MATRIX MULTIPLIER

VHDL DESIGN AND IMPLEMENTATION REPORT

Dec 20/2018

By: Bruk Tekalgne

Zewdie Habtie

Contents

List of Figures and Tables

1. Introduction

In many numerical algorithms matrix multiplication is a central operation. Hence, much work has been invested in making **matrix multiplication algorithms** efficient. Applications of matrix multiplication in computational problems are found in many fields including scientific computing and pattern recognition and in seemingly unrelated problems such counting the paths through a graph. Many different algorithms have been designed for multiplying matrices on different types of hardware, including parallel and distributed systems, where the computational work is spread over multiple processors (perhaps over a network).

Directly applying the mathematical definition of matrix multiplication gives an algorithm that takes time on the order of n^3 to multiply two $n \times n$ matrices ($\Theta(n^3)$ in big O notation).

2. Algorithms to be implemented

2.1. 2's complement multiplication

Computers basically don't "understand" negative numbers whether it is real number or integer. But the users of computers, humans, need the computers to do, for example, mathematical operations on negative numbers. So, in order to fulfill this requirement of users computers are made to have various representations of numbers, i.e. 2's complement, which makes it easier to do such operations. So in our project we make use of multiplier extensive number of times in which we do the multiplication by representing the numbers in 2's complement. In order to do such a conversion we make use the **signed()** type caster in **ieee std** library.

2.2. 2's complement addition

One of the advantages of 2's complement representation is that you can under go any arithmetic manipulation without paying a special attention to the sign bit. So in our project we used the numbers that are converted to 2's complement prior to the addition operation, specifically for the multiplication operation. So that the representation continues and are used for addition.

2.3 Matrix Multiplication

Matrix multiplication is basically the collection of addition and multiplications of elements in the given two matrices. The definition of matrix multiplication is that if $C = AB$ for an $n \times m$ matrix A and an $m \times p$ matrix B, then C is an $n \times p$ matrix with entries

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

Equation 1: Matrix Multiplication

Let's explore further, suppose we have two matrices with the following configurations:

Matrix a: N x M, where N is the number of rows and M is the number of columns.
Matrix b: M x P, where M is the number of rows and P is the number of columns.

Then, the matrix result of the multiplication of these two matrices will be C, with N rows and P columns:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mp} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{np} \end{bmatrix}$$

Figure 1: Matrix Multiplication in Block Form

Now, in order to determine the result value of C matrix, the next operations should be performed:

$$\begin{aligned} c_{11} &= a_{11}\, b_{11} + a_{12}\, b_{21} + \ldots + a_{1m}\, b_{m1} \\ c_{12} &= a_{11}\, b_{12} + a_{12}\, b_{22} + \ldots + a_{1m}\, b_{m2} \\ c_{1p} &= a_{11}\, b_{1p} + a_{12}\, b_{2p} + \ldots + a_{1m}\, b_{mp} \\ c_{21} &= a_{21}\, b_{11} + a_{22}\, b_{21} + \ldots + a_{2m}\, b_{m1} \\ c_{22} &= a_{21}\, b_{12} + a_{22}\, b_{22} + \ldots + a_{2m}\, b_{m2} \\ c_{2p} &= a_{21}\, b_{1p} + a_{22}\, b_{2p} + \ldots + a_{2m}\, b_{mp} \\ c_{n1} &= a_{n1}\, b_{11} + a_{n2}\, b_{21} + \ldots + a_{nm}\, b_{m1} \\ c_{n2} &= a_{n1}\, b_{12} + a_{n2}\, b_{22} + \ldots + a_{nm}\, b_{m2} \\ c_{np} &= a_{n1}\, b_{1p} + a_{n2}\, b_{2p} + \ldots + a_{nm}\, b_{mp} \end{aligned}$$

Figure 2: Matrix Multiplication result element computation

As we can see, to determine the value of $\mathbf{c}_{(i,j)}$, it is required to multiply each value of $\mathbf{a}_{(i,\,0..m)}$ by $\mathbf{b}_{(0..n,j)}$ then add all the multiplied numbers.

3. Architecture Design and Implementation

As mentioned above we tried to explore various options on the implementation. In this section we will explain the choice we make and the potential benefits with respect to other implementations.

3.1. Implementation Strategy

From our analysis of the problem, we devised an algorithm which loops over the indices *i* from 1 through *N* and *j* from 1 through *P*, and k from 1 through M and/or M-1 generating circuit and the connections. In the inner most loop the iterations from 1 to M are used to generate Multipliers and iterations from 1 to M-2 are used to generate Adders. In the outer loops of the pseudocode we make the connections between the outputs of the corresponding multipliers to the adders, and adders with adders which finally produces the result C matrix.

Note: *the indexing in array of vhdl begins with zero but the algorithm is made to begin with one. This indexing is used in the pseudocode for the better user experience assuming the user has less programming experience.*

The pseudocode of the algorithm is as follows:

```
• Input: matrices $A$ of size **M x N** and $B$ of size **N x P**
• Let $C$ be a matrix of **N x P**
• Allocate a ***product*** 3d matrix P of size M, N, P – for interconnections
• Allocate a ***sum*** 3d matrix of size M-1, N, P – for adder interconnections
• For $i$ from 1 to **N**:
    • For $j$ from 1 to **P**:
        • For $k$ from 1 to **M**:
            • Set product[ i , j, k] <= $A[i, j] \times B[k, j]$
        • For $k$ form 1 to **M-1:**
            • If k == 1
                • sum[i ,j ,k] <= product[ i, j, k] + product[i, j, k+1]
            • If K > 1 and k < M – 1
                • sum[i ,j ,k] <= product[ i, j, k] + sum[i ,j ,k - 1]
            • If k == M – 2
                • $C[i, j] \leftarrow$ product[ i, j, k] + sum[i ,j ,k - 1]
```

In the above pseudocode, the internal connections may seem to have a lot of number with respect to the conventional Matrix multipliers which usually uses latches (for caching) in the implementation. However, at a cost of large interconnection our algorithm **minimizes the number** of **Transistors** inside since there is no latching. This algorithm may not be very effective with **FPGA (Field Programmable Gate Array) tools** since they do have lots of Cells prefabricated on board, But It is very useful when the implementation is done in **ASIC (Application Specific Integrated Circuit)**. In addition, the system is also very fast because there is no caching or synchronization inside, assuming the delays in the connections and gates are very small compared to clock period in a sequential circuit.

3.2 Dependency Diagram

The diagram blow show the existing relation between blocks. Meanings for each block is found below

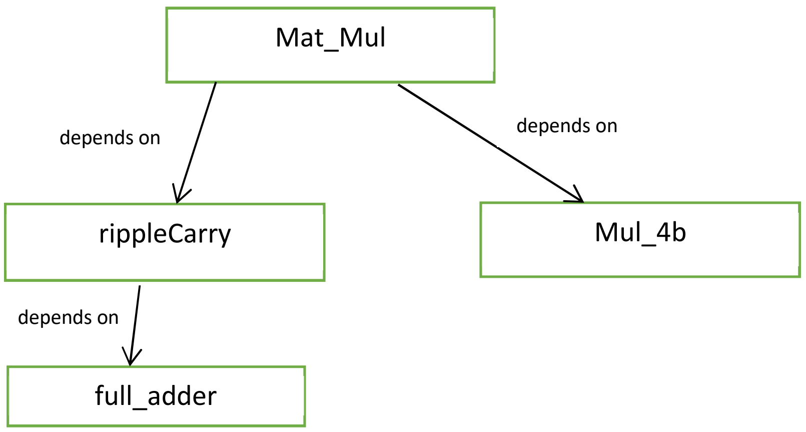


Figure 3: Block Dependency Diagram

Table 1: Acronyms and Descriptions in the dependency diagram

Acronym	**Description**
full_adder	Full Adder
Ripple_carry	Ripple Carry Adder
Mul_4b	Multiplication block
Mat_Mul	Matrix Multiplier

3.3. Block Diagram

This diagram shows a proposed architecture to be implemented to solve the multiplication matrix problem. Please refer to the next section for further information about each block (3.4). As we can see in the figure below, the block diagram to perform the matrix multiplication uses many instances of ***MUL*** and ***RCA***, note that these instances creation are executed per each cell in the output matrix. This Figure 5 shows the blocks required just only for one generic cell in the output matrix so it can be used for the other cells.

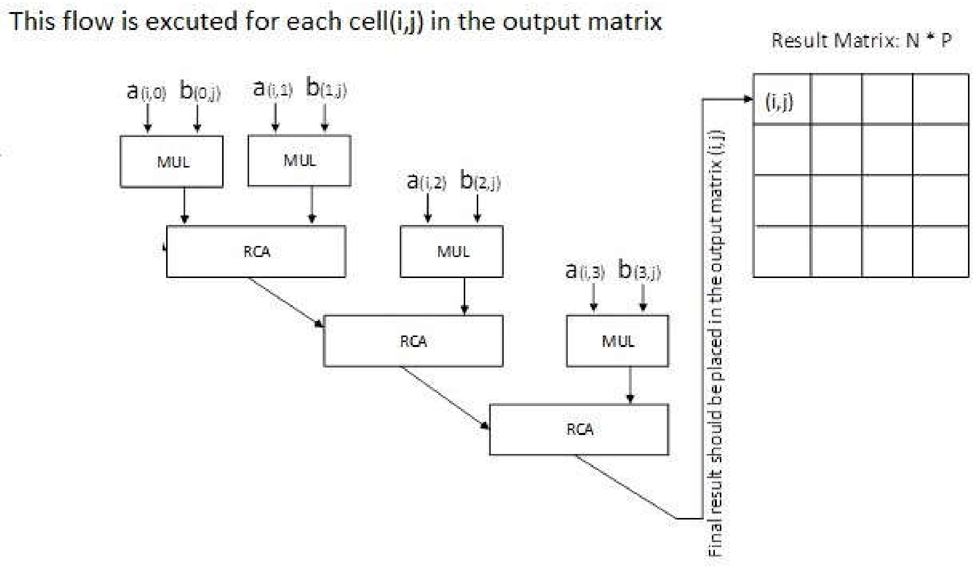
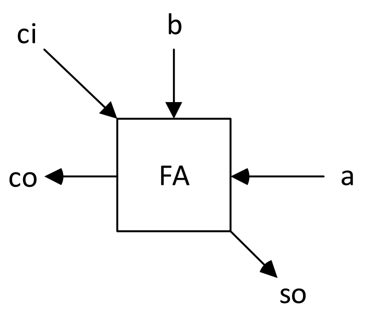
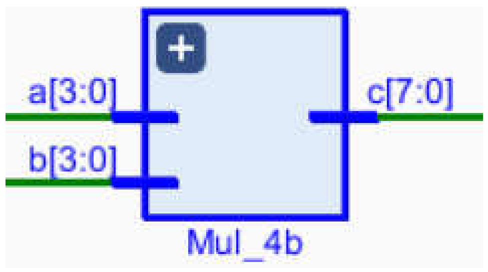

This flow is excuted for each cell(i,j) in the output matrix
Result Matrix: N * P
a(i,0) b(0,j)
a(i,1) b(1,j)
MUL
MUL
a(i,2) b(2,j)
RCA
MUL
a(i,3) b(3,j)
RCA
MUL
RCA
Final result should be placed in the output matrix (i,j)
(i,j)

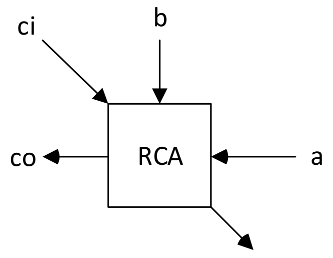
Figure 4: Matrix Multiplier for 3 element multiplication and addition process

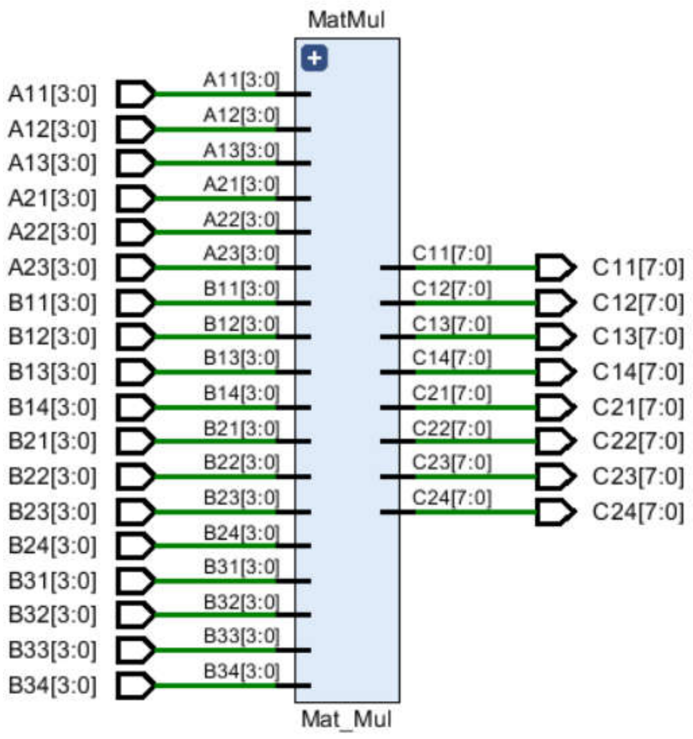
3.4 Block Specification

This section lists each component specification for further information and understanding.

Block name	Full Adder – **FA**		
Description			
This block does the bit (a) a bit (b) addition. Note that an input carry bit (ci) can be received and it will take part of the addition operation. As a result a bit is released (so) and a possible carry out bit (co) will be reported.			
Port name	Port direction	Port type	Description
ci	Input	std_logic	Carry in bit.
a	Input	std_logic	A-bit.
b	Input	std_logic	B-bit.
co	Output	std_logic	Carry out bit.
so	Output	std_logic	Result bit.

Block name	Multiplier		
Description			
This block does the 2's complement multiplication of 4 bit inputs and give 8 bit output.			
Port name	Port direction	Port type	Description
A[3:0]	Input	std_logic_vector(3 downto 0)	A bit vector.
B[3:0]	Input	std_logic_vector(3 downto 0)	B bit vector.
c[7:0]	Output	std_logic_vector(7 downto 0)	C bit vector.

Block name	Ripple Carry Adder – **RCA**		
Description			
RCA block performs the addition of two vectors of bits. It uses the existing FA block which already implements the bit to bit addition.			
Port name	Port direction	Port type	Description
ci	Input	std_logic	Carry in bit.
a	Input	std_logic_vector(2*Nbit-1 downto 0)	A-bits vector.
b	Input	std_logic_vector(2*Nbit-1 downto 0)	B-bits vector.
co	Output	std_logic	Carry out bit.
so	Output	std_logic_vector(2*Nbit-1 downto 0)	Result-bits vector.

Block name	Matrix Multipler schematics
Description	 MatMul A11[3:0] A12[3:0] A13[3:0] A21[3:0] A22[3:0] A23[3:0] B11[3:0] B12[3:0] B13[3:0] B14[3:0] B21[3:0] B22[3:0] B23[3:0] B24[3:0] B31[3:0] B32[3:0] B33[3:0] B34[3:0] C11[7:0] C12[7:0] C13[7:0] C14[7:0] C21[7:0] C22[7:0] C23[7:0] C24[7:0] Mat_Mul
This block performs the multiplication of two 2*3 and 3*4 matrices. All the elements are represented in 4 bits. What is seen from the diagram is the schematic of synthesis with wrapper of 2 by 3 and 3 by 4 input constraints and 2 by 4 output constraint.	

Port name	Port direction	Port type	Description
A	Input	2d array of std_logic_vector(3 downto 0)	Multiplicand
B	Input	2d array of std_logic_vector(3 downto 0)	Multiplier
P	Output	2d array of std_logic_vector(3 downto 0)	Product

4. Test Plan

In order to test the matrix multiplier we need to define specific configuration for the sizes of operand matrices. Also, we need to analyze possible input conditions to show operations that could cause faces. In this chapter we will describe our testing configurations and input conditions.

4.1. Test configuration

For testing process, the next configuration of matrices are used:

Matrix 1: N x M = 2 rows and 3 columns.

Matrix 2: M x P = 3 rows and 4 columns. Which means the values

N = 2, M= 3, P = 4

These matrices have elements represented in 2's complement on 4 bits so these values are in this interval [-8, 7]. Moreover, the result matrix has the following configuration:

Output Matrix: N x P = 2 rows and 4 columns. The stored values in this output matrix are represented in 2's complement too but on 8 bits.

4.2. Test bench

The frequent problem with such matrix multiplier architectures is if the connections are done **wrong**. In this case it looks working fine for some cases i.e. *matrices of similar element* value and it shows error for other cases i.e. *matrices of different element* value. So, our testing configuration is made to address the possible interconnection errors by varying the values of input matrix element. Also, for testing the system with the specified configuration, we wrote a test bench code which set the test setting the values of N=2, M=3 and P=3 in the test bench architecture definition.

```
---* Here specify the sizes of the matrixes to be given to the matrix multiplier.*----
generic map( N => 2, M => 3, P => 4)
```

Code Snippet 1: Testing configuration

Also we gave the values to the test bench signal in hexadecimal notation. Each is represented in a vector format like:

```
Atb <= ((x"3",x"3",x"3"),(x"1",x"1",x"1"));
Btb <= ((x"2",x"2",x"2",x"2"),(x"2",x"2",x"2",x"2"),(x"2",x"2",x"2",x"2"));
wait for 100 ns;
```

Code Snippet 2: Describing the matrix values in vector of hex numbers

The matrix can be seen in the wave form window Msgs column like:

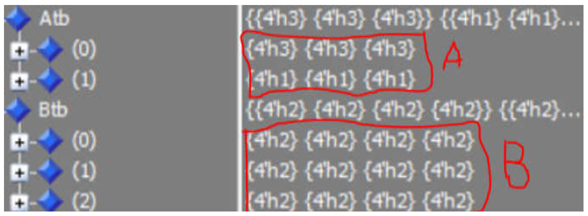
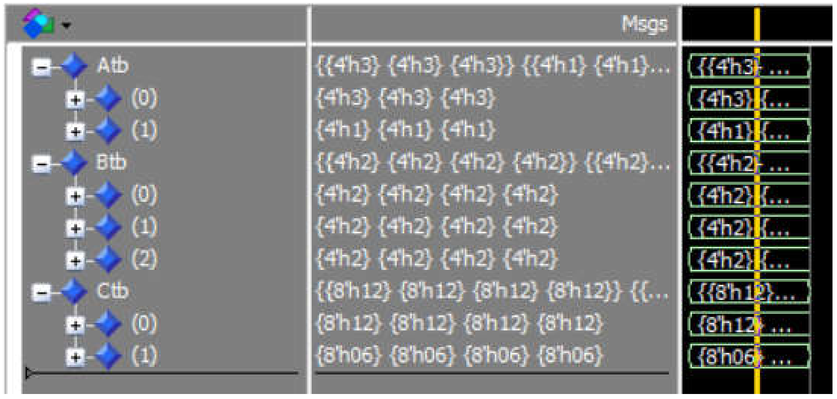

```
Atb      {{4'h3} {4'h3} {4'h3}} {{4'h1} {4'h1}...
 (0)     {4'h3} {4'h3} {4'h3}
 (1)     {4'h1} {4'h1} {4'h1}
Btb      {{4'h2} {4'h2} {4'h2} {4'h2}} {{4'h2}...
 (0)     {4'h2} {4'h2} {4'h2} {4'h2}
 (1)     {4'h2} {4'h2} {4'h2} {4'h2}
 (2)     {4'h2} {4'h2} {4'h2} {4'h2}
```

Figure 5: Test value setups for the matrices

The cases we explicitly observe include:

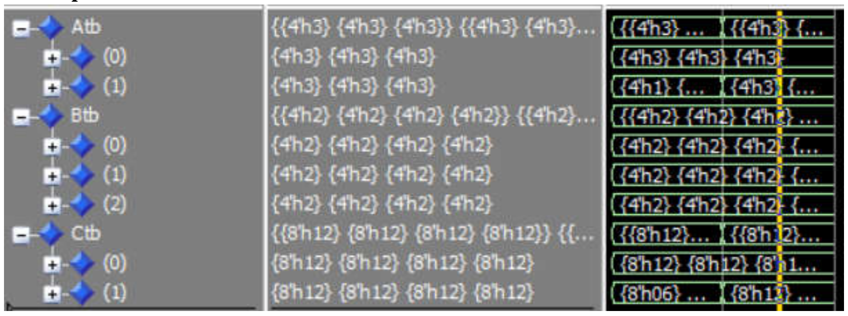
i) Multiplication of matrices of positive numbers:



```
                Msgs
Atb             {{4'h3} {4'h3} {4'h3}} {{4'h1} {4'h1}...   {{4'h3} ...
  (0)           {4'h3} {4'h3} {4'h3}                       {4'h3} {...
  (1)           {4'h1} {4'h1} {4'h1}                       {4'h1} {...
Btb             {{4'h2} {4'h2} {4'h2} {4'h2}} {{4'h2}...   {{4'h2} ...
  (0)           {4'h2} {4'h2} {4'h2} {4'h2}                {4'h2} {...
  (1)           {4'h2} {4'h2} {4'h2} {4'h2}                {4'h2} {...
  (2)           {4'h2} {4'h2} {4'h2} {4'h2}                {4'h2} {...
Ctb             {{8'h12} {8'h12} {8'h12} {8'h12}} {{...    {{8'h12}...
  (0)           {8'h12} {8'h12} {8'h12} {8'h12}            {8'h12} ...
  (1)           {8'h06} {8'h06} {8'h06} {8'h06}            {8'h06} ...
```

Figure 6: Test 1

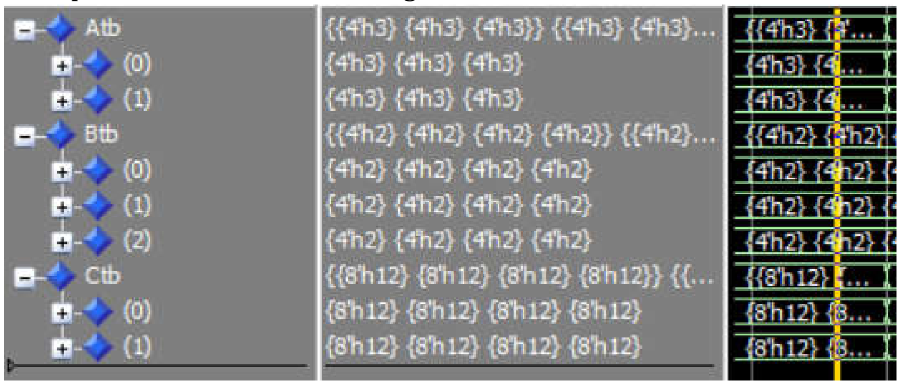
ii) Multiplication of matrices of different element values:



```
Atb             {{4'h3} {4'h3} {4'h3}} {{4'h3} {4'h3}...   {{4'h3} ...    {{4'h3} {...
  (0)           {4'h3} {4'h3} {4'h3}                       {4'h3} {4'h3} {4'h3}
  (1)           {4'h3} {4'h3} {4'h3}                       {4'h1} {...    {4'h3} {...
Btb             {{4'h2} {4'h2} {4'h2} {4'h2}} {{4'h2}...   {{4'h2} {4'h2} {4'h2} ...
  (0)           {4'h2} {4'h2} {4'h2} {4'h2}                {4'h2} {4'h2} {4'h2} {...
  (1)           {4'h2} {4'h2} {4'h2} {4'h2}                {4'h2} {4'h2} {4'h2} {...
  (2)           {4'h2} {4'h2} {4'h2} {4'h2}                {4'h2} {4'h2} {4'h2} {...
Ctb             {{8'h12} {8'h12} {8'h12} {8'h12}} {{...    {{8'h12}...    {{8'h12}...
  (0)           {8'h12} {8'h12} {8'h12} {8'h12}            {8'h12} {8'h12} {8'h1...
  (1)           {8'h12} {8'h12} {8'h12} {8'h12}            {8'h06} ...    {8'h12} ...
```

Figure 7: Test 2

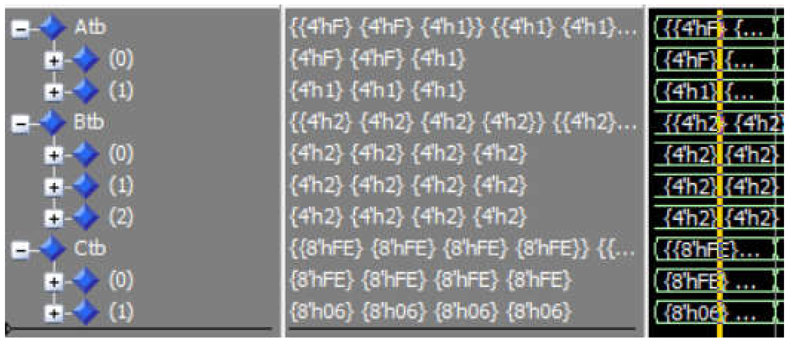
iii) Multiplication of matrices having different element each one:



```
Atb             {{4'h3} {4'h3} {4'h3}} {{4'h3} {4'h3}...   {{4'h3} {4'...
  (0)           {4'h3} {4'h3} {4'h3}                       {4'h3} {4...
  (1)           {4'h3} {4'h3} {4'h3}                       {4'h3} {4...
Btb             {{4'h2} {4'h2} {4'h2} {4'h2}} {{4'h2}...   {{4'h2} {4'h2}
  (0)           {4'h2} {4'h2} {4'h2} {4'h2}                {4'h2} {4'h2}
  (1)           {4'h2} {4'h2} {4'h2} {4'h2}                {4'h2} {4'h2}
  (2)           {4'h2} {4'h2} {4'h2} {4'h2}                {4'h2} {4'h2}
Ctb             {{8'h12} {8'h12} {8'h12} {8'h12}} {{...    {{8'h12} {...
  (0)           {8'h12} {8'h12} {8'h12} {8'h12}            {8'h12} {8...
  (1)           {8'h12} {8'h12} {8'h12} {8'h12}            {8'h12} {8...
```

Figure 8: Test 3

iv) Multiplication of negative number by positive. note here that **4'hF** means **-1** in 2's complement and 8'hFE means -2 in 2's complement:



```
Atb     {{4'hF} {4'hF} {4'h1}} {{4'h1} {4'h1}...   {{4'hF} {...
  (0)   {4'hF} {4'hF} {4'h1}                       {4'hF} {...
  (1)   {4'h1} {4'h1} {4'h1}                       {4'h1} {...
Btb     {{4'h2} {4'h2} {4'h2} {4'h2}} {{4'h2}...   {{4'h2} {4'h2}
  (0)   {4'h2} {4'h2} {4'h2} {4'h2}                {4'h2} {4'h2}
  (1)   {4'h2} {4'h2} {4'h2} {4'h2}                {4'h2} {4'h2}
  (2)   {4'h2} {4'h2} {4'h2} {4'h2}                {4'h2} {4'h2}
Ctb     {{8'hFE} {8'hFE} {8'hFE} {8'hFE}} {{...   {{8'hFE}...
  (0)   {8'hFE} {8'hFE} {8'hFE} {8'hFE}            {8'hFE} ...
  (1)   {8'h06} {8'h06} {8'h06} {8'h06}            {8'h06} ...
```

Figure 9: Test 4

5. Synthesis

Design without synthesis is good for nothing, so we proceed to synthesizing the design. However, it is difficult to synthesize the N*M and M*P multiplier in hardware. Since the exact values should be known in order to construct the electronic circuit. As a result we decide to use the test bench criteria specified in the project description, which is N = 2, M = 3, P = 4. So In order to insert these limits we wrote a wrapper code, i.e. **Mat_Mul_wrapper.vhdl**, which basically map/set these numbers.

After the design elaboration and synthesis using Xilinx Vivado tool we take the following observations:

I. Only 3.05 % (536 out of 17600) of the available LUT is used from the slice logic.
II. 9 LUTs are used from the memory logic possibly with the effect of no latching

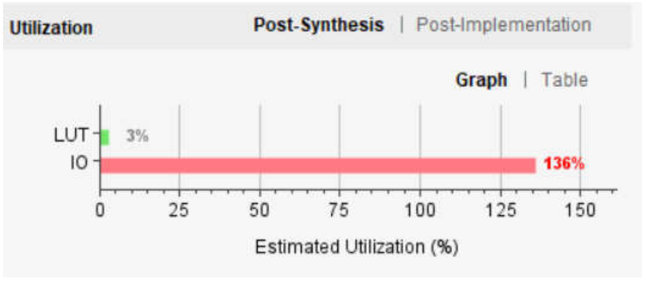
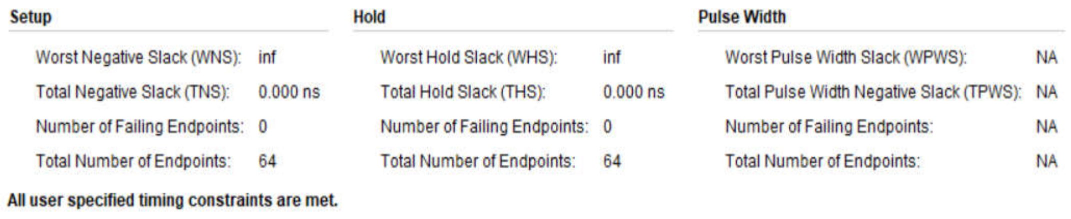


Figure 10: Synthesis Utilization

III. Also the worst negative slack has a zero value because our system is purely combinational and involves no latching in the design

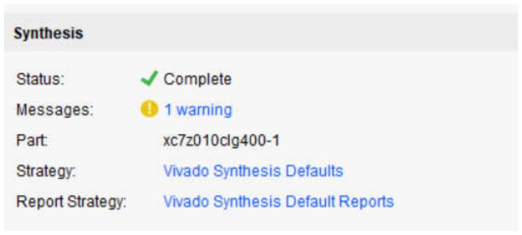


Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	inf	Worst Hold Slack (WHS):	inf	Worst Pulse Width Slack (WPWS):	NA
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	NA
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	NA
Total Number of Endpoints:	64	Total Number of Endpoints:	64	Total Number of Endpoints:	NA

All user specified timing constraints are met.

Figure 11: Timing report

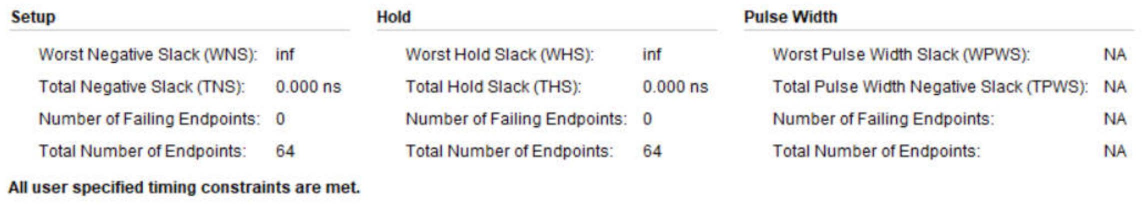
IV. There is 1 warning saying *"[Constraints 18-5210] No constraint will be written out*". This is expected of our design because the number of inputs, i.e. 72 input bits, can't be fit on Xilinx Zybo board.



Synthesis

Status:	✓ Complete
Messages:	1 warning
Part:	xc7z010clg400-1
Strategy:	Vivado Synthesis Defaults
Report Strategy:	Vivado Synthesis Default Reports

V. In the design timing summary we observed that all the timing constraints are met. Since our system is not constrained with clock the WNS and TNS are 0.00ns. Delays are only due to wire connections, assuming the gates have 0 delay compared to the user, which are to be measured after implementation.



Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	inf	Worst Hold Slack (WHS):	inf	Worst Pulse Width Slack (WPWS):	NA
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	NA
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	NA
Total Number of Endpoints:	64	Total Number of Endpoints:	64	Total Number of Endpoints:	NA

All user specified timing constraints are met.

VI. We tried to implement the design to see the propagation timing and power results. Since no constraint is specified the tool tries to automatically create constraint and implement the circuit. But, eventually the tool stop implementation saying "*Place Design Error*", specifically "**I/O implementation impossible"** and **"I/O placer failed to find solution"**. This happens due to a large number of input and output pins required in our system. I.e. if we see inputs pins number only A and B = 2*3 + 3*4 = 18; the number of bits per element is 4; so the total number of input pins are = 4 * 18 = 72 input pins which is much larger than the number of pins in Zybo board.

6. Conclusion

Matrix multipliers have been becoming the center of attention with the rapid boost of machine learning. Circuits good at solving linear and nonlinear equation problems contribute to this machine learning progress. So In order to compute such solutions fast we need processors which can manipulate matrices.

In our project we developed, simulated and synthesized a matrix multiplier. We couldn’t be able to measure the time delay for the system because the gates are assumed to be with zero delay which is wrong and the interconnection delays are not also measured because it can’t be implemented. The good feature is there is a small number of transistors relative to latched versions which will have good impact with regard to the cost, if it is used for mass production.

References

[1] Diligent Newsletter, What are constraint files, https://reference.digilentinc.com/learn/software/tutorials/vivado-xdc-file, visited on 12/2018,

[3]. Skiena, Steven (2008). *The Algorithm Design Manual*. Springer. pp. 45–46, 401–403.