CS/COE 0447

Plexers and Simple Logic Minimization

wilkie (with content borrowed from:

Jarrett Billingsley

Dr. Bruce Childers)

Exam:

- Pretty good.
- Half the class got an A.
  - Lenient grading to avoid some of your F's meant lots of A's. Boo.
  - Exam 2 is generally harder, in a fashion, imo, so I'm not worried. ☺

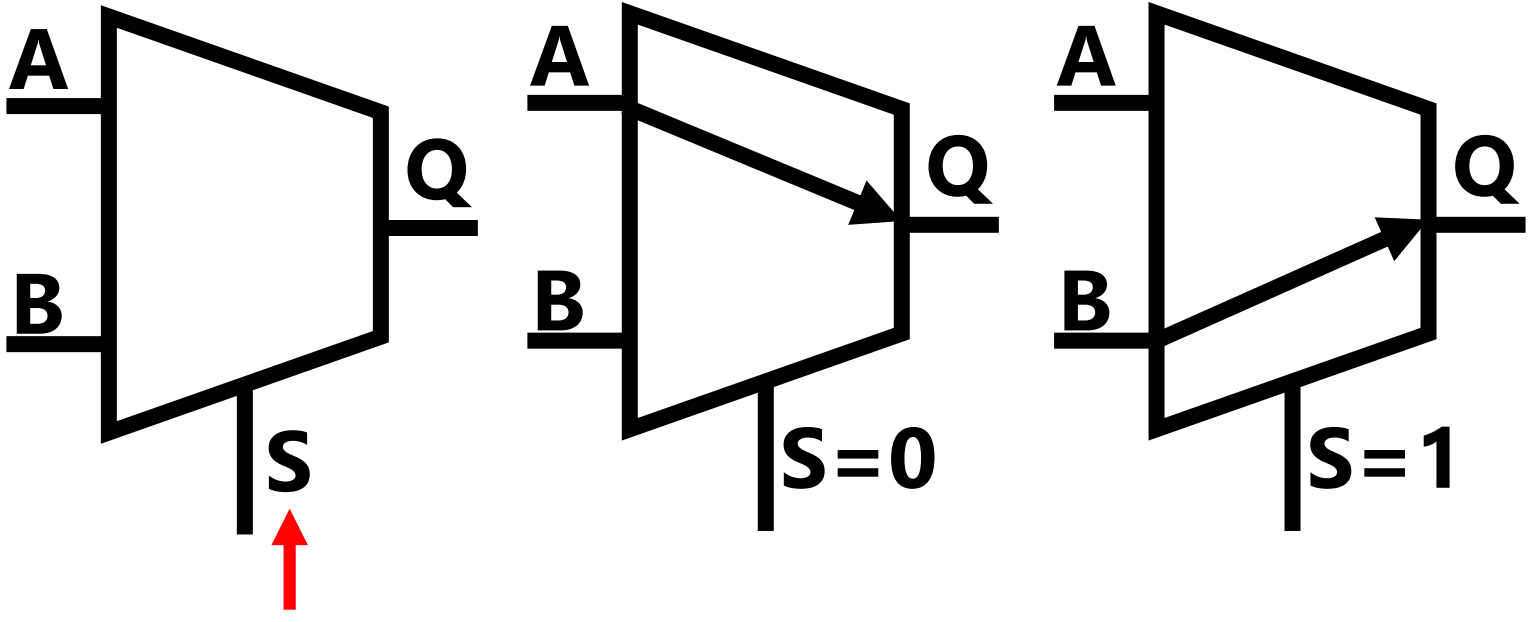
- Some notes:
  - $2^0 = 1$ (not 0!! I'm a tad concerned!!)
  - The smallest unsigned integer is 0, naturally. ($2^0 - 1$, if you must)
  - Some of you said largest is: $2^{10}+2^9+2^8+2^7+2^6+2^5+2^4+2^3+2^2+2^1+2^0$
    - True! But that's the same as $2^{11} - 1$ so I hate it. ☺ (but you get credit anyway, <u>but not on the final</u>)

Muxes, Demuxes, and Decoders

When you need particular data

Hardware that makes decisions

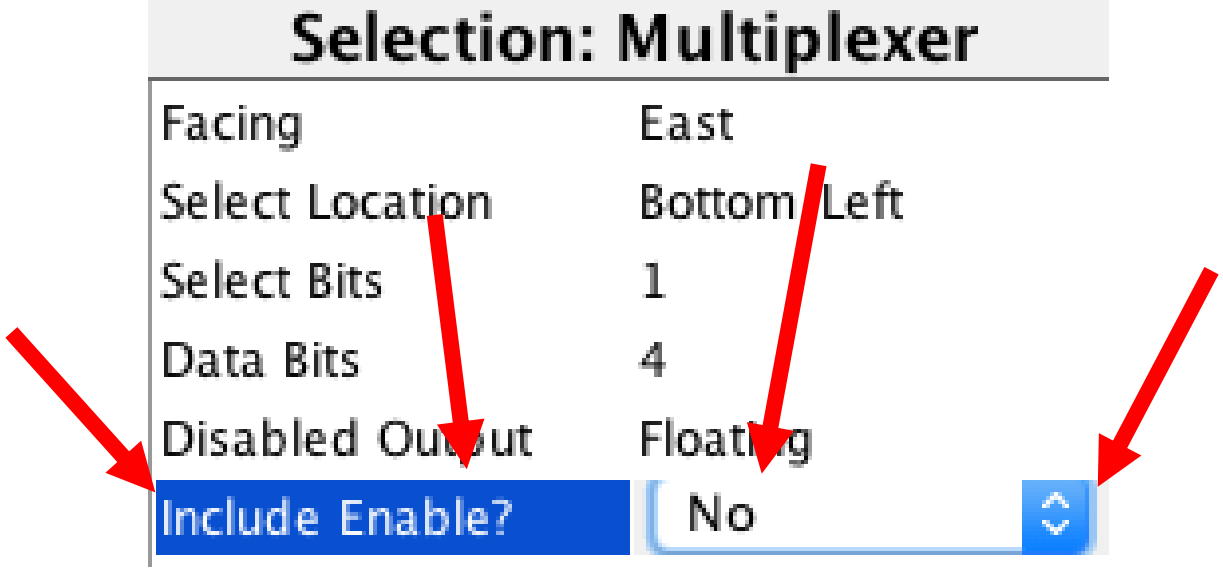
- A **multiplexer** (**mux**) outputs one of its inputs based on a **select.**

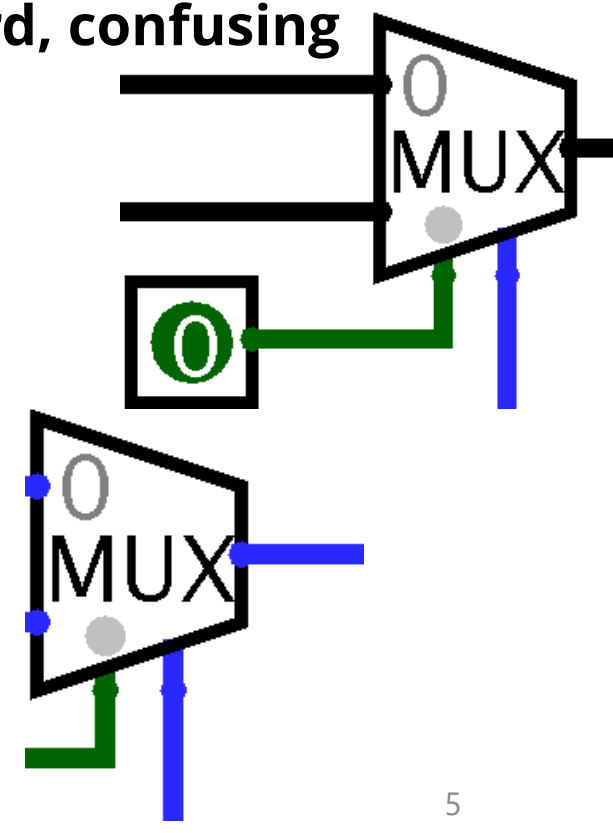


This is the select input.

What's that enable input?

- If you don't understand tristate buses or high impedance states, **turn off the *enable* input.**
- If you ever see **blue wires**, **you are in weird, confusing territory.**
- if you know this stuff, fine, but otherwise…





Doing everything and throwing most of it away

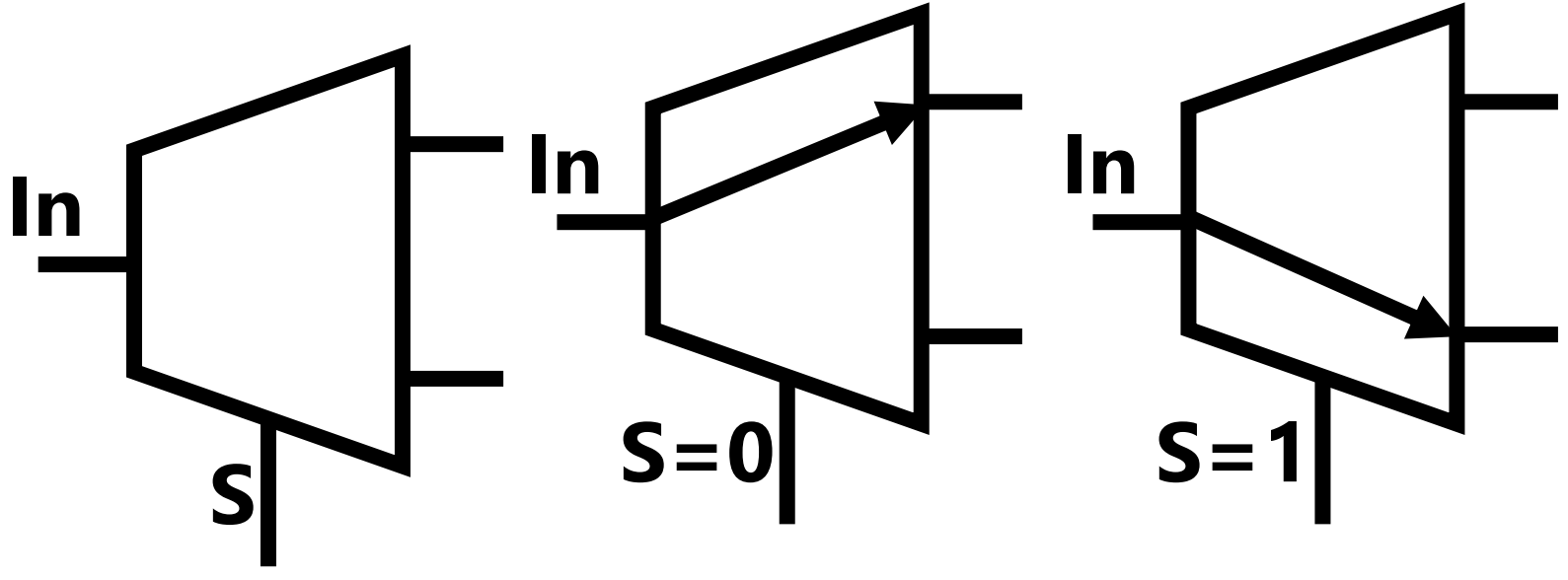
- I want a circuit that does this:

```
if(select == 1)
    output = A - B
else
    output = A + B
```

- Let's see what that looks like
- A mux is like a **hardware if-else statement**
- But unlike in software…
  - The "condition" comes at the "end" (the output)
  - Instead of doing one *or* the other, we **do both, choose the one that we care about, and ignore the rest!**

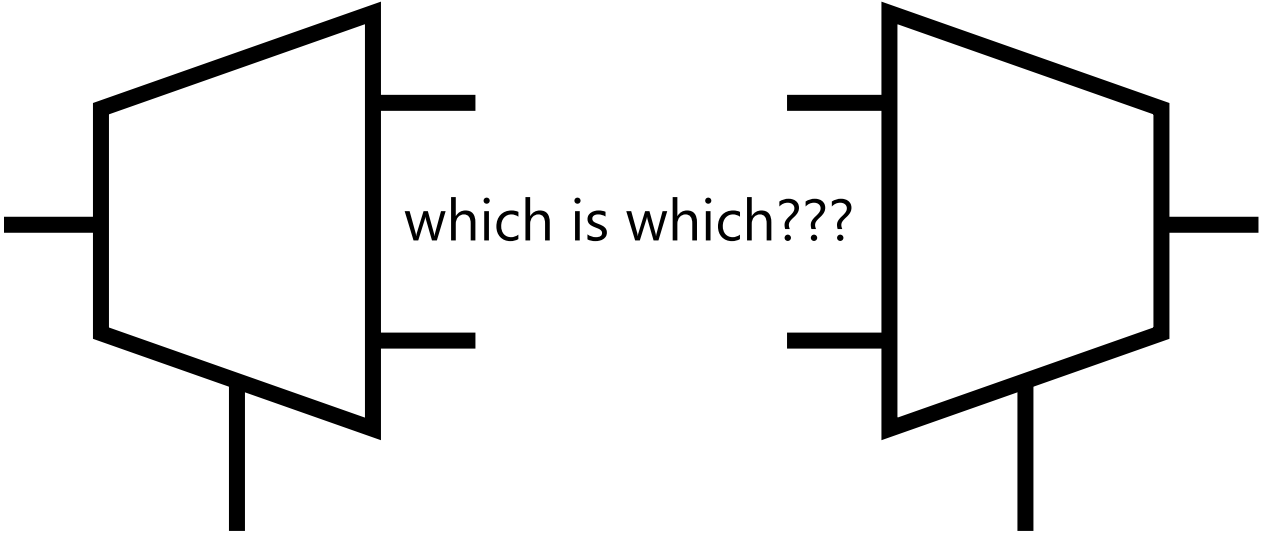
Demultipliexers

- A **demux** does the opposite: it sends its input to one of its outputs
- **The rest of the outputs are 0s**



Looking in a mirror

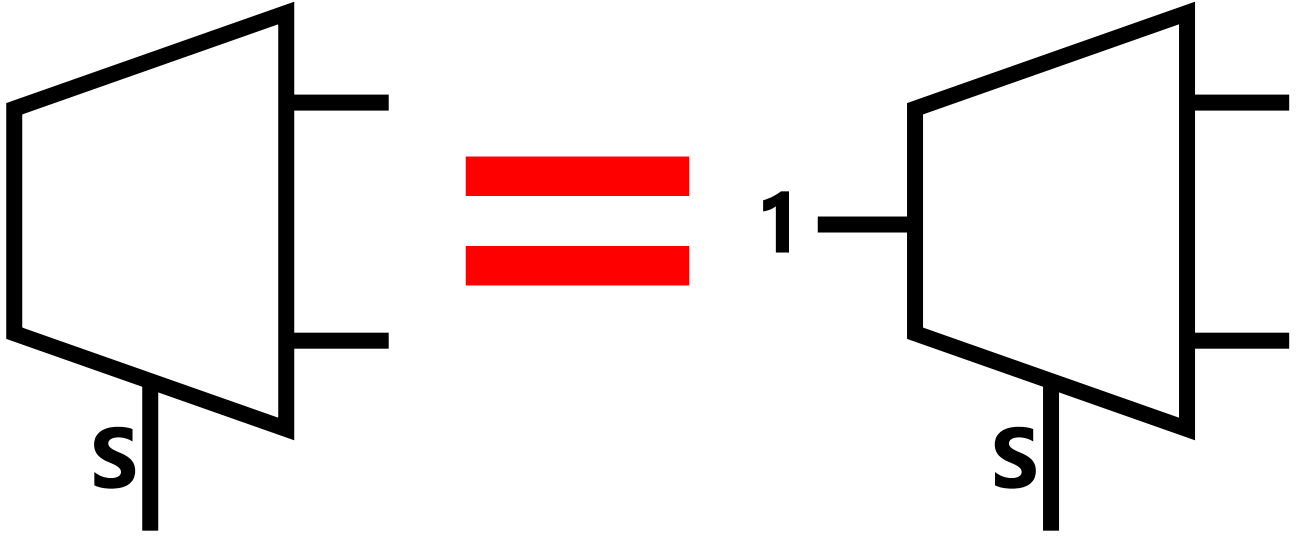
- it can be confusing if all you see is this:



Logisim distinguishes these, and
I'll try to too, with arrows

Decoders

- A **decoder** is like a 1-bit demux whose input is always 1



exactly one output is 1,
and the rest are 0s

Uses for decoders and demuxes

- Unless you're using tristate (blue) wires, they're not too useful...

- They let us create data buses and such.
  - Lets many stateful components share data wires, essentially.
  - It's a nice optimization, but we don't *have to* do this.

- Most of the time, you don't have to "direct" a signal to a location.
  - Instead, you **hook up the inputs to everything that needs them**.

- Since these are more about encapsulating state, we'll use them more when we get to **sequential logic**.

Combinational vs Sequential Logic

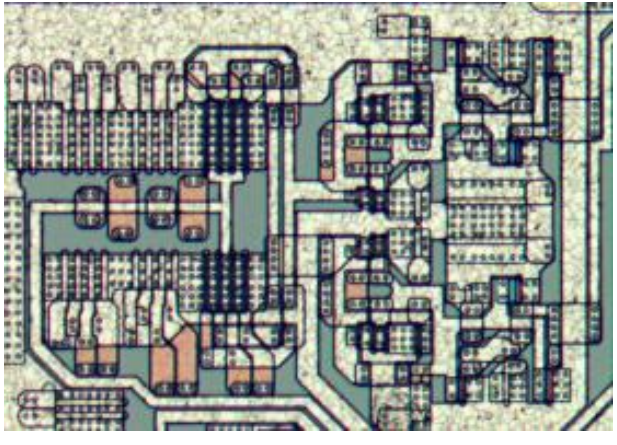
- **Combinational logic:** the outputs of a circuit depend entirely on their ***current*** inputs
  - AND, OR, NOT, XOR gates
  - adders
  - muxes, demuxes, and decoders
  - Basically: It converts an input to an output in one go.
- **Sequential logic** is coming up soon
  - the outputs can depend on the **current *and previous*** inputs
  - it *remembers*
  - Basically: It actually computes: the process changes the state of the machine, which in turn changes *how* it converts input to output.
  - Built from sections of combinational logic.
- Logic minimization techniques only work on **combinational** logic!
  - …or combinational *pieces* of a larger sequential circuit

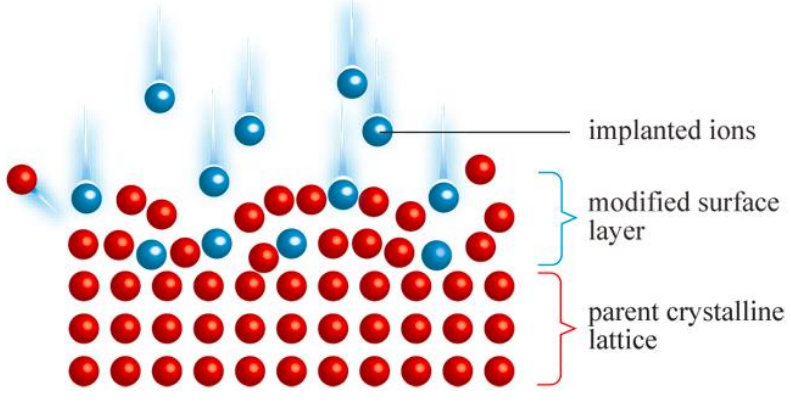
How ICs are made

(another For Fun™ section)

How ICs (integrated circuits) are made

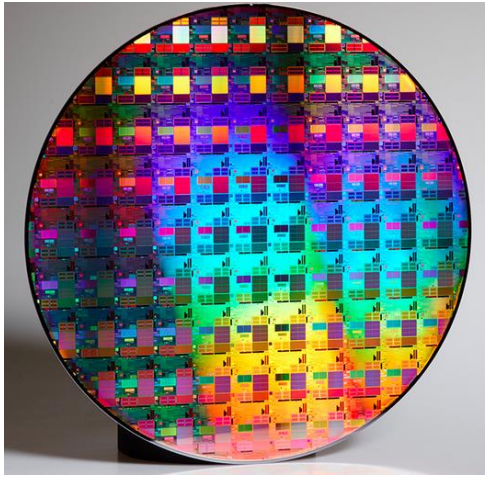
- silicon is purified and grown into a **monolithic crystal** *(extremely expensive)*
- this is sliced thinly to make **wafers**
  - gooey caramel is put between them to make **stroopwafel**
- a series of complicated photochemical processes do things like:
  - change its electrical properties
  - make wires to connect things
  - make inert insulating layers

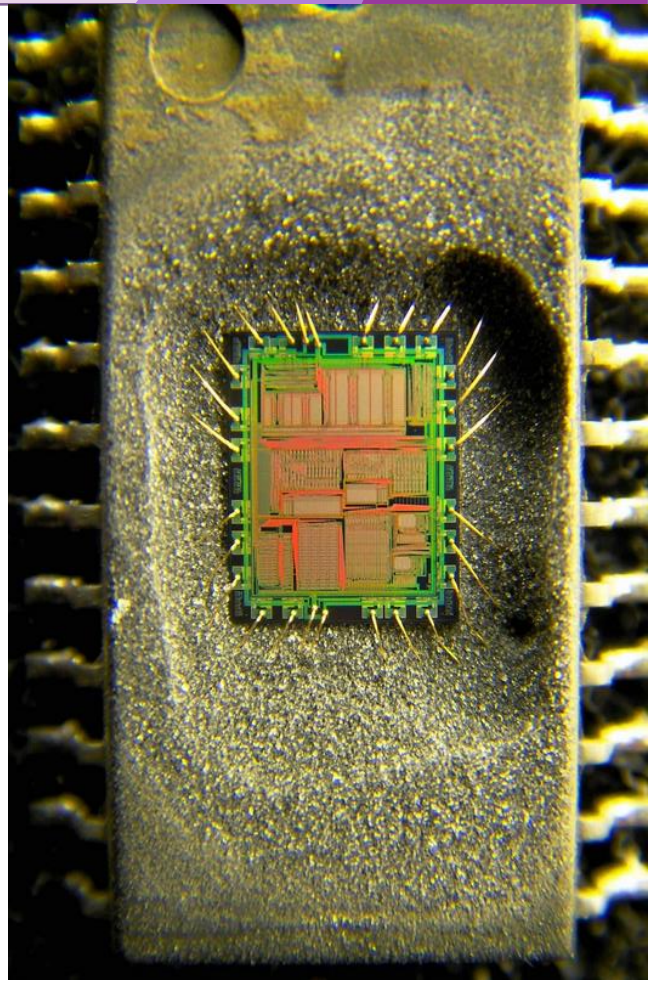




How ICs (integrated circuits) are made

- many ICs are printed on one wafer
- the wafers are **diced** (chopped up)
- the ICs are **tested**
- the ICs are **mounted** in a package →
- they're tested *again*
- *then* they're ready to sell





Manufacturing yield

- ICs are **tiny and complex**
- silicon crystals can have **defects**
- a tiny speck of dust during production can **ruin an entire chip**
- the **yield** is the **percentage of usable chips**
  - **bigger** chips have **smaller** yields: more opportunities for mistakes!
- the **size of the silicon** is **the biggest factor** in the price of an IC
  - huge ICs (several *cm* on each side!), such as very high-resolution camera sensors, can cost **tens of thousands of dollars!**
- manufacturers can also **bin** resulting chips
  - bug-free ones can be sold as Core i7s for *lotsa money*
  - slightly malformed ones can be sold as Core i5s and i3s
  - the ones they sweep off the floor are the Celerons

Logic Minimization

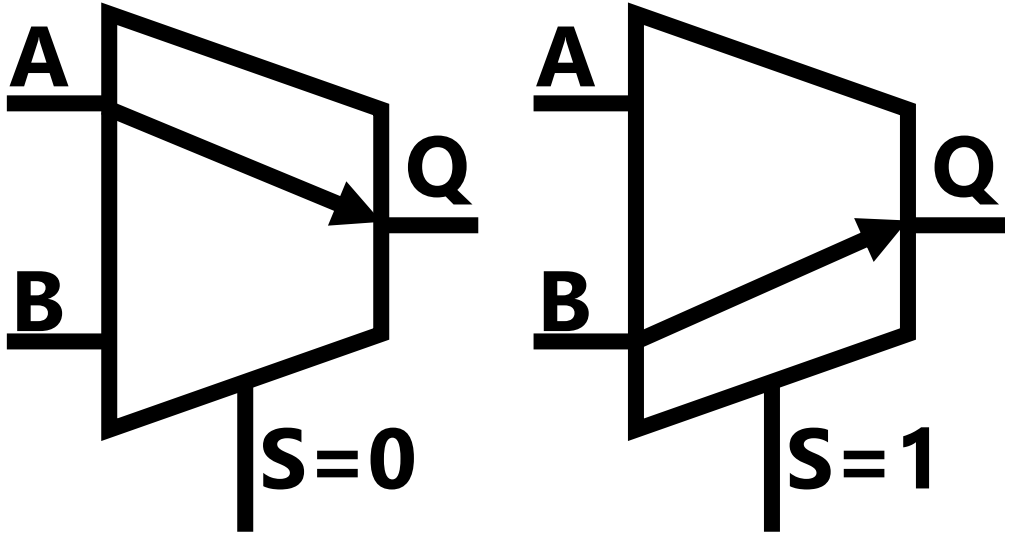
Making Smaller Circuits

Silicon is expensive, rocks are slow

- **logic minimization** means using the **smallest number** of gates/transistors possible to implement a boolean function
  - a **boolean function** is anything we've talked about
  - it has boolean inputs, and boolean outputs
- fewer transistors means:
  - smaller area:
    - **cheaper chips!**
    - **more stuff** on one chip!
    - **smaller chance** of manufacturing **defects!**
  - less gate delay:
    - **faster circuits!**

A first try

- Let's make a truth table for a **two-input 1-bit multiplexer.**



S	A	B	Q
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

I Couldn’t Even Care One Bit

- We don't even *care* about the *un*selected input
- We can put **X** in the inputs we don't care about
- We call these **don't cares**
  - yep, really
- What these mean is:
  - when we make this into a boolean function, we can **ignore** those inputs
    - we won't even need to write em

S	A	B	Q
0	0	X	0
0	0	X	0
0	1	X	1
0	1	X	1
1	X	0	0
1	X	1	1
1	X	0	0
1	X	1	1

Making a Boolean Expression

- Let's make a boolean expression from the truth table:
  1. find all rows with an **output of 1**
  2. for each one, write an AND of all the inputs, with NOTs on the 0s
  3. eliminate duplicate terms
  4. OR the remaining terms together

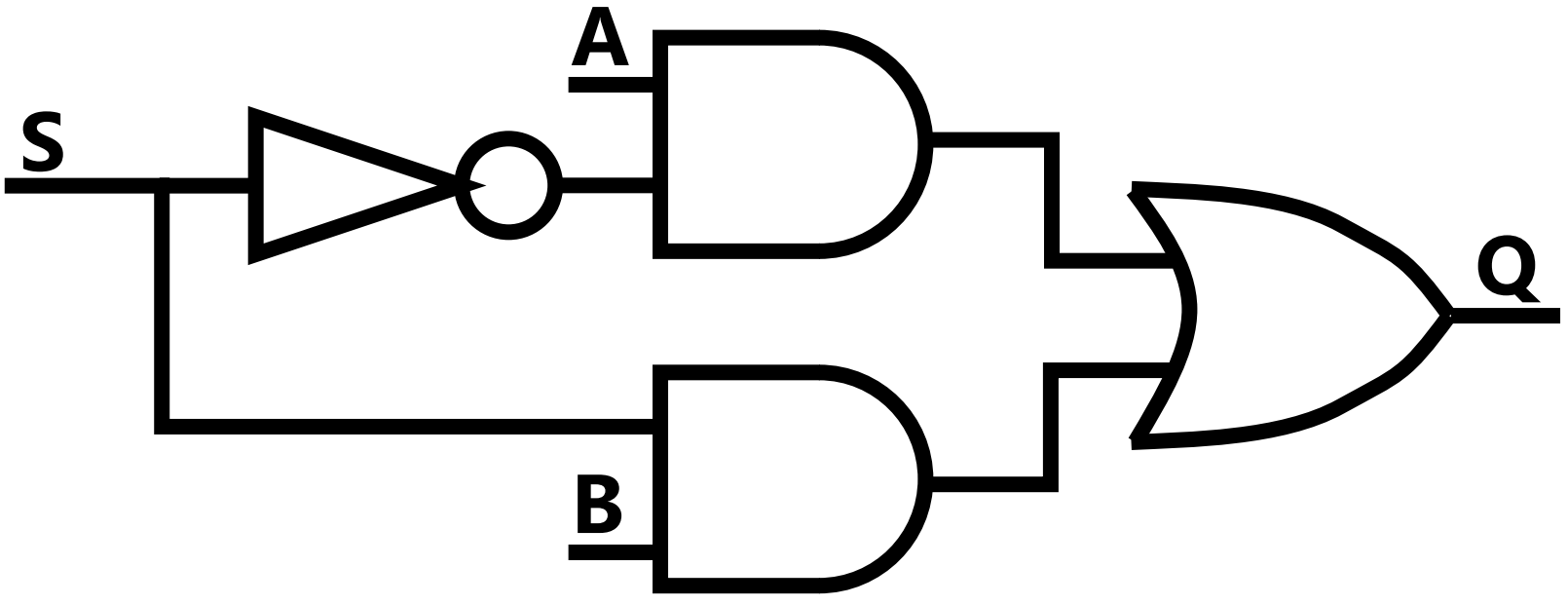
$$\mathbf{Q = \bar{S}A + SB}$$

- This is what we call a **sum-of-products**
  - An OR of multiple ANDed terms

	S	A	B	Q
	0	0	X	0
	0	0	X	0
$\bar{S}A$	0	1	X	1
~~$\bar{S}A$~~	0	1	X	1
	1	X	0	0
SB	1	X	1	1
	1	X	0	0
~~SB~~	1	X	1	1

Turning that expression into gates

- making $\mathbf{Q = \bar{S}A + SB}$ into gates is pretty straightforward:



Limitations of the last method

- If we use that method on the C_o of a full adder:

$$C_{out} = \bar{A}BC_{in} + A\bar{B}C_{in} + AB\overline{C_{in}} + ABC_{in}$$

A	B	C_{in}	C_{out}	
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	1	$\bar{A}BC_{in}$
1	0	0	0	
1	0	1	1	$A\bar{B}C_{in}$
1	1	0	1	$AB\overline{C_{in}}$
1	1	1	1	ABC_{in}

- It feels too complex, somehow
- Each NOT gate uses 2 transistors
- Each AND/OR gate uses 6
- This will use **72 transistors**
  - *just* for the carry
  - of *one* one-bit addition
  - Hmmmmmmmmmmmmmmmm mmmmmmmmmmmmmmmmmmm mmmmmmmmmmmmmmmmmmm mmmmmmmmmmm next time