HO CHI MINH UNIVERSITY OF TECHNOLOGY

ELECTRICAL ELECTRONIC ENGINEERING

Electronic Department

----------o0o----------

LOGIC DESIGN / LOGIC SYNTHESIS

LAB 2

MATRIX MULTIPLICATION

INSTRUCTORS: Tran Hoang Linh

Nguyen Tuan Hung

STUDENT : Ha Gia Huy _ 2051117

HO CHI MINH CITY, 2024

Table of Content

List of Figure

I. Overview

The problem to solve is the matrix equation:

> Y(24-bit) = A(8-bit) * B(8-bit) + C(16-bit)

=> where A represents a 128x128 matrix composed of 8-bit integers, B is a 128x1 column vector containing 8-bit integers, C is a 128x1 column vector comprising 16-bit integers, and Y denotes a 128x1 column vector consisting of 24-bit integers. The choice of these specific integer widths is a deliberate consideration to prevent any loss of precision during intermediate calculations, as commonly encountered in matrix-accumulation operations. In the provided materials, A, B, and C are predetermined and static. During configuration, I will incorporate A, B, and C into my setup by transferring them onto the board. However, my system's design should accommodate arbitrary matrices of identical dimensions. The primary task of my system is to execute the Multiply-Accumulate (MAC) calculation. Rather than presenting a complex user interface to examine each element of the resulting matrix (Y), my system will streamline the process by computing the sum of all values within Y and displaying the resultant 24-bit sum.

My system is designed to display two different values on the hex displays based on the state of SW0. When SW0 is set to 1, the hex display will showcase the number of clock ticks required to complete the computation, represented in hexadecimal format. On the other hand, if SW0 is cleared to 0, the hex display will present the 24-bit result of the calculation, encoded as 6 hexadecimal numerals. This functionality provides flexibility in monitoring either the computational time or the actual result based on the user's preference.

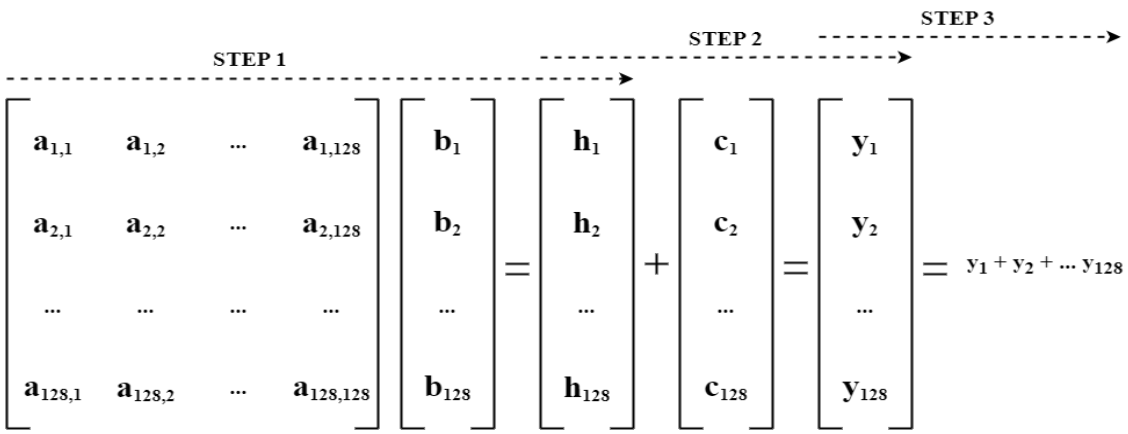

Figure 1. Flowchart

II. Theory

For matrix multiplication, the number of columns in the first matrix must be equal to the number of rows in the second matrix. The resulting matrix, known as the matrix product, has the number of rows of the first and the number of columns of the second matrix. The product of matrices A and B is denoted as H.

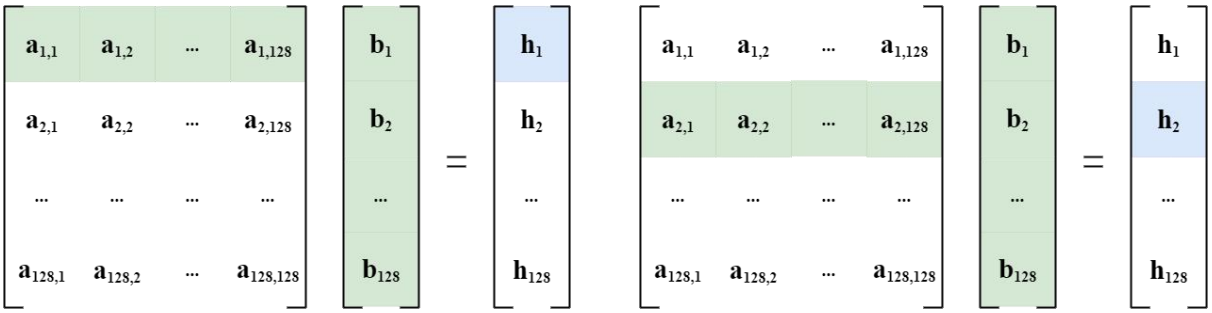

Figure 2. Matrix Multiplication

The sum of two matrices of the same order is a matrix obtained by adding together corresponding elements of the original two matrices. Matrix addition is commutative and associative.

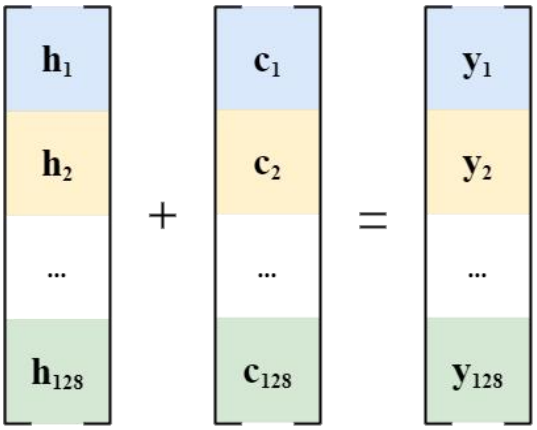

Figure 3. Matrix Addition

In computing, the multiply–accumulate (MAC) is a common step that computes the product of two numbers and adds that product to an accumulator. The hardware unit that performs the operation is known as a multiplier–accumulator (MAC unit); the operation itself is also often called a MAC operation. The MAC operation modifies an accumulator:

$$A <= A + (B * C)$$

III. Design Strategy

In this project, we are provided several modules and files (*romA_128x128.v, romB_128x1.v, romC_128x1.v, multiplier_8816.v, matA.mif, matB.mif, matC.mif*). Firstly, we execute matrix multiplication between romA and romB. Multiplying each rows of romA with romB. Rom modules operate based on clock signal to generate data by address (dual port). There are 128x128 = 16384 data in romA. Each clock, two data will be generated, so that we need 16384/2 = 8192 cycles to get all value in romA. More romA modules are integrated to design, more cycles will be optimized.

Number of romA	Number of cycles
1 (single port)	16384
1 (dual port)	8192
2 (dual port)	4096
4 (dual port)	2048
8 (dual port)	1024
16 (dual port)	512
32 (dual port)	256
64 (dual port)	128 (not enough resource on hardware)

Because increasing number romB reduce not much cycles and make design more complex, I use only one romB for execute with 32 romAs.

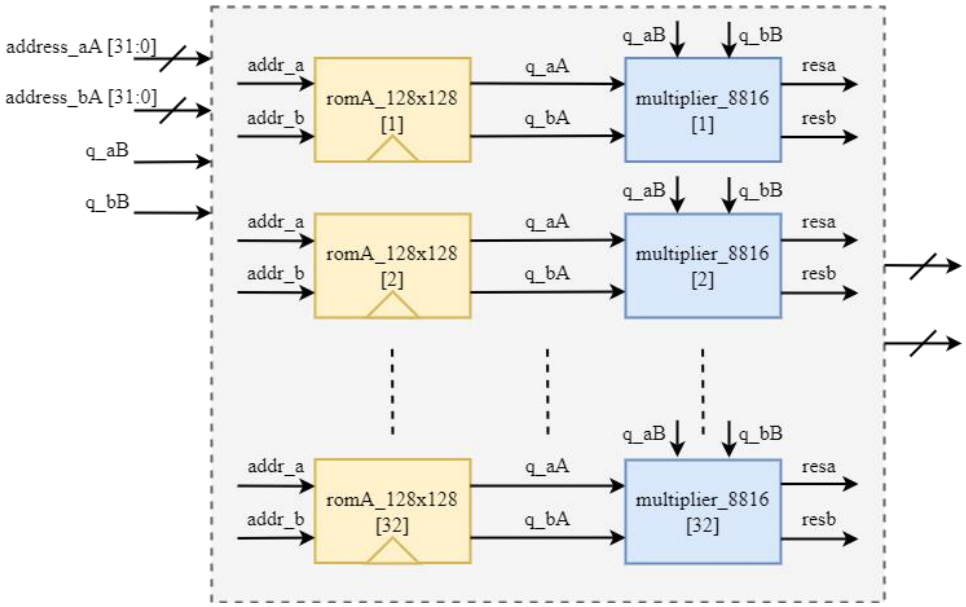

Figure 4. Multi-Multiplier

Multiply–Accumulate (MAC) module will hold and accumulate until multiplying 128 values in each rows of romA with romB for finish a term. And it will flush before going to new term.

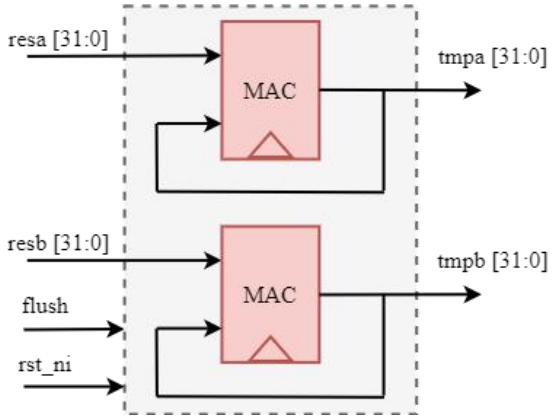

Figure 5. Multiply Accumulate

Data memory (DMEM) is an important module to store result of above multiplication. There are 128 registers in this memory. Each register can be store 16-bit value (the largest value of matrix multiplication of two 8-bit matrix is 16-bit).

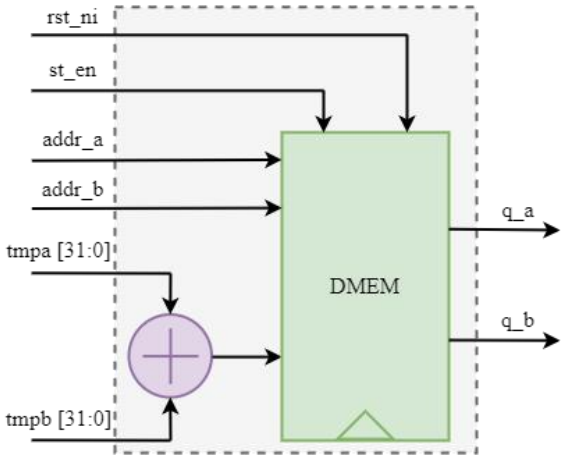

Figure 6. Data Memory

Control Unit is responsible for sending signals and controlling dataflow of system. My systems must be reset initially to set up data. More specifically, address_aA and address_aB start from 0 and increase 2 units every clocks (0, 2, 4, …) whereas address_bA and address_bB begin from 1 and increase 2 units every clocks (1, 3, 5, …). Moreover, because I used 32 romA as demonstrate above, I need 32 addresses respectively. Each of addresses to control romA away 32 units (0, 32, 64, 96,

128 …). Ultimately, another responsibility of Control Unit is active/inactive signals such as flush, st_en for MAC, DMEM and done for main system.

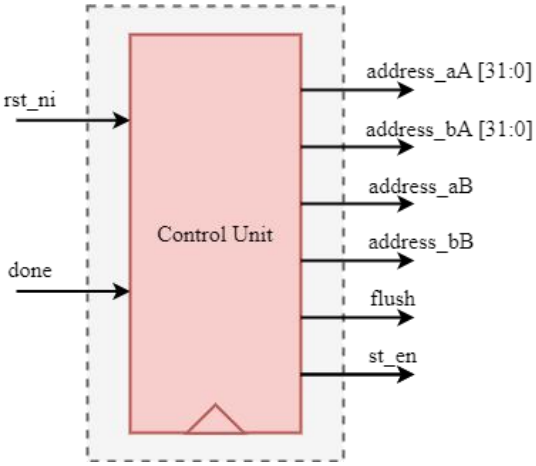

Figure 7. Control Unit

Finally, I calculate sum of elements in matix_AB and matrix_C. Designing a counter to count number of clock ticks starting from reset signal active and ending to done signal active. The data will be shown in HEX via ChipInterface module (provided and modified). This data will depend on SW0 (0 for result and 1 for clock ticks). KEY0 is reponsible for reset function.

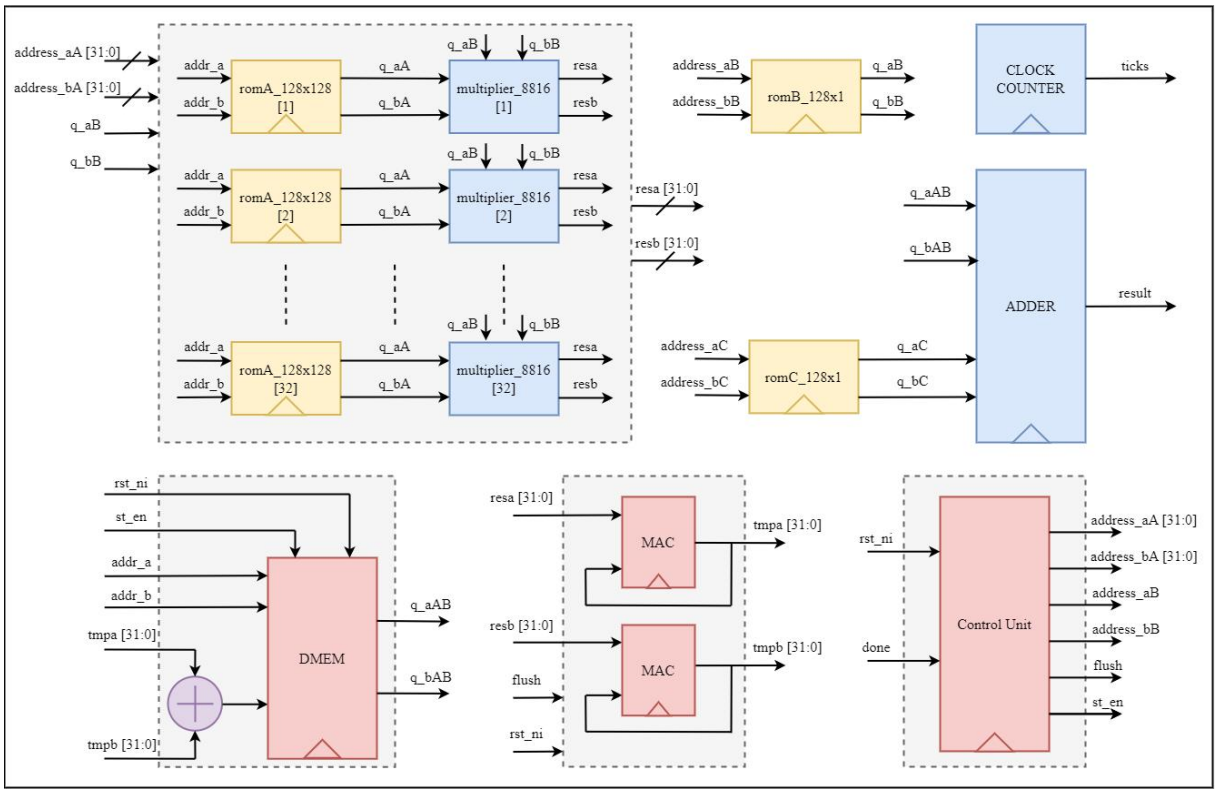

Figure 8. Block Diagram

IV. Verification Strategy

We use a provided python script *generate_matrix.py* to generate a random problem. Random problem generated from this python program also prints out the answer of Ab sum; C sum and MMA product.

```
D:\Lab\Lab2>python 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: 0x1008215c (trunc 08215c)
C sum: 0x003c150f (trunc 3c150f)
MMA product: 0x1044366b (trunc 44366b)
```

Figure 9. Random Matrix

After using VERILATOR simulator, we can check sumAB, sumC and MMA product, number of clock ticks. As a result, our design work correctly to generate expected outcomes.

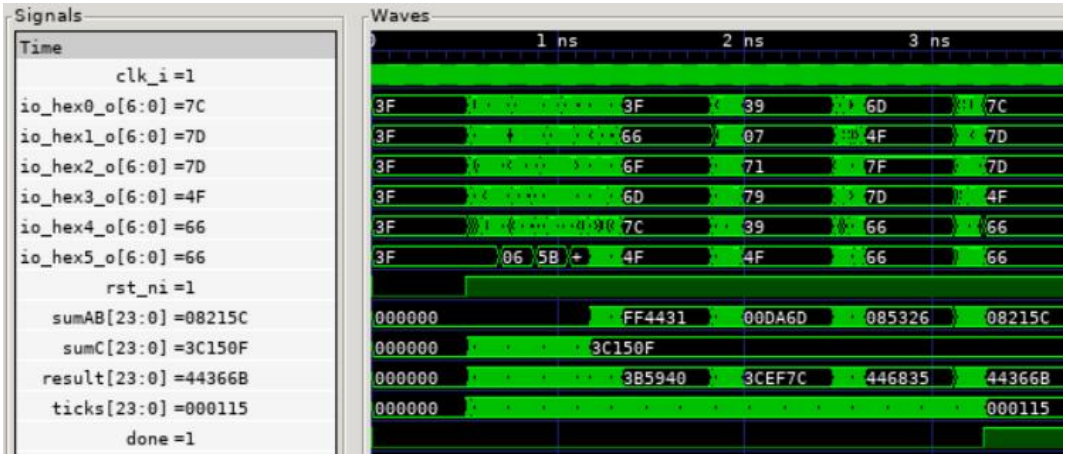

Figure 10. Verilator Simulation for Random Matrix

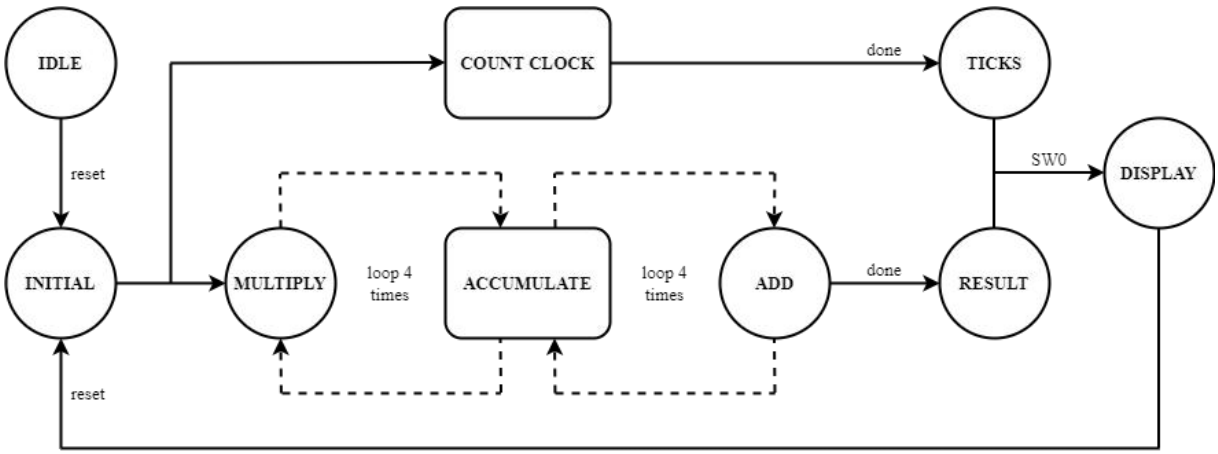

Figure 11. Finite State Machine

V. Simulating on Quartus

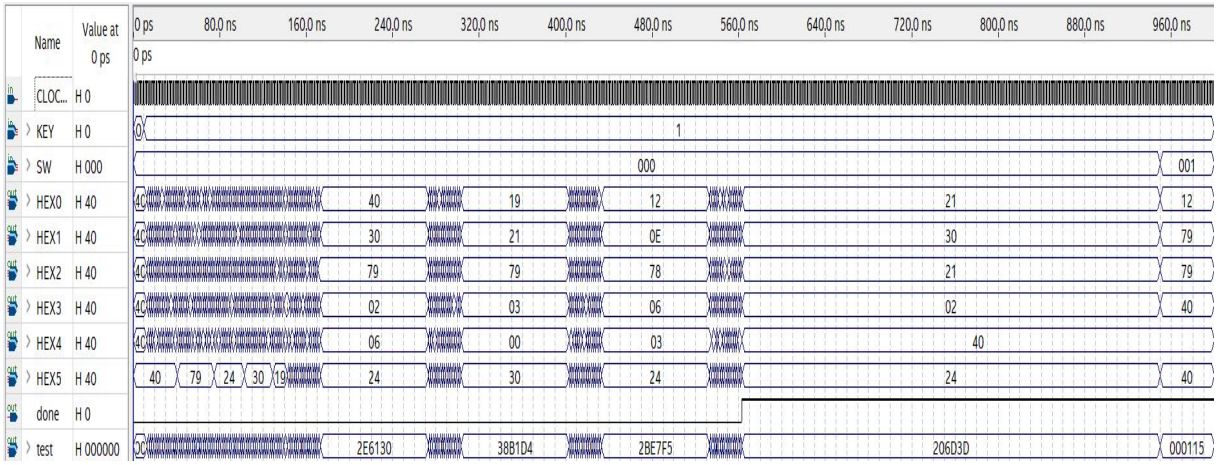

Figure 12. *24'h206D3D* result for Mat_1

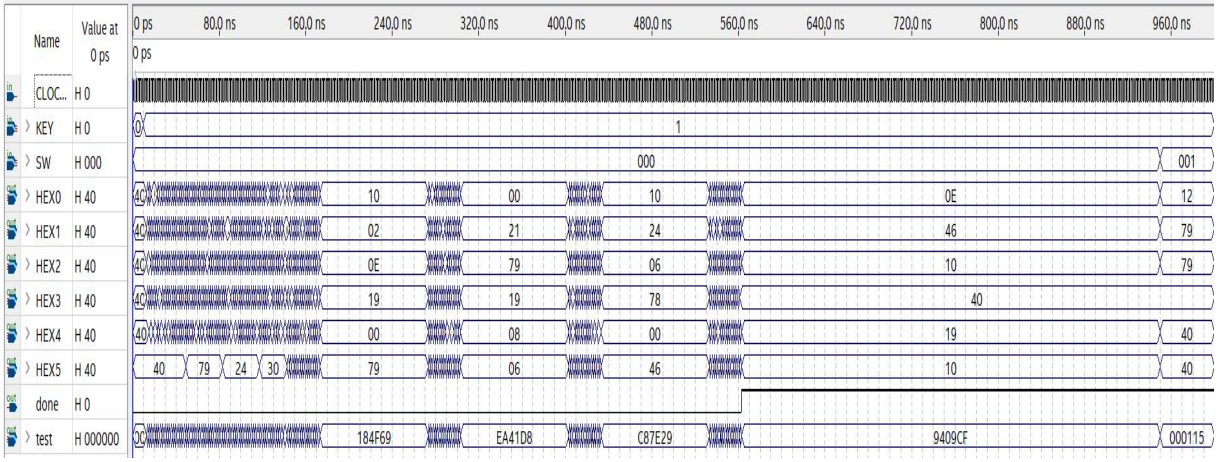

Figure 13. *24'h9409CF* result for Mat_2

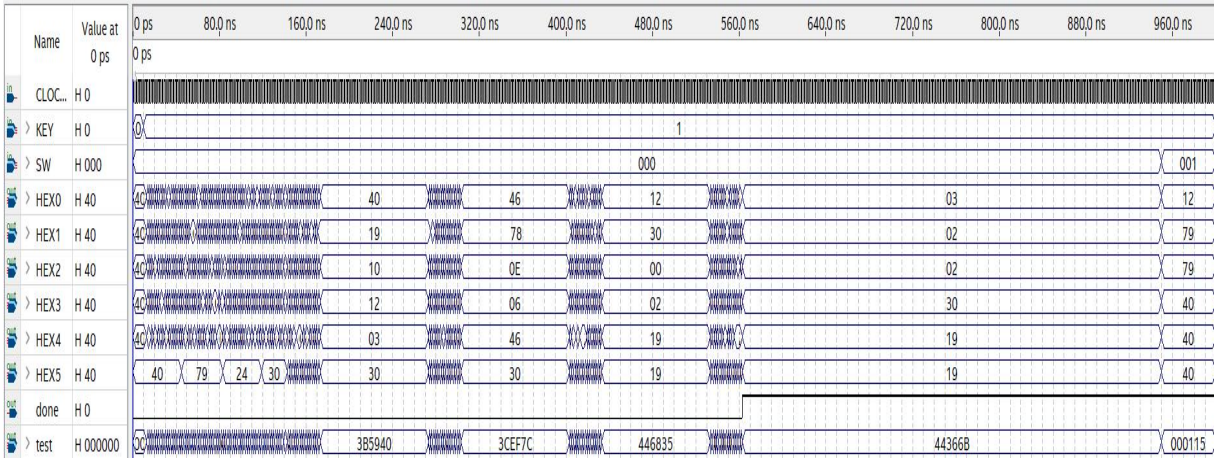

Figure 14. *24'h44366B* result for Mat_3

VI. Implementing on Hardware

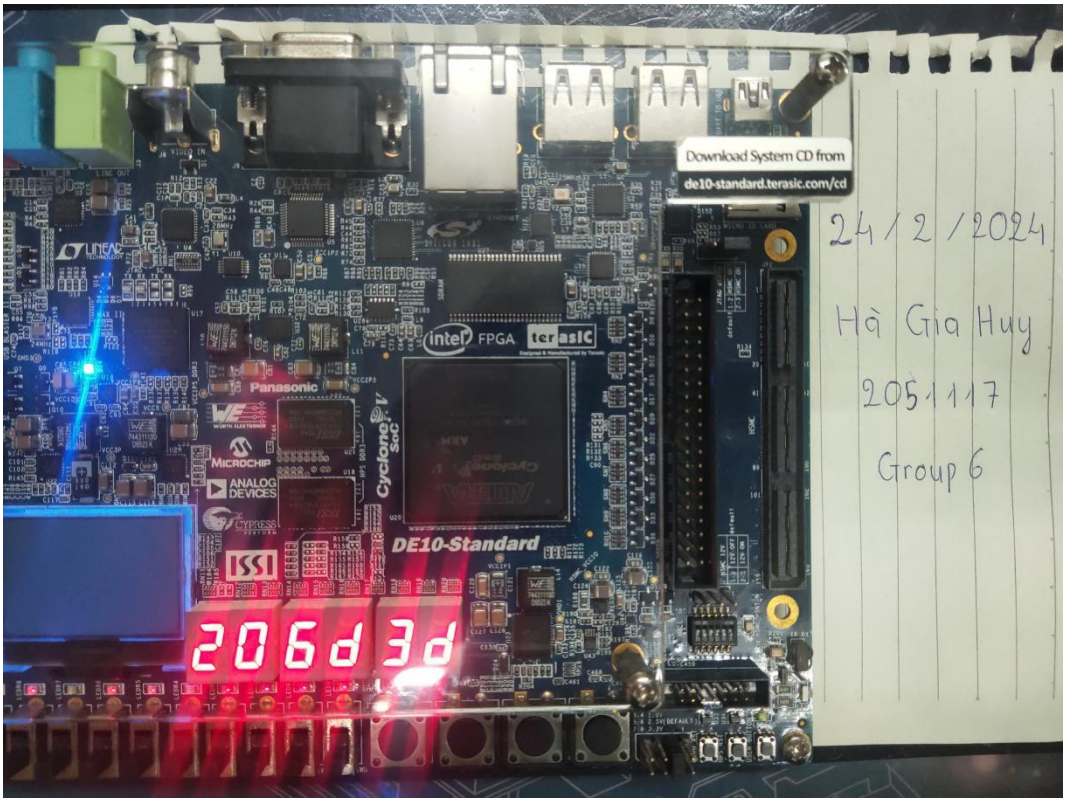

Figure 15. Result for Mat_1

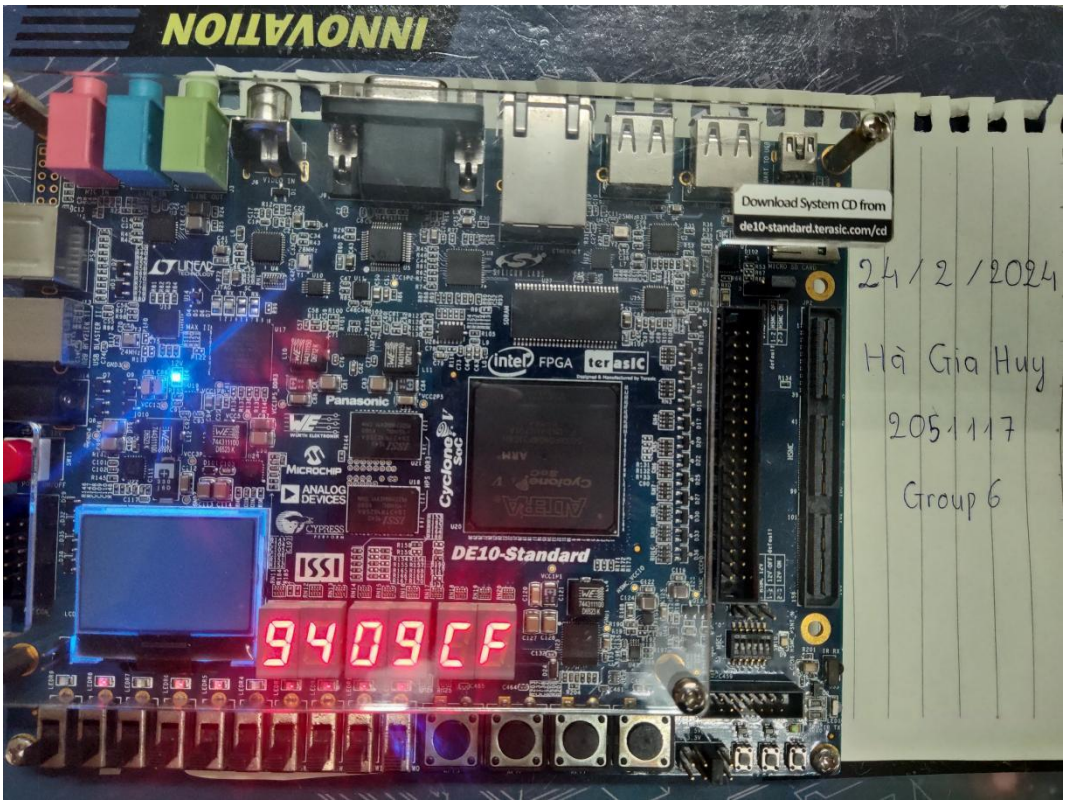

Figure 16. Result for Mat_2

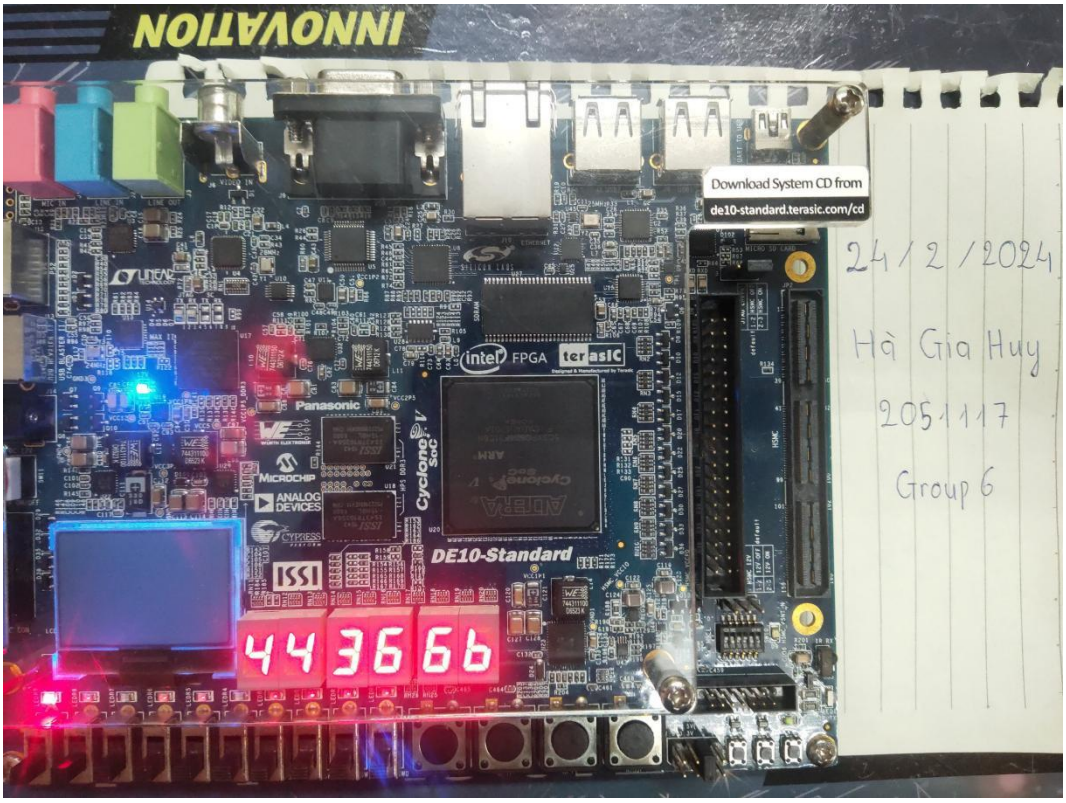

Figure 17. Result for Mat_3

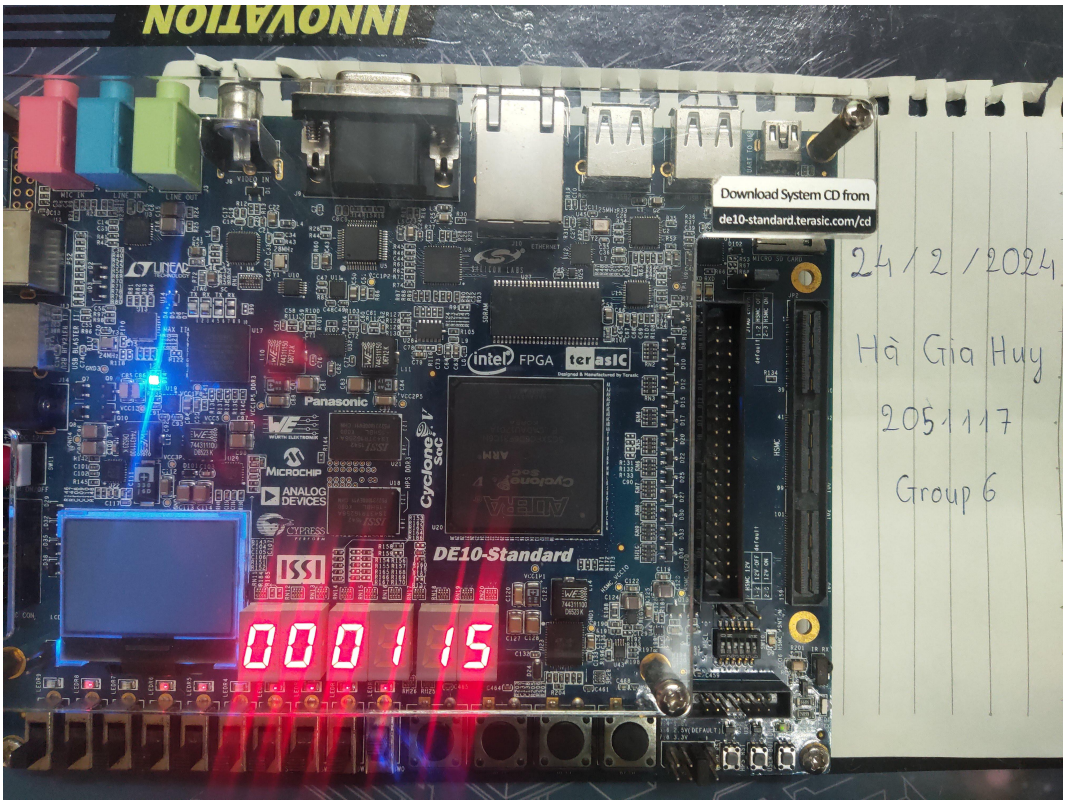

Figure 18. Number of Clock Ticks (*in hex*)

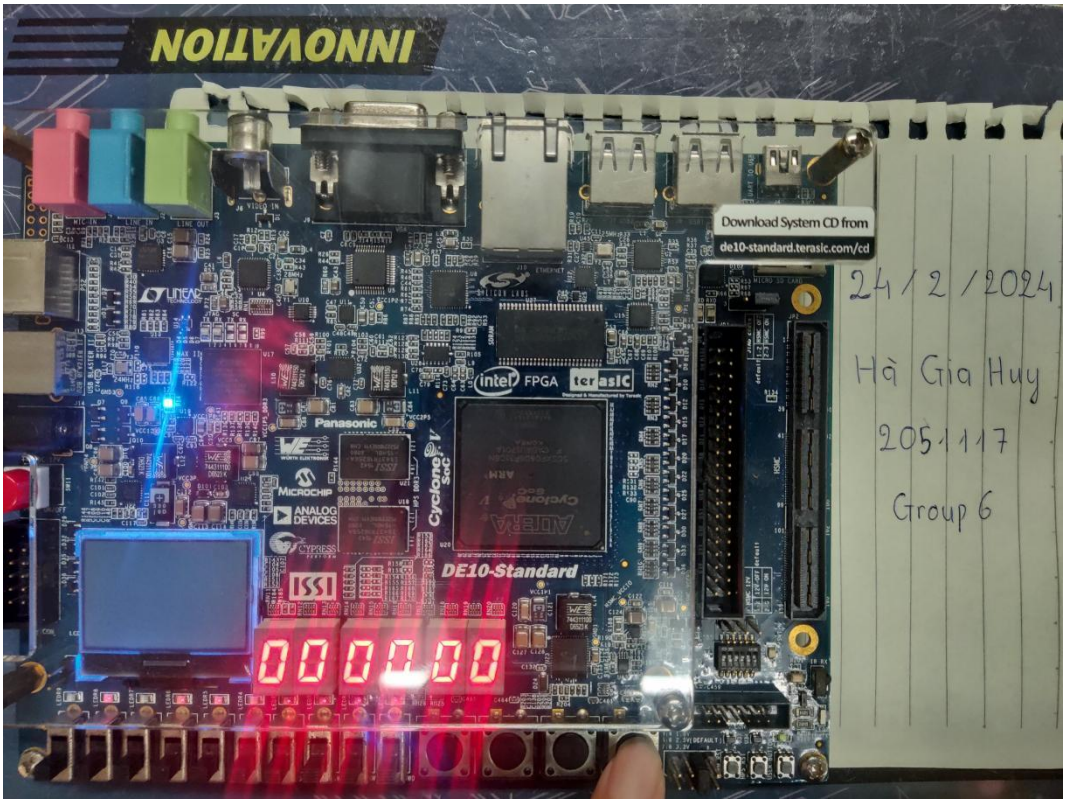

Figure 19. Reset Operation

The Fmax column is what were interested in. Keep this value over 70MHz by breaking up large blobs of combinational logic with registers, and your design is guaranteed to work "as you coded it." Let it fall below 70MHz, and all bets are off. For this project, my system is required to meet the timing requirements of the 70MHz clock, as shown in that 1100mV, 85C model.

Slow 1100mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	71.14 MHz	71.14 MHz	CLOCK_50	

Figure 20. Timing Analysis

It will have a listing for the clock I just created and constrained. Quartus will be able to measure my critical paths accurate and decide if my design is feasible.

Clocks

<<Filter>>

	Clock Name	Type	Period	Frequency	Rise	Fall	Duty Cycle	Divide by
1	CLOCK_50	Base	20.000	50.0 MHz	0.000	10.000		

Figure 21. Timing Constraints

VII. Conclusion

In short, I review arithmetic knowledge about matrix operator such as multiplication, addition and multiply accumulator in this project. Moreover, it it a good project to practice my coding skills, systemverilog algorithm and organize logic blocks. The major target of this project have complete with expected results for matrix_1s and matrix_2s. In addition, I also generate a new random matrix to triple check my design and get targeted value. About the cycle of design, getting it to run faster is worth more, see the grading sheet. 512 clock cycles is effectively your "speed of light" target and my system get 277 clock cycles whereas making sure it not truncating intermediate values for accurate result. Furthermore, keeping Fmax over 70MHz also very important to my design is guaranteed to work "as you coded it."