EEE 248/ CNG 232
Logic Design

Project 1

Yıldız Alara Köymen

2453389

TABLE OF CONTENTS

1.1 Introduction

This experiment is an introduction to logic design. Our first objective in section 1.2.1 is to design a logic unit. This logic unit has two inputs and eight outputs consisting of eight operations: and, or, nand, nor, xor, xnor, not A and not B. These are the basic gates we have learned during the course. We used structural code for each logic gate and later simulated them using MODELSIM. The MODELSIM simulation helped us to abstract and find out whether the code was working correctly or not.

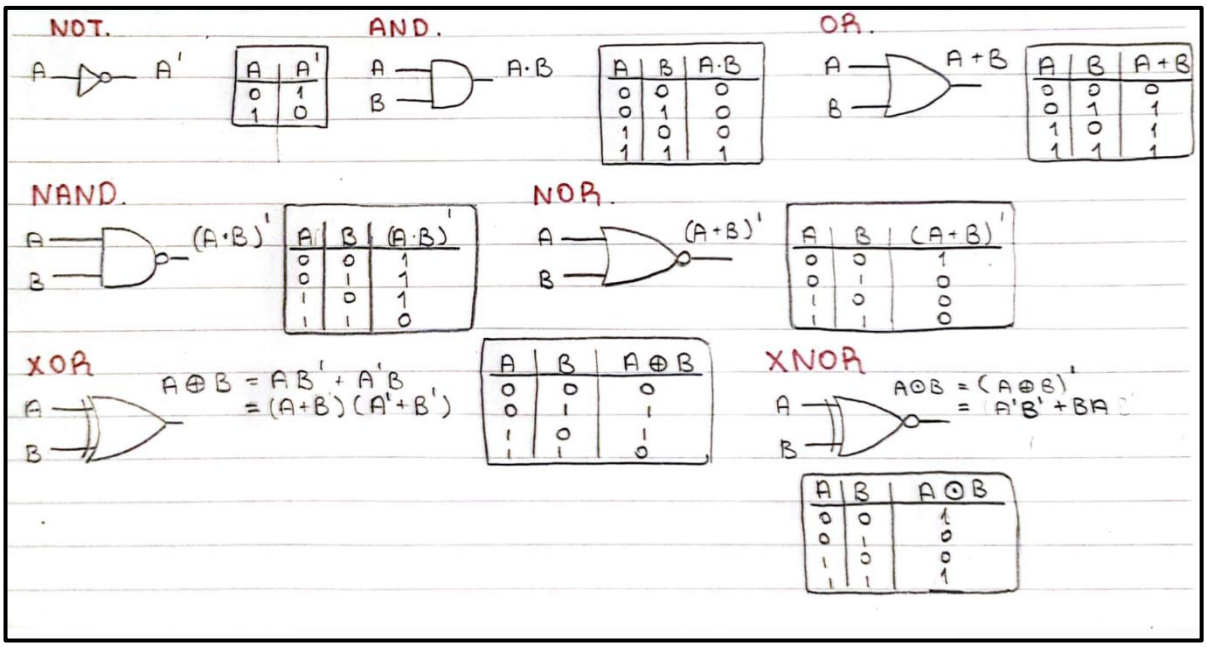
In section 1.2.2 we were asked to design a bakery profit calculator. This calculator must work based on the rules made for the bakery. For example, it is not possible for the bakery to produce more than two kinds of pastries. Based on rules such as this we drew a truth table, putting don’t cares for impossible cases. Then the Karnaugh map method was used to find an equation. This formula created a base line for us to draw a schematic and code the Verilog HDL code necessary for the design.

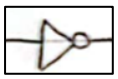
1.2 Preliminary Work: Logic Unit

1. For the logic gates given below.

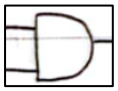
   i. Derive the truth table and Boolean expression for each logic gate.

   ii. Draw the corresponding logic schematics for all gates.

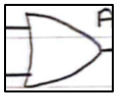




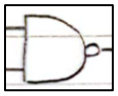
NOT: The gate works as an inverter; it takes the input and derives it as its complement. A 1 input becomes 0 and a 0 input becomes 1.



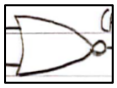
AND: The and gate only returns true if all the values are 1. We can think of it as the universal quantifier.



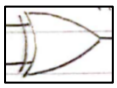
OR: The or gate only returns false if all the values are false. We can think of it as the existential quantifier.



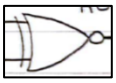
NAND: The nand gate is the inverse of the and statement. We can think of it as an and gate followed by a not gate.



NOR: The nor gate is the inverse of the or statement. We can think of it as an or gate followed by a not gate.



XOR: The xor gate will return false if both the inputs are the same.



XNOR: The xnor gate will return false if both the inputs are different. We can think of it as a xor gate with a not gate after it.

2.

i. Write a structural Verilog code for each logic gate and simulate using Modelsim. In your report show your simulation outputs.

To write a structural code we need to implement the lower-level sub systems already included in Quartus. This means that instead of assigning a desired outcome to an output, these logic functions are used instead. For example, while writing the module *not_gate* we are required to use the not(output, input) function. This situation is paralleled for the other structural codes written for this question. To achieve the structural method we used these functions for each module:

```
module not_gate: not(output, input)
module and_gate: and(output, input)
module or_gate: or(output, input)
module nand_gate: nand(output, input)
module nor_gate: nor(output, input)
module xor_gate: xor(output, input)
module xnor_gate: xnor(output, input)
```

After writing the desired code for each gate, MODELSIM was a must to simulate them. Before the simulation we needed to write testbench codes for the gates. The testbench generates the input conditions and drives them into the ports of the DUT. DUT stands for Device Under Test. *reg* and wire is used for testbench codes instead of input and output as the outputs don't store any value during simulation. The output value is driven by the DUT not the testbench. In essence, An *initial* block drives in a series of stimulus patterns that are written resembling a truth table. *#100* written on each line means that every stimulus pattern is driven for one hundred pico second intervals. MODELSIM takes these input patterns and simulates the gates. We can observe the waveforms to see if the outputs are accurate. We can use this information to confirm the truth tables we drew for 1.i.

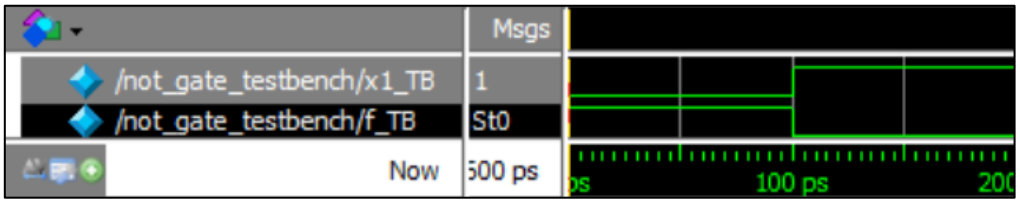
NOT GATE:

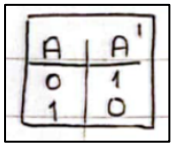
```
1
2  module not_gate(x1, f); //structural code
3
4  output f;
5  input x1;
6
7  not (f,x1);
8
9  endmodule
```

NOT GATE TESTBENCH:

```
1
2  module not_gate_testbench();
3
4  reg x1_TB;
5  wire f_TB;
6
7  not_gate DUT(x1_TB, f_TB);
8
9  initial
0
1      begin
2
3          x1_TB = 1'b0; #100;
4          x1_TB = 1'b1; #100;
5
6      end
7
8  endmodule
```

NOT GATE MODELSIM TESTBENCH SIMULATION:





The input x1 and the output are accurate for the truth table. While the input is 0 the output is 1, while the input is 1, the output is 0.

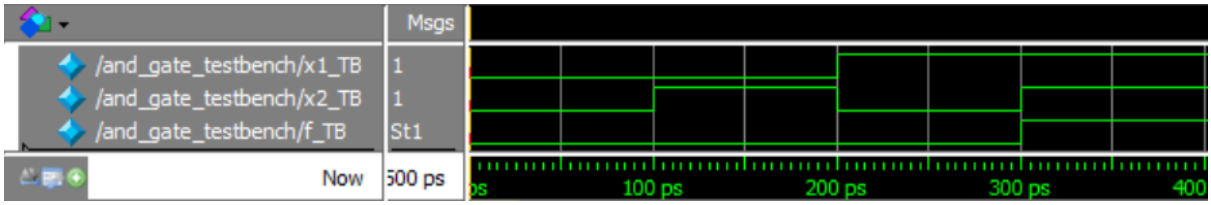
AND GATE:

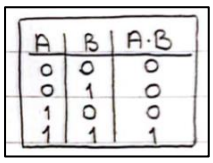
```
1
2  module and_gate(x1, x2, f); //structural code
3
4  input x1, x2;
5  output f;
6
7  and(f, x1, x2);
8
9  endmodule
```

AND GATE TESTBENCH:

```
1
2  module and_gate_testbench();
3
4  reg x1_TB, x2_TB;
5
6  wire f_TB;
7
8  and_gate DUT(x1_TB, x2_TB, f_TB);
9
10 initial
11
12    begin
13
14        x1_TB = 1'b0; x2_TB = 1'b0; #100;
15        x1_TB = 1'b0; x2_TB = 1'b1; #100;
16        x1_TB = 1'b1; x2_TB = 1'b0; #100;
17        x1_TB = 1'b1; x2_TB = 1'b1; #100;
18
19    end
20
21 endmodule
```

AND GATE MODELSIM TESTBENCH SIMULATION:





The output was only 1 during the 300 – 400 PS mark when both the inputs were 1.

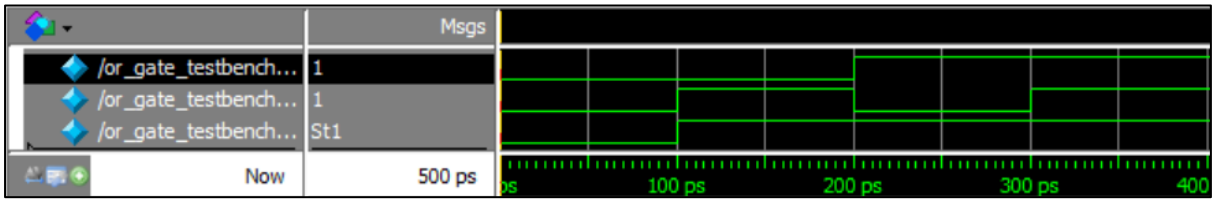
OR GATE:

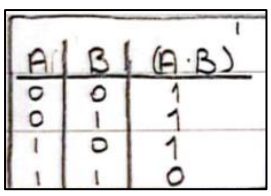
```
1
2  module or_gate(x1, x2, f); //structural code
3
4  input x1, x2;
5
6  output f;
7
8  or(f, x1, x2);
9
10 endmodule
```

OR GATE TESTBENCH:

```
1
2  module or_gate_testbench();
3
4  reg x1_TB, x2_TB;
5
6  wire f_TB;
7
8  or_gate DUT(x1_TB, x2_TB, f_TB);
9
10 initial
11
12    begin
13
14       x1_TB = 1'b0; x2_TB = 1'b0; #100;
15       x1_TB = 1'b0; x2_TB = 1'b1; #100;
16       x1_TB = 1'b1; x2_TB = 1'b0; #100;
17       x1_TB = 1'b1; x2_TB = 1'b1; #100;
18
19    end
20
21 endmodule
```

OR GATE MODELSIM TESTBENCH SIMULATION:





The output was only 0 during the 0 – 100 PS mark when both the inputs were 0.

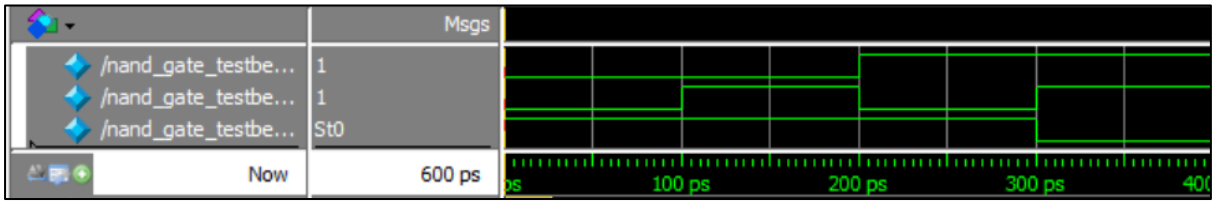
NAND GATE:

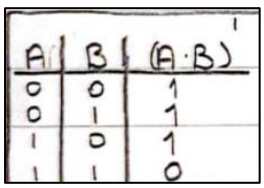
```
1
2  module nand_gate(x1, x2, f); //structural code
3
4  input x1, x2;
5  output f;
6
7  nand(f, x1, x2);
8
9  endmodule
```

NAND GATE TESTBENCH:

```
1
2  module nand_gate_testbench();
3
4  reg x1_TB, x2_TB;
5
6  wire f_TB;
7
8  nand_gate DUT(x1_TB, x2_TB, f_TB);
9
0  initial
1
2     begin
3
4        x1_TB = 1'b0; x2_TB = 1'b0; #100;
5        x1_TB = 1'b0; x2_TB = 1'b1; #100;
6        x1_TB = 1'b1; x2_TB = 1'b0; #100;
7        x1_TB = 1'b1; x2_TB = 1'b1; #100;
8
9     end
0
1  endmodule
```

NAND GATE MODELSIM TESTBENCH SIMULATION:





The output was only 0 during the 300 – 400 PS mark when both the inputs were 1.

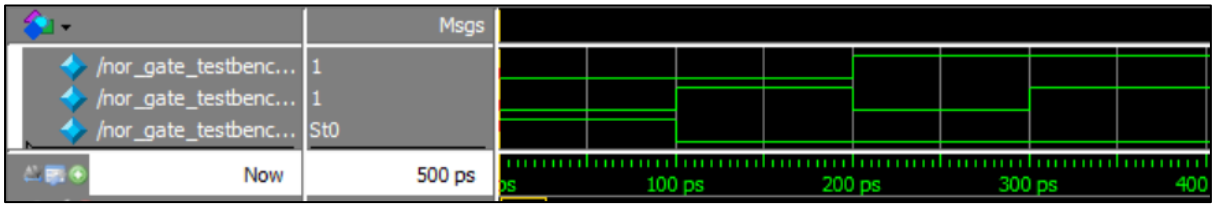
NOR GATE:

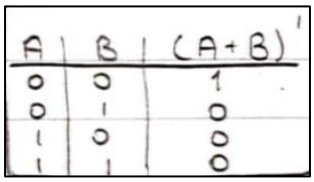
```
1
2  module nor_gate(x1, x2, f); //structural code
3
4  input x1, x2;
5  output f;
6
7  nor(f, x1, x2);
8
9  endmodule
```

NOR GATE TESTBENCH:

```
1
2  module nor_gate_testbench();
3
4  reg x1_TB, x2_TB;
5
6  wire f_TB;
7
8  nor_gate DUT(x1_TB, x2_TB, f_TB);
9
10 initial
11
12    begin
13
14       x1_TB = 1'b0; x2_TB = 1'b0; #100;
15       x1_TB = 1'b0; x2_TB = 1'b1; #100;
16       x1_TB = 1'b1; x2_TB = 1'b0; #100;
17       x1_TB = 1'b1; x2_TB = 1'b1; #100;
18
19    end
20
21 endmodule
```

NOR GATE MODELSIM TESTBENCH SIMULATION:





The output was only 1 during the 0 – 100 PS mark when both the inputs were 0.

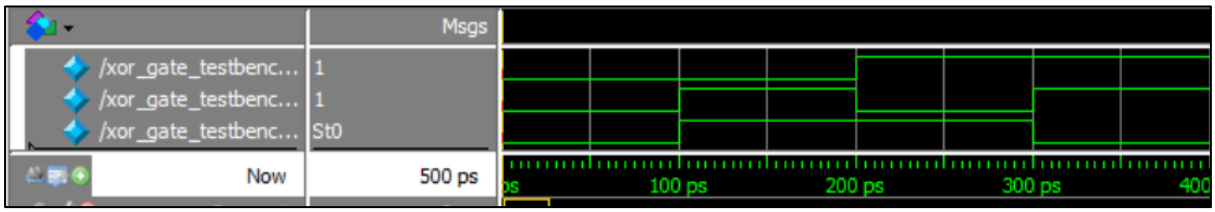
XOR GATE:

```
1
2  module xor_gate(x1, x2, f); //structural code
3
4  input x1, x2;
5  output f;
6
7  xor(f, x1, x2);
8
9  endmodule
```

XOR GATE TESTBENCH:

```
1
2  module xor_gate_testbench();
3
4  reg x1_TB, x2_TB;
5
6  wire f_TB;
7
8  xor_gate DUT(x1_TB, x2_TB, f_TB);
9
10 initial
11
12    begin
13
14       x1_TB = 1'b0; x2_TB = 1'b0; #100;
15       x1_TB = 1'b0; x2_TB = 1'b1; #100;
16       x1_TB = 1'b1; x2_TB = 1'b0; #100;
17       x1_TB = 1'b1; x2_TB = 1'b1; #100;
18
19    end
20
21 endmodule
```

XOR GATE MODELSIM TESTBENCH SIMULATION:



A	B	A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0

The outputs were 1 during the 100 – 200 mark when the inputs were 0,1 and the 200 – 300 mark when the inputs were 1,0.

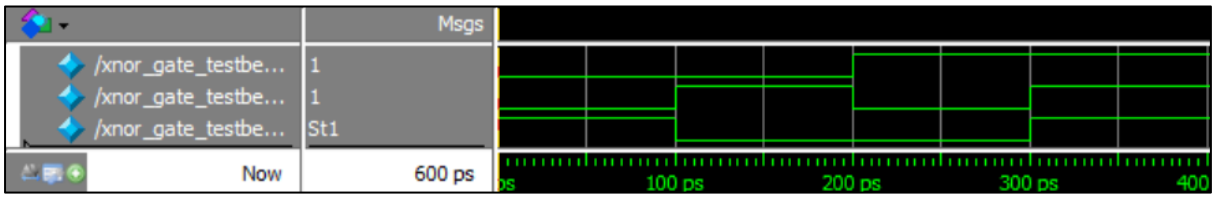
XNOR GATE:

```
1
2  module xnor_gate(x1, x2, f); //structural code
3
4  input x1, x2;
5  output f;
6
7  xnor(f, x1, x2);
8
9  endmodule
```

XNOR GATE TESTBENCH:

```
1
2  module xnor_gate_testbench();
3
4  reg x1_TB, x2_TB;
5
6  wire f_TB;
7
8  xnor_gate DUT(x1_TB, x2_TB, f_TB);
9
10 initial
11
12    begin
13
14       x1_TB = 1'b0; x2_TB = 1'b0; #100;
15       x1_TB = 1'b0; x2_TB = 1'b1; #100;
16       x1_TB = 1'b1; x2_TB = 1'b0; #100;
17       x1_TB = 1'b1; x2_TB = 1'b1; #100;
18
19    end
20
21 endmodule
```

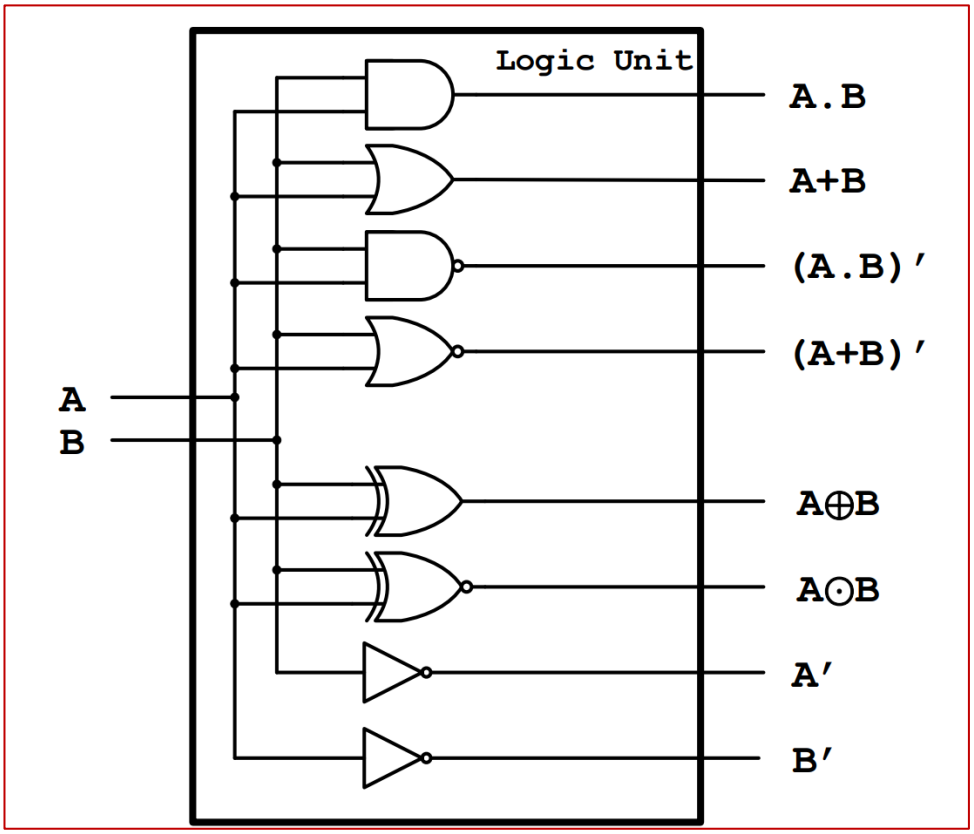
XNOR GATE MODELSIM TESTBENCH SIMULATION:



A	B	A ⊙ B
0	0	1
0	1	0
1	0	0
1	1	1

The outputs were 1 during the 0 – 100 mark when the inputs were 0,0 and the 300 – 400 mark when the inputs were 1,1.

3. Write a structural Verilog code and implement all logic gates together as depicted in figure, simulate your design, and verify the functionality using Modelsim. Compare your results with 1.i and 1.ii. This implementation will then be used in PROJECT 2 as logic unit

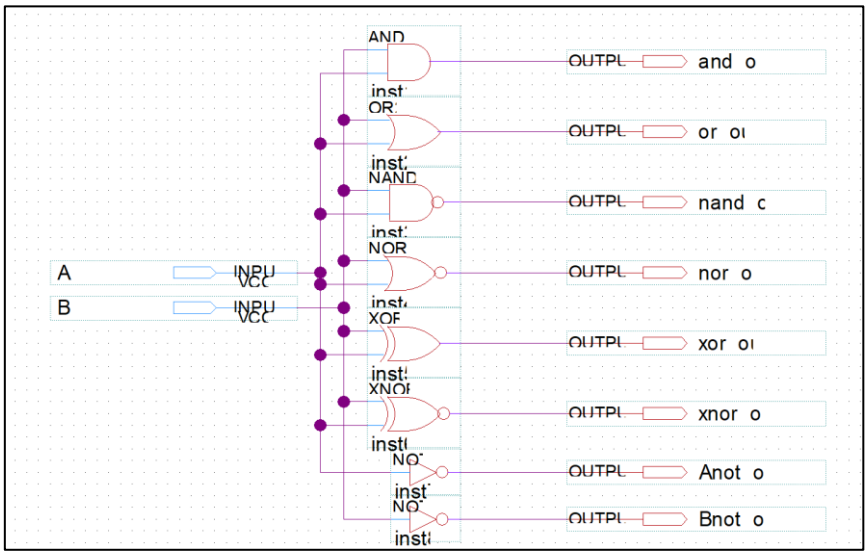


LOGIC UNIT STRUCTURAL VERILOG CODE:

```
1
2  module logic_unit(A, B, and_out, or_out, nand_out, nor_out, xor_out, xnor_out, Anot_out, Bnot_out);
3
4  input A, B;
5  output and_out, or_out, nand_out, nor_out, xor_out, xnor_out, Anot_out, Bnot_out;
6
7  and(and_out, A, B);
8  or(or_out, A, B);
9  nand(nand_out, A, B);
10 nor(nor_out, A, B);
11 xor(xor_out, A, B);
12 xnor(xnor_out, A, B);
13 not(Anot_out, A);
14 not(Bnot_out, B);
15
16 endmodule
```

We used the same functions as before to implement this code. There are eight functions and eight outputs for each function. We used the gates that we had used in section 1.2.1. The structural code is written as explained on page 6.

SCHEMATIC OF LOGIC UNIT:



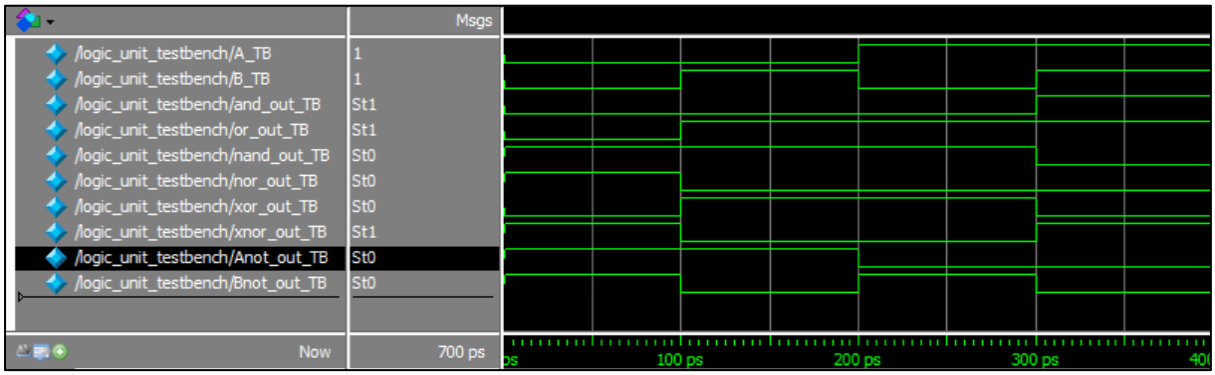
The schematic was made by examining the figure. The figure was very effective to visualize, code and implement the logic unit.

LOGIC UNIT TESTBENCH CODE:

```
module logic_unit_testbench();

reg A_TB, B_TB;

wire and_out_TB, or_out_TB, nand_out_TB, nor_out_TB, xor_out_TB, xnor_out_TB, Anot_out_TB, Bnot_out_TB;

logic_unit DUT(A_TB, B_TB, and_out_TB, or_out_TB, nand_out_TB, nor_out_TB, xor_out_TB, xnor_out_TB, Anot_out_TB, Bnot_out_TB);

initial

    begin
    
    A_TB = 1'b0; B_TB = 1'b0; #100;
    A_TB = 1'b0; B_TB = 1'b1; #100;
    A_TB = 1'b1; B_TB = 1'b0; #100;
    A_TB = 1'b1; B_TB = 1'b1; #100;
    
    end
endmodule
```

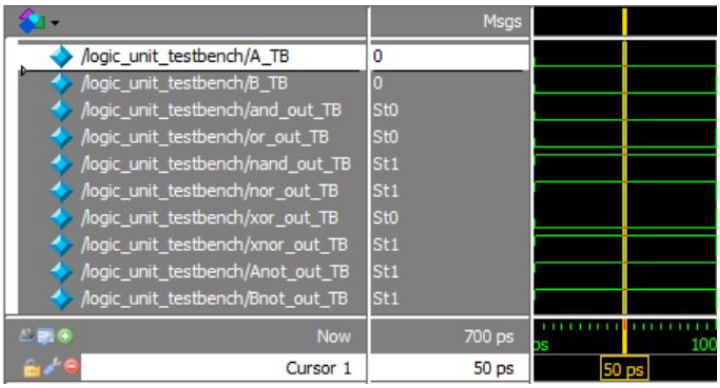
The testbench code was written as explained on page 6. Four rows were needed as each gate had four inputs.

MODELSIM SIMULATION OF LOGIC UNIT:



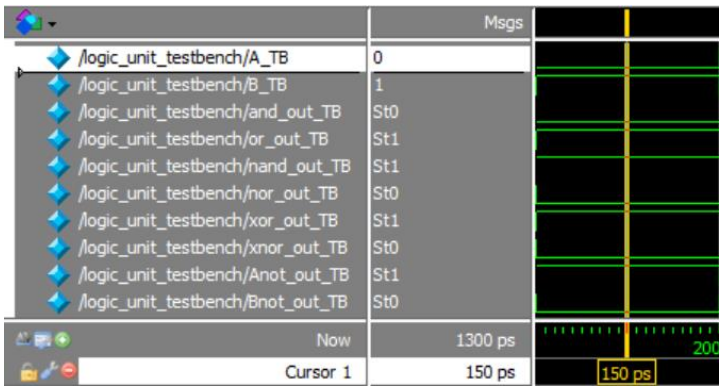
A	B	and	or	nand	nor	xor	xnor	A'	B'
0	0	0	0	1	1	0	1	1	1
0	1	0	1	1	0	1	0	1	0
1	0	0	1	1	0	1	0	0	1
1	1	1	1	0	0	0	1	0	0

I compiled the truth tables that we wrote to make it easier to compare to the logic unit simulation.



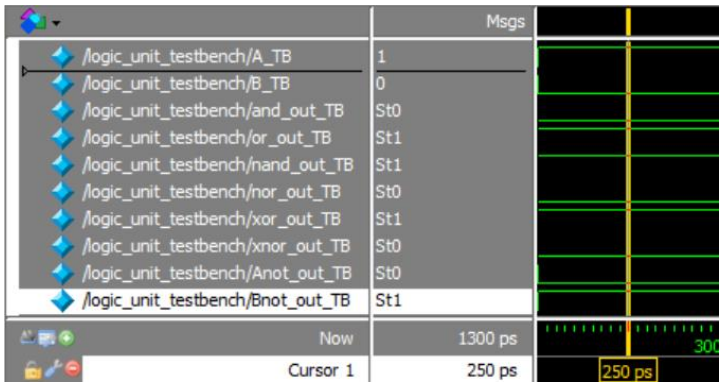
For the 0 to 100 PS: mark the inputs are A = 0 and B = 0. This is the same as the first row of the truth table. We can see that the outputs are the same as the truth table outputs.

and: 0, or: 0, nand: 1, nor: 1, xor: 0, xnor: 1, Anot: 1, Bnot: 1



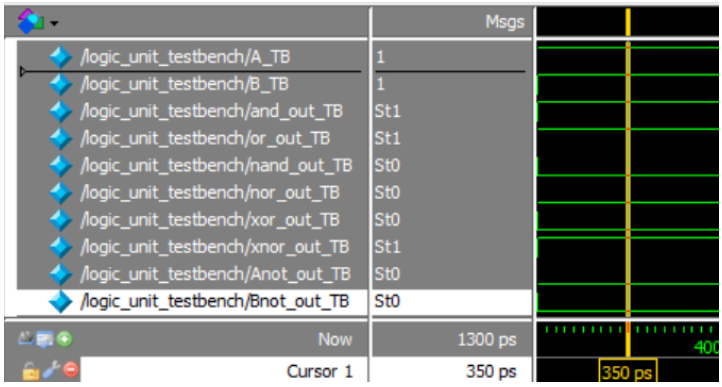
100 to 200 PS: mark is the same as the second row.

and: 0, or: 1, nand: 1, nor: 0, xor: 1, xnor: 0, Anot: 1, Bnot: 0



The 200 to 300 PS: is the same as the third row.

and: 0, or: 1, nand: 1, nor: 0, xor: 1, xnor: 0, Anot: 0, Bnot: 1



The 300 to 400 PS: is the same as the last row.

and: 1, or: 1, nand: 0, nor: 0, xor: 0, xnor: 1, Anot: 0, Bnot: 0

1.3 Preliminary Work: Bakery Profit Calculator

1.2.2 Bakery Profit Calculator

In this design, you will implement a simple digital circuit to compute total profits made by a bakery based on the kinds of deserts that they are baking. Bakery owner has to follow the set of guidelines below for setting up the bakery.

1. Baker can bake donuts, brownies, eclairs, and croissants on his bakery but cannot produce all at the same time.
2. The bakery does not have enough space to bake more than 2 different types of deserts. Therefore, his bakery never has more than 2 types of deserts at any particular time.
3. Each type of the desert earns a certain level of profit for the bakery. Donuts earn him a profit of 4 units, brownies earn him a profit of 2 units, eclairs earn a profit of 3 units and croissants being small livestock earn him a profit of 1 unit.
4. The total profit of the bakery is the sum of profits made by each type of desert that is baked with the exceptions provided below.
5. Brownies and croissants have very different ingredients, so Baker needs to spend extra for the sourcing of ingredients for each separately. Therefore, if he plans to bake brownies and croissants together then his profits go down by 1 unit.
6. Similarly, if he bakes donuts and croissants his profits go down by 1 unit because of extra sourcing costs. Additionally, donuts have a chance to require more labor than croissants on the bakery. This further reduces his profits by 1 unit.
7. However, croissants and eclairs do very well if baked together. They have many similarities especially with respect to the kind of ingredients they have. Also due to the huge demand for eclairs and croissants, the profit they make goes up by 1 unit if croissants and eclairs are baked together.

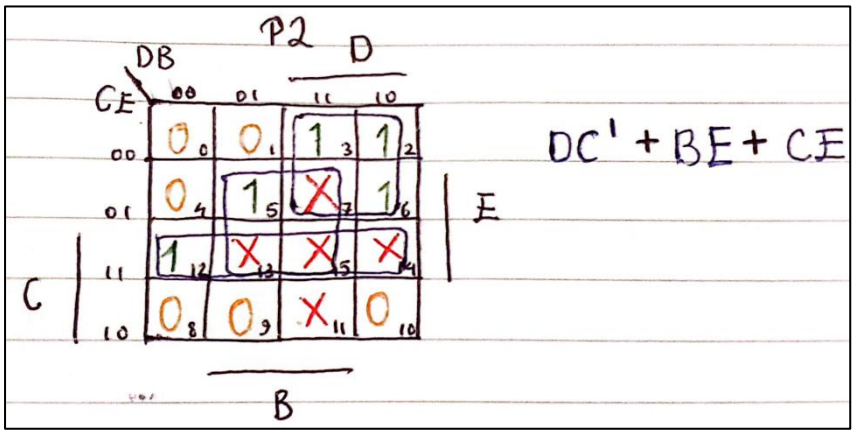
Your objective is to design a digital circuit which takes in the types of deserts on the bakery and outputs a binary number representing the corresponding profit. The four input bits are C (Croissants), E (Eclairs), D (Donuts), and B (Brownies), and the output bits are, P2, P1, P0, such that P2 is the most significant bit. Begin by creating a truth table containing all of the input combinations. Because there are four input bits, your table should contain 24 = 16 rows. Then fill in the output columns of your truth table based on the guidelines provided above. For outputs that are not possible due to restrictions on the bakery, insert 'X' to represent "don't cares." Once your truth table is complete, use Karnaugh Maps as discussed in class to simplify the Boolean algebra expressions for each of the output bits. To conclude the design, draw a schematic of the final digital circuit. Be sure to label all inputs and outputs.

C	E	D	B	P2	P1	P0	
0	0	0	0	0	0	0	=> 0
0	0	0	1	0	1	0	=> 2
0	0	1	0	1	0	0	=> 4
0	0	1	1	1	1	0	=> 6
0	1	0	0	0	1	1	=> 3
0	1	0	1	1	0	1	=> 5
0	1	1	0	1	1	1	=> 7
0	1	1	1	X	X	X	
1	0	0	0	0	0	1	=> 1
1	0	0	1	0	1	0	=> 2
1	0	1	0	0	1	1	=> 3
1	0	1	1	X	X	X	
1	1	0	0	1	0	1	=> 5
1	1	0	1	X	X	X	
1	1	1	0	X	X	X	
1	1	1	1	X	X	X	

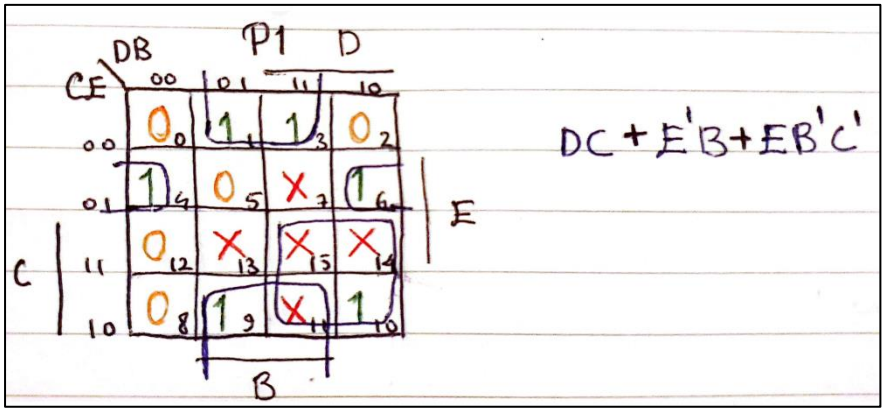
Based on the conditions given above, we each created a truth table. Then compared them to make sure it is correct. We put "don't care" conditions for when more than two inputs exist because the bakery is said to be unable to produce more than two types of pastries. For the rest we summed up the profits made for each pastry. In conditions where brownies and croissants were made, we took 1 unit of profit off. For donuts and croissants, we took 2 units of profit off. For croissants and eclairs, we added 1 unit of profit.

Later we drew KMAPs for each output.

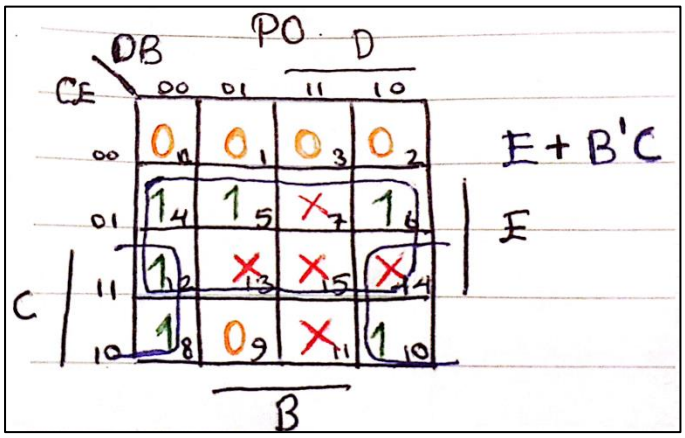
P2:



P1:



P0:



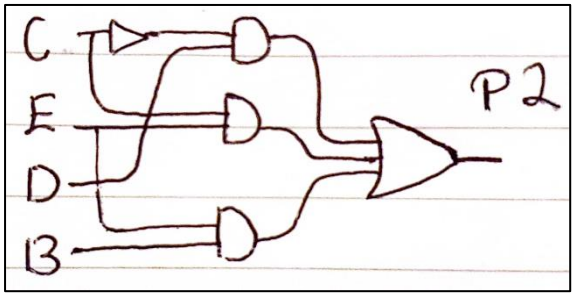
After concluding the outputs as:

- P2 = DC’ + BE + CE
- P1 = DC + E’B + EB’C’
- P0 = E + B’C

I drew schematics for each output so we can easily create the big Quartus schematic with all the outputs.

P2:

C prime and D connected to an and gate.



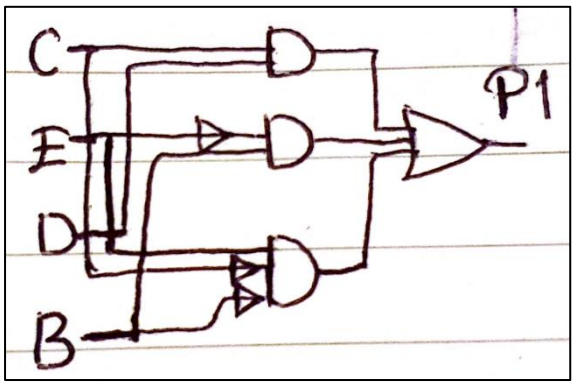
C and E connected to an and gate.

B and E connected to an and gate.

All of them are connected to an or gate

P1:

C and D connected to an and gate.

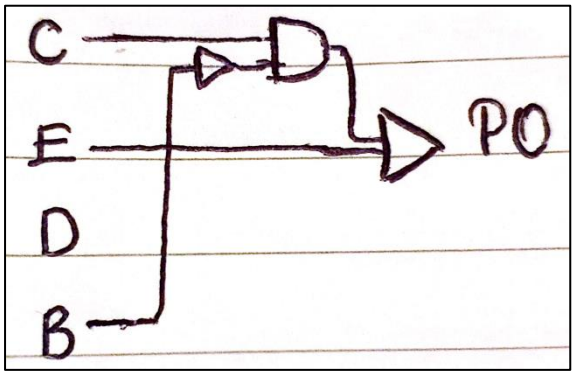


E prime and B connected to an and gate.

E, B prime and C prime is connected to an and gate.

All of them are connected to an or gate.

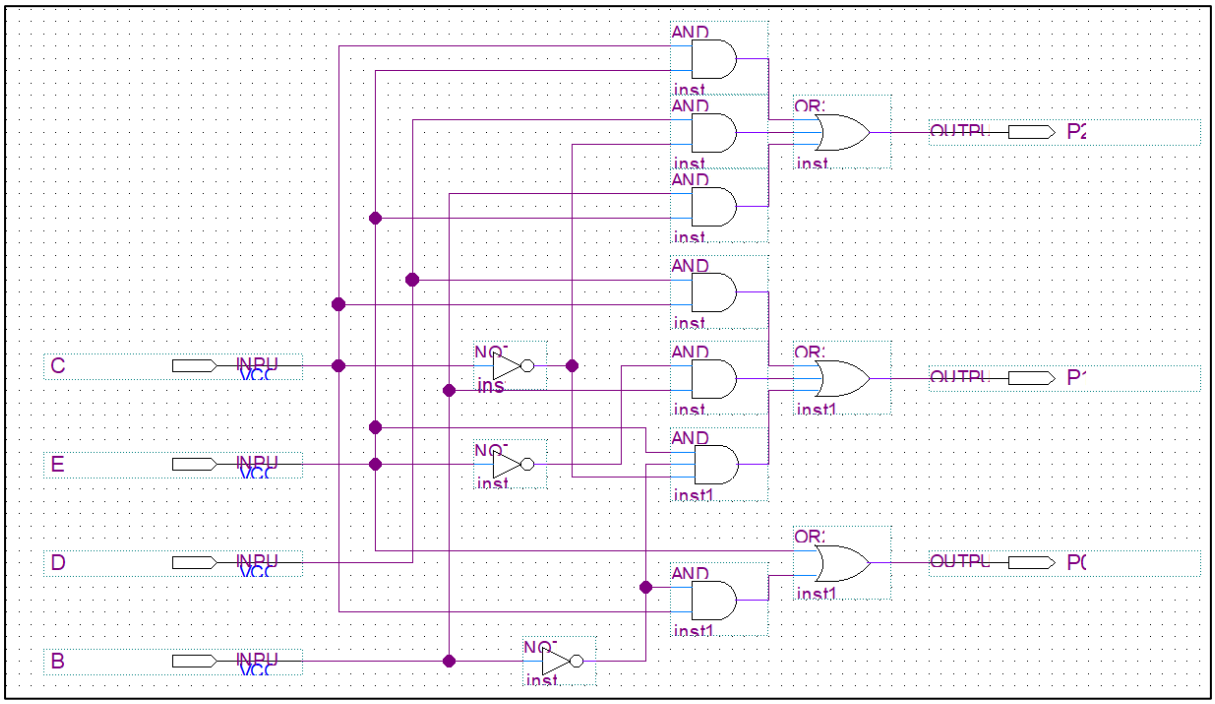
P0:



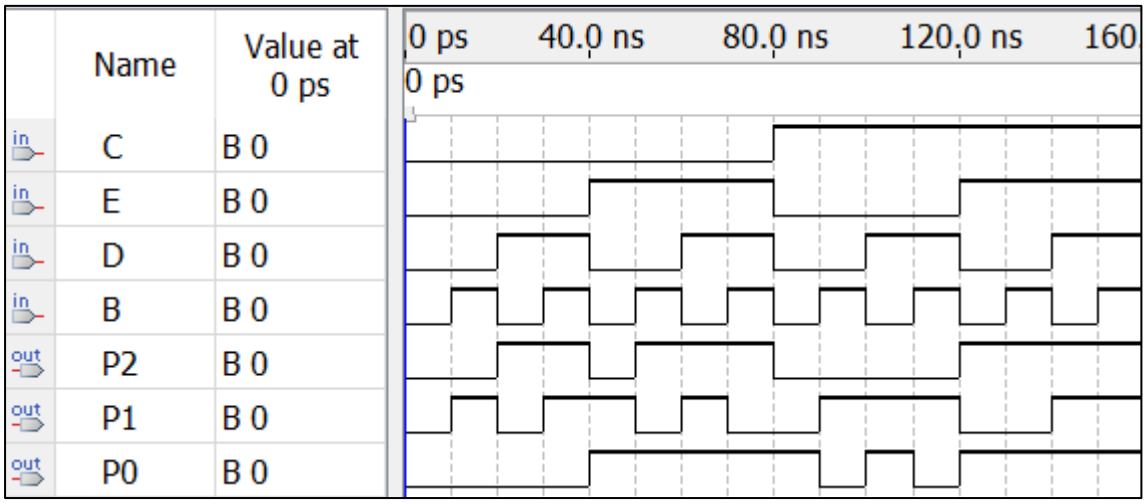
B prime and C is connected to an and gate.

And this is connected with E to an or gate.

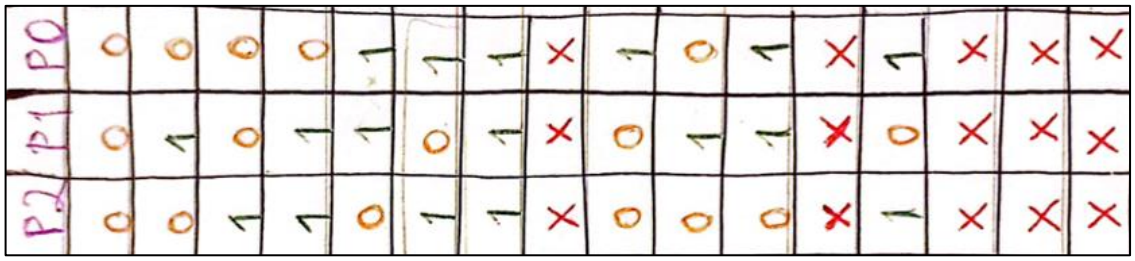
Drawing for each output made it easier to create the Bakery Profit Calculator schematic.



We also simulated the schematic to make sure the scheme was correct.



- **P0: 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1**
- **P1: 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1**
- **P2: 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1**



Except the "don't care" values, the outputs match therefore the schematic is confirmed to be correct.

Now, based on the schematic we wrote our structural bakery profit calculator Verilog code.

```
1
2   module bakery_profit_calculator(P2, P1, P0, C, E, D, B);
3
4   input C; //croissants
5   input E; //eclairs
6   input D; //donuts
7   input B; //brownies
8
9   output P2, P1, P0;
10  wire C_not, E_not, B_not;
11
12  not(C_not, C);
13  not(E_not, E);
14  not(B_not, B);
15
16  // P2:
17  wire w1, w2, w3;
18
19  and(w1, D, C_not);
20  and(w2, B, E);
21  and(w3, C, E);
22
23  or(P2, w1, w2, w3);
24
25  // P1:
26  wire w4, w5, w6;
27
28  and(w4, D, C);
29  and(w5, E_not, B);
30  and(w6, E, B_not, C_not);
31
32  or(P1, w4, w5, w6);
33
34  // P0:
35  wire w7;
36
37  and(w7, B_not, C);
38
39  or(P0, w7, E);
40
41
42  endmodule
```

However, we quickly realized that writing the code as structural meant that we couldn’t assign the “don’t care” values. So, we wrote a behavioral code to use instead. For a behavioral code, we assigned each bit for the output based on each input. Similarly to a testbench we wrote somewhat of a truth table for the inputs. The difference however was that we used conditional statements for each row. For example, for row one of P2, we can read it as: if C is 0, if E is 0, if D is 0 if B is 0 then P2 is 0.

Unfortunately, we didn’t know how to combine the statements so that we only had to write one condition with different outputs. So, we settled on created three assign blocks for each output.

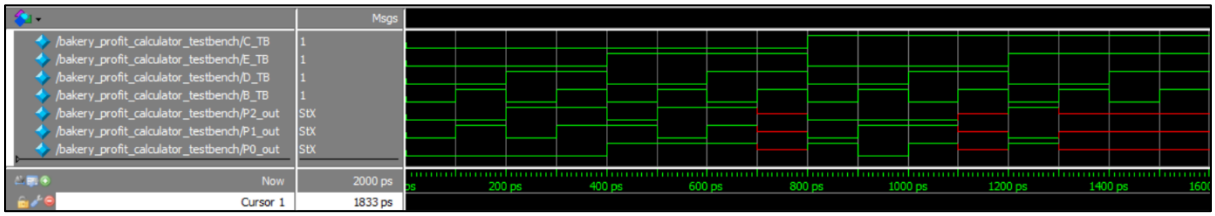
```
module bakery_profit_calculator_behavioral(P2, P1, P0, C, E, D, B);

input C; //croissants
input E; //eclairs
input D; //donuts
input B; //brownies

output P2, P1, P0;

assign P2 = ((C == 1'b0) && (E == 1'b0) && (D == 1'b0) && (B == 1'b0)) ? 1'b0:
            ((C == 1'b0) && (E == 1'b0) && (D == 1'b0) && (B == 1'b1)) ? 1'b0:
            ((C == 1'b0) && (E == 1'b0) && (D == 1'b1) && (B == 1'b0)) ? 1'b1:
            ((C == 1'b0) && (E == 1'b0) && (D == 1'b1) && (B == 1'b1)) ? 1'b1:
            ((C == 1'b0) && (E == 1'b1) && (D == 1'b0) && (B == 1'b0)) ? 1'b0:
            ((C == 1'b0) && (E == 1'b1) && (D == 1'b0) && (B == 1'b1)) ? 1'b1:
            ((C == 1'b0) && (E == 1'b1) && (D == 1'b1) && (B == 1'b0)) ? 1'b1:
            ((C == 1'b0) && (E == 1'b1) && (D == 1'b1) && (B == 1'b1)) ? 1'bx:
            ((C == 1'b1) && (E == 1'b0) && (D == 1'b0) && (B == 1'b0)) ? 1'b0:
            ((C == 1'b1) && (E == 1'b0) && (D == 1'b0) && (B == 1'b1)) ? 1'b0:
            ((C == 1'b1) && (E == 1'b0) && (D == 1'b1) && (B == 1'b0)) ? 1'b0:
            ((C == 1'b1) && (E == 1'b0) && (D == 1'b1) && (B == 1'b1)) ? 1'bx:
            ((C == 1'b1) && (E == 1'b1) && (D == 1'b0) && (B == 1'b0)) ? 1'b1:
            ((C == 1'b1) && (E == 1'b1) && (D == 1'b0) && (B == 1'b1)) ? 1'bx:
            ((C == 1'b1) && (E == 1'b1) && (D == 1'b1) && (B == 1'b0)) ? 1'bx:
            1'bx;

assign P1 = ((C == 1'b0) && (E == 1'b0) && (D == 1'b0) && (B == 1'b0)) ? 1'b0:
            ((C == 1'b0) && (E == 1'b0) && (D == 1'b0) && (B == 1'b1)) ? 1'b1:
            ((C == 1'b0) && (E == 1'b0) && (D == 1'b1) && (B == 1'b0)) ? 1'b0:
            ((C == 1'b0) && (E == 1'b0) && (D == 1'b1) && (B == 1'b1)) ? 1'b1:
            ((C == 1'b0) && (E == 1'b1) && (D == 1'b0) && (B == 1'b0)) ? 1'b1:
            ((C == 1'b0) && (E == 1'b1) && (D == 1'b0) && (B == 1'b1)) ? 1'b0:
            ((C == 1'b0) && (E == 1'b1) && (D == 1'b1) && (B == 1'b0)) ? 1'b1:
            ((C == 1'b0) && (E == 1'b1) && (D == 1'b1) && (B == 1'b1)) ? 1'bx:
            ((C == 1'b1) && (E == 1'b0) && (D == 1'b0) && (B == 1'b0)) ? 1'b0:
            ((C == 1'b1) && (E == 1'b0) && (D == 1'b0) && (B == 1'b1)) ? 1'b1:
            ((C == 1'b1) && (E == 1'b0) && (D == 1'b1) && (B == 1'b0)) ? 1'b1:
            ((C == 1'b1) && (E == 1'b0) && (D == 1'b1) && (B == 1'b1)) ? 1'bx:
            ((C == 1'b1) && (E == 1'b1) && (D == 1'b0) && (B == 1'b0)) ? 1'b0:
            ((C == 1'b1) && (E == 1'b1) && (D == 1'b0) && (B == 1'b1)) ? 1'bx:
            ((C == 1'b1) && (E == 1'b1) && (D == 1'b1) && (B == 1'b0)) ? 1'bx:
            1'bx;

assign P0 = ((C == 1'b0) && (E == 1'b0) && (D == 1'b0) && (B == 1'b0)) ? 1'b0:
            ((C == 1'b0) && (E == 1'b0) && (D == 1'b0) && (B == 1'b1)) ? 1'b0:
            ((C == 1'b0) && (E == 1'b0) && (D == 1'b1) && (B == 1'b0)) ? 1'b0:
            ((C == 1'b0) && (E == 1'b0) && (D == 1'b1) && (B == 1'b1)) ? 1'b0:
            ((C == 1'b0) && (E == 1'b1) && (D == 1'b0) && (B == 1'b0)) ? 1'b1:
            ((C == 1'b0) && (E == 1'b1) && (D == 1'b0) && (B == 1'b1)) ? 1'b1:
            ((C == 1'b0) && (E == 1'b1) && (D == 1'b1) && (B == 1'b0)) ? 1'b1:
            ((C == 1'b0) && (E == 1'b1) && (D == 1'b1) && (B == 1'b1)) ? 1'bx:
            ((C == 1'b1) && (E == 1'b0) && (D == 1'b0) && (B == 1'b0)) ? 1'b1:
            ((C == 1'b1) && (E == 1'b0) && (D == 1'b0) && (B == 1'b1)) ? 1'b0:
            ((C == 1'b1) && (E == 1'b0) && (D == 1'b1) && (B == 1'b0)) ? 1'b1:
            ((C == 1'b1) && (E == 1'b0) && (D == 1'b1) && (B == 1'b1)) ? 1'bx:
            ((C == 1'b1) && (E == 1'b1) && (D == 1'b0) && (B == 1'b0)) ? 1'b1:
            ((C == 1'b1) && (E == 1'b1) && (D == 1'b0) && (B == 1'b1)) ? 1'bx:
            ((C == 1'b1) && (E == 1'b1) && (D == 1'b1) && (B == 1'b0)) ? 1'bx:
            1'bx;

endmodule
```

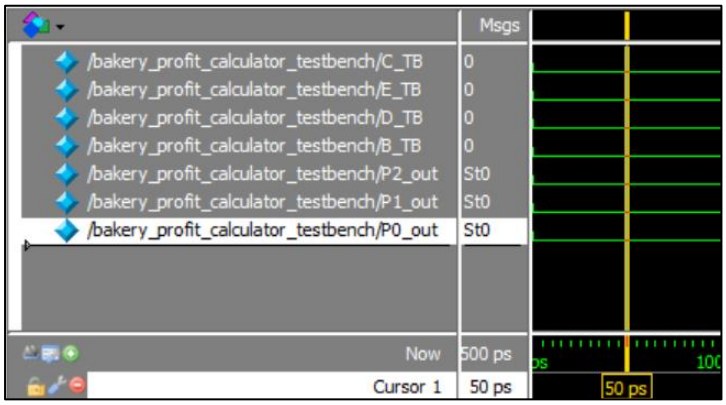
It was time to test the code with a simulation. Therefore, a testbench was created as explained before.

```
module bakery_profit_calculator_testbench();

reg C_TB, E_TB, D_TB, B_TB;

wire P2_out, P1_out, P0_out;

bakery_profit_calculator_behavioral DUT(P2_out, P1_out, P0_out, C_TB, E_TB, D_TB, B_TB);

initial

	begin
	
		C_TB = 1'b0; E_TB = 1'b0; D_TB = 1'b0; B_TB = 1'b0; #100;
		C_TB = 1'b0; E_TB = 1'b0; D_TB = 1'b0; B_TB = 1'b1; #100;
		C_TB = 1'b0; E_TB = 1'b0; D_TB = 1'b1; B_TB = 1'b0; #100;
		C_TB = 1'b0; E_TB = 1'b0; D_TB = 1'b1; B_TB = 1'b1; #100;
		C_TB = 1'b0; E_TB = 1'b1; D_TB = 1'b0; B_TB = 1'b0; #100;
		C_TB = 1'b0; E_TB = 1'b1; D_TB = 1'b0; B_TB = 1'b1; #100;
		C_TB = 1'b0; E_TB = 1'b1; D_TB = 1'b1; B_TB = 1'b0; #100;
		C_TB = 1'b0; E_TB = 1'b1; D_TB = 1'b1; B_TB = 1'b1; #100;
		C_TB = 1'b1; E_TB = 1'b0; D_TB = 1'b0; B_TB = 1'b0; #100;
		C_TB = 1'b1; E_TB = 1'b0; D_TB = 1'b0; B_TB = 1'b1; #100;
		C_TB = 1'b1; E_TB = 1'b0; D_TB = 1'b1; B_TB = 1'b0; #100;
		C_TB = 1'b1; E_TB = 1'b0; D_TB = 1'b1; B_TB = 1'b1; #100;
		C_TB = 1'b1; E_TB = 1'b1; D_TB = 1'b0; B_TB = 1'b0; #100;
		C_TB = 1'b1; E_TB = 1'b1; D_TB = 1'b0; B_TB = 1'b1; #100;
		C_TB = 1'b1; E_TB = 1'b1; D_TB = 1'b1; B_TB = 1'b0; #100;
		C_TB = 1'b1; E_TB = 1'b1; D_TB = 1'b1; B_TB = 1'b1; #100;
	
	end

endmodule
```

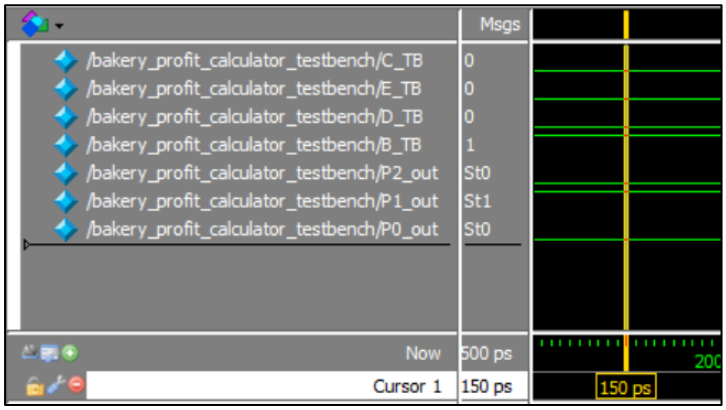
There were 16 rows because there are four inputs.

MODELSIM SIMULATION

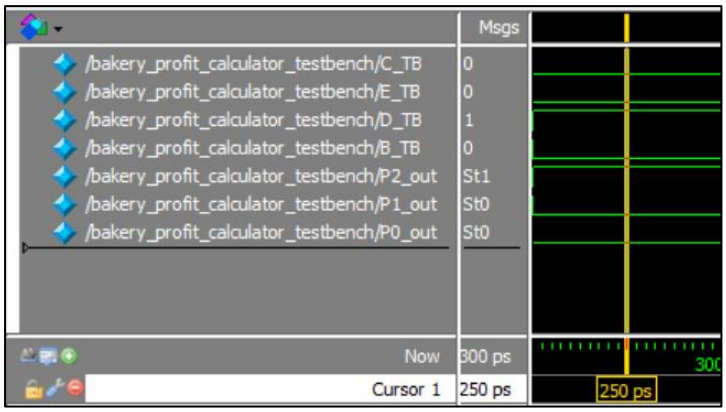




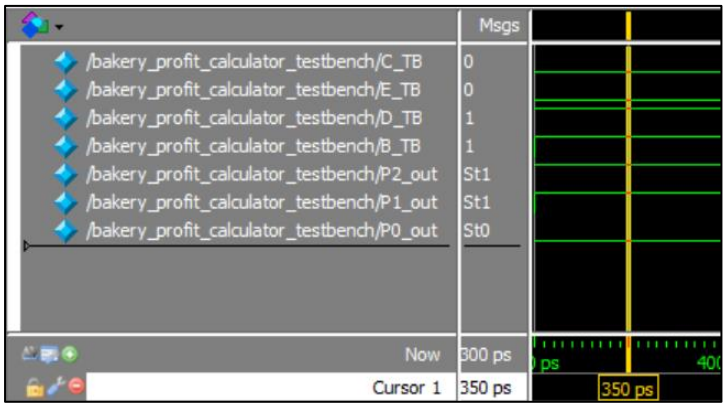
For between 0 – 100 PS:
The outputs are all 0.



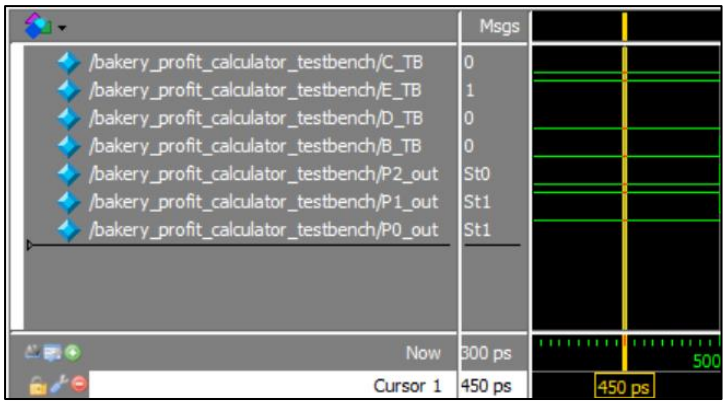
For between 100 – 200 PS: The outputs are 0, 1 and 0. So the profit made is 2 units.



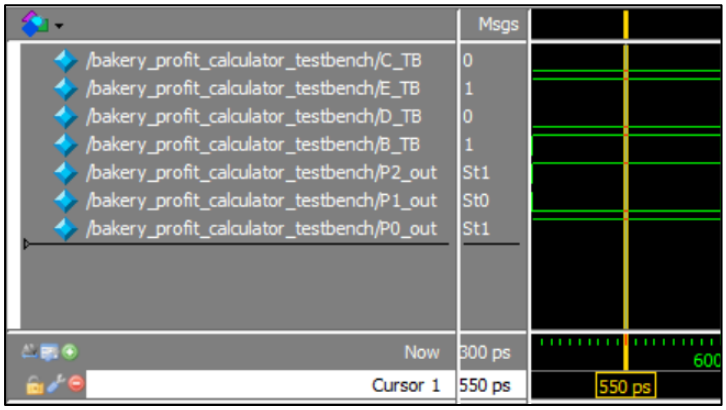
For between 200 – 300 PS: The outputs are 1, 0,0. So the profit made is 4 units.



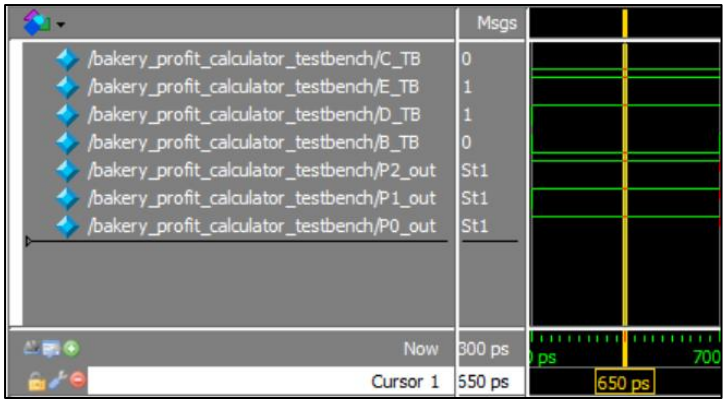
For between 300 – 400 PS: The outputs are 1, 1, 0. So the profit made is 6 units.



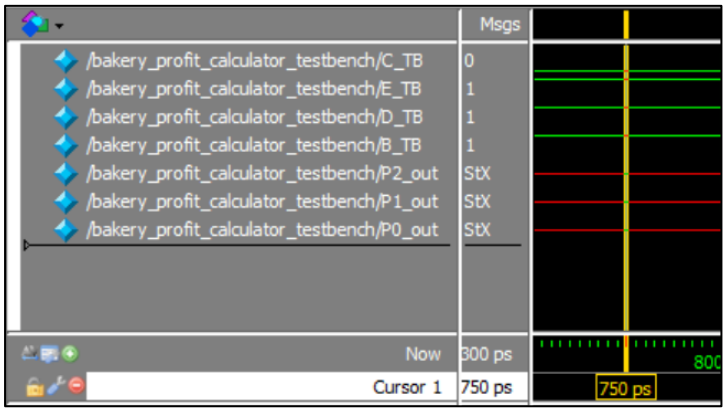
For between 400 – 500 PS: The outputs are 0, 1, 1. So the profit made is 3 units.



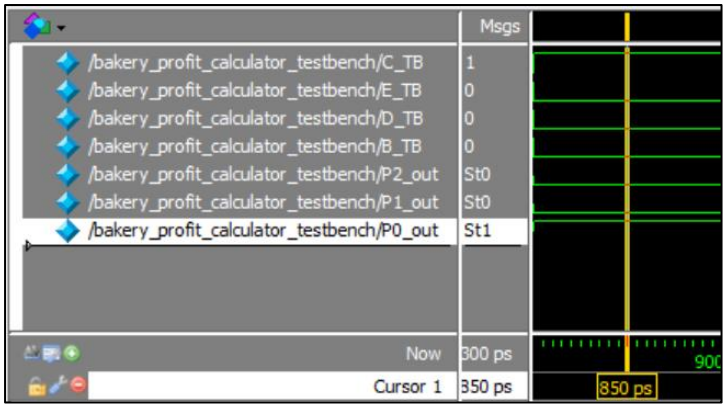
For between 500 – 600 PS: The outputs are 1, 0, 1. So the profit made is 5 units.



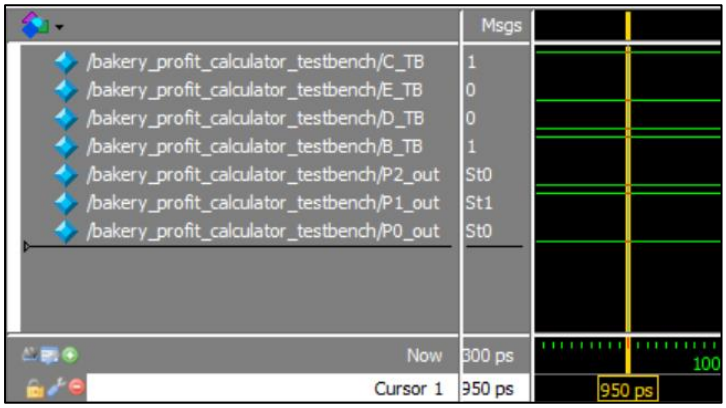
For between 600 – 700 PS: The outputs are 1, 1, 1. So the profit made is 7 units.



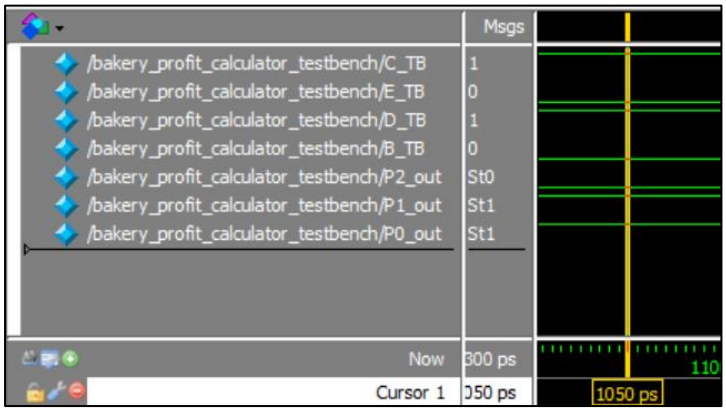
For between 700 – 800 PS: The outputs are X, X, X. This is because there are three inputs which is not possible.



For between 800 – 900 PS: The outputs are 0, 0, 1. So the profit made is 1 unit.

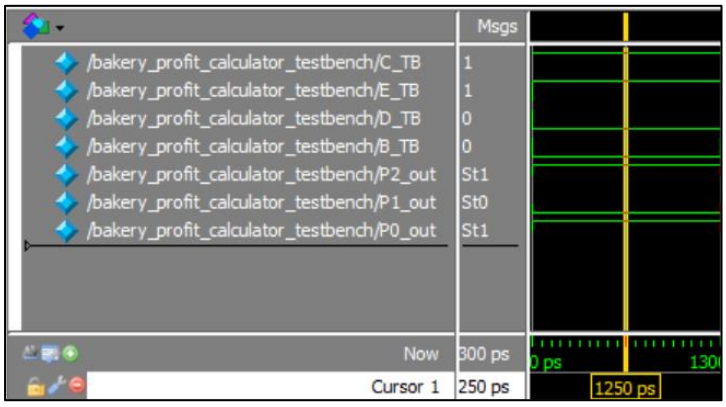


For between 900 – 1000 PS: The outputs are 0, 1, 0. So the profit made is 2 units.



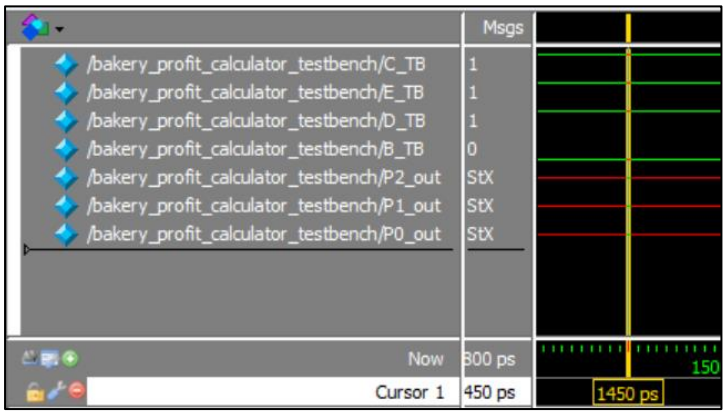
For between 1000 – 1100 PS: The outputs are 0, 1, 1. So the profit made is 3 units.

For between 1100 – 1200 PS: The outputs are X, X, X. This is because there are three inputs which is not possible.

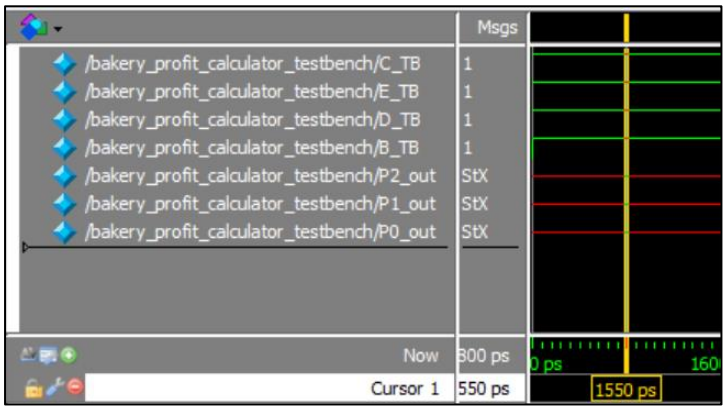


For between 1200 – 1300 PS: The outputs are 1, 0, 1. So the profit made is 5 units.

For between 1300 – 1400 PS: The outputs are X, X, X. This is because there are three inputs which is not possible.



For between 1400 – 1500 PS: The outputs are X, X, X. This is because there are three inputs which is not possible.



For between 1500 – 1600 PS: The outputs are X, X, X. This is because there are three inputs which is not possible.

The outputs are matching the truth table. The bakery profit calculator is working.

1.4 Conclusion:

We created a working Logic Unit and a Bakery Profit Calculator. The Logic Unit is outputting the outputs of the gates: AND, OR, NAND, NOR, XOR, XNOR and NOT. The Bakery profit calculator outputs the profit for each pastry combination.

First, we drew tables. Then created schematics. Wrote codes based on the tables and the schematics. Then we simulated the code we wrote. After tracing the simulation, we concluded that our work was correct.