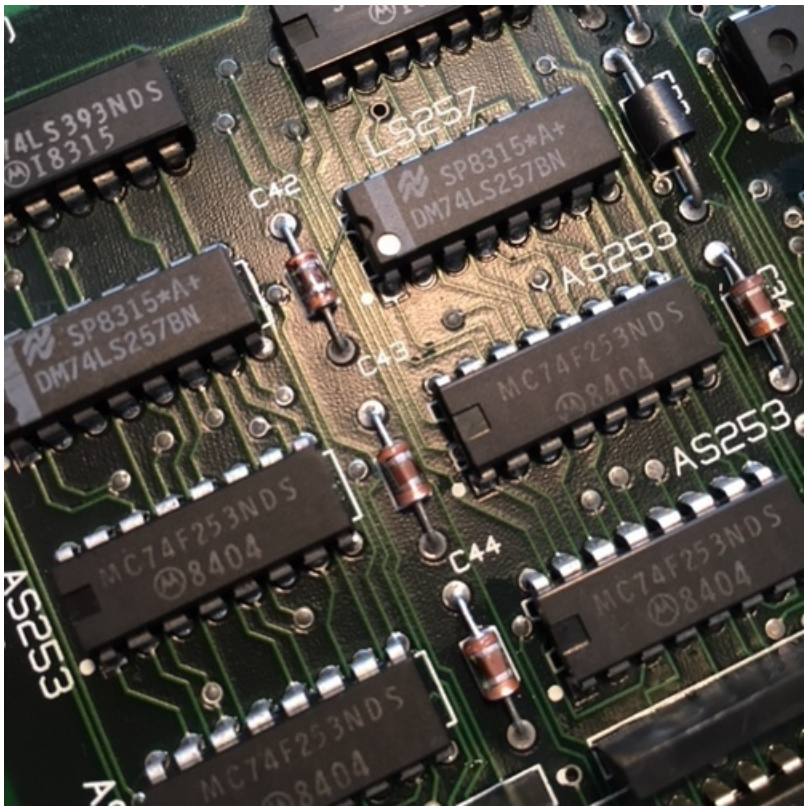
adafruit learning system

Digital Circuits 3: Combinational Circuits

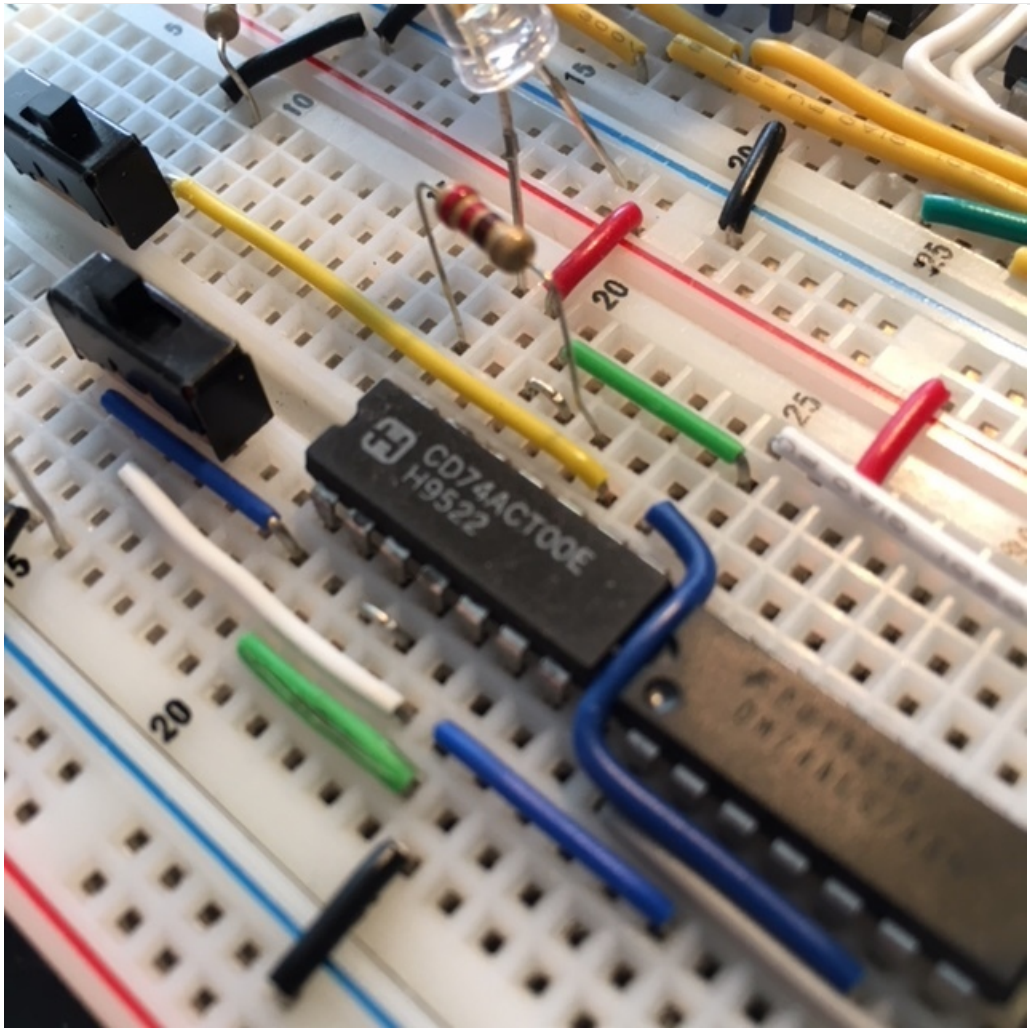
Created by Dave Astels



Last updated on 2018-08-22 04:06:11 PM UTC

Guide Contents

Overview



A combinational logic circuit is a circuit whose outputs only depend on the current state of its inputs. In mathematical terms, the each output is a function of the inputs. These functions can be described using logic expressions, but is most often (at least initially) using truth tables.

Logic gates are the simplest combinational circuits. As we saw in part 1, their output is a very simple function of their inputs describable with a very simple truth table. Naturally, the more inputs there are, the larger the truth table.

This part of the series will have a look at some general classes of combinational circuits and some representative chips from each class.

Decoders

Dealing with multiple signals

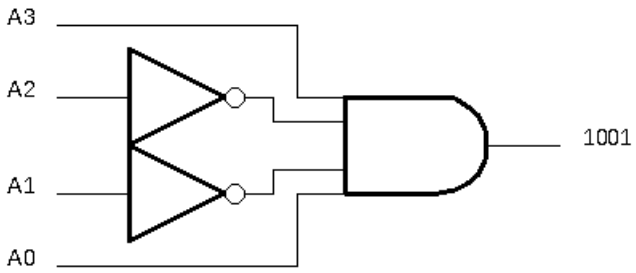
Say we have 4 input signals. How do we talk about them? How do we talk about the values on them?

Typically we use the letters A, B, and so forth to denote groups of related inputs. Each individual input is denoted by subscripting the name: **A**0, **A1**, etc. **A0** is the ones digit, **A1** is the two's, **A3**, is the four's, and so on. If **A0** is high, **A1** is low, **A2** is low, and **A3** is high, the bit pattern of **A** is `1001`, or `0b1001`, i.e. `9`.

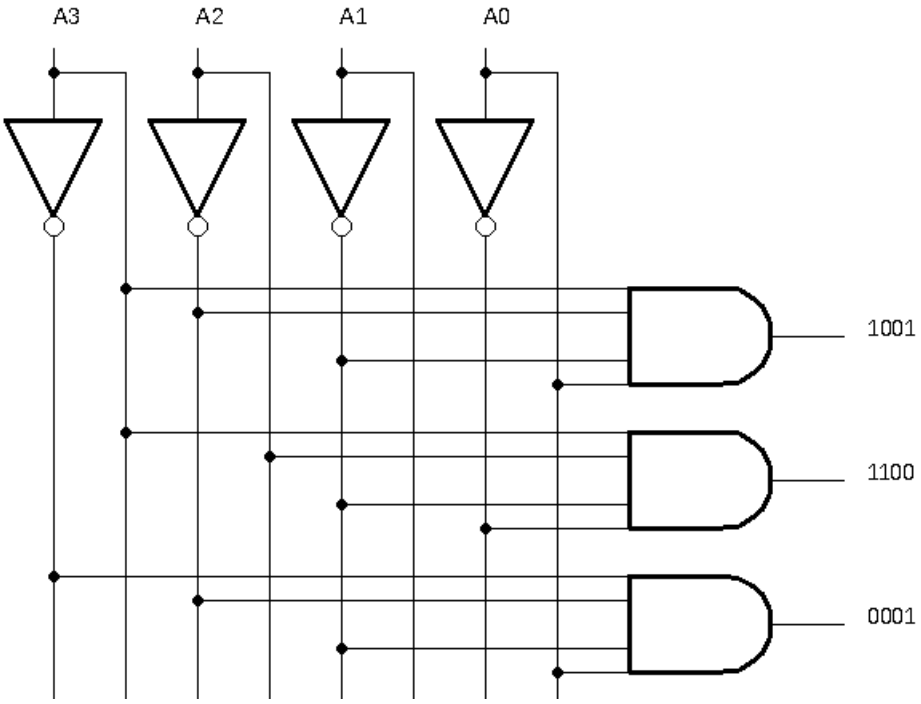
Outputs are usually named O, or Q, or Y. For a group of related outputs we use the same naming as inputs: Q0, Q1, and so on.

Pattern detection

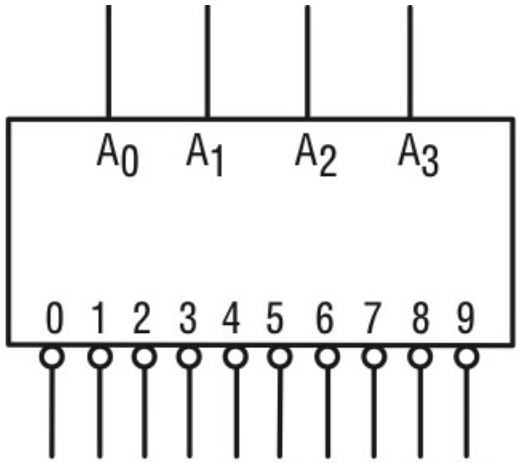
By pattern I mean a specific combination of highs and lows on a set of signals. For example, let's take the case where we have 4-bit input signal **A** as above and we want to detect when it has the value 1001. What exactly is this pattern? Well, **A3** is high AND **A2** is low AND **A1** is low AND **A0** is high. That's four inputs all ANDed together. It just so happens that the 7421 contains two 4-input AND gates. Just what we need. However, some of the inputs are high and some are low. We can use NOT gates (aka *inverters*) to invert the signals that need to be low in the pattern so that they are high going into the AND gate when the pattern is present.



It's very often the case that we will want to decode various patterns present on the inputs. In general we use inverters to generate negated versions of the inputs. Then we can select the versions (negated or not) for each AND gate that matches a pattern to be recognised. For example:

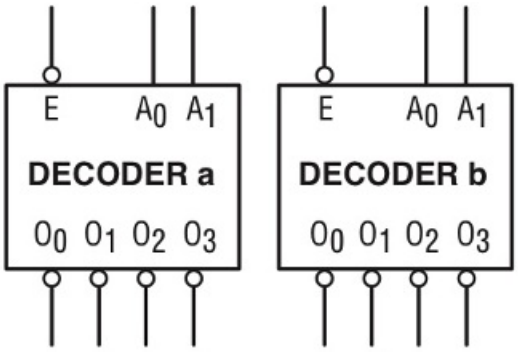


This is so common, in fact, that there are chips specifically for this. They are call *one-of-n decoders*. For example the 7442 is a 1-of-10 decoder. It has four inputs as we described above, and 10 outputs corresponding to input patterns of 0000 - 1001.

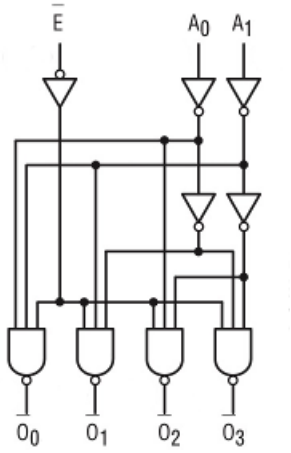


Notice the little circles on each output? This means the output is *active low*. That just means that when the output is *active* it has a logic *low* value, rather than a logic high. So, when the corresponding value is present on the inputs (1001 for example) the corresponding output (9) will be low and the rest will be high.

Another useful decoder is the 74139 dual 1-of-4 decoder. This 16 pin chip contains two 1-of-4 decoders, with a the added feature of an enable input (which is quite common). The decoder works as you would expect with the addition that if the active low enable input is high, all the active low outputs are high regardless of the A inputs. When the enable is low, the decoder operates as usual, setting the corresponding output low.

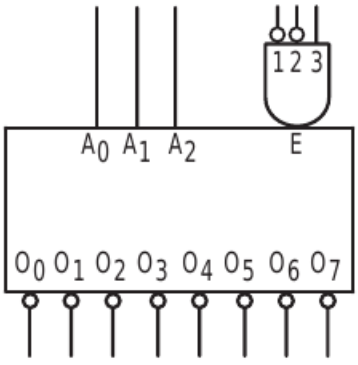


Internally, these decoders look very much like the circuits above. Here's one of the 1-of-4 decoders.

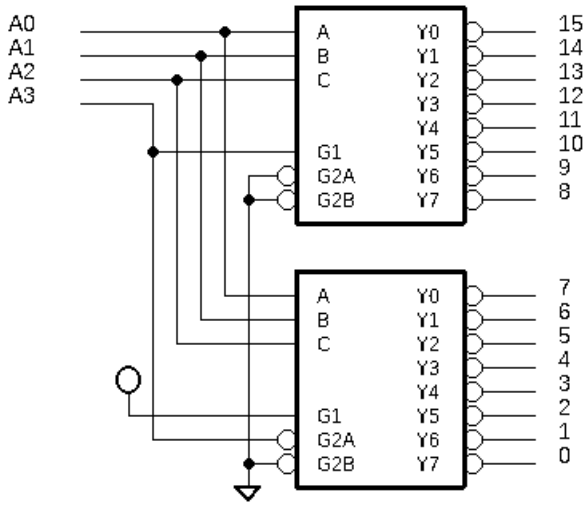


Notice how NAND gates are used instead of ANDs as I've used above. That's why decoder output are typically active low. Recall that NAND gates are the simplest gates to make, requiring fewer transistors and less space.

Another useful decoder is the 74138 1-of-8. This takes 3 input lines and decodes them to 8 active low outputs.



An interesting feature of this chip is its 3 enable inputs: 2 active low and 1 active high. This is very useful when combining them in make a larger (wider) 1-of-n decoders. A 1-of-16, for example.



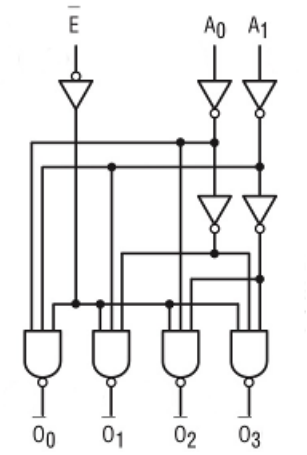
Here, the **A3** input is connected to an active low enable of the lower 1-of-8 decoder, and to the active high enable of the higher one. This way, when **A** is between `0b0000` and `0b0111`, inclusive, (**A3** is low) the lower decoder is enabled. When **A** is `0b1000` to `0b1111`, inclusive, (**A3** is high) the higher decoder is enabled.

Demultiplexers and Multiplexers

Demultiplexer

A demultiplexer is very much like a decoder with an enable. The idea is that you select one of the outputs and route an input signal to it. In fact the 74139 1-of-4 decoder in the previous section is also known as a 1-of-4 demultiplexer. When the E input is high, the selected output is high (ok, they all are but that's not the point). When E is low, the selected output is low. The E input is effectively routed to the selected output. If E varies over time, the selected output will vary in the same way.

Below is the logic diagram of the internals of one of the demultiplexers, as we've seen before. You can see that the enable (we can think of it as the data input for the demultiplexer) feeds into all of the output gates. Each of those NAND gates detect one of the possible patterns on the A inputs, and hence one of the 4 outputs to which the enable/data input is routed.



Fan-In, Fan-Out

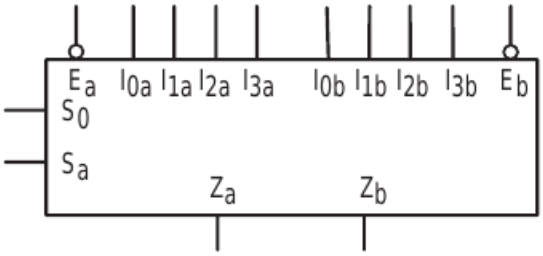
It's interesting to note that the enable/data input is immediately inverted and that **NAND** gates are used to generate the outputs. What's happening is that the logic is done in an active-high way, with the input and outputs inverted to externally be active-low. Even more interesting is that the A inputs are also immediately inverted, then inverted again. Recall the previous diagram where the normal and inverted inputs are used to select the various patterns (since for a given pattern, all inputs to the matching **AND** gate must be high, so the negated form of any that need to be low is used). But why the double negation? That is more of an implementation issue. You know how you'll see that an microcontroller GPIO pin can supply up to some number of milliamps and you can drive an LED directly but you need to do something special to drive a high current load (like a relay or motor)? That's what's going on here. Each input requires a certain amount of current. Chip designers try very had to ensure that each input of a chip presents only a single load to whatever it is connected to. Each output can provide enough current to drive a certain number of input loads, noted as it's fan-out. Negating each input as above ensure that it only presents a single load. Otherwise it would present 3: a NOT gate input and 2 AND gate inputs. When you design logic circuits you need to keep track of how many inputs are connected to each output so as not to exceed each output's fan-out. If you ignore that your circuit could work strangely or not at all.

Multiplexer

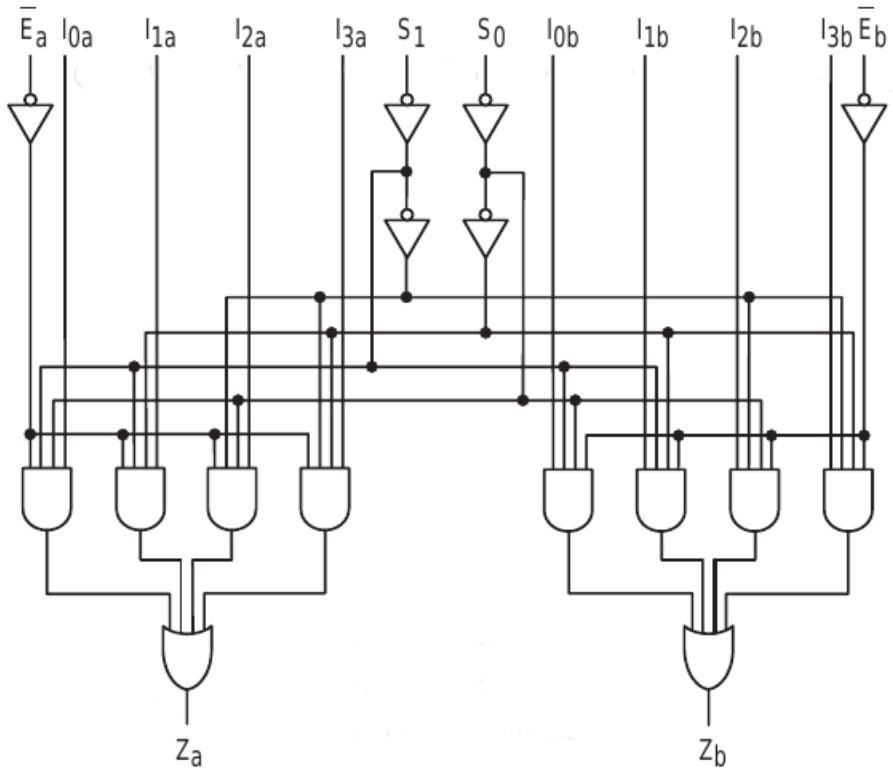
Multiplexers are the opposite of demultiplexers: they route a selected input to a single output.

The 74153 is typical of a multiplexer. It is, in fact, a dual 4-input multiplexer. The two halves aren't completely

independent, though: they share input selection inputs. Each multiplexer does have its own enable signal.



If we look inside, we can see how it works.



Here the data inputs are named I0a-I2a and I0b-I3b. The a inputs selectively get routed to Za, and the b inputs get routed to Zb.

The signal group S selects which input gets routed to the output, for each of the two multiplexers.

Finally, each side has its own active low enable input, when it's low the routed input appears on the output, when it's high the output is low. Unlike recent circuits, this one is active high.

Notice the use of the AND-OR combination that I wrote about in Digital Circuits 1 (https://adafru.it/BJk).

Adders and Subtractors

Computers are good at math. Computers, as we've seen, are made out of simple gates. Gates just do simple logic functions like AND and OR, not math like addition and subtraction. How do we reconcile this?

Simple... we make circuits out of logic gates that can do math. In this section we'll have a look at adders and subtractors.

This also provides a few good learning opportunities to bring out some lessons having to do with digital circuit design.

Let's start simply: adding 2 1-bit numbers. Recall from math class that adding numbers results in a sum and a carry. It's no different here. With two one bit numbers we have 4 distinct cases:

1. 0 + 0 = 0 with no carry
2. 0 + 1 = 1 with no carry
3. 1 + 0 = 1 with no carry
4. 1 + 1 = 0 with a carry

Since we are dealing with binary numbers, and each binary digit corresponds to a logic value, let's express this as a truth table:

A	B	S	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

What does that remind you of? Well, Sum is **A XOR B**, and Cout is **A AND B**!

Boolean expressions

It's common when writing boolean expressions to use operators rather than gate names:

- a vertical centered dot in place of AND
- + in place of OR
- a circled + in place of XOR
- a bar over a negated expression rather than NOT

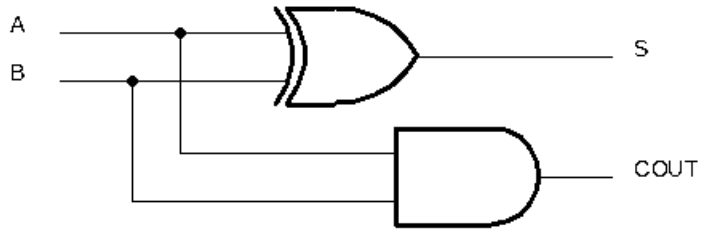
Using this notation, the expressions describing the above truth table are:

$$\begin{aligned} S &= A \oplus B \\ C_{out} &= A \cdot B \end{aligned}$$

We'll use this format from now on.

Back to the adder

So the logic circuit to add two one bit numbers would be:



Binary addition for adding more than single digit numbers is the same as you learned in school for decimal: you add the two corresponding digits ***and the carry from the digit adder to the immediate right*** to give a sum digit and a carry. So our single digit adder must support an incoming carry. What we have above is referred to as a *half adder*, since is really only does part/half of the job.

What we need to do is expand on this idea to include an incoming carry. Here's the truth table:

C_{in}	A	B	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Logic simplifcation

As your logic circuits (as well as the associated truth tables and equations) get larger and more complex, it's useful to have some tools and techniques to help simplify them. Why simplify them? Mostly to require fewer gates. That means fewer chips, less silicon, fewer connections, smaller boards, faster circuits, etc. The simpler you can make a circuit and get the same job done, the better.

One useful tool was introduced by Maurice Karnaugh in 1953: Karnaugh Maps (https://adafru.it/BJv). Let's go through an example to see how it works.

To use karnaugh Maps we need to put the truth table in terms of an OR of AND terms. These AND terms correspond to the rows in the truth table contain a logical 1 for the output in question. For the half adder we had:

$$\begin{aligned} S &= \overline{A} \cdot B + A \cdot \overline{B} \\ C_{out} &= A \cdot B \end{aligned}$$

And for the full adder the equations are:

$$\begin{aligned} S &= \overline{C_{in}} \cdot \overline{A} \cdot B + \overline{C_{in}} \cdot A \cdot \overline{B} + C_{in} \cdot \overline{A} \cdot \overline{B} + C_{in} \cdot A \cdot B \\ C_{out} &= \overline{C_{in}} \cdot A \cdot B + C_{in} \cdot \overline{A} \cdot B + C_{in} \cdot A \cdot \overline{B} + C_{in} \cdot A \cdot B \end{aligned}$$

A Karnaugh Map is a two dimensional table that has 2^n cells if there are n inputs. Adjacent rows and columns can differ by the negation of a single input. Here's the maps for the half adder:

$S(A,B):$

A \ B	0	1
0	0	1
1	1	0

$C_{out}(A,B):$

A \ B	0	1
0	0	0
1	0	1

The way it works is that the row labelled "A" corresponds to the A input being high, the other row corresponds to A being low. Similarly with the columns and the B input. The 1 in the Cout map corresponds to the case when both A and B are high.

There's no simplification to be done on the half adder, it's trivial. The full adder is another story. There are the maps for it.

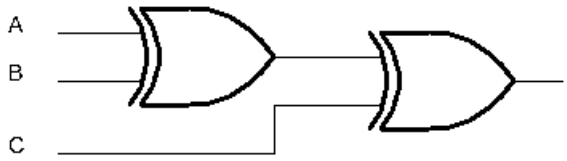
$S(A,B,C_{in}):$

B \ A, C_{in}	00	01	11	10
0	0	1	0	1
1	1	0	1	0

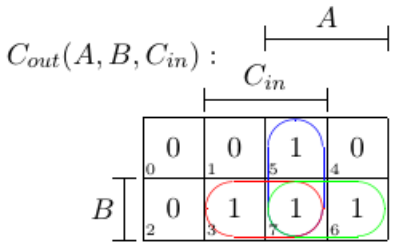
$C_{out}(A,B,C_{in}):$

B \ A, C_{in}	00	01	11	10
0	0	0	1	0
1	0	1	1	1

What you are looking for to simplify using a map is groups of 1s that are some power of 2 in size. In the Sum map above, there are none. Each 1 is separate. That pattern is indicative of an XOR of all three inputs. That can be achieved by chaining XOR gates.



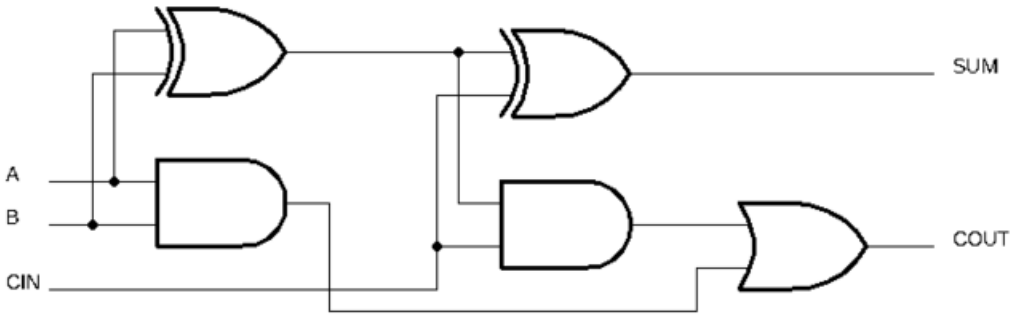
The Carry out is a different story, though. There are three groups of two:



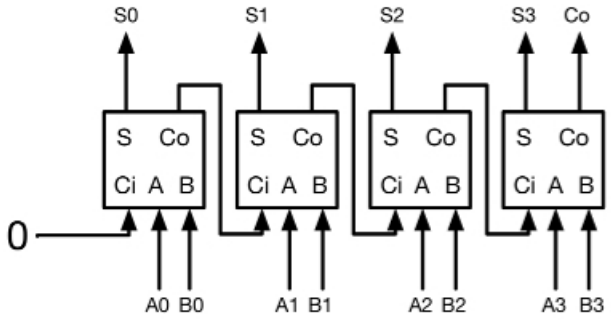
The green circle is the A . B term, leaving the other two 1s to be covered. In both those cases Cin is high and A and B differ. That is the definition of XOR, and so we can rewrite the equation to replace some ANDs, ORs, and NOTs with an XOR.

$$C_{out} = A \cdot B + C_{in} \cdot (\overline{A} \cdot B + A \cdot \overline{B})$$
$$C_{out} = A \cdot B + C_{in} \cdot (A \oplus B)$$

Interestingly, A . B and A XOR B are both outputs of a half adder as shown above. We need another XOR and another AND. In fact we can use two half adders along with an additional OR gate to build the full adder as shown below.



This full adder only does single digit addition. Multiple copies can be used to make adders for any size binary numbers. By default the carry-in to the lowest bit adder is 0*. Carry-out of one digit's adder becomes the carry-in to the next highest digit's adder. The carry-out of the highest digit's adder is the carry-out of the entire operation.



This is pretty typical of digital circuits that work on data: if you can design a circuit to work on single bit data, multiple copies can usually be used together to operate on bigger data.

* A CPU/MCU will have a carry bit in its flag register that can be used as the carry-in for addition operations. The carry out from such operations will be stored in that flag for future use. This allows operations on data larger than can be added at at one time.

Half subtractor

As before, I'll start with subtracting 1-bit numbers, generating a difference and a borrow. A will be the minuend and B will be the subtrahend. I.e.the circuit will compute A - B.

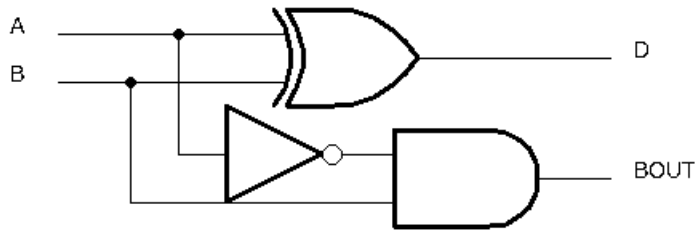
Here's the truth table:

A	B	D	B_{out}
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Converting that to equations:

$$\begin{aligned} D &= A \oplus B \\ B_{out} &= \overline{A} \cdot B \end{aligned}$$

This gives converts easily to a circuit very similar to the half adder. The only difference is the inverter on A for the computation of the borrow.



Full subtractor

Here's the truth table and corresponding maps for the full subtractor, which takes into account an incoming borrow. I'll skip the step of writing out the equations, as the maps can easily be constructed directly from the truth table.

A	B	B_{in}	D	B_{out}
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

$D(A, B, B_{in})$:

B \ A, B_{in}	00	01	11	10
0	0 (0)	1 (1)	0 (5)	1 (4)
1	1 (2)	0 (3)	1 (7)	0 (6)

$B_{out}(A, B, B_{in})$:

B \ A, B_{in}	00	01	11	10
0	0 (0)	1 (1)	0 (5)	0 (4)
1	1 (2)	1 (3)	1 (7)	0 (6)

As before, the next step is to find the groups in the map in order to simplify the logic.

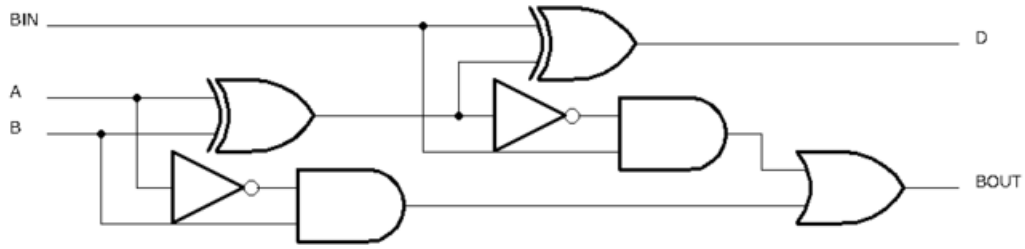
$B_{out}(A, B, B_{in})$:

B \ A, B_{in}	00	01	11	10
0	0 (0)	1 (1)	0 (5)	0 (4)
1	1 (2)	1 (3)	1 (7)	0 (6)

Taking the red group first, we have:

$$
\begin{aligned}
B_{out} &= \overline{A} \cdot B + B_{in} \cdot A \cdot B + B_{in} \cdot \overline{A} \cdot \overline{B} \\
B_{out} &= \overline{A} \cdot B + B_{in} \cdot (A \cdot B + \overline{A} \cdot \overline{B}) \\
B_{out} &= \overline{A} \cdot B + B_{in} \cdot (\overline{A \oplus B})
\end{aligned}
$$

From the half subtractor, we have various pieces of this, and can do the same thing we did with the full adder: use a couple half-subtractors and an OR gate:

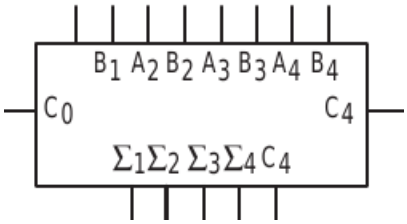


As with the full adder, full subtractors can be strung together (the borrow output from one digit connected to the borrow input on the next) to build a circuit to subtract arbitrarily long binary numbers.

Notice that subtractors are almost the same as adders. In fact a single circuit is generally used for both, with some "controllable invertors" being used to switch between operations. Going further than that, a CPU contains an Arithmetic-and-Logic-Unit (aka ALU) that takes two numbers, and an operation selector to configure it to perform one of a variety of arithmetic or logic operations.

Adder on a chip

This was an interesting exercise, but we'll never need to build an adder from gates. There are adder chips that can be dropped into our designs. The 7483 is one example.



Recap

There's a lot in this section:

- describing a circuit's desired behaviour with a truth table,
- extracting a AND-OR equation for each output column of the table,
- simplifying those equations using Karnough Maps,
- implementing the simplified equations using logic gates, and
- looking for similarities in existing designs to leverage work already done.

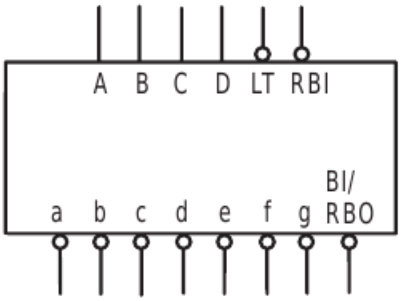
While you can always implement a truth table by ORing ANDed terms, with NOTs in the right places, it's usually not the most efficient. When using discrete gate ICs a one goal is always to minimize the chip count; fewer chips means a faster circuit, using less power, and generating less heat. These things aren't as relevant when you are using MCUs (they are when designing them, though), but it's a fun puzzle to solve.

Converters

We saw that decoders, multiplexers, and demultiplexers all deal with (depending on which they are) a single input or output corresponding to a selection number. Converters are similar on the surface: there's a set of input signals that represent a value (i.e. a binary number) and some outputs. Where they differ is that the input value isn't used to select one of the outputs; all the outputs are always relevant/significant. A converter does just what the name implies: convert one set of values to another. They embody a function in that you put a bit pattern in and get a corresponding bit pattern out.

A good example of a converter is a BCD* to 7-segment converter. Each group of 4 binary digits directly represents a single decimal digit. As such it can be converted to a pattern to be displayed on a 7-segment display.

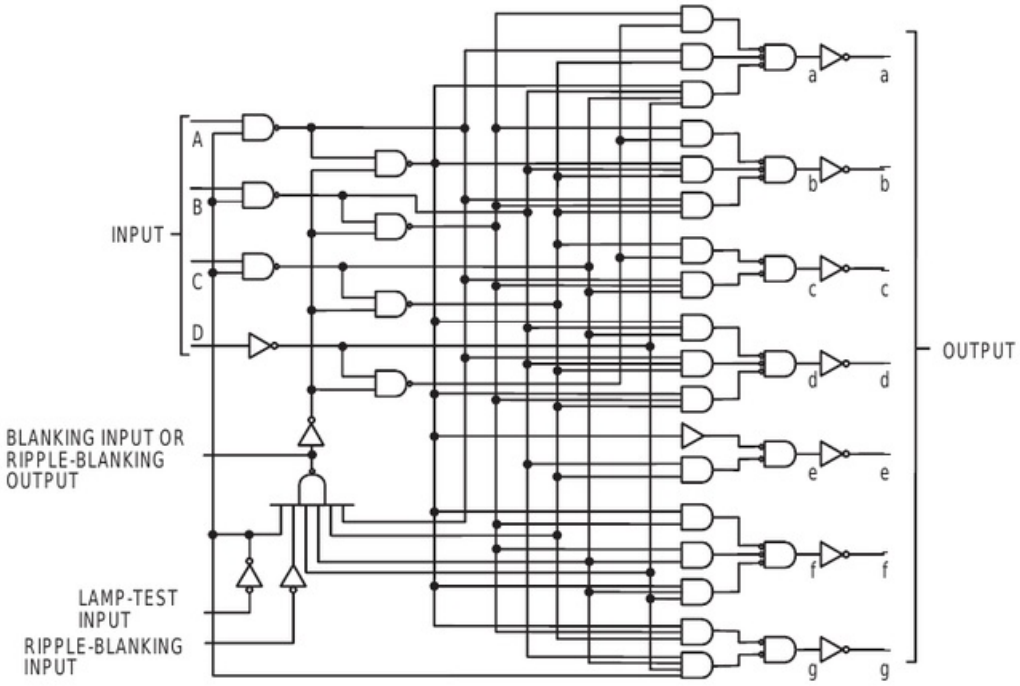
The 7447 is one such converter.



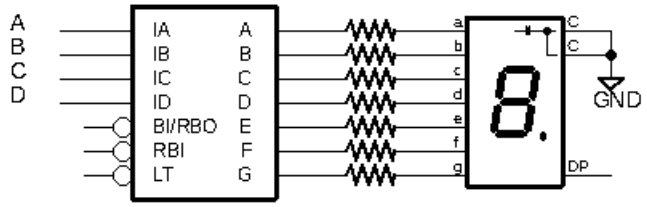
The DCBA inputs make up the value, 0-9, to convert. The a-g outputs connect to the segments in a 7-segment LED display. This input labelling is somewhat different than the use of A0, A1, A2, and A3 that we've seen previously. You'll find that there isn't just one standard naming convention. Sigh.

Don't worry about the LT, R, RBI, and RBO signals. They provide support for blanking leading and/or trailing zeros and aren't involved in the actual conversion.

If we look inside the chip, we can see that it uses simple gates that we have seen plenty of: NOT, AND, and NOR (the ANDs with inverted inputs).



This version of the BCD to 7-segment converter has the advantage in that it is designed to directly drive the LEDs by including output drivers. This makes the resulting circuit very simple.



* BCD: Binary Coded Decimal uses 4 binary digits to store a single decimal digit, i.e numbers 0-9. Each subsequent 4-bit group (usually called a nybble) represents a separate decimal digit. The decimal number 3571 would be represented in binary as 0x0DF3 though, but in BCD as 0b0011010101110001. Not nearly as concise but it does have advantages when decimal numbers (especially the input and output of them) are the focus.

Hands-on

For a hands on exercise you can try designing a 4-bit to hex 7-segment converter. There is just such an exercise written up on the Computer Science department site of The East Tennessee State University page (https://adafru.it/BJw). All the information you need is in that exercise. Once change: disregard where it talks about *your* segment; instead design the complete decoder (i.e. to drive all 7 segments). We'll be revisiting this decoder in a future project in this series.

I encourage you to work through the design and, if you can, purchase the required gate chips (I usually go to Digikey (https://adafru.it/BJr) for that) and breadboard your solution to see it in action. The digital input and output boards in part 2 of this series (https://adafru.it/BJl) can be handy for that.

Alternatively, there are various logic simulators available that you could use to test your design. That said, it's more fun to actually get something physical built and working.

Series Index

1. Binary, Boolean, and Logic (https://adafru.it/BJk)
2. Some Tools (https://adafru.it/BJl)
3. Combinational Circuits (https://adafru.it/BJm)
4. Sequential Circuits (https://adafru.it/BJn)
5. Memories (https://adafru.it/Bji)
6. An EPROM Emulator (https://adafru.it/BIT)
7. MCUs... how do they work? (https://adafru.it/BJo)