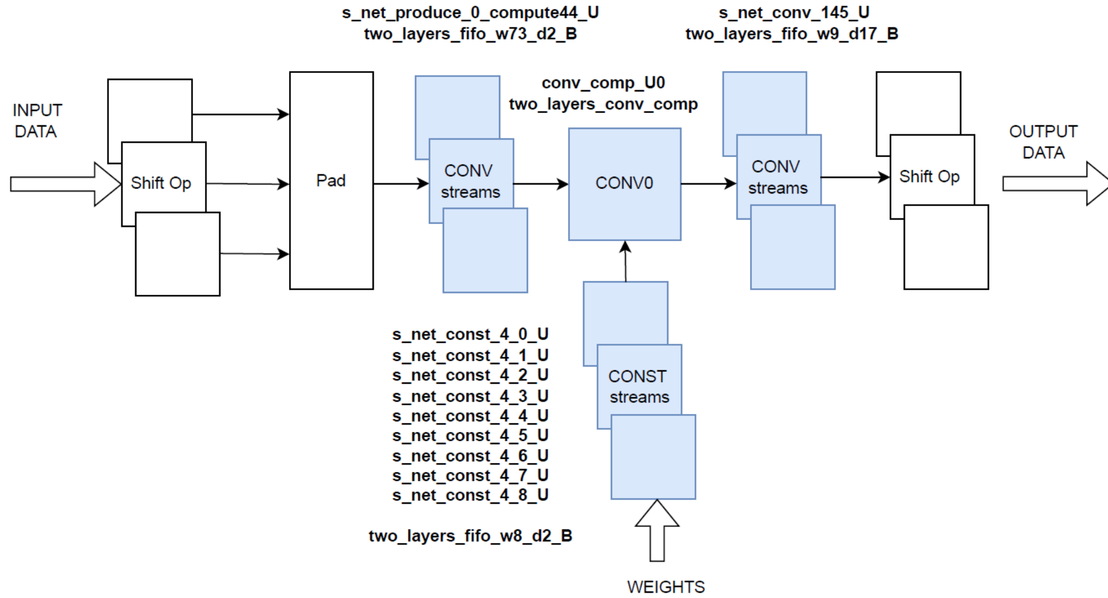

Figure 4.3: *Modified blocks to change clock domains*

The input FIFO named *two_ layers_ fifo_ w73_ d2_ B* and the output FIFO named *two_ layers_ fifo_ w9_ d17_ B* are modified to work as an asynchronous FIFO with two clocks. Additionally, the *s_ const_ 4_ ** parameter blocks are now driven by the second clock.

4.2 RTL simulation Results

To ensure the proper functioning of the two-layer neural network, the second clock must be assigned to the convolution block, parameter streams, and FIFO buffers. Additionally, the FIFOs are configured according to the parameters discussed in the previous paragraph. Conducting a behavioral simulation is the next step, which involves selecting appropriate clock period values and defining the relationship between the two clocks in the top-level simulation. As the design processes two images, the predicted values become available in the output buffer immediately after the computation stage. These results are written as output data and can be viewed directly on the simulation waveform.

The following are the predicted values for both images.

Predicted values for image 1									
0xf7	0x0b	0x04	0x08	0x00	0x03	0xf3	0x05	0xf4	0x09

Predicted values for image 2									
0x18	0x0f	0xfe	0x00	0xea	0xf9	0xd3	0xfc	0x13	0x13

The table below presents a comparison of the design in two scenarios. In the first scenario, the two clocks are operating at the same frequency, and this is meant to simulate the design at its operative frequency. This helps in obtaining accurate results and provides a golden model for comparison with the second scenario. In the second scenario, the same design is tested with clk2 set to double frequency. This is done to investigate the performance of the design accelerating the convolution block. The results obtained from the second scenario remain correct, as evidenced by Figure 4.4. Additionally, the simulation time for completing the same task has been reduced by nearly half.

RTL Behavioral Simulation Results			
clk1	clk2	Image 1/2	Image 2/2
100 MHz	100 MHz	[10.71%] @987455000ps	[100.00%] @1970595000ps
100 MHz	200 MHz	[5.68%] @495975000ps	[100.00%] @987545000ps

Figure 4.4: *RTL simulation with clk2 at 200 MHz*

Figure 4.4 shows that when the convolution block is accelerated by double the operating frequency, the simulation produces the expected results and reduces the simulation time. However, when attempting to triple the second clock's frequency to 300 MHz while keeping the rest of the system at 100 MHz, the results become corrupted, and the output FIFO writing operation does not occur correctly.

After simulating the RTL design, an inference script is used to load the same design onto the hardware board.

4.3 Inference

The design is inferred by downloading an overlay into the PL during boot time. An overlay typically includes a bitstream that configures the FPGA fabric, a Vivado TCL file that identifies the available IP, and a Python API that exposes the IPs as attributes. To load an overlay, the Pynq class *overlay* can be used. Once the overlay is loaded, the IPs are treated as objects of the class and their registers can be written to. Notably, this is the case with the clocking wizard IP, where the clk2 frequency can be adjusted by writing the divisor and the multiplier with respect to the VCO frequency of reference. This can be achieved by writing to the dynamic reconfiguration registers located in the register space, as described in [21].

To infer the design, a bash script is used, which calls different TCL scripts. These scripts define the board characteristics, the dataset used, and the allocation of input and output buffers. Two Vivado files, the bitstream file (.bit) and the hardware handoff file (.hwh), are required to load the design. The HWH (hardware handoff) file is generated automatically from the Vivado IP Integrator block design. PYNQ uses it to automatically identify the Zynq system configuration, IP versions, interrupts, resets, and other control signals. Based on this information, PYNQ can modify some parts of the system configuration automatically, enable or disable features, and connect signals to corresponding Python methods. To program the Ultra96-V2, Jupyter Notebook is used to run Python scripts. Jupyter Notebook is a free web app for live coding, data visualization, and document creation. It supports multiple programming languages, with Python being the most popular. It allows users to run code and view output data interactively. Once the overlay has been instantiated and downloaded, two DMA objects are created. One object is used to send data through the input channel, and the other is used to receive data through the output channel. An input buffer is allocated for reading data from memory. Similarly, an output buffer is also allocated to write data to the memory. The read channel reads data from the PS DRAM and writes it to the stream buffer, while the write channel reads data from the stream and writes it back to the PS DRAM.

Below is the Python script used to vary the frequency of the clocks. First, the clocking wizard IP is accessed as an object of the overlay class, and the necessary configurations are written to change the frequency. Once the correct values are written, the status register is checked, and the inference can be executed as soon as the clock is stable. Finally, the results are displayed on the console.

Algorithm 2 Algorithm for dynamically reconfigure clocks

Require: import pynq
Require: from pynq import Overlay

```
overlay = Overlay('./overlay/design_1.bit')
dma = overlay.axi_dma_0                          ▷ DMA instance
in_buffer definition
out_buffer definition

clk = overlay.clk_wiz_0                          ▷ Clock Wizard instance
clkin = clk.read(0x200)
clk1 = clk.read(0x208)
clk2 = clk.read(0x214)

                                   ▷ Clocks Dynamic Reconfiguration
clk.write(0x200,0x00000601)                      ▷ pll clock [8:15]
clk.write(0x208,0x00000009)                      ▷ clk1 = 100 MHz
clk.write(0x214,0x00000009)                      ▷ clk2 = 100 MHz
clk.write(0x25C,0x00000003)      ▷ Load bit for clock reconfiguration
status = clk.read(0x04)    ▷ Status bit to 0 during reconfiguration
while status ≠ 0x1 do
    wait
end while

Inference
```

4.3.1 Results expectation in HW

Once the design is processed, the console displays several results, including the total time, power, and energy consumed by the design. Two key indicators are the FPS and the neural network's accuracy. FPS stands for frames processed per second, while accuracy represents the degree of precision with which the neural network predicts results. The neural network generates predicted values which are then compared with the golden model. The accuracy is determined by calculating how many predicted values match the correct results. These predicted values are obtained from the output buffer and are the maximum values. The FPS calculation, instead, depends on the inference's total time and the batch size and number of batches.

The expected behavior is that as the frequency rises, the frames per second increase while losing accuracy. If the frequency is plotted against both the FPS and accuracy, two graphs similar to Figure 4.5 and Figure 4.6 are expected.

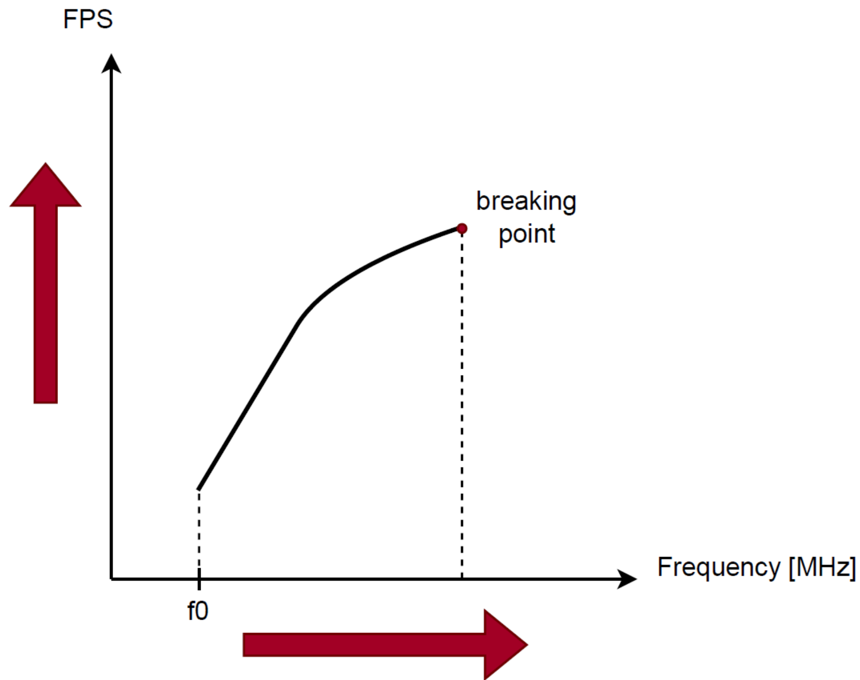

Figure 4.5: *Expected Result for FPS*

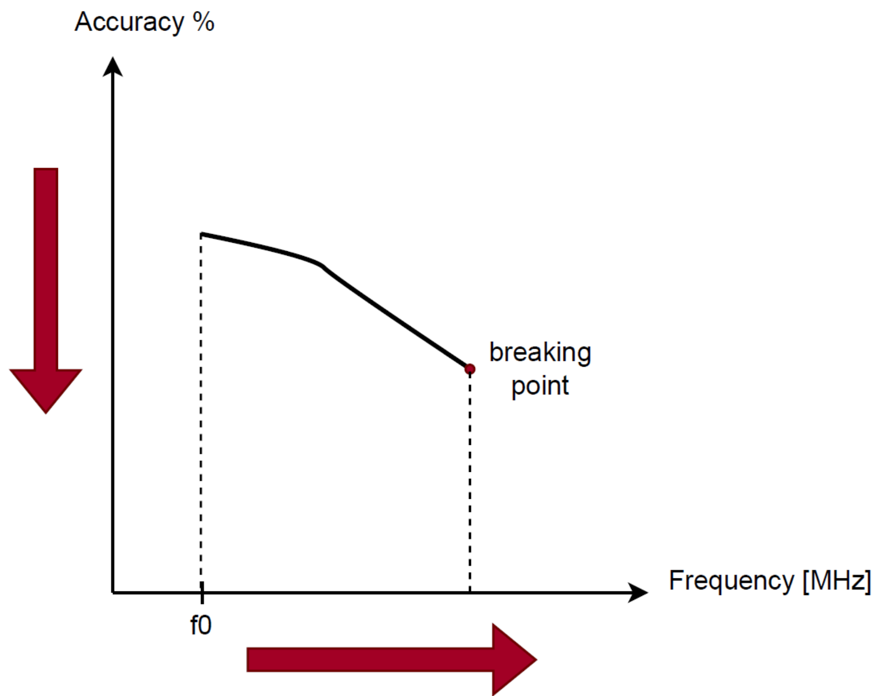

Figure 4.6: *Expected Result for the accuracy*

4.4 Results

The solution for the two-layer NN with the two clocks at the same frequency achieves an accuracy of 20.63% with a throughput of 1014 FPS on the Ultra96-v2 board. Frequency and accuracy have an inverse relationship. This means that decreasing the clock period will reduce accuracy. The starting design has low accuracy, so by increasing the frequency, the accuracy becomes too low, and thus, the values predicted by the neural network are not reliable.

The results obtained for FPS (frames per second) and accuracy are given below.

FPS, Accuracy vs. Frequency		
Frequency clk2	FPS	Accuracy
100 MHz	1014	20.63%
200 MHz	3046	12.51%
300 MHz	–	10.26%

The figures below show the plot for the throughput and accuracy (Figure 4.7 and 4.8, respectively). As anticipated, the number of frames increases with the frequency. The trend is as expected; however, after reaching 300 MHz, the writing is corrupted, leading to a faulty design.

Hardware tests are conducted by first loading the design with the second clock set to the same frequency as the main clock driving the rest of the circuit. Once it is confirmed that the design has worked with the new FIFO management and the addition of the secondary clock, the frequency can be increased by writing to the register through the Python script described in paragraph 4.3. To dynamically reconfigure the clock frequency, it is necessary to compute the divisor of the reference clock to obtain the desired frequency. Once the clock is ready and the status bit is high, the inference can be run without reloading the overlay. Because reloading the overlay will reconfigure the clock parameters to the ones chosen in the generated bitstream. After rerunning the inference, the results are printed on the console and saved in a text file.

After analyzing the data presented in the table and the graphs below, it can be observed that the accelerated block can be speeded up to 2 times compared to the operative frequency. However, beyond 200 MHz, the design starts malfunctioning. At 200 MHz, there is a significant decrease in accuracy. At 300 MHz, the FPS (frames per second) is lower than at 200 MHz, and the accuracy is very low. Additionally, even if the DMA is not blocking and the inference is brought out, the output buffer is filled with zeros, indicating that the design is not working correctly after 200 MHz.

Prior RTL simulations have demonstrated that the convolution block can achieve a frequency acceleration of up to twice that of the system's other components. The latter hardware tests validate this result.

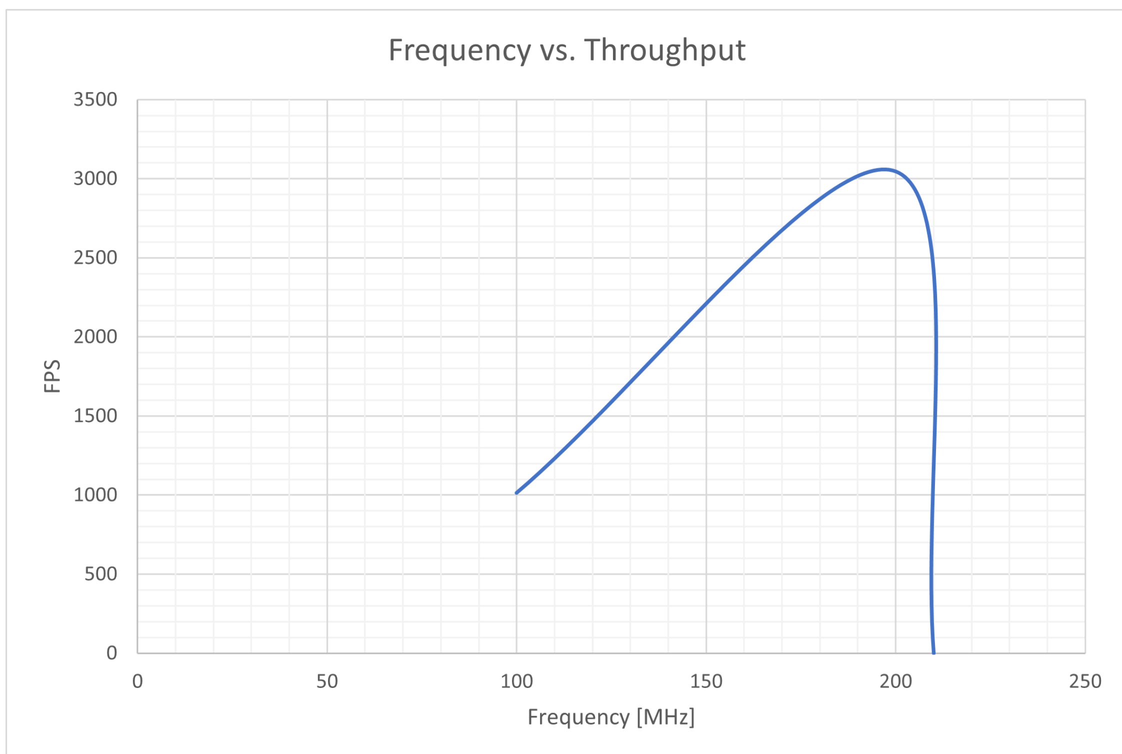

Figure 4.7: *Neural Network Troughput*

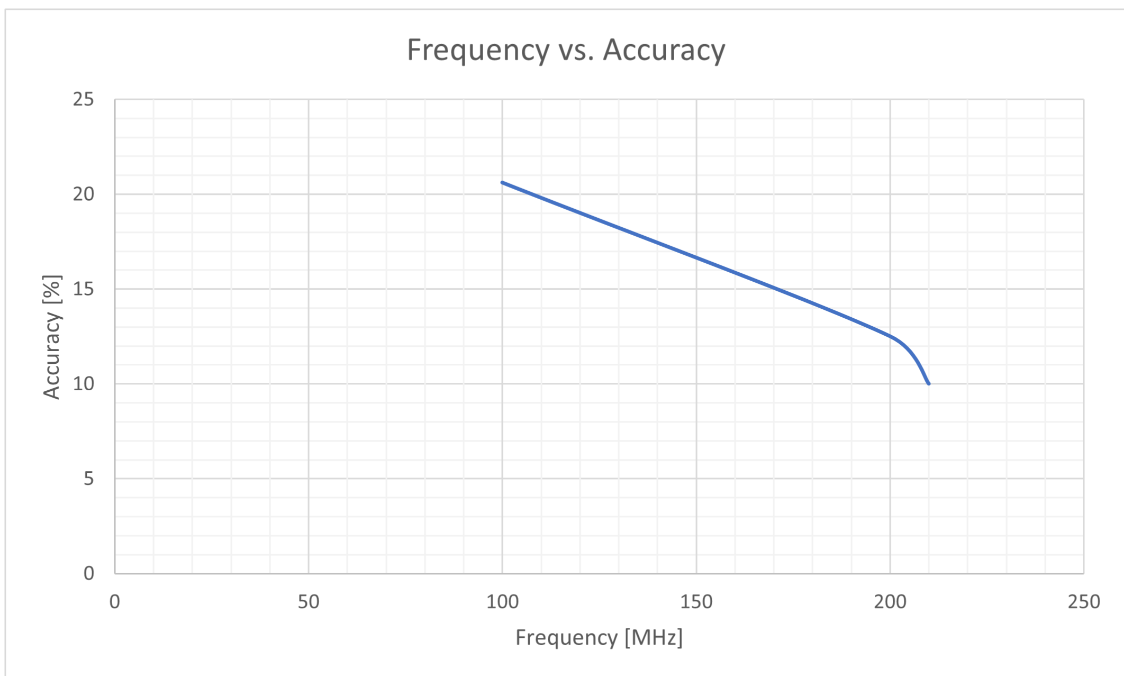

Figure 4.8: *Neural Network Accuracy*

Chapter 5

Conclusions and Future Work

This thesis presents a detailed discussion of the implementation of the Razor circuit along with two methods for error detection. The design is further modified to reconfigure the frequency in hardware.

An automated design flow has been developed to modify the original architecture by adding the Razor circuit and all the logic necessary for error detection and correction. The Razor technique has been integrated into only a small portion of the design. The division of the circuit into small blocks allows only the block where the error is detected to stall for one clock cycle, while the rest of the circuit continues working without halting. The design's clock tree has been modified to implement Razor by managing the Clocking Wizard IP in relation to the design processing system. Once the Razor architecture is integrated into the design, the neural network design is tested in hardware on both Kria KV260 and Ultra96-V2 boards. The testing includes two steps. Firstly, the error detection process is checked, which involves the Razor cells and the multiplexers on the input paths. The results of this step are reported. Secondly, the error recovery methodology is implemented and tested.

Two error detection methods are evaluated and integrated into the design flow. The Razor technique has been investigated and adapted to the proposed architecture. The first method utilizes the empty signal's behavior to stall the pipeline when an error occurs. However, results have been inconclusive due to timing violations along Razor circuit paths. The second method, which involves clock gating, has also been integrated into the design flow but found unsuitable on FPGAs for highly constrained gating, where timing is crucial. A post-implementation simulation has been conducted to investigate the reasons for the timing violations associated with both methods. Unfortunately, the problem persists even after the simulation, indicating that the implementation of the Razor methodology is not entirely successful. In fact, only the error detection part has been proven to work in both simulation and hardware. The design that incorporates the Razor circuitry for error detection

has been tested, but the accuracy of the results from the neural network is quite low. A potential next step in this work could be to apply the automated implementation flow to integrate the Razor technique into larger neural networks with higher accuracy. This would increase the performance and frequency, leading to more precise outcomes.

In order to conduct a thorough evaluation of the performance of the design of the two-layer neural network architecture, a new project has been initiated. The primary objective of this project is to determine the maximum frequency at which the design can operate properly. To achieve this objective, it has been decided to accelerate only one part of the design, specifically the same block on which the Razor technique was previously implemented. The goal is to identify the breaking point of the design and the frequency at which the circuit has maximum performance with proper operation. The preexisting source and simulation files generated by Vitis HLS are being utilized. A second clock has been added to the modified HDL code to accelerate the convolution block identified earlier via Razor methodology. Additionally, some FIFOs are modified, and clock domain crossing has been considered to ensure that the circuit operates properly. The design's initial stage of testing involves RTL behavioral simulations, followed by hardware testing using the Ultra96-V2. The simulations report the correct predicted values of the network. Parameters such as FPS and accuracy are the results of hardware inference. The results of the RTL simulation demonstrate that the convolution block currently under examination can be accelerated by a factor of two in comparison to the frequency of the remaining design. Furthermore, the same behavior of the design is visible when it is loaded in hardware. This was verified through the results obtained by running the block twice the primary frequency.

Modifying the error recovery strategy can improve the current work. It is essential to investigate the timing violations that occur due to the propagation of the error signal through the design, as these can significantly impact the system's reliability. By identifying the root cause of the timing violations, a more effective solution can be found.

One way that can be tried to address the issue is to incorporate a universal enable signal that traverses all blocks. This signal should be gated with the error signal to prevent all flip-flops from enabling simultaneously. As previously stated, another potential improvement is to apply an automated implementation process to larger neural networks. This approach can lead to better initial accuracy and more room to enhance performance.

It is worth noting that this thesis did not focus on power consumption and area overhead. Therefore, to fully optimize the system, it would be beneficial to conduct a study that examines these aspects after implementing Razor. This study can help identify areas where the system can be optimized and improved

to reduce power consumption and minimize area overhead without compromising system performance.

Bibliography

[1] Daniel Gomez-Prado and Maciej Ciesielski. *A Tutorial on FPGA Routing.* Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, USA.

[2] Edward Stott, Joshua M. Levine, Peter Y.K. Cheung and Nachiket Kapre. *Timing Fault Detection in FPGA-based Circuits.* In: 2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines. 2014, pp. 96-99. DOI: 10.1109/FCCM.2014.32.

[3] Joseph Crop, Evgeni Krimer, Nariman Moezzi-Madani, Robert Pawlowski, Thomas Ruggeri, Patrick Chiang and Mattan Erez. *Error Detection and Recovery Techniques for Variation-Aware CMOS Computing: A Comprehensive Review.* In: 2011 Journal of Low Power Electronics and Applications. 2011, pp. 334-356. DOI:10.3390/jlpea1030334.

[4] Bharadwaj Amrutur, Nandish Mehta, Satyam Dwivedi and Ajit Gupte. *Adaptative Techniques to Reduce Power in Digital Circuits.* In: 2011 Journal of Low Power Electronics and Applications. 2011, pp.261-276. DOI:10.3390/jlpea1020261.

[5] Jose Nunez-Yanez. *Energy proportional computing in commercial FPGAs with adaptive voltage scaling.* In: FPGAworld '13: Proceedings of the 10th FPGAworld Conference. 2013, pp.1–5. https://doi.org/10.1145/2513683.2513689.

[6] Edward Stott, Joshua M. Levine, Peter Y.K. Cheung and Nachiket Kapre. *Measuring Timing Errors in FPGA-based Circuits.* In: 2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). 2014, DOI:10.1109/FCCM.2014.32.

[7] Dan Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner and Trevor Mudge. *Razor: a low-power pipeline based on circuit-level timing speculation.* In: 2003 Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-36. 2003, DOI: 10.1109/MICRO.2003.1253179.

[8] Filippo Minnella, Teodoro Urso, Mihai T. Lazarescu and Luciano Lavagno. *Design and Optimization of Residual Neural Network Accelerators for Low-Power FPGAs Using High-Level Synthesis.* In: arXiv:2309.15631, 2023.

https://doi.org/10.48550/arXiv.2309.15631.

[9] Keiron O'Shea and Ryan Nash. *An Introduction to Convolutional Neural Networks.* In: arXiv:1511.08458, 2015. https://doi.org/10.48550/arXiv.1511.08458.

[10] AMD. *Vitis AI.* https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html.

[11] Xilinx. *FINN.* xilinx.github.io/finn.

[12] Michaela Blott, Thomas B Preußer, Nicholas J Fraser, Giulio Gambardella, Kenneth O'brien, Yaman Umuroglu, Miriam Leeser, and Kees Vissers. Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 11(3):1–23, 2018.

[13] Alessandro Pappalardo. Xilinx/brevitas, 2023.

[14] Pythorch framework. https://pytorch.org/features/.

[15] Matthew Fojtik, David Fick, Yejoong Kim, Nathaniel Pinckney, David Money Harris, David Blaauw and Dennis Sylvester. *Bubble Razor: Eliminating Timing Margins in an ARM Cortex-M3 Processor in 45 nm CMOS Using Architecturally Independent Error Detection and Correction.* In: IEEE Journal of Solid-State Circuits (Volume: 48, Issue: 1, January 2013). 2012, pp.66-81, DOI: 10.1109/JSSC.2012.2220912.

[16] Bharath Naidu Vangapandu and Anu Chalil. *FPGA Implementation of High-Performance Montgomery Modular Multiplication with Adaptive Hold Logic.* In: 2022 6th International Conference on Computing Methodologies and Communication (ICCMC). 2022, DOI: 10.1109/ICCMC53470.2022.9754043.

[17] Ushio Jimbo, Ryota Shioya and Masahiro Goshima. *Application of Timing Fault Detection to Rocket Core on FPGA.* In: 2018 Sixth International Symposium on Computing and Networking Workshops (CANDARW). 2018, DOI: 10.1109/CANDARW.2018.00041.

[18] Alexander Brant, Ameer Abdelhadi, Douglas H.H. Sim, Shao Lin Tang, Michael Xi Yue, and Guy G.F. Lemieux. *Safe overclocking of tightly coupled CGRAs and processor arrays using razor.* In: 2013 IEEEInt. Symp. onField-Programmable CustomComputingMachines(FCCM). 2013, pp.37–44.

[19] AMD. *PYNQ.* https://pynq.readthedocs.io/en/v2.3/pynq_overlays.html.

[20] AMD. *XPM_FIFO_ASYNC.* https://docs.xilinx.com/r/en-US/ug953-vivado-7series-libraries/XPM_FIFO_ASYNC.

[21] AMD. *Clocking Wizard IP Product Guide.* https://docs.amd.com/r/en-US/pg065-clk-wiz/Register-Space.