# CSCE 312: Computer Organization

David Kebo Houngninou

# Digital Logic Design

# Bitwise operators

| Operator | Description |
|---|---|
| ~<br>Binary NOT | Each bit of the output is 1 if the bit of x is 0, otherwise it is 0. |
| &<br>Binary AND | Each bit of the output is 1 if the bit of x AND the bit of y is 1, otherwise it's 0. |
| \|<br>Binary OR | Each bit of the output is 0 if the bit of x AND the bit of y is 0, otherwise it's 1. |
| ^<br>Binary XOR | Each bit of the output is 1 if the bit of x is the one complement of the bit of y, otherwise it's 0 |
| <<<br>Binary Left Shift | x << y returns x with the bits shifted to the left by y places. Performs a multiplication of x by $2^y$. |
| >><br>Binary Right Shift | x >> y returns x with the bits shifted to the right by y places. Performs a floor division of x by $2^y$. |

---

# Logical operators

Be careful. Do not mix bitwise operators and logical operators!

This is a common mistake in C programming.

Logical operators are: `&&, ||, !`

- View 0 as “False”, anything nonzero is “True”
- Always return 0 or 1

| Operator | Description | Example |
|---|---|---|
| && | Logical AND. If both operands are non-zero, then the condition is true. | (A && B) is false. |
| \|\| | Logical OR. If any of the operands is non-zero, then the condition is true. | (A \|\| B) is true. |
| ! | Logical NOT. Reverses the logical state of its operand. If a condition is true, then the Logical NOT makes it false. | !(A && B) is true. |

# Truth tables

We can describe a Boolean function by a truth table giving the values of the function for each combination of bits in the bit vectors.

A truth table has one column for each input variable, and one column for the output variable.

| a | b | f |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |



# NOT (~)

The NOT of a binary number is the value obtained by inverting all the bits in the binary number.

- We flip 0 to 1
- We flip 1 to 0

Truth table:

# AND (&)

The AND of a set of operands is 1 if and only if all of the operands are 1's.

Truth table:



# OR (I)

The OR of a set of operands is 1 if at least one of the operands is a 1.

Truth table:

# XOR (^)

The XOR of a set of operands is 1 if both the operands differ.

Truth table:



# Left shift: (<<)

A logical shift is a bitwise operation that shifts all the bits of its operand.

n << x is n with the bits shifted to the left by x places.

Note: the left shift performs a multiplication of n by $2^x$.

# Right shift: (>>)

A logical shift is a bitwise operation that shifts all the bits of its operand.

n >> x is n with the bits shifted to the right by x places.

Note: the right shift performs a floor division of n by $2^x$.



# Universal logic gates

A universal gate is a gate that can implement any Boolean function without any other gate.

The NAND and NOR gates are universal gates.

The NAND and NOR gates are economical and easier to fabricate and are the basic gates used in all IC digital logic families.

# Number system: Binary addition

Rules for binary addition

0 + 0 = 0

0 + 1 = 1

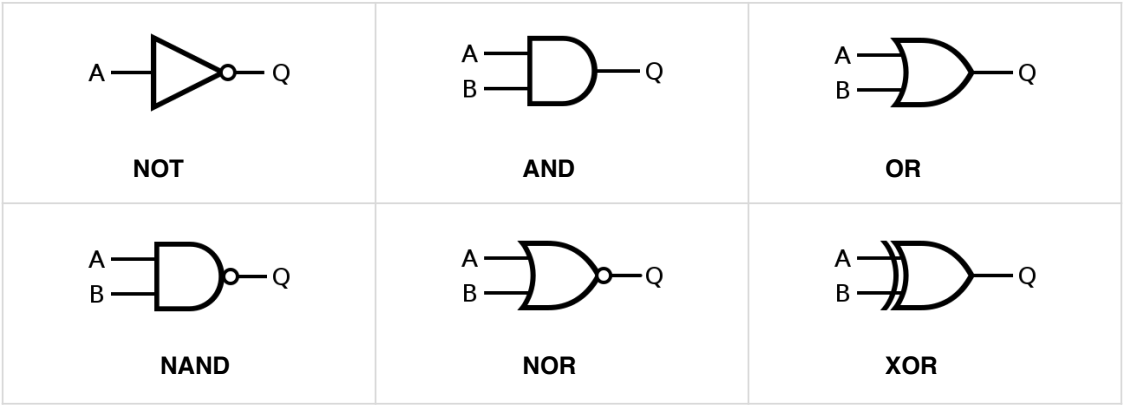
1 + 0 = 1

1 + 1 = 10

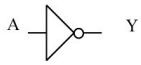
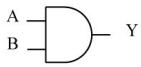
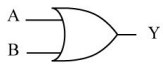
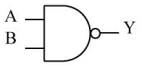
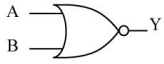
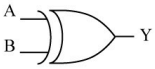
for 1 + 1, we write down a zero in the right-most column and carry over a one to the next column.



# Symbols for logic gates



# Symbols for logic gates

| | Truth Table | Gate Symbol | Boolean | Verilog |
|---|---|---|---|---|
| NOT | A \| Y<br>0 \| 1<br>1 \| 0 |  | $Y = \overline{A}$ | `assign y = ~a;` |
| AND | A B \| Y<br>0 0 \| 0<br>0 1 \| 0<br>1 0 \| 0<br>1 1 \| 1 |  | $Y = A \wedge B$ | `assign y = a & b;` |
| OR | A B \| Y<br>0 0 \| 0<br>0 1 \| 1<br>1 0 \| 1<br>1 1 \| 1 |  | $Y = A \vee B$ | `assign y = a \| b;` |
| NAND | A B \| Y<br>0 0 \| 1<br>0 1 \| 1<br>1 0 \| 1<br>1 1 \| 0 |  | $Y = \overline{A \wedge B}$ | `assign y = ~(a & b);` |
| NOR | A B \| Y<br>0 0 \| 1<br>0 1 \| 0<br>1 0 \| 0<br>1 1 \| 0 |  | $Y = \overline{A \vee B}$ | `assign y = ~(a \| b);` |
| XOR | A B \| Y<br>0 0 \| 0<br>0 1 \| 1<br>1 0 \| 1<br>1 1 \| 0 |  | $Y = A \oplus B$ | `assign y = a ^ b;` |



# Boolean functions with digital circuits

We can use a Karnaugh Map (K-Map) to determine a minimized Boolean expression of f from a truth table.

| a | b | c | f |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | **1** |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | **1** |
| 1 | 1 | 0 | **1** |
| 1 | 1 | 1 | **1** |

What is the function f ?

| ab \ c | 0 | 1 |
|---|---|---|
| 00 | | |
| 01 | | 1 |
| 11 | 1 | 1 |
| 10 | | 1 |

$f = (b \wedge c) \vee (a \wedge c) \vee (a \wedge b)$

This form is called a sum of products

# Adding bits

How do computers add bits?

We can build an adder function with two inputs and two outputs.

The half adder adds two input bits: a and b.

The output bits are: sum (s) and carry out (cout).

The carry out is an overflow.

We can get the Boolean expression for the sum (s) and carry out (cout) functions.

s = (¬a ∧ b) ∨ (a ∧ ¬b)

cout = a ∧ b

| inputs | | outputs | |
|---|---|---|---|
| **a** | **b** | **s** | **cout** |
| 0 | 0 | 0 | 0 |
| 0 | 1 | **1** | 0 |
| 1 | 0 | **1** | 0 |
| 1 | 1 | 0 | **1** |

half-adder truth table

| a \ b | 0 | 1 |
|---|---|---|
| 0 | | 1 |
| 1 | 1 | |

sum (s) k-map

| a \ b | 0 | 1 |
|---|---|---|
| 0 | | |
| 1 | | 1 |

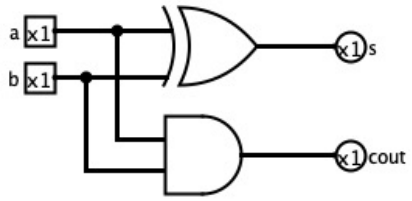
cout k-map



# Half adder

Using the Boolean expression for the sum (s) and carry out (cout) functions we create the circuit for the half adder.

The half-adder uses one XOR gate and one AND gate.



# Full adder

To add more than 2 bits we need a full-adder.

We can build a full-adder using two half adders.

The full adder adds three input bits: a, b and cin.

The output bits are: sum (s) and carry out (cout).

We can get the Boolean expression for the sum (s) and carry out (cout) functions.

s = ?

cout = ?

| inputs | | | outputs | |
|---|---|---|---|---|
| a | b | cin | s | cout |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

full-adder truth table



# Full adder

| ab \ cin | 0 | 1 |
|---|---|---|
| 00 | | 1 |
| 01 | 1 | |
| 11 | | 1 |
| 10 | 1 | |

sum (s) k-map

| ab \ cin | 0 | 1 |
|---|---|---|
| 00 | | |
| 01 | | 1 |
| 11 | 1 | 1 |
| 10 | | 1 |

cout k-map

| inputs | | | outputs | |
|---|---|---|---|---|
| a | b | cin | s | cout |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | **1** | 0 |
| 0 | 1 | 0 | **1** | 0 |
| 0 | 1 | 1 | 0 | **1** |
| 1 | 0 | 0 | **1** | 0 |
| 1 | 0 | 1 | 0 | **1** |
| 1 | 1 | 0 | 0 | **1** |
| 1 | 1 | 1 | **1** | **1** |

full-adder truth table

$s = (\neg a \wedge \neg b \wedge cin) \vee (\neg a \wedge b \wedge \neg cin) \vee (a \wedge \neg b \wedge \neg cin) \vee (a \wedge b \wedge cin)$

$s = a \oplus b \oplus cin$ (simplified)

$cout = (a \wedge b) \vee (a \wedge cin) \vee (b \wedge cin)$

# Full adder

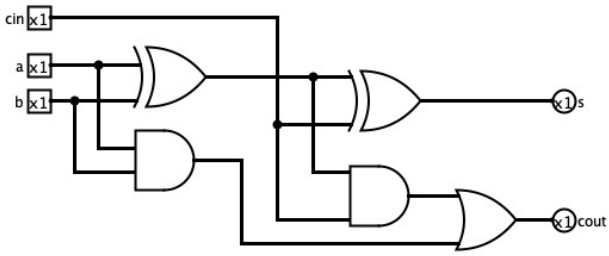
Using the Boolean expression for the sum (s) and carry out (cout) functions we create the circuit for the full adder.

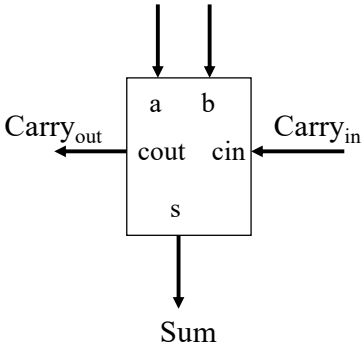
The full-adder uses XOR gates and AND gates.

sum = a ⊕ b ⊕ cin

cout = (a ∧ b) ∨ (a ∧ cin) ∨ (b ∧ cin)

Note: cout is also called a majority function!

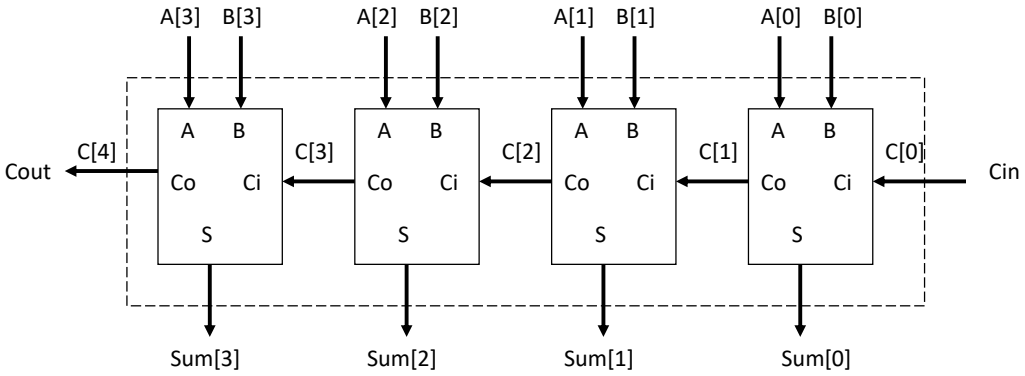


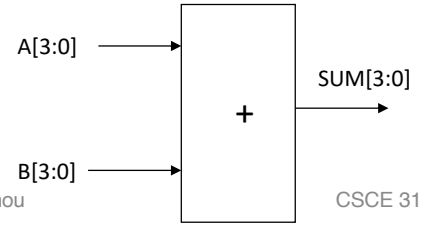






# 4 Bit Ripple Carry Adder

We can use a multiple 1-bit full adders to build a larger full adder



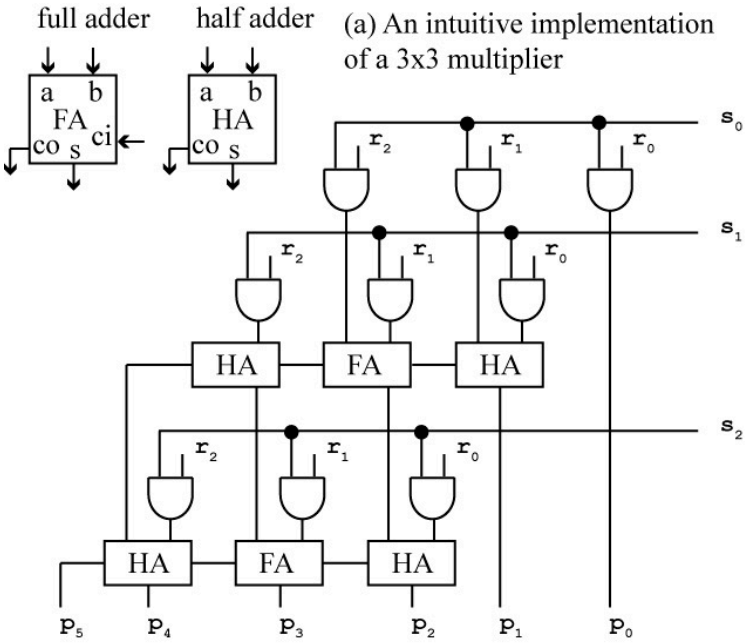
# Fixed-Point Multipliers

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| multiplicand | | | $r_2$ | $r_1$ | $r_0$ | | |
| multiplier | | x | $s_2$ | $s_1$ | $s_0$ | | |
| partial product | | | $s_0*r_2$ | $s_0*r_1$ | $s_0*r_0$ | | |
| | | $s_1*r_2$ | $s_1*r_1$ | $s_1*r_0$ | | | |
| | + | $s_2*r_2$ | $s_2*r_1$ | $s_2*r_0$ | | | |
| | $p_5$ | $p_4$ | $p_3$ | $p_2$ | $p_1$ | $p_0$ | product |

| Binary | Decimal |
|---|---|
| 1 1 1 | 7 |
| x 1 0 1 | x 5 |
| 1 1 1 | 35 |
| 0 0 0 | |
| + 1 1 1 | |
| 1 0 0 0 1 1 = 35 | |



# Array Multiplier Structure




(a) An intuitive implementation of a 3x3 multiplier

(b) A Verilog implementation of a 3x3 multiplication using the '*' operator

```
//3x3 multiplier
//using the '*' operator

module mult3x3 (a, b, y);

input [2:0] a,b;
output [5:0] y;

//do 3x3 multiply
assign y = a * b;

endmodule
```

# Selecting bits

A multiplexer (MUX) is a circuit that chooses one of two inputs based on a select input.

The MUX has three input bits: a, b, s.

The output bit is: out.

We can get the Boolean expression for the out functions.

out = ?

| inputs | | | outputs |
|---|---|---|---|
| s | a | b | out |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

MUX truth table

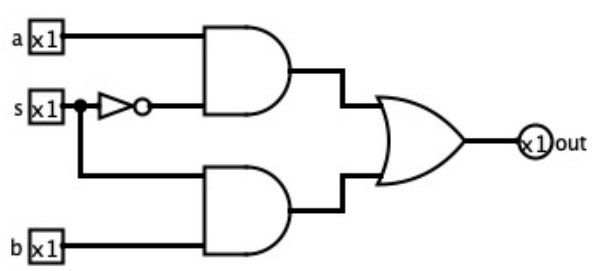


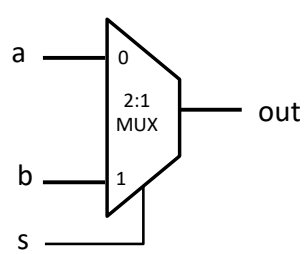
# 2:1 Multiplexer

The MUX is the logic-level version of the if/else statement.

```
if (s)
out = b;
else
out = a;
```

out = (a ∧ ~s) ∨ (b ∧ s)



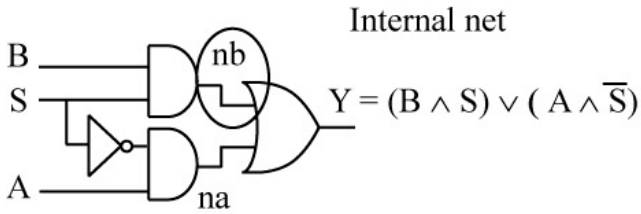
# 2:1 Multiplexer

Truth Table

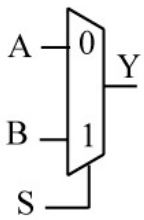
| S | B | A | Y |
|---|---|---|---|
| 0 | x | 0 | 0 |
| 0 | x | 1 | 1 |
| 1 | 0 | x | 0 |
| 1 | 1 | x | 1 |

x - don't care

Gate Schematic



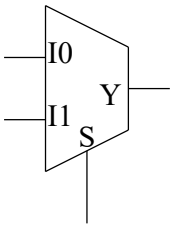

Symbol





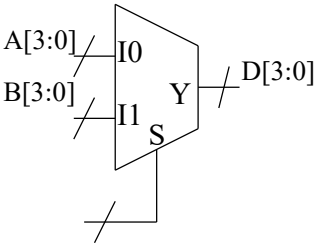

# Combinational Building Blocks

1 bit Multiplexer (2:1 MUX)




if S = 0, then Y = I0
if S = 1, then Y = I1

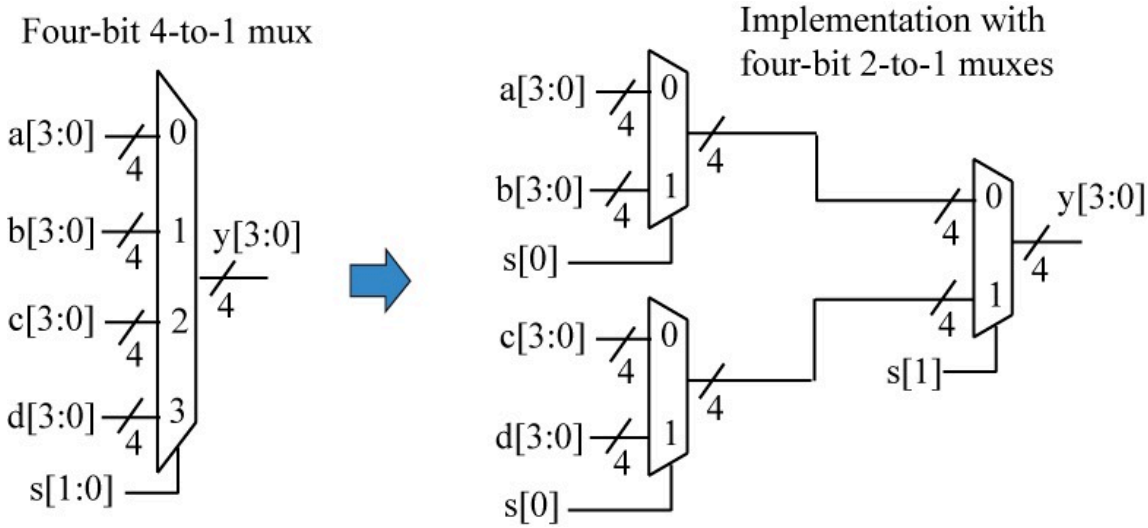
Y = I0 S' + I1 S




Muxes are often used to select groups of bits arranged in busses.

How many wires are in each bus ?

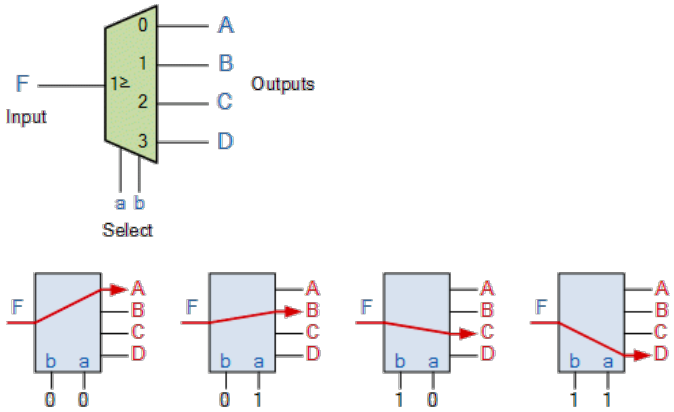
# Multiplexers






# Demultiplexer

A demultiplexer is a combinational logic circuit that switches one input line to one of the separate output lines.



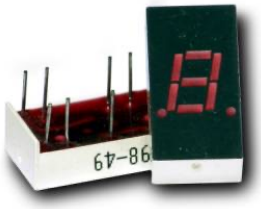

1-to-4 Channel De-multiplexer

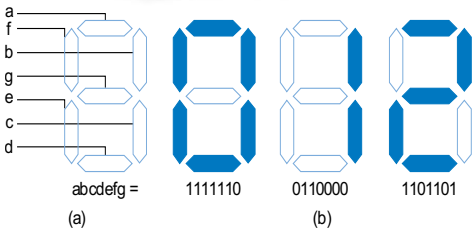
| inputs | | | outputs |
|---|---|---|---|
| F | a | b | out |
| 0 | 0 | 0 | A = 0 |
| 0 | 0 | 1 | B = 0 |
| 0 | 1 | 0 | C = 0 |
| 0 | 1 | 1 | D = 0 |
| 1 | 0 | 0 | A = 1 |
| 1 | 0 | 1 | B = 1 |
| 1 | 1 | 0 | C = 1 |
| 1 | 1 | 1 | D = 1 |

# Multiple-Output: BCD to 7-Segment Converter

TABLE 2-4 4-bit binary number to seven-segment display truth table

| w | x | y | z | a | b | c | d | e | f | g |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |






a = w'x'y'z' + w'x'yz' + w'x'yz + w'xy'z + w'xyz' + w'xyz + wx'y'z' + wx'y'z

b = w'x'y'z' + w'x'y'z + w'x'yz' + w'x'yz + w'xy'z' + w'xyz + wx'y'z' + wx'y'z

c = ?



# Decoder

A decoder transforms an n-bit binary input, by setting exactly on the $2^n$ bits outputs to 1.

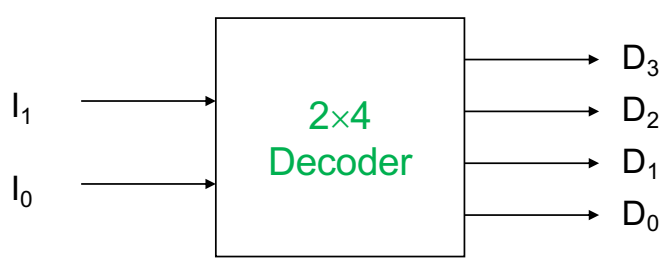
For an n-bit input, a decoder has $2^n$ outputs.

$n \times 2^n$ Device

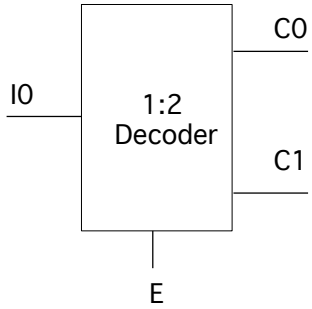
$n$ encoded inputs

$2^n$ decoded outputs

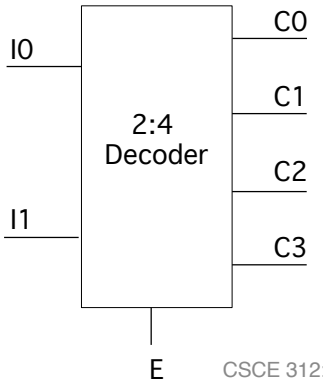
| $I_1$ | $I_0$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 |



# Decoders (with Enable)



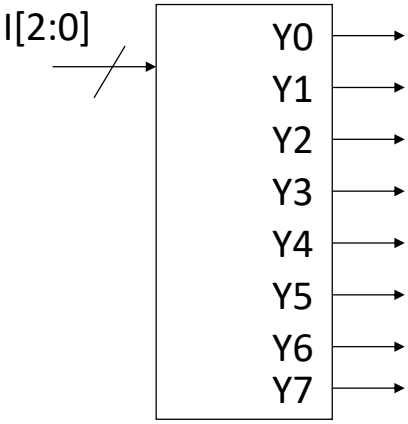

| E | I0 | C1 | C0 |
|---|---|---|---|
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |
| 0 | X | 0 | 0 |




| E | I1 | I0 | C3 | C2 | C1 | C0 |
|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| 0 | X | X | 0 | 0 | 0 | 0 |



# Decoder




if I= 000 then Y0=1 else Y0=0;
if I= 001 then Y1=1 else Y1=0;
if I= 010 then Y2=1 else Y2=0;
if I= 011 then Y3=1 else Y3=0;
if I= 100 then Y4=1 else Y4=0;
if I= 101 then Y5=1 else Y5=0;
if I= 110 then Y6=1 else Y6=0;
if I= 111 then Y7=1 else Y7=0;

# Summary

Combinational Logic Design Process

| Step | Description |
|---|---|
| Capture the function | Create a truth table or equations, to describe the desired behavior of the combinational logic. |
| Convert to equations | Create an equation for each output by ORing all the minterms for that output. Simplify the equations if desired. |
| Implement as a gate-based circuit | For each output, create a circuit corresponding to the output's equation. |