VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
OFFICE FOR INTERNATIONAL STUDY PROGRAM
FACULTY OF ELECTRICAL AND ELECTRONICS ENGINEERING

———————— * ————————

LOGIC DESIGN / LOGIC SYNTHESIS
Lab 2: Matrix Multiply Project

Instructor : **Dr. Trần Hoàng Linh**
TA : **M.S. Nguyễn Tuấn Hùng**
Subject : **Logic Design / Logic Synthesis**
Fullname : **Lương Triển Thắng**
Student ID : **2051194**

Ho Chi Minh City, October 02, 2023

Contents

List of Figures

1. Overview

In this project, the primary objective is to explore and harness the specialized hardware features available in modern FPGAs (Field-Programmable Gate Arrays). While in the past, I have focused on utilizing CLBs (Configurable Logic Blocks) for combinational and sequential circuitry, this project marks a significant departure as I delve into the realm of hardware multiply-accumulate units and dual-port Embedded RAMs within the FPGA.

The overarching aim of this endeavor is to perform matrix multiplication with a specific set of requirements: `Y`, a 24-bit result, is to be calculated as the product of two 8-bit matrices, `A` and `B`, with the addition of a 16-bit matrix `C`. This operation not only demands correctness but also prompts me to explore optimization strategies to reduce the number of cycles required for the computation. Additionally, I aspire to leverage as many of the FPGA's resources as possible, aiming to maximize its utilization while striving to increase the F_{max}, a key parameter denoting the maximum achievable clock frequency.

Throughout the course of this project, the design underwent several iterations and revisions, each bringing me closer to the desired outcome. The culmination of these efforts was a significant achievement: achieving an impressive F_{max} of 160 MHz while completing the matrix multiplication task in just 268 cycles. These milestones demonstrate not only the effective utilization of FPGA hardware features but also a profound understanding of optimization strategies and real-world applications in matrix multiplication.

In this report, I will delve into the methods employed, challenges faced, and the results obtained during the course of this project. By sharing this journey, my aim is to provide valuable insights into the intricate world of FPGA design and the exciting possibilities it presents in terms of accelerating complex computations and achieving high-performance goals.

2. Background

2.1 Matrix - Vector Multiplication

This project involves the design of a hardware implementation for a matrix-vector multiplication, followed by an addition operation with another vector. This fundamental mathematical operation, often referred to as a fused multiply-add (FMA) operation, finds extensive applications in various fields, with a particular emphasis on machine learning (ML) and digital signal processing (DSP).

In the realm of machine learning, the hardware implementation of matrix-vector multiplication followed by vector addition plays a pivotal role in neural networks and deep learning models. These operations are the backbone of many neural network layers, such as fully connected layers, convolutional layers, and recurrent layers. Efficient hardware implementations of these operations are crucial for accelerating the training and inference processes of ML models, enabling real-time or near-real-time predictions in applications ranging from image recognition to natural language processing.

Digital signal processing, on the other hand, heavily relies on matrix-vector multiplication and vector addition for a wide range of applications. In audio processing, for instance, this hardware design can be employed for filtering, convolution, and spectral analysis, allowing for enhanced audio signal manipulation and analysis. Moreover, in wireless communication systems, the efficient execution of these operations is essential for tasks like channel equalization, beamforming, and modulation-demodulation processes, contributing to the optimization of data transmission and reception.

2.2 Project Overview

The problem at hand revolves around solving a matrix equation of the form:

```
Y(24-bit) = A(8-bit) * B(8-bit) + C(16-bit) = F(23-bit) + C(16-bit)
```

This equation presents a unique challenge and set of requirements due to the varying data widths and the need to maintain precision throughout the computation.

In this scenario, we are working with three key matrices and a resultant matrix:

- A 128×128 matrix `A` containing 8-bit unsigned integers.
- A 128×1 column vector `B` containing 8-bit unsigned integers.
- Another 128×1 column vector `C` containing 16-bit unsigned integers.
- The target matrix `Y`, which is also a 128×1 column vector `But` contains 24-bit unsigned integers.

2.3 Design Idea

2.3.1 Matrix Multiplication

To tackle this complex matrix equation, let's start by breaking it down, beginning with the matrix-vector multiplication. The fundamental idea behind this operation is quite straightforward: we aim to decompose the 128×128 matrix `A` into 128 individual 128×1 column vectors. This transformation effectively turns the problem into performing 128

separate matrix-vector multiplications, each involving a 128×1 vector from `A` and an 8-bit column vector from `B`. This results in a total of 128 intermediate 128×1 vectors.

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & \cdots & a_{0,127} \\ a_{1,0} & a_{1,1} & a_{1,2} & \cdots & a_{1,127} \\ a_{2,0} & a_{2,1} & a_{2,2} & \cdots & a_{2,127} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{127,0} & a_{127,1} & a_{127,2} & \cdots & a_{127,127} \end{bmatrix} \times \begin{bmatrix} b_{0,0} \\ b_{1,0} \\ b_{2,0} \\ \vdots \\ b_{127,0} \end{bmatrix} + \begin{bmatrix} c_{0,0} \\ c_{1,0} \\ c_{2,0} \\ \vdots \\ c_{127,0} \end{bmatrix} = \begin{bmatrix} a_{0,0} \\ a_{1,0} \\ a_{2,0} \\ \vdots \\ a_{127,0} \end{bmatrix} \times \begin{bmatrix} b_{0,0} \\ b_{1,0} \\ b_{2,0} \\ \vdots \\ b_{127,0} \end{bmatrix}$$
$$+ \begin{bmatrix} a_{0,1} \\ a_{1,1} \\ a_{2,1} \\ \vdots \\ a_{127,1} \end{bmatrix} \times \begin{bmatrix} b_{0,0} \\ b_{1,0} \\ b_{2,0} \\ \vdots \\ b_{127,0} \end{bmatrix}$$
$$+ \quad \vdots$$
$$+ \begin{bmatrix} a_{0,127} \\ a_{1,127} \\ a_{2,127} \\ \vdots \\ a_{127,127} \end{bmatrix} \times \begin{bmatrix} b_{0,0} \\ b_{1,0} \\ b_{2,0} \\ \vdots \\ b_{127,0} \end{bmatrix}$$
$$+ \begin{bmatrix} c_{0,0} \\ c_{1,0} \\ c_{2,0} \\ \vdots \\ c_{127,0} \end{bmatrix}$$

After executing these multiplications for all 128 pairs, we are left with a set of 128 intermediate 128×1 vectors. To derive the final result vector `Y`, we proceed by summing all these intermediate vectors element-wise. This summation process consolidates the 128 intermediate vectors into a single 128×1 result vector, which adheres to the specified data width of 24 bits.

In essence, this breakdown simplifies the complex 128×128 matrix-vector multiplication problem into a sequence of more manageable matrix-vector multiplications and subsequent summations. This approach facilitates the efficient hardware implementation of the operation while ensuring that the desired precision and format (24-bit) of the result vector `Y` are maintained throughout the process.

In the context of this project, we'll use the vector `F` to represent the result of the matrix `A` multiplied by vector `B`.

Now, let's move on to the addition step involving vector `C`. This operation is relatively straightforward. After obtaining the intermediate result vector `F` through the matrix-vector multiplication, we proceed to add vector `C` to it.

In essence, the final result vector `Y` is formed by adding the intermediate result vector `F` (which is the outcome of the matrix-vector multiplication) to vector `C`. This step effec-

tively combines the contributions of both the matrix-vector multiplication and the vector addition, resulting in the desired 128×1 column vector `Y`, which adheres to the specified data width of 24 bits.

2.3.2 Matrix – Accumulator

To achieve the matrix-vector multiplication and addition operations efficiently, I will employ a fundamental hardware component known as a Multiply-Accumulator or MAC unit. The MAC unit is a specialized arithmetic unit designed for precisely this type of computation, where multiplication and addition are performed in a single operation.

In a typical MAC unit, you have a multiplier that takes two operands, multiplies them, and feeds the result into an accumulator. The accumulator then accumulates or adds up the multiplied values over multiple clock cycles, allowing for iterative accumulation and precise calculation of the final result.

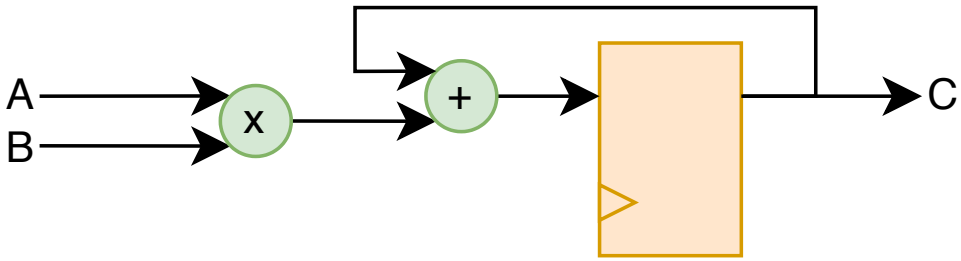

Figure 2.1: Multiply – Accumulator (MAC)

However, in this project, I will make a slight modification to the traditional MAC unit design. Instead of using a single multiplier, I will employ two separate multipliers. These two multipliers will take the necessary input operands, perform their respective multiplications, and then the results will be summed up. This sum, representing the accumulation of the products from both multipliers, will be passed into the accumulator unit.

This modification introduces parallelism into the operation, allowing for the simultaneous processing of multiple elements in the matrix-vector multiplication. By using two multipliers, I can enhance the throughput of the computation, potentially reducing the number of clock cycles required to complete the entire operation.

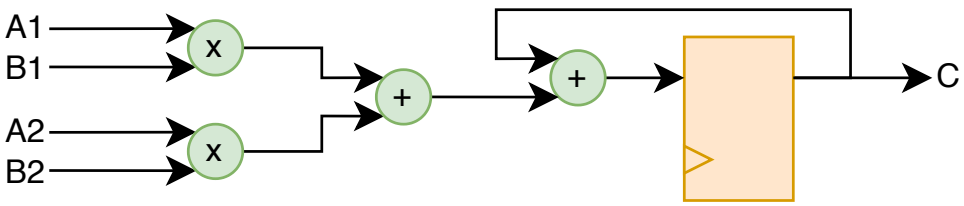

Figure 2.2: Modified Multiply – Accumulator

2.3.3 Two-Port RAM/ROM

In this project, we will employ two-port RAMs (Random Access Memory) as a key component of our design. These specialized memory units have two independent read ports,

which means they can simultaneously access two distinct memory locations, making them ideal for applications where multiple data accesses need to occur concurrently.

However, in our specific application, we will repurpose these two-port RAMs to function as ROMs (Read-Only Memories). A ROM is a type of memory that stores data that cannot be modified or written to during normal operation, making it perfect for storing constant values like the matrix `A`, vector `B`, and vector `C` in our matrix multiplication and addition operation.

To adapt the two-port RAMs into ROMs, we will simply disable the write enable (`WE`) signal. The dual-port capability of these ROMs will be invaluable for our application, as it will allow us to read two distinct values from two different memory addresses at the same time. This concurrent access will significantly enhance the efficiency of our hardware implementation, as we can fetch two data in parallel.

Figure 2.3: Two-Port ROM

To further enhance the parallelization of our hardware design, we have the option to instantiate additional ROMs for matrix `A` and vector `B`. This approach allows us to increase the degree of parallelism in our computations, potentially reducing the overall computation time and improving performance. However, it's crucial to keep in mind the limitations and constraints of the FPGA device resources.

2.3.4 ROM addressing

We know that the matrix A is 128×128 and vector B is 128×1, and that the ROM is capable of addressing 2 values at once, the opportunity arises for 2 values to be gathered from both matrix A and vector B in a single cycle. Importantly, this naturally aligns with the fact that 2 values from vector B can be the multiplicands for 2 values from matrix A.

When `addr_a = 0` and `addr_b = 1`,

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & \cdots & a_{0,126} & a_{0,127} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,126} & a_{1,127} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,126} & a_{2,127} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{127,0} & a_{127,1} & a_{127,2} & a_{127,3} & \cdots & a_{126,126} & a_{127,127} \end{bmatrix} \times \begin{bmatrix} b_{0,0} \\ b_{1,0} \\ b_{2,0} \\ b_{3,0} \\ \vdots \\ b_{126,0} \\ b_{127,0} \end{bmatrix}$$

When `addr_a = 2` and `addr_b = 3`,

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & \cdots & a_{0,126} & a_{0,127} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,126} & a_{1,127} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,126} & a_{2,127} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{127,0} & a_{127,1} & a_{127,2} & a_{127,3} & \cdots & a_{126,126} & a_{127,127} \end{bmatrix} \times \begin{bmatrix} b_{0,0} \\ b_{1,0} \\ b_{2,0} \\ b_{3,0} \\ \vdots \\ b_{126,0} \\ b_{127,0} \end{bmatrix}$$

When `addr_a = 126` and `addr_b = 127`, all values of the first row are traversed, at this point, the first value of vector `F` should be calculated.

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & \cdots & a_{0,126} & a_{0,127} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,126} & a_{1,127} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,126} & a_{2,127} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{127,0} & a_{127,1} & a_{127,2} & a_{127,3} & \cdots & a_{126,126} & a_{127,127} \end{bmatrix} \times \begin{bmatrix} b_{0,0} \\ b_{1,0} \\ b_{2,0} \\ b_{3,0} \\ \vdots \\ b_{126,0} \\ b_{127,0} \end{bmatrix}$$

When `addr_a = 128` and `addr_b = 129`, and considering that the addresses for ROM B are represented using only 7 bits, the addresses for ROM B will roll over to `0` and `1`.

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & \cdots & a_{0,126} & a_{0,127} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,126} & a_{1,127} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,126} & a_{2,127} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{127,0} & a_{127,1} & a_{127,2} & a_{127,3} & \cdots & a_{126,126} & a_{127,127} \end{bmatrix} \times \begin{bmatrix} b_{0,0} \\ b_{1,0} \\ b_{2,0} \\ b_{3,0} \\ \vdots \\ b_{126,0} \\ b_{127,0} \end{bmatrix}$$

ROM addressing for 32 ROMs

Using a single ROM `A` wouldn't be considered optimized for the parallelization of the calculation due to its limitation of having only two output ports. This limitation would result in the necessity of approximately 8192 cycles just to read all the values stored in the ROM, which is not conducive to the project's efficiency goals. To achieve the desired level of parallelism, it is recommended that multiple ROMs be employed. In this case, the decision has been made to use 32 ROMs, which significantly reduces the number of cycles needed to traverse all the values to a mere 256 cycles.

For addressing ROM `A`, a total of 14 bits are required. To optimize the addressing process, these 14 bits can be divided into three distinct parts:

1. The first part comprises the least significant 7 bits, which are used to address the 128 values within a row.
2. The second part is the next 5 bits, serves the purpose of selecting one of the 32 rows. In this project, 32 ROMs have been chosen to tie these bits to the corresponding rows.

3. The last 2-bit part is designated for selecting one of the four 32-row sections, allowing for efficient access to the desired data.

The addressing process has been optimized, and the number of cycles required to access all the necessary values has been minimized. This approach aligns with the goal of achieving efficient parallelization and enhanced performance in the hardware design.

	7'd0	7'd1	7'd2	7'd3	⋯	7'd126	7'd127
{2'd0,5'd00}	$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	⋯	$a_{0,126}$	$a_{0,127}$
{2'd0,5'd01}	$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	⋯	$a_{1,126}$	$a_{1,127}$
{2'd0,5'd02}	$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	⋯	$a_{2,126}$	$a_{2,127}$
⋮	⋮	⋮	⋮	⋮	⋱	⋮	⋮
{2'd0,5'd31}	$a_{31,0}$	$a_{31,1}$	$a_{31,2}$	$a_{31,3}$	⋯	$a_{31,126}$	$a_{31,127}$
{2'd1,5'd00}	$a_{32,0}$	$a_{32,1}$	$a_{32,2}$	$a_{32,3}$	⋯	$a_{32,126}$	$a_{32,127}$
{2'd1,5'd01}	$a_{33,0}$	$a_{33,1}$	$a_{33,2}$	$a_{33,3}$	⋯	$a_{33,126}$	$a_{33,127}$
{2'd1,5'd02}	$a_{34,0}$	$a_{34,1}$	$a_{34,2}$	$a_{34,3}$	⋯	$a_{34,126}$	$a_{34,127}$
⋮	⋮	⋮	⋮	⋮	⋱	⋮	⋮
{2'd1,5'd31}	$a_{63,0}$	$a_{63,1}$	$a_{63,2}$	$a_{63,3}$	⋯	$a_{63,126}$	$a_{63,127}$
{2'd2,5'd00}	$a_{64,0}$	$a_{64,1}$	$a_{64,2}$	$a_{64,3}$	⋯	$a_{64,126}$	$a_{64,127}$
{2'd2,5'd01}	$a_{65,0}$	$a_{65,1}$	$a_{65,2}$	$a_{65,3}$	⋯	$a_{65,126}$	$a_{65,127}$
{2'd2,5'd02}	$a_{66,0}$	$a_{66,1}$	$a_{66,2}$	$a_{66,3}$	⋯	$a_{66,126}$	$a_{66,127}$
⋮	⋮	⋮	⋮	⋮	⋱	⋮	⋮
{2'd2,5'd31}	$a_{95,0}$	$a_{95,1}$	$a_{95,2}$	$a_{95,3}$	⋯	$a_{95,126}$	$a_{95,127}$
{2'd3,5'd00}	$a_{96,0}$	$a_{96,1}$	$a_{96,2}$	$a_{96,3}$	⋯	$a_{96,126}$	$a_{96,127}$
{2'd3,5'd01}	$a_{97,0}$	$a_{97,1}$	$a_{97,2}$	$a_{97,3}$	⋯	$a_{97,126}$	$a_{97,127}$
{2'd3,5'd02}	$a_{98,0}$	$a_{98,1}$	$a_{98,2}$	$a_{98,3}$	⋯	$a_{98,126}$	$a_{98,127}$
⋮	⋮	⋮	⋮	⋮	⋱	⋮	⋮
{2'd3,5'd31}	$a_{127,0}$	$a_{127,1}$	$a_{127,2}$	$a_{127,3}$	⋯	$a_{126,126}$	$a_{127,127}$

2.3.5 High Fan-out Ports

In FPGA-based designs, one common challenge that arises when dealing with high fan-out is the issue of signal propagation delay. High fan-out occurs when a single signal needs to be distributed to a large number of destinations, such as when connecting the output of a ROM to multiple multipliers, as mentioned in the example.

When a signal is directly connected to a large number of destination points, it can lead to longer signal paths and increased propagation delay. This can result in timing issues, where some components receive the signal later than others, potentially causing synchronization problems and affecting the overall functionality of the design.

To mitigate the challenges posed by high fan-out, one effective strategy is to use registers as buffers. Registers act as intermediate storage elements that can isolate the high fan-out signal from the original source, allowing it to be distributed more evenly and efficiently to its multiple destinations. In essence, the signal is first loaded into a register, and then the copies of the signal are distributed from the registers to the respective destinations. But there is a catch is that it would consume one clock cycle.

In the case of the ROM `B` output connected to 12 multipliers, using registers as buffers

can help alleviate the high fan-out issue. Instead of connecting the ROM output directly to the multipliers, you can connect it to a set of registers. Each register can then feed its value to a group of multipliers. For example, you can connect the ROM output to 4 registers, and each of these registers can be connected to 3 multipliers. This distribution strategy effectively divides the high fan-out signal into smaller groups, reducing the load on each register and ensuring a more balanced and controlled distribution of the signal.

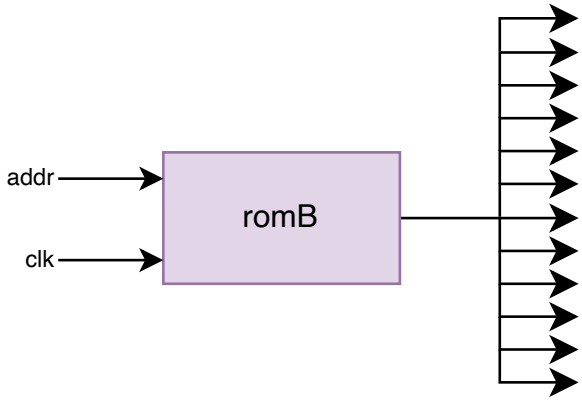

Figure 2.4: High Fan-Out Issue

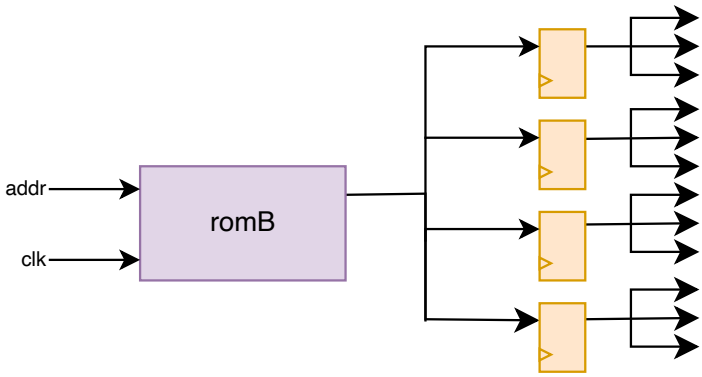

Figure 2.5: High Fan-Out Fixed with buffer registers

3. Hardware Design

3.1 RAM Controller

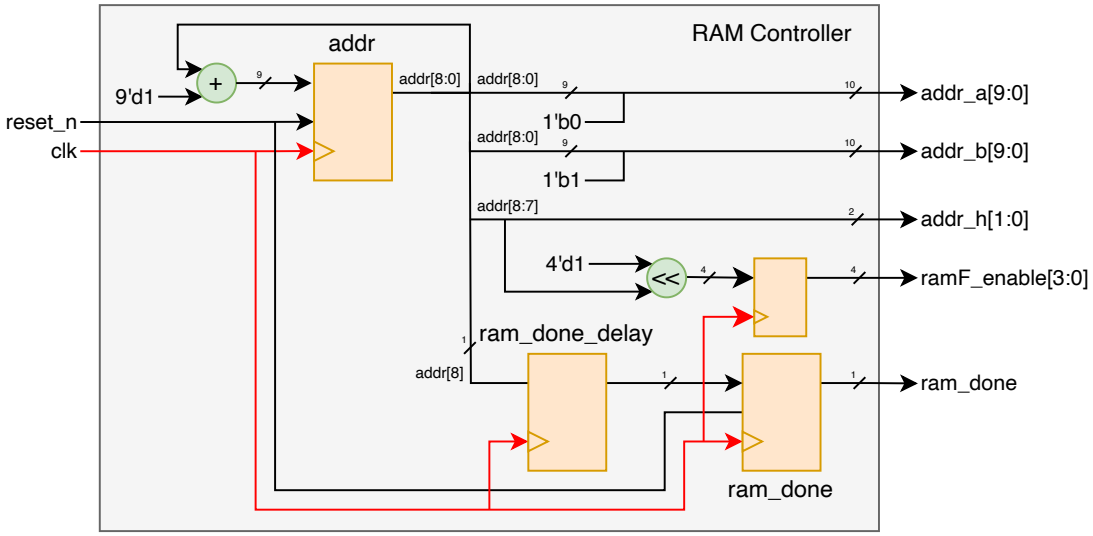

Figure 3.1: RAM Controller Block Diagram

The module incorporates two critical internal registers that play pivotal roles in its functionality:

- `addr`: This internal register serves as the backbone of the RAM Controller. It meticulously keeps track of the current address being accessed. Specifically, `addr` represents the upper part of the `addr_a` and `addr_b` signals.
- `ram_done_delay`: This register introduces a one-cycle delay for the `ram_done` signal. By doing so, it ensures that the `ram_done` signal remains synchronized with other components in the top module.

The RAM Controller module provides several crucial output signals that play integral roles in the overall operation:

- `addr_a` and `addr_b`: These output signals are responsible for representing the addresses used to access ROMs A, B and C. The differentiation between `addr_a` and `addr_b` is based on whether the least significant bit (LSB) of these addresses is 0 (for `addr_a`) or 1 (for `addr_b`). This allows us to get two values from two addresses at the same time.
- `ram_done`: When asserted, it signals that all addresses have been traversed, marking the completion of the matrix - vector multiplication and addition operation.
- `ramF_enable`: This multi-bit output signal holds a vital role in enabling write operations to the `ramF` (result accumulator).
- `addr_h`: The `addr_h` signal represents the high-order address bits, they determine which part of ROMs will be accessed during the matrix-vector multiplication and addition process.

3.2 Matrix Multiplier

The module performs a 1×2 matrix multiplication with a 2×1 matrix, resulting in a 1×1 matrix (scalar value).

$$\begin{bmatrix} a_{00} & a_{01} \end{bmatrix} \times \begin{bmatrix} b_{00} \\ b_{10} \end{bmatrix} = [a_{00}b_{00} + a_{01}b_{10}] = [c_{00}]$$

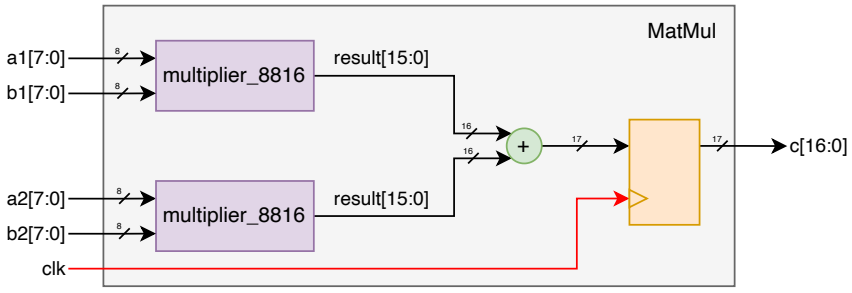

Figure 3.2: MatMul Block Diagram

3.3 ROM C to registers

This module is responsible for loading values from ROM C module, into an array of registers, I called it RAM `C`.

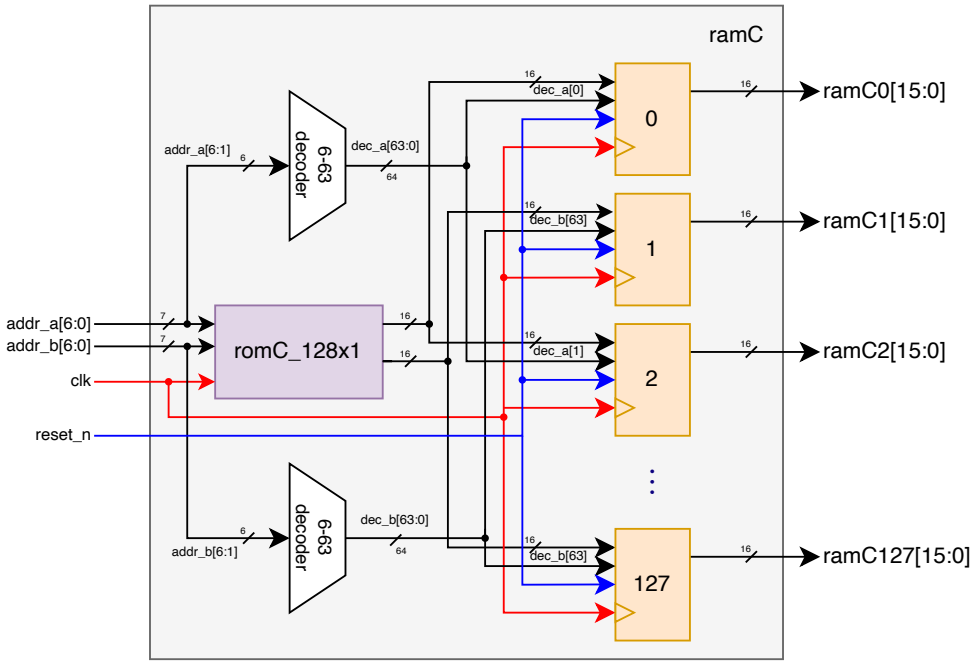

Figure 3.3: RAM `C` Block Diagram

3.4 Accumulator Block

The module `accum_8x32` is designed to perform accumulation operations on an array of 16-bit data values, which are the output of `MatMul`, represented as `accum_i`, and store the results in another array result. The `enable` signal in the module is used to control whether the accumulation operation should be performed. Its purpose is to enable or disable the accumulation process based on its state.

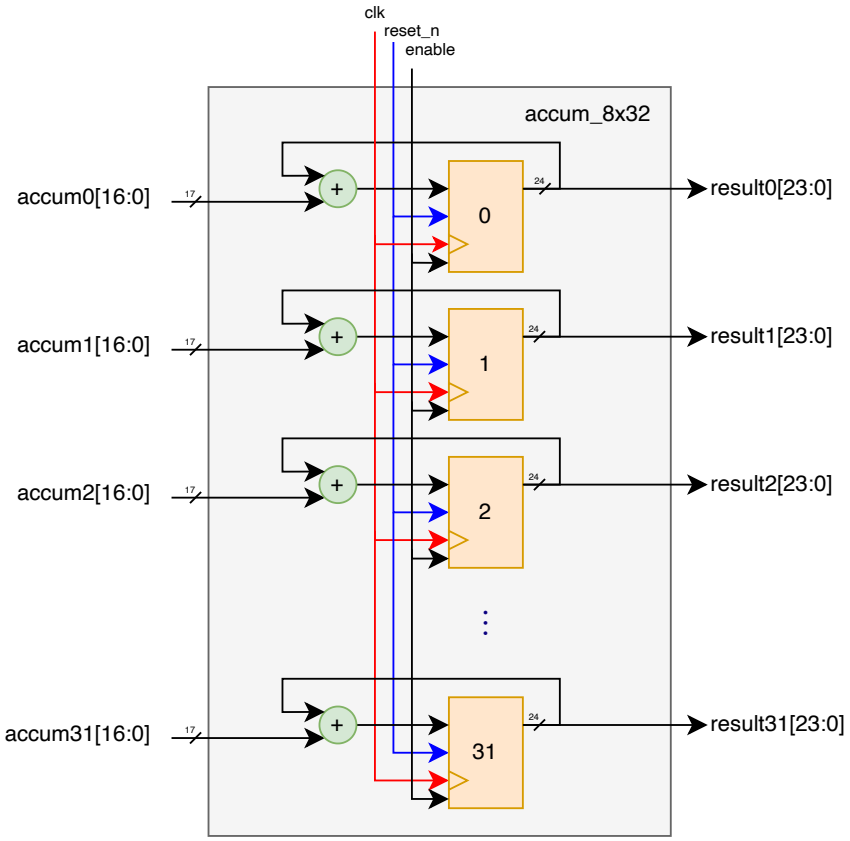

Figure 3.4: Accumulator Block Block Diagram

3.5 Adder Tree

An adder tree, also known as a carry-save adder tree, is a digital circuit used in computer arithmetic and digital signal processing to perform multi-bit addition quickly and efficiently. It is particularly useful for adding multiple binary numbers together in parallel.

The basic idea behind an adder tree is to break down a multi-bit addition problem into a series of smaller additions, which can be performed in parallel. Each stage of the adder tree reduces the number of bits in the operands while simultaneously preserving any carry bits generated during addition. The carry bits are then propagated up the tree to be added to the next stage.

The adder tree approach significantly reduces the critical path delay compared to performing 128 sequential additions. It also provides better performance and utilizes parallelism efficiently.

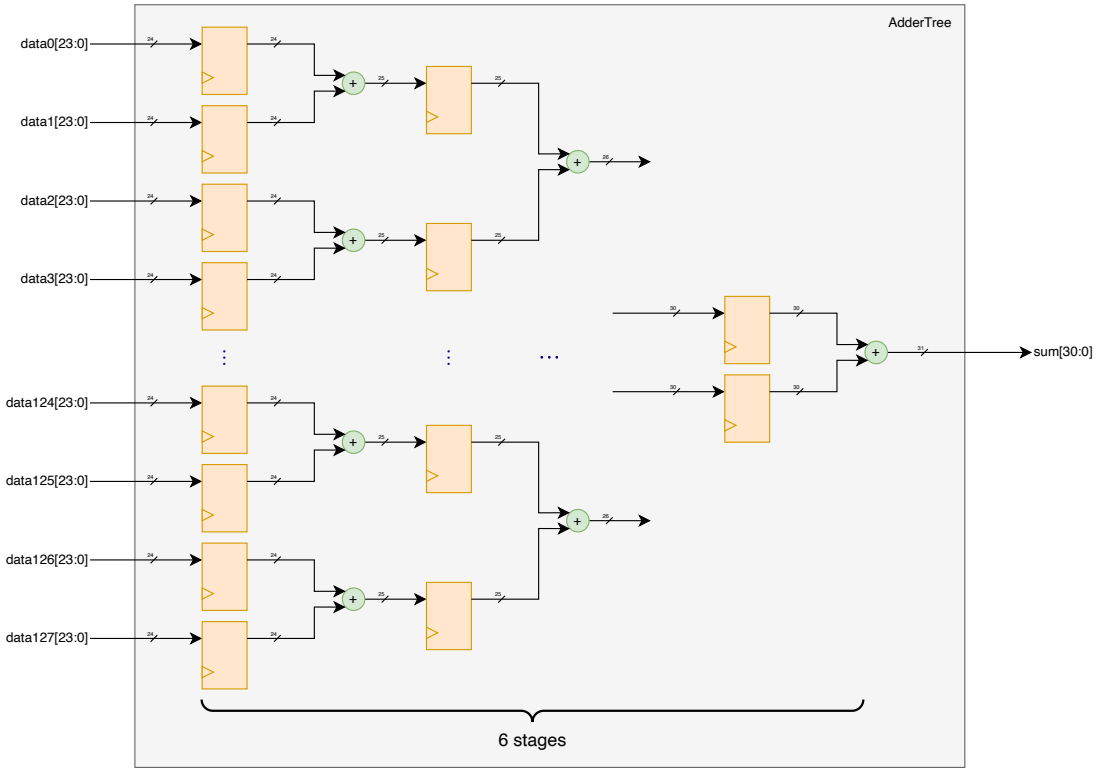

Figure 3.5: Adder Tree Block Diagram

3.6 Main Module

3.6.1 Combinational Modules

In the hardware design, the capabilities of 32 ROM `A` blocks and 32 `MatMul` units have been harnessed to maximize parallelization. This results in the processing of a total of 32 values concurrently in each cycle. This level of parallelism has been made possible by the utilization of 2-port ROMs, allowing two values to be accessed from each ROM simultaneously. As a result, the first 32 values of RAM `F` can be efficiently calculated in just 64 cycles.

Following this initial computation, the remaining calculations for the entire RAM `F` are performed. With the foundation laid by the first 32 values, the processing is extended to cover the full range of 128 values in RAM `F`. An additional 192 cycles are required for this, resulting in a total of 256 cycles to compute the entirety of RAM `F`.

Once the complete RAM `F` has been obtained, the next step in the computation involves the addition of RAM `C`. By combining the results of RAM `C` and RAM `F`, the final `Y` vector is generated.

Finally, the pivotal stage of the computation is reached, where the need arises to sum up the 128 values stored in RAM `Y`. This is where the Adder Tree comes into play. With careful design and parallel processing, the sum of these 128 values is efficiently computed, and this task is achieved within 6 cycles.

The module will output the sum of 128 values of vector `Y`, the number of cycles that is needed for the calculation, and the `done` signal indicating the process has been done.

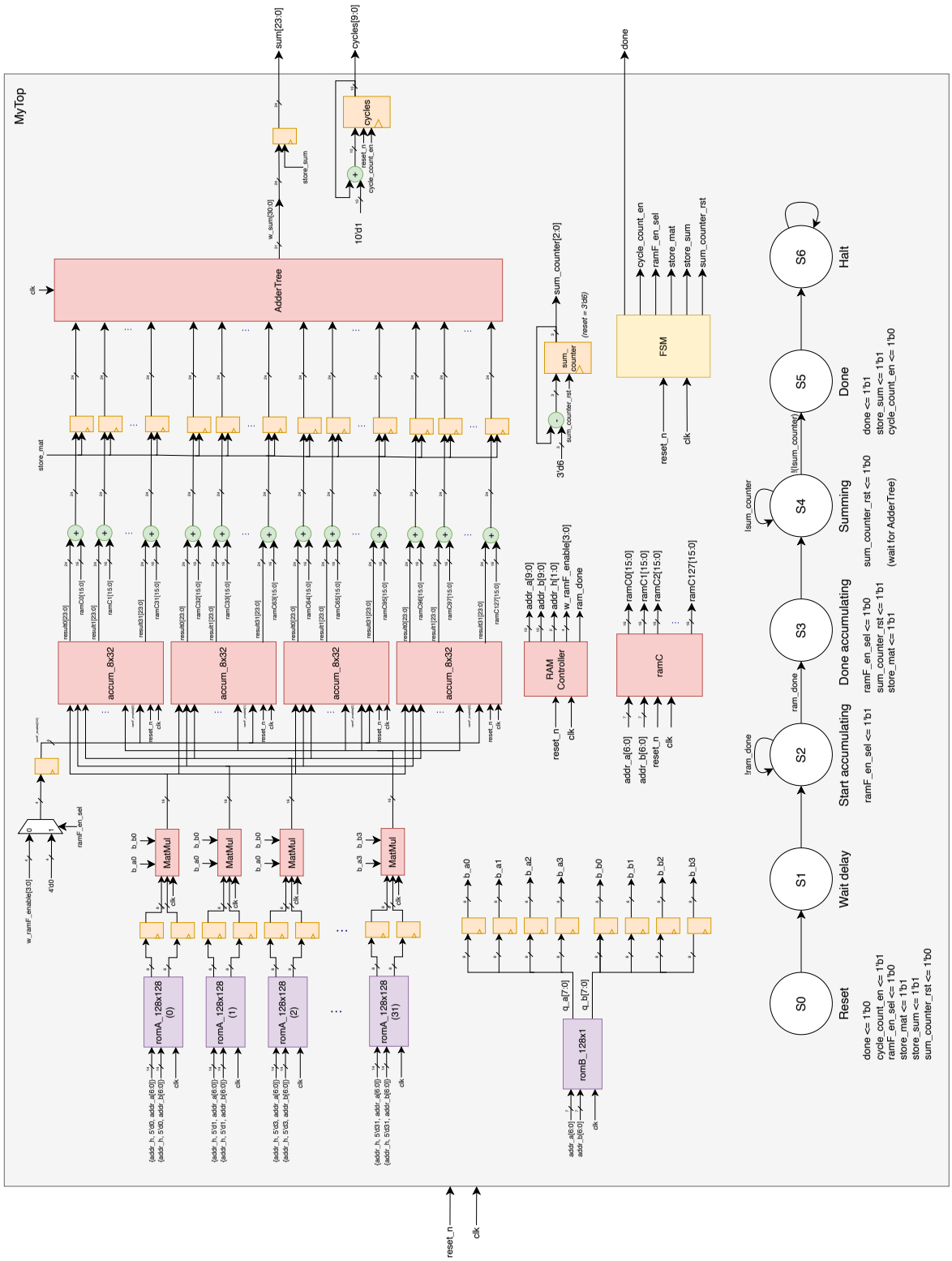

Figure 3.6: Main Module Block Diagram

3.6.2 State Machines

`S1`: **Reset and Initialization:** In this initial state, all control registers are reset, and the cycle counter is enabled. It marks the beginning of the operation, setting the stage for subsequent phases.

`S2`: **Delay for Fan-Out Registers:** The FSM transitions to this state to allow for a delay in the fan-out registers. This step is essential for ensuring proper synchronization within the design.

`S3`: **Start Accumulating:** This state initiates the accumulation process, beginning with the accumulation of the first 32 values. It continues to accumulate values in this state unless the `ram_done` signal is asserted by the RAM Controller, indicating that it has completed traversing all 128×128 values.

`S4`: **Done Accumulating:** Upon completion of the accumulation phase, this state is entered. It is responsible for asserting signals that trigger the start of the summing process for all 128 values.

`S5`: **Wait for Summing in the Adder Tree:** In this state, the FSM pauses and waits for a specific duration, which is six cycles, to allow for the summing operation within the Adder Tree to conclude.

`S6`: **Operation Completion:** Upon successful completion of the summing process in the Adder Tree, this state is entered. It is responsible for asserting the "done" signal, storing the sum produced from the Adder Tree, and stopping the cycle counter. This marks the end of the operation, and the desired result is obtained.

`S7`: **Halt:** The final state serves as a halt state, where the operation has concluded, and the FSM remains idle. It is a standby state, signifying the end of the active operation.

3.6.3 Notes

The question may arise: "Why is there a need to calculate all 128 values of vector `Y` at the end, instead of accumulating them during the calculation process?" The answer lies in the fundamental objective of this project, which is focused on matrix multiplication.

Accumulating the values of vector `Y` during the calculation process would indeed be an option, but it would detract from the core purpose of matrix multiplication. The essence of matrix multiplication involves the systematic multiplication and accumulation of individual elements to produce a resultant vector. By calculating all 128 values separately and subsequently summing them, we adhere to this fundamental mathematical operation.

Due to these hardware constraints, it may not be feasible to visualize and display all the intermediate values as they are calculated. Therefore, the approach of accumulating the values at the end allows for a final confirmation of the result of the matrix multiplication process.

3.7 Top Module (ChipInterface)

This module serves as an interface between input signals, a calculation module named MyTop, and a seven-segment display controller called SevenSegmentControl.

The inputs to ChipInterface include a clock signal (`CLOCK_50`), a set of switches (`SW`), and a set of keys or buttons (`KEY`). It also handles clock synchronization and reset logic by connecting `CLOCK_50` to an internal clock signal (`clk`) and `KEY[0]` to an internal reset signal (`reset_n`).

ChipInterface connects the output signals HEX0 through HEX5 to a seven-segment display, displaying values based on the state of switches, either the sum values or cycles.

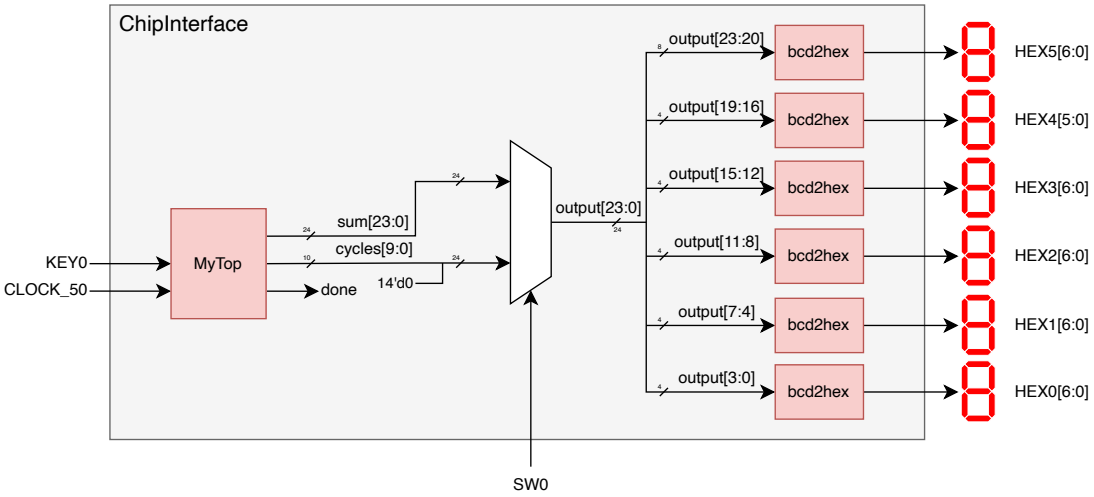

Figure 3.7: ChipInterface Block Diagram

4. Result

4.1 Synthesis Result

Fitter Status	Successful - Sun Oct 1 13:59:07 2023
Quartus Prime Version	22.1std.0 Build 915 10/25/2022 SC Standard Edition
Revision Name	lab2
Top-level Entity Name	ChipInterface
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	6,452 / 41,910 (15 %)
Total registers	12707
Total pins	45 / 499 (9 %)
Total virtual pins	0
Total block memory bits	4,197,376 / 5,662,720 (74 %)
Total RAM Blocks	514 / 553 (93 %)
Total DSP Blocks	32 / 112 (29 %)
Total HSSI RX PCSs	0 / 9 (0 %)
Total HSSI PMA RX Deserializers	0 / 9 (0 %)
Total HSSI TX PCSs	0 / 9 (0 %)
Total HSSI PMA TX Serializers	0 / 9 (0 %)
Total PLLs	0 / 15 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 4.1: Resources Usage

Slow 1100mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	169.92 MHz	169.92 MHz	CLOCK_50	

Figure 4.2: F_{max} result for Slow 1100mV 85C Model

4.2 Simulation Problem

When simulating a design that includes multiple block RAMs instantiated as part of the FPGA implementation, the conversion from the `.mif` (Memory Initialization File) format to the simulation-friendly format (typically `.ver` or equivalent) can indeed be time-consuming, especially when you have 32 block RAMs to initialize. To address this issue and speed up the simulation process, there are two common approaches: using Synopsys VCS, or using modeled Block RAMs.

In modeled Block RAMs approach, I create Verilog modules that model the behavior of block RAMs without the need for `.mif` files. These modeled block RAMs can be instantiated in simulation testbench and can provide initial data or mimic the behavior of the actual block RAMs. This approach has the advantage of eliminating the need for `.mif` to `.v` conversion and can significantly reduce simulation startup time.

```
1 `ifdef SIMULATION
2 module romA_128x128$ (
3     input [13:0] address_a,
4     input [13:0] address_b,
```

```
    input clock,
    output reg [7:0] q_a,
    output reg [7:0] q_b
);

reg [7:0] rom_data [0:16383]; // 128*128 ROM, 16K 8-bit words

initial begin
    $readmemh("matA.hex", rom_data);
end

always @(posedge clock) begin
    // Read from Port A
    q_a <= rom_data[address_a];

    // Read from Port B
    q_b <= rom_data[address_b];
end
endmodule
`else
// Altera RAM/ROM
`endif
```

To use this, the `SIMULATION` macro should be defined. Also, you need to convert the `.mif` file into `.hex` file. The Python script to convert will be attached in the zip file.

4.3 Simulation and Hardware Result

Two sets of pre-generated ROM data are available, and their corresponding results are 24-bit values: `24'h206D3D` and `24'h9409CF`. To validate the correctness of this data, I developed a Python script that decodes the `.mif` file and recalculates the values independently. The script is included in the provided zip file.

4.3.1 Matrices Set 1

```
@2685: Finished!
@2685: The sum is 206d3d
@2685: Total cycles: 10c

@3695: Reset!! The current sum is 000000

@6465: Finished!
@6465: The sum is 206d3d
@6465: Total cycles: 10c
** Note: $stop
   Time: 6475 ps  Iteration: 0  Instance: /TB
```

Figure 4.3: Simulation Result using QuestaSim for `mat1`

```
Chronologic VCS simulator copyright 1991-2018
Contains Synopsys proprietary information.
Compiler version O-2018.09-SP2-2_Full64; Runtime version O-2018.09-SP2-2_Full64;  Oct  1 22:59 2023
@2685: Finished!
@2685: The sum is 206d3d
@2685: Total cycles: 10c

@3695: Reset!! The current sum is 000000

@6465: Finished!
@6465: The sum is 206d3d
@6465: Total cycles: 10c
$stop at time 6475 Scope: TB File: TB.sv Line: 51
```

Figure 4.4: Simulation Result using Synopsys VCS for `mat1`

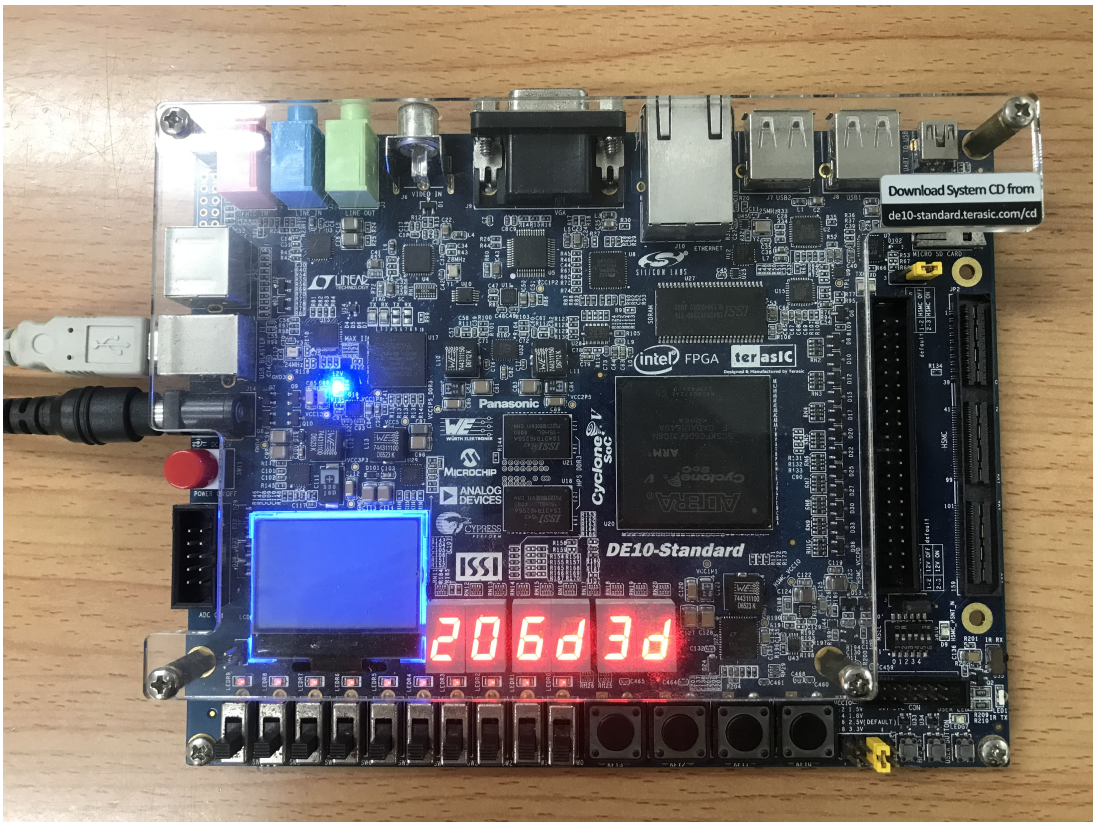

Figure 4.5: Hardware Result for `mat1`

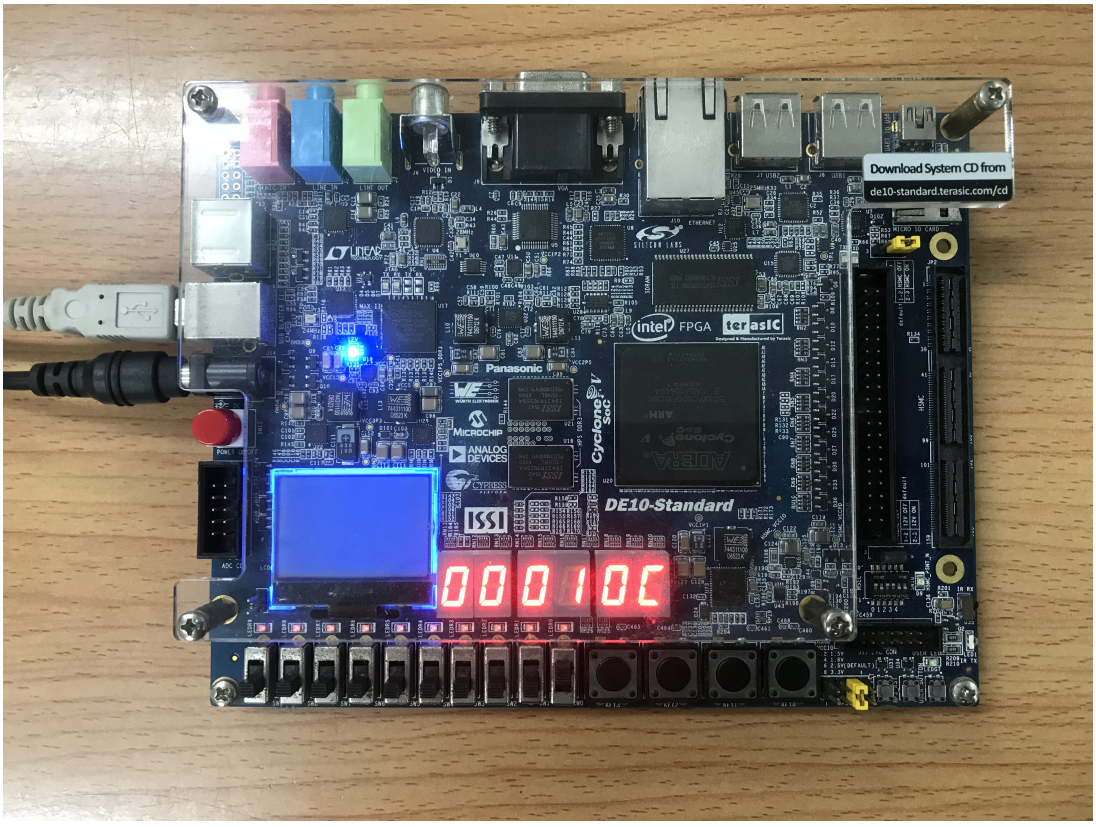

Figure 4.6: Hardware Result for total cycles used when switching `SW0` to high

The total number of cycles utilized for the calculation amounts to `0x10C`, as observed both in the simulation and the hardware results, equating to a total of 268 cycles.

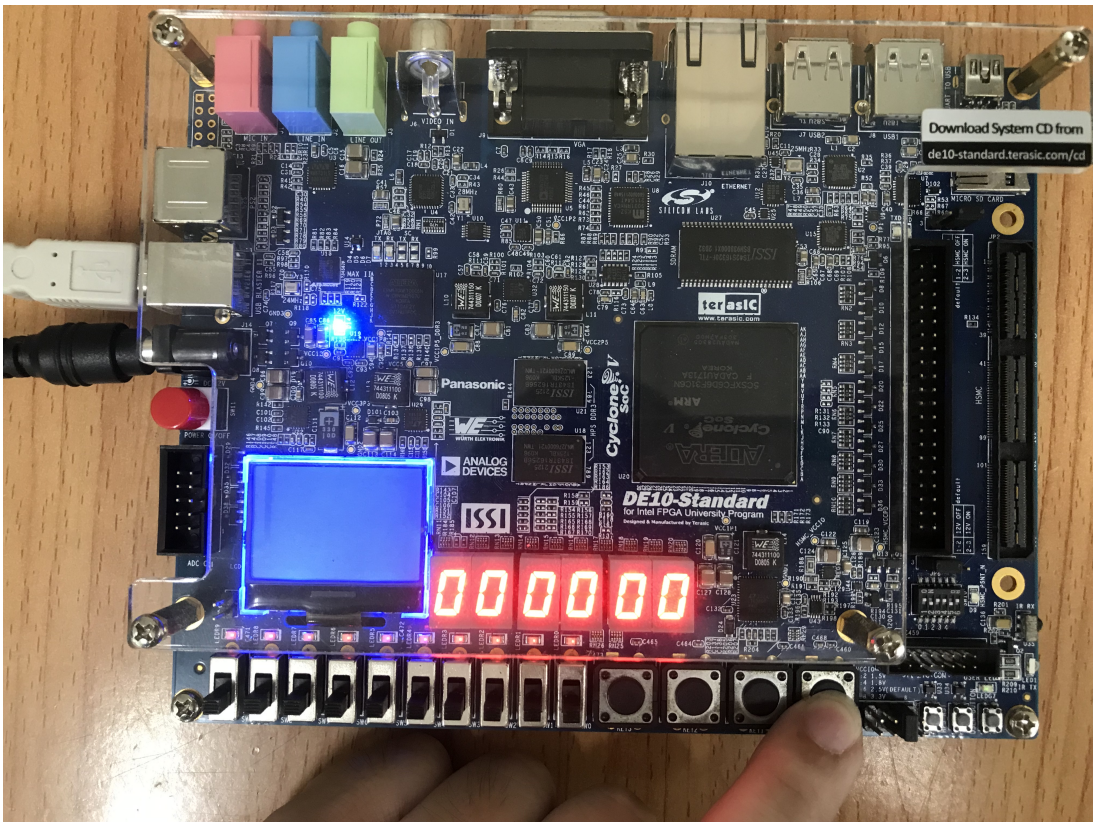

Figure 4.7: Hardware Result when holding the `KEY0` (reset)

4.3.2 Matrices Set 2

```
Chronologic VCS simulator copyright 1991-2018
Contains Synopsys proprietary information.
Compiler version O-2018.09-SP2-2_Full64; Runtime version O-2018.09-SP2-2_Full64;  Oct  1 23:14 2023
@2685: Finished!
@2685: The sum is 9409cf
@2685: Total cycles: 10c

@3695: Reset!! The current sum is 000000

@6465: Finished!
@6465: The sum is 9409cf
@6465: Total cycles: 10c
$stop at time 6475 Scope: TB File: TB.sv Line: 51
```

Figure 4.8: Simulation Result using Synopsys VCS for `mat2`

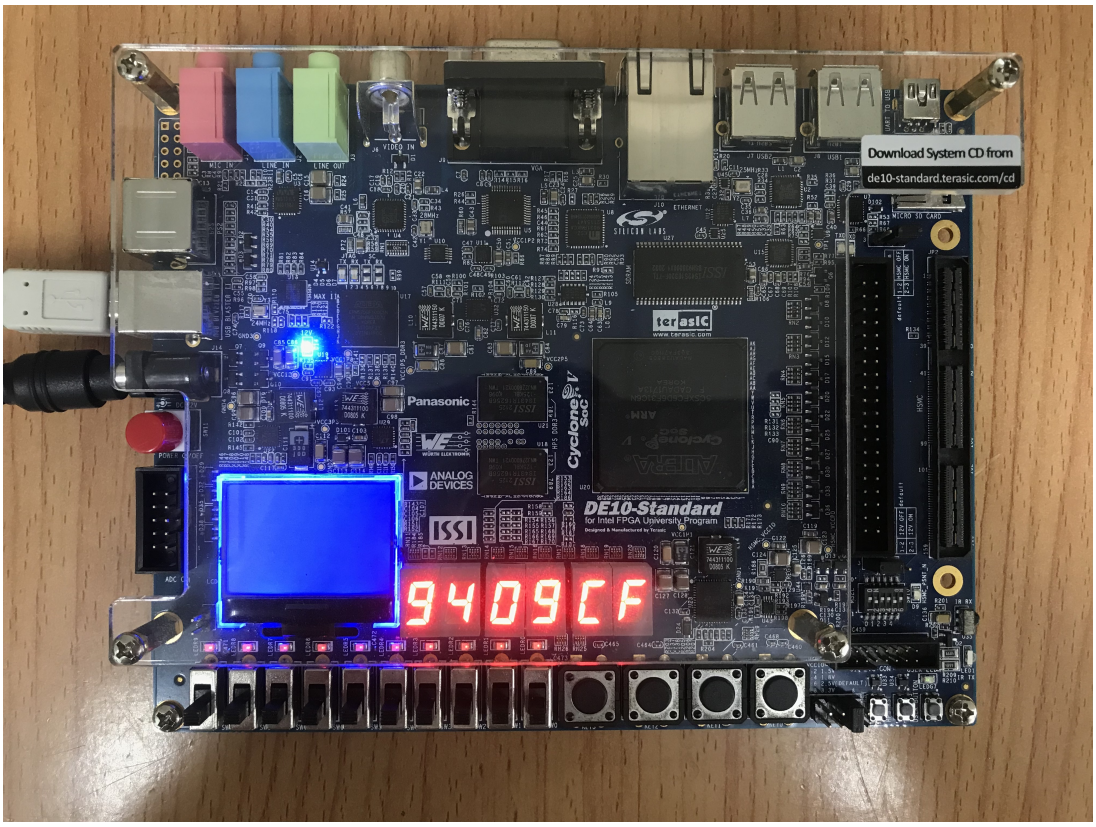

Figure 4.9: Hardware Result for `mat2`

Upon comparing the simulation results with the hardware results, it is evident that they both yield the same values as those mentioned above. This alignment between the computed values from the simulation and the hardware design serves as a strong confirmation of the correctness of the hardware implementation.

4.3.3 Matrices Set 3

To provide further confirmation of the correctness of the design, the `generate_matrix.py` script is used to generate a new random set of matrices. Subsequently, I conducted simulations and programmed the FPGA to observe the results obtained from this fresh data set.

```
$ python3 ./generate_matrix.py matA_3.mif matB_3.mif matC_3.mif
Matrix files created: matA_3.mif matB_3.mif matC_3.mif
Ab sum: 0x0edd5a54 (trunc dd5a54)
C sum: 0x0044b31c (trunc 44b31c)
MMA product: 0x0f220d70 (trunc 220d70)
```

Figure 4.10: Output of the Python script

```
Chronologic VCS simulator copyright 1991-2018
Contains Synopsys proprietary information.
Compiler version O-2018.09-SP2-2_Full64; Runtime version O-2018.09-SP2-2_Full64;  Oct  1 23:15 2023
@2685: Finished!
@2685: The sum is 220d70
@2685: Total cycles: 10c

@3695: Reset!! The current sum is 000000

@6465: Finished!
@6465: The sum is 220d70
@6465: Total cycles: 10c
$stop at time 6475 Scope: TB File: TB.sv Line: 51
```

Figure 4.11: Simulation Result using Synopsys VCS for `mat3`

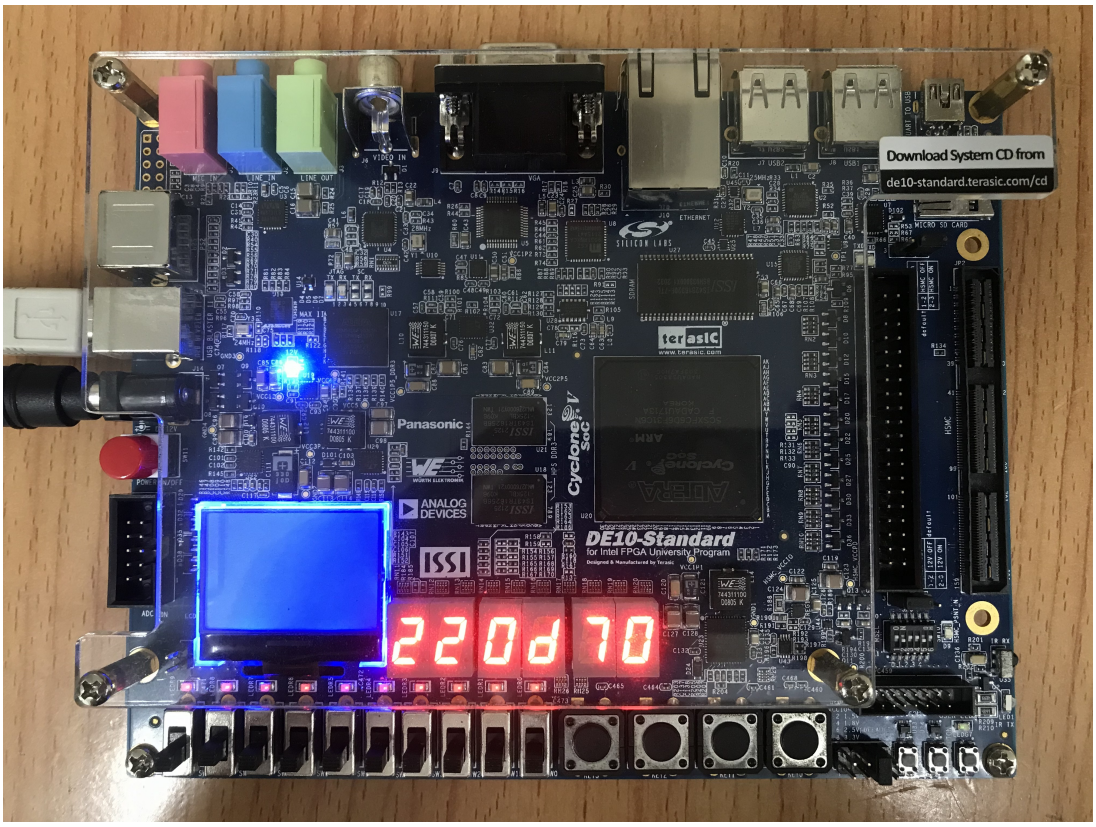

Figure 4.12: Hardware Result for `mat3`

5. Conclusion

In conclusion, this project has demonstrated the successful design and implementation of a hardware matrix multiplication system, leveraging FPGA's inherent capabilities. The primary objective of efficiently performing matrix-vector multiplication and accumulation has been realized, resulting in the generation of a 128-element vector. The design strategically utilized hardware features, including Multiply-Accumulate (MAC) units and dual-port Embedded RAMs, to optimize computational performance.

The project's execution followed a well-structured plan, with the matrix-vector multiplication process meticulously divided into multiple phases. Parallelization and resource utilization were at the forefront, leading to a significant reduction in computation time. A notable highlight was the incorporation of multiple of 2-port ROMs and matrix multipliers, which expedited simultaneous calculations through simultaneous data retrieval.

Comprehensive validation efforts, including simulations and hardware testing, consistently confirmed the correctness of the hardware design. Pre-generated ROM data and Python scripts were employed for verification, and further testing with new generated values reaffirmed the system's reliability.

In terms of performance, the hardware implementation successfully achieved the intended matrix multiplication and accumulation objectives while efficiently managing computational load. The total cycle count for the calculation, verified both in simulation and hardware, stood at 268 cycles.

Furthermore, it's noteworthy that the achieved maximum frequency (F_{max}) reached around 160MHz, showcasing the system's capability to handle medium-speed operations while maintaining accuracy and stability.