I. Project Statement

I.a. Objective

The objective is to learn to understand, design, and simulate sequential logic devices and blocks that make up an 8-bit processor with Harvard architecture.

I.b. Project Description

The project consists of generating an 8-bit processor whose goal is to allow data to be written into memory, and then writing into another memory the operations I want it to perform and it will send the results back. This processor is composed of 6 basic blocks not counting the registers; the basic blocks are the pulse generating machine (GCM), the program counter (PC), the data and instruction memories, the arithmetic logic unit (ALU), and the instruction decoder.

I.c. Methodology

- **Problem statement:** We sought to understand what the problem was and obtained base data to propose a solution.
- **Problem solving:** An abstract idea was created on how to solve the problem.
- **Component research:** It was investigated which components were suitable for solving the problem.
- **Design:** A design was created with these components to provide a real solution to the problem.
- **Simulation and verification:** It was simulated and verified that the design met and solved the problem.

II. Theoretical Framework

II.a. Digital counters

Digital or electronic counters are synchronous and/or asynchronous sequential type circuits, which have a clock-type input (a square pulse) that activates a series of logic circuits to output a number in a code format that another component such as a microprocessor, a 7-SEG display or an LCD display can understand.

Their purpose is to increment and/or decrement the number with each clock pulse they receive. The output codes are generally binary, although they can also have hexadecimal, octal outputs, among others.

II.a.1. Synchronous counters

They are mainly composed of flip-flops that have a clock signal, which are activated simultaneously, executing each flip-flop at the time of the square or clock pulse. It can also be said that the operations of the flip-flops are executed in parallel, unlike asynchronous counters which are serial. [1]

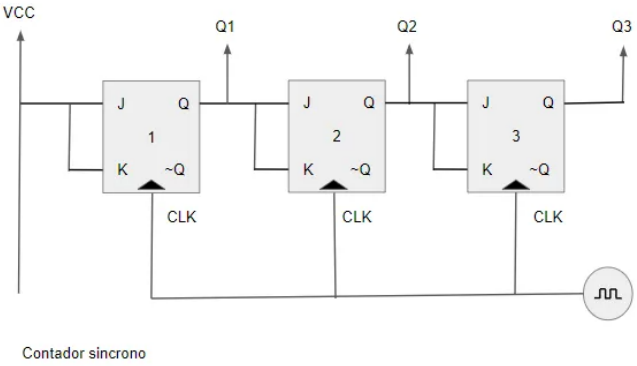

Figure 1: Synchronous counter diagram

II.b. ROM

Read-Only Memory, or ROM, is a type of memory where information is programmed or recorded into the device, and during normal operation it only allows read access.

In ROM, there are two main parts: the address and the stored data. The address is the number that identifies which data is being accessed.

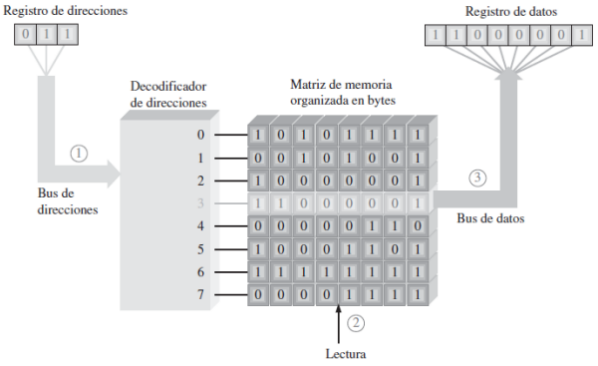

Figure 2: ROM

II.c. Ports

The microcontroller ports are the communication point between the microcontroller and the external world. Through them, electronic control processes can be performed on power devices, instrumentation, telemetry, etc. [5]

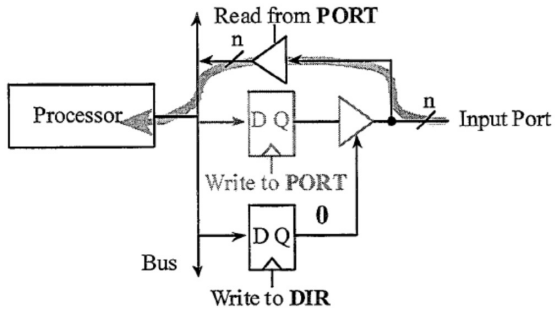

Figure 3: Bidirectional port

II.d. Extra truth tables

NAND		
A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0

XNOR		
A	B	Q
0	0	1
0	1	0
1	0	0
1	1	1

NOR		
A	B	Q
0	0	1
0	1	0
1	0	0
1	1	0

II.e. Interrupt

An interrupt is a signal that activates a series of processes whose goal is to stop the main program to execute higher-priority operations in the processor.

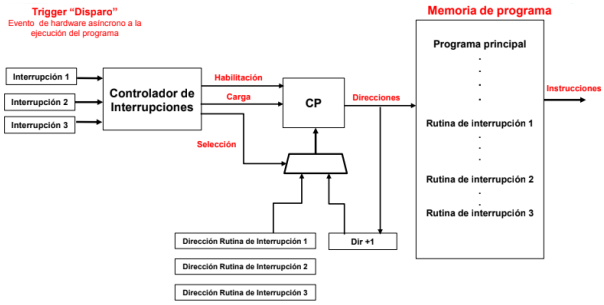

Figure 4: Interrupt diagram in a microcontroller [6]

As shown in Figure 4, different interrupts can exist, so it is necessary for them to have a priority which will determine the execution order. Then they go through a masking stage and then to the interrupt controller. The interrupt controller is responsible for stopping the program counter, storing its last address, then loading the interrupt addresses to execute them, and finally returning the program counter to its original state so the main program can continue.

III. Design

In the design section it will be divided into 4 parts: first, the 3 components designed which are: the counter, the multiplexer, and the ROM. The last section will consist of unifying all these components.

III.a. Cycle Generating Machine (GCM)

For the GCM, it was required to design a counter that counts from 0 to 7, which are the necessary outputs to activate the different processes of the processor. Theoretically, only 5 outputs are needed but we require 3 zeros or 3 spaces between operations so that the information loads properly.

NOTE: The Flip-flop used is the T Flip-Flop and its truth table can be seen in [2]

State Table											
Present State				Next State				T Flip-Flop			
Q_A	Q_B	Q_C	Q_D^+	Q_A^+	Q_B^+	Q_C^+	Q_D^+	T_A	T_B	T_C	T_D
0	0	0	0	0	0	0	1	0	0	0	1
0	0	0	1	0	0	1	0	0	0	1	1
0	0	1	0	0	0	1	1	0	0	0	1
0	0	1	1	0	1	0	0	0	1	1	1
0	1	0	0	0	1	0	1	0	0	0	1
0	1	0	1	0	1	1	0	0	0	1	1
0	1	1	0	0	1	1	1	0	0	0	1
0	1	1	1	0	0	0	0	0	1	1	1

After this, we proceed with the table to create our Karnaugh maps of the 5 Flip-Flops

T_A	00	01	11	10
00	0	0	0	0
01	0	0	0	x
11	x	x	x	x
10	x	x	x	x

T_B	00	01	11	10
00	0	0	1	0
01	0	0	1	x
11	x	x	x	x
10	x	x	x	x

T_C	00	01	11	10
00	0	1	1	0
01	0	1	1	x
11	x	x	x	x
10	x	x	x	x

T_D	00	01	11	10
00	1	1	1	1
01	1	1	1	x
11	x	x	x	x
10	x	x	x	x

Then, once this is obtained, we get the functions of each T input which are:

$$T_A = 0 \tag{1}$$

$$T_B = Q_D Q_C \tag{2}$$

$$T_C = Q_D \tag{3}$$

$$T_D = 1 \tag{4}$$

Once we have the counter, with the following pulse diagram in Figure 5 we create the necessary AND gates so that they are activated in that order.

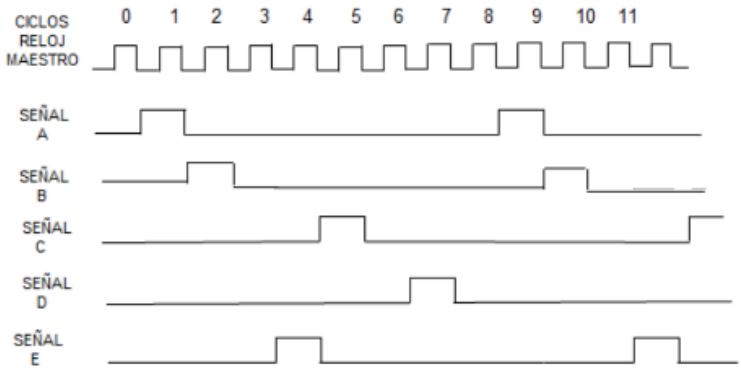

Figure 5: Pulse diagram for the GCM

For this, the first pulse which would be 0000 in the GCM is created using an AND gate where all the inputs are negated and the output is signal A. Then for signal B, it is the next GCM state which would be 0001, so the inputs from Qa to Qc are negated and Qd is normal, and so on until all signals sent by the GCM are in order as shown in Figure 6.

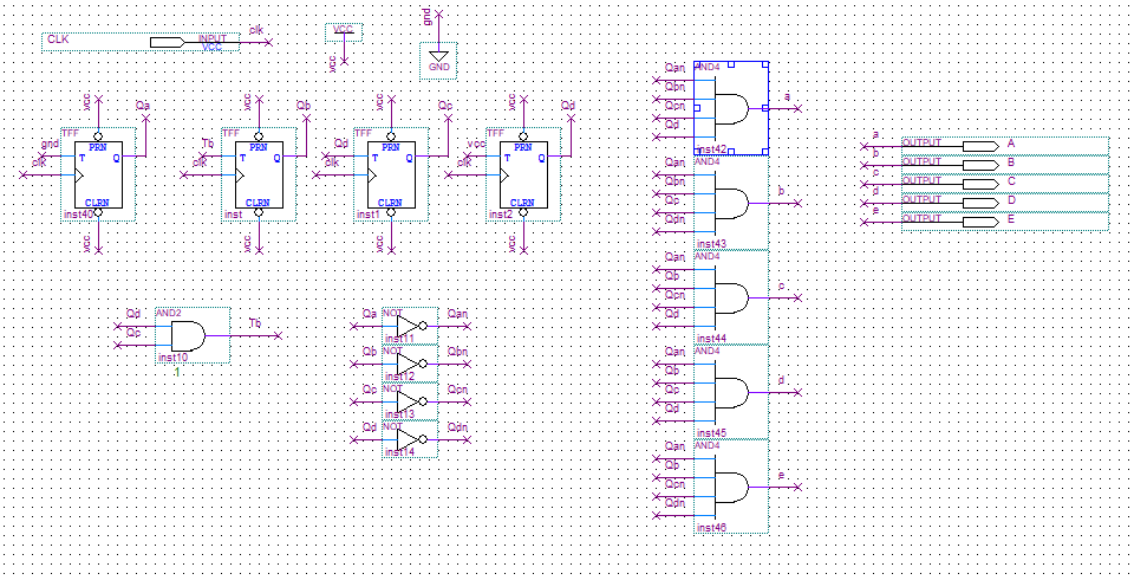

Figure 6: GCM

III.b. Program Counter (PC)

For the program counter, it was realized that in the truth table of the T flip-flop together with the state table we can observe a phenomenon where the first flip-flop is 1, the second is the input of the first, the third is the AND of the two previous ones, and so on. This happens when we want to make a synchronous counter of order 2^n where n is the number of flip-flops that follow this sequence. So, to count 8 bits, 8 flip-flops are needed and it will count up to 256, thus allowing us to have 256 instructions and data.

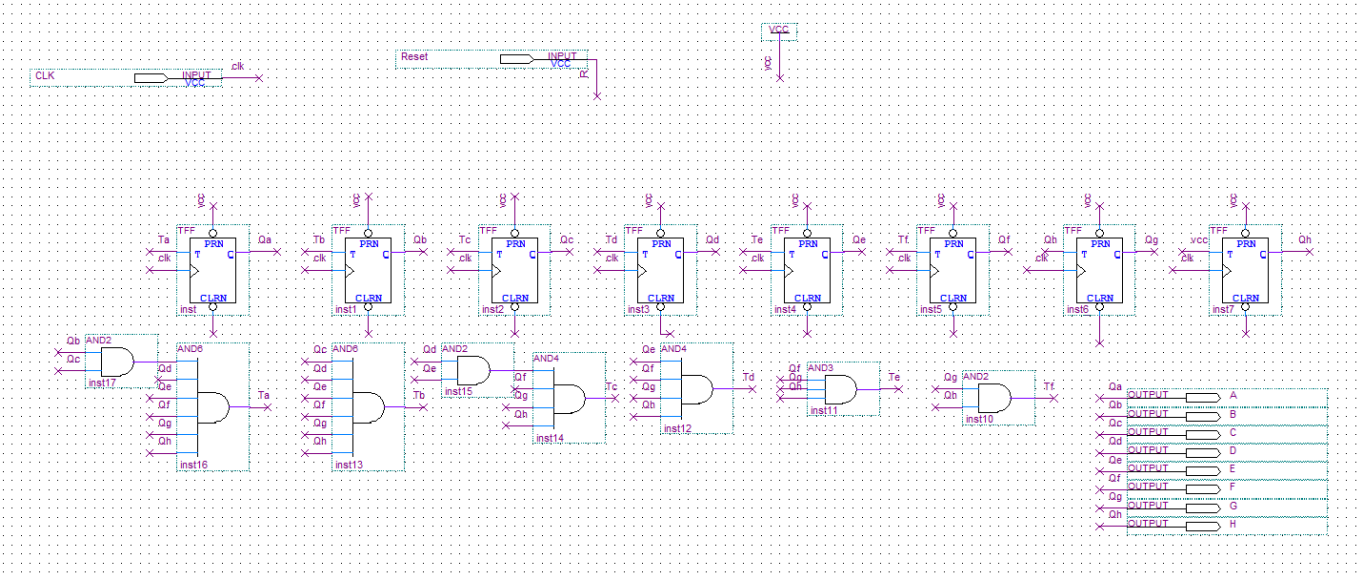

Figure 7: PC

In the figure, the above is shown.

III.c. Instruction Register

To design the instruction register, D Flip-Flops were used since these only store 1 bit and output what they have stored without altering it. In this setup, 8 Flip-Flops are needed since the instructions are 8-bit in size, where the connection is parallel input – parallel output. Also, the clock of the flip-flops is all connected to the same GCM output that gives the instruction to store the data. This register is connected to the memory that stores the instructions.

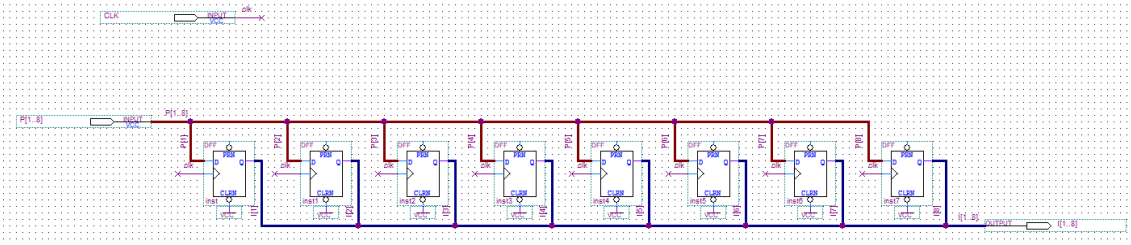

Figure 8: Instruction Register

III.d. Data Register

The data register follows the same principle and operation as described above in the instruction register, the only difference is that this one is connected to the memory that stores the data.

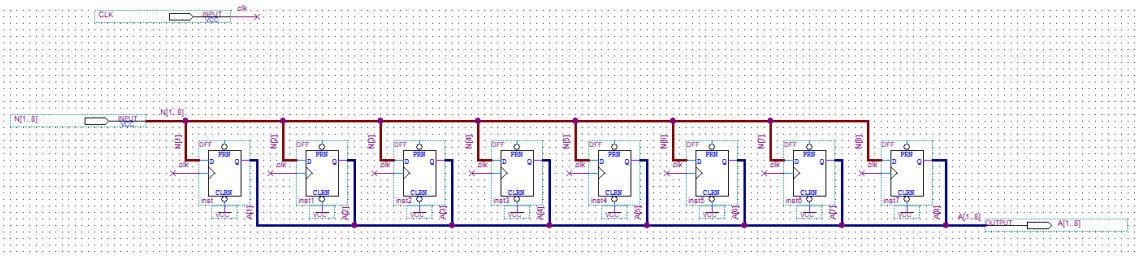

Figure 9: Data Register

III.e. Temporary Accumulator

The accumulators, both the temporary and the regular one, are basically registers that store the output of the operation. Just like the registers mentioned earlier, they have 8 flip-flops with parallel-parallel connection. This register serves as temporary data storage since if it were stored in a single accumulator, problems could arise when the number being operated on in the ALU changes at the same time.

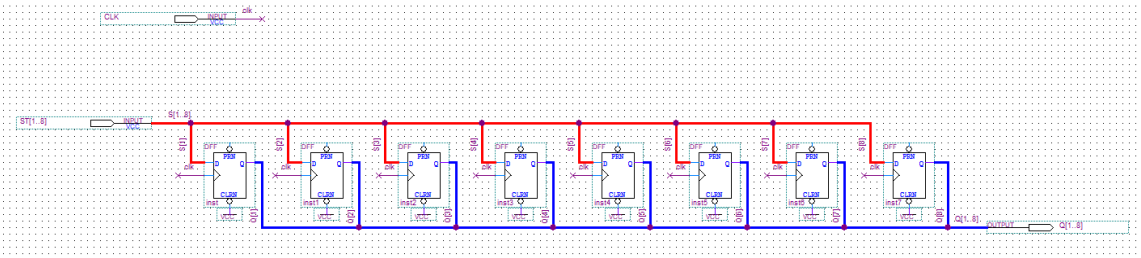

Figure 10: Temporary Accumulator

III.f. Accumulator

The function of the accumulator is to provide the final result and feed the ALU with the “B” number to perform operations either with it or with another number stored in the data memory “A”. This is a copy of the temporary accumulator design.

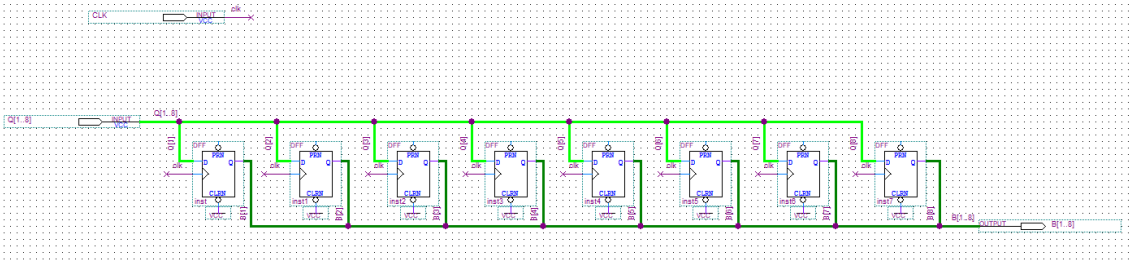

Figure 11: Acumulator

III.g. 8 × 16 Instruction Decoder

The decoder's purpose is to send which instruction to store in the accumulators. It receives the 8-bit instruction from the instruction register, decodes this number, and enables the tristate of one of the 10 operations, thus storing this data in the accumulator. So, when 00000000 is sent, instruction 1 is enabled; when 00000001 is sent, instruction 2 is enabled; and so on until reaching instruction 00001111, which would be 16.

Number	Instruction
00000000	No operation
00000001	Load Data
00000010	Subtract
00000011	Shift
00000100	Invert
00000101	Add
00000110	XOR
00000111	OR
00001000	AND
00001001	Reset
00001010	Port Read
00001011	Port Write
00001100	Timer
00001101	Configure Port
00001110	End of Interrupt
00001111	Configure Mask

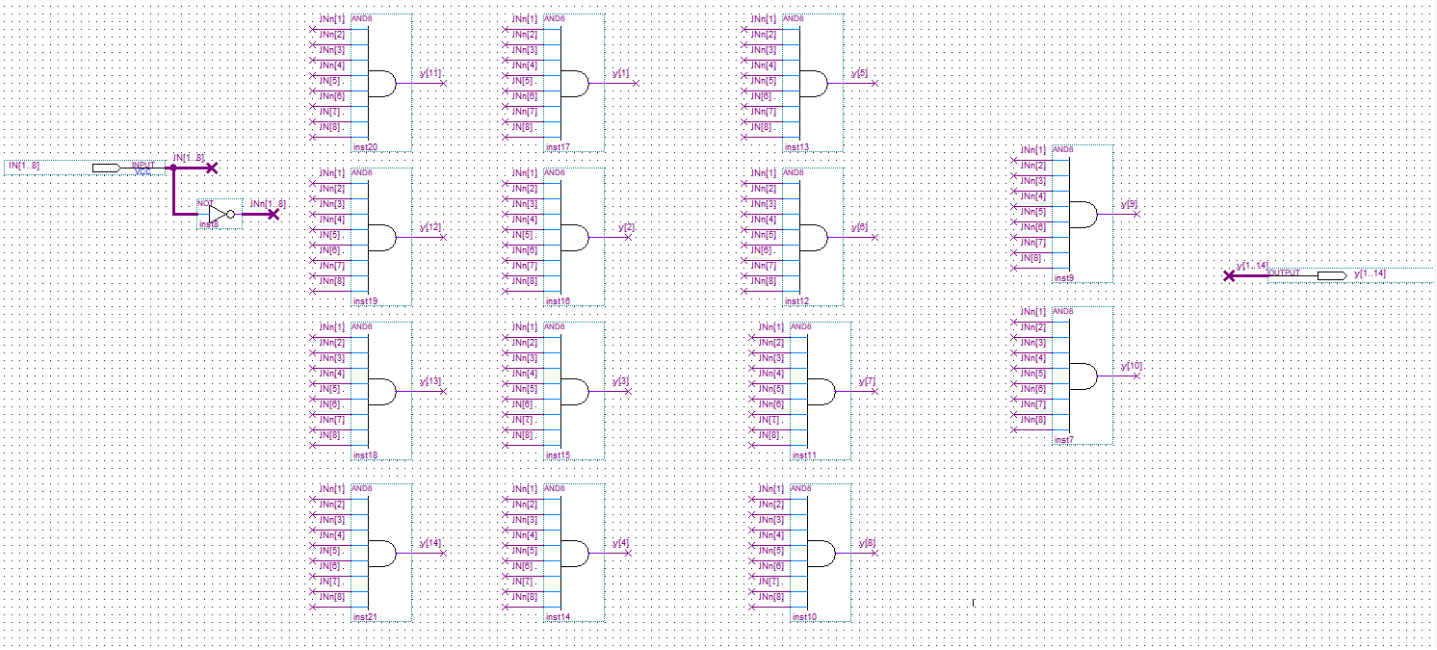

Figure 12: 8X14 Decoder

Truth Table

Instruction	Input Number A[1:8]								Decoder Output y[1:14]														
Number	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	y[1]	y[2]	y[3]	y[4]	y[5]	y[6]	y[7]	y[8]	y[9]	y[10]	y[11]	y[12]	y[13]	y[14]
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
2	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0
3	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
4	0	0	0	0	0	0	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0
5	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
6	0	0	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
12	0	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0
13	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
14	0	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1

III.h. Arithmetic Logic Unit (ALU)

The arithmetic logic unit is responsible for performing operations with the numbers it receives; it contains both a number .ᴬ.ªnd a number "B"which are the operands.

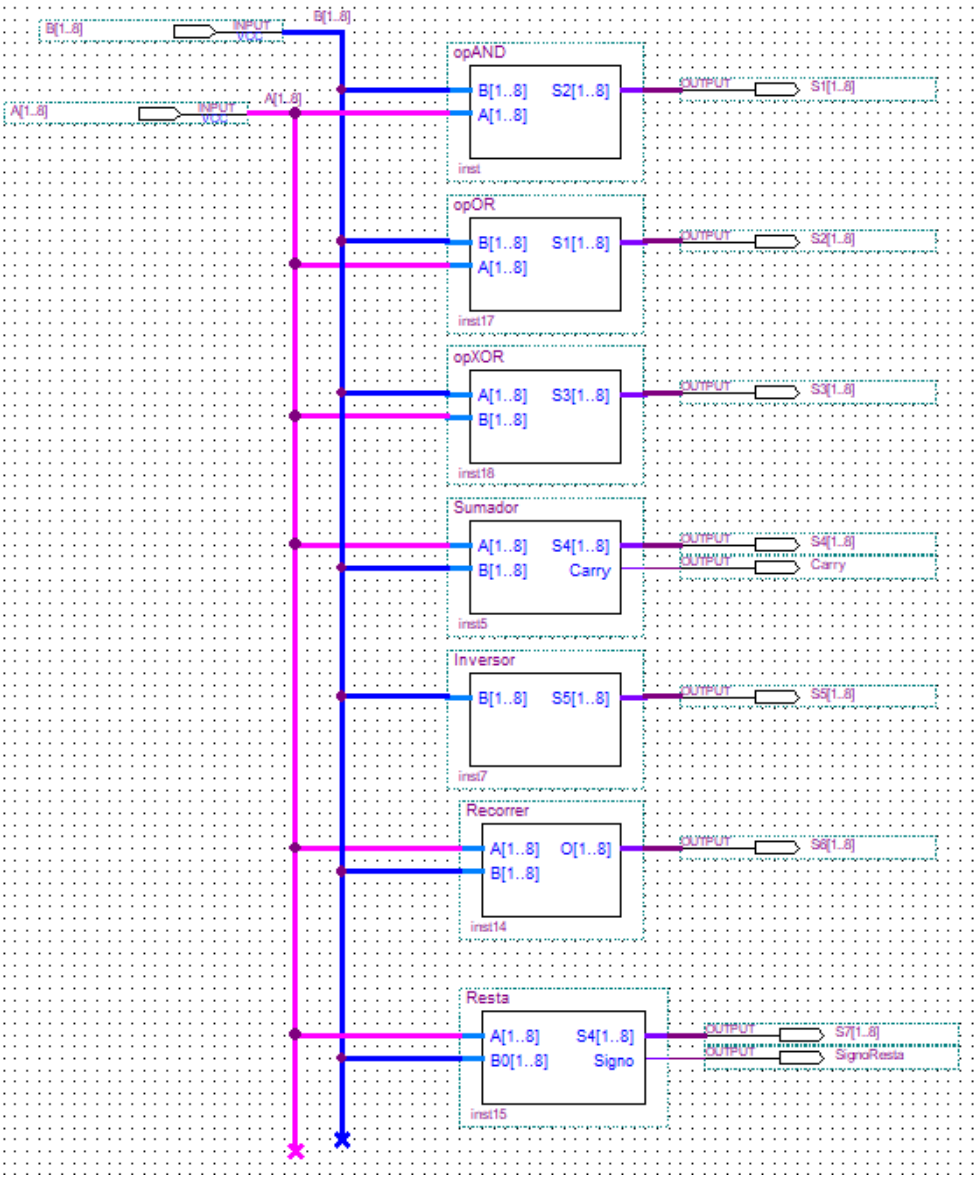

Figure 13: ALU

III.h.1.

AND In the AND operation, this logical function is performed between the two 8-bit numbers A and B. The truth table is:

AND		
A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

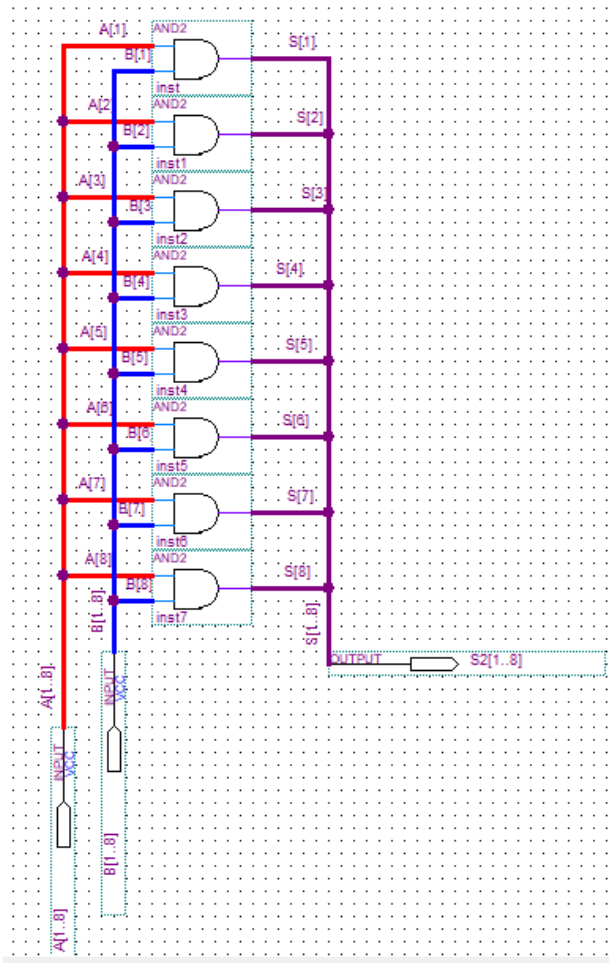

Figure 14: AND

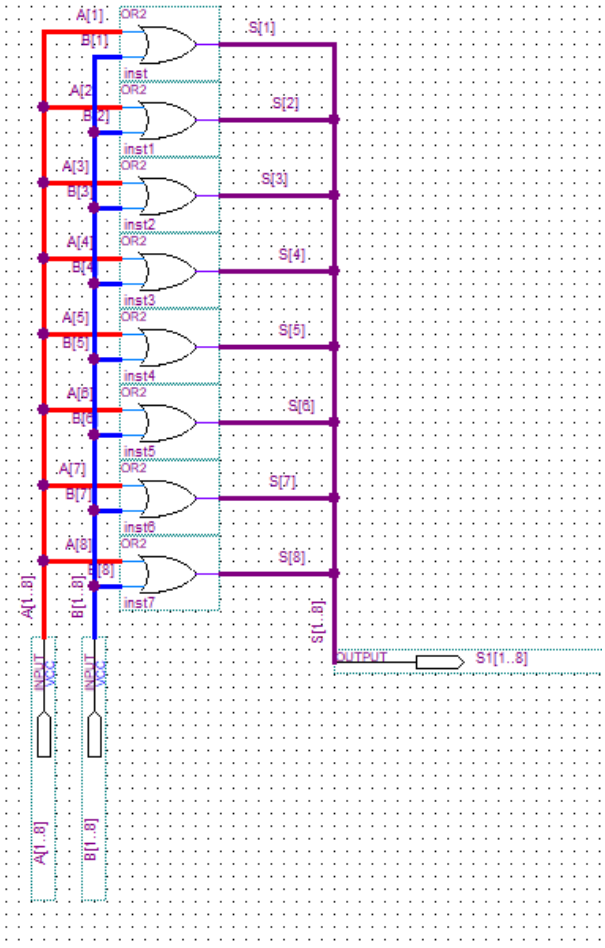

Figure 15: OR

III.h.2. OR

In the OR operation, this logical function is performed between the two 8-bit numbers A and B. The truth table is:

OR		
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1

III.h.3. XOR

In the XOR operation, this logical function is performed between the two 8-bit numbers A and B. The truth table is:

XOR		
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

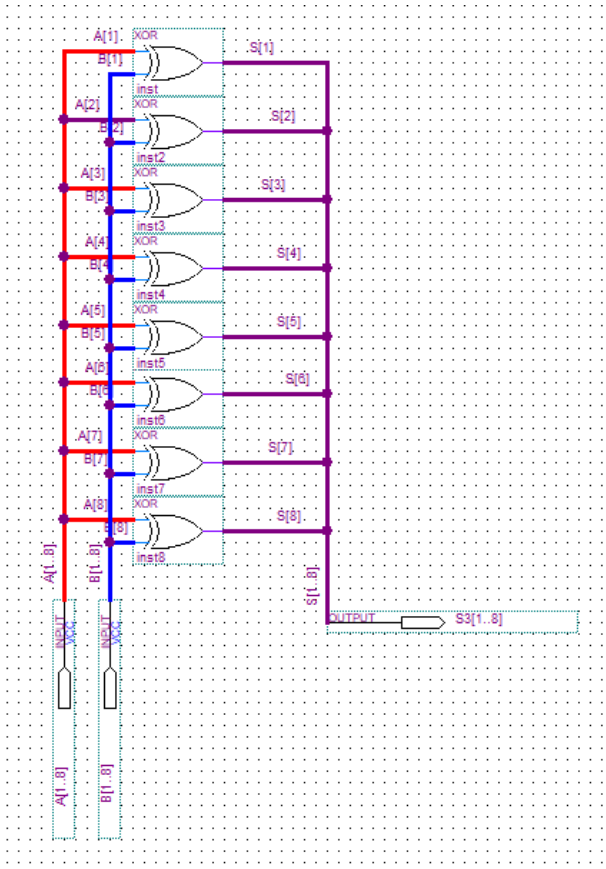

Figure 16: XOR

III.h.4. Addition

For the addition operation, I designed a 1-bit Full-Adder **??** which I connected as shown in figure 17. There were 8 of these full adder blocks because the numbers received were 8 bits. They were connected so that the carry from the least significant bit entered the next operation between the two-bit numbers until reaching the most significant bit, where if the result was a number larger than 8 bits, the overflow flag would be set, and this goes to the program output indicating that the representation size was exceeded.

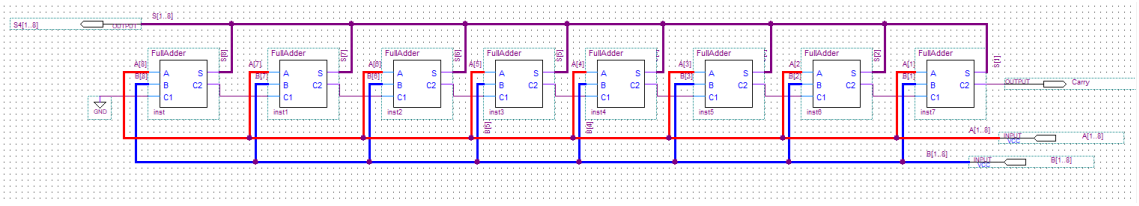

Figure 17: Addition

In the next figure, we can see the internal design of the 1-bit full-adder block.

III.h.5.

Inverter In the inverter operation, the logical NOT function is performed on number B. The truth table is:

NOT	
A	Q
0	1
0	1
1	0
1	0

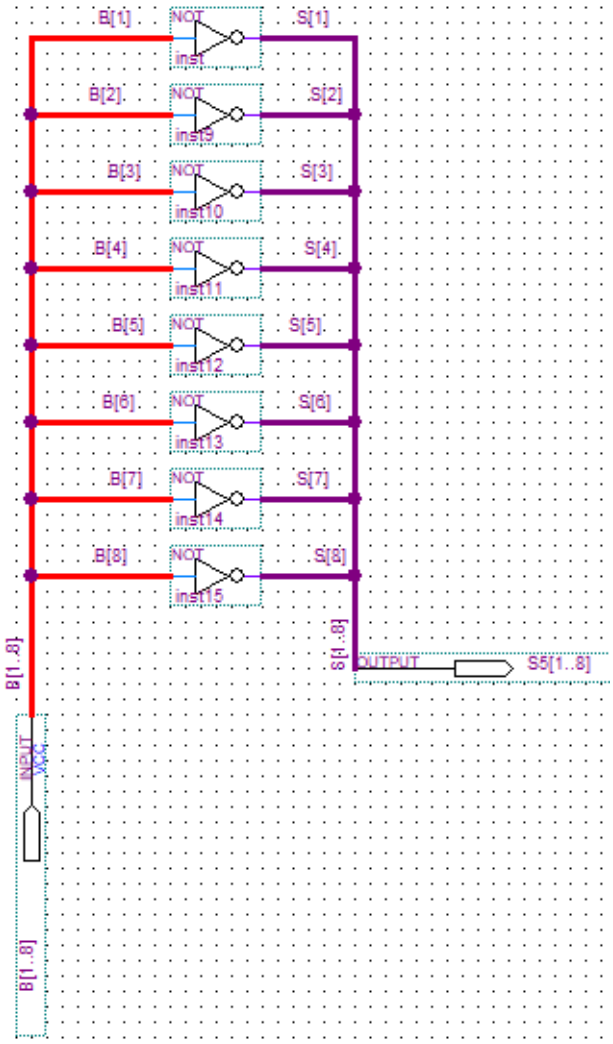

Figure 19: Inverter

III.h.6. Shift

For the shift operation, two main blocks were used: first, an 8X8 decoder which took the output from the data register to see how much it had to shift; then 8 blocks where each shifted the number a different amount. For example, with the shift instruction and the number 6, the number was shifted 6 places and the empty spaces were filled with 0.

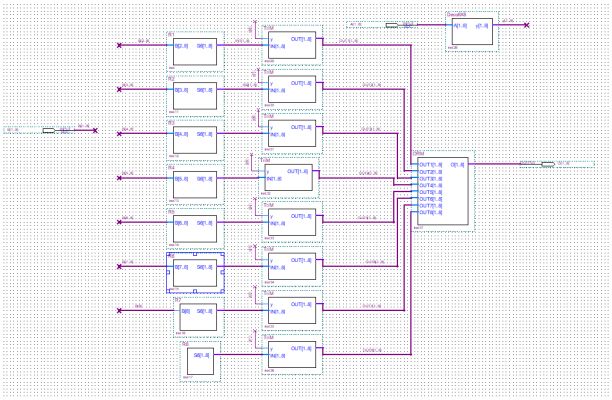

Figure 20: Shift

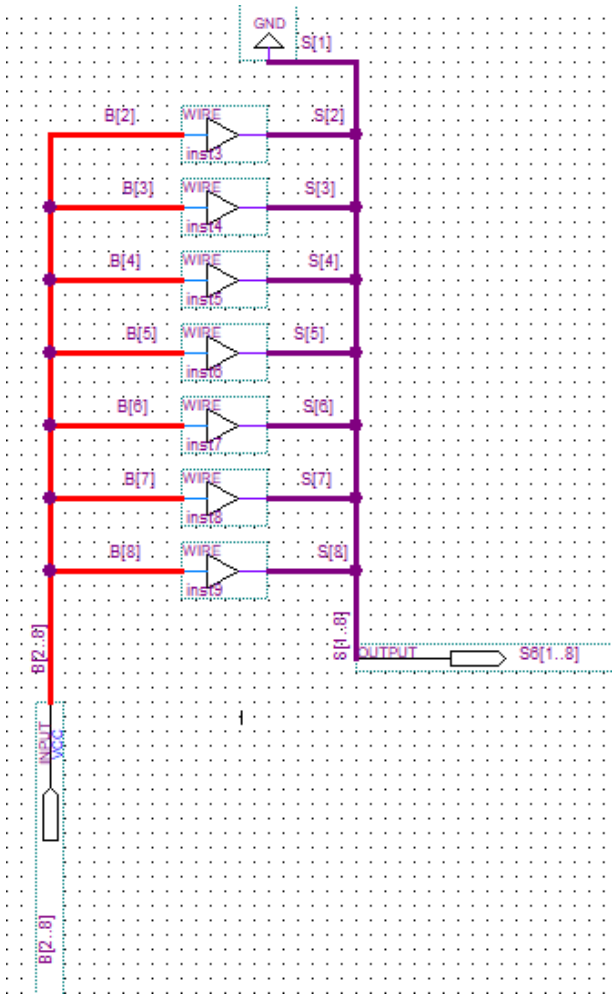

Figure 21: 1-bit Shift

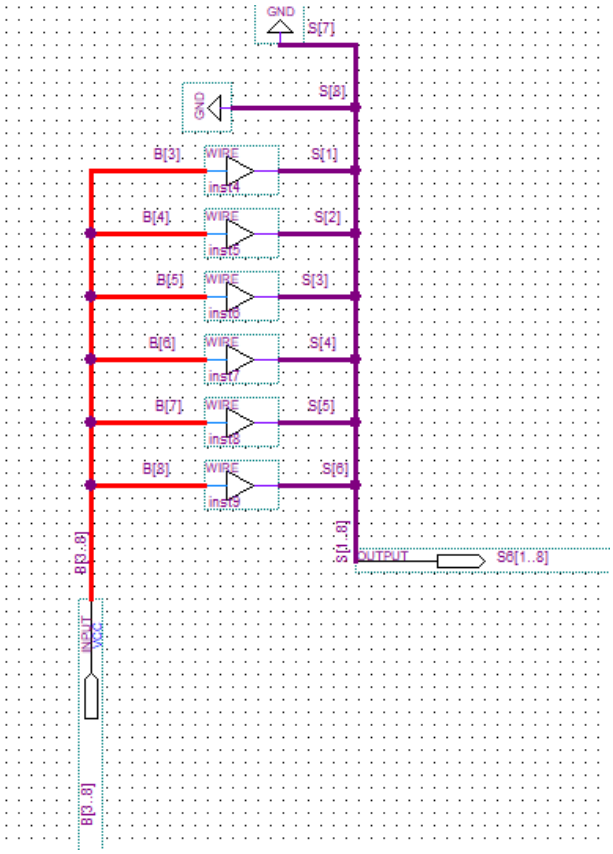

Figure 22: 2-bit Shift

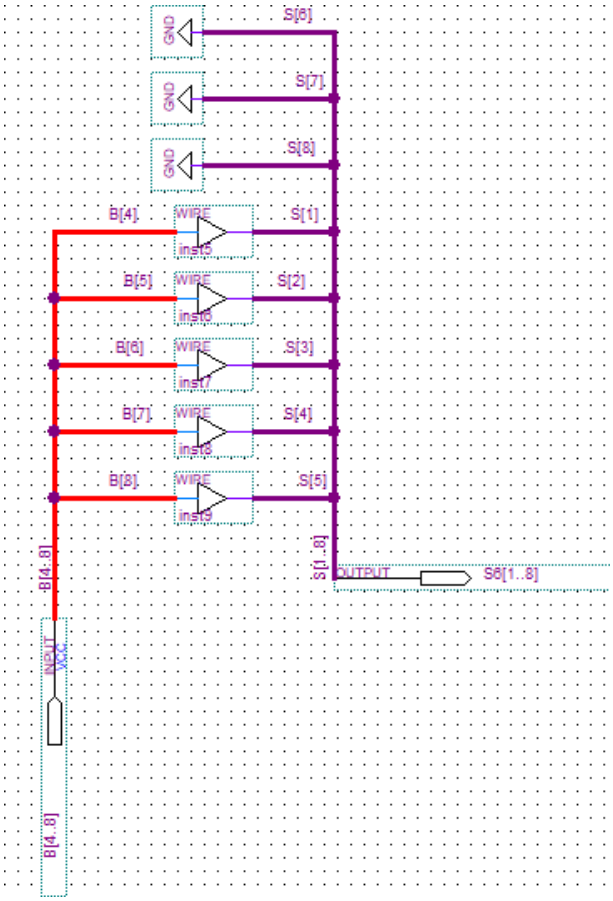

Figure 23: 3-bit Shift

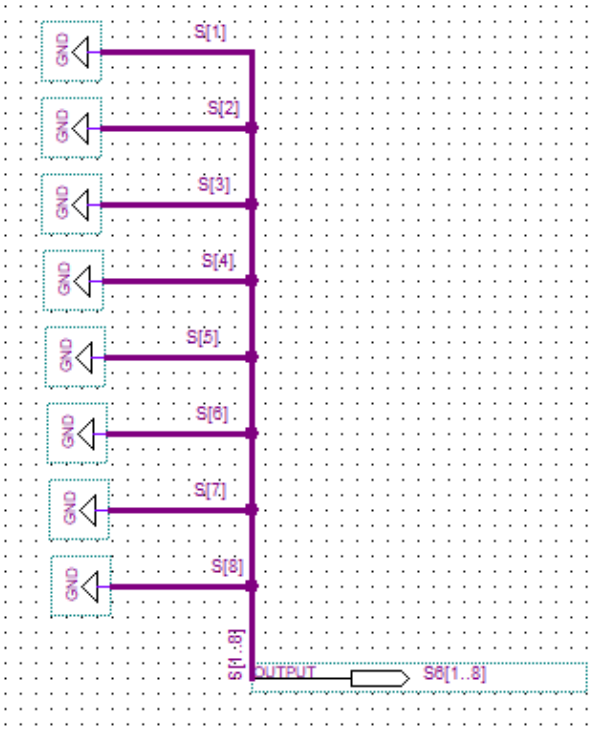

Figure 24: 8-bit Shift

In the previous figures, we can observe how the sequences from 1 to 8+ were constructed. In the following figure 25, we can see how the decoder is composed, where depending on the data, the sequence was determined. When the number is 1000 in binary or 8 in decimal, we can see that the output is all zeros, so it was added in the decoder that when the 4th, 5th, 6th, 7th, or 8th bit are on, the result will be 00000000.

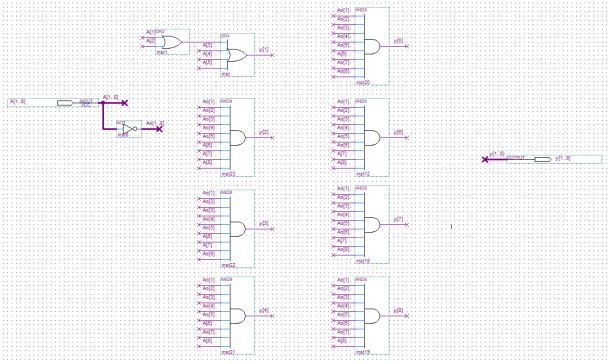

Figure 25: 8x8 Decoder

III.h.7. Subtraction

For subtraction, only what we already had from the full-adders **??** and the adder 17 was used. To this, the 2's complement 27 was added, which consists of inverting a number and adding 1 to obtain its 2's complement

notation. Then, we add it with number A, and at the end, we get the result in 2's complement. At this point, if the number was negative, the "Sign"flag turns off indicating it is negative, and it passes through another 2's complement block to obtain the magnitude of the number while the flag indicates its sign.

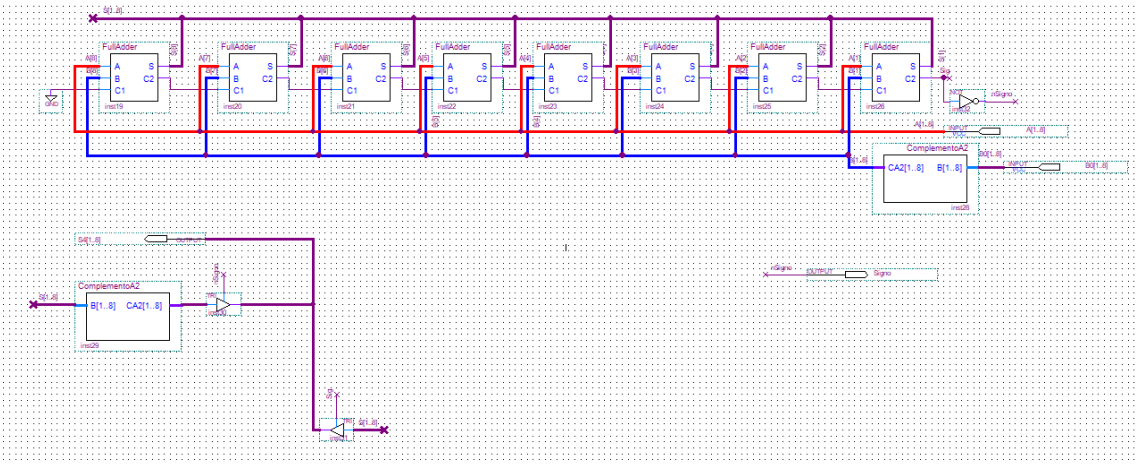

Figure 26: Subtraction

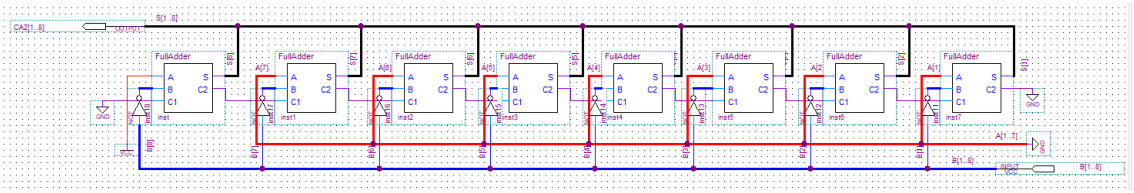

Figure 27: 2's Complement

III.i. Timer

For the timer, I divided it into three main circuits: the counter, the comparator, and the combinational logic.

The counter was taken from the PC design, which counts from 0 to 255 as it is 8 bits. It starts counting once it detects the decoder signal 12. While counting, the comparator, which consists of 8 XNOR gates that output 1 when both inputs are equal and 0 otherwise, allows us to detect when the input number we want to count and the counter number are equal. When all the XNORs are on, this goes to a NAND gate that sets the output to 0 once this happens.

The combinational logic is as follows: We have a clock signal (CL) that updates the flip-flops of the counter (clk). This clock first passes through an AND gate, where the condition to send the clock to the flip-flops is that the number currently in the counter is not the desired number. So the clock input is CL AND CU (meaning the condition has not yet been met).

Then the same clock signal goes to a tristate, where the enable condition is that the counter's number is equal to 0 or that the counter has not yet reached the desired number. If either condition is true, the clock passes to a flip-flop that reads the decoder bit and stores and transmits it to the resets of the counter.

This means that the counters never start because the decoder output y[13] is 0, activating the counter reset and preventing it from advancing from 0. The clock of this flip-flop keeps updating the flip-flop's state looking for when a 1 signal arrives. When this happens, the resets deactivate allowing it to start counting. To allow the microprocessor to keep working, the clock to the flip-flop reading the decoder signal is blocked

while the counter is not zero or hasn't reached the desired number, so the flip-flop state does not change and holds 1 until the desired number is reached, stopping the count at that time.

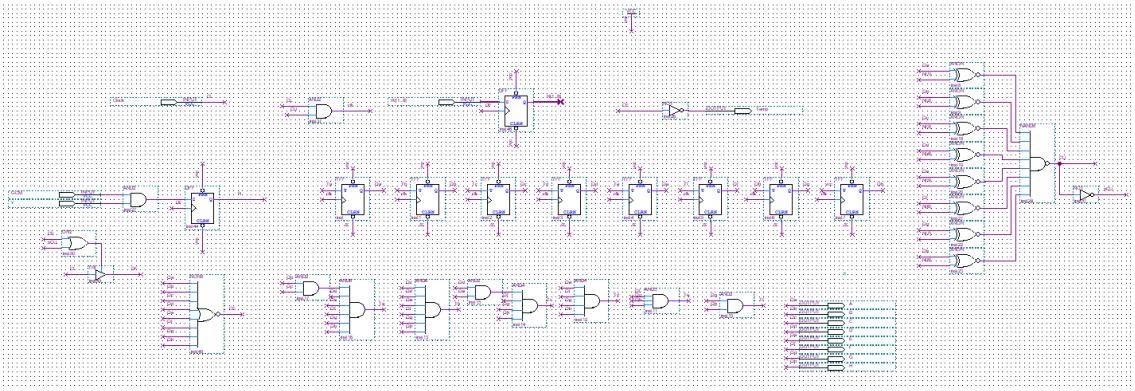

Figure 28: Timer

At this point, there is an error, which is that the counter does not reset to 0, and the flag does not drop until after 8 cycles required to change the instruction because y[13] continues receiving 1. Therefore, a final condition is added with an AND gate representing that for the flip-flop to be 1, the decoder must send the signal and the GCM must have its load signal active on the registers plus the next signal. This will always be true, so it does not affect the operation, but causes the flip-flop to return to 0 after one cycle, so no problems occur counting from 1 to 8.

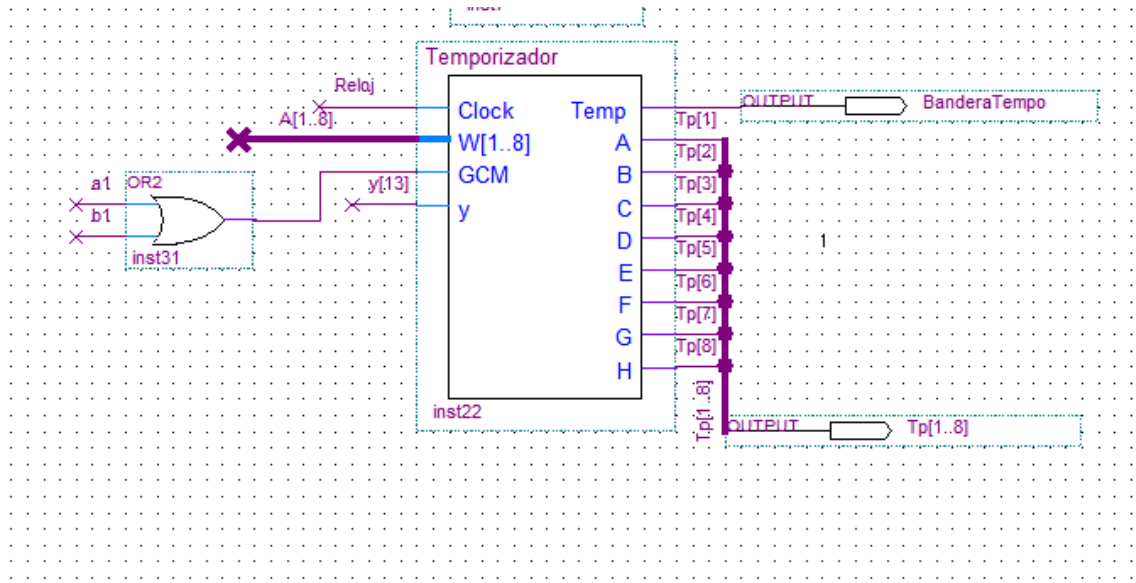

Figure 29: Timer Block

III.j. Input/Output Port

For the port, we made a port that works as input or output depending on the configuration. First, the instructions in the decoder 12 were defined, which correspond to 11, 12, and 14.

The first instruction is 14, which configures the port. When configuring, we get the number stored in the data memory, which has two values: 0 to configure as read and 1 as write. Once configured, we can send the next instruction, which can be either read or write. If we choose write, it writes the data in the accumulator

to the port (B[1..8]) and this is shown on the bidirectional pins.

For reading, a multiplexer is required to stop the data flow from memory to the register and send the data from the port to the register. The multiplexer was designed as a 2x1 as shown in figure 31.

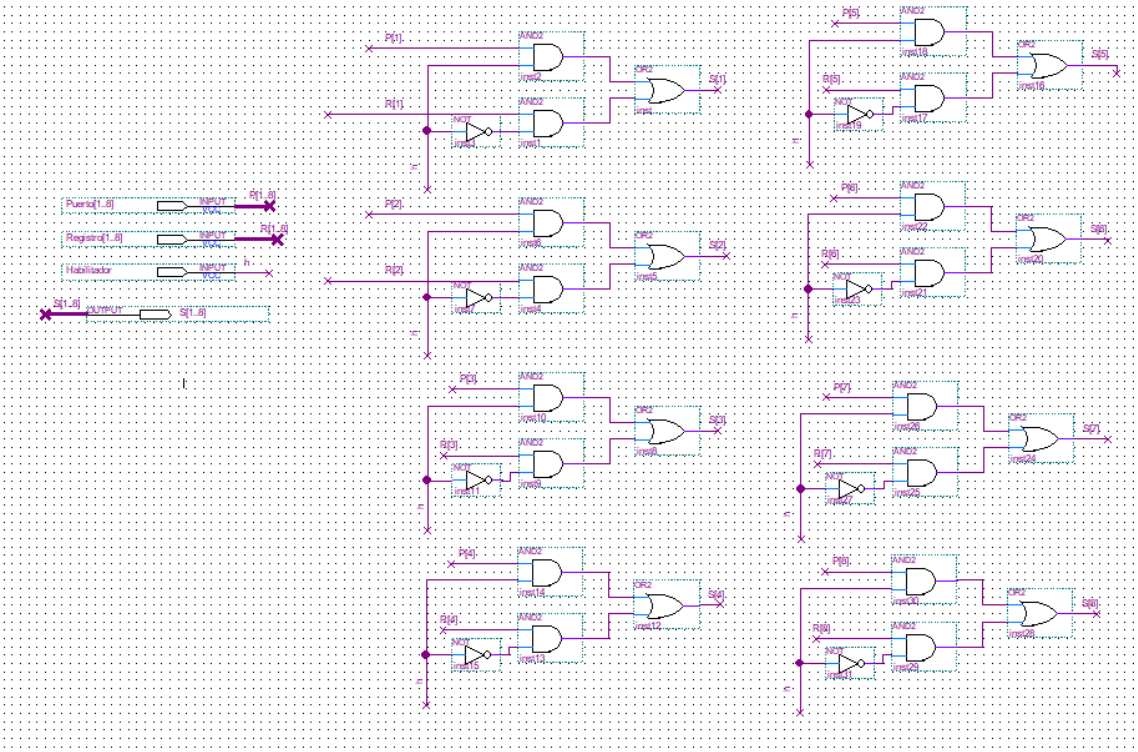

Figure 30: 2x1 Multiplexer

Once it selects whether the data comes from the data memory or from the port, the port is ready to send its data.

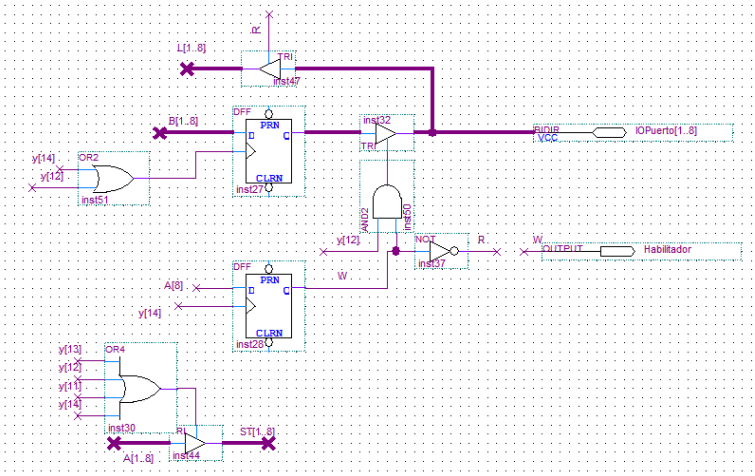

Figure 31: Port Logic

IV. Interrupt

As seen in Figure 4, the first thing we need is to have a port configured for writing and a masking mechanism. The port used will be the one developed earlier, and the masking logic was created as follows:

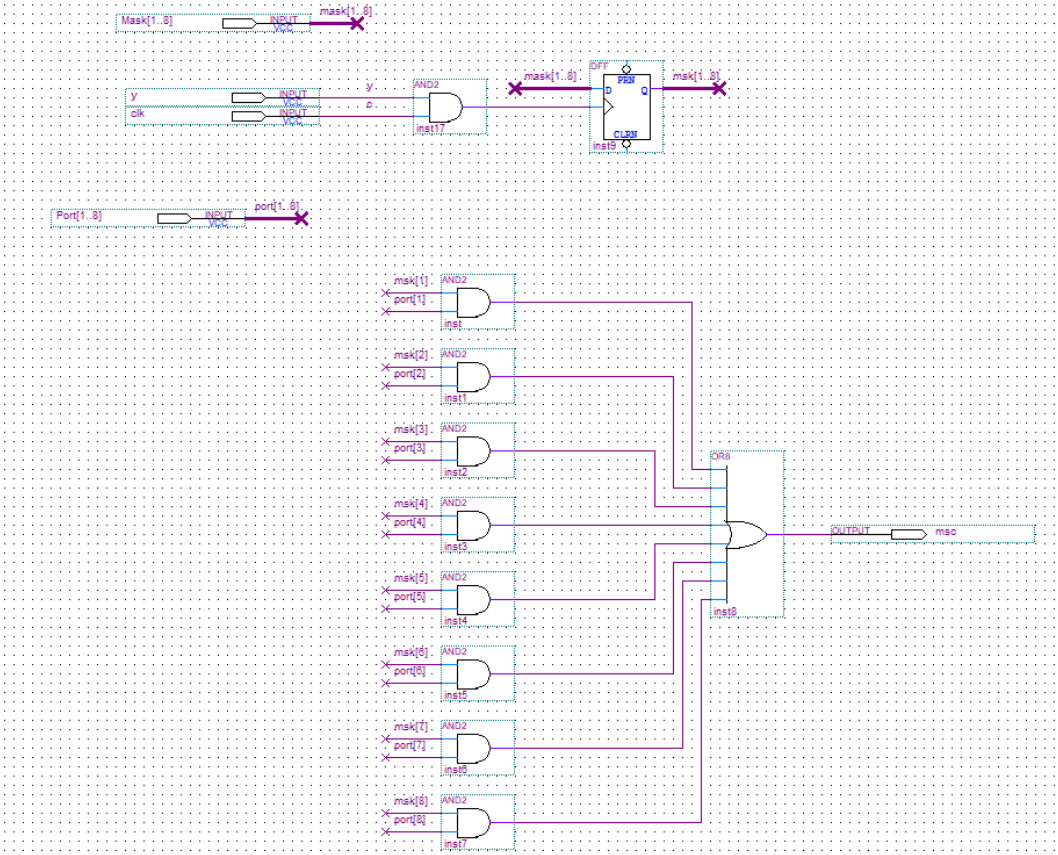

Figure 32: Masking

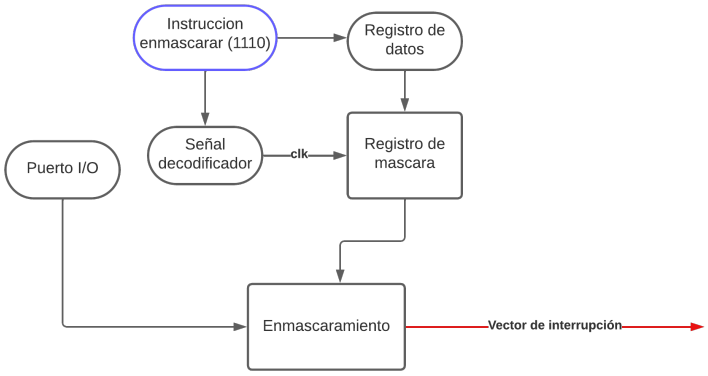

Figure 33: Masking Diagram

First, we generate a new instruction in our decoder, which is the masking instruction. When activated, it obtains the value from the data register and turns on the corresponding bits for the port. Then, as observed in Figure 33, we have our mask input, which is the output of the data register and is stored in flip-flops that

hold the same value as long as it is not reactivated, and y[16], which is the decoder signal for masking. After this, it passes through AND gates activated depending on the data register, and finally through an 8-input OR gate. Thus, if an active bit is detected from the port, it passes through, and the block sends a signal to activate the interrupt.

Next, we generate our interrupt controller.

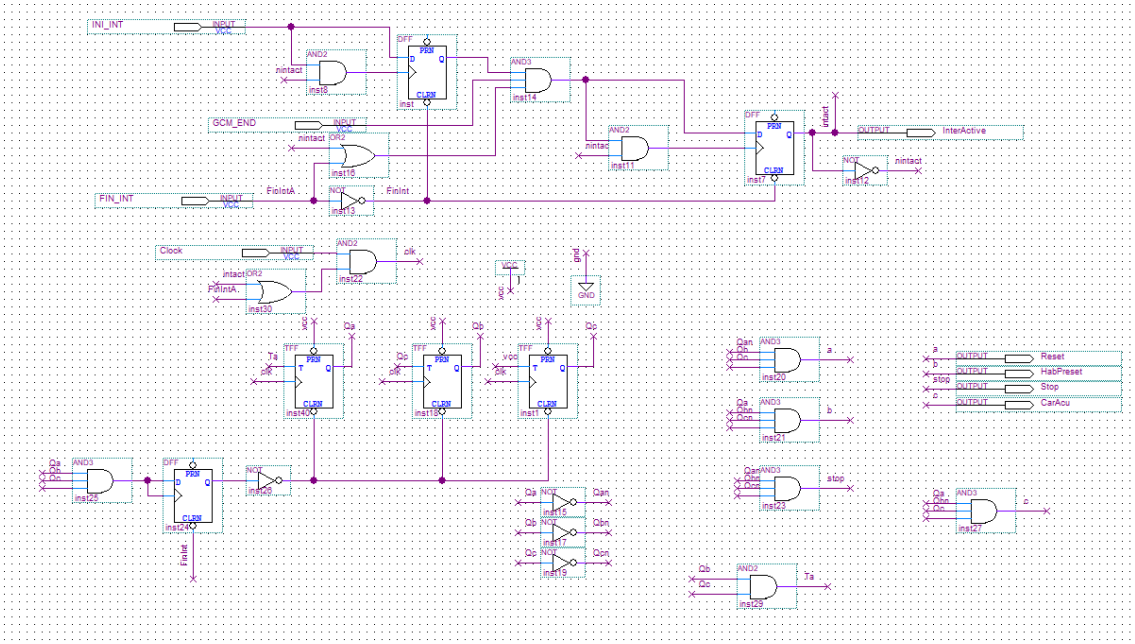

Figure 34: Interrupt Controller

For the interrupt controller, two main parts were created: combinational logic to raise the active interrupt flag and a state machine.

The interrupt flag, as seen in Figure 34, is raised when two conditions occur: an active interrupt signal arrives from the mask and the GCM is in its last signal. This last condition ensures that before activating the interrupt, the GCM finishes its current cycle if it had started one.

Then, we have a state machine responsible for stopping the GCM once the interrupt flag is raised. At the first signal, it sends reset signals to the PC and accumulator. In the second state, it loads the presets, which is the interrupt address to the PC and clears the temporary accumulator. In the last state, it sends a clock signal to the accumulator so it loads the temporary accumulator information and clears it. Once this is done, the GCM is reactivated. When an end-of-interrupt signal arrives, the process repeats, but the previous data is loaded back into the program counter and accumulator.

State Table								
Present State			Next State			Flip-Flop T Inputs		
Q_A	Q_B	Q_C	Q_A^+	Q_B^+	Q_C^+	T_A	T_B	T_C
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	0	1	1	0	0	1
0	1	1	1	0	0	1	1	1
1	0	0	1	0	1	0	0	1
1	0	1	1	1	0	0	1	1
1	1	0	1	1	1	0	0	1
1	1	1	0	0	0	1	1	1

After generating the state table for our counter, we created the Karnaugh maps:

T_A	00	01	11	10
0	0	0	1	0
1	0	0	1	0

T_B	00	01	11	10
0	0	1	1	0
1	0	1	1	0

T_C	00	01	11	10
0	1	1	1	1
1	1	1	1	1

Thus, once obtained, the functions for each T input are:

$$T_A = Q_C Q_D \tag{5}$$

$$T_B = Q_C \tag{6}$$

$$T_C = 1 \tag{7}$$

Following the signal diagram observed in Figure 35, we generated the combinational logic using AND gates to produce the signals.

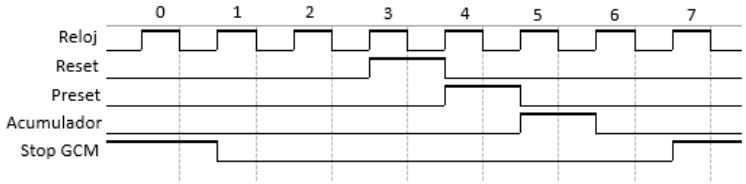

Figure 35: State Machine Signals of the Interrupt Controller

- 1. First, we stop the GCM
- 2. Then we wait two clock pulses to avoid issues
- 3. We send the reset signal
- 4. We send the preset signal
- 5. We send the load signal to the accumulator
- 6. Wait one clock pulse
- 7. We reactivate the GCM

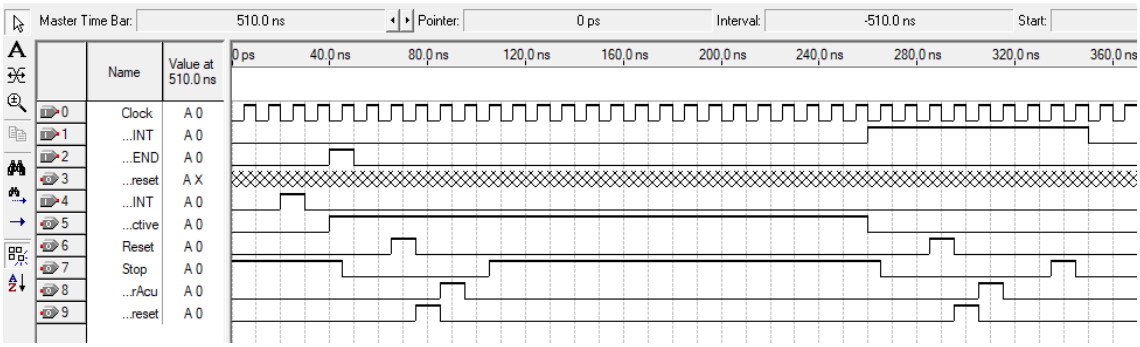

Figure 36: State Machine Signals of the Interrupt Controller

Everything described above can be seen in Figure 36, where first an interrupt signal arrives but the flag is not raised until the last GCM cycle signal arrives. Once it arrives, the interrupt is activated, the GCM

is stopped, and three pulses are generated in the following order: Reset, Preset, Load Accumulator. When these three signals finish, the GCM is reactivated and the instructions at the interrupt memory addresses are executed. Then the end-of-interrupt instruction arrives, which again generates the same three pulses while the GCM is stopped and loads the previous values to continue the main program.

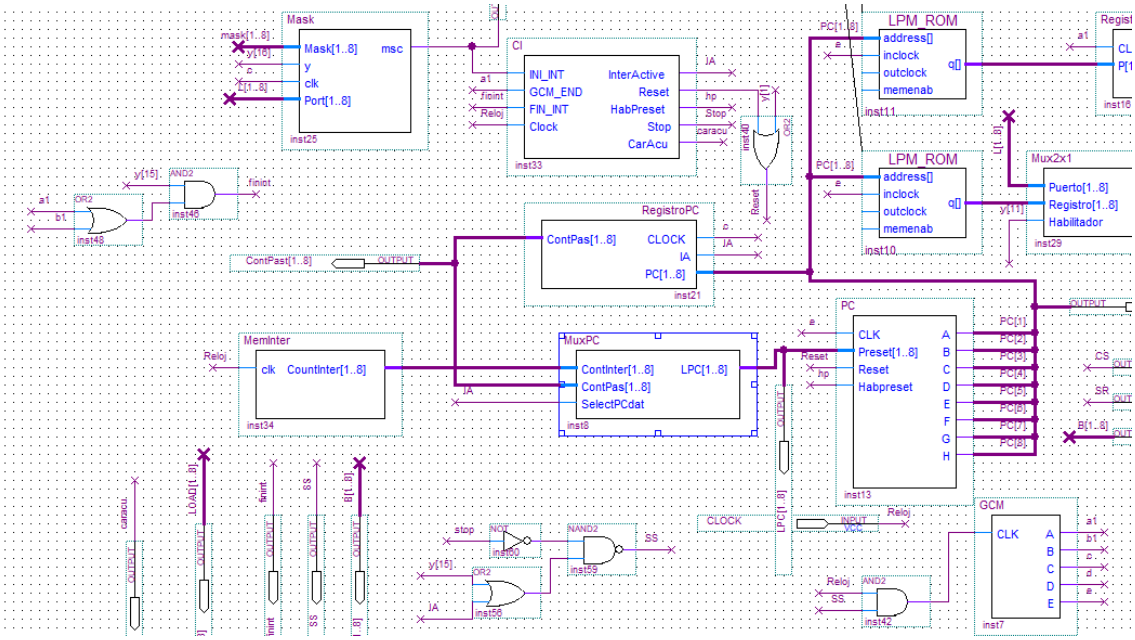

Figure 37: Mask, Interrupt Controller, Multiplexer, Registers, and PC Changes Associated with the Interrupt

As observed in Figure 37, the changes made include the two blocks: the masking and the interrupt controller, both connected to the PC and accumulator. Also, there is a fixed register holding the interrupt memory address, which in this case is 200, and another register storing the last program counter state before the interrupt activates.

This image also shows the multiplexer, which selects the data loaded to the program counter depending on whether the interrupt is active or not.

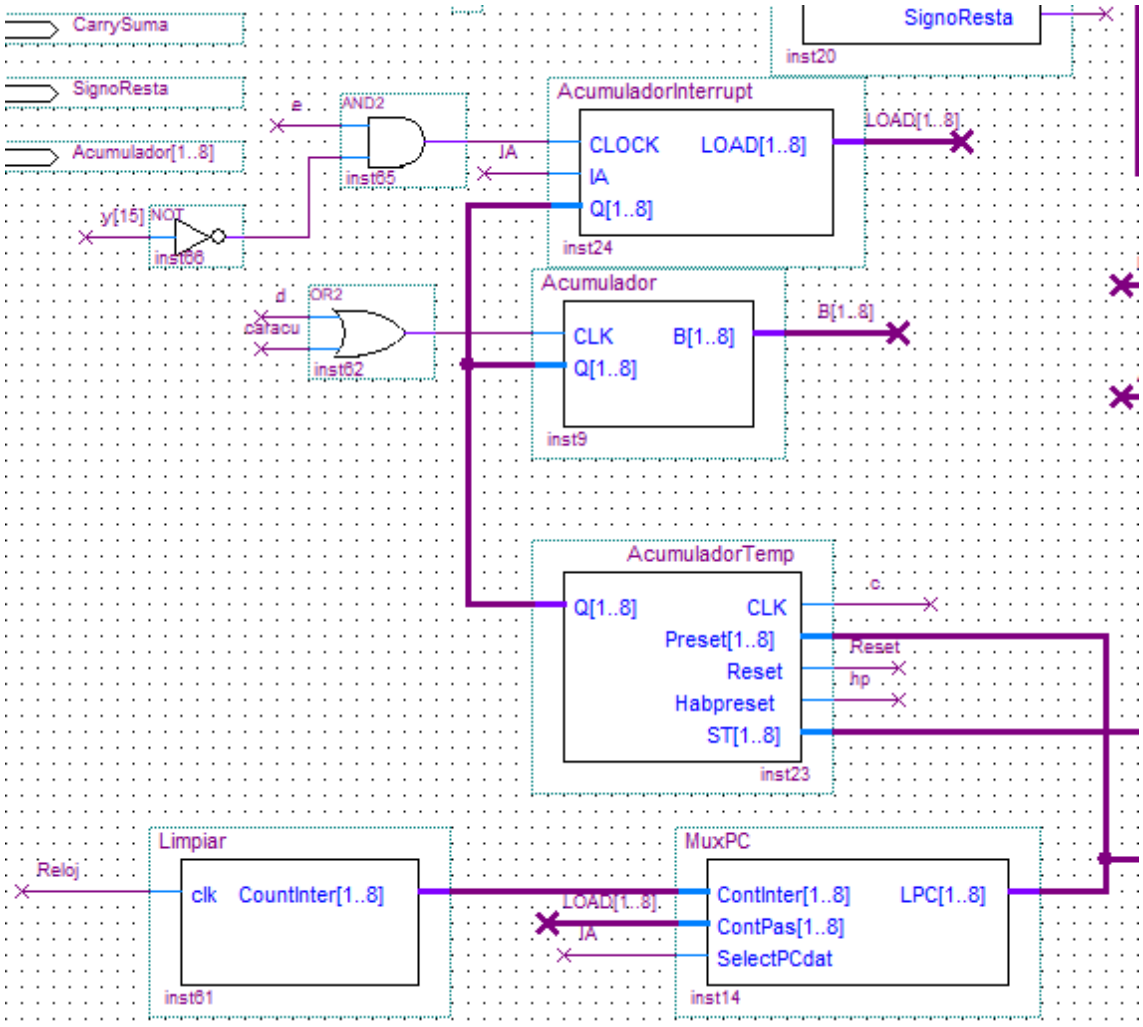

Figure 38: State Machine Signals of the Interrupt Controller

Figure 38 is similar to the previous one but applied to the accumulator, where the only change is the register loaded once the interrupt activates, which is 00000000 to clear the accumulator.

To summarize, the interrupt process is:

- 1. Configure the mask
- 2. Detect the port interrupt
- 3. Stop the GCM
- 4. Send reset and then preset signals
- 5. Load the interrupt address to the PC
- 6. Enable the GCM
- 7. Execute the interrupt
- 8. Detect end of interrupt
- 9. Stop the GCM
- 10. Send reset and preset signals

- 11. Load previous data to PC and accumulator
- 12. Enable the GCM and continue the program

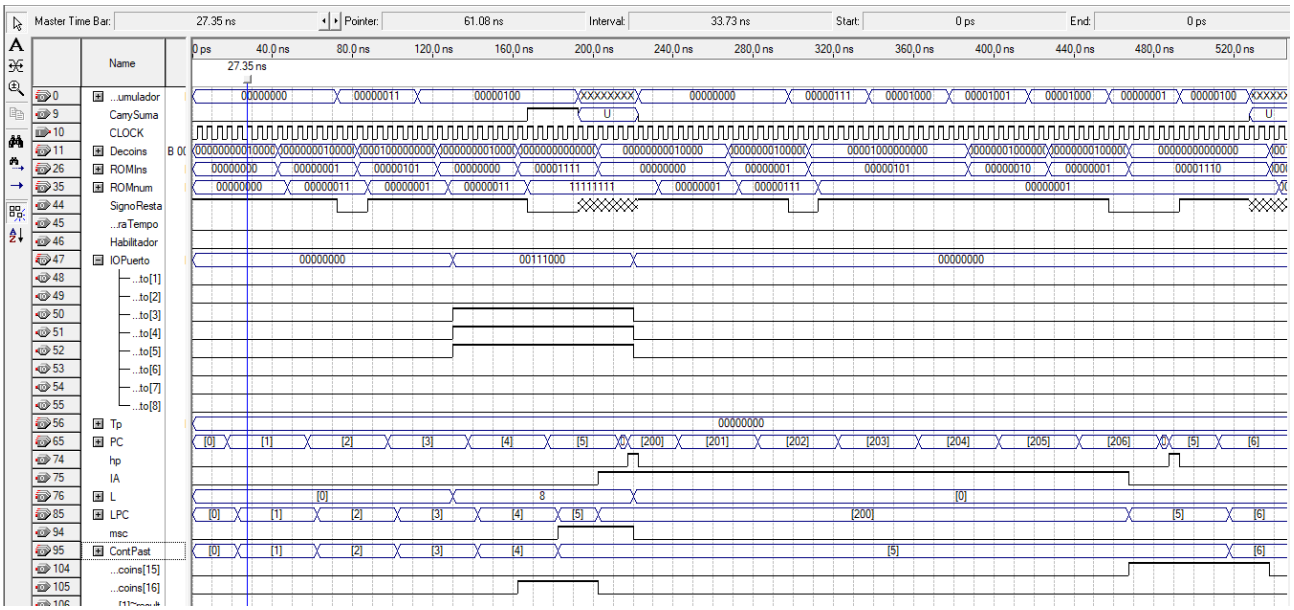

Figure 39: Complete Processor Diagram with Interrupt

In Figure 39, it can be seen how the processor is executing a main program consisting of additions, then an interrupt occurs. The PC jumps to the interrupt address 200, a number is loaded, several operations are applied, and finally the end-of-interrupt instruction is executed, which reloads the previous values into the PC and accumulator.

IV.a. Complete 8-bit Processor

For the complete processor, blocks were created from all previous diagrams and connected in order. First, the GCM was connected to the PC, registers, and accumulators. From the PC, connections go to the memories to select the data to output. The data is stored in the data and instruction registers. From the instruction register, the decoder receives which operation is desired and simultaneously the data register sends data to the ALU to perform all operations simultaneously. Then the decoder activates the tri-state of the desired operation, and this result goes to the temporary accumulator where it waits for a pulse and finally is sent to the accumulator.

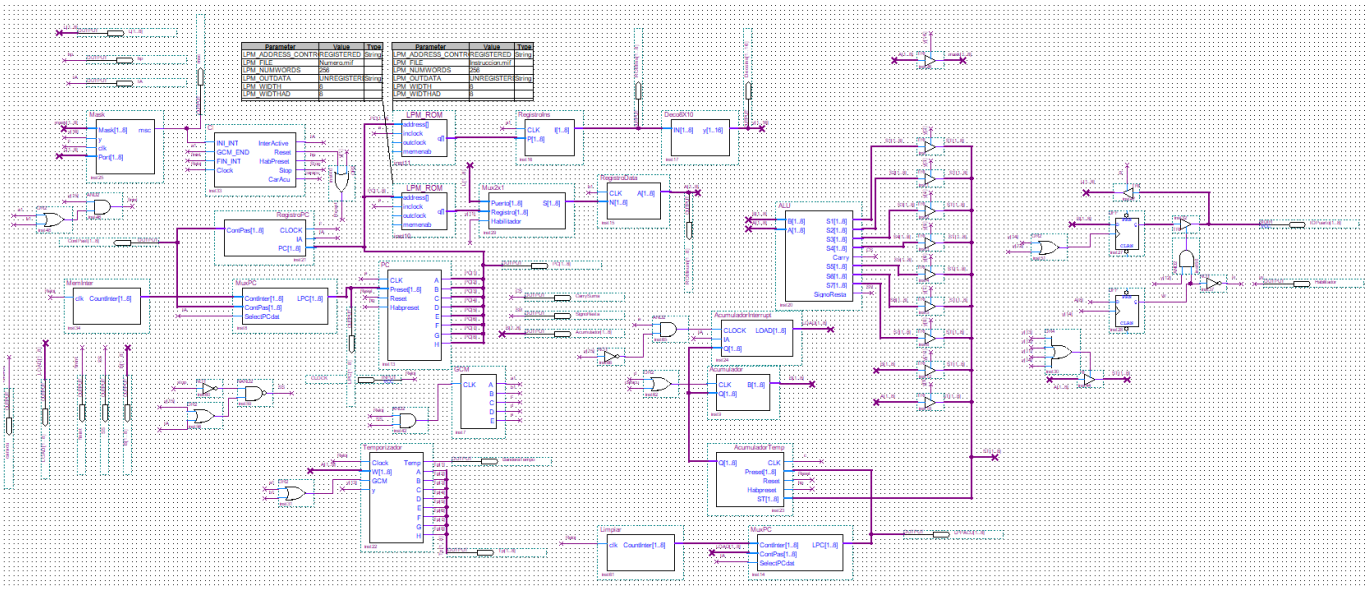

Figure 40: 8-bit Processor

V. Flowchart and Commented Code

To better understand the code, I divided it into 8 main sections. First, we have the GCM, which sends pulses in a certain order to control the processes of our machine. Then, it sends a pulse to our Instruction Register and Data Register (2) that stores data at position 0 of the memories. From there, data passes to our Arithmetic Logic Unit and 8×10 Decoder (3) that performs the operation with the sent data, and the decoder receives the desired operation output and sends it to the ALU. Next, a pulse is sent to the Program Counter (4) which counts to the next number and sends it to the ROM Memories (5) to select the output data and the next instruction (these are not loaded until another cycle passes and the register clocks are enabled). Finally, two clock pulses are sent to the Accumulators (6) . In the temporary accumulator, data is temporarily stored so as not to compromise the ALU, since the number we are operating would be loaded immediately if there were no two registers here. The last pulse from the GCM goes to the accumulator, giving our final result. After this, two more sections were added: the Input/Output Port (8) , which defines with the first instruction whether it is input or output, and then sends the instruction to read or write. When reading, it takes the value sent from the simulation and writes it to the data register; when writing, it writes the value in the accumulator and sends it through the port to the bidirectional pin. Then we have the Timer (7) , which receives a signal from the decoder to start counting and counts up to the selected value in the data memory. While counting, it has a flag raised.

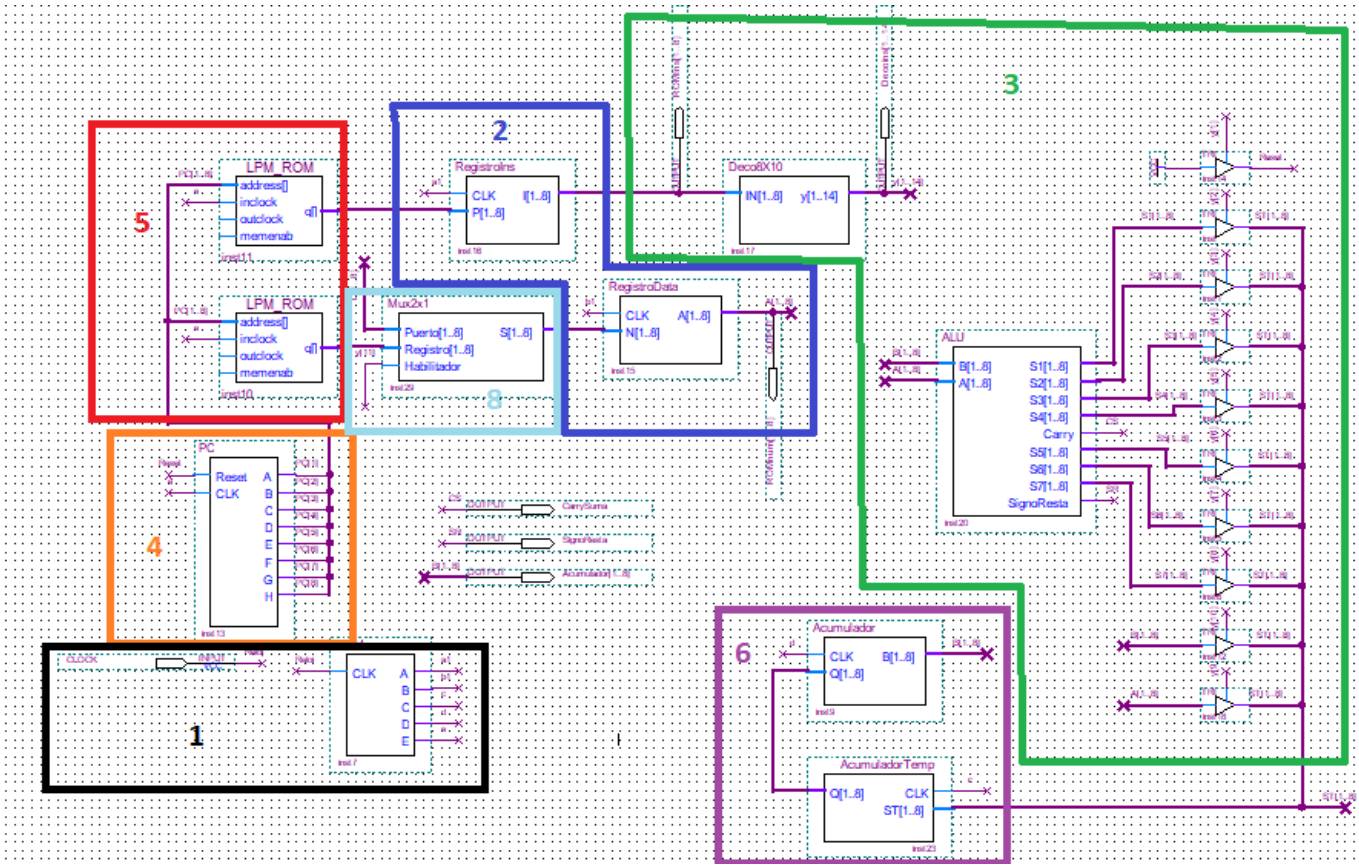

Figure 41: Timing Diagram Part 1

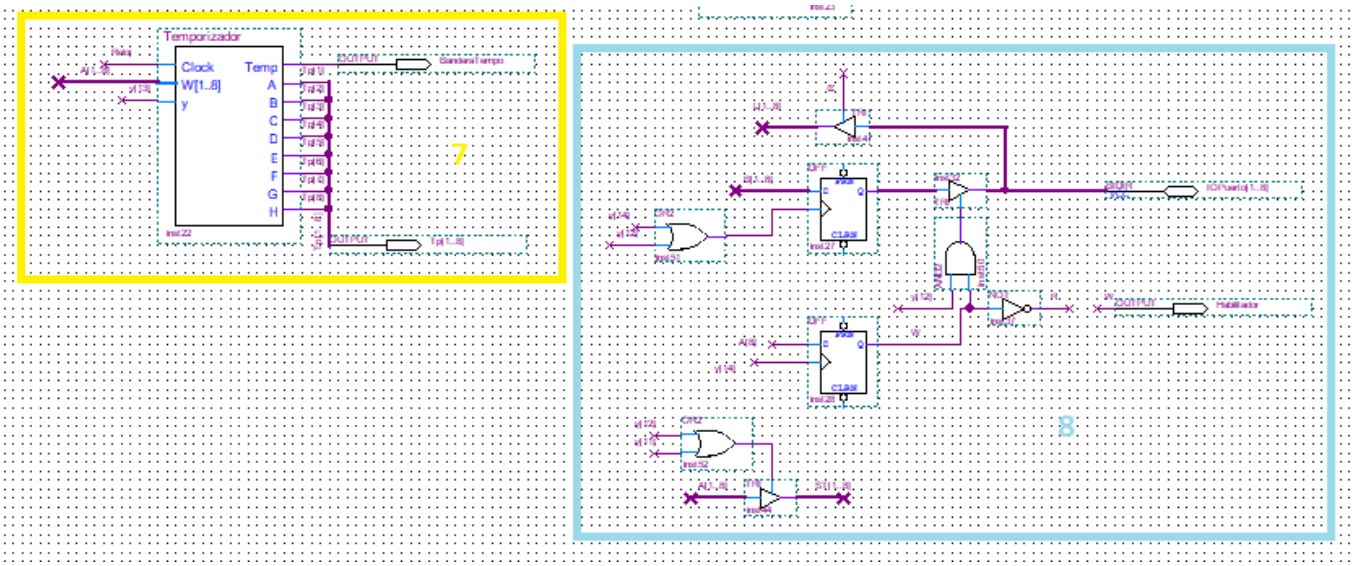

Figure 42: Timing Diagram Part 2

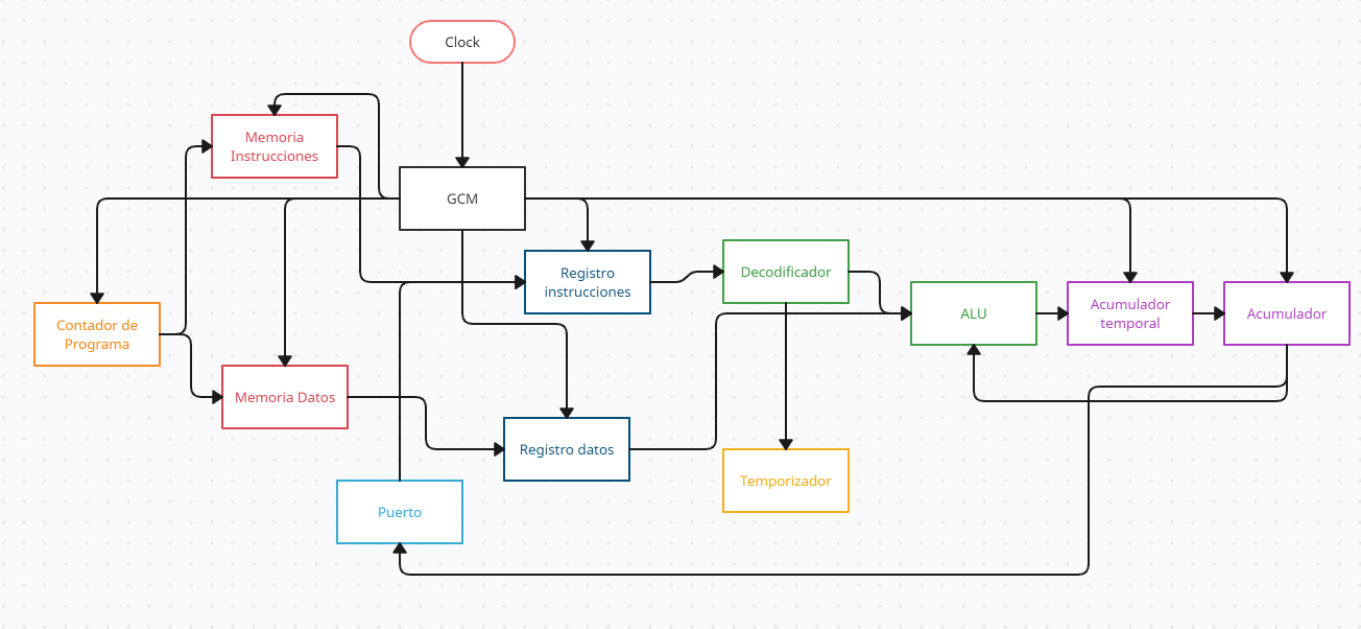

Figure 43: Flowchart

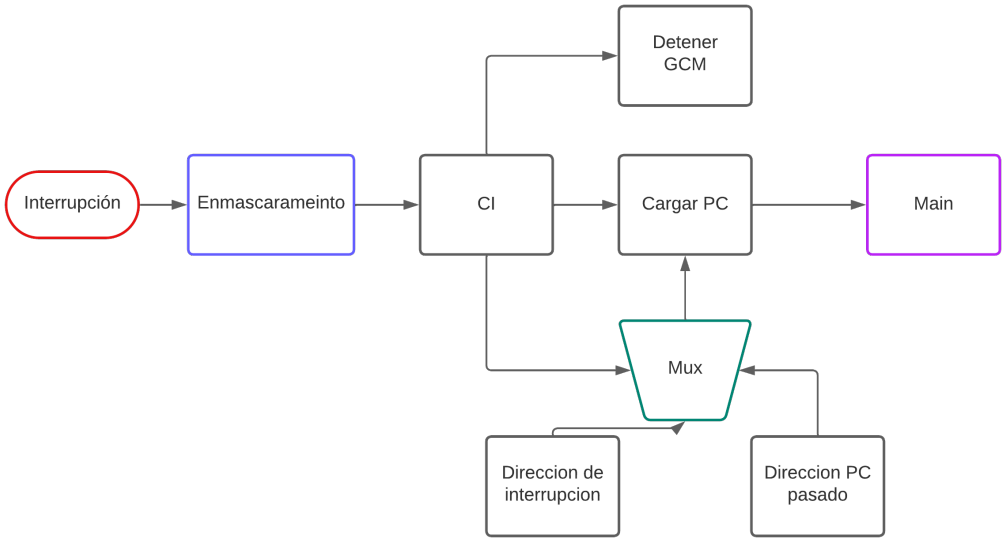

Figure 44: Interrupt Flowchart

VI. Results and Conclusions

Once we have our BDF, we proceed to create a waveform file that will simulate our processor.

The simulations were done separately to verify that the blocks worked independently and finally unified into a single simulation. This section will not show all simulations done, as it would be excessive information,

and these must work correctly for the final simulation.

First, we have the decoder, as seen it receives the instruction 00000001 whose output is 0000000010, as shown in Figure 45.

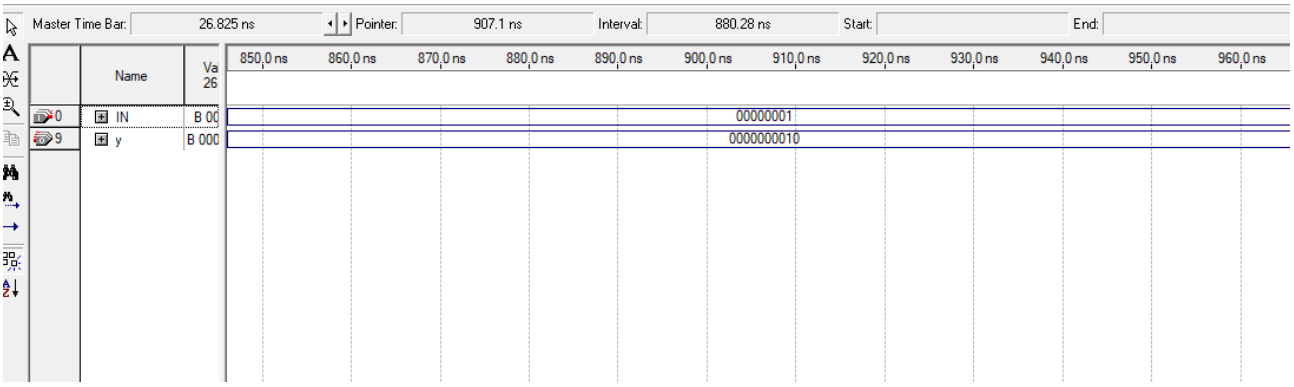

Figure 45: 8X10 decoder simulation

As shown in Table **??**, if the instruction is 00000000 the output should be the no-operation which in the decoder is 00000001, this can be verified in the simulation of Figure 46.

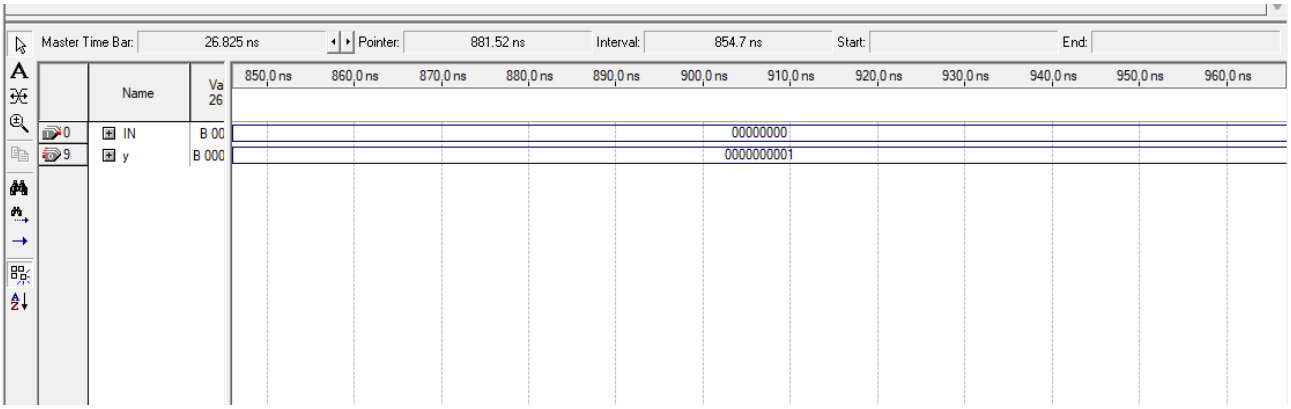

Figure 46: 8X10 decoder simulation

To verify the functionality of the AND instruction, we introduced from memory the numbers 10011011 and 11101010; the output should be 10001010, which is confirmed in Figure 47.

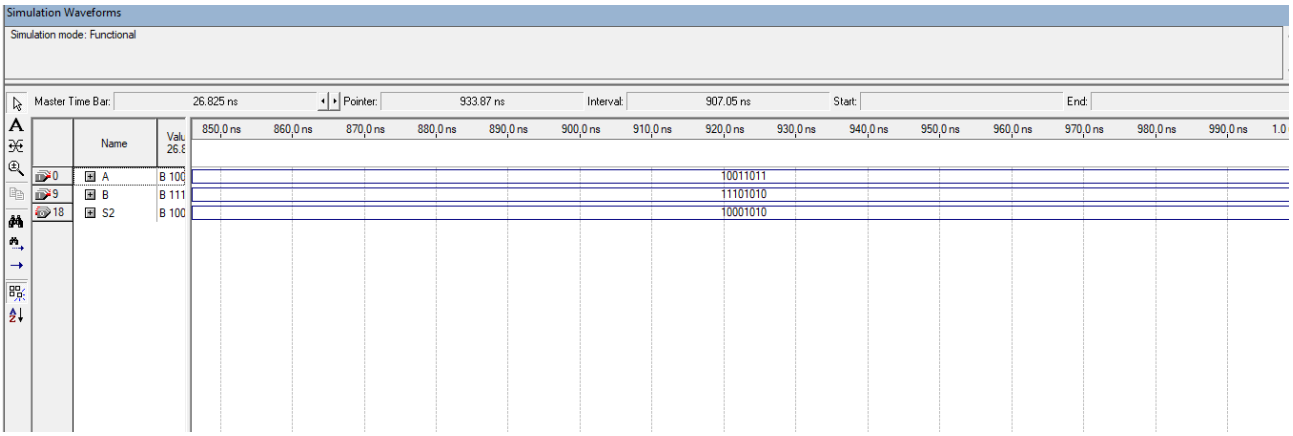

Figure 47: AND operation simulation

Then, for the OR instruction, we compared the numbers 10100111 and 00000000, which results in 10100111, confirmed in Figure 48.

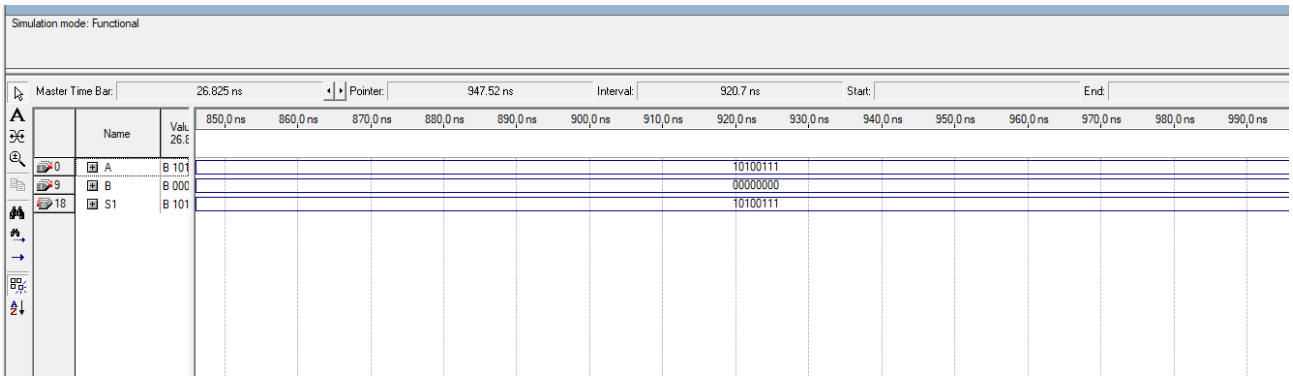

Figure 48: OR operation simulation

Next, for the XOR instruction, we compared the numbers 10110011 and 11010010, which results in 01100001, confirmed in Figure 49.

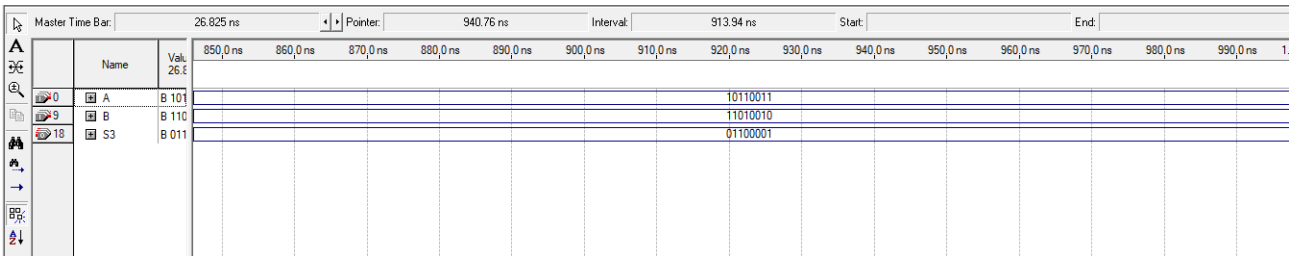

Figure 49: XOR operation simulation

For the inversion, which is the logical NOT operation, we introduced the number 10101010 and it should return 01010101, which is fulfilled as shown in Figure 50.

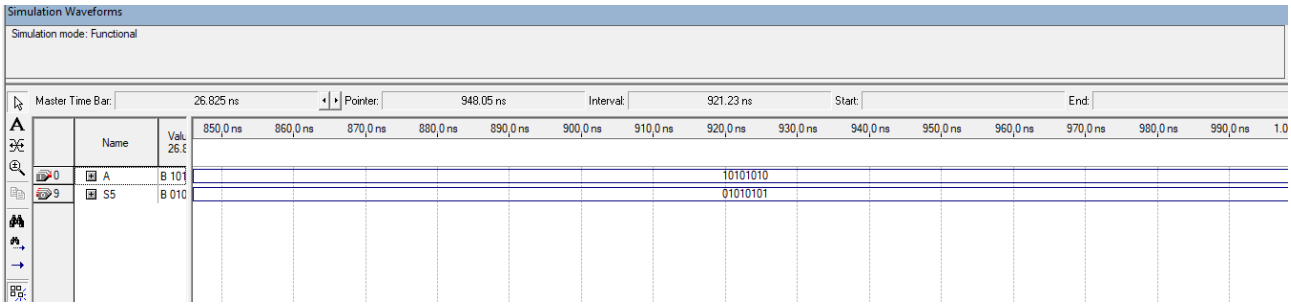

Figure 50: Inversion operation simulation

For addition, first we verified adding 0 plus 16, which in binary is 00000000 plus 00011111, resulting in 00011111 or decimal 16, confirmed in 51.

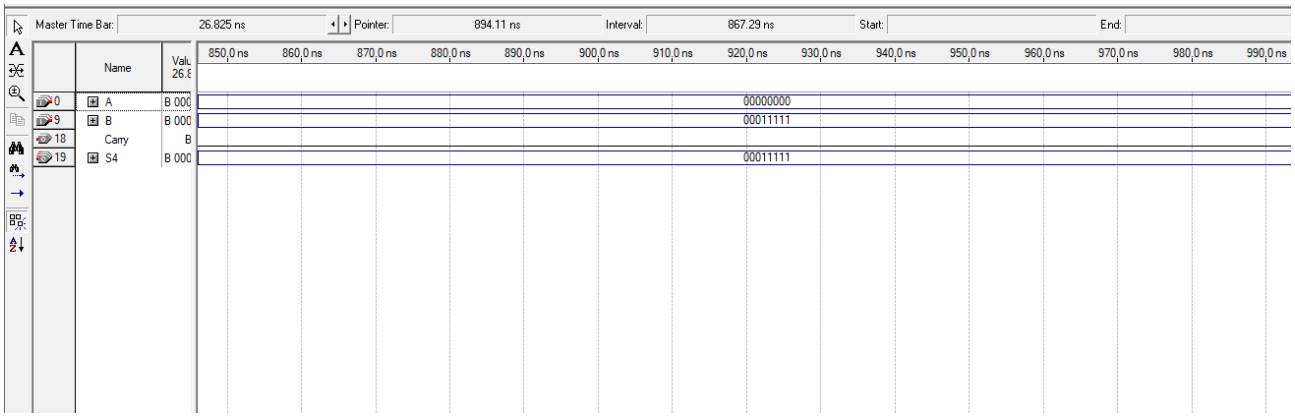

Figure 51: Addition operation simulation

For the second addition, we verified adding 31 plus 31 which should yield 62; in binary 00011111 plus 00011111 results in 00111110, confirmed in 52.

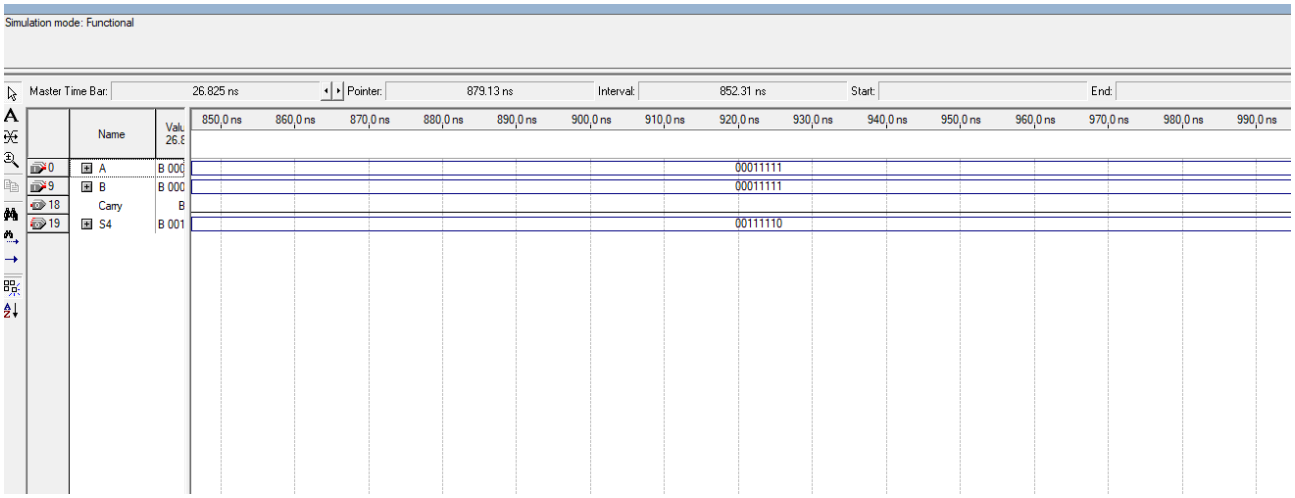

Figure 52: Addition operation simulation

For the third addition, we verified the carry function adding 255 plus 31 which should yield 286, but since it exceeds the representable size, the carry is activated and the output should be 30; in binary 11111111 plus

00011111 results in 00011110 with carry active, confirmed in 53.

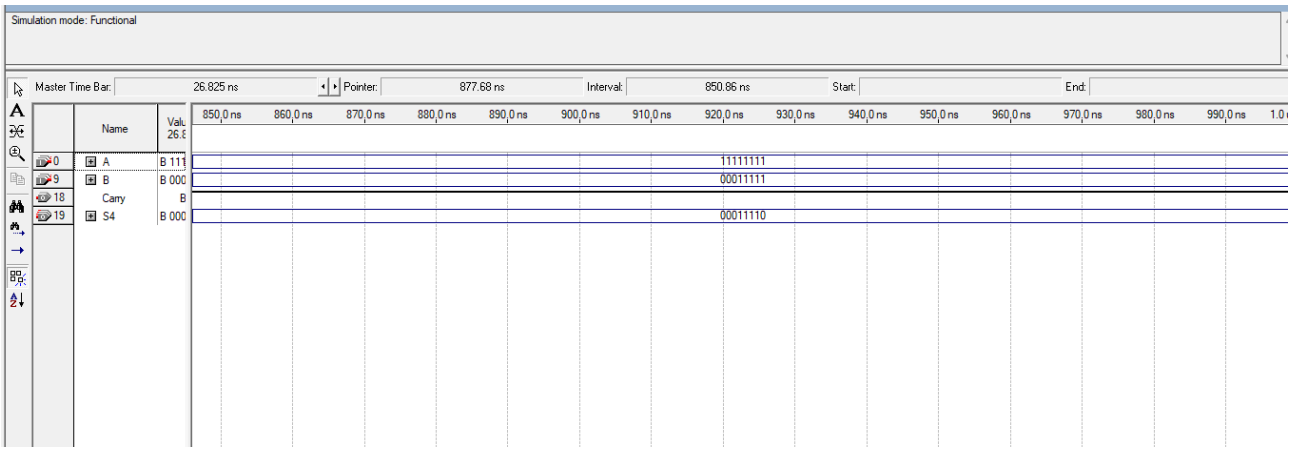

Figure 53: Addition operation simulation with carry active

In Figure 54, we can see the operation of the program counter, which counts from 0 to 255.

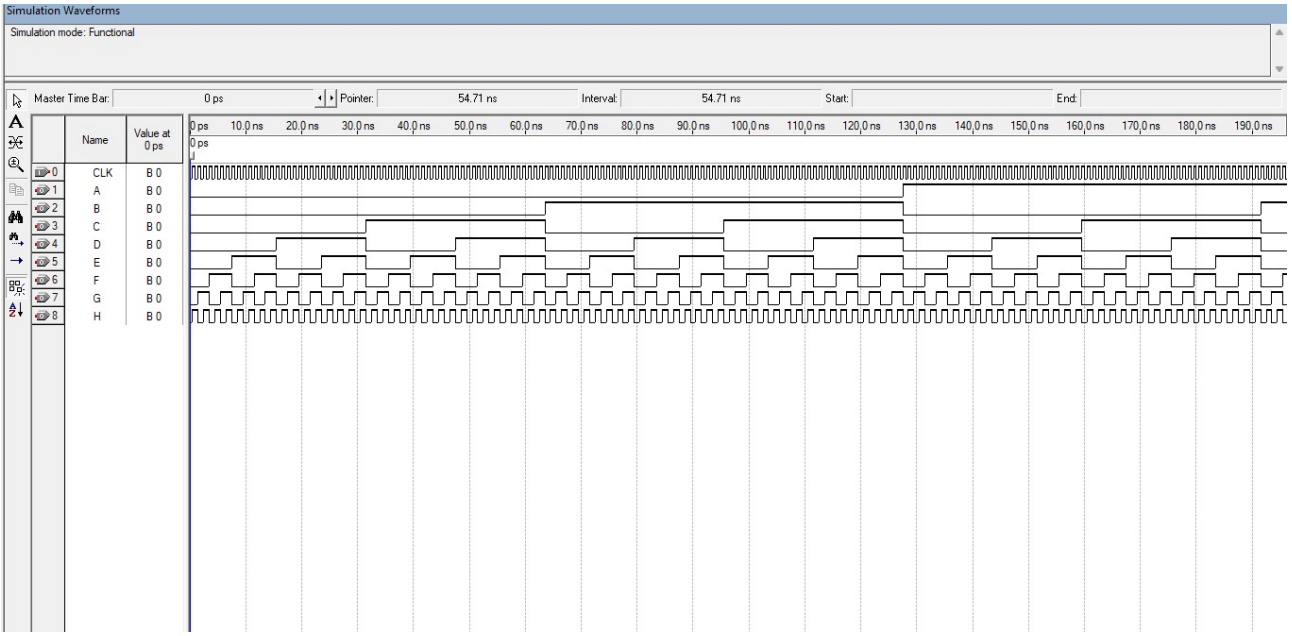

Figure 54: Program counter simulation

Addr	+000	+001	+010	+011	+100	+101	+110	+111
00000000	00000011	00000111	00000001	00000010	00000001	00000000	00000000	00000000
00001000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00010000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00011000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00100000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00101000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00110000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00111000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
01000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
01001000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
01010000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
01011000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
01100000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
01101000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
01110000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
01111000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
10000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
10001000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
10010000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
10011000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
10100000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
10101000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
10110000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
10111000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
11000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
11001000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
11010000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
11011000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
11100000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
11101000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
11110000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
11111000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000

((a)) Data .mif file

Addr	+000	+001	+010	+011	+100	+101	+110	+111
00000000	00000001	00000101	00000101	00000011	00000010	00001001	00000000	00000000
00001000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00010000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00011000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00100000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00101000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00110000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00111000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
01000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
01001000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
01010000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
01011000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
01100000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
01101000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
01110000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
01111000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
10000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
10001000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
10010000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
10011000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
10100000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
10101000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
10110000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
10111000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
11000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
11001000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
11010000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
11011000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
11100000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
11101000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
11110000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
11111000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000

((b)) Instruction .mif file

Figure 55: Set of data and instructions

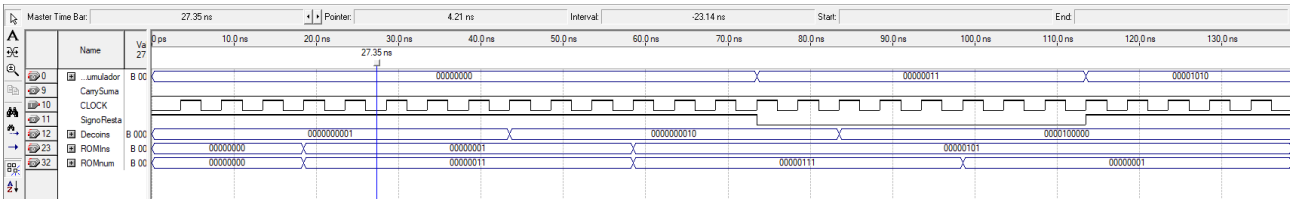

Figure 56: 8-bit processor simulation

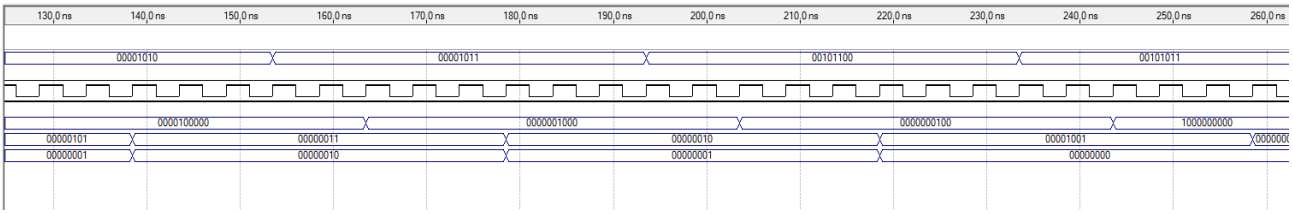

Figure 57: 8-bit processor simulation

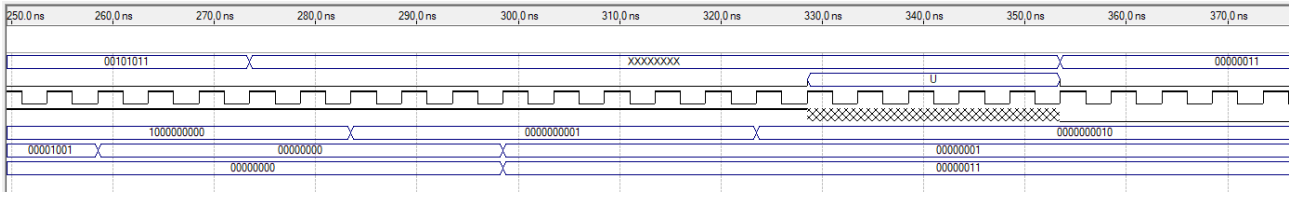

Figure 58: 8-bit processor simulation

For our final simulation, a total of 6 different instructions were used. The first was load, which loads the first number from memory 00000011 to the accumulator; then the second instruction was to add the previous number with 00000111 which, as seen in 56, results in 00001010; next, as the third instruction, 00000001 was

added to the previous number resulting in 00001011; the fourth instruction was a shift by 2, so as seen in 57 the result is 00101100; to this number, as the 5th instruction, we subtract 1 resulting in 00101011, as seen in the same figure; finally, the reset instruction was used which restarts the counter, and as observed in Figure 58 the program restarts executing the instructions again.

For the second simulation, the new instructions added were tested, which are the ports and timers.

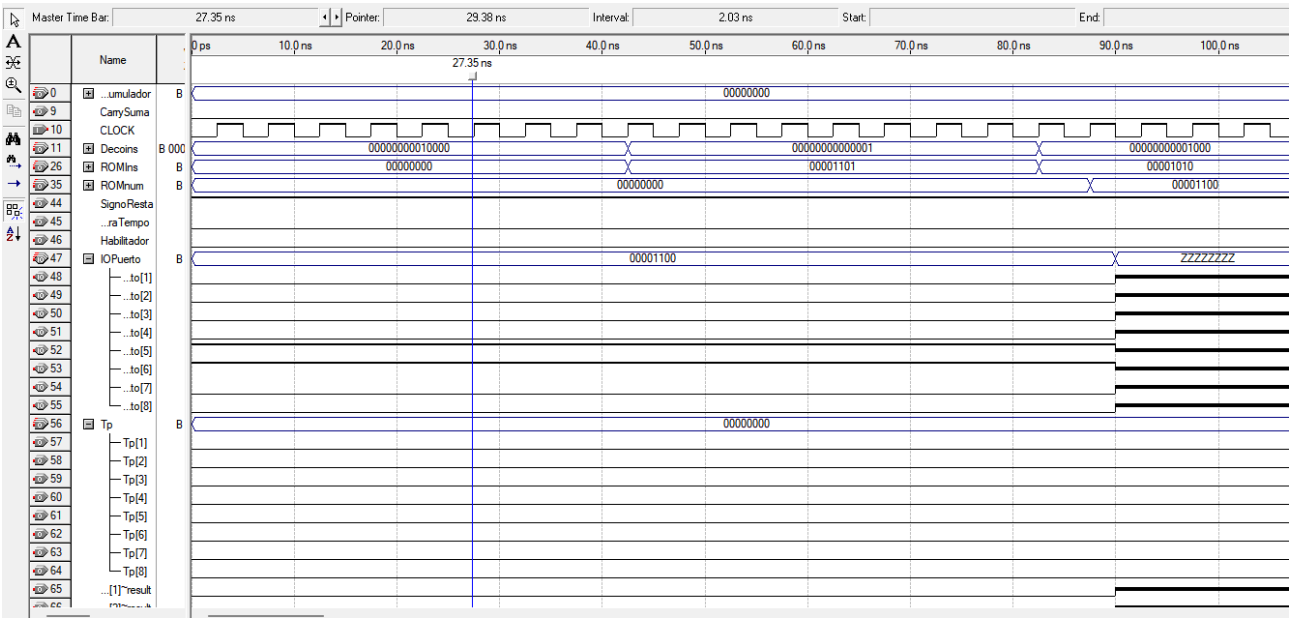

Figure 59: 8-bit processor simulation with added instructions

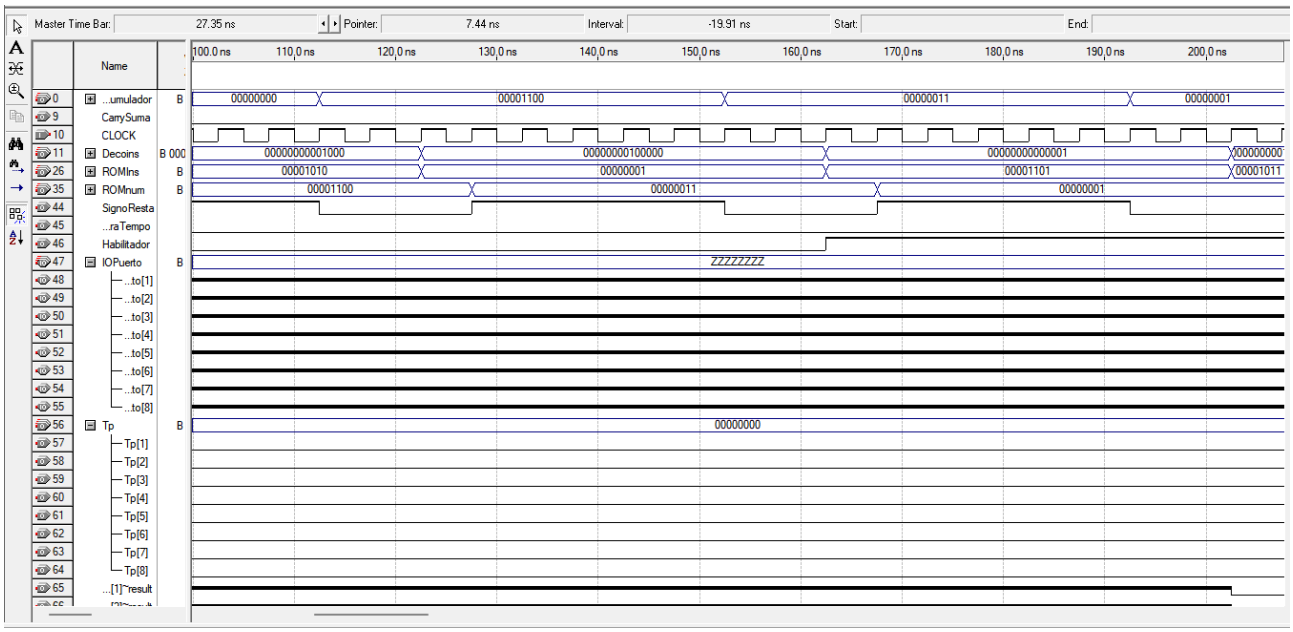

Figure 60: 8-bit processor simulation with added instructions

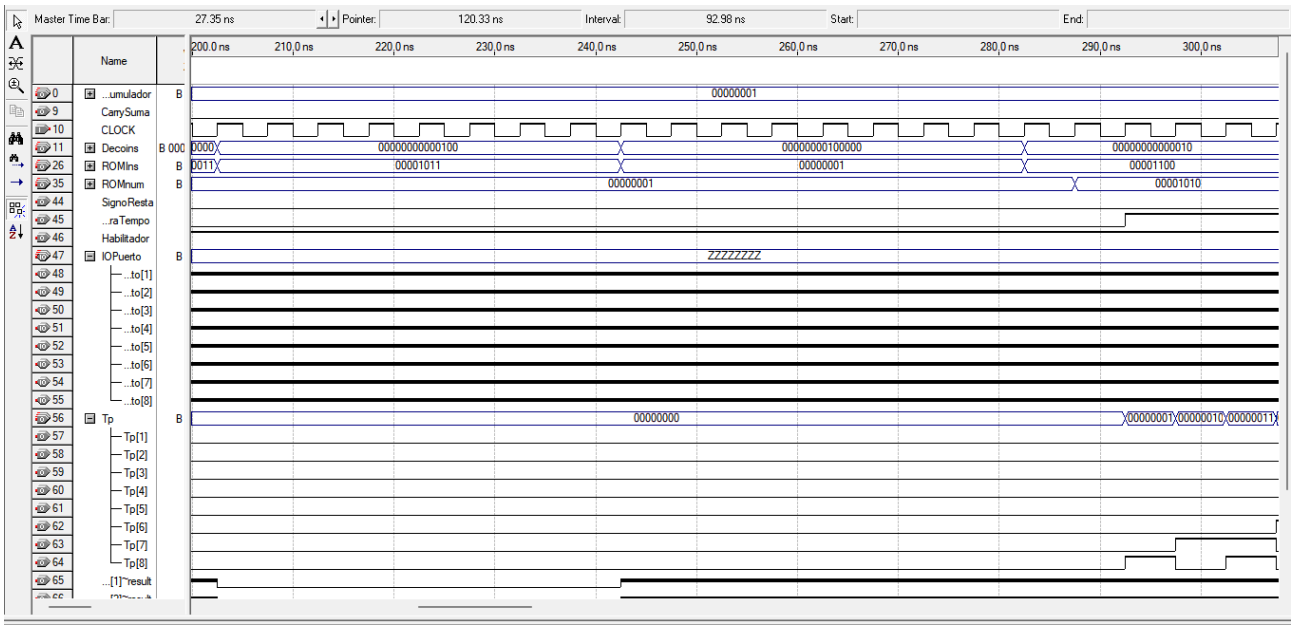

Figure 61: 8-bit processor simulation with added instructions

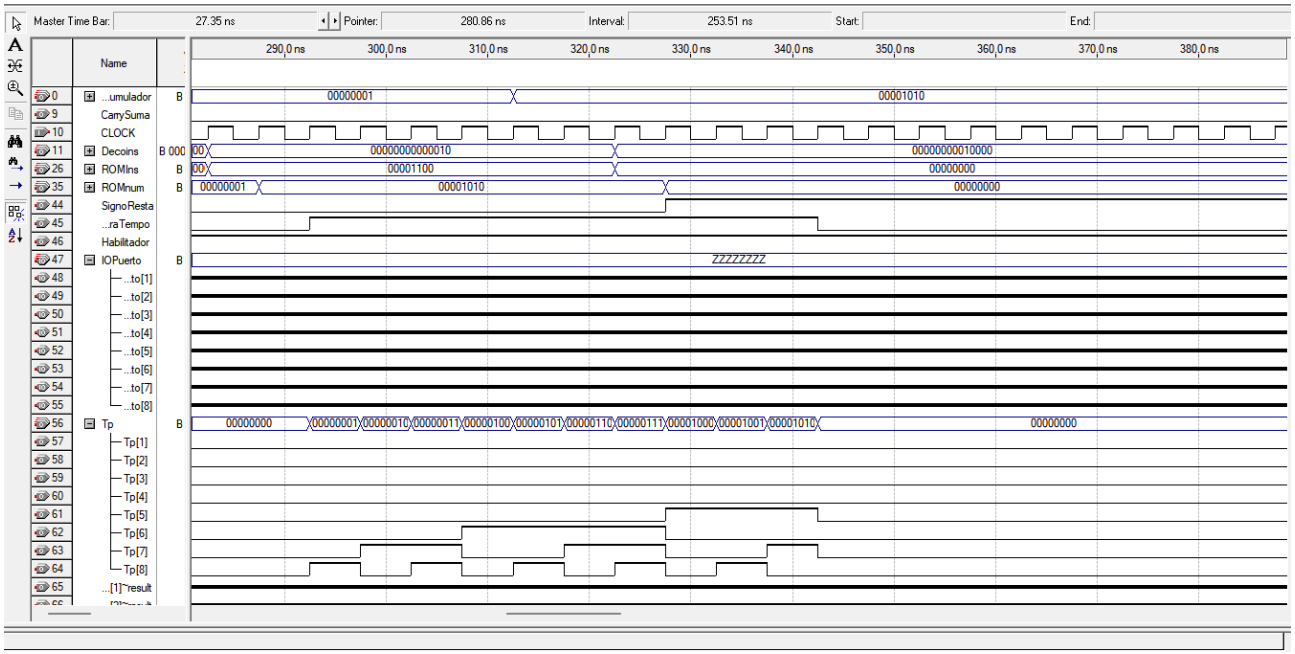

Figure 62: 8-bit processor simulation with added instructions

In conclusion, designing an 8-bit processor requires a deep understanding of processor architecture as well as skills in hardware description languages, circuit design and simulation, and problem solving. Through this project, valuable skills can be acquired in digital circuit design, understanding the internal workings of processors, and troubleshooting. This helped us understand more effectively the topics covered in class regarding how a microprocessor and its constituent machines operate. Ports were added to allow peripheral input as well as a timer to enable interrupts in the future.

Referencias

[1] https://electronicacompleta.com/contadores-digitales/

[2] http://hyperphysics.phy-astr.gsu.edu/hbasees/Electronic/jkflipflop.html

[3] Valvano, J. W. (2013). Embedded Systems. Jonathan W. Valvano.

[4] Bai, Y. (2015). Practical microcontroller engineering with ARM technology. John Wiley Sons.

[5] https://www.paginaspersonales.unam.mx/files/5190/Asignaturas/1415/Archivo3.2375.pdf

[6] Saul de la Rosa. (2019) Interrupciones. Pag 11. Recuperado de presentación vista en clase.