

# Lecture 7 – Logic implementation

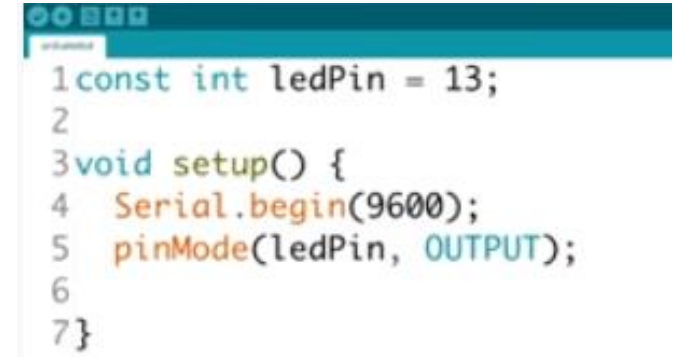
Dr. Aftab M. Hussain,  
Assistant Professor, PATRIoT Lab, CVEST

## Chapter 3



# Arduino coding

- Embedded systems are generally operating in a loop – either waiting for an input to process or operating something continuously
- For that to happen, we need to “setup” the microcontroller pins in either input mode or output mode
- This and other things you can do in the setup function
- This is only run once when the microcontroller boots
- Once this is done, we run the loop function indefinitely
- This function can be trained to continuously read a particular input and process a certain output for it
- Like continuously read serial input and output a certain value for a particular input



```
1const int ledPin = 13;
2
3void setup() {
4  Serial.begin(9600);
5  pinMode(ledPin, OUTPUT);
6
7}
```

# Product of Sums

- The minimized Boolean functions derived from the K-map in the previous lecture were expressed in sum-of-products form
- With a minor modification, the product-of-sums form can be obtained
- The 1's placed in the squares of the map represent the minterms of the function
- From this observation, we see that the complement of a function is represented in the map by the squares not marked by 1's
- If we mark the empty squares by 0's and combine them into valid adjacent squares, we obtain a simplified sum-of-products expression of the complement of the function (i.e., of  $F'$ )
- The complement of  $F'$  gives us back the function  $F$  in product-of-sums form (a consequence of DeMorgan's theorem)

# Product of Sums

- Simplify the following Boolean function into (a) sum-of-products form and (b) product-of-sums form:

$$F(w, x, y, z) = \sum(0, 1, 2, 5, 8, 9, 10)$$

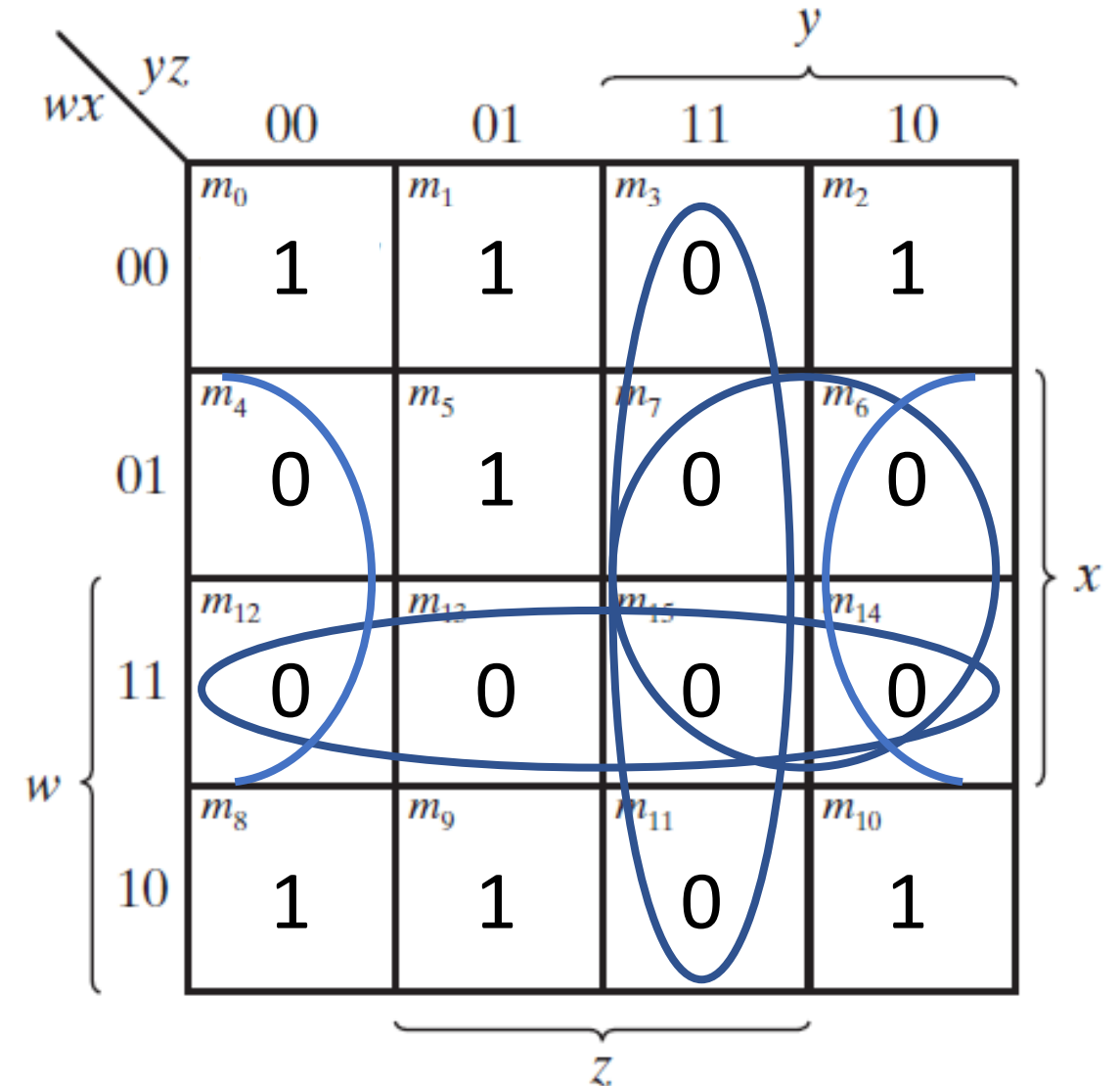
- Two clusters of four squares:  $x'z'$  and  $x'y'$
- One cluster of two squares:  $w'y'z$
- Thus, the function is:  
$$F = x'z' + x'y' + w'y'z$$

		$y$			
		$yz$		11	10
$w$	$x$	00	01	11	10
	00	$m_0$ 1	$m_1$ 1	$m_3$ 0	$m_2$ 1
	01	$m_4$ 0	$m_5$ 1	$m_7$ 0	$m_6$ 0
	11	$m_{12}$ 0	$m_{13}$ 0	$m_{15}$ 0	$m_{14}$ 0
	10	$m_8$ 1	$m_9$ 1	$m_{11}$ 0	$m_{10}$ 1



# Product of Sums

- Thus, the function is:  
$$F = x'z' + x'y' + w'y'z$$
- Now, let us see the complement of the function:  $F'$
- This can be obtained from clustering all the 0s together
- We have four clusters of four (prime implicants)
- Of these, the essential prime implicants are:  $yz, wx, xz'$
- Thus,  $F' = wx + yz + xz'$



# Product of Sums

- The original function is:
$$F = x'z' + x'y' + w'y'z$$
- Thus, the complement function is
$$F' = wx + yz + xz'$$
- Now, we can obtain F back from F' using the DeMorgan's theorems
- Thus,
$$F = (w' + x')(y' + z')(x' + z)$$
- Hence, the actual simplest implementation of a function can be through its complement (product of sum)

A 4x4 Karnaugh map for the complement function  $F' = wx + yz + xz'$ . The map is labeled with variables  $w, x, y, z$  and their complements. The rows are labeled  $w$  (00, 01, 11, 10) and the columns are labeled  $x$  (00, 01, 11, 10). The cells are labeled  $m_0$  through  $m_{15}$ . The values in the cells are:  $m_0=1, m_1=1, m_3=0, m_2=1$  (row 00);  $m_4=0, m_5=1, m_7=0, m_6=0$  (row 01);  $m_{12}=0, m_{13}=0, m_{15}=0, m_{14}=0$  (row 11);  $m_8=1, m_9=1, m_{11}=0, m_{10}=1$  (row 10). Blue circles highlight the prime implicants: a circle around  $m_3$  and  $m_{11}$  (representing  $xz'$ ), a circle around  $m_3$  and  $m_7$  (representing  $yz$ ), and a circle around  $m_3$  and  $m_{15}$  (representing  $wx$ ). There are also circles around  $m_4$  and  $m_8$ ,  $m_5$  and  $m_9$ ,  $m_6$  and  $m_{10}$ , and  $m_7$  and  $m_{11}$ .

$w \backslash x \backslash yz$	00	01	11	10
00	$m_0$ 1	$m_1$ 1	$m_3$ 0	$m_2$ 1
01	$m_4$ 0	$m_5$ 1	$m_7$ 0	$m_6$ 0
11	$m_{12}$ 0	$m_{13}$ 0	$m_{15}$ 0	$m_{14}$ 0
10	$m_8$ 1	$m_9$ 1	$m_{11}$ 0	$m_{10}$ 1

# Product of Sums

- Let us see if we can directly get the product of sum form from the map
- Obtain a product of sum logic circuit for the following function:

$$F(w, x, y, z) = \sum (0, 1, 3, 4, 5, 9, 11, 13, 15)$$

- We get two clusters of four:  $yz'$  and  $wz'$
- One cluster of two:  $w'xy$
- Thus,  $F = (y' + z)(w' + z)(w + x' + y')$

		$y$			
		$yz$		11	10
$w$	$x$	00	01	11	10
	00	$m_0$ 1	$m_1$ 1	$m_3$ 1	$m_2$ 0
	01	$m_4$ 1	$m_5$ 1	$m_7$ 0	$m_6$ 0
	11	$m_{12}$ 0	$m_{13}$ 1	$m_{15}$ 1	$m_{14}$ 0
$w$	10	$m_8$ 0	$m_9$ 1	$m_{11}$ 1	$m_{10}$ 0



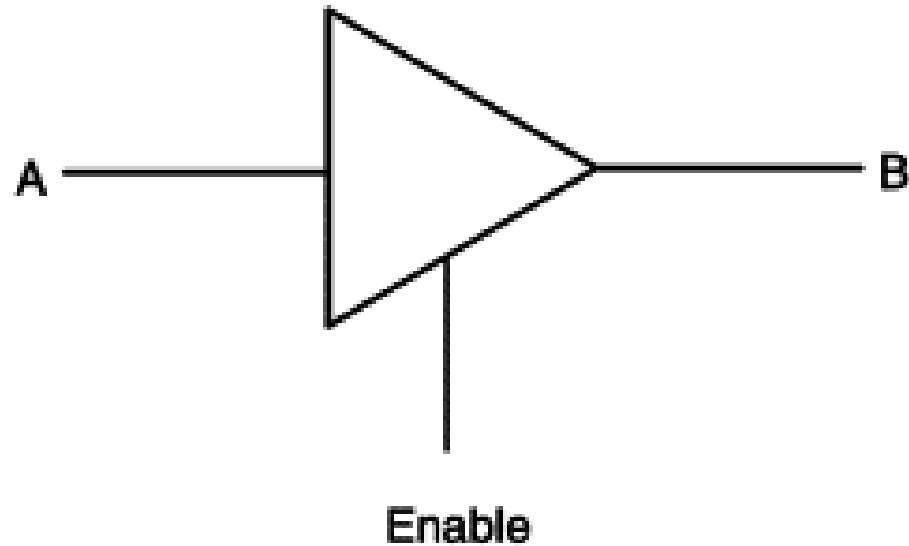
# The Other states

---

- This is REALLY important
- Apart from the two states of 0 and 1, there is a third state called the high impedance state denoted by Z
- When we say a particular pin is at HIGH or LOW state, we are assuming a driver behind it, i.e., the pin is *driven* to HIGH or LOW value
- This can be done through a transistor connecting the pin to either ground or  $V_{cc}$
- However, if we do not connect a pin to either of HIGH or LOW, the pin is said to be in high impedance state or in Z state
- This concept is used extensively in the digital logic world to control buses

# The Other states

- Most logic gates only output HIGH or LOW, the third state is generally obtained using tristate buffers
- These are used just before the bus connections



Enable	A	B
0	0	Z
0	1	Z
1	0	0
1	1	1

# The Other states

---

- Another choice is that we can have either 0 or 1 (but not both together)
- This is called the don't care state – or a condition in the logic function that follows that we do not care what the output in a particular case is, ie, for a particular set of inputs
- This is represented as X
- This can be either 0 or 1 and both are equally acceptable while forming logic circuits for a given function
- Consider a simple two variable statement: If x then y
- We can interpret this as we need to know the value of y when x is TRUE, but when x is FALSE, we DON'T CARE!

# The Other states

- Consider a simple two variable statement: If A then B
- One way we can interpret this as we need to know the value of B when A is TRUE, but when B is FALSE, we DON'T CARE!
- If this is the case, we can make the truth table for the function as shown
- In this case, because X can take either 0 or 1 value, we can simply make the desired function using an AND gate

A	B	F
0	0	X
0	1	X
1	0	0
1	1	1

# The Other states

- Simplify the Boolean function

$$F(w, x, y, z) = \sum (1, 3, 7, 11, 15)$$

- We have one cluster of four:  $yz$  and one cluster of two:  $w'x'z$

- Thus, the function is

$$F = yz + w'x'z$$

		$y$			
		$yz$		11	10
$w$	$x$	00	01	11	10
	00	$m_0$ 0	$m_1$ 1	$m_3$ 1	$m_2$ 0
	01	$m_4$ 0	$m_5$ 0	$m_7$ 1	$m_6$ 0
	11	$m_{12}$ 0	$m_{13}$ 0	$m_{15}$ 1	$m_{14}$ 0
$w$	10	$m_8$ 0	$m_9$ 0	$m_{11}$ 1	$m_{10}$ 0

# The Other states

- Now, consider the same function

$$F(w, x, y, z) = \sum (1, 3, 7, 11, 15)$$

- which has the don't-care conditions

$$d(w, x, y, z) = \sum (0, 2, 5)$$

- Now, we have two clusters of four squares:  $yz$  and  $w'z$
- This is because  $m_5$  can be 1, and it is ok if  $m_0$  and  $m_2$  are 0
- Thus, the function is  $F = yz + w'z$
- The function can also be simplified as  $F = yz + w'x'$

		$y$			
		$yz$		11	10
$w$	$x$	00	01	11	10
	00	$m_0$ X	$m_1$ 1	$m_3$ 1	$m_2$ X
	01	$m_4$ 0	$m_5$ X	$m_7$ 1	$m_6$ 0
	11	$m_{12}$ 0	$m_{13}$ 0	$m_{15}$ 1	$m_{14}$ 0
$w$	10	$m_8$ 0	$m_9$ 0	$m_{11}$ 1	$m_{10}$ 0



# The Other states

- Functions  $F = yz + w'z$  and  $F = yz + w'x'$  are different functions
- However, both expressions include minterms 1, 3, 7, 11, and 15 that make the function  $F$  equal to 1
- The don't-care minterms 0, 2, and 5 are treated differently in each expression
- The first expression includes minterms 0 and 2 with the 1's and leaves minterm 5 with the 0's
- The second expression includes minterm 5 with the 1's and leaves minterms 0 and 2 with the 0's
- The two expressions represent two functions that are not algebraically equal

		$y$			
		$yz$		11	10
$w$	$x$	00	01	11	10
	00	$m_0$ X	$m_1$ 1	$m_3$ 1	$m_2$ X
	01	$m_4$ 0	$m_5$ X	$m_7$ 1	$m_6$ 0
	11	$m_{12}$ 0	$m_{13}$ 0	$m_{15}$ 1	$m_{14}$ 0
	10	$m_8$ 0	$m_9$ 0	$m_{11}$ 1	$m_{10}$ 0

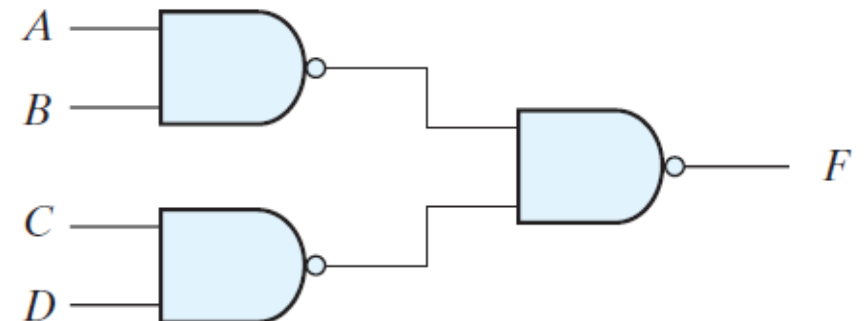
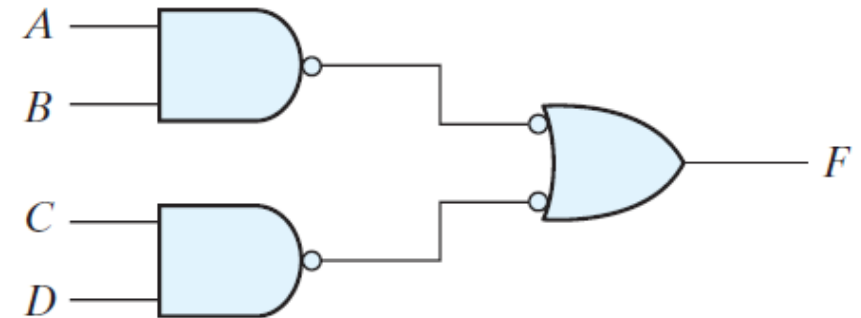
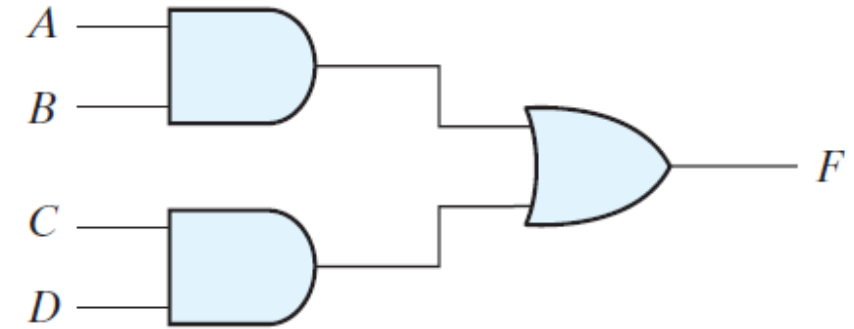
# The Other states

- The key question is: Are these the simplest possible representations for the given function?  $F = yz + w'z$  and  $F = yz + w'x'$
- What if we look at the product of sum simplification?
- We can get a cluster of 8 squares:  $z'$
- And a cluster of four squares:  $wy'$
- Thus, the function can be represented as  $F = z(w' + y)$

		$y$			
		$yz$	00	01	11
$w$	00	$m_0$ X	$m_1$ 1	$m_3$ 1	$m_2$ X
	01	$m_4$ 0	$m_5$ X	$m_7$ 1	$m_6$ 0
	11	$m_{12}$ 0	$m_{13}$ 0	$m_{15}$ 1	$m_{14}$ 0
	10	$m_8$ 0	$m_9$ 0	$m_{11}$ 1	$m_{10}$ 0

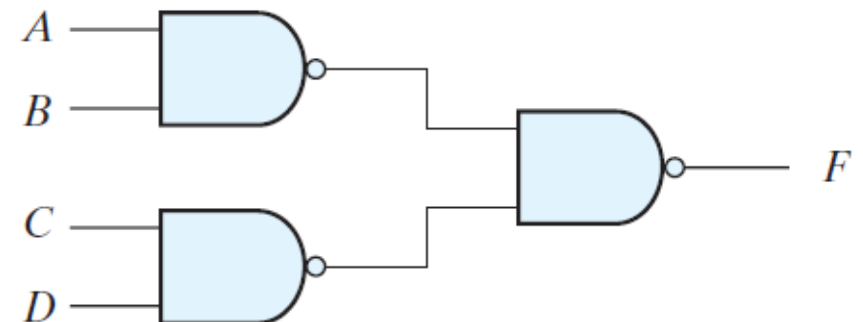
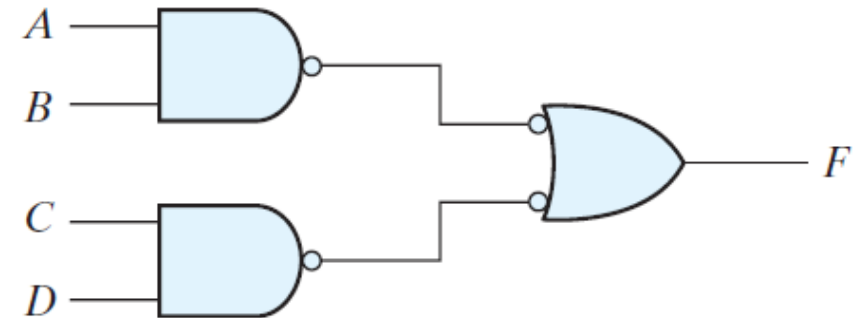
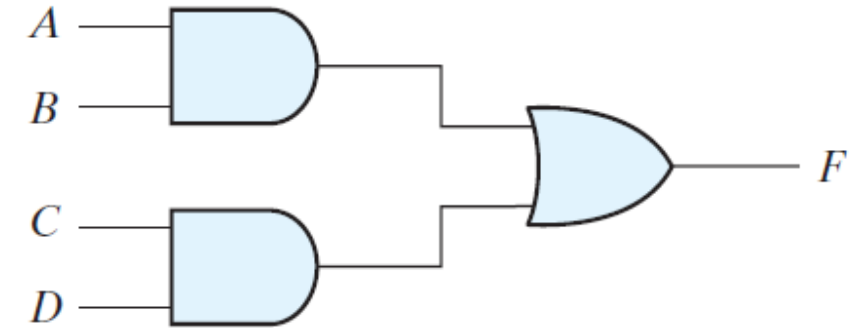
# NAND implementations

- Remember that NAND and NOR are universal gates – thus, we should be able to make any implementation using NAND and NOR gates
- Consider a simple AND-OR implementation (sum of products)
- We can put two complementary operation in the middle of the wire
- The first layer AND gates become NAND and the second layer OR gate can be made NAND using the DeMorgan theorem



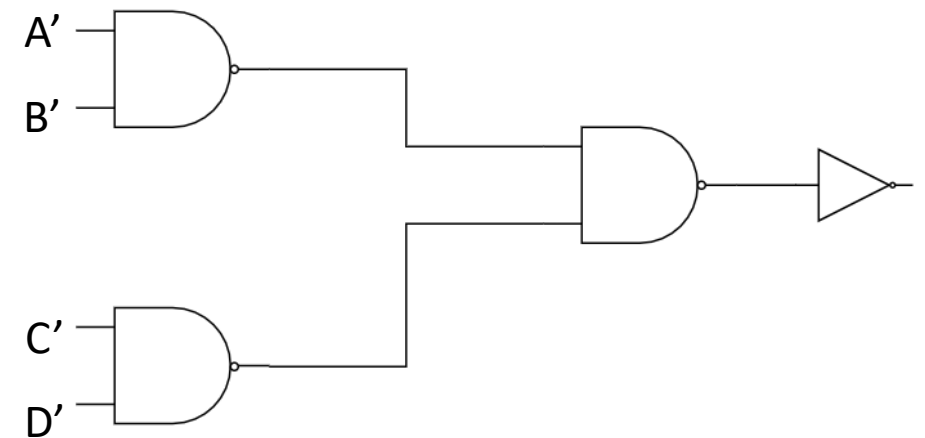
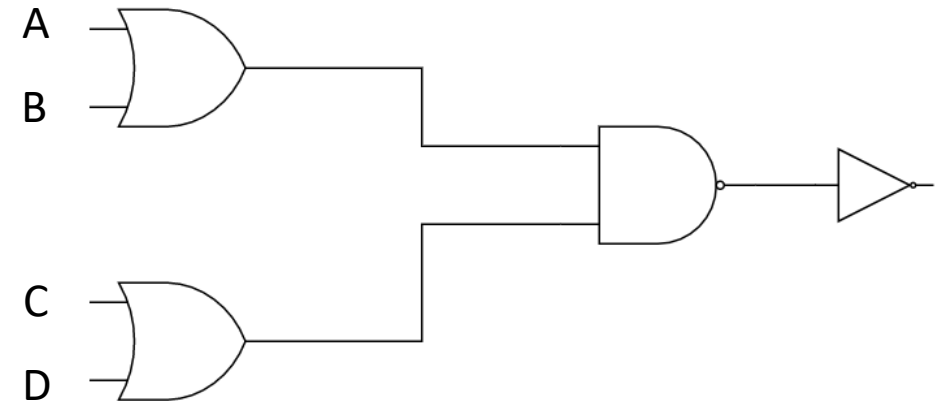
# NAND implementations

- The procedure for obtaining the NAND logic diagram from a Boolean function is as follows:
  1. Simplify the function and express it in sum-of-products form
  2. Draw a NAND gate for each product term of the expression that has at least two literals. The inputs to each NAND gate are the literals of the term. This procedure produces a group of first-level gates
  3. Draw a single NAND gate with inputs coming from outputs of first-level gates.
  4. A term with a single literal requires an inverter in the first level. However, if the single literal is complemented, it can be connected directly to an input of the second level NAND gate



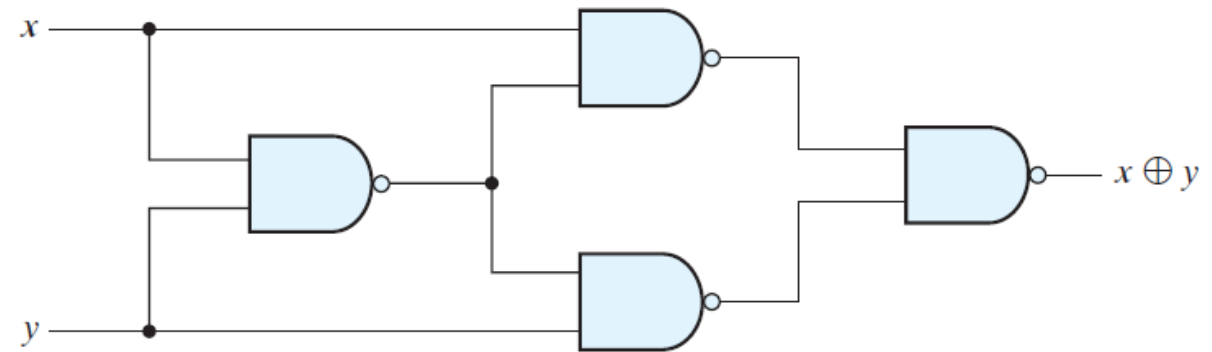
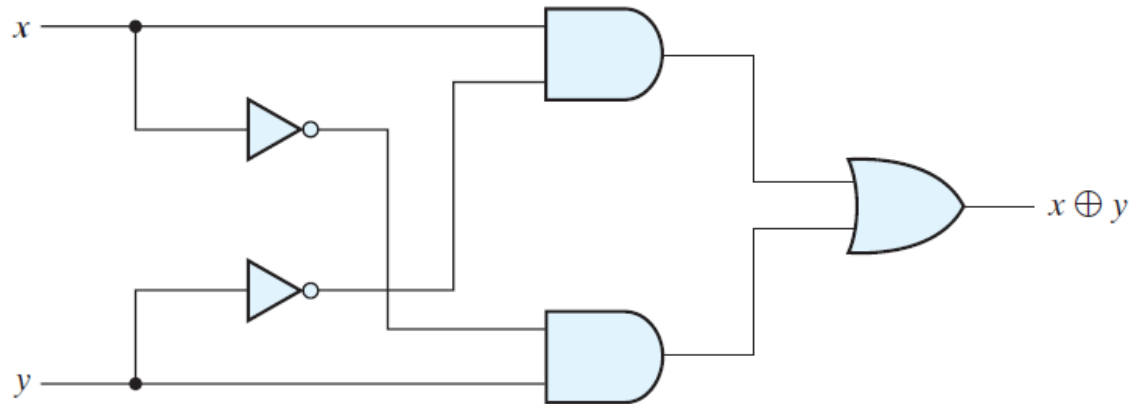
# NAND implementations

- In case of OR-AND implementation (product of sum)
- We have to complement the output to get a NAND gate at the second level
- Then we complement the inputs we are providing and obtain NAND gates at the first level
- The NOR operation is the dual of the NAND operation
- Therefore, all procedures and rules for NOR logic are the duals of the corresponding procedures and rules developed for NAND logic



# ExOR gate

- ExOR is weird because there is no easy way to make an ExOR gate other than simply implementing its logic function





# ExOR gate

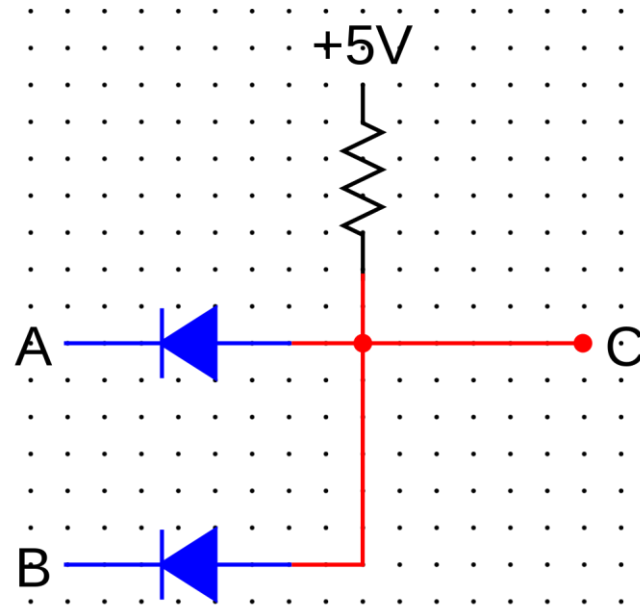
- ExOR is weird because there is no easy way to make an ExOR gate other than simply implementing its logic function

$A \backslash BC$		$B$			
		00	01	11	10
$A$	0	$m_0$	$m_1$ 1	$m_3$	$m_2$ 1
	1	$m_4$ 1	$m_5$	$m_7$ 1	$m_6$

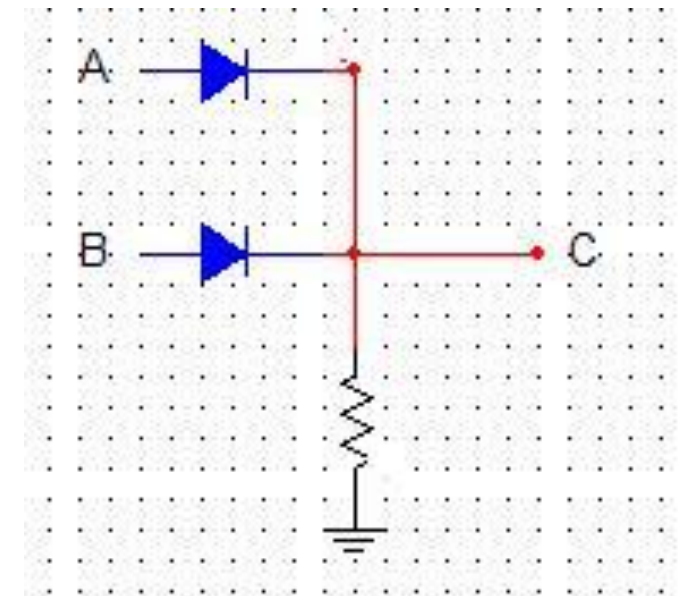
$AB \backslash CD$		$C$			
		00	01	11	10
$A$	00	$m_0$	$m_1$ 1	$m_3$	$m_2$ 1
	01	$m_4$ 1	$m_5$	$m_7$ 1	$m_6$
	11	$m_{12}$	$m_{13}$ 1	$m_{15}$	$m_{14}$ 1
	10	$m_8$ 1	$m_9$	$m_{11}$ 1	$m_{10}$

# Wired AND and Wired OR

- This is a way of making AND and OR logic gates without the use of complex components like transistors
- A wired logic connection is a logic gate that implements Boolean algebra (logic) using only components such as diodes and resistors
- Diodes in parallel can be configured to obtain wired AND and wired OR by using a “pull-up” resistor or a “pull-down” resistor



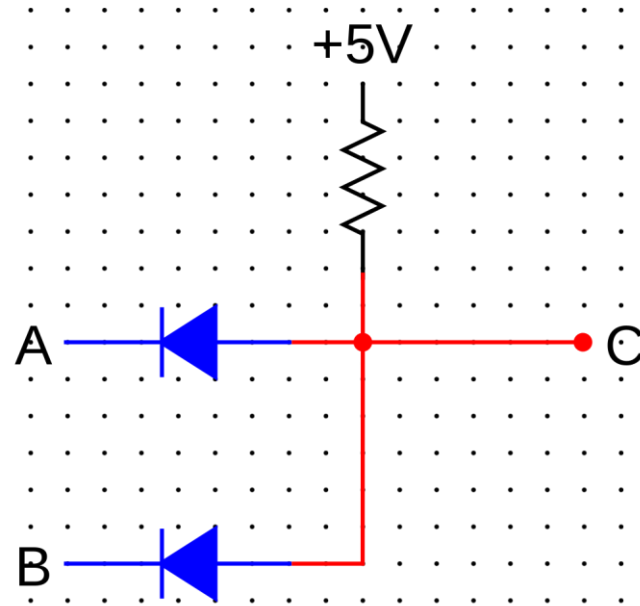
Wired AND



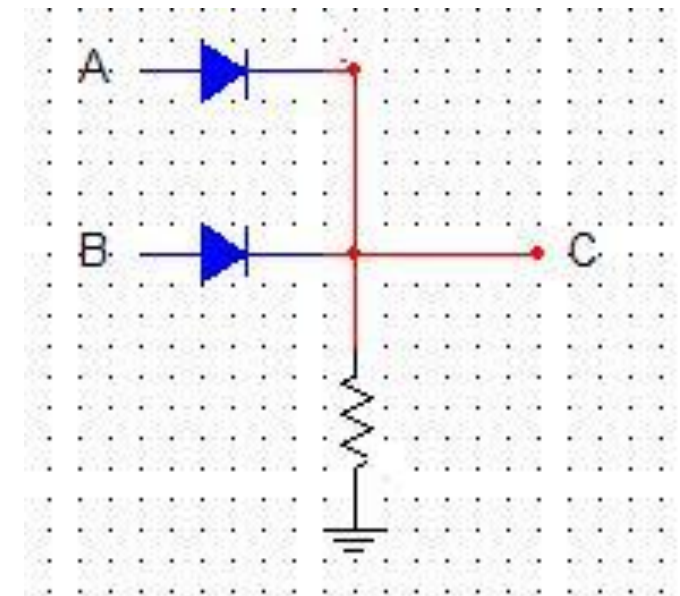
Wired OR

# Wired AND and Wired OR

- Advantages
  - Simple to implement
  - Low-cost
- Problems:
  - Wired AND cannot drive the voltage to LOW and Wired OR cannot drive the voltage to HIGH
  - Very high power for certain states
  - May not be compatible in case of multiple levels of logic



Wired AND



Wired OR