Project I: Adder/Subtractor

Computer Organization CSC 34300 - EF

By: Sebastian Grygorczuk

Tabel of Content:

Objective:

The objective of this lab is to create a “smart” 4 bit adder/subtractor circuit which would use 2:1 multiplex for operation code, use LEDs for zero, negative, overflow and carry out flags, LEDs for the binary representation, and two seven segment displays one for decimal representation of zero to nine and the second to display zero, one, plus and minus, allowing us to have both signed and unsigned outputs. This will be accomplished through building smaller circuits such as Half-Wave Adder, Full-Adder, Multiplex and ect., and verify them by comparing them to the ones found in the LPM libraries with purpose of using these smaller circuit to construct the desired circuit meanwhile proving the understanding of base concepts such as binary addition/subtraction, two’s complement, designing and implementing of the system using Block Diagram/Schematic File and VHDL.

Overview:

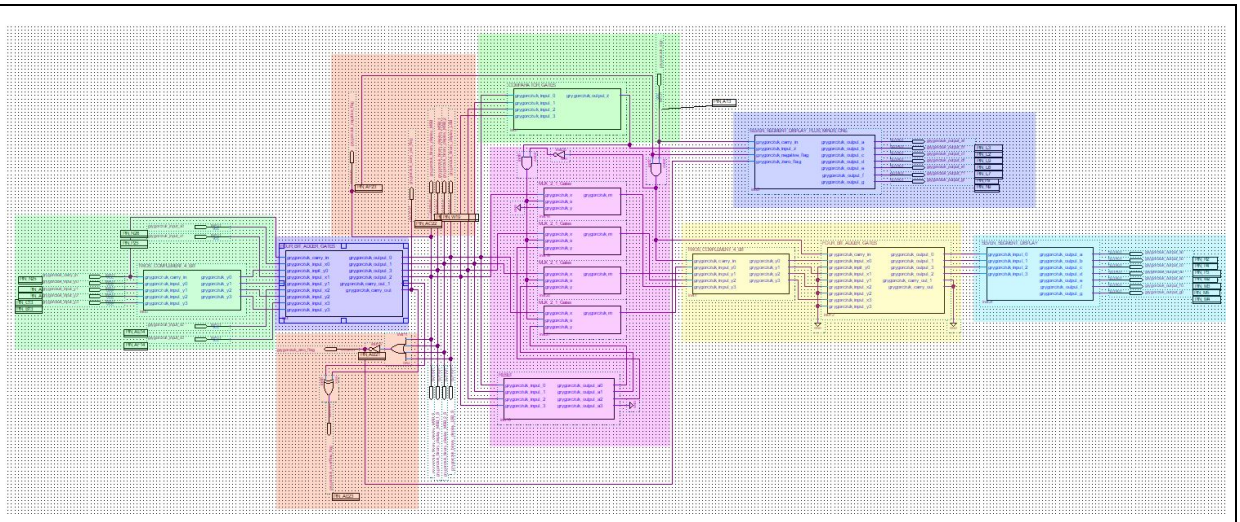

Figure 1. Complete System

Above is a full schematic of the system, I’ve color the individual subsystem in a manner that I believe would be easier to inspect them. In the following parts I will explain the function and design of each highlighted area. Below is the VHDL code that works same as the design in Figure 1.

```
library ieee;
use ieee.std_logic_1164.all;

entity PROJECT_ADD_SUB is
```

```
	port(
		grygorczuk_y0, grygorczuk_y1, grygorczuk_y2, grygorczuk_y3 :in std_logic;
		grygorczuk_x0, grygorczuk_x1, grygorczuk_x2, grygorczuk_x3 :in std_logic;
		grygorczuk_carry_in, grygorczuk_sign :in std_logic;
		grygorczuk_flag_carry_out,grygorczuk_flag_zero, grygorczuk_flag_overflow,
grygorczuk_negative : out std_logic;
		grygorczuk_output_0, grygorczuk_output_1, grygorczuk_output_2,
grygorczuk_output_3: out std_logic;
		grygorczuk_a2, grygorczuk_b2, grygorczuk_c2, grygorczuk_d2,
grygorczuk_e2, grygorczuk_f2, grygorczuk_g2 : out std_logic;
		grygorczuk_a1, grygorczuk_b1, grygorczuk_c1, grygorczuk_d1,
grygorczuk_e1, grygorczuk_f1, grygorczuk_g1 : out std_logic
	);
end PROJECT_ADD_SUB;

architecture PROJECT_ADD_SUB_LOGIC of PROJECT_ADD_SUB is
signal Y_OUTPUT_0,Y_OUTPUT_1,Y_OUTPUT_2,Y_OUTPUT_3, CARRY_OUT_2,
CARRY_OUT_3, OUTPUT_0,OUTPUT_1,OUTPUT_2,OUTPUT_3, NEG, ZERO, Z,
RESET_S, A_OUTPUT_0,A_OUTPUT_1,A_OUTPUT_2,A_OUTPUT_3,M_OUTPUT_0,
M_OUTPUT_1, M_OUTPUT_2,M_OUTPUT_3, C_OUTPUT_0, C_OUTPUT_1,
C_OUTPUT_2, C_OUTPUT_3, CARRY_OUT_2_2, CARRY_OUT_3_2,  O_OUTPUT_0,
O_OUTPUT_1, O_OUTPUT_2, O_OUTPUT_3: std_logic;

component TWOS_COMPLEMENT is
	port(
		grygorczuk_y0, grygorczuk_y1, grygorczuk_y2, grygorczuk_y3,
grygorczuk_s_1 : in std_logic;
		grygorczuk_m0,grygorczuk_m1,grygorczuk_m2,grygorczuk_m3 : out std_logic
	);
end component;

component FOUR_BIT_ADDER is
	port(
		grygorczuk_carry_in_2 : in std_logic;
		grygorczuk_carry_x0, grygorczuk_carry_x1, grygorczuk_carry_x2,
grygorczuk_carry_x3: in std_logic;
		grygorczuk_carry_y0, grygorczuk_carry_y1, grygorczuk_carry_y2,
grygorczuk_carry_y3: in std_logic;
		grygorczuk_carry_out_2, grygorczuk_carry_out_3, grygorczuk_o0,
grygorczuk_o1, grygorczuk_o2, grygorczuk_o3: out std_logic
	);
end component;

component COMPARATOR is
	port(
		grygorczuk_o0, grygorczuk_o1, grygorczuk_o2, grygorczuk_o3 : in std_logic;
		grygorczuk_z : out std_logic
```

```
	);
end component;

component SEVEN_SEGMENT_D2 is
	port(
		grygorczuk_sign, grygorczuk_input_z, grygorczuk_negative_flag,
grygorczuk_zero_flag : in std_logic;
		grygorczuk_output_a, grygorczuk_output_b, grygorczuk_output_c,
grygorczuk_output_d, grygorczuk_output_e, grygorczuk_output_f, grygorczuk_output_g : out
std_logic
	);
end component;

component RESET_1 is
	port(
		grygorczuk_input_0, grygorczuk_input_1, grygorczuk_input_2,
grygorczuk_input_3 : in std_logic;
		grygorczuk_output_0, grygorczuk_output_1, grygorczuk_output_2,
grygorczuk_output_3: out std_logic
	);
end component;

component MUX is
	port(
		grygorczuk_x, grygorczuk_y, grygorczuk_s : in std_logic;
		grygorczuk_m : out std_logic
	);
end component;

component SEVEN_SEGMENT_D1 is
	port(
		grygorczuk_input_3, grygorczuk_input_2, grygorczuk_input_1,
grygorczuk_input_0 : in std_logic;
		grygorczuk_output_a, grygorczuk_output_b, grygorczuk_output_c,
grygorczuk_output_d, grygorczuk_output_e, grygorczuk_output_f, grygorczuk_output_g : out
std_logic
	);
end component;

begin

TWOS_COMPLEMENT_1: TWOS_COMPLEMENT port map(grygorczuk_y0, grygorczuk_y1,
grygorczuk_y2, grygorczuk_y3, grygorczuk_carry_in,
Y_OUTPUT_0,Y_OUTPUT_1,Y_OUTPUT_2,Y_OUTPUT_3);
FOUR_BIT_ADDER_1 : FOUR_BIT_ADDER port map(grygorczuk_carry_in, grygorczuk_x0,
grygorczuk_x1, grygorczuk_x2,
grygorczuk_x3,Y_OUTPUT_0,Y_OUTPUT_1,Y_OUTPUT_2,Y_OUTPUT_3, CARRY_OUT_2,
```

```
CARRY_OUT_3, OUTPUT_0,OUTPUT_1,OUTPUT_2,OUTPUT_3);

grygorczuk_flag_carry_out <= CARRY_OUT_3;
grygorczuk_flag_zero <=  not (OUTPUT_0 or OUTPUT_1 or OUTPUT_2 or OUTPUT_3);
grygorczuk_flag_overflow <= CARRY_OUT_2 xor CARRY_OUT_3;
grygorczuk_negative <= OUTPUT_3;

NEG <= OUTPUT_3;
ZERO <=  not (OUTPUT_0 or OUTPUT_1 or OUTPUT_2 or OUTPUT_3);

grygorczuk_output_0 <= OUTPUT_0;
grygorczuk_output_1 <= OUTPUT_1;
grygorczuk_output_2 <= OUTPUT_2;
grygorczuk_output_3 <= OUTPUT_3;

COMPARATOR_1: COMPARATOR port map(OUTPUT_0, OUTPUT_1, OUTPUT_2,
OUTPUT_3, Z);
SEVEN_SEGMENT_D2_1: SEVEN_SEGMENT_D2 port map(grygorczuk_sign, Z, NEG,
ZERO, grygorczuk_a2, grygorczuk_b2, grygorczuk_c2, grygorczuk_d2, grygorczuk_e2,
grygorczuk_f2, grygorczuk_g2);

RESET_S <=  not (grygorczuk_sign and NEG) and Z;
RESET_1_1: RESET_1 port map(OUTPUT_0,OUTPUT_1,OUTPUT_2,OUTPUT_3,
A_OUTPUT_0,A_OUTPUT_1,A_OUTPUT_2,A_OUTPUT_3);

MUX_0: MUX port map(OUTPUT_0,A_OUTPUT_0, RESET_S, M_OUTPUT_0);
MUX_1: MUX port map(OUTPUT_1,A_OUTPUT_1,  RESET_S, M_OUTPUT_1);
MUX_2: MUX port map(OUTPUT_2,A_OUTPUT_2, RESET_S, M_OUTPUT_2);
MUX_3: MUX port map(OUTPUT_3,'0',RESET_S, M_OUTPUT_3);

TWOS_COMPLEMENT_2: TWOS_COMPLEMENT port map(M_OUTPUT_0, M_OUTPUT_1,
M_OUTPUT_2, M_OUTPUT_3, grygorczuk_sign and NEG, C_OUTPUT_0
,C_OUTPUT_1,C_OUTPUT_2,C_OUTPUT_3);
FOUR_BIT_ADDER_2 : FOUR_BIT_ADDER port map(grygorczuk_sign and
NEG,C_OUTPUT_0
,C_OUTPUT_1,C_OUTPUT_2,C_OUTPUT_3,'0','0','0','0',CARRY_OUT_2_2,
CARRY_OUT_3_2, O_OUTPUT_0, O_OUTPUT_1, O_OUTPUT_2, O_OUTPUT_3);
SEVEN_SEGMENT_D1_1: SEVEN_SEGMENT_D1 port map(O_OUTPUT_0,
O_OUTPUT_1, O_OUTPUT_2, O_OUTPUT_3,grygorczuk_a1, grygorczuk_b1,
grygorczuk_c1, grygorczuk_d1, grygorczuk_e1, grygorczuk_f1, grygorczuk_g1);

end PROJECT_ADD_SUB_LOGIC;
```

Code 1. Project Add Sub

Part 1: Inputs and Two's Complement

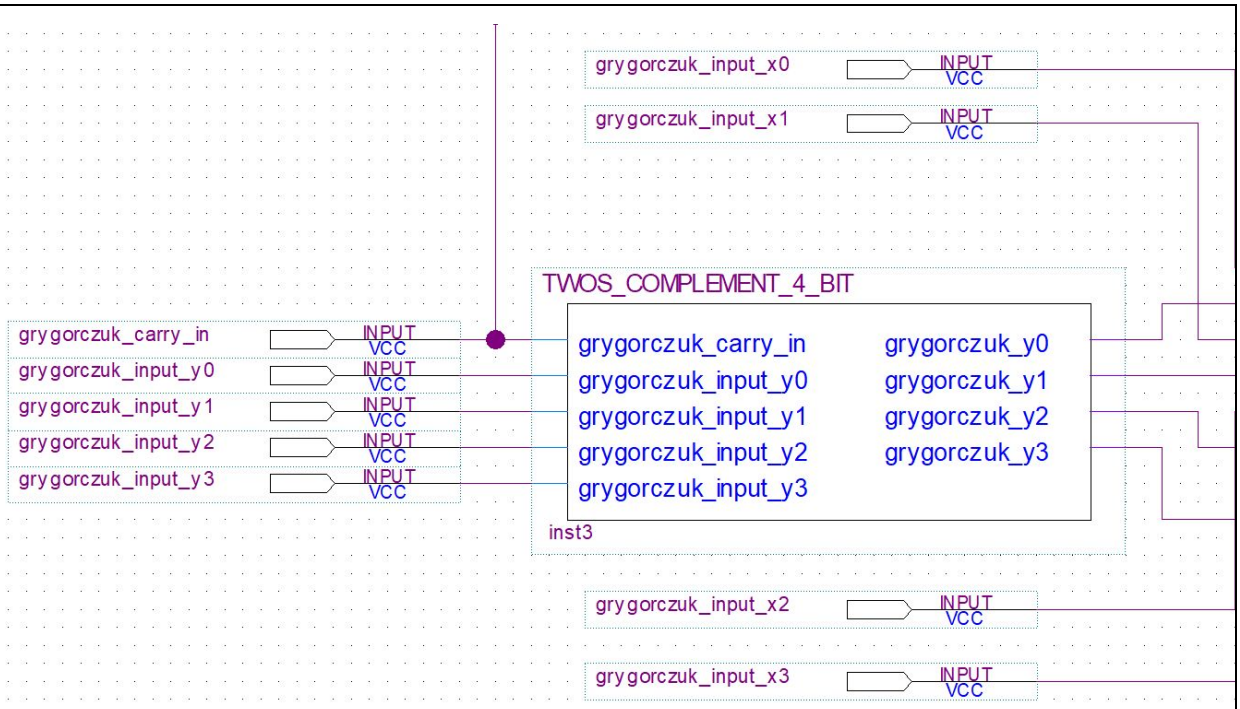

Figure 2.

Above we see the overview of the inputs and the two's complement used to invert an input to simulate it being subtracted. There are nine given inputs x0-x3, y0-y3 and carry in with x0 and y0 being the least significant bits and x3 and y3 being most significant bits. Such that

X3 X2 X1 X0
Y3 Y2 Y1 Y0
0 1 0 0

Is equivalent to 4. In the given set up the x0-x3 inputs are directly used in the system while y0-y3 go through two's complement based on the state of carry in. If carry in is 0 then the operation is read as "+" and y0-y3 pass without a change meanwhile if carry in is 1 then the operation is read as "-" and y0-y3 is inverted. Thus given

Input: Y3 Y2 Y1 Y0 CI
0 1 0 0 0

Output: Y3 Y2 Y1 Y0

0 1 0 0

Input: Y3 Y2 Y1 Y0 CI
0 1 0 0 1

Output: Y3 Y2 Y1 Y0
1 0 1 1

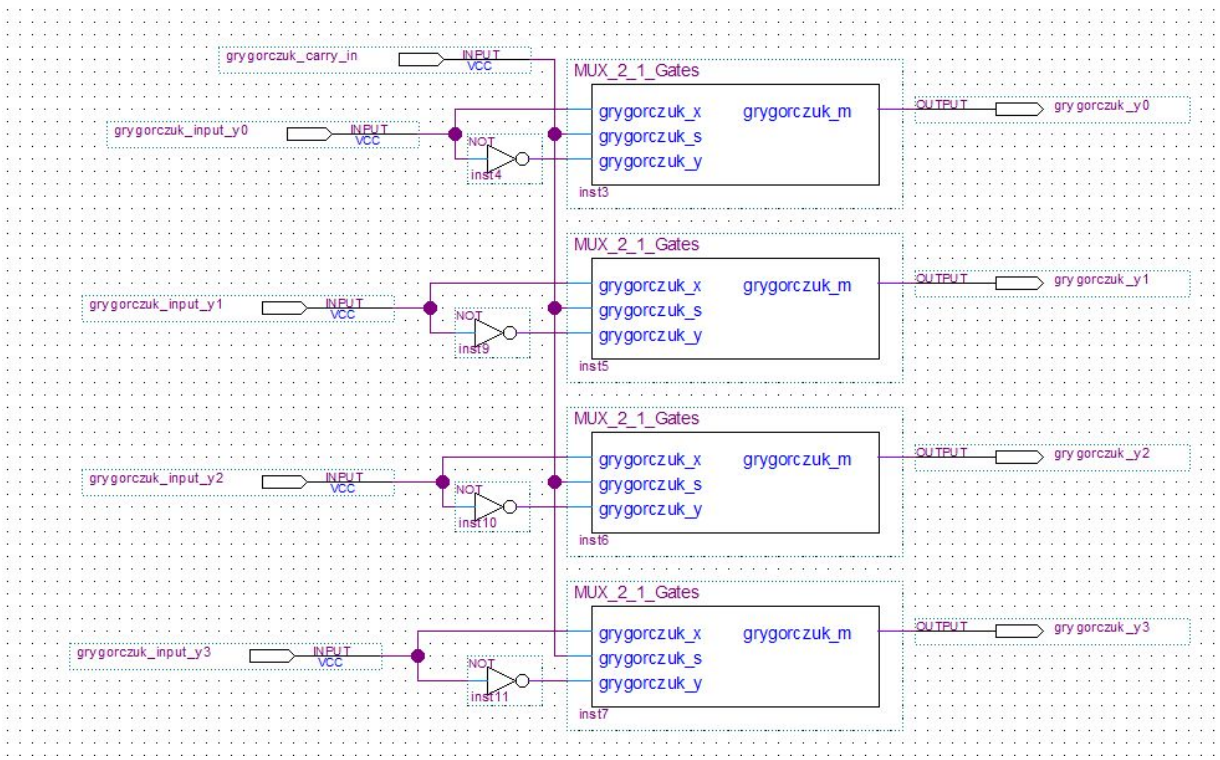

Figure 3. Here is the inside of the TWOS_COMPLEMENT_4_BIT component seen a in Figure 2.

```
library ieee;
use ieee.std_logic_1164.all;

entity TWOS_COMPLEMENT is
	port(
		grygorczuk_y0, grygorczuk_y1, grygorczuk_y2, grygorczuk_y3,
grygorczuk_s_1 : in std_logic;
		grygorczuk_m0,grygorczuk_m1,grygorczuk_m2,grygorczuk_m3 : out std_logic
	);
end TWOS_COMPLEMENT;
```

```
architecture TWOS_COMPLEMENT_LOGIC of TWOS_COMPLEMENT is
signal OUTPUT_0,OUTPUT_1,OUTPUT_2,OUTPUT_3: std_logic;
component MUX is
        port(
                grygorczuk_x, grygorczuk_y, grygorczuk_s : in std_logic;
                grygorczuk_m : out std_logic
        );
end component;

begin
MUX_0: MUX port map(grygorczuk_y0, not grygorczuk_y0,grygorczuk_s_1, OUTPUT_0);
MUX_1: MUX port map(grygorczuk_y1, not grygorczuk_y1,grygorczuk_s_1, OUTPUT_1);
MUX_2: MUX port map(grygorczuk_y2, not grygorczuk_y2,grygorczuk_s_1, OUTPUT_2);
MUX_3: MUX port map(grygorczuk_y3, not grygorczuk_y3,grygorczuk_s_1, OUTPUT_3);

grygorczuk_m0 <= OUTPUT_0;
grygorczuk_m1 <= OUTPUT_1;
grygorczuk_m2 <= OUTPUT_2;
grygorczuk_m3 <= OUTPUT_3;

end TWOS_COMPLEMENT_LOGIC;
```

Code Two's Complement VHDL

Above is the inside of the TWOS_COMPLEMENT_4_BIT it's made up for four 2:1 multiplexers which take in the input and its inverse and based on the state of the carry in input the system will give us the original inputs or their inverses. Below is an example of this shown in waveform simulation using 0 1 0 0 as input with varying carry in inputs.

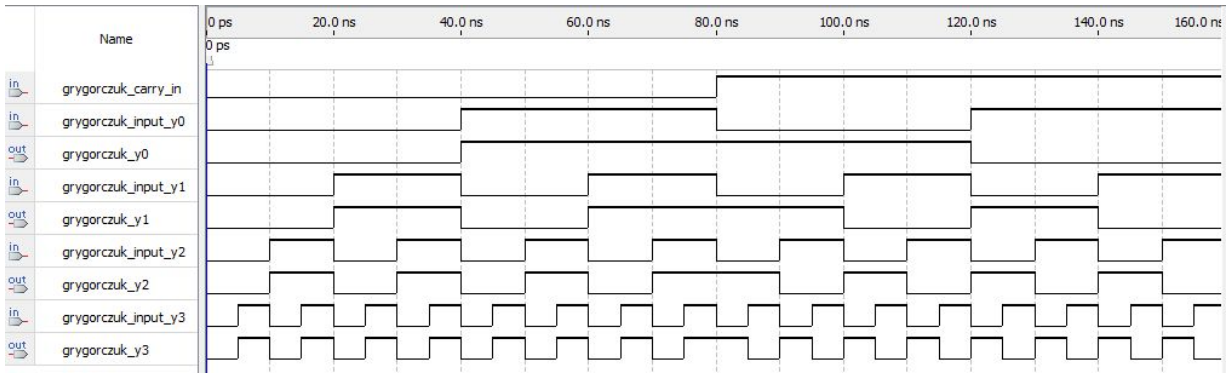

Figure 4. Waveform of the Two's Complement Subsystem

Just to show that the system works on the fundamental level we can zoom in on a singular 2:1 multiplexer as shown in Figure 5 and we can compare it to the premade 2:1 multiplexer from the Library of Parameterized Modules (LPM) shown in Figure 6, and compare their waveforms.
Formula of 2:1 Multiplexer: m = xs' + ys

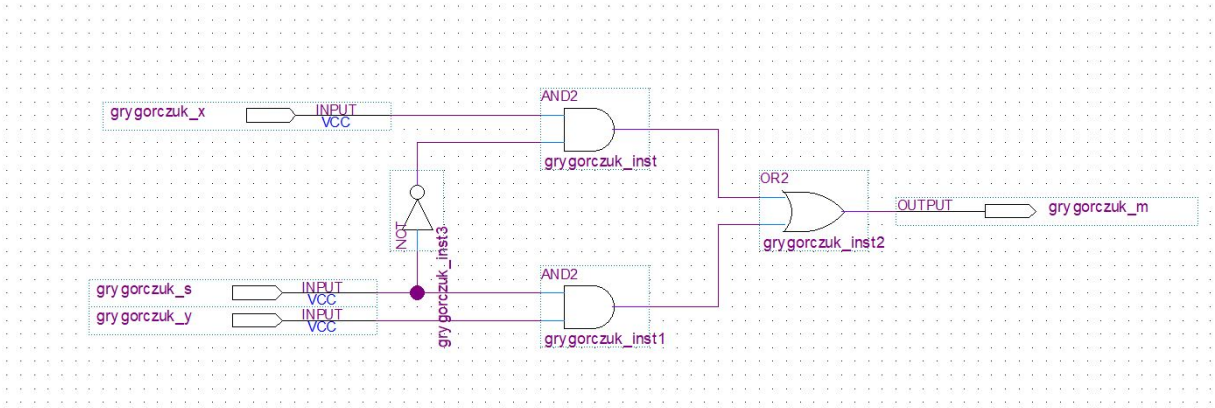

Figure 5. 2:1 Multiplexer

```
library ieee;
use ieee.std_logic_1164.all;

entity MUX is
	port(
		grygorczuk_x, grygorczuk_y, grygorczuk_s : in std_logic;
		grygorczuk_m : out std_logic
	);
end MUX;

architecture MUX_LOGIC of MUX is
signal AND_0, AND_1: std_logic;


begin
AND_0 <= grygorczuk_x and not grygorczuk_s;
AND_1 <= grygorczuk_y and grygorczuk_s;
grygorczuk_m <= AND_0 or AND_1;

end MUX_LOGIC;
```

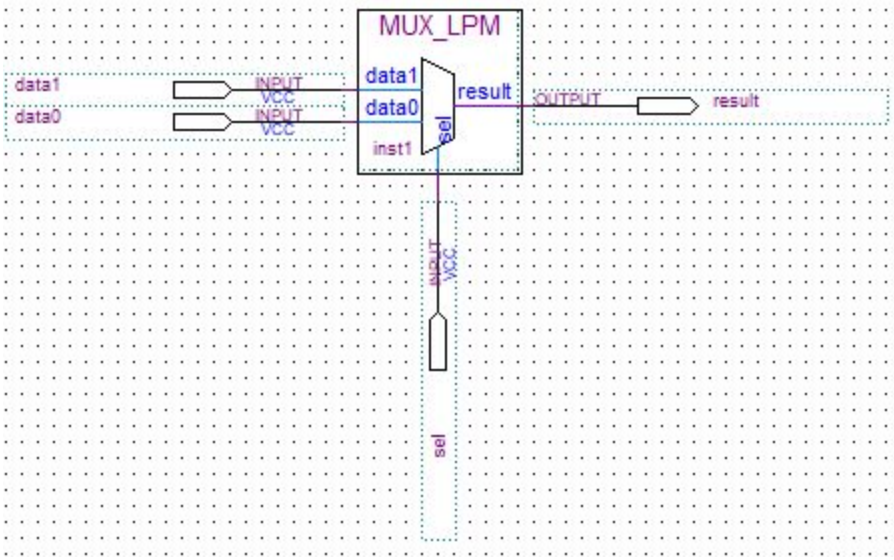

Figure 6. 2:1 Multiplexer LPM

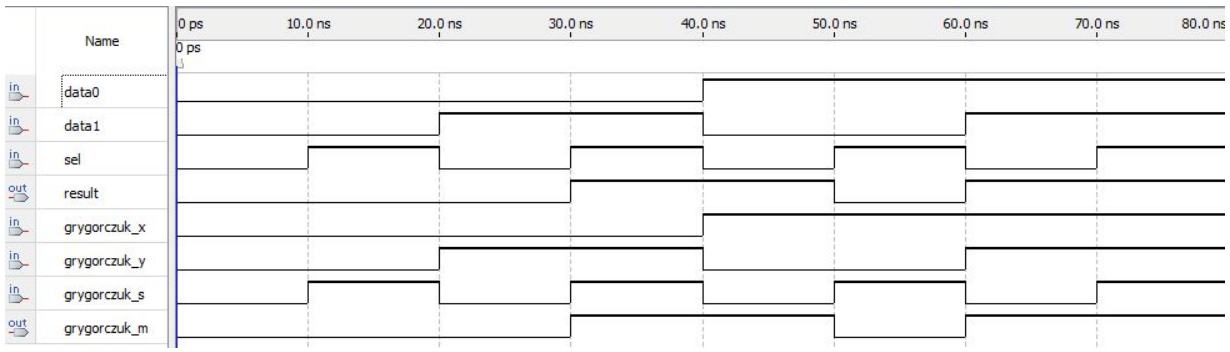

Figure 7. 2:1 Multiplexer Waveform

In Figure 7, we see the first four waves being data0, data1, sel and result and the following for are x,y,s and m we can see that both are the same and we can also verify the result with a truth table for a 2:1 Mux shown below.

X	Y	S	M
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Tabel 1. Truth Table of 2:1 Mux

With this verification we can be sure that so far the results of the inputs are given to the 4 bit adder correctly and we can proceed to the next sub system.

Part 2: Four Bit Adder

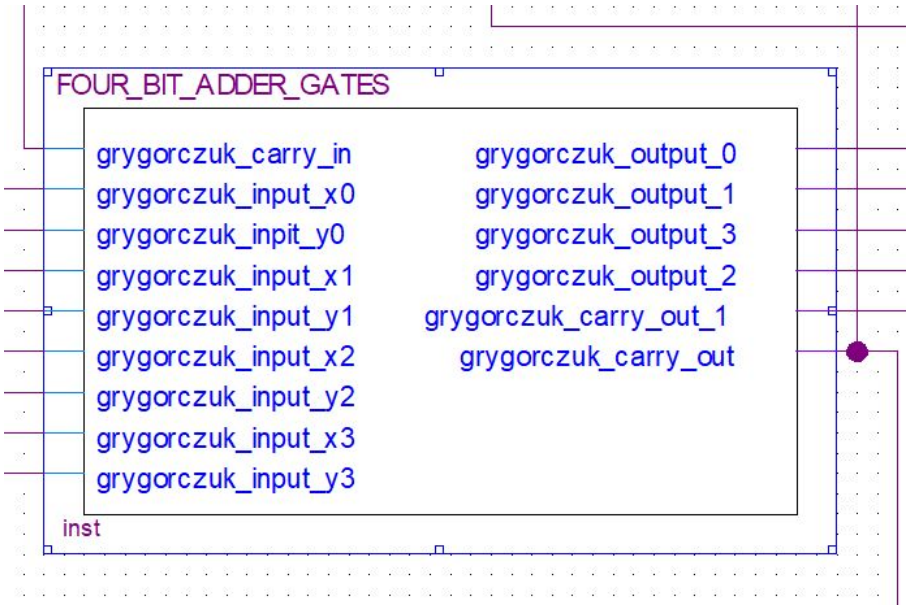

Figure. 8 Four Bit Adder

In this part of the system we take the nine given inputs and add them together. Carry in is used to act as plus one in case two's complement was executed in previous step. Once the inputs have been processed it will give out an output and a carry out output. To show that this works as intended let's start at the smallest part of the subsystem and build up towards the FOUR_BIT_ADDER_GATES block.

First we start off with a half added given by these functions

Output = Input_1 + Input_2

Carry Out = (Input_1)(Input_2)

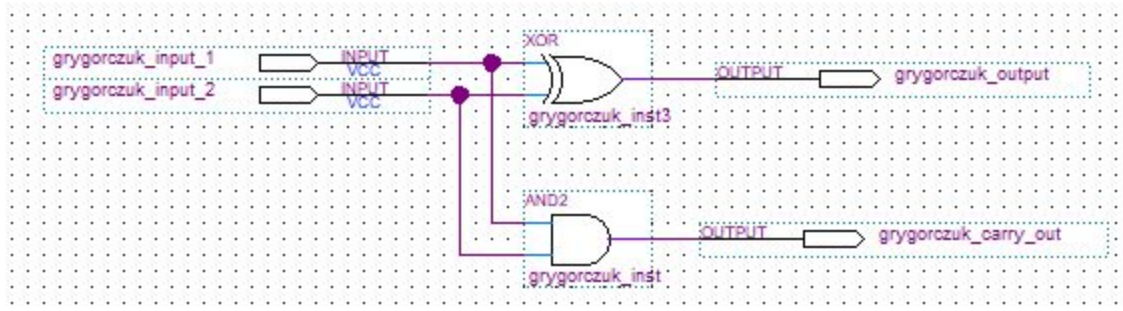

Figure 9. Half-Adder

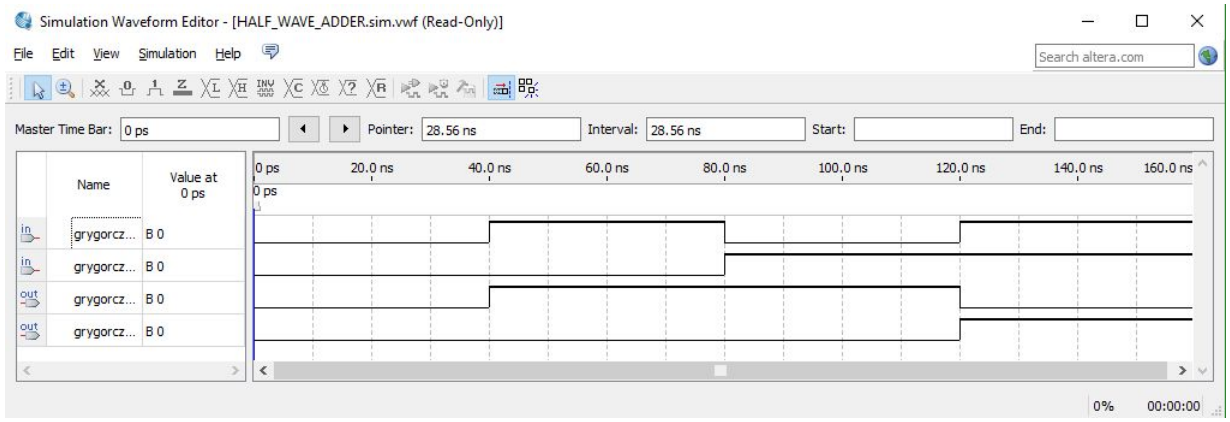

Figure 10. Half Adder Waveform

```
library ieee;
use ieee.std_logic_1164.all;

entity HALF_ADDER is
	port(
		grygorczuk_input_1, grygorczuk_input_2 : in std_logic;
```

```
                grygorczuk_output, grygorczuk_carry_out : out std_logic
        );
end HALF_ADDER;

architecture HALF_ADDER_LOGIC of HALF_ADDER is
begin
        grygorczuk_output <= grygorczuk_input_1 xor grygorczuk_input_2;
        grygorczuk_carry_out <= grygorczuk_input_1 and grygorczuk_input_2;
end HALF_ADDER_LOGIC;
```

Code . Half Adder VHDL

The half adder now will allow us to add a single bit together. Using this fundamental component I've created a full adder shown below

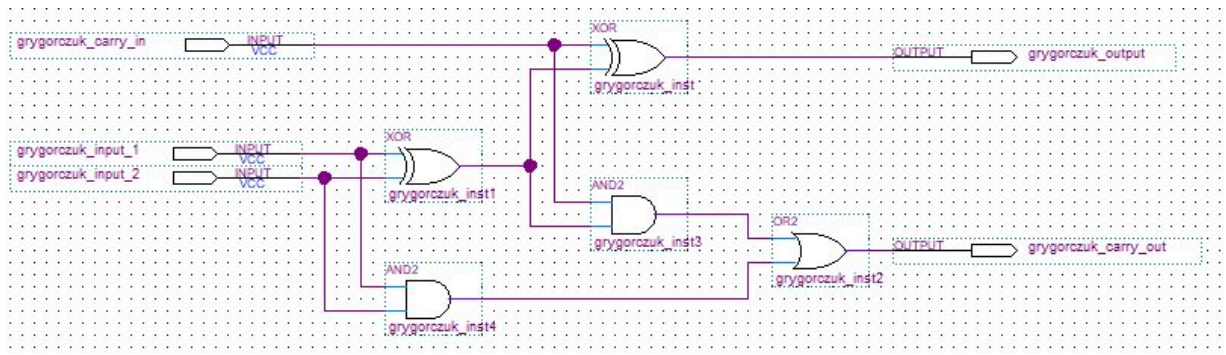

Figure 11.1 Full Adder made of gates

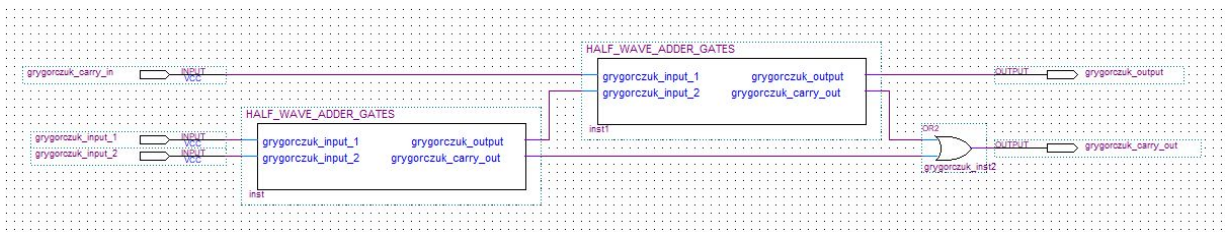

Figure 11.2 Full Adder made of half adders.

```
library ieee;
use ieee.std_logic_1164.all;

entity FULL_ADDER is
        port(
                grygorczuk_input_3, grygorczuk_input_4, grygorczuk_carry_in: in std_logic;
                grygorczuk_output, grygorczuk_carry_out_2 : out std_logic
```

```
        );
end FULL_ADDER;

architecture FULL_ADDER_LOGIC of FULL_ADDER is
signal HALF_ADDER_OUTPUT_1,HALF_ADDER_OUTPUT_2,
HALF_ADDER_CARRY_OUT_1, HALF_ADDER_CARRY_OUT_2 : std_logic;
component HALF_ADDER
port
        (
                grygorczuk_input_1, grygorczuk_input_2 : in std_logic;
                grygorczuk_output, grygorczuk_carry_out : out std_logic
        );
end component;

begin
HALF_ADDER_1: HALF_ADDER port map(grygorczuk_input_3, grygorczuk_input_4,
HALF_ADDER_OUTPUT_1, HALF_ADDER_CARRY_OUT_1);
HALF_ADDER_2: HALF_ADDER port map(grygorczuk_carry_in,
HALF_ADDER_OUTPUT_1, HALF_ADDER_OUTPUT_2, HALF_ADDER_CARRY_OUT_2);
grygorczuk_output <= HALF_ADDER_OUTPUT_2;
grygorczuk_carry_out_2 <= HALF_ADDER_CARRY_OUT_1 or
HAIF_ADDER_CARRY_OUT_2;
end FULL_ADDER_LOGIC;
```

Code. Full Adder Using Half Adder VHDL

With the Full Adder we introduce the carry in input which will be necessary when subtracting is the operation. Below in the figure 12 we can see the waveform of the full adder.

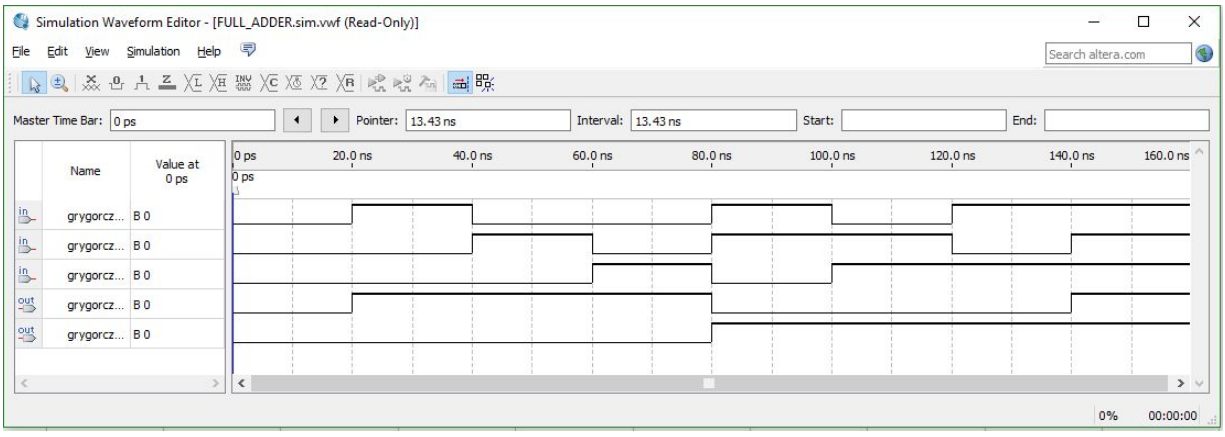

Figure 12. Full Adder

Shown above we see that if any input is 1 we get an output of 1 and carry of 0, if we have two inputs of 1 we get and output of 0 and carry of 1 and if all inputs are 1 then output and carry both will be 0. With the full adder working condition I put four of these in series creating a four bit adder shown in Figure 13.

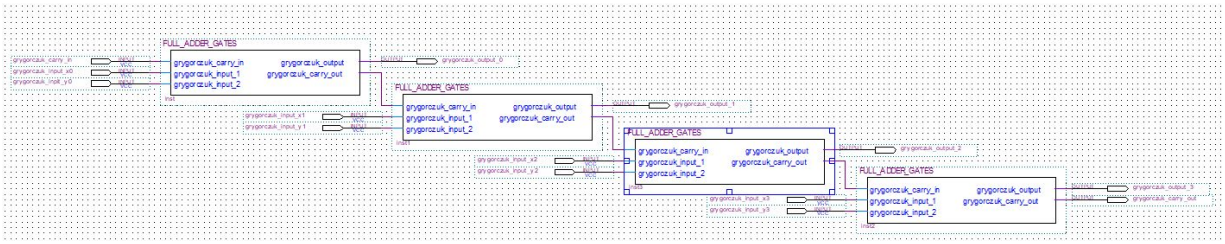

Figure 13. Four Bit Adder

```
library ieee;
use ieee.std_logic_1164.all;

entity FOUR_BIT_ADDER is
	port(
		grygorczuk_carry_in_2 : in std_logic;
		grygorczuk_carry_x0, grygorczuk_carry_x1, grygorczuk_carry_x2,
grygorczuk_carry_x3: in std_logic;
		grygorczuk_carry_y0, grygorczuk_carry_y1, grygorczuk_carry_y2,
grygorczuk_carry_y3: in std_logic;
		grygorczuk_carry_out_3, grygorczuk_carry_o0, grygorczuk_carry_o1,
grygorczuk_carry_o2, grygorczuk_carry_o3: out std_logic
	);
end FOUR_BIT_ADDER;

architecture FOUR_BIT_ADDER_LOGIC of FOUR_BIT_ADDER is
signal OUTPUT_0,OUTPUT_1, OUTPUT_2,OUTPUT_3,CARRY_OUT_0 ,CARRY_OUT_1
,CARRY_OUT_2, CARRY_OUT_3 : std_logic;
component FULL_ADDER
port
	(
		grygorczuk_input_3, grygorczuk_input_4, grygorczuk_carry_in: in std_logic;
		grygorczuk_output, grygorczuk_carry_out_2 : out std_logic
	);
end component;

begin
FULL_ADDER_0: FULL_ADDER port map(grygorczuk_carry_in_2, grygorczuk_carry_x0,
grygorczuk_carry_y0, OUTPUT_0,CARRY_OUT_0);
FULL_ADDER_1: FULL_ADDER port map(CARRY_OUT_0, grygorczuk_carry_x1,
grygorczuk_carry_y1, OUTPUT_1,CARRY_OUT_1);
```

```
FULL_ADDER_2: FULL_ADDER port map(CARRY_OUT_1,grygorczuk_carry_x2,
grygorczuk_carry_y2,OUTPUT_2, CARRY_OUT_2);
FULL_ADDER_3: FULL_ADDER port map(CARRY_OUT_2, grygorczuk_carry_x3,
grygorczuk_carry_y3, OUTPUT_3,CARRY_OUT_3);

grygorczuk_carry_out_3 <= CARRY_OUT_3;
grygorczuk_carry_o0 <= OUTPUT_0;
grygorczuk_carry_o1 <= OUTPUT_1;
grygorczuk_carry_o2 <= OUTPUT_2;
grygorczuk_carry_o3 <= OUTPUT_3;

end FOUR_BIT_ADDER_LOGIC;
```

Code . Four Bit Adder VHDL

Here we create the inputs x0-x3 and y0-y3 with the input carry used to add one in case two's complement was executed. The system will x0 with y0 and initial carry carry_in giving out output_0, x1 with y1 and carry_out of 0th full adder and will give the output_1, x2 with y2 and carry_out of 1st full adder and will give the output_2, and finally x3 with y3 and carry_out of 2nd full adder and will give the output_3 and the carry out of the system. We can look at the waveform to see if the additions are done correctly seem in figure 14.

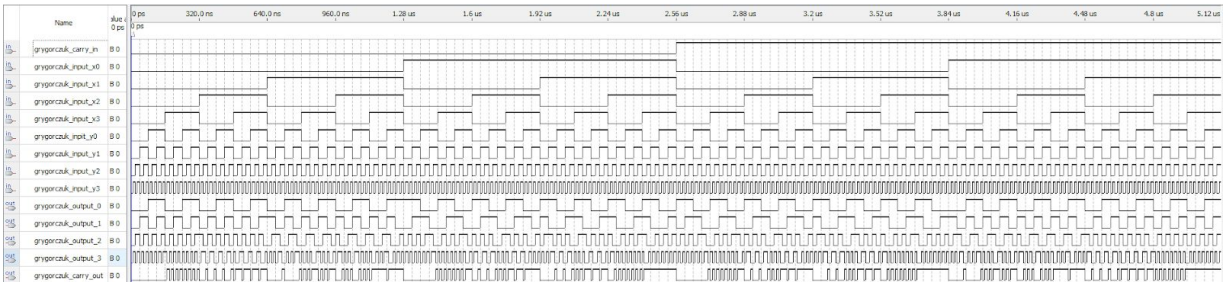

Figure 14.1 Waveform of a Four Bit Adder

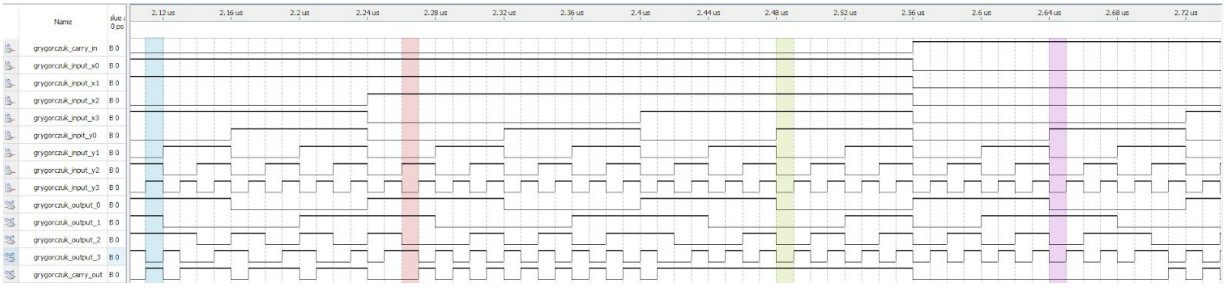

Figure 14.2 Zoom In

Above in Figure 14.1 we see that there are many possibilities of add between the nine input, so I zoomed in roughly in the middle and picked four random points to see if their additions are correctly computed.

Blue Carry In 0 0 0 0 (0) X3 X2 X1 X0 1 0 1 1 (11) Y3 Y2 Y1 Y0 1 1 0 0 (12) Output: O3 O2 O1 O0 0 1 1 1 (7) Carry Out 0 0 0 1 (16)	Red Carry In 0 0 0 0 (0) X3 X2 X1 X0 0 1 1 1 (7) Y3 Y2 Y1 Y0 0 1 0 0 (4) Output: O3 O2 O1 O0 1 0 1 1 (11) Carry Out 0 0 0 0 (0)
Green Carry In 0 0 0 0 (0) X3 X2 X1 X0 1 1 1 1 (15) Y3 Y2 Y1 Y0 0 0 0 1 (1) Output: O3 O2 O1 O0 0 0 0 0 (1)7 bgg Carry Out 0 0 0 1 (16)	Purple Carry In 0 0 0 1 (1) X3 X2 X1 X0 0 0 0 0 (0) Y3 Y2 Y1 Y0 0 0 0 1 (1) Output: O3 O2 O1 O0 0 0 1 0 (2) Carry Out 0 0 0 0 (0)

Tabel 2. Visual representation of the strips shown in Figure 14.2

Table 2 shows us that the additions are done correctly and thus this subsystem is compacted to the symbol shown in Figure 8. Once the computation is complete we have to check for several output that this new information presents us with. First is if there is a carry out signal.

Part 3: Carry Out, Negative Flags and 4 LED Binary Display

The simplest of the information to process from this addition/subtraction are the Carry Out Flag, Negative Flag and th 4 LEDs.

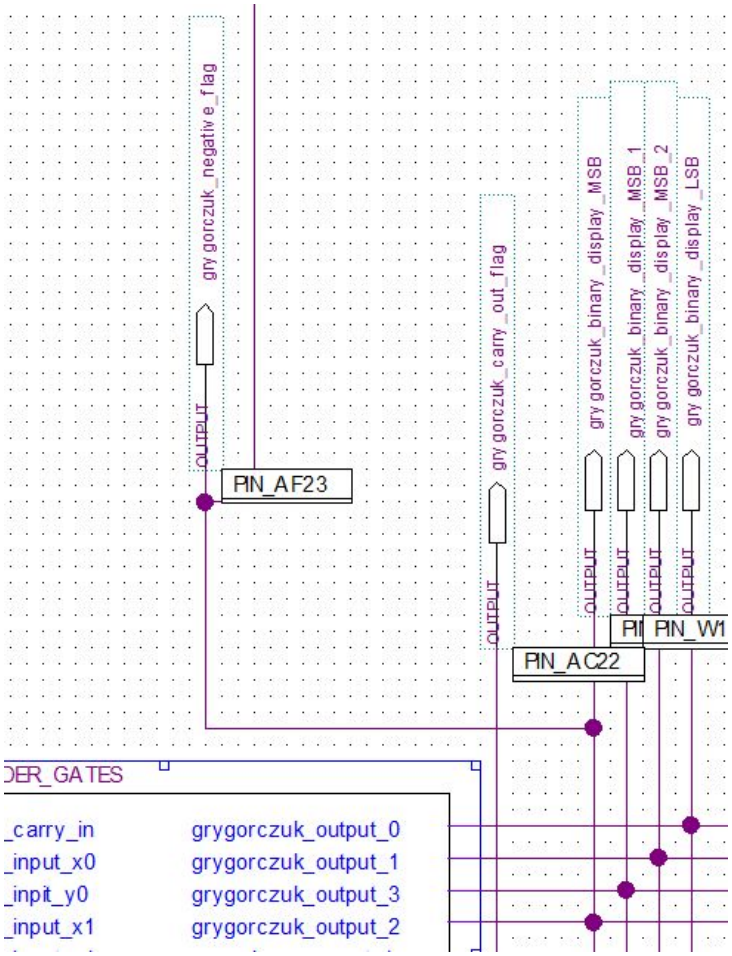

Figure 15. Carry Out, Negative Flags and 4 LED Binary Display

In Figure 15, we see that the Negative Flag is connected to the MSB, if the bit is 0 the number is positive and if the bit is 1 the number is negative. The Negative Flag will be crucial in display the information on the seven segment display. We also see that the four output of MSB, MSB_1, MSB_2 and LSB are connected out output_3, output_2, output_1, and output_0 respectively to be used as a binary display of the outputs. Lastly Figure 15 shows the carry out flag is connected directly to the carry out output of the FOUR_BIT_ADDER_GATE system.

Part 4: Zero and Overflow Flags

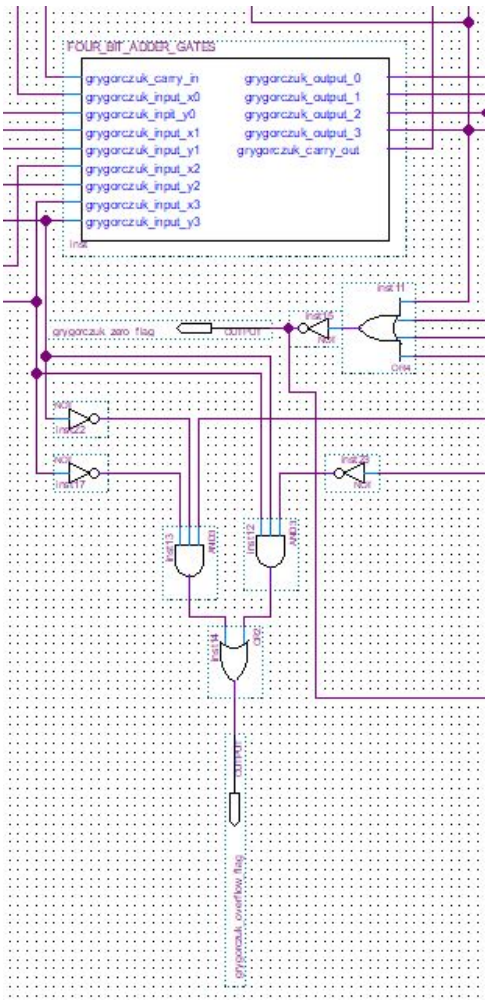

Figure 16. Zero and Overflow Flags

Here we take a look at flags that require us to manipulate the information of the outputs to check for the conditions of the the number being zero or the two numbers adding up out of the scope of the addition. In figure 17, we see that the zero flag is given by an inverse of a four pin or gate, the or gate takese in the four binary outputs, if all of them are zero then the flag is set to one. The zero flag is necessary in the inputs of one of the seven segment display. The function of the zero flag is

Zero = (Output_0 + Output_0 + Output_0 + Output_0)’

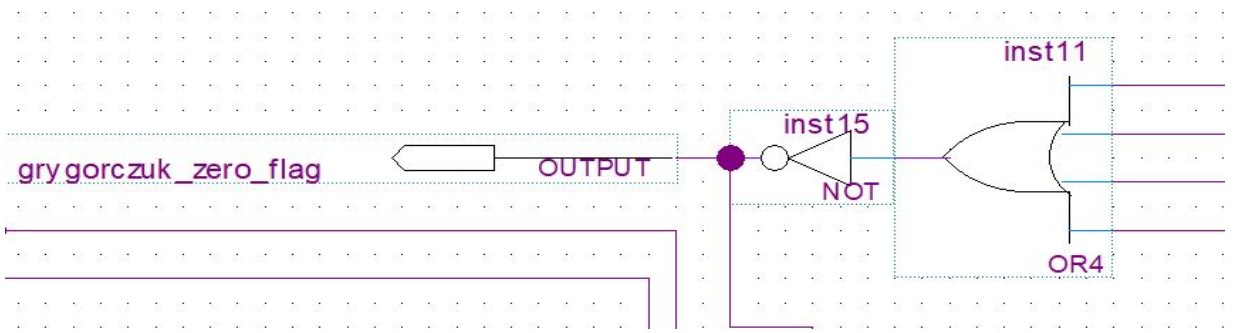

Figure. 17 Zero Flag

The overflow flag is made up by checking the values of the last carry out flag and the second to last carry out flag run through a xor gate.

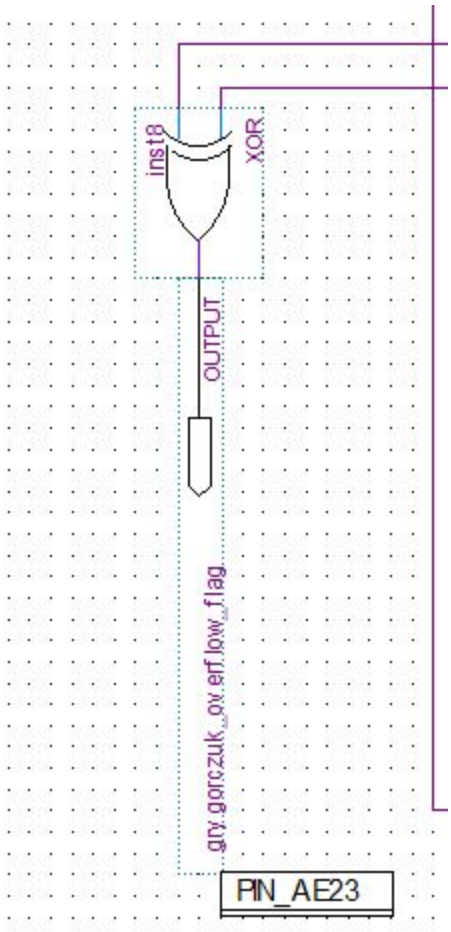

Figure 18. Overflow Flag

Part 5: Comparator

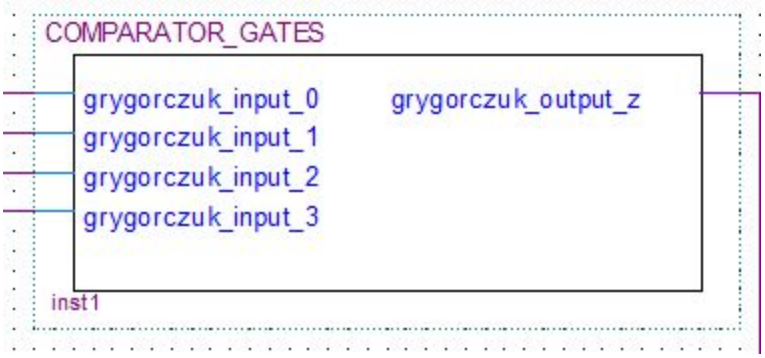

Figure 19. Comparator

The comparator is a subsystem used track whether the output value is larger than nine, if the value is larger than nine it will output a one to be displayed on the second digit seven segment display, if its less than one it will send a zero to show on the second digit seven segment. It takes in the direct output of the four bit adder result and returns zero or one. Below we can see the the inside of the comparator it has four inputs however LSB doesn't affect it so it's just grounded while the other three form this formula

Z = (input_1)(input_3) + (input_1)(input_2)

In Figure 21, we can see the waveform of the comparator proving that anything less than ten gives a zero output whine anything larger than nine gives out a one output. In this subsystem I started using the nand gates to construct and/or gates and the two symbols can be inspected in Figures 22.1 and 22.2.

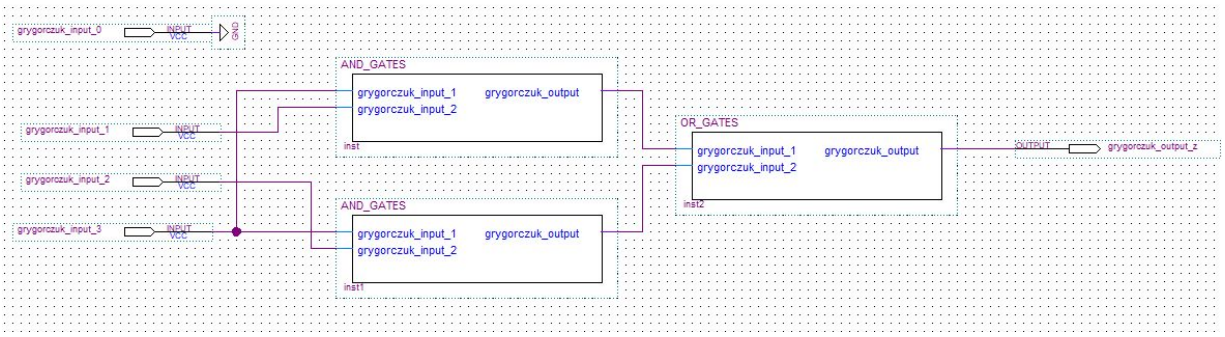

Figure 20. Comparator made of gates

```
library ieee;
use ieee.std_logic_1164.all;

entity COMPARATOR is
	port(
		grygorczuk_o0, grygorczuk_o1, grygorczuk_o2, grygorczuk_o3 : in std_logic;
		grygorczuk_z : out std_logic
	);
end COMPARATOR;

architecture COMPARATOR_LOGIC of COMPARATOR is
signal AND_0, AND_1: std_logic;

begin
AND_0 <= grygorczuk_o1 and grygorczuk_o3;
AND_1 <= grygorczuk_o2 and grygorczuk_o3;
grygorczuk_z <= AND_0 or AND_1;

end COMPARATOR_LOGIC;
```

Code Comparator VHDL

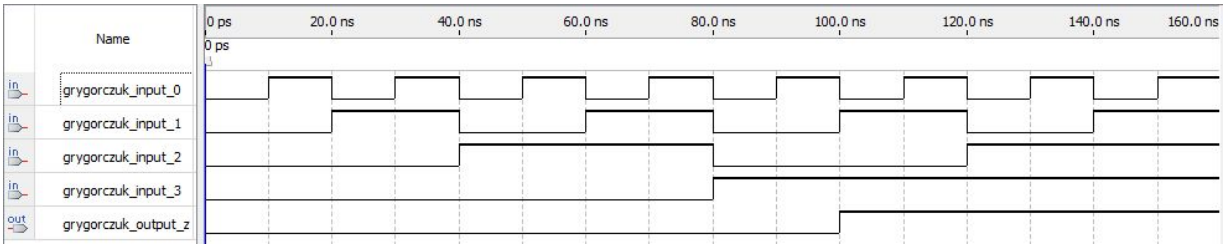

Figure 21. Comparator Waveform

	i3	i2	i1	i0	z
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	0
5	0	1	0	1	0

6	0	1	1	0	0
7	0	1	1	1	0
8	1	0	0	0	0
9	1	0	0	1	1
10	1	0	1	0	1
11	1	0	1	1	1
12	1	1	0	0	1
13	1	1	0	1	1
14	1	1	1	0	1
15	1	1	1	1	1

Tabel 3. Comparator Truth Table

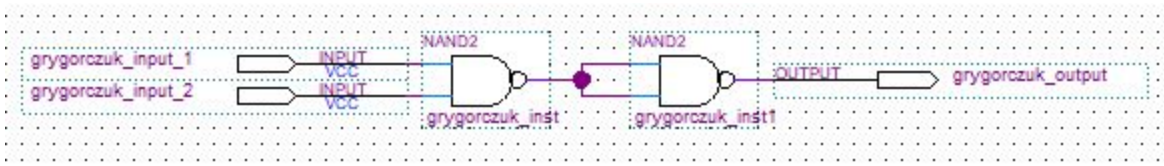

Figure 22.1 And gate made from nand gates

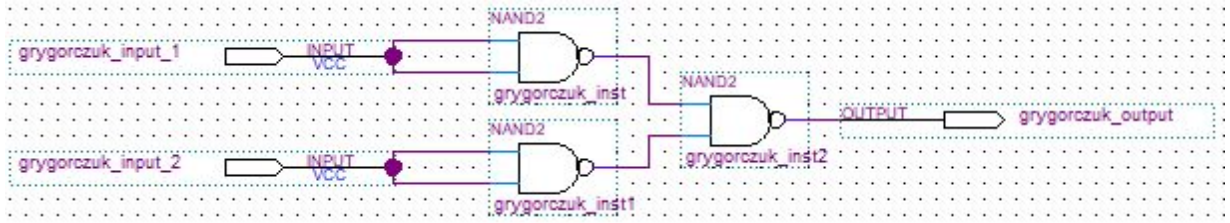

Figure 22.2 Or gate made from nand gates

To prove that these function the same way that regular and/or gates work Figures 23.1 and 23.2 show the wave from on my and/or gate compared to waveform of regular and/or gates.

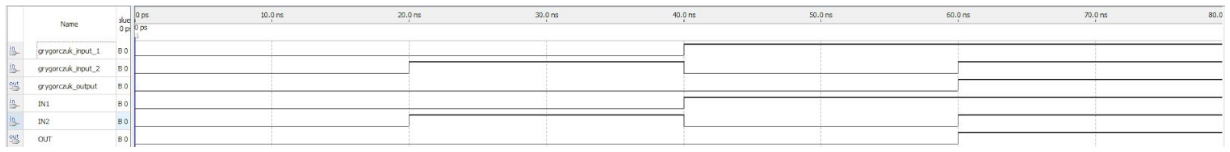

Figure 23.1 And Waveform

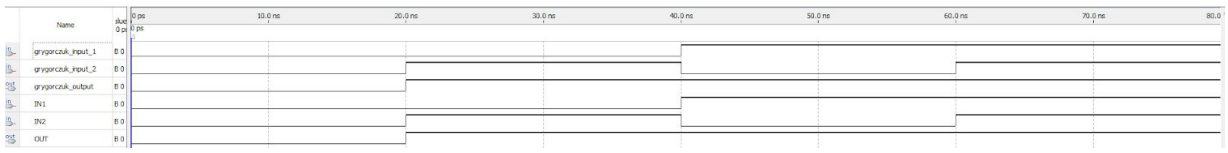

Figure 23.2 Or Waveform

The figure above show that the and/or gates function the same way.

Part 6: Plus, Minus, One, Zero 7 Segment Display

The first seven segment display is used to show plus, zero and minus when the system is display signed numbers and zero or one when displaying unsigned numbers.

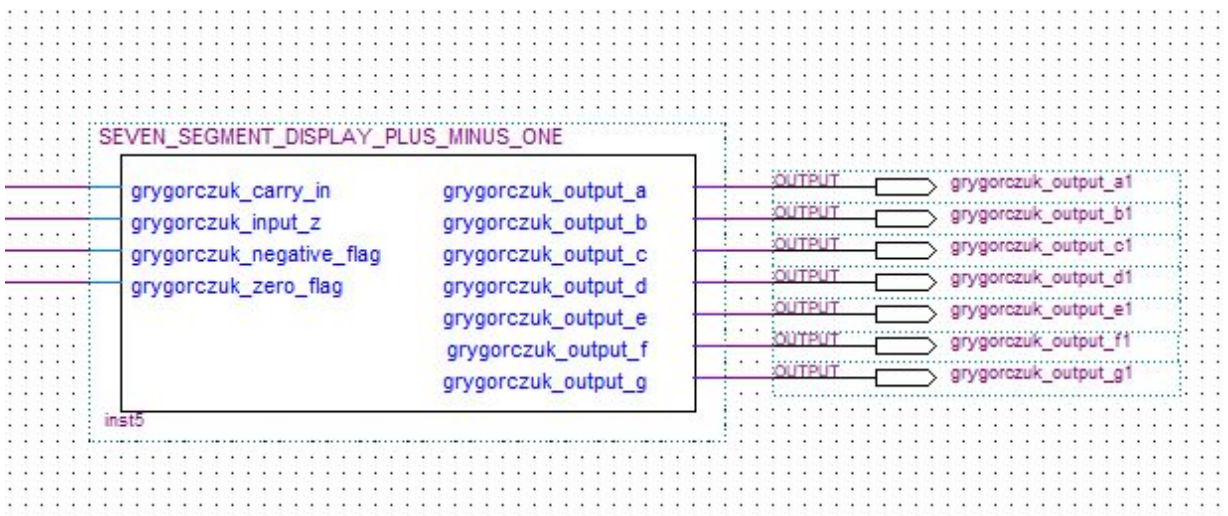

Figure 24. Seven segment display digit two

In this system we take four inputs, carry_in, z, negative flag and zero flag. The carry in is to tell the system if we're showing the numbers as signed or unsigned, z tells the system to display one if the numbers are unsigned, negative flag tells the system to display plus or minus and the zero flag is there to tell the system to display zero.

	Carry In	Z	Neg	Zero	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	1
1	0	0	0	1	1	1	1	1	1	1	1
2	0	0	1	0	1	1	1	1	1	1	1

3	0	0	1	1	1	1	1	1	1	1	1
4	0	1	0	0	1	0	0	1	1	1	1
5	0	1	0	1	1	0	0	1	1	1	1
6	0	1	1	0	1	0	0	1	1	1	1
7	0	1	1	1	1	0	0	1	1	1	1
8	1	0	0	0	1	0	0	1	1	1	0
9	1	0	0	1	1	1	1	1	1	1	1
10	1	0	1	0	1	1	1	1	1	1	0
11	1	0	1	1	1	1	1	1	1	1	1
12	1	1	0	0	1	0	0	1	1	1	0
13	1	1	0	1	1	1	1	1	1	1	1
14	1	1	1	0	1	1	1	1	1	1	0
15	1	1	1	1	1	1	1	1	1	1	1

Tabel 4. Truth Table of the Seven Segment Display

I’ve create this truth table to display the symbols as explained prior and below is the schematic for it

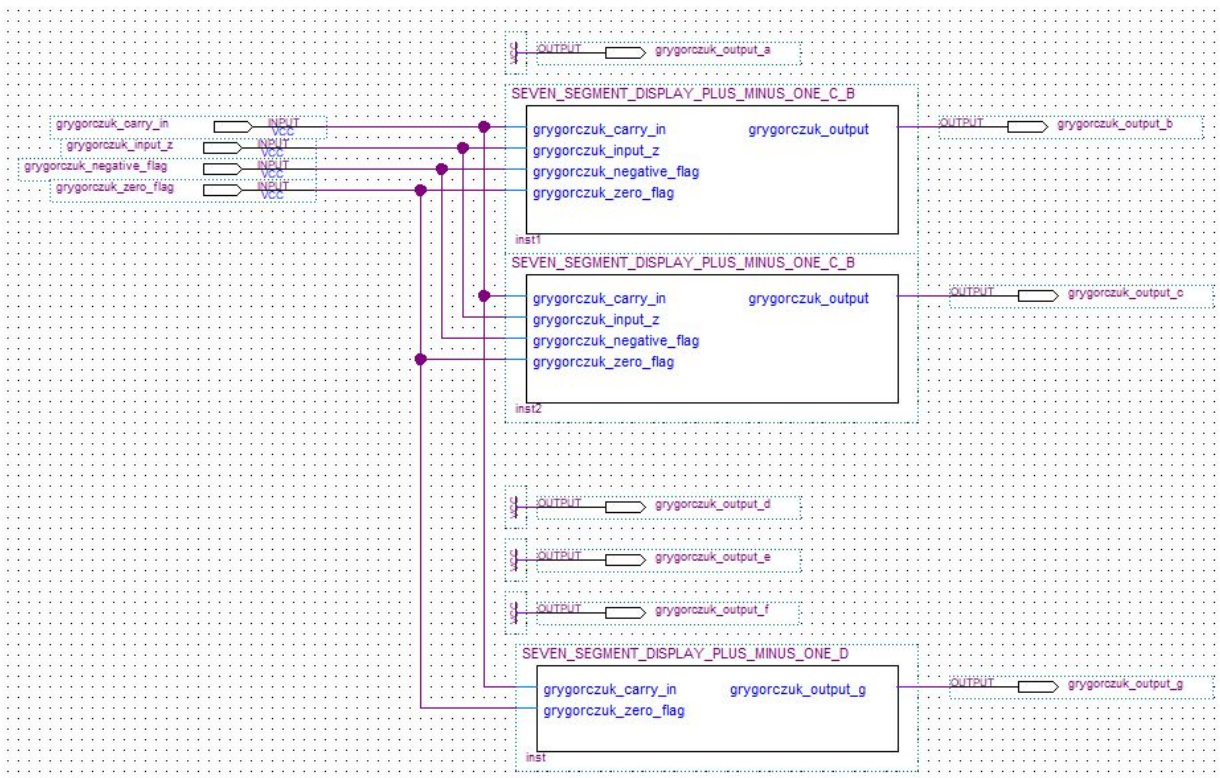

Figure 25. Seven segment display digit two inside

```
library ieee;
use ieee.std_logic_1164.all;

entity SEVEN_SEGMENT_D2 is
	port(
		grygorczuk_sign, grygorczuk_input_z, grygorczuk_negative_flag,
grygorczuk_zero_flag : in std_logic;
		grygorczuk_output_a, grygorczuk_output_b, grygorczuk_output_c,
grygorczuk_output_d, grygorczuk_output_e, grygorczuk_output_f, grygorczuk_output_g : out
std_logic
	);
end SEVEN_SEGMENT_D2;

architecture SEVEN_SEGMENT_D2_LOGIC of SEVEN_SEGMENT_D2 is
signal AND_0, AND_1, AND_2, OR_0, OR_1 : std_logic;

begin

AND_0 <= not grygorczuk_sign and not grygorczuk_input_z;
AND_1 <= grygorczuk_sign and grygorczuk_negative_flag;
AND_2 <= grygorczuk_sign and grygorczuk_zero_flag;
```

```
OR_0 <= AND_0 or AND_1;
OR_1 <= OR_0 or AND_2;

grygorczuk_output_a <= '1';
grygorczuk_output_b <= OR_1;
grygorczuk_output_c <= OR_1;
grygorczuk_output_d <= '1';
grygorczuk_output_e <= '1';
grygorczuk_output_f <= '1';
grygorczuk_output_g <= not grygorczuk_sign or grygorczuk_zero_flag;

end SEVEN_SEGMENT_D2_LOGIC;
```

Code Seven Segment D2

To prove that is shows the same thing as the truth table here is the waveform of the system.

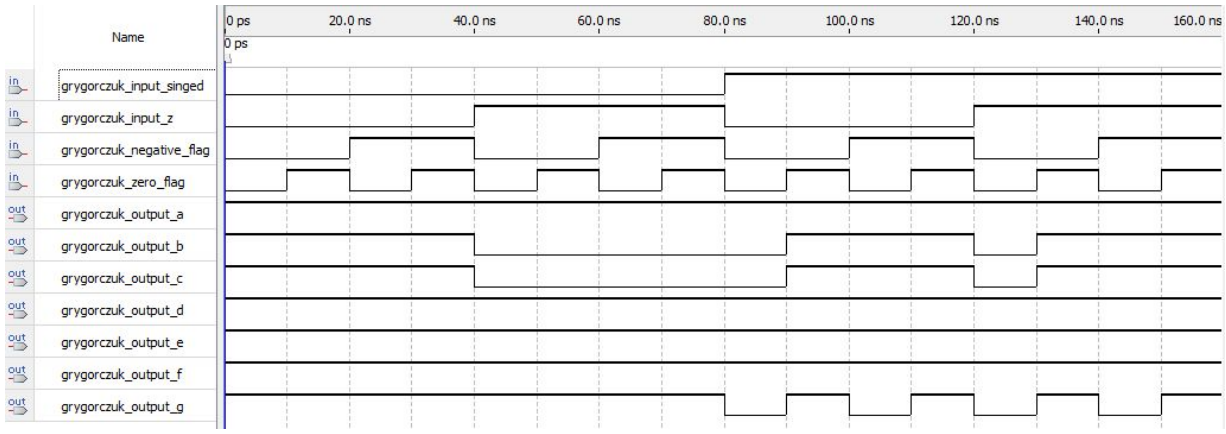

Figure 26. Waveform of the seven segment display digit two inside

Below are the circuits used to create the SEVEN_SEGMENT_DISPLAY_PLUS_ONE_G and SEVEN_SEGMENT_DISPLAY_PLUS_ONE_B_C with their formulas being

B/C = (carry_in)'(z)' + (carry_in)(negative_flag) + (carry_in)(zero_flag)

G = carry_in' + zero_flag

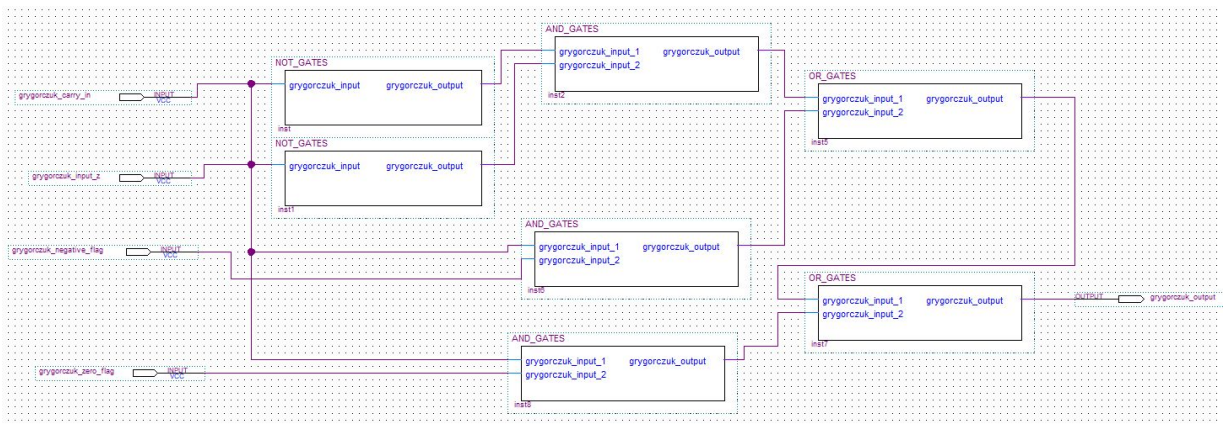

Figure 27.1 Circuit of SEVEN_SEGMENT_DISPLAY_PLUS_ONE_B_C

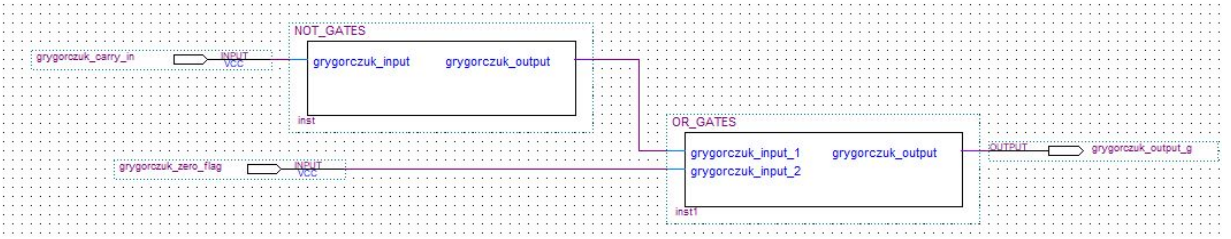

Figure 27.2 SEVEN_SEGMENT_DISPLAY_PLUS_ONE_D

Part 7: Reset

Next we have the reset button, to make sure that the number on the first digit seven segment display goes back to zero and starts count up against once the output of the four bit adder is larger than nine.

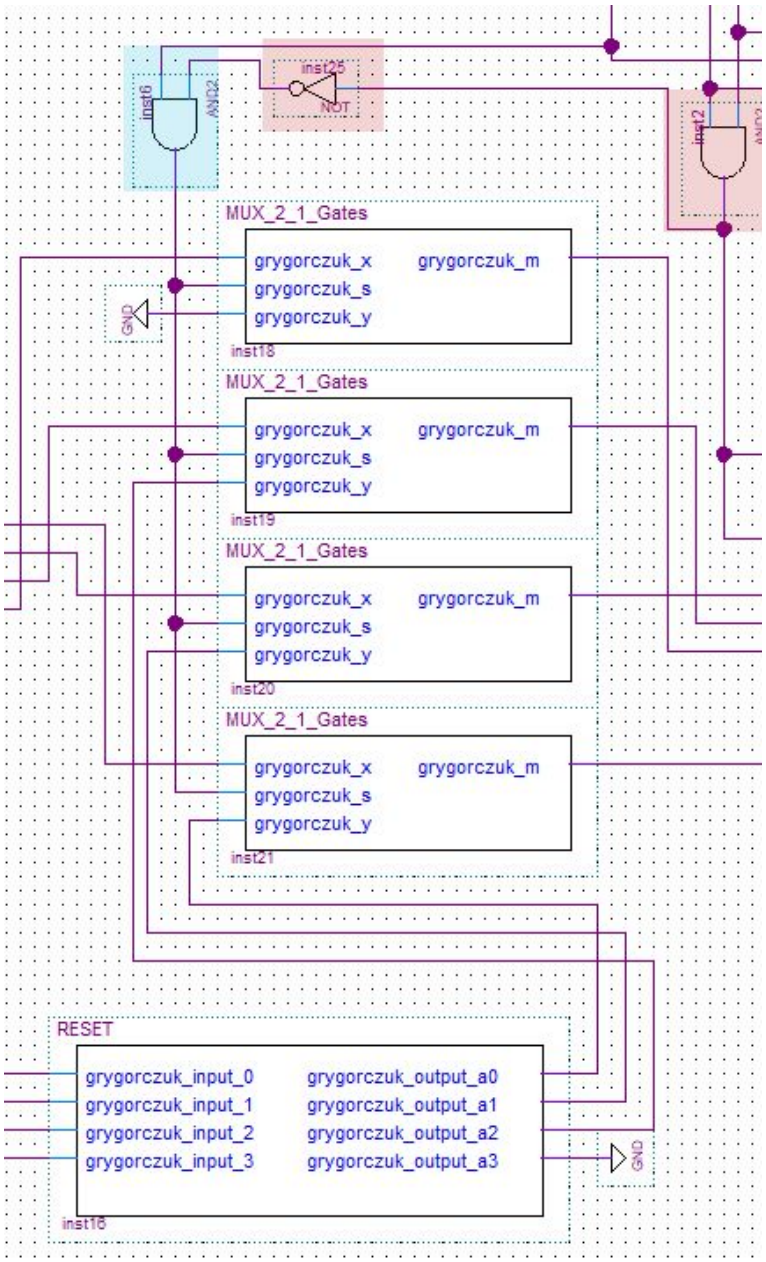

Figure 28. Reset

There are two sides to this system, the first is the reset symbol which keeps track between zero and nine then goes back to zero to five when the output is larger than nine. The multiplexers than take the initial outputs and those created by the reset system. On the other side is the switch signal to the multiplexers, that signal is made up of three major inputs; z from the comparator, negative flag and carry in. The signal function is

S = z[(negative_flag)(carry_in)]’

The z signal is there to add one to the second digit seven segment display while the negative_flag and carry_in signals are there to make sure the numbers aren't reseted and can later be converted to their signed equivalents.

Looking inside the reset system in Figure 29

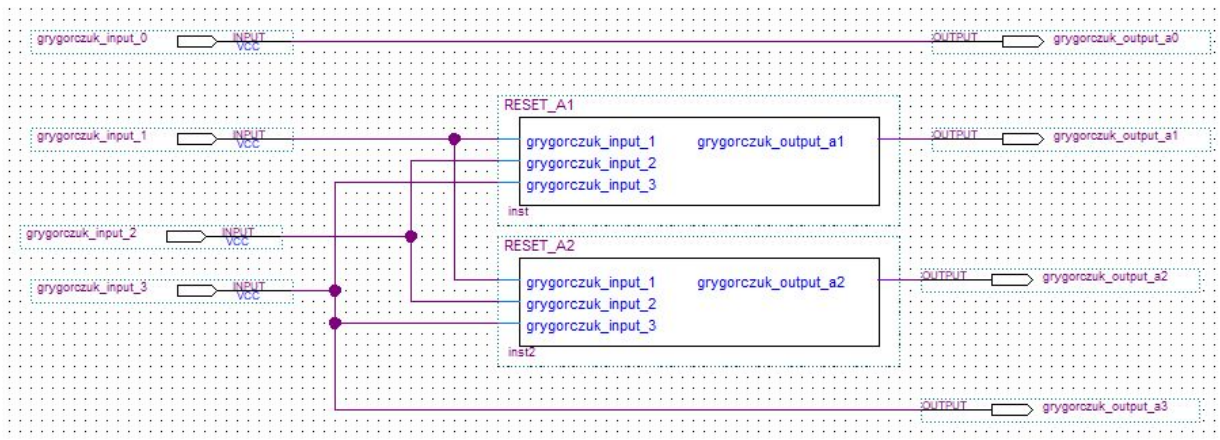

Figure 29. Inside of the Reset system

```
library ieee;
use ieee.std_logic_1164.all;

entity RESET_1 is
	port(
		grygorczuk_input_0, grygorczuk_input_1, grygorczuk_input_2,
grygorczuk_input_3 : in std_logic;
		grygorczuk_output_0, grygorczuk_output_1, grygorczuk_output_2,
grygorczuk_output_3: out std_logic
	);
end RESET_1;

architecture RESET_LOGIC of RESET_1 is
signal AND_0, AND_1,OR_0, AND_2, AND_3, OR_1 : std_logic;

begin

AND_0 <= grygorczuk_input_1 and not grygorczuk_input_3;
AND_1 <= not grygorczuk_input_1 and grygorczuk_input_2 and grygorczuk_input_3;
OR_0 <= AND_0 or AND_1;

AND_2 <= grygorczuk_input_2 and not grygorczuk_input_3;
AND_3 <= grygorczuk_input_1 and grygorczuk_input_2;
OR_1 <= AND_2 or AND_3;
```

```
grygorczuk_output_0 <= grygorczuk_input_0;
grygorczuk_output_1 <= OR_0;
grygorczuk_output_2 <= OR_1;
grygorczuk_output_3 <= grygorczuk_input_3;

end RESET_LOGIC;
```

Code Reset VHDL

	i3	i2	i1	i0	a3	a2	a1	a0
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	1	0	0	0	1	0
3	0	0	1	1	0	0	1	1
4	0	1	0	0	0	1	0	0
5	0	1	0	1	0	1	0	1
6	0	1	1	0	0	1	1	0
7	0	1	1	1	0	1	1	1
8	1	0	0	0	1	0	0	0
9	1	0	0	1	1	0	0	1
10	1	0	1	0	1	0	0	0
11	1	0	1	1	1	0	0	1
12	1	1	0	0	1	0	1	0
13	1	1	0	1	1	0	1	1
14	1	1	1	0	1	1	0	0
15	1	1	1	1	1	1	0	1

Tabel 5. Reset Truth Table

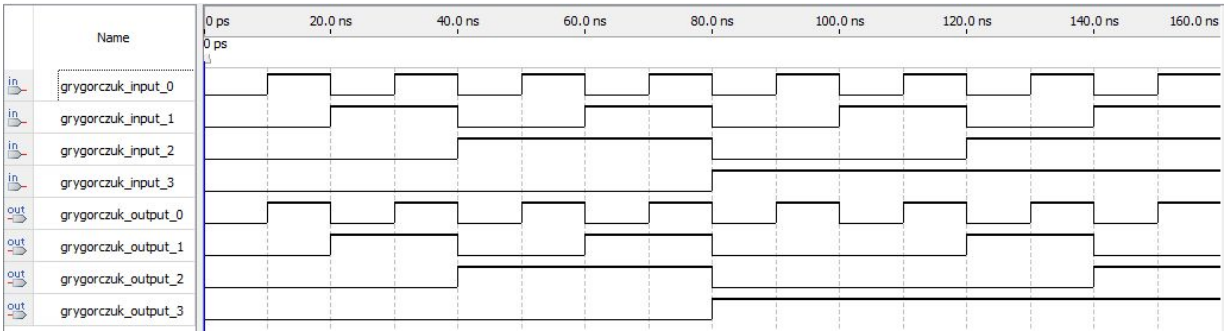

Figure 30. Waveform of Reset

With these modified waveforms the number would reset to zero through five once the outputs of the four bit adder are larger than nine. Below you can see how the A1 and A2 signals are created.

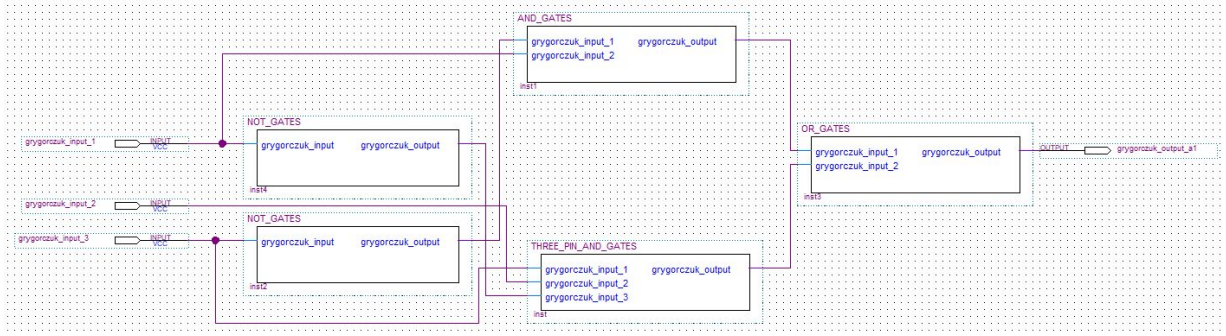

Figure 31.1 Reset A1

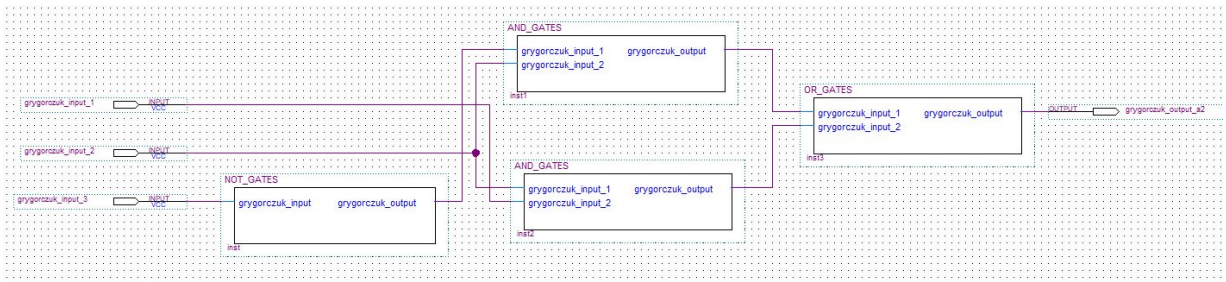

Figure 31.2 Reset A2

Part 8: Two's Complement and 4 Bit Adder

Here we have the combination of two systems previously before, we use the two's complement and the four bit adder to transform our output into its complement if they are negative. There probably is a much more efficient way to do this but I was running out and this worked.

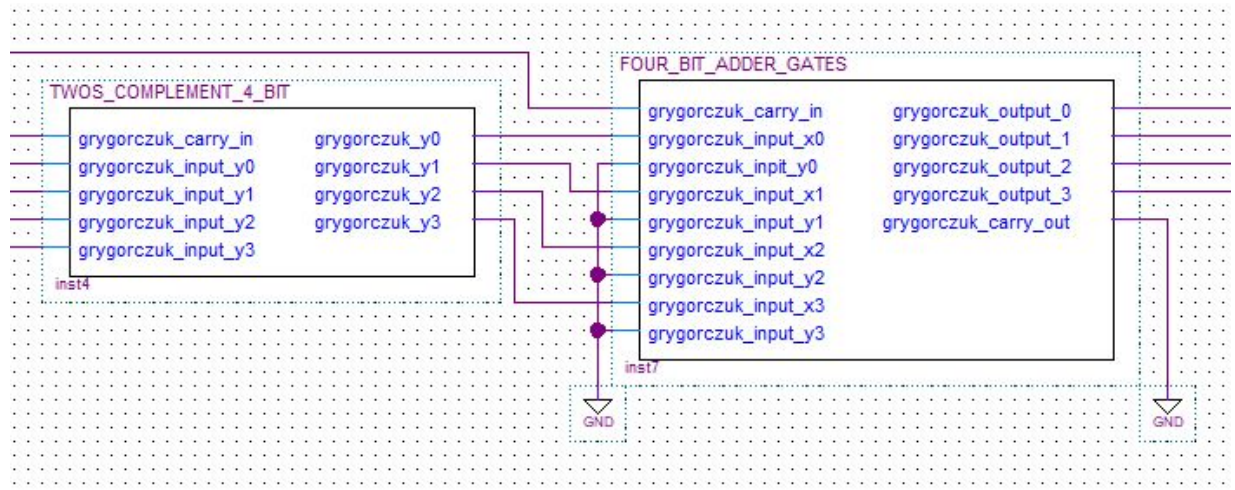

Figure 32. Two's Complement and 4 Bit Adder

The inputs for the two's complement come from the multiplexers that were affected by the reset and comparator, with the carry in being an and signal between the initial carry in and the negative flag. Once that works the four bit adder is used to add the one in two's complement. This subsystem will either keep the initial signals as they were or will invert them.

Part 9: 0-9 Seven Segment Display

This is the last component of the whole system it takes in all the information done by comparator, the reset and the two's complement. All those previous subsystems are used to show zero through nine on the seven segment display.

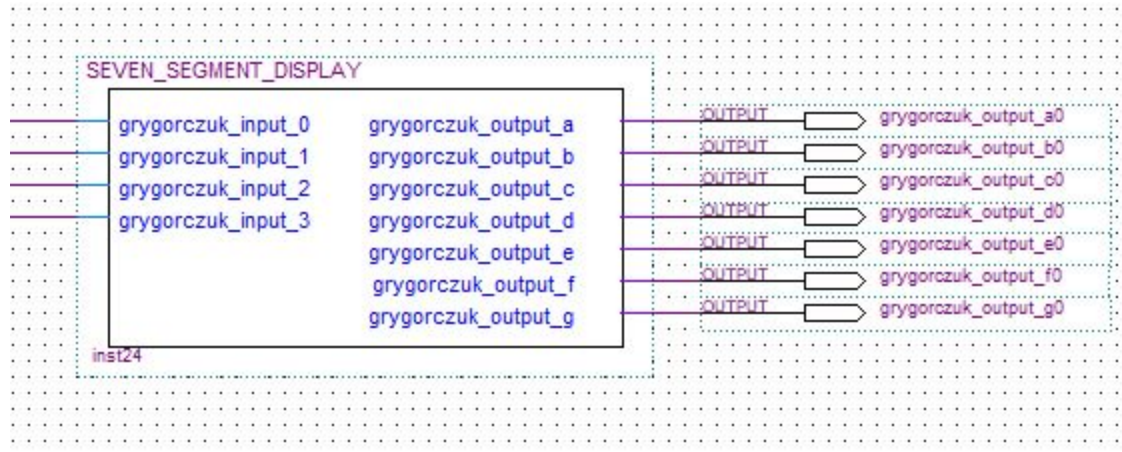

Figure 33. Seven Segment Display

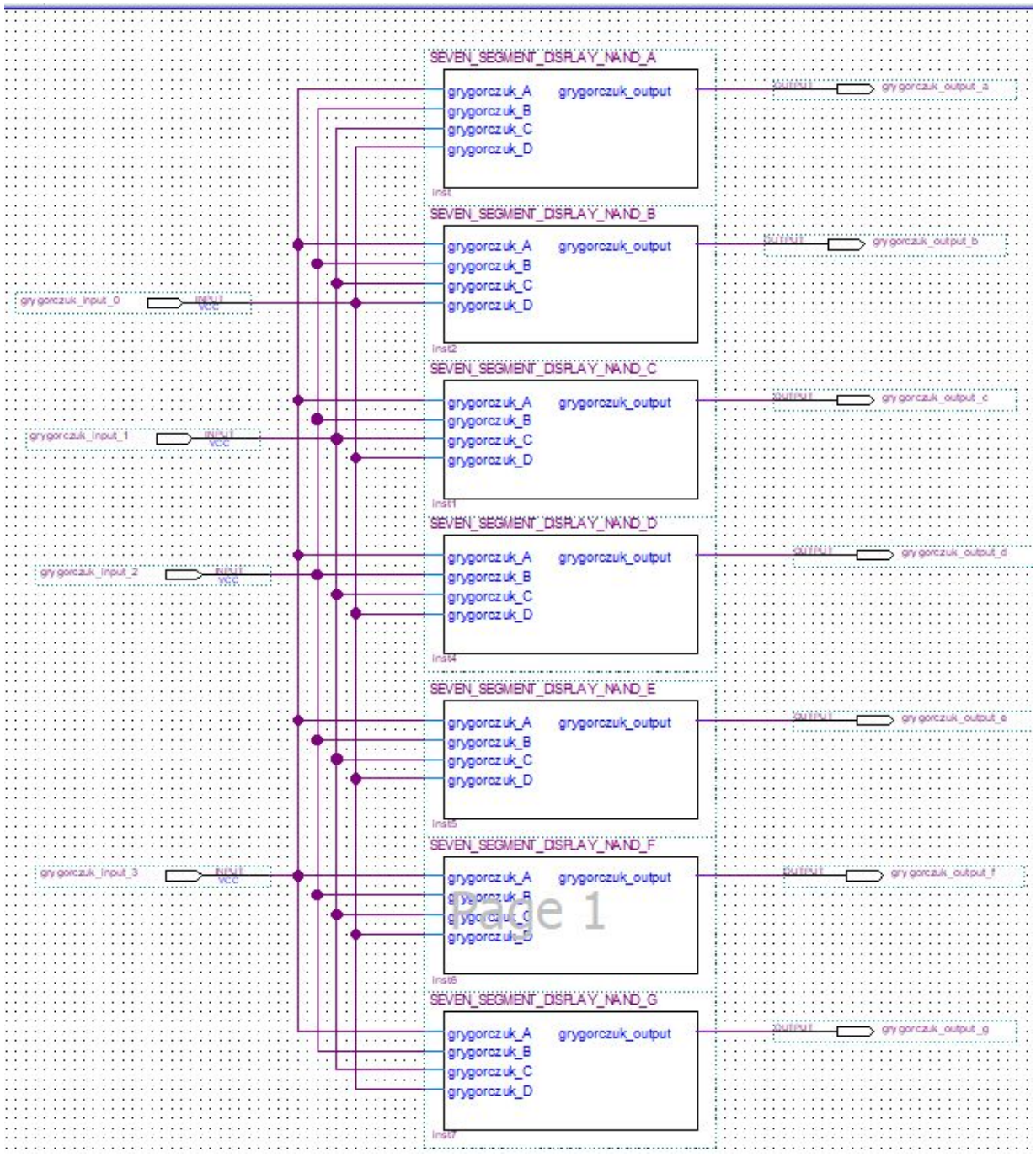

Figure 34. Inside of the Seven Segment Display

```
library ieee;
use ieee.std_logic_1164.all;

entity SEVEN_SEGMENT_D1 is
```

```
	port(
		grygorczuk_input_3, grygorczuk_input_2, grygorczuk_input_1,
grygorczuk_input_0 : in std_logic;
		grygorczuk_output_a, grygorczuk_output_b, grygorczuk_output_c,
grygorczuk_output_d, grygorczuk_output_e, grygorczuk_output_f, grygorczuk_output_g : out
std_logic
	);
end SEVEN_SEGMENT_D1;

architecture SEVEN_SEGMENT_D1_LOGIC of SEVEN_SEGMENT_D1 is
signal AND_0, AND_1, A, AND_2, AND_3, B, C, AND_4, AND_5, AND_6, D, AND_7,
AND_8, AND_9, E, AND_10, AND_11, AND_12, F, AND_13, AND_14, G: std_logic;

begin

AND_0 <= not grygorczuk_input_3 and grygorczuk_input_2 and not grygorczuk_input_0;
AND_1 <= not grygorczuk_input_3 and not grygorczuk_input_2 and not grygorczuk_input_1
and grygorczuk_input_0;
A <= AND_0 or AND_1;

AND_2 <= not grygorczuk_input_3 and grygorczuk_input_2 and not grygorczuk_input_1 and
grygorczuk_input_0;
AND_3 <= not grygorczuk_input_3 and grygorczuk_input_2 and grygorczuk_input_1 and not
grygorczuk_input_0;
B <= AND_2 or AND_3;

C <= not grygorczuk_input_3 and not grygorczuk_input_2 and grygorczuk_input_1 and not
grygorczuk_input_0;

AND_4 <= not grygorczuk_input_2 and not grygorczuk_input_1 and grygorczuk_input_0;
AND_5 <= not grygorczuk_input_3 and grygorczuk_input_2 and not grygorczuk_input_1 and
not grygorczuk_input_0;
AND_6 <= not grygorczuk_input_3 and grygorczuk_input_2 and grygorczuk_input_1 and
grygorczuk_input_0;
D <= AND_4 or AND_5 or AND_6;

AND_7 <= not grygorczuk_input_3 and grygorczuk_input_0;
AND_8 <= not grygorczuk_input_2 and not grygorczuk_input_1 and grygorczuk_input_0;
AND_9 <= not grygorczuk_input_3 and grygorczuk_input_2 and not grygorczuk_input_1;
E <= AND_7 or AND_8 or AND_9;

AND_10 <= not grygorczuk_input_3 and not grygorczuk_input_2 and grygorczuk_input_0;
AND_11 <= not grygorczuk_input_3 and not grygorczuk_input_2 and grygorczuk_input_1;
AND_12 <= not grygorczuk_input_3 and grygorczuk_input_1 and grygorczuk_input_0;
F <= AND_10 or AND_11 or AND_12;

AND_13 <= not grygorczuk_input_3 and not grygorczuk_input_2 and not grygorczuk_input_1;
```

```
AND_14 <= not grygorczuk_input_3 and grygorczuk_input_2 and grygorczuk_input_1 and
grygorczuk_input_0;
G <= AND_13 or AND_14;

grygorczuk_output_a <= A;
grygorczuk_output_b <= B;
grygorczuk_output_c <= C;
grygorczuk_output_d <= D;
grygorczuk_output_e <= E;
grygorczuk_output_f <= F;
grygorczuk_output_g <= G;

end SEVEN_SEGMENT_D1_LOGIC;
```

Code Seven Segment Display 0-9

The seven segment display is made to display numbers zero through nine Figure 35 shows the waveform of this system and Tabel 6 confirms the simulation.

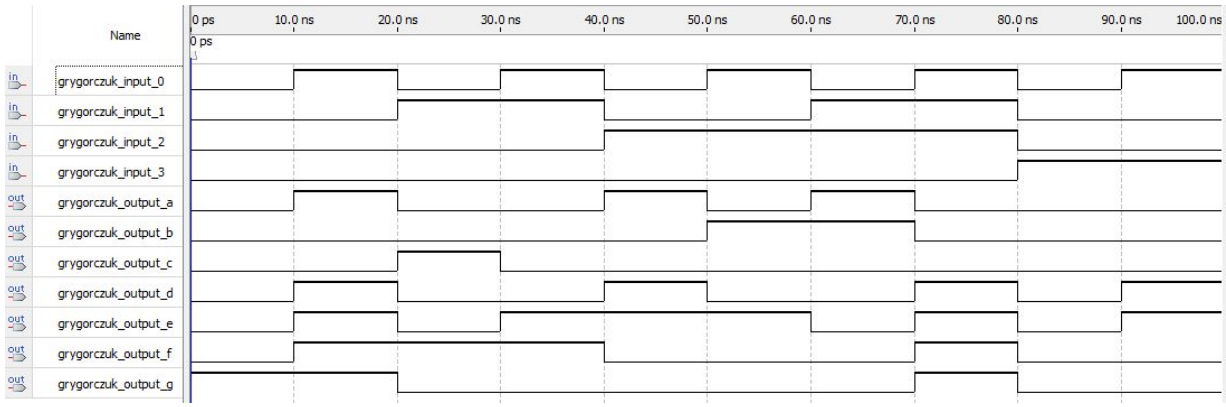

Figure 35. Inside of the Seven Segment Display Waveform

	i3	i2	i1	i0	a	b	c	d	e	f	g
0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	1	1	0	0	1	1	1	1
2	0	0	1	0	0	0	1	0	0	1	0
3	0	0	1	1	0	0	0	0	1	1	0
4	0	1	0	0	1	0	0	1	1	0	0
5	0	1	0	1	0	0	0	0	1	0	0

6	0	1	1	0	1	0	0	0	1	0	0
7	0	1	1	1	0	0	0	1	0	1	1
8	1	0	0	0	0	1	0	0	1	0	0
9	1	0	0	1	0	1	0	1	1	0	0

Tabel 6. Inside of the Seven Segment Display Truth Table

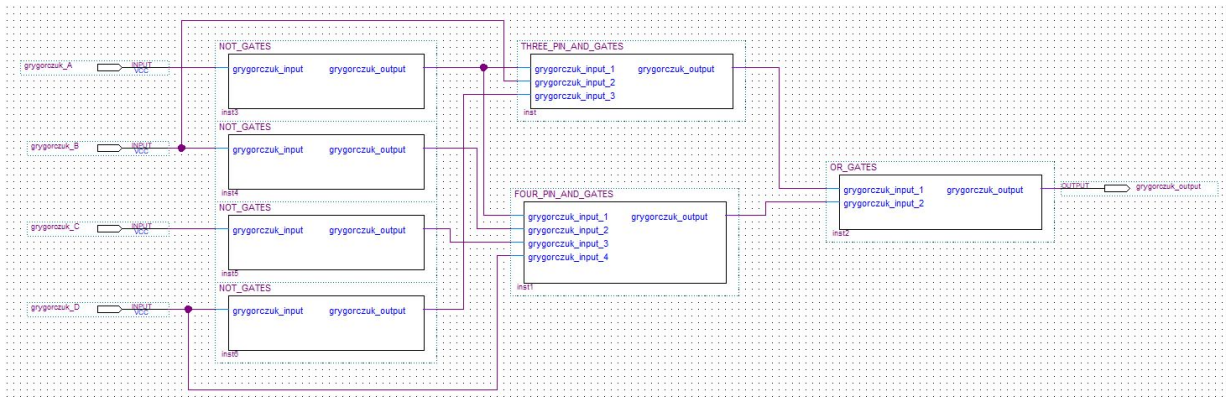

Figure 36.1 Output A System

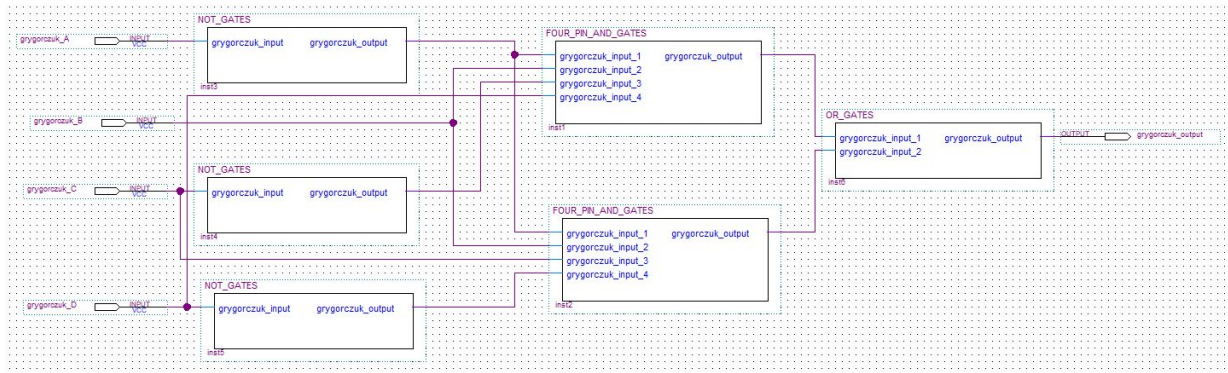

Figure 36.2 Output B System

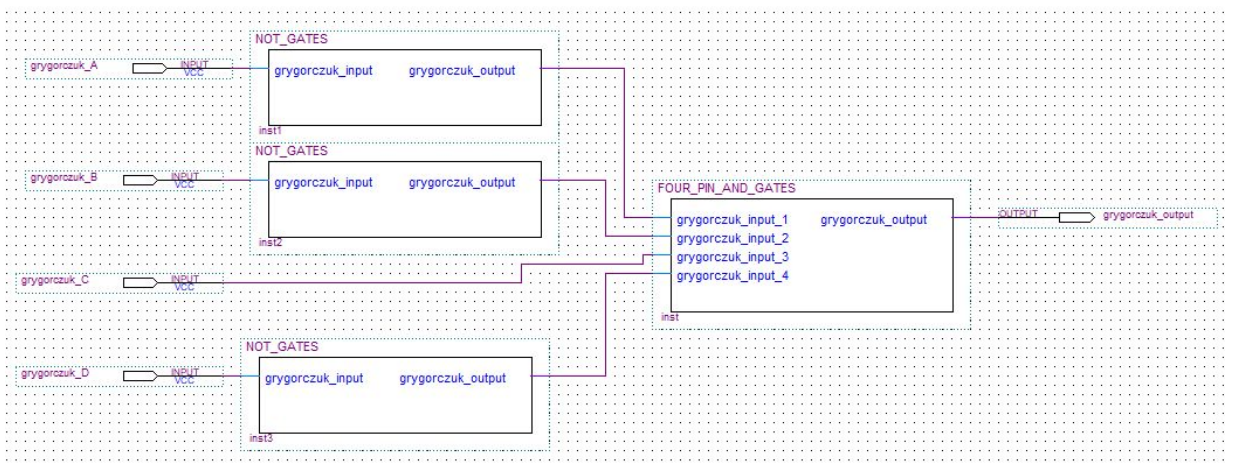

Figure 36.3 Output C System

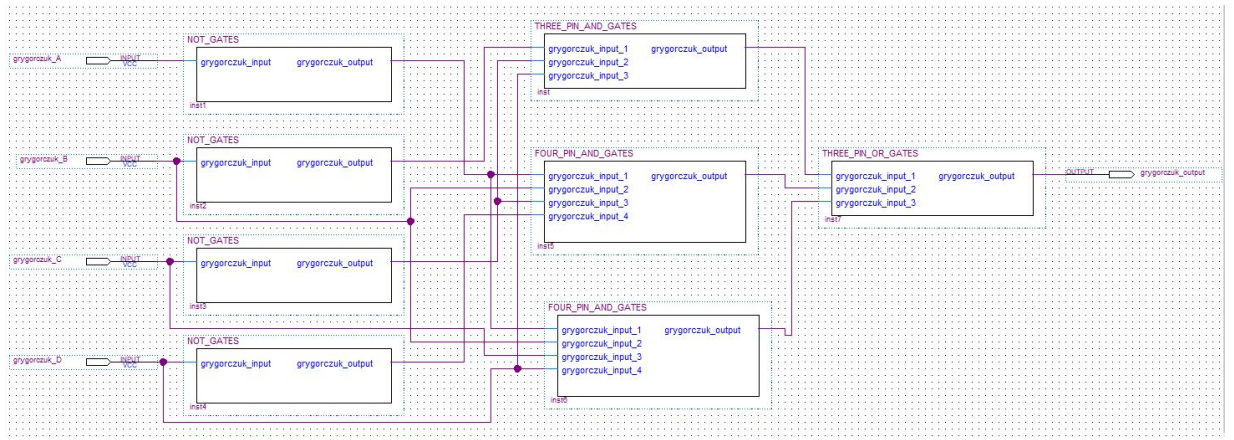

Figure 36.4 Output D System

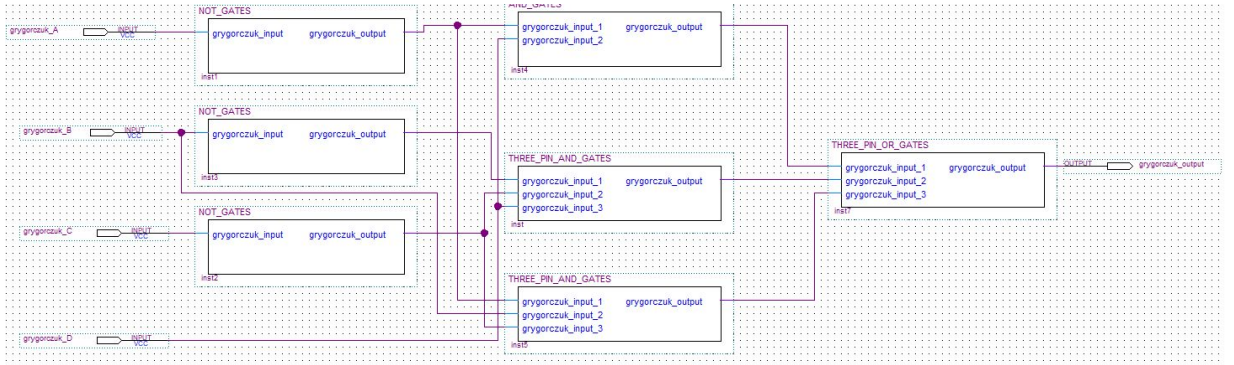

Figure 36.5 Output E System

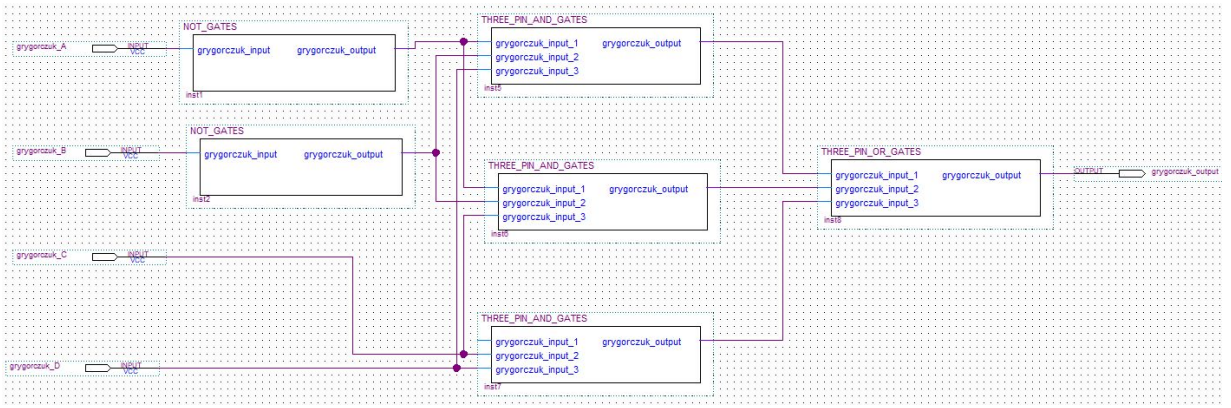

Figure 36.6 Output F System

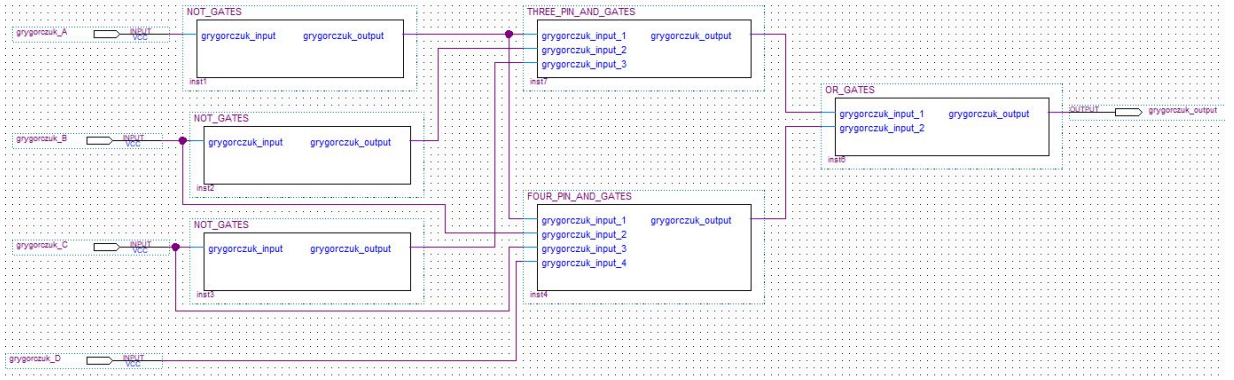

Figure 36.7 Output G System

Part 10: Board

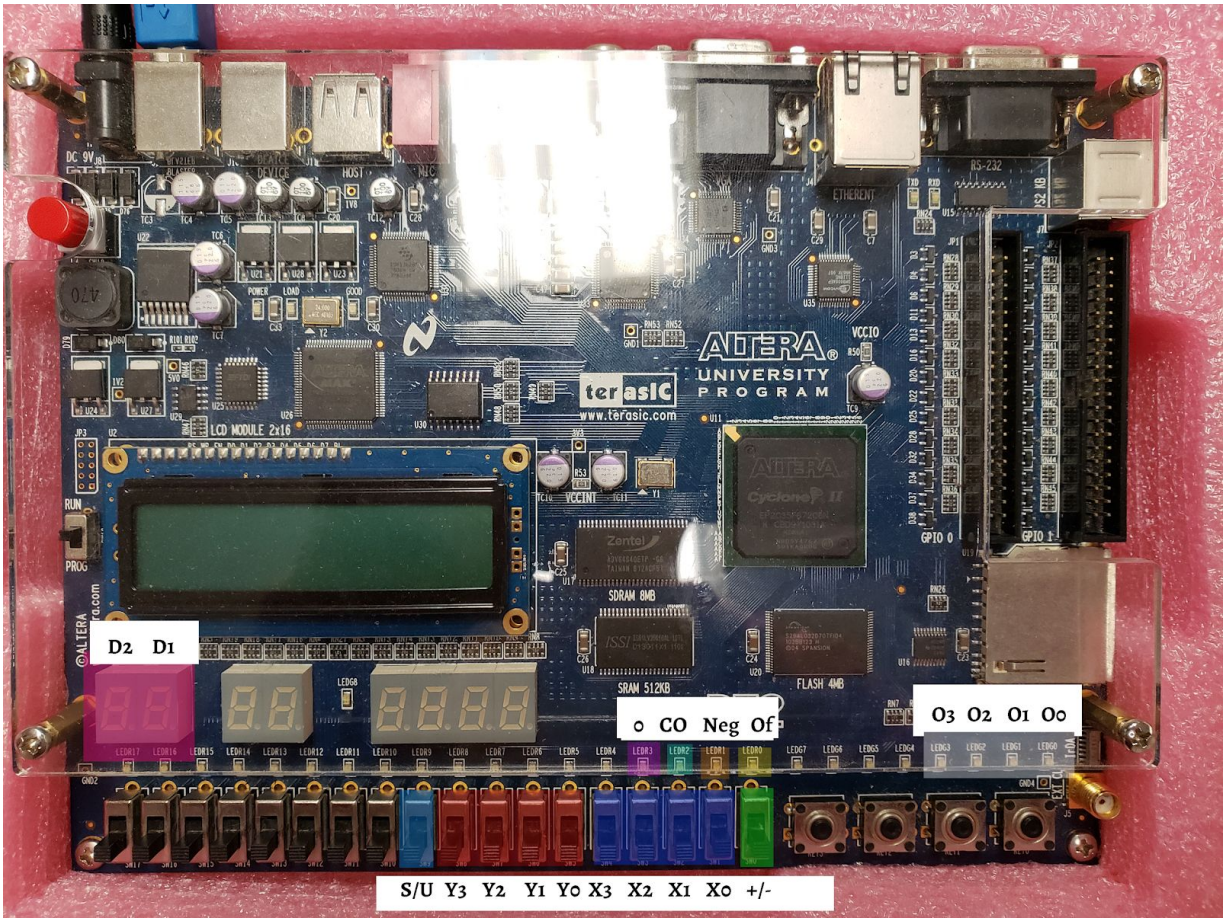

Figure 37. Board and how it's programed

Above is the board with the information on how it's programed and below is the input code used to assign the pins to the board layout.

```
To, Location
grygorczuk_carry_in, PIN_N26
grygorczuk_input_x0, PIN_N25
grygorczuk_input_x1 PIN_P25
grygorczuk_input_x2, PIN_AE14
grygorczuk_input_x3, PIN_AF14
grygorczuk_input_y0, PIN_AD13
grygorczuk_input_y1, PIN_AC13
grygorczuk_input_y2, PIN_C13
grygorczuk_input_y3, PIN_B13
grygorczuk_sign, PIN_A13
```

```
grygorczuk_output_a1, PIN_L3
grygorczuk_output_b1, PIN_L2
grygorczuk_output_c1, PIN_L9
grygorczuk_output_d1, PIN_L6
grygorczuk_output_e1, PIN_L7
grygorczuk_output_f1, PIN_P9
grygorczuk_output_g1, PIN_N9
grygorczuk_output_a0, PIN_R2
grygorczuk_output_b0, PIN_P4
grygorczuk_output_c0, PIN_P3
grygorczuk_output_d0, PIN_M2
grygorczuk_output_e0, PIN_M3
grygorczuk_output_f0, PIN_M5
grygorczuk_output_g0, PIN_M4
grygorczuk_overflow_flag, PIN_AE23
grygorczuk_negative_flag, PIN_AF23
grygorczuk_zero_flag, PIN_AB21
grygorczuk_carry_out_flag, PIN_AC22
grygorczuk_binary_display_LSB, PIN_AE22
grygorczuk_binary_display_MSB_2, PIN_AF22
grygorczuk_binary_display_MSB_1, PIN_W19
grygorczuk_binary_display_MSB, PIN_V18
```

Code. Pin Assignment

Part 11: Results

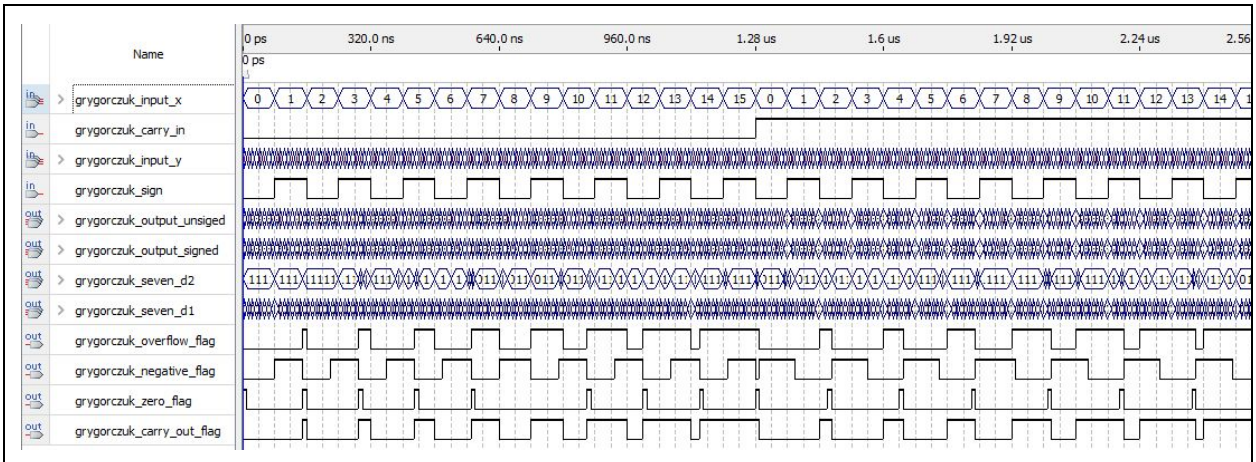

Figure 37. Output Waveform

Below the results for the circuit all put together, in Figure 37 we see a large collection of test function and in Figure 38 is a close on on roughly middle of the waveform with the tables 7 and 8 showing results.

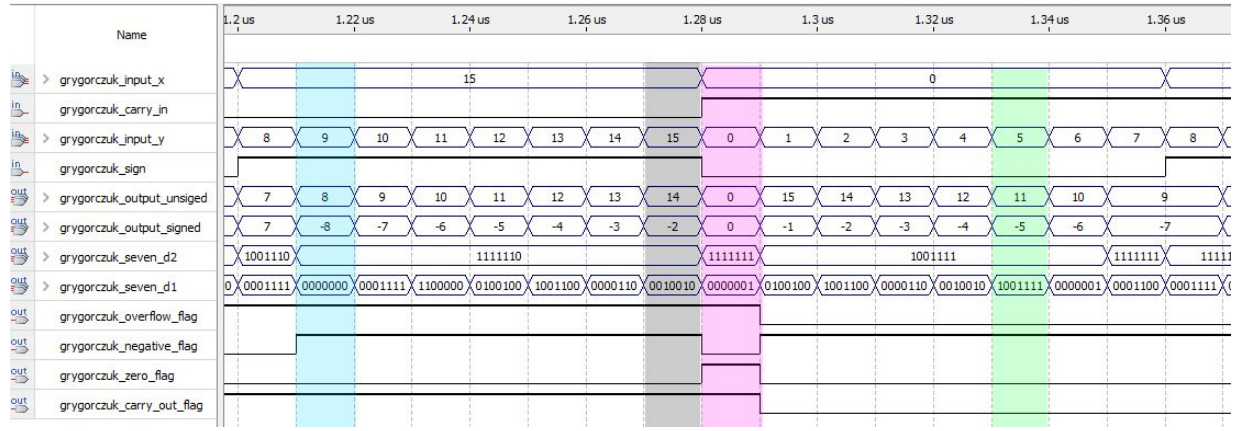

Figure 38. Output Waveform Close up

000 0001	0
100 1111	1
001 0010	2
000 0110	3
100 1100	4
010 0100	5
110 0000	6
000 1111	7
000 0000	8
110 0000	9
100 1110	+
111 1110	-
111 1111	BLANK

Tabel 7. Translation for Seven Segment Display

	Blue	Gray	Pink	Green
Input X	15	15	0	0
Add/Sub	+	+	-	-
Input Y	9	15	0	5
Signed/ Unsigned	Signed	Signed	Unsigned	Unsigned
Display 2	-	-	BLANK	1
Display 1	8	2	0	1
Overflow	1	1	1	0
Negative	1	1	0	1
Zero	0	0	1	0
Carry Out	1	1	1	0

Tabel 8. Results of Close Up

With this confirmation I set up the list to assign the pins using the list shown below.

Part 12: Conclusion

In this lab I've learned how to use the Quaruts system and how to create block diagrams in conjunction with how to code in VHDL. This lab worked as a great refresher on the design process, going from truth table to boolean function to implementation through either the block diagram or VHDL, to verifying this information using waveform diagrams. Now more specifically I've learned binary arithmetic, two's complement, and construction of decoder.