# Advanced Embedded Systems

# Answers for Assignment 1

# Silvia Lucia Sanna – 70/90/00053

## Task 1 – Block Design Creation

For this task I used the ready-to-use example from Vivado using Project Template “Base MicroBlaze” and using the board “Artix-7”.

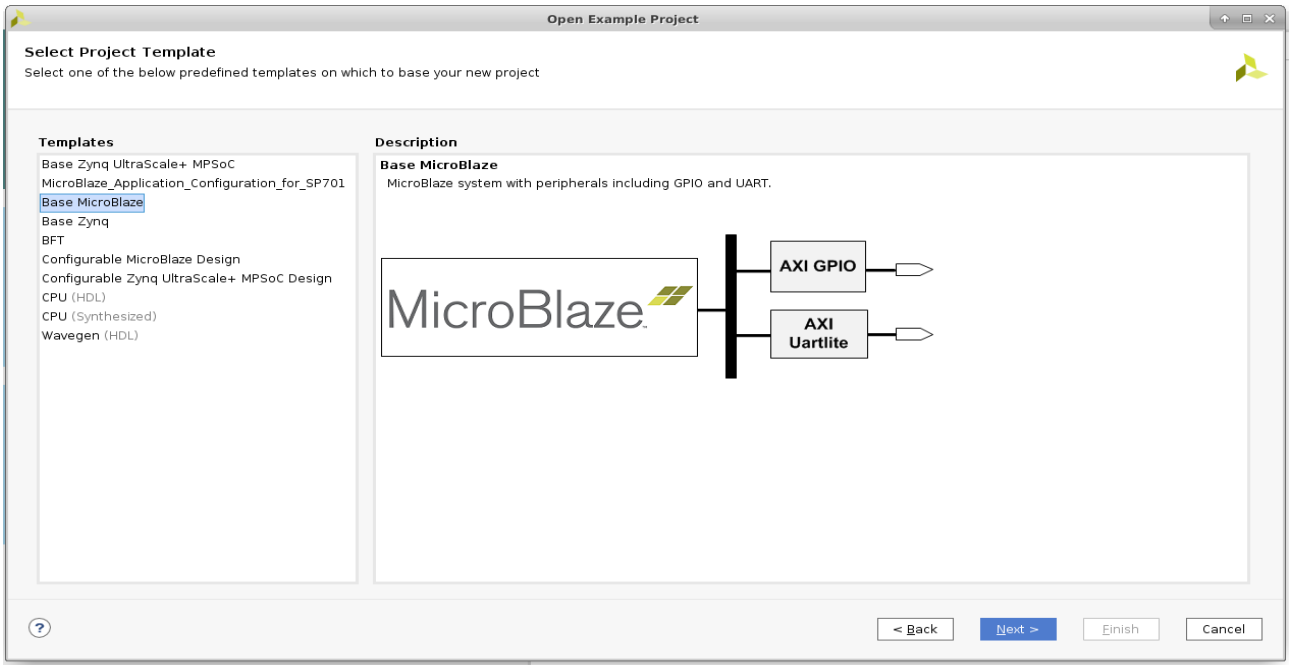



*Figure 1: project Base MicroBlaze*

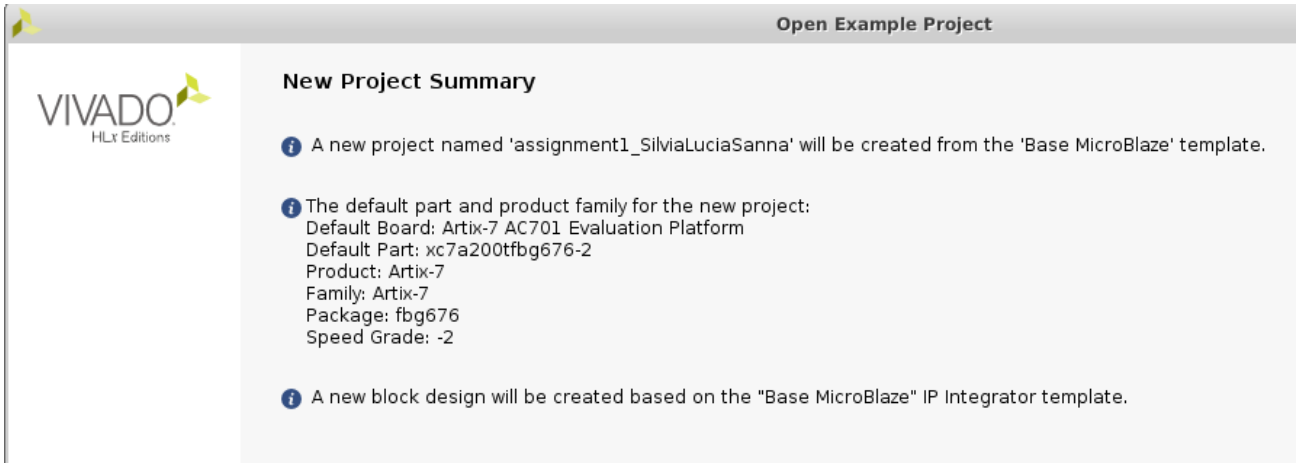



*Figure 2: project summary with board description*

After the project creation, when I opened it, I already had the project design block.

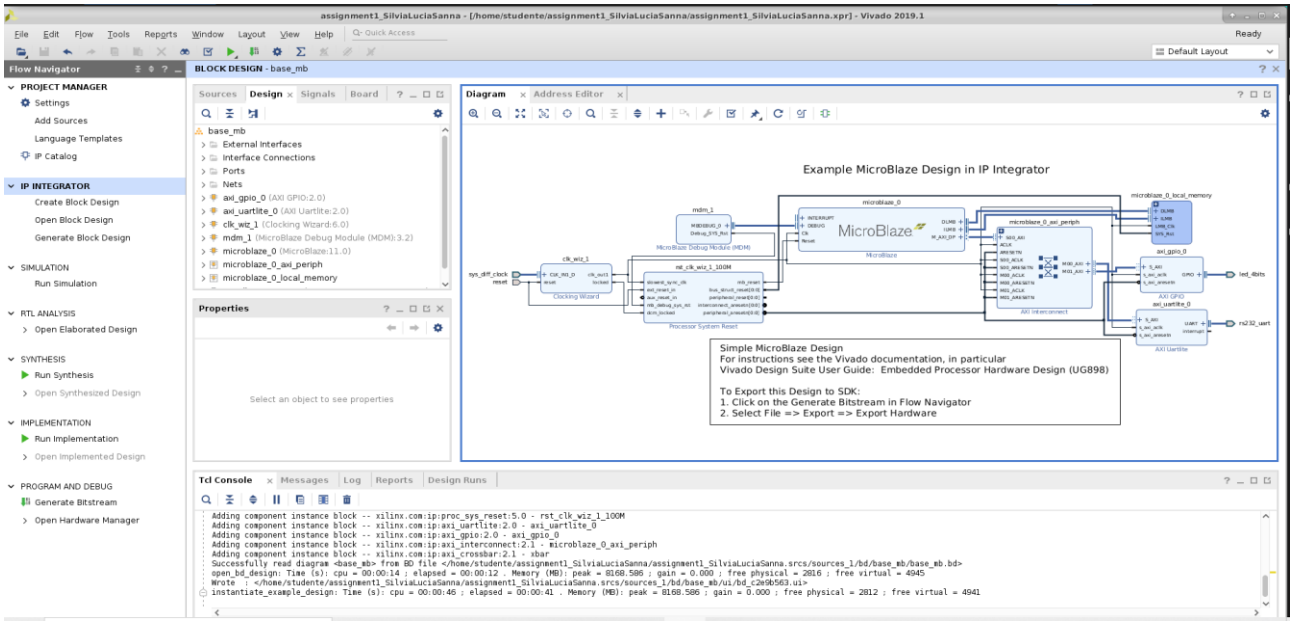



*Figure 3: block design*

By double clicking on “MicroBlaze” I opened all microprocessor options and in the “Advanced → Buses” I put a tick on “Enable Trace Bus Interface”. I could have ticked it also clicking on “Next” from first page in the options.

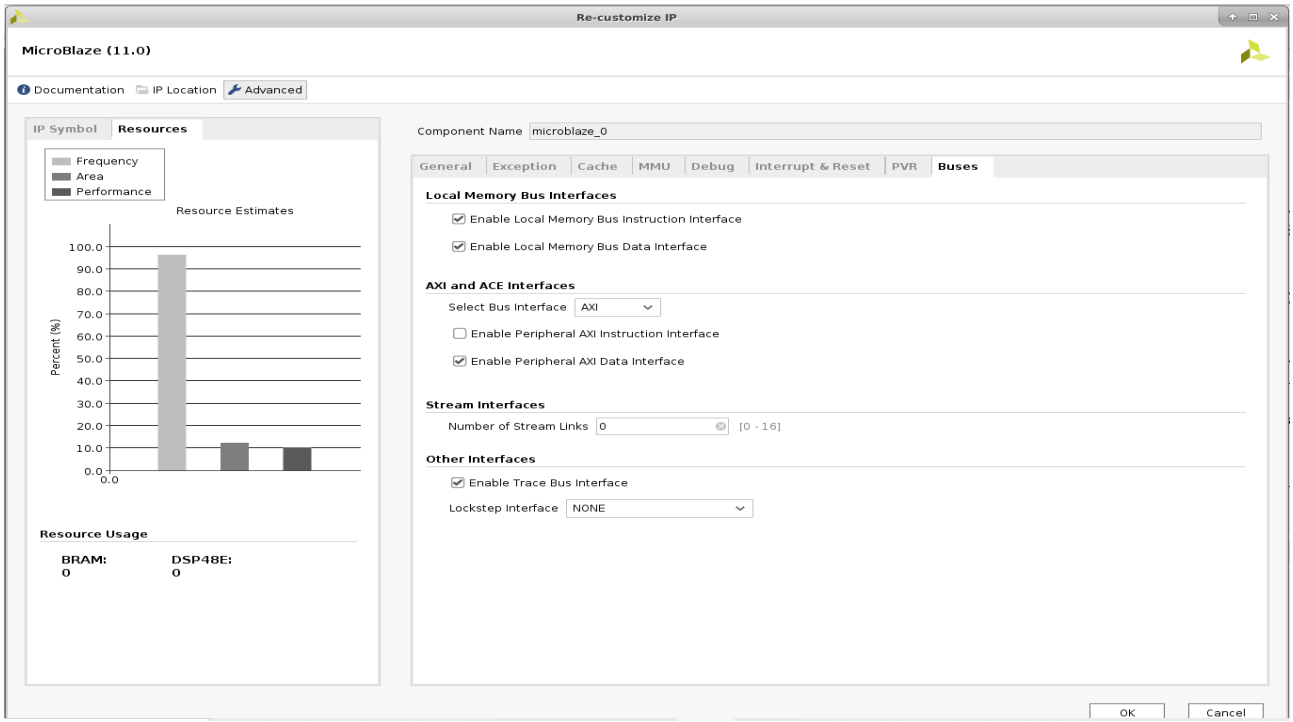



*Figure 4: Enable Trace Bus Interface*

Before starting to code I had to make the synthesis (in the left window I have a lot of different functions like “Run Implementation”, “Run Simulation”, “Run Synthesis”) and export the hardware (File → Export → Export Hardware).

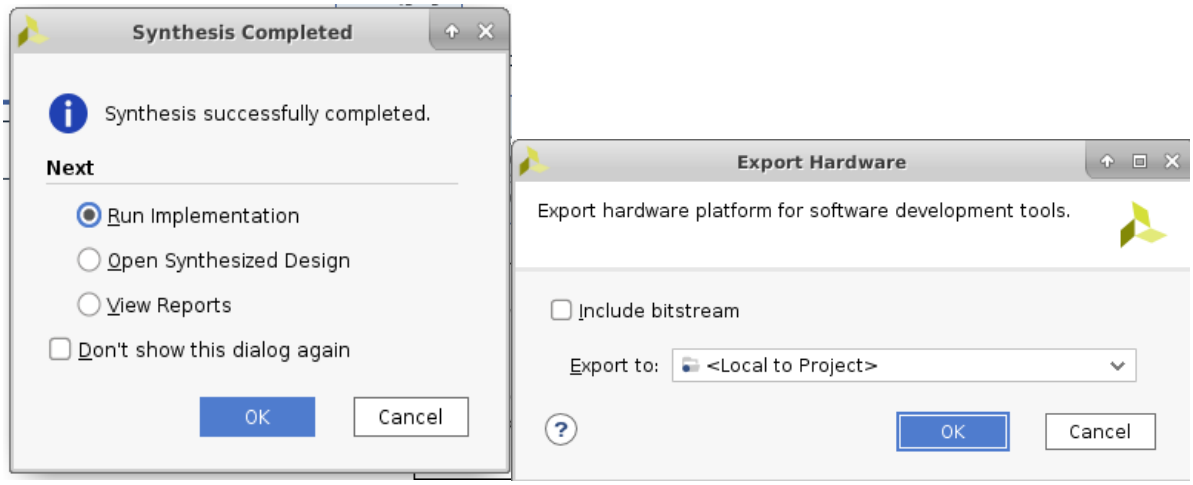



*Figure 5: Left: synthesis completed. Right: export hardware.*

## *Block Design description*

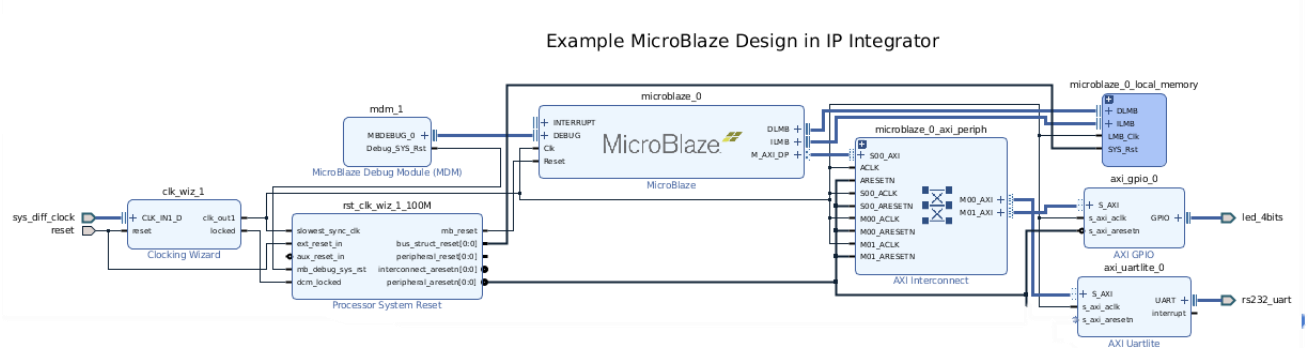



*Figure 6: block design detailed*

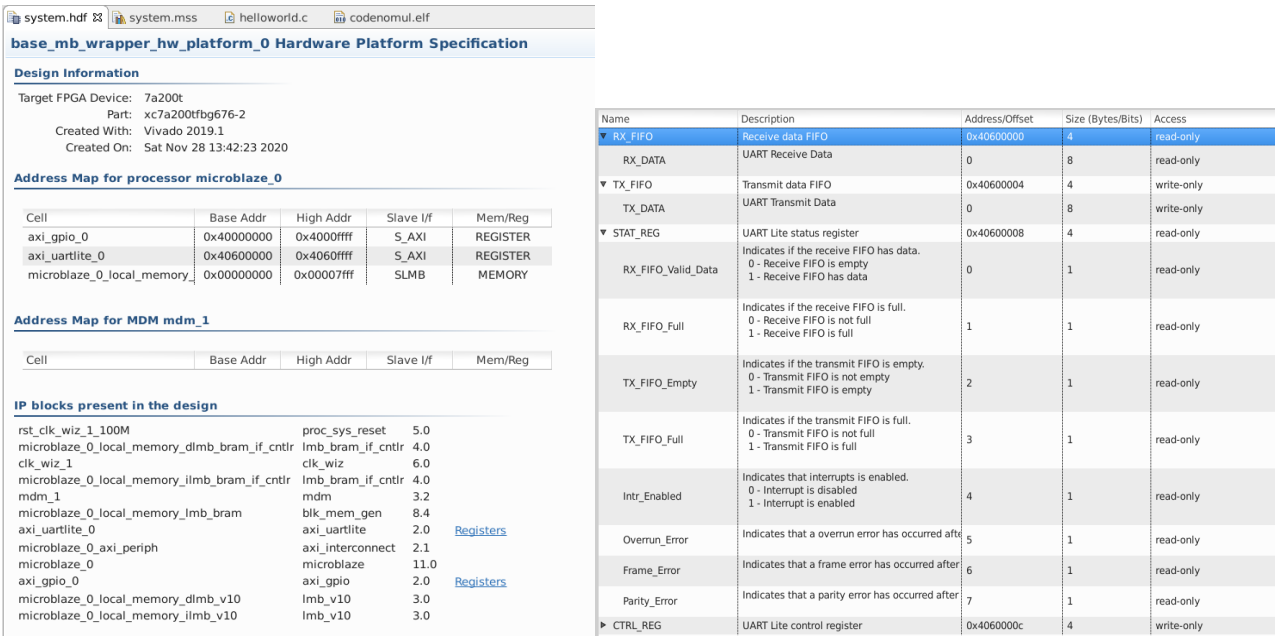
The hardware used in this project is what you find in the ready-to-use Vivado’s projects, named “Base MicroBlaze”. This project has the following hardware components:

- A “clocking wizard” made with differential clock and a reset signal.
- A “processor system reset” to send the correct signal clock and reset to the microprocessor.
- A “MicroBlaze debug module” which enables the signals to debug the microprocessor.
- The “MicroBlaze” which is our microprocessor.

- The “AXI Interconnected” which is the module performing the AXI interconnections between AXI peripherals and microblaze.
- AXI GPIO and AXI Uartlite which are peripherals respectively for “general purpose input output” and uart. This last one is very important for our project because is the peripheral through which we can communicate with the external, connect the FPGA board to the computer and make them communicate.
- A “MicroBlaze local memory” which is the memory of the microprocessor.

*Memory map*

The memory map is a very important structure in the microprocessor that contains how the memory of the peripherals is structured, so from which address to which one we find what. In our case we have from address 0x40600000 and 0x4060ffff we have the uralite. In particular if we click on Registers in “IP blocks present in the design” we can see detailed offsets of Uartlite for example or even GPIO.



system.hdf | system.mss | helloworld.c | codenomul.elf

**base_mb_wrapper_hw_platform_0 Hardware Platform Specification**

**Design Information**

Target FPGA Device: 7a200t
Part: xc7a200tfbg676-2
Created With: Vivado 2019.1
Created On: Sat Nov 28 13:42:23 2020

**Address Map for processor microblaze_0**

| Cell | Base Addr | High Addr | Slave I/f | Mem/Reg |
|---|---|---|---|---|
| axi_gpio_0 | 0x40000000 | 0x4000ffff | S_AXI | REGISTER |
| axi_uartlite_0 | 0x40600000 | 0x4060ffff | S_AXI | REGISTER |
| microblaze_0_local_memory | 0x00000000 | 0x00007fff | SLMB | MEMORY |

**Address Map for MDM mdm_1**

| Cell | Base Addr | High Addr | Slave I/f | Mem/Reg |
|---|---|---|---|---|

**IP blocks present in the design**

| | | | |
|---|---|---|---|
| rst_clk_wiz_1_100M | proc_sys_reset | 5.0 | |
| microblaze_0_local_memory_dlmb_bram_if_cntlr | lmb_bram_if_cntlr | 4.0 | |
| clk_wiz_1 | clk_wiz | 6.0 | |
| microblaze_0_local_memory_ilmb_bram_if_cntlr | lmb_bram_if_cntlr | 4.0 | |
| mdm_1 | mdm | 3.2 | |
| microblaze_0_local_memory_lmb_bram | blk_mem_gen | 8.4 | |
| axi_uartlite_0 | axi_uartlite | 2.0 | Registers |
| microblaze_0_axi_periph | axi_interconnect | 2.1 | |
| microblaze_0 | microblaze | 11.0 | |
| axi_gpio_0 | axi_gpio | 2.0 | Registers |
| microblaze_0_local_memory_dlmb_v10 | lmb_v10 | 3.0 | |
| microblaze_0_local_memory_ilmb_v10 | lmb_v10 | 3.0 | |

| Name | Description | Address/Offset | Size (Bytes/Bits) | Access |
|---|---|---|---|---|
| RX_FIFO | Receive data FIFO | 0x40600000 | 4 | read-only |
| RX_DATA | UART Receive Data | 0 | 8 | read-only |
| TX_FIFO | Transmit data FIFO | 0x40600004 | 4 | write-only |
| TX_DATA | UART Transmit Data | 0 | 8 | write-only |
| STAT_REG | UART Lite status register | 0x40600008 | 4 | read-only |
| RX_FIFO_Valid_Data | Indicates if the receive FIFO has data. 0 - Receive FIFO is empty 1 - Receive FIFO has data | 0 | 1 | read-only |
| RX_FIFO_Full | Indicates if the receive FIFO is full. 0 - Receive FIFO is not full 1 - Receive FIFO is full | 1 | 1 | read-only |
| TX_FIFO_Empty | Indicates if the transmit FIFO is empty. 0 - Transmit FIFO is not empty 1 - Transmit FIFO is empty | 2 | 1 | read-only |
| TX_FIFO_Full | Indicates if the transmit FIFO is full. 0 - Transmit FIFO is not full 1 - Transmit FIFO is full | 3 | 1 | read-only |
| Intr_Enabled | Indicates that interrupts is enabled. 0 - Interrupt is disabled 1 - Interrupt is enabled | 4 | 1 | read-only |
| Overrun_Error | Indicates that a overrun error has occurred afte | 5 | 1 | read-only |
| Frame_Error | Indicates that a frame error has occurred after | 6 | 1 | read-only |
| Parity_Error | Indicates that a parity error has occurred after | 7 | 1 | read-only |
| CTRL_REG | UART Lite control register | 0x4060000c | 4 | write-only |

*Figure 7: Left: memory map of all peripherals. Right: Uart addresses.*

## Task 2 – Code at high level (written in C using SDK) and low level (assembly code in .elf file)

To write the C code, first of all after the synthesis I had to launch the SDK. So, in Vivado I clicked on File → Launch SDK

When it opened, there already was a file called helloworld.c, I modified it with the correct lines of code I needed.

```c
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xuartlite_l.h" //library for uart

#define SIZEFROMNUMB 53 //70/90/00053
#define INPUT_SIZE 8 //my matriculation number is 53 so I do 5+3=8
#define UART_BASE_ADDRESS 0x40600000 //first memory cell in uart
#define kernelsize 3
#define ysize INPUT_SIZE - kernelsize +1

void convolution (uint64_t *ptry, uint32_t *ptrx, uint32_t *ptrw);
void receive (uint32_t *ptr, int forsize);
void send (uint64_t *ptr, int forsize);

int main()
{


    uint32_t x[INPUT_SIZE]; //integers depends on compilator: we need 32 bit int
    uint32_t w[kernelsize]; //w is the weight vector
    uint64_t y[ysize]; //output vector
    //X
    receive(x, INPUT_SIZE); //fill x with data in the .txt file received from uart
    //w
    receive(w, kernelsize); //fill w with data in the .txt file received from uart
    //make convolution
    convolution (y, x, w); //fill y making convolution between x and w
    //transmit y
    send(y, ysize); //transmit y to the uart


```

*Figure 8: main c code*

For simulation time simplicity I selected as INPUT_SIZE 8 (the sum between 5 and 3, the last two numbers of my student id), the size of the kernel was 3 and so the result of my convolution was 8-3+1 (ysize). I selected x and w made by integers of 32 bits: each row of x and w is made by 4 bytes. My y is made by integers of 64 bits (the max representation by multiplying two 32 bits integers, even if in some cases there is overflow, which for lack of time I did not take in count, deciding to ignore as windows calculator does for example). After x and w creation, I

had to fill in with the input received by the uart. After this I made the convolution (fill in the y) and transmit it to the uart.

```
36 void convolution (uint64_t *ptry, uint32_t *ptrx, uint32_t *ptrw){
37     for (int j=0; j<ysize; j++){
38         ptry[j] = (ptrx[j]*ptrw[0]) + (ptrx[j+1]*ptrw[1]) + (ptrx[j+2]*ptrw[2]);
39     }
40 }
41
42 void receive (uint32_t *ptrarray, int forsize){ //this is only to send one byte;
43     //if I want to send more bytes an internal for is needed with also a shift in the ith element each byte
44     for (int i=0; i<forsize; i++){
45         for (int j=0; j<4; j++){ //x and w are 32 bit, uart reads 8bit each time
46             ptrarray[i] += (XUartLite_RecvByte(UART_BASE_ADDRESS)) << (j*8);
47             //mask because if we take a var with more than 8 bit I filter it
48         }
49
50     }
51 }
52
53 void send (uint64_t *ptrarray, int forsize) {
54     for (int i=0; i<forsize; i++){
55         for (int j=0; j<8; j++) { //y is 64bit and uart can accept only 8 bit each time
56             uint8_t data = ptrarray[i] >> (j*8);
57             XUartLite_SendByte(UART_BASE_ADDRESS, data);
58         }
59     }
60 }
```

*Figure 9: from up to down: function convolution (lines 36-40), function receive (lines 42-51), function send (lines 53-60)*

I implemented the convolution making a simple sum between the multiplication with each element of x between each element of w and shifting the w on x length by on element each time. I fill the y with the result of each iterations.

For the function receive I made a cycle during the length of the array I had to fill in when the reception was on. Each element of my x and w was made by 32 bits but the uart can only work with 8 bits at a time: so I made another cycle for each element, to send one byte at a time. To not lose the first received values, I made a shift of 8 bits at each iteration. To make the receiving I used the ready-to-use function XUartLite_RecvByte which checks if the data is valid (if FIFO is not empty, so if RX_FIFO_Valid_Data is set to 0) and when the data is valid it receive it (as we can later see on the simulation wave forms).

In the send function I had to cycle in y length, but my y was made by 64 bits and again uart can work only with 8 bits each time: I made an internal cycle for the 8 bytes of each y element. Here again, to not lose values I made a shift to right to send the correct byte at a time. To make the trasmission I used the

ready-to-use function XUartLite_SendByte which checks if the data is valid (if FIFO is not full, so if TX_FIFO_Full is set to 0) and when the data is valid it transmits it (as we can later see on the simulation wave forms).

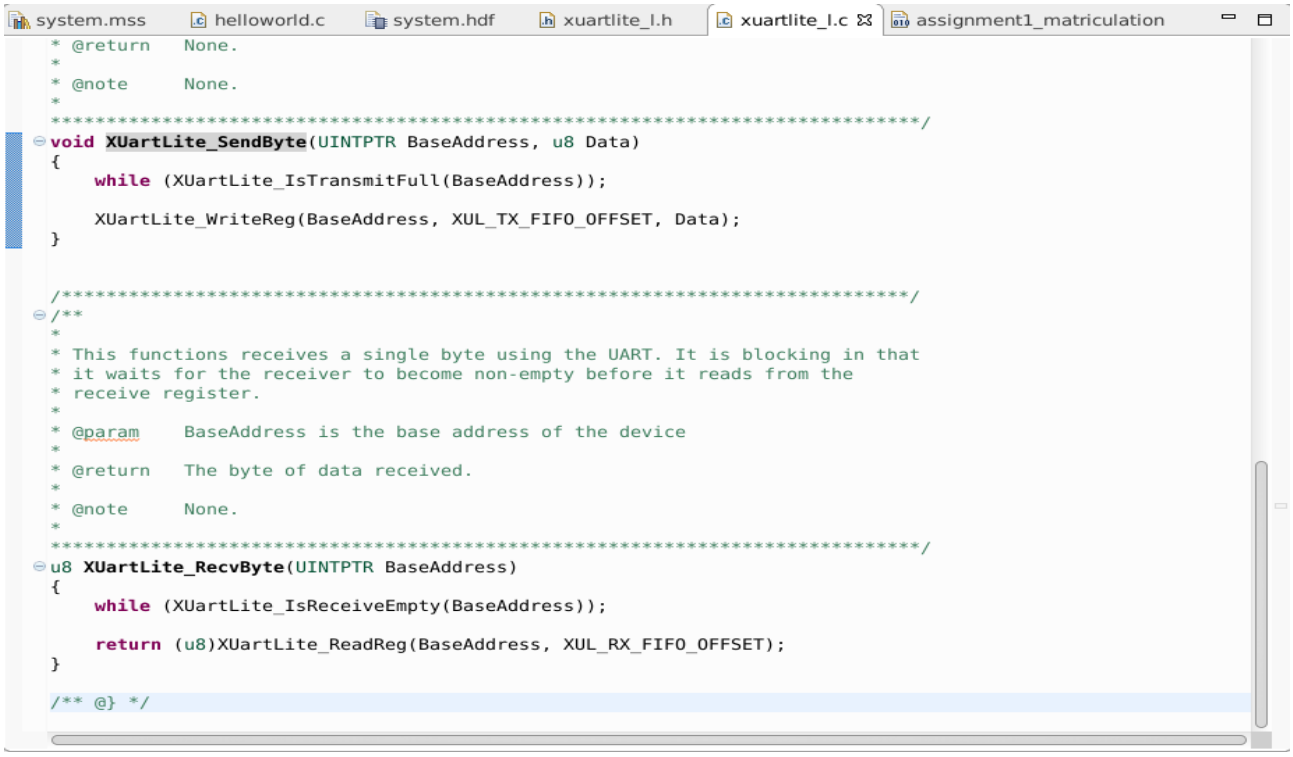



*Figure 10: How functions XUartLite_SendByte and XUartLite_RecvByte are implemented*

Assembly:

I attached in my project my assembly commented line by line, named in file “assembly_nomulSanna.txt”.

I have noticed that my assembly is composed by 20 instructions repeated appropriately. I briefly describe here how they work as in the comments sometimes I only wrote the meaning for that case.

- Arithmetic operations:
  - addk: makes the arithmetic form, is in the form addk rD,rA,rB so makes the sum between content of registers A and B and stores in register D.
  - addi: add immediate is in the form addi rD,rA,IMM so the sum of the content of register rA with IMM is stored in register rD.
  - addik: is the same of addi but also bit K is set and so the carry flag will keep its previous value despite the current addition. So we can say this is a no-operation.
  - cmp: comparison operation between two register, is in the form cmp rD,rA,rB so the content of register rA is subtracted from content of register rB and stored in rD. This is a signed operation and rD shows the relation between register A and B. In my assembly I find this instruction everytime before closing the for cycle (is one of the last two instructions).
- Load:
  - imm: loads IMM value into a temporary register and locks this value to be used by following instruction. In my assembly this instruction is always before instruction brlid **XX**
  - lbui: immediately loads a byte unsigned from memory address obtained by the sum of content rA with IMM and stored in least significant byte of rD. In my assembly I found this function at the end of function “send” to store the value to be sent, in fact is in the function XUartLite_SendByte.

    - lwi: is the same of lbui but loads 8 bytes (a word). In my assembly I find this instruction everytime I have to reload something from the memory above all at the end of the functions or in the for cycles.
- Store:
    - sbi: stores byte immediate, is in the form sbi rD,rA,IMM so stores the content of least significant byte of register D in the memory location obtained by adding content of register A to IMM. This instruction is in my assembly only once in send function before the call to XUartLite_SendByte function.
    - swi: is the same of sbi but instead of working with byte works with word (8 bytes = 1 word). In my assembly I find this instruction everytime I want to store something above all in the initialization of arrays x, w, y.
- Logical operations:
    - or: logical or, is in the form or rD,rA,rB so it makes the or between content of register A and content of register B and stores it in register D. In the assembly I have found this operation most of the time as a no-operation: makes the or between 0 and 0 storing in 0.
    - andi: logical and with immediate, is placed in the form andi rD,rA,IMM so content of rA is anded with IMM and stored in rD. I find this in my assembly only in functions send and receive when I make respectively the right shift and left shift.
    - xori: logical XOR immediate, is in the form xori rD,rA,IMM and so content of register A is xored with value IMM and stored in register D. I only find this instruction once in my assembly in the send function during right shifting.
    - srl: shift right logical, is in the form srl rD,rA so shifts logically the content or rA one bit to the right and stores the result on rD.
- Branch: branches to the instruction located at address IMM there are different forms of branches and in my assembly I have found:

- bri: branch immediately to instruction set in location IMM.
- brlid: branch and link immediate with delay. The L bit is valid, this instruction will link automatically. Above all there is set also bit D standing that there is a delay slot, in fact this instruction is followed by a no-operation instruction.
- bgei: branches immediately if the register is greater or equal to 0, to instruction located at address IMM.
- blti: branch immediately if register is less than 0, to instruction located at address IMM.
- bequi: branch immediately if the register is equal to 0, to instruction located at address IMM.
- beqid: is the same of bequi but there is added a delay time, so bit D is set and is followed by an instruction nop.
- bneid: branch immediately if register is not equal to 0 to instruction IMM.

## Task 3 – Run Simulation and see functions time

Before doing the simulation, I had to write a .txt file which was called on the testbench to receive values from the uart. The testbench is very important in this case because we cannot physically connect the uart to our PC and send and receive data, so the testbench emulates this process. The .txt file containes exactly all inputs needed but stored in a specific way. From the function $readmemh at the end of testbench, I had to create the .txt file in a specific way. This means that the file had to contain in each row 1 byte made by hex numbers: readmemh works with 1 byte at a time made by hex characters. The file was composed with two hex numbers in each row: I produced them randomly with a little Python script.

```
hex = [0,1,2,3,4,5,6,7,8,9,"A","B","C", "D","E", 'F']
print(hex)
import random
dimbytes = 80
txt = []
for i in range(dimbytes):
    row = str(random.choice(hex)) + str(random.choice(hex)) + '\n'
    with open ("txttb.txt", 'a') as f:
        f.write(row)
```

*Figure 11: Python script to produce my input.txt file*

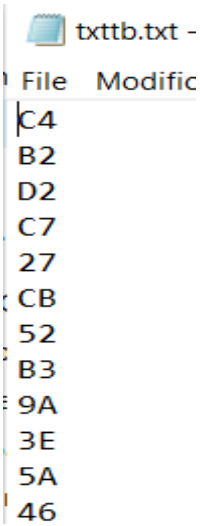



*Figure 12: input.txt file*

And here you can also see a short preview of 3 bytes in my .txt file. The way of filling this is so important: we could choose Little Endian or Big Endian.

To run the Simulation, I first saved and built the code. This process creates me a .elf file in the Debug directory where I am working. Then I have to associate this .elf to Vivado: Sources → Simulation Sources → delete .elf file. Go to Tools → Associate Elf → select Simulation Sources, sim_1, click on 3 points → browse until find the correct .elf file.

After associating the file I had to put the correct testbench that you gave us. So I went to Sources → Simulation Sources → sim_1 → _tb file and copy the correct testbench. On this one I had to change the path for

the .txt input file and also I changed time needed to receive data from UART (from 300 to 100).

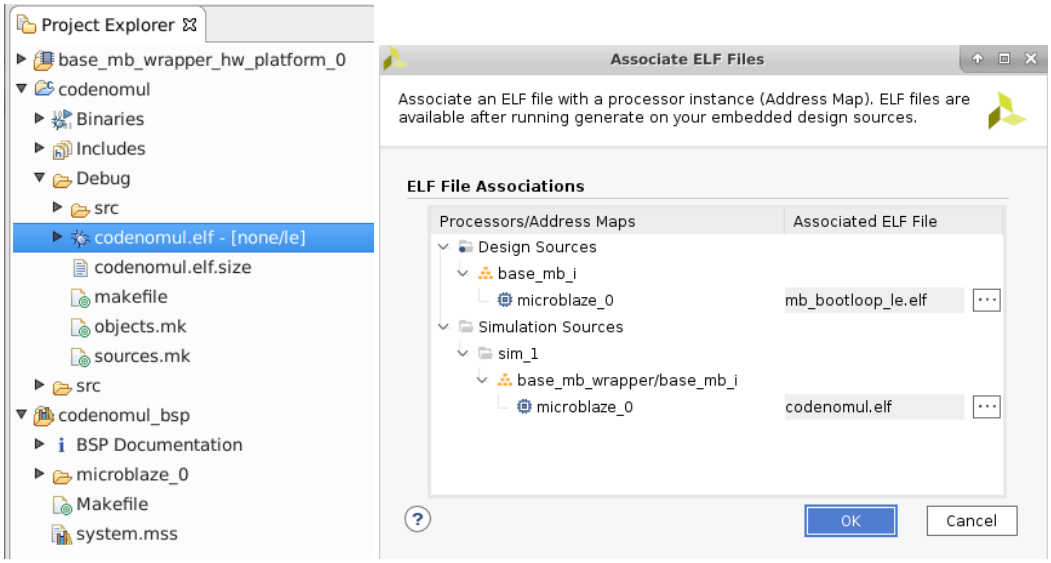



*Figure 13: Left: .elf file. Right: Associate file.*

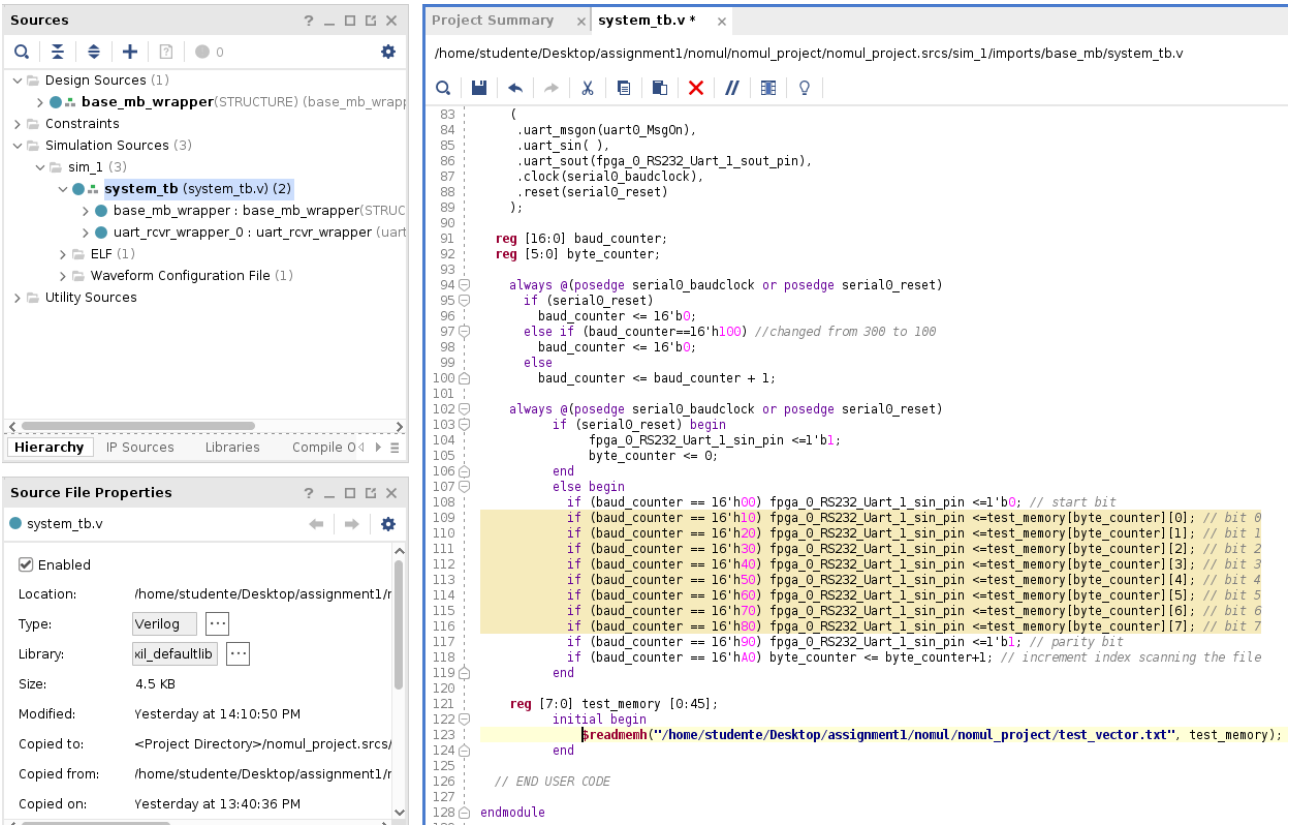



*Figure 14: testbench: where to find it and changed lines (line 97, line 123)*

After making this I can come back to Vivado and Run the simulation. In the left windows I have different options: make synthesis, make implementation, run simulation.

I select run simulation, run behavioural simulation and wait some minutes the end of the simulation.

But first, when I start, the simulation will be done for some nanoseconds, so in the upper windows I can select how many microseconds I would like to make the simulation run. I selected 20 ms. I do not know why but after more or less 8ms it was blocked, so I reclicked run and fortunately it continued.

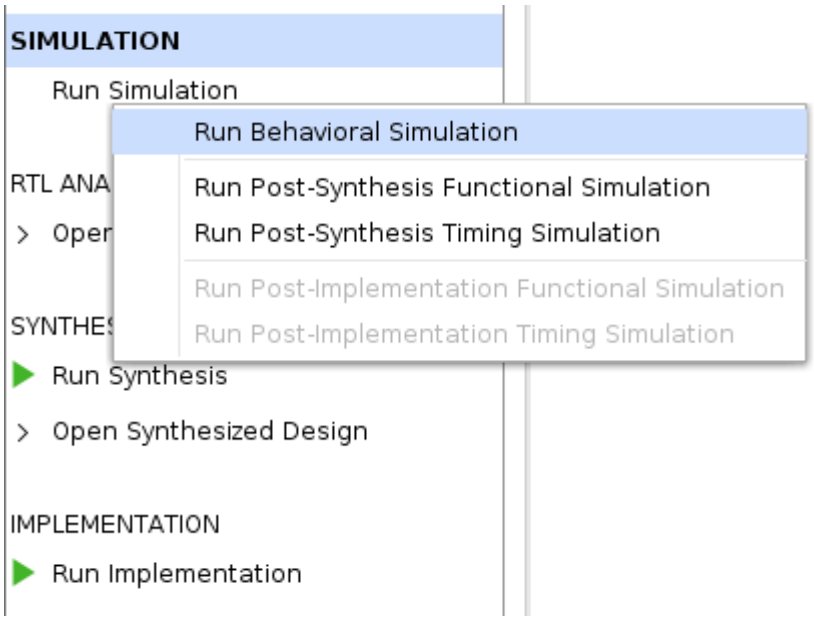



*Figure 15: Run simulation - Run Behavioral Simulation*

When the simulation finished, I had to watch how many seconds did each function take. So, I came back to the SDK and in the .elf file I searched for the name of each function. I note down the first and last address of each function. Then, in the simulation on Vivado, I selected the wire "Instr_Addr" where there was the value of the instruction currently running in that moment. I right clicked on that wire, made "Find value" and inserted the value of each function at the start and at the end. Unfortunately some functions had very very near start/end points (the end point of the previous one was very near to the start point of the current one), so even if I made the marker for each start and end, some of them were too near and Vivado like

“deleted” them, but I took notes on the times and even screenshot of each function duration.

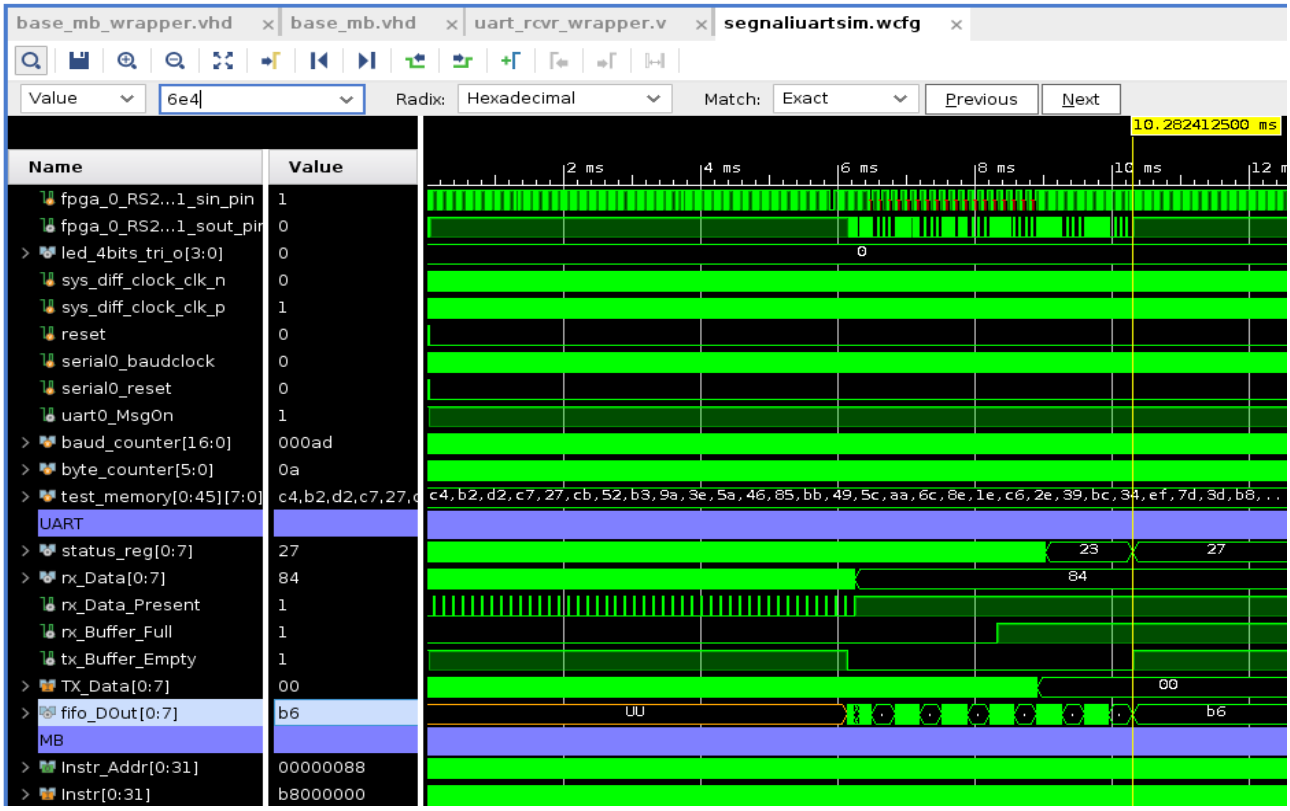



*Figure 16: total simulation time*

How to understand this green graph? On signal rx_Data_Present I see when the data is received: this signal becomes high for the all duration of the received byte which is present on signal rx_Data[0:7]. Instead signal tx_Buffer_Empty becomes low for the duration of the all the transmit time and function. The sent byte is stored in signal fifo_DOut[0:7]. In signal Instr_Addr[0:31] there is the instruction address of the currently executing instruction. In test_memory there is stored all my test_vector.txt file, the one used to receive bytes for fill in x and w arrays.

| Function name | Start time | End time | Duration | Cycles |
|---|---|---|---|---|
| main() | 3,3025 | 8900,6225 | 8897,32 | 889732 |
| receive()<br>one to recv X and one for w | 3,5525<br>4417,1425 | 4417,2325<br>6090,5225 | 4413,68<br>1673,38 | 441368<br>167338 |
| convolution() | 6090,6425 | 6135,7725 | 45,13 | 4513 |
| send() | 4417,1525 | 8900,5425 | 4483,39 | 448339 |

*Figure 17: table with start, end, duration time and cycles for each function. Values are expressed in microseconds.*

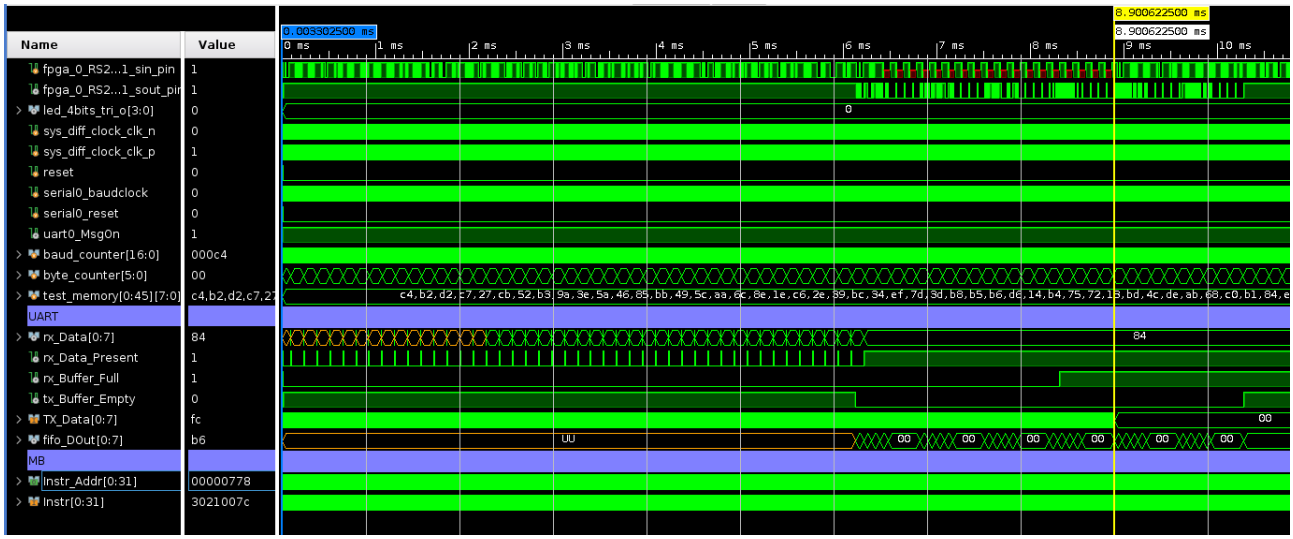



*Figure 18: main start and end*

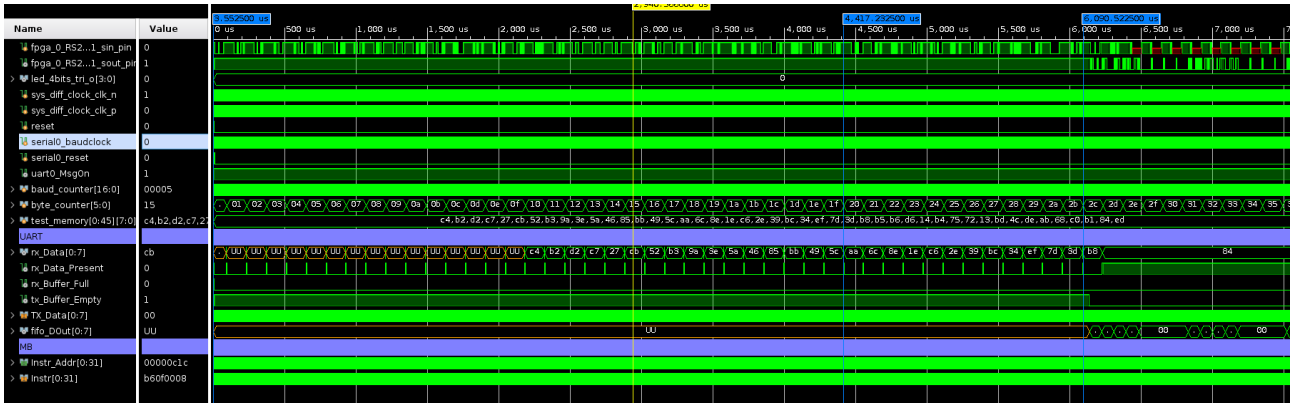



*Figure 19: receive function start and end*

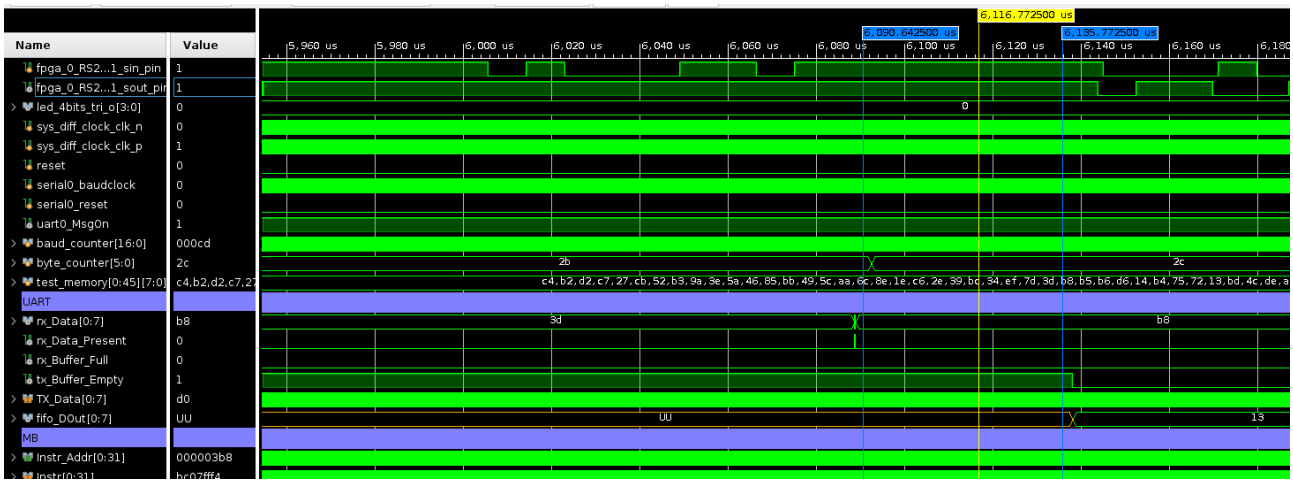



*Figure 20: convolution function start and end*

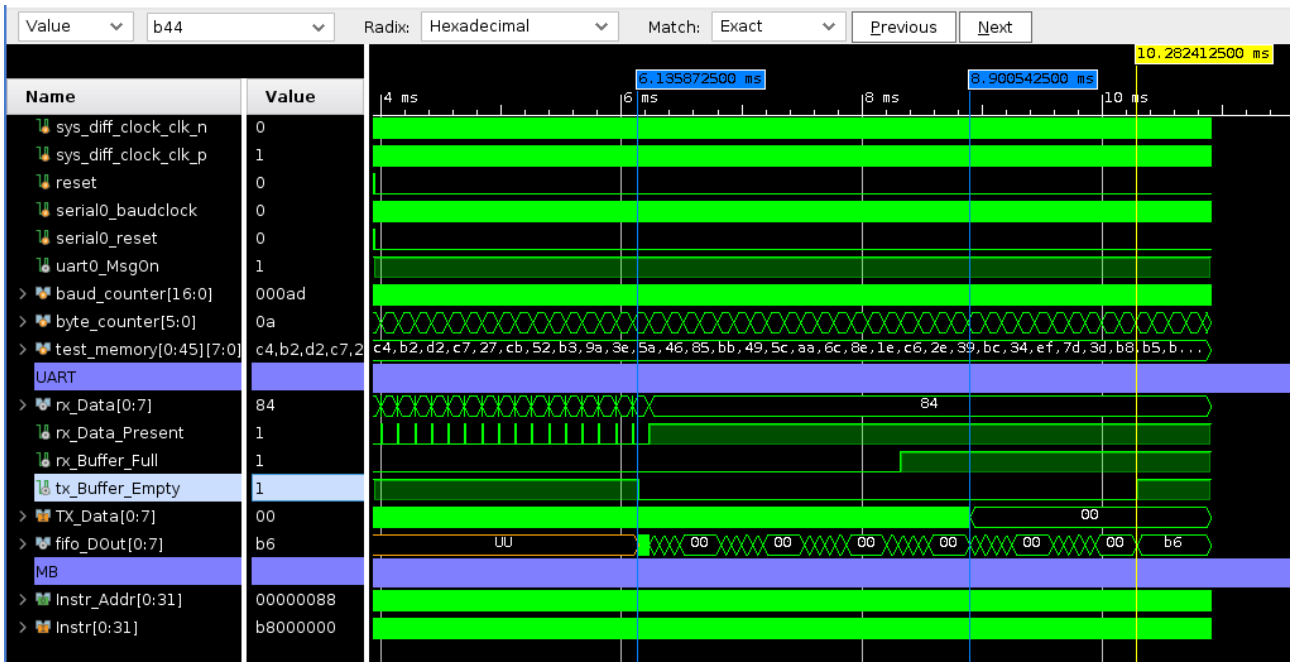



*Figure 21: transmit function start and end*

About the send function, I had a doubt why does the main and also send function end before the tx_Buffer_Empty comes up. Effectively main and send end when the last one y value should be sent. So, they end at the last external for cycle on send function, this because the last y value is sent to the uart but the transmission time is set by uart, so everything is working well.

After this my question is: are my send and receive functions performing well? So, I decided to zoom in in the first byte sent to see if it was really “C4” as it should (it is the first byte in my .txt file). It was, as you can see in the following picture. The uart receives bytes in this way but stores them in little endian as I specify later.

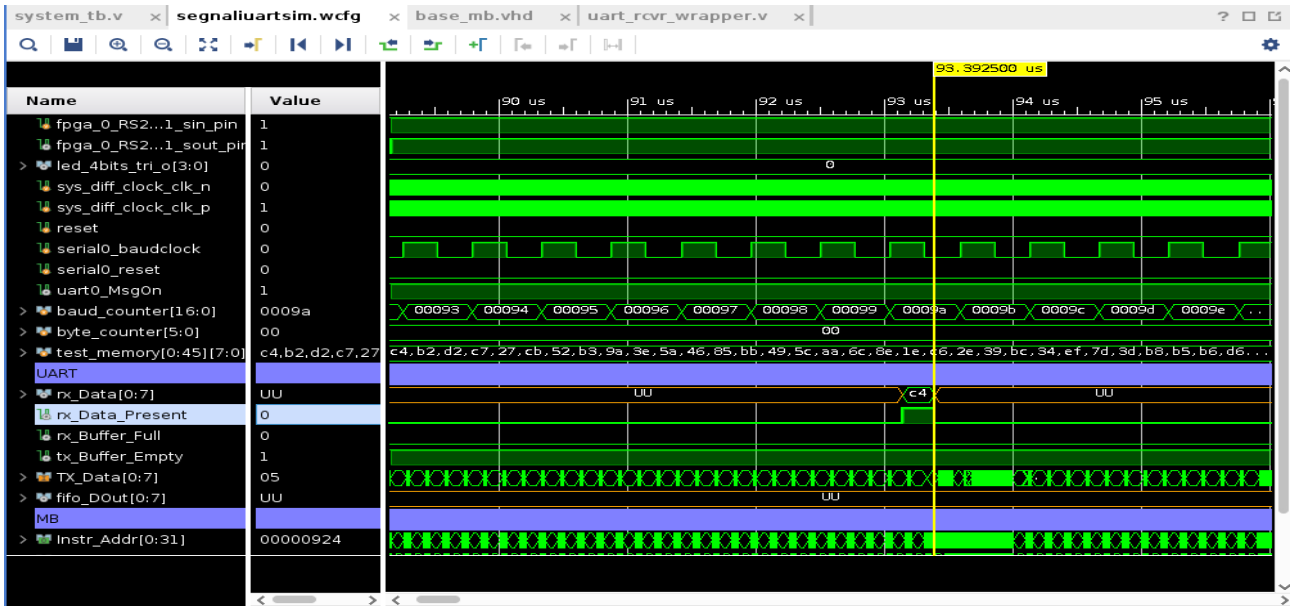



*Figure 22: send first byte*

I did the same for the transmit function: I zoomed in in the first byte sent to see if it really was the right one. How to know this? I made a simple proof by making the convolution function between the first 8 numbers of my .txt file (for x array) and the following 3 numbers of my .txt file (for w array). My first byte should be “13” and it was.

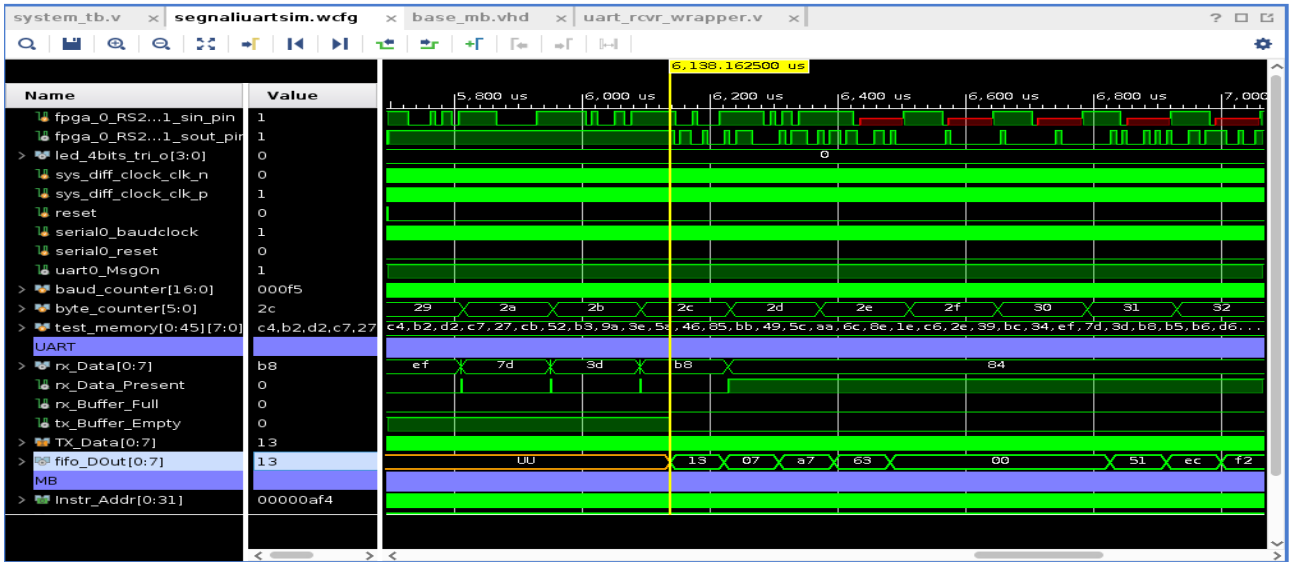



*Figure 23: first transmit bytes*

To prove that my convolution was right and so that the correct first byte sent was right, I checked my program.

```c
int main()
{

    unsigned long x [8] = {0xC7D2B2C4, 0xB352CB27, 0x465A3E9A, 0x5C49BB85, 0x1E8E6CAA, 0xBC392EC6, 0x3D7DEF34, 0xD6B6B5B8};
    unsigned long w [3] = {0x7275B414, 0xDE4CBD13, 0xB1C068AB};
    int ysize = 8-3+1; //x len - w len +1
    unsigned long long y[ysize];
    for (int j=0; j<ysize; j++){
        y[j] = (x[j]*w[0]) + (x[j+1]*w[1]) + (x[j+2]*w[2]);
        printf("0x%x \n", y[j]);}
    return 0;
}
```

input

```
main.c:20:20: warning: format '%x' expects argument of type 'unsigned int', but argument 2 has type 'long long unsigned int' [-Wformat=]
0x63a70713
0x9cf2ec51
0xbca0ed75
0x45b6b44
0x431493b6
0x4319263c
```

*Figure 24: proof right convolution*

output.txt -

File Modifica

```
13
07
a7
63
00
00
00
00
51
ec
f2
9c
00
00
00
00
```

The send and the receive stores in a way that is called little endian: the storing starts from the least significant bit (the one on the right) and ends in the most significant bit (the one on the left). For this reason, in my .txt file the number “C4B2D2C7” is stored in the x as “C7D2B2C4” (even if received as “C4B2D2C7”) and in the output file the number “000063a70713” is transmitted and stored as “1307a76300000”.

I find this file on the directory of my project, folder project.sim → sim_1 → behave → xsim → .out file (is the only one).

*Figure 25: output.txt file with y values*

## Task 4 – Add MUL64 and rerun simulation, recompute times

Making a double click on the microblaze in the block design, I can modify and see some microprocessor options. In advanced options, “General” tab, I had to select MUL64.

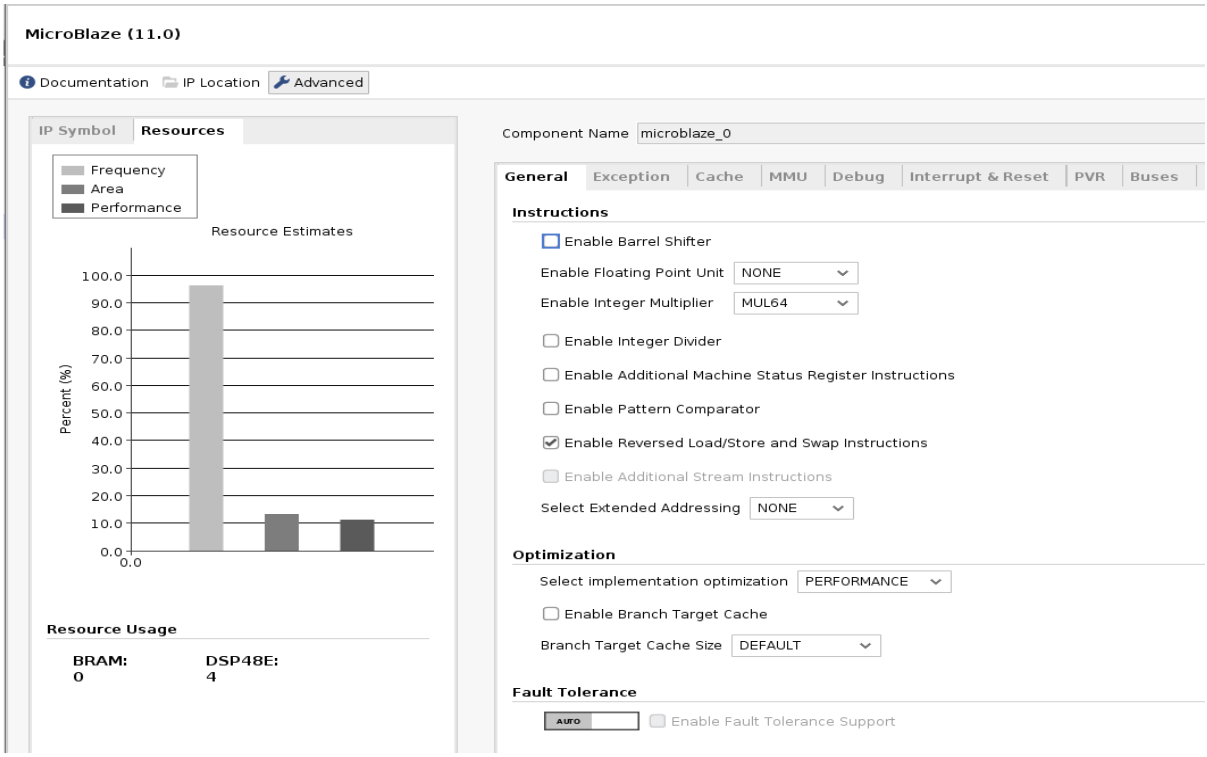



*Figure 26: mul64 option on microblaze*

I also changed clock options to make it work at 10%

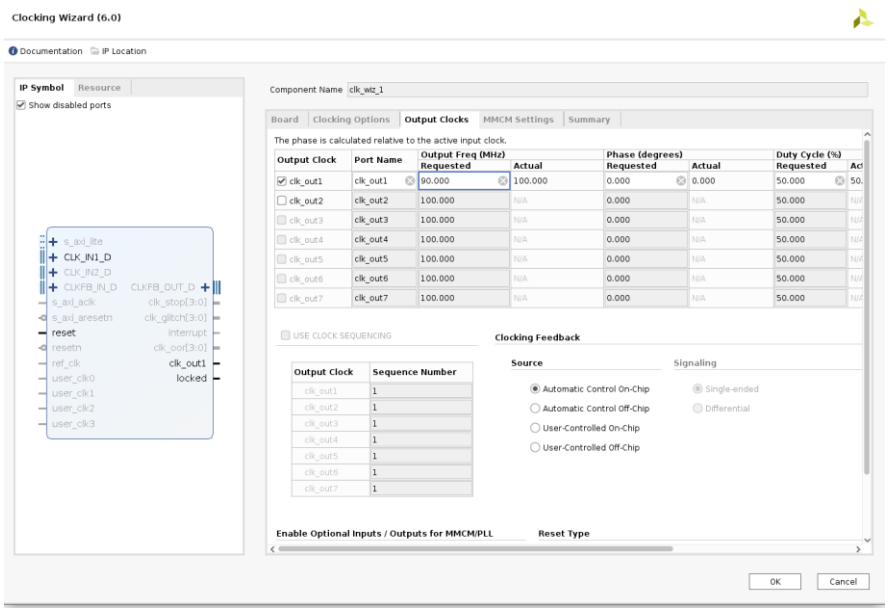



*Figure 27: change clock frequency*

After that, as I changed something on the hardware, I had to re synthetize it then re-export hardware. When it finished, I opened SDK, re-saved and re-build the code. I associate another .elf file (in Vivado: Tools → Associate ELF File) and made another simulation.

I do not know why but my times did not changed. So, I watched at the assembly and noticed that even the registers for each function did not changed. I decided to search for "muli" instruction (this must compare as I selected mul64 and must be in the convolution so that there are not so many addk) but there was no trace.

I thought that maybe I made a mistake in the hardware, so I tried to delete the sdk directory in the project and re-synthetize, re-export, re-code, re-save, re-build. No trace of "muli" even here.

Maybe this is not the right solution, so I decided to made another project starting with the correct hardware. So, I did again points 1-2: I launched Vivado → Open Example Project → Base MicroBlaze → put a name to the project → select Artix-7 as board

Then I had to change the microBlaze configurations: double click on microblaze, Enable Trace Interface on advanced options buses, Mul64 in Enable Integer Multipler in General options. After that I also changed clock settings by double clicking on that IP.

After that, as I changed the hardware, I had to remake synthesis and re-export hardware. To make the code I had to launch SDK: File → New → Application Project → choose name. Then on the left I had the project name and in src I found the "helloworld.c" file that I had to change with the correct code which you can see at point 2. I saved and built the code so then in nameproject → Debug I found the "nameproject.elf" which is assembly code. In this code I searched for instruction "muli" which was present in the convolution function instead of a lot of addi or addik. So now the multiplier option worked.

Now I came back to Vivado and in the Sources on the left I deleted from Sources → Simulation Sources → sim1 → ELF →

“executable.elf”. I had to associate the correct one: Tools → Associate Elf, second project → browse in the directories and when found clicked on ok.

Before making the simulation I also had to put the correct testbench to emulate the behaviour between the board and the user. So, in Sources on left side → Simulation Sources → Sim1 → file_tb.v modify with the correct one. I also had to add the .txt file in the correct path.

To see the right signals in the waveform I also had to browse on them.

Now I simulated and I noticed that the main lasted a little bit less (more or less 0.020427778 ms). This because of very less duration of convolution function (it is much shorter) but also a less duration in functions send and receive. This is due to a replacement with a lot of addi and addik instructions with one muli instruction: so for different addition operations we can just use one multiplication operation (as we know from math concepts) which is faster.

| Function name | Start time | End time | Duration | Cycle |
|---|---|---|---|---|
| main() | 5,650277 | 8880,194722 | 8874,54 | 798709 |
| receive()<br>one to recv X and one for w | 5,928055<br>4418,283611 | 4418,183611<br>6091,261388 | 4412,25551<br>1672,977 | 397103<br>150568 |
| convolution() | 6091,394722 | 6097,0725 | 5,677778 | 511 |
| send() | 6091,2725 | 8880,105833 | 2788,8333 | 250995 |

*Figure 28: table with start, end, duration time and cycles for each function with mul64 configuration. Expressed in microseconds.*

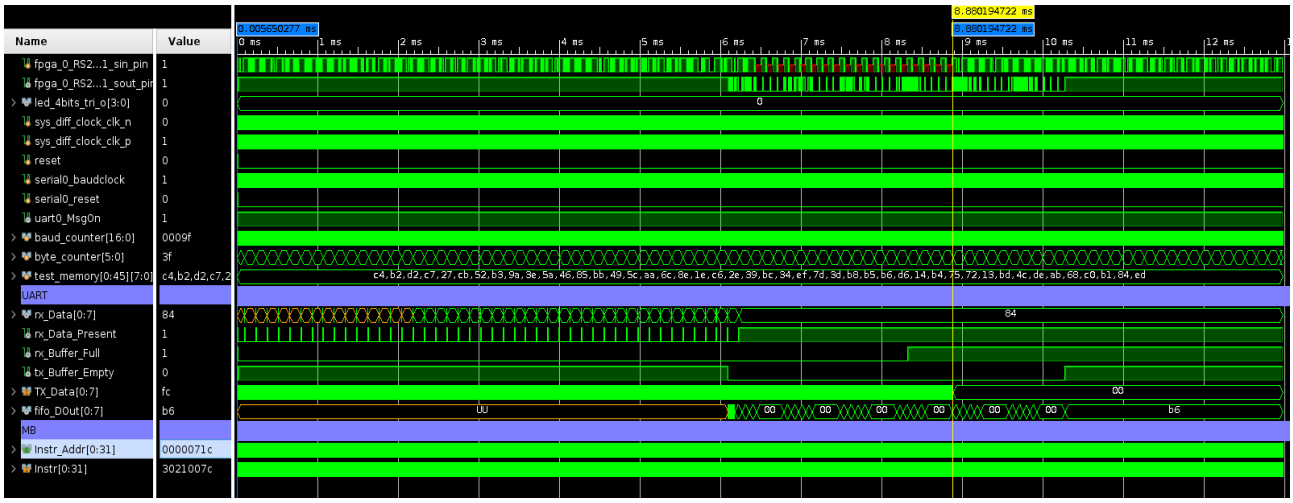



*Figure 29: main duration in mul64*

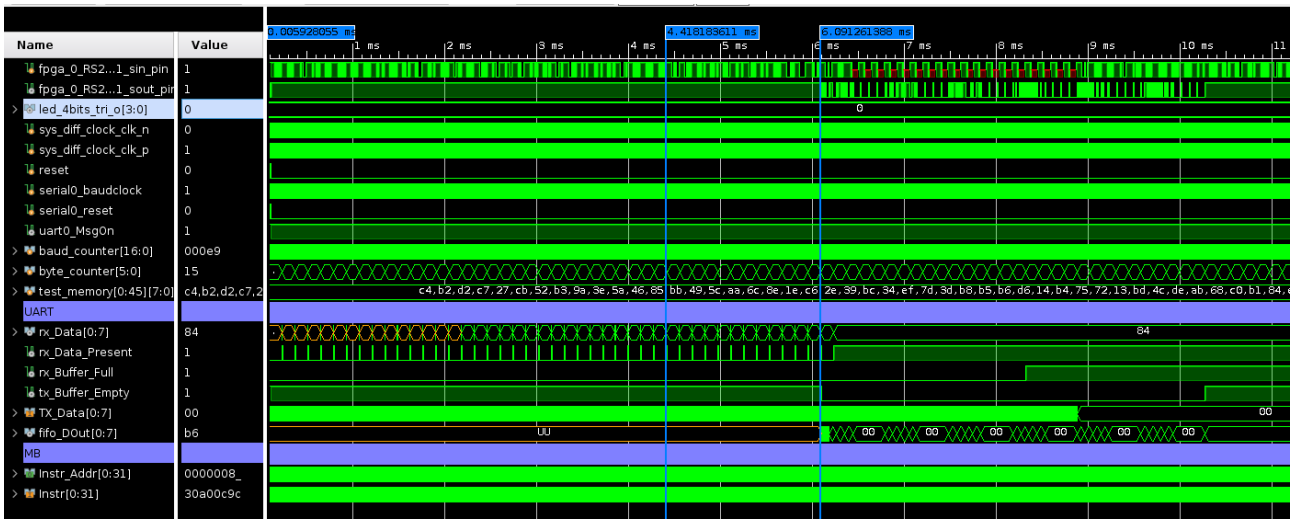



*Figure 30: receive function duration with mul64*

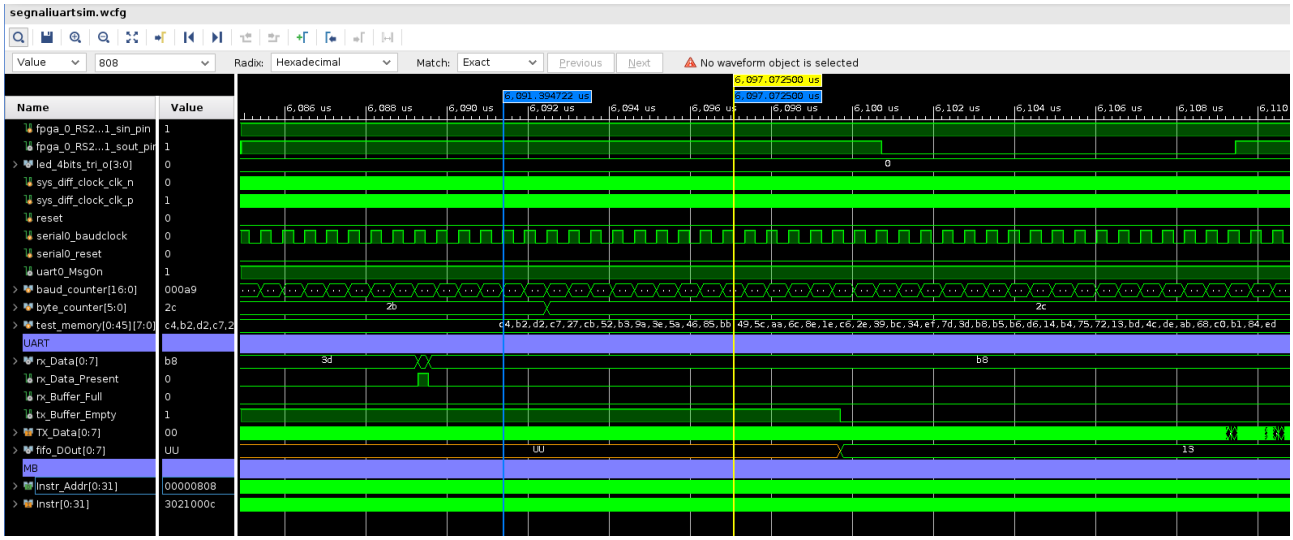



*Figure 31: convolution function duration with mul64*

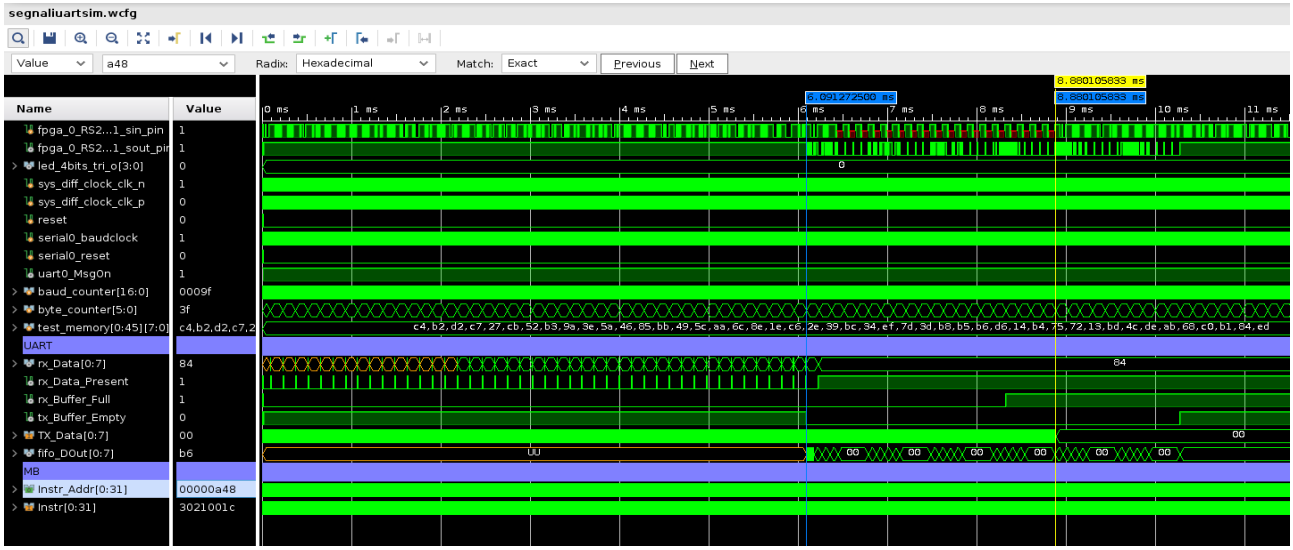



*Figure 32: send function duration with mul64*

With respect to receive function I saw a preculiarity: in the waveform I also found its start address also during receive function. I did not see this in the simulation without mul64, so could be this a way for the microprocessor to move up the previous instruction?

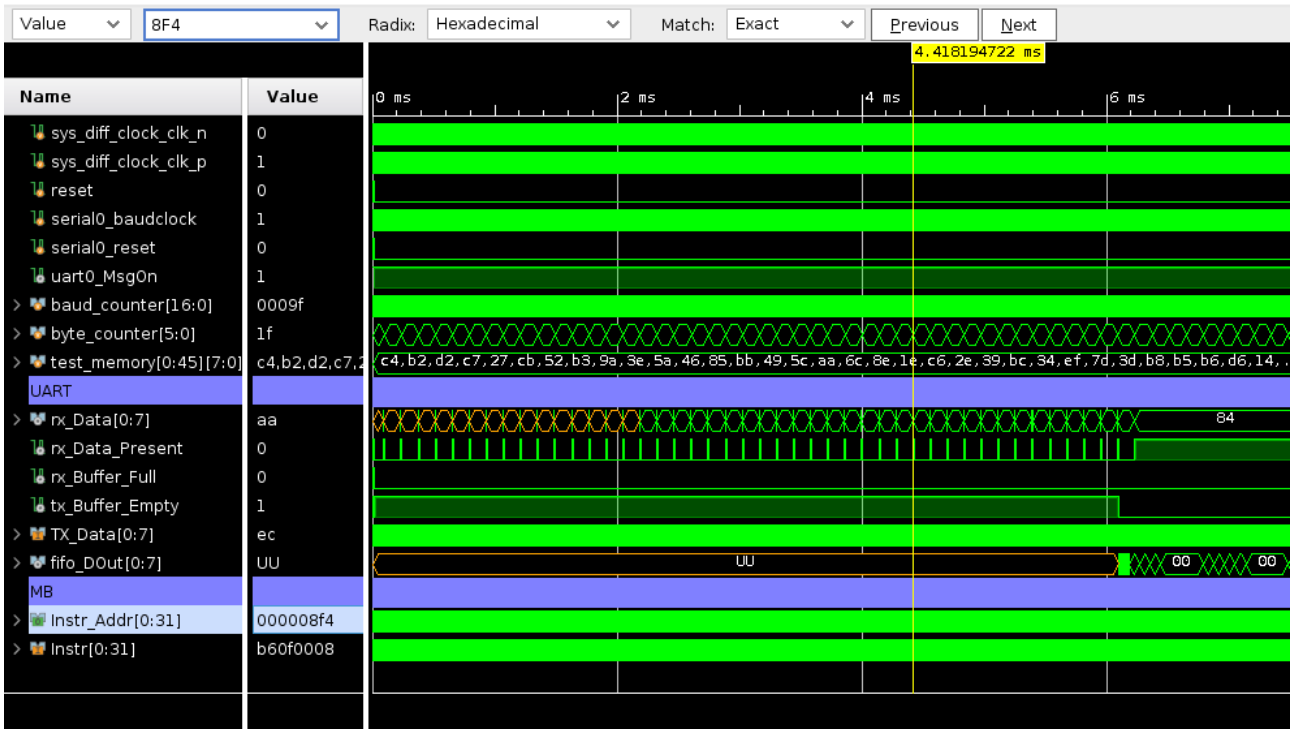



*Figure 33: instruction 8F4 (start of send function) also found before effectively send*

Regarding the assembly I did not commented the one obtained with mul64 because it has the same instructions of the project without mul apart from the following ones:

- mul: this is the instruction for multiplication, it has the form mul rD,rA,rB and takes the content of register A and B, makes the multiplication and stores the least significant word of the result on register D (the most significant byte is discarded, so this on is in little endian). Both registers A and B are 32-bit and register D (as is their multiplication) is 64-bit (two 32-bits multiplied by each other give a 64-bit)
- muli: it is the same of mul but has the form mul rD,rA,IMM and so here the multiplication is between the content of one register (rA) and a value. IMM, the value, is a 32-bit integer.

In my assembly I find this instructions in the convolution function when I make the convolution operation (the sum of the product of 3 elements of x with the 3 elements of w); in the function receive when I make the left shift to store the values in x and w; in the send function when I make the right shift to send y values.

**Conclusions**

So, with this assignment I learned how to create a project to make communicate a microprocessor with the external world. This was done without the use of multiplier and also using it with 64 bit. Here down you can see the difference in time and cycles.

| Function | Dur nomul | Cyc nomul | Dur mul64 | Cyc mul64 |
|---|---|---|---|---|
| Main() | 8897,32 | 889732 | 8874,54 | 798709 |
| Receive() | 4413,68<br>1673,38 | 441368<br>167338 | 4412,25551<br>1672,977 | 397103<br>150568 |
| Convolution() | 45,13 | 4513 | 5,677778 | 511 |
| Send () | 4483,39 | 448339 | 2788,8333 | 250995 |

*Figure 34: comparison table between no mul (second and third columns) and with mul64 (4th and last column). Expressed in microseconds.*