University of Pisa

MSc in Computer Engineering

Electronic and Communication Systems

Academic Year 21/22

Matrix Multiplier

Report and VHDL implementation

Luca Tartaglia

Kamran Mehravar

Electronic Project – Matrix Multiplier

Contents

1. Introduction

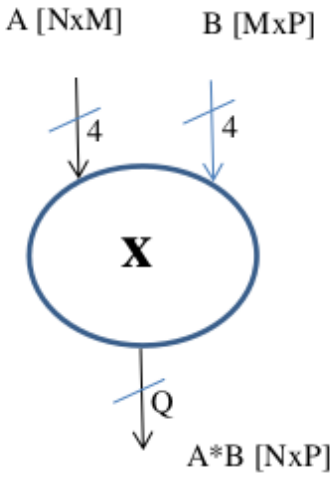
1.1 Problem Description

Design the VHDL description of a matrix multiplier with the following characteristics:

- input matrices of size (NxM) and (MxP) with elements represented in 2's complement on 4 bits
- Output matrix of size (NxP) with elements represented in 2's complement on Q bits. The value of Q has to be determined in order to avoid any finite arithmetic's error.

For the test-bench simulation, please consider the following values of the parameters: N=2, M=3

and P=4.



1.2 State-of-Art of Digital Multipliers

A binary multiplier is an electronic circuit used in digital electronics with the purpose to multiply binary numbers. To implement a digital multiplier can be used a variety of techniques and most of them involve computing the set of partial products, which are then summed together using binary adders.

Binary multiplication is the process of multiplying binary numbers and is performed doing exactly the same multiplication as with decimal numbers.

As for the decimal multiplication one of the most used algorithm is the **shift and add**, i.e. Doing the partial products (which is 0 or the first number), shifting one position left and then adding them together (using a binary addition).

$$
\begin{array}{cccccccc}
 & & & x_3 & x_2 & x_1 & x_0 \\
 & & & y_3 & y_2 & y_1 & y_0 \\
\hline
 & & & x_3y_0 & x_2y_0 & x_1y_0 & x_0y_0 \\
 & & x_3y_1 & x_2y_1 & x_1y_1 & x_0y_1 & \\
 & x_3y_2 & x_2y_2 & x_1y_2 & x_0y_2 & & \\
x_3y_3 & x_2y_3 & x_1y_3 & x_0y_3 & & & & \\
\hline
p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}
$$

Figure 1. example of binary multiplication

Using **binary numbers,** it must be pay particularly attention on which representation is been used, because, modern computers uses the **two's complement** representation, so the sign is embedded in the number itself. That force the multiplication process to be adapted to handle two's complement numbers, and that complicates the process of the multiplication.

For this reason, were be designed specific algorithms to handle the multiplication with 2's complement numbers. One of the most important is the Booth algorithm, that is an algorithm designed to multiply two signed binary numbers in 2's complement notation.

The **hardware implementation** of a digital multiplier can be realized by using 2 different approaches:

The first approach is called **ROM-based implementation** and it consist to use a ROM where are saved all the results of the multiplication of 2 words (for example A and B).

So, this architecture takes in input A and B words, and return as output the multiplication values stored in the ROM.

This approach has two big problems:

1. It would take a large amount of memory (e.g. for 2 words of 16 bit it would need 2^{32} words of 32 bits = 16 Gigabytes of memory)
2. The speed of memory access would be too slow.

For this reason, this approach is not acceptable in digital signal processes.

The second approach is to implement a **parallel multiplier**, that is doing each single operation bit by bit. All can be split into 3 general steps:

1. generating the **partial product** on k bits, using the representations of the operands (e.g. X and Y) on n and m bits respectively.

   the partial product can be calculated using a Full Adder and where there is not the carry-in it can be used a Half Adder.

2. reducing the partial product, sizing it on k bits (that can be in the range [0, n+m-1]
3. computing the final product, summing all the partial products.

To realize the blocks concerning the operations required are needed five different block types:

1. ***Product:*** is an AND that takes in input two values x,y and gives p as the product in output.
2. ***Half Adder***: it is a Full Adder where the carry-in is 0
3. ***Full Adder:*** here the carry is present and it will be summed (c_i is the carry-in)
4. ***Product+ Half Adder****:* it makes both the sum and the product between x and y
5. ***Product + Full Adder****:* same as the block 4 but takes into account the carry. -in

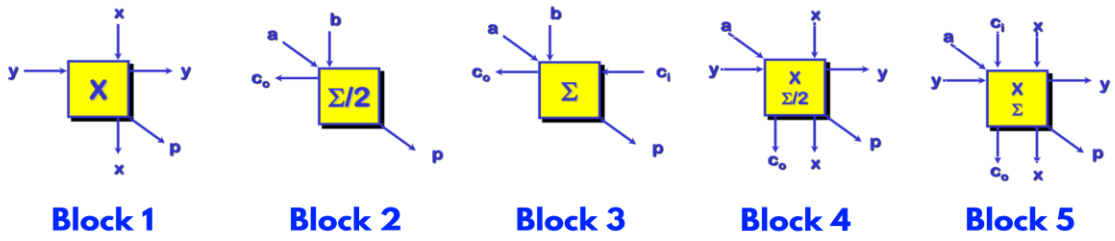


Figure 2. block types

The overall scheme of a parallel multiplier is represented in the figure below:

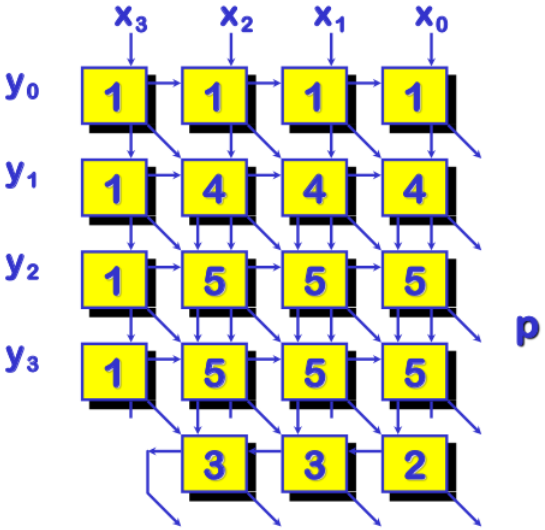


Figure 3. full parallel multiplier figure

As is possible to note in the figure, the type 1 block provides all the partial products, the type 4 and 5 blocks are the Half Adders and Full Adders and the final blocks, of 2 and 3 type, represent a sort of ripple carry adder that makes the final sum.

The latter scheme is based on a 4x4 multiplier, but in general the are needed:

- N^2 AND port
- N(N-2) Full Adder
- N Half Adder

1.3 Matrix Multiplication

Matrix multiplication is a binary operation that produces a matrix from two matrices. 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 AB.

Formally it can be defined that, given two matrices $A \in \mathbb{Z}^{n*m}$, $B \in \mathbb{Z}^{m*p}$

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \ddots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{pmatrix} \quad B = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1p} \\ b_{21} & b_{22} & \dots & b_{2p} \\ \vdots & \vdots & \ddots & \dots \\ b_{m1} & b_{m2} & \dots & b_{mp} \end{pmatrix}$$

Figure 4. Matrices A and B

The matrix product $P = AB \in \mathbb{Z}^{n*p}$ is defined as:

$$P = \begin{pmatrix} p_{11} & p_{12} & \dots & p_{1p} \\ p_{21} & p_{22} & \dots & p_{2p} \\ \vdots & \vdots & \ddots & \dots \\ p_{n1} & p_{n2} & \dots & p_{np} \end{pmatrix}$$

Figure 5. Matrix P = AB

Where each element can be calculated as follow:

$$p_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{im}b_{mj} = \sum_{k=1}^{m} a_{ik}\, b_{kj}$$

Figure 6. Formula for calculate each element of P

for i=1 to m and j=1 to p.

From this a simple algorithm can be constructed to compute the matrix multiplication, which loops over the indices i from 1 to n and j from 1 to p.

1.4 Finite Arithmetic sizing

In order to avoid any finite arithmetic's error on q (number of bits of the matrix P elements) are assumed the followed assumption:

First of all, by definition the interval of all possible representable numbers in 2's complement with N bits is:

$$[-2^{N-1}: 2^{N-1} - 1]$$

Formula 1. *range of possible numbers in 2's complement*

so in the case study the range of the representable number with 4 bits is [-8 : 7].

Knowing the latter, the maximum number (real part) having 4 bits is 64.

The sum of 2 binary numbers on n bits can always be displayed on *n+1* bits, instead the product of two numbers in 2's complement on n bits can always be displayed on *n+n = 2n* bits.

Moreover, doing the matrix multiplication operation it is possible to have the sum of the maximum number *m* times (where *m* is the number of rows of matrix A and columns of matrix B), so it is possible to define the maximum computable number for each cell in this way:

$$k = (2^{N-1} * 2^{N-1}) * m = 2^{2N-2}$$

Formula 2. *Max representable number for each cell*

In this case, having matrices 2x3 and 3x4 *m* = 3, so the maximum representable number is 64*3 = 192.

By definition it is possible to calculate the number of bits needed to represent a number in 2's complement as follow:

$$N = \lceil log_2(x + 1) \rceil + 1 if x \geq 0$$

$$N = \lceil log_2(|x|) \rceil if x < 0$$

Formula 3. *Formula to calculate the number of bits given a decimal number*

At this point, knowing that the maximum number is representable with *Formula 1* and the number of bits having a number is computable with *Formula 3* is possible to conclude that in general the number of bits q is:

$$q = \lceil log_2(k + 1) \rceil + 1$$

Formula 4. *formula to compute q*

And in this specific case the number of *q bits* to represent an element of matrix P without finite arithmetic's errors is :

$$q = \lceil log_2(192 + 1) \rceil + 1 = \lceil 7{,}5924 \rceil + 1 = 9 bits$$

Formula 5. *Computation of q with 4 bits*

2. Matrix Multiplier Implementation

2.1 Algorithm

The algorithm used to compute the matrix multiplication is the following:

```
1    for i from 1 to n
2        for j from 1 to p
3            for k from 1 to m
4                Pij = Pij + (Aik * Bkj) ;
5            end for ;
6        end for ;
7    end for ;
```

This is a simple algorithm constructed with loops over the indices *i* from 1 through n and *j* from 1 through p, computing each element of the resulting matrix as follows:

$$c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}.$$

***Formula 6**. computation to calculate entries of resulting matrix P*

this algorith takes $\Theta(nmp)$. To simplify it can be assumed that all the matrices are square of size n*n, so the computation time is $\Theta(n^3)$.

2.2 Model Architecture

The block diagram for the implemented architecture is the following:

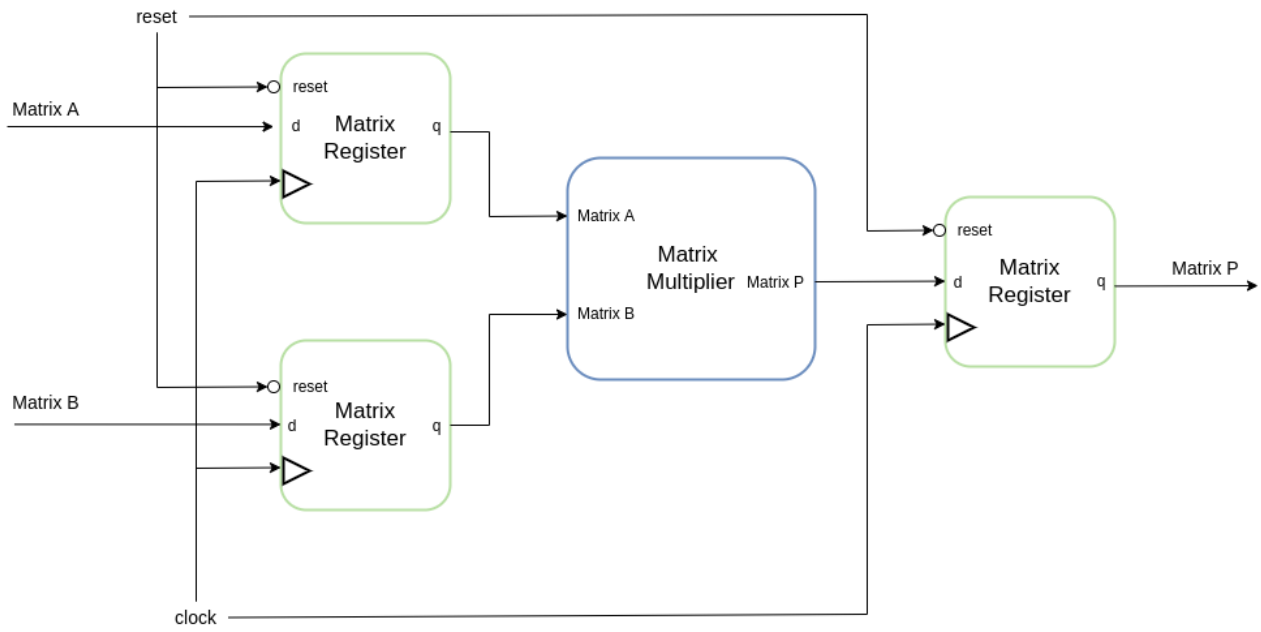


Figure 7. Matrix Multiplier block diagram

Where the green blocks are the input/output registers, the blue block is the logic core (combinatorial circuit) of the matrix multiplier and the matrices (A,B and P) are matrices of bits (type defined).

In the following chapter all the components are explained in detail.

3. VHDL Implementation

In the following chapter is presented the VHDL code for all components of the Matrix Multiplier. It is used the VHDL 2008 syntax (that allow to use unconstrained *std_ulogic_vector* and *std_logic_vector*).

In Modelsim, to use this language syntax it must be set the option in the “Project Compiler Settings” section.

3.1 Bit Matrix type definition

It is declared a custom type of array called BitMatrix. It is the matrix type used in all modules of the matrix multiplier.

It is chosen to delegate the constraining to the single component in order to make more dynamic the whole structure.

```
18 package Common is
19     type BitMatrix is array(natural range<>,natural range<>) of std_ulogic_vector;
20 end Common;
21
22 package body Common is
23 end Common;
```

3.2 Matrix Register

The matrix register is a D-FlipFlop extended to hold a matrix of bytes (BitMatrix type). Its VHDL entity is defined as follows:

```
 9 entity MatrixRegister is
10     generic (
11         matrixRowNumber : positive;
12         matrixColNumber : positive;
13         cellBits        : positive
14     );
15     port (
16         clock   : in std_ulogic;
17         reset   : in std_ulogic;
18         d       : in BitMatrix(0 to matrixRowNumber-1,0 to matrixColNumber-1)(cellBits downto 0);
19         q       : out BitMatrix(0 to matrixRowNumber-1,0 to matrixColNumber-1)(cellBits downto 0)
20     );
21 end MatrixRegister;
```

3.3 Matrix Multiplier

This part is the logic core of the entire module. The matrix multiplier is a combinatorial circuit, below is show the entity definition:

```
11 entity MatrixMultiplier is
12     generic (
13             matrixAColNumber : positive;
14             matrixBColNumber : positive;
15             matrixARowNumber : positive;
16             matrixBRowNumber : positive
17         );
18     port (
19         matrixA : in BitMatrix(0 to matrixARowNumber-1,0 to matrixAColNumber-1)(3 downto 0);
20         matrixB : in BitMatrix(0 to matrixBRowNumber-1,0 to matrixBColNumber-1)(3 downto 0);
21         matrixP : out BitMatrix(0 to matrixARowNumber-1,0 to matrixBColNumber-1)(8 downto 0) -- Cell sizing for matrix P elements according to
22                                                                                              -- the formula to avoid finite arithmetics errors
23     );
24 end MatrixMultiplier;
```

taking into account the *finite arithmetic sizing problem* (chapter 1.4) some considerations are done:

The problem is focused on sizing the matrix P elements (line 21) and the considerations on what implement were two:

1. assign 9 bits to matrix P elements (as the computation in this specific case return)
2. assign “dynamically” to matrix P elements using the VHDL transcription of the bits computation formula (formula 4, chapter 1.4) as is shown below:

```
25
26 --matrixP : out BitMatrix(0 to matrixARowNumber-1,0 to matrixBColNumber-1)(integer((ceil(log2(real(((real(64)*real(matrixAColNumber)) + real(1) )))))) downto 0 )
27
```

After the study, assumed that the second method does not add any advantage, it was decided to follow the first consideration in order to maintain the reliability of the code.

Finally, the combinatorial logic function that implement the matrix multiplication is the following:

```
29 architecture beh of MatrixMultiplier is
30     function  MatrixMultiplication  ( a : BitMatrix; b:BitMatrix) return BitMatrix is
31
32     --the use of variables is due to the fact that there is a for cycle, so it must be updated values immediately,
33     --not after the function/process (delta delay rule for signals and variables, that updates
34     --signals after the end of the function/process).
35     variable i,j,k : integer:=0;
36     variable product : BitMatrix(0 to matrixARowNumber-1,0 to matrixBColNumber-1)(8 downto 0 ):=(others => (others => (others => '0')));
37     begin
38         for i in 0 to matrixARowNumber-1 loop
39             for j in 0 to matrixBColNumber-1 loop
40                 for k in 0 to matrixAColNumber-1 loop
41                     product(i,j) := std_ulogic_vector(signed(product(i,j)) + (signed(a(i,k)) * signed(b(k,j))));
42                 end loop;
43             end loop;
44         end loop;
45         return product;
46     end MatrixMultiplication;
47 begin
48     matrixP <= MatrixMultiplication(matrixA,matrixB);
49 end beh;
50
```

The implementation of the algorithm described in the chapter 2.1 force the uses of variables.

As commented in the file, the use of **variables** (only inside the process) is due to the fact that having a *for* cycle and knowing that the variables, with respect to signals, instantaneously resolve the assignment and not after the process termination (delta delay rule).

3.4 Matrix Multiplier Architecture

The matrix maultiplier architecture is the entire architecture composed by the input matrices, matrix A and matrix B, the output matrix P and by the clock and the reset.

Its entity is shown below:

Fig 8 matrix multiplier architecture

??????

4. Test Plan

The test plan for matrix multiplier was designed to test some different case, so, for this purpose, it was written a python program, called *TB_generator.py*. This program dynamically generates the *MatrixMultiplier_tb.vhd* files depending on which case it is decided to simulate.

One of the main goal of the test plan is to test the **finite arithmetic sizing resolution**. So this test is focused on extreme values, i.e. among all the configurations that a matrix can assume it can be considered the worst case when limit values are all placed in a row of the first matrix and in a column of the second matrix.

Doing so it is possbile to reproduce the largest positive values (192) and the largest negative values (-168) and see if the sizing of *q* (number of bits for matrix P elements) is congruent with the study.

Moreover the test plan involves the simulation of a case with **random values** for the matrices A and B and another simulation with **values chosen** by the user.

4.1 Python Testbench generator

As just mentioned, this python program generates the testbench files in order to make different types of simulation.

The program let the user choose between 4 different modes:

1. Assign values to matrices A and B in order to produce the largest positive number
2. Assign values to matrices A and B in order to prodcue the largest negative number
3. Generate randomly the values for matrices A and B
4. Let the user choose the values for Matrices A and B

```
**TESTBENCH GENERATOR, Choose the mode
MODE 1: Assign values to matrices A and B in order to produce the largest positive number
MODE 2: Assign values to matrices A and B in order to produce the largest negative number
MODE 3: Generate randomly the values for the matrices A and B
MODE 4: Create your own matrices A and B
```

One time the mode is choosen the program creates the MatrixMultiplier_tb.vhd files and shows the binary matrices written inside the *stimulus process*

```
BINARY MATRIX A:
 (("1000","1000","1000"),("0000","0000","0000"));
BINARY MATRIX B:
 (("1000","0000","0000","0000"),("1000","0000","0000","0000"),("1000","0000","0000","0000"));

**MatrixMultiplier_tb.vhd File correctly generated**
```

Moreover, the program computes the real matrix multiplication operation and returns the values into Result.txt file, so it is possible to compare the real values with the values returned by the simulation.

```
**MATRIX MULTIPLICATION RESULTS**

MATRIX A ( 2 x 3 ):
 [[-8 -8 -8]
 [ 0  0  0]]

MATRIX B ( 3 x 4 ):
 [[-8  0  0  0]
 [-8  0  0  0]
 [-8  0  0  0]]

MATRIX P = AB (MATRIX MULTIPLICATION RESULT):
 [[192   0   0   0]
 [  0   0   0   0]]

**Result.txt File correctly generated**
```

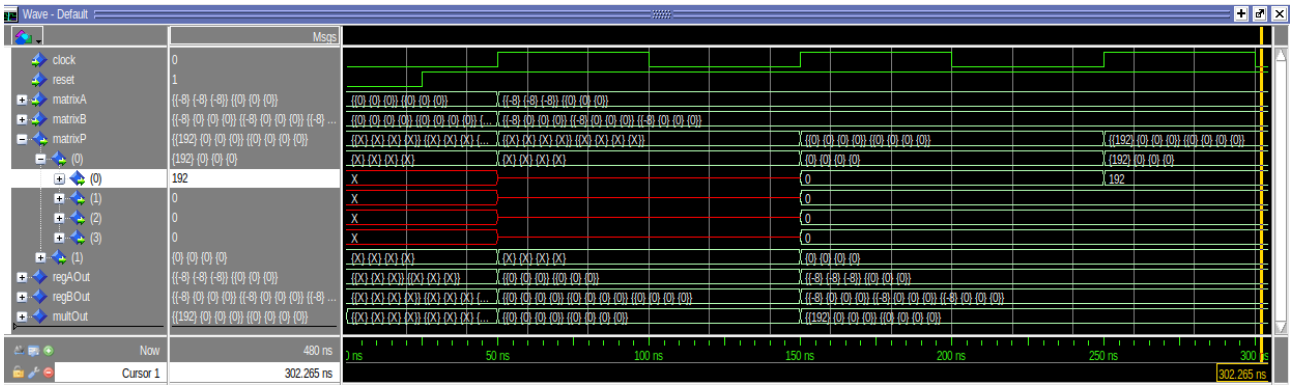
It is also important to point out that the program dynamically read the matrices A and B sizes, so it is possible to use the program to generate testbench files for matrices of any size.

4.2 Testbench implementation

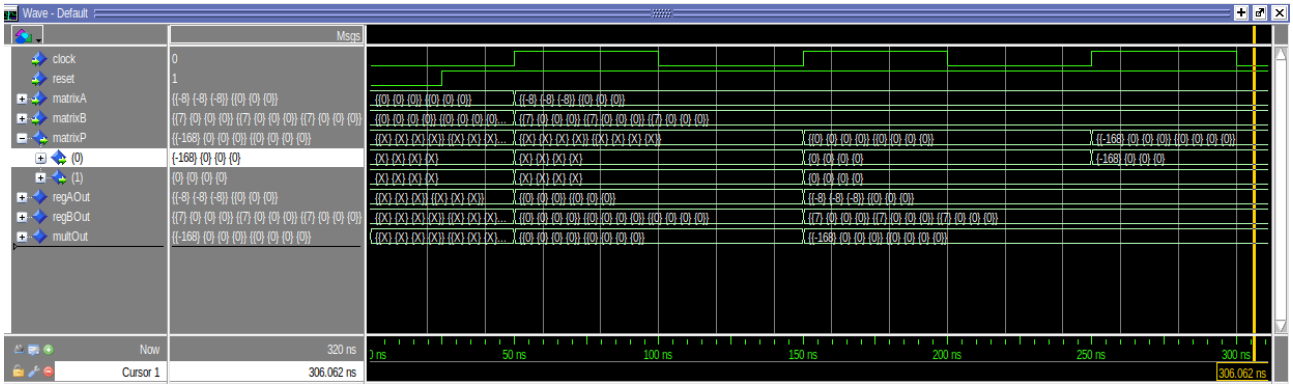
One time the MatrixMultiplier_tb.vhd file is created it is used to run the simulation in Modelsim.

Below are shown all 4 cases just explained:

First test: All elements in first row of A and in first column of B are equal to $(-8)_{10} \Leftrightarrow (1000)_2$. The result must be $(192)_{10}$ in Matrix P cell $p_{0,0}$:



Second test: All elements in first row of A are equal to $(-8)_{10} \Leftrightarrow (1000)_2$. All elements in first column of B are equal to $(7)_{10} \Leftrightarrow (0111)_2$. The result must be $(-168)_{10}$ in Matrix P cell $p_{0,0}$:



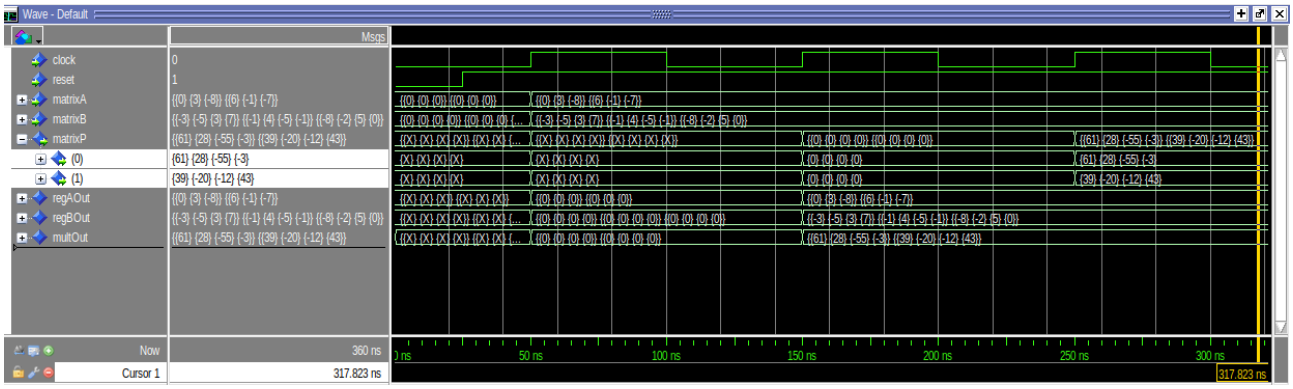
Third Test: In this case are used the matrix multiplication results, given by the python program, and are compared againts the simulations results. Is possible to observe that the results match.

```
**MATRIX MULTIPLICATION RESULTS**

MATRIX A ( 2 x 3 ):
 [[ 0  3 -8]
 [ 6 -1 -7]]

MATRIX B ( 3 x 4 ):
 [[-3 -5  3  7]
 [-1  4 -5 -1]
 [-8 -2  5  0]]

MATRIX P = AB (MATRIX MULTIPLICATION RESULT):
 [[ 61  28 -55  -3]
 [ 39 -20 -12  43]]
```



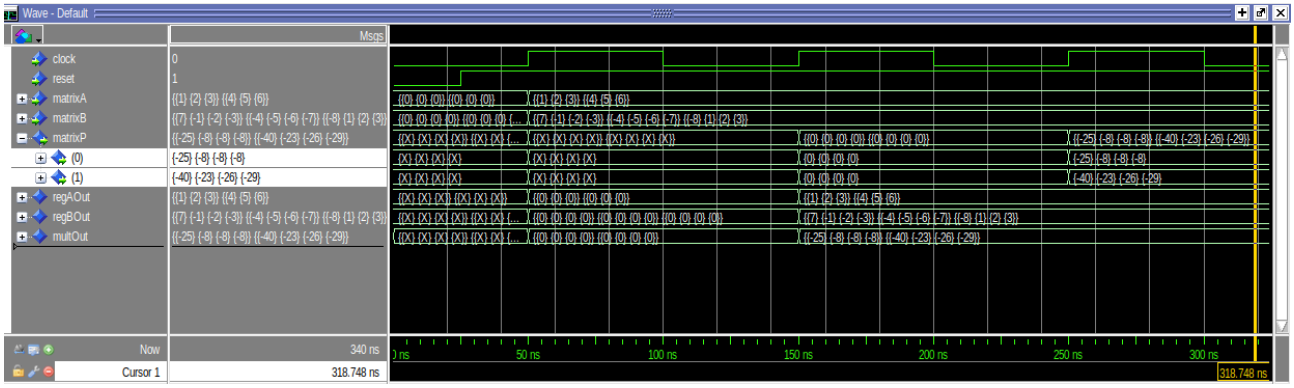
Fourth Test: Same as the third test, the results return the same values for matrix P both in the python program and in the simulation.

```
**MATRIX MULTIPLICATION RESULTS**

MATRIX A ( 2 x 3 ):
 [[1 2 3]
 [4 5 6]]

MATRIX B ( 3 x 4 ):
 [[ 7 -1 -2 -3]
 [-4 -5 -6 -7]
 [-8  1  2  3]]

MATRIX P = AB (MATRIX MULTIPLICATION RESULT):
 [[-25  -8  -8  -8]
 [-40 -23 -26 -29]]
```



5. Synthesis and Implementation Logic (Vivado part)

Like 2-D rotation matrices, we instantiate matrix dimensions in Vivado to $N = M = P = 2$. (In order for Vivado to recognize some VHDL-2008 syntactic constructs, there are several ways to run VHDL-2008 files with Vivado. We can go to the Source File Properties window, and set Type: VHDL 2008 from the drop-down of available file types. The Vivado tool then sets that the file type to VHDL-2008. we can also set files to VHDL-2008 with the *set_property* command in the *Tcl* Console.
The syntax is as follows: *set_property FILE_TYPE {VHDL 2008} [get_files <file>.vhd]*
Finally, in the Non-Project or *Tcl* flow, the command for reading in VHDL has VHDL-2008 is as follows: *read_vhdl -vhdl2008 <file>.vhd*
If we want to read in more than one file, we can either use multiple read_vhdl commands or multiple files with one command, as follows: *read_vhdl -vhdl2008 {a.vhd b.vhd c.vhd}*

5.1. Phase 1: Register Transfer Level design (RTL)

This phase results in the following design:

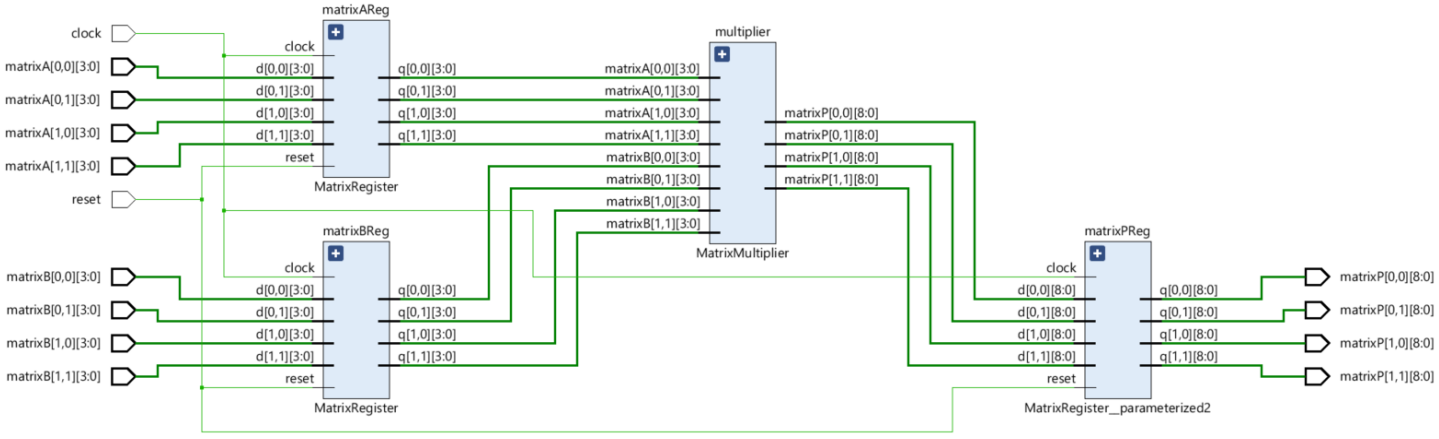


Figure 8.1. *RTL Design from Vivado*

It's also worth noting how the multiplier's loop-unrolled architecture appears:

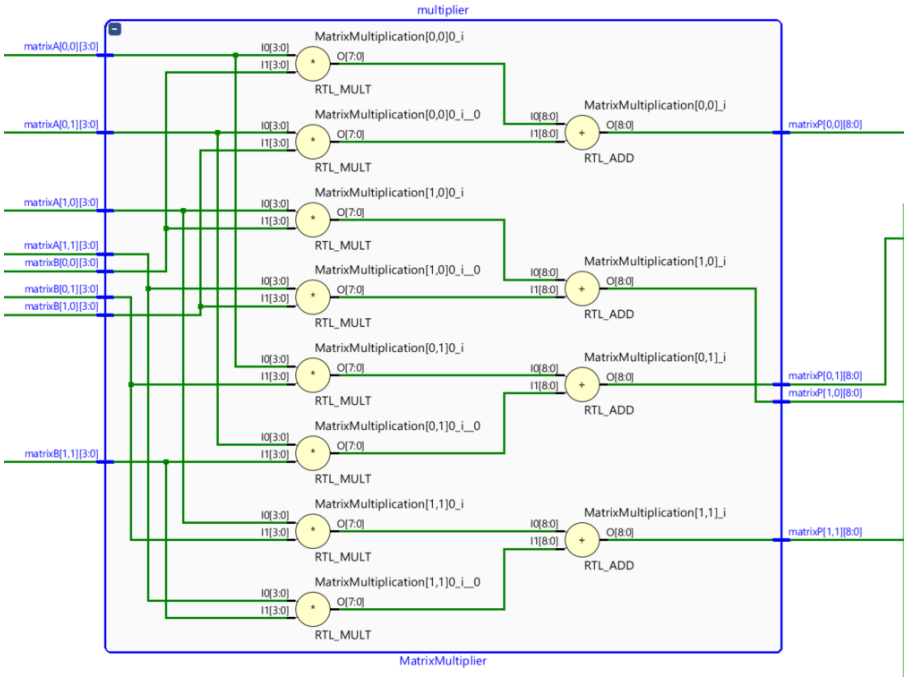


***Figure 8.2**. Inside the MatrixMultiplier*

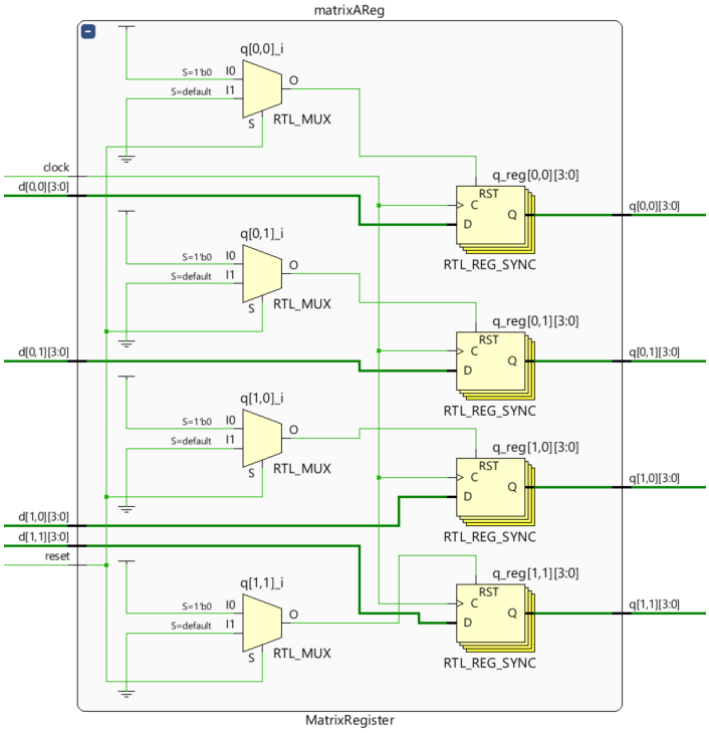


***Figure 8.3**. Inside the MatrixRegister*

5.2. Phase 2: Synthesis

5.2.1 Critical path

We can examine the report timing in the following table after adding the constraint clock with a period of

$\mathbf{t}_{clk} = 125\mathbf{Mhz}$:

There are no negative slacks, therefore an 8ns clock is definitely sufficient for the resulting design. Furthermore, the worst negative slack is 2.278 ns, indicating that the clock can be quicker than this amount.

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	2.278 ns	Worst Hold Slack (WHS):	0.221 ns	Worst Pulse Width Slack (WPWS):	3.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	36	Total Number of Endpoints:	36	Total Number of Endpoints:	69

All user specified timing constraints are met.

***Figure 9**. Timing report from VIVADO*

The SLACK is the temporal margin between the stabilization of signal and is

$$TSLACK = TCLK + TSUP.$$

We can observe the worst path thanks to Vivado:

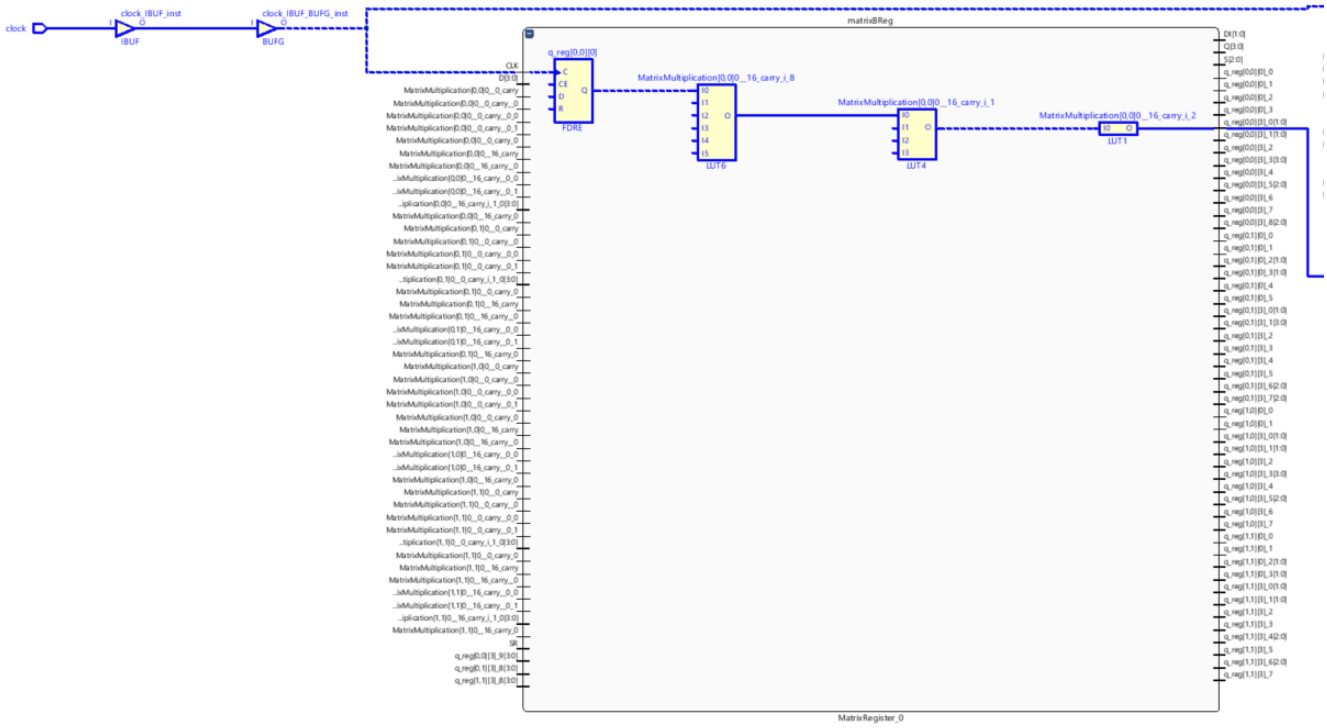


***Figure 10.1:** Worst path from VIVADO*

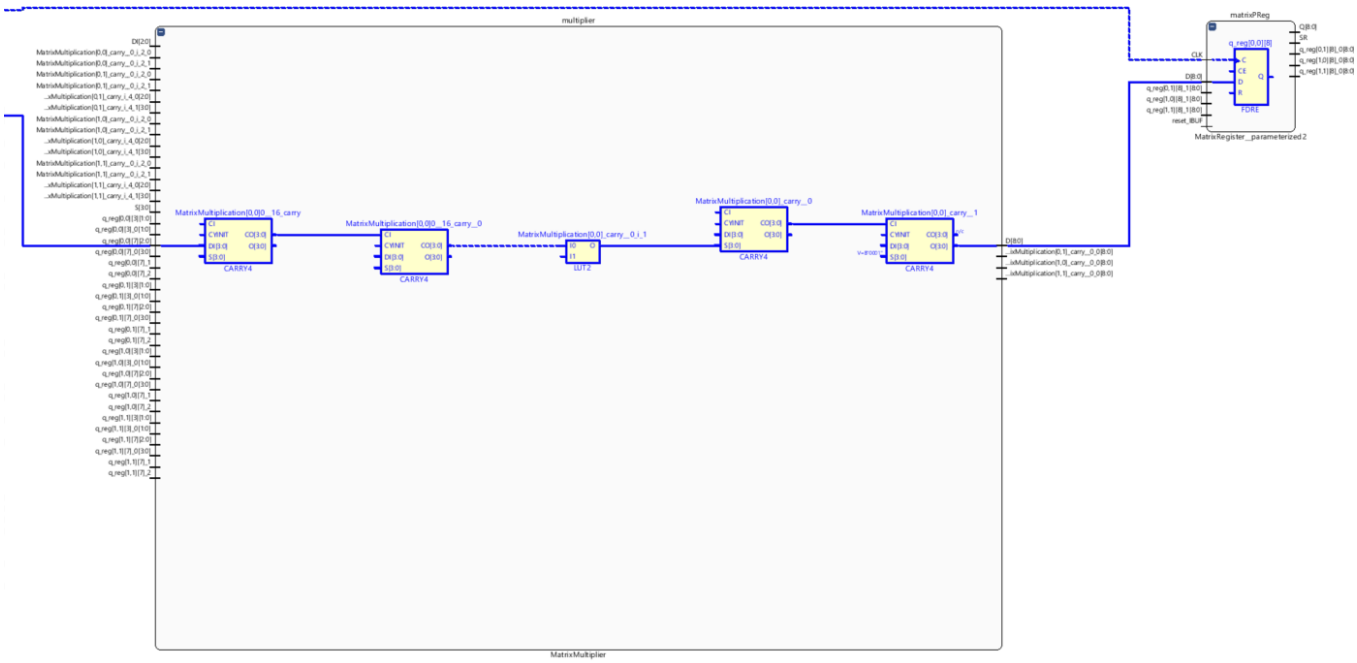


Figure 10.2: *Worst path from VIVADO*

We can calculate how much quicker we can drive our clock given the slack:

$$t_{clk} = t_{setup} + t_{p\text{-}logic} + t_{c+q} + t_{slack}$$

The minimal clock may be found using the same formula. Only slack can be changed since other times are set:

$$t_{clkmin} = t_{setup} + t_{p\text{-}logic} + t_{c+q} + t_{slackmin}$$

Because we are in the limit scenario, the value of slack is zero. If we take the second form out of the first equation, we get:

$$t_{clkmin} - t_{clk} = 0 - t_{slack} \Rightarrow t_{clkmin} = t_{clk} - t_{slack} \Rightarrow f_{max} = \frac{1}{t_{clk} - t_{slack}}$$

So, in the end, we get:

$$f_{max} = \frac{1}{tclk} = \frac{1}{8\ ns - 2.278} = \frac{1}{5.722\ ns} \approx 174Mhz$$

When we look at the timing report after applying this value, we see that the worst negative slack is equal to zero, which is exactly what we want.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.000 ns	Worst Hold Slack (WHS):	0.221 ns	Worst Pulse Width Slack (WPWS):	2.361 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	36	Total Number of Endpoints:	36	Total Number of Endpoints:	69

All user specified timing constraints are met.

Figure 11: *Timing report with tclock = 5.722ns*

5.2.2 Power consumption

The most recent report is on ***Power consumption***. This is a very preliminary estimate, but it gives you a good understanding of the circuit's power needs.

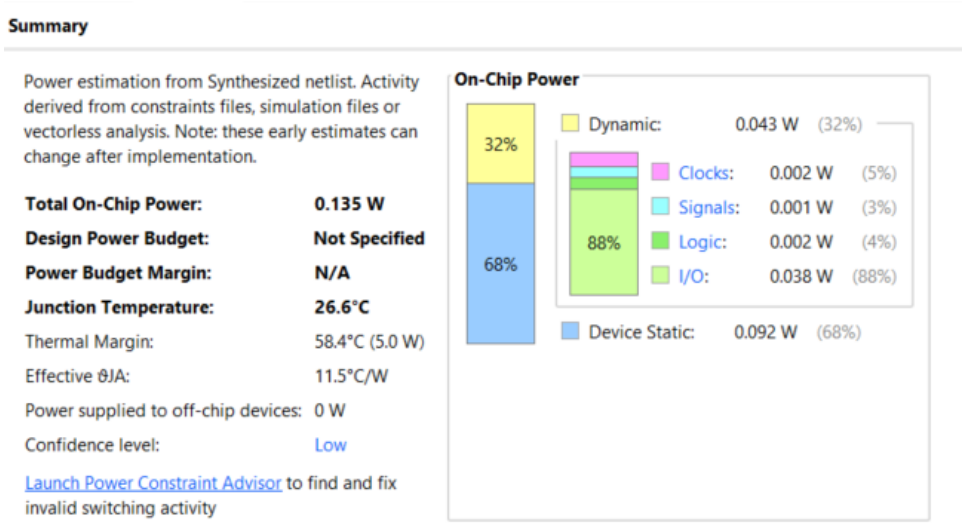


Figure 12: *Power consumption report*

The average power usage is more than 100mW. Furthermore, the percentage of static power is clearly higher than the percentage of dynamic power. As a result, the switching activity in this circuit might be considered minimal.

5.2.3 Utilization report

The ***utilization report*** obtained are as follows:

Resource	Utilization	Available	Utilization %
LUT	205	17600	1.16
FF	68	35200	0.19
IO	70	100	70.00

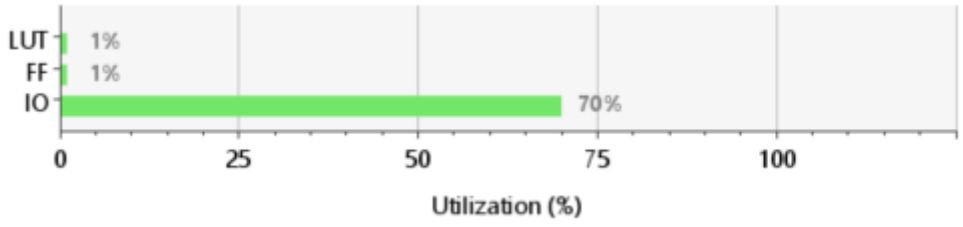


Figure 13: *Utilization report*

5.3. Phase 3: Implementation

Vivado will perform the location and route in this step, with some beneficial improvements. In typically, this step is preceded by I/O Planning, in which the FPGA's I/O Physical Ports are linked to the Elaborated Design's I/O Ports. However, the FPGA does not have enough ports to accomplish a 16-bit input and output. As a result, the implementation will be done in Out of Context Mode, which eliminates the need for I/O planning.

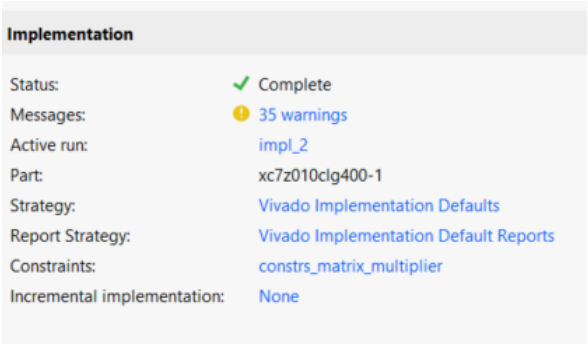


Figure 14: *Implementation Result*

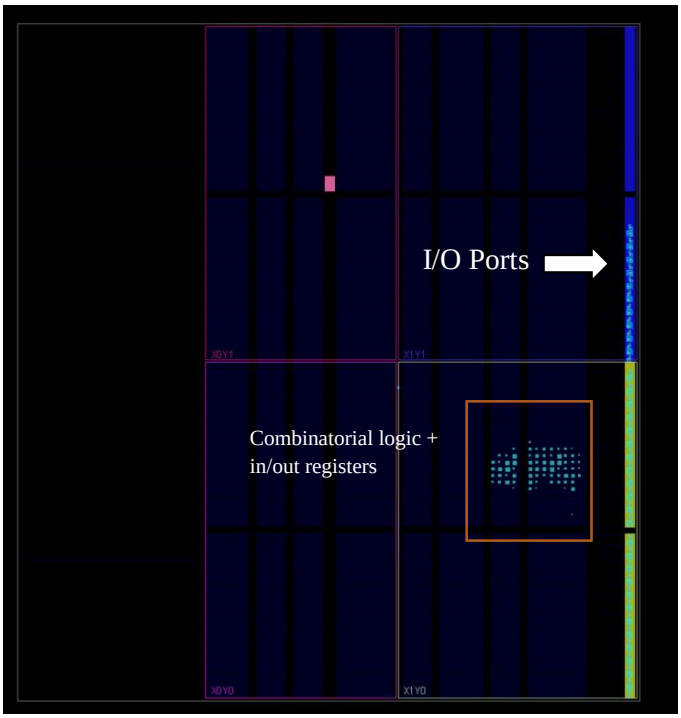


Figure 15: *Implemented design from VIVADO*

5.4. Warnings

5.4.1 Warnings Analysis

The Implementation's Warnings are the following:

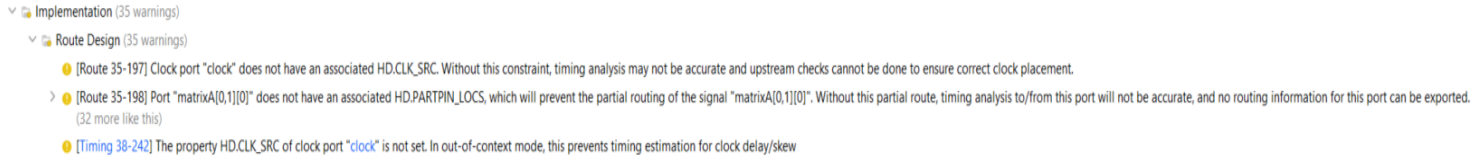


Figure 16: *Implementation Warnings*

They're all internal Vivado warnings that may be disregarded.

5.4.2 Timing Warnings

There is a warning concerning the lack of a delay limitation. This will determine whether or not our ASIC can fulfil the timing requirements of the external devices to which it is linked. If these timings are not reached, our ASIC will be unable to communicate with the external devices to which it is designed.

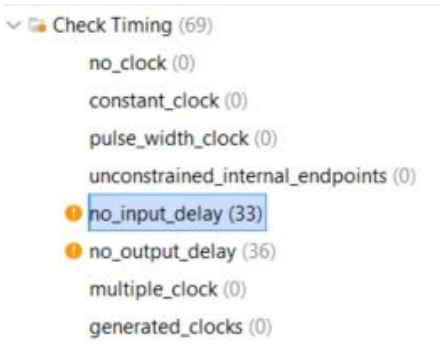


Figure 17*. Timing warning from VIVADO*

6. Conclusions

We will have a functional matrix multiplier at the conclusion of this project. It might be implemented using more space-efficient methods, such as Strassen's, to reduce the area occupied by combinatorial logic for multiplication and to parallelize combinatorial circuits, shortening the path between in-out registers. This might result in less restrictive clock frequency constraints, enhancing the module's total speed. This is crucial when discussing matrix multiplication since it is commonly used in graphics processors to perform 2-D and 3-D transformations on pictures and models.