Computer Science 324
Computer Architecture
Mount Holyoke College
Fall 2009

Topic Notes: Digital Logic

Our goal for the next couple of weeks is to gain a reasonably complete understanding of how we can start with basic transistor technology and build up through added complexity and abstractions to build all of the components we need to build a computer that will execute machine code such as that of the MIPS ISA. We will visit many of the topics most of you have seen previously, but will go into more depth on some, and spend time implementing these circuits in the digital logic lab.

Basic Physics

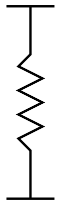
At the lowest level, today's computers are just very complex electrical circuits.

We will only look at the most basic ideas from physics to describe how some of the basic digital logic building blocks can be constructed.

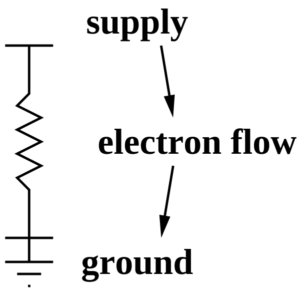
Resistors

In nature, electrical potential wants to equalize. To maintain a potential, the electrons must be separated by an insulating material. A conductive material will allow the potential to equalize.

In an electrical circuit, we place a *resistor* to establish a potential difference between points.

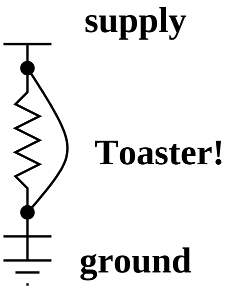


In a circuit, the electrons want to go from a power supply to ground, but an appropriate resistor prevents this from happening too quickly.



Typically, the power supplies for our circuits will be +5V (5 volts).

If we place a wire that forms a path around our resistor, we have a problem: we make a toaster. (recall your physics: $V = IR$)

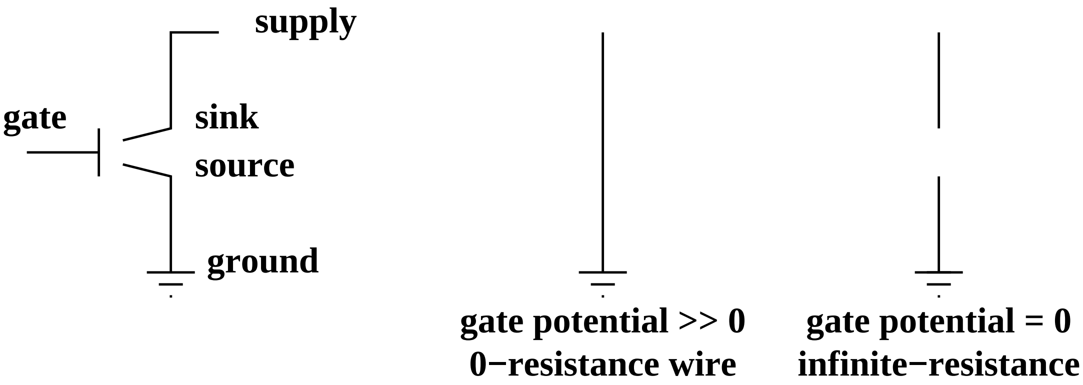


We want to avoid conducting all of our electricity like that, so be careful.

For this class, we'll want to make sure we have a path from supply to ground, but always with resistance along the way. We won't worry much about it beyond that.

Transistors

The *transistor*, invented in 1948, is the key device that allows us to build the kinds of logic circuits that we will study.



This is a *field-effect transistor (FET)*. For physicists, this is a *continuous* device – a variable resistor, maybe an amplifier.

For our Computer Science purposes, we only care about this device's behavior at +5V or 0V (i.e., 1 or 0, true or false).

Realistically, 0-1V is a reasonable 0, 2-5V is a reasonable 1, and 1-2V is illegal as it may cause unpredictable behavior.

The behavior of the transistor follows these rules:

- If a potential (+5V, or 1, or true) is placed on the `gate`, the transistor acts like a wire.

- If no potential is placed on the `gate` the transistor acts like a broken wire – one with infinite resistance.

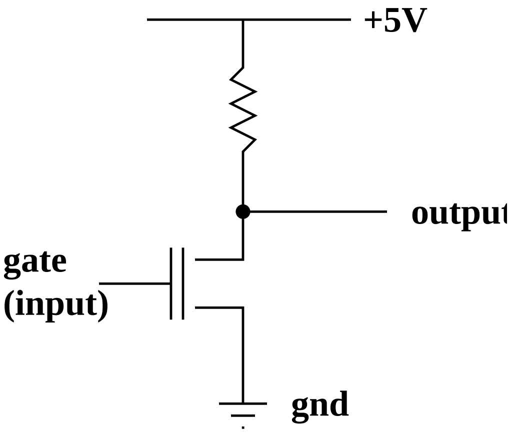
The transistor has some semiconducting material at the gate and that causes a delay in the electron flow. When we switch the gate from true to false or false to true, it takes some time to change its behavior.

This *gate delay* is small, but keeps us from building faster computers. We have to wait for the electrons.

Modern computers will have 50-100 billion of these switches, which can be just a few atoms across, allowing about 1 trillion "gate operations" per second.

The Inverter

Consider this circuit:

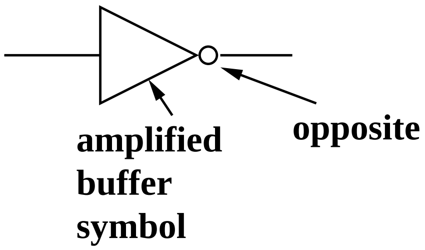


The transistor part becomes a 0Ω or $\infty\Omega$ resistor, depending on the gate `input`, so the potential at `output` will be either 0V or +5V, depending on the value of the `input` at the gate.

`input`=0 produces `output`=1 (+5V) since the `output` has no connection to ground
`input`=1 produces `output`=0 (0V) since the `output` is connected (or "pulled") to ground

This is an *inverter*!

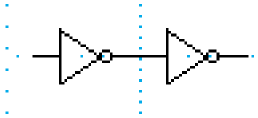
in	out
0	1
1	0



In the symbol, the triangle is for an *amplified buffer* and the circle on the tip means “opposite” or “invert”.

The “buffer” just slows the signal down.

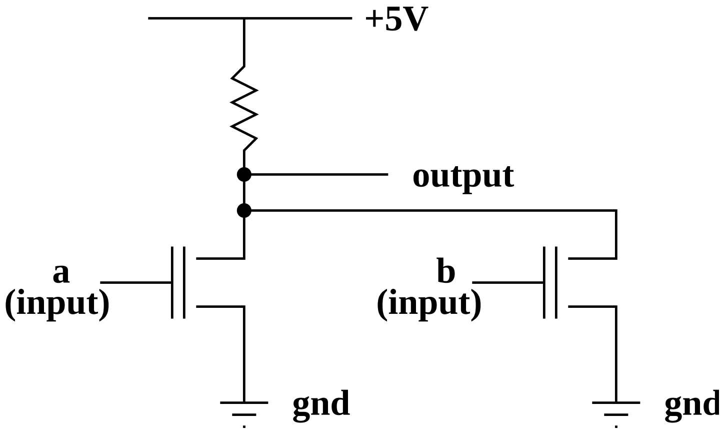
Since !!b=b, putting two inverters in series produces an output equal to the input, but such a circuit will build the strength of a signal and slow it down. Both can be useful, as we will see.



in	out
0	0
1	1

Constructing NOR and NAND

Now consider this circuit:



What does this circuit do? We have two transistors.

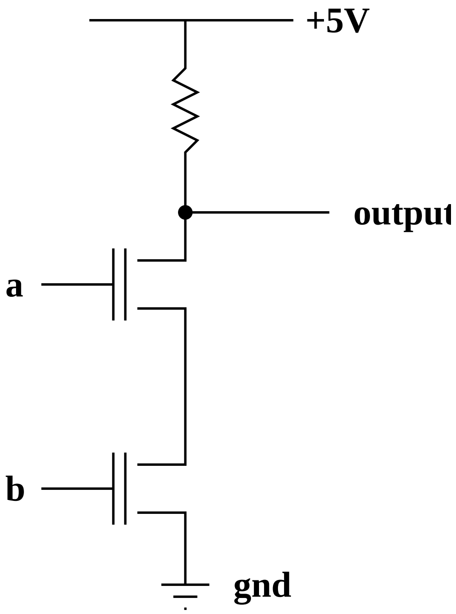
Which wires are connected or broken when we present values of `a` and `b` on the input? What happens to the potential?

When either `a` is 1 or `b` is 1, the `output` will be pulled to ground. If both are 0, there is no path from `output` to ground.

a/b	0	1
0	1	0
1	0	0

This is an inverted OR – the NOR: ¬∨ (not OR)

What if we put our two transistors in series?



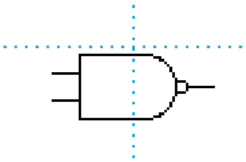
Now our output looks like this:

a/b	0	1
0	1	1
1	1	0

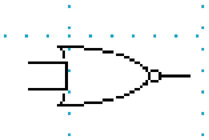
This is an inverted AND – a NAND: ¬∧ (not AND)

Abstractions of Physical Gates

Our lowest level of abstraction is to take our transistor-based circuits and abstract to these physical digital logic gates:



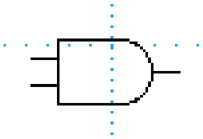
NAND gate



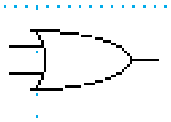
NOR gate

We know how to build them, but no longer need to think about how they work (except maybe on a homework problem or an exam question).

We assume the existence of inverters, NAND, NOR gates and we use these to build other (more complex) gates:



AND gate



OR gate

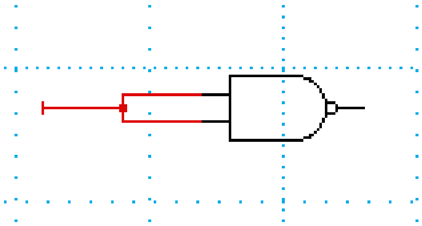
Universality of Certain Gates

We can use these five gates to construct a variety of circuits.

Two of these gates are *universal*: NAND and NOR.

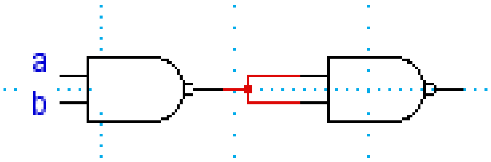
Any circuit that we can build with all 5 available can be built with only NAND or only NOR gates.

For example, if we wire the same signal to both inputs of a NAND:



This is an inverter!

If you have only NAND gates, you can build an AND gate:



We can do similar things to build NOR, OR from NAND.

We can also construct all other gates out of only NORs. Left as an exercise. (hint: DeMorgan's Laws)

Representing a Mathematical Function

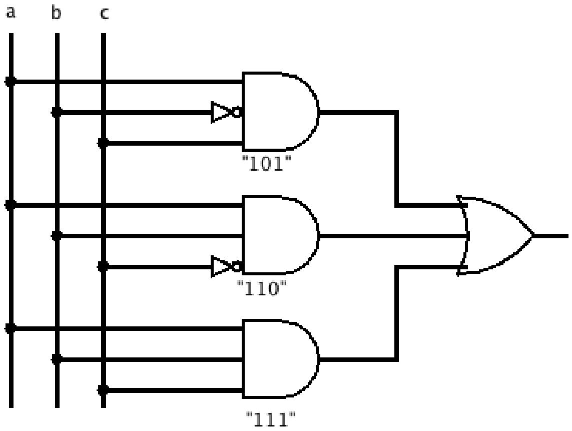
We now wish to build a circuit to compute a given function:

input	f(input)
000	0
001	0
010	0
011	0
100	0
101	1
110	1
111	1

To construct a circuit for this, take a set of AND gates, one for each "1" in the function, stick inverters (represented just by little circles on the inputs) on the inputs that are 0.

Then hook up all the AND gate outputs to an OR gate, the output is the function.

For the above function:



Logisim Circuit:
`/home/jteresco/shared/cs324/examples/logisim/firstexample.circ`

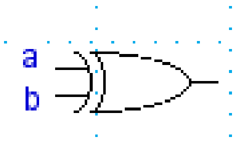
We can do this for any binary function!

For a function of an n-bit value as an m-bit value, we can construct m of these, and compute any function we want (sqrt, sin, whatever).

We can almost always construct a simpler but equivalent circuit with fewer gates than needed by this approach.

Circuit simplification, in general, is a very hard problem.

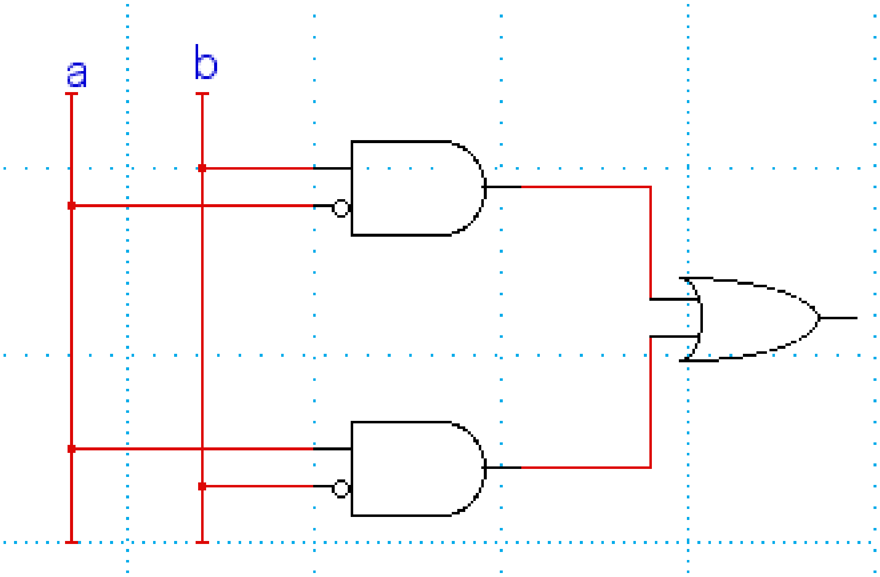
How about a circuit to compute exclusive OR from the other 5 gates?



Moreover, what is the fewest number of gates needed?

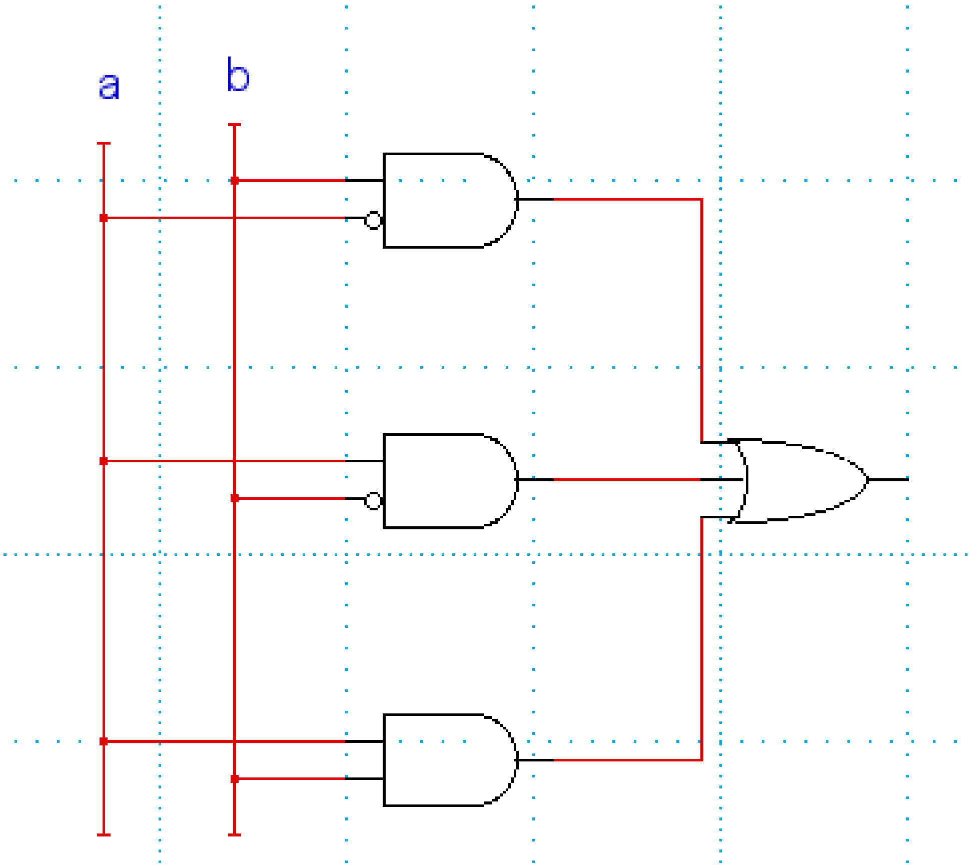
We can do this with the technique we used previously, make a truth table (note Gray code ordering):

a	b	out
0	0	0
0	1	1
1	1	0
1	0	1

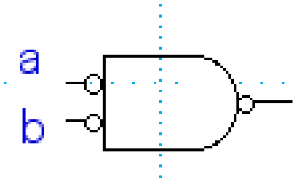


How about OR (kind of silly, yes, but we can do it):

a	b	out
0	0	0
0	1	1
1	1	1
1	0	1



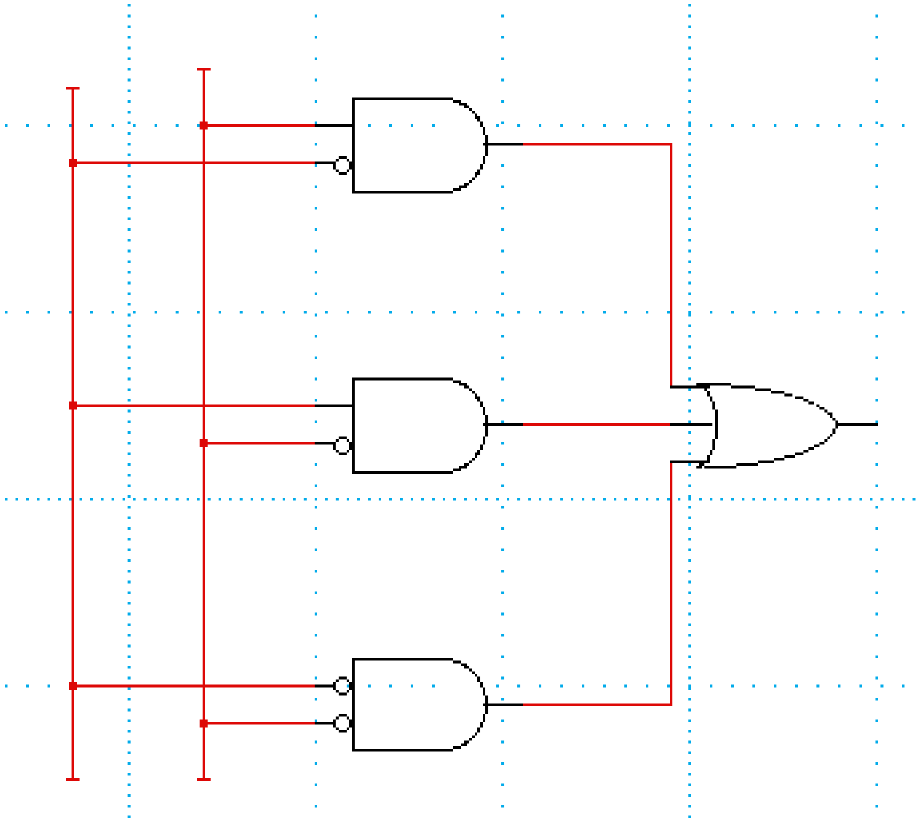
Of course, this seems pretty wasteful. Even if we didn't just want to use an OR gate, we could compute the opposite function and invert:



How about implementing NAND?

a	b	out
0	0	1
0	1	1
1	1	0
1	0	1

Which goes directly to:



We can save some inverters by having a and $\overline{a}$, b and $\overline{b}$ then only regular AND gates.

By doing this, we save two inverters. That's good.

Of course, if we wanted to simplify a circuit for NAND in real life, we'd probably just use NAND...

The point: there are many cases where we will generate a circuit algorithmically and it won't generate the simplest circuit.

Simplification of Circuits

We looked at how we could use AND, OR, and NOT gates to compute any function of n inputs.

We already saw one trick to simplify. If we use the inverse of an input more than once, we can invert the signal once and connect the inverted signal to all of the places that want that input.

We can also notice quite easily that if our truth table for the function being computed has more 1's than 0's, we might want to compute the inverse of the function and invert the output at the end.

But there's certainly more we can do.

Let's consider this function:

a	b	c	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

We could draw a circuit to do this, but it's going to be fairly complex. 5 3-way AND gates feeding into a 5-way OR gate, and 3 inverters.

To consider how we can simplify this, let's write this in a *sum of products* form:

$$f = \overline{a}b\overline{c} + \overline{a}bc + a\overline{bc} + a\overline{b}c + abc$$

where "multiplication" represents an AND operation, and "addition" represents an OR operation.

But we can notice some things about this expression that will allow us to simplify it. Note that between the terms $\overline{a}b\overline{c}$ and $\overline{a}bc$ that if $a = 0$ and $b = 1$, it doesn't matter what c is, the result is always 1. So we can effectively cancel out those c's:

$$f = \overline{a}b + a\overline{bc} + a\overline{b}c + abc$$

Same thing when $a = 1$ and $b = 0$. c doesn't matter. So we can simplify further:

$$f = \overline{a}b + a\overline{b} + abc$$

This leads to a simpler circuit.

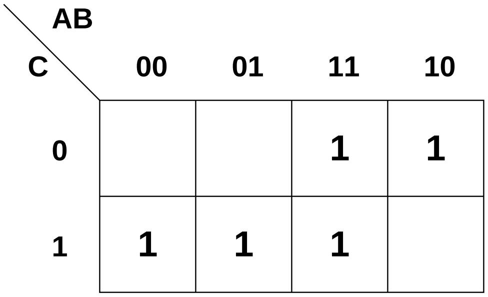
But we can do even better with the subtle observation that we can combine the same term more than once. Also note in our original expression that when $a = 1$ and $c = 1$, b doesn't matter. So we can leave out b from our last term and reduce the size of one of our AND gates in the corresponding circuit:

$$f = \overline{a}b + a\overline{b} + ac$$

Karnaugh Maps

A mechanism to perform these simplifications was proposed in 1953 by Karnaugh.

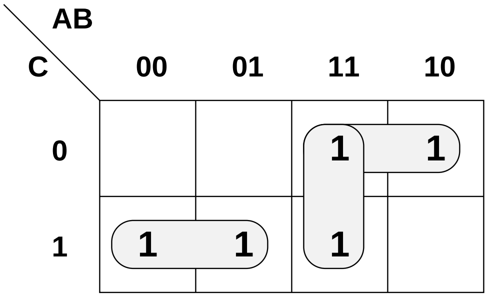
We draw our truth table in an odd format:



AB C	00	01	11	10
0			1	1
1	1	1	1	

Note the odd ordering of the patterns for AB – gray code. These differ in only one bit.

Next, we look for pairs (or quads) of adjacent 1's in the map, and circle them.



AB C	00	01	11	10
0			1	1
1	1	1	1	

Each circle encompasses two (or 4 or more) outputs that can be combined, since they differ only in one bit.

We can then choose a subset of these circles that "cover" all of our 1's with circles (even if they're just the one square), and we have a simplified sum-of-products expression that will lead to a simpler circuit.

We can cover a 1 with more than one circle, but there's no need to cover multiple times.

So in this case, we have several options. The simplest options:

$$\overline{a}c + ab + a\overline{c}$$

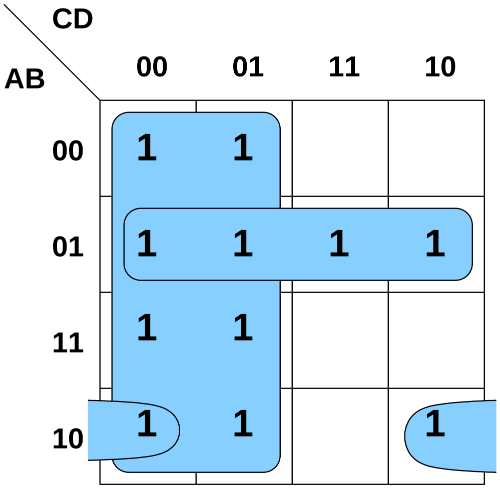
or

$$\overline{a}c + bc + a\overline{c}$$

just as we figured out before.

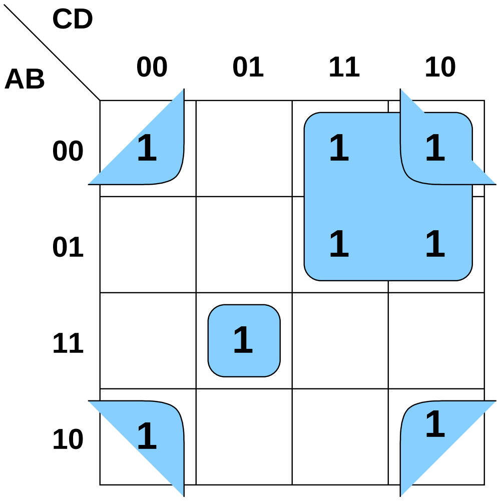
We can consider larger examples: a 4-input function that leads to a 4x4 K-map.

Note that we can circle groups of 4, 8.



This one corresponds to

$$f = \overline{c} + \overline{a}b + a\overline{bd}$$



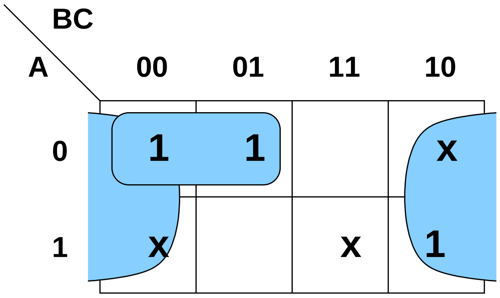
This one corresponds to

$$f = \overline{bd} + \overline{a}c + ab\overline{c}d$$

In some cases, we don't care about certain combinations of input. For example:

a	b	c	f
0	0	0	1
0	0	1	1
0	1	0	x
0	1	1	0
1	0	0	x
1	0	1	0
1	1	0	1
1	1	1	x
c			

The *x* entries indicate those input value that we don't care about. They can be 0 or 1: whatever makes our circuit simpler.



We can choose to cover (or not), the don't care entries in our K-map.

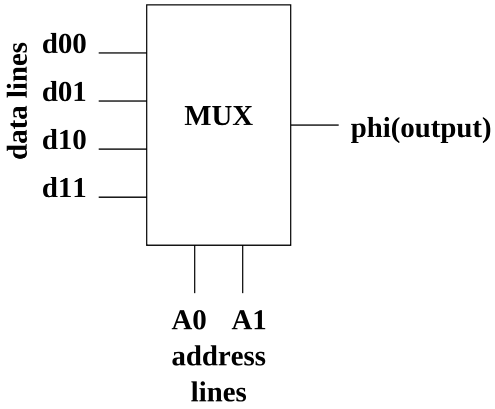
The circling above corresponds to

$$f = \overline{a}\overline{b} + \overline{c}$$

Multiplexers and Demultiplexers

Suppose we have a shared telephone line – we want any one of a number of incoming lines to be connected to an output line.

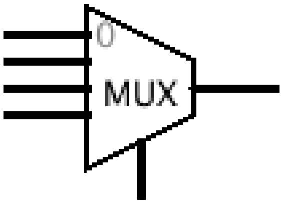
We want this device:



A *multiplexer* – picks which of several inputs gets passed to a single output line.

If A=00, we want d_{00} connected to ϕ, others disconnected, etc.

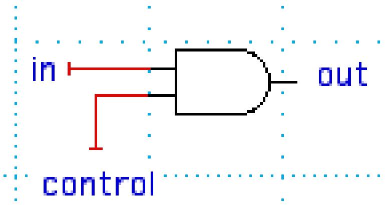
In Logisim, the symbol looks like this:



Logisim Circuit: `/home/jteresco/shared/cs324/examples/logisim/mux.circ`

How can we implement this with the tools we have so far?

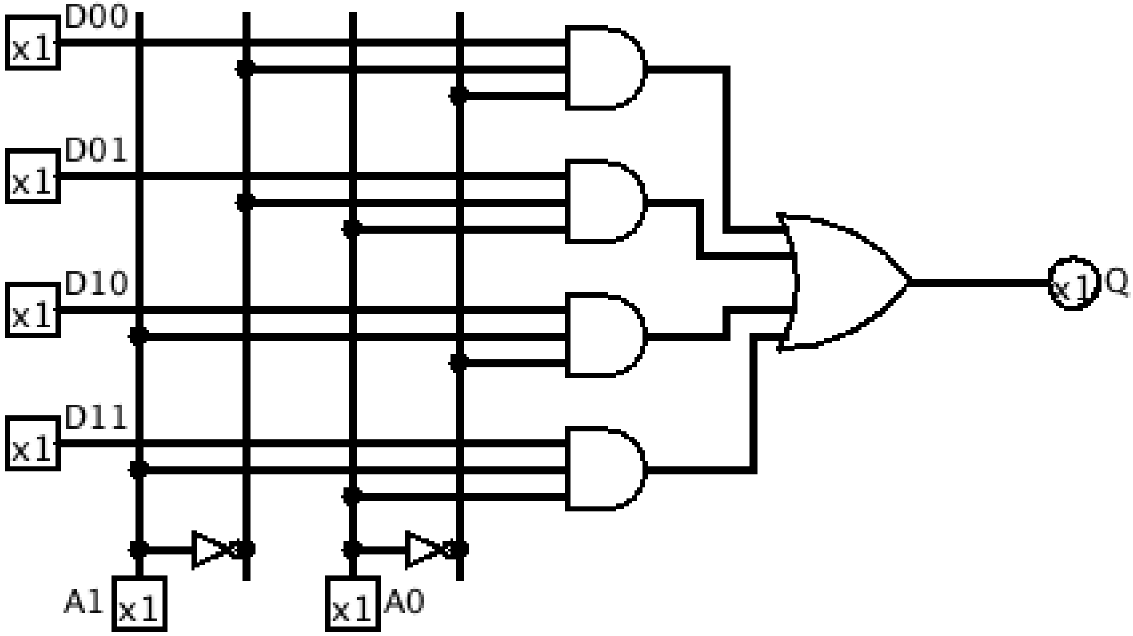
Let's first think about how an AND gate can be used as a control device:



If the control is high (1), the input is passed on to the output.

If the control is low (0), the input is irrelevant and a 0 is always placed on the output.

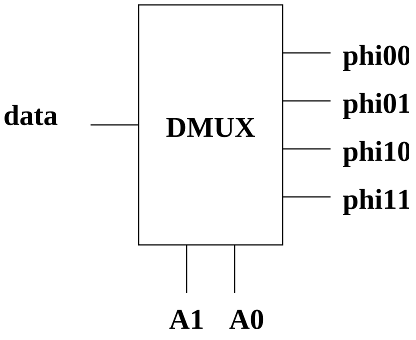
With this in mind, we can build a circuit for the multiplexer:



Logisim Circuit:
`/home/jteresco/shared/cs324/examples/logisim/4to1mux.circ`

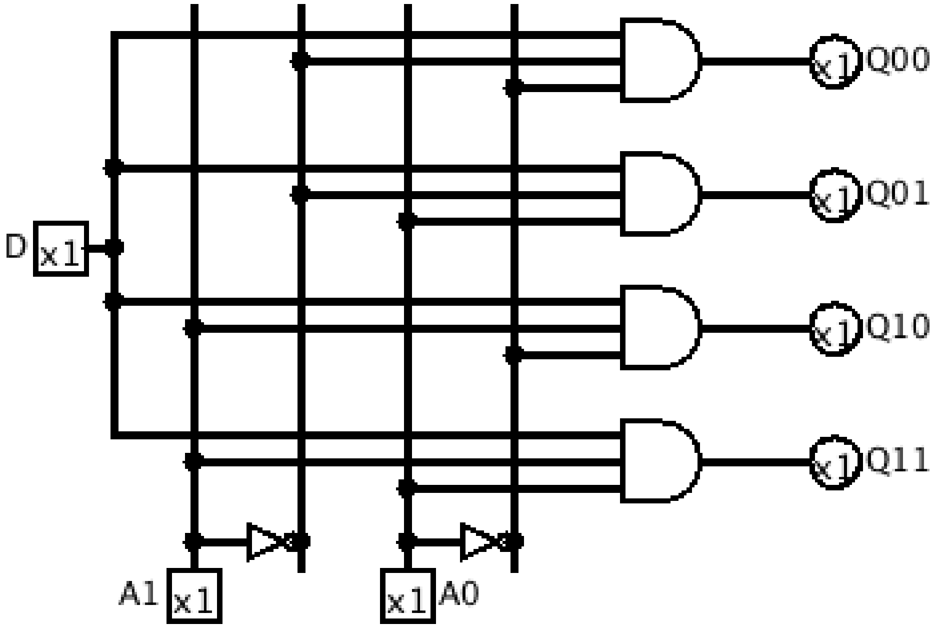
Three of the four AND gates are guaranteed to produce 0 (that is, to mask out their d input). One will pass through its d input to the output.

The opposite of this is the *demultiplexer*



Here, the address lines select which of several outputs get the value from the input, while other outputs get 0's.

And we can do it as such:



Logisim Circuit:
`/home/jteresco/shared/cs324/examples/logisim/1to4dmux.circ`

Encoders and Decoders

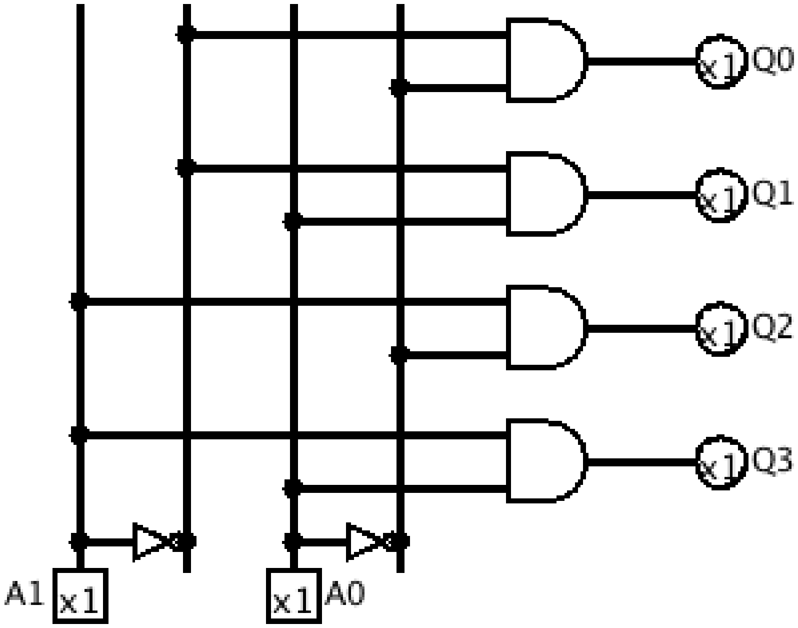
A *decoder* selects one of several output lines based on a coded input signal.

Typically, we have n input and 2^n output lines.

A 2-to-4 decoder:

a	b	0	1	2	3
0	0	1	0	0	0
0	1	0	1	0	0
1	1	0	0	1	0
1	0	0	0	0	1

A circuit to do it:



Logisim Circuit:
`/home/jteresco/shared/cs324/examples/logisim/2to4decoder.circ`

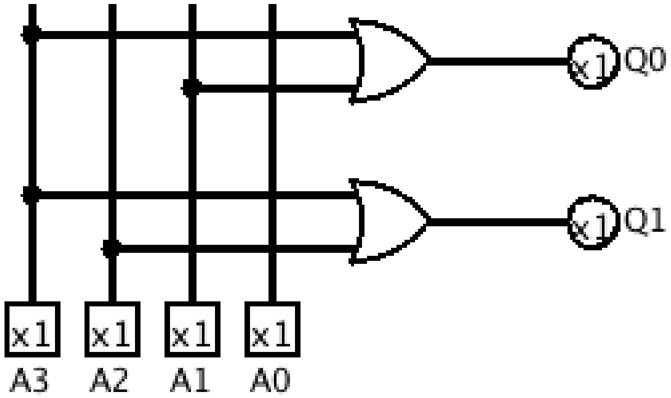
The opposite of this is the *encoder*, where one of several input lines is high, and the output is a code.

Typically, an encoder has 2^n input lines and n output lines.

A 4-to-2 encoder (assuming only legal inputs – where exactly one input line is high):

a_3	a_2	a_1	a_0	ϕ_1	ϕ_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

This is a weird situation, as there are really 16 input combinations, but only 4 are considered "legal". Assuming no illegal inputs, we can construct a circuit:



Logisim Circuit:
`/home/jteresco/shared/cs324/examples/logisim/4to2encoder.circ`

This is not especially satisfying. Our outputs don't even depend on a_0!

But these are still potentially useful. Consider a very hypothetical situation where we have 16 buttons, exactly one of which must be pressed at any given time (perhaps this is a voting machine in an election with 16 candidates). Each of these buttons, when on, will turn on a light corresponding to the button in another location.

One simple way to make this happen is to connect the 16 buttons with 16 wires to 16 lights. But with encoders and decoders, we can use fewer "long" wires. Connect the output of the 16 buttons to a 16-to-4 encoder. We essentially "encode" a value 0-15 as a 4-bit number and send that 4-bit number over 4 wires. On the other end, we use a 4-to-16 decoder to decode the 4-bit value back to 0-15, exactly one of which will be high and light up one of our 16 lights.

Priority Encoders

More likely, we would want what is called a *priority encoder*, where there is a priority of inputs, making all combinations legal. We could give priority to either low-numbered or high-numbered inputs.

For low input priority, we'd have this truth table:

a_0	a_1	a_2	a_3	ϕ_0	ϕ_1
1	1	1	1	0	0
1	1	1	0	0	0
1	1	0	1	0	0
1	1	0	0	0	0
1	0	1	1	0	0
1	0	1	0	0	0
1	0	0	1	0	0
1	0	0	0	0	0
0	1	1	1	0	1
0	1	1	0	0	1
0	1	0	1	0	1
0	1	0	0	0	1
0	0	1	1	1	0
0	0	1	0	1	0
0	0	0	1	1	1

Multi-input Gates, Fan Out

As our circuits have become more complex, we have seen multi-input gates. For AND, we can draw any number of inputs symbolically, put a slash through the inputs with a wire to specify large numbers.

We draw these, but we might actually buy chips that provide only 2-input gates.

We can construct a 3-way AND from 2 2-way ANDS.

We can construct a 4-input and from 3 2-way ANDs:

Is this bad? Well, it's not natural, looks kind of like a loop, as our information cascades through the gates.

A "tree-like" structure is better. At the least, there are only 2 gate delays before we have the answer.

Even so, we are stuck with $n - 1$ 2-way gates to implement an n-way gate.

In the 4-input case, the difference in gate delay isn't such a big deal, but think about the 64-input AND. The linear approach leads to a circuit with 63 gate delays, while the tree approach has only 6.

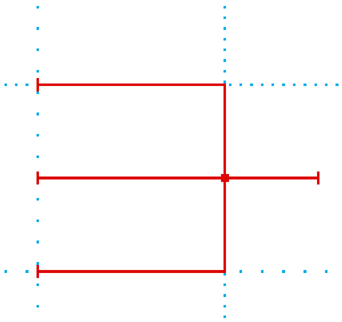
So when we're constructing an n-input gate, we will have

- $n - 1$ gate equivalents charged to
    1. transistor budget
    2. real estate on the chip/board
- and $O(\log n)$ gate delay

Fan In and Fan Out

A wire in our circuit must be driven by +5V input, GND, or the output of some gate.

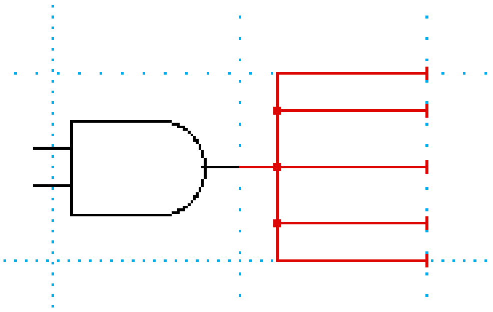
This *fan in* is not a good idea.



This could be a short circuit – avoid it. It would be bad for your grade on a circuit design and bad for the gates if you were to wire it up, as the gates might get a signal coming in the wrong way.

It is called a *wired or* since if it does what we intend, it would behave like an OR gate.

Fan out is allowed but is limited by the gate power.



The practical limit is 4 or 5 other gates powered by the output of a gate.

High *gate load* will mean weak signals.

A solution: bigger, stronger gates...but bigger, stronger gates are slower (more gate delay).

Another solution: boost your signals as needed with a buffer, but this also introduces gate delay.

Adders

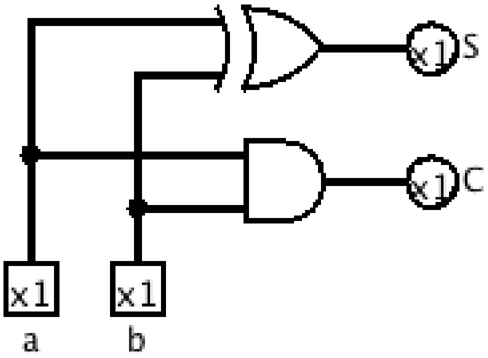
Our next goal is to develop circuits to do addition. Ultimately, we would like to be able to add 8- or 16- or 32-bit 2’s complement numbers together, but to start, we’ll try adding two bits together.

Half Adders

Recall this table from our discussion of binary arithmetic:

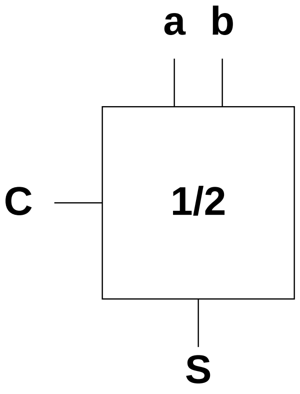
+	0	1
0	00	01
1	01	10

So if I have two one-bit values, a and b, I can get their sum and the carry out with this circuit:



Logisim Circuit:
`/home/jteresco/shared/cs324/examples/logisim/halfadder.circ`

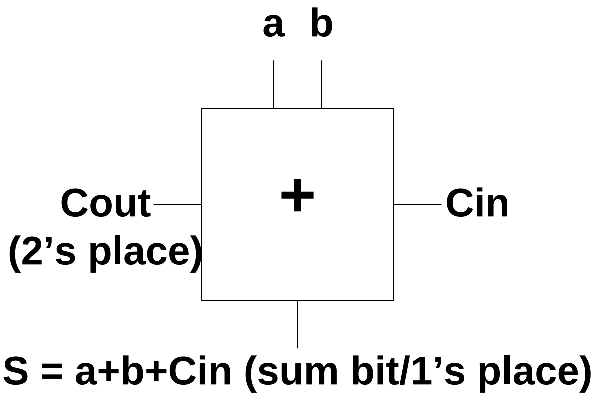
This is called a *half adder*. We represent it with this symbol:



This in itself isn't especially useful, but we'll use this as a building block for what we really want...

Full Adders

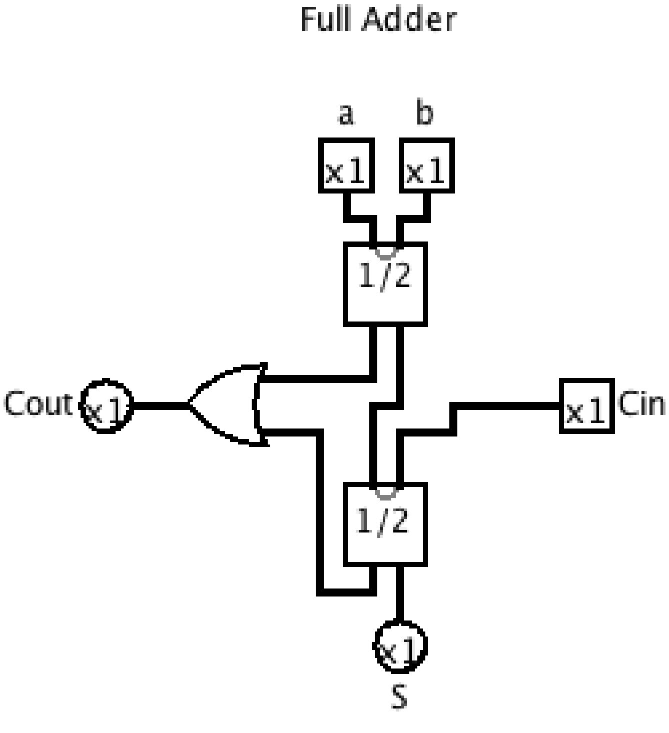
The *full adder*:



This adds a and b, two one-bit values, plus a carry in, to produce a sum bit S and a carry out bit C_{out}.

2 bits is enough to hold the sum, since the range of results is 0-3.

We can construct this from two half adders:

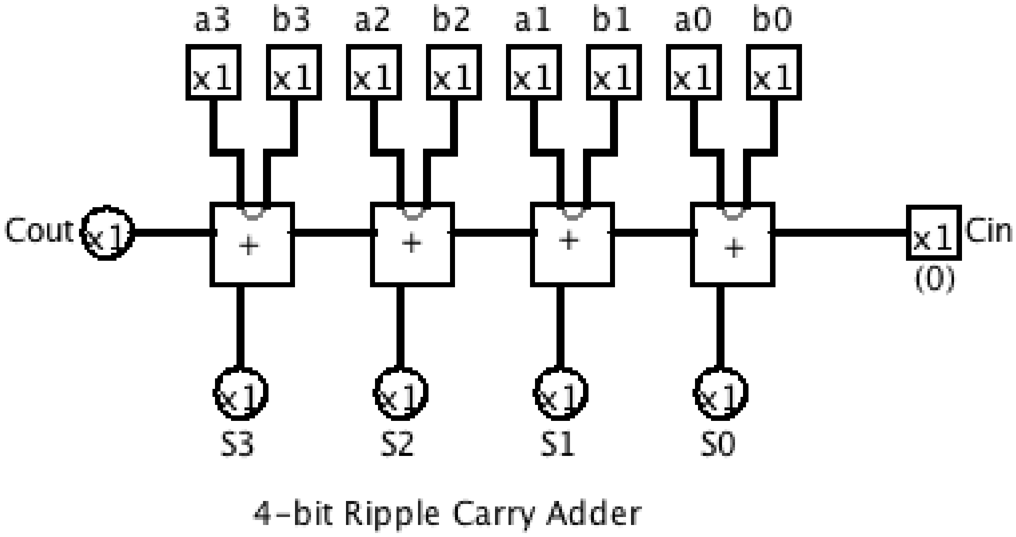


Logisim Circuit: Circuit `fulladder` in
`/home/jteresco/shared/cs324/examples/logisim/adders.circ`

This in itself is still not especially useful, but these can be used to build a multi-bit adder.

Ripple Carry Adder

For example, 4 of these can be chained together to construct a 4-bit adder.



Logisim Circuit: Circuit `4bitrcadder` in
`/home/jteresco/shared/cs324/examples/logisim/adders.circ`

This is called a *ripple carry adder*, since the carry bits ripple along through the circuit.

Think about how the carry is propagated sequentially and has to travel down the chain. This is slow!

For an n-bit ripple carry adder, we have $O(n)$ gates and this requires $O(n)$ gate delays to get the right answer (for sure).

Think about how this works. It works for 2's complement!

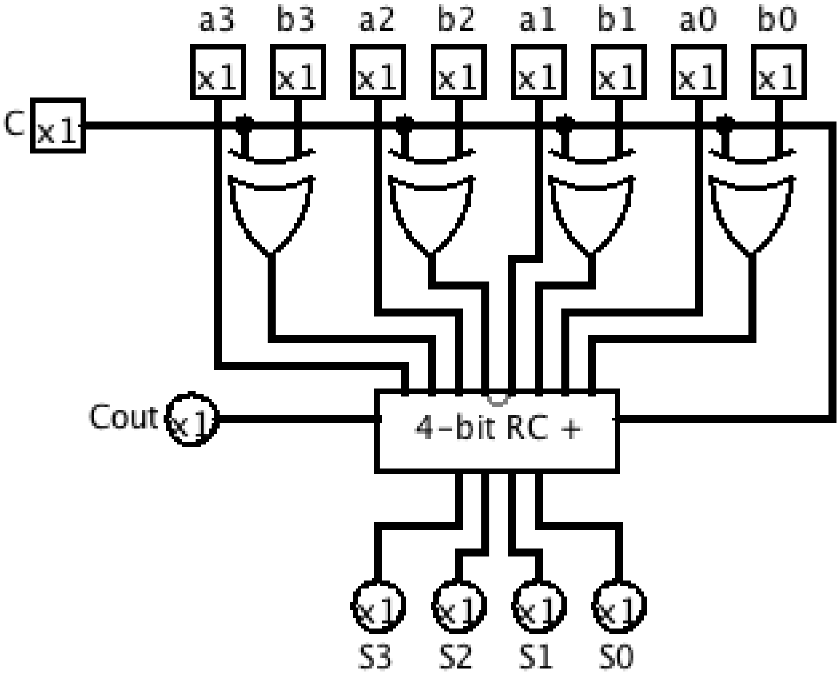
This has relatively poor performance because of the ripple aspect of it, but it is actually used. We just need to make sure we wait long enough before trusting the answers.

We can extend this to any number of bits, but note that it is expensive in both the number of gates and in gate delay.

Subtractors

We could consider building a circuit to do subtraction, but we have these adders that can deal with 2's complement numbers. We can use this to build a subtractor.

In fact, we can do this just by augmenting our 4-bit adder with one extra input.



Logisim Circuit: Circuit `4bitaddsub` in `/home/jteresco/shared/cs324/examples/logisim/adders.circ`

The control line C is called the subtract/$\overline{\text{add}}$ line.

When C is 1, this computes $a - b$, when it's 0, it computes $a + b$.

Why does this work?

Recall that for 2's complement, we get -x from x by inverting the bits and adding 1.

$$a - b \equiv a + (-b) \equiv a + (\bar{b} + 1) \equiv (a + \bar{b}) + 1$$

If C is high, all b bits will be inverted by the XOR gates and the entire 4-bit adder's carry-in line will be 1 (taking care of the second part).

Aside: note how XOR is a "not equals gate" and the control line makes them function as inverters when it (C) is high.

We have built a general-purpose adder.

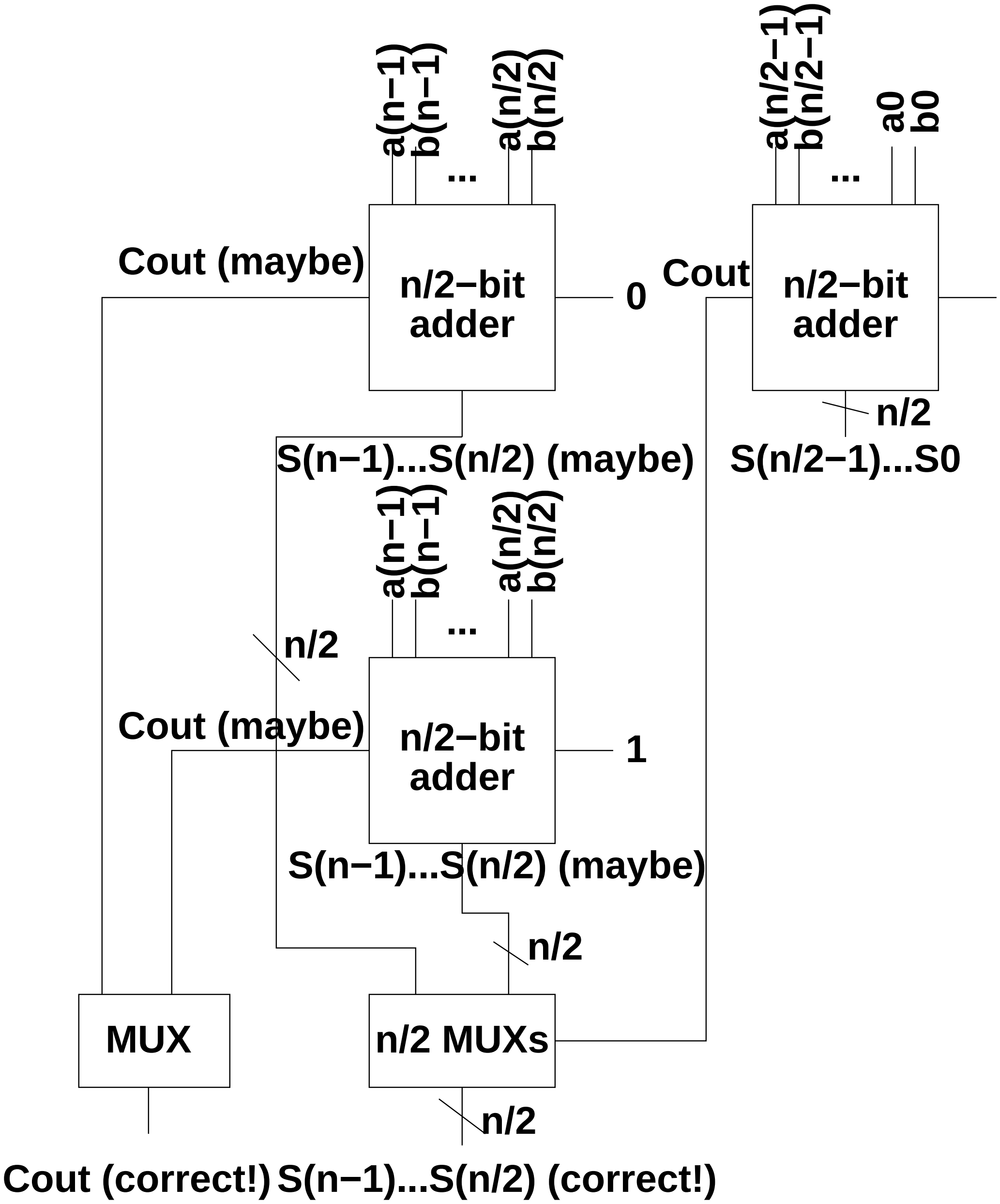
Speeding Up an Adder

Let's see what happens if we break our n-bit adder in half.

We can add 2 $\frac{n}{2}$-bit numbers (in parallel) and combine them into our answer.

We just have to think about what happens when the bottom half results in a carry out.

Consider this:



We compute the bottom $\frac{n}{2}$ bits for sure and easily.

We compute both possibilities for the top $\frac{n}{2}$ bits, one with carry in, one without.

Then, when the carry in arrives from the bottom half, we use a set of $\frac{n}{2} + 1$ MUXs and use the carry out from the bottom half to select which input (the top bits plus carry out) to pass through!

Some notes about this approach:

- We can make the low-order one a few bits smaller, so the carry out is already delivered to the MUXs when the high-order ones finish.
- This costs more space (bigger circuit) but saves time.

- We can do this recursively! But we don’t need to create the whole tree to do it. We only need twice the space to do this.
- Difficulties: it’s hard to lay out on the chip (wires want to cross).
- Realistically, standard ripple-carry addition is used for values of size 16 bits or less. It would likely be broken down recursively for larger operand sizes.