ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 242E

DIGITAL CIRCUITS LABORATORY EXPERIMENT REPORT

EXPERIMENT NO	:	2
EXPERIMENT DATE	:	26.03.2021
LAB SESSION	:	FRIDAY - 10.30
GROUP NO	:	G17

GROUP MEMBERS:

150180024 : ŞULE BEYZA KARADAĞ

150190710 : SENİHA SERRA BOZKURT

150190024 : AHMET FURKAN KAVRAZ

SPRING 2021

Contents

1 INTRODUCTION [10 points]

We found the expression with the lowest cost for combinational logic circuits and implemented them. We used methods we learned in Digital Circuits Course (BLG 231E), like Karnaugh Map, Quine–McCluskey algorithm and Prime implicant chart creation.

2 MATERIALS AND METHODS [40 points]

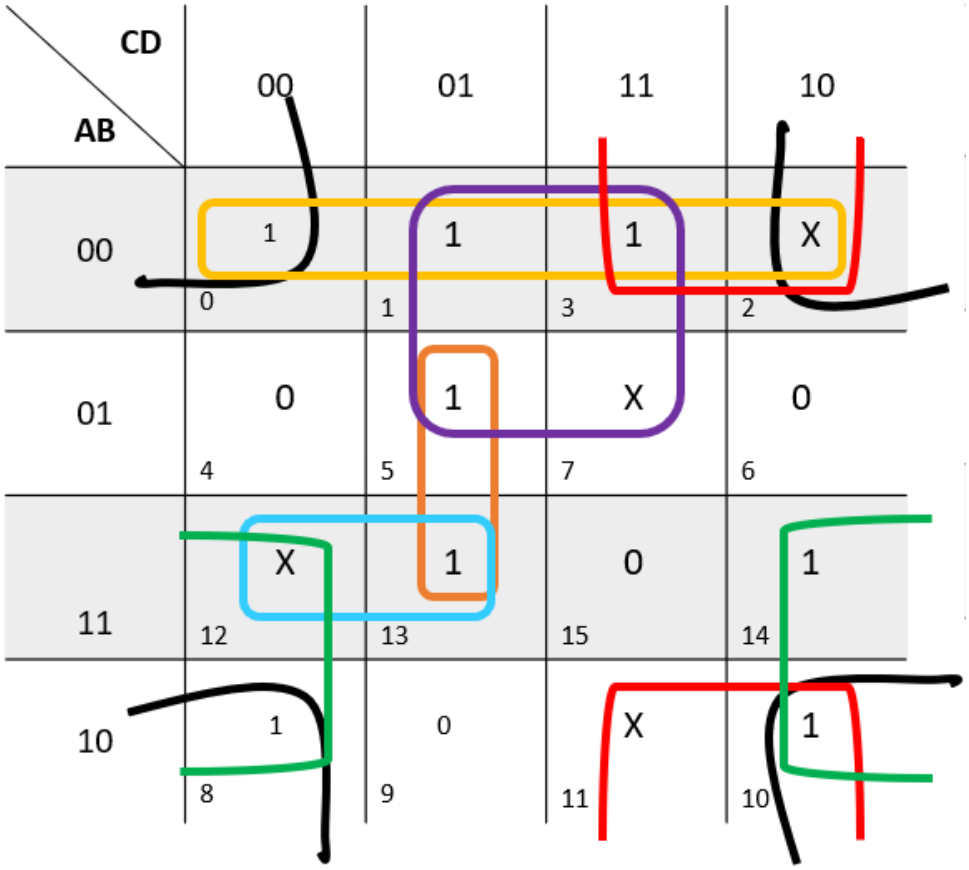
2.1 Preliminary

In the preliminary part, we found prime implicants of the function F_1 using different methods and determined the expression with the minimum cost, then we designed and drew the circuit of it with using logic gates and plexers.

2.1.1 Part 1-a

We implement the true points in the Karnaugh Map and we set all don't care values to 1 for the given function F_1 and obtained the following prime implicants:

B'.C + B.C'.D + A'.B' + A.D' + A.B.C' + A'.D + B'.D'



2.1.2 Part 1-b

In the Quine-McCluskey method, minterms are compared to all other minterms.

Num.	a	b	c	d	
0	0	0	0	0	√
1	0	0	0	1	√
2	0	0	1	0	√
8	1	0	0	0	√
3	0	0	1	1	√
5	0	1	0	1	√
10	1	0	1	0	√
12	1	1	0	0	√
7	0	1	1	1	√
11	1	0	1	1	√
13	1	1	0	1	√
14	1	1	1	0	√

We clustered the 1-generating input combinations depending on the number of 1's they have.

Then we compared the inputs that are in the neighboring clusters and grouped them by removing 1 variable that has different values for two inputs.

Num.	a	b	c	d	
0,1	0	0	0	-	√
0,2	0	0	-	0	√
0,8	-	0	0	0	√
1,3	0	0	-	1	√
1,5	0	-	0	1	√
2,3	0	0	1	-	√
2,10	-	0	1	0	√
8,10	1	0	-	0	√
8,12	1	-	0	0	√
3,7	0	-	1	1	√
3,11	-	0	1	1	√
5,7	0	1	-	1	√
5,13	-	1	0	1	
10,11	1	0	1	-	√
10,14	1	-	1	0	√
12,13	1	1	0	-	
12,14	1	1	-	0	√

Groups with 2 points We repeated all steps until there are no more groups can be formed. If two input combinations are grouped, they cannot be prime implicant alone.

Num.	a	b	c	d
0,1,2,3	0	0	-	-
0,2,1,3	0	0	-	-
0,2,8,10	-	0	-	0
0,8,2,10	-	0	-	0
1,3,5,7	0	-	-	1
1,5,3,7	0	-	-	1
2,3,10,11	-	0	1	-
2,10,3,11	-	0	1	-
8,10,12,14	1	-	-	0
8,12,10,14	1	-	-	0

Groups with 4 points There are no more group combinations we can create at this step and we got same input combinations, it is enough to write the prime implicants of the same combinations once.

The input combinations that are grouped are marked. Prime implicants are the combinations that are not grouped.

2.1.3 Part 1-c

The true points that are covered by a prime implicant are represented with an 'X' and placed to prime implicant's row according to their number index. Then we calculated the cost of each prime implicant according the cost criteria.

	0	1	3	5	8	10	13	14	COST
bc'd				X			X		7
abc'							X		7
a'b'	X	X	X						6
a'd		X	X	X					5
b'd'	X				X	X			6
b'c			X			X			5
ad'					X	X		X	5

In this chart, 14 is the distinguished point. ad' is an essential prime implicant, its column and row that it covers are removed.

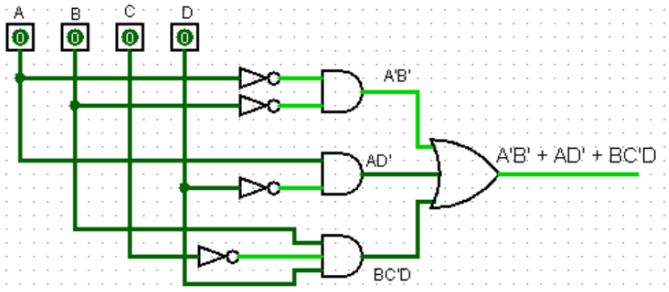
	0	1	3	5	13	COST
bc'd				X	X	7
abc'					X	7
a'b'	X	X	X			6
a'd		X	X	X		5
b'd'	X					6
b'c			X			5

$bc'd$ covers abc', $a'b'$ covers $b'd'$, $a'd$ covers $b'c$ and their costs are equal to each other. The covered rows are removed.

	0	1	3	5	13	COST
bc'd				X	X	7
a'b'	X	X	X			6
a'd		X	X	X		5

In this chart, 0 and 13 are distinguished points, therefore $a'b'$ and $bc'd$ are selected as an essential prime implicant. Their rows and columns that they cover are removed from the chart. Therefore, all true points of the F_1 are covered.

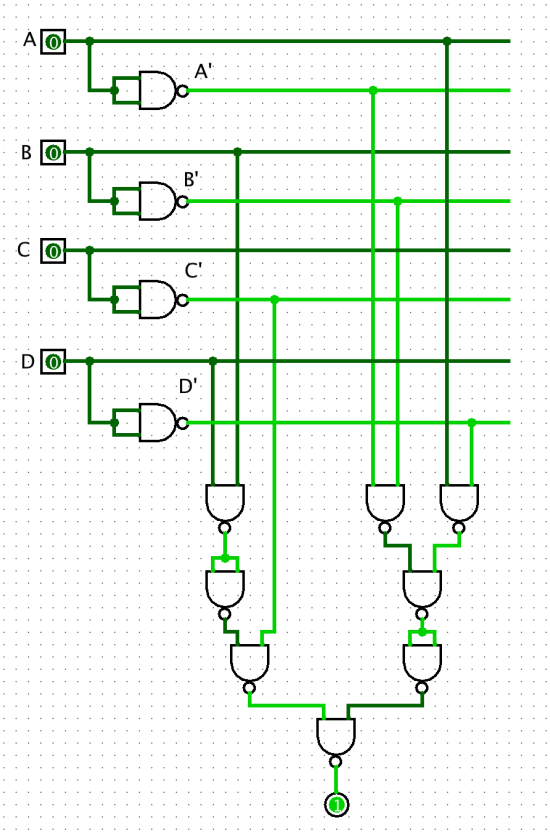
2.1.4 Part 1-d



1. We connected A to NOT gate to obtain A'.
2. We connected B to NOT gate to obtain B'.
3. We connected A' and B' to AND gate to obtain A'.B'.
4. We connected D to NOT gate to obtain D'.
5. We connected A and D' to AND gate to obtain A.D'.
6. We connected C to NOT gate to obtain C'.
7. We connected B, C' and D to AND gate to obtain B.C'.D.

8. We connected A'.B', A.D' and B.C'.D to OR gate to obtain A'B' + AD' + BC'D.

2.1.5 Part 1-e



1. We connected A and A to NAND gate to obtain A'.

2. We connected B and B to NAND gate to obtain B'.

3. We connected C and C to NAND gate to obtain C'.

4. We connected D and D to NAND gate to obtain D'.

For left subtree:

5. We connected B and D to NAND gate to obtain (B.D)'.

6. We connected (B.D)' and (B.D)' to NAND gate to obtain (B.D).

7. We connected (B.D) and C' to NAND gate to obtain (B'+C+D').

For right subtree:

8. We connected A' and B' to NAND gate to obtain (A'.B')'.

9. We connected A and D' to NAND gate to obtain (A.D')'.

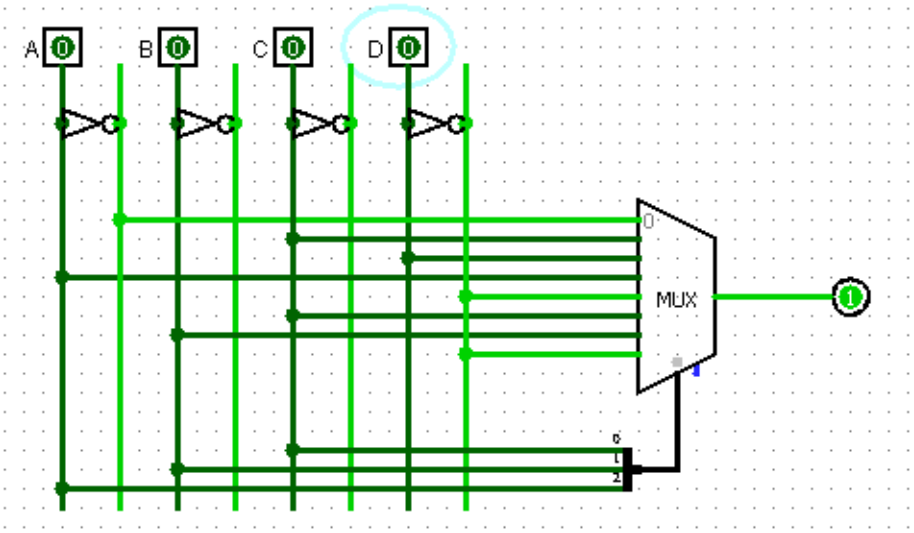
10. We connected (A'.B')' and (A.D')' to NAND gate to obtain ((A'.B')'.(A.D')')'.

11. We connected ((A'.B')'.(A.D')')' and ((A'.B')'.(A.D')')' to NAND gate to obtain (A'.B')'.(A.D')'.

For root:

12. We connected (C'.(B.D))' and (A'.B')'.(A.D')' to NAND gate to obtain ((C'.(B.D))'.(A'.B')'.(A.D')')'.

2.1.6 Part 1-f



We connected to:

0^{th} index of Multiplexer to A'.

1^{st} index of Multiplexer to C.

2^{nd} index of Multiplexer to D.

3^{rd} index of Multiplexer to A.

4^{th} index of Multiplexer to D'.

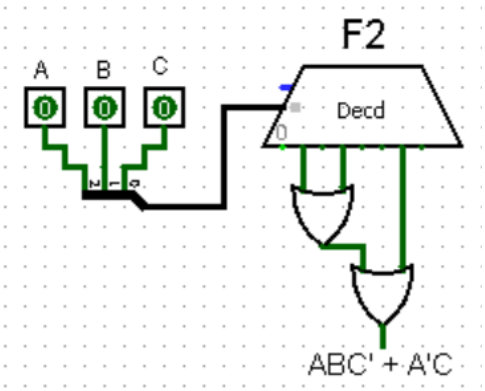
5^{th} index of Multiplexer to C.

6^{th} index of Multiplexer to B.

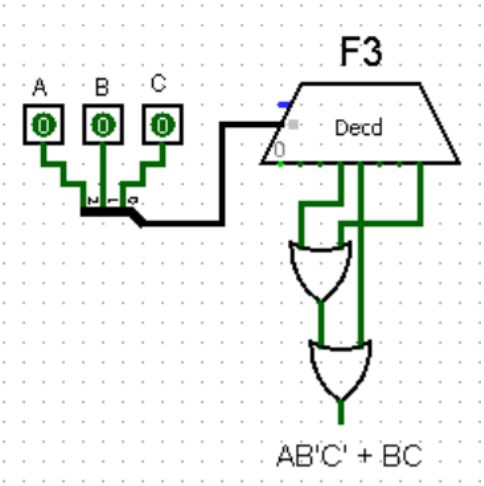
7^{th} index of Multiplexer to D'.

In the final, outputs of function are same with the truth table.

2.1.7 Part 2



1. We connected A, B and C to DECODER to obtain Inputs.
2. We connected A'B'C and A'BC to OR gate to obtain (A'B'C + A'BC) = A'C.
3. We connected A'C and ABC' to OR gate to obtain ABC' + A'C.



1. We connected A, B and C to DECODER to obtain Inputs.
2. We connected A'BC and ABC to OR gate to obtain (A'BC + ABC) = BC.
3. We connected BC and AB'C' to OR gate to obtain BC + AB'C'.

2.2 Experiment

2.2.1 Part 1

```
module and_gate_2input(
    input A,
    input B,
    output C
    );
    assign C = A & B;
endmodule
```

```
module and_gate_3input(
    input A,
    input B,
    input C,
    output D
    );
    assign D = A & B & C;
endmodule
```

AND gates are created by assigning the result of applied "AND" operation between inputs to output. As seen above, one of them gets 2 inputs and the other gets 3 inputs.

```
module or_gate_3input(
    input A,
    input B,
    input C,
    output D
    );
    assign D = A | B | C;
endmodule
```

OR gate is created by assigning the result of applied "OR" operation between inputs to output.

```
module not_gate(
    input A,
    output B
    );
    assign B = ~A;
endmodule
```

NOT gate is created by assigning result of applying "NOT" operation of input to output.

```
module pre1_d(
    input A,
    input B,
    input C,
    input D,
    output E
    );
    wire var1, var2, var3;
    wire not_A, not_B, not_C, notD;

    not_gate not0(A, not_A);
    not_gate not1(B, not_B);
    not_gate not2(C, not_C);
    not_gate not3(D, not_D);

    and_gate_2input and1(not_A, not_B, var1);
    and_gate_2input and2(A    , not_D, var2);
    and_gate_3input and3(B, not_C, D,  var3);

    or_gate_3input or1(var1, var2, var3, E);
endmodule
```

- First, we applied NOT operation to all of the inputs. Because as seen in the **Preliminary-1.d part**, not operation of all inputs separately are needed. So we obtained A', B', C', D'.
- Then, we applied AND operation to A' and B' to obtain the expression A'.B'
- Then, we applied AND operation to A and D' to obtain the expression A.D'
- Then, we applied AND operation to B, C' and D to obtain the expression B.C'.D

- In final, we applied OR operation to A'.B', A.D' and B.C'.D to obtain the expression A'.B' + A.D'+ B.C'.D

2.2.2 Part 2

In order to create NAND gate:

1. We connected A and B to AND gate. We obtained A.B and assigned it to not_C variable.

2. We connected not_C to NOT gate and assigned it to output C. Therefore we got NAND operation.

```
module nand_gate(
    input A,
    input B,
    output C
    );
    wire not_C;

    and_gate_2input and0(A, B, not_C);
    not_gate not0(not_C, C);

endmodule
```

```
module pre1_e(
    input A,
    input B,
    input C,
    input D,
    output E
    );
    wire var1, var2, var3, var4, var5, var6, var7;
    wire not_A, not_B, not_C, notD;

    nand_gate nand0(A, A, not_A);
    nand_gate nand1(B, B, not_B);
    nand_gate nand2(C, C, not_C);
    nand_gate nand3(D, D, not_D);

    //left
    nand_gate nand4(D, B, var1);
    nand_gate nand5(var1, var1, var2);
    nand_gate nand6(var2, not_C, var3);

    //right
    nand_gate nand7(not_A, not_B, var4);
    nand_gate nand8(A, not_D, var5);
    nand_gate nand9(var4, var5, var6);
    nand_gate nand10(var6, var6, var7);
    //root
    nand_gate nand11(var7, var3, E);
endmodule
```

To obtain the expression:

1. First, we need NOT operation of all inputs separately. As seen in the **Preliminary-1.e part**, we connected A, B, C and D with themselves to a NAND gate.

2. Then, in order to obtain left part of the circuit; we connected:

- D and B with NAND gate, and assigned it to var1.
- var1 with itself with NAND gate to obtain NOT operation, and assigned it to var2.
- var2 and not_C with NAND gate assigned to var3.

3. In order to obtain right part of the circuit; we connected:

- not_A and not_B with NAND gate, and assigned it to var4.
- A and not_D with NAND gate, and assigned it to var5.
- var4 and var5 with NAND gate, and assigned it to var6.
- var6 with itself with NAND gate to obtain NOT operation, and assigned it to var7.

4. Finally, we connected the left part with the right part. In order to do that, we used NAND gate to connect var3 and var7.

2.2.3 Part 3

```
module and_gate_4input(
    input A,
    input B,
    input C,
    input D,
    output E
    );
    assign E = A & B & C & D;
endmodule
```

→ Here, AND gate is are created by assigning the result of applied "AND" operation between 4 inputs to output. As seen below, we later will use it in multiplexer design.

NOT gate is created by assigning result of applying "NOT" operation of input to output.

```
module not_gate(
    input A,
    output B
    );
    assign B = ~A;
endmodule
```

```
module or_gate_8input(
    input A,
    input B,
    input C,
    input D,
    input E,
    input F,
    input G,
    input H,
    output I
    );
    assign I = A | B | C | D | E | F | G | H;
endmodule
```

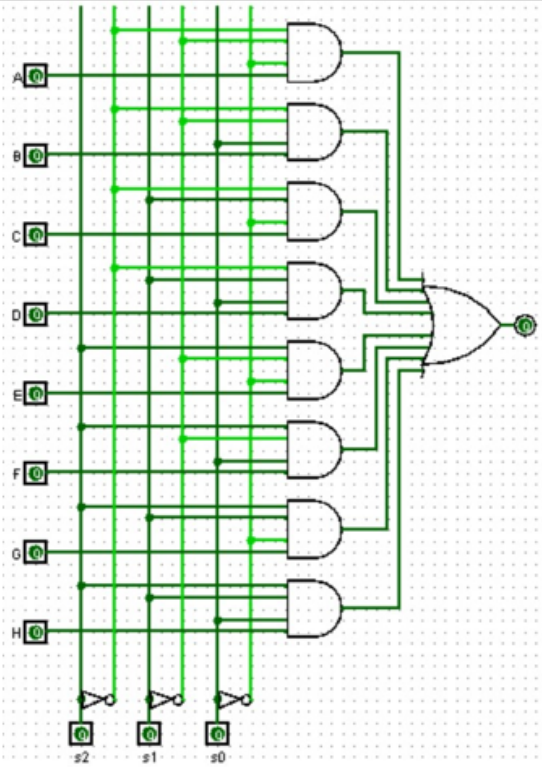
And here, we implemented OR gate with 8 inputs, to use in the design of 8:1 multiplexer with 3 selectors.

Then, 8:1 Multiplexer is created with using 4-input AND gates and NOT gates. You can see it in the page above. And the explanation is:

We used 3 selectors for the multiplexer to choose. Not operation of the selectors are needed for the internal structure of the multiplexer. In order to obtain those, we applied NOT operation to all selectors.

We connected them with appropriate input combinations via AND gates to get the parts in our expression. Finally, we connected what we achieved with OR gate. The reason we use AND gates is to reset combinations that will not participate in our OR gate, because 0 is dominant in the AND operation.

```
module multiplexer_8input(
    input A,
    input B,
    input C,
    input D,
    input E,
    input F,
    input G,
    input H,
    input s0, //selector 1
    input s1, //selector 2
    input s2, //selector 3
    output O
    );
    wire not_s0, not_s1, not_s2;
    wire OA, OB, OC, OD, OE, OF, OG, OH;

    not_gate not0(s0, not_s0);
    not_gate not1(s1, not_s1);
    not_gate not2(s2, not_s2);
```




```
    and_gate_4input o0(not_s0, not_s1, not_s2, A, OA); // 000
    and_gate_4input o1(s0    , not_s1, not_s2, B, OB); // 001
    and_gate_4input o2(not_s0, s1    , not_s2, C, OC); // 010
    and_gate_4input o3(s0    , s1    , not_s2, D, OD); // 011
    and_gate_4input o4(not_s0, not_s1, s2    , E, OE); // 100
    and_gate_4input o5(s0    , not_s1, s2    , F, OF); // 101
    and_gate_4input o6(not_s0, s1    , s2    , G, OG); // 110
    and_gate_4input o7(s0    , s1    , s2    , H, OH); // 111

    or_gate_8input or0(OA, OB, OC, OD, OE, OF, OG, OH, O);
```

	a	b	c	d	F
0	0	0	0	0	**1**
1	0	0	0	1	**1**
2	0	0	1	0	**X**
3	0	0	1	1	**1**
4	0	1	0	0	**0**
5	0	1	0	1	**1**
6	0	1	1	0	**0**
7	0	1	1	1	**X**
8	1	0	0	0	**1**
9	1	0	0	1	**0**
10	1	0	1	0	**1**
11	1	0	1	1	**X**
12	1	1	0	0	**X**
13	1	1	0	1	**1**
14	1	1	1	0	**1**
15	1	1	1	1	**0**

As you can see from the truth table next column, we don't need all of the input combinations to obtain this Boolean expression (A'.B' + A.D'+ B.C'.D).

- The don't care values are the outputs with numbers 2 (A'.B'.C.D'), 7 (A'.B.C.D), 11 (A.B'.C.D), 12 (A.B.C'.D').
- The 0 values are the outputs with numbers 4 (A'.B.C'.D'), 6 (A'.B.C.D'), 9 (A.B'.C'.D), 15 (A.B.C.D).

So the remaining expressions are:

- 0 (A'.B'.C'.D'), - 3 (A'.B'.C.D), - 8 (A.B'.C'.D'), - 13 (A.B.C'.D),
- 1 (A'.B'.C'.D), - 5 (A'.B.C'.D), - 10 (A.B'.C.D'), - 14 (A.B.C.D').

By using these 1 outputs, and the don't care values we would like, we connected the inputs to 8:1 MUX and then get the required outputs for us. The following code block represents the described approach:

We get the don't care valued input 2 as 1, here, so inputs 0 + 2
= A'.B'.C'.D' + A'.B'.C.D' → By Distributivity:
= A'.B'.(C' + C).D' → By Inversion:
= A'.B'.1.D'→ By Identity:
= A'.B'.D'.....(X)

Inputs 3 + 1 = A'.B'.C'.D + A'.B'.C.D → By Distributivity:
= A'.B'.(C'+ C).D → By Inversion:
= A'.B'.1.D → By Identity:
= A'.B'.D.....(Y)

Concatenating them both: X + Y
= A'.B'.D + A'.B'.D' → By Distributivity:
= A'.B'.(D + D') → By Inversion:
= A'.B'.1 → By Identity = A'B'

Inputs 5 + 13 = A'.B.C'.D + A.B.C'.D → By Distributivity:
= (A' + A).B.C'.D → By Inversion:
= 1.B.C'.D → By Identity = B.C'.D

Get the don't care value 12 as 1, here, so inputs 12 + 14
= A.B.C'.D' + A.B.C.D' → By Distributivity:
= A.B.(C' + C).D' → By Inversion:
= A.B.1.D' → By Identity:
= A.B.D'.....(W)

Inputs 8 + 10 = A.B'.C'.D' + A.B'.C.D' → By Distributivity:
= A.B'.(C'+C).D' → By Inversion:
= A.B'.1.D' → By Identity:
= A.B'.D'.....(Z)

Concatenating them both: W + Z
= A.B.D' + A.B'.D' → By Distributivity:
= A.(B + B').D' → By Inversion:
= A.1.D' → By Identity = A.D'
So the expression obtained: A'.B' + A.D'+ B.C'.D

```
module pre1_f(
    input A,
    input B,
    input C,
    input D,
    output E
    );
    wire not_A, not_D;

    not_gate not1(A, not_A);
    not_gate not2(D, not_D);

    multiplexer_8input mux(not_A, C, D, A, not_D, C, B, not_D, C, B, A, E);

endmodule
```

2.2.4 Part 4

```
module or_gate_2input(
    input A,
    input B,
    output C
    );
    assign C = A | B;
endmodule
```

OR gate is created by assigning the result of applied "OR" operation between inputs to output.

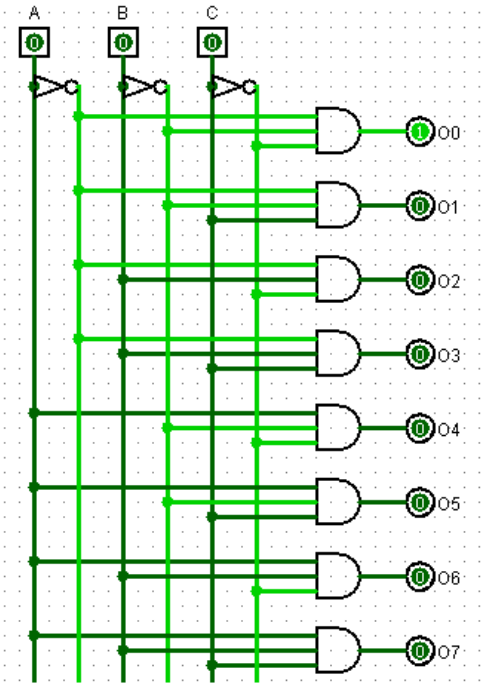
```
module not_gate(
    input A,
    output B
    );
    assign B = ~A;
endmodule
```

NOT gate is created by assigning result of applying "NOT" operation of input to output.

```
module and_gate_3input(
    input A,
    input B,
    input C,
    output D
    );
    assign D = A & B & C;
endmodule
```

AND gate is created by assigning the result of applied "AND" operation between inputs to output. As seen above, one of them gets 2 inputs and the other gets 3 inputs.

```
module decoder_3input(
    input s0,
    input s1,
    input s2,
    output O0,
    output O1,
    output O2,
    output O3,
    output O4,
    output O5,
    output O6,
    output O7
    );
    wire not_s0, not_s1, not_s2;

    not_gate not0(s0, not_s0);
    not_gate not1(s1, not_s1);
    not_gate not2(s2, not_s2);

    and_gate_3input and0(not_s0, not_s1, not_s2, O0);
    and_gate_3input and1(not_s0, not_s1, s2    , O1);
    and_gate_3input and2(not_s0, s1    , not_s2, O2);
    and_gate_3input and3(not_s0, s1    , s2    , O3);
    and_gate_3input and4(s0    , not_s1, not_s2, O4);
    and_gate_3input and5(s0    , not_s1, s2    , O5);
    and_gate_3input and6(s0    , s1    , not_s2, O6);
    and_gate_3input and7(s0    , s1    , s2    , O7);
endmodule
```



3:8 Decoder 3 control inputs and 8 outputs. According to the values of selectors, only one output gets the value 1, others will be 0.

Decoder is created with using 3-input AND gates and NOT gates. Not operation of the selectors are needed again for the internal structure. In order to obtain those, we applied NOT operation to all selectors.

We connected them with appropriate input combinations via AND gates. Finally, we obtained each output. The reason we use AND gates and NOT gates is to reset all outputs except 1 output.

```
module F2(
    input A,
    input B,
    input C,
    output D
    );
    wire var0, var1, var2, var3, var4, var5, var6, var7;
    wire or_output;
    decoder_3input decoder(A, B, C, var0, var1, var2, var3, var4, var5, var6, var7);

    or_gate_2input or1(var1, var3, or_output);
    or_gate_2input or2(or_output, var6, D);
endmodule
```

1. We connected A, B and C to DECODER to obtain required expressions.

2. We connected 1st output of decoder (A'B'C) and 3rd output of decoder (A'BC) to OR gate to obtain or_output = A'C.

3. We connected or_output and 6th output of decoder (ABC') to OR gate to obtain output D = ABC' + A'C.

```
module F3(
    input A,
    input B,
    input C,
    output D
    );
    wire var0, var1, var2, var3, var4, var5, var6, var7;
    wire or_output;
    decoder_3input decoder(A, B, C, var0, var1, var2, var3, var4, var5, var6, var7);

    or_gate_2input or1(var3, var7, or_output);
    or_gate_2input or2(or_output, var4, D);
endmodule
```

1. We connected A, B and C to DECODER to obtain required expressions.

2. We connected 3rd output of decoder (A'BC) and 7th output of decoder (ABC) to OR gate to obtain or_output = BC.

3. We connected or_output and 4th output of decoder(AB'C') to OR gate to obtain output D = BC + AB'C'.

3 RESULTS [15 points]

The simulation results we obtained after completing each step of the experiments resulted the same with their truth tables drawn above. This concludes the fact that our experiment ended up being coherant.

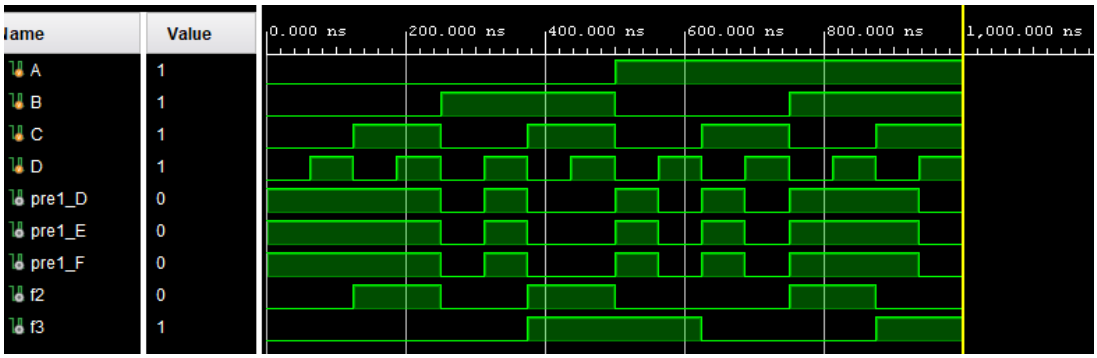
```
`timescale 1ns / 1ps

module test();

    reg A;
    reg B;
    reg C;
    reg D;
    wire prel_D;
    wire prel_E;
    wire prel_F;
    wire f2;
    wire f3;

    prel_d uut1 (A, B, C, D, prel_D);
    prel_e uut2 (A, B, C, D, prel_E);
    prel_f uut3 (A, B, C, D, prel_F);
    F2     uut4 (A, B, C,    f2    );
    F3     uut5 (A, B, C,    f3    );


    initial begin
        A = 0;   B = 0; C = 0;  D = 0;  #62.5;
        A = 0;   B = 0; C = 0;  D = 1;  #62.5;
        A = 0;   B = 0; C = 1;  D = 0;  #62.5;
        A = 0;   B = 0; C = 1;  D = 1;  #62.5;
        A = 0;   B = 1; C = 0;  D = 0;  #62.5;
        A = 0;   B = 1; C = 0;  D = 1;  #62.5;
        A = 0;   B = 1; C = 1;  D = 0;  #62.5;
        A = 0;   B = 1; C = 1;  D = 1;  #62.5;
        A = 1;   B = 0; C = 0;  D = 0;  #62.5;
        A = 1;   B = 0; C = 0;  D = 1;  #62.5;
        A = 1;   B = 0; C = 1;  D = 0;  #62.5;
        A = 1;   B = 0; C = 1;  D = 1;  #62.5;
        A = 1;   B = 1; C = 0;  D = 0;  #62.5;
        A = 1;   B = 1; C = 0;  D = 1;  #62.5;
        A = 1;   B = 1; C = 1;  D = 0;  #62.5;
        A = 1;   B = 1; C = 1;  D = 1;  #62.5;
    end
endmodule
```

	a	b	c	d	F
0	0	0	0	0	**1**
1	0	0	0	1	**1**
2	0	0	1	0	**X**
3	0	0	1	1	**1**
4	0	1	0	0	**0**
5	0	1	0	1	**1**
6	0	1	1	0	**0**
7	0	1	1	1	**X**
8	1	0	0	0	**1**
9	1	0	0	1	**0**
10	1	0	1	0	**1**
11	1	0	1	1	**X**
12	1	1	0	0	**X**
13	1	1	0	1	**1**
14	1	1	1	0	**1**
15	1	1	1	1	**0**

You can see the simulation inputs and their given times to complete each step of their part of the simulation next to this column.

To carry out each step of the experiment, we commented unused inputs and used the rest of the inputs, and arranged the time slots of each, accordingly.

- First, for **Part 1-d**, we used regs A, B, C and D to implement pre1_d.
- After that, in **Part 1-e** we formed the function pre1_e using regs A, B, C and D.
- Then, in **Part 1-f**, we implemented the pre1_f function using all regs.
- Finally, in **Part 2**, we used regs A, B and C and simulated the functions F2 and F3.



4 DISCUSSION [25 points]

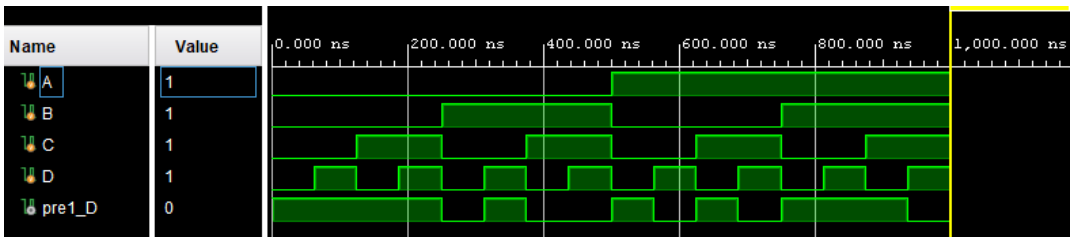
In the preliminary **part-1a** and **part-1b**, firstly, we find prime implicants using Karnaugh and Quine-McCluskey methods.

Later on this, in **preliminary part-1c**, the expression with minimum cost is found according to given cost criteria.

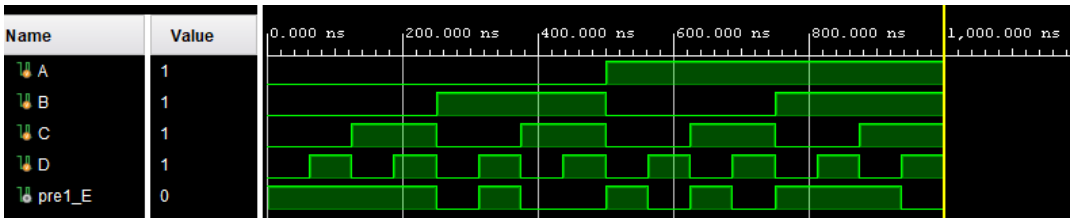
After that, in **preliminary part-1d**, **part-1e** and **part-1f**, the lowest cost expression is designed and has been drawn according to permitted gates and/or plexers.

Next, in **preliminary part-2**, the functions F_2 = abc' + a'c and F_3 = ab'c' + bc are drawn using only one 3:8 DECODER and 2-input OR gates.

We then implemented the circuit that we have designed in **Preliminary 1.d**. section using NOT, AND, and OR modules (which we designed beforehand). Following the implementation, we ran simulations for various input combinations to validate our design. It turns out to be coherent.

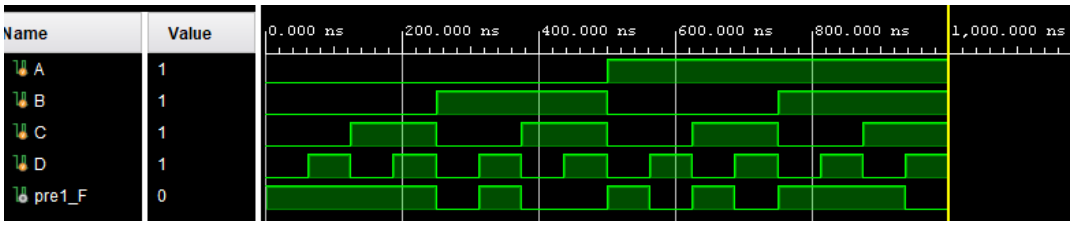


After that, we implemented the circuit that we have designed in **Preliminary 1.e**. section using only NAND modules (which we designed beforehand). Following the implementation, we ran simulations for various input combinations to validate our design.

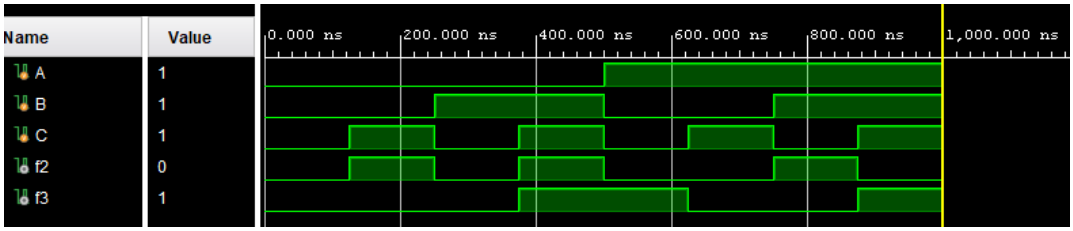


We implemented the circuit that we have designed in **Preliminary 1.f**. section using

a 8:1 MUX and NOT gates. We wrote the multiplexer and NOT gate as modules and we used them in our implementations. Following the implementation, we ran simulations for various input combinations to validate our design.



We implemented the circuit that we have designed in **Preliminary 2** section using a 3:8 DECODER and OR gates. We wrote the DECODER and OR gate as modules and we used them later in our implementations. Following the implementation, we ran simulations for various input combinations to validate your design.



5 CONCLUSION [10 points]

We used different methods to obtain prime implicants. We learned how to:

- express functions and implement them in circuits using AND, OR, NOT, NAND gates, multiplexer and decoder.

- the internal structure of plexers and implementing these to our Verilog codes.

- express the functions which contain "and, or, not" operations with using only NAND gates.

- design and draw the lowest cost expression.

Drawing the internal structures of plexers and schemes of functions helped us to implement the modules.

The difficulties we faced are using programs in different environment(**Mac OS**), languages and group work.