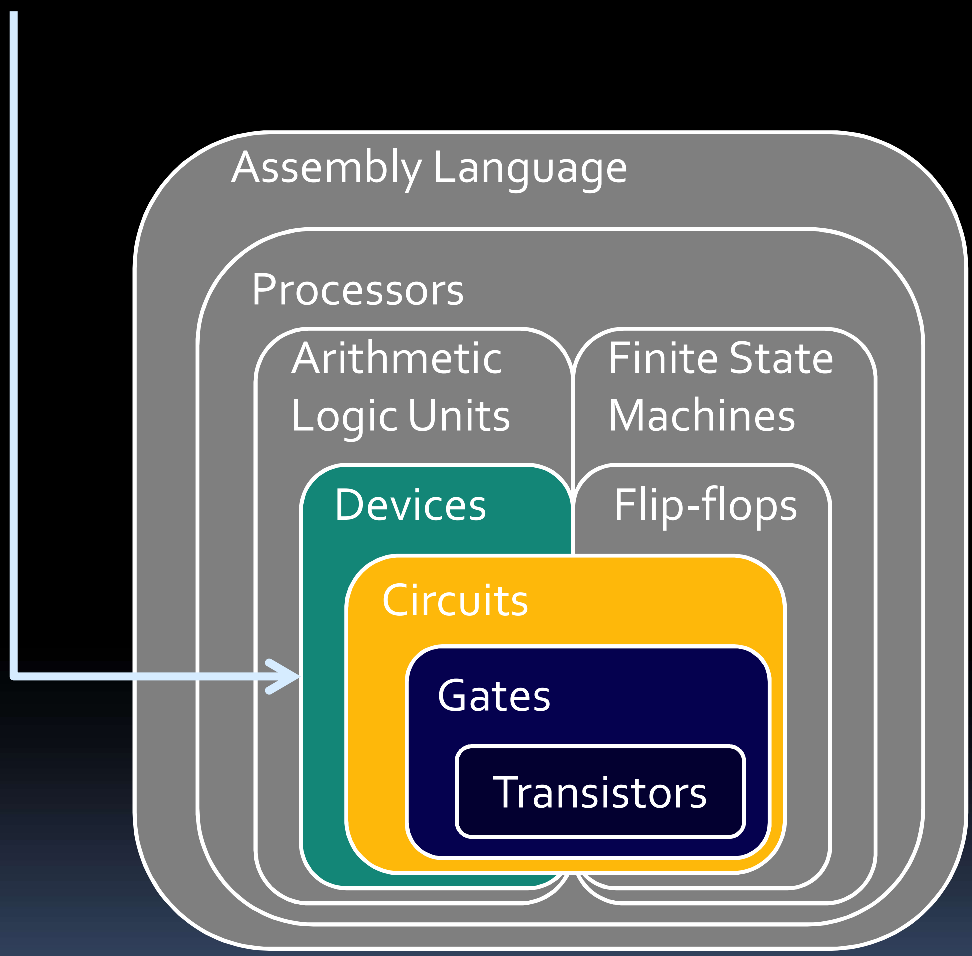
# Logical Devices

# We are here


Assembly Language
Processors
Arithmetic Logic Units
Finite State Machines
Devices
Flip-flops
Circuits
Gates
Transistors

# Building up from gates…

- Some common and more complex structures:
  - Multiplexers (aka *mux*)
  - Decoders
    - Seven-segment decoders
  - Adders (half and full)
  - Subtractors
  - Comparators

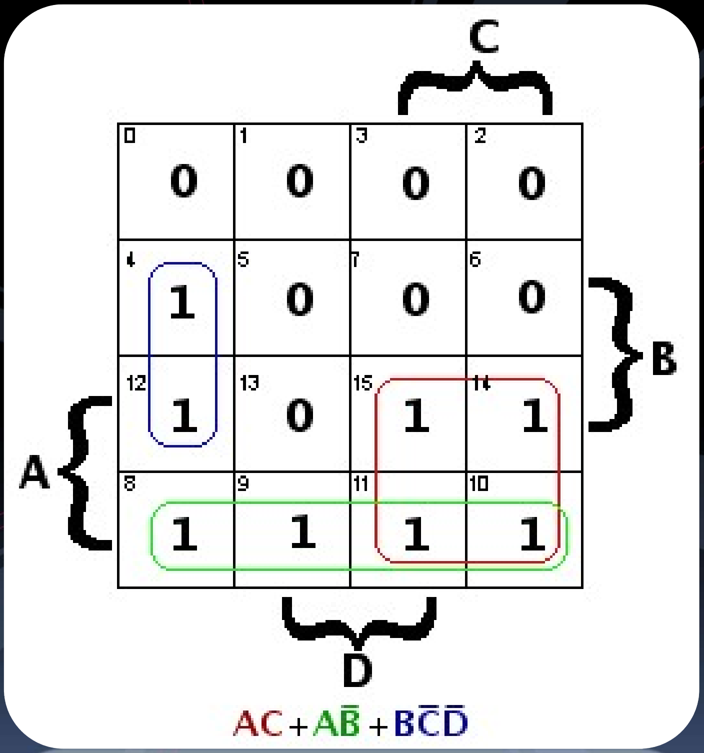
These are all combinational circuits

# Combinational Circuits

- *Combinational Circuits* are any circuits where the outputs rely strictly on the inputs.
  - Everything we've done so far and what we'll do today is all combinational logic.

- Another category is *sequential circuits* that we will learn in the next few weeks.

# More Karnaugh Maps



# Karnaugh map review

| | $\overline{B}\cdot\overline{C}$ | $\overline{B}\cdot C$ | $B\cdot C$ | $B\cdot\overline{C}$ |
|---|---|---|---|---|
| $\overline{A}$ | 0 | 0 | 1 | 0 |
| $A$ | 1 | 0 | 1 | 1 |

- K-maps provide an illustration of a circuit's minterms (or maxterms), and a guide to how neighbouring terms may be combined.

$$Y = \overline{A}\cdot B\cdot C + \boxed{A\cdot\overline{B}\cdot\overline{C} + A\cdot B\cdot\overline{C}} + A\cdot B\cdot C$$

# Karnaugh map review

| | $\overline{B}\cdot\overline{C}$ | $\overline{B}\cdot C$ | $B\cdot C$ | $B\cdot\overline{C}$ |
|---|---|---|---|---|
| $\overline{A}$ | 0 | 0 | 1 | 0 |
| $A$ | 1 | 0 | 1 | 1 |

- K-maps provide an illustration of a circuit's minterms (or maxterms), and a guide to how neighbouring terms may be combined.

$$Y = \overline{A}\cdot B\cdot C + A\cdot\overline{B}\cdot\overline{C} + A\cdot B\cdot\overline{C} + A\cdot B\cdot C$$
$$= B\cdot C + A\cdot\overline{C}$$

# Reminder on Reducing Circuits

- Eliminating variables in K-Maps by drawing larger (>1 element) rectangular groupings results in a circuit with a lower cost function.
- The resulting expression is still in sum-of-products form.
  - But, if simplified, it is *no longer in sum-of-minterms form*.
- Note: It is not only the number of gates that matters when reducing circuits, but also the number of inputs to each gate.

# K-Maps – Different Notations

## A 3-variables map example

| A | B | C | Y |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

$$Y = B \cdot C + A \cdot \overline{C}$$

Important!

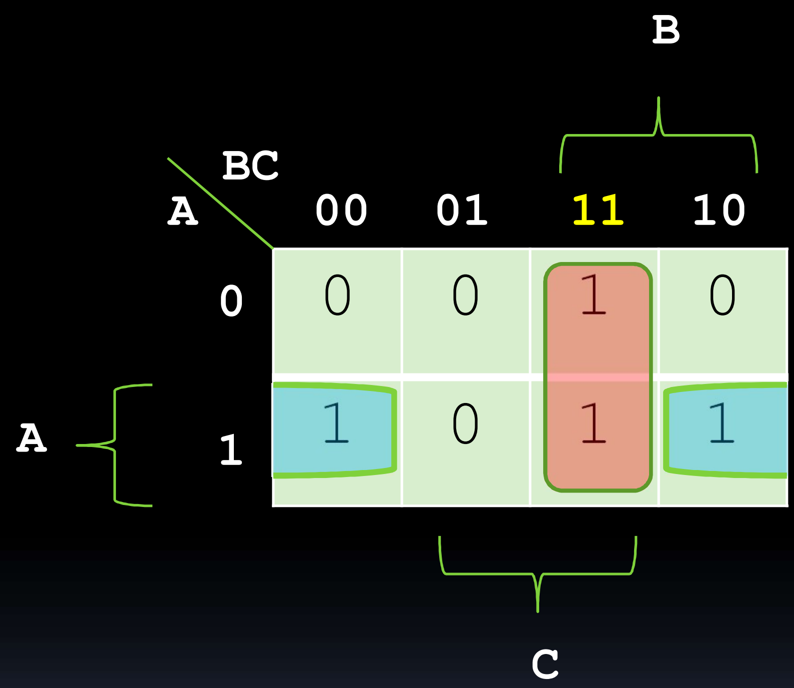
| A \ BC | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 |

Using either notation is fine!

Important!

| | $\overline{B}\,\overline{C}$ | $\overline{B}\,C$ | $BC$ | $B\,\overline{C}$ |
|---|---|---|---|---|
| $\overline{A}$ | 0 | 0 | 1 | 0 |
| $A$ | 1 | 0 | 1 | 1 |

# Helpful Hint



# More Examples w/ K-Maps

F1 =

B

| A \ BC | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |

A

C

F2 =

| A \ BC | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

F3 =

| A \ BC | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |

F4 =

| A \ BC | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 |

# More Examples w/ K-Maps

**F1 = 1**

B

| A \ BC | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |

A

C

**F2 = A**

| A \ BC | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

**F3 = C′**

| A \ BC | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |

**F4 = B+C**

| A \ BC | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 |

# Karnaugh map example

- Create a circuit with four inputs (A, B, C , D), and two outputs (X, Y):
  - The output X is high whenever two or more of the inputs are high.
  - The output Y is high when three or more of the inputs are high.

| A | B | C | D | X | Y |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | | |
| 0 | 0 | 0 | 1 | | |
| 0 | 0 | 1 | 0 | | |
| 0 | 0 | 1 | 1 | | |
| 0 | 1 | 0 | 0 | | |
| 0 | 1 | 0 | 1 | | |
| 0 | 1 | 1 | 0 | | |
| 0 | 1 | 1 | 1 | | |
| 1 | 0 | 0 | 0 | | |
| 1 | 0 | 0 | 1 | | |
| 1 | 0 | 1 | 0 | | |
| 1 | 0 | 1 | 1 | | |
| 1 | 1 | 0 | 0 | | |
| 1 | 1 | 0 | 1 | | |
| 1 | 1 | 1 | 0 | | |
| 1 | 1 | 1 | 1 | | |

# Karnaugh map example

- Create a circuit with four inputs (A, B, C , D), and two outputs (X, Y):
  - The output X is high whenever two or more of the inputs are high.
  - The output Y is high when three or more of the inputs are high.

| A | B | C | D | X | Y |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 |

# Karnaugh map example

X:

| | $\overline{C}\cdot\overline{D}$ | $\overline{C}\cdot D$ | $C\cdot D$ | $C\cdot\overline{D}$ |
|---|---|---|---|---|
| $\overline{A}\cdot\overline{B}$ | 0 | 0 | 1 | 0 |
| $\overline{A}\cdot B$ | 0 | 1 | 1 | 1 |
| $A\cdot B$ | 1 | 1 | 1 | 1 |
| $A\cdot\overline{B}$ | 0 | 1 | 1 | 1 |

X =

# Karnaugh map example

X:

| | $\bar{C}\cdot\bar{D}$ | $\bar{C}\cdot D$ | $C\cdot D$ | $C\cdot\bar{D}$ |
|---|---|---|---|---|
| $\bar{A}\cdot\bar{B}$ | 0 | 0 | 1 | 0 |
| $\bar{A}\cdot B$ | 0 | 1 | 1 | 1 |
| $A\cdot B$ | 1 | 1 | 1 | 1 |
| $A\cdot\bar{B}$ | 0 | 1 | 1 | 1 |

$$X = A\cdot B + C\cdot D + B\cdot D + B\cdot C + A\cdot D + A\cdot C$$

# Karnaugh map example

Y:

| | $\overline{C}\cdot\overline{D}$ | $\overline{C}\cdot D$ | $C\cdot D$ | $C\cdot\overline{D}$ |
|---|---|---|---|---|
| $\overline{A}\cdot\overline{B}$ | 0 | 0 | 0 | 0 |
| $\overline{A}\cdot B$ | 0 | 0 | 1 | 0 |
| $A\cdot B$ | 0 | 1 | 1 | 1 |
| $A\cdot\overline{B}$ | 0 | 0 | 1 | 0 |

$$Y = A\cdot B\cdot D + B\cdot C\cdot D + A\cdot B\cdot C + A\cdot C\cdot D$$

# Alternative for X: Maxterms

X:

| | C+D | $C+\overline{D}$ | $\overline{C}+\overline{D}$ | $\overline{C}+D$ |
|---|---|---|---|---|
| **A+B** | 0 | 0 | 1 | 0 |
| $A+\overline{B}$ | 0 | 1 | 1 | 1 |
| $\overline{A}+\overline{B}$ | 1 | 1 | 1 | 1 |
| $\overline{A}+B$ | 0 | 1 | 1 | 1 |

X =

# Alternative for X: Maxterms

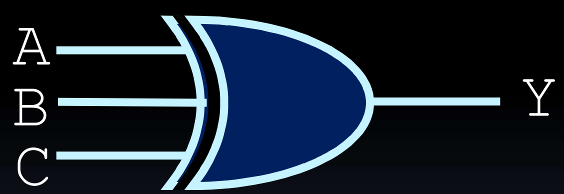
X:

| | C+D | $C+\overline{D}$ | $\overline{C}+\overline{D}$ | $\overline{C}+D$ |
|---|---|---|---|---|
| A+B | 0 | 0 | 1 | 0 |
| $A+\overline{B}$ | 0 | 1 | 1 | 1 |
| $\overline{A}+\overline{B}$ | 1 | 1 | 1 | 1 |
| $\overline{A}+B$ | 0 | 1 | 1 | 1 |

X = (A+C+D)·(B+C+D)·(A+B+C)·(A+B+D)

# Karnaugh map review

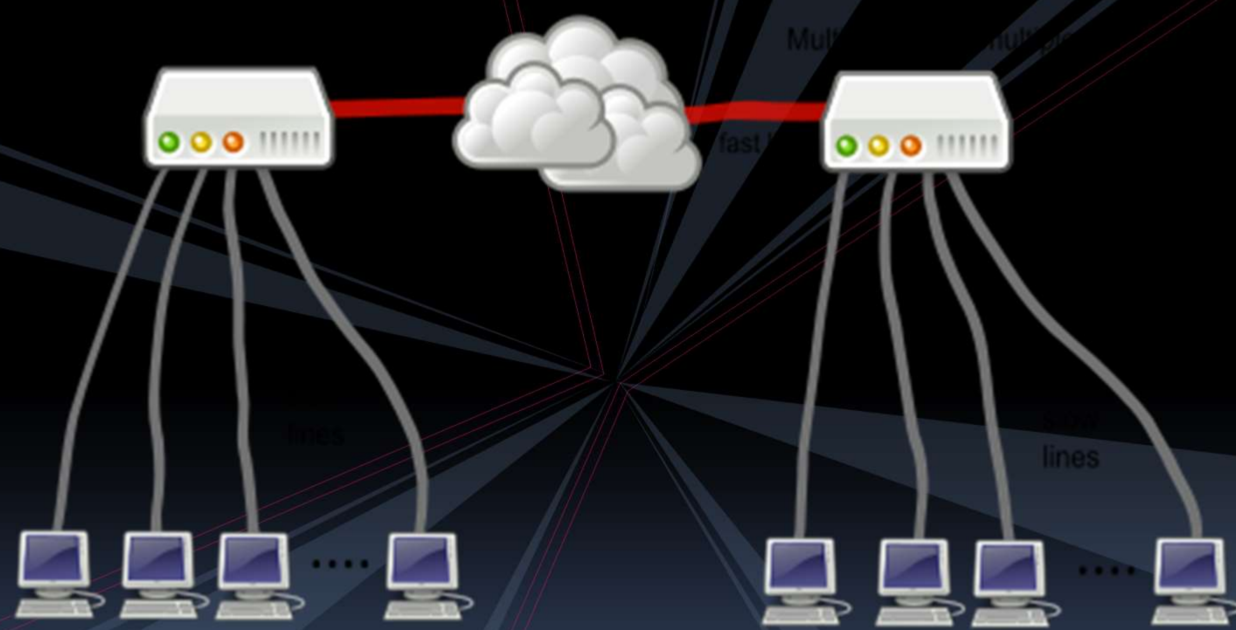
- Note: There are cases where no combinations are possible. K-maps cannot help in these cases.
- Example: Multi-input XOR gates.
  - Output is 1 iff odd number of inputs is 1.




| | $\overline{B} \cdot \overline{C}$ | $\overline{B} \cdot C$ | $B \cdot C$ | $B \cdot \overline{C}$ |
|---|---|---|---|---|
| $\overline{A}$ | 0 | 1 | 0 | 1 |
| $A$ | 1 | 0 | 1 | 0 |

$$Y = \overline{A} \cdot \overline{B} \cdot C + A \cdot \overline{B} \cdot \overline{C} + \overline{A} \cdot B \cdot \overline{C} + A \cdot B \cdot C$$

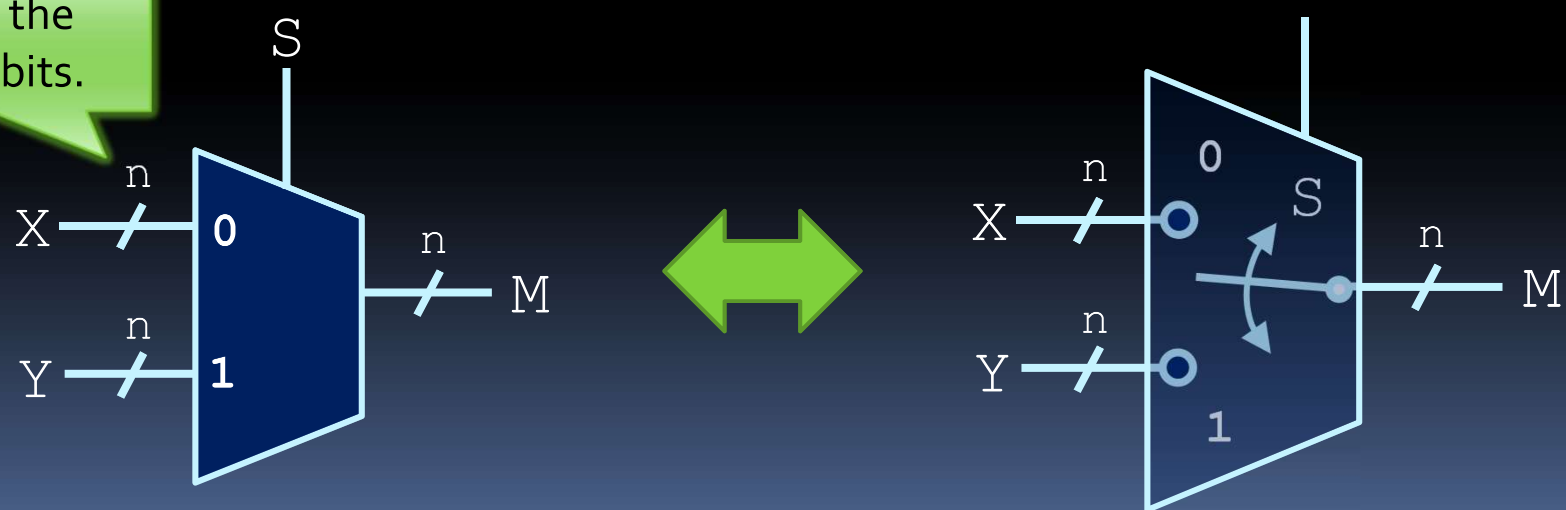
# Multiplexers



# Logic devices

- Certain structures are common to many circuits, and have block elements of their own.
  - e.g., Multiplexers (short form: mux)
  - <u>Behaviour:</u> Output is X if S is 0, and Y if S is 1:
    - S is the select input; X and Y are the data inputs.

n specifies the number of bits.



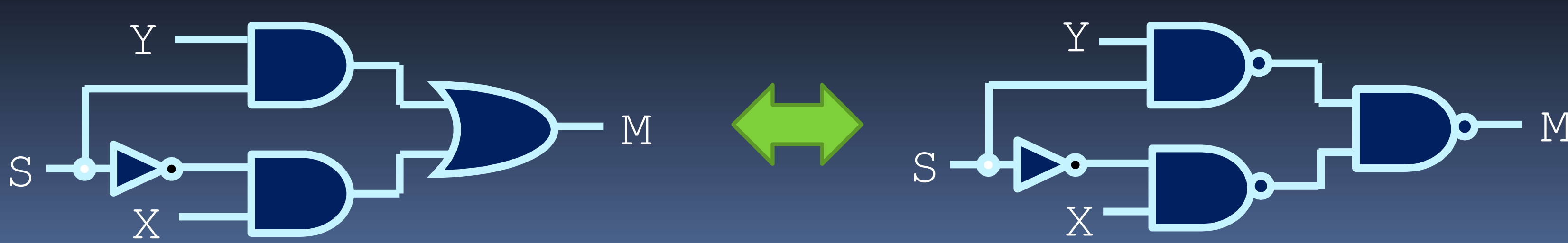

2-to-1 mux

# Multiplexer design

| X | Y | S | M |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

| | $\overline{Y}\cdot\overline{S}$ | $\overline{Y}\cdot S$ | $Y\cdot S$ | $Y\cdot\overline{S}$ |
|---|---|---|---|---|
| $\overline{X}$ | 0 | 0 | 1 | 0 |
| $X$ | 1 | 0 | 1 | 1 |

$$M = Y\cdot S + X\cdot\overline{S}$$



# Multiplexers in Verilog

- A four-input multiplexer, created with gates.
  - Note that four input lines require two select bits to choose the output.

```
module mux_gates( select, d, q );

input[1:0] select;
input[3:0] d;
output     q;

wire       q, q1, q2, q3, q4;
wire       not_s0, not_s1;
wire[1:0]  select;
wire[3:0]  d;

not n1( not_s0, select[0] );
not n2( not_s1, select[1] );

and a1( q1, not_s0, not_s1, d[0]  );
and a2( q2, select[0], not_s1, d[1]  );
and a3( q3, not_s0, select[1], d[2]  );
and a4( q4, select[0], select[1], d[3]  );

or o1( q, q1, q2, q3, q4 );

endmodule
```

# Multiplexers in Verilog

- Another four-input mux, this time implemented using boolean notation

In Lab2 you need to implement a 4-to-1 mux differently, using hierarchical design.

```
module mux_logic( select, d, q );

input[1:0] select;
input[3:0] d;
output q;

wire q;
wire[1:0] select;
wire[3:0] d;

assign q = (~select[1]&~select[0]) &d[0]  |
           (~select[1]&select[0])  &d[1]  |
           (select[1]&~select[0])  &d[2]  |
           (select[1]&select[0])   &d[3];

endmodule
```

specify input

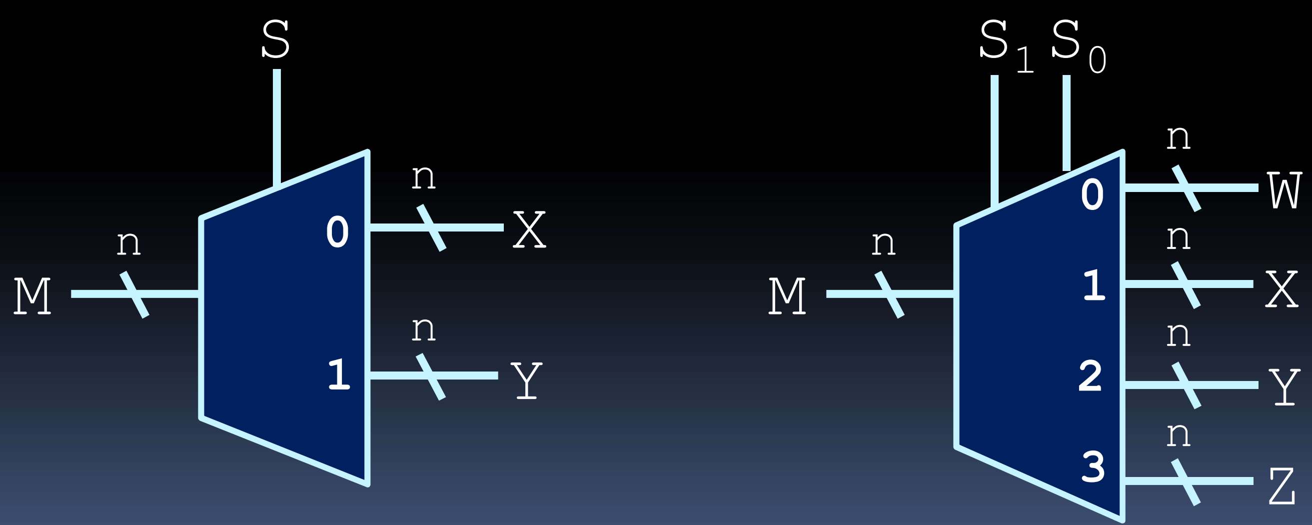
specify value

# Multiplexer uses

- Muxes are very useful whenever you need to select from multiple input values.
  - Example: surveillance video monitors, digital cable boxes, routers.

# Demultiplexers

- Related to decoders: demultiplexers.
  - Does multiplexer operation, in reverse.
  - <u>Example:</u> modems receiving Internet data.



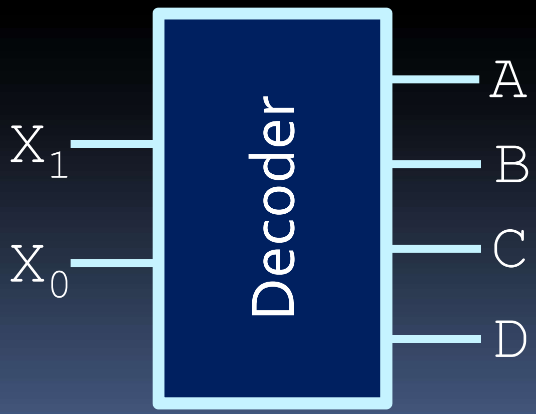
# Mux + Demux


Source:
https://upload.wikimedia.org/wikipedia/commons/e/eo/Telephony_multiplexer_system.gif

# Decoders

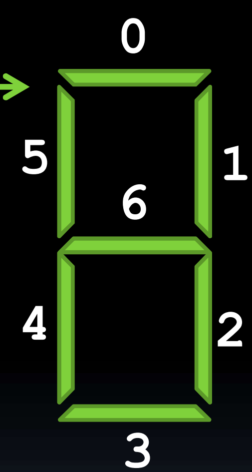
# Decoders

- Decoders are essentially translators.
  - Translate from the output of one circuit to the input of another.
  - Think of them as providing a mapping between 2 different encodings!
- <u>Example:</u> Binary signal splitter
  - Activates one of four output lines, based on a two-digit binary number.

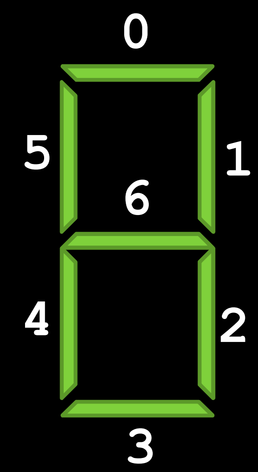


# 7-segment decoder

- Common and useful decoder application.
  - Translate from a 4-digit binary number to the seven segments of a digital display.
  - Each output segment has a particular logic that defines it.
  - <u>Example:</u> Segment `0`
    - Activate for values: `0`, `2`, `3`, `5`, `6`, `7`, `8`, `9`.
    - In binary: `0000`, `0010`, `0011`, `0101`, `0110`, `0111`, `1000`, `1001`.
  - <u>First step:</u> Build the truth table and K-map.



# 7-segment decoder



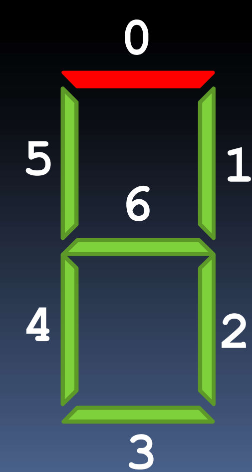

- These segments are "active-low", meaning that setting it low turns it on.
- Example: Displaying digits 0-9
  - Assume input is a 4-digit binary number
  - Segment 0 (top segment) is low whenever the input values are 0000, 0010, 0011, 0101, 0110, 0111, 1000 or 1001, and high whenever input number is 0001 or 0100.
  - This create a truth table and map like the following....

# 7-segment decoder

| $X_3$ | $X_2$ | $X_1$ | $X_0$ | $HEX_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |

| | $\overline{X}_1\cdot\overline{X}_0$ | $\overline{X}_1\cdot X_0$ | $X_1\cdot X_0$ | $X_1\cdot\overline{X}_0$ |
|---|---|---|---|---|
| $\overline{X}_3\cdot\overline{X}_2$ | 0 | 1 | 0 | 0 |
| $\overline{X}_3\cdot X_2$ | 1 | 0 | 0 | 0 |
| $X_3\cdot X_2$ | x | x | x | x |
| $X_3\cdot\overline{X}_2$ | 0 | 0 | x | x |

- $HEX0 = \overline{X}_3\cdot\overline{X}_2\cdot\overline{X}_1\cdot X_0 + \overline{X}_3\cdot X_2\cdot\overline{X}_1\cdot\overline{X}_0$
- But wait…what about input values 1010 to 1111?



# “Don’t care” values

- Input values that will never happen or are not meaningful in a given design, and so their output values do not have to be defined.
  - Recorded as ‘X’ in truth-tables and K-Maps.

- In the K-maps we can think of these “don’t care” values as either 0 or 1 depending on what helps us simplify our circuit.
  - Note: you do NOT replace the X with a 0 or 1, you just include it in a grouping as needed.

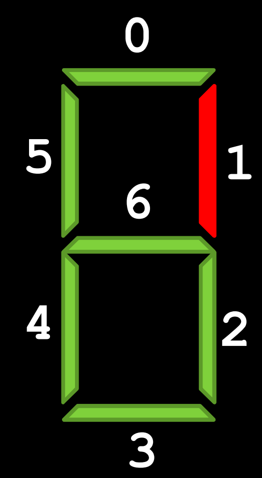
# “Don’t care” values

- New equation for HEX0:

| | $\overline{x_1} \cdot \overline{x_0}$ | $\overline{x_1} \cdot x_0$ | $x_1 \cdot x_0$ | $x_1 \cdot \overline{x_0}$ |
|---|---|---|---|---|
| $\overline{x_3} \cdot \overline{x_2}$ | 0 | 1 | 0 | 0 |
| $\overline{x_3} \cdot x_2$ | 1 | 0 | 0 | 0 |
| $x_3 \cdot x_2$ | x | x | x | x |
| $x_3 \cdot \overline{x_2}$ | 0 | 0 | x | x |

$$\mathrm{HEX0} = \overline{x_3} \cdot \overline{x_2} \cdot \overline{x_1} \cdot x_0 + x_2 \cdot \overline{x_1} \cdot \overline{x_0}$$

# Again for segment 1

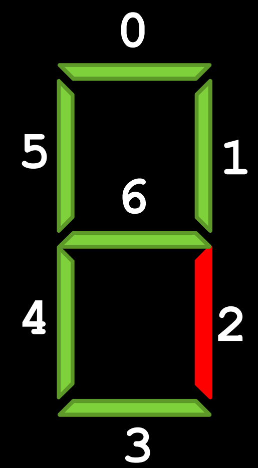



| $x_3$ | $x_2$ | $x_1$ | $x_0$ | $HEX_1$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |

| | $\bar{x}_1 \cdot \bar{x}_0$ | $\bar{x}_1 \cdot x_0$ | $x_1 \cdot x_0$ | $x_1 \cdot \bar{x}_0$ |
|---|---|---|---|---|
| $\bar{x}_3 \cdot \bar{x}_2$ | 0 | 0 | 0 | 0 |
| $\bar{x}_3 \cdot x_2$ | 0 | 1 | 0 | 1 |
| $x_3 \cdot x_2$ | x | x | x | x |
| $x_3 \cdot \bar{x}_2$ | 0 | 0 | x | x |

$$HEX1 = x_2 \cdot \bar{x}_1 \cdot x_0 + x_2 \cdot x_1 \cdot \bar{x}_0$$

# Again for segment 2




| $x_3$ | $x_2$ | $x_1$ | $x_0$ | $HEX_2$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |

| | $\overline{x}_1 \cdot \overline{x}_0$ | $\overline{x}_1 \cdot x_0$ | $x_1 \cdot x_0$ | $x_1 \cdot \overline{x}_0$ |
|---|---|---|---|---|
| $\overline{x}_3 \cdot \overline{x}_2$ | 0 | 0 | 0 | 1 |
| $\overline{x}_3 \cdot x_2$ | 0 | 0 | 0 | 0 |
| $x_3 \cdot x_2$ | x | x | x | x |
| $x_3 \cdot \overline{x}_2$ | 0 | 0 | x | x |

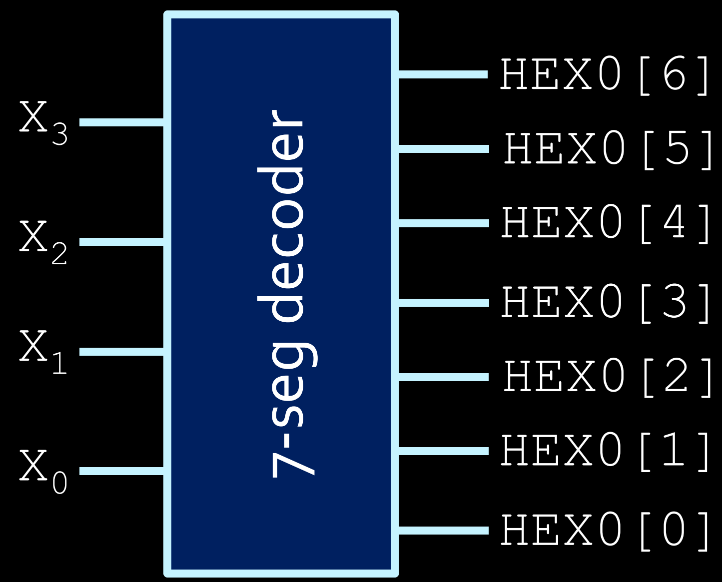
$$HEX2 = \overline{x}_2 \cdot x_1 \cdot \overline{x}_0$$

# Verilog for 7-segment display

```
//Seven segment decoder for BCD inputs from 0 to 9
module seven_seg_decoder(S,HEX0);

input [3:0]S;
output [6:0]HEX0;

assign HEX0[0]=(~S[3]&~S[2]&~S[1]&S[0])|(S[2]&~S[1]&~S[0]);
assign HEX0[1]=(S[2]&~S[1]&S[0])|(S[2]&S[1]&~S[0]);
assign HEX0[2]=~S[2]&S[1]&~S[0];
...  // remaining equations left as an exercise

endmodule
```

# The final 7-seg decoder

- Decoders all look the same, except for the inputs and outputs.
- Unlike other devices, the implementation differs from decoder to decoder.



# Another “don’t care” example

*(not related to decoders)*

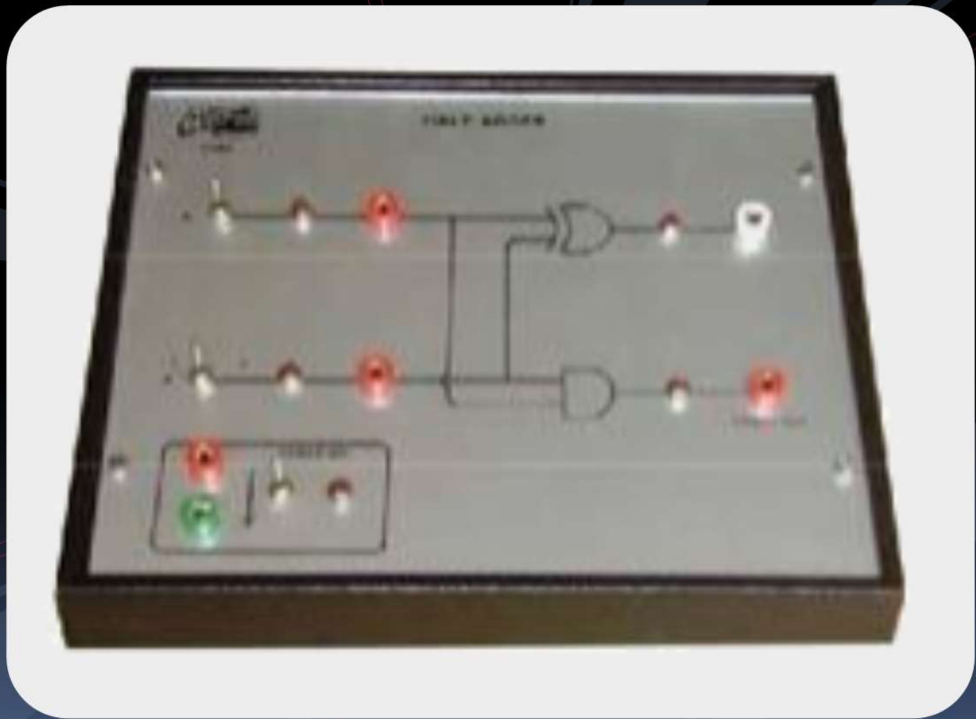
- Climate control fan:
  - The fan should turn on (F) if the temperature is hot (H) or if the temperature is cold (C), depending on whether the unit is set to A/C or heating (A).

| H | C | A | F |
|---|---|---|---|
| 0 | 0 | 0 | |
| 0 | 0 | 1 | |
| 0 | 1 | 0 | |
| 0 | 1 | 1 | |
| 1 | 0 | 0 | |
| 1 | 0 | 1 | |
| 1 | 1 | 0 | |
| 1 | 1 | 1 | |

| | $\overline{H}\cdot\overline{C}$ | $\overline{H}\cdot C$ | $H\cdot C$ | $H\cdot\overline{C}$ |
|---|---|---|---|---|
| $\overline{A}$ | 0 | 1 | X | 0 |
| $A$ | 0 | 0 | X | 1 |

$$F = A\cdot H + \overline{A}\cdot C$$

# Adder circuits



# Adders

- Also known as binary adders.
  - Small circuit devices that add two digits together.
  - Combined together to create iterative combinational circuits.
- Types of adders:
  - Half adders (HA)
  - Full adders (FA)
  - Ripple Carry Adder

# Review of Binary Math

- Each digit of a decimal number represents a power of 10:

$$\mathbf{258} = 2\times10^2 + 5\times10^1 + 8\times10^0$$

- Each digit of a binary number represents a power of 2:

$$\mathbf{01101_2} = 0\times2^4 + 1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0$$
$$\mathbf{= 13_{10}}$$

# Decimal to Binary Conversion

- Let's say I give you number 11 in decimal. How would you represent this in binary?
  - Keep dividing by 2 and write down the remainders!

Use the quotient from previous row.

11 in decimal is 1011 in binary!

| Number | Quotient = Number / 2 | Remainder = Number % 2 | |
|---|---|---|---|
| 11 | | | |
| | | | |
| | | | |
| | | | |

# Decimal to Binary Conversion

- Let's say I give you number 11 in decimal. How would you represent this in binary?
  - Keep dividing by 2 and write down the remainders!

11 in decimal is 1011 in binary!

Use the quotient from previous row.

| Number | Quotient = Number / 2 | Remainder = Number % 2 | |
|---|---|---|---|
| 11 | 5 | 1 | Least Significant Bit |
| 5 | 2 | 1 | |
| 2 | 1 | 0 | |
| 1 | 0 | 1 | Most Significant Bit |

# Hexadecimal Numbers

- Base 16 numbers, where valid values are:
  - 0 to 9 as in decimal, and
  - 10 is A
  - 11 is B
  - ..
  - 15 is F
- Writing a binary number in hex(-adecimal):
  - 0000010111111010 = 0000 0101 1111 1010 = 0x05fa
  - In Verilog (more about this in the handout of Lab 3):
    - 16’b0000_0101_1111_1010
    - 16’h05FA (16’h05fa is fine too)

Hex numbers are typically expressed as 0x____

# Unsigned binary addition

- 27 + 53

27 = 00011011

53 = 00110101

```
  1 1 1 1 1 1
  00011011
 +00110101
 ---------
  01010000
```

01010000

$80_{10}$

# Unsigned binary addition

- 27 + 53

  27 = 00011011

  53 = 00110101

```
   1 1 1 1 1 1
  00011011
 +00110101
 ---------
  01010000
```

$80_{10}$

**01010000**

- 95 + 181

  01011111

  +10110101

```
   1 1 1 1 1 1 1 1
   01011111
  +10110101
  ---------
  100010100
```

carry bit

$20_{10}$ ??

**00010100**

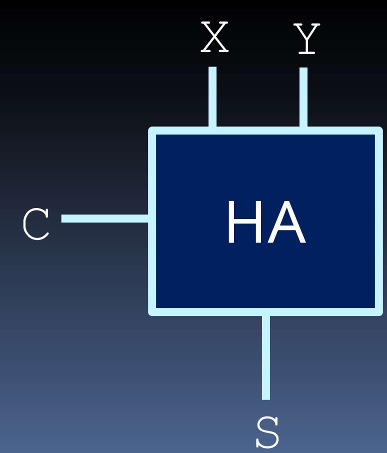
With 8 bits we can only represent unsigned numbers 0 to 255 !

# Half Adders

- A 2-input, 1-bit width binary adder that performs the following computations:

| | | | | |
|---|---|---|---|---|
| X | 0 | 0 | 1 | 1 |
| +Y | +0 | +1 | +0 | +1 |
| CS | 00 | 01 | 01 | 10 |

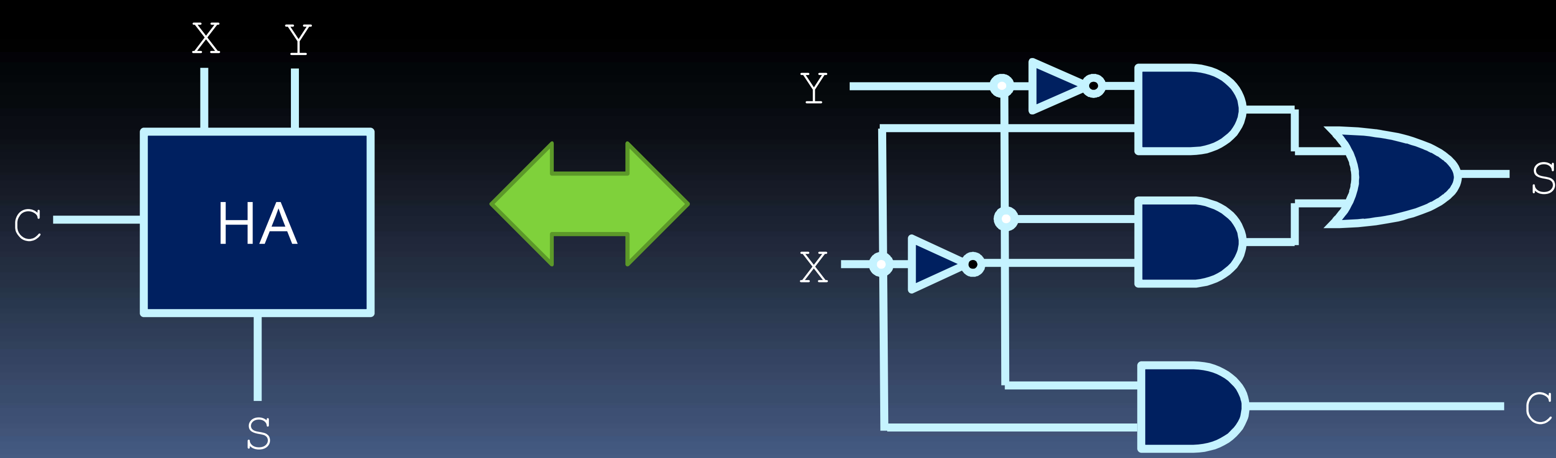
- A half adder adds two bits to produce a two-bit sum.
- The sum is expressed as a sum bit S and a carry bit C.



# Half Adder Implementation

- Equations and circuits for half adder units are easy to define (even without Karnaugh maps)

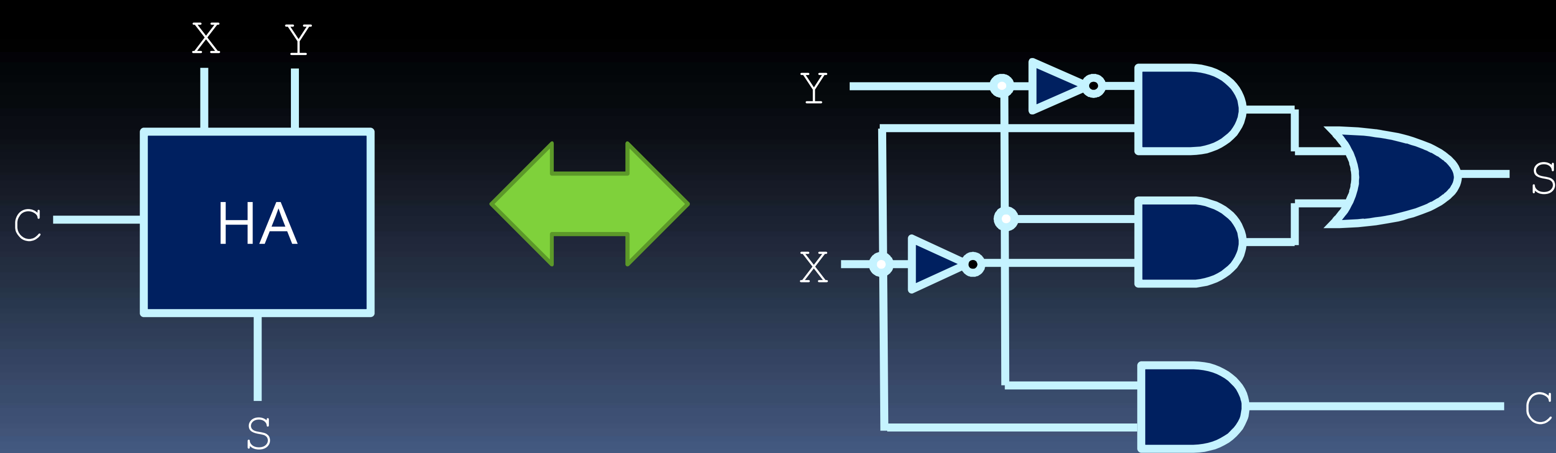
C = S =



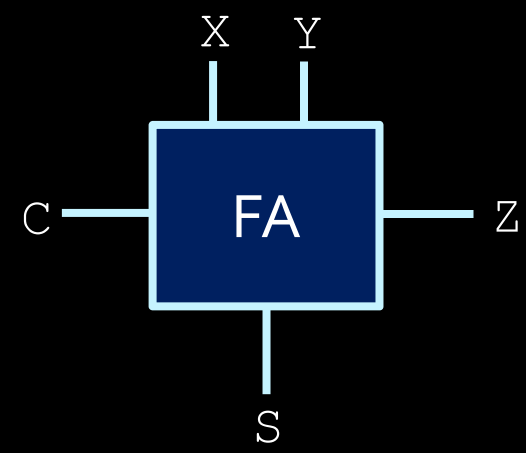
# Half Adder Implementation

- Equations and circuits for half adder units are easy to define (even without Karnaugh maps)

$$C = X \cdot Y \qquad S = X \cdot \overline{Y} + \overline{X} \cdot Y$$
$$= X \oplus Y$$



# Full Adders




- Similar to half-adders, but with another input Z, which represents a carry-in bit.
  - C and Z are sometimes labeled as $C_{out}$ and $C_{in}$.
- When Z is 0, the unit behaves exactly like a half adder.
- When Z is 1:

| | | | | |
|---|---|---|---|---|
| X | 0 | 0 | 1 | 1 |
| +Y | +0 | +1 | +0 | +1 |
| +Z | +1 | +1 | +1 | +1 |
| CS | 01 | 10 | 10 | 11 |

# Full Adder Design

| X | Y | Z | C | S |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

| C | $\bar{Y}\cdot\bar{Z}$ | $\bar{Y}\cdot Z$ | $Y\cdot Z$ | $Y\cdot\bar{Z}$ |
|---|---|---|---|---|
| $\bar{X}$ | 0 | 0 | 1 | 0 |
| X | 0 | 1 | 1 | 1 |

| S | $\bar{Y}\cdot\bar{Z}$ | $\bar{Y}\cdot Z$ | $Y\cdot Z$ | $Y\cdot\bar{Z}$ |
|---|---|---|---|---|
| $\bar{X}$ | 0 | 1 | 0 | 1 |
| X | 1 | 0 | 1 | 0 |

$$C = X\cdot Y + X\cdot Z + Y\cdot Z$$

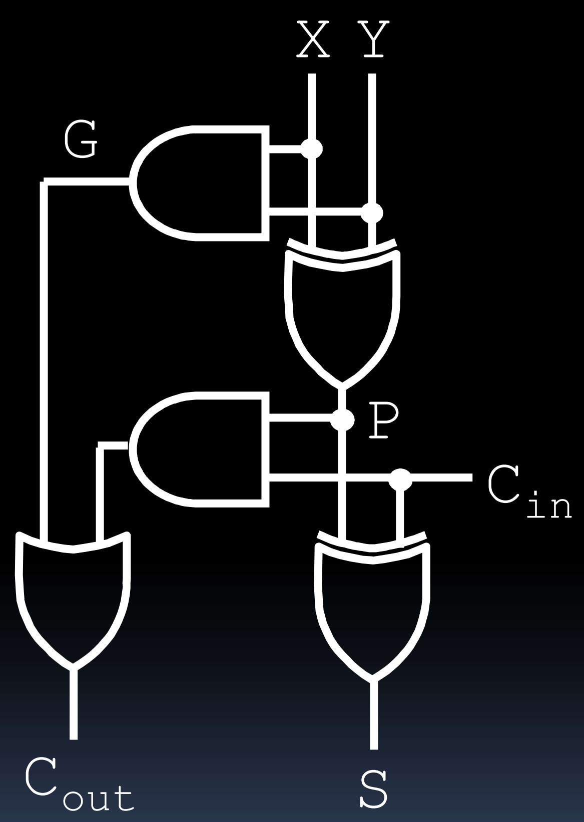
$$S = X \oplus Y \oplus Z$$

# Full Adder Design

- The $C$ term can also be rewritten as:

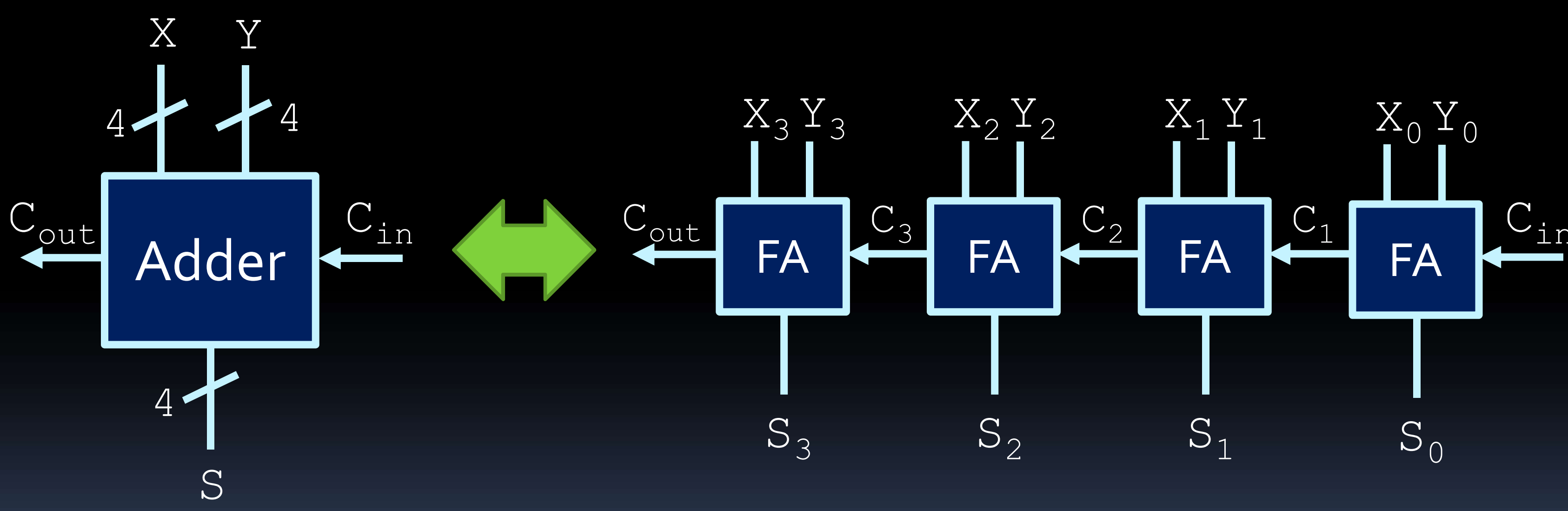
$$C = X \cdot Y + (X \oplus Y) \cdot Z$$

- Two terms come from this:
  - $X \cdot Y$ = carry generate ($G$).
  - $X \oplus Y$ = carry propagate ($P$).
- Results in this circuit →



# Ripple-Carry Binary Adder

- Full adder units are chained together in order to perform operations on signal vectors.



# Adders in Verilog

- Verilog code that implements a half adder unit.

```
module half_adder (in_x, in_y, out_sum, out_carry);

input  in_x;
input  in_y;
output out_sum;
output out_carry;

assign out_sum = in_x^in_y;
assign out_carry = in_x&in_y;

endmodule
```

# Adders in Verilog

- Verilog code that implements a full adder unit.

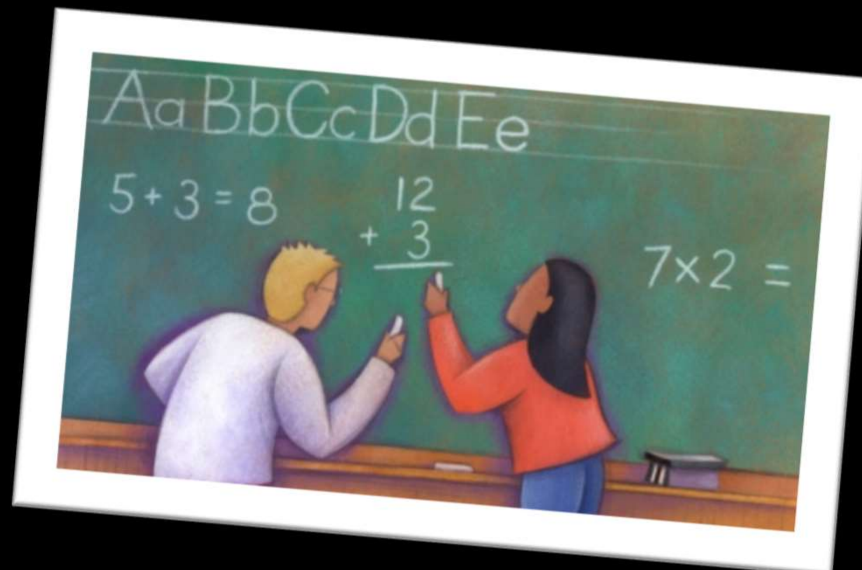
```
module full_adder(sum,cout,a,b,cin);

output sum, cout;
input a, b, cin;

assign sum = a^b^cin;
assign cout = (a&b)|(cin&(a^b));

endmodule
```

```
module full_adder(sum,cout,a,b,cin);

output sum, cout;
input a,b,cin;

assign {cout,sum}=a+b+cin;

endmodule
```

# The role of $C_{in}$

- Why can't we just have a half-adder for the smallest (right-most) bit?
- We could, if we were only interested in addition. But the last bit allows us to do subtraction as well!
  - Time for a little fun with subtraction!

# Fun with Subtraction




1. Find a partner.
2. Have each person choose a five-digit binary number.
3. Take the smaller number, and invert all the digits.
4. Add this inverted number to the larger one.
5. Add one to the result.
6. Check what the result is…

# Subtractors

- Subtractors are an extension of adders.
  - Basically, perform addition on a negative number.
- Before we can do subtraction, need to understand negative binary numbers.
- Two types:
  - Unsigned = a separate bit exists for the sign; data bits store the positive version of the number.
  - Signed = all bits are used to store a 2's complement negative number.
    - More common, and what we use for this course.

# Two's complement

- First step: getting 1's complement:
  - Given number $X$ with $n$ bits, take $(2^n-1)-X$
  - Negates each individual bit (bitwise NOT).

```
01001101   →   10110010
11111111   →   00000000
```

- 2's complement = (1's complement + 1)

```
01001101   →   10110011
11111111   →   00000001
```

Know this!

- Note: Adding a 2's complement number to the original number produces a result of zero.

# Signed subtraction

- Negative numbers are generally stored in 2's complement notation.
  - Reminder: 1's complement → bits are the bitwise NOT of the equivalent positive value.
  - 2's complement → one more than 1's complement value; results in zero when added to equivalent positive value.
    - Subtraction can then be performed by using the binary adder circuit with negative numbers.

# Signed representations

| Decimal | Unsigned | Signed 2’s |
|---|---|---|
| 7 | 111 | --- |
| 6 | 110 | --- |
| 5 | 101 | --- |
| 4 | 100 | --- |
| 3 | 011 | 011 |
| 2 | 010 | 010 |
| 1 | 001 | 001 |
| 0 | 000 | 000 |
| -1 | --- | 111 |
| -2 | --- | 110 |
| -3 | --- | 101 |
| -4 | --- | 100 |

# Rules about signed numbers

- When thinking of signed binary numbers, there are a few useful rules to remember:
    - The largest positive binary number is a zero followed by all ones.
    - The binary value for $-1$ has ones in all the digits.
    - The most negative binary number is a one followed by all zeroes.

- There are $2^n$ possible values that can be stored in an $n$-digit binary number.
    - $2^{n-1}$ are negative, $2^{n-1}-1$ are positive, and one is zero.
    - For example, given an 8-bit binary number:
        - There are 256 possible values
        - One of those values is zero
        - 128 are negative values (11111111 to 10000000)
        - 127 are positive values (00000001 to 01111111)

-1 to -128

1 to 127

# Practice 2’s complement!

- Assume 4-bit signed representation, write the following decimal numbers in binary:
  - 2 => 0010
  - -1 => 1111
  - 0 => 0000
  - 8 => Not possible to represent in 4 digits!
  - -8 => 1000
- What is max positive number? => 7 (or $2^{4-1} - 1$)
- What is min negative number? => -8 (or $-2^{4-1}$)

# At the core of subtraction

- Subtraction of a number is simply the addition of its negative value.
  - Where the negative value is found using the 2's complement process.

  - $7 - 3 = 7 + (-3)$
  - $-3 - 2 = -3 + (-2)$

# Signed Subtraction example

- 7 − 3

```
   0111
  -0011
  -----
   0111
  +1101
  -----
  10100
```

discarded: 1

$0100 = 4_{10}$

- -3 − 2

```
   1101
  -0010
  -----
   1101
  +1110
  -----
  11011
```

discarded: 1

$1011 = -5_{10}$

# What about bigger numbers?

- 53 – 27

```
   00110101
  -00011011
```

```
             00110101
discarded   +11100101
            100011010
```

$00011010 = 26_{10}$

- 27 – 53

```
   00011011
  -00110101
```

```
             00011011
discarded   +11001011
            011100110
```

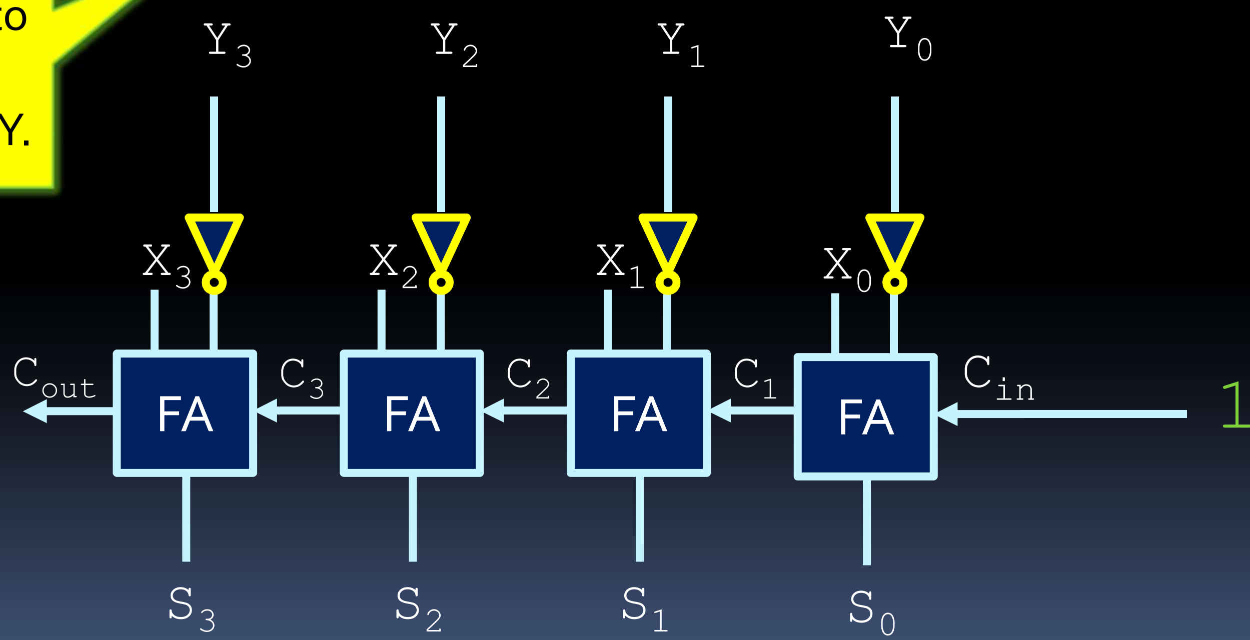
$11100110 = -26_{10}$

# Subtraction circuit

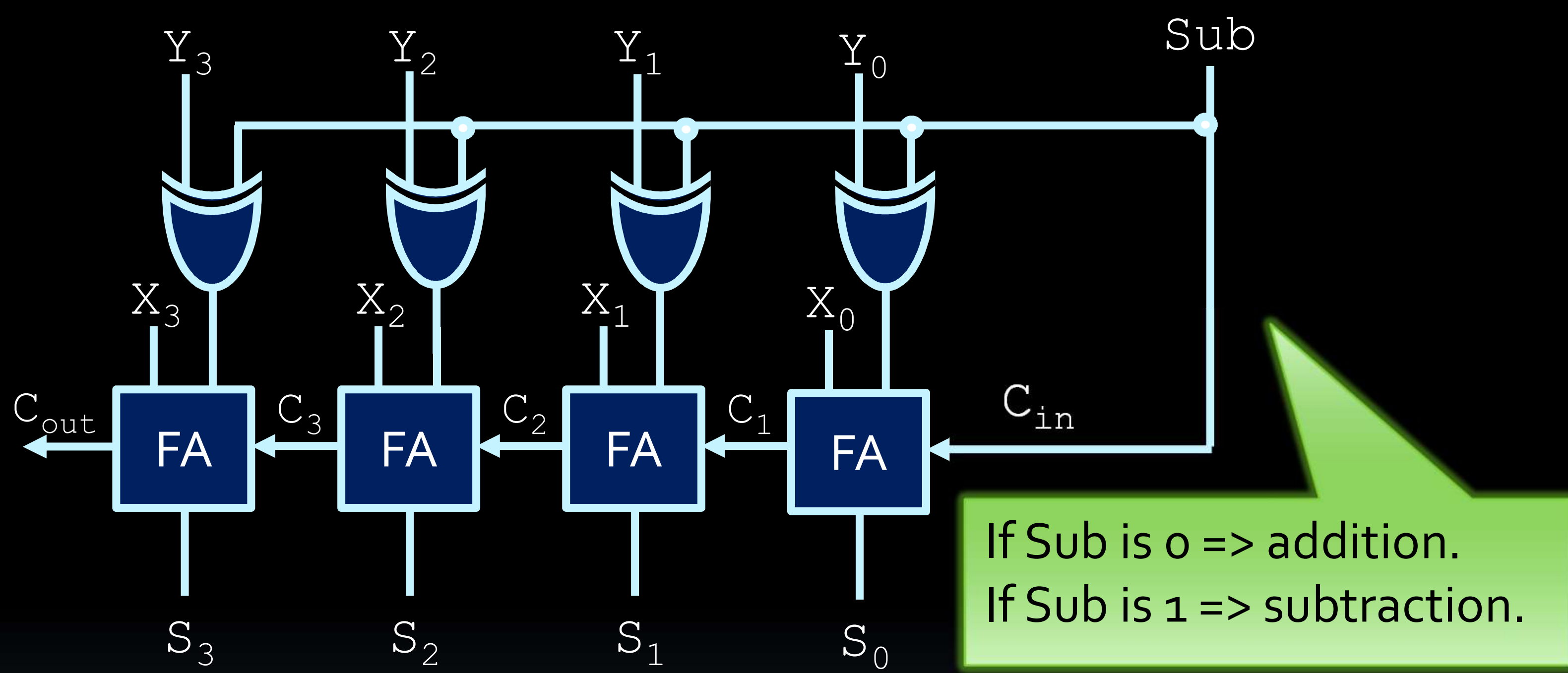
- 4-bit subtractor: X – Y
  - X plus 2's complement of Y
  - X plus 1's complement of Y plus 1

Feed 1 as Carry-In in the least significant FA.

Use NOT gates to get the 1's complement of Y.



# Addition/Subtraction circuit




- The full adder circuit can be expanded to incorporate the subtraction operation.
  - Remember: 2's complement = 1's complement + 1
    - We need Sub fed as $C_{in}$

# Food for Thought

- What happens if we add these two positive signed binary numbers `0110` + `0011` (i.e., 6 + 3) ?
  - The result is `1001`.
  - But that is a negative number (−7)! ☹
- What happens if we add the two negative numbers `1000` + `1111` (i.e., −8 + (−1))?
  - The result is `0111` with a carry-out. ☹
- We need to know when the result might be wrong.
  - This is usually indicated in hardware by the Overflow flag!
  - More about this when we'll talk about processors.

# Unsigned subtraction

- General algorithm:
  1. Get the 2's complement of the subtrahend (the term being subtracted).
  2. Add that value to the minuend (the term being subtracted from).
  3. If there is an end carry ($C_{out}$ is high), the final result is positive and does not change.
  4. If there is no end carry ($C_{out}$ is low), get the 2's complement of the result and add a negative sign to it (or set the sign bit high).
- Special case for signed subtraction:
  - Sign and magnitude representation (using a sign bit).

# Unsigned subtraction example

- 53 – 27

```
           00110101
          -00011011
          ---------
           00110101
carry bit +11100101
          ---------
          100011010

no need for sign bit
           00011010
```

- 27 – 53

```
              00011011
             -00110101
             ---------
              00011011
no carry bit +11001011
             ---------
             011100110

needs a sign bit
             -00011010
```

# Sign & Magnitude Representation

- The Sign part: one bit is designated as the sign (+/-).
  - `0` for positive numbers
  - `1` for negative numbers
- The Magnitude part: Remaining bits store the positive (i.e., unsigned) version of the number.

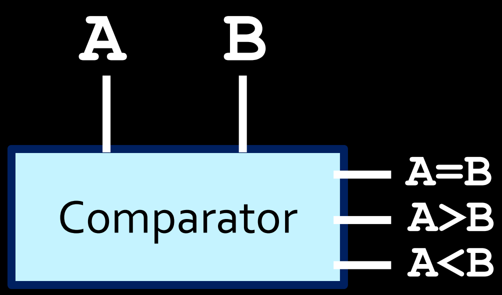
- Example: 4-bit binary numbers:
  - `0110` is `6` while `1110` is `–6` (most significant bit is the sign)
  - What about `0000` and `1000`? => zero (two ways)
- Sign-magnitude computation is more complicated.
  - 2's complement is what today's systems use!

# Comparators

# Comparators

- A circuit that takes in two input vectors, and determines if the first is greater than, less than or equal to the second.
- How does one make that in a circuit?

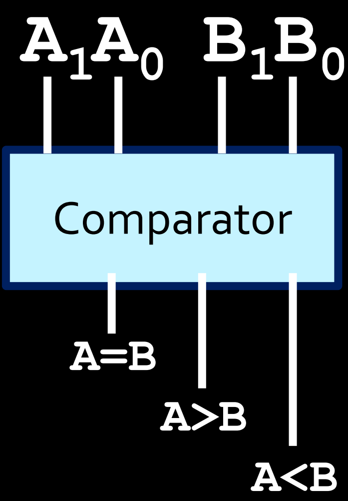
# Basic Comparators




- Consider two binary numbers A and B, where A and B are one bit long.
- The circuits for this would be:
  - A==B: $A \cdot B + \overline{A} \cdot \overline{B}$
  - A>B: $A \cdot \overline{B}$
  - A<B: $\overline{A} \cdot B$

| A | B |
|---|---|
| 0 | 0 |
| 0 | 1 |
| 1 | 0 |
| 1 | 1 |

# Basic Comparators

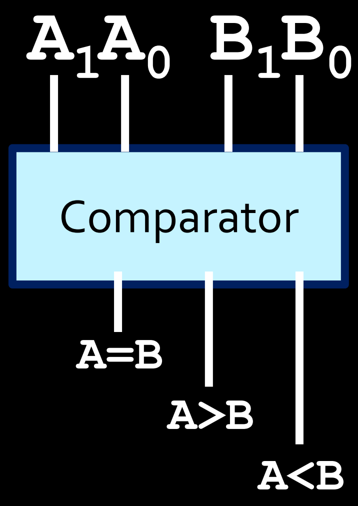



- What if A and B are two bits long?
- The terms for this circuit for have to expand to reflect the second signal.
- For example:
  - A==B: $(A_1 \cdot B_1 + \overline{A}_1 \cdot \overline{B}_1) \cdot (A_0 \cdot B_0 + \overline{A}_0 \cdot \overline{B}_0)$

Make sure that the values of bit 1 are the same

Make sure that the values of bit 0 are the same

# Basic Comparators



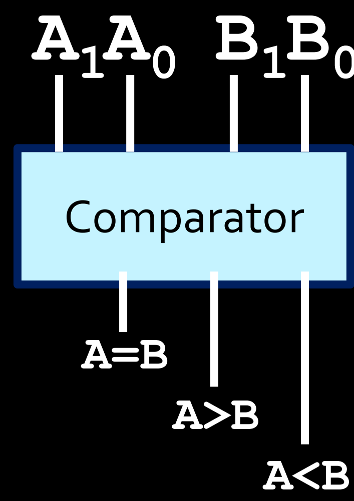

- What about checking if A is greater or less than B?
  - A>B: $A_1 \cdot \overline{B_1} + (A_1 \cdot B_1 + \overline{A_1} \cdot \overline{B_1}) \cdot (A_0 \cdot \overline{B_0})$
  - A<B: $\overline{A_1} \cdot B_1 + (A_1 \cdot B_1 + \overline{A_1} \cdot \overline{B_1}) \cdot (\overline{A_0} \cdot B_0)$

Check if first bit satisfies condition

If not, check that the first bits are equal…

…and then do the 1-bit comparison

# Basic Comparators




- The final circuit equations for two-input comparators are shown below.
  - Note the sections they have in common!

  - A==B: $(A_1 \cdot B_1 + \overline{A}_1 \cdot \overline{B}_1) \cdot (A_0 \cdot B_0 + \overline{A}_0 \cdot \overline{B}_0)$
  - A>B: $A_1 \cdot \overline{B}_1 + (A_1 \cdot B_1 + \overline{A}_1 \cdot \overline{B}_1) \cdot (A_0 \cdot \overline{B}_0)$
  - A<B: $\overline{A}_1 \cdot B_1 + (A_1 \cdot B_1 + \overline{A}_1 \cdot \overline{B}_1) \cdot (\overline{A}_0 \cdot B_0)$

# General Comparators

- The general circuit for comparators requires you to define equations for each case.
- Case #1: Equality
  - If inputs A and B are equal, then all bits must be the same.
  - Define $X_i$ for any digit $i$:
    - (equality for digit $i$)

$$X_i = A_i \cdot B_i + \overline{A_i} \cdot \overline{B_i}$$

  - Equality between A and B is defined as:

$$A{=}{=}B \; : \; X_0 \cdot X_1 \cdot \ldots \cdot X_n$$

# Comparators

- **Case #2:** $A > B$
  - The first non-matching bits occur at bit $i$, where $A_i=1$ and $B_i=0$. All higher bits match.
  - Using the definition for $X_i$ from before:

$$A{>}B = A_n \cdot \overline{B}_n + X_n \cdot A_{n-1} \cdot \overline{B}_{n-1} + \ldots + A_0 \cdot \overline{B}_0 \cdot \prod_{k=1}^{n} X_k$$

- **Case #3:** $A < B$
  - The first non-matching bits occur at bit $i$, where $A_i=0$ and $B_i=1$. Again, all higher bits match.

$$A{<}B = \overline{A}_n \cdot B_n + X_n \cdot \overline{A}_{n-1} \cdot B_{n-1} + \ldots + \overline{A}_0 \cdot B_0 \cdot \prod_{k=1}^{n} X_k$$

# Comparator truth table

- Given two input vectors of size $n=2$, output of circuit is shown at right.

| Inputs | | | | Outputs | | |
|---|---|---|---|---|---|---|
| $A_1$ | $A_0$ | $B_1$ | $B_0$ | $A < B$ | $A = B$ | $A > B$ |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 1 | 0 |

# Comparator example (cont'd)

A<B:

| | $\overline{B}_0 \cdot \overline{B}_1$ | $B_0 \cdot \overline{B}_1$ | $B_0 \cdot B_1$ | $\overline{B}_0 \cdot B_1$ |
|---|---|---|---|---|
| $\overline{A}_0 \cdot \overline{A}_1$ | 0 | 1 | 1 | 1 |
| $A_0 \cdot \overline{A}_1$ | 0 | 0 | 1 | 1 |
| $A_0 \cdot A_1$ | 0 | 0 | 0 | 0 |
| $\overline{A}_0 \cdot A_1$ | 0 | 0 | 1 | 0 |

$$LT = B_1 \cdot \overline{A}_1 + B_0 \cdot B_1 \cdot \overline{A}_0 + B_0 \cdot \overline{A}_0 \cdot \overline{A}_1$$

# Comparator example (cont'd)

A=B:

| | $\overline{B}_0 \cdot \overline{B}_1$ | $B_0 \cdot \overline{B}_1$ | $B_0 \cdot B_1$ | $\overline{B}_0 \cdot B_1$ |
|---|---|---|---|---|
| $\overline{A}_0 \cdot \overline{A}_1$ | 1 | 0 | 0 | 0 |
| $A_0 \cdot \overline{A}_1$ | 0 | 1 | 0 | 0 |
| $A_0 \cdot A_1$ | 0 | 0 | 1 | 0 |
| $\overline{A}_0 \cdot A_1$ | 0 | 0 | 0 | 1 |

$$EQ = \overline{B}_0 \cdot \overline{B}_1 \cdot \overline{A}_0 \cdot \overline{A}_1 + B_0 \cdot \overline{B}_1 \cdot A_0 \cdot \overline{A}_1 + B_0 \cdot B_1 \cdot A_0 \cdot A_1 + \overline{B}_0 \cdot B_1 \cdot \overline{A}_0 \cdot A_1$$

# Comparator example (cont'd)

A>B:

| | $\overline{B_0} \cdot \overline{B_1}$ | $B_0 \cdot \overline{B_1}$ | $B_0 \cdot B_1$ | $\overline{B_0} \cdot B_1$ |
|---|---|---|---|---|
| $\overline{A_0} \cdot \overline{A_1}$ | 0 | 0 | 0 | 0 |
| $A_0 \cdot \overline{A_1}$ | 1 | 0 | 0 | 0 |
| $A_0 \cdot A_1$ | 1 | 1 | 0 | 1 |
| $\overline{A_0} \cdot A_1$ | 1 | 1 | 0 | 0 |

$$GT = \overline{B}_1 \cdot A_1 + \overline{B}_0 \cdot \overline{B}_1 \cdot A_0 + \overline{B}_0 \cdot A_0 \cdot A_1$$

# Comparators in Verilog

- Implementing a comparator can be done by putting together the circuits as shown in the previous slide, or by using the comparison operators to make things a little easier:

```
module comparator_4_bit (a_gt_b, a_lt_b, a_eq_b, a, b);

input [3:0] a, b;
output a_gt_b, a_lt_b, a_eq_b;

assign a_gt_b = (a > b);
assign a_lt_b = (a < b);
assign a_eq_b = (a == b);

endmodule
```

# Comparing larger numbers

- As numbers get larger, the comparator circuit gets more complex.
- At a certain level, it can be easier sometimes to just process the result of a subtraction operation instead.
  - Easier, less circuitry, just not faster.