UPPSALA
UNIVERSITET

IT mINS 25 014
Degree project 30 credits
10 2025

Systolic Array Based Neural Network Accelerator Design with TangramFP MAC Unit

Power Consumption Evaluation with Performance and Area Trade-Off Analysis on FPGA

Abdelmojeb Eltaib Elsidig Mohammednour

Master's Program in Embedded Systems

UPPSALA
UNIVERSITET

Systolic Array Based Neural Network Accelerator Design with TangramFP MAC Unit

Abdelmojeb Eltaib Elsidig Mohammednour

Abstract

This thesis presents the design, implementation, and evaluation of a systolic array architecture enhanced with the TangramFP multiply–accumulate (MAC) unit. TangramFP reduces dynamic power consumption by skipping ineffectual partial products during IEEE-754 floating-point operations with controlled error bounds with the mode estimation unit. This work investigates how such unit-level energy savings can be scaled to array-level computation through the integration of TangramFP MACs into systolic arrays.

The design included optimizing the MAC for single-cycle operation, developing a dedicated mode estimation unit to guide TangramFP computations, and integrating of these components into a pipelined systolic array prototype with mode propagation. The system was verified through behavioral, post-synthesis, and post-implementation simulations, and deployed as part of SoC architectures on FPGA platforms (Zynq-7000 and Kintex-7). Software control and real-time monitoring via the Integrated Logic Analyzer (ILA) were also implemented to validate functionality.

A Comprehensive evaluation was performed on the configurations of the systolic array (2×2, 4×4, and 8×8) at precisions FP16 and FP32. Results show a consistent dynamic power reduction —up to 50% at smaller sizes and lower precision— flattening around 25% at larger arrays, with clock rates stabilizing around 71 MHz for FP16 and 52 MHz for FP32. These findings confirm that TangramFP delivers scalable power efficiency without sacrificing correctness, offering a promising pathway for energy-efficient floating-point systolic arrays suited to AI acceleration, despite scalability limits imposed by FPGA resources.

Faculty of Science and Technology

Uppsala University, Place of publication Uppsala

Supervisor: Yuan Yao Subject reader: Stefanos Kaxiras

Examiner: Bengt Jonsson

Contents

1 Introduction **1**
- 1.1 Systolic Arrays . . . 1
- 1.2 TangramFP . . . 2
- 1.3 TangramFP Mechanism . . . 3

2 Methodology **7**
- 2.1 Development Flow . . . 7
- 2.2 Hardware Design Methodology . . . 8
  - 2.2.1 System Architecture Definition . . . 8
  - 2.2.2 Integer–MAC Systolic Array Prototype . . . 8
- 2.3 TangramFP Integration . . . 9
  - 2.3.1 Single-Cycle MAC Redesign . . . 9
  - 2.3.2 Mode-Selection Unit . . . 9
- 2.4 System Integration in Vivado . . . 9
- 2.5 Software Flow . . . 9
- 2.6 Real time Debugging . . . 9
- 2.7 Tools and Equipment . . . 10
- 2.8 Evaluation . . . 10

3 Design **11**
- 3.1 Systolic Array Architecture . . . 11
  - 3.1.1 Designing the Systolic Array . . . 11
  - 3.1.2 Building Basic Systolic Array with Integer MAC . . . 12
  - 3.1.3 Module Interface . . . 12
  - 3.1.4 Streaming Input Data A . . . 12
  - 3.1.5 Data Movement Inside the Array . . . 13
  - 3.1.6 Streaming-out the Output . . . 15
- 3.2 Integrating TangramFP MAC . . . 15
  - 3.2.1 Mode Generation for TangramFP Systolic Array . . . 16
  - 3.2.2 Adaptation of TangramFP . . . 20
- 3.3 Interfacing . . . 22
- 3.4 Integrating into SoC . . . 23
- 3.5 Suggested Optimizations for Different Applications . . . 24
- 3.6 Pipelining . . . 26

4 Implementation **27**
- 4.1 Behavioral Simulation . . . 27
- 4.2 Synthesis . . . 27
  - 4.2.1 Synthesizing Systolic Array First Structure . . . 27
  - 4.2.2 Redesign of Considerations . . . 28

4.2.3 I/O Triangular Delays Instead of Counter-select Delays . . . 29
4.2.4 Pipeline and Control . . . 30
4.3 Simulation and Synthesis . . . 32
4.4 Implementation and PS-PL Testing . . . 34
4.5 RTL Improvements for Timing Closure . . . 37

5 Evaluation **41**
5.1 Resource Utilization . . . 41
5.1.1 Systolic Array Utilization for Different Sizes . . . 42
5.2 Timing Evaluation . . . 42
5.3 Static Power Analysis . . . 44
5.4 Switching Activity Based Power Analysis . . . 45
5.5 Discussion . . . 46

6 Conclusion **49**

7 Limitations and Future Work **51**
7.1 False Maximum Exponent and Over Approximation Mode . . . 51
7.2 Carry Chain and High Performance . . . 51
7.3 Memory Inference in FPGA . . . 52
7.4 Generic and Realistic Systolic Array . . . 52
7.5 Tiling . . . 52
7.6 Leveraging TangramFP for Bandwidth . . . 53
7.6.1 Internal Memory and Systolic Array Bandwidth . . . 53
7.6.2 External Transmission Bandwidth . . . 53

List of Figures

1.1 FP formats for DNN training. 3
1.2 3:8 and 1:5:5 splits featuring three modes of operation: i) Full Mode; ii) SkipBD Mode; iii) AC Mode Yao et al. [2024] 4
1.3 Addition stage of the first three modes 5

3.1 Systolic array 11
3.2 Counter-select based feeding A input to the systolic array 13
3.3 Counter of the entire operation 13
3.4 Counter-select based buffering output 15
3.5 mode selection unit for the entire matrix multiplication 17
3.6 Generating one column of modes per clock cycle 18
3.7 mode selection unit for the for 1 column of A and all rows of W . . 18
3.8 mode input triangular delay 19
3.9 Propagation of mode inside the systolic array 19
3.10 simplifying Carry-save 3 input adder 21
3.11 Configurations for cut,P, and Q cases 23
3.12 Abstract organization of the systolic array interface 24
3.13 Block design of SoC with systolic array IP 25

4.1 The schematic shows that the optimizer has removed the output signals 29
4.2 Reorganization of systolic array, interface, mode selection unit . . . 30
4.3 Changing from counter based delays into triangular delays 30
4.4 State machine for receiving 31
4.5 Systolic array pipeline controller 32
4.6 Waveform for systolic array pipeline simulation 33
4.7 SoC matrix-result data communication in ILA 35
4.8 Multiple matrix multiplications in a pipeline 36
4.9 Systolic array design in SoC system using Microblaze in Kintex-7 . 39
4.10 ILA capture transaction of 4 multiplicand matrices to the systolic array and the results of the multiplications 40

5.1 LUT utilization of the systolic array and the mode selection unit for different sizes and FP precision 42
5.2 Carry traversing on carry chains inside a critical path in the device after implementation 44
5.3 Static power estimation for the systolic array, whole system, and mode unit of sizes 2×2, 4×4, and 8×8 in 16 and 32 bits. 44
5.4 Dynamic power estimation of the systolic array system: (a) whole design with AXI interface, (b) systolic array only, (c) mode select unit, and (d) power difference between Full and Skip mode. 46

List of Tables

4.1 Key delays timing in the systolic array 31

5.1 LUT utilization report for MAC 16 bit 41
5.2 LUT utilization report for MAC 32 bit 42
5.3 Minimum clock period and maximum clock rate for different systolic array sizes 43

Chapter 1

Introduction

1.1 Systolic Arrays

Since the introduction of systolic arrays in the 1980s, they have been extensively researched and widely adopted in various applications, particularly those requiring high-throughput and parallel computation. Originally proposed to address the need for efficient hardware implementations of linear algebra operations, systolic arrays consist of regular tightly coupled processing elements (PEs) that rhythmically compute and pass data through a network—much like the pulse of a heartbeat, which inspired the term "systolic." With the rapid advancement and growing dominance of artificial intelligence (AI), systolic arrays have gained renewed attention as a foundational architecture for domain-specific accelerators, particularly in deep learning. Modern neural networks rely heavily on matrix multiplication and tensor operations, which naturally map to the structured and parallel processing capabilities of systolic arrays. This has made them a key component in the design of specialized AI hardware, such as Google's Tensor Processing Units (TPUs), which leverage systolic array structures to achieve high energy efficiency and performance. The inherent scalability, regularity, and data reuse patterns of systolic arrays make them particularly suitable for FPGA and ASIC implementations in embedded systems, enabling efficient deployment of machine learning models in resource-constrained environments. As AI applications continue to expand into diverse areas of life—from healthcare to autonomous systems—the role of systolic arrays as a high-performance computing substrate remains relevant and increasingly critical.

Although systolic arrays have long been recognized as efficient substrates for dense matrix computation, recent research highlights that they are still actively evolving to address energy and efficiency challenges, especially for embedded and edge deployment. Seong et al. [2023] proposed a cycle-reduced diagonal systolic array that reroutes data diagonally, reducing execution cycles by over 30% without additional hardware overhead Seong et al. [2023]. Similarly, Sestito et al. [2025] introduced the Triangular Input Movement (TrIM) for convolutional networks, demonstrating that carey chosen data-flows can cut memory accesses by an order of magnitude compared to conventional mappings. These works emphasize that optimizing systolic arrays is not only about arithmetic throughput but also about minimizing when and how data move. At the same time, the trend towards flexible and adaptive systolic architectures is clear. For example, the Precision and Dimension-Variable Flexible Systolic Array (PD-FSA) adapts its compute cycles to layer precision and size,

achieving a nearly 3× cycle reduction in some workloads Seong et al. [2023]. The Liu et al. [2020] with their Systolic Tensor Array (STA) further shows how structured sparsity (DBB) can save power and area by reducing unnecessary multiplications, though it is restricted to INT8 inference [4]. Such designs confirm that adaptivity — in precision, sparsity, or data-flow — is a central strategy for power efficiency. Other works pursue energy reduction from different angles. Li et al. [2023] compared digital compute-in-memory (DCIM) with TPU-like systolic arrays and found that DCIM offers area and power savings, but struggles with latency scalability; their hybrid systolic DCIM achieved up to 47% higher compute efficiency Li et al. [2023]. In FPGAs, building parameterized systolic arrays through HDL generate constructs enables designs to scale in array size or partition workloads using divide-and-conquer methods Zunin and Romanova [2022]. These strategies underline that scalability and configurability are just as important as raw performance.

1.2 TangramFP

TangramFP Yao et al. [2024] introduced a new approach to mitigate the inefficiency of floating-point multiply-accumulate operations by identifying and skipping ineffectual partial products at runtime. Unlike format-level changes such as FP16, bfloat16, or INT8, TangramFP preserves IEEE-754 floating-point compliance while reducing dynamic power by selectively disabling parts of the multiplier that do not affect the final result. In its 16-bit implementation, TangramFP demonstrated mean ULP errors within FP16 bounds and achieved up to 36% savings in dynamic power.

The novelty of TangramFP lies in exploiting a fundamental property of floating-point addition: when a very small number is added to a much larger one, the smaller number is effectively discarded after exponent alignment. Conventional multipliers always compute the full product, even if much of it is later be shifted out and wasted. TangramFP anticipates this at runtime by segmenting the mantissas into partial products and activating only those segments that contribute meaningfully to the final result. This makes it possible to skip unnecessary multiplications and reduce power without significant accuracy loss.

TangramFP addresses a gap in the current design space: while most systolic arrays adopt INT8, bfloat16, or other reduced-precision formats to control energy, few architectures attempt to make full IEEE floating-point computation more efficient. This positions TangramFP uniquely as a power-aware, yet standards-compliant solution capable of delivering energy savings with near full FP precision.

Preliminary work extended TangramFP to 32-bit floating-point (FP32) and confirmed its potential in both software models and HDL prototypes. These studies established TangramFP as a promising building block for energy-efficient floating-point accelerators.

This thesis advances the concept by embedding TangramFP into a systolic array architecture. While previous optimizations targeted array topology, precision scaling, or sparsity exploitation, TangramFP introduces a complementary axis of efficiency: mode-selective computation inside the MAC datapath. The central question explored in this work is whether TangramFP's unit-level energy savings can scale to systolic arrays, resulting reduced power consumption at the array level while preserving numerical accuracy and FPGA efficiency.

This thesis integrates the TangramFP in 16/32-bit floating-point mode-selective MAC unit into a systolic array. By embedding approximation-aware MACs into a parallel architecture, this work investigates whether the power savings at the unit

level scale to array level computation. Compared to the approaches discussed above —topology optimization, precision adaptation, sparsity exploitation, and CIM— the TangramFP systolic array introduces a new axis of efficiency: mode-selective computation embedded directly in the MAC datapath.

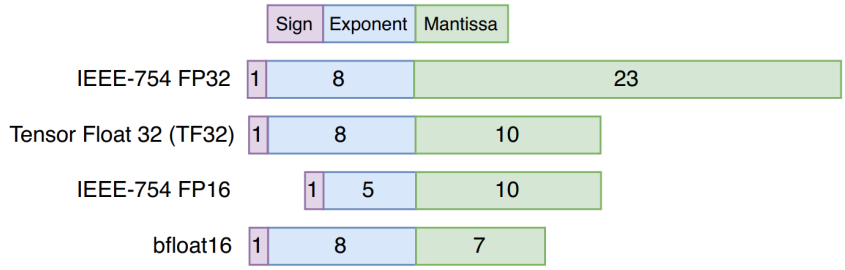


Figure 1.1: FP formats for DNN training.

1.3 TangramFP Mechanism

In the TangramFP, the floating-point representation of the IEEE-754 is adopted for 16-bit, and in this report the same standard is used for FP-32. The IEEE-754 standard stores the floating-point numbers in three sections, 1 bit for the sign of the number, the second section stores the exponent with a bias depending on the precision of the number to have the range from 0 up to double the maximum exponent that the FP-format covers. In the 16 bit the exponent ranges from 0 up to 30 in 5 bits length with a bias value of 15 so the FP16 covers the exponent range from -14 to 15. In addition to special cases, first when the exponent is all 1s meaning the number either NaN or infinity based on the state of the mantissa, second when the exponent is 0 the exponent is interpreted as -14 with no hidden significant bit in the mantissa. The mantissa of the IEEE754 represents the fraction of the floating point number after being put in the binary form 1.xxx where the whole number 1 is removed and considered as significant hidden bit. In FP32 the exponent bit width is 8, giving an exponent range from -126 up to 127 with subnormal values with exponent -126. Subnormal numbers in the standard IEEE-754 represents a class of very small numbers at the lowest exponent that a specific precision can represent, it is represented by setting the exponent to zero which means that the exponent is the lowest value and the hidden significant bit is considered as zero not 1, the lowest exponent for 16-bit is -14 and for 32-bit is -126. In Tangram the choice of IEEE-754 format gives a relatively long bit width for the mantissa, which gives an opportunity to utilize the tangram process to increase the precision in the approximation of the multiplication while minimizing the power consumption.

Yao et al. [2024] The main idea of tangram is to divide the mantissa of the two multiplication operands into smaller parts. $X * Y$ where X is segmented into $1 : A : B$ and Y into $1 : C : D$ where the length of A, C is p and the length of B, D is q and $U = (A : B), V = (C : D)$, then the multiplication operation is performed as

$2^{2(p+q)} + (U + V) * 2^{p+q} + AC * 2^{2q} + (AD + BC) * 2^{q} + BD$

Each term in the equation above corresponds to a mode of operation that represents an expected alignment shift that will affect the term or part of it. When the exponent difference between the multiplication results and the added value of the sum is expected to be higher than 0 but less than the first threshold, then the B*D

term is discarded and its multiplication is skipped, the mode of operation is called Skip_BD, and when the difference is between the first threshold and the second, B*D and $(A * D + B * C)$ are skipped with the mode AC_only. Whenever the difference is greater than threshold 2 then the entire multiplication is skipped as the result will likely yield a small number when added to the sum it results in the sum itself. TangramFP proposes different forms of segmentation of the mantissa with levels for modes of operation for the approximation.

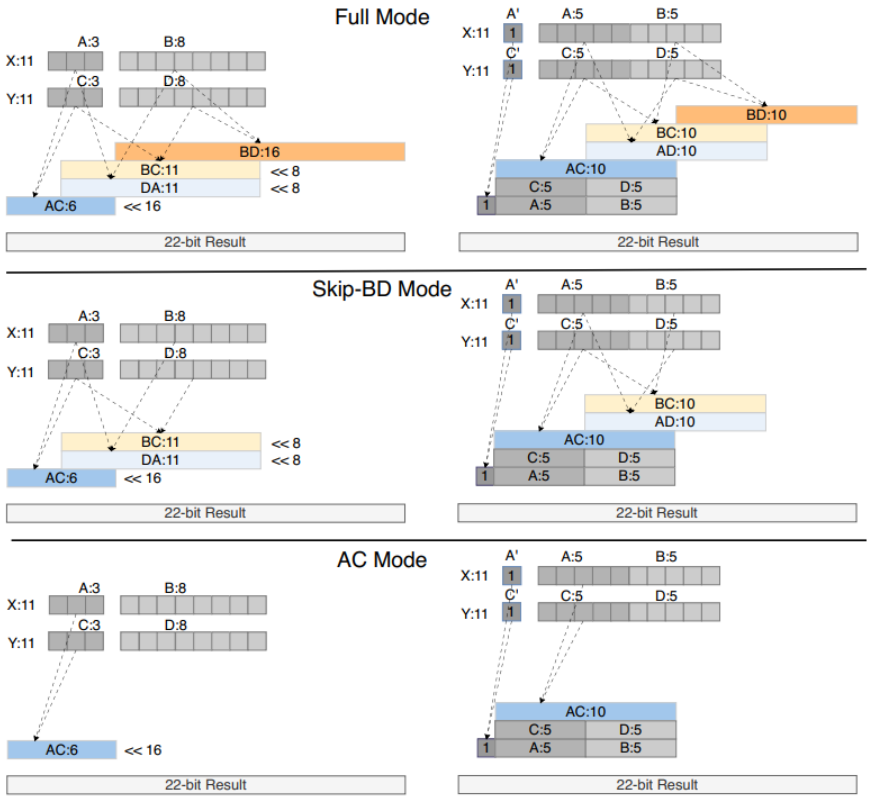


Figure 1.2: 3:8 and 1:5:5 splits featuring three modes of operation: i) Full Mode; ii) SkipBD Mode; iii) AC Mode Yao et al. [2024]

In TangramFP-32 the same mechanism is implemented with dividing the mantissa into two parts in addition to the hidden significant 1 with p = 12 and q = 11 as a close extension to the split 1:5:5 in the 16 bit Tangram.

As illustrated in both figures Fig. 1.3 and Fig. 1.2, the multiplication is divided into partial products where each term of the equation is multiplied selectively and separately in an independent Dadda reduction matrix, and then these partial products are added to form the final multiplication result. When in the operation $a * b + c$ where a or b is a subnormal number, the multiplication result will likely depend on the other input, so the subnormal number can be approximated to be the lowest normal number in the floating-point standard by replacing the zero exponent with 1 representing the lowest exponent. Then the mantissa is replaced by zero, meaning that the old mantissa is rounded up. This conversion adds up the hidden significant 1, and then the operation mode is determined according to the difference between the $a * b$ exponent and the c exponent. On the other hand, if c is the subnormal number, then it will be converted to the lowest normal number in the same way as above, but the mode of multiplication will be the Full mode, as the result totally

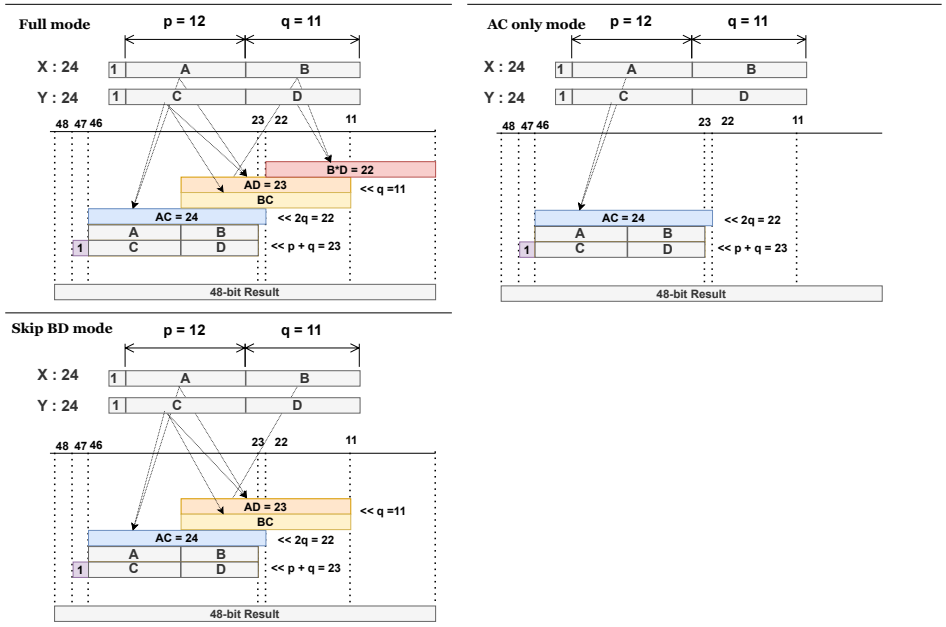


Figure 1.3: Addition stage of the first three modes

depends on multiplication $a * b$.

When the exponent of any of the inputs is all ones, this indicates that either the number is NaN or infinity; in any way this means that the whole operation will result in a NaN/infinity value in that case the whole operation multiplication and addition is skipped and the NaN directed to the output, similarly if the exponent of the multiplication is exceeding the maximum exponent covered by the standard FP such as above 254 in FP32 that means the final result will be infinity. In this case, the mode Skip will be selected, the addition is avoided, and the MAC unit will return infinity as output.

Chapter 2

Methodology

In this chapter, we describe the methodology followed to design, implement, and evaluate the integration of TangramFP into a systolic array architecture to accelerate AI inference models. The process covers the stages of architectural reasoning, Register Level Logic (RTL) prototyping, verification, system integration, hardware synthesis, and performance evaluation. A bottom-up design flow was adopted, starting from the smallest hardware unit and progressively composing higher-level modules until reaching the complete design. This approach allowed each sub-module to be verified independently before integration, ensuring correctness and focusing optimization efforts on the unit currently under development. While this method emphasizes localized optimization, it also leaves room for re-optimizing sub-modules later if integration or system-level evaluation reveals further improvement opportunities.

2.1 Development Flow

The overall development flow followed a structured sequence of design and verification steps.

The process began with the architecture specification, where the functionality of systolic array for matrix multiplication was formally defined. At this stage, we described the role of the processing elements (PEs), the datapath to stream and accumulate data, and the control mechanisms that govern synchronization. Defining these aspects provided a blueprint for the subsequent design stages.

Following the specification, the next step is RTL prototyping. Each architectural unit was implemented in VHDL using parameterization with generic blocks. This ensured reconfigurability at compile time, allowing the same design framework to scale to different matrix sizes or adapt to FPGA resource constraints. RTL prototyping provided the first hardware-level realization of the system.

The next stage involved functional simulation. Verification began with the lowest-level modules (Dadda adders) and progressed upward, consistent with the bottom-up methodology. Simulation was used to validate datapath correctness, control flow, and inter-module communication, ensuring that each unit behaved according to specification before moving to synthesis. After verification, synthesis and implementation were performed on the top-level systolic array module. This step mapped the design to the target FPGA fabric, leading to timing analysis, power estimation, and resource utilization.

To confirm correctness under realistic hardware constraints, post-implementation timing and/or functional simulations were performed at the top level of the systolic

design. The simulation validated the functional behavior while accounting for mapped resources, propagation delays, and FPGA-specific optimizations.

After that, there were two paths available, first testing and reporting the design in the post implementation stage for power, timing, and resource utilization; this only requires Vivado tools. The second is to integrate the design into a system and deploy it on a FPGA board to run in real time. Both paths are important and complement each other, hence both have been applied.

For the real time run the design was then prepared for system-level integration. This required exploring communication mechanisms and suitable architectures to connect the systolic array as an IP accelerator within a larger computing system. SoC architecture with AXI-DMA protocol is chosen because it is easy to develop and it requires the least external resources i.e. require only same FPGA board.

Alongside the hardware integration, software development was performed to enable interaction with the accelerator. A test application was created to serve as a simple AI agent, streaming matrices to the hardware, triggering computation, and retrieving results.

Finally, the design was subjected to end-to-end testing. A bitstream was generated, deployed on the FPGA board, and executed under workloads in real time. During this phase, examination of the PS-PL transaction through real time debug is conducted. On the other side, tests and reports were conducted for resource utilization and power consumption estimation. The reporting is conducted for different sizes of the systolic array 2×2, 4×4, 8×8 for TangramFP MAC unit 16 bit and 32 bits then comparative analysis is carried out.

2.2 Hardware Design Methodology

2.2.1 System Architecture Definition

The design scope was restricted to inference-only matrix multiplication, where weights remain static and inputs are streamed during execution. Each PE in the systolic array contained a TangramFP based MAC unit that supports four operating modes: Full, Skip_BD, AC_only, and Skip.

The data flow followed a classic systolic structure: for matrix multiplication WxA, a N×N grid where the values of matrix A were streamed column-wise, while the static weight matrix W was held static row-wise. Partial sums propagated through the array using carry-save accumulation stages. This architecture enabled efficient reuse of weights, minimized data movement, and provided pipeline-friendly computation.

2.2.2 Integer–MAC Systolic Array Prototype

Before integrating TangramFP, an integer-based systolic array prototype was developed as a proof of concept. The prototype featured a flattened input structure, a simple streaming logic to feed data to the systolic array, and the instantiation of PEs containing integer MAC units. Behavioral simulation validated the correctness of data streaming, accumulation, and output consistency, ensuring that the systolic structure operated as intended.

2.3 TangramFP Integration

2.3.1 Single-Cycle MAC Redesign

The integration of TangramFP required redesigning the PE MAC datapath for single-cycle execution. First, mode inference logic (responsible for exponent comparison) was extracted from the MAC core to simplify its critical path. The redesigned MAC fused multiplier and adder stages into a single pipeline-balanced datapath. The optimization aimed to meet frequency constraints while preserving functional correctness. To confirm these improvements, post-synthesis gate-level simulations were performed, validating numerical correctness for the one-cycle datapath.

2.3.2 Mode-Selection Unit

A dedicated mode-selection unit was introduced to configure the TangramFP MACs at runtime. This unit performed exponent-based estimation, comparing the sums of products against the maximum exponent to determine the appropriate MAC mode. The system dynamically generated per-PE configurations, balancing accuracy and performance trade-offs. Careful consideration was given to the hardware cost of the mode-selection logic, as it had a direct impact on scalability.

2.4 System Integration in Vivado

To make the systolic array usable as part of a larger system, it was packaged as a custom IP block in Xilinx Vivado. This required creating suitable interfaces and communication mechanisms between the accelerator and the system memory. Integration was performed through a Vivado block design, where the systolic array IP was connected to additional modules such as DDR memory controllers and interconnect logic. After integration, bitstream generation was completed, including synthesis, placement, routing, and export of the hardware description for deployment on the FPGA platform.

2.5 Software Flow

The software workflow was designed to complement the hardware accelerator. The first step was identifying a suitable FPGA development board and configuring its board support package. Next, driver development was carried out to enable communication between the processor and the systolic array IP through the chosen interface.

On top of this, a lightweight test application was implemented to act as an AI Agent. The application streamed input matrices to the hardware, triggered computation, and retrieved results. These results were then compared against a reference software model to confirm end-to-end correctness.

2.6 Real time Debugging

During the development of the SoC block design, a System Integrated Logic Analyzer (ILA) IP is included. The ILA allows to capture certain signals and debug in real time with break points in the software in Vitis and waveform display in Vivado hardware manager. This feature gives a chance to see exact signal values during run, therefore

transaction between Direct Memory Access controller (DMA) and the systolic array are propped to the ILA in addition to control signals and some inner signals in the systolic array. The capture of the input/out transaction in the systolic array gives a clear picture about the total time the systolic array spent processing the data. Also, it helps in debugging for the correctness of the output result, as when the design is a black box, it becomes difficult to determine if the error is coming from the hardware design or the software side.

2.7 Tools and Equipment

The methodology relied on industry-standard tools and platforms. The RTL design and simulation were performed in VHDL. The FPGA implementation used the Xilinx Vivado/Vitis toolchain, providing synthesis, simulation, IP packaging, and software integration. The deployment targeted suitable Xilinx FPGA boards, selected on the basis of availability and resource compatibility with the systolic array design.

2.8 Evaluation

In this stage . Resource utilization metrics such as LUTs, FFs, BRAMs, and DSPs were recorded to quantify the hardware cost of the design. The timing analysis determined the maximum operating frequency and latency for different matrix sizes in addition to identifying critical paths and timing violations such as hold and setup violations. Finally, power consumption was estimated using Vivado Power analysis tools for static estimation and switch activity based estimation to realistically estimate dynamic power consumption. Together, these evaluations provided insight into the expected power cost of the systolic array in contrast to resource utilization.

Chapter 3

Design

3.1 Systolic Array Architecture

3.1.1 Designing the Systolic Array

There are many architectures to implement the systolic array depending on the intended functionality and the hardware constraints in terms of communication bandwidth and the internal hardware resources. Since in this work the focus is on the implementation of TangramFP MAC unit in a systolic array to accelerate a neural network, then three points will be considered here; first, it will be satisfactory for the scope to assume that the systolic array is intended for inference and not for training purposes. The difference is that in inference operations, the weight matrix does not change while the input to the inference module is new every time. Second, it is not necessary to consider non square matrix multiplications and other optimization perspectives. Third, the TangramFP is designed in a way to receive accumulated results and then perform multiplication and add the result to the received accumulations and pass over the new accumulation. This entails a pipelined accumulation. Consequently, the architecture of the systolic array will be set as processing element (PE) units in the $N \times N$ form with fixed W input and streamed input form matrix A on a column by column basis, where each clock cycle a new column from A is entered and each column in the $N \times N$ array passes on these columns horizontally. Each PE calculates the partial accumulated result and then passes these results down in the grid. Finally, the result is taken from the last row.

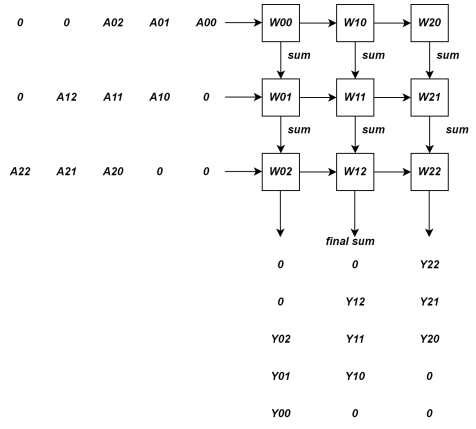


Figure 3.1: Systolic array

3.1.2 Building Basic Systolic Array with Integer MAC

As a starting point, we design a functioning systolic array according to the description above. The dynamic of the array is the focus regardless of the type of MAC unit used and intended application. The design directive here will be to have the systolic design scalable to any number of rows and columns. However, here the number of columns equals the number of rows. the architecture implementation will be split into major parts:

1. Input/Output interface module.
2. Input data streaming.
3. Data movement inside the array.
4. Buffering the output.

The PE for this design will be a simple multiply-add unit that receives three inputs A, B, C to perform $A^*B + C$ and give back sum. All inputs will be of the same size and unsigned integers. In this stage, no synthesis is required; thus, the functionality of the array is measured in behavioral simulation to test the data flow and verify its readiness to the next step in design.

3.1.3 Module Interface

In this stage, interfacing is not important, so a port of flat matrices is enough with all data communication is intended to be with a testbench.

Buffering Input and Output

Currently, after receiving data from external sources, the matrices A and W are kept in a $N \times N$ register file to prepare data streaming to the systolic array. For W there is no streaming; instead, there are $N \times N$ lines to connect each register to a corresponding PE in the array. Similarly, the result is buffered in an $N \times N$ register file to prepare it for sending to the external receiver. In this stage we can just flatten the result to one vector of length $N * N * number\_width$.

3.1.4 Streaming Input Data A

From Fig. 3.1 A is streamed in slice by slice each clock cycle, each slice represents a complete column in A. We notice that not all elements of the A column arrive at the same time to the array, with each element in the column being delayed one clock cycle from the element above it, so that when an element in the column reaches its PE it synchronizes with the computed partial product from the above PE.

One way to achieve that, at least in this stage, is by having a counter that starts at the beginning of the operation and resets after the total time needed from the start until the complete matrix is produced. Fig. 3.2 shows how the delaying logic is implemented with which for each position in column A_input an element is taken from A_input_matrix that corresponds to the required delay; in other words, instead of delaying through buffering, the delay is carried out by calculating the index to select a previous element. The index calculation of a matrix (n,m) is chosen as $(i, counter - i)$ where i is the position of the element in the A_input register and by subtracting $counter - i$ we can select the element as if it was delayed. In this

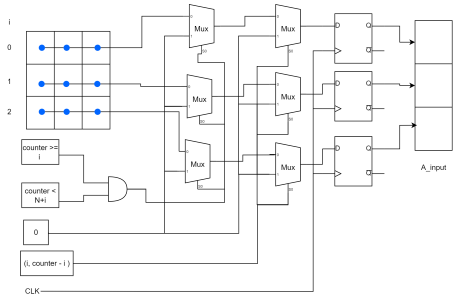


Figure 3.2: Counter-select based feeding A input to the systolic array

approach, the index selection must be guarded so that no negative index is produced or exceeds the full size of the matrix by restricting the selection to be when the counter is greater than or equal to i and the counter is less than $N + i$, N being the size of one dimension of the matrix.

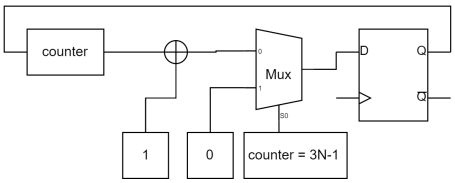


Figure 3.3: Counter of the entire operation

When no element is selected, in case the first element in the column is delayed or the row is fed completely, the default propagated value is 0. This structure is sufficient for testing the systolic array, and we can revise it later when this step is completed successfully.

3.1.5 Data Movement Inside the Array

In this stage of development, the goal is to build and verify this part of the design. The movement of data inside the array follows the same depiction in Fig. 3.1, as stated earlier, however, there are some points in implementing the structure and functionality in VHDL/Vivado.

VHDL Implementation

To implement the systolic array structure described in Fig. 3.1 we can initially determine the dimension of the two matrices and then instantiate the required PE units and create the needed registers and connectors. However, we lose flexibility to change the size of the array and we would need to build a new systolic array each time we try a new matrix size. Alternatively, we can use the generate method in VHDL with two for loops to instantiate a PE at each i,j position. In this approach due diligence is required to manage the interconnection among the PEs input output from the generated grid.

The generate construct allows for definition of signals inside its scope alongside any process combinational or clocked. Having that, we cannot identify a specific

Listing 3.1: Systolic array core

```
1     gen_rows: for i in 0 to N-1 generate
2         gen_cols: for j in 0 to N-1 generate
3             signal sum_in, sum_out : std_logic_vector(Pre_out-1 downto 0);
4             signal A_in , B_in : std_logic_vector(Pre_in-1 downto 0);
5         begin
6            -- Define Inputs for  First  Row/Column
7             A_in <= A_input(i) when j = 0 else A_pipe(i,j-1);
8             B_in <= B_matrix(j, i);
9             sum_in <= (others => '0') when (i = 0) else C_matrix(i-1, j);
10            C_matrix(i, j) <= sum_out;
11            sum_output(j) <= C_matrix(N-1, j);
12            pipe : if j <N-1 generate
13                process(clk) begin
14                if rising_edge(clk) then
15                    if n_rst = '0' then
16                        A_pipe(i,j) <= (others => '0');
17                    else
18                        A_pipe(i,j) <= A_in ;
19                    end if;
20                end if;
21                end process;
22            end generate pipe;
23           PE: MAC
24                port map (
25                    clk   => clk,
26                    n_rst => n_rst,
27                    a     => A_in,
28                    b     => B_in,
29                    c     => sum_in,
30                    sumout => sum_out
31                );
32        end generate;
33    end generate;
```

signal from all generated signals to interconnect them in the way described earlier. However, we can create a matrix $N \times N$ for each interconnection and have each cell of that matrix connect to a signal inside the generate block. At each clock cycle, the partial results from a row in the array of PEs are passed down to the next row; this functionality does not need a separate clock process in the scope of generate block, because these PEs produce results at the clock event. Inside the generate block, each PE output is assigned to sumout; a signal defined inside the inner generate loop to create it for every PE in the array. this sumout in turn will be passed to a corresponding C_matrix matrix cell while the input sum is taken from C_matrix. The scope of C_matrix is the entity architecture where it is defined, this allows it to be visible and hand over data in its last row to an output register vector sum_output. On the other hand, instead of passing the input A into a PE then task the PE to pass along the input in the next clock cycle, a feature that would require adding this functionality inside any MAC unit intended to be used as PE in the array, it is better to have this functionality of the systolic array inside the generate block scope. Similarly to C_matrix, A_pipe is an array of registers, it is possible to have the same build up as C_matrix, but the dimension needed for A_pipe is $N \times N - 1$

and having a square array of registers would waste resources and energy. But we cannot easily put this condition inside the generate block and expect it to have the correct behavior. It requires its own generate block inside the nested generate block, as shown in Listing 3.1.

3.1.6 Streaming-out the Output

The output buffering is designed the same way as feeding input to the systolic array. The difference here is that the process is reversed, which requires demultiplexing to distribute the result into different positions in the result array of registers. Also, in this time the data are intended to be delayed, but they are already produced in a stair shaped delay. Nevertheless, the calculation of the indices is the same, given that a constant $(N+1)$ plus any delay in the input is subtracted from the counter value to rest it to zero. $N+1$ is a calculated time until the first output is available. In addition, the same conditions are needed to avoid negative or out of bound indices with an additional condition counter that has reached $N+3$ to start registering output.

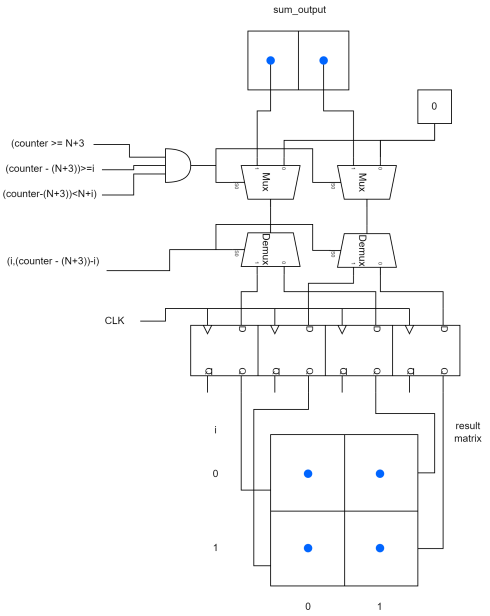


Figure 3.4: Counter-select based buffering output

3.2 Integrating TangramFP MAC

After verifying the functionality of the systolic array core, the next step is to integrate the TangramFP MAC unit in place of the PE used earlier. The TangramFP used here was developed as an experimental unit in a previous project to test energy reduction as a result of different multiplication modes. So, it had mode inference logic embedded inside which worked for comparing only the exponents of the input sum and the exponent of the addition of the two multiplication operands. To integrate it here it requires first to separate the mode inference from the MAC, second, the previous MAC operated on two clock cycle period first clock to produce multiplication and the second to perform addition part, this setting was intended for high

frequency clock operation. In this project, it is better to have a single-cycle MAC, thus re-optimizing and re-organizing the MAC unit is required.

3.2.1 Mode Generation for TangramFP Systolic Array

The TangramFP MAC unit operates on multiple multiplication modes; Full, Skip_BD, AC_only, and Skip. These modes are determined based on the amount of exponent alignment shift that occurs when two floating point numbers are added with different exponents. The aliment shift is performed on the smaller number, and if the smaller number is the result of the multiplication of the $A * B + C$ operation, then part of the precision of the mantissa is lost, and therefore the TangramFP provides an estimation of precision loss and avoids the multiplication of parts that will produce that part. This means that the MAC unit receives the mode from an external source. In matrix multiplication, accumulation is performed on the dot product of each row of the multiplier and each column of the multiplicand matrix. For one row and one column dot product, the amount of alignment shift is the difference between the final sum exponent and each exponent of the product of two corresponding elements, but the final sum exponent will result from normalizing the result after all multiplications and additions are performed. If the operation is performed progressively, the exponent of the accumulation currently completed at a certain time can be used to estimate the shift alignment. However, this gives less accurate estimate of the shift alignment for all multiplications in the dot product, suppose that the temporary accumulated sum exponent is small, then upon comparison the estimation would assign a higher precision multiplication, as the operation progresses, one or more multiplications increase the exponent of the accumulated sum, which means that some of the multiplications that were already performed could have been performed with more reduction, for example, a multiplication performed with full mode and could have been performed with Skip resulting in decrease effectiveness of the mode estimation.

The addition does not affect the exponent of the result significantly as the result will mostly keep the exponent of the bigger number, conversely, the multiplication always gives an equal or greater than the sum of the exponents of the two numbers, meaning that the biggest exponent in the A x B vector dominates and the final result exponent either equals to or differs slightly from it. This can be utilized so that the multiplication exponents in the dot products are compared, then a maximum exponent is selected as an estimate of the accumulation exponent, then used to compare with the exponent of each multiplication to estimate the shift alignment and decide accordingly the mode of multiplication. Based on the last method of mode selection, it does not include the mantissa nor the sign part, rather mainly depends only on the exponent part of the floating point number, which means less cost computationally and in area utilization. In this section, we will explore the possible options for the design of the mode selection unit.

The mode selection operation contains multiple steps: first select a pair of row and column and extract exponents, second perform exponent addition, third determine what is the maximum exponent of the added exponents, fourth for each multiplication exponent subtract it from the maximum exponent, and finally, determine the mode according to the difference. One straightforward way is to calculate the modes for all multiplications of all rows and columns of matrix A and B that can be parallelized by using N^3 adders to produce all multiplication exponents, $N^2 * (N - 1)$ comparators, in the following step N^3 subtractors, leading to a final

step N^3 multiplexers to determine the mode for all needed multiplication operations.

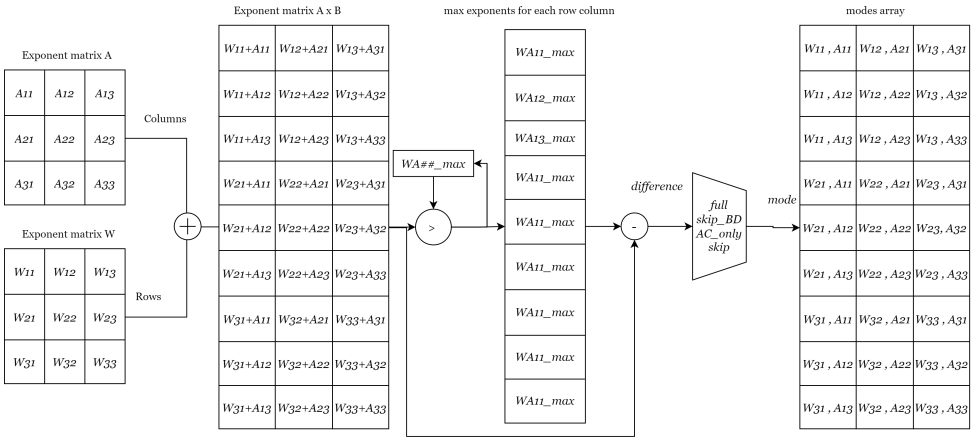


Figure 3.5: mode selection unit for the entire matrix multiplication

The Fig. 3.5 describes the mode unit for all operations. The whole process is in combinational logic and for timing purposes the final result can be clocked into a register. Despite the simplicity in design, this approach is too expensive from an area utilization point of view, especially if the design needed to be deployed to a small FPGA board. Additionally, in this work, the systolic array works in a pipelined fashion where the weights are loaded to each PE and the input is fed sequentially, each generated mode is intended for a specific PE; this means that not all the modes are needed at the same time, and after directing a mode to its target it is no longer needed. Accordingly, then there is no point on building a large modes unit and thus can be reduced to a smaller unit that produces only the needed mode at the time.

An alternative is to generate a single column of modes for one column of PEs each clock cycle. A single column of PEs would have a row of the weight matrix, i.e. the multiplier matrix. Although this idea sounds appealing, it has multiple issues; when this process is applied to the systolic array, we find that the required number of columns of the modes at each clock cycle increases to N and then decreases, as we can see in the diagram Fig. 3.6, which means that the full structure of parallel generation of modes is unavoidable. Additionally, between the first and the last clock cycles more than one PE column expects mode input resulting in having a direct connection to each node in the systolic array, eventually we will end up with the same structure in Fig. 3.5 even though at some points it generates a single column modes.

A less parsimonious way about resource utilization comes from considering the systolic array as a box that receives at every clock cycle a new input and propagates it inside to each node in a row. Each input represents a column vector from the multiplicand matrix, this vector then propagates horizontally to each column of PEs. This signifies that each column traverses all the rows of the multiplier matrix. Considering that at any operation on a PE in the array requires its own mode, then if we produce all the required modes for a single column vector of the multiplicand matrix with all the rows in the multiplier matrix, we will achieve; first a uniformity of mode generation to be at any given clock cycle an $N \times N$ array of modes is generated and it would suffice for the complete journey of the input vector inside the systolic array, second the architecture will reduce significantly from N^3 registers,

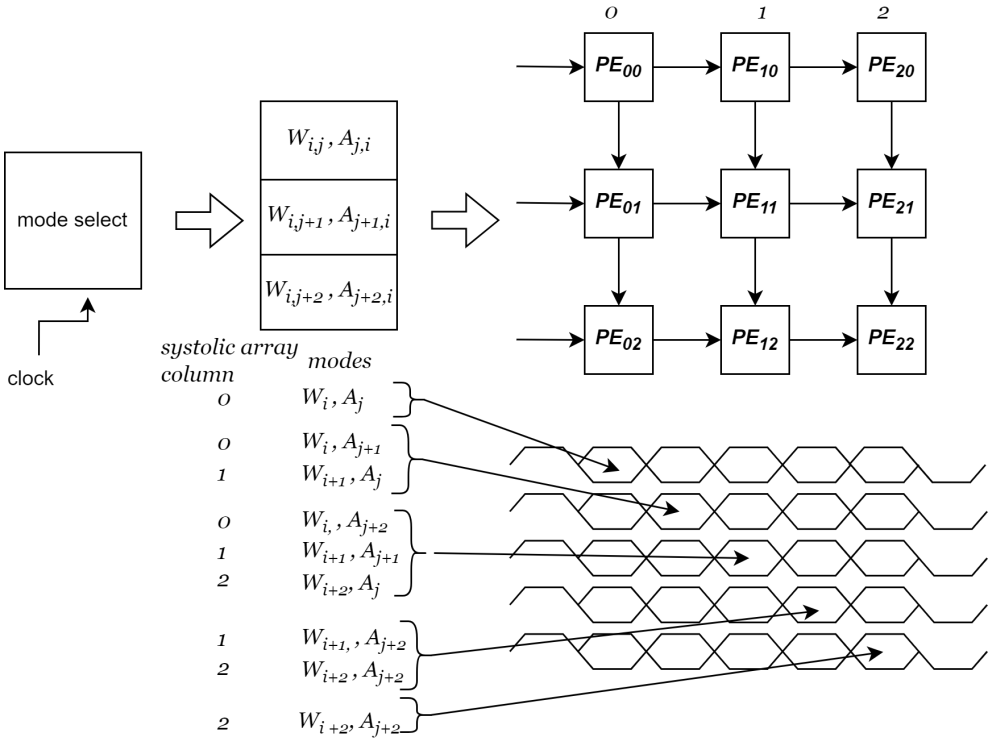


Figure 3.6: Generating one column of modes per clock cycle

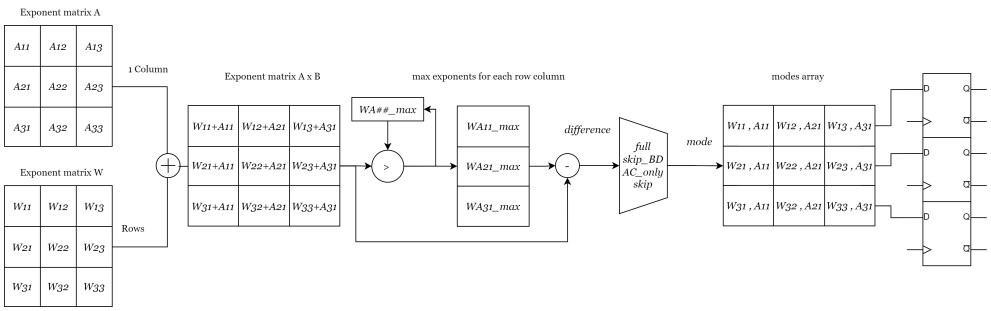


Figure 3.7: mode selection unit for the for 1 column of A and all rows of W

adders, subtractors and multiplexers into N^2 and $N * (N - 1)$ comparators. The Fig. 3.7 describes this organization of the mode unit, we will adopt it in this work because it is compact and satisfies the requirements for the systolic array.

The mode unit in Fig. 3.7 generates at each clock cycle an array of mode with size $N \times N$ while the input port in the systolic array takes a column not an array. In consideration of the form of mode delivery to the doorstep of the systolic array along with directing modes on time to their respective PEs, there is no limitation on organizations to achieve that. We will try to think of a simple organization to input, delay, and propagate the modes inside the systolic array.

Triangular Delay on Mode Input

We can first concatenate all the modes in each row of the mode array into single vector of elements of dimension $N \times 1$. This approach is convenient with VHDL im-

plementation especially in using generate method to have compile reconfigurability to change the systolic array size. Also, this structure enables the creation of variable length shift registers. The reason to have a variable length registers is that not all the delays are equal, rather the increase linearly based on the register position in the column, and having all of them with equal length would misallocate half of the built resources.

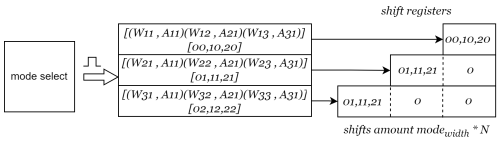


Figure 3.8: mode input triangular delay

Propagation of Mode Inside the Systolic Array

On the contrary to matrix input, modes are not reused by other PEs which makes the pipeline mechanism used for matrix multiplicand does not apply, yet at each hop an input takes to the next PE, the corresponding mode must be available at that stage on time otherwise none of the multiplications would perform according to the correct mode. Since the selected mode unit produces a $N \times 1$ array of modes where each cell is directed to a row input port of the systolic array, we could use a shift register with length $N * N * mode_{width}$ similar to the input pipeline we described earlier; however, we already established that not all modes are needed at any level and after being used there will be no purpose for them; therefore, this will result in significant waste of resources and power. We can alternatively create in each row in the systolic array a chain of registers acting as a shift register, but the difference is that each shift to the right takes only the modes that will be needed forward and discards the mode that was already taken by a corresponding PE as shown in Fig. 3.9. Having All modes in the row packed together in a single victor also makes the implementation convenient and reconfigurable for any size of the systolic array.

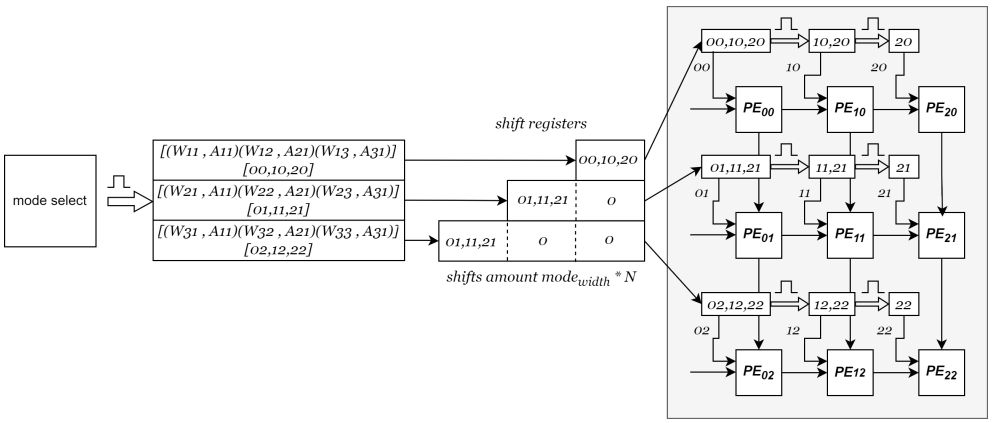


Figure 3.9: Propagation of mode inside the systolic array

3.2.2 Adaptation of TangramFP

This project relies on the implementation in VHDL of a TangramFP 32 bit based on the TangramFP 16 work in Yao et al. [2024]. However, the implemented MAC operated on double clock cycles for multiplication and addition, the design was very specific for a certain ratio of the cut where in mantissa split to $1 : A : B$ with lengths $1 : P : Q$, P has to be greater than Q limiting choices of cut feature in TangramFP settings, also with using different precisions the HDL code has to be modified so that the dimensions of partial products align in intermediate additions. Additionally, the mode selection is computed inside the MAC based on a single pair of exponents from the received accumulation and the addition of the multiplication operands. In this section, we explain the adaptations and optimizations carried out in the MAC implementation to fit the systolic array operation.

Double-cycle to a Single-cycle

In order to reduce MAC cycles into a single cycle operation we first need to identify the critical path in the MAC design before removing the intermediate register. In normal operation when there is no NaN input nor the multiplication or addition is deactivated, the multiplication operands go through preparation step, then through Dadda multipliers and Carry-Save Adders (CSA), then the result is normalized and forwarded to the addition step. the multiplier unit needs also a mode for the operation, despite being computed in parallel but if it is slower than the preparation it could add to the delay. The CSA adders units contain at the last step a carry propagation adder with delay proportion to the sum width, adding to that normalization includes counting leading zeros in the mantissa. As a first step, we can remove the normalization and full number construction from the multiplier and forward the raw mantissa multiplication result to the adder and modify the normalization at the adder to account for this change. At this point the MAC multiply-Accumulate becomes Fused but not completely, since the multiplier has multiple CSA adder units with carry propagation. A real fully fused operation will require redesigning the multiplier, and at this point we think this will go beyond the scope of the project, hence we try this reduction then if its necessary for timing, we will come back to it later.

Mode as Input

The mode unit consumes a significant portion of the MAC utilization area, energy, and computation time; removing it makes the MAC a light weight, as well as it is necessary for mode estimation to be computed in the matrix level, as we discussed earlier.

Optimizations in the Multiplier's Adders

In the TangramFP multiplier with a split ratio $1 : P : Q$ initially $P > Q$, the two mantissas $(X, Y) : 1 : A : B$ and $1 : C : D$ and $U = (A : B)$ and $V = (C : D)$, the multiplication is calculated as follows:

$$2^{2(p+q)} + (U + V) * 2^{p+q} + AC * 2^{2q} + (AD + BC) * 2^{q} + BD$$

The term $(AD + BC)$ is computed in one of the CSA adders with 4 input vectors (CSA4i) and AC and $(U + V)$ are passed to another instant of the CSA4i. Since

these products are not of the same weight, BD being the least significant part is concatenated with the summation of $AC + U + V$, and then the most significant bit $2^{2(p+q)}$ is padded with zeros to align with the other partial results. At the final sum these three components are fed to the CSA adder with 3 input vectors, the last adder would contain a large amount of zeros, it would improve the timing if we eliminate the propagation of zero carry along and confine it to where $(AD + BC)$ align with $AC + U + V$, and with $2^{2(p+q)}$ and change the CSA to 2 inputs as shown in Fig. 3.10.

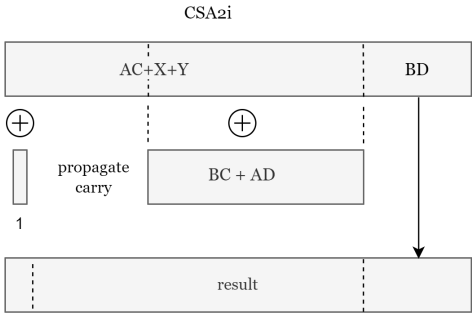


Figure 3.10: simplifying Carry-save 3 input adder

Reconfigurability

The focal point in the TangramFP is that it is reconfigure at run time which Dadda multipliers are active based on the mode. we follow on that and equip the design of the MAC with other runtime reconfigurability, as well as compile time reconfigurability to widen the range of use without need to redesign the MAC and walk through the verification workflow every time.

Run-time Reconfigurability

In addition to reconfigurable Dadda multipliers, there are edge cases where the whole multiplier is bypassed, such as when one of the operands being zero, then the entire multiplier logic is deactivated. Both the adder and the multiplier are deactivated when any of the three inputs is infinity, or NaN or the result of the multiplication would lead to an overflow beyond the target precision could represent. when any subunit of the MAC is bypassed, a corresponding flag is raised so that the final output in the MAC top module accounts for that special case. Other reconfigurability features are added inside the adder, alternating between addition or subtraction based on the difference between the sign of the addition operands. It seemed convenient to perform subtraction instead of adding compliments. Also, counting the leading zeros with Leading Zeros Counter (LZC) is activated only when the input sum is zero, and we need only to normalize the multiplied mantissa or the subtraction is active. The reconfiguration of addition/subtraction and activation of the LZC were present in the original implementation of the MAC, but were modified to account for the row input of the non-normalized mantissa.

Compile-time Reconfigurability

The compile-time reconfigurability in the MAC unit has two purposes and consequently targets different scopes of parameters inside the MAC. The first one, we

want to be able to change the operation precision of the floating point number so that whenever we need to test or downgrade the precision later in the study we do not face a great wall of re-adapt the MAC as will take the focus away from the main problem at hand reset the work at an earlier stage; additionally, it will open the options to reuse the design for further studies. The other reason is that TangramFP proposes multiple ways to split the mantissa and the cut ratio; when changing the cut length, it changes the alignment of intermediate products also, the precision of the floating point numbers affects that, whether the mantissa width is an odd number or even, or the cut makes $P = Q$, $P > Q$, and $Q > P$. Moreover, the split form if changed for example from $1 : P : Q$ to $P : Q$ the number of Dadda multipliers will change and CSA adders would change in input numbers and formation. Up to now we managed to universalize the cut ratio to be any number in the mantissa width range at compile time, because having it reconfigure at run time would mean alternative configuration has to be synthesized and currently we need minimal area for resource availability, in the case of more spacious FPGA we can just change that from using conditioned generate into enable signals. The split formation is important because it will open a wider window for utilizing TangramFP in other aspects such as reducing the Bandwidth of data traffic; however, we deferred that for later time when it is necessary as for now we are interested testing the current design, although the current design is closer to achieve that reconfigurability.

Floating Number Precision In order to be able to use the MAC for any Precision chosen, all the operations inside the MAC sub-modules are parameterized and passed on as generics. However, with one exception, because we were concerned about MAC timing, the LZC is specifically optimized for the least number of carry forwarding units in FPGA for FP16 and FP32. For total universal MAC for all precisions, it can be changed to a for loop or parallel universal LZC.

Cut, P, and Q As we can see in Fig. 3.11 the length of the input addition operands to CSA4i depends on whether P is greater or equal to Q or otherwise; additionally, in the case of $Q > P$, BD becomes longer and UV/PV is only shifted to the left by $(P + Q)$ while AC is shifted by $2Q$, which means that BD overlaps with the added value of them. In order not to add a fifth input to the CSA4i adder or a third input to the final csa2i, we can append the overlapping part to AC and the remaining of BD to the result of the previous adder when feeding to the CSA2i adder.

Dadda Multiplier Initially, the Dadda multiplier was designed to multiply signed numbers of equal length. Since its purpose here is to multiply mantissas, we do not need the sign related logic inside the Dadda, nor we need to have extra idle input for the sign, so these are removed. When changing P and Q and they are not equal, then either we append zeros to the shorter operand and thus we will have additional rows/columns in the Dadda reduction stages, which increase area and power even if with slight amount, it is undesirable and avoidable, as a result we adapted Dadda to receive different sizes of input.

3.3 Interfacing

The design of the interface was constructed after testing with behavioral simulation to determine the operational correctness of the systolic array. An interface is neces-

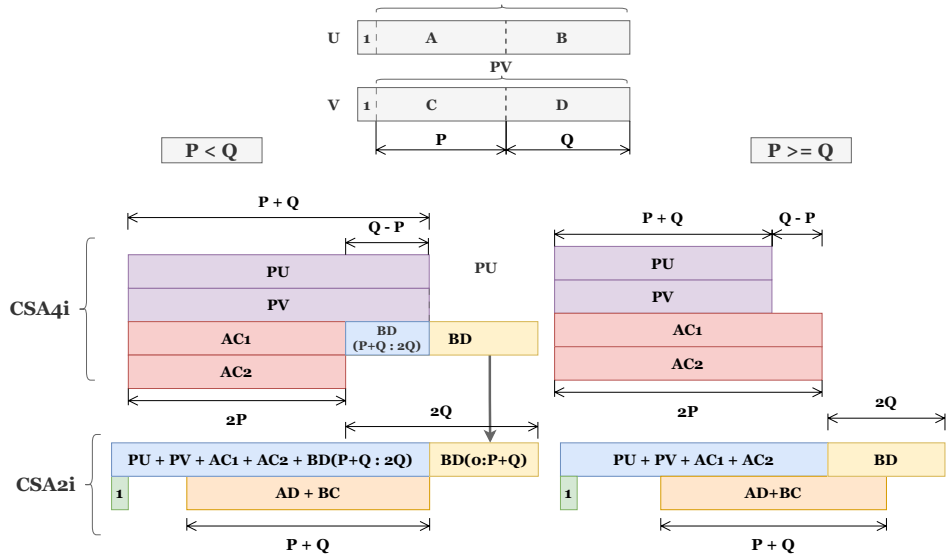


Figure 3.11: Configurations for cut,P, and Q cases

sary especially for the systolic array at this point receives two $N \times N$ matrices in parallel, which means $2 * N * N * precision$ port pins. This is a very high number of ports for a target device, and then moving to next step to test after synthesis and implementation will be impossible. It is possible to make an interface just for post-synthesis/implementation functional simulation; however, from a practical point of view, it is better to build an interface for a specific target so that testing is not repeated when the whole design is modified or embedded in a bigger system. We have also explored options on the types of system integration, the communication medium and protocol, and the FPGA target device. These explorations are not carried out in the preliminary design phase, but after the first synthesis attempt, and we will discuss it further in the implementation chapter.

3.4 Integrating into SoC

The main study aim is focused around the design and evaluation of TangramFP based systolic array, then building an SoC systems seems out of scope. However, in order to fully test the system, end-to-end testing is necessary to verify that the systolic array is performing as expected; furthermore, it will give insights about the design choices and trade-offs.

Since we have opted to have the size of the systolic array and its PE as configurable system, choosing the target device will not be problematic despite having a bigger device will provide more options in the systolic array design. The current available device is Zynq-7000 on Minized board, it is relatively small, especially our initial design choice for the systolic array is to store a complete weight matrix, so that the matrix multiplication will be carried out in a single send-receive operation. Another option was Kintex-7 on the Genesys-2 board; this device does not have an SoC onboard; but has a larger number of cells. It is possible to integrate a Microblaze soft core or build an interface to communicate data from scratch; for this stage, the Zynq-7000 will serve the purpose of system-level debugging.

Choosing to build with SoC makes the process easier and straightforward, espe-

cially for determining data communication. Vivado provides a block design methodology, and we will utilize it to build a system constituting of a Zynq-7000 Processing unit (PS) where the data preparation is handled, simulating an AI inference model and the wrapped systolic array which will be on the Programmable Logic (PL) side in the FPGA board. For communication between PS and PL, the AXI streaming protocol with Direct Memory Access controller (DMA) is the straightforward choice, as shown in the diagram below, the PS is connected to the PL (the systolic array), the PS will send two matrices and receives their multiplication result. Moreover, with the SoC system, the option of creating a communication channel with a PC device becomes simple.

The block design also contains a system ILA IP block to capture the data transferred and the control signals.

The organization of the interface is designed as an encapsulating module for the systolic array, as shown in Fig. 3.12. It consists of a state machine that receives data, enabling the systolic array, buffering the result, and then sending back the result over AXI. This structure of the system is intended for the first implementation and testing at the system level, after which the design will be revised according to the findings of the implementation and operation of the system.

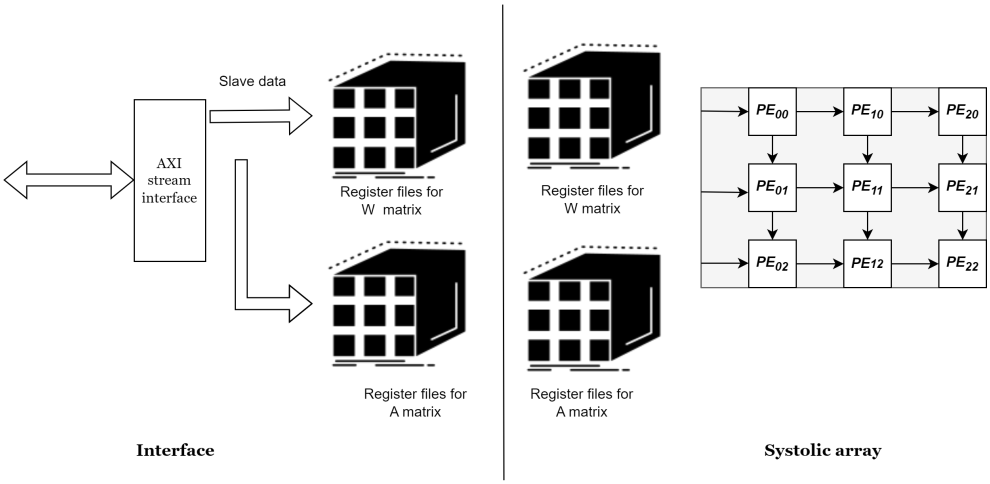


Figure 3.12: Abstract organization of the systolic array interface

3.5 Suggested Optimizations for Different Applications

We want to point out after applying the modification that the initial decision to have the weight matrix as fixed inside the systolic array needs to be reconsidered. Because the initial design assumed that the systolic array would perform full matrix multiplication at once, however, after the reduction in size, the full matrix multiplication with the usual size in AI inference models would have to be in stages and tiles. At this level, the only change to account for that is to duplicate the same mechanism applied for input A and sequentialize the state machine to always receive W first and then A. We will not pursue that at this level, as it is not the core focus of the work and leave it for future work. Additionally, we would expect that if tile multiplication is set, the effective mode deactivation of multipliers will slightly decrease because the mode selection is computed based on the current transmitted part of the whole matrix.

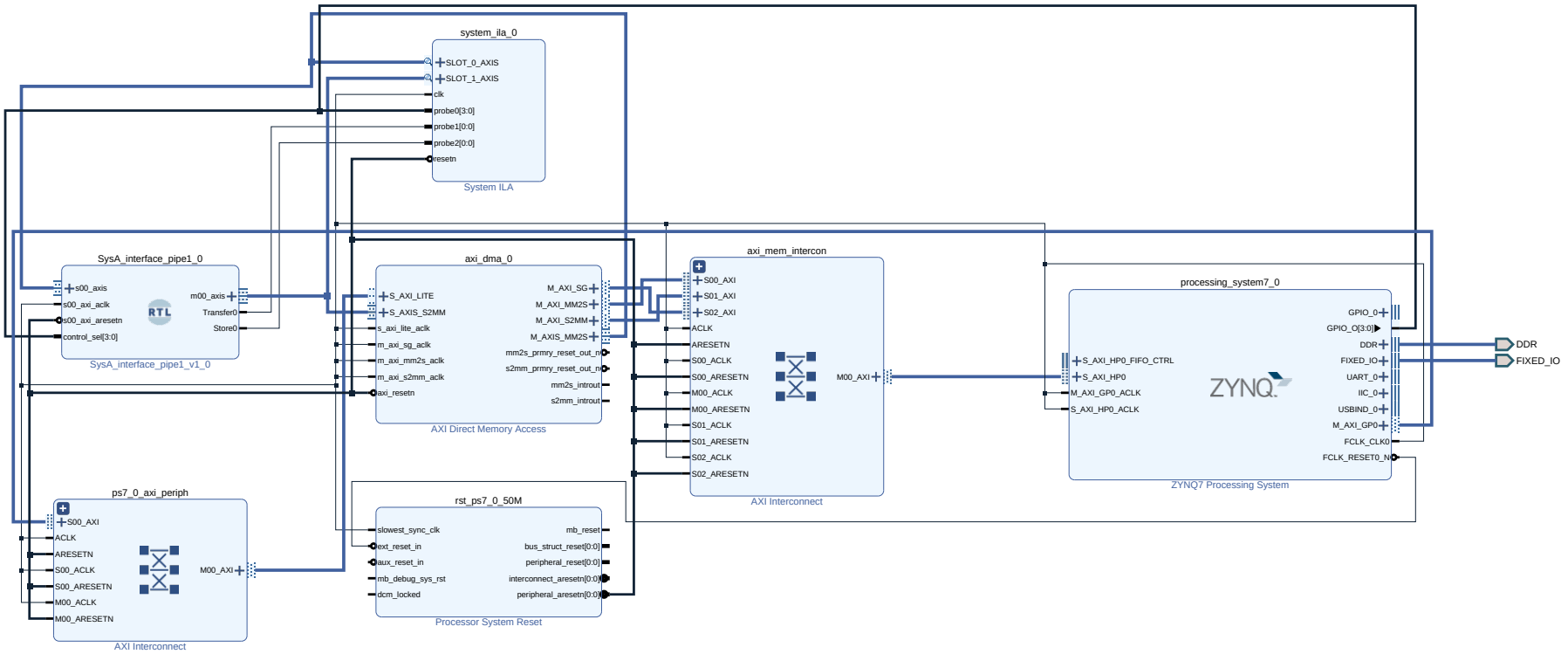


Figure 3.13: Block design of SoC with systolic array IP

3.6 Pipelining

Although the focus is on TangramFP and the systolic array, it is vital to add an embedded system perspective, since the design choice is SoC and the target device is Zynq-7000, nonetheless, for operational and effective systolic array a pipeline mechanism is required. Because if the systolic array system operates one multiplication at a time, the total waiting time from sending the data to receiving the result will be:

$$\frac{2 * N * N * precision}{channel\_width} + calculation\ time$$

Pipelining would utilize the calculation time to overlap with the transmission of data, increasing the effectiveness. However, the design of the pipeline depends on multiple design choices of the interfacing module that will affect the utilization area of the FPGA board such as channel width and buffering choices. As a result, the design of the pipeline mechanism will be carried out after the first implementation along with the decisions made then.

Chapter 4

Implementation

In this chapter, we discuss the implementation steps that were performed in this work. The implementation includes:

- synthesis and implementation.
- Post synthesis/implementation simulations.
- Redesign optimizations for several modules.
- Design parameters in the system block design.
- Exporting hardware and software development.

4.1 Behavioral Simulation

The behavioral simulation is carried out for every module independently to verify its behavior from the smallest module up to the wrapping top module. Also, simulation is performed with integrated modules at intermediate levels form sub-modules in the TangramFP MAC, up to the systolic array interface, for timing synchronization of data flow and correct computations. The simulation is performed with Vivado native simulator with testbench codes with single random generated values, random matrix test values, and selected edge cases in the case of testing the MAC or the multiplier/adder jointly or separately.

4.2 Synthesis

4.2.1 Synthesizing Systolic Array First Structure

The first synthesis for the systolic array module was performed according to the initial design with the mode selection unit, as described in the previous chapter. This design was neither integrated into an accelerated system nor was it encapsulated with an interface. The synthesis was successful; however, there was a severe warning because the number of input port pins was too high for any target device, and as a result the implementation cannot be performed. The target device at this point was Gensys-2 equipped with Kintex FPGA. It is important to note that Kintex does not have a processing system (PS), and in order to have a systolic array as an accelerator, we would need to create an appropriate utilizing system. As a result, we have to build an interface for the systolic array and limit the input/output port to a reasonable size. As we discussed earlier, we merged the step of creating an interface with both

adding a communication protocol and creating a control state machine to handle data transmission and controlling the systolic array operation.

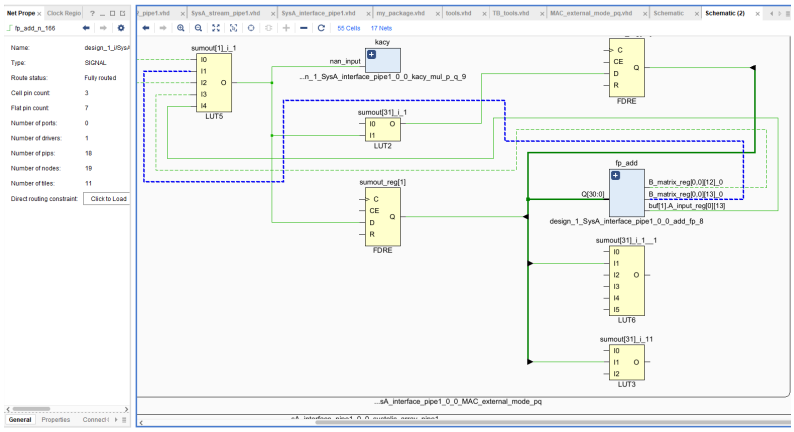
Once again we have attempted to implement the design in Vivado, shown in Fig. 3.12, but this time we have moved to Minized board equipped with Zynq-7000 PS. since we have an AXI we included the interface and the complete system with the PS. The size of the systolic array was 4×4 and the PE unit was TangramFP 32-bit. In this attempt, the result was also an area over-utilization. The reason is obvious that our design is not sensitive to resource devices, and Zynq-7000 is meant for small applications; thus we need to look at our design and make meaningful changes to reduce the area.

However, post synthesis simulation was possible for the design where the DUT was the interface module and the testbench acted as the PS side with AXI-DMA behavior, sending data and receiving results. We discovered that the systolic array did not behave as a computation device but always gave the same answer, and verified with the synthesis schematic that Vivado during synthesis and implementation optimization of the design has removed output of the last output of the last row of the PEs, and the outgoing data from the interface is grounded. Upon inspection of the synthesis log files, it was shown that optimizations were carried out due to constant propagation.

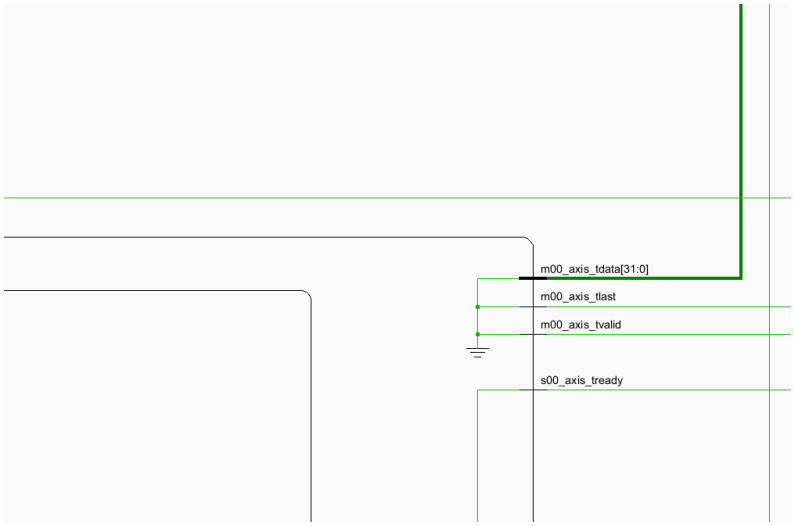
The cause of this optimization is that the optimizer does not correctly interpret the assignment of the last row of the array in the generate loop sum_output(j) <= C_matrix(N-1, j) in Listing 3.1 Line 11, and because it was not directed to an output port in the entity but buffered in the same entity, it interpreted that there is no change in the assignment and it is better to reduce redundant signals. As a result, we made changes to address the issues of over-utilization and over-optimization of the synthesis tool. In order to prove that this is the case, we proceeded before attempting any changes in the structure but reduced the size of the systolic array to 2×2 and changed the precision to TangramFP 16-bits. Additionally, the output precision in the TangramFP is twice the precision of the input, so in order to reduce further utilization area, we have added a precision down-grader at the end of the systolic array to reduce the buffer and the output width to a half. Then we proceeded to develop the software part and added an ILA IP module to the block design. The result on the terminal verified that indeed the output is always constant and the signals tlast and tvalid never rise when tested with the ILA debugging, as a result we applied modifications to the design as we see below.

4.2.2 Redesign of Considerations

In order to solve the issues that were encountered earlier, the systolic array design is reorganized. The reorganization did not affect the core of the systolic implementation, but it; first separated the systolic array into its own entity, second the mode selection unit was forwarded the entire matrices of A and W from inside the systolic entity, which duplicated the number of buffers, as well as there was duplication in buffers in the interface and the systolic entity. These buffers are all synthesized into register files, which means that the duplication is in the LUTs and the Flip-flops and so on. Further, the weight data were forwarded during the reception without buffering in the interface directly to the systolic array, as it will be fixed, and for the mode selection unit the data input is limited only to the exponent size rather than the complete FP number. For input A, when received through AXI, it is kept in one buffer inside the interface as described in Fig. 4.2, when the computation



(a) The multiplier Kacy output is not connected



(b) The output master tdata,tlast,and tvalid are all grounded

Figure 4.1: The schematic shows that the optimizer has removed the output signals

is commenced, the interface controller starts to forward a column from A to the systolic entity every clock event and slices the exponents of that column and directs it to the mode selection unit.

4.2.3 I/O Triangular Delays Instead of Counter-select Delays

One of the major modifications that are required is to move from counter-select shown in Fig. 3.2 and Fig. 3.4 delays that we have implemented earlier to using triangular buffers to synchronize inputs to the systolic array. The reason is that first we are separating the systolic array in a separate entity operating in FIFO manner only controlled with enable from the interface, second the counter-select delays are sensitive to changes in delays in any of the inputs, and third, it complicates the design when we try to add pipelining mechanism to the operation. However, a counter or multiple counters are still needed in the interface controller to trigger different stages of the operation in the pipeline. The following Fig. 4.3 shows the triangular in the systolic array architecture. Its important to point out this might add extra registers in addition to the utilized area, but this is also can be overcome by using Brams instead

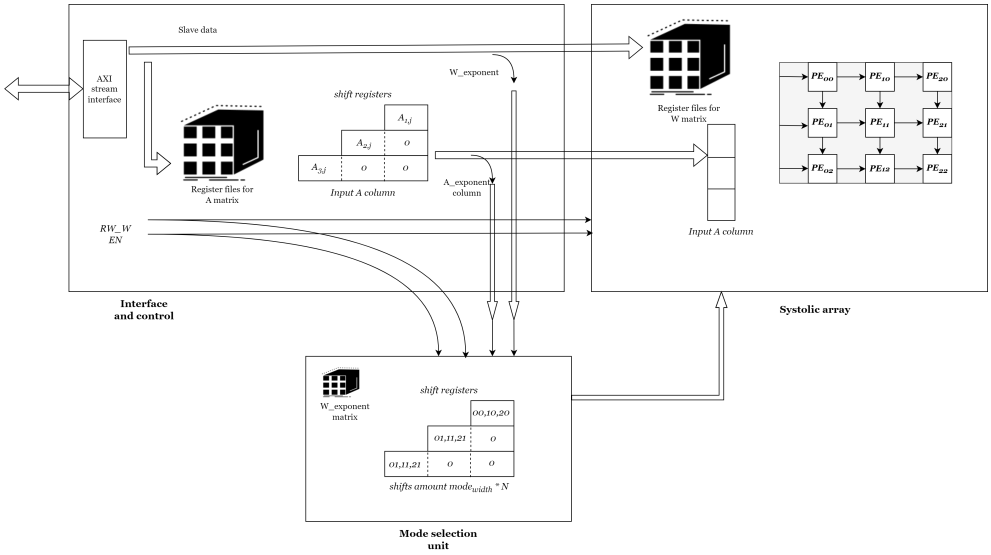


Figure 4.2: Reorganization of systolic array, interface, mode selection unit

of register files to store received data, by storing each column of A and in one location in the memory. In this stage, we have not implemented Brams as it will tie the design into specific size and application, and currently we are interested as much as possible in having the design generic and applicable to various configurations. In another angle, this modification will elongate the time between receiving data and the start of computation; however, based on the behavioral simulation the triangular buffer in the outputs delays the first output by N clock cycles, but the last output is only delayed by one clock cycle, on the other side, it opens the door to pipelining.

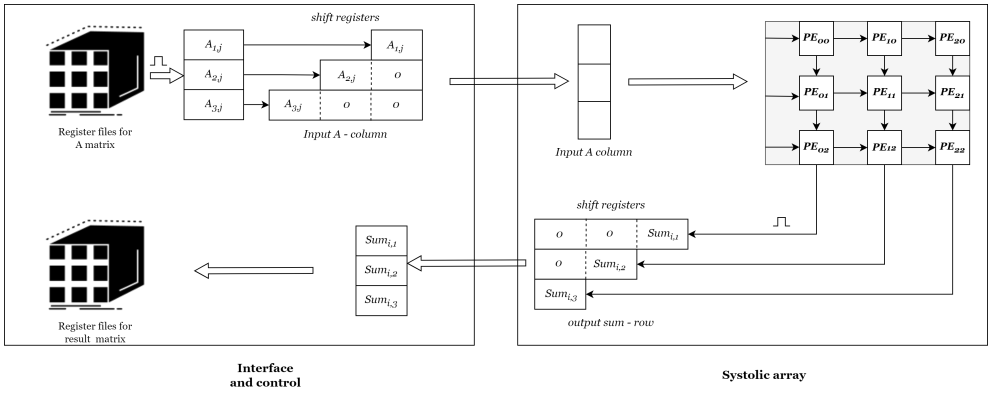


Figure 4.3: Changing from counter based delays into triangular delays

4.2.4 Pipeline and Control

In order to create a pipelined operation, we need to identify time points at every step of the operation so that we can split the operation into multiple stages that can overlap. In the following table Table 4.1, we identify key time points during the operation that were identified from analysis and inspection of the simulation waveform.

In this design, the AXI transmits one number at a clock cycle, so to transmit the

delays	A buffer	input triangular	Synch delay A	In systolic array	output triangular	total
first element out	1	N-1	2	1	1	N+4
first column out	1	N-1	2	1	N	2N+3
last column out	N	N-1	2	N	1	3N+2

Table 4.1: Key delays timing in the systolic array

$N \times N$ matrix, it takes N^2 cycles. While feeding the input to the systolic array is performed in column rate which takes N cycles to feed all the data of A, this means that the interface can start receiving a new A data the moment the first column is fed into the systolic array, which is one clock cycle after the start of computation. On the other hand, when the first output is ready it takes N cycles to store the full result matrix; however, the transmission of the result is a bottle-neck and takes N^2 to transmit the complete result, which means that the new result of the new A, the multiplication should not be out until at least the first column of the output is completely transmitted i.e. after N clock cycles. As a result, since the reception of data takes N^2, it adds more sufficient delay between inputs that cover the window between outputs; consequently, we can start immediately receiving inputs through the AXI channel.

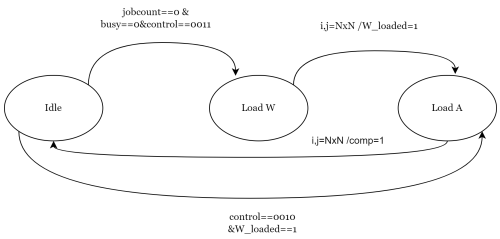


Figure 4.4: State machine for receiving

Figures Fig. 4.4 and Fig. 4.5 show the structure and mechanism of the controller implemented for the systolic array with pipeline. The operation of the controller is detailed as follows:

State machine at the start of the state machine waits in idle state for the control signal to trigger transitions to first receive weights matrix. Then it transitions to the load_W state, from load_W the next state is load_A. when load_W is done, it sets the flag B_loaded to allow the next time that idle state can transition to load_A as the weight matrix is already available. When load_A is finished, it transitions back to idle and sets the comp flag to trigger the start of the systolic operation.

Alternating counters and jobcount To enable pipelining, two alternating counters counter1 and counter2 are sufficient to track matrix multiplication from start to end. Since after the completion of feeding the input to the systolic array, it is still necessary to track the time to expect the output. Each counter is activated alternatively and both reset when they reach the total clock count at the last output column of the systolic array. The jobcount counter is incremented

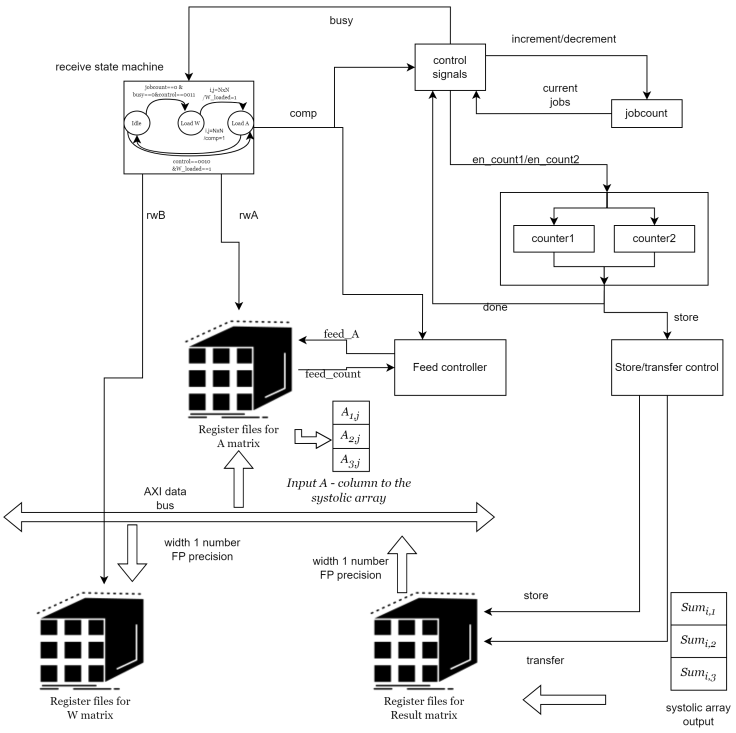


Figure 4.5: Systolic array pipeline controller

when the comp flag is triggered at the exit of the load_A state, and decremented when any of the two counters resets.

Controls and flags The controller has the following flag signals:

1. busy : when set, it prevents the transition to load_W as the weights are still in use.
2. en : enables operations of the systolic array.
3. en_count1 and en_count2 : enable signals for the two counters to start counting from 0.
4. store : is set when any of the two counters reaches the time at which the first output column will come out of the systolic array to start buffering output; it resets when the counter resets.
5. feed_A and feed_count: The feed_A controller sets feed_A when comp is set and rests it when feed_count reaches N.

4.3 Simulation and Synthesis

After applying the modifications and implementing the pipeline as described above, the design was simulated first, the operation was verified as we can see in the following wavform figure. Then it was synthesized for 4×4 size and FP32 precision. The result for Zynq-7000 still over-utilization, this time with slight difference than the maximum available logic cells in the device.

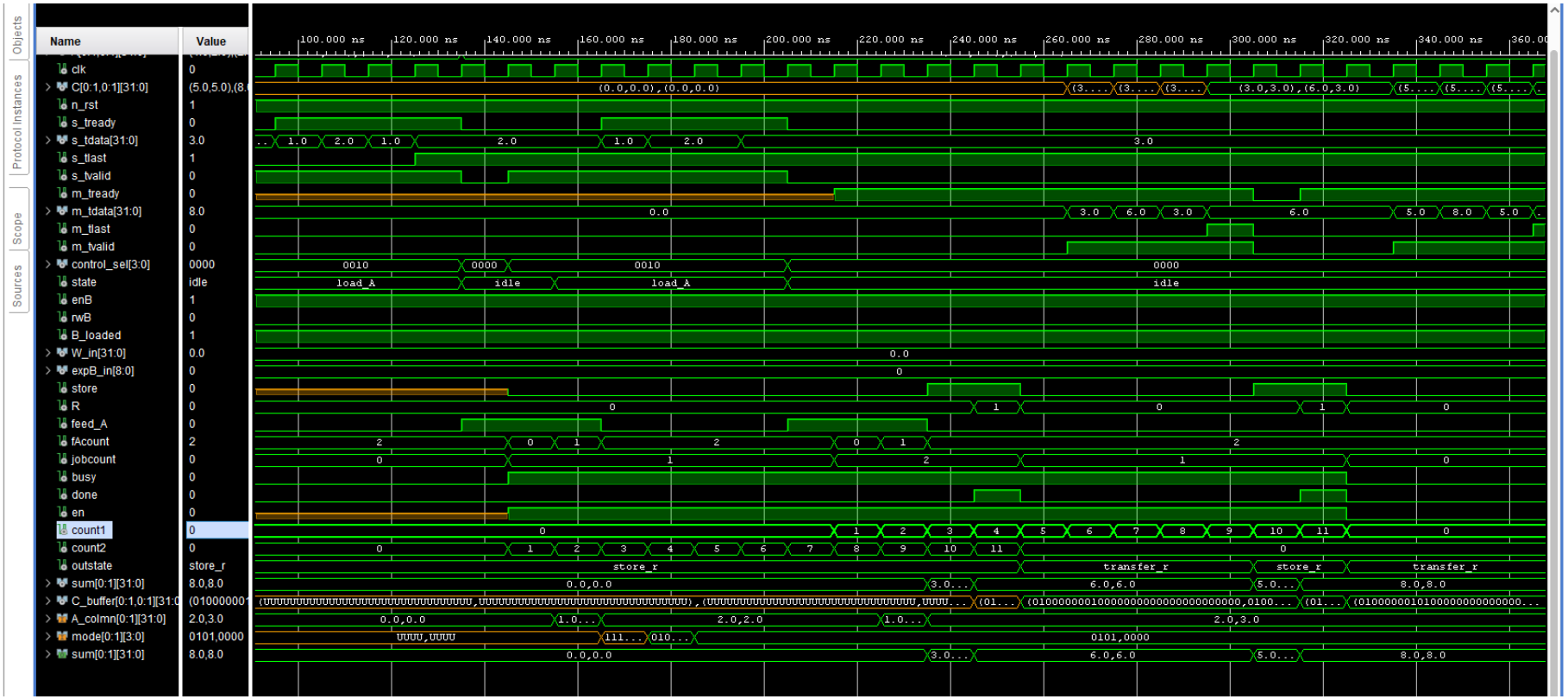


Figure 4.6: Waveform for systolic array pipeline simulation

4.4 Implementation and PS-PL Testing

As a result of the last implementation report, the size has been reduced to 2×2 size and the floating point precision has been reduced to FP16. Then the complete SoC system Fig. 3.13 is synthesized and deployed to FPGA. In order to test in real time the operation of the systolic SoC system, a software companion is developed in Vitis to run on the PS side of the Zynq-7000 SoC. The software initializes the hardware and sends two matrices to the PL through AXI DMA and signal commands through regular GPIOs and receives the results of the multiplication. Fig. 4.7 shows the multiplicand matrices sent to the systolic array PL and the results sent back to the PS. In this configuration, the weight matrix is sent then fixed for many multiplications with multiplicand matrices.

The waveform showed correct numerical operation; however, in this test configuration it is not obvious how much advantage the pipeline provides in acceleration performance. In Fig. 4.8 a tight pipeline configuration is tested by modifying the AXI DMA receiver to receive multiple matrices as one packet in scatter-gather mode before terminating reception, while in the PL the pipeline keeps track of how many multiplicand matrices were received and then does not raise the master tlast "m00_axi_tlast" signal unless sending the last element of the last matrix.

Despite that, the implementation of the systolic array in SoC system on Zynq-7000 device is running correctly, there are multiple points that need to be addressed; first, the size of the systolic array is very small 2×2, this may give an idea on performance and power reduction across non-Full modes but not enough, as for 4 PEs in the 2×2 configuration does not offer much parallelism advantage. In addition, the pipeline has to always be full so that the accumulation of power reduction is significant. Another point is that the mode selection operation is performed along the row columns of the systolic array configuration, which makes generated modes pairs of full and non-full mode; this is a major issue in measuring the potential power reduction or performance. Moreover, larger design can be synthesized with a smaller board, even "opt_design" can be run on it, however, since the run is over utilizing implementation and place and route can not be performed meaning the design has not been placed into real hardware; consequently, the power analysis will not reflect realistic estimations. The second point is that, since the Zynq-7000 could not accommodate larger designs, it is not possible to compare the performance, power reduction, and verification in different sizes. Third, currently, although running the SoC on real data has successfully produced correct results, during implementation stage the Vivado synthesizing tool showed timing failure with respect to setup time. This means either the systolic array should operate on a very slow clock rate or serious improvements in RTL design must be curried out. Both solutions are tried as detailed in the following section.

From a resource perspective, there are only two hardware devices available for this study, one is the Zynq-7000 and two the Genesys-2 board with Kintex device. Kintex is a pure FPGA without the PS side; this means that in order to replicate the same design of the SoC in the Zynq-7000 device, the design has to be adapted for the device. The adaptation includes using a Microblaze softcore instead of the Zynq-7000 processor and managing the additional units that are not built into the Microblaze, such as UART IP, DDR memory generator, and then the integration of all components in the block design. The Figure 4.9 shows the block design of the integrated systolic array in the SoC system using the Microblaze softcore on the Kintex-7 device on Genesys-2 board. Building on that, it was possible then to try

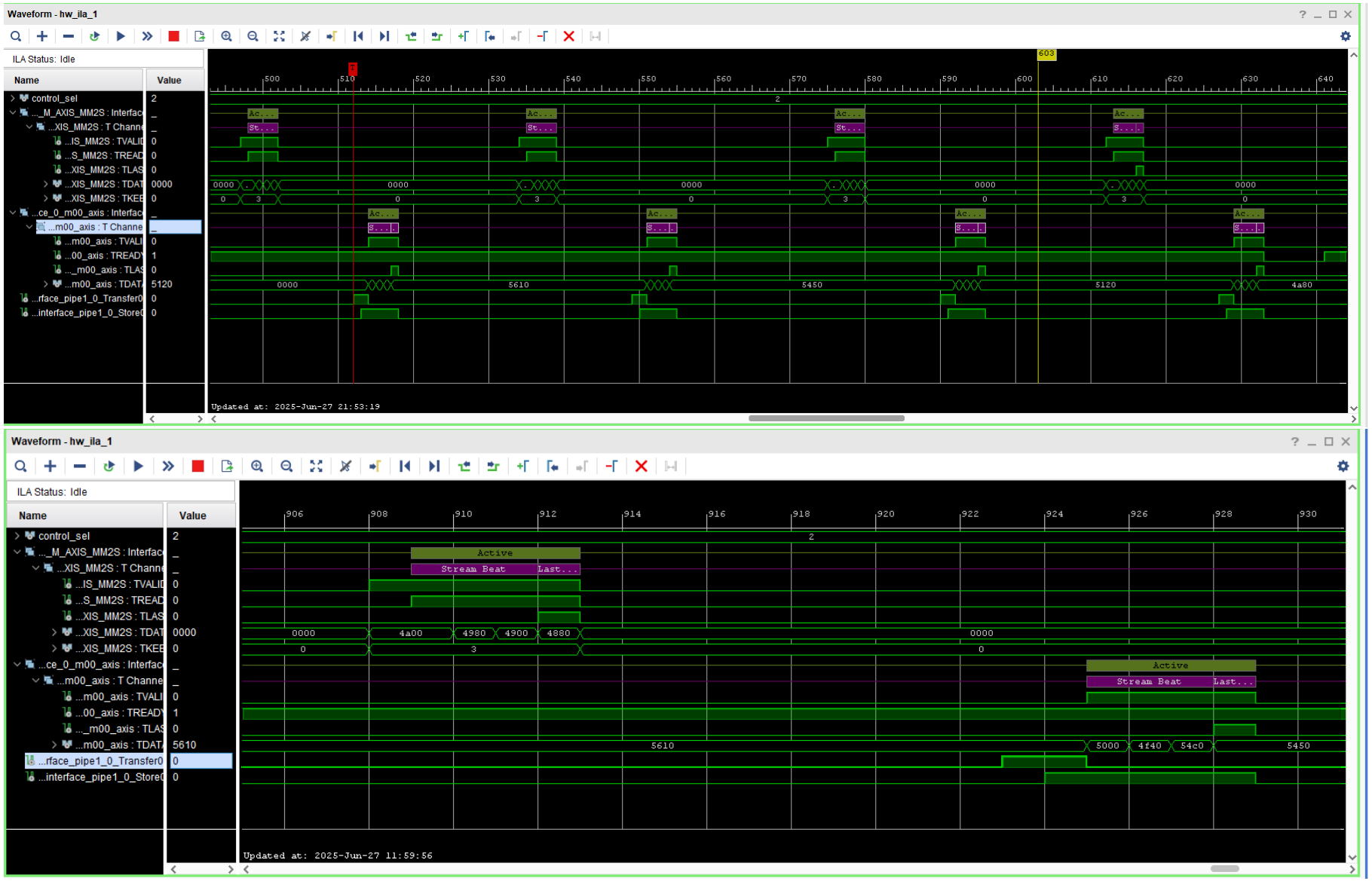


Figure 4.7: SoC matrix-result data communication in ILA

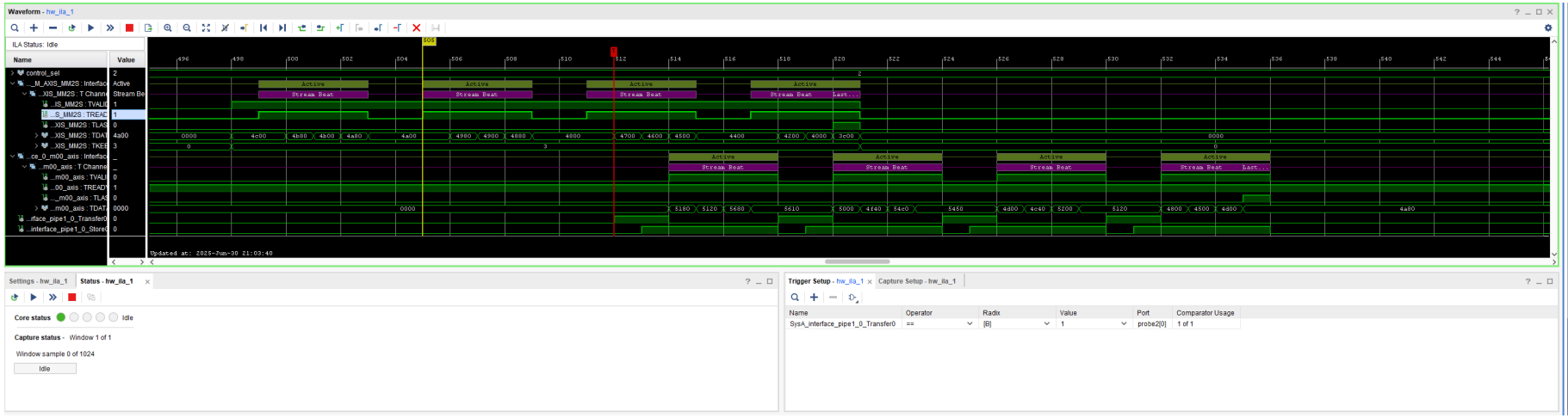


Figure 4.8: Multiple matrix multiplications in a pipeline

different sizes of the systolic array 2×2, 4×4 and 8×8 with 16 bit floating point precision, 2×2 and 4×4 with 32 bit precision. In Fig. 4.10 shows the ILA capture of multiplicands matrices sent to the systolic array pipeline and the result of the multiplication of these matrices with the weight matrix. The configuration of the systolic array is 8×8 array size with PE units of a 16 bit precision single-cycle MAC with thresholds for mode approximations 5 and 10.

4.5 RTL Improvements for Timing Closure

The design deployed into Zynq-7000 device can pass timing at clock rate of 30 MHz, this rate is very low. It is not the main focus of the work to improve the clock rate or achieve the fastest acceleration possible, but multiple improvements in the RTL design for the FPGA are possible while keeping the main characteristics of the design. The improvements targeted all the hierarchy of the design from interfacing to the mantissa multiplier inside the MAC unit in two directives; one, modification of RTL coding so that it maintains the same logic while reducing the utilization footprint in path or LUTs number, and two, remove the preprocessing of detecting special cases or normalizing the input outside the MAC unit to decrease the setup time.

1. Parallelizing the combinational setting of the latch enables the Dadda multipliers by placing the processing logic in the higher level entity.

2. Detection of zero, Nan, and infinity are removed from inside the MAC and placed in the pipeline interface for two reasons. First, then the mode unit can benefit from the obtained information into the mode detection procedure, second, the detection is performed during reception stage where each element received 3 parallel detectors record if it is zero, Nan, or infinity. this will decrease path and complexity in MAC operation.

3. In the adder unit, there is no significant change in the logic except using variables in the processes and some rearranging to stimulate shorter paths during synthesis.

4. In the mode unit, the operation of generating the mode vectors is split into two parts, first estimating the maximum result exponent, which will be computed combinationally immediately after the A column is passed into the unit. Second, at the next rising edge of the clock, the rest of the operation is completed.

5. In the MAC unit top module, processes are simplified by using smaller process blocks and using single bit flags inside the conditions instead of using the entire signal vector; for example, in the condition if(mantissa == 0){some logic} is split into 2 parts first, a single bit flag is detected and set, then for the logic inside the condition becomes if(flag == 1){the logic}. This improves timing and resource utilization, especially when the same condition is needed multiple times in different locations.

6. Regarding infinity and NaN inputs and results, the mode detection and MAC procedures are improved to distinguish between the two occurrences.

These improvements are not final, and they can continue for more desirable results, but this is a different scope of work, and here the purpose is to reach a

satisfactory level by which performing dynamic power analysis or any other measurements can be significant.

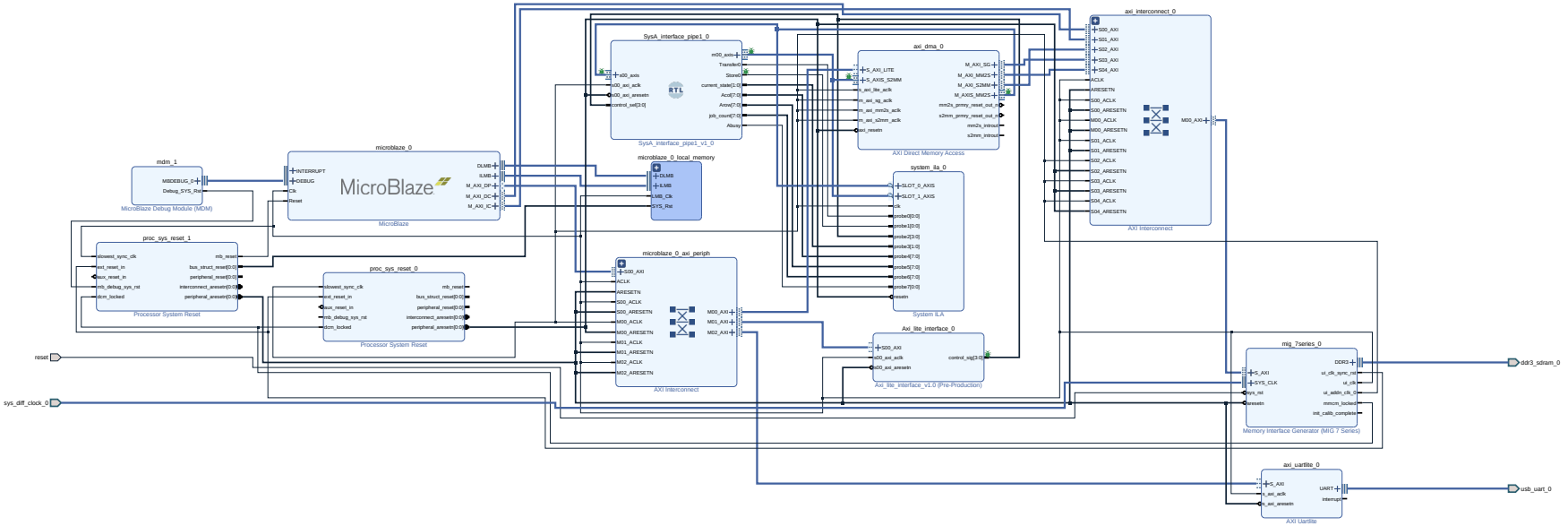


Figure 4.9: Systolic array design in SoC system using Microblaze in Kintex-7

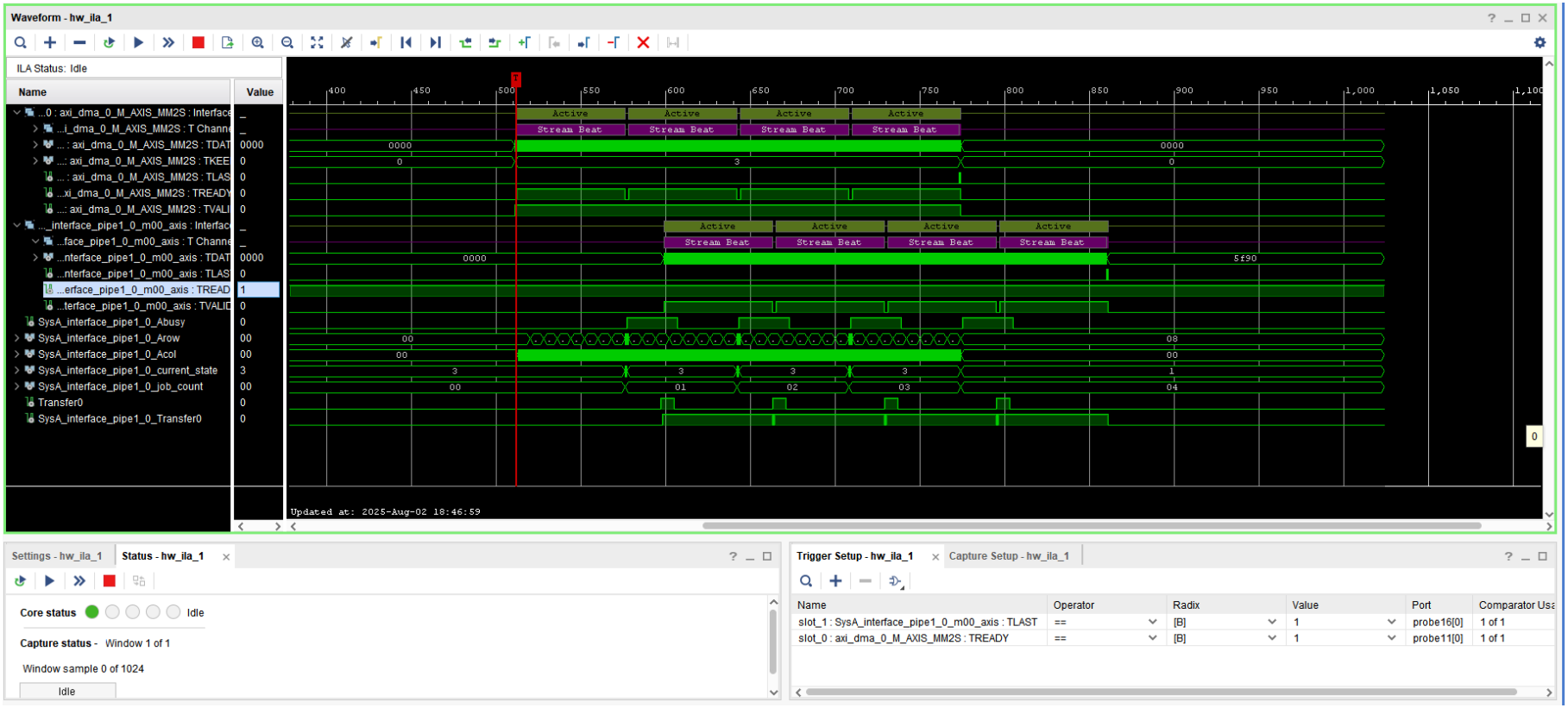


Figure 4.10: ILA capture transaction of 4 multiplicand matrices to the systolic array and the results of the multiplications

Chapter 5

Evaluation

This chapter will discuss the implementation evaluation for a systolic array in two major metrics; the area utilization of the design and the dynamic power reduction that would result from switching off some of the Dadda multipliers and corresponding parts in the design as a result of multiplication modes (Skip_BD, AC_only, Skip). In preparation for the testing, the systolic array design is synthesized separately i.e. without SoC integration.

5.1 Resource Utilization

First, it is important to have a sense of the amount of utilization area each part of the design takes. Here, the exact numbers are not as important as how they grow when increasing the size of the systolic array. To measure multiple utilization of these components, each part was isolated and synthesized separately because the synthesizer tool tends to optimize the design merging registers or flattening some parts to improve timing or reduce area based on the strategy in play; this is not a disadvantage, but for measurement purposes, it is better to confine the optimizations inside one module per synthesis.

The following two tables Table 5.1 and Table 5.2 show the synthesis utilization for a single MAC unit in a 16-bit floating point configuration and 32 bits, respectively. The number of LUTs, Flip-Flops are taken as a basis to signify and read the trend of the resource utilization along the size increase of the systolic array and the precision of the operands, for all two main parts of the systolic array design i.e. systolic array stream (without pipeline or interface) and mode selection unit.

Site Type	**Used**	**Available**	**Util%**
Slice LUTs*	966	14400	6.71
LUT as Logic	966	14400	6.71
LUT as Memory	0	6000	0.00
Slice Registers	102	28800	0.35
Register as Flip Flop	32	28800	0.11
Register as Latch	70	28800	0.24
F7 Muxes	19	8800	0.22
F8 Muxes	0	4400	0.00

Table 5.1: LUT utilization report for MAC 16 bit

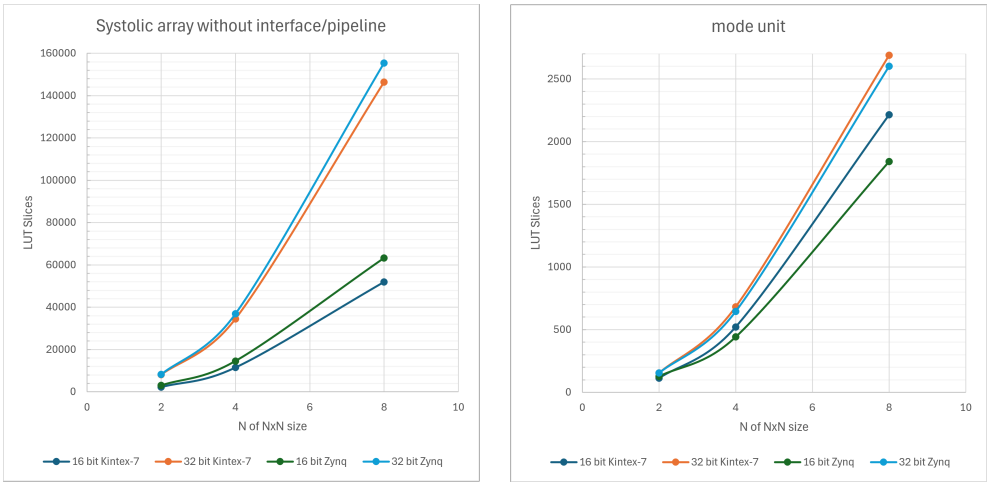


Figure 5.1: LUT utilization of the systolic array and the mode selection unit for different sizes and FP precision

Site Type	Used	Available	Util%
Slice LUTs*	2627	14400	18.24
LUT as Logic	2627	14400	18.24
LUT as Memory	0	6000	0.00
Slice Registers	204	28800	0.71
Register as Flip Flop	64	28800	0.22
Register as Latch	140	28800	0.49
F7 Muxes	0	8800	0.00
F8 Muxes	0	4400	0.00

Table 5.2: LUT utilization report for MAC 32 bit

5.1.1 Systolic Array Utilization for Different Sizes

Although the Zynq-7000 device was over-utilized during synthesis, it gave a sense of trend LUT usage, as can be seen from the similarity in Fig. 5.1 between the curves of utilization of optimized synthesis on the Zynq-7000 device and those from the place and route on the Kintex-7. It is logical to have lower LUTs after place and route especially when the implementation strategy was aggressive exploration in order to reduce critical paths for hold time preservation, but in the mode unit the numbers actually increased after implementing on Kintex, and the reason for that is during the last improvement round additional logic was added to improve the overall numerical accuracy with respect to the special cases of the multiplication.

5.2 Timing Evaluation

The result of these improvements can be summarized in table Table 5.3 showing the clock rate that was achieved in all the configurations tested of the systolic array in addition to the Worst Negative setup Slack time, Worst Hold Slack and Worst Pulse Width Slack. The last column shows the maximum possible clock frequency computed for these designs on the current hardware with the formula

$1/(clockperiod - WNS)$. The WHS shows very small values, indicating that the place and route process has passed hold requirement with critical margin. A possible explanation for that is the imbalance in the data paths between flip-flops when more emphasis was put to minimize data path delays while synthesizing the systolic array as standalone with clock data as external inputs may incur larger clock skew. For instance, in the timing report for the systolic design size 8×8 with 16-bit precision, the small hold slack of 0.065 ns occurs because the FF-to-FF path has minimal logic delay 0.399 ns while experiencing significant clock skew 0.264 ns. In this direct connection between flip-flops, the routing delay between them 0.221 ns plus minimal logic 0.178 ns barely exceeds the clock distribution delay difference of 0.264 ns and the flip-flop's internal hold requirement of 0.070 ns. This creates a timing-critical situation where the data signal arrives only slightly before it is no longer required to be stable, leaving minimal margin for process, voltage, and temperature variations. We leave the discussion about hold slack here without engaging further in possible improvements as long as it is safe for performing post implementation simulations and running the required tests.

size	precision	clock period ns	clock rate MHz	WNS ns	WHS ns	WPWS ns	max clock rate MHz
2×2	16	14	71	1.368	0.050	6.358	79.16402787
4×4	16	14	71	0.0949	0.063	6.358	71.91605958
8×8	16	14	71	0.0880	0.065	6.358	71.88039103
2×2	32	17	58.8	0.056	0.06	7.858	59.01794145
4×4	32	18	55.5	0.024	0.071	8.358	55.62972853
8×8	32	19	52.6	0.069	0.07	8.858	52.82341134

Table 5.3: Minimum clock period and maximum clock rate for different systolic array sizes

On the other hand, with regard to the setup time represented in the table under WNS, the major issue is the long carry chains. These carry chains are placed in two key locations; in the multiplier partial products adders and in the adder unit after multiplication. The effect of these chains worsens when using 32 bit precision. as can be seen in Fig. 5.2 that a signal is traversing carry chains in the device making the path critical and barely preserving the setup time with low setup slack. One possible remedy is to delay the last carry propagation in the addition of the partial products and unify it with the adder carry propagation, achieving a full fusion of multiply-accumulate operation. It is Also possible to fuse all the additions across the PEs into a big adder at the output of the systolic array. These solutions are large enough to change the scope of this work and thus are left for future investigations.

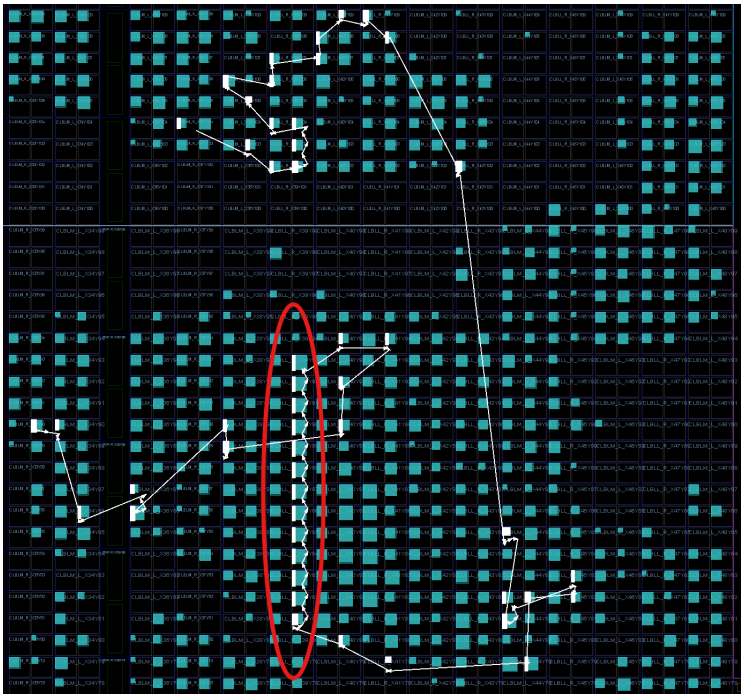
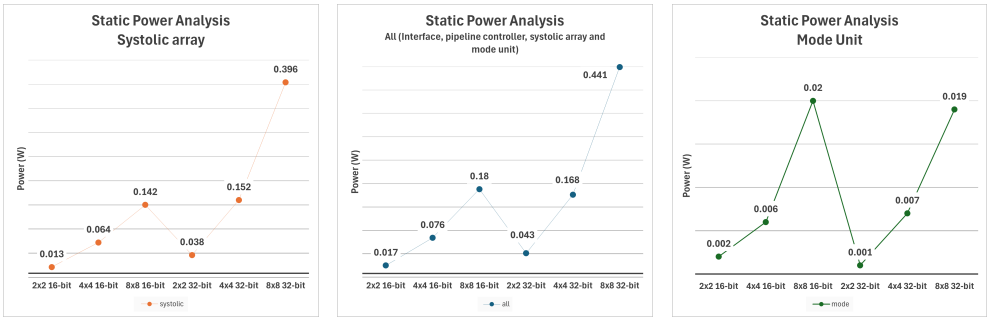


Figure 5.2: Carry traversing on carry chains inside a critical path in the device after implementation

5.3 Static Power Analysis

After completing the examination of the design in multiple configurations from the perspective of time and resource utilization, it is important to analyze the estimated power consumption. In this regard, Vivado provides a tool to estimate and analyze the power behavior of the design based both on default assumptions and on signal switching activity collected from post-implementation timing simulation.



(a) Systolic array (b) Whole system (c) Mode unit

Figure 5.3: Static power estimation for the systolic array, whole system, and mode unit of sizes 2×2, 4×4, and 8×8 in 16 and 32 bits.

This section shows the results from estimating the power consumption in the systolic array. The graphs show the sum of the estimated consumed power, including signal activity, clocks, IO, and logic in selected units in the hierarchy; the power consumed inside the mode unit depicted in Fig. 5.3c as separate and the power in the

systolic array in Fig. 5.3a without control logic or transmission management, then all the systolic design which sums the mode and the systolic array and additional control/transmission logic and signals in Fig. 5.3b.

The power analysis tool was run under default parameters with typical process, commercial grade for temperature, and default switching activities as described by 12.5 for toggle rate and 0.5 static probability. The mode power consumption increases exponentially with the rise of the systolic array size but basically there is no difference between 16 and 32 bits and that's understandable because the size of the registers inside the mode unit do not change significantly from 16 bit with 5 bits and 7 bits for the exponents operation and 8 and 10 for 32 bits, besides depending on how much optimization took place during place and root stages, the result might seem counter intuitive like in the case of 2×2 with 16 bit having higher power than its counterpart in 32 bits. On the other hand in the systolic array and thus consequently in the overall system, it is very clear that an increase in the size has a quadratic effect on the power, while the power consumption approximately doubles when changing from 16 to 32 bits with the same size of the systolic array.

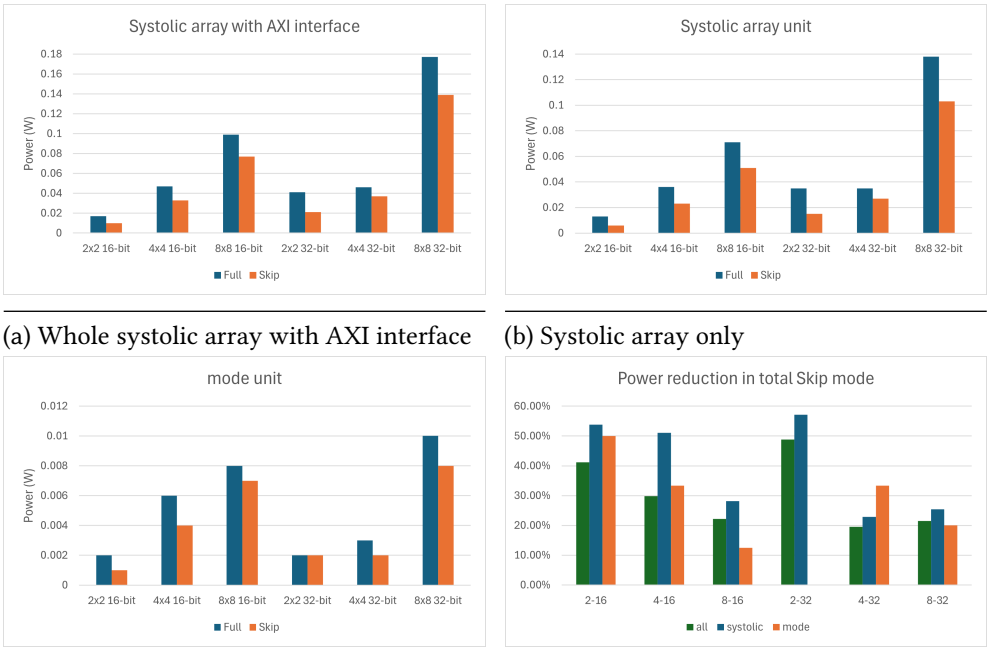
It is important to note that this estimation is an approximation and not a realistic measurement of the system, and still it is necessary to conduct another estimation based on switching activity of the placed and routed netlist in hardware through post-implementation simulations as discussed in the following section.

5.4 Switching Activity Based Power Analysis

In order to measure the dynamic power reduction that will be induced from mode selection of the Tangram process, a complete design of the systolic array with its pipeline interface and the mode unit is synthesized and implemented for the sizes 2×2, 4×4 and 8×8 for both 16 bit and 32 bit FP precisions. The plan is to run Post-implementation timing simulation and capture the signal switching activity in a .saif file then run the power analysis tool in Vivado to estimate the hierarchical power consumption in the device. To measure the reduction first, the simulation will run to stimulate full mode multiplication in all multiplication are run in the systolic array by inputting alternating equal numbers, because if the input was steady all the time then the switching activity will reduce to minimum leading to incorrect power estimation with respect to the tested mode. To achieve that, the weight matrix remains constant with equal numbers say all 1s, then input matrix will have equal values across the columns but different across the rows, this way all the multiplication modes will be Full mode.

The second step is to stimulate all Skip modes multiplication across all PEs in the systolic array, and this is achieved by having weight matrix with 0s while keeping the same input matrix conflagration from the first step, this ensures that approximately the inner circuitry remains active to notice in power reduction in the analysis that can clearly be attributed to general Skip mode in the operation. Third, running the analysis from switching activity extracted from multiplying realistic data used in AI, model in order to measure the advantage of using Tangram multiplier in this particular design on AI acceleration from the power perspective.

The three figures; Fig. 5.4a, Fig. 5.4b, and Fig. 5.4c show the power estimations based on the switching activity recorded from the simulations. They show the contrast between having Full mode and Skip mode in the overall system, inside the systolic stream, and in the mode unit only. The power consumption increases with the size of the systolic array; nevertheless, the difference between the Skip mode



(a) Whole systolic array with AXI interface

(b) Systolic array only

(c) Mode select unit only

(d) Difference between Full and Skip mode

Figure 5.4: Dynamic power estimation of the systolic array system: (a) whole design with AXI interface, (b) systolic array only, (c) mode select unit, and (d) power difference between Full and Skip mode.

situations and the Full mode is significant. It is important to note that during the cases 8×8 16-bit, 4×4 32-bit, and 8×8 32-bit running post-implementation timing simulation was not feasible due to the computer performance limitations where the simulations were run, but was compensated with post-implementation functional simulation to log the switching activities.

The figure Fig. 5.4d shows that the reduction can reach around 50% for the systolic unit while it decreases as the size goes up in the mode unit and over the whole the system, it is very clear that at 8×8 16-bit, 4×4 32-bit and 8×8 32-bit the percentage is lower between 20 and 30, and it is not clear if this is because the data are collected from functional simulation and hence less realistic switching activity than timing simulation or simply these represent more dense architecture leading to higher base line consumption or may be both; however, the former may have more weight. In any case, the lowest percentage of power difference being around 22% for the systolic array is a significant value and is not far behind the reduction value achieved in TangramFP Yao et al. [2024] for a single multiplier in Skip mode.

5.5 Discussion

The module design is evaluated in terms of timing, area utilization, and estimated power consumption. We observe that the achievable clock rate decreases and then stabilizes around 71 MHz for FP16 and 52 MHz for FP32 as the systolic array size increases. A similar trend is seen in power reduction: at smaller sizes and/or lower precision, the reduction is large, but as the array grows, the reduction flattens toward 25%. With caution, we suggest that the clock rate may asymptotically approach 70 MHz for FP16 and 50 MHz for FP32 at very large sizes (e.g. 1024×1024). This

interpretation is supported by the fact that the transition from a 4×4 array (16 MAC units) to an 8×8 array (64 MAC units) represents a large increase in complexity, yet the corresponding change in clock rate and power reduction was comparatively small.

Chapter 6

Conclusion

This work investigated the integration of TangramFP into systolic array architectures to achieve power reduction through selective disabling of ineffectual partial products in floating-point computation. The research successfully developed, verified, and implemented a complete accelerator design that spanned from individual MAC units to the systolic array interface and was integrated into SoC systems on both the Zynq-7000 and Kintex-7 FPGA platforms.

The primary objective was determining whether TangramFP's unit-level energy savings could scale to array-level implementations while maintaining IEEE-754 compliance. The integration required developing a mode estimation and selection unit to control TangramFP optimizations in the systolic array, along with a specialized synchronous control path for mode signals in the systolic structure. This dual-data-path architecture extends conventional systolic arrays by synchronizing computational data with mode instructions for each MAC unit. Evaluation of the implemented design of the systolic array in multiple configurations (2×2, 4×4, and 8×8) in the precision FP16 and FP32 demonstrated consistent dynamic power reductions of up to 50% at smaller sizes and lower precision, stabilizing at approximately 25% for larger configurations. The clock rates scaled predictably, asymptotically approaching 71 MHz for FP16 and 52 MHz for FP32 implementations.

This work demonstrates that power-aware computation can be successfully integrated into systolic arrays without fundamentally disrupting their parallel processing capabilities. The results validate TangramFP as a scalable mechanism for reducing power consumption in floating-point computation while preserving IEEE-754 compliance, offering an alternative to reduced-precision approaches that maintains numerical fidelity within acceptable error bounds. Although FPGA resource constraints limited the evaluation to the 8×8 array size, the predictable scaling behavior suggests that the architecture would maintain stable performance at larger scales. The research establishes the first successful integration of TangramFP into systolic array architectures and provides a reusable framework for embedding data-dependent power optimization techniques into parallel computing systems.

Chapter 7

Limitations and Future Work

7.1 False Maximum Exponent and Over Approximation Mode

Subsequent evaluation of the mode unit revealed a major limitation in the estimation of the multiplication modes. The mode estimation so far relies on determining the maximum exponent among the multiplications in the dot product $max(exp\_AB_i = exp\_A_i + exp\_B_i)$ then the mode is decided by $diff_i = Max\_exponent - exp\_AB_i$. In a specific case when the max exponent corresponds to a certain sign while a closer or equal other exponent has belongs to a number of opposite sign to the maximum exponent then during the addition of the products the operation between the two numbers becomes subtraction and because the exponents are close or equal in value their subtraction leads to a much lower exponent in reality, this lower exponent will in turn require less disabled partial products in the multiplication in the next number. While in this mode module, $diff_i$ is higher because $max_exponent$ is not dynamically changed with regard to whether there is another exponent closer in value and of opposite sign number. These limitations did not affect the research findings, as the focus was on power consumption given the fact that a mode is Full or Skip.

One possible solution is to add to the module unit a functionality of grouping the exponents in two groups according to the sign of their numbers then determine the maximum in the two groups, the possibility of this case happens when the two groups have maximums closer in value based on preset threshold, then when that is true an estimate of a new max exponent is determined. If there is still an exponent in the opposite group that is closer in value to the estimate max, then a new estimate is calculated again. The maximum should not be less than the remaining exponents. Fun fact, with this solution the condition of selecting Full mode if the $diff$ is negative then becomes relevant and a typical case of operation. This solution may complicate the mode unit, but it is important to have a very accurate mode selection at all times because the strong side of the TangramFP solution is to reduce power with high accuracy (low ULP like IEEE-754) therefore, improving mode estimation in parallel setting is worth investigating in further details in future work.

7.2 Carry Chain and High Performance

We have seen that the long carry chain has led to slow clock rate, and it affects scaling the systolic array design to high performance designs or large size architectures. The TangramFP multiplier contains multiple Dadda multipliers and adders of the partial

products; AC multiplier produces AC_1 and AC_2 vectors with U and V are all fed to 4 input adders to produce F_x, at the end of the Carry Save Ahead (CSA) operation a carry is propagated in the final stage which is seen in the carry chain. Parallel to that AD_1, AD_2, BC_1, and BC_2 are summed to give F_y. then both are fed into optimized 2 input adders which also contain carry propagation. After that in the adder unit the multiplication result is also added to the passed over accumulation, which again has a carry propagation. In this work a semi fused multiply-add is achieved, but it is interesting to investigate solutions that involve complete fused operation in the MAC where the carry propagation happens only once after adding the multiplication to the accumulation. Further, an investigation can go beyond the TangramFP towards delaying the carry propagation at the end of the dot product at the systolic array output and study how much the performance can improve.

7.3 Memory Inference in FPGA

One of the issues that complicated the implementation of FPGA was the inability to utilize the built-in RAMs because always multiple data (N) are accessed at the same clock and written at the same clock. One solution was to have an N RAM blocks to be accessed together, but this is also an issue when scaling the design to larger sizes while trying to use the same constraint platform. Further studies could try to investigate possible options to use the FPGA RAM while maintaining the same level of reconfigurability and optimizing area utilization.

7.4 Generic and Realistic Systolic Array

In this work, we have not given much focus to the reality of AI operation in our systolic design. AI systems generally have much larger matrix multiplication not always in the square shape and will include different types of operations depending on the model. Therefore, we see a dedicated study into creating a realistic implementation of TangramFP based systolic array, which in turn require a generic mode estimator suitable to work in non-square sizes and or if the operation is for example convolution/correlation not just matrix multiplication.

7.5 Tiling

Considering matrix multiplication in inference AI models deployed in edge AI, having such small systolic array requires a tiling system to segment and compile large matrix multiplication. One straightforward solution could be by designing another tiling unit to manage the accumulation of partial matrices and buffer the output. However, since tiling requires streaming both matrix inputs, in that situation the results are accumulated locally. We can utilize that by having the same tiling accumulation and control logic embedded inside the systolic array where the overall result is accumulated in the same MAC. This will save area utilization and can give more advantage in delaying the long carry chain and let it be done finally in the last clock cycle or even be split into multiple cycles. Special attention will have to be given to tracking the exponents, such as when increments that should occur when multiplications overflow.

7.6 Leveraging TangramFP for Bandwidth

7.6.1 Internal Memory and Systolic Array Bandwidth

TangramFP can be utilized to reduce the size of the area utilization by nearly half. This can be obtained by reducing the data connections and registers by half for all the mantissas using the tangram $P : Q$ the operation can be organized first using the organization where both matrices are streamed in and the accumulation is performed in the PE, the same organization suitable for tiling. Then each matrix multiplication is split into two repeated flows. The first flow is the exponents and the high half of the mantissa in this flow the Pes will ignore the input if the mode is Skip, the second flow for the lower halves of the mantissas and each PE will ignore the input if the mode is not or Skip_BD. The time of operation will nearly double; however, by increasing the clock rate the performance could be higher as the arithmetic operations will be in shorter numbers.

7.6.2 External Transmission Bandwidth

One of the major limitations of the systolic array is the transmission delay. When the size of the array was small, such as 2 or maximum 4 the transmission can be pipelined with the computation to shadow some of the delay, but this delay grows exponentially with the size increase while the computation delay is linearly growing with the size; as a result pipeline will end up as serial operation and everything is bottle-necked by the transmission. Increasing the channel width will not solve the problem, especially in SoC as a channel such as AXI can increase to a maximum width of 64 bits, meaning dividing the transmission delay by 2 or 4 and if N is large the delay still bottle-necking the operation.

It might be useful to investigate the possibility of utilizing TangramFP to decrease the amount of data transmission. This can be arranged by sending only the first half of the mantissas, then only sending the halves that will be needed by the two modes of and Skip_BD. The Tangram segmentation thus must be two way cut $P : Q$ instead of $1 : P : Q$. also modes will have to be computed before transmission. Consequently, the multiplication exponents can be sent instead of the exponents of the operands, as the addition will already have taken place in the mode estimator/selector. Architecture possibility may be with an arrangement where there are two accelerators, one near memory to estimate the modes and the other is the systolic array.

However, for a lower half to not be sent at all, it means that the modes that are associated with it will all have to be AC_only or Skip. This might be of lower probability depending on the data probability distribution, so a preliminary study is necessary to show that the distribution of the modes is regarding a systolic array and how much reduction in Bandwidth is achievable.

Bibliography

Wantong Li, Junmo Lee, and Shimeng Yu. Optimization strategies for digital compute-in-memory from comparative analysis with systolic array. In *2023 IEEE 5th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pages 1–5, 2023. doi: 10.1109/AICAS57966.2023.10168651.

Zhi-Gang Liu, Paul N. Whatmough, and Matthew Mattina. Systolic tensor array: An efficient structured-sparse gemm accelerator for mobile cnn inference. *IEEE Computer Architecture Letters*, 19(1):34–37, 2020. doi: 10.1109/LCA.2020.2979965.

Gyubin Seong, Jong Kang Park, and Jong Tae Kim. Fpga implementation of cycle-reduced diagonal data flow systolic array for edge device ai. In *2023 20th International SoC Design Conference (ISOCC)*, pages 99–100, 2023. doi: 10.1109/ISOCC59558.2023.10396567.

Cristian Sestito, Shady Agwa, and Themis Prodromakis. Trim, triangular input movement systolic array for convolutional neural networks: Architecture and hardware implementation. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 72(5):2263–2273, 2025. doi: 10.1109/TCSI.2024.3522351.

Yuan Yao, Xiaoyue Chen, Hannah Atmer, and Stefanos Kaxiras. Tangramfp: Energy-efficient, bit-parallel, multiply-accumulate for deep neural networks. In *2024 IEEE 36th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 1–12, 2024. doi: 10.1109/SBAC-PAD63648.2024.00009.

V.V. Zunin and I.I. Romanova. Parameterized computing module generator based on a systolic array. In *2022 IEEE International Conference on Industry 4.0, Artificial Intelligence, and Communications Technology (IAICT)*, pages 217–220, 2022. doi: 10.1109/IAICT55358.2022.9887460.