Digital Design with Truth tables and K-Maps

Instructor: Dr. Vinicius Prado da Fonseca (vpradodafons@online.mun.ca)

MEMORIAL UNIVERSITY
www.mun.ca

Digital Design

Translate the problem's definition into a circuit using gates

Set of steps that translates a problem into gates:

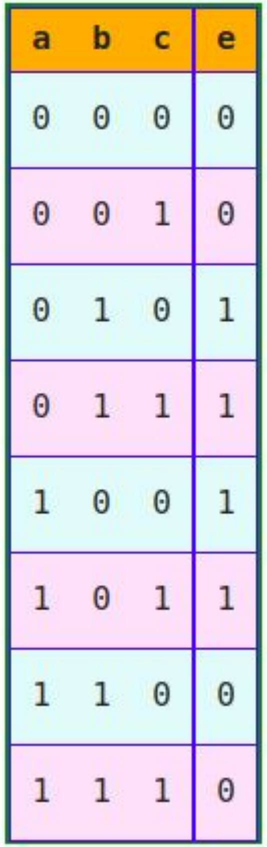
- Use a truth table to describe a solution to the problem.
- Write the truth table as a K-map.
- Use the K-map to produce a minimized Boolean expression.
- Use the Boolean expression to design a circuit.
- Run the examples using the course repo/devcontainer

Circuit to detect 2 ≤ (a, b, c) ≤ 5

- Detects the range 2 to 5
- Each row of the table can represent an integer, $000_2 = 0_{10}$, $001_2 = 1_{10}$, ... $111_2 = 7_{10}$

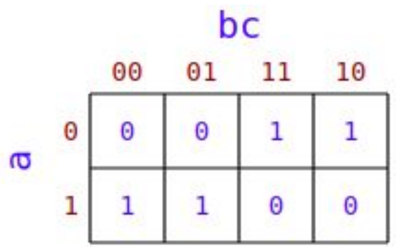
Circuit to detect 2 ≤ (a, b, c) ≤ 5

- Detects the range 2 to 5
- Each row of the table can represent an integer, 000 = 0, 001 = 1, ... 111 = 7

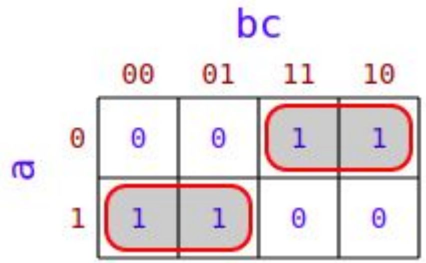


a	b	c	e
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Circuit to detect 2 ≤ (a, b, c) ≤ 5

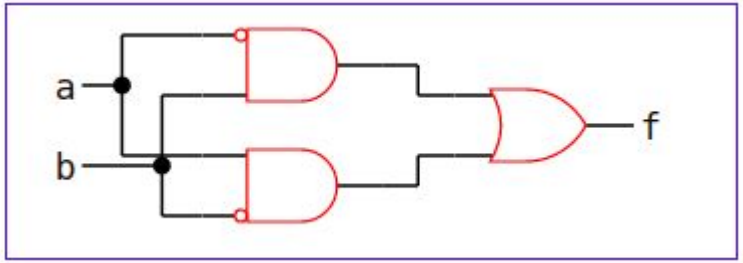


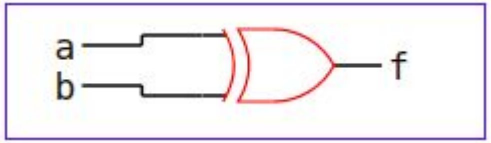
a \ bc	00	01	11	10
0	0	0	1	1
1	1	1	0	0

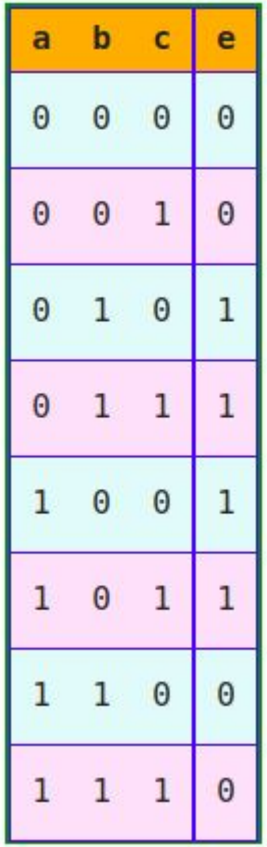


a \ bc	00	01	11	10
0	0	0	1	1
1	1	1	0	0

$$\overline{a}\cdot b + a\cdot\overline{b} = a \oplus b.$$







a	b	c	e
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

- Verilog implementation

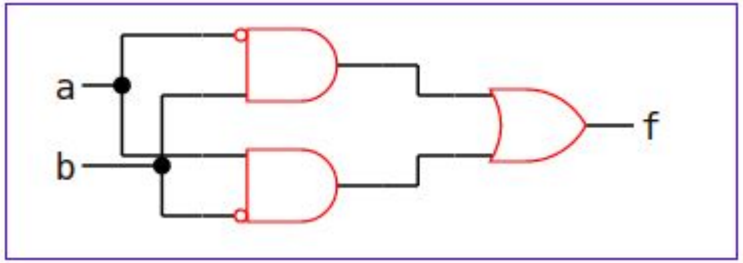
Circuit to detect 2 ≤ (a, b, c) ≤ 5 (Verilog)

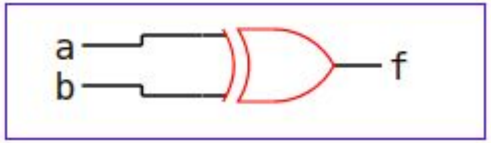
```
module detect2_5(output logic y, input logic a, b, c);
    logic o1, o2, nota, notb;
    not #2 n1(nota, a);
    not #2 n2(notb, b);
    and #2 a1(o1, nota, b);
    and #2 a2(o2, notb, a);
    or #2 or1(y, o1, o2);
endmodule
```

```
module detect2_5(output logic y, input logic a, b, c);

    xor #2 x1(y, a, b);

endmodule
```

```
module detect2_5(output logic y, input logic a, b, c);

        assign y = a ^ b;

endmodule
```

$$\overline{a} \cdot b + a \cdot \overline{b} = a \oplus b.$$





Circuit to detect 2 ≤ (a, b, c) ≤ 5 (Verilog tb)

```
`timescale 1ns / 1ns

module main;
    logic a, b, c, f;
    integer all_inputs, i;

    detect2_5 dut (f, a, b, c);

    assign {a,b,c} = all_inputs[2:0];

    initial begin
        $display("t=%04d abc f", $time);
        i = 0;
        repeat ( 8 ) begin
            all_inputs = i;

            #10;
            $display("t=%04d %b%b%b %b", $time, a, b, c, f);
            i = i + 1;
        end
    end
endmodule
```

Add 3 to 3-bits

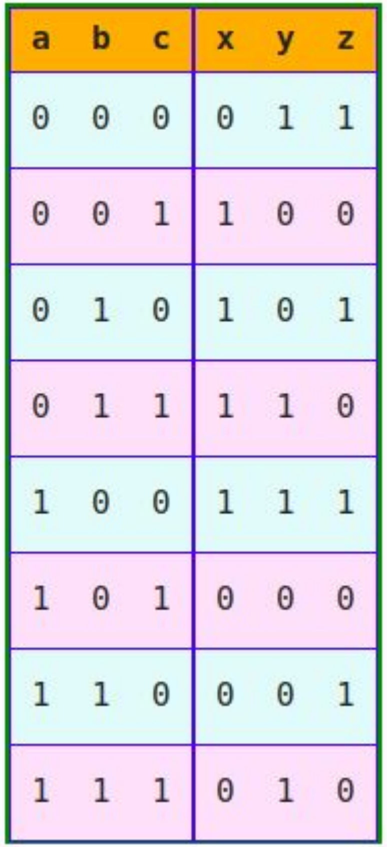
The truth table for a 3-input 3-output circuit that adds 011 (3_{10})

We have 3 outputs, 3 equations: x, y and z.

Add 3 to 3-bits

The truth table for a 3-input 3-output circuit that adds 011 (3_{10})

We have 3 outputs, 3 equations: x, y and z.

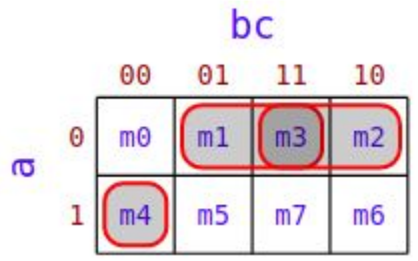


a	b	c	x	y	z
0	0	0	0	1	1
0	0	1	1	0	0
0	1	0	1	0	1
0	1	1	1	1	0
1	0	0	1	1	1
1	0	1	0	0	0
1	1	0	0	0	1
1	1	1	0	1	0

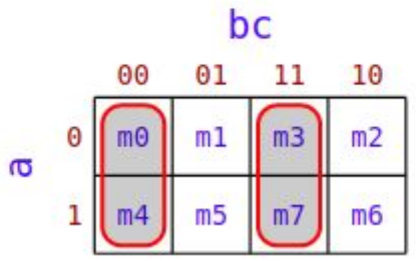
Add 3 to 3-bits

The truth table for a 3-input 3-output circuit that adds 011 (3_{10})

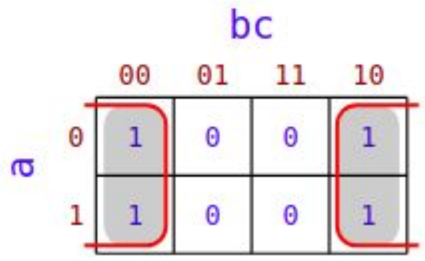
x=

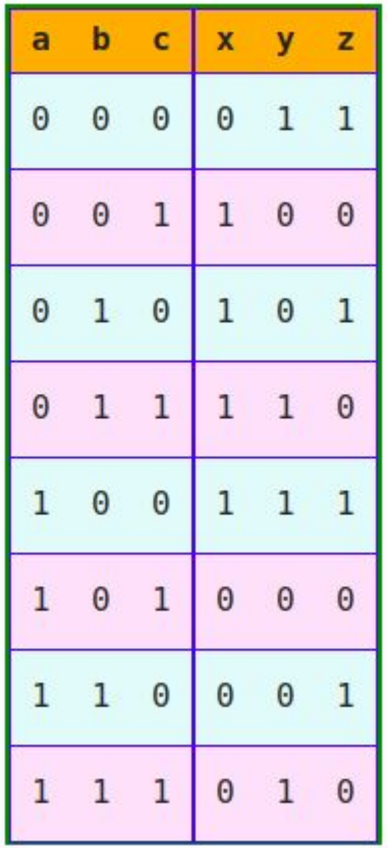


y=



z=



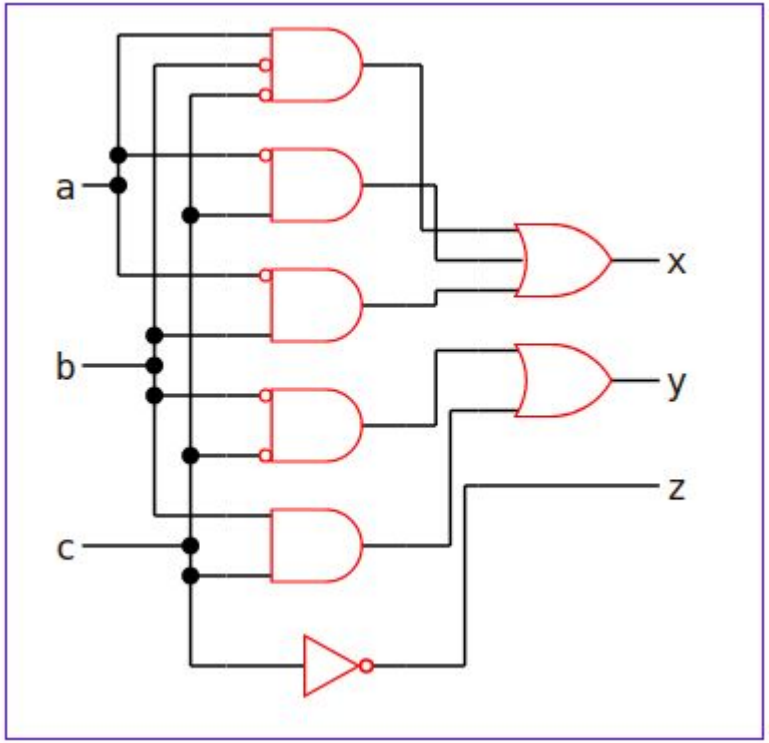


a	b	c	x	y	z
0	0	0	0	1	1
0	0	1	1	0	0
0	1	0	1	0	1
0	1	1	1	1	0
1	0	0	1	1	1
1	0	1	0	0	0
1	1	0	0	0	1
1	1	1	0	1	0

Add 3 to 3-bits

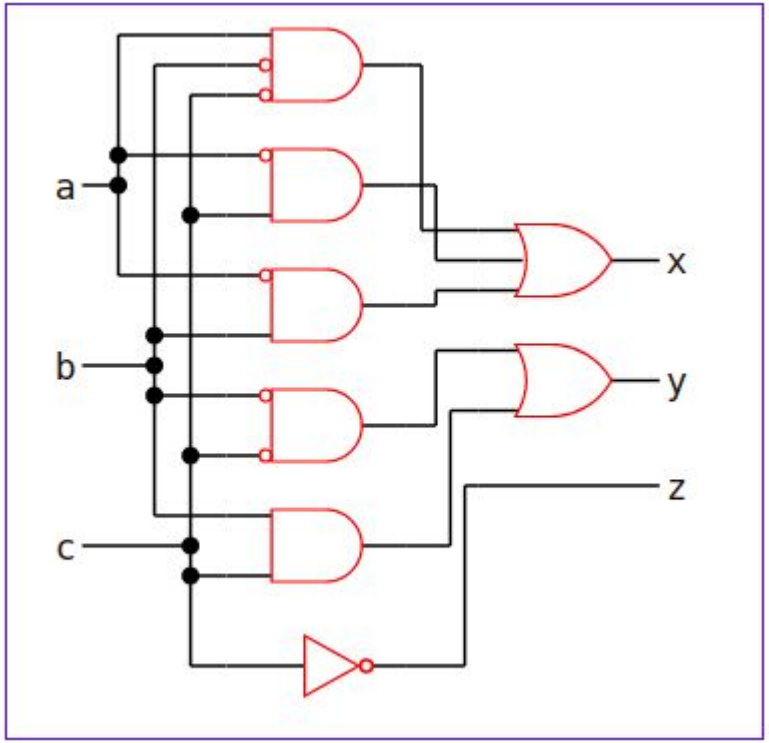
The minimized Boolean expressions are:

$$x = a \cdot \overline{b} \cdot \overline{c} + \overline{a} \cdot c + \overline{a} \cdot b$$
$$y = \overline{b} \cdot \overline{c} + b \cdot c$$
$$z = \overline{c}$$



Add 3 to 3-bits

```
module add_3(output logic x, y, z, input logic a, b, c);

    logic n_a, n_b, n_c;
    logic i_0, i_1, i_2, i_3, i_4;

    not (n_a, a);
    not (n_b, b);
    not (n_c, c);


    and a_0(i_0, a, n_b, n_c);
    and a_1(i_1, n_a, c);
    and a_2(i_2, n_a, b);
    and a_3(i_3, n_b, n_c);
    and a_4(i_4, b, c);

    or (x, i_0, i_1, i_2);  // x = a&~b&~c | ~a&c | ~a&b
    or (y, i_3, i_4);       // y = ~b&~c | b&c

    buf (z, n_c);           // z = ~c
endmodule
```



Add 3 to 3-bits

Functional verilog using Boolean expressions:

```
module add_3(output logic x, y, z, input
logic a, b, c);
    assign x = a&~b&~c | ~a&c | ~a&b;
    assign y = ~b&~c | b&c;
    assign z = ~c;
endmodule
```

$$x = a \cdot \overline{b} \cdot \overline{c} + \overline{a} \cdot c + \overline{a} \cdot b$$
$$y = \overline{b} \cdot \overline{c} + b \cdot c$$
$$z = \overline{c}$$

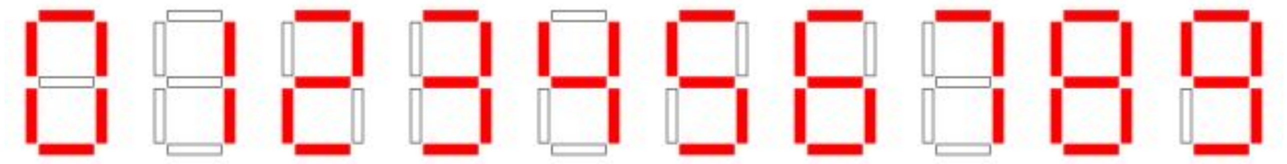
Add 3 to 3-bits

Testbench

```
`timescale 1ns/1ns

module main;
    logic a, b, c;
    logic x, y, z;
    integer all_inputs, i;

    add_3 dut(x,y,z,a,b,c);

    assign {a,b,c} = all_inputs[2:0];

    initial begin
        i = 0;
        #10 $display("t=%04d abc xyz", $time);
        repeat (8) begin
            all_inputs = i;
            #10 $display("t=%04d %b%b%b %b%b%b", $time, a, b, c, x, y, z);
            i = i + 1;
        end
    end
endmodule
```

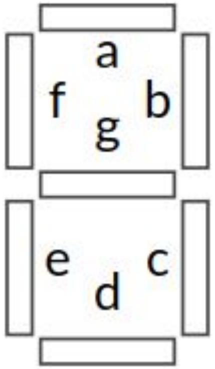
Seven Segment Displays (segment a) [HERE]

The digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 can be displayed with seven segments



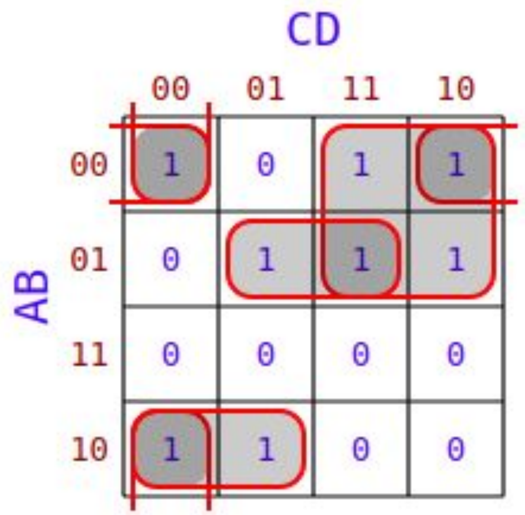
Seven Segment Displays (segment a)

- The seven segments can be labeled with a, b, c, d, e and f
- The digit 1 is displayed when segments b and c are turned on

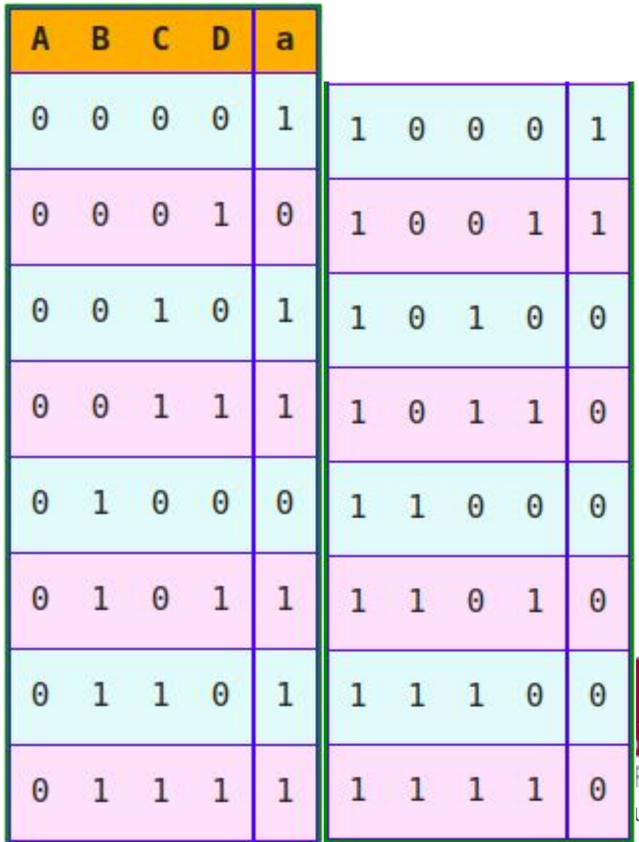


Seven Segment Displays (segment a)

The K-map for the a-segment controller is:



AB \ CD	00	01	11	10
00	1	0	1	1
01	0	1	1	1
11	0	0	0	0
10	1	1	0	0



A	B	C	D	a
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

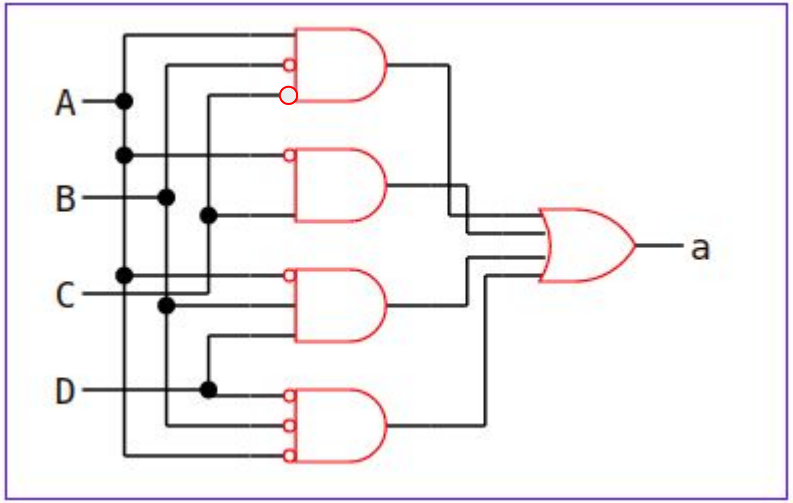
Seven Segment Displays (segment a)

The prime implicants are: $\overline{A} \cdot \overline{B} \cdot \overline{D}$, $\overline{B} \cdot \overline{C} \cdot \overline{D}$, $A \cdot \overline{B} \cdot \overline{C}$, $\overline{A} \cdot C$, and $\overline{A} \cdot B \cdot D$

The essential prime implicants: $A \cdot \overline{B} \cdot \overline{C}$, $\overline{A} \cdot C$, and $\overline{A} \cdot B \cdot D$

One possible minimal Boolean expression is:

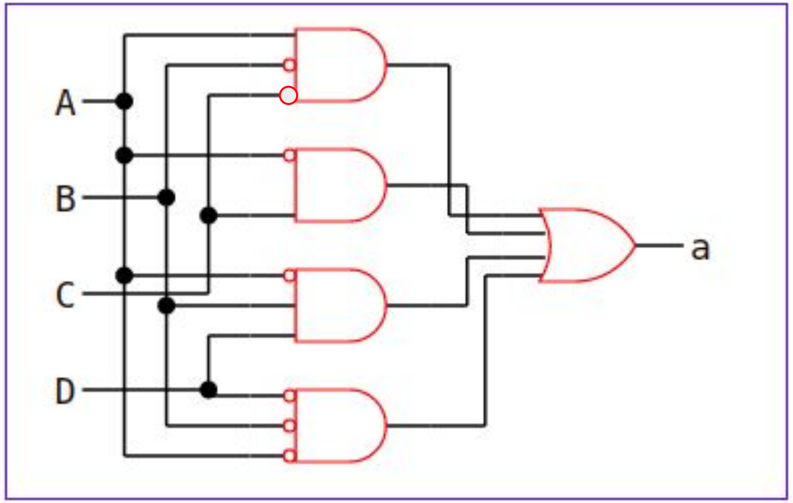
$$A \cdot \overline{B} \cdot \overline{C} + \overline{A} \cdot C + \overline{A} \cdot B \cdot D + \overline{A} \cdot \overline{B} \cdot \overline{D}$$



Seven Segment Displays (segment a)

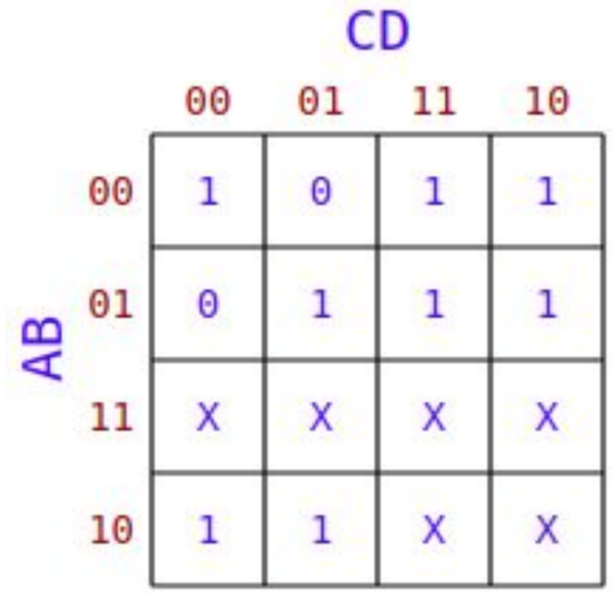
```
module seg_a(output logic out_a, input logic a, b,
c, d);

    logic n_a, n_b, n_d;
    logic i_0, i_1, i_2, i_3;

    not (n_a, a);
    not (n_b, b);
    not (n_c, c);
    not (n_d, d);

    and a_0(i_0, a, n_b, n_c);
    and a_1(i_1, n_a, c);
    and a_2(i_2, n_a, b, d);
    and a_3(i_3, n_d, n_b, n_a);

    or (out_a, i_0, i_1, i_2, i_3);

endmodule
```

$$A \cdot \overline{B} \cdot \overline{C} + \overline{A} \cdot C + \overline{A} \cdot B \cdot D + \overline{A} \cdot \overline{B} \cdot \overline{D}$$



Don't Cares in Truth Tables

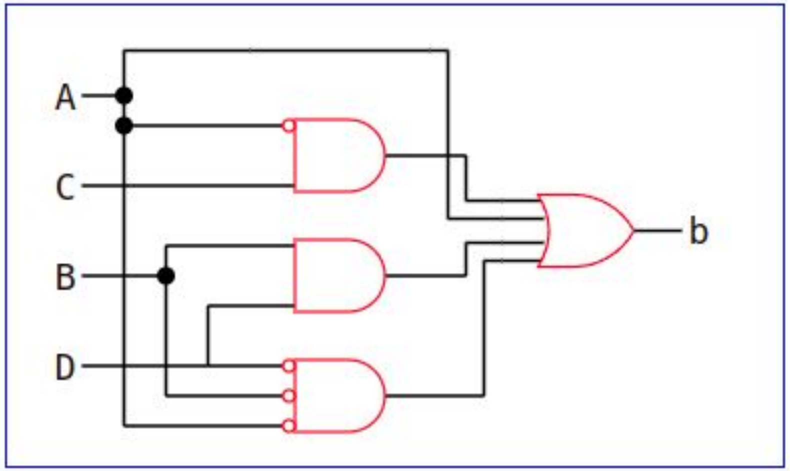
- 4-bit -> 16 possible values.
- 3-bit -> 8 possible values
- Ten decimal digits only require 10 values.
- 6 values -> don't care values
  - 11 - 15
  - Can be used to further simplify the Boolean expression
- A K-map with don't cares for the a-segment is:

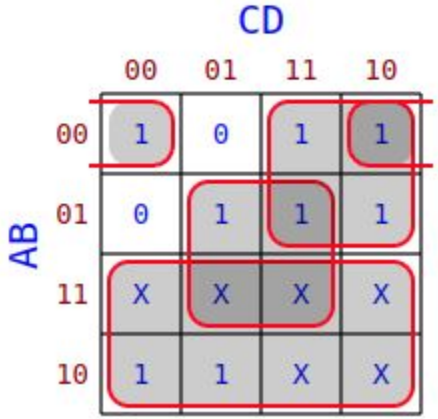


AB \ CD	00	01	11	10
00	1	0	1	1
01	0	1	1	1
11	X	X	X	X
10	1	1	X	X

Don't Cares in Truth Tables

$A + B \cdot D + \overline{A} \cdot C + \overline{A} \cdot \overline{B} \cdot \overline{D}$

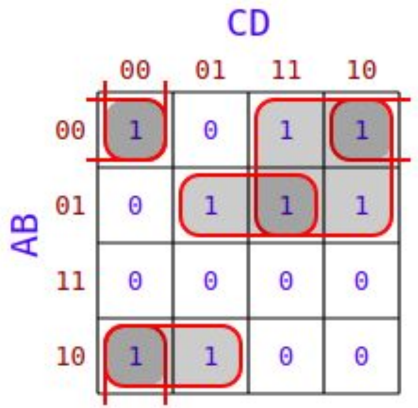


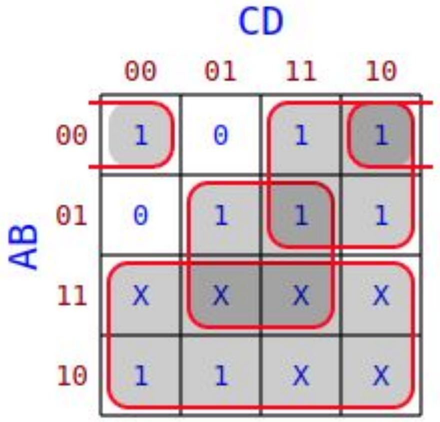


AB \ CD	00	01	11	10
00	1	0	1	1
01	0	1	1	1
11	X	X	X	X
10	1	1	X	X

Don't Cares in Truth Tables

```
module seg_a(output logic out_a, input logic a, b, c, d);

    assign out_a = c | a | b&d | ~b&~d; // not same as notes
    //assign out_a = a | b&d | ~a&c | ~a&~b&~d;

endmodule
```

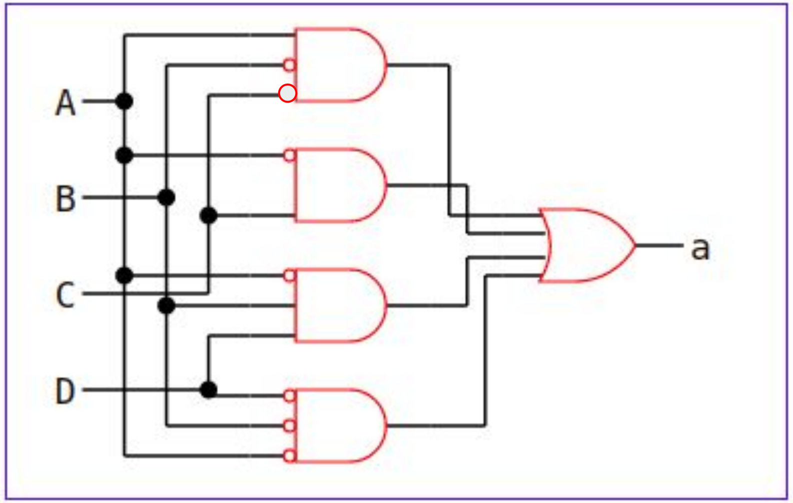


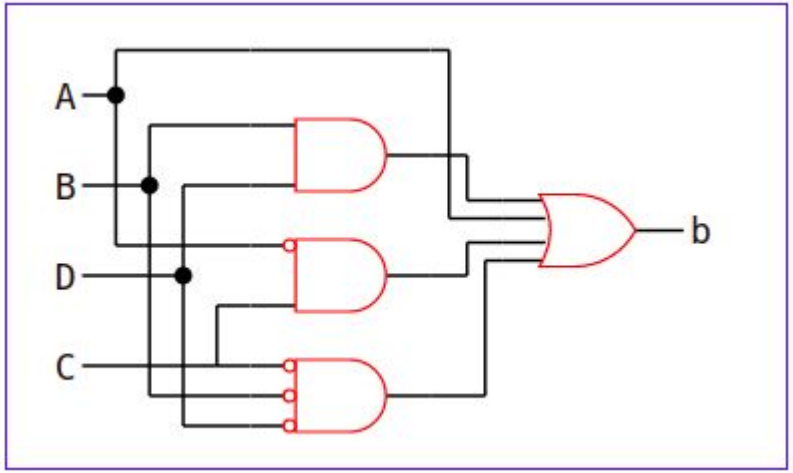


MEMORIAL UNIVERSITY
www.mun.ca

Don't Cares in Truth Tables

Before and after





Online notes

- More examples
  - Practice implementing in verilog
  - Previous lecture, 10 - Verilog For Combinational Logic, also has some examples
  - Use the repo and devcontainer
- Last Unit 2 lecture

Questions?

Next: Unit 3, Sequential Logic, FSM

Lab 3 material

- Here we finish the lab 3 material
- Get the lab manual from:

Contents > Labs > Lab support files > Lab Manual