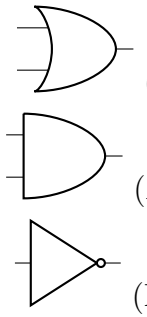
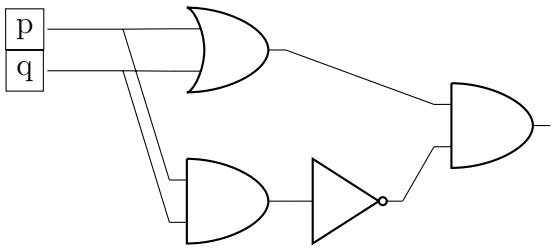
6 March 2015

Today we will be abstracting away the physical representation of logical gates so we can more easily represent more complex circuits. Specifically, we use the shapes themselves to represent what type of gate is being used.



Let's note that we *cannot* run two wires together without a gate. We *can* split a wire to be used as input multiple times. Finally, we will not *for the moment* allow a gate's output to be reused as input to itself.

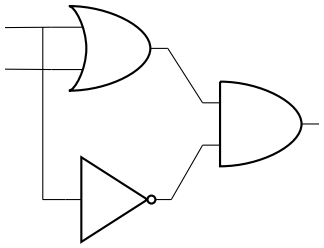
So, for a more complicated example:



p	q	$p \vee q$	$p \wedge q$	$\neg(p \wedge q)$	$(p \vee q) \wedge \neg(p \wedge q)$
T	T	T	T	F	F
T	F	T	F	T	T
F	T	T	F	T	T
F	F	F	F	T	F

Using a truth table, we can examine this circuit to recognize that it's output is the XOR of p and q.

So let's look at another example, except this time we will start from the proposition instead of the circuit: $(x \vee y) \wedge \neg x$.



x	y	$x \vee y$	$\neg x$	$(x \vee y) \wedge \neg x$
1	1		0	0
1	0		0	0
0	1	1	1	1
0	0	0		0

Notice that, in the truth table above, we were able to save some time by only checking one side of the final AND expression; if we get a false, we need not calculate the other value!

Binary Expansion

Binary is a different way of expressing numbers, which works just like base-10:

Decimal	Binary	Binary Expansion
0	0	$0 \cdot 2^0$
1	1	$1 \cdot 2^0$
2	10	$0 \cdot 2^0 + 1 \cdot 2^1$
3	11	$1 \cdot 2^0 + 1 \cdot 2^1$
4	100	$0 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2$
5	101	$1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2$
6	110	$0 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2$
7	111	$1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2$

Proposition 1. $\forall n \in \mathbb{Z}_{\geq 0}, \exists$ an expression of the form

$$n = a_k 2^k + a_{k-1} 2^{k-1} + \cdots + a_1 2^1 + a_0 2^0$$

such that $a_i \in \{0, 1\} \forall i$. A binary representation of an integer is simply a 0/1 string which lists the coefficients a_i in order.

Proof. The proof is by induction, and left as an exercise. □

Long Addition

Notice that we need to 'carry' extra digits whenever we add numbers (just as we did in decimal addition):

$$\begin{array}{cccc} & 1 & 0 & 0 \\ & 1 & 0 & 1 \\ \hline 1 & 0 & 0 & 1 \end{array}$$

So, as we build a circuit to add binary numbers, we will first start with a *half-adder*: it takes in two inputs (digits), and returns two outputs (the sum and the 'carry' bit). Our table for a half adder may look something like this:

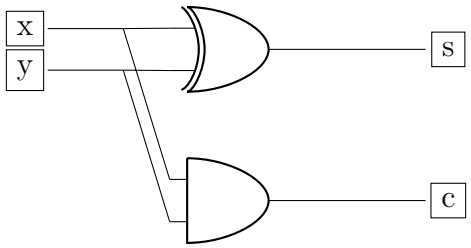
x	y	s	c
1	1	0	1
1	0	1	0
0	1	1	0
0	0	0	0

9 March 2015

We were attempting to put together a combinatorial circuit for basic addition.

Last week, we had just discussed the idea of a *half-adder*, which has two inputs (a bit from each number), and two outputs (one 'sum' bit and one 'carry' bit).

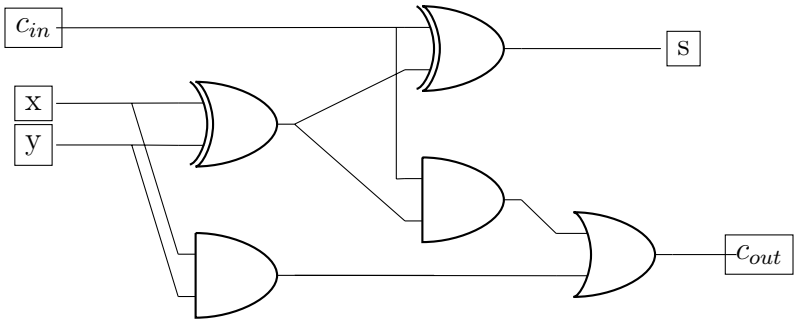
x	y	s	c
1	1	0	1
1	0	1	0
0	1	1	0
0	0	0	0



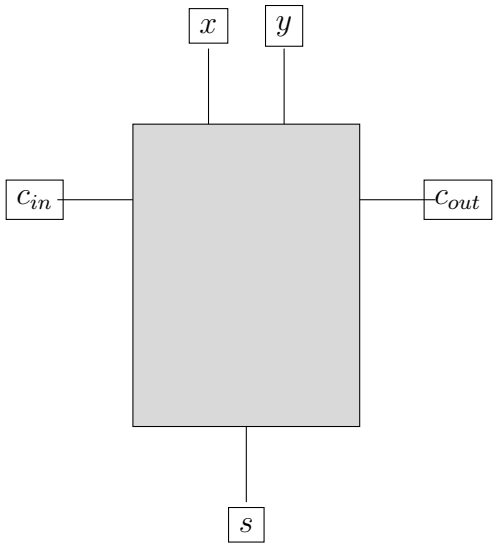
Note: the top gate in the above diagram is a Xor *gate.*

But notice that this does not allow us to consider whether a carry bit *already* existed. So this is what a full adder needs: a third input.

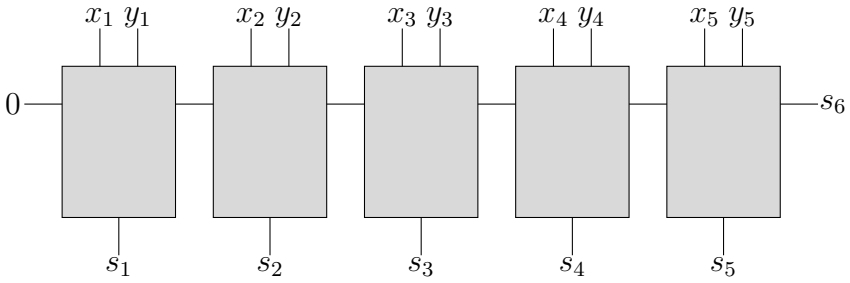
x	y	c_{in}	s	c_{out}
1	1	1	1	1
1	1	0	0	1
1	0	1	0	1
1	0	0	1	0
0	1	1	0	1
0	1	0	1	0
0	0	1	1	0
0	0	0	0	0



If we take these full-adders and 'black box' them into a new, constructed gate, which looks like the following:



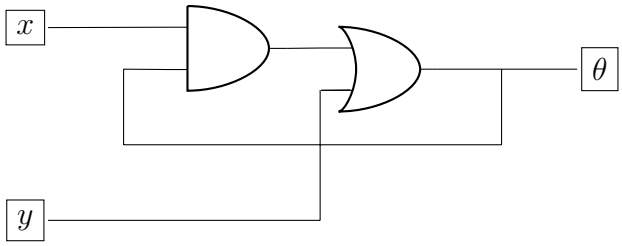
Then we can string them together sequentially to make the 'ripple adder', which adds any two binary numbers. For example, with 5 digits:



Because the carry bit 'ripples' across all the 1-bit full adders, this is known as a ripple-adder sometimes. And, because we need to *wait* for it to ripple across, using this actually slows down addition slightly. By looking for places where both x and y have a 0 bit, we can actually start a *new* ripple adder at the following point, since we don't need to wait for the carry bit to know it will be a 0.

Feedback

Before, we did not allow feedback, but now we will allow it! This will allow us to have a concept of state.

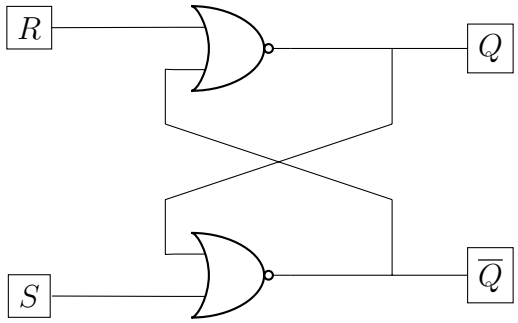


x	y	θ_{old}	θ_{new}	Action
1	0	1	1	Hold
1	0	0	0	Hold
*	1	1	1	Set 1
*	1	0	1	Set 1
0	0	1	1	Set 0
0	0	0	0	Set 0

As you can see, we are able to change and store the value of θ by feeding in certain values to x and y.

11 March 2015

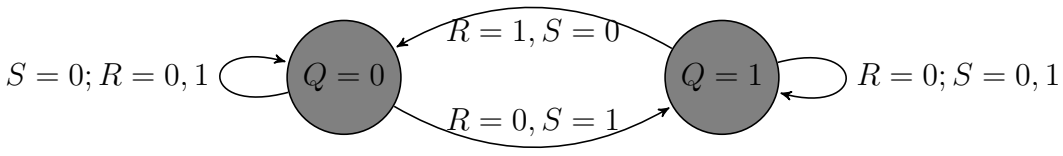
We return to feedback circuits! But we're going to move onto a more complex gate called an 'R-S Latch'.



Q is the stored value of the circuit. We need to consider that Q already has a value before we give the latch inputs.

- Consider $R = S = 0$. If Q started at 0, we get that Q stays at 0. If $Q = 1$, we still get that Q stays at 1, so $R = S = 0$ is a hold state.
- Consider $R = 0, S = 1$. Because of $S = 1$, we know that $\overline{Q}$ will be 0. If we start with $Q = 1$, then nothing changes; but if $\overline{Q}$ was 1 before, we will change it to 0, and change Q to 1. Thus, $R = 0, S = 1$ changes Q to 1.
- This circuit is symmetric, so we know that $R = 1, S = 0$ changes Q to 0.
- If we set both $R = 1$ and $S = 1$, then we get that $Q = \overline{Q} = 0$, which is a contradiction – thus, we never want to set $R = 1$ and $S = 1$ simultaneously.

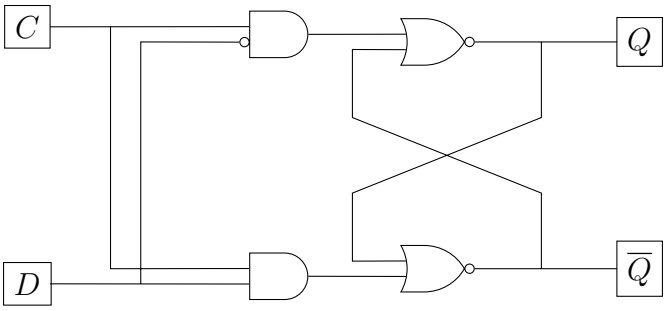
Another way of thinking about states like this is with a state diagram, like the following, where each :



Clock Input

A clock is an periodically alternating signal that switches between 0 and 1 that flips on some consistent, discrete time scale.

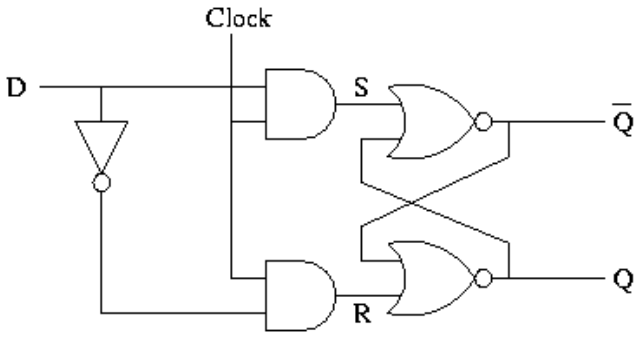
Using a clock input we can put together a 'D-latch', which acts like a safer RS-Latch. A D-latch has two inputs: a clock input C, and an input D.



Notice that, by using both D and $\neg D$ in the two AND gates, we prevent the possibility of both R and S being 1. Instead, whenever the clock is on, D is written to Q. Manually looking at the RS-latch truth table will confirm this.

13 March 2015

Last time, we formed an R/S-latch (2 binary inputs, Q and $\neg Q$ as output) and a D-latch (added clock input).

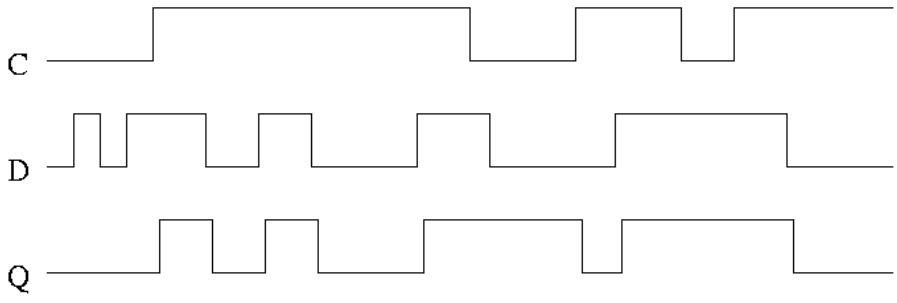


The clock input was an alternating signal that was flipping between 0 and 1 on some time scale. To figure out the output of a D-latch, we have to think about 1) whether the latch is open or closed (ie. the clock is 0 or 1); and 2) what value Q takes on with that input.

How do we think about this? We have a stored-value Q. Take D in and record the value of D to Q. Or the value of D comes in but Q does not change, it holds. Open the latch, write the value of D to Q. Close the latch, hold the value of Q until the next time the latch is opened.

We can think about an R/S-latch process with state diagrams.

We can also think about the subset of a D-latch that comes after the AND gates as an R/S-latch with inputs $C \wedge \neg D$ and $C \wedge D$.
If $C = 0 \implies S = 0, R = 0 \implies Q$ holds.
If $C = 1, D = 1 \implies Q \to 1$. Q gets the value of D.
If $C = 1, D = 0 \implies Q \to 0$. Q gets the value of D.



One limitation of this model is that things can change the entire time that the clock is open. We'd like to restrict/know better when things can change. If you know the instant that things are read/written, that's more powerful than just knowing the timeframe during which the change can happen.

How do we do this? We're going to box off the D-latch, and use two of them to create a Flip-Flop circuit. (Note: we don't use the first $\neg Q$ in the bigger circuit).
Q switches to match D at the moment that C flips from $1 \to 0$ – this is called a "falling edge" circuit.
Q switches to match D at the moment that C flips from $0 \to 1$ – this is called a "rising edge" circuit.

The value of D can be fed in at any point up until Q switches. This gives us efficiency in how we use our circuits.