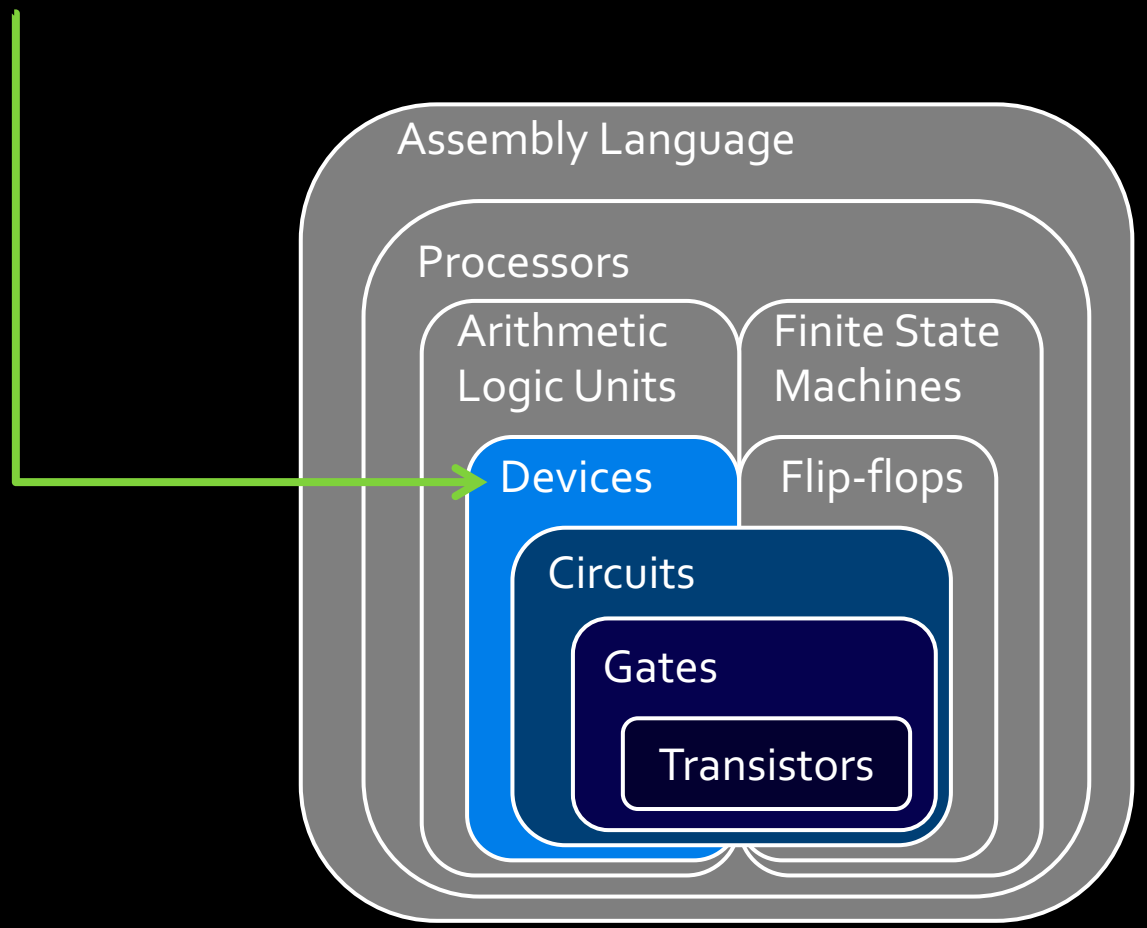
CSC258 Week 3

We are here

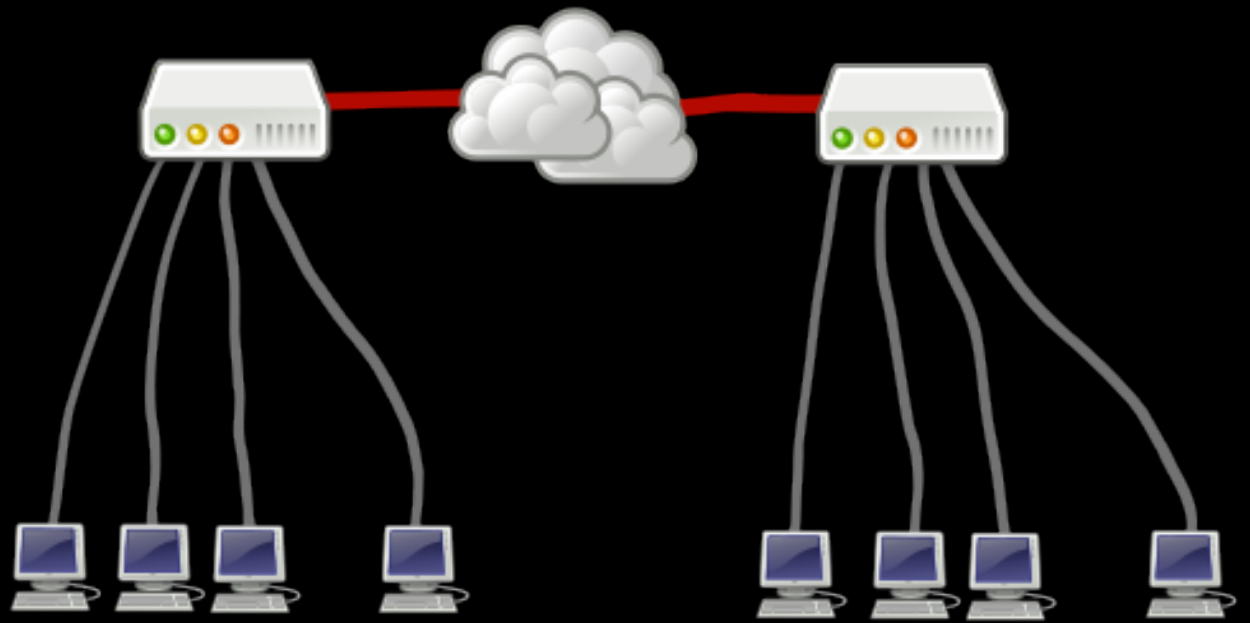

Assembly Language
Processors
Arithmetic Logic Units
Finite State Machines
Devices
Flip-flops
Circuits
Gates
Transistors

Logical Devices

Building up from gates…

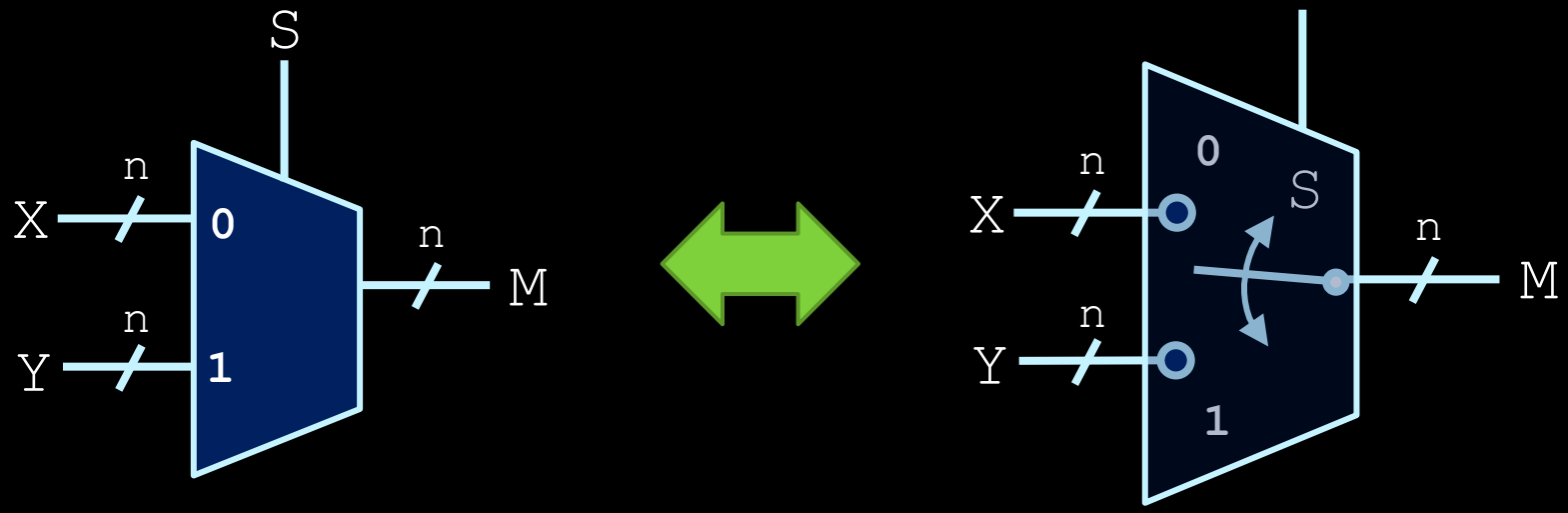
- Some common and more complex structures:
  - Multiplexers (MUX)
  - Adders (half and full)
  - Subtractors
  - Decoders
    - Seven-segment decoders
  - Comparators

Multiplexers



Logical devices

- Certain structures are common to many circuits, and have block elements of their own.
  - e.g. Multiplexers (short form: mux)
  - Behaviour: Output is `X` if `S` is `0`, and `Y` if `S` is `1`, i.e., S selects which input can go through

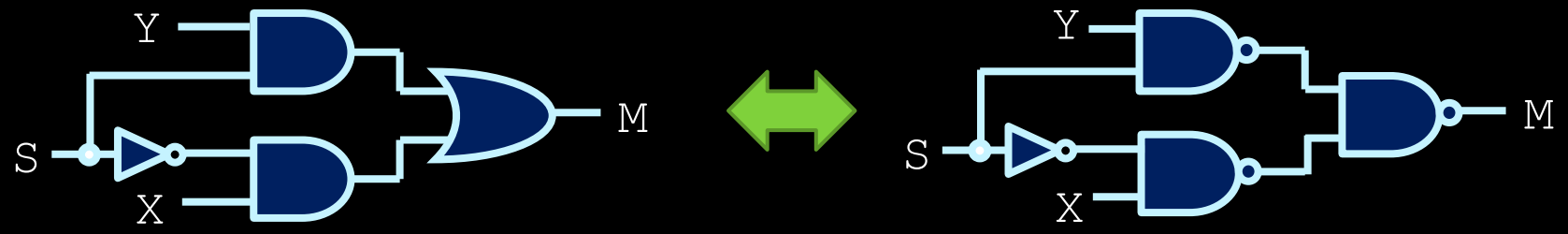


Multiplexer design

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**

	$\bar{Y}\cdot\bar{S}$	$\bar{Y}\cdot S$	$Y\cdot S$	$Y\cdot\bar{S}$
$\bar{X}$	0	0	1	0
X	1	0	1	1

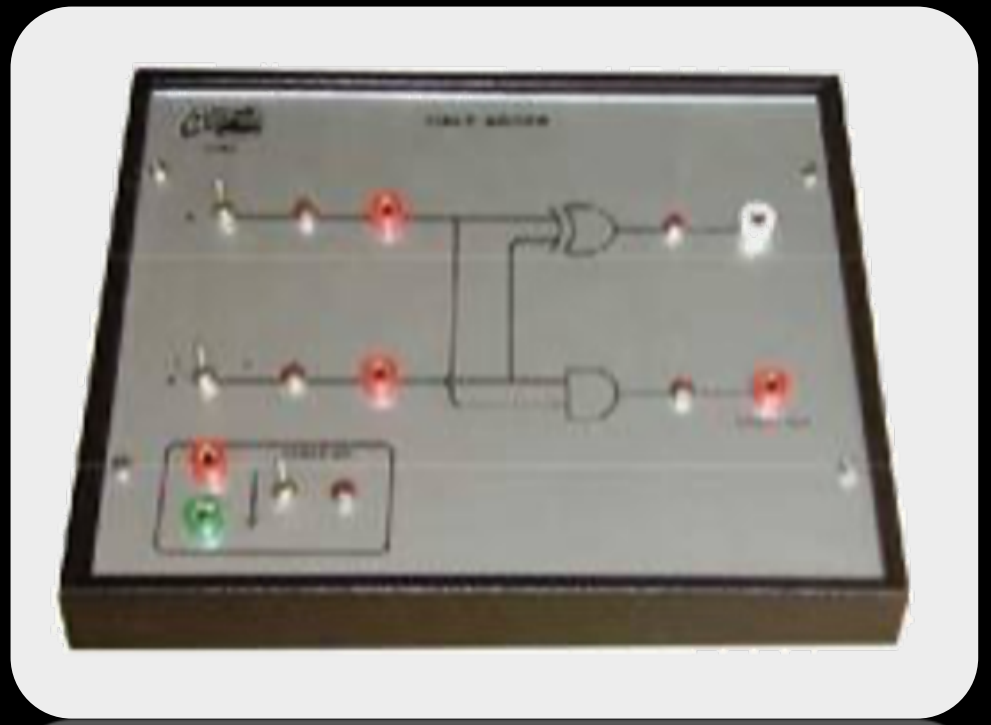
$$M = Y\cdot S + X\cdot\bar{S}$$



Multiplexer uses

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

Adder circuits



Adders

- Also known as binary adders.
  - Small circuit devices that add two 1-bit number.
  - Combined together to create iterative combinational circuits – add multiple-bit numbers
- Types of adders:
  - Half adders (HA)
  - Full adders (FA)
  - Ripple Carry Adder
  - Carry-Look-Ahead Adder (CLA)

Review of Binary Math

Review of Binary Math

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

$$\mathbf{258} = 2x10^2 + 5x10^1 + 8x10^0$$

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

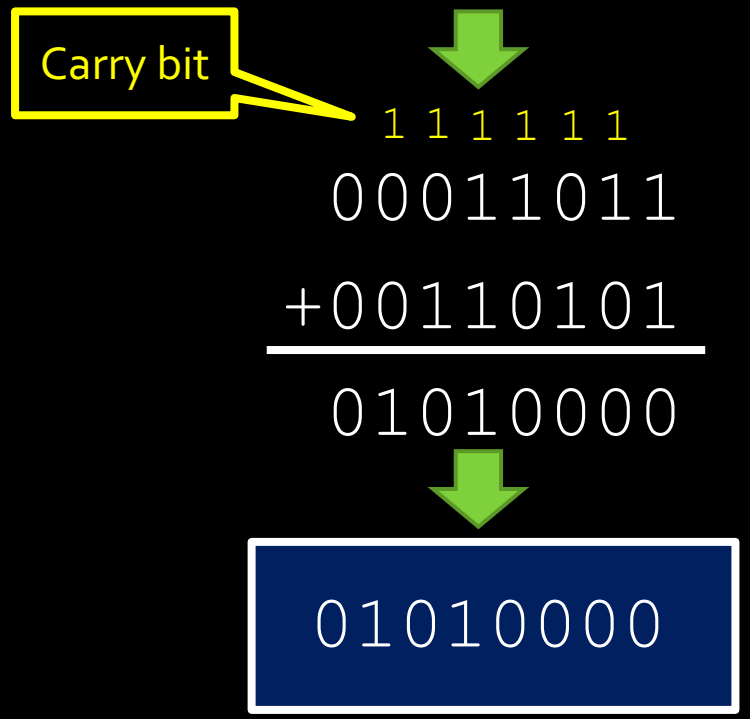
$$\mathbf{01101_2} = 0x2^4 + 1x2^3 + 1x2^2 + 0x2^1 + 1x2^0$$
$$= \mathbf{13_{10}}$$

Unsigned binary addition

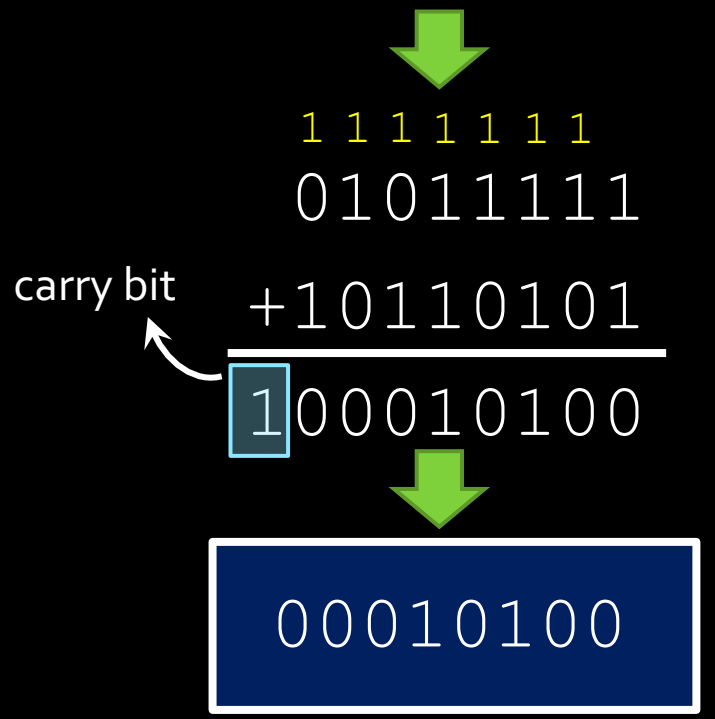
- 27 + 53

27 = 00011011

53 = 00110101



- 95 + 181



Half Adder

Input: two 1-bit numbers
Output: 1-bit sum and 1-bit carry

Half Adders

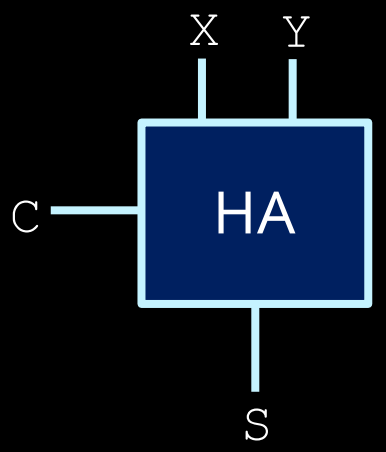
- A 2-input, 1-bit width binary adder that performs the following computations:

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

C = X?Y

S = X?Y

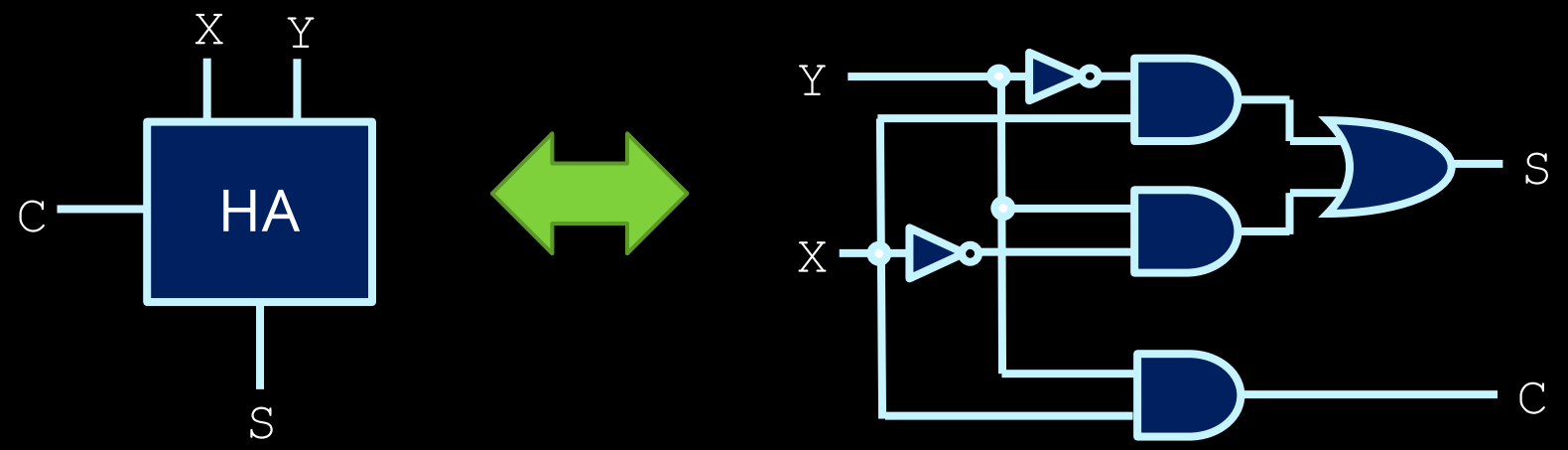
- A half adder adds two bits to produce a two-bit sum.
- The sum is expressed as a sum bit S and a carry bit C.



Half Adder Implementation

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

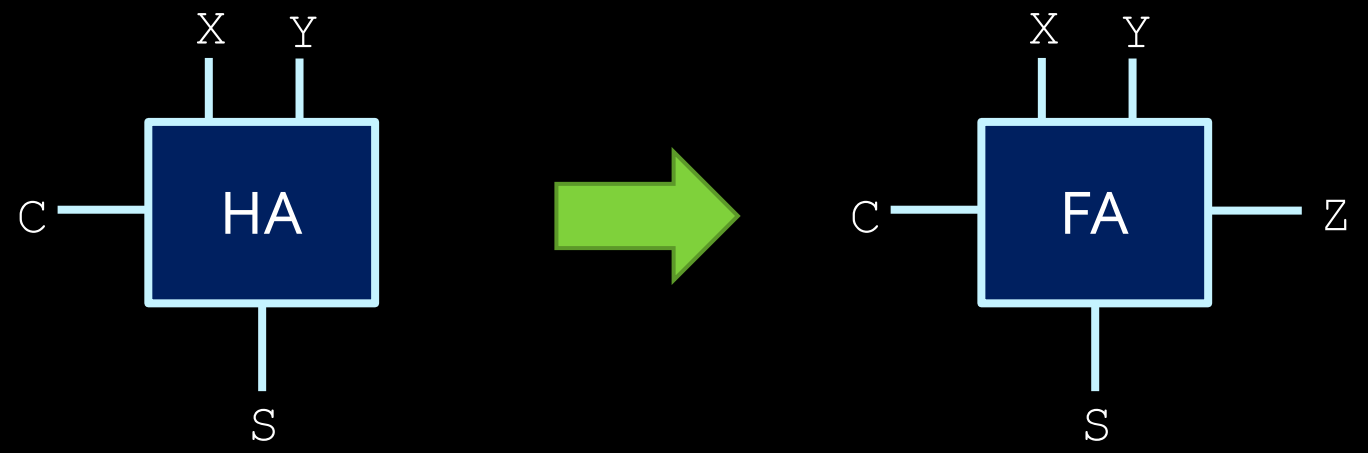
$$C = X \cdot Y \qquad S = X \cdot \overline{Y} + \overline{X} \cdot Y = X \text{ xor } Y$$



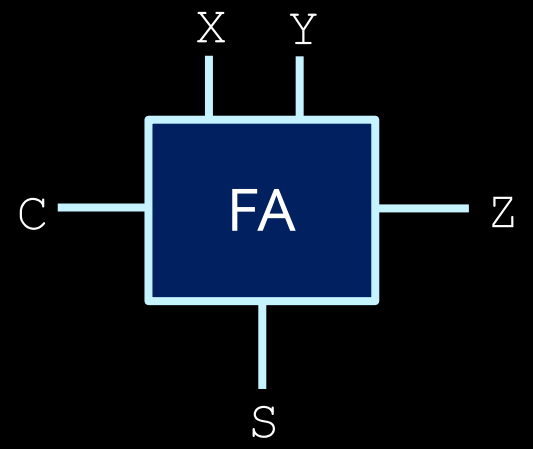
A half adder outputs a carry-bit,
but does not take a carry-bit as input.

Full Adder

takes a carry bit as input



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	$\overline{Y}\cdot\overline{Z}$	$\overline{Y}\cdot Z$	$Y\cdot Z$	$Y\cdot\overline{Z}$
$\overline{X}$	0	0	1	0
X	0	1	1	1

S	$\overline{Y}\cdot\overline{Z}$	$\overline{Y}\cdot Z$	$Y\cdot Z$	$Y\cdot\overline{Z}$
$\overline{X}$	0	1	0	1
X	1	0	1	0

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

$$C = X\cdot Y + (X \text{ xor } Y)\cdot Z$$

$$S = X \text{ xor } Y \text{ xor } Z$$

For gate reuse (X xor Y) considering both C and S

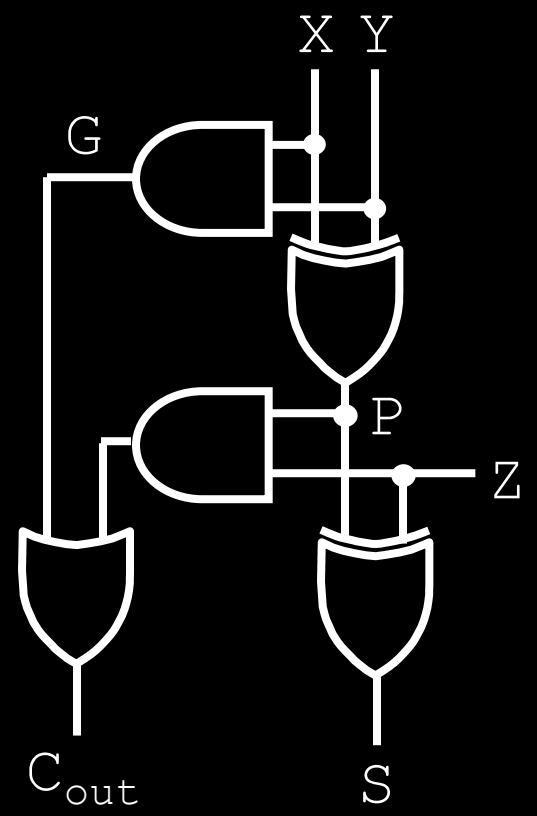
Full Adder Design

S = X xor Y xor Z

- The C term can also be rewritten as:

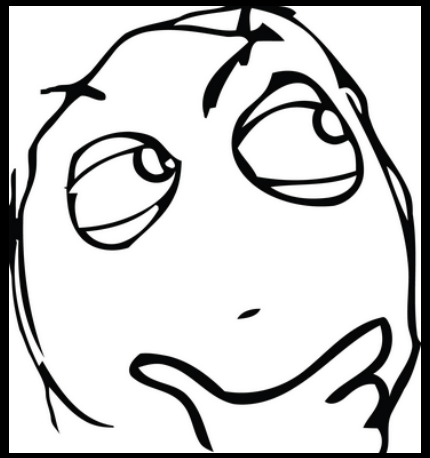
C = X·Y + (X xor Y)·Z

- Two terms come from this:
  - X·Y = carry generate (G).
    - Whether X and Y generate a carry bit
  - X xor Y = carry propagate (P).
    - Whether carry will be propagated to Cout
- Results in this circuit →

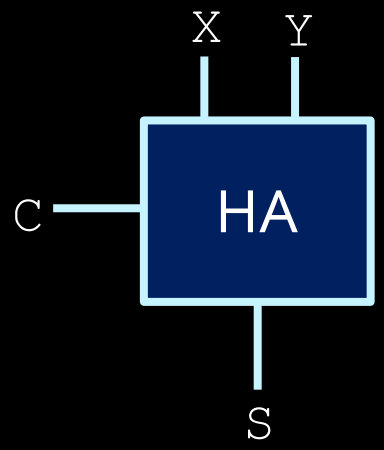


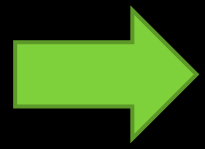
Now we can add one bit properly, but most of the numbers we use have more than one bits.

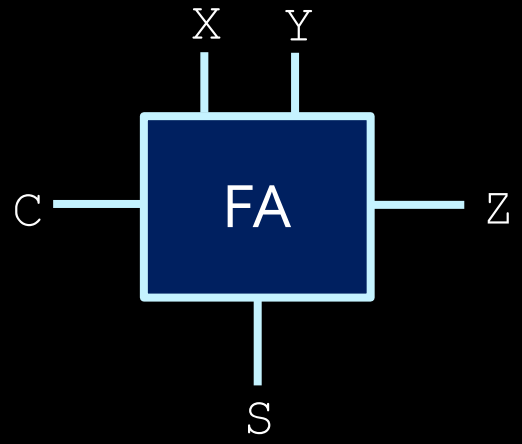
- int, unsigned int: 32 bits (architecture-dependent)
- short int, unsigned short int: 16 bits
- long long int, unsigned long long int: 64 bit
- char, unsigned char: 8 bits

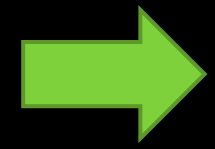


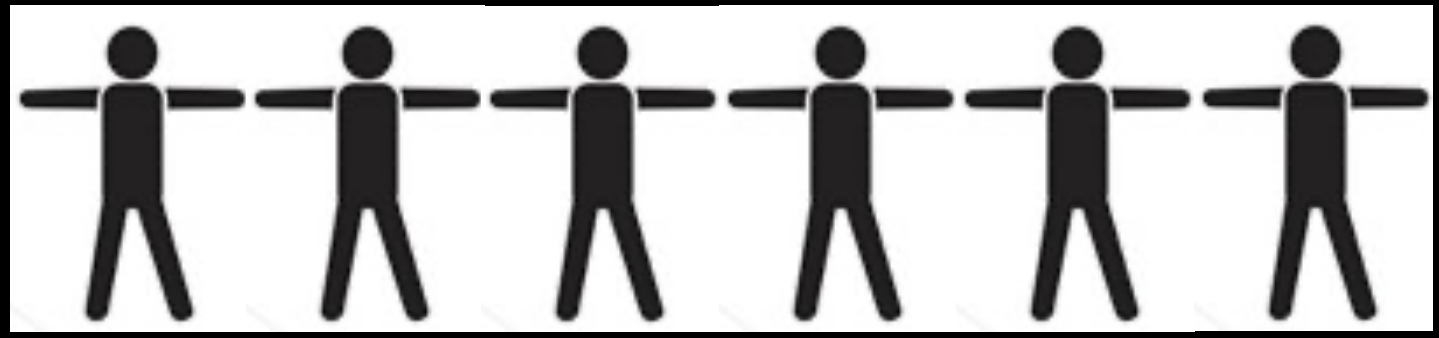
How do we add multiple-bit numbers?


X
Y
HA
C
S




X
Y
FA
C
Z
S





Each full adder takes in a carry bit and outputs a carry bit.

Each full adder can take in a carry bit which is output by another full adder.

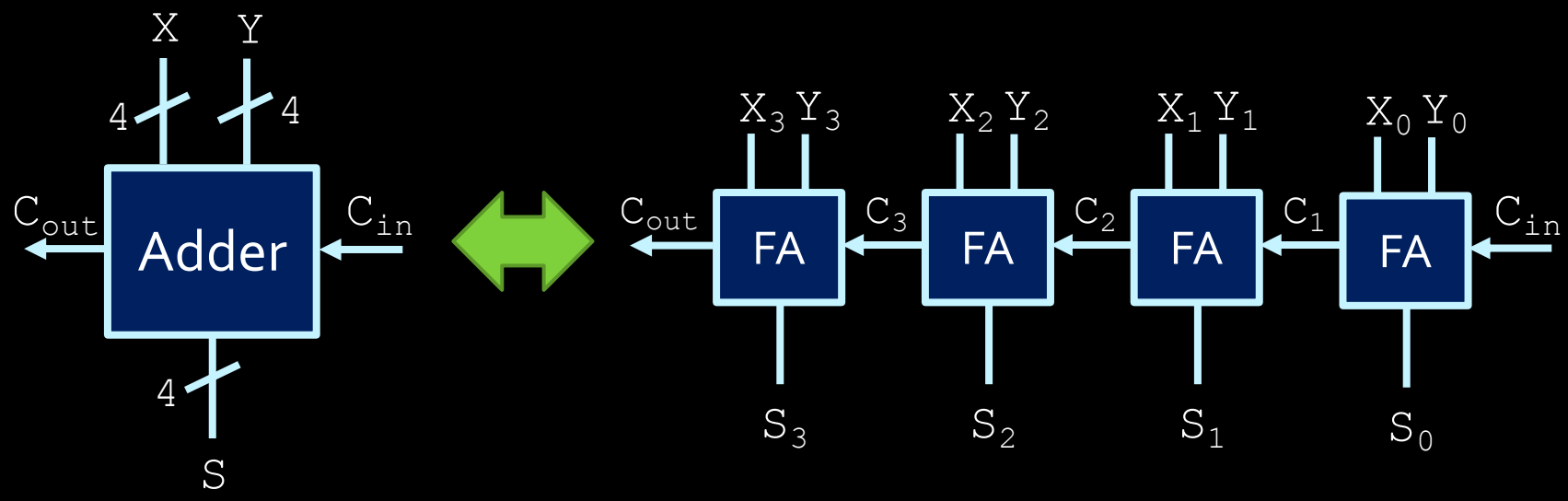
That is, they can be **chained** up.

Ripple-Carry Binary Adder

Full adders chained up,
for multiple-bit addition

Ripple-Carry Binary Adder

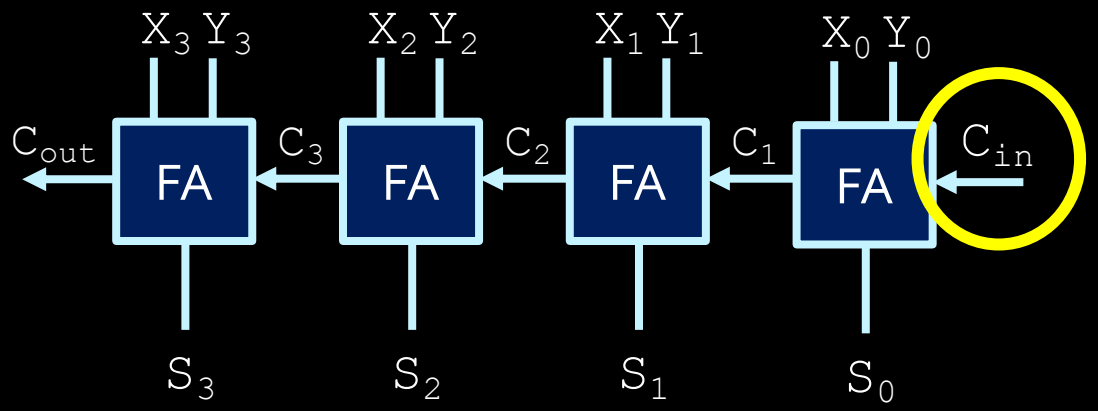
- Full adder units are chained together in order to perform operations on signal vectors.



$S_3S_2S_1S_0$ is the sum of $X_3X_2X_1X_0$ and $Y_3Y_2Y_1Y_0$

The role of C_{in}

- Why can't we just have a half-adder for the smallest (right-most) bit?
- Because if we can use it to do **subtraction**!



Let's play a game…

1. Pick two numbers between 0 and 31
2. Convert both numbers to **5-bit** binary form
3. **Invert** each digit of the smaller number
4. Add up the big binary number and the inverted small binary number
5. **Add 1** to the result, keep the lowest 5 digits
6. Convert the result to a decimal number

What do you get?

You just did subtraction without doing subtraction!

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.

Two's complement

- Need to know how to get 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.

(2's complement of A) + A = 0.

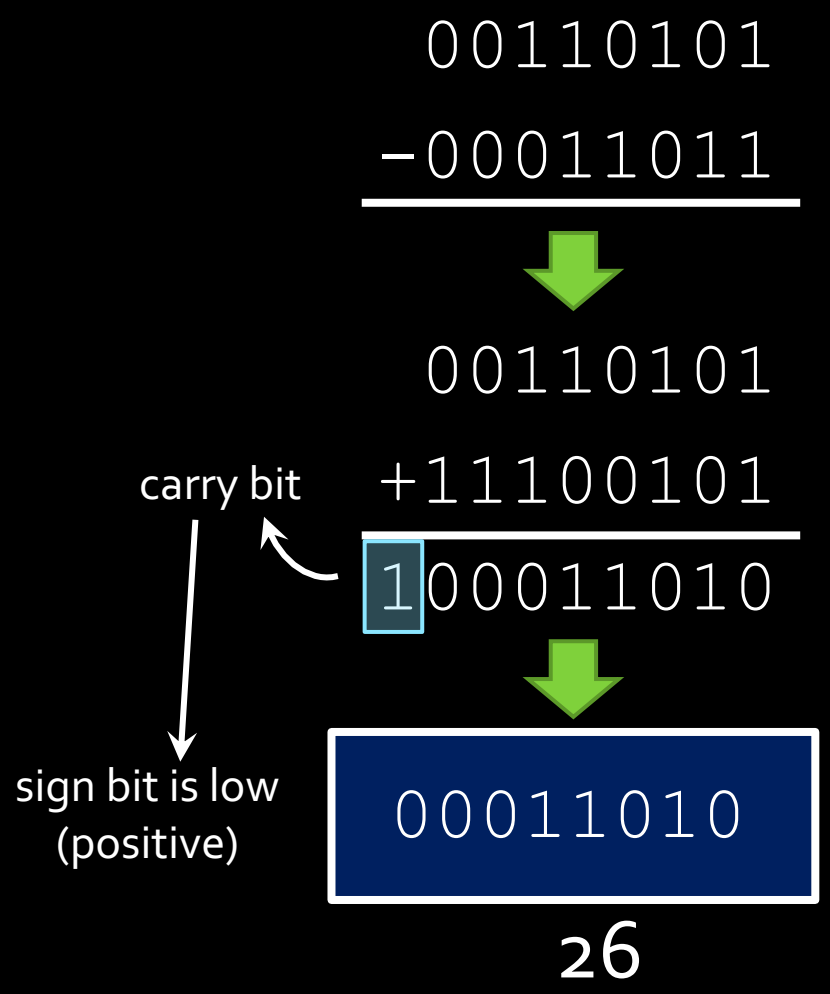
The 2's complement of A is like -A

Unsigned subtraction (separate sign bit)

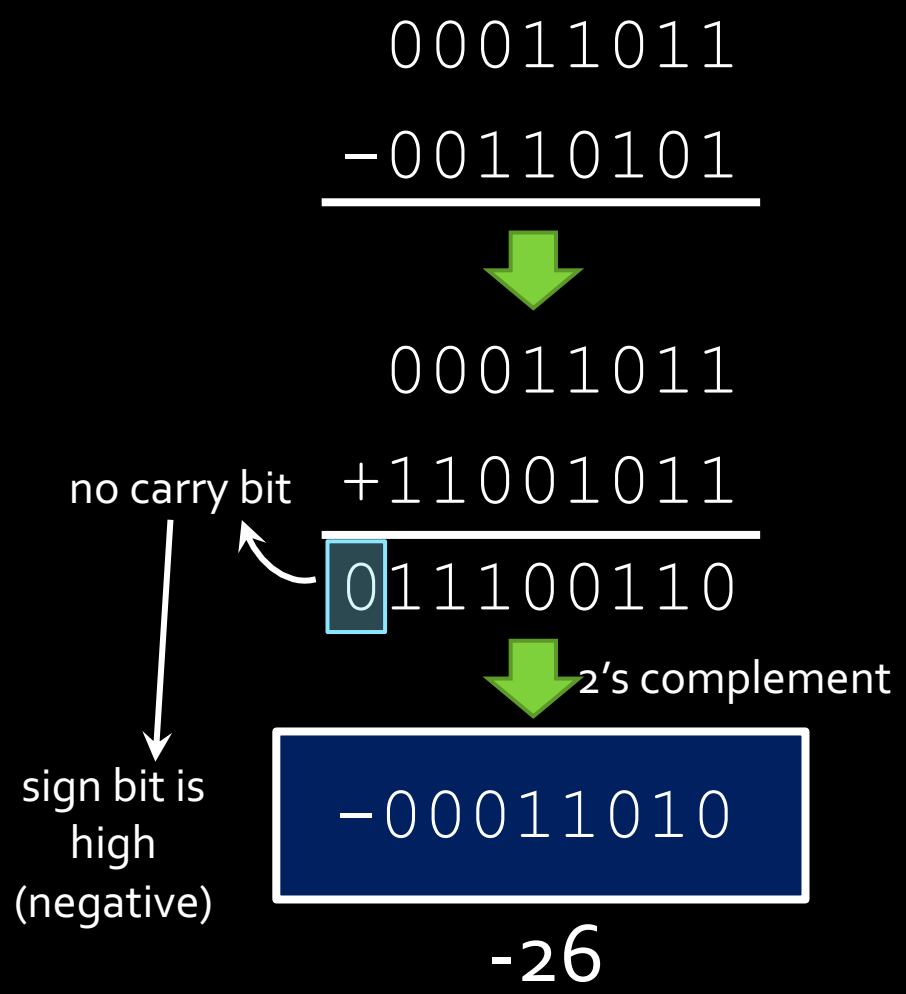
- General algorithm for **A - B**:
  1. Get the 2's complement of B (-B)
  2. Add that value to **A**
  3. If there is an end carry (C_{out} is high), the final result is positive and does not change.
  4. If there is no end carry (C_{out} is low), get the 2's complement of the result (B-A) and add a negative sign to it, or set the sign bit high (-(B-A) = A-B).

Unsigned subtraction example

- 53 – 27



- 27 – 53



Signed subtraction (easier)

- Store negative numbers in 2's complement notation.
  - Subtraction can then be performed by using the binary adder circuit with negative numbers.
  - To compute A – B, just do A + (-B)
  - Need to get -B first (the 2's complement of B)

Signed subtraction example (6-bit)

- 21 – 23

- 23 is 010111
- 21 is 010101
- -23 is 101001 (2’s complement of 32)
- 21-23 is 111110 which is -2

Signed addition example (6-bit)

- 21 + 23

- 23 is 010111
- 21 is 010101
- 23+21: 101100
- This is -20!
- The supposed result 44 is exceeding the range of 6-bit signed integers. This is called an overflow.

Now you understand C code better

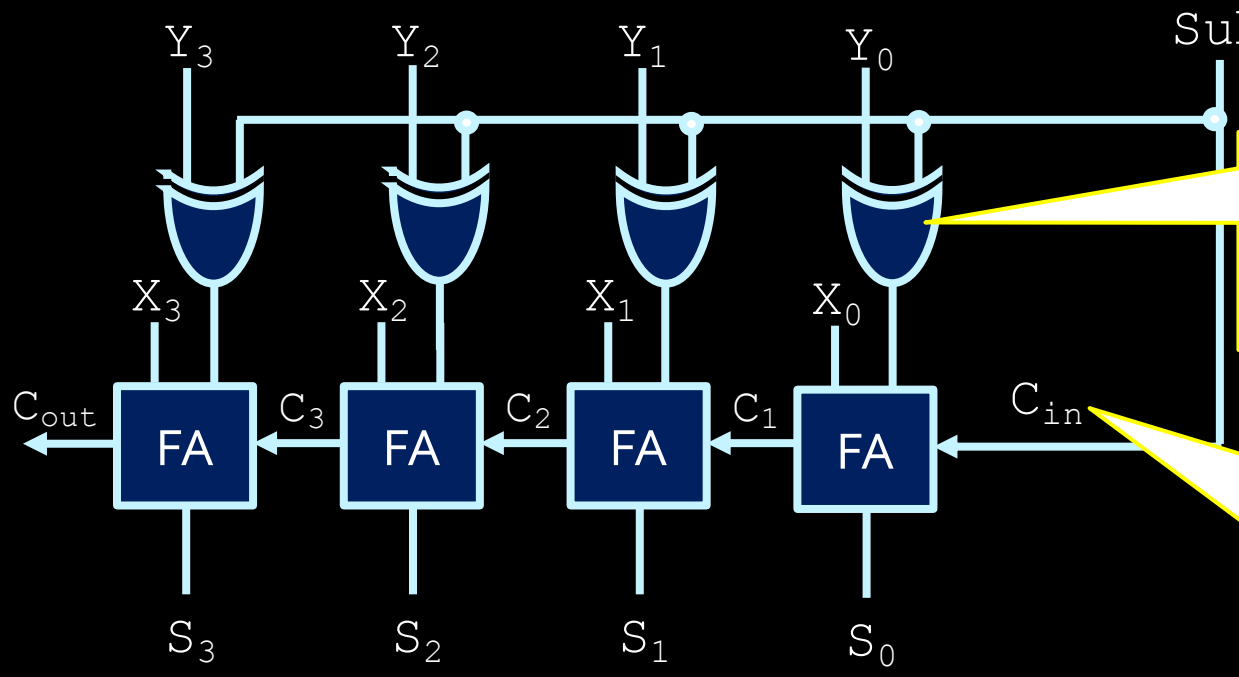
```
#include <stdio.h>


int main()
{
    /* char is 8-bit integer */
    signed char a = 100;
    signed char b = 120;
    signed char s = a + b;
    printf("%d\n", s);
}
```

Trivia about sign numbers

- The largest positive 8-bit signed integer?
  - `01111111 = 127` (`0` followed by all `1`)
- The smallest negative 8-bit signed integer?
  - `10000000 = -128` (`1` followed by all `0`)
- The binary form 8-bit signed integer `-1`?
  - `11111111` (all one)
- For n-bit signed number there are 2^n possible values
  - 2^{n-1} are negative numbers (e.g. 8 bit, -1 to -128)
  - $2^{n-1}-1$ are positive number (e.g. 8 bit, 1 to 127)
  - and a zero

-128: 10000000 (signed)

Subtraction circuit



Invert all the digits (if sub = 1)

Add 1, so getting 2's complement

- If sub = 0, S = X + Y
- If sub = 1, S = X – Y

One circuit, both adder or subtractor

Decoders

What is a decoder?

Decoder

…

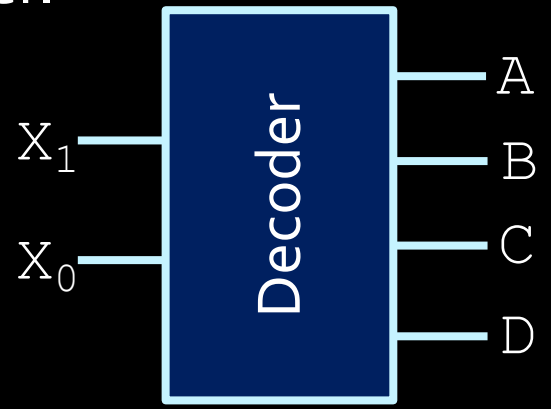
"number 1"
"number 2"
"number 3"
…
"number 10"
"rock!"
"good job!"
"….."

5-bit input, encoded original information

The original information

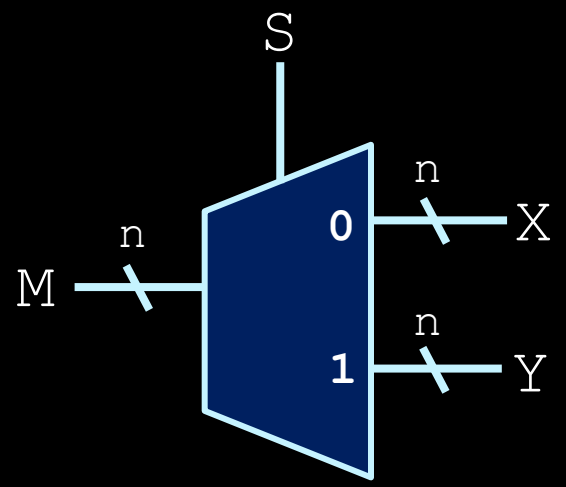
Decoders

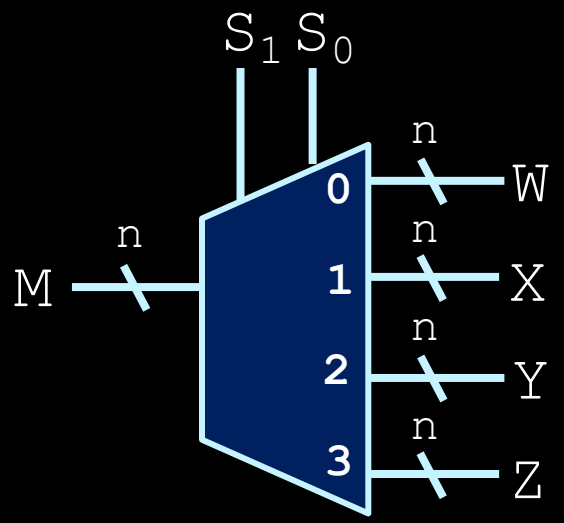
- Decoders are essentially translators.
  - Translate from the output of one circuit to the input of another.
- Example: Binary signal splitter
  - Activates one of four output lines, based on a two-digit binary number.



Demultiplexers

- Related to decoders: demultiplexers.
  - Does multiplexer operation, in reverse.



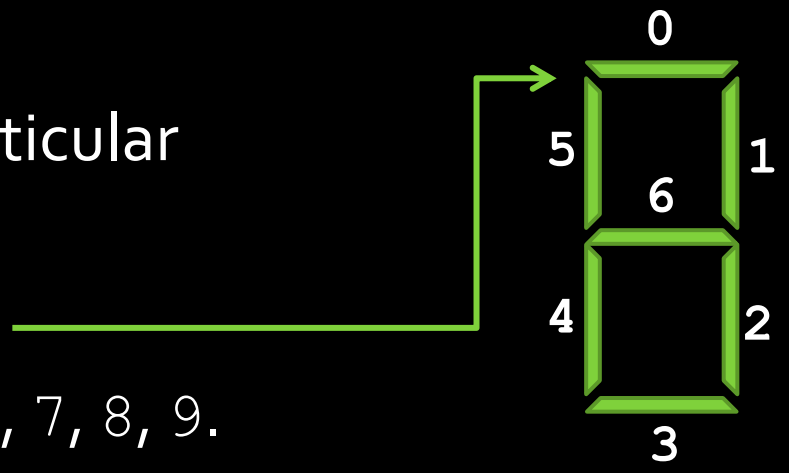


Multiplexer:
Choose one from multiple inputs as output

Demultiplexer:
One input chooses from multiple outputs

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.
  - Example: Segment 0
    - Activate for values: 0, 2, 3, 5, 6, 7, 8, 9.
    - In binary: 0000, 0010, 0011, 0101, 0110, 0111, 1000, 1001.
  - First step: Build the truth table and K-map.

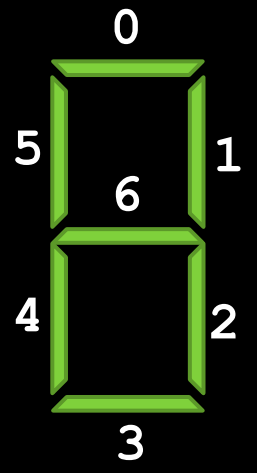


Note

What we talk about here is NOT the same as what we did in Lab 2

- In labs we translate numbers 0, 1, 3, 4, 5, 6 to displayed letters such as (H, E, L, L, O, _, E, L, I)
  - This is specially defined for the lab
- Here we are talking about translating 0, 1, 2, 3, 4,…, to displayed 0, 1, 2, 3, 4, …
  - This is more common use

7-segment decoder



- For 7-seg decoders, turning a segment on involves driving it low. (active low)
  - (In Lab 2, we treated it like active high. It's OK because Logisim does auto-conversion to make it work).
  - i.e., Assuming a 4-digit binary number, segment `0` is low whenever input number is `0000`, `0010`, `0011`, `0101`, `0110`, `0111`, `1000` or `1001`, and high whenever input number is `0001` or `0100`.
  - This creates a truth table and map like the following…

7-segment decoder

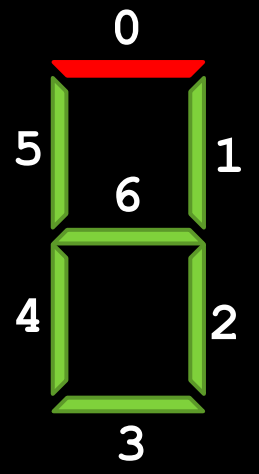
x_3	x_2	x_1	x_0	HEX_0
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0

6 rows missing!

1010 ~ 1111

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

- $HEX0 = \bar{x}_3 \cdot \bar{x}_2 \cdot \bar{x}_1 \cdot x_0 + \bar{x}_3 \cdot x_2 \cdot \bar{x}_1 \cdot \bar{x}_0$
- But what about input values from 1010 to 1111?



"Don't care" values

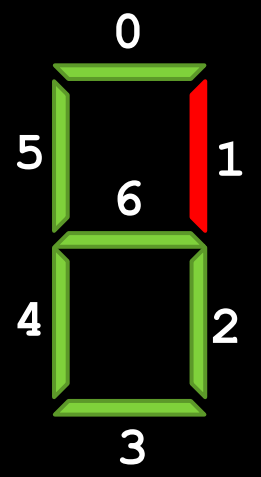
- Some input values will never happen, so their output values do not have to be defined.
  - Recorded as 'X' in the Karnaugh map.
- These values can be assigned to whatever values you want, when constructing the final circuit.

	$\overline{x}_1 \cdot \overline{x}_0$	$\overline{x}_1 \cdot x_0$	$x_1 \cdot x_0$	$x_1 \cdot \overline{x}_0$
$\overline{x}_3 \cdot \overline{x}_2$	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

$$HEX0 = \overline{x}_3 \cdot \overline{x}_2 \cdot \overline{x}_1 \cdot x_0 + x_2 \cdot \overline{x}_1 \cdot \overline{x}_0$$

Boxes can cover "x"'s, or not, whichever you like.

Again for segment 1

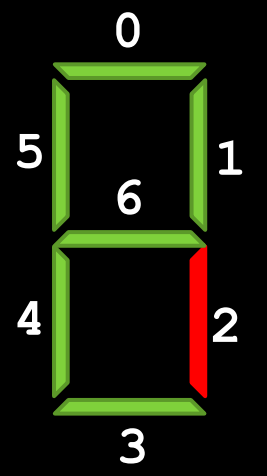


x_3	x_2	x_1	x_0	HEX_1
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0

	$\bar{x}_1 \cdot \bar{x}_0$	$\bar{x}_1 \cdot x_0$	$x_1 \cdot x_0$	$x_1 \cdot \bar{x}_0$
$\bar{x}_3 \cdot \bar{x}_2$	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



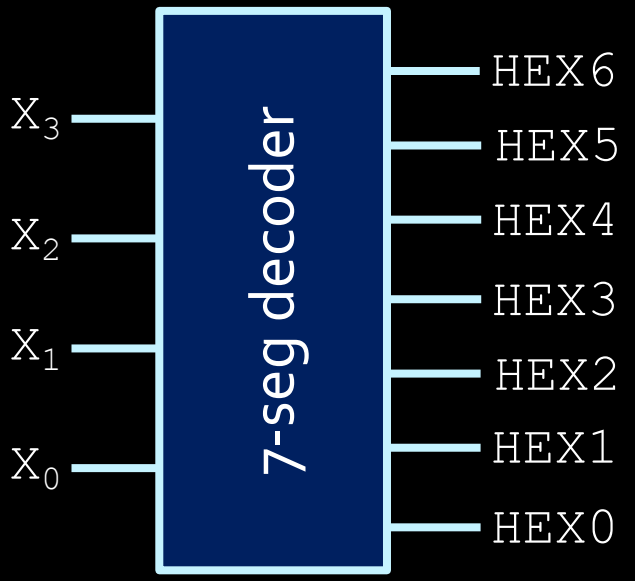
x_3	x_2	x_1	x_0	HEX_2
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0

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

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

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.

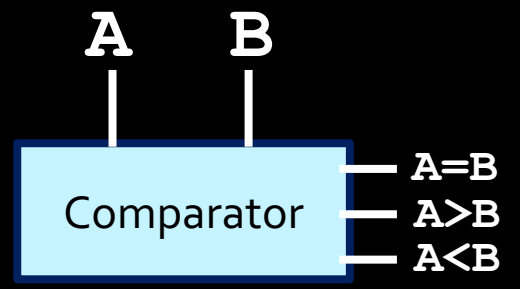


Comparators (leftover from last week)

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:

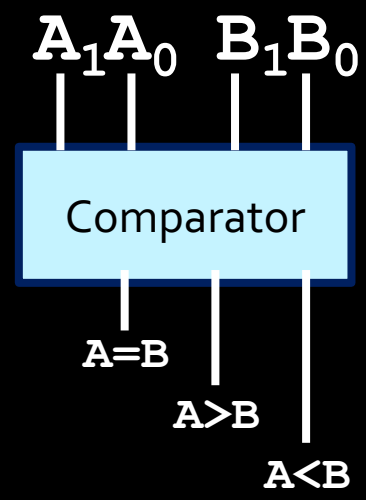
$$A \cdot \overline{B}$$

  - A<B:

$$\overline{A} \cdot B$$

A	B
0	0
0	1
1	0
1	1

Basic Comparators

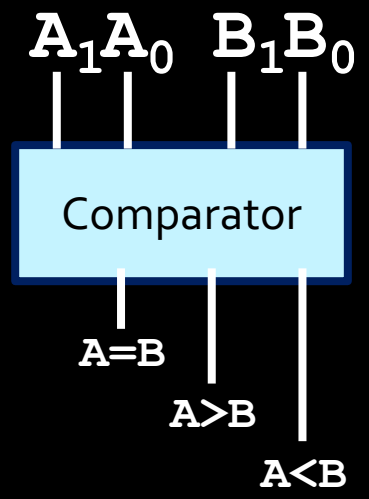


- What if A and B are two bits long?
- The terms for this circuit for have to expand to reflect the second signal.
- For example:
  - A==B: $(A_1 \cdot B_1 + \overline{A}_1 \cdot \overline{B}_1) \cdot (A_0 \cdot B_0 + \overline{A}_0 \cdot \overline{B}_0)$

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:

$$A_1 \cdot \overline{B}_1 + (A_1 \cdot B_1 + \overline{A}_1 \cdot \overline{B}_1) \cdot (A_0 \cdot \overline{B}_0)$$

Check if first bit satisfies condition

If not, check that the first bits are equal…

…and then do the 1-bit comparison

  - A<B:

$$\overline{A}_1 \cdot B_1 + (A_1 \cdot B_1 + \overline{A}_1 \cdot \overline{B}_1) \cdot (\overline{A}_0 \cdot B_0)$$

A > B if and only if A1 > B1 or (A1 = B1 and Ao > Bo)

Comparing large numbers

- The circuit complexity of comparators increases quickly as the input size increases.
- For comparing large number, it may make more sense to just use a subtractor.
  - Subtract and then check the sign bit.

Today we learned

How a computer does following things

- Control the flow of signal (mux and demux)
- Arithmetic operations: adder, subtractor
- Decoder
- comparators

Next week:

- Sequential circuits: circuits that have **memories**.