EECS 370 - Lecture 8

Combinational Logic

Announcements

Floating Point Arithmetic

See end of slides for bonus material (not covered in HW or exams)

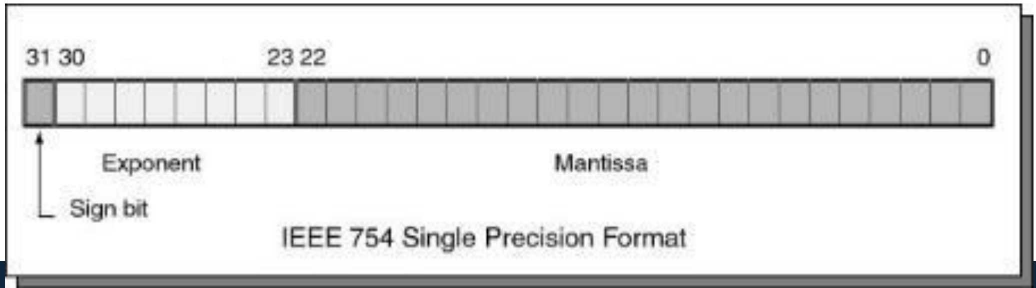
Why floating point

- Have to represent real numbers somehow
- Rational numbers
  - Ok, but can be cumbersome to work with
- Fixed point
  - Do everything in thousandths (or millionths, etc.)
  - Not always easy to pick the right units
  - Different scaling factors for different stages of computation
- <u>**Scientific notation: this is good!**</u>
  - Exponential notation allows HUGE dynamic range
  - Constant (approximately) relative precision across the whole range

IEEE Floating point format (single precision)

- Sign bit: (0 is positive, 1 is negative)
- Significand: (also called the *mantissa*; stores the 23 most significant bits after the decimal point)
- Exponent: used biased base 127 encoding
  - Add 127 to the value of the exponent to encode:
  - -127 → 00000000   1 → 10000000
  - -126 → 00000001   2 → 10000001
  - …   …
  - 0 → 01111111   128 → 11111111
- How do you represent zero ? Special convention:
  - Exponent: -127 (all zeroes), Significand 0 (all zeroes), Sign + or -

Some other exception cases (e.g. NaN) we won't cover



Floating Point Representation

$$10.625_{10} \Rightarrow 1010.101_{2}$$

- Step 1: convert from decimal to binary
  - 1st bit after "binary" point represents 0.5 (i.e. 2^{-1})
  - 2nd bit represents 0.25 (i.e. 2^{-2})
  - etc.

Floating Point Representation

$$10.625_{10} \Rightarrow 1010.101_2$$

$$1.010101 \times 2^3$$

- Step 2: normalize number by shifting binary point until you get 1.XXX * 2^Y

Floating Point Representation

$$\mathbf{10.625}_{10} \Rightarrow \mathbf{1010.101}_{2}$$

$$\mathbf{1.010101} \times \mathbf{2}^{3}$$

This must be a 1!
So don't store it.

+/-	Exponent (3)	Significand (1010101)
1 bit	8 bits	23 bits

- Step 3: store relevant numbers in proper location (ignoring initial 1 of significand)

Floating Point Representation

$$10.625_{10} \Rightarrow 1010.101_{2}$$

$$\color{red}{1}.010101 \times 2^{3}$$

This must be a 1!
So don't store it.

+/-	Exponent (3)	Significand (1010101)
1 bit	8 bits	23 bits

$$10.625_{10} = 0 \quad 10000010 \quad 01010100000000000000000$$

Class Problem

Poll

- What is the value of the following IEEE 754 floating point encoded number?

1 | **10000101** | **01011001000000000000000**

1 = -
10000101 = 133 – 127 -> exponent 6
01011001 = mantissa

-1.01011001 x 2^6

-1010110.01
-(2^6+2^4+2^2+2^1+2^-2)
-(64+16+4+2+1/4)

-86.25

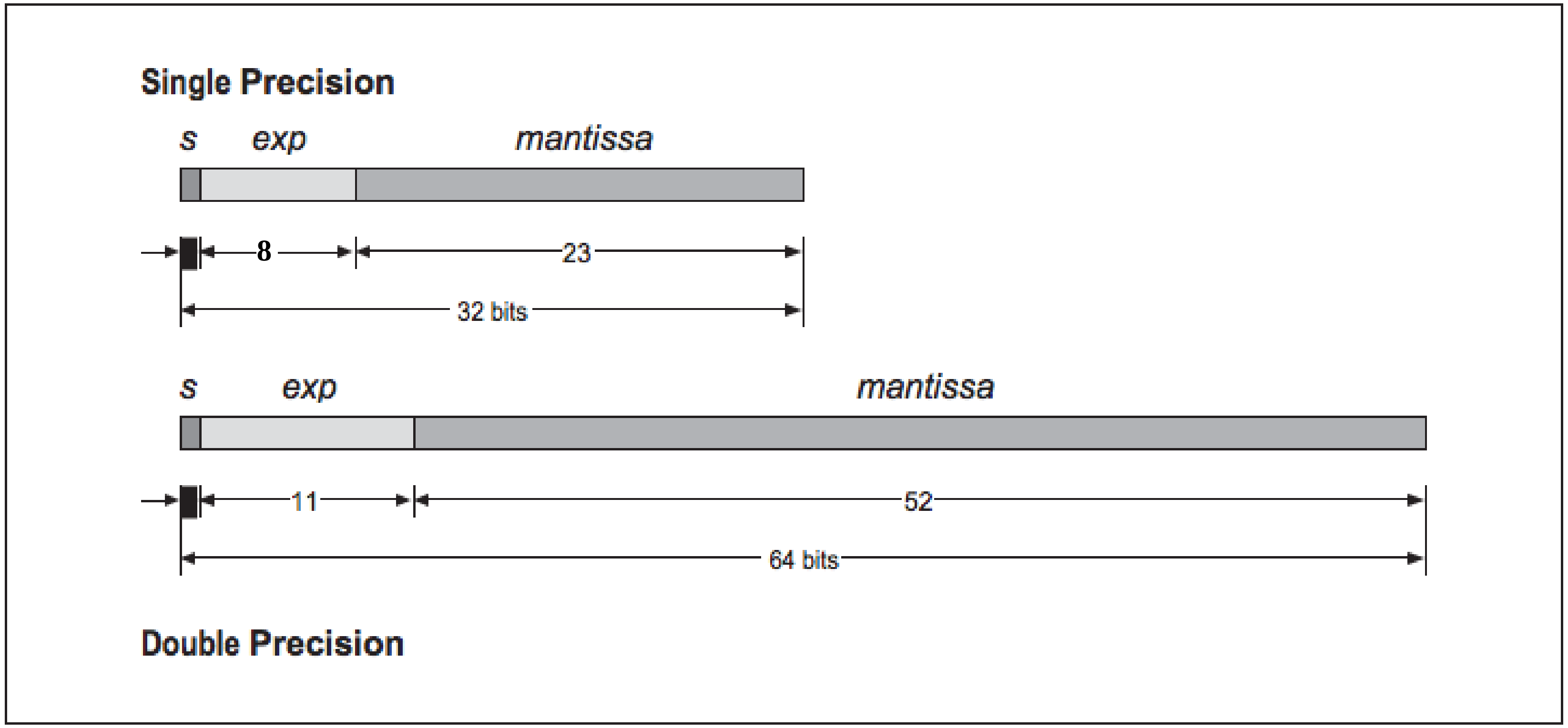
What matters to a CS person?

- When you are adding small numbers to big numbers, the result may not change
  - E.g. $1.00 \times 10^3 + 1.00 \times 10^1 = 1000+1=1.00\times10^3$
- This can be a real problem when writing scientific code.
  - For the above example, imagine you did that addition a million times
  - You'd still have 1000 when the answer should be 1,001,000
- So you need to be aware of the issue.
  - This is why most people use "double" instead of "float"
  - The problem can still exist, it's just less likely.

More precision and range

- We've described IEEE-754 binary32 floating point format, i.e. "single precision" ("float" in C/C++)
  - 24 bits precision; equivalent to about 7 decimal digits
  - $3.4 * 10^{38}$ maximum value
  - Good enough for most but not all calculations
- IEEE-754 also defines larger binary64 format, "double precision" ("double" in C/C++)
  - 53 bits precision, equivalent to about 16 decimal digits
  - $1.8 * 10^{308}$ maximum value
  - Most accurate physical values currently known only to about 47 bits precision, about 14 decimal digits

Single (“float”) precision



Next few lectures: Digital Logic

- Lectures 1-7:
  - LC2K and ARMv8/LEGv8 ISAs
  - Converting C to Assembly
  - Function Calls
  - Linking
- **Today:**
  - **Floating Point**
  - **Combinational Logic**
- Next lecture:
  - Sequential Logic

Up Until Now…

- We've covered high-level C code to an executable
  - Compilation
  - Assembly
  - Linking
  - Loading
- Now, we'll talk about the hardware that runs this code
  - First step: the basics of digital logic

Next 3 Lectures

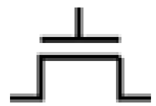
1. **Combinational Logic:**
   - **Basics of electronics; logic gates, muxes, decoders**

2. Sequential Logic
   - Clocks, latches and flip-flops

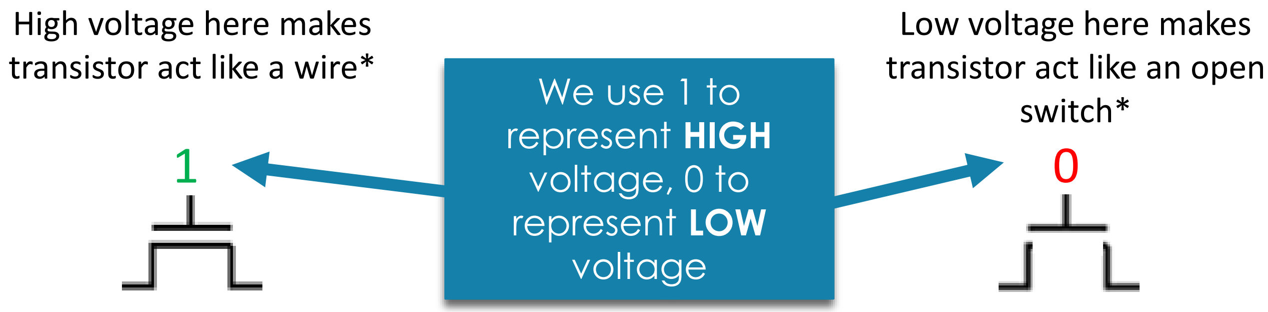
3. State Machines
   - Building a simple processor

Transistors

- At the heart of digital logic is the transistor
- Electrical engineers draw it like this



- The physics is complicated, but at the end of the day, all it is a **really** small and **really** fast electric switch



**Yeah, yeah, circuits people. It's a lot more complicated than that. This abstraction is fine for 370.*

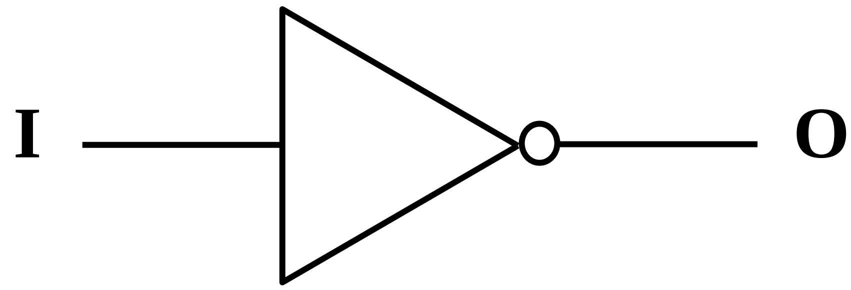
Basic gate: Inverter

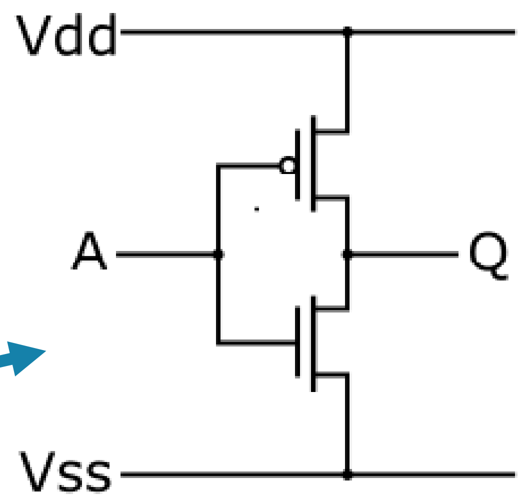
CS abstraction
- logic function

Truth Table

I	O
0	**1**
1	**0**

Schematic symbol (CS/EE)





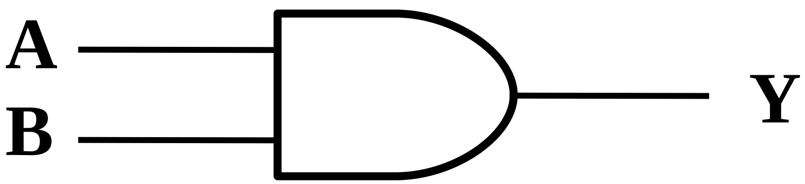
We won't focus on this part in 370. Take 270 and 312 to learn more

Basic gates: AND and OR

AND

Truth Table

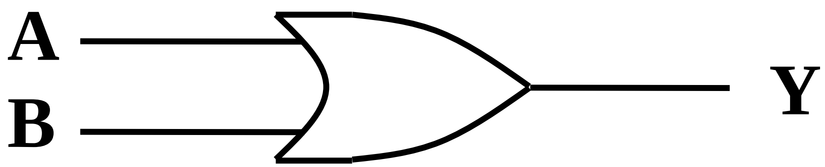
A	B	Y
0	**0**	**0**
0	**1**	**0**
1	**0**	**0**
1	**1**	**1**



OR

Truth Table

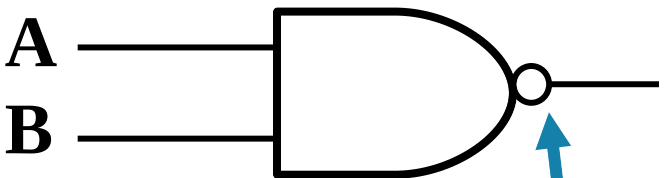
A	B	Y
0	**0**	**0**
0	**1**	**1**
1	**0**	**1**
1	**1**	**1**



Basic gate: NAND

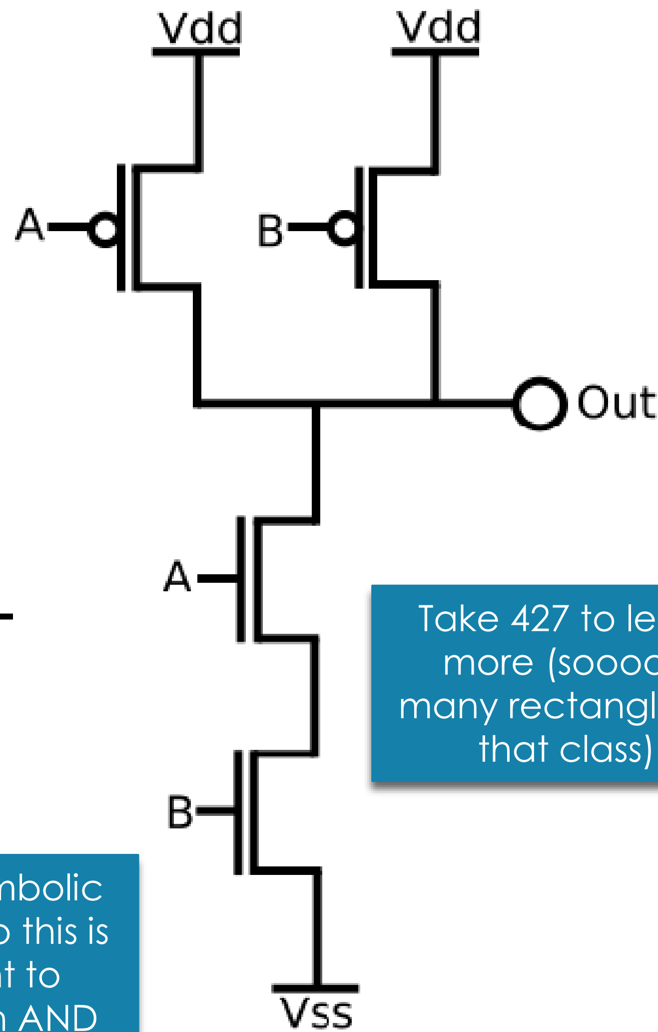
Truth Table

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

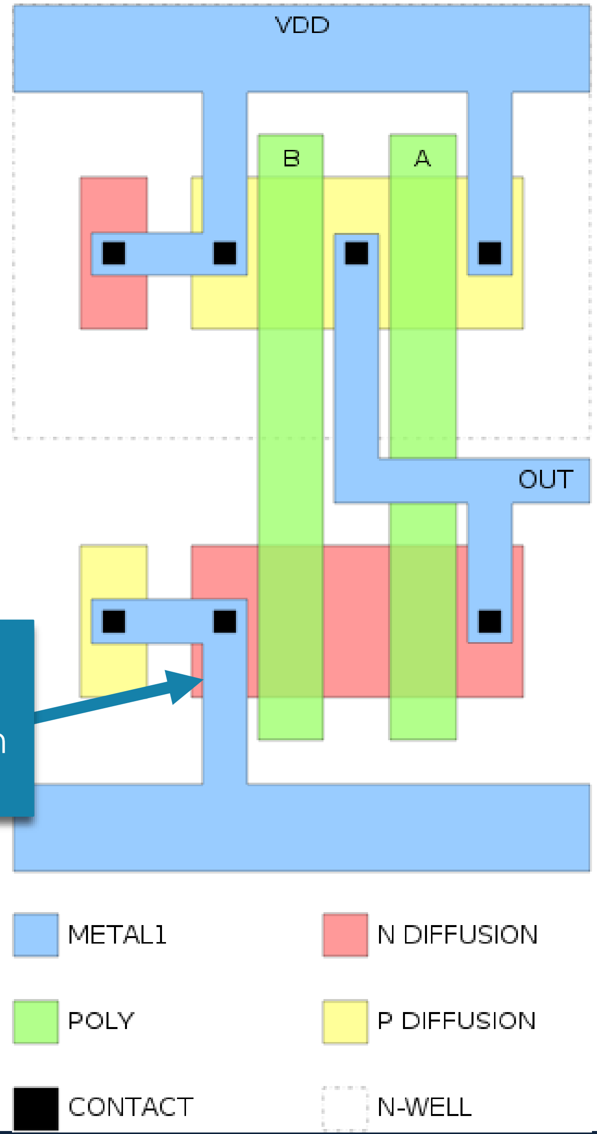


Bubble is symbolic for "invert", so this is equivalent to negating an AND gate

Transistor-level schematic



Layout schematic

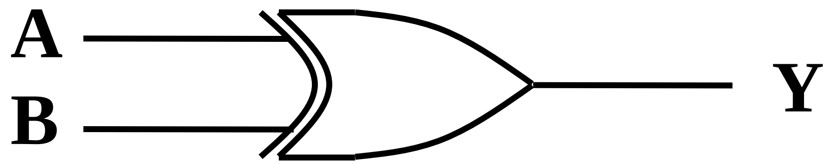


Take 427 to learn more (sooooo many rectangles in that class)

Basic gate: XOR (eXclusive OR)

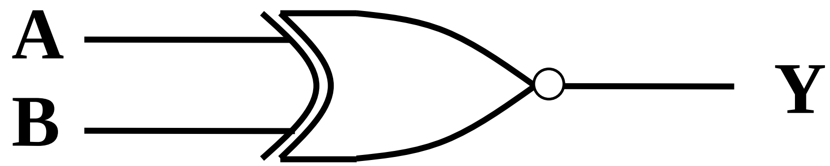
Truth Table

A	B	Y
0	0	
0	1	
1	0	
1	1	



<u>Poll</u>: How do we fill in the truth table for XOR? XNOR?

XNOR Gate



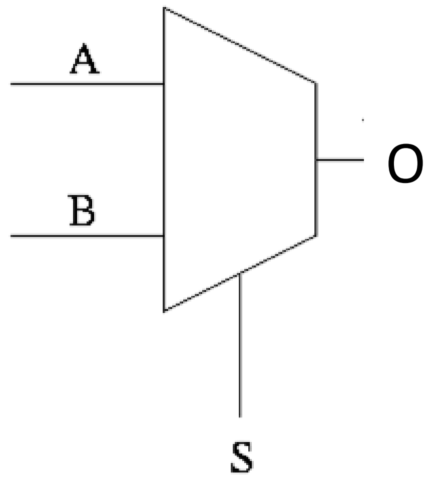
Building Complexity: Selecting

- We want to design a circuit that can select between two inputs (multiplexer or **mux**)
- Let's do a one-bit version
  1. Draw a truth table

A	B	S	O
0	**0**	**0**	
0	**0**	**1**	
0	**1**	**0**	
0	**1**	**1**	
1	**0**	**0**	
1	**0**	**1**	
1	**1**	**0**	
1	**1**	**1**	

Poll: How do we fill in the truth table for this?

Symbol

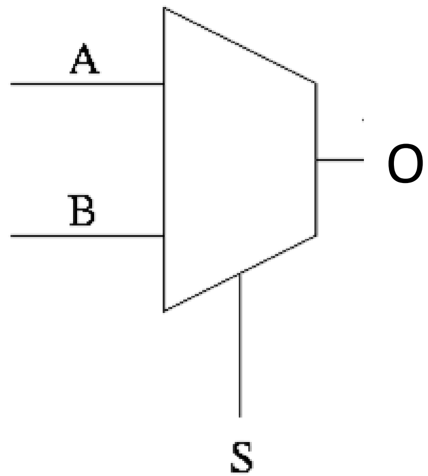


O = S ? B : A

Building Complexity: Selecting

- We want to design a circuit that can select between two inputs (multiplexor or **mux**)
- Let's do a one-bit version
    1. Draw a truth table

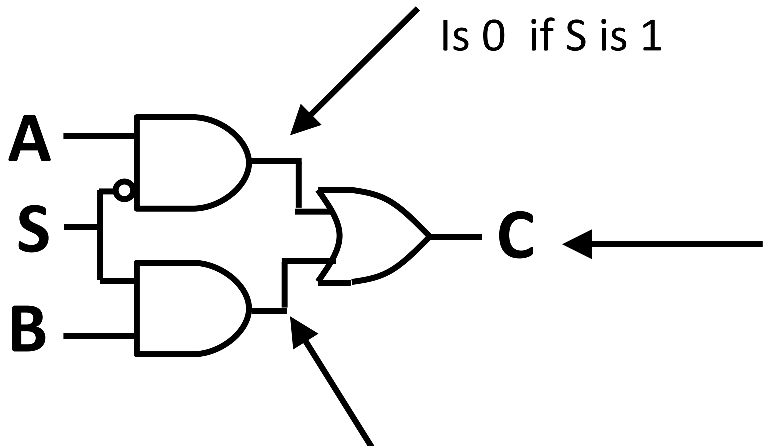
Symbol



A	B	S	O
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**

Circuit

Is A if S is 0
Is 0 if S is 1



O = S ? B : A
Is A if S is 0
Is B if S is 1

Is 0 if S is 0
Is B if S is 1

Muxes are universal! A 2^N entry truth table can be implemented by passing each ouput value into an input of a 2^N–to-1 mux

Building Complexity: Addition

- We want to design a circuit that performs binary addition
- Let's start by adding two bits
  - Design a circuit that takes two bits (A and B) as input
    - Generates a sum and carry bit (S and C)

1. Make a truth table
2. Design a circuit

$$\begin{array}{r} 10011 \\ +\,00110 \\ \hline \end{array}$$

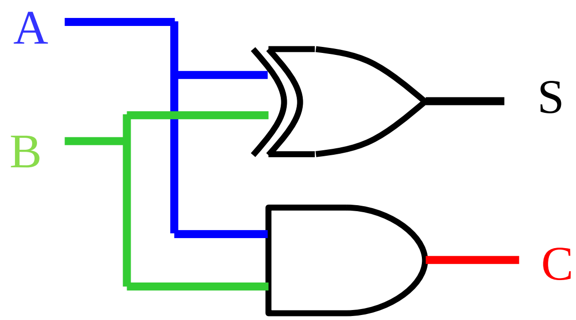
A	B	C	S

Building Complexity: Addition

- We want to design a circuit that performs binary addition
- Let's start by adding two bits
  - Design a circuit that takes two bits (A and B) as input
    - Generates a sum and carry bit (S and C)

1. Make a truth table
2. Design a circuit

```
 0 1 1 0
 1 0 0 1 1
+0 0 1 1 0
----------
 1 1 0 0 1
```

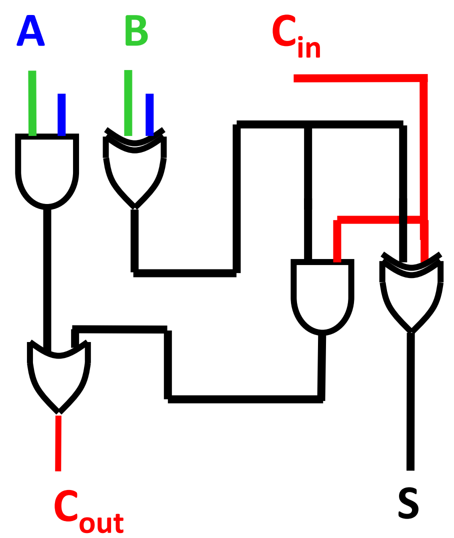


A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Building Complexity: Addition

- Now we can add two bits, but how do we deal with carry bits?
- This is a **full adder**
  - We have to design a circuit that can add three bits
    - Inputs: A, B, Cin
    - Outputs: S, Cout
  1. Design a truth table
  2. Circuit
- This is a **full adder**

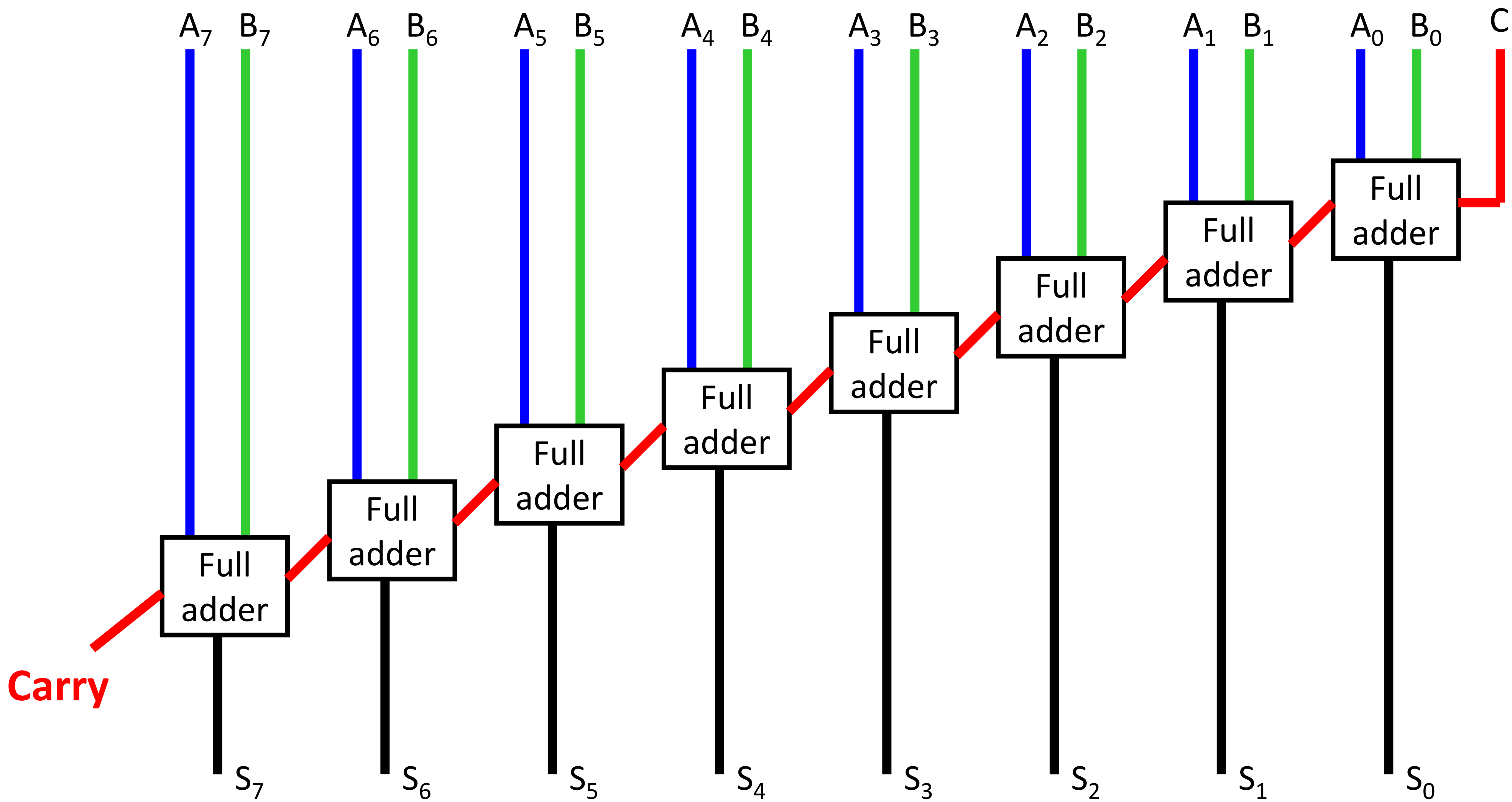
```
  0 1 1 0
  1 0 0 1 1
+ 0 0 1 1 0
-----------
  1 1 0 0 1
```



Cin	A	B	Cout	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

8-bit Ripple Carry Adder

If we invert B's bits and set C to 1, we also have a subtractor! Why?

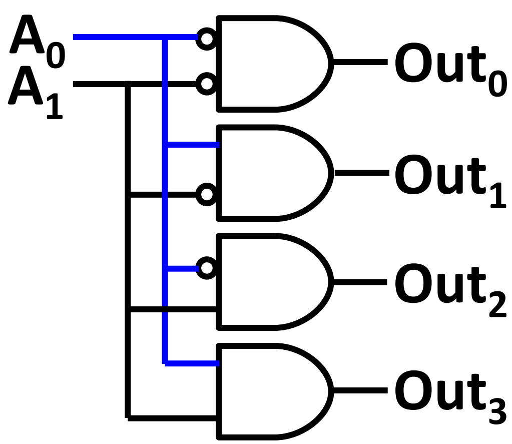


This will be very slow for 32 or 64 bit adds, but is sufficient for our needs

Building Complexity: Decoding

- Another common device is a decoder
  - Input: N-bit binary number
  - Output: 2^N bits, exactly one of which will be high
  - Allows us to **index** into things (like a register file)

Decoder

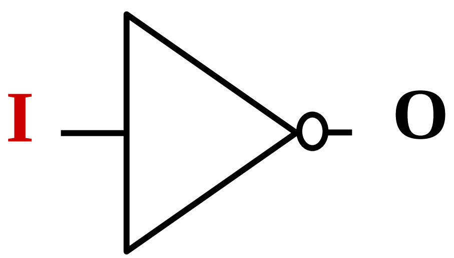


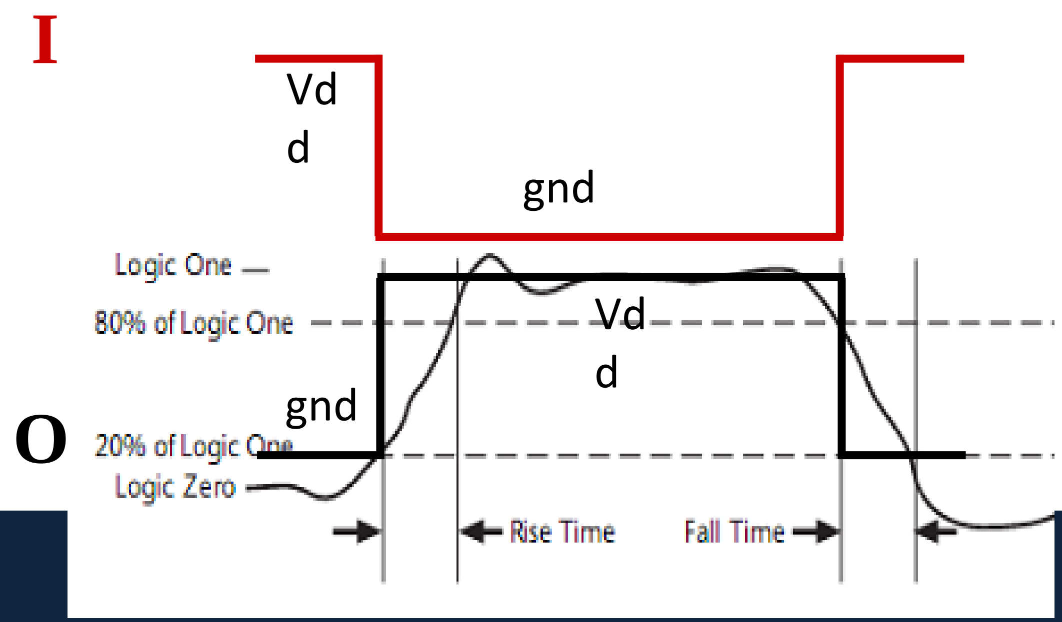
Poll: What will be the output for 101?

Propagation delay in combinational gates

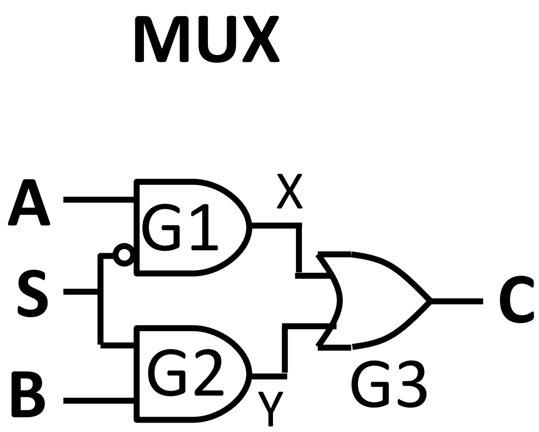
- Gate outputs do not change exactly when inputs do.
  - Transmission time over wires (~speed of light)
  - Saturation time to make transistor gate switch

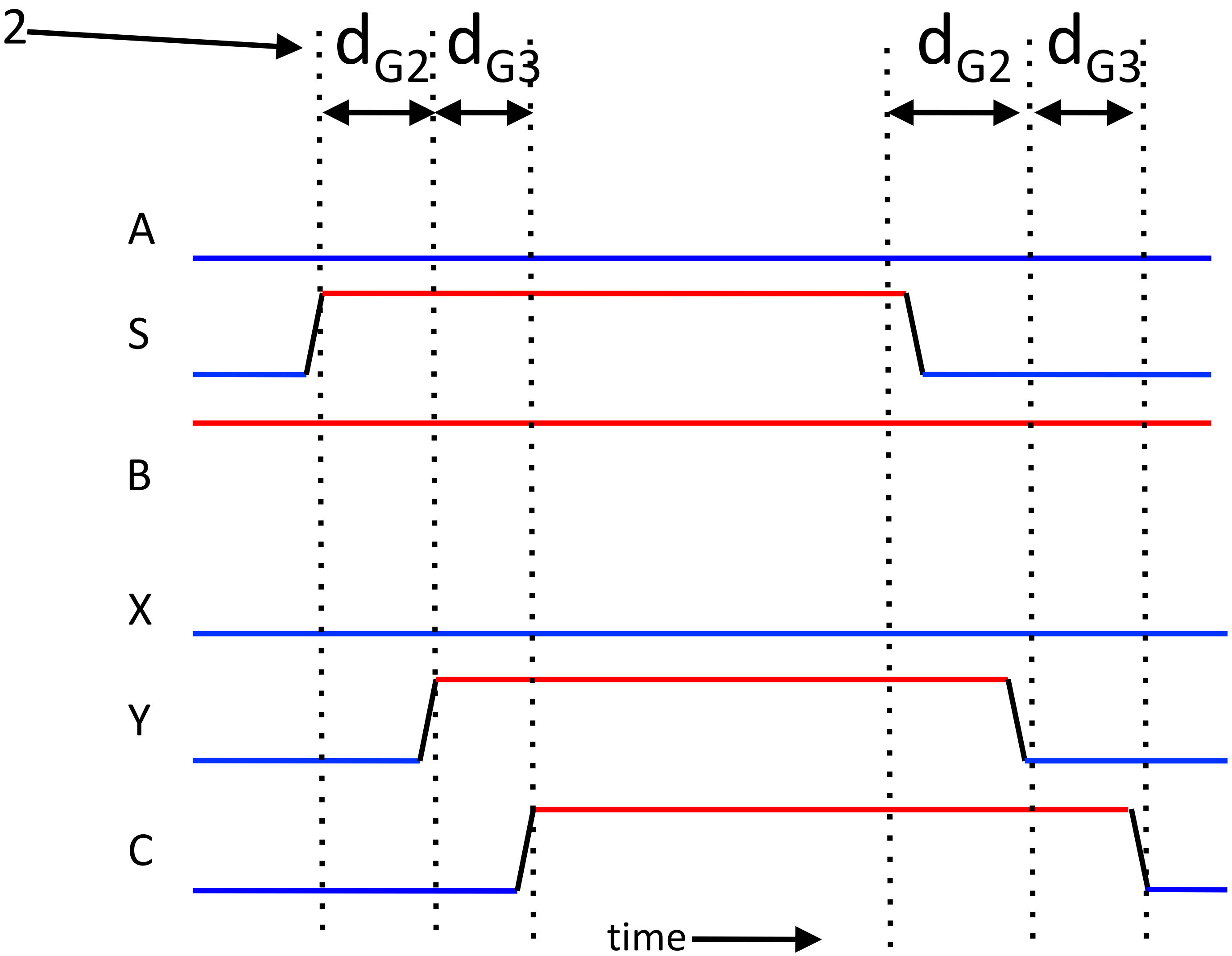
⇒ Every combinatorial circuit has a propagation delay
(time between input and output stabilization)





Timing in Combinational Circuits



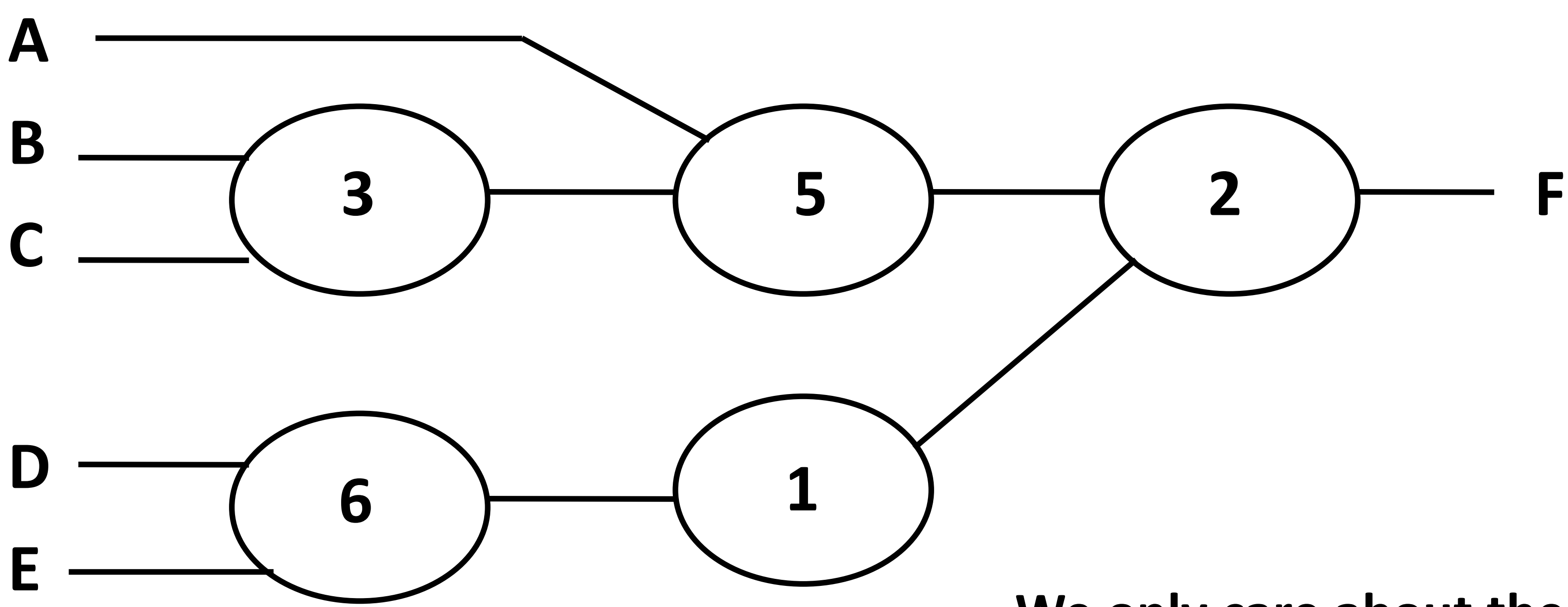


What is the input/output delay (or simply, delay) of the MUX?

What is the delay of this Circuit?

Each oval represents one gate, the type does not matter

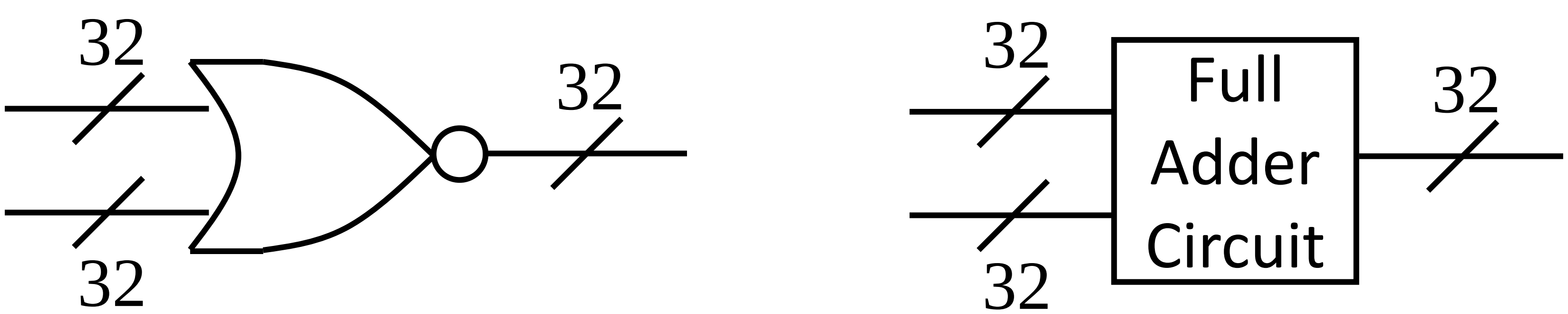
<u>Poll</u> : What is the delay?



We only care about the longest path, or <u>critical path</u>

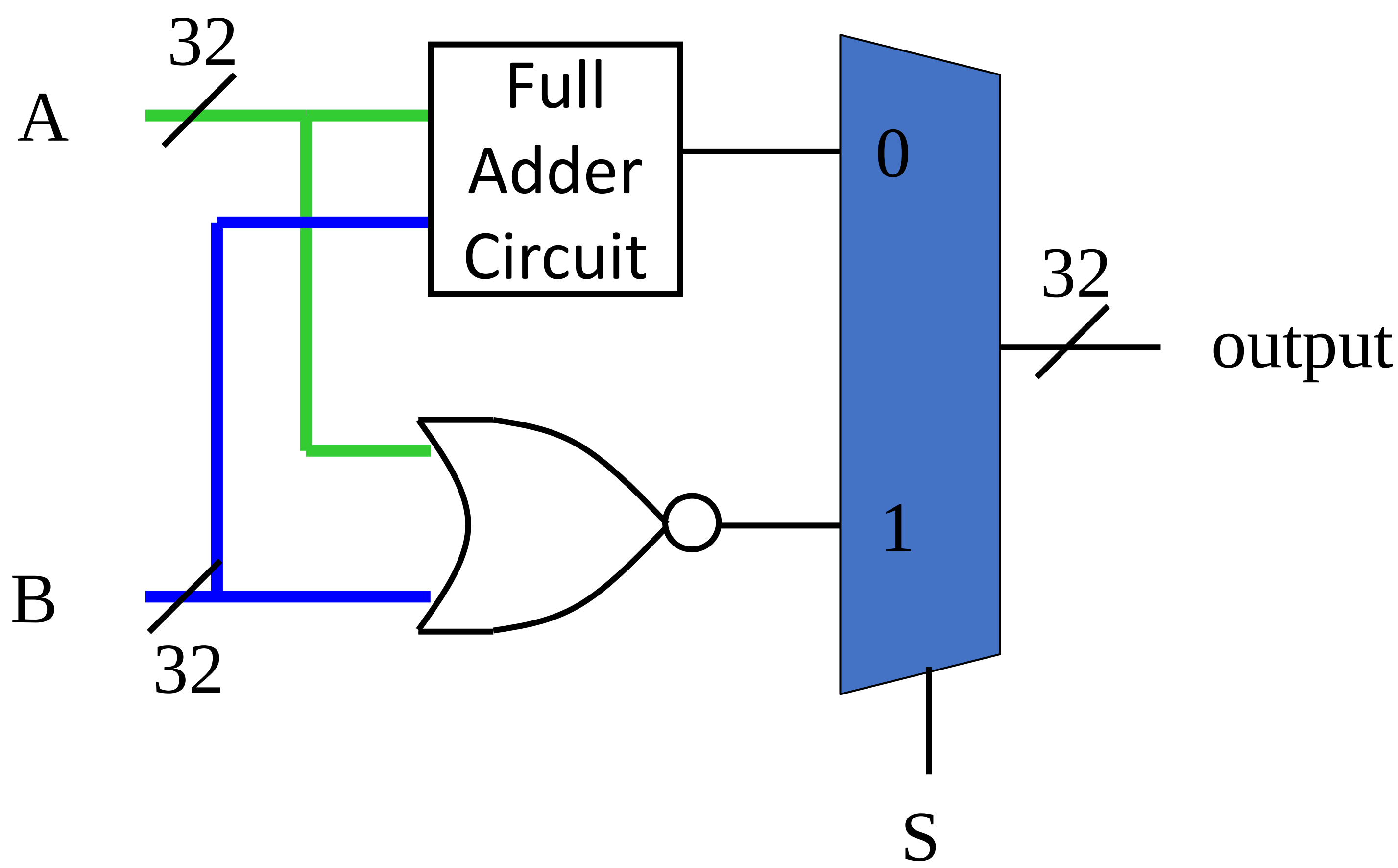
Exercise

- Use the blocks we have learned about so far (full adder, NOR, mux) to build this circuit
  - Input A, 32 bits
  - Input B, 32 bits
  - Input S, 1 bit
  - Output, 32 bits
  - When S is low, the output is A+B, when S is high, the output is NOR(a,b)
- Hint: you can express multi-bit gates like this:



Exercise

- This is a basic ALU (Arithmetic Logic Unit)
- It is the heart of a computer processor!



Next Time

- Logic circuits that "remember"
  - Aka "sequential logic"
- Lingering questions / feedback? I'll include an anonymous form at the end of every lecture: https://bit.ly/3oXr4Ah

BONUS Floating Point Slides

Bonus slides – this material is not testable

Not testable

- This material is here for those folks that may care.
  - **You *may* find it useful when considering the gap between representations**
  - But the material isn't directly testable.
- It is interesting if you are into that kind of thing.
- It can be useful if you are going to do scientific programming for a living.
- So it is provided as a reference, but isn't part of the class (we may cover a bit of it in lecture if we have time)

Not testable

Floating point multiplication

- Add exponents (don't forget to account for the bias of 127)
- Multiply significands (don't forget the implicit **1** bits)
- Renormalize if necessary
- Compute sign bit (simple exclusive-or)

Floating point multiply

Not testable

$$10.625_{10} = 1010.101_2$$

$$10_{10} = 1010_2$$

0	10000010	01010100000000000000000
	+	×
0	10000010	01000000000000000000000
	-127	
0	10000101	10101001000000000000000

$$
\begin{array}{r}
1\,0\,1\,0\,1\,0\,1 \\
\times \qquad 1\,0\,1 \\
\hline
1\,0\,1\,0\,1\,0\,1 \\
1\,0\,1\,0\,1\,0\,1\,0\,0 \\
\hline
1\,1\,0\,1\,0\,1\,0\,0\,1
\end{array}
$$

$$1101010.01_2 = 106.25_{10}$$

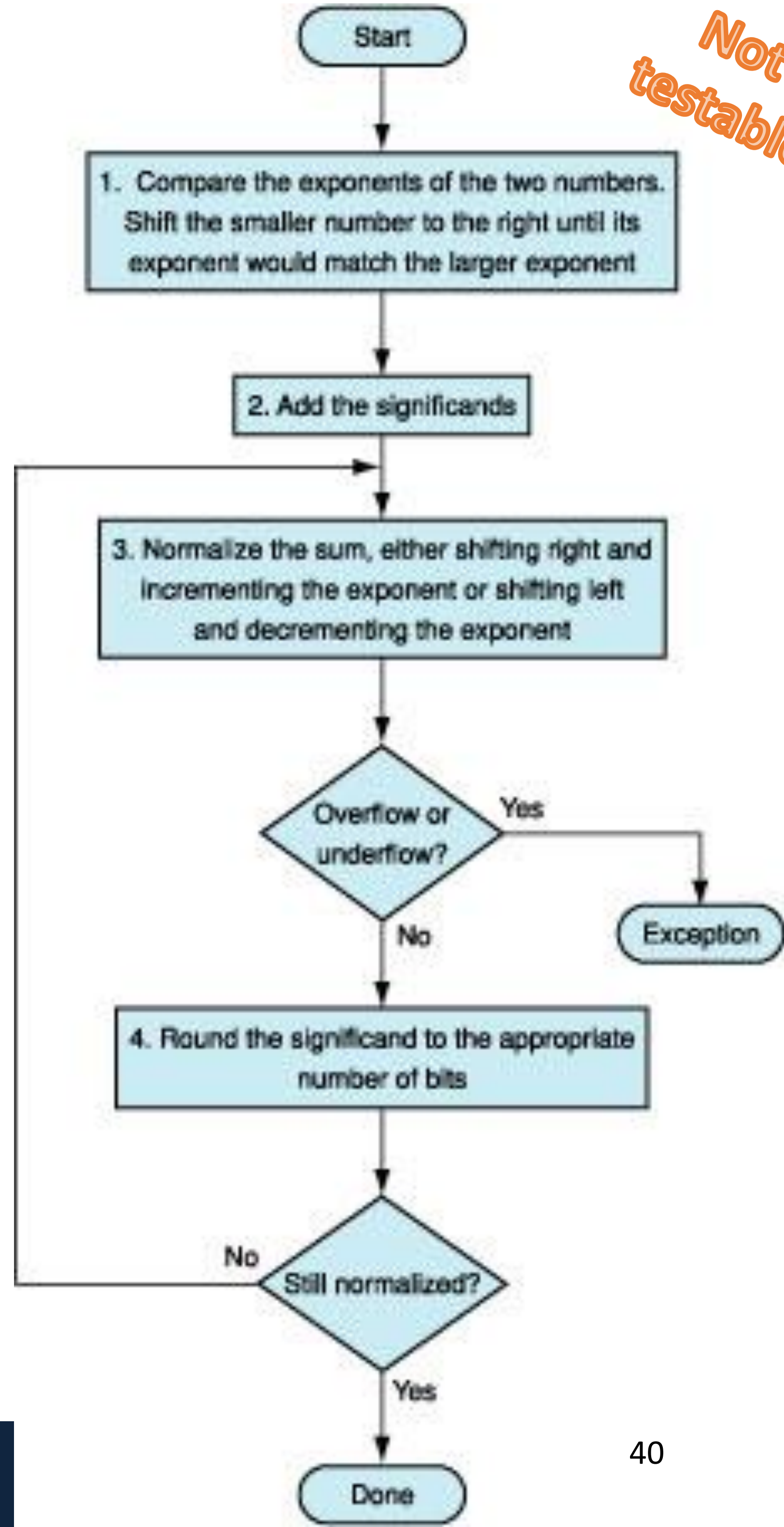
Floating point addition

Not testable

- More complicated than floating point multiplication!
- If exponents are unequal, must shift the significand of the smaller number to the right to align the corresponding place values
- Once numbers are aligned, simple addition (could be subtraction, if one of the numbers is negative)
- Renormalize (which could be messy if the numbers had opposite signs; for example, consider addition of +1.5000 and – 1.4999)
- Added complication: rounding to the correct number of bits to store could denormalize the number, and require one more step

Floating point Addition

1. Shift smaller exponent right to match larger.
2. Add significands
3. Normalize and update exponent
4. Check for "out of range"



Not testable

Class Problem

Show how to add the following 2 numbers using IEEE floating point addition: 101.125 + 13.75

Not testable

Class Problem

101.125	0	10000101	10010100100000000000000
13.75	0	10000010	10111000000000000000000

Shift by 6-3 = 3

Shift mantissa by difference in exponent

00110111000000000000000

Note: When shifting to the right, the first shift should put the implicit 1, then 0's

Sum Significands

```
  1 1 0 0 1 0 1 0 0 1
+ 0 0 0 1 1 0 1 1 1 0
---------------------
  1 1 1 0 0 1 0 1 1 1
```

Sum didn't overflow, so no re-normalization needed

0	10000101	11001011100000000000000

= 114.875

Class Problem

Not testable

Show how to add the following 2 numbers using IEEE floating point addition: 117.125 + 13.75

Not testable

Class Problem

117.125	0	10000101	11010100100000000000000
13.75	0	10000010	10111000000000000000000

Shift by 6-3 = 3

Shift mantissa by difference in exponent

00110111000000000000000

Note: When shifting to the right, the first shift should put the implicit 1, then 0's

Sum Significands

```
  1 1 1 0 1 0 1 0 0 1
+ 0 0 0 1 1 0 1 1 1 0
---------------------
1 0 0 0 0 0 1 0 1 1 1
```

0	10000110	00000101110000000000000

= 130.875

Sum overflows, re-normalize by adding one to exponent and shifting mantissa by one