University of Pisa
MSc in Computer Engineering
Electronic Systems
2017/2018

Matrix Multiplier

Report and VHDL implementation

Federico ROSSI

Contents

1 Introduction

1.1 State-of-art digital multipliers

Shift and add This is the most common multiplication algorithm (i.e. the one we do by hand). Given two base β numbers X, C on n digits and a same base number Y on m digits, we want to compute $P = X * Y + C$. Thus the iterative algorithm:

$$\begin{cases} P_0 = C \\ P_{i+1} = y_i * \beta^i * X + P_i \end{cases}$$

The generic iteration P_i is called *partial product.* At the end, the iteration term P_m will be exactly P. This algorithm is quite complex to implement with a sequential network.

Booth algorithm The idea here is focusing on 0 bit chains in multiplicand (also chains of 1 are meaningful, since they can be obtained by subtracting two numbers with chains of 0. Example: Instead of multiplying by $(01111)_2$ one can readily multiply by $(10000-00001)_2$, thus implying two simple shifts).

Suppose to multiply $m = m_{x-1}m_{x-2}...m_1m_0$ and $r = r_{y-1}r_{y-2}...r_1r_0$ $(m*r)$. Before starting, we have to define values for A (value to add), S (value to subtract) and initial value for product P (all of them will have size $x+y+1$)

- Most significant bits of A are set to m, remaining $y+1$ to 0;
- Most significant bits of B are set to $-m$ in 2's complement, remaining as before.
- Most significant x bits of P to 0 chained on the right with value of r, others at 0.

Thus the algorithm iterates y times this operation: takes 2 LSB from P, computes **Operation** according to their value, then shifts arithmetically P to the right by 1 digit and starts over. At the end, the product is obtained dropping the LSB from P.

p_i	p_{i-1}	$p_{i-1}-p_i$	**Operation**
0	0	0	Nothing. Middle of 0 chain
1	0	-1	Beginning of 1 chain. Add S to P
1	1	0	Nothing. Middle of 1 chain
0	1	1	End of 1 chain. Add A to P

Wallace tree algorithm This algorithm is a very hardware-efficient one. Suppose to multiply $m = m_{x-1}m_{x-2}...m_1m_0$ and $r = r_{x-1}r_{x-2}...r_1r_0$ $(m*r)$. It has basically three steps:

1. Multiply each bit of one of the argument, by each bit of the other, producing x^2 results (wires). Eeach result is assigned a weight w, according to digit positions of the two bits:
$$m_a * r_b \Leftrightarrow w = 2^{a+b} \tag{1}$$
(Note that this first multiplication is basically an AND operation);
2. Take any group of 3 wires with same weights and sum them with a *Full Adder*, resulting in a wire with same weight plus another wire with the next higher weight. Then take any group of 2 wires with same weights and sum them with a *Half adder*, resulting in wires as before. If there is only one wire left, connect it to next layer of the multiplier;
3. Sum all the wires from the last reduction layer into the product P;

Comparing this algorithm to the first one, here the multiplier will yield to the result within $log(x)$ reduction layers, being x the number of digits in operands, instead of x w.r.t the first algorithm. Moreover this algorithm can be combined with Booth encoding to perform fewer additions than the basic Wallace algorithm.

1.2 Matrix multiplication algorithm

Given two matrices $A \in \mathbb{Z}^{n*m}, B \in \mathbb{Z}^{m*p}$ the result will be:

$$P \in \mathbb{Z}^{n*p}, P_{ij} = \sum_{k=1}^{m} A_{ik}B_{kj} \tag{2}$$

A simple algorithm to compute multiplication is the following:

```
for i in 0 to rowNumberA−1 loop
    for j in 0 to columnNumberB−1 loop
        for k in 0 to columnNumberA−1 loop
            P(i)(j) := P(i)(j) + (A(i)(k) * B(k)(j));
        end loop;
    end loop;
end loop;
```

1.3 Finite arithmetic sizing

In order to avoid any finite arithmetic's error in the operation:

```
P(i)(j) := P(i)(j) + (A(i)(k) * B(k)(j));
```

we need to size the cell element of P. We know that the sum of two numbers in 2's complement on n bits can always be displayed on $n+1$ bits. Moreover we know that product of two numbers in 2's complement on n bits can always be displayed on $n+n=2n$ bits. The latter comes from:

$$-\frac{\beta^n}{2} \le a \le \frac{\beta^n}{2} - 1, -\frac{\beta^n}{2} \le b \le \frac{\beta^n}{2} - 1, p = a*b \Rightarrow -\frac{\beta^{2n}}{4} \le c \le \frac{\beta^{2n}}{4} \tag{3}$$

So, focusing on our computations, the term p_{ij} will be the sum of m numbers on 8 bits (being m the number of columns in left matrix). Since m can be represented in 2's complement on l bits:

$$l = \lceil log_2(m) \rceil + 1 = \lfloor log_2(m) \rfloor + 2 \tag{4}$$

Thus, the maximum value we can obtain from the computation is (this holds both for positive and negative values):

$$p_{ij} \le m*c \le 2^{2n-2} * 2^{\lfloor log_2(m) \rfloor + 2 - 1} = \frac{2^{2n+\lfloor log_2(m) \rfloor}}{2} \tag{5}$$

This means that the number q of bits representing p_{ij} without arithmetic's errors will be:

$$q = 8 + \lfloor log_2(m) \rfloor \tag{6}$$

2 Implemented architecture

2.1 High-level architecture

A block diagram for the implemented architecture is the following:

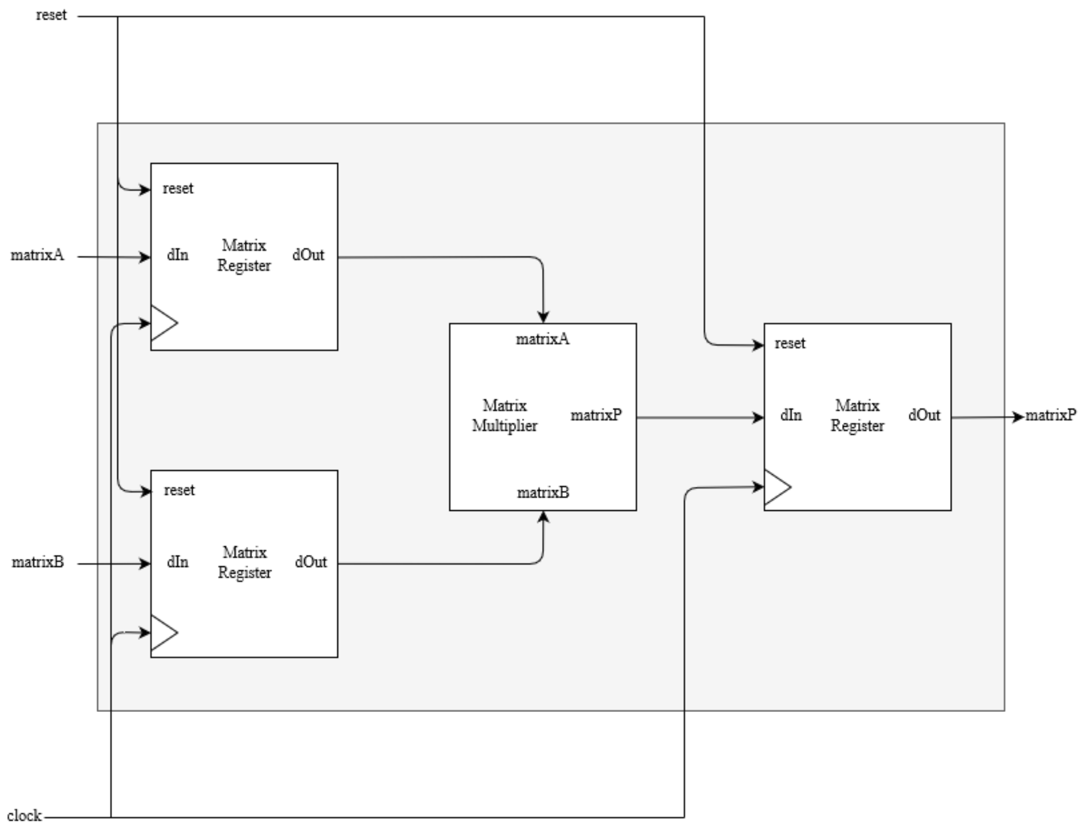


Figure 1: Block diagram for the Matrix Multiplier

2.2 Bit matrix type

The BitMatrix type has been defined in a package called `Common`:

```
package Common is
    type BitMatrix is array(natural range<>,natural range<>) of std_ulogic_vector;
end Common;
```

Note that in order to use this type of syntax, delegating constraining of vector to the modules that use the type (that is - a standard feature from VHDL 2008), Modelsim project has been configured to be compiled with flag -2008.

2.3 Matrix Register

The matrix register is a D-FlipFlop extended to hold a matrix of bytes. The VHDL entity is:

```
entity MatrixRegister is
    generic (
            RowNumber : positive;
            ColNumber : positive;
            cellBits  : positive
    );
    port (
        clock : in std_ulogic;
        reset : in std_ulogic;
        dIn   : in BitMatrix(0 to RowNumber-1,0 to ColNumber-1)(cellBits downto 0);
        dOut  : out BitMatrix(0 to RowNumber-1,0 to ColNumber-1)(cellBits downto 0)
    );
end MatrixRegister;
```

2.4 Matrix Multiplier

This is the core of the entire module. The multiplier is a combinatorial circuit with this entity:

```
entity MatrixMultiplier is
    generic (
            AColNumber : positive;
            BColNumber : positive;
            ARowNumber : positive;
            BRowNumber : positive
        );
    port (
        matrixA : in BitMatrix(0 to ARowNumber-1,0 to AColNumber-1)(3 downto 0);
        matrixB : in BitMatrix(0 to BRowNumber-1,0 to BColNumber-1)(3 downto 0);
        matrixP : out BitMatrix(0 to ARowNumber-1,0 to BColNumber-1)(q downto 0 )
    );
end MatrixMultiplier;
```

Where *q* has to be expanded with:

```
integer(real(7)+floor(log2(real(AColNumber)))
```

Finally, the combinatorial logic function that implements matrix multiplication is:

```
function  Multiply  ( a : BitMatrix; b:BitMatrix) return BitMatrix is
 variable i,j,k : integer:=0;
 variable prod : BitMatrix(...,...)(...):=(others => (others => (others => '0')));
 begin
 for i in 0 to matrixARowNumber-1 loop
  for j in 0 to matrixBColNumber-1 loop
   for k in 0 to matrixAColNumber-1 loop
    prod(i,j) := std_ulogic_vector(signed(prod(i,j)) + (signed(a(i,k)) * signed(b(k,j))));
   end loop;
  end loop;
 end loop;
 return prod;
end Multiply;
```

3 Test Plan

Test plan for multiplier has the main goal to point out any finite arithmetic's error related to undersizing of result. So among all the values that a cell of n bits can assume:

$$-\frac{\beta^n}{2} \leq a \leq \frac{\beta^n}{2} - 1 \tag{7}$$

we focus on the extreme values. Again, among all the configurations that a matrix can assume with this values, we can consider the worst case when limit values are all placed in a row of the first matrix and in a column of the second matrix.

$$\begin{pmatrix} a_1 & \cdots & a_m \\ \cdots & & \\ \cdots & & \end{pmatrix} \times \begin{pmatrix} b_1 & \cdots & \cdots \\ \vdots & & \\ b_m & & \end{pmatrix} \tag{8}$$

So the worst "path" of multiplications and sums will be $\sum_{i=1}^{m} a_i * b_i$. Therefore we have either

$$b_i, a_i = -\frac{\beta^n}{2}, \forall i = 1...m \tag{9}$$

to reproduce the largest positive value, and

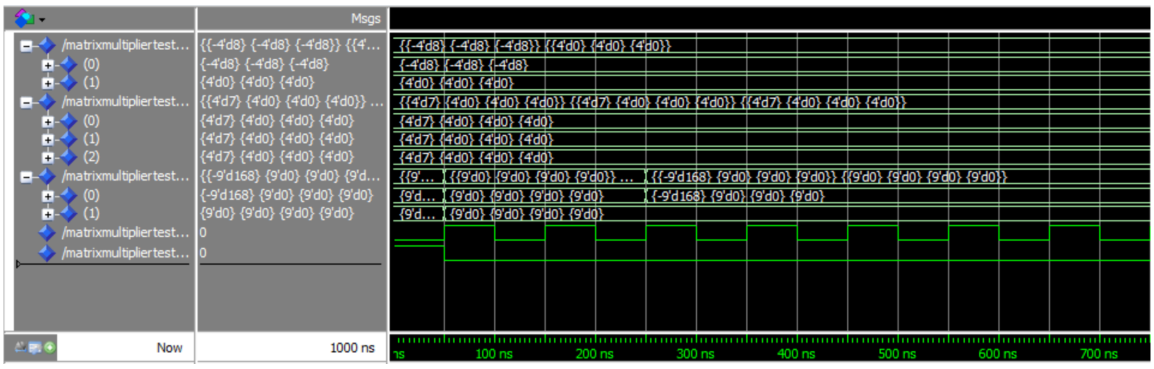
$$a_i = -\frac{\beta^n}{2}, b_i = \frac{\beta^n}{2} - 1, \forall i = 1...m \tag{10}$$

to reproduce the largest (absolute value) negative number.

3.1 Test-bench implementation

We now define a test-bench to simulate in Modelsim the behaviour of the multiplier in those two conditions. In both of them we have matrix $A \in \mathbb{Z}^{2*3}$ and matrix $B \in \mathbb{Z}^{3*4}$, thus resulting product $P \in \mathbb{Z}^{2*4}$

First 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 $(7)_{10} \Leftrightarrow (0111)_2$. We expect to obtain $(-168)_{10}$ in $P_{0,0}$, and this is what we obtain:

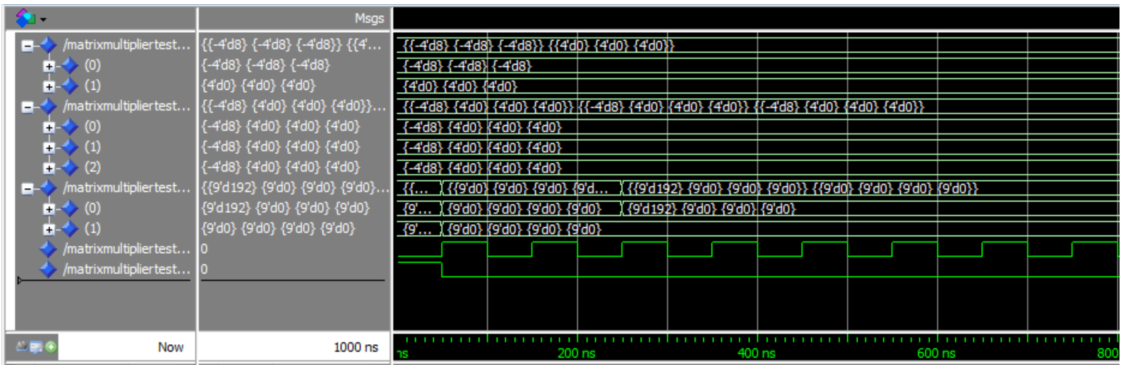


```
                                  Msgs
/matrixmultipliertest...  {{-4'd8} {-4'd8} {-4'd8}} {{4'...   {{-4'd8} {-4'd8} {-4'd8}} {{4'd0} {4'd0} {4'd0}}
  (0)                     {-4'd8} {-4'd8} {-4'd8}              {-4'd8} {-4'd8} {-4'd8}
  (1)                     {4'd0} {4'd0} {4'd0}                 {4'd0} {4'd0} {4'd0}
/matrixmultipliertest...  {{4'd7} {4'd0} {4'd0} {4'd0}} ...    {{4'd7} {4'd0} {4'd0} {4'd0}} {{4'd7} {4'd0} {4'd0} {4'd0}} {{4'd7} {4'd0} {4'd0} {4'd0}}
  (0)                     {4'd7} {4'd0} {4'd0} {4'd0}          {4'd7} {4'd0} {4'd0} {4'd0}
  (1)                     {4'd7} {4'd0} {4'd0} {4'd0}          {4'd7} {4'd0} {4'd0} {4'd0}
  (2)                     {4'd7} {4'd0} {4'd0} {4'd0}          {4'd7} {4'd0} {4'd0} {4'd0}
/matrixmultipliertest...  {{-9'd168} {9'd0} {9'd0} {9'd...    {{9'... {{9'd0} {9'd0} {9'd0} {9'd0}} ...  {{-9'd168} {9'd0} {9'd0} {9'd0}} {{9'd0} {9'd0} {9'd0} {9'd0}}
  (0)                     {-9'd168} {9'd0} {9'd0} {9'd0}       {9'd... {9'd0} {9'd0} {9'd0} {9'd0}      {-9'd168} {9'd0} {9'd0} {9'd0}
  (1)                     {9'd0} {9'd0} {9'd0} {9'd0}          {9'd... {9'd0} {9'd0} {9'd0} {9'd0}
/matrixmultipliertest...  0
/matrixmultipliertest...  0

Now                       1000 ns    ns  100 ns  200 ns  300 ns  400 ns  500 ns  600 ns  700 ns
```

Figure 2: First test from Modelsim

Second test All elements in first row of A and in first column of B are equal to $(-8)_10 \Leftrightarrow (1000)_2$. We expect to obtain $(192)_{10}$ in $P_{0,0}$, and this is what we obtain:



```
                                  Msgs
/matrixmultipliertest...   {{-4'd8} {-4'd8} {-4'd8}} {{4'...
    (0)                    {-4'd8} {-4'd8} {-4'd8}
    (1)                    {4'd0} {4'd0} {4'd0}
/matrixmultipliertest...   {{-4'd8} {4'd0} {4'd0} {4'd0}}...
    (0)                    {-4'd8} {4'd0} {4'd0} {4'd0}
    (1)                    {-4'd8} {4'd0} {4'd0} {4'd0}
    (2)                    {-4'd8} {4'd0} {4'd0} {4'd0}
/matrixmultipliertest...   {{9'd192} {9'd0} {9'd0} {9'd0}...
    (0)                    {9'd192} {9'd0} {9'd0} {9'd0}
    (1)                    {9'd0} {9'd0} {9'd0} {9'd0}
/matrixmultipliertest...   0
/matrixmultipliertest...   0

Now                        1000 ns
```

```
{{-4'd8} {-4'd8} {-4'd8}} {{4'd0} {4'd0} {4'd0}}
{-4'd8} {-4'd8} {-4'd8}
{4'd0} {4'd0} {4'd0}
{{-4'd8} {4'd0} {4'd0} {4'd0}} {{-4'd8} {4'd0} {4'd0} {4'd0}} {{-4'd8} {4'd0} {4'd0} {4'd0}}
{-4'd8} {4'd0} {4'd0} {4'd0}
{-4'd8} {4'd0} {4'd0} {4'd0}
{-4'd8} {4'd0} {4'd0} {4'd0}
{{...  {{9'd0} {9'd0} {9'd0} {9'd...  {{9'd192} {9'd0} {9'd0} {9'd0}} {{9'd0} {9'd0} {9'd0} {9'd0}}
{9'...  {9'd0} {9'd0} {9'd0} {9'd0}  {9'd192} {9'd0} {9'd0} {9'd0}
{9'...  {9'd0} {9'd0} {9'd0} {9'd0}
```

Figure 3: Second test from Modelsim

4 Synthesis and implementation

In Vivado we instantiate matrix dimensions to $N = M = P = 2$, like 2-D rotation matrixes. (In order to let Vivado recognise some VHDL-2008 syntax constructs it was necessary to execute this command into the *tcl console*: `set_property FILE_TYPE {VHDL 2008} [get_files *.vhd]`

4.1 Step 1: RTL Design

First step ends without any warning, producing this layout:

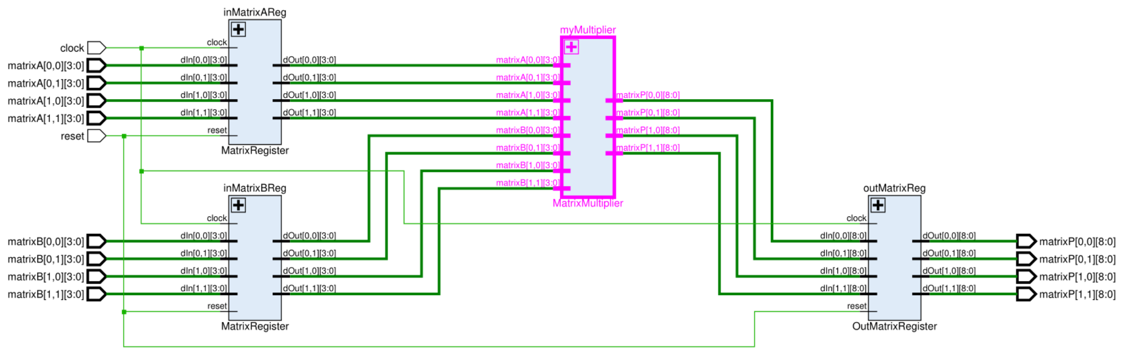


Figure 4: RTL Design from Vivado

Moreover it is interesting how the loop-unrolled design for the multiplier appears:

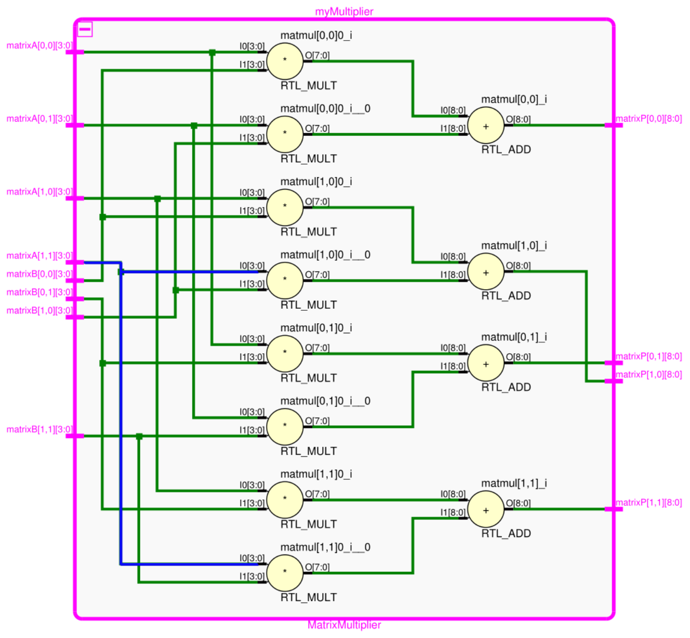
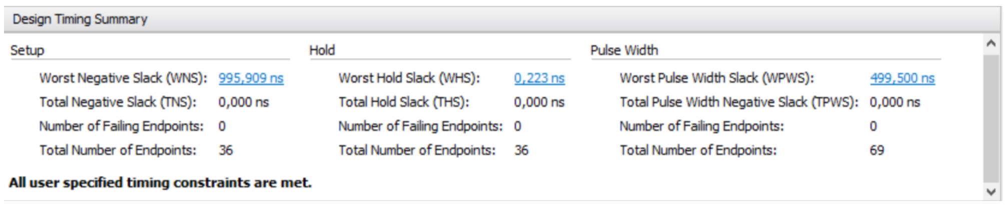


Figure 5: RTL Design from Vivado

4.2 Step 2: Synthesis

We choose a clock frequency constraint of $t_{clk} = 1Mhz$ and we start the synthesis. It completes without warnings.

Timing report Timing report succeeds saying that timing constraints are met:



Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	995,909 ns	Worst Hold Slack (WHS):	0,223 ns	Worst Pulse Width Slack (WPWS):	499,500 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 6: Timing report from VIVADO

However we have some warnings reporting missing delays for input and output wires (clock excluded). This is because we have to guarantee the correct timing by giving an estimation for delays to be considered when using this IP, both when providing inputs and using outputs. At the moment this is out of our purposes, so we will ignore these warnings. As expected the worst path is represented by one of the paths between input registers out and output register input, that traverses the combinatorial logic circuit of the multiplier core.

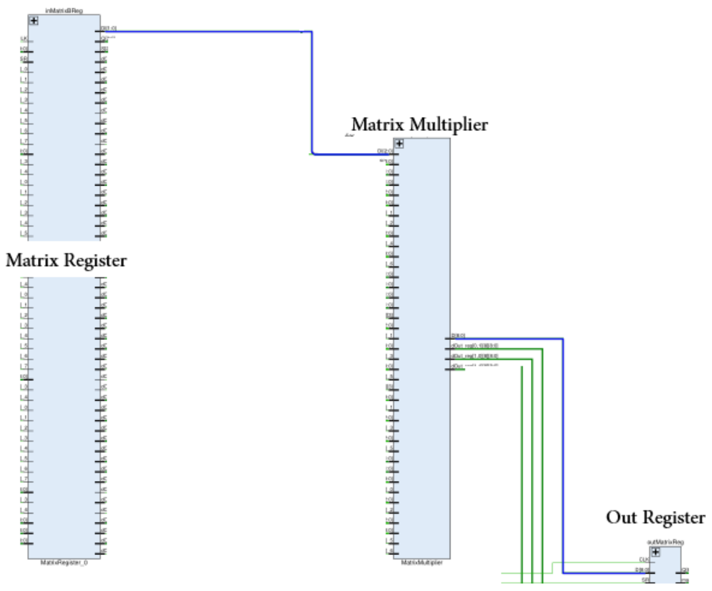


Figure 7: Worst path from VIVADO (other wiring is not displayed)

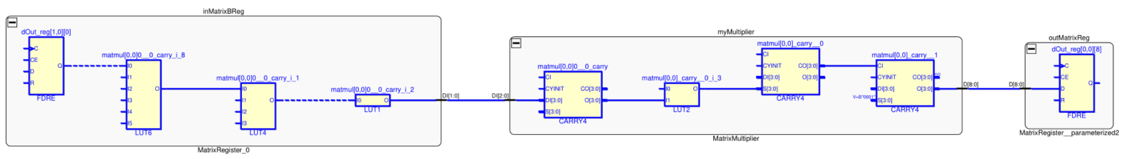


Figure 8: Worst path from VIVADO

Given that slack we can compute how much faster we can drive our clock:

$$t_{clk} = t_{setup} + t_{p-logic} + t_{c-q} + slack \tag{11}$$

The same expression holds with the minimum clock time possible (all the terms except the slack are fixed):

$$t_{clkmin} = t_{setup} + t_{p-logic} + t_{c-q} + slackmin \tag{12}$$

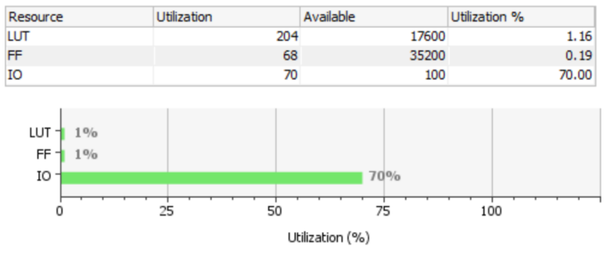
In this case the value of slack is 0 since we are in the limit case. If we subtract the second from the first equation:

$$t_{clkmin} - t_{clk} = 0 - slack \Rightarrow tclkmin = tclk - slack \Rightarrow f_{max} = \frac{1}{tclk - slack} \tag{13}$$

Now, substituting numbers, we obtain:

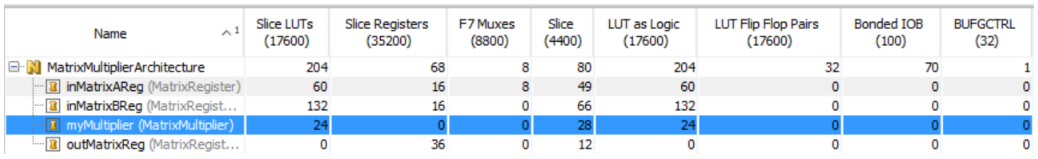
$$f_{max} = \frac{1}{tclk - slack} = \frac{1}{1000ns - 995.909ns} = \frac{1}{4.091ns} \approx 244Mhz \tag{14}$$

Utilisation report Utilisation report shows that we are using most of the I/O pads of the Zync7000 Soc. This makes sense since matrix signals combined with 4-bits cells, although with small matrices, are very I/O expensive.



Resource	Utilization	Available	Utilization %
LUT	204	17600	1.16
FF	68	35200	0.19
IO	70	100	70.00

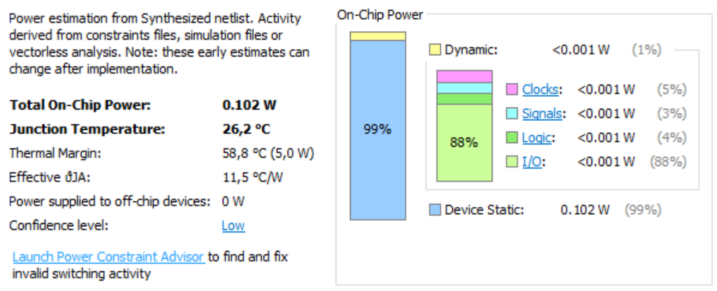
Figure 9: Utilisation report from VIVADO



Name	Slice LUTs (17600)	Slice Registers (35200)	F7 Muxes (8800)	Slice (4400)	LUT as Logic (17600)	LUT Flip Flop Pairs (17600)	Bonded IOB (100)	BUFGCTRL (32)
MatrixMultiplierArchitecture	204	68	8	80	204	32	70	1
inMatrixAReg (MatrixRegister)	60	16	8	49	60	0	0	0
inMatrixBReg (MatrixRegist...	132	16	0	66	132	0	0	0
myMultiplier (MatrixMultiplier)	24	0	0	28	24	0	0	0
outMatrixReg (MatrixRegist...	0	36	0	12	0	0	0	0

Figure 10: Utilisation report from VIVADO

Power report Power report estimates power on chip at $0.102W$, where 99% is related to static power dissipation.



Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	**0.102 W**
Junction Temperature:	**26,2 °C**
Thermal Margin:	58,8 °C (5,0 W)
Effective ϑJA:	11,5 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:	<0.001 W	(1%)
Clocks:	<0.001 W	(5%)
Signals:	<0.001 W	(3%)
Logic:	<0.001 W	(4%)
I/O:	<0.001 W	(88%)
Device Static:	0.102 W	(99%)

Figure 11: Power report from VIVADO

4.3 Step 3: Implementation

Implementation succeeds with a warning, telling us that we are not using the PS7 processor of the Zynq SoC, but we can ignore it. Implementation reports tell us likely the same things as previous step. Slack is reduced to $\approx 994,639ns$ (due to I/O ports being actually assigned, thus more delays to consider in timing constraints), therefore the max frequency (computed as before) is $f_{max} \approx 186Mhz$.

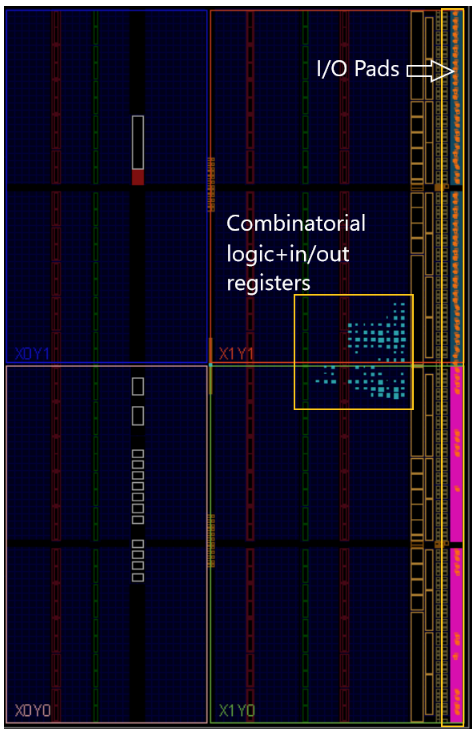


Figure 12: Implemented design from VIVADO

5 Conclusions

From this work we obtained a working matrix multiplier, further work may include using more space-efficient algorithms, like the Strassen's one in order to minimize the area used by the combinatorial logic for the multiplication and also to parallelize combinatorial circuit, reducing the path between in-out registers. This could clearly lead to less constraining bonds on clock frequency, increasing the overall speed of the module. This is central when speaking about matrix multiplication, since it is used extensively in graphic processors to compute 2-D and 3-D transforms on images and models.