UNIVERSITY OF PISA

COMPUTER ENGINEERING MASTRER DEGREE

Matrix Multiplier

Electronics and Communication systems' Project

Giada Anastasi

2019-2020

Contents

1 Introduction

The project goal is to define a VHDL description of a system that could compute the multiplication between two matrices. Starting from a matrix A (NxM) and a matrix B (MxP) made of 4-bit two's complement numbers, we want to obtain a matrix C (NxP) made of Q-bit two's complement numbers, where Q must be choosen in order to avoid finite arithmetic errors.

1.1 State-of-the-art multipliers

There are different algorithms to compute the matrix multiplication:

1.1.1 Array multiplier

Based on "shift and add" algorithm: 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:

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

This algorithm is quite difficult to implement with a sequential network, due to more number of components used to make it. It consumes a lot of power and time.

1.1.2 Booth algorithm

One of most important problem of array multiplier is its execution speed, so a possible solution is that one used by Booth algorithm, that improves performances reducing the number of iterations. In this algorithm three bits are scanned at the same time and the result makes the present pair, which includes two bits and the higher bit of an adjacent lower order pair belongs to the third bit. Then, the booth logic converts the triplets into a set of five control signals, used to perform operations.
Suppose to multiply $m = m_{x-1}m_{x-2}...m_1m_0$ and $r = r_{y-1}r_{y-2}...r_1r_0$. 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 tom, 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

1.1.3 Modified booth multiplier

To improve the efficiency of booth multiplier, the booth encoder here performs various steps simultaneously. In this way the speed of the multiplier increases and the multiplier circuit is get shortened.

1.1.4 Wallace algorithm

This algorithm is a very hardware-efficient one. Suppose to multiply $m = m_{x-1}m_{x-2}...m_1m_0$ and $r = r_{y-1}r_{y-2}...r_1r_0$. 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:

$$m_a * r_b \Leftrightarrow w = 2^{a+b}$$

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 themultiplier;

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:

$$P \in \mathbb{Z}^{n*p}, P_ij = \sum_{k=1}^{m} A_ikB_kj$$

A simple algorithm to compute this multiplication is:

```
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

To avoid any finite arithmetic's error, we have to pay attention to the following 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 two's complement on n bits can always be displayed on $n+1$ bits and that the product of these numbers will be displayed on $n+n=2n$ bits. The latter results comes from the following formula:

$$-\frac{\beta^n}{2} \leq a \leq \frac{\beta^n}{2} - 1, -\frac{\beta^n}{2} \leq b \leq \frac{\beta^n}{2} - 1, p = a * b \Rightarrow -\frac{\beta^{2n}}{4} \leq p \leq \frac{\beta^{2n}}{4} - 1$$

In our case $n = 8$ so

$$-\frac{2^8}{2} \leq p \leq \frac{2^8}{2} - 1$$

Every element of the final matrix will be obtained as a sum of m number like p, where m is the number of columns in left matrix. Since m can be represented in two's complement on l bits:

$$l = \lceil \log_2 m \rceil + 1 = \lfloor \log_2 m \rfloor + 2$$

This means that the maximum value we can obtain (for positive values) is:

$$p_{ij} \leq m * c \leq 2^{2n-2} * 2^{\lfloor \log_2 m \rfloor + 2 - 1} = \frac{2^{2n + \lfloor \log_2 m \rfloor}}{2}.$$

In conclusion the number q of bits representing p_{ij} without arithmetic's errors will be:

$$q = 8 + \lfloor \log_2 m \rfloor.$$

2 Implemented architecture

2.1 High-level architecture

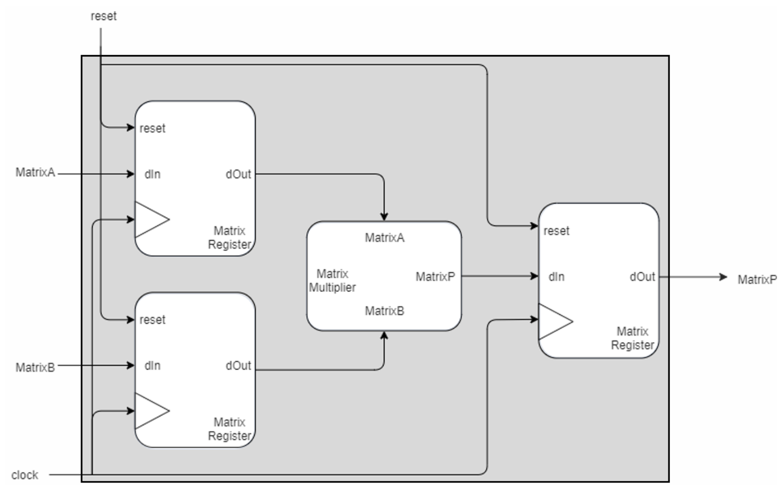


Figure 2.1: Block diagram for Matrix Multiplier

As we can see in the Figure 2.1, we need :

- Three MatrixRegister Modules
- One MatrixMultiplier Module

2.2 Bit matrix type

To define the input of the Matrix Register, that is a Matrix of bytes.
Defined in **Common.vhd** file:

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

2.3 Matrix Register

It is a D-FlipFlop Register 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

Core part of the entire module: it receives the two matrix, A and B, in input and has as output the Product matrix P.

```
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)
        (integer(real(7)+floor(log2(real(matrixAColNumber)))) downto 0 )
    );
end MatrixMultiplier;
```

After the definition of all the modules, we define the combinatorial logic function that implements matrix multiplication :

```
function Multiply (a:BitMatrix; b:BitMatrix) return BitMatrix is
  variable i,j,k : integer:=0;
  variable product : BitMatrix(0 to matrixARowNumber-1,0 to matrixBColNumber-1)
    (integer(real(7)+floor(log2(real(matrixAColNumber)))) downto 0 )
    :=(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
           product(i,j) := std_ulogic_vector(signed(product(i,j))
           + (signed(a(i,k)) * signed(b(k,j))));
        end loop;
      end loop;
    end loop;
    return product;
end Multiply;
```

3 Test Plan

This phase has the main goal to find any finite arithmetic's error related to under-sizing of result. A cell of a matrix is a number in two's complement representation over n bits so this value **a** is:

$$-\tfrac{\beta^n}{2} \leq a \leq \tfrac{\beta^n}{2} - 1$$

We focus our attention on the extreme values. Amoung all the configurations that a matrix can assume with this values, we can consider the worst case. So when the limit values are all placed in a row of the first matrix and in a column of the second matrix. In this case, the worst "combination" of multiplication and sum will be

$$\sum_{i=1}^{m} a_i * b_i$$

with a_i and b_i elements of the worst row of matrixA and column of matrixB. As seen before, we have that

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

to reproduce the largest positive value, and

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

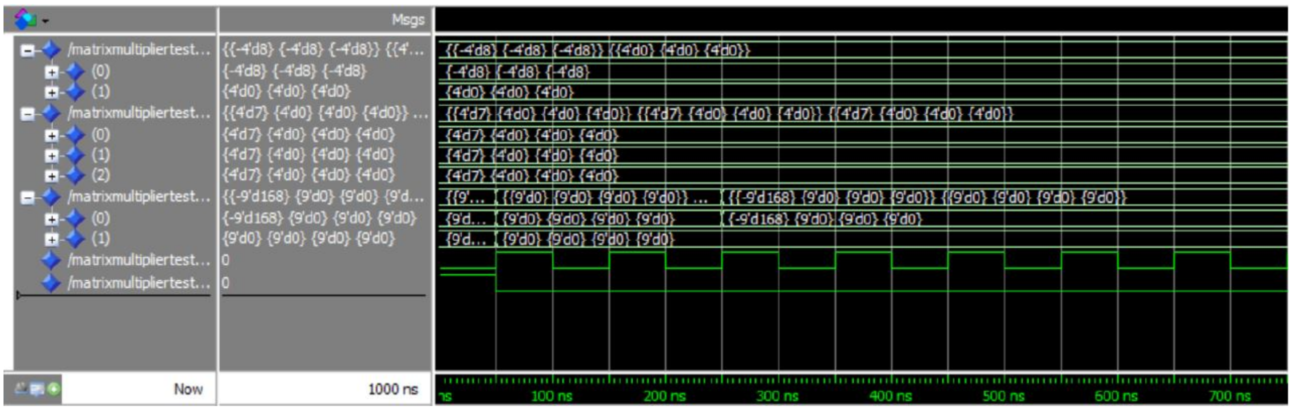
to reproduce the largest, talking about absolute value, negative number.

3.1 Test-bench implementation

In this phase our aim is to define a test-bench to simulate in Modelism 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 the product matrix $P \in \mathbb{Z}^{2*4}$

3.1.1 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:

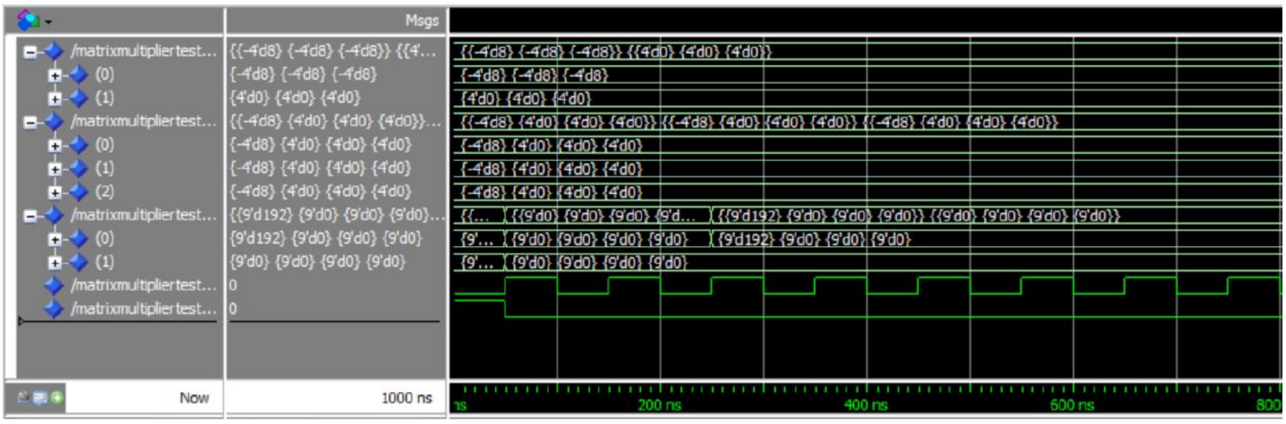

```
                                            Msgs
/matrixmultipliertest...   {{-4'd8} {-4'd8} {-4'd8}} {{4'...
  (0)                      {-4'd8} {-4'd8} {-4'd8}
  (1)                      {4'd0} {4'd0} {4'd0}
/matrixmultipliertest...   {{4'd7} {4'd0} {4'd0} {4'd0}} ...
  (0)                     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

Figure 3.1: First test results

3.1.2 Second test

All elements in first row of A and in first column of B are equal to $(-8)_{10} \Leftrightarrow (1000)_2$. We expected 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}
ns
200 ns
400 ns
600 ns
800

Figure 3.2: Second test results

4 Synthesis and Implementation

Using **Vivado**, we istantiate matrix dimensions to $N = M = P = 2$. Before all the following steps we have to execute in **tcl console** the following command: `set_property FILE_TYPE VHDL 2008 [get_files *.vhd]`

4.1 Step 1: RTL Design

This step produce the following layout:

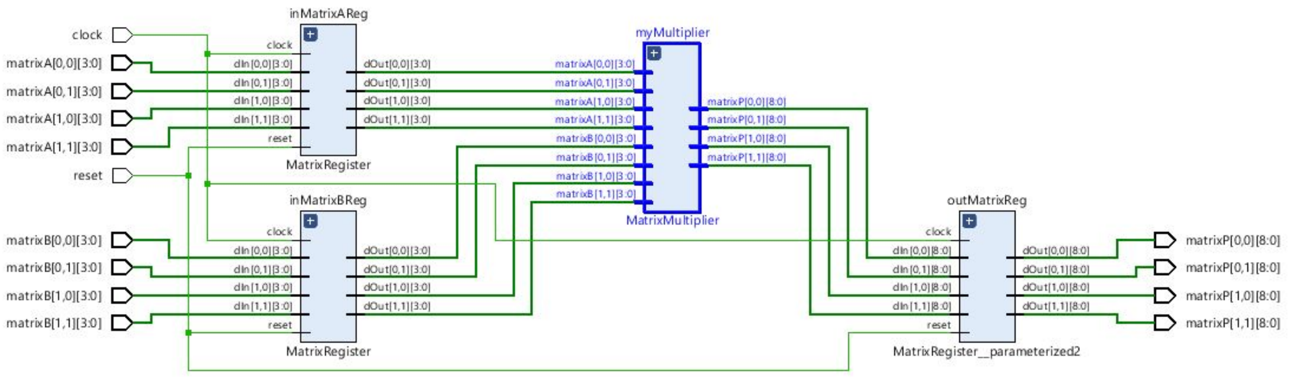


Figure 4.1: RTL Analysis from Vivado

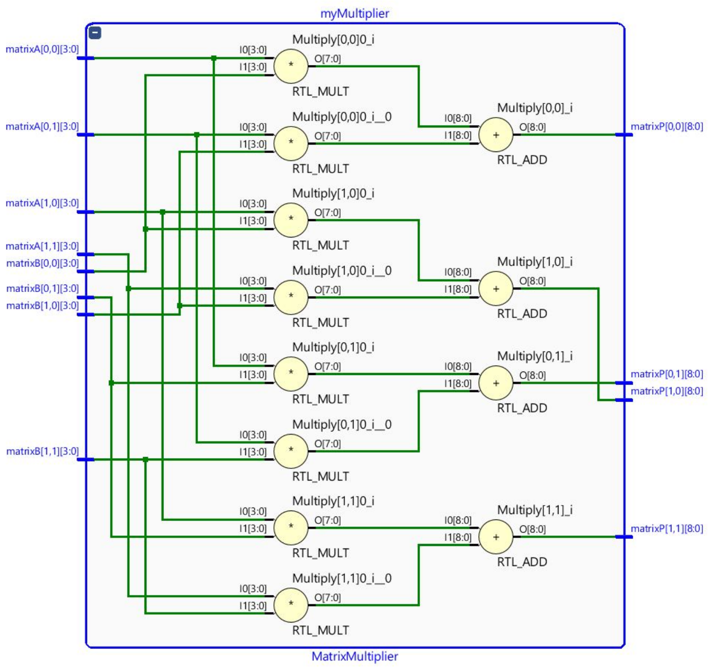
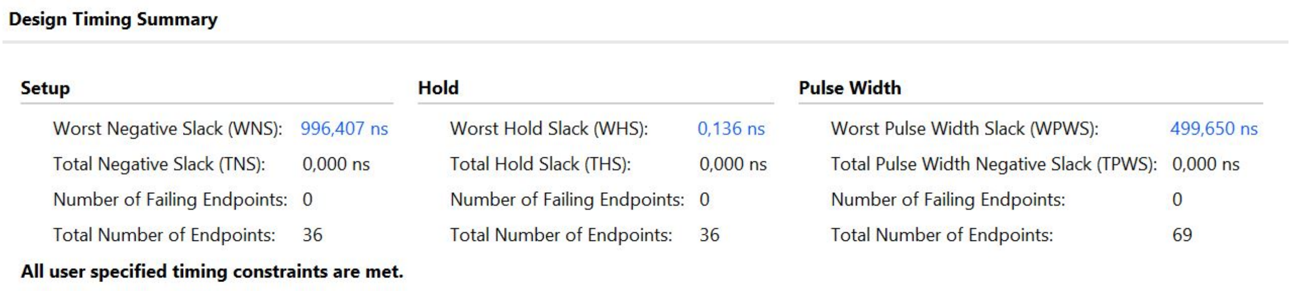


Figure 4.2: Inside the MatrixMultiplier

4.2 Step 2: Synthesis

4.2.1 Critical path

After adding the constraint clock with a period of $t_{clk} = 1Mhz$, we can analyze the report timing showed in the following table:



Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	996,407 ns	Worst Hold Slack (WHS):	0,136 ns	Worst Pulse Width Slack (WPWS):	499,650 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

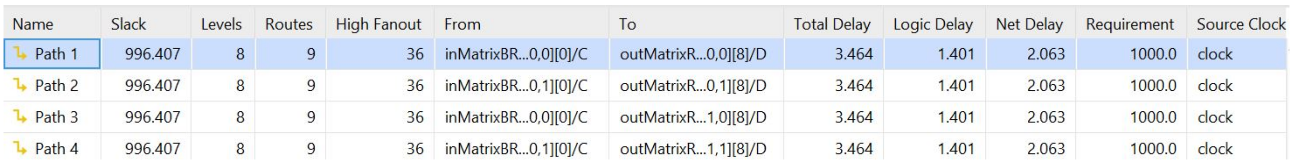
All user specified timing constraints are met.

Figure 4.3: Timing report from Vivado

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

$$T_{SLACK} = T_{CLK} + T_{SUP}.$$

Thanks to `Vivado` we can see what is the worst path:



Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock
Path 1	996.407	8	9	36	inMatrixBR...0,0][0]/C	outMatrixR...0,0][8]/D	3.464	1.401	2.063	1000.0	clock
Path 2	996.407	8	9	36	inMatrixBR...0,1][0]/C	outMatrixR...0,1][8]/D	3.464	1.401	2.063	1000.0	clock
Path 3	996.407	8	9	36	inMatrixBR...0,0][0]/C	outMatrixR...1,0][8]/D	3.464	1.401	2.063	1000.0	clock
Path 4	996.407	8	9	36	inMatrixBR...0,1][0]/C	outMatrixR...1,1][8]/D	3.464	1.401	2.063	1000.0	clock

Figure 4.4: Table from which we choose worst path

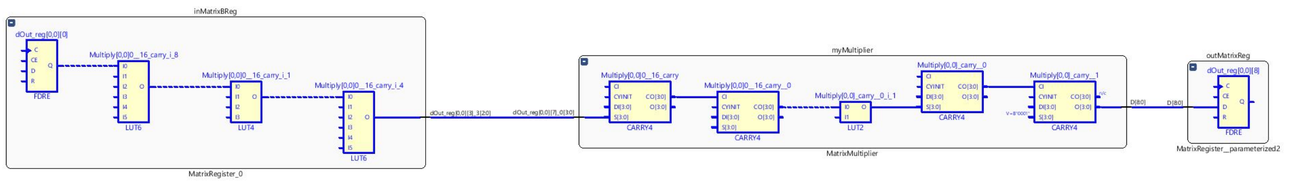


Figure 4.5: Worst path from Vivado

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

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

Same expression can be used to find the minimum clock. Change only slack because other times are fixed:

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

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

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

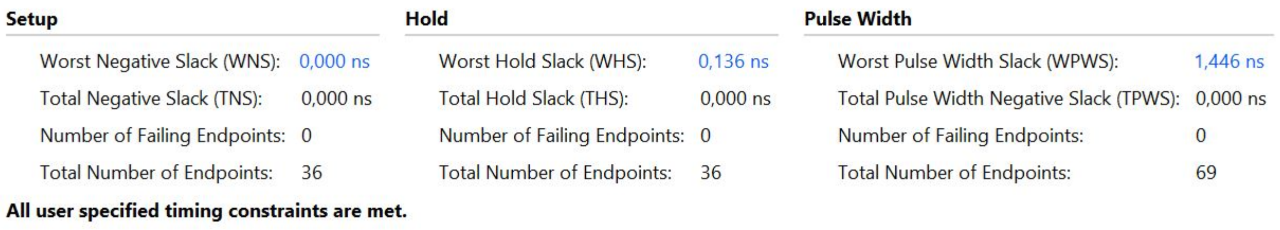
So, at the end we obtain

$$f_{max} = \frac{1}{t_{clk} - t_{slack}} = \frac{1}{1000ns - 996.407ns} = \frac{1}{3.593ns} \approx 278Mhz$$

and from this we have

$$T_{max} = \frac{1}{f_{max}} = \frac{1}{244Mhz} = 3.593ns$$

If we look timing report after using this value, we have `worst negative slack` equal to zero, and that's what we expect.



Setup

Worst Negative Slack (WNS):	0,000 ns
Total Negative Slack (TNS):	0,000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	36

Hold

Worst Hold Slack (WHS):	0,136 ns
Total Hold Slack (THS):	0,000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	36

Pulse Width

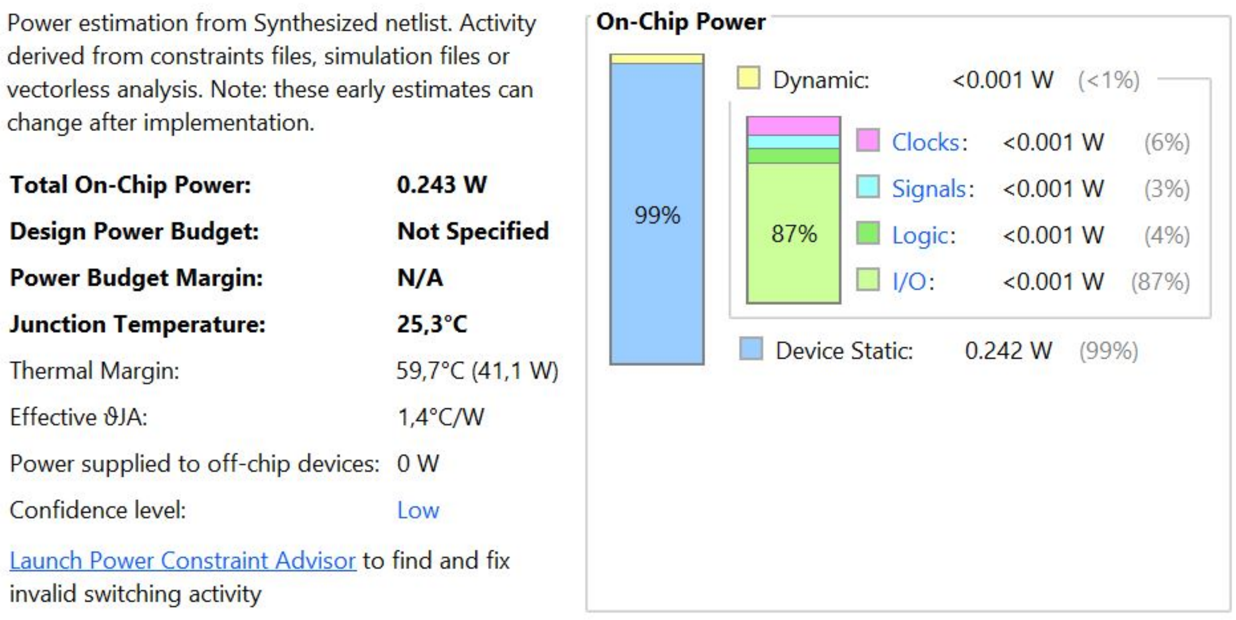
Worst Pulse Width Slack (WPWS):	1,446 ns
Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	69

All user specified timing constraints are met.

Figure 4.6: Timing report with $t_{clock} = 3.593ns$

4.2.2 Power consumption

As we can see in the following figure, the 99% of power on chip 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:	**0.243 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25,3°C**
Thermal Margin:	59,7°C (41,1 W)
Effective ϑJA:	1,4°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

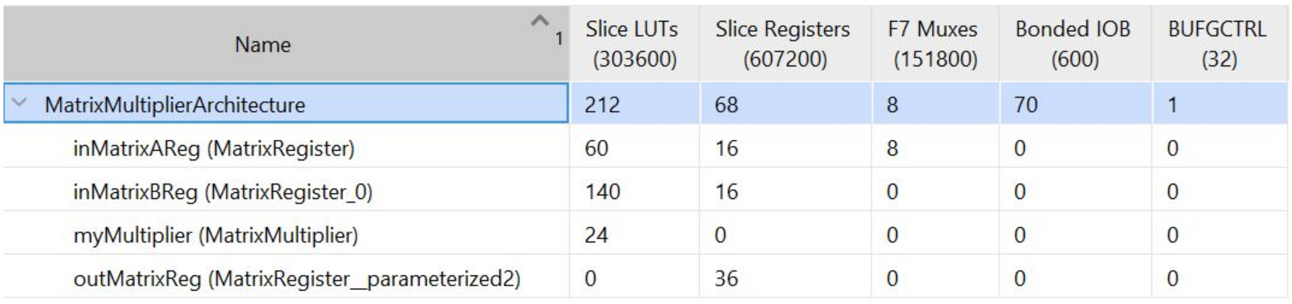
Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

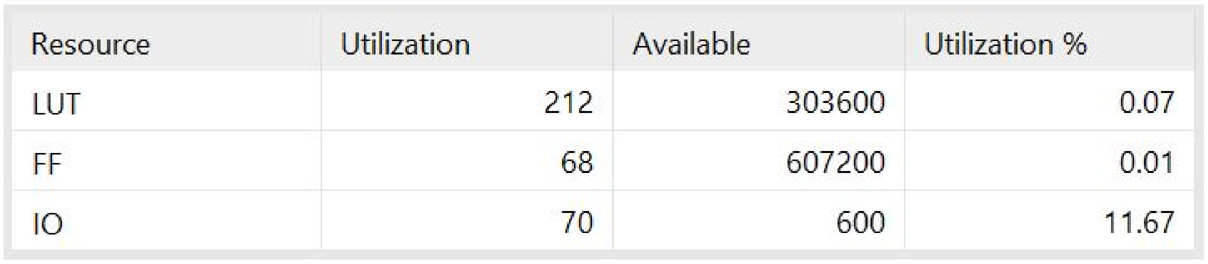
🟨	Dynamic:	<0.001 W	(<1%)
🟪	Clocks:	<0.001 W	(6%)
🟦	Signals:	<0.001 W	(3%)
🟩	Logic:	<0.001 W	(4%)
🟩	I/O:	<0.001 W	(87%)
🟦	Device Static:	0.242 W	(99%)

Figure 4.7: Power report from Vivado

4.2.3 Utilization report



Name	Slice LUTs (303600)	Slice Registers (607200)	F7 Muxes (151800)	Bonded IOB (600)	BUFGCTRL (32)
MatrixMultiplierArchitecture	212	68	8	70	1
inMatrixAReg (MatrixRegister)	60	16	8	0	0
inMatrixBReg (MatrixRegister_0)	140	16	0	0	0
myMultiplier (MatrixMultiplier)	24	0	0	0	0
outMatrixReg (MatrixRegister__parameterized2)	0	36	0	0	0



Resource	Utilization	Available	Utilization %
LUT	212	303600	0.07
FF	68	607200	0.01
IO	70	600	11.67

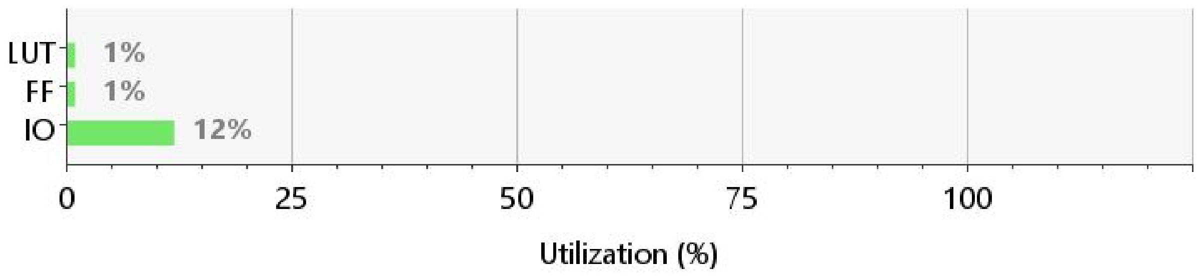


Figure 4.8: Utilization report from Vivado

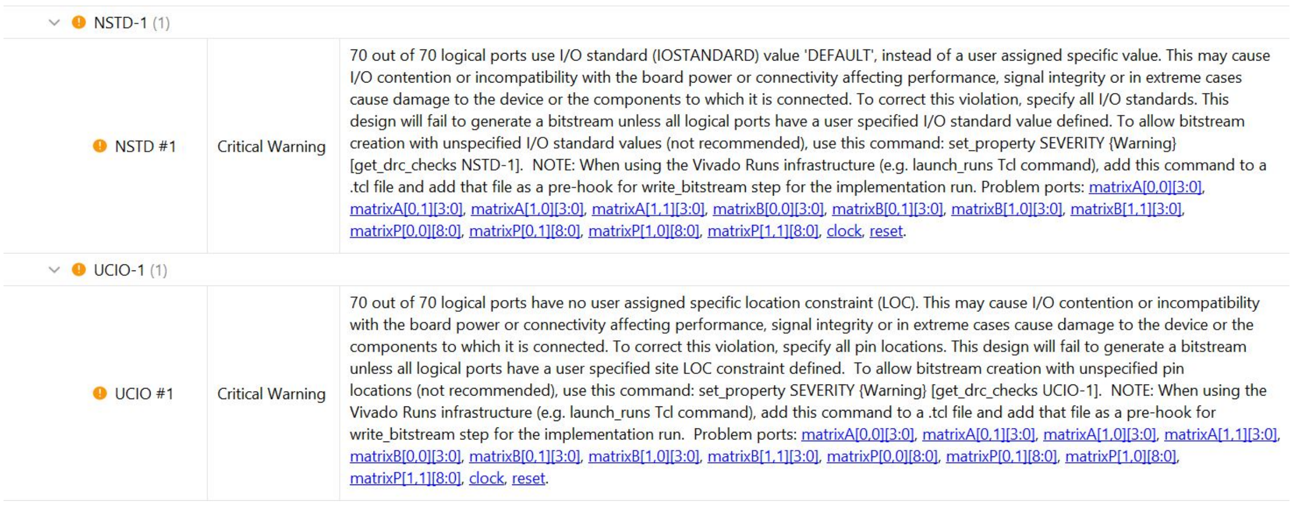
4.3 Step 3: Implementation

This step report us 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 of previous step. Slack is reduced to $\approx 995,158ns$ due to I/O ports being actually assigned, thus more delays to consider in timing constraint. So the max frequency is $f_{max} = 206,5Mhz$.

4.4 Warnings

4.4.1 Critical warnings

As we can see in the following figure, we have two critical warnings that want to notify customers that we need to set `IOSTANDARD` and `PACKAGE PIN` in order to protect devices from damage that could be caused by the tools randomly choosing a pin location, or `IOSTANDARD` without knowledge of the board voltage or connections. This critical warnings is due to the lack of information about our board. This warnings is solved up if we map on a board with enough I/O pin and specify the value.



NSTD-1 (1)

NSTD #1	Critical Warning	70 out of 70 logical ports use I/O standard (IOSTANDARD) value 'DEFAULT', instead of a user assigned specific value. This may cause I/O contention or incompatibility with the board power or connectivity affecting performance, signal integrity or in extreme cases cause damage to the device or the components to which it is connected. To correct this violation, specify all I/O standards. This design will fail to generate a bitstream unless all logical ports have a user specified I/O standard value defined. To allow bitstream creation with unspecified I/O standard values (not recommended), use this command: set_property SEVERITY {Warning} [get_drc_checks NSTD-1]. NOTE: When using the Vivado Runs infrastructure (e.g. launch_runs Tcl command), add this command to a .tcl file and add that file as a pre-hook for write_bitstream step for the implementation run. Problem ports: matrixA[0,0][3:0], matrixA[0,1][3:0], matrixA[1,0][3:0], matrixA[1,1][3:0], matrixB[0,0][3:0], matrixB[0,1][3:0], matrixB[1,0][3:0], matrixB[1,1][3:0], matrixP[0,0][8:0], matrixP[0,1][8:0], matrixP[1,0][8:0], matrixP[1,1][8:0], clock, reset.

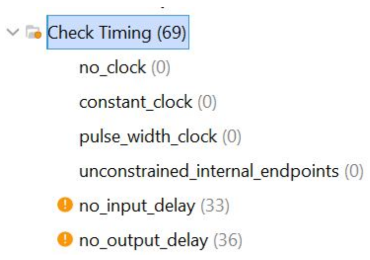
UCIO-1 (1)

UCIO #1	Critical Warning	70 out of 70 logical ports have no user assigned specific location constraint (LOC). This may cause I/O contention or incompatibility with the board power or connectivity affecting performance, signal integrity or in extreme cases cause damage to the device or the components to which it is connected. To correct this violation, specify all pin locations. This design will fail to generate a bitstream unless all logical ports have a user specified site LOC constraint defined. To allow bitstream creation with unspecified pin locations (not recommended), use this command: set_property SEVERITY {Warning} [get_drc_checks UCIO-1]. NOTE: When using the Vivado Runs infrastructure (e.g. launch_runs Tcl command), add this command to a .tcl file and add that file as a pre-hook for write_bitstream step for the implementation run. Problem ports: matrixA[0,0][3:0], matrixA[0,1][3:0], matrixA[1,0][3:0], matrixA[1,1][3:0], matrixB[0,0][3:0], matrixB[0,1][3:0], matrixB[1,0][3:0], matrixB[1,1][3:0], matrixP[0,0][8:0], matrixP[0,1][8:0], matrixP[1,0][8:0], matrixP[1,1][8:0], clock, reset.

Figure 4.9: Critical warnings from Vivado

4.4.2 Time warnings

Warning about the lack of a constraint on the delay. This will decide wheter our ASIC can meet the timings of external devices it is connected to. If these timings are not met, then our ASIC cannot be used with external devices to which it is supposed to interface.



```
Check Timing (69)
    no_clock (0)
    constant_clock (0)
    pulse_width_clock (0)
    unconstrained_internal_endpoints (0)
    no_input_delay (33)
    no_output_delay (36)
```

Figure 4.10: Time warnings from Vivado

5 Conclusions

At the end of this project we obtain a working matrix multiplier. It could be implemented with more space efficient alghorithms, 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 lead to less constraining bounds 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.

Appendix

MatrixMultiplierArchitecture.vhd code

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;        -- for addition & counting
use ieee.numeric_std.all;               -- for type conversions
use ieee.math_real.all;                 -- for the ceiling and log constant calculation functions
library work;
use work.all;
use work.Common.all;                    -- contains definition of BitMatrix type

-- Define new entity for vivado tests

entity MatrixMultiplierArchitecture is
        port (
                clock   : in std_ulogic;
                reset   : in std_ulogic;
                matrixA : in BitMatrix(0 to 1,0 to 1)(3 downto 0);
                matrixB : in BitMatrix(0 to 1,0 to 1)(3 downto 0);
                matrixP : out BitMatrix(0 to 1,0 to 1)(integer(real(7)+floor(log2(real(2)))) downto 0 )
        );
end MatrixMultiplierArchitecture;

architecture MatrixMultiplierArchitecture_arc of MatrixMultiplierArchitecture is

        -- define output sizing as a constant to ease readability later on.

        constant outBits : positive := integer(real(7)+floor(log2(real(2))));

        --import component to handle matrix

        component MatrixRegister
                generic (
                        matrixRowNumber : positive;
                        matrixColNumber : positive;
                        cellBits         : positive
                );
                port (
                        clock : in std_ulogic;
                        reset : in std_ulogic;
                        dIn   : in BitMatrix(0 to matrixRowNumber-1,0 to matrixColNumber-1)(cellBits downto 0);
                        dOut  : out BitMatrix(0 to matrixRowNumber-1,0 to matrixColNumber-1)(cellBits downto 0)
                );
        end component MatrixRegister;

    -- import component to compute multiplication between matrix

        component MatrixMultiplier
                generic (
                        matrixAColNumber : positive;
                        matrixBColNumber : positive;
                        matrixARowNumber : positive;
                        matrixBRowNumber : positive
                );
                port (
                        matrixA  : in BitMatrix(0 to matrixARowNumber-1,0 to matrixAColNumber-1)(3 downto 0);
                        matrixB  : in BitMatrix(0 to matrixBRowNumber-1,0 to matrixBColNumber-1)(3 downto 0);
                        matrixP  : out BitMatrix(0 to matrixARowNumber-1,0 to matrixBColNumber-1)(outBits downto 0 )
                );
        end component MatrixMultiplier;

    -- tests done with N=M=P=2

        signal regAOut : BitMatrix(0 to 1,0 to 1)(3 downto 0);
        signal regBOut : BitMatrix(0 to 1,0 to 1)(3 downto 0);
        signal multOut : BitMatrix(0 to 1,0 to 1)(outBits downto 0 );
begin

inMatrixAReg : MatrixRegister         generic map (matrixColNumber => 2, matrixRowNumber => 2,cellBits => 3)
```

```
                                      port map (dIn => matrixA,clock => clock, reset => reset,dOut => regAOut);
inMatrixBReg : MatrixRegister         generic map (matrixColNumber => 2, matrixRowNumber => 2, cellBits => 3)
                                      port map (dIn => matrixB,clock => clock, reset => reset,dOut => regBOut);
myMultiplier : MatrixMultiplier       generic map(
                                        matrixAColNumber => 2,
                                        matrixARowNumber => 2,
                                        matrixBColNumber => 2,
                                        matrixBRowNumber => 2
                                      )
                                      port map(matrixA => regAOut,matrixB =>regBOut,matrixP=>multOut);
outMatrixReg : MatrixRegister         generic map (matrixColNumber => 2, matrixRowNumber => 2, cellBits => outBits)
                                      port map(dIn =>multOut, dOut=>matrixP, clock => clock, reset => reset);
end MatrixMultiplierArchitecture_arc;
```