

Lab 2 Report

Name: Mert Can Bilgin

Student ID: 2453025

PRELIMINARY WORK AND EXPERIMENTAL RESULTS

2.2.2 2-TO-1 MULTIPLEXER

QUESTION 2.2.2.1

Design a 2-to-1 multiplexer such that a 4-bit signal and its 1's complement will be its inputs. Introduce a *Select* (S) input so that one of the inputs will be the output of the multiplexer. It is sufficient to show the multiplexer symbol (and the symbol of any other gates that you use for the corresponding design) in the answer, and label all the inputs and outputs.

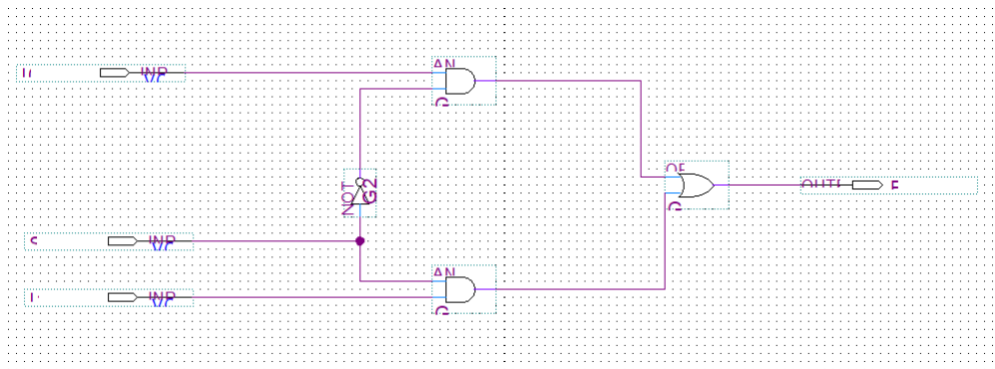


Figure 1: Mux2to1 Schematic

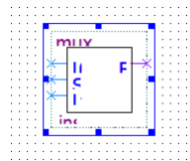


Figure 2: Mux Symbol

2. Write a Verilog code module to implement the multiplexer designed in 2.2.2.1. Take the 1's complement of the input signal inside your Verilog code, i.e. there is no inverter gate available. Please follow the procedural approach of behavioural modelling and use parametric design principles.

```
1.module mux_2x1
2. #(parameter size = 4)(
3. input [size - 1: 0] w0,
4. input s,
5. output reg [size - 1: 0] f
6.);
7. always @(w0,s)
8. begin
9. f = s?~w0:w0;
10. end
11.endmodule
```

Figure 3: Verilog Code of Mux 2-to-1

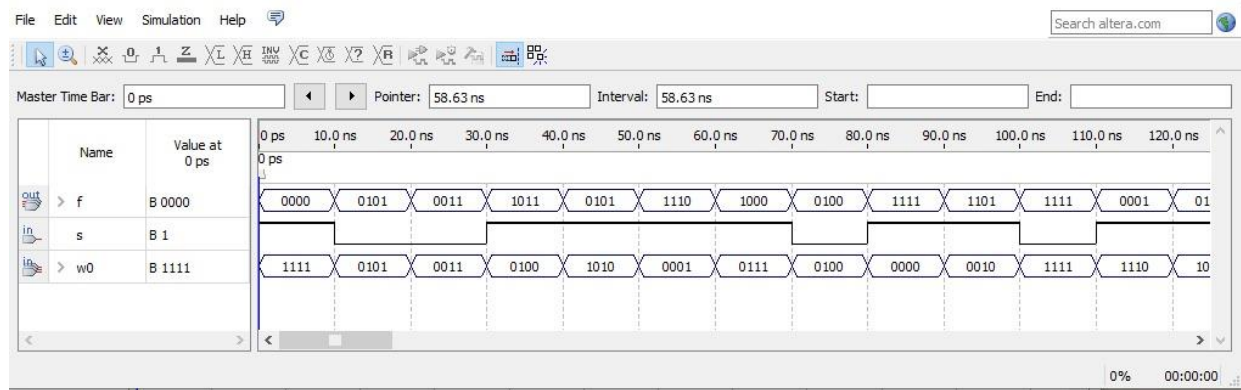


Figure 4: Mux 2-to-1 WaveForm

```

1 module mux_2x1_tb();
2
3     reg [3:0] I;
4     reg S;
5
6     wire [3:0] F;
7
8     mux_2x1 DUT(I, S, F);
9
10    always
11    begin
12        I = 4'b0000; S = 1'b0; #100;
13        I = 4'b0001; S = 1'b0; #100;
14        I = 4'b0010; S = 1'b0; #100;
15        I = 4'b0011; S = 1'b0; #100;
16        I = 4'b0100; S = 1'b0; #100;
17        I = 4'b0101; S = 1'b0; #100;
18        I = 4'b0110; S = 1'b0; #100;
19        I = 4'b0111; S = 1'b0; #100;
20        I = 4'b1000; S = 1'b1; #100;
21        I = 4'b1001; S = 1'b1; #100;
22        I = 4'b1010; S = 1'b1; #100;
23        I = 4'b1011; S = 1'b1; #100;
24        I = 4'b1100; S = 1'b1; #100;
25        I = 4'b1101; S = 1'b1; #100;
26        I = 4'b1110; S = 1'b1; #100;
27        I = 4'b1111; S = 1'b1; #100;
28    end
29 endmodule

```

Figure 5: TestBench of Mux 2-to-1



Figure 6: Simulation of Mux 2-to-1

Comments:

- 2-to-1 Multiplexer selects one of the binary and gives the output according to the selection input. As we can see in the Waveform and Simulation, when selection is 0 we get the same output; however, we get the output as 1's complement of the input. I can be sure that the simulation is true by comparing with the truth table.

2.2.3 4-BIT ADDER/SUBTRACTOR

QUESTION 2.2.3.1

A full Adder has three inputs, A, B, CIN (Carry-IN), and two outputs, SUM and COUT (Carry- OUT). The output represents the result from computing binary addition of the three inputs. Derive a 1-bit Full adder truth-table and write down the logic expressions for SUM and COUT.

A	B	Cin	Sum	cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 7: Truth Table of Full Adder

Binary Expression for Sum:

$$\text{Sum}(A, B, \text{Cin}) = A'B'C + A'BC' + AB'C' + ABC = A \oplus B \oplus C$$

Binary Expression for cout:

$$\text{Cout}(A, B, \text{Cin}) = A'BC + AB'C + ABC' + ABC = BC + AC + AB$$

QUESTION 2.2.3.2

Write a behavioral Verilog code to implement the full adder designed in 2.2.3.1.

```
1 module FullAdder1bit (A,B,Cin,sum,cout) ;
2
3 output sum, cout;
4 input A, B, Cin;
5
6 assign sum = (A ^ B) ^ Cin;
7 assign cout = (B & Cin) | (A & Cin) | (A & B);
8
9 endmodule
10
```

Figure 8: Verilog Code of 1-bit Full Adder

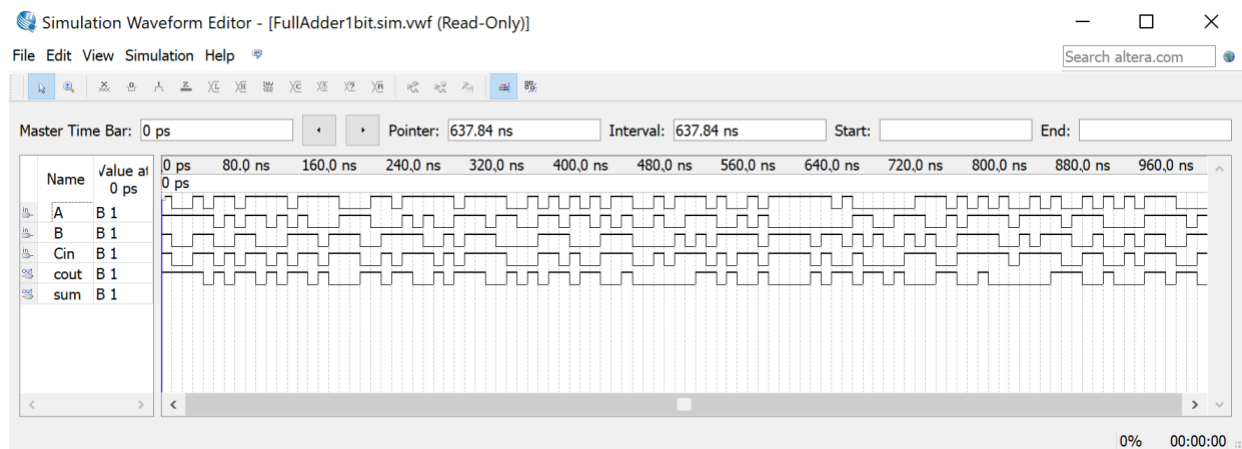


Figure 9: WaveForm of 1-bit Full Adder

```
1 module FullAdder1bit_tb();
2
3 reg A, B, Cin;
4 wire sum, cout;
5
6 FullAdder1bit DUT(A, B, Cin, sum, cout);
7
8 initial
9 begin
10 A = 1'b0; B = 1'b0; Cin = 1'b0; #100;
11 A = 1'b0; B = 1'b0; Cin = 1'b1; #100;
12 A = 1'b0; B = 1'b1; Cin = 1'b0; #100;
13 A = 1'b0; B = 1'b1; Cin = 1'b1; #100;
14 A = 1'b1; B = 1'b0; Cin = 1'b0; #100;
15 A = 1'b1; B = 1'b0; Cin = 1'b1; #100;
16 A = 1'b1; B = 1'b1; Cin = 1'b0; #100;
17 A = 1'b1; B = 1'b1; Cin = 1'b1; #100;
18 end
19 endmodule
20
```

Figure 10: TestBench of 1-bit Full Adder

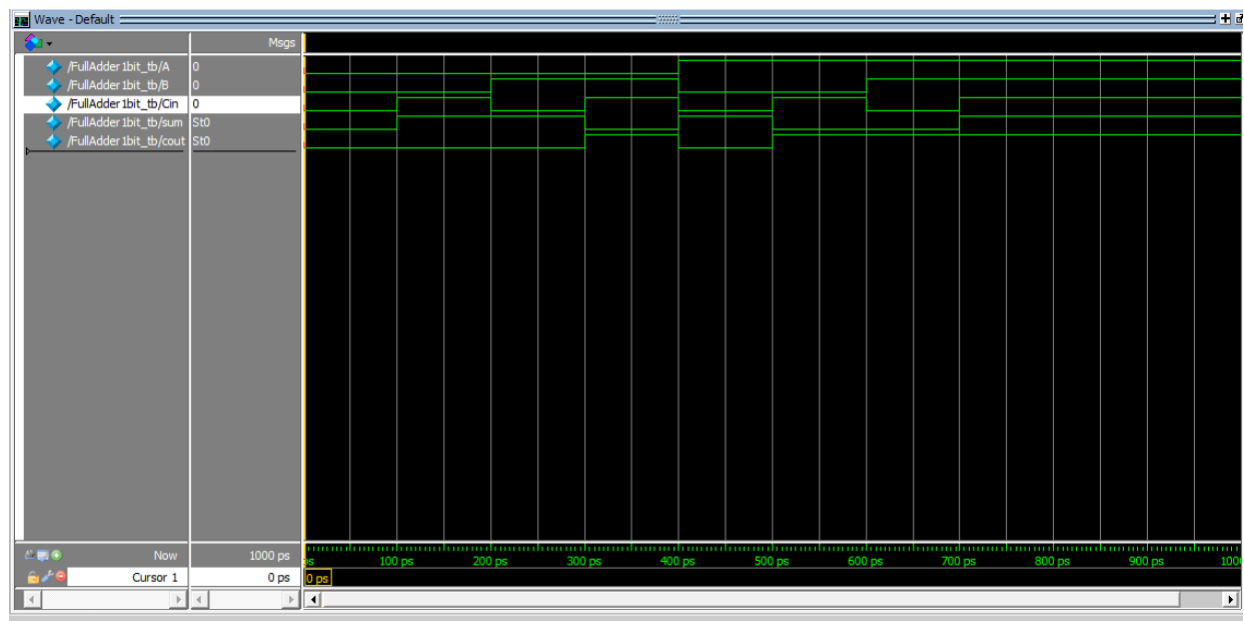


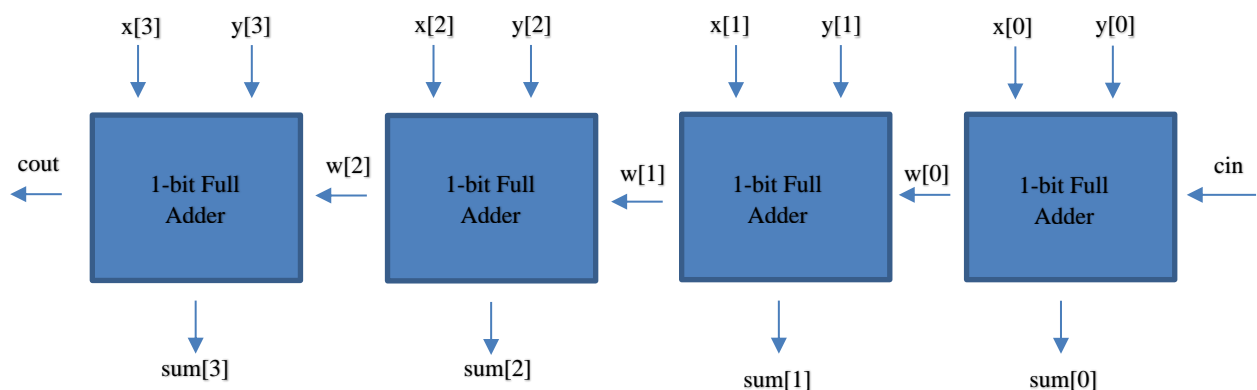
Figure 11: Simulation of 1-bit Full Adder

Comments:

- 1-bit Full Adder takes 2 two 1-bit binary input and a carry-in, and it gives the sum with carry-out.
- It sums the two binary inputs for us.
- I can be sure that my results are correct by comparing with the truth table.

QUESTION 2.2.3.3

Use the symbol in Figure 3(a) for 1-bit Full-Adder and indicate how would you interconnect four such adders in order to build a 4-bit Ripple-Carry-Adder for which a symbol is depicted in figure 3(b).



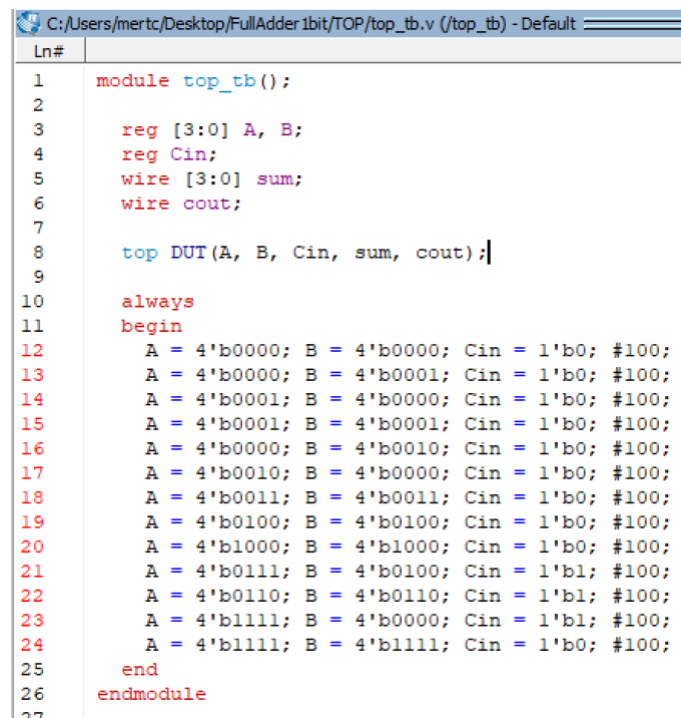
I used four 1-bit Adder to build 4-bit Ripple-Carry-Adder. After the first full adder, I set the input as wires instead of Cin. I connected them with these wires, and all 1-bit sums give the result at the end.

QUESTION 2.2.3.4

Write a parametrized Verilog code using 2.2.3.2 to implement 4-bit ripple carry adder.

```
1  module top(A, B, Cin, sum, cout);
2
3      parameter size = 4;
4
5      input [size - 1: 0] A, B;
6      input Cin;
7
8      output [size - 1: 0] sum;
9      output cout;
10
11     wire [size - 2: 0] W;
12
13     genvar i;
14     generate
15         for(i = 0; i < size; i = i + 1) begin: top
16             if(i == 0)
17                 FullAdder1bit U1(A[i], B[i], Cin, sum[i], W[i]);
18             else if(i == size-1)
19                 FullAdder1bit U2(A[i], B[i], W[i - 1], sum[i], cout);
20             else
21                 FullAdder1bit U3(A[i], B[i], W[i - 1], sum[i], W[i]);
22             end
23         endgenerate
24     endmodule
```

Figure 12: Parameterized Verilog Code of 4-bit Full Adder



The screenshot shows a Verilog testbench file named top_tb.v. The code defines a module top_tb() that instantiates the 4-bit full adder module top. It sets up registers for inputs A, B, and Cin, and wires for outputs sum and cout. A series of 16 test cases are listed, each with specific binary values for A, B, and Cin, and a delay of 100 time units. The test cases cover all possible combinations of 4-bit inputs and carry-in values.

```
C:/Users/mertc/Desktop/FullAdder1bit/TOP/top_tb.v (/top_tb) - Default
Ln#
1  module top_tb();
2
3      reg [3:0] A, B;
4      reg Cin;
5      wire [3:0] sum;
6      wire cout;
7
8      top DUT(A, B, Cin, sum, cout);
9
10     always
11     begin
12         A = 4'b0000; B = 4'b0000; Cin = 1'b0; #100;
13         A = 4'b0000; B = 4'b0001; Cin = 1'b0; #100;
14         A = 4'b0001; B = 4'b0000; Cin = 1'b0; #100;
15         A = 4'b0001; B = 4'b0001; Cin = 1'b0; #100;
16         A = 4'b0000; B = 4'b0010; Cin = 1'b0; #100;
17         A = 4'b0010; B = 4'b0000; Cin = 1'b0; #100;
18         A = 4'b0011; B = 4'b0011; Cin = 1'b0; #100;
19         A = 4'b0100; B = 4'b0100; Cin = 1'b0; #100;
20         A = 4'b1000; B = 4'b1000; Cin = 1'b0; #100;
21         A = 4'b0111; B = 4'b0100; Cin = 1'b1; #100;
22         A = 4'b0110; B = 4'b0110; Cin = 1'b1; #100;
23         A = 4'b1111; B = 4'b0000; Cin = 1'b1; #100;
24         A = 4'b1111; B = 4'b1111; Cin = 1'b0; #100;
25     end
26 endmodule
27
```

Figure 13: TestBench of 4-bit Full Adder

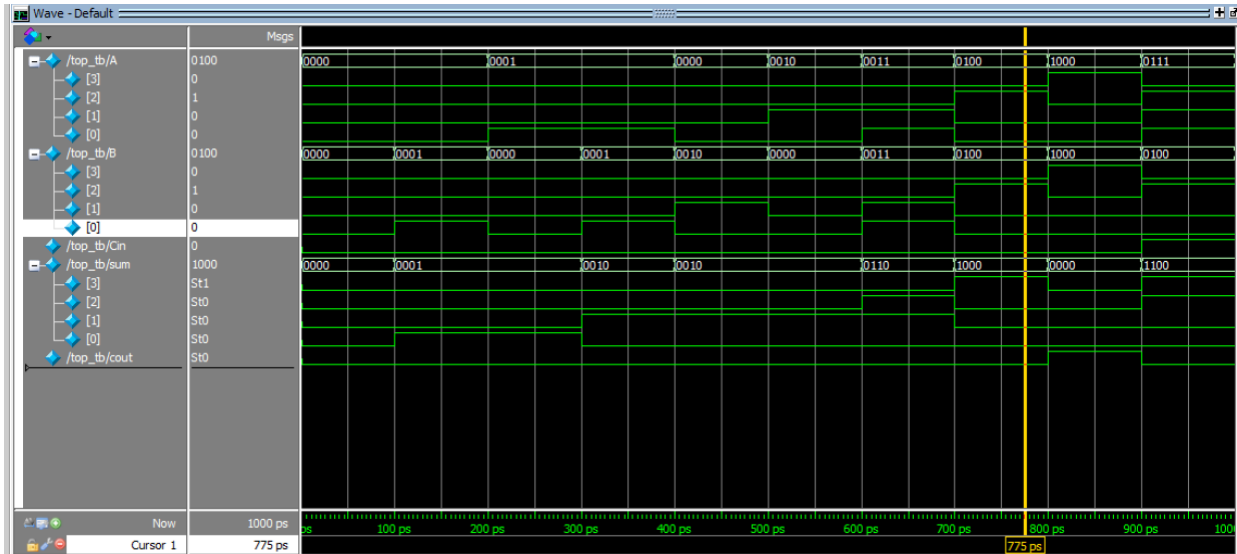


Figure 14: Simulation of 4-bit Full Adder

Comments:

- 1-bit Full adder take the sum of two 1-bit binary number.
- 4-bit Full adder take the sum of two 4-bit binary number. I used hierarchical style in my design, so there are four 1-bit Full Adder in the 4-bit Full adder.
- I can ensure that my simulation results are true by adding the numbers one by one, or comparing with the truth table. For example, in 700-800ps A is 0100 and B is 0100 and the sum of them is 1000 with cout 0.

QUESTION 2.2.3.5

Use structural (hierarchical) design principles and modify the Verilog code in 2.2.3.4 in order to create a 4-bit adder/subtractor circuit. Please note that instead of “CIN” signal (Carry-IN) used in 2.2.3.4, use “OP” (operation) signal as depicted in Figure 4. This will make it easier to get the 2’s complement of a number when it is necessary. Note that when the “OP” signal changes, the operation performed also changes. Also, please consider that “OP” signal also affects one of the input signals coming to the addition/subtraction unit.

```

1 module AdderSubtractor(A, B, OP, sum, cout);
2
3     parameter size = 4;
4
5     input [size - 1: 0] A, B;
6     input OP;
7
8     output [size - 1: 0] sum;
9     output cout;
10
11     wire [size - 1: 0] W;
12     wire [size - 1: 0] middleW;
13
14     genvar i;
15     generate
16     for(i = 0; i < size; i = i + 1) begin: AdderSubtractor
17         if(i == 0)
18             begin
19                 mux21 U4(B[i], OP, W[i]);
20                 FullAdder1bit U1(A[i], W[i], OP, sum[i], middleW[i]);
21             end
22         else if(i == size-1)
23             begin
24                 mux21 U5(B[i], OP, W[i]);
25                 FullAdder1bit U2(A[i], W[i], middleW[i - 1], sum[i], cout);
26             end
27         else
28             begin
29                 mux21 U6(B[i], OP, W[i]);
30                 FullAdder1bit U3(A[i], W[i], OP, sum[i], middleW[i]);
31             end
32         end
33     endgenerate
34 endmodule
35
36
37

```

Figure 15: Verilog Code of Adder/Subtractor

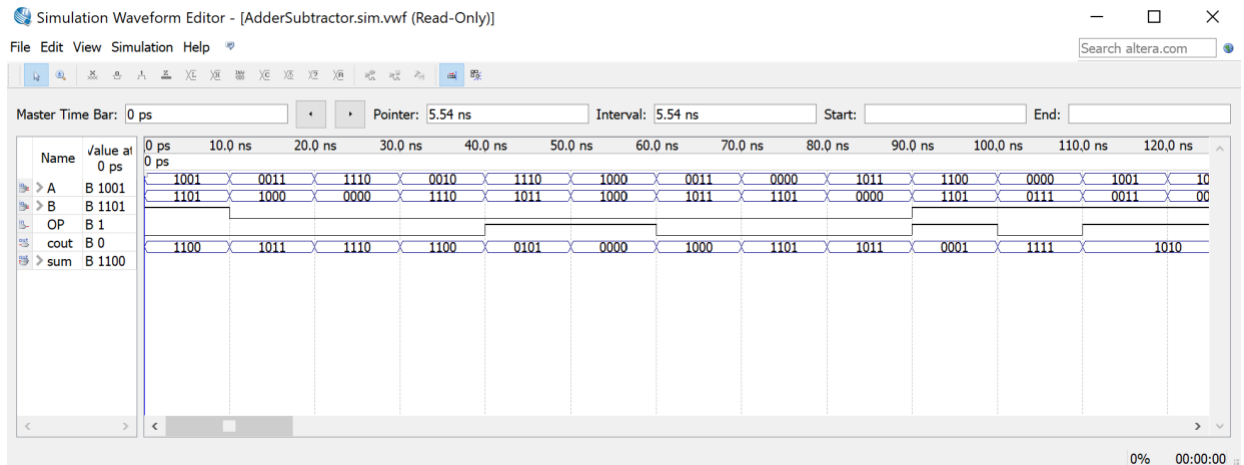


Figure 16: Waveform of Adder/Subtractor

```

1 module AdderSubtractor_tb();
2
3     reg [3:0] A, B;
4     reg OP;
5     wire [3:0] sum;
6     wire cout;
7
8     AdderSubtractor DUT(A, B, OP, sum, cout);
9
10    always
11    begin
12        A = 4'b0000; B = 4'b0000; OP = 1'b0; #100;
13        A = 4'b0000; B = 4'b0001; OP = 1'b0; #100;
14        A = 4'b0001; B = 4'b0000; OP = 1'b0; #100;
15        A = 4'b0001; B = 4'b0001; OP = 1'b0; #100;
16        A = 4'b0000; B = 4'b0010; OP = 1'b0; #100;
17        A = 4'b0010; B = 4'b0000; OP = 1'b0; #100;
18        A = 4'b0011; B = 4'b0011; OP = 1'b0; #100;
19        A = 4'b0100; B = 4'b0100; OP = 1'b0; #100;
20        A = 4'b1000; B = 4'b1000; OP = 1'b0; #100;
21        A = 4'b0111; B = 4'b0100; OP = 1'b1; #100;
22        A = 4'b0110; B = 4'b0110; OP = 1'b1; #100;
23        A = 4'b1111; B = 4'b0000; OP = 1'b1; #100;
24        A = 4'b1111; B = 4'b1111; OP = 1'b0; #100;
25    end
26 endmodule

```

Figure 17: TestBench of Adder/Subtractor

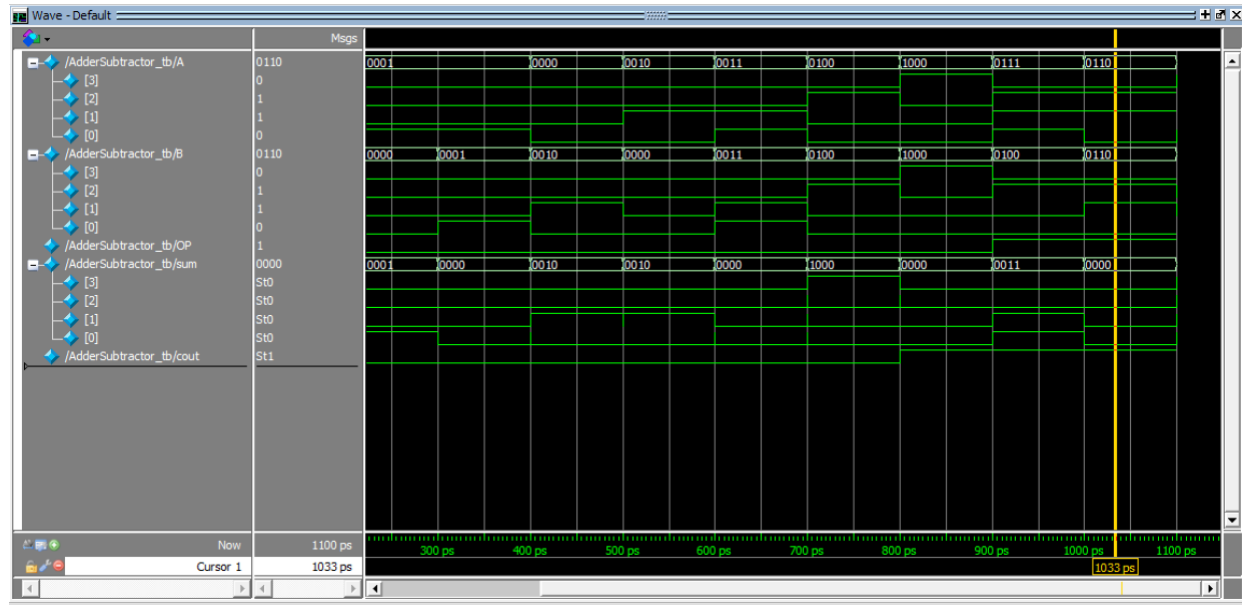


Figure 18: Simulation of Adder/Subtractor

Comments:

- If the operation signal is '1', Adder/Subtractor operates subtraction. Otherwise, it operates addition. It completes the subtraction by taking the 2's complement of the second number B. To taking the 2's complement of a number I used a multiplexer as we seen in the Lab Manuel.
- I can be sure that my simulation results are true by comparing the results with the truth table, or operation the process by hand. For instance, A is 0110 and B is 0110 between 1000ps and 1100ps. Also the OP is 1, so we need to operate subtraction. If we subtract 0110 from 0110, we get 0000. We can see the same result in the simulation.

2.2.4 4-BIT COMPARATOR

QUESTION 2.2.4.1

A 2-bit comparator has two inputs A1, A0 and B1, B0 and three outputs, A<B, A>B, A = B. Derive a 2-bit comparator truth-table and using K-MAPs write down the simplified logic expressions for A<B, A>B, A = B.

A1	A0	B1	B0	A < B	A > B	A = B
0	0	0	0	0	0	1
0	0	0	1	1	0	0
0	0	1	0	1	0	0
0	0	1	1	1	0	0
0	1	0	0	0	1	0
0	1	0	1	0	0	1
0	1	1	0	1	0	0
0	1	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	0
1	0	1	0	0	0	1
1	0	1	1	1	0	0
1	1	0	0	0	1	0
1	1	0	1	0	1	0
1	1	1	0	0	1	0
1	1	1	1	0	0	1

A[1]A[0] \ B[1]B[0]					
		00	01	10	11
00	0	1	1	1	
01	0	0	1	1	
10	0	0	0	0	
11	0	0	1	0	

A < B

⇒

Expression for $A < B$

$$A[1]'B[1] + A[0]'B[1]B[0] + A[1]'A[0]'B[0]$$

		B[1]B[0]			
		00	01	10	11
A[1]A[0]	00	0	0	0	0
	01	1	0	0	0
	10	1	1	0	1
	11	1	1	0	0

A > B

⇒

Expression for $A > B$

$$A[1]B[1]' + A[0]B[1]'B[0]' + A[1]A[0]B[0]'$$

		B[1]B[0]			
		00	01	10	11
A[1]A[0]	00	1	0	0	0
	01	0	1	0	0
	10	0	0	1	0
	11	0	0	0	1

A = B

⇒

Expression for $A = B$

$$A[1]'A[0]'B[1]'B[0]' + A[1]'A[0]B[1]'B[0] + A[1]A[0]B[1]B[0] + A[1]A[0]'B[1]B[0]'$$

It can be simplified as:

$$A[1]'B[1]'(A[0]'B[0]' + A[0]B[0]) + A[1]B[1](A[0]B[0] + A[0]'B[0]')$$

Then,

$$(A[0]B[0] + A[0]'B[0]')(A[1]B[1] + A[1]'B[1]')$$

QUESTION 2.2.4.2

Write a parametrized Verilog code to implement 4-bit comparator module.

```
1 module Comparator(A, B, LT, GT, EQ);
2
3   parameter size = 4;
4
5   input [size - 1: 0] A, B;
6   output LT, GT, EQ;
7   reg LT, GT, EQ;
8
9   always@ (A,B)
10  begin
11    if (A > B)
12    begin
13      LT = 1'b0;
14      EQ = 1'b0;
15      GT = 1'b1;
16    end
17
18    else if (A == B)
19    begin
20      LT = 1'b0;
21      EQ = 1'b1;
22      GT = 1'b0;
23    end
24
25    else
26    begin
27      LT = 1'b1;
28      EQ = 1'b0;
29      GT = 1'b0;
30    end
31  end
32 endmodule
33
34
```

Figure 19: Verilog Code of 4-bit Comparator

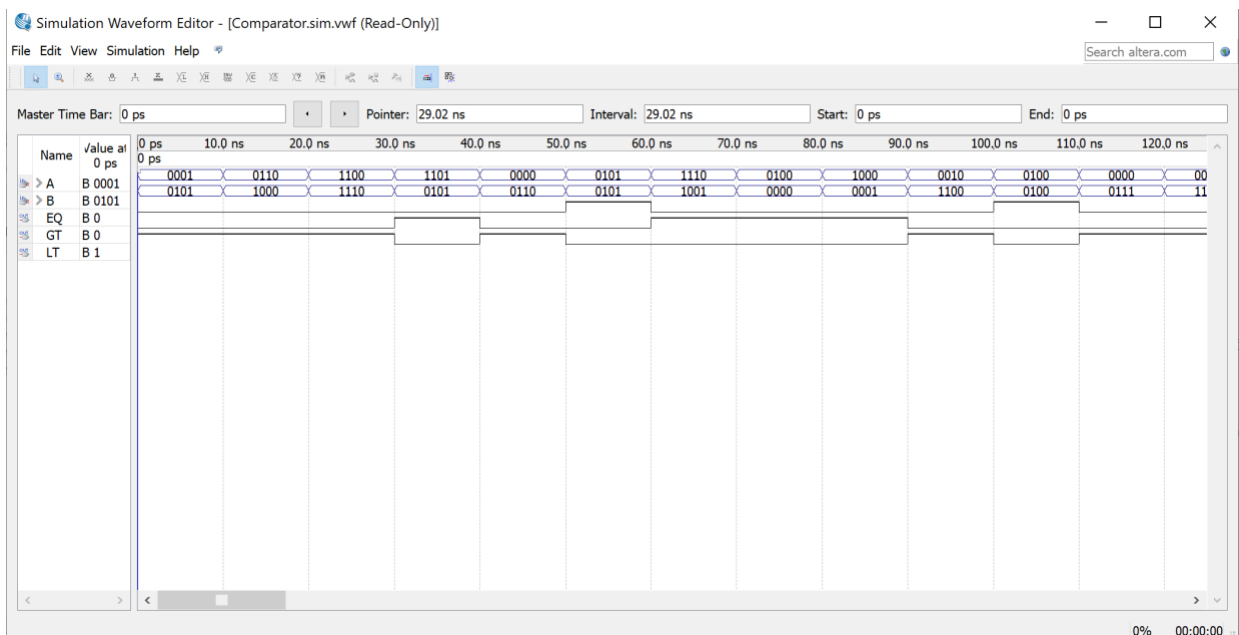


Figure 20: Waveform of 4-bit Comparator

QUESTION 2.2.4.3

Write a testbench for your design using Modelsim® and simulate your comparator Verilog code.

```
1 module Comparator_tb();
2
3 reg [3:0] A, B;
4 wire LT, GT, EQ;
5
6 Comparator DUT(A, B, LT, GT, EQ);
7
8 always
9 begin
10     A = 4'b0000; B = 4'b0000; #100;
11     A = 4'b0000; B = 4'b0001; #100;
12     A = 4'b0001; B = 4'b0000; #100;
13     A = 4'b0001; B = 4'b0001; #100;
14     A = 4'b0000; B = 4'b0010; #100;
15     A = 4'b0010; B = 4'b0000; #100;
16     A = 4'b0011; B = 4'b0011; #100;
17     A = 4'b0100; B = 4'b0100; #100;
18     A = 4'b1000; B = 4'b1000; #100;
19     A = 4'b0111; B = 4'b0100; #100;
20     A = 4'b0110; B = 4'b0110; #100;
21     A = 4'b1111; B = 4'b0000; #100;
22     A = 4'b1111; B = 4'b1111; #100;
23 end
24 endmodule
```

Figure 21: Testbench of 4-bit Comparator

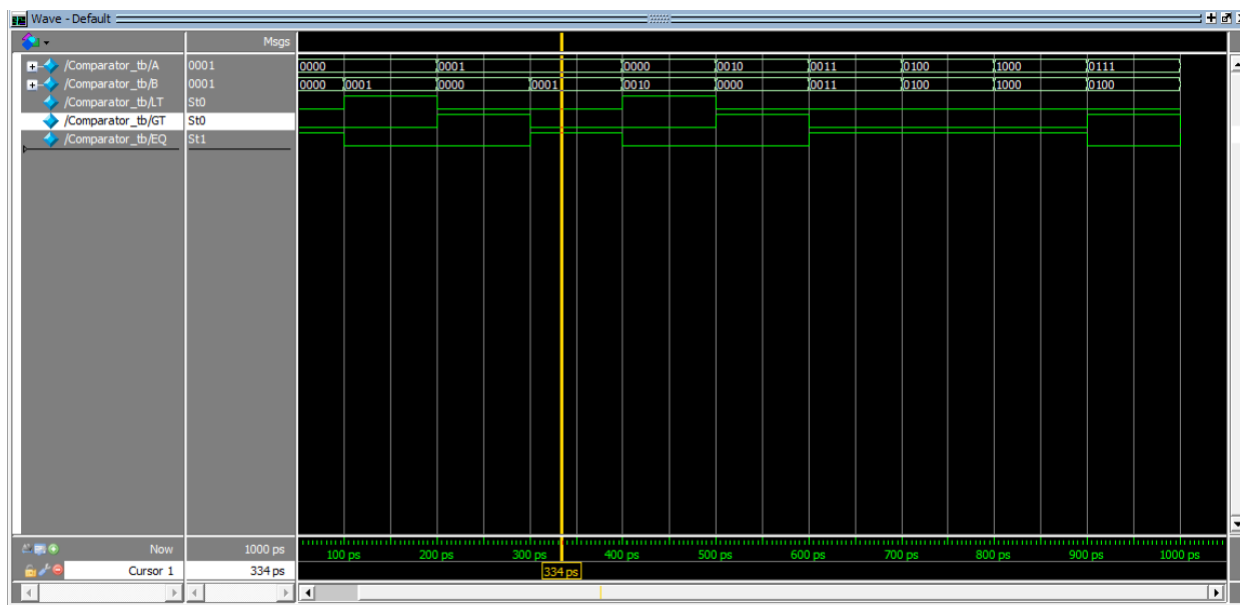


Figure 22: Simulation of 4-bit Comparator

Comments:

- The comparator is used for comparing two binary numbers
- It has eight inputs and three outputs.
- Firstly, I derived the truth table of the 4-bit comparator, then I simplified it by using K-Maps. I implemented verilog code by using them. The code is parameterized, so the size can be changed.
- I can be sure that my results are correct by comparin the simulation results with the truth table. For example between 300ps and 400ps A and B are 0001, they are equal, so we can see that EQ is 1 at this situtation. Also, between 900ps and 1000ps, A is 0111 and B is 0100, A is greater than B, so we can see GT is 1 at that point.

2.2.5 ARITHMETIC UNIT

QUESTION 2.2.5.1

After a 4-bit Adder/Subtractor and a 4-bit Comparator module have been implemented and tested in 2.2.3 and 2.2.4, combine these two modules as shown in Figure 5 in one structural Verilog HDL design named as **ARITHMETIC_UNIT**.

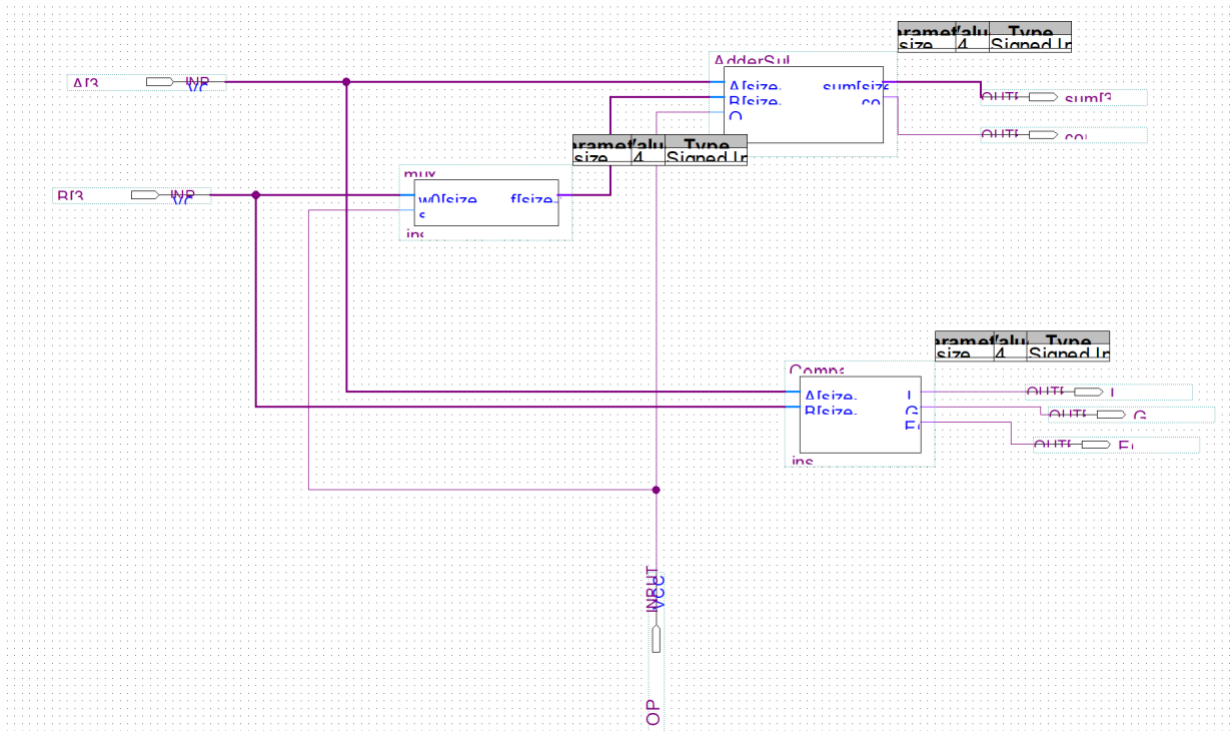


Figure 23: Schematic of Arithmetic Unit

```

mux21.v      FullAdder1bit.v      AdderSubtractor.v
1  module ARITHMETIC_UNIT(A, B, OP, sum, cout, LT, GT, EQ);
2
3  input [3:0] A, B;
4  input OP;
5  output [3:0] sum;
6  output LT, GT, EQ, cout;
7
8  AdderSubtractor U1(A, B, OP, sum, cout);
9  Comparator U2(A, B, LT, GT, EQ);
10
11 endmodule
12

```

Figure 24: Verilog Code of Arithmetic Unit

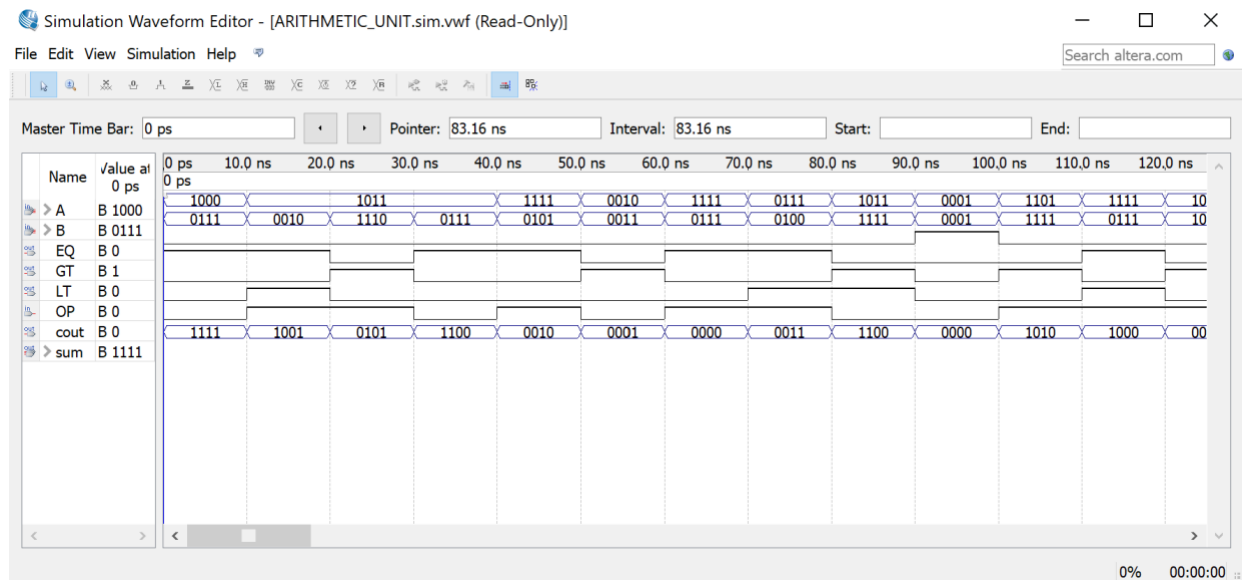


Figure 25: Waveform of Arithmetic Unit

QUESTION 2.2.5.2

Write a testbench for your design using Modelsim® and simulate your arithmetic unit.

```
1  module ARITHMETIC_UNIT_tb();
2
3      reg [3:0] A, B;
4      reg OP;
5      wire [3:0] sum;
6      wire cout, LT, GT, EQ;
7
8      ARITHMETIC_UNIT DUT(A, B, OP, sum, cout, LT, GT, EQ);
9
10     always
11     begin
12         A = 4'b0000; B = 4'b0000; OP = 1'b0; #100;
13         A = 4'b0000; B = 4'b0001; OP = 1'b0; #100;
14         A = 4'b0001; B = 4'b0000; OP = 1'b0; #100;
15         A = 4'b0001; B = 4'b0001; OP = 1'b0; #100;
16         A = 4'b0000; B = 4'b0010; OP = 1'b0; #100;
17         A = 4'b0010; B = 4'b0000; OP = 1'b0; #100;
18         A = 4'b0011; B = 4'b0011; OP = 1'b0; #100;
19         A = 4'b0100; B = 4'b0100; OP = 1'b0; #100;
20         A = 4'b1000; B = 4'b1000; OP = 1'b0; #100;
21         A = 4'b0111; B = 4'b0100; OP = 1'b1; #100;
22         A = 4'b0110; B = 4'b0110; OP = 1'b1; #100;
23         A = 4'b1111; B = 4'b0000; OP = 1'b1; #100;
24         A = 4'b1111; B = 4'b1111; OP = 1'b0; #100;
25     end
26 endmodule
27
28
```

Figure 26: Testbench of Arithmetic Unit

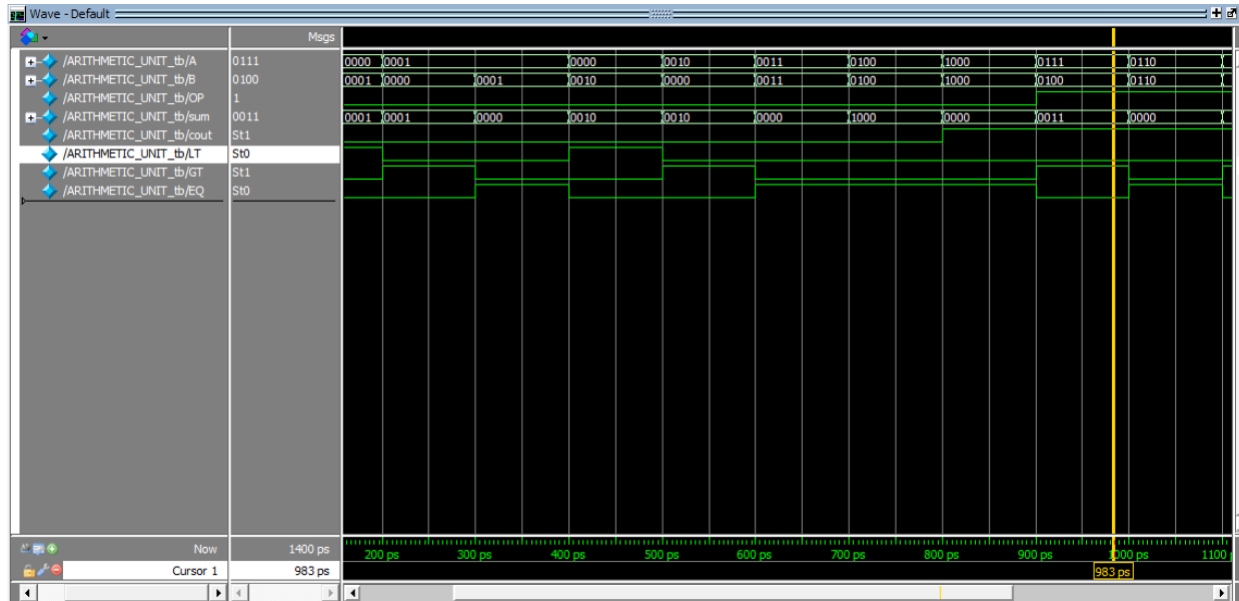


Figure 27: Simulation of Arithmetic Unit

Comments:

- Arithmetic Unit operates the comparison, addition and subtraction.
- It has two 4-bits inputs, one 1-bit OP input and four output which are sum, carry-out, LT, GT and EQ in this experiment.
- I get the arithmetic unit from baby steps in this experiment. Firstly, I created a multiplexer, then I created an 1-bit adder. I used the 1-bit adder to create 4-bit adder. After that, I used the multiplexer to create subtractor. Then, I created a comparator, and I completed the arithmetic unit.
- I can ensure that my simulation results are correct by comparing the simulation results with the truth tables, or operating the operations by hand. For instance, between 900ps and 1000ps, A is 0111 and B is 0100. We see that GT is '1', also arithmetic logic unit should operate subtraction because OP signal is '1'. When we compare the result, $0111 - 0100$ is 0011 as we see in the simulation.

2.2.6 DECODER

QUESTION 2.2.6.1

DE0 board has 4 adjacent 7-Segment Display. By considering this, use combinational logic design principles to design a 7-Segment Display decoding logic such that when a 4-bit signed binary number is entered, the number will show up on the rightmost 7-Segment Display. Also, if the number is negative, the sign will be displayed on the 7-Segment Display next to the one that shows the number (Rightmost two 7-Segment Display will be employed). For example, when the user enters 0001, two of the vertically aligned LEDs of the rightmost 7-Segment Display should light up. Similarly, when the user enters 1110 (-2) in binary, five of the LEDs on the rightmost 7-Segment Display should light up, and the LED located in the middle of the adjacent 7-Segment Display should light up to indicate the number is negative. Please *optimize* your logic as much as possible to yield the lowest cost i.e. the lowest number of input switches, gates, literals, and gate inputs. Input-to-Output delay is *not* a concern in this application.

Hint: Remember each 7-Segment Display on DE0 board is characterized by 8 independent **active low** outputs (including the dot in the lower right) that need to be all defined to determine the displayed character, as depicted in Table 4.

By the help of table 4, I implemented the design as 4-bit input and, 16-bit output. Output[0] will represent the LED A, output[1] will represent the LED B, and so on. I used procedural behavioral modelling in this design. It makes the life easier, because we can only implement what should be happen in specific cases.

QUESTION 2.2.6.2

Write a Verilog code to implement the decoder module designed in Section 2.2.6.1. Please follow the procedural approach of behavioural modelling. For more information about procedural approach, you can refer to your first lab manual or textbook.

```

1  module Decoder(number, LED);
2
3  input  [3:0] number;
4  output [15:0] LED;
5
6  reg [15:0] LED;
7
8  always@ (number)
9  begin
10     case (number)
11         // ABCDEFGHIJKLMNOP
12         4'b1000: LED = 16'b1111110100000001;
13         4'b1001: LED = 16'b1111110100011111;
14         4'b1010: LED = 16'b1111110101000001;
15         4'b1011: LED = 16'b1111110101001001;
16         4'b1100: LED = 16'b1111110110011001;
17         4'b1101: LED = 16'b1111110100001101;
18         4'b1110: LED = 16'b1111110100100101;
19         4'b1111: LED = 16'b1111110110011111;
20         4'b0000: LED = 16'b1111111100000011;
21         // ABCDEFGHIJKLMNOP
22         4'b0001: LED = 16'b1111111110011111;
23         4'b0010: LED = 16'b11111111100100101;
24         4'b0011: LED = 16'b11111111100001101;
25         4'b0100: LED = 16'b11111111110011001;
26         4'b0101: LED = 16'b11111111101001001;
27         4'b0110: LED = 16'b11111111101000001;
28         4'b0111: LED = 16'b11111111100011111;
29         default: LED = 16'b1111111111111111;
30     endcase
31 end
32 endmodule

```

Figure 28: Verilog Code of Decoder

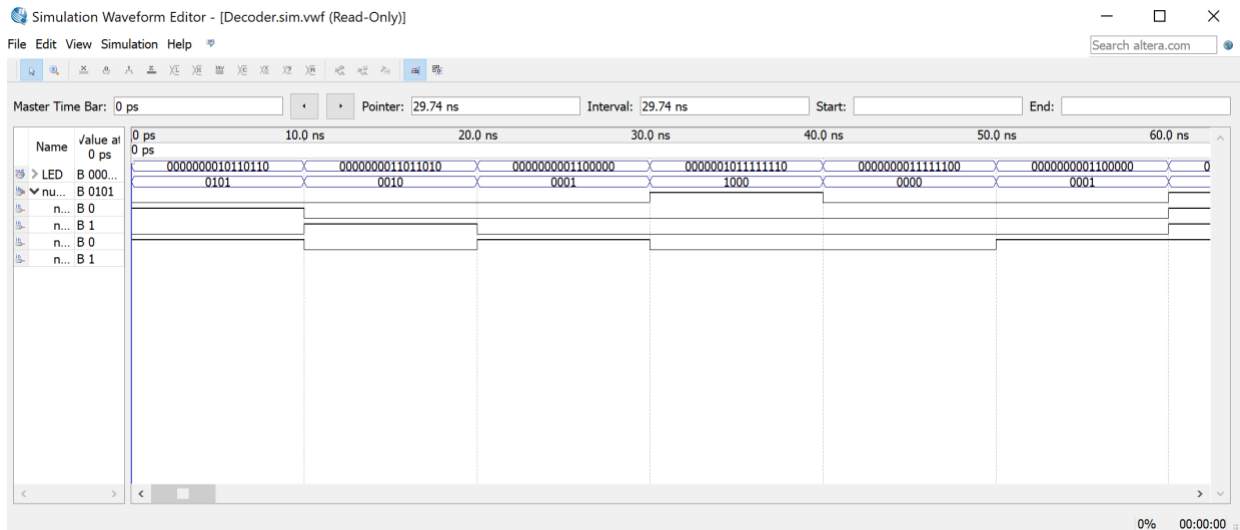


Figure 29: Waveform of Decoder

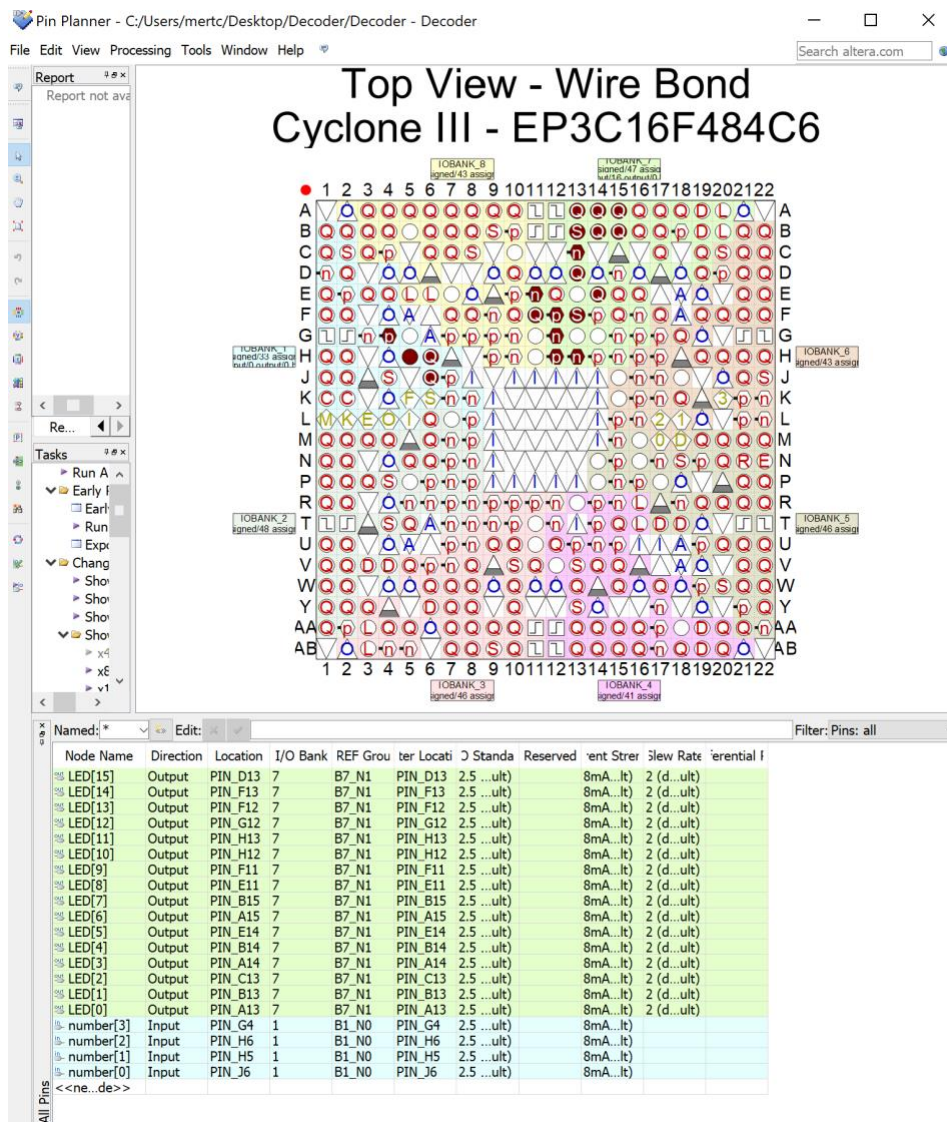


Figure 30: Pin Planner of Decoder

```

C:\Users\mertc\Desktop\Decoder\Modelsim\Decoder_tb.v
Ln#
1  module Decoder_tb();
2
3  reg [3:0] number;
4  wire [15:0] LED;
5
6  Decoder DUT(number, LED);
7
8  always
9  begin
10 number = 4'b1000; #100;
11 number = 4'b1001; #100;
12 number = 4'b1010; #100;
13 number = 4'b1011; #100;
14 number = 4'b1100; #100;
15 number = 4'b1101; #100;
16 number = 4'b1110; #100;
17 number = 4'b1111; #100;
18 number = 4'b0000; #100;
19 number = 4'b0001; #100;
20 number = 4'b0010; #100;
21 number = 4'b0011; #100;
22 number = 4'b0100; #100;
23 number = 4'b0101; #100;
24 number = 4'b0110; #100;
25 number = 4'b0111; #100;
26
27 end
28 endmodule
29

```

Figure 31: TestBench of Decoder

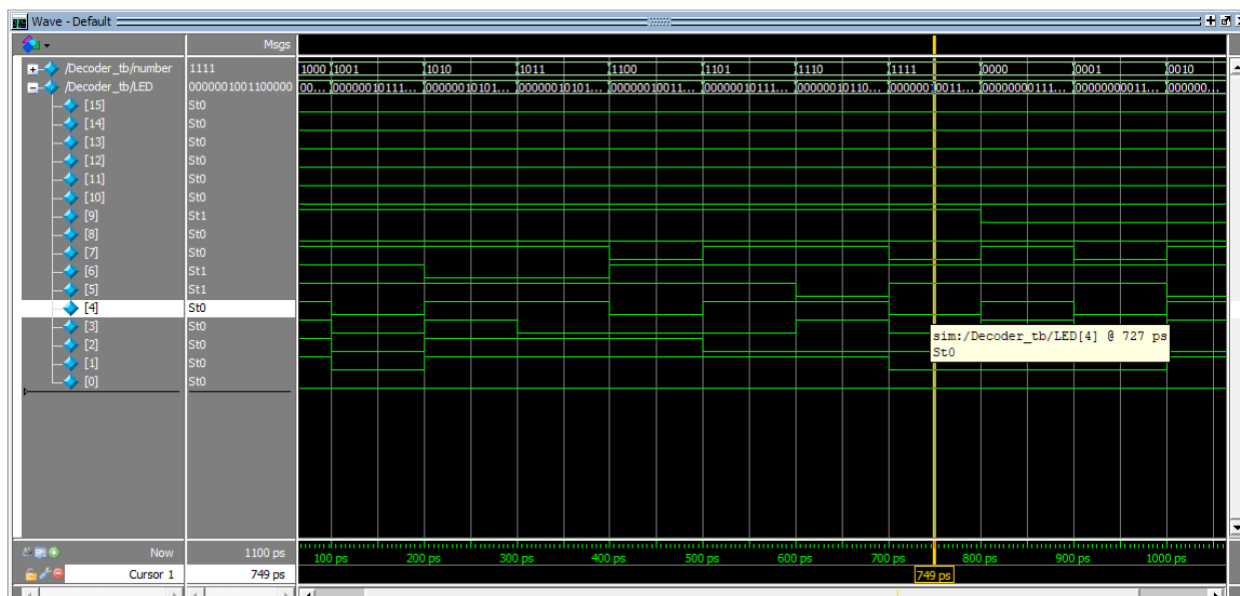


Figure 32: Simulation of Decoder

Comments:

- After designing the decoder, I implemented the verilog code. I have used procedural behavioral approach in my code. It made coding easier. Then, I assigned the pin planner according to the DE0 board.
- When I completing simulation and waveform, I just changed the active lows to active highs temporarily because it makes clearer when I check my results with the simulation.
- I am sure that my results are true by checking the leds from the table in Lab2 manuel. For instance, between 700ps and 800ps the number is 1111 which is -1. When we check the table G, J and K should light, and we can see them in the simulation.