**Electronic Project – Matrix Multiplier**

1. Introduction
   1. Problem Description
   2. State-of-art of digital multipliers
   3. Matrix multiplication
   4. Finite arithmetic sizing
2. Matrix Multiplier Implementation
   1. Algorithm
   2. Model Architecture
3. VHDL Implementation
   1. Bit Matrix type definition
   2. Matrix Register
   3. Matrix Multiplier
   4. Matrix Multiplier Architecture
4. Test plan
   1. Python Testbench generator
   2. Testbench implementation
5. Synthesis and Implementation Logic (Vivado part)
   1. RTL design
   2. Synthesis
      1. Timing report and critical path
      2. Device Utilization
      3. Power consumption
      4. Warning analysis (if any)
   3. Implementation
      1. timing report and critical path
      2. Device utilization
      3. Power Consuption
      4. Warning analysis (if any)
6. Conclusions

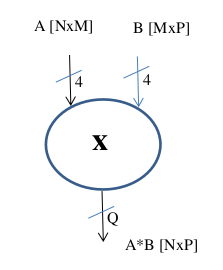
**1. Introduction**

**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.



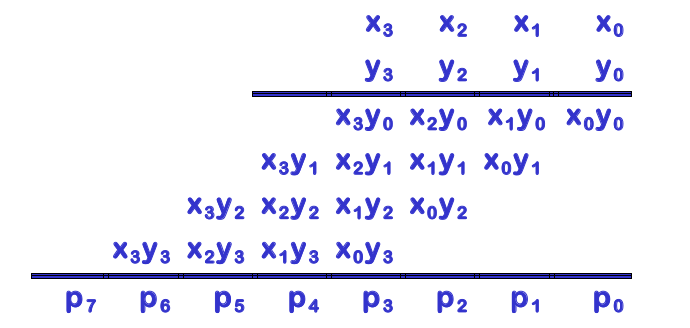
**1.2 State-of-Art of Digital Multipliers**

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 togheter 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 togheter (using a binary addition).

Fig 1 . example of binary multiplication



Using **binary numbers** it must be pay particulary 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 complicatest 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 wolud take a large amount of memory (e.g. for 2 words of 16 bit it wolud need 232 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 an Half Adder.

1. reducing the partial product, sizing it on k bits (that can be in the range [0, n+m-1]
2. 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(ci 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

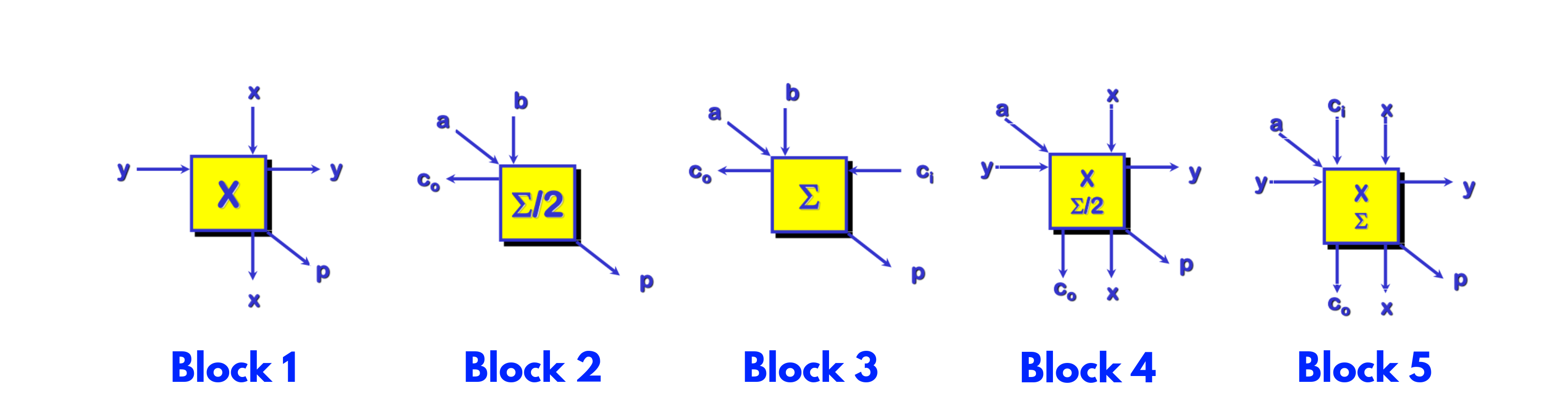


figura 5 tipi di blocco

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

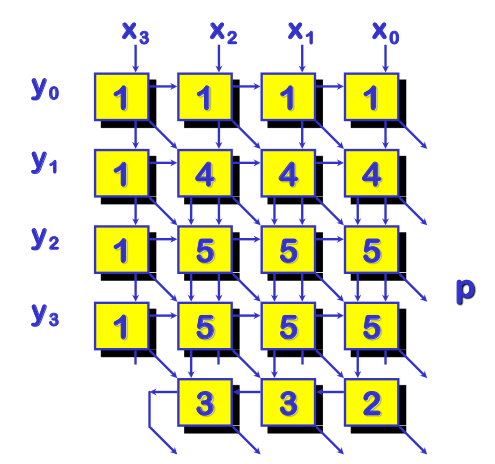


figura parallel multiplier completo (pag 80)

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:

* N2 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 ∈ ℤn\*m , B ∈ ℤm\*p

Fig . Matrices A and B

The matrix product P = AB ∈ ℤn\*p is defined as:

Fig. Matrix P = AB

Where each element can be calculated as follow:

Fig. 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:

*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:

*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:

*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:

*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 :

*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:



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

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

**2.2 Model Architecture**

The block diagram for the implemented architecture is the following:

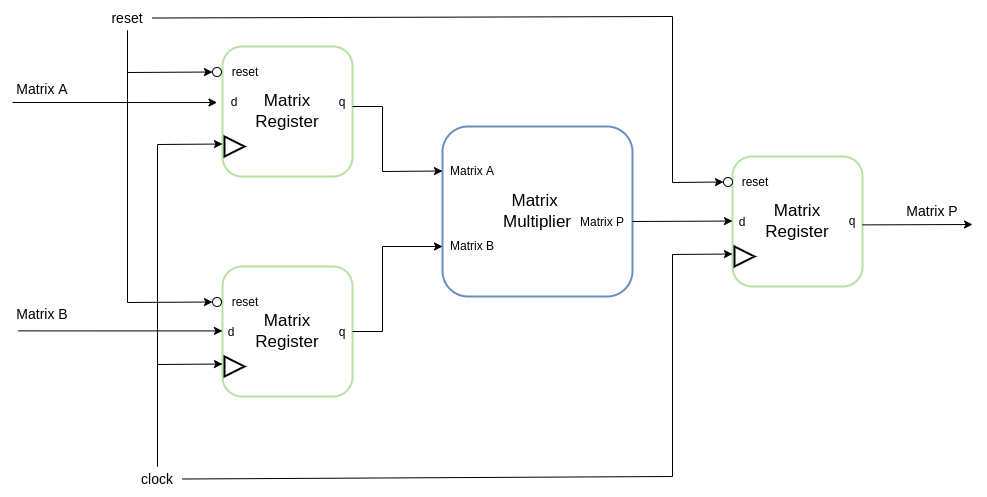


Figure. Matrix Multiplier block diagram

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**

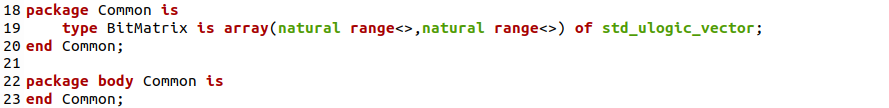
In the following chapter is presented the VHDL code for all components of the Matrix Multiplier. It is used the VHDL 2008 syntax (that allow to use unconstrained *std\_ulogic\_vector* and *std\_logic\_vector* ).

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**

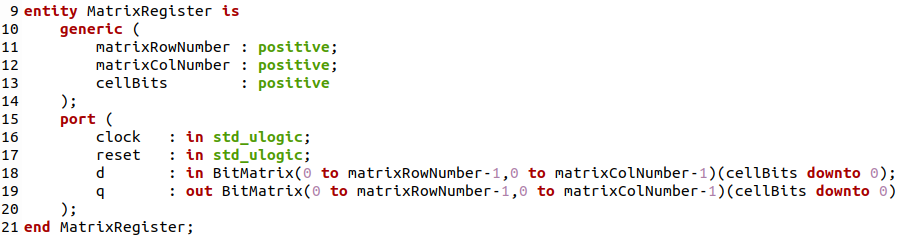
It is declared a custom type of array called BitMatrix. It is the matrix type used in all modules of the matrix multiplier.

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



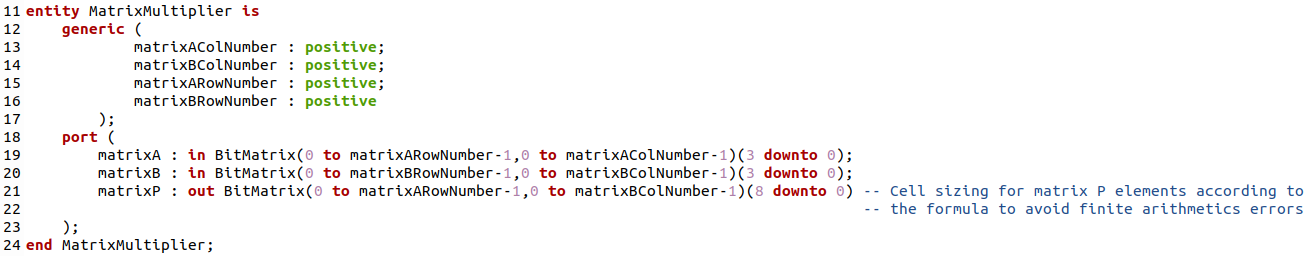
**3.2 Matrix Register**

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



**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:



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

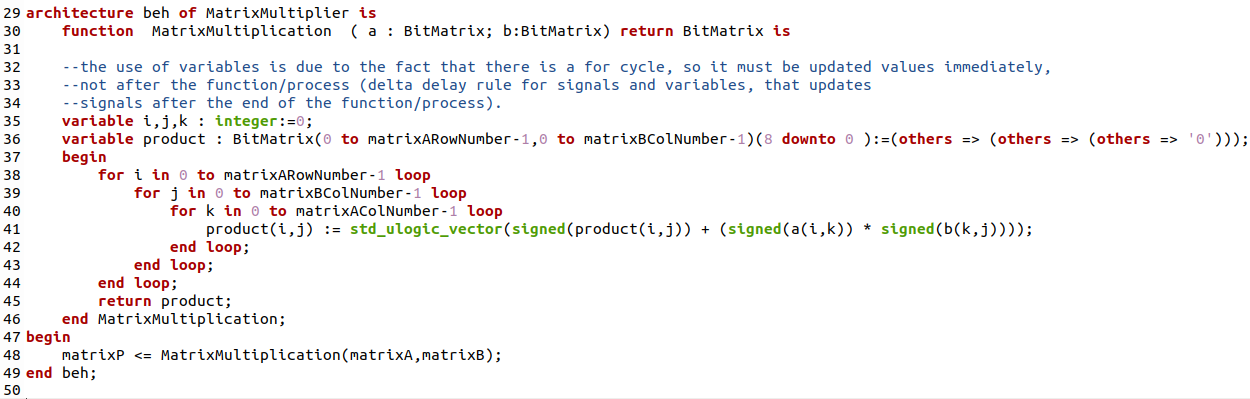
The problem is focused on sizing the matrix P elements (line 21) and the considerations on what implement were two:

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:



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:



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:

figura 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.

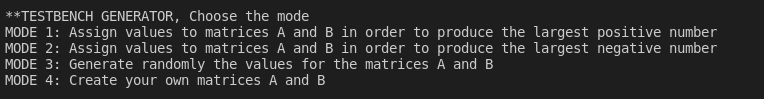
Moreover the test plan involves the simulation of a case with **random values** for the matrices A and B and another simulation with **values chosen** by the user.

**4.1 Python Testbench generator**

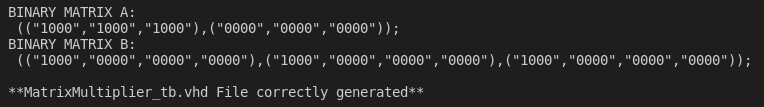
As just mentioned, this python program generates the testbench files in order to make different types of simulation.

The program let the user choose between 4 different modes:

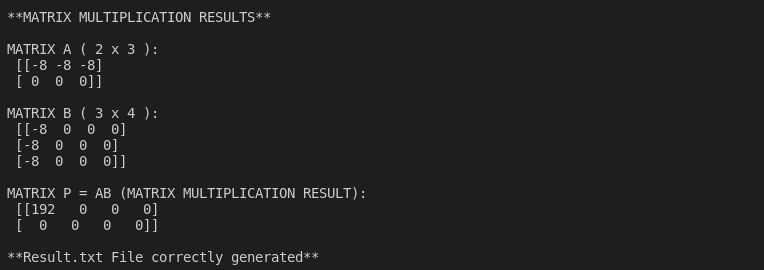
1. Assign values to matrices A and B in order to produce the largest positive number
2. Assign values to matrices A and B in order to prodcue the largest negative number
3. Generate randomly the values for matrices A and B
4. Let the user choose the values for Matrices A and B



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



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.



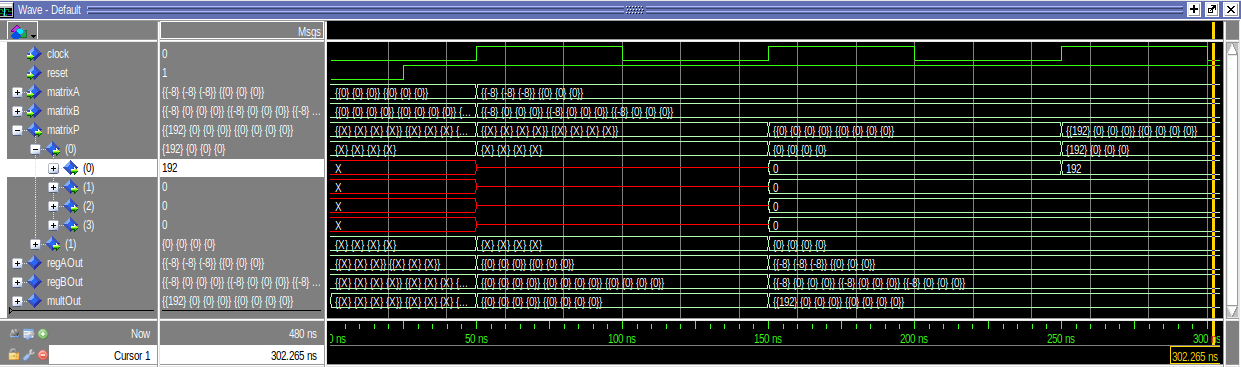
It is also important to point out that the program dynamically read the matrices A and B sizes, so it is possible to use the program to generate testbench files for matrices of any size.

**4.2 Testbench implementation**

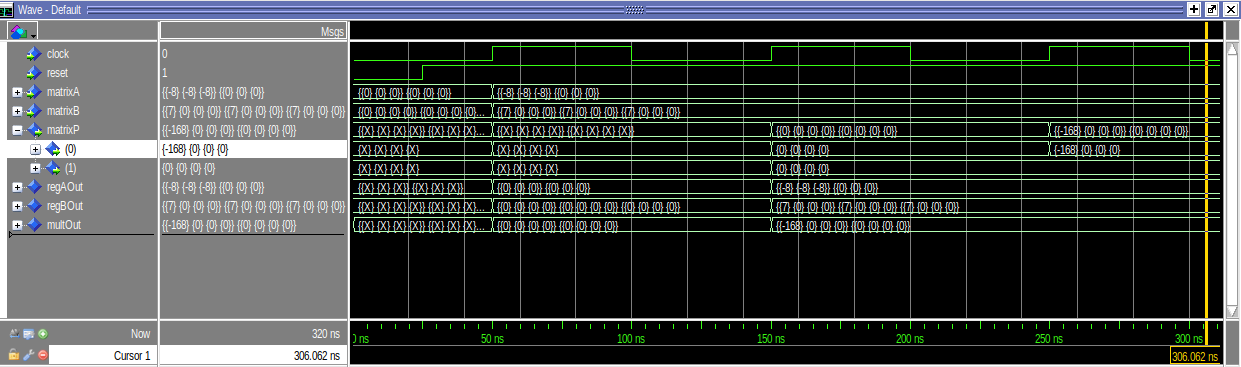
One time the MatrixMultiplier\_tb.vhd file is created it is used to run the simulation in Modelsim.

Below are shown all 4 cases just explained:

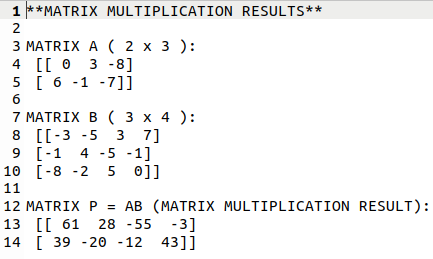
**First test:** All elements in first row of A and in first column of B are equal to (−8)10⇔(1000)2 . The result must be (192)10 in Matrix P cell*p0,0* :

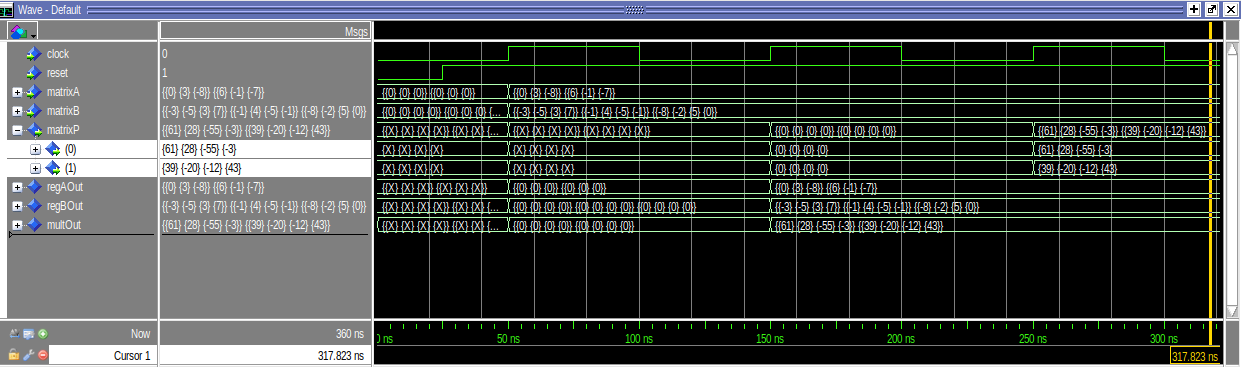


**Second test:** All elements in first row of A are equal to (−8)10 ⇔ (1000)2 . All elements in first column of B are equal to (7)10 ⇔ (0111)2 . The result must be (-168)10 in Matrix P cell*p0,0* :



**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.



****

**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.

