EEE 248/ CNG 232
Logic Design

Project 2

Yıldız Alara Köymen

2453389

TABLE OF CONTENTS

2.1: Introduction:

In this lab we used combinational circuits such as multiplexer, adder, decoder, comparator to design an arithmetic logic unit (ALU).

2.1.1: Objective: What is an ALU?

Arithmetic logic unit is a combinational circuit that performs arithmetic and bitwise operations.

The ALU created for this lab includes operations:

- Add
- Subtract
- Compare
- And
- Or
- Nand
- Nor
- Xor
- Xnor
- Not

2.2 Preliminary Work:

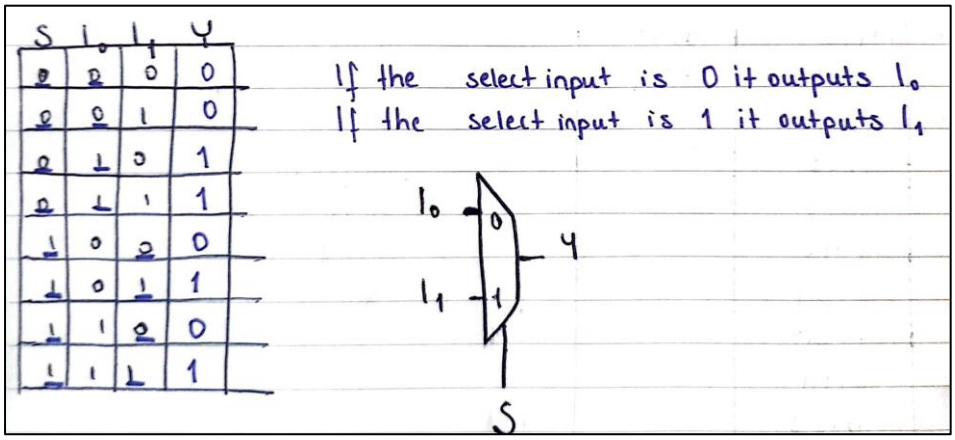
2.2.2: 2-to-1 MUX

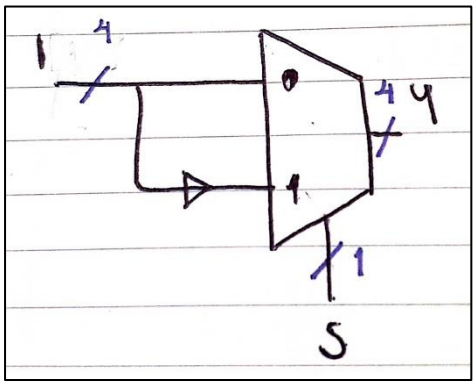
1 Design a 2-to-1 multiplexer such that a 4-bit signal and its 1's complement will be its inputs. Introduce a Select (S) input so that one of the inputs will be the output of the multiplexer. It is sufficient to show the multiplexer symbol (and the symbol of any other gates that you use for the corresponding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

Table 1: 2-to-1 Multiplexer Operation

Select	Multiplexer Output
0	Input [3..0]
1	~Input [3..0]

A multiplxer would output a specific value based on the select(S) input. If select is 0 it would output I0, if its 1 it would output I1 in this case:



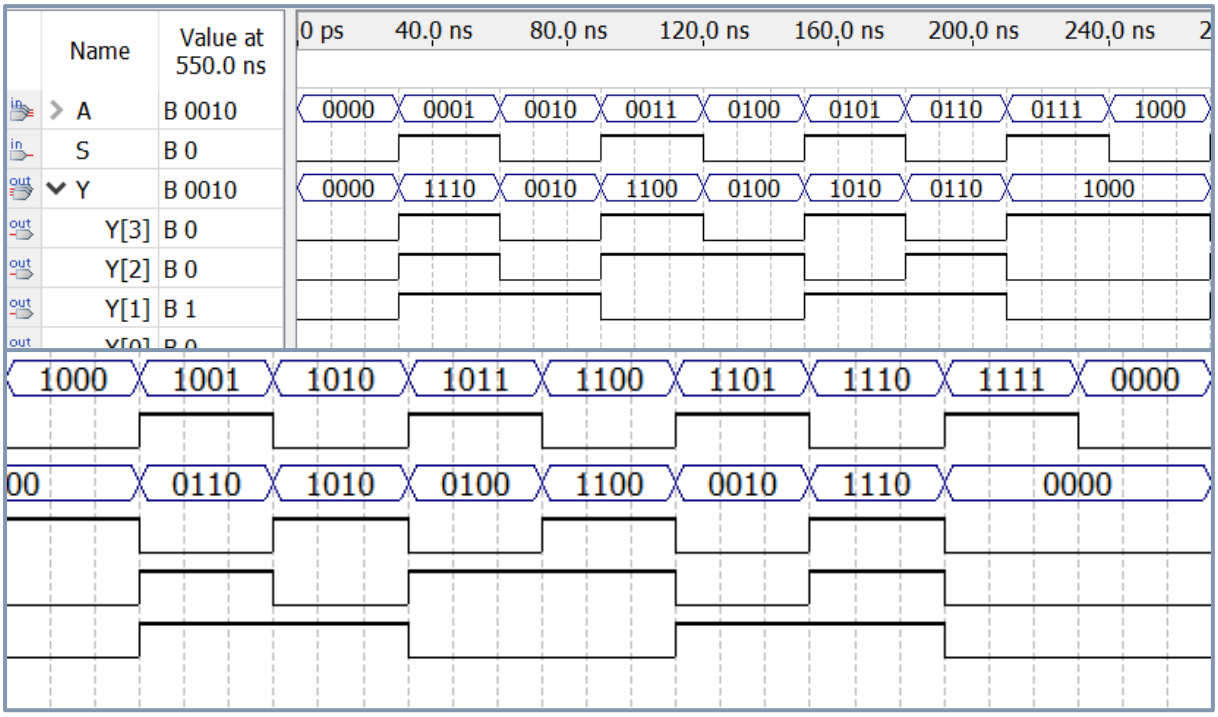


Since I1 and I0 are complements of eachother. We can choose to take the compliment of I1 and send it to the input instead.

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
2  module two_to_one_MUX(A, S, Y);
3
4  parameter size = 4;
5
6  input [size -1 :0] A;
7  input S;
8
9  output [size -1 :0] Y;
10
11 wire [size -1 :0] B;
12
13 genvar i;
14
15 generate
16 ⟶for(i = 0; i < size; i = i + 1)begin: two_to_one_MUX
17 ⟶⟶
18 ⟶⟶assign B[i] =(A[i] == 1'b0) ? 1'b1:
19 ⟶⟶⟶⟶⟶⟶⟶⟶⟶⟶⟶⟶1'b0;
20 ⟶end
21 endgenerate
22
23 assign Y = (S == 1'b0)? A:
24 ⟶⟶⟶⟶⟶⟶⟶⟶ B;
25
26
27 endmodule
```

Two make this code behavioral and also parametrized, we used the for loop and put the assign statements in there. Since looping statements only accure in procedural blocks, the assignment would be done proceduraly. With the increase of i, the index of A and B changes. In each loop the compenent of A's bit will be written to B, in the same index. After the loop, we would lastly assign an answer based on the signal value. If its 0 it should output A, if its 1 it should output B.

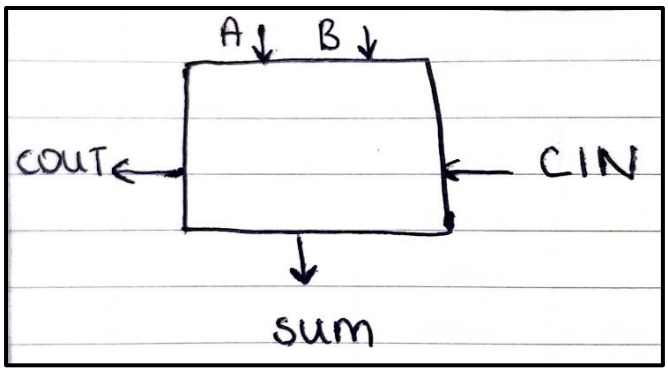


When S is 0: A is outputted.

When S is 1: A’ is outputted.

2.2.3: 4-BIT ADDER/SUBTRACTOR

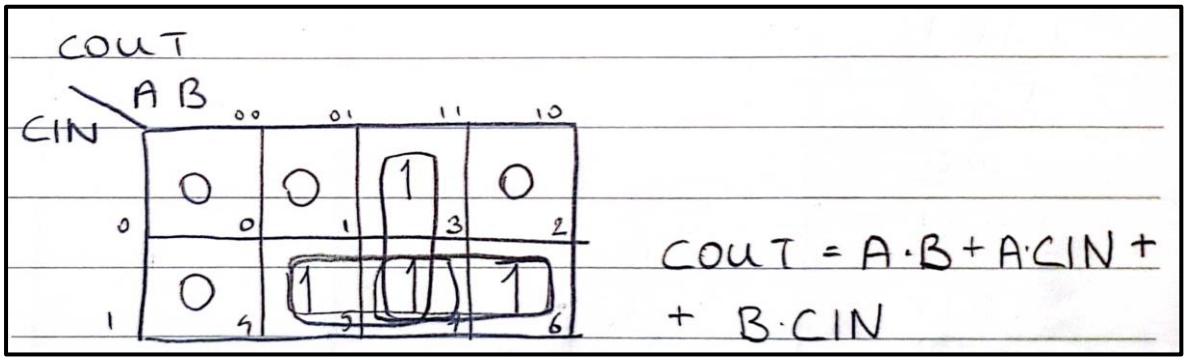
1 A full Adder has three inputs, A, B, CIN (Carry-IN), and two outputs, SUM and COUT (Carry-OUT). The output represents the result from computing binary addition of the three inputs. Derive a 1-bit Full adder truth-table and write down the logic expressions for SUM and COUT.



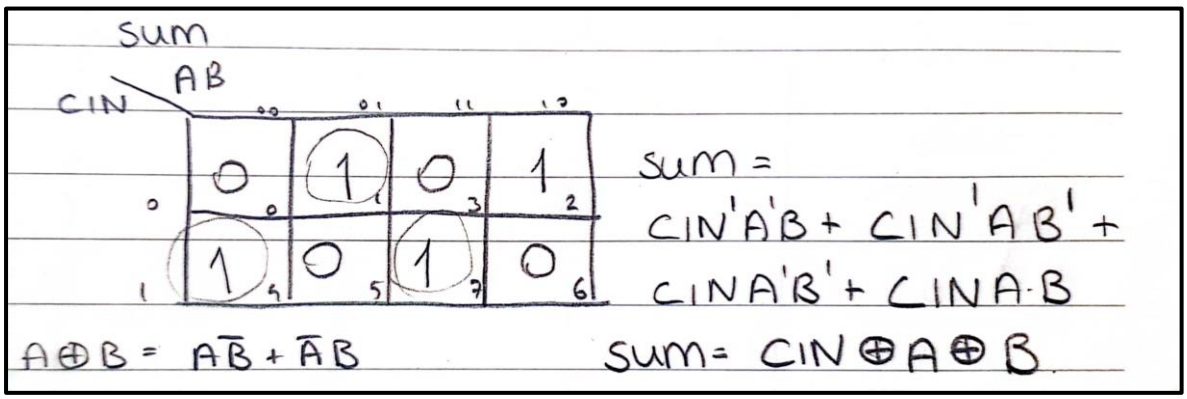
CIN	A	B	COUT	SUM
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

In its very essence Adder Subtractor adds CIN, A and B together. We write the operation result on SUM and if there is overflow, the other bit would go to COUT. We will think of CIN as a carry-in, and get that value ourselves as we keep designing the Arithmetic Unit.

COUT: A*B + A*CIN + B*CIN:



SUM: CIN ⊕ A ⊕ B:

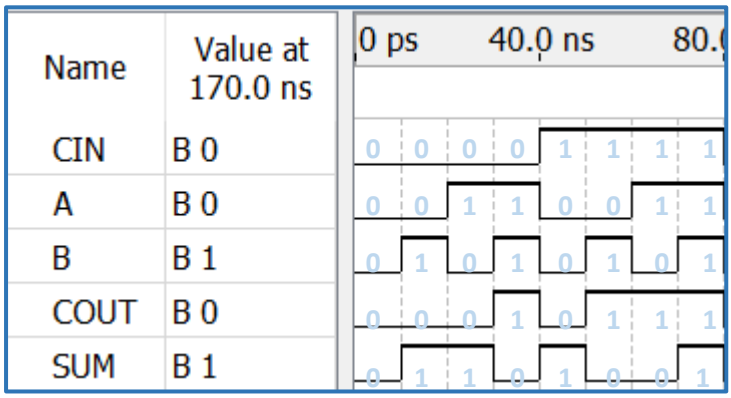


2 Write a behavioral Verilog code to implement the full adder designed in 2.2.3.1

```
module Full_Adder(CIN, A, B, COUT, SUM);

input CIN, A, B;
output COUT, SUM;

assign COUT =    ((CIN == 1'b0)&&(A == 1'b0)&&(B == 1'b0)) ? 1'b0:
                 ((CIN == 1'b0)&&(A == 1'b0)&&(B == 1'b1)) ? 1'b0:
                 ((CIN == 1'b0)&&(A == 1'b1)&&(B == 1'b0)) ? 1'b0:
                 ((CIN == 1'b0)&&(A == 1'b1)&&(B == 1'b1)) ? 1'b1:
                 ((CIN == 1'b1)&&(A == 1'b0)&&(B == 1'b0)) ? 1'b0:
                 ((CIN == 1'b1)&&(A == 1'b0)&&(B == 1'b1)) ? 1'b1:
                 ((CIN == 1'b1)&&(A == 1'b1)&&(B == 1'b0)) ? 1'b1:
                                                             1'b1;

assign SUM =     ((CIN == 1'b0)&&(A == 1'b0)&&(B == 1'b0)) ? 1'b0:
                 ((CIN == 1'b0)&&(A == 1'b0)&&(B == 1'b1)) ? 1'b1:
                 ((CIN == 1'b0)&&(A == 1'b1)&&(B == 1'b0)) ? 1'b1:
                 ((CIN == 1'b0)&&(A == 1'b1)&&(B == 1'b1)) ? 1'b0:
                 ((CIN == 1'b1)&&(A == 1'b0)&&(B == 1'b0)) ? 1'b1:
                 ((CIN == 1'b1)&&(A == 1'b0)&&(B == 1'b1)) ? 1'b0:
                 ((CIN == 1'b1)&&(A == 1'b1)&&(B == 1'b0)) ? 1'b0:
                                                             1'b1;

endmodule
```

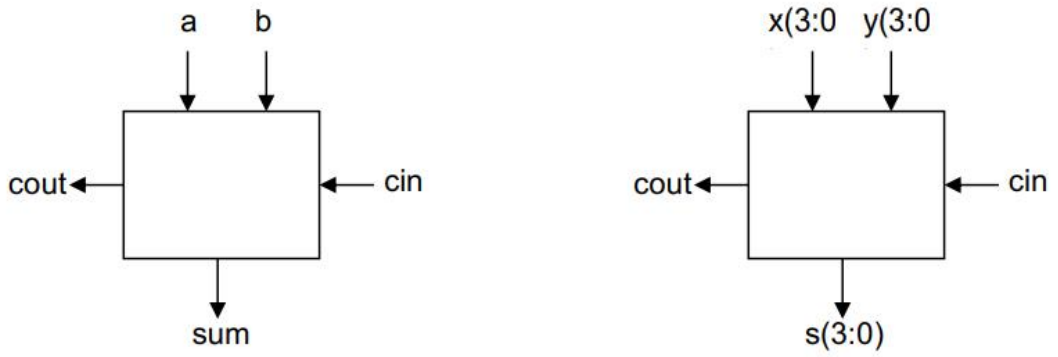
CIN	A	B	COUT	SUM
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Writing the truth tables we have created in the procedural assign statements.



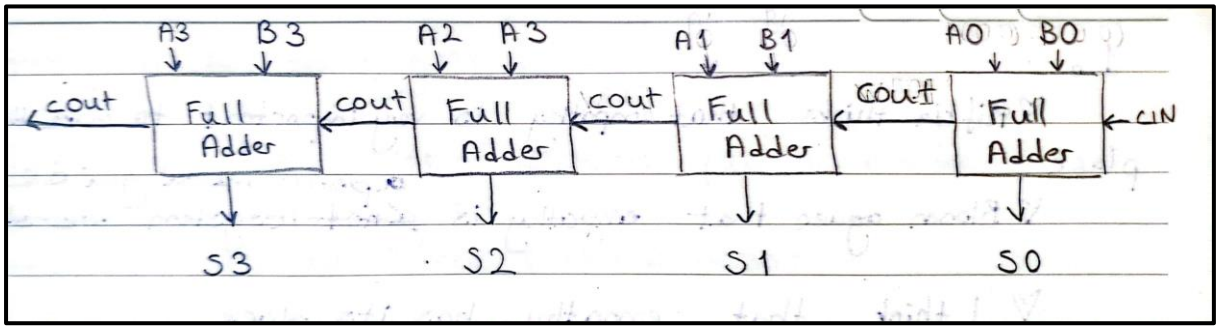
As you can see on the annotated quartus waveform, the truth table values match.

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 Figure 3(b).



(a) (b)

Figure 3: (a) 1-bit Full Adder Symbol, (b) 4-bit Ripple Carry Adder Symbol



The full adders would need to be connected in fours to build a four bit full adder. The carry-out would become the carry-in for the middle and last full adder. Just like the binary addition we do on paper, we would need to add the carry-outs. Each bit of A and B would enter each full adder and do the calculation with the carry-outs. Each full adder would output the each bit of the summation.

The logic would be similar to saying:

	1	1	1	1	1	1		← Carry outs
		1	1	1	1	0	1	= 61 ← A
+			1	0	1	1	1	= 23 ← B
1	0	1	0	1	0	0		= 84 ← carry-out + sum

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

```
1
2  module Four_Bit_Ripple_Carry_Adder(CIN, A, B, COUT, SUM);
3
4  parameter size = 4;
5
6  input[size -1: 0] A, B;
7  input CIN;
8
9  output [size -1: 0] SUM;
10 output COUT;
11
12 wire [size -2: 0] W;
13
14 genvar i;
15
16 generate
17 ⟶for(i = 0; i < size; i = i + 1) begin: Four_Bit_Ripple_Carry_Adder
18 ⟶
19 ⟶⟶if(i == 0)
20 ⟶⟶⟶Full_Adder U1(CIN, A[i], B[i], W[i], SUM[i]);
21 ⟶
22 ⟶⟶else if(i == size -1)
23 ⟶⟶⟶Full_Adder U1(W[i - 1], A[i], B[i], COUT, SUM[i]);
24 ⟶
25 ⟶⟶else
26 ⟶⟶⟶Full_Adder U1(W[i - 1], A[i], B[i], W[i], SUM[i]);
27 ⟶end
28 endgenerate
29
30 endmodule
```

With the generate block we can make a for loop and count an i value. The i value would be the index of the Arrays A and B.

if i == 0:

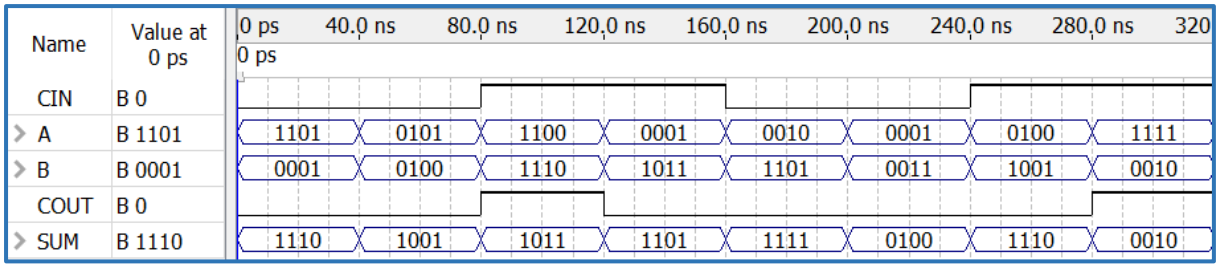
The first full adder is initiated, this needs to be done seperately because the Full_Adder(0) has the input CIN instead of COUT.

else:

The middle full adders are initiated. The input for them needs to be COUT of the previous Full_Adder. The output is written in a wire.

else if:

The last full adder is initiated. It outputs the actual COUT.



As it can be seen from the waveform the addition is done correctly. The COUT is 1 when there is an overflown bit at the end of addition.

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. See table

6 Table 2. This will make it easier to code your unit since you are just using 1-bit adder unit to design adder/subtractor.

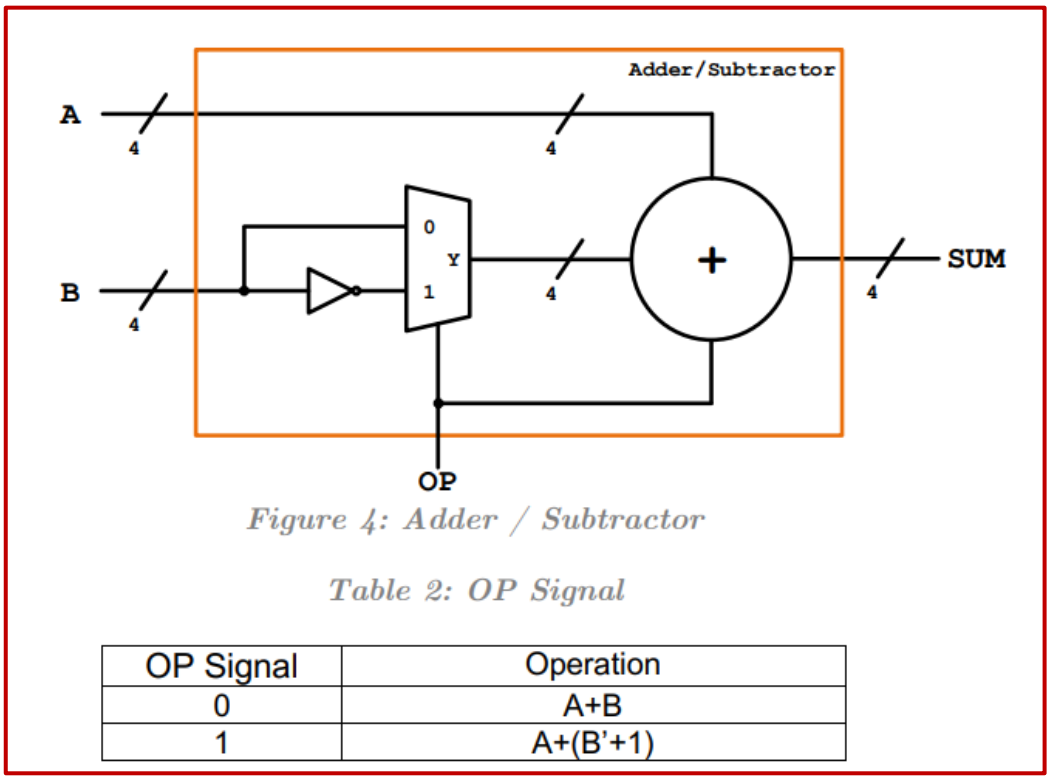


Figure 4: Adder / Subtractor

Table 2: OP Signal

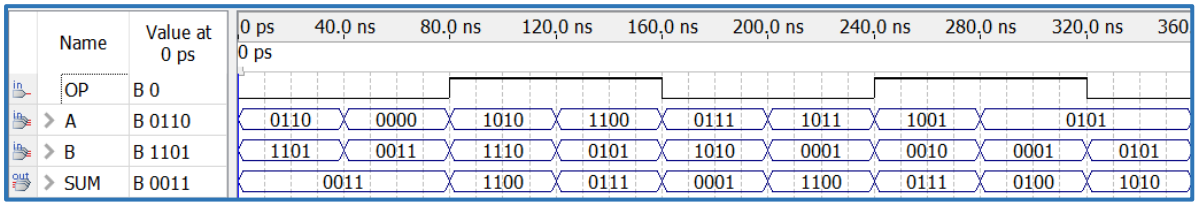
OP Signal	Operation
0	A+B
1	A+(B'+1)

When we add the two-to-one mux that we have created before something interesting occurs. When the signal OP is 1 the mux would output the complement of B. However the OP is also used as an input for the CIN. This means that one is added to B's complement. This would make a 2s complement. So the addition would become A plus B's 2s complement. Also known and **binary subtraction**. When OP is 0 the addition would occur as usual.

Also when we create the Adder/Subtractor we don't take the previous COUT to account anymore.

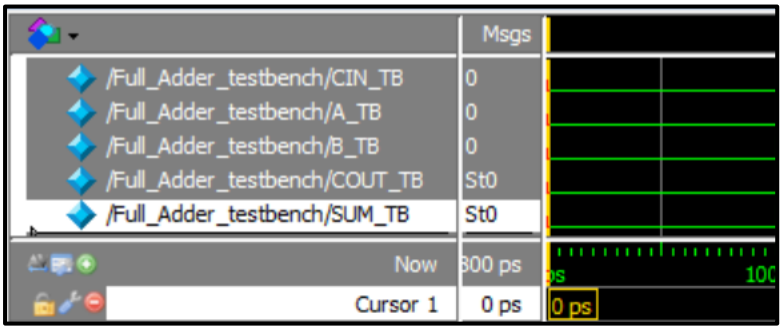
The design is very similar to four_bit_ripple_carry_adder. The inputs for some have changed.

```
1
2   module Adder_Subtractor(A, B, OP, SUM);
3
4   parameter size = 4;
5
6   input [size -1:0] A, B;
7   input OP;
8
9   output [size -1:0] SUM;
10
11  wire [size -1:0] W_B;
12  wire [size -2:0] W;
13
14  two_to_one_MUX U1(B, OP, W_B);
15
16  genvar i;
17
18 generate
19 ⟶for(i = 0; i < size; i = i + 1) begin: Adder_Subtractor
20  ⟶
21  ⟶⟶if(i == 0)
22  ⟶⟶⟶Full_Adder U2(OP, A[i], W_B[i], W[i], SUM[i]);
23  ⟶
24  ⟶⟶else if(i == size -1)
25  ⟶⟶⟶Full_Adder U2(W[i - 1], A[i], W_B[i], COUT, SUM[i]);
26  ⟶
27  ⟶⟶else
28  ⟶⟶⟶Full_Adder U2(W[i - 1], A[i], W_B[i], W[i], SUM[i]);
29  ⟶end
30  endgenerate
31
32  endmodule
33
```

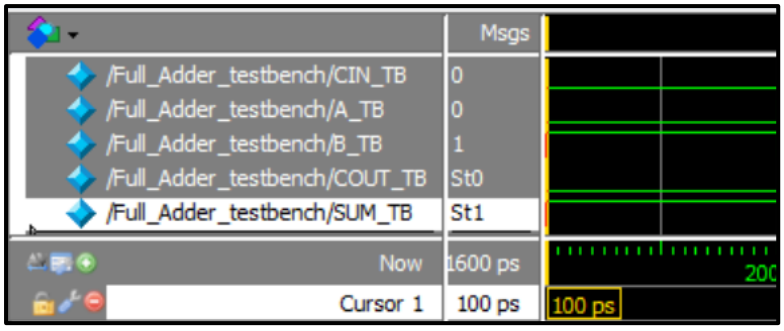


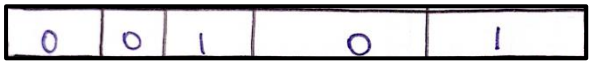
7 Refer to Modelsim® tutorial and write a testbench for your designs using Modelsim® and simulate your Verilog codes in 2.2.3.2 - 2.2.3.5. By showing all inputs and outputs on waveform.

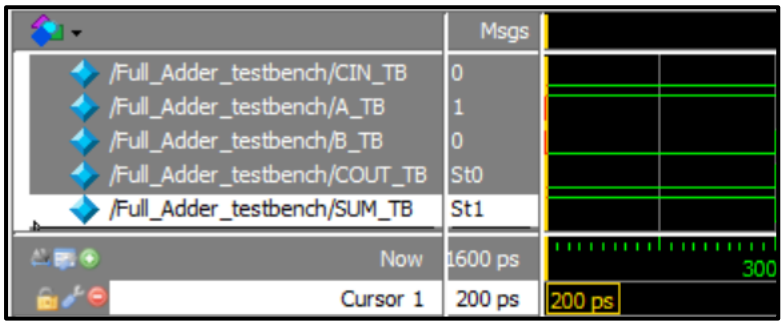
```
1
2  module Full_Adder_testbench();
3
4  reg CIN_TB, A_TB, B_TB;
5  wire COUT_TB, SUM_TB;
6
7  Full_Adder DUT(CIN_TB, A_TB, B_TB, COUT_TB, SUM_TB);
8
9  initial
0
1  begin
2
3      CIN_TB = 1'b0; A_TB = 1'b0; B_TB = 1'b0; #100;
4      CIN_TB = 1'b0; A_TB = 1'b0; B_TB = 1'b1; #100;
5      CIN_TB = 1'b0; A_TB = 1'b1; B_TB = 1'b0; #100;
6      CIN_TB = 1'b0; A_TB = 1'b1; B_TB = 1'b1; #100;
7      CIN_TB = 1'b1; A_TB = 1'b0; B_TB = 1'b0; #100;
8      CIN_TB = 1'b1; A_TB = 1'b0; B_TB = 1'b1; #100;
9      CIN_TB = 1'b1; A_TB = 1'b1; B_TB = 1'b0; #100;
0      CIN_TB = 1'b1; A_TB = 1'b1; B_TB = 1'b1; #100;
1
2  end
3
4  endmodule
```



CIN	A	B	COUT	SUM
0	0	0	0	0

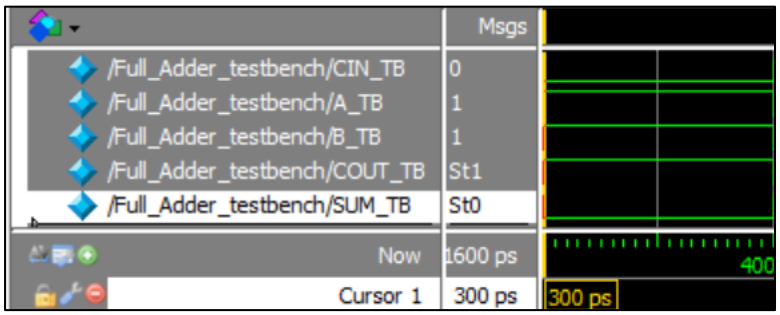

Msgs
/Full_Adder_testbench/CIN_TB
/Full_Adder_testbench/A_TB
/Full_Adder_testbench/B_TB
/Full_Adder_testbench/COUT_TB
/Full_Adder_testbench/SUM_TB
0
0
1
St0
St1
Now
1600 ps
Cursor 1
100 ps
100 ps

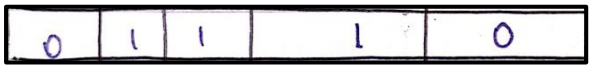

0 0 1 0 1

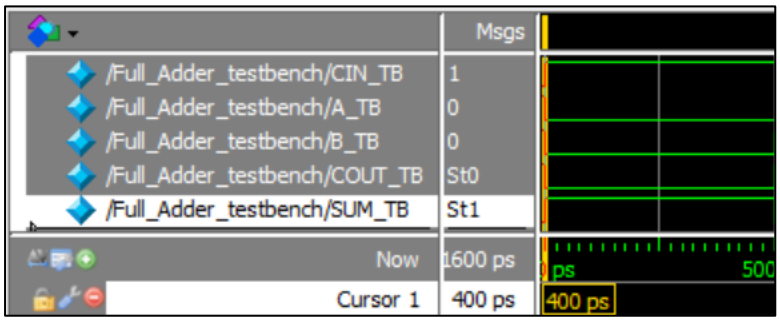

Msgs
/Full_Adder_testbench/CIN_TB
/Full_Adder_testbench/A_TB
/Full_Adder_testbench/B_TB
/Full_Adder_testbench/COUT_TB
/Full_Adder_testbench/SUM_TB
0
1
0
St0
St1
Now
1600 ps
Cursor 1
200 ps
200 ps

,

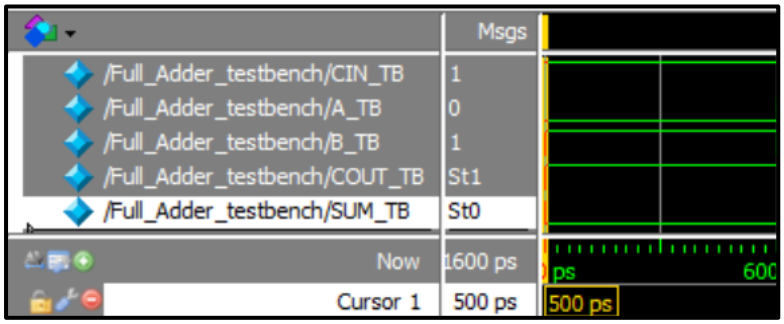

0 1 0 0 1


Msgs
/Full_Adder_testbench/CIN_TB
/Full_Adder_testbench/A_TB
/Full_Adder_testbench/B_TB
/Full_Adder_testbench/COUT_TB
/Full_Adder_testbench/SUM_TB
0
1
1
St1
St0
Now
1600 ps
Cursor 1
300 ps
300 ps

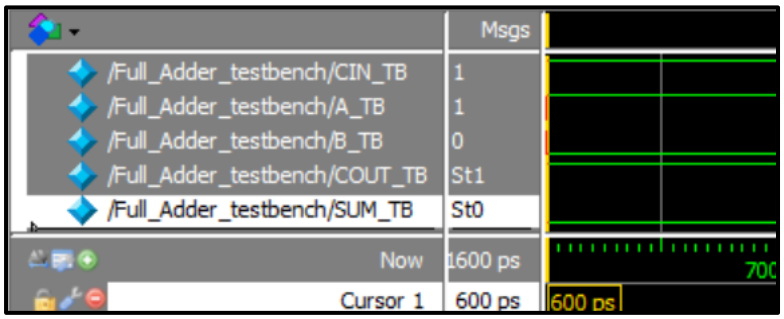

0 1 1 1 0

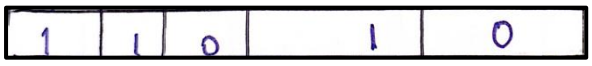

Msgs
/Full_Adder_testbench/CIN_TB
/Full_Adder_testbench/A_TB
/Full_Adder_testbench/B_TB
/Full_Adder_testbench/COUT_TB
/Full_Adder_testbench/SUM_TB
1
0
0
St0
St1
Now
1600 ps
Cursor 1
400 ps
400 ps

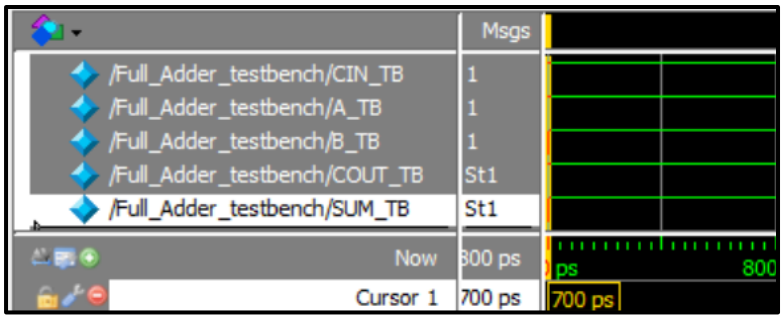

1 0 0 0 1

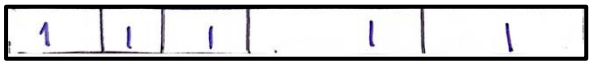

Msgs
/Full_Adder_testbench/CIN_TB
/Full_Adder_testbench/A_TB
/Full_Adder_testbench/B_TB
/Full_Adder_testbench/COUT_TB
/Full_Adder_testbench/SUM_TB
1
0
1
St1
St0
Now
1600 ps
Cursor 1
500 ps
500 ps


1
0
1
1
0


Msgs
/Full_Adder_testbench/CIN_TB
/Full_Adder_testbench/A_TB
/Full_Adder_testbench/B_TB
/Full_Adder_testbench/COUT_TB
/Full_Adder_testbench/SUM_TB
1
1
0
St1
St0
Now
1600 ps
Cursor 1
600 ps
600 ps


1
1
0
1
0

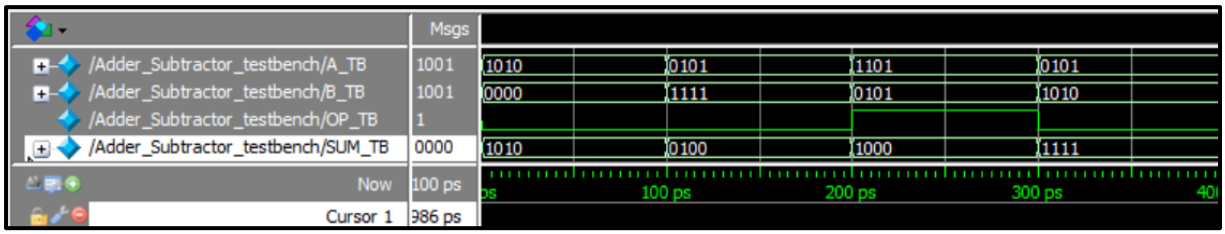

Msgs
/Full_Adder_testbench/CIN_TB
/Full_Adder_testbench/A_TB
/Full_Adder_testbench/B_TB
/Full_Adder_testbench/COUT_TB
/Full_Adder_testbench/SUM_TB
1
1
1
St1
St1
Now
800 ps
Cursor 1
700 ps
700 ps

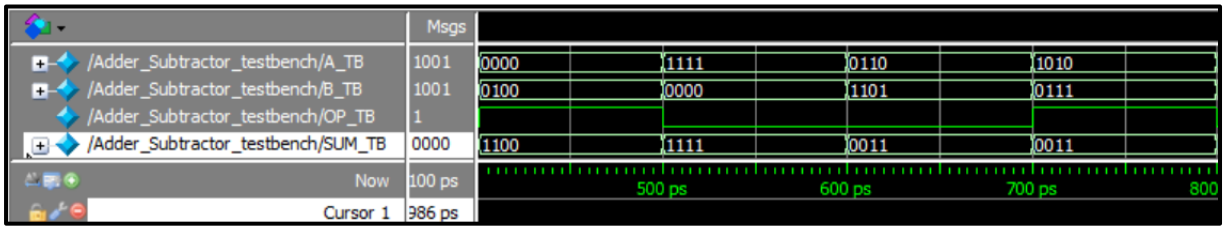

1
1
1
1
1

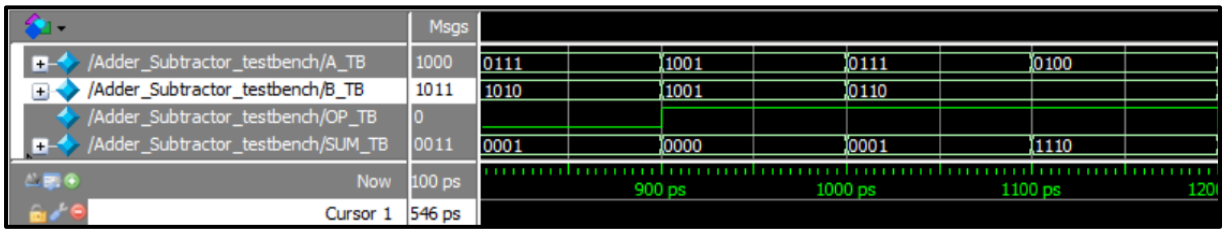
```
1
2   module Adder_Subtractor_testbench();
3
4   parameter size = 4;
5
6   reg [size -1:0] A_TB, B_TB;
7   reg OP_TB;
8
9   wire [size -1:0] SUM_TB;
10
11  Adder_Subtractor DUT(A_TB, B_TB, OP_TB, SUM_TB);
12
13  initial
14
15  ⟶begin
16  ⟶
17  ⟶⟶OP_TB = 1'b0; A_TB = 4'b1010; B_TB = 4'b0000; #100;
18  ⟶⟶OP_TB = 1'b0; A_TB = 4'b0101; B_TB = 4'b1111; #100;
19  ⟶⟶OP_TB = 1'b1; A_TB = 4'b1101; B_TB = 4'b0101; #100;
20  ⟶⟶OP_TB = 1'b0; A_TB = 4'b0101; B_TB = 4'b1010; #100;
21  ⟶⟶OP_TB = 1'b1; A_TB = 4'b0000; B_TB = 4'b0100; #100;
22  ⟶⟶OP_TB = 1'b0; A_TB = 4'b1111; B_TB = 4'b0000; #100;
23  ⟶⟶OP_TB = 1'b0; A_TB = 4'b0110; B_TB = 4'b1101; #100;
24  ⟶⟶OP_TB = 1'b1; A_TB = 4'b1010; B_TB = 4'b0111; #100;
25  ⟶⟶OP_TB = 1'b0; A_TB = 4'b0111; B_TB = 4'b1010; #100;
26  ⟶⟶OP_TB = 1'b1; A_TB = 4'b1001; B_TB = 4'b1001; #100;
27  ⟶⟶OP_TB = 1'b1; A_TB = 4'b0111; B_TB = 4'b0110; #100;
28  ⟶⟶OP_TB = 1'b1; A_TB = 4'b0100; B_TB = 4'b0110; #100;
29  ⟶⟶OP_TB = 1'b0; A_TB = 4'b0110; B_TB = 4'b1110; #100;
30  ⟶⟶OP_TB = 1'b1; A_TB = 4'b0000; B_TB = 4'b0101; #100;
31  ⟶⟶OP_TB = 1'b0; A_TB = 4'b0011; B_TB = 4'b1010; #100;
32  ⟶⟶OP_TB = 1'b0; A_TB = 4'b1000; B_TB = 4'b1011; #100;
33  ⟶end
34
35  endmodule
```

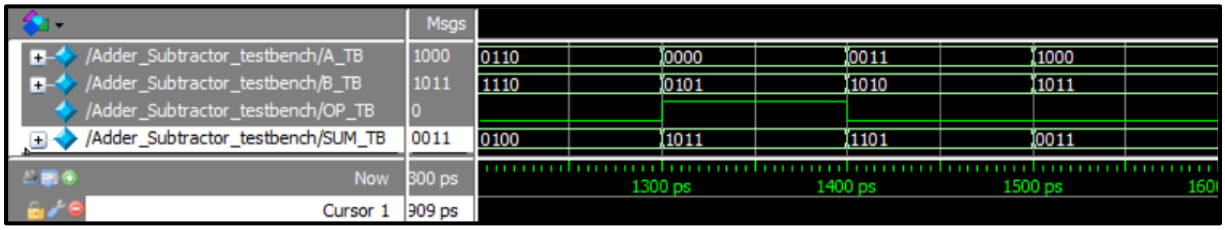
Random values were written for the testbench.

All outputs in waveform:

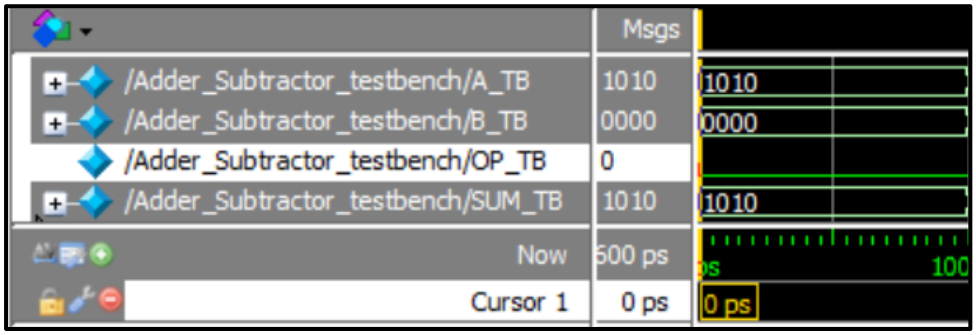




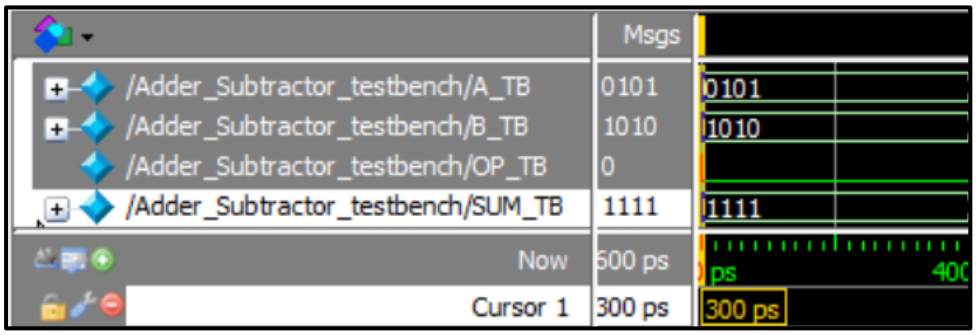




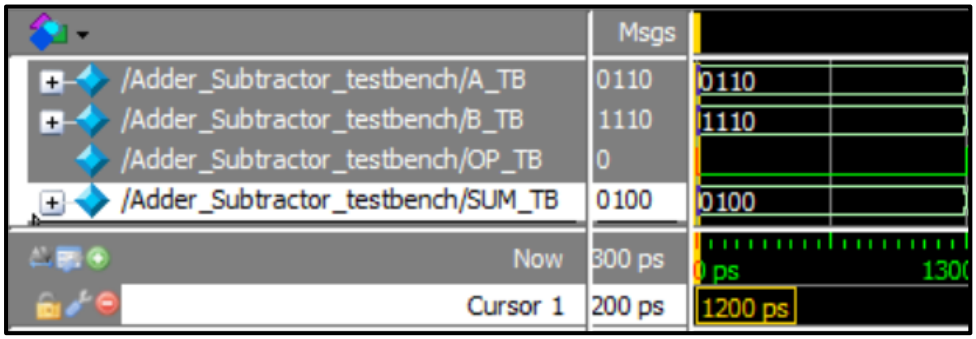
Let's check some of them:



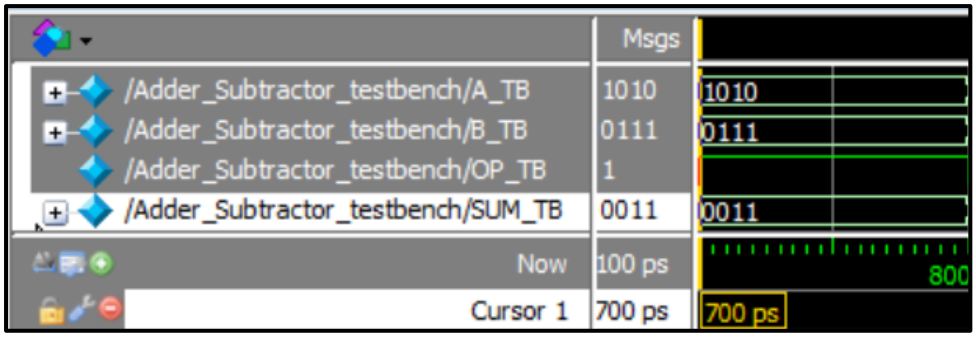
Op is 0 so we only do addition. 1010 + 0000 is 1010.



Op is 0 so we only do addition. 0101 + 1010 is 1111.



Op is 0 so we only do addition. 0110 + 1110 is 10100 but we ignore the overflow so it becomes 0100.



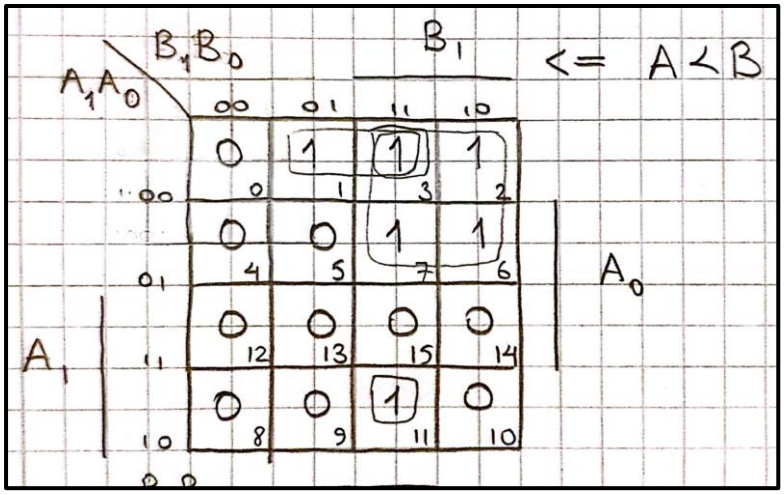
Op is 1 so we do subtraction. 2s complement of B is 1001. 1010 + 1001 is 10011 but we ignore the overflow so it becomes 0100.

2.2.4: 4-BIT COMPARATOR

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.

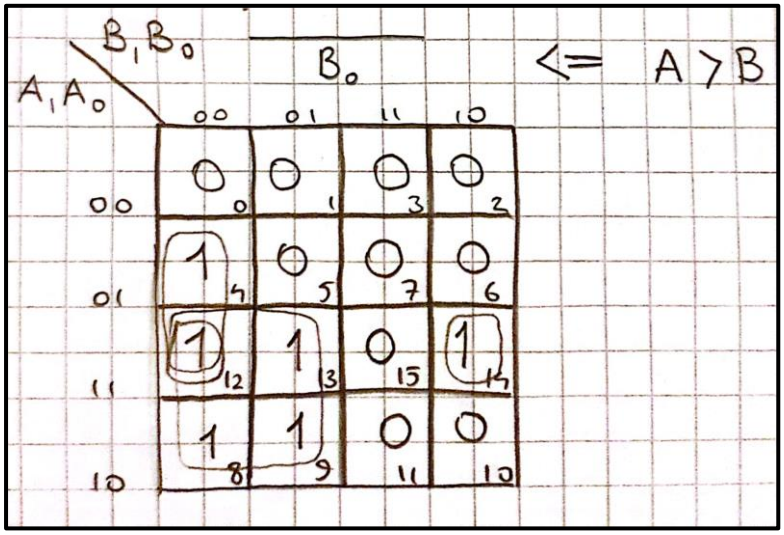
A	B	A<B	A>B	A=B
0 0	0 0	0	0	1
0 0	0 1	1	0	0
0 0	1 0	1	0	0
0 0	1 1	1	0	0
0 1	0 0	0	1	0
0 1	0 1	0	0	1
0 1	1 0	1	0	0
0 1	1 1	1	0	0
1 0	0 0	0	1	0
1 0	0 1	0	1	0
1 0	1 0	0	0	1
1 0	1 1	1	0	0
1 1	0 0	0	1	0
1 1	0 1	0	1	0
1 1	1 0	0	1	0
1 1	1 1	0	0	1

Very simple, when B is bigger than A, A < B is 1. When A is bigger than B, A > B is 1. If A equals to B, A = B is 1. Only one of them could be 1.



$$= B_1 \cdot A_1 + \overline{A_1} \cdot \overline{A_0} \cdot B_0 + B_0 B_1 \cdot \overline{A_0}$$

$$\overline{A_1} \cdot B_1 + \overline{A_1} \cdot \overline{A_0} \cdot B_0 + \overline{A_0} \cdot B_0 B_1$$



$$= A_1 \cdot \overline{B_1} + A_1 \cdot A_0 \overline{B_0} + A_0 \overline{B_0}\ \overline{B_1}$$

$$= A_1 \cdot \overline{B_1} + A_1 \cdot A_0 \overline{B_0} + A_0 \overline{B_1}\ \overline{B_0}$$

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

For a 4-bit comparator parametrized verilog code, we first need to implement the 2-bit comparator. We used behavioral modeling for the 2-bit comparator.

```

module two_bit_comparator(A1,A0,B1,B0, B_big_A, A_big_B, A_eq_B);

input A1,A0,B1,B0;
output B_big_A, A_big_B, A_eq_B;

assign B_big_A =  ((A1 == 1'b0)&&(A0 == 1'b0)&&(B1 == 1'b0)&&(B0 == 1'b0))? 1'b0:
						((A1 == 1'b0)&&(A0 == 1'b0)&&(B1 == 1'b0)&&(B0 == 1'b1))? 1'b1:
						((A1 == 1'b0)&&(A0 == 1'b0)&&(B1 == 1'b1)&&(B0 == 1'b0))? 1'b1:
						((A1 == 1'b0)&&(A0 == 1'b0)&&(B1 == 1'b1)&&(B0 == 1'b1))? 1'b1:
						((A1 == 1'b0)&&(A0 == 1'b1)&&(B1 == 1'b0)&&(B0 == 1'b0))? 1'b0:
						((A1 == 1'b0)&&(A0 == 1'b1)&&(B1 == 1'b0)&&(B0 == 1'b1))? 1'b0:
						((A1 == 1'b0)&&(A0 == 1'b1)&&(B1 == 1'b1)&&(B0 == 1'b0))? 1'b1:
						((A1 == 1'b0)&&(A0 == 1'b1)&&(B1 == 1'b1)&&(B0 == 1'b1))? 1'b1:
						((A1 == 1'b1)&&(A0 == 1'b0)&&(B1 == 1'b0)&&(B0 == 1'b0))? 1'b0:
						((A1 == 1'b1)&&(A0 == 1'b0)&&(B1 == 1'b0)&&(B0 == 1'b1))? 1'b0:
						((A1 == 1'b1)&&(A0 == 1'b0)&&(B1 == 1'b1)&&(B0 == 1'b0))? 1'b0:
						((A1 == 1'b1)&&(A0 == 1'b0)&&(B1 == 1'b1)&&(B0 == 1'b1))? 1'b1:
						((A1 == 1'b1)&&(A0 == 1'b1)&&(B1 == 1'b0)&&(B0 == 1'b0))? 1'b0:
						((A1 == 1'b1)&&(A0 == 1'b1)&&(B1 == 1'b0)&&(B0 == 1'b1))? 1'b0:
						((A1 == 1'b1)&&(A0 == 1'b1)&&(B1 == 1'b1)&&(B0 == 1'b0))? 1'b0:
																								1'b0;
```

```

assign A_big_B =  ((A1 == 1'b0)&&(A0 == 1'b0)&&(B1 == 1'b0)&&(B0 == 1'b0))? 1'b0:
						((A1 == 1'b0)&&(A0 == 1'b0)&&(B1 == 1'b0)&&(B0 == 1'b1))? 1'b0:
						((A1 == 1'b0)&&(A0 == 1'b0)&&(B1 == 1'b1)&&(B0 == 1'b0))? 1'b0:
						((A1 == 1'b0)&&(A0 == 1'b0)&&(B1 == 1'b1)&&(B0 == 1'b1))? 1'b0:
						((A1 == 1'b0)&&(A0 == 1'b1)&&(B1 == 1'b0)&&(B0 == 1'b0))? 1'b1:
						((A1 == 1'b0)&&(A0 == 1'b1)&&(B1 == 1'b0)&&(B0 == 1'b1))? 1'b0:
						((A1 == 1'b0)&&(A0 == 1'b1)&&(B1 == 1'b1)&&(B0 == 1'b0))? 1'b0:
						((A1 == 1'b0)&&(A0 == 1'b1)&&(B1 == 1'b1)&&(B0 == 1'b1))? 1'b0:
						((A1 == 1'b1)&&(A0 == 1'b0)&&(B1 == 1'b0)&&(B0 == 1'b0))? 1'b1:
						((A1 == 1'b1)&&(A0 == 1'b0)&&(B1 == 1'b0)&&(B0 == 1'b1))? 1'b1:
						((A1 == 1'b1)&&(A0 == 1'b0)&&(B1 == 1'b1)&&(B0 == 1'b0))? 1'b0:
						((A1 == 1'b1)&&(A0 == 1'b0)&&(B1 == 1'b1)&&(B0 == 1'b1))? 1'b0:
						((A1 == 1'b1)&&(A0 == 1'b1)&&(B1 == 1'b0)&&(B0 == 1'b0))? 1'b1:
						((A1 == 1'b1)&&(A0 == 1'b1)&&(B1 == 1'b0)&&(B0 == 1'b1))? 1'b1:
						((A1 == 1'b1)&&(A0 == 1'b1)&&(B1 == 1'b1)&&(B0 == 1'b0))? 1'b1:
																								1'b0;
```

```
40
41  assign A_eq_B =   ((A1 == 1'b0)&&(A0 == 1'b0)&&(B1 == 1'b0)&&(B0 == 1'b0))?1'b1:
42  ->->->->->->((A1 == 1'b0)&&(A0 == 1'b0)&&(B1 == 1'b0)&&(B0 == 1'b1))?1'b0:
43  ->->->->->->((A1 == 1'b0)&&(A0 == 1'b0)&&(B1 == 1'b1)&&(B0 == 1'b0))?1'b0:
44  ->->->->->->((A1 == 1'b0)&&(A0 == 1'b0)&&(B1 == 1'b1)&&(B0 == 1'b1))?1'b0:
45  ->->->->->->((A1 == 1'b0)&&(A0 == 1'b1)&&(B1 == 1'b0)&&(B0 == 1'b0))?1'b0:
46  ->->->->->->((A1 == 1'b0)&&(A0 == 1'b1)&&(B1 == 1'b0)&&(B0 == 1'b1))?1'b1:
47  ->->->->->->((A1 == 1'b0)&&(A0 == 1'b1)&&(B1 == 1'b1)&&(B0 == 1'b0))?1'b0:
48  ->->->->->->((A1 == 1'b0)&&(A0 == 1'b1)&&(B1 == 1'b1)&&(B0 == 1'b1))?1'b0:
49  ->->->->->->((A1 == 1'b1)&&(A0 == 1'b0)&&(B1 == 1'b0)&&(B0 == 1'b0))?1'b0:
50  ->->->->->->((A1 == 1'b1)&&(A0 == 1'b0)&&(B1 == 1'b0)&&(B0 == 1'b1))?1'b0:
51  ->->->->->->((A1 == 1'b1)&&(A0 == 1'b0)&&(B1 == 1'b1)&&(B0 == 1'b0))?1'b1:
52  ->->->->->->((A1 == 1'b1)&&(A0 == 1'b0)&&(B1 == 1'b1)&&(B0 == 1'b1))?1'b0:
53  ->->->->->->((A1 == 1'b1)&&(A0 == 1'b1)&&(B1 == 1'b0)&&(B0 == 1'b0))?1'b0:
54  ->->->->->->((A1 == 1'b1)&&(A0 == 1'b1)&&(B1 == 1'b0)&&(B0 == 1'b1))?1'b0:
55  ->->->->->->((A1 == 1'b1)&&(A0 == 1'b1)&&(B1 == 1'b1)&&(B0 == 1'b0))?1'b0:
56  ->->->->->->->->->->->->->->->->->->->->->->->->->->->->->->->->->->->->->->->1'b1;
57
58  endmodule
```

```
1
2   module four_bit_comparator(A,B, B_big_A, A_big_B, A_eq_B);
3
4   parameter size = 4;
5   parameter biggest_index = 3;
6
7   input [size -1: 0]A, B;
8   output B_big_A, A_big_B, A_eq_B;
9
10  wire [size -2:0]w1,w2,w3;
11
12  genvar i;
13
14 generate
15 ->for(i = 0; i < biggest_index; i = i + 1) begin: four_bit_comparator
16  ->
17  ->->if(i == 0)
18  ->->->two_bit_comparator U1(A[i + 1], A[i], B[i + 1], B[i], w1[i], w2[i], w3[i]);
19  ->->
20  ->->else if(i == biggest_index -1)
21  ->->->two_bit_comparator U1(w2[1], w2[0], w1[1], w1[0], B_big_A, A_big_B, A_eq_B);
22  ->->
23  ->->else
24  ->->->two_bit_comparator U1(A[i + 2], A[i + 1], B[i + 2], B[i + 1], w1[i], w2[i], w3[i]);
25  ->->
26  ->->
27  ->->
28  ->end
29  endgenerate
30
31
32  endmodule
```

The four_bit_comparator needs to be parametrized. So we used the generate block again.

if == 0: (start)

A1, A0 and B1, B0 would go to the two_bit_comparator function. The outputs would be written to w1(B > A), w2(A > B), w3(A = B).

else: (middle)

A3, A2 and B3, B2 would go to the two_bit_comparator function. The outputs would be written to w1(B > A), w2(A > B), w3(A = B).

Else if: (end)

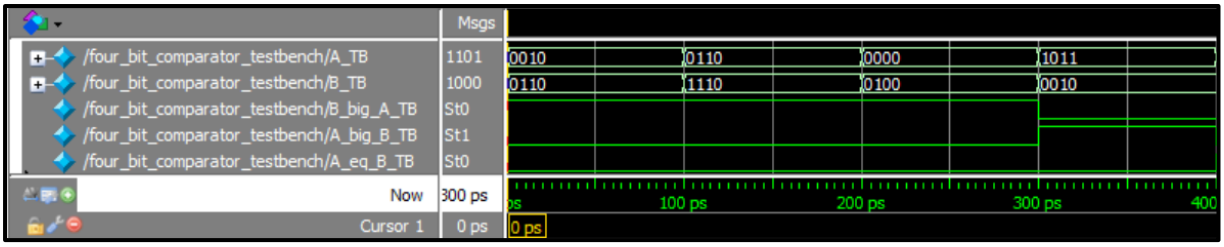
w2[1], w2[0] and w1[1], w1[0] would go to the two_bit_comparator function. So (A > B) and (B > A) would be compared. If they're equal, A = B would be 1. If w1 is bigger B > A becomes 1. If w2 is bigger A > B becomes 1.

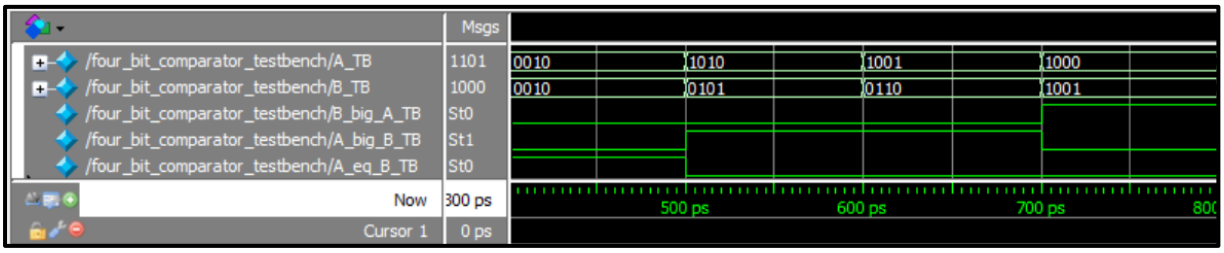
3 Write a testbench for your design using Modelsim® and simulate your comparator Verilog code.

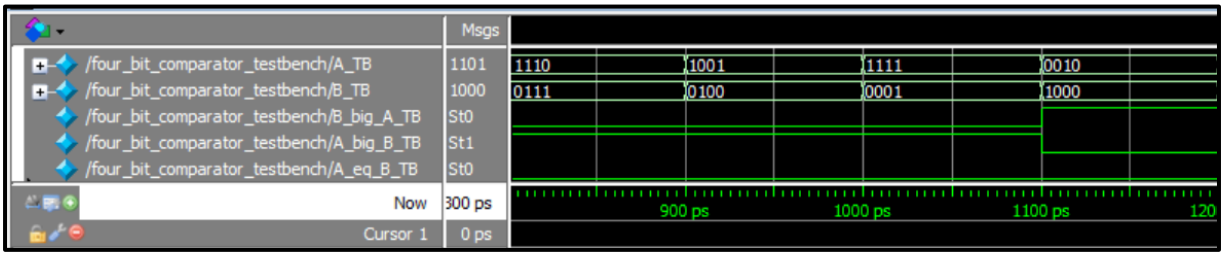
```
1
2   module four_bit_comparator_testbench();
3
4   parameter size = 4;
5
6   reg [size-1:0] A_TB, B_TB;
7   wire B_big_A_TB, A_big_B_TB, A_eq_B_TB;
8
9   four_bit_comparator DUT(A_TB, B_TB, B_big_A_TB, A_big_B_TB, A_eq_B_TB);
10
11  initial //random values
12
13  begin
14  
15      A_TB = 4'b0010; B_TB = 4'b0110; #100;
16      A_TB = 4'b0110; B_TB = 4'b1110; #100;
17      A_TB = 4'b0000; B_TB = 4'b0100; #100;
18      A_TB = 4'b1011; B_TB = 4'b0010; #100;
19      A_TB = 4'b0010; B_TB = 4'b0010; #100;
20      A_TB = 4'b1010; B_TB = 4'b0101; #100;
21      A_TB = 4'b1001; B_TB = 4'b0110; #100;
22      A_TB = 4'b1000; B_TB = 4'b1001; #100;
23      A_TB = 4'b1110; B_TB = 4'b0111; #100;
24      A_TB = 4'b1001; B_TB = 4'b0100; #100;
25      A_TB = 4'b1111; B_TB = 4'b0001; #100;
26      A_TB = 4'b0010; B_TB = 4'b1000; #100;
27      A_TB = 4'b0001; B_TB = 4'b0101; #100;
28      A_TB = 4'b0111; B_TB = 4'b0111; #100;
29      A_TB = 4'b1111; B_TB = 4'b0110; #100;
30      A_TB = 4'b1101; B_TB = 4'b1000; #100;
31  end
32
33  endmodule
```

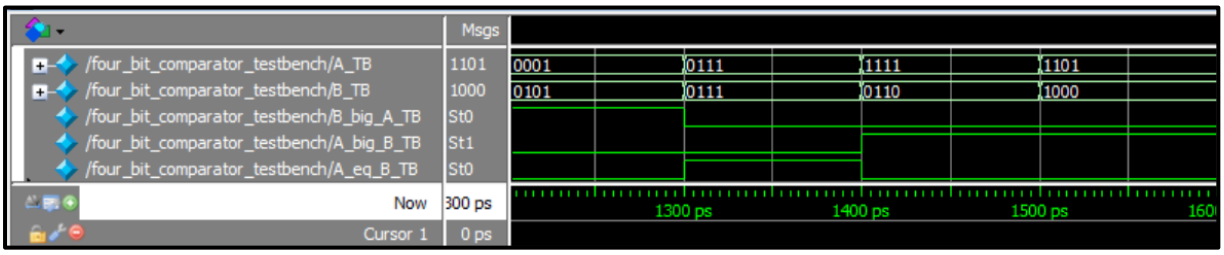
Random values are written for the testbench.

All outputs in waveform:

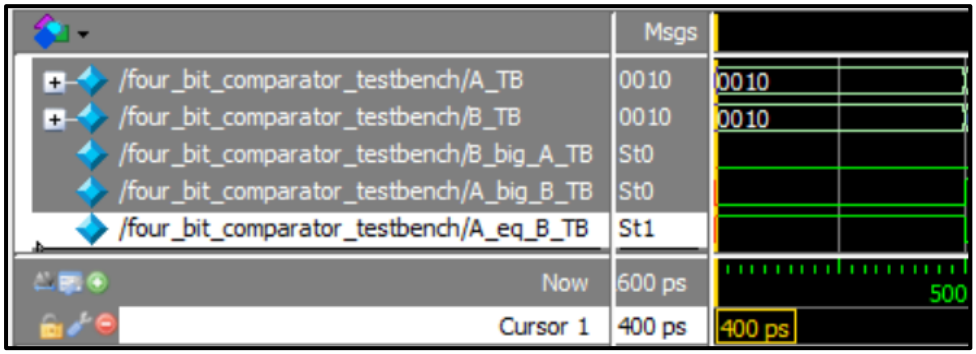




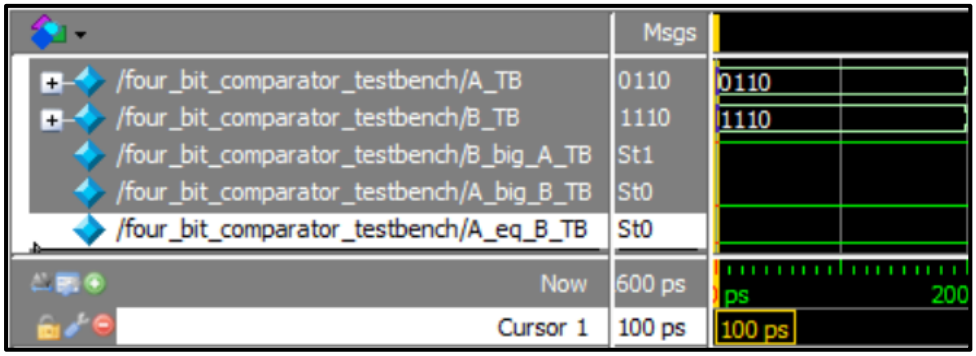




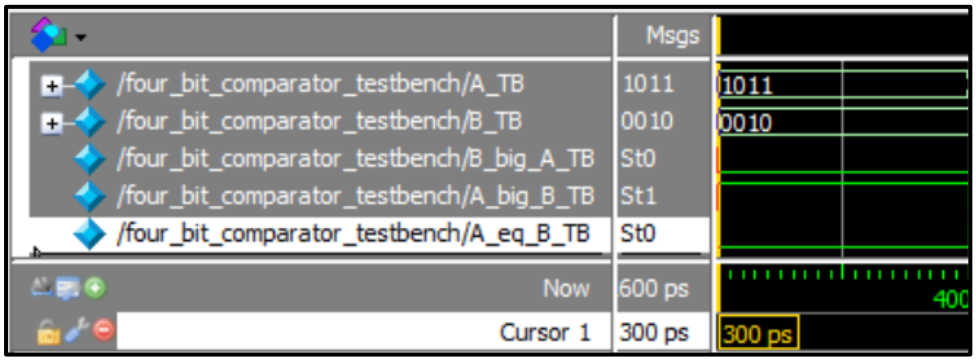
Let’s check some of them:



0010 and 0010 are equal so A = B is 1, rest are 0.



1110 is bigger than 0110 so B > A is 1, rest are 0.



1011 is bigger than 0010 so A > B is 1, rest are 0.

2.2.5: ARITHMETIC UNIT

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.

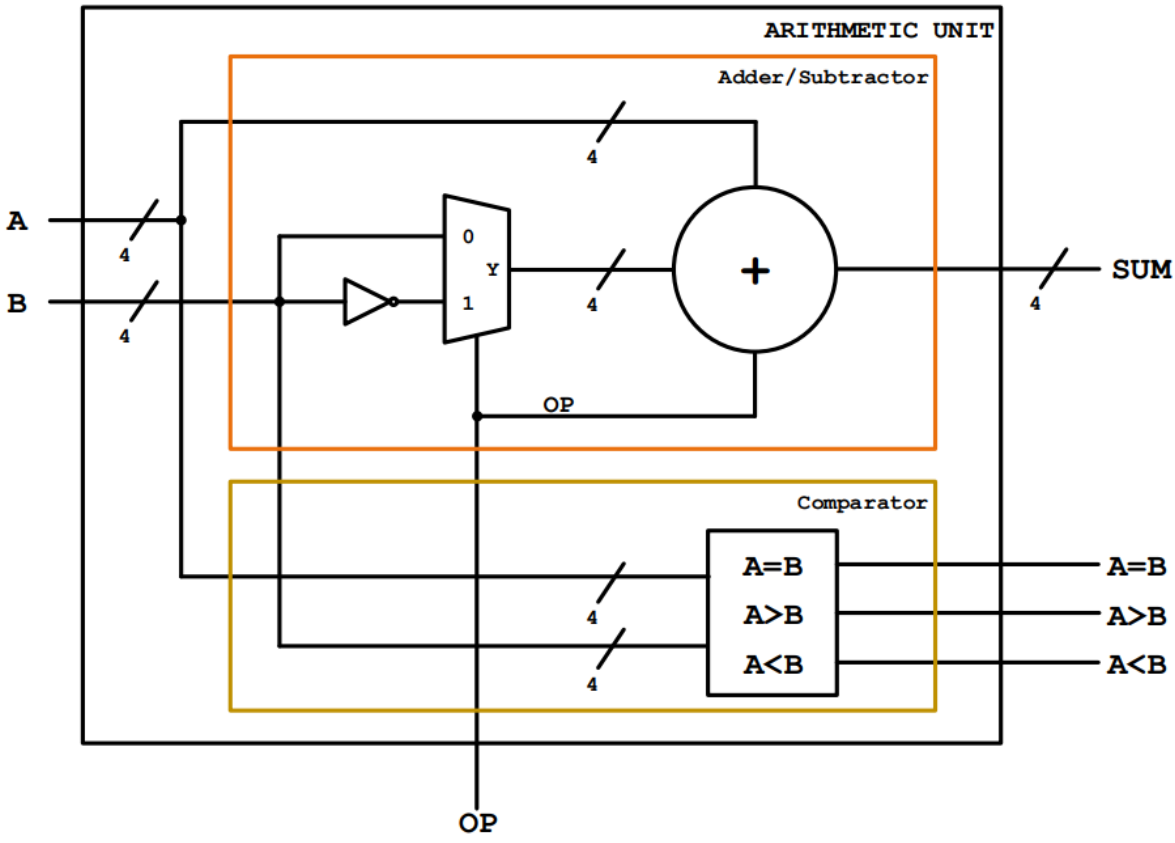


Figure 5: Arithmetic Unit

```
1
2  module ARITHMETIC_UNIT(A, B, OP, SUM, A_eq_B, A_big_B, B_big_A);
3
4  parameter size = 4;
5
6  input [size -1:0] A, B;
7  input OP;
8
9  output [size -1:0] SUM;
10 output A_eq_B, A_big_B, B_big_A;
11
12 Adder_Subtractor U1(A, B, OP, SUM);
13
14 four_bit_comparator U2(A, B, B_big_A, A_big_B, A_eq_B);
15
16 endmodule
```

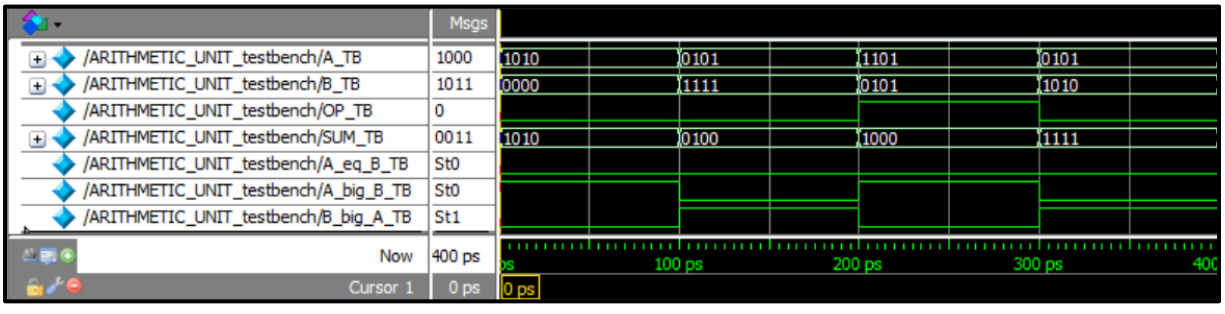
We combined the Adder_Subtractor and four_bit_comparator.

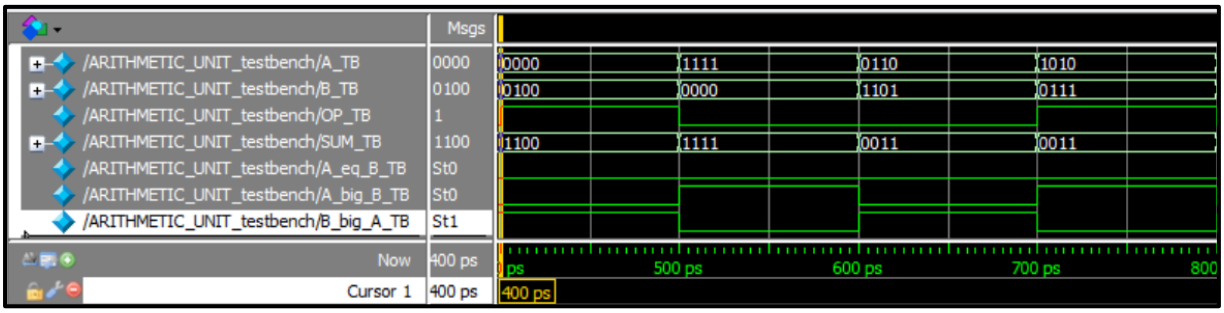
2 Write a testbench for your design using Modelsim® and simulate your arithmetic unit.

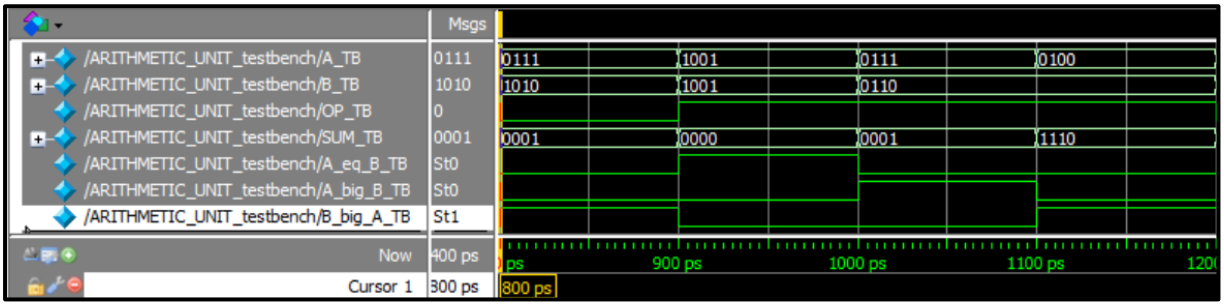
```
1
2   module ARITHMETIC_UNIT_testbench();
3
4   parameter size = 4;
5
6   reg [size -1:0] A_TB, B_TB;
7   reg OP_TB;
8
9   wire [size -1:0] SUM_TB;
10  wire A_eq_B_TB, A_big_B_TB, B_big_A_TB;
11
12  ARITHMETIC_UNIT DUT(A_TB, B_TB, OP_TB, SUM_TB, A_eq_B_TB, A_big_B_TB, B_big_A_TB);
13
14  initial
15
16  ⟶begin
17  ⟶
18  ⟶⟶OP_TB = 1'b0; A_TB = 4'b1010; B_TB = 4'b0000; #100;
19  ⟶⟶OP_TB = 1'b0; A_TB = 4'b0101; B_TB = 4'b1111; #100;
20  ⟶⟶OP_TB = 1'b1; A_TB = 4'b1101; B_TB = 4'b0101; #100;
21  ⟶⟶OP_TB = 1'b0; A_TB = 4'b0101; B_TB = 4'b1010; #100;
22  ⟶⟶OP_TB = 1'b1; A_TB = 4'b0000; B_TB = 4'b0100; #100;
23  ⟶⟶OP_TB = 1'b0; A_TB = 4'b1111; B_TB = 4'b0000; #100;
24  ⟶⟶OP_TB = 1'b0; A_TB = 4'b0110; B_TB = 4'b1101; #100;
25  ⟶⟶OP_TB = 1'b1; A_TB = 4'b1010; B_TB = 4'b0111; #100;
26  ⟶⟶OP_TB = 1'b0; A_TB = 4'b0111; B_TB = 4'b1010; #100;
27  ⟶⟶OP_TB = 1'b1; A_TB = 4'b1001; B_TB = 4'b1001; #100;
28  ⟶⟶OP_TB = 1'b1; A_TB = 4'b0111; B_TB = 4'b0110; #100;
29  ⟶⟶OP_TB = 1'b1; A_TB = 4'b0100; B_TB = 4'b0110; #100;
30  ⟶⟶OP_TB = 1'b0; A_TB = 4'b0110; B_TB = 4'b1110; #100;
31  ⟶⟶OP_TB = 1'b1; A_TB = 4'b0000; B_TB = 4'b0101; #100;
32  ⟶⟶OP_TB = 1'b0; A_TB = 4'b0011; B_TB = 4'b1010; #100;
33  ⟶⟶OP_TB = 1'b0; A_TB = 4'b1000; B_TB = 4'b1011; #100;
34  ⟶end
35
36  endmodule
```

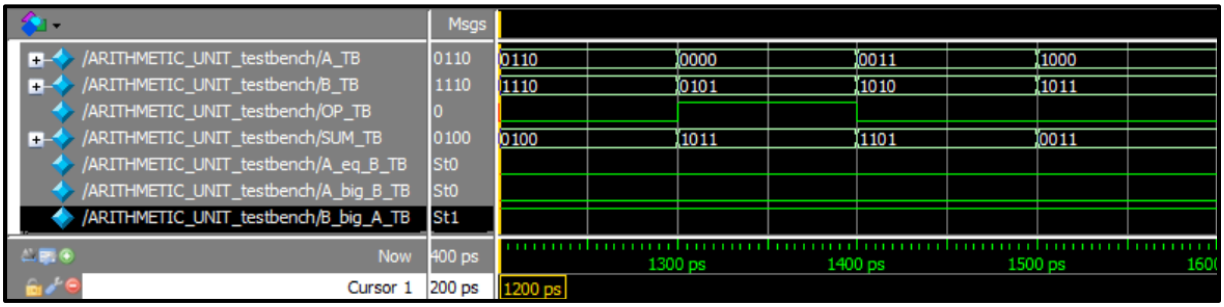
Random values are written for the testbench.

All outputs in waveform:

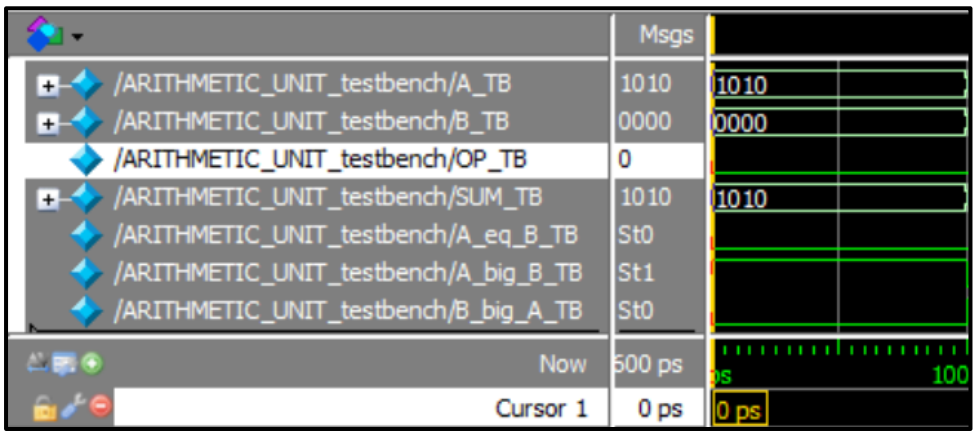




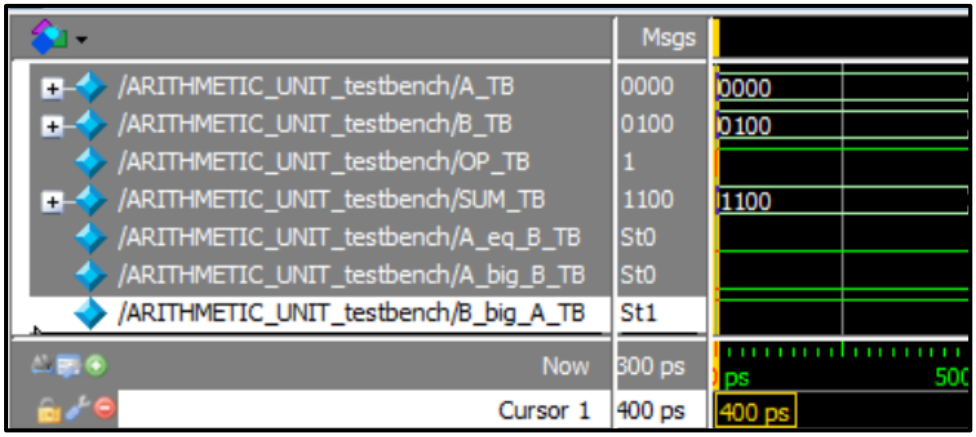




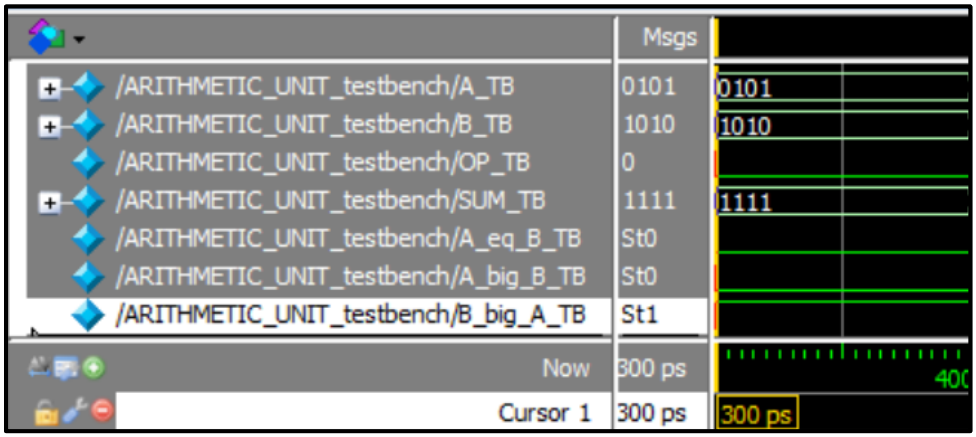
Let's check some of them:



1010 is bigger than 0000 so A > B is 1, A = B and B > A is 0.
Op is 0 so 1010 + 0000 = 1010.



0100 is bigger than 0000 so B > A is 1, A = B and A > B is 0. Op is 1 so 0000 + 1100 = 1100.

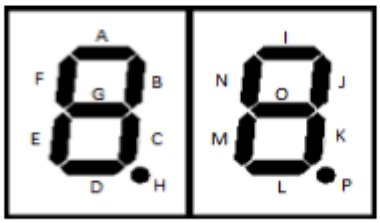


1010 is bigger than 0101 so B > A is 1, A = B and A > B is 0. Op is 0 so 0101 + 1010 = 1111.

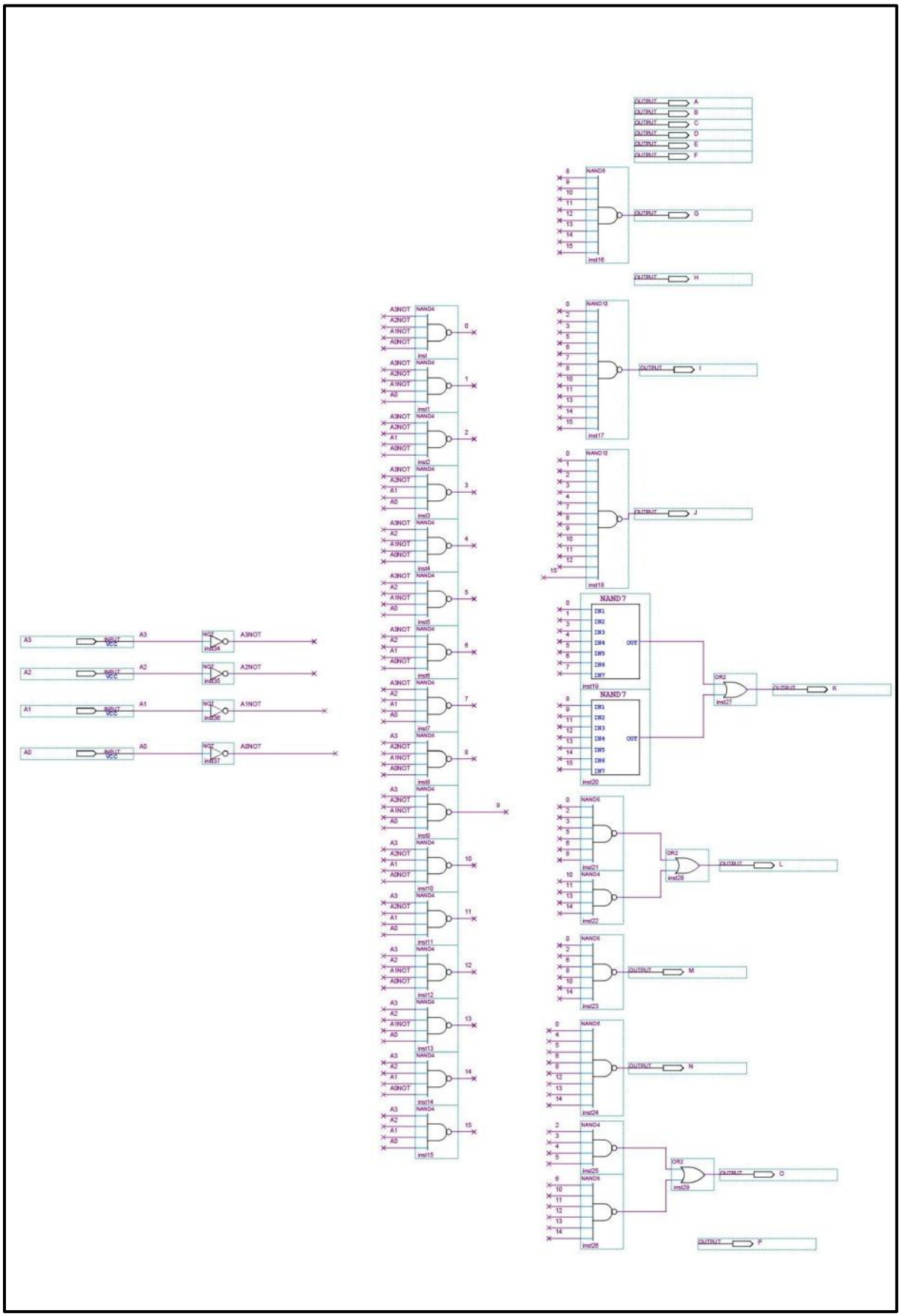
2.2.6: Decoder

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.

Table 3: Mapping of DE0 7-Segment Display to Decoder



BINARY VALUE	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
-8	OFF	OFF	OFF	OFF	OFF	OFF	ON	OFF	ON	ON	ON	ON	ON	ON	ON	OFF
-7	OFF	OFF	OFF	OFF	OFF	OFF	ON	OFF	ON	ON	ON	OFF	OFF	OFF	OFF	OFF
-6	. . .															
…	. . .															
+6	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	ON	OFF	ON	ON	ON	ON	ON	OFF
+7	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	ON	ON	ON	OFF	OFF	OFF	OFF	OFF


A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
NAND7
NAND7

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.

A decoder seperates n amount of inputs to 2^n amount of inputs. So our 4 inputs are going to 16 inputs.

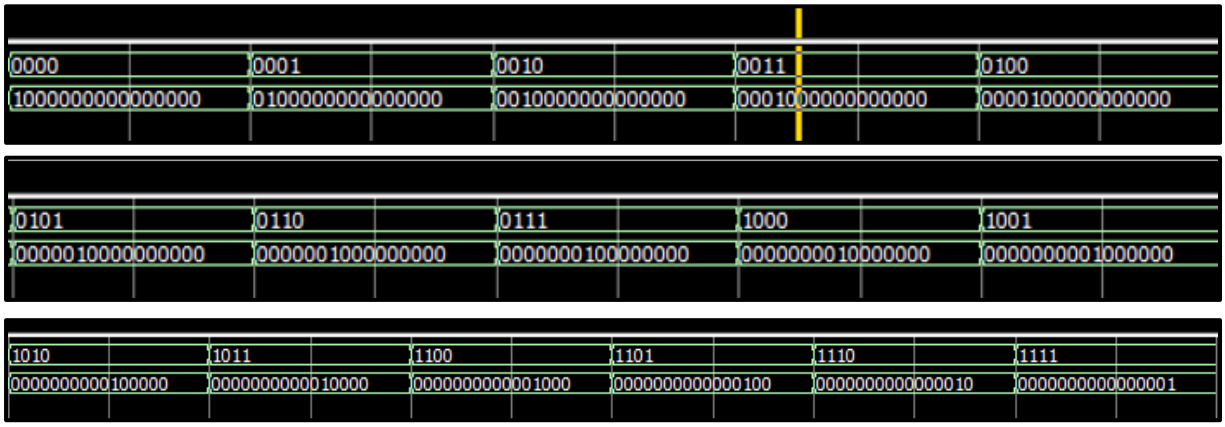
```
1
2   module decoder_4to16(output wire [15:0] F , input wire [3:0] ABCD);
3
4      assign F[15] = ~ABCD[3] & ~ABCD[2] & ~ABCD[1] & ~ABCD[0];
5      assign F[14] = ~ABCD[3] & ~ABCD[2] & ~ABCD[1] &  ABCD[0];
6      assign F[13] = ~ABCD[3] & ~ABCD[2] &  ABCD[1] & ~ABCD[0];
7      assign F[12] = ~ABCD[3] & ~ABCD[2] &  ABCD[1] &  ABCD[0];
8      assign F[11] = ~ABCD[3] &  ABCD[2] & ~ABCD[1] & ~ABCD[0];
9      assign F[10] = ~ABCD[3] &  ABCD[2] & ~ABCD[1] &  ABCD[0];
10     assign F[9]  = ~ABCD[3] &  ABCD[2] &  ABCD[1] & ~ABCD[0];
11     assign F[8]  = ~ABCD[3] &  ABCD[2] &  ABCD[1] &  ABCD[0];
12     assign F[7]  =  ABCD[3] & ~ABCD[2] & ~ABCD[1] & ~ABCD[0];
13     assign F[6]  =  ABCD[3] & ~ABCD[2] & ~ABCD[1] &  ABCD[0];
14     assign F[5]  =  ABCD[3] & ~ABCD[2] &  ABCD[1] & ~ABCD[0];
15     assign F[4]  =  ABCD[3] & ~ABCD[2] &  ABCD[1] &  ABCD[0];
16     assign F[3]  =  ABCD[3] &  ABCD[2] & ~ABCD[1] & ~ABCD[0];
17     assign F[2]  =  ABCD[3] &  ABCD[2] & ~ABCD[1] &  ABCD[0];
18     assign F[1]  =  ABCD[3] &  ABCD[2] &  ABCD[1] & ~ABCD[0];
19     assign F[0]  =  ABCD[3] &  ABCD[2] &  ABCD[1] &  ABCD[0];
20
21  endmodule
```

The decoder seperates 4 inputs to 16 inputs.

Enable	Inputs				outputs														
E	I_3	I_2	I_1	I_0	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
0	X	X	X	X	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
1	0	1	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
1	1	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

We refered to lab 3 decoder code for this behavioral decoder code.

All outputs in waveform:



Binary Input	Decimal equivalent	LEDs
0000	0	IJKLMN
0001	1	JK
0010	2	IJLMO
0011	3	IJKLO
0100	4	JKNO
0101	5	IKLNO
0110	6	IKLMNO
0111	7	IJK
1000	-8	GIJKLMNO
1001	-7	GIJK
1010	-6	GIKLMNO
1011	-5	GIKLNO
1100	-4	GJKNO
1101	-3	GIJKLO
1110	-2	GIJLMO
1111	-1	GJK

We used this table to transform the decoder into 7segment code.

I	0	2	3	5	6	7	-8	-2	-3	-5	-6			
J	0	1	2	3	4	7	-8	-1	-2	-3	-4			
K	0	1	3	4	5	6	7	-8	-1	-3	-4	-5	-6	
L	0	2	3	5	6	-8	-2	-3	-5	-6				
M	0	2	6	-8	-2	-6								
N	0	4	5	6	-8	-4	-5	-6						
O	2	3	4	5	6	-8	-2	-3	-4	-5	-6			
G	-1	-2	-3	-4	-5	-6	-8							

From this table we found what outputs corresponded to what

A3	A2	A1	A0	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
0	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

LED.

B15, B14, B12, B11, B9, B8, B7, B5, B4, B3, B2, B0 needs to be used.

```
module seven_segment(in,A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P);

input [3:0] in;
output A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P;
wire [15:0] w;

decoder_4to16 U1(w, in);

   assign A = 1;
   assign B = 1;
   assign C = 1;
   assign D = 1;
   assign E = 1;
   assign F = 1;
   assign H = 1;
   assign P = 1;


   assign I =  ((w[15] == 1) | (w[13] == 1) | (w[12] == 1) | (w[10] == 1) | (w[9] == 1) | (w[8] == 1) | (w[7] == 1) | (w[6] == 1) | (w[5] == 1) | (w[4] == 1) | (w[2] == 1) | (w[1] == 1)) ? 1'b0:
               1'b1;
   assign J =  ((w[15] == 1) | (w[14] == 1) | (w[13] == 1) | (w[12] == 1) | (w[11] == 1) | (w[8] == 1) | (w[7] == 1) | (w[6] == 1) | (w[3] == 1) | (w[2] == 1) | (w[1] == 1) | (w[0] == 1)) ? 1'b0:
               1'b1;
   assign K =  ((w[15] == 1) | (w[14] == 1) | (w[12] == 1) | (w[11] == 1) | (w[10] == 1) | (w[9] == 1) | (w[8] == 1) | (w[7] == 1) | (w[6] == 1) | (w[5] == 1) | (w[4] == 1) | (w[3] == 1) | (w[2] == 1) | (w[0] == 1)) ? 1'b0:
               1'b1;
   assign L =  ((w[15] == 1) | (w[13] == 1) | (w[12] == 1) | (w[10] == 1) | (w[9] == 1) | (w[7] == 1) | (w[5] == 1) | (w[4] == 1) | (w[2] == 1) | (w[1] == 1)) ? 1'b0:
               1'b1;
   assign M =  ((w[15] == 1) | (w[13] == 1) | (w[9] == 1) | (w[7] == 1) | (w[5] == 1) | (w[1] == 1)) ? 1'b0:
               1'b1;
   assign N =  ((w[15] == 1) | (w[11] == 1) | (w[10] == 1) | (w[9] == 1) | (w[7] == 1) | (w[5] == 1) | (w[4] == 1) | (w[3] == 1)) ? 1'b0:
               1'b1;
   assign O =  ((w[13] == 1) | (w[12] == 1) | (w[11] == 1) | (w[10] == 1) | (w[9] == 1) | (w[7] == 1) | (w[5] == 1) | (w[4] == 1) | (w[3] == 1) | (w[2] == 1) | (w[1] == 1)) ? 1'b0:
               1'b1;
   assign G =  ((w[7] == 1) | (w[6] == 1) | (w[5] == 1) | (w[4] == 1) | (w[3] == 1) | (w[2] == 1) | (w[1] == 1) | (w[0] == 1)) ? 1'b0:
               1'b1;


endmodule
```

Then this code was created.

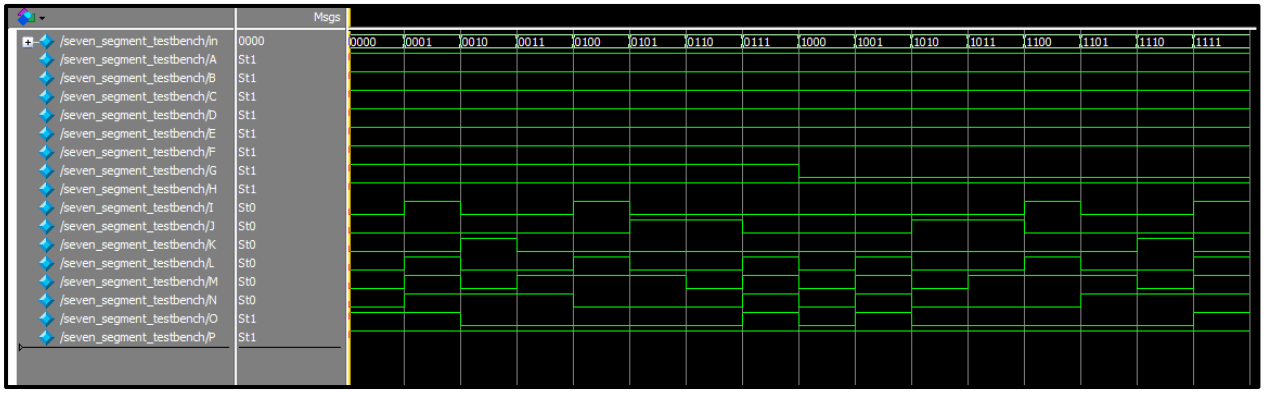
Only when one of B15, B14, B12, B11, B9, B8, B7, B5, B4, B3, B2, B0 are 1 will the output I be 0.

LED's are active-low so A, B, C, D, E, F, H, and P are assigned positive.

Seven Segment Testbench:

```
1
2   module seven_segment_testbench();
3
4   reg [3:0]in;
5   wire A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P;
6
7   seven_segment DUT(in,A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P);
8
9   initial
10     begin
11        in=4'b0000; #100;
12        in=4'b0001; #100;
13        in=4'b0010; #100;
14        in=4'b0011; #100;
15        in=4'b0100; #100;
16        in=4'b0101; #100;
17        in=4'b0110; #100;
18        in=4'b0111; #100;
19        in=4'b1000; #100;
20        in=4'b1001; #100;
21        in=4'b1010; #100;
22        in=4'b1011; #100;
23        in=4'b1100; #100;
24        in=4'b1101; #100;
25        in=4'b1110; #100;
26        in=4'b1111; #100;
27
28
29
30     end
31
32  endmodule
33
```

Seven Segment Simulation



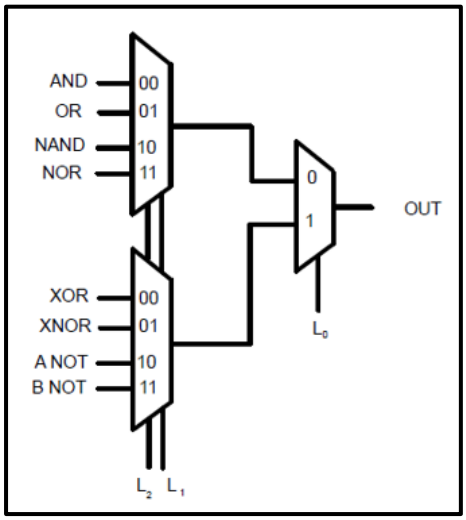
The outputs are the same as the table we created.

2.2.7: Logic Unit With Multiplexers

1 Using the basic gates in LAB 1, design the logic unit by combining with 4-to-1 and 2-to-1 Multiplexers as to fulfil the correct operation as described in Table 5.

Table 5: Logic Unit Operations

Select			Multiplexer Output
L2	L1	L0	F
0	0	0	A.B
0	0	1	A⊕B
0	1	0	A+B
0	1	1	A⊙B
1	0	0	(A.B)'
1	0	1	A'
1	1	0	(A+B)'
1	1	1	B'



We will need to connect the MUX's as such to give only one output.

2 Write a parametrized and structural verilog HDL code named as LOGIC_UNIT using the Verilog code that have been implemented in LAB 1 for basic gates and implement the logic unit designed in 2.2.7.1.

```
module logic_unit(A,B,L2,L1,L0,F);

parameter size=4;

input [size-1:0]A,B;
input L2,L1,L0;
output [size-1:0]F;
wire [size-1:0]w1,w2,w3,w4,w5,w6,w7,w8,w10,w11;

genvar i;

generate
for(i=0; i<size; i=i+1)begin: top

basic_gates U1(A[i],B[i],w1[i],w2[i],w3[i],w4[i],w5[i],w6[i],w7[i],w8[i]); //basic gates' outputs are sent to two different 4to1 mux's

MUX_4to1 U2(w1[i],w2[i],w3[i],w4[i],L2,L1,w10[i]); //
                                                   //4to1 MUX' outputs are sent to 2to1 MUX
MUX_4to1 U3(w5[i],w6[i],w7[i],w8[i],L2,L1,w11[i]);
                                                   //

MUX_2to1 U4(w10[i],w11[i],L0,F[i]); // output of 2to1 MUX is the output of logic unit
end
endgenerate

endmodule
```

We used the basic gates written in Project 1

```
module basic_gates(A,B,and_out,or_out,nand_out,nor_out,xor_out,xnor_out,anot_out,bnot_out);

input A,B;
output and_out,or_out,nand_out,nor_out,xor_out,xnor_out,anot_out,bnot_out;

and (and_out,A,B);
or (or_out,A,B);
nand (nand_out,A,B);
nor (nor_out,A,B);
xor (xor_out,A,B);
xnor (xnor_out,A,B);
not (anot_out,A);
not (bnot_out,B);

endmodule
```

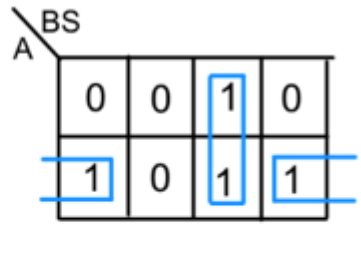
After the basic_gates give away their output, the MUX's generate. First the 4 to 1's then the 2 to 1. The output will be based on the multiplexers.

```
1
2  module MUX_2to1(A,B,S,F);
3
4  input A,B,S;
5  output F;
6  wire w1,w2,w3;
7
8  not U1(w1,S);
9
0  and U2(w2,A,w1);
1  and U3(w3,B,S);
2  or  U4(F,w2,w3);
3
4  ---->---->---->---->
5  endmodule
6
```

```
1
2  module MUX_4to1(A,B,C,D,S1,S0,F);
3
4  input A,B,C,D,S1,S0;
5  output F;
6  wire w1,w2;
7
8  MUX_2to1 U1(A,B,S0,w1);
9  MUX_2to1 U2(C,D,S0,w2);
0  MUX_2to1 U3(w1,w2,S1,F);
1
2  endmodule
```

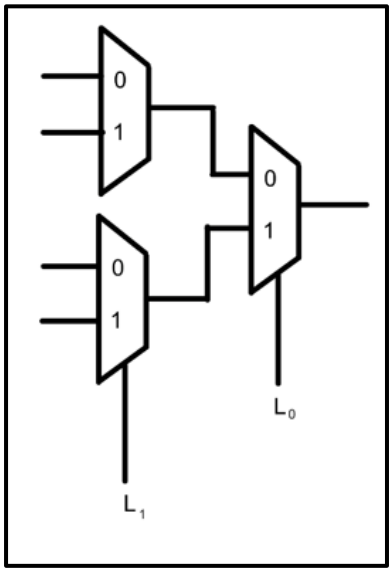
2 to 1 MUX was constructed with a KMAP and truth table:

A B S	F
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



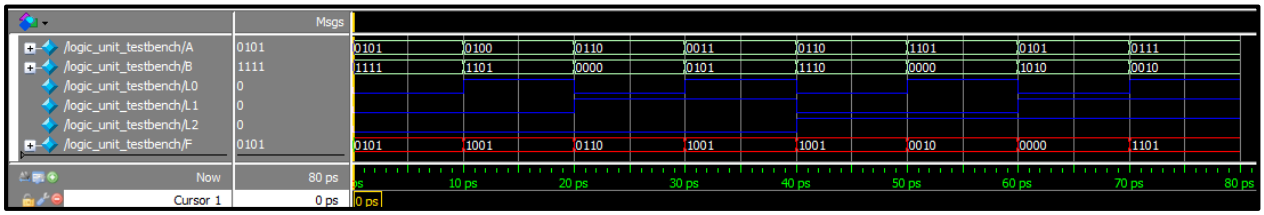
A'S+BS

4 to 1 MUX was constructed with the 2 to 1 MUX's.



Logic Unit Test Bench

```
module logic_unit_testbench();

reg [3:0]A,B;
reg L2,L1,L0;
wire [3:0]F;

logic_unit DUT(A,B,L2,L1,L0,F);

initial
begin
    L2=0; L1=0; L0=0;    A=0101; B=0111; #10;          //(L2,L1,L0) = (0,0,0) -> AND
    L2=0; L1=0; L0=1;    A=0100; B=1101; #10;          //(L2,L1,L0) = (0,0,1) -> XOR
    L2=0; L1=1; L0=0;    A=1110; B=0000; #10;          //(L2,L1,L0) = (0,1,0) -> OR
    L2=0; L1=1; L0=1;    A=1011; B=0101; #10;          //(L2,L1,L0) = (0,1,1) -> XNOR
    L2=1; L1=0; L0=0;    A=1110; B=0110; #10;          //(L2,L1,L0) = (1,0,0) -> NAND
    L2=1; L1=0; L0=1;    A=1101; B=0000; #10;          //(L2,L1,L0) = (0,0,1) -> A NOT
    L2=1; L1=1; L0=0;    A=0101; B=0010; #10;          //(L2,L1,L0) = (1,1,0) -> NOR
    L2=1; L1=1; L0=1;    A=1111; B=1010; #10;          //(L2,L1,L0) = (1,1,1) -> B NOT

end
endmodule
```



2.2.8: Arithmetic Logic Unit (ALU) Top Level Design

1 After all components required for ALU is designed, implemented and tested. Create a top level structural (hierarchical) Verilog HDL file called ALU and instantiate and connect all the components as depicted in Figure 2.

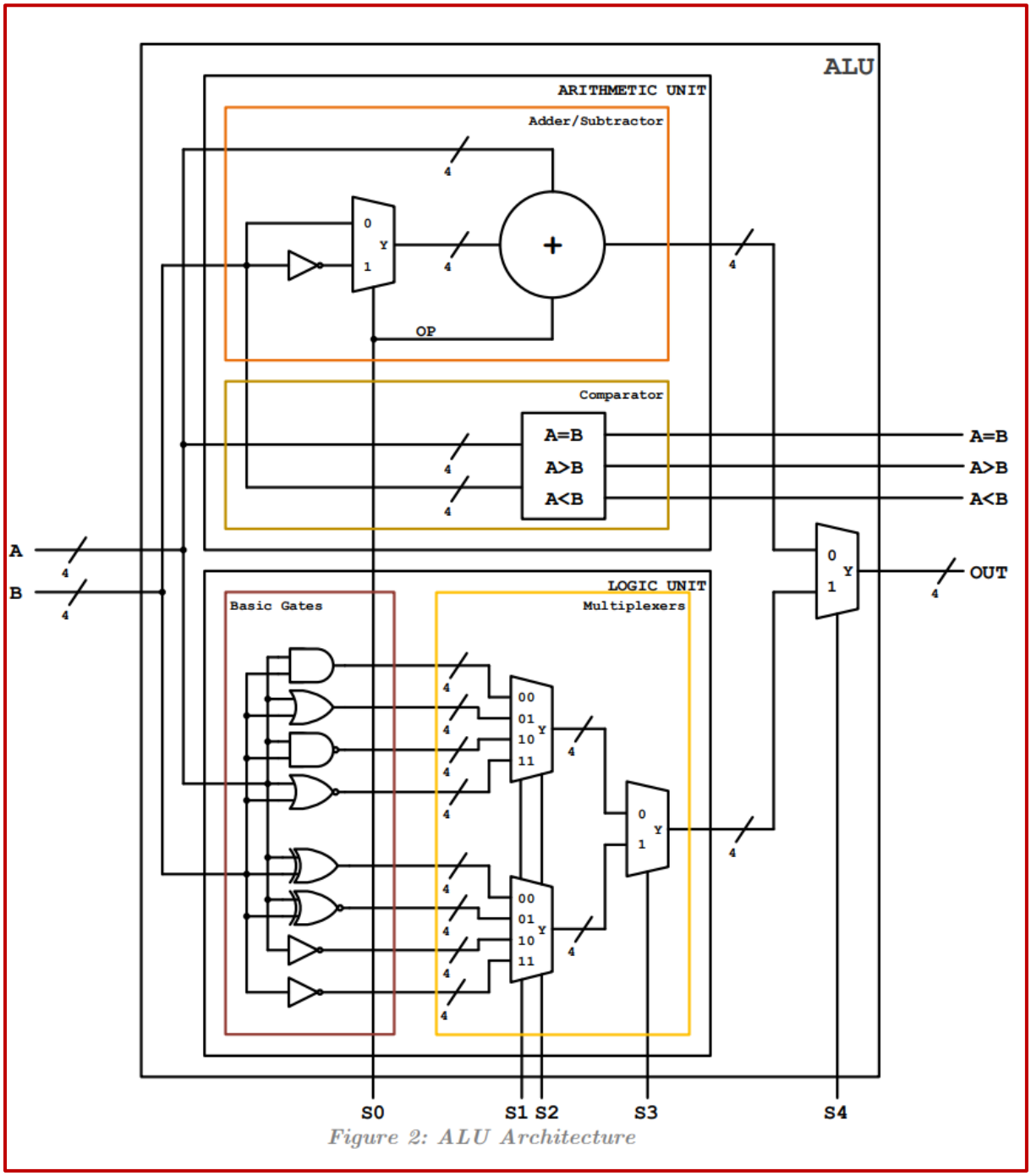
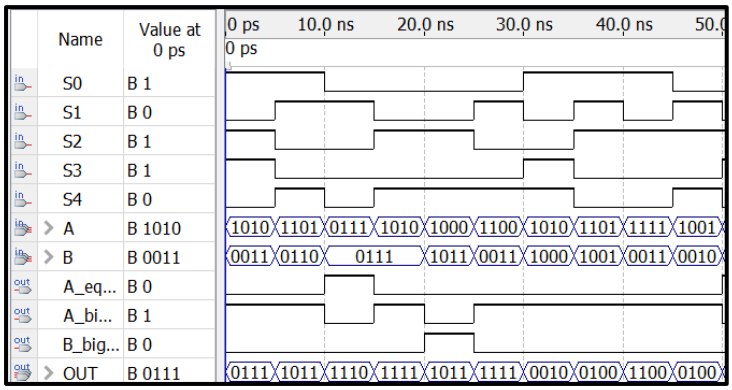


Figure 2: ALU Architecture

```
1
2  module ALU(A, B, S0, S1, S2, S3, S4, A_eq_B, A_big_B, B_big_A, OUT);
3
4  parameter size = 4;
5
6  input [size -1: 0] A, B;
7  input S0, S1, S2, S3, S4;
8
9  output [size -1: 0] OUT;
10 output A_eq_B, A_big_B, B_big_A;
11
12 wire [size -1:0] SUM;
13 wire [size -1:0] F;
14
15 genvar i;
16
17 ARITHMETIC_UNIT U1(A, B, S0, SUM, A_eq_B, A_big_B, B_big_A);
18 logic_unit U2(A, B, S1, S2, S3, F);
19
20 generate
21 ⟶for(i = 0; i < size; i = i + 1)begin: ALU
22 ⟶⟶MUX_2to1 U3(SUM[i], F[i], S4, OUT[i]);
23 ⟶end
24 endgenerate
25
26 endmodule
27
```

Op of arithmetic unit becomes, S0. L2, L1, L0 of logic unit becomes S1, S2, S3. And lastly there is a multiplexer with select S4 that ties them together.



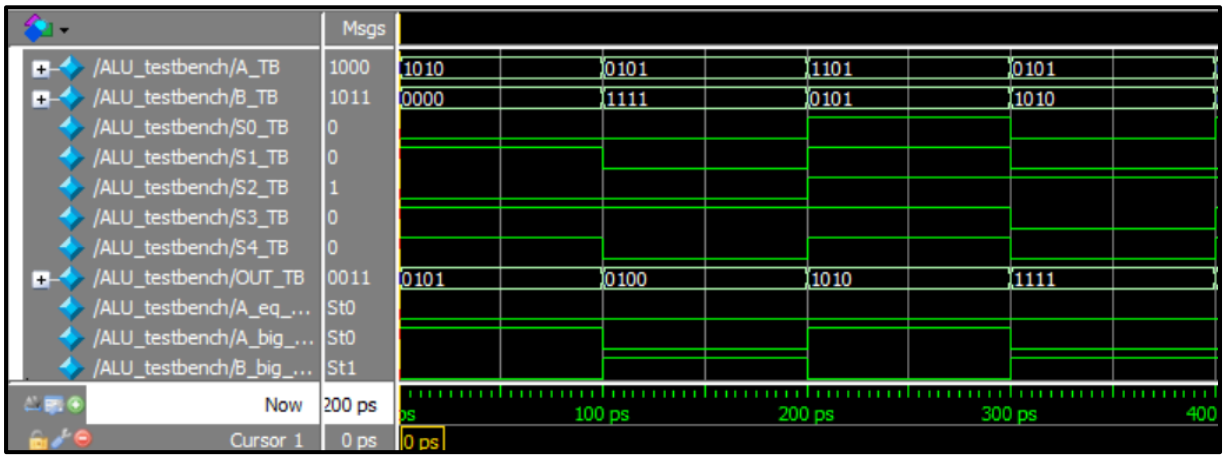
We checked the code using Quartus simulation.

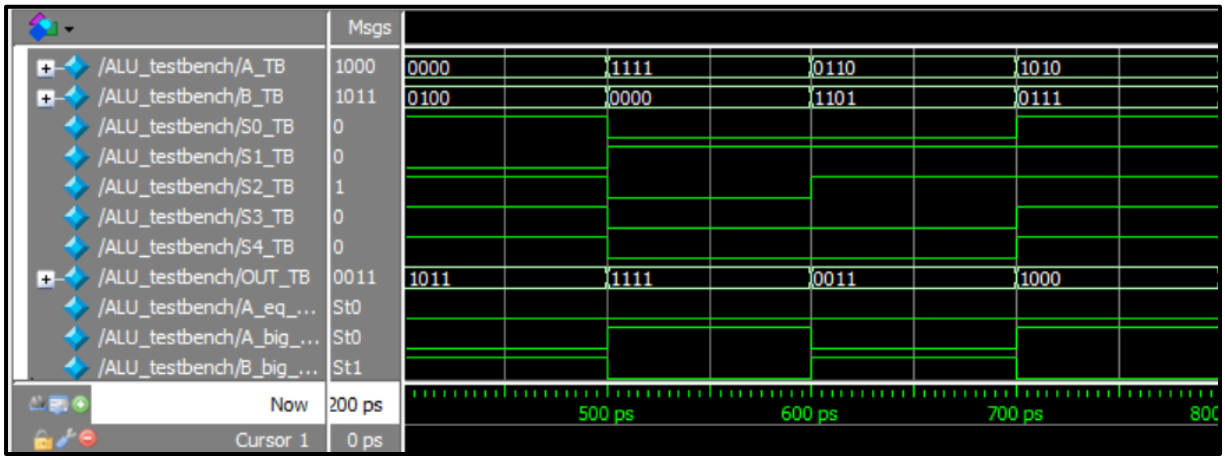
2 Write a testbench for your design using Modelsim® and simulate your ALU.

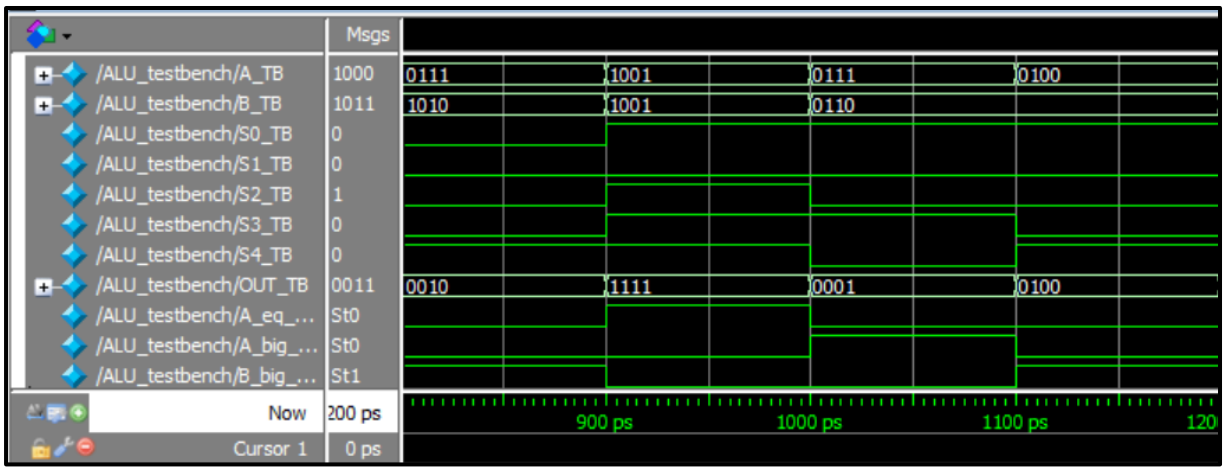
```
module ALU_testbench();

parameter size = 4;

reg [size -1:0] A_TB, B_TB;
reg S0_TB, S1_TB, S2_TB, S3_TB, S4_TB;

wire [size -1:0] OUT_TB;
wire A_eq_B_TB, A_big_B_TB, B_big_A_TB;

ALU DUT(A_TB, B_TB, S0_TB, S1_TB, S2_TB, S3_TB, S4_TB, A_eq_B_TB, A_big_B_TB, B_big_A_TB, OUT_TB);

initial
	
begin
	
	S0_TB = 1'b0; S1_TB = 1'b1; S2_TB = 1'b0; S3_TB = 1'b1; S4_TB = 1'b1; A_TB = 4'b1010; B_TB = 4'b0000; #100;
	S0_TB = 1'b0; S1_TB = 1'b0; S2_TB = 1'b0; S3_TB = 1'b1; S4_TB = 1'b0; A_TB = 4'b0101; B_TB = 4'b1111; #100;
	S0_TB = 1'b1; S1_TB = 1'b1; S2_TB = 1'b1; S3_TB = 1'b1; S4_TB = 1'b1; A_TB = 4'b1101; B_TB = 4'b0101; #100;
	S0_TB = 1'b0; S1_TB = 1'b0; S2_TB = 1'b1; S3_TB = 1'b0; S4_TB = 1'b0; A_TB = 4'b0101; B_TB = 4'b1010; #100;
	S0_TB = 1'b1; S1_TB = 1'b0; S2_TB = 1'b1; S3_TB = 1'b1; S4_TB = 1'b1; A_TB = 4'b0000; B_TB = 4'b0100; #100;
	S0_TB = 1'b0; S1_TB = 1'b1; S2_TB = 1'b0; S3_TB = 1'b0; S4_TB = 1'b0; A_TB = 4'b1111; B_TB = 4'b0000; #100;
	S0_TB = 1'b0; S1_TB = 1'b1; S2_TB = 1'b1; S3_TB = 1'b0; S4_TB = 1'b0; A_TB = 4'b0110; B_TB = 4'b1101; #100;
	S0_TB = 1'b1; S1_TB = 1'b1; S2_TB = 1'b1; S3_TB = 1'b1; S4_TB = 1'b1; A_TB = 4'b1010; B_TB = 4'b0111; #100;
	S0_TB = 1'b0; S1_TB = 1'b0; S2_TB = 1'b0; S3_TB = 1'b0; S4_TB = 1'b1; A_TB = 4'b0111; B_TB = 4'b1010; #100;
	S0_TB = 1'b1; S1_TB = 1'b0; S2_TB = 1'b1; S3_TB = 1'b1; S4_TB = 1'b1; A_TB = 4'b1001; B_TB = 4'b1001; #100;
	S0_TB = 1'b1; S1_TB = 1'b0; S2_TB = 1'b0; S3_TB = 1'b1; S4_TB = 1'b0; A_TB = 4'b0111; B_TB = 4'b0110; #100;
	S0_TB = 1'b1; S1_TB = 1'b0; S2_TB = 1'b0; S3_TB = 1'b0; S4_TB = 1'b1; A_TB = 4'b0100; B_TB = 4'b0110; #100;
	S0_TB = 1'b0; S1_TB = 1'b1; S2_TB = 1'b0; S3_TB = 1'b0; S4_TB = 1'b0; A_TB = 4'b0110; B_TB = 4'b1110; #100;
	S0_TB = 1'b1; S1_TB = 1'b0; S2_TB = 1'b1; S3_TB = 1'b1; S4_TB = 1'b1; A_TB = 4'b0000; B_TB = 4'b0101; #100;
	S0_TB = 1'b0; S1_TB = 1'b1; S2_TB = 1'b0; S3_TB = 1'b1; S4_TB = 1'b1; A_TB = 4'b0011; B_TB = 4'b1010; #100;
	S0_TB = 1'b0; S1_TB = 1'b0; S2_TB = 1'b1; S3_TB = 1'b0; S4_TB = 1'b0; A_TB = 4'b1000; B_TB = 4'b1011; #100;
	
	
end
endmodule
```

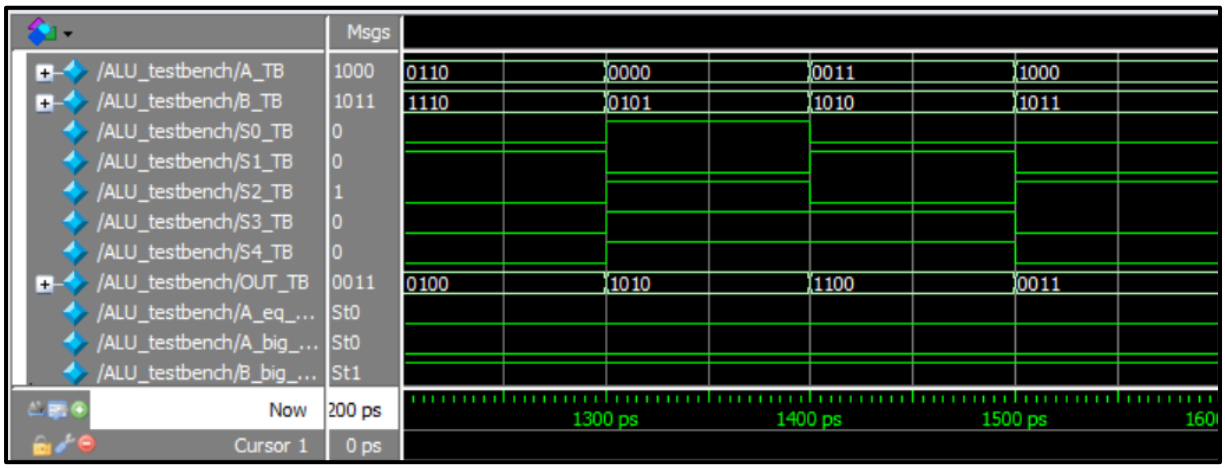
Random values are written for the testbench.

All outputs in waveform:




Msgs
/ALU_testbench/A_TB
/ALU_testbench/B_TB
/ALU_testbench/S0_TB
/ALU_testbench/S1_TB
/ALU_testbench/S2_TB
/ALU_testbench/S3_TB
/ALU_testbench/S4_TB
/ALU_testbench/OUT_TB
/ALU_testbench/A_eq_...
/ALU_testbench/A_big_...
/ALU_testbench/B_big_...
1000
1011
0
0
1
0
0
0011
St0
St0
St1
0000
0100
1111
0000
0110
1101
1010
0111
1011
1111
0011
1000
Now
200 ps
Cursor 1
0 ps
500 ps
600 ps
700 ps


Msgs
/ALU_testbench/A_TB
/ALU_testbench/B_TB
/ALU_testbench/S0_TB
/ALU_testbench/S1_TB
/ALU_testbench/S2_TB
/ALU_testbench/S3_TB
/ALU_testbench/S4_TB
/ALU_testbench/OUT_TB
/ALU_testbench/A_eq_...
/ALU_testbench/A_big_...
/ALU_testbench/B_big_...
1000
1011
0
0
1
0
0
0011
St0
St0
St1
0111
1010
1001
1001
0111
0110
0100
0010
1111
0001
0100
Now
200 ps
Cursor 1
0 ps
900 ps
1000 ps
1100 ps


Msgs
/ALU_testbench/A_TB
/ALU_testbench/B_TB
/ALU_testbench/S0_TB
/ALU_testbench/S1_TB
/ALU_testbench/S2_TB
/ALU_testbench/S3_TB
/ALU_testbench/S4_TB
/ALU_testbench/OUT_TB
/ALU_testbench/A_eq_...
/ALU_testbench/A_big_...
/ALU_testbench/B_big_...
1000
1011
0
0
1
0
0
0011
St0
St0
St1
0110
1110
0000
0101
0011
1010
1000
1011
0100
1010
1100
0011
Now
200 ps
Cursor 1
0 ps
1300 ps
1400 ps
1500 ps

The last multiplexer will determine which operations will be done.

If S4 = 0 then,

In arithmetic unit:

If S0 = 1:

Subtraction.

If S0 = 0:

Addition.

If S4 = 1 then,

In logic unit:

If S3 = 1:

If S2 = 1:

If S1 = 1:

B'

If S1 = 0;

A xnor B

If S2 = 0:

If S1 = 1:

A'

If S0 = 0:

A xor B

If S3 = 0:

If S2 = 1:

If S1 = 1:

A nor B

If S1 = 0:

A or B

If S2 = 0:

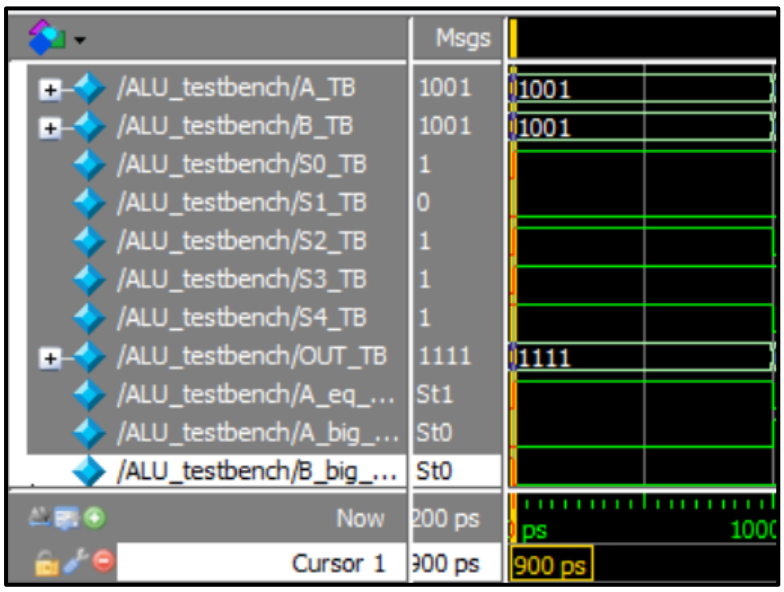
If S1 = 1:

A nand B

If S1 = 0:

A and B

Let’s check some of them:



1001 = 1001

A = B → 1

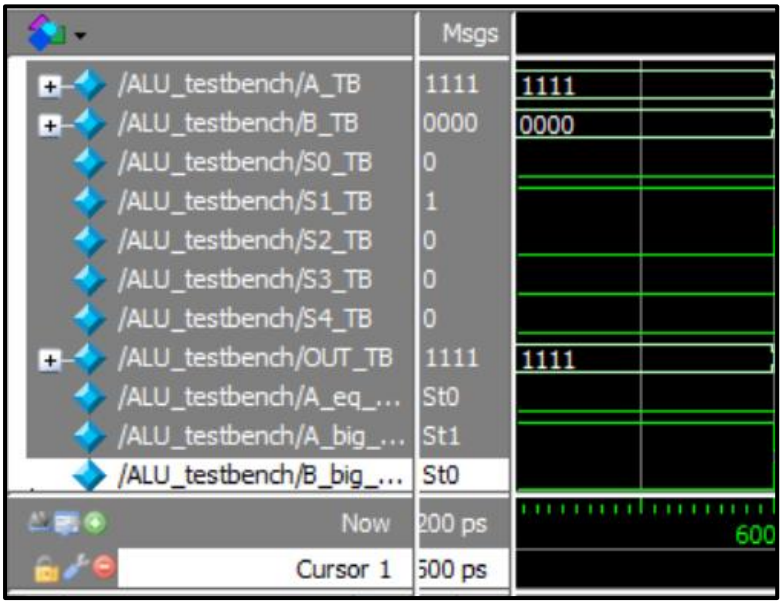
S4 = 1: logic unit

S3 = 1:

S2 = 1:

S1 = 0: A xnor B

1001 xnor 1001 = 1111



1111 > 0000

A > B → 1

S4 = 0: arithmetic unit

S0 = 0: addition

1111 + 0000 = 1111