Introduction

This document contains a brief overview of how the various functional components of a processor can be constructed from Boolean logic gates. This information is not required for any of the CS24 coursework, but it is a pretty big claim that one can implement arithmetic, signal routing, and memory all with logic gates. Therefore, for the curious student, this document reviews some of the basic approaches to implementing such constructs entirely with logic gates.

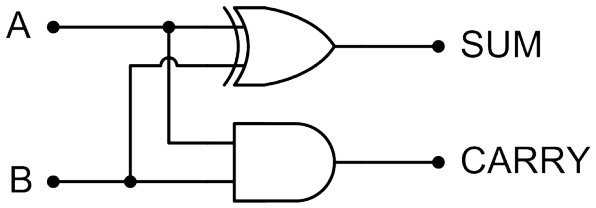
An important note about the circuits in this document: These are not necessarily the *best* ways to construct such functional components; in fact, these tend to be the simplest conceptually, but are also slower and more costly than the techniques frequently used in CPU design. For the purposes of CS24, we just want to see that these things are possible.

Adders

Binary addition can be implemented very easily with a simple circuit called a **half-adder**. A half-adder is responsible for adding two input bits, and producing a sum and a carry from the inputs. The half-adder truth table is as follows:

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

And, logic that computes this set of results is as follows:



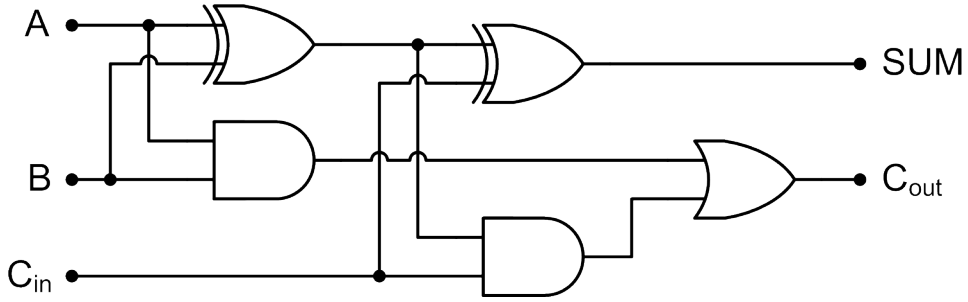
Now, to actually compute a *w*-bit sum from two *w*-bit inputs, we need to incorporate a carry-in into the mix. The truth table including carry-in, C_{in}, becomes:

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

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

Again, a very simple way to do this is to use two half-adders connected together, into a circuit called a **full-adder**. The first half-adder takes care of computing the sum and carry

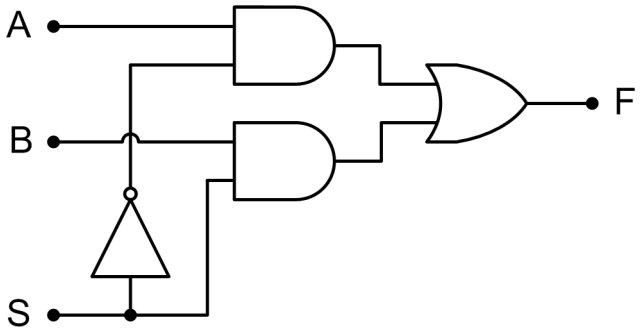
from adding A and B, and the second half-adder computes the sum and carry-out from adding together the result of (A+B) and the carry-in. The circuit is as follows:



Multiplexers and Demultiplexers

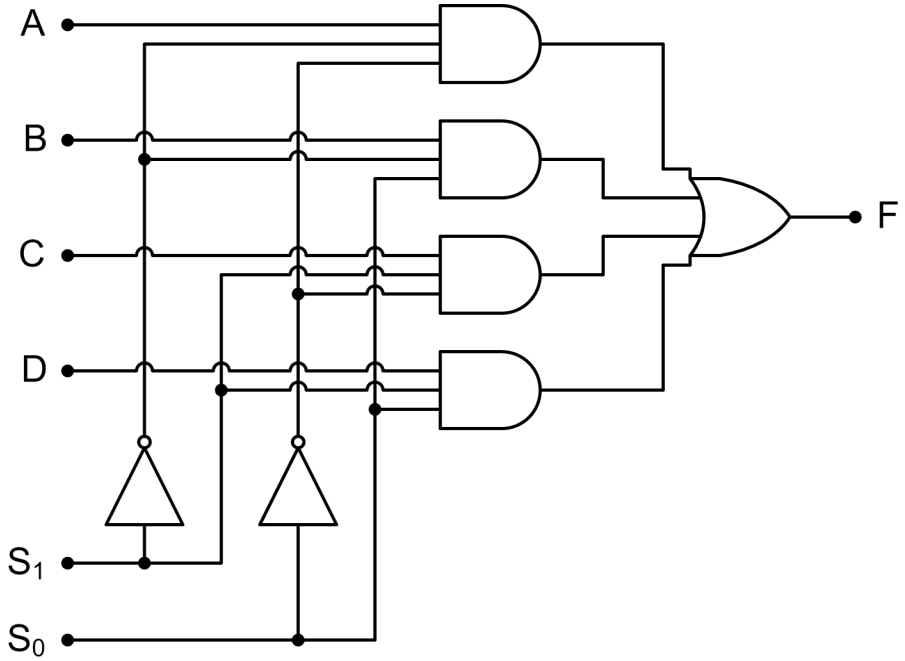
Another important feature for implementing a processor is the ability to route a signal to one of *N* possible outputs, or to route one of *N* possible inputs to a single output. This is the role that multiplexers and demultiplexers play.

Multiplexers (abbreviated MUXes) are circuits that take *N* inputs (almost always, *N* is a power of 2, to simplify addressing), and *A* address inputs ($N = 2^A$) to select which input is routed to the output. These circuits are quite straightforward to construct, although they can get rather tedious when routing a large number of signals. Here is a simple 2-to-1 multiplexer with one address line:



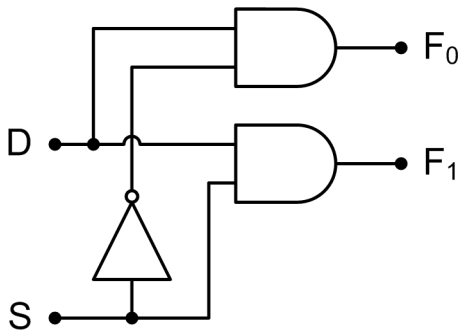
It is easy to see that when S is false, A will be routed through to F, and when S is true, B will be routed through to F. It couldn't get much simpler.

A 4-1 multiplexer uses the exact same concept, but is obviously substantially more complex:



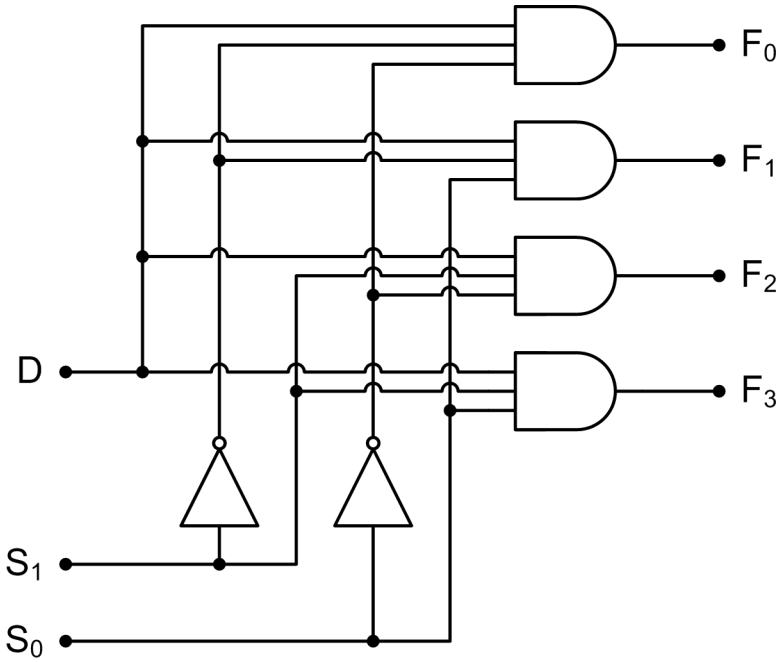
With this multiplexer, two address lines are necessary to select the proper input to route to the output. The inverters and AND-gates are configured to select A when (S_1, S_0) = 00, B when (S_1, S_0) = 01, C when (S_1, S_0) = 10, and D when (S_1, S_0) = 11.

Demultiplexers (abbreviated DEMUXes) are very similar to multiplexers, except that they route a single input signal to exactly one of *N* outputs. The logic is almost identical to the multiplexers; the changes are completely obvious. Here is a 1-to-2 demultiplexer circuit:



Whereas the 2-to-1 multiplexer had several data inputs, here we tie all AND-gates to the single data input D, and then we use S to select which AND-gate we will pass the signal through. Thus, if S = 0, F_0 = D and F_1 = false. If S = 1, F_0 = false, and F_1 = D.

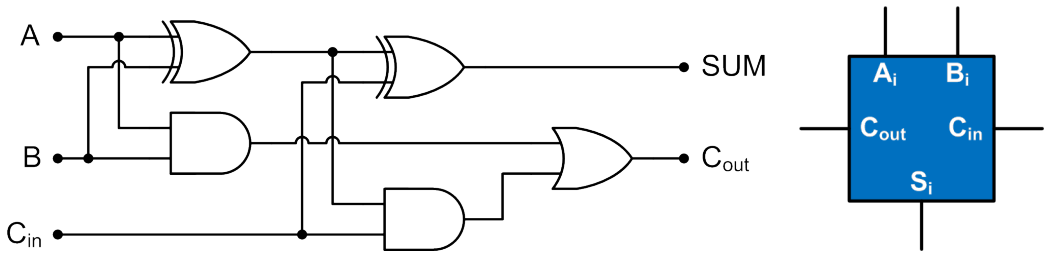
Correspondingly, here is a 1-4 demultiplexer:



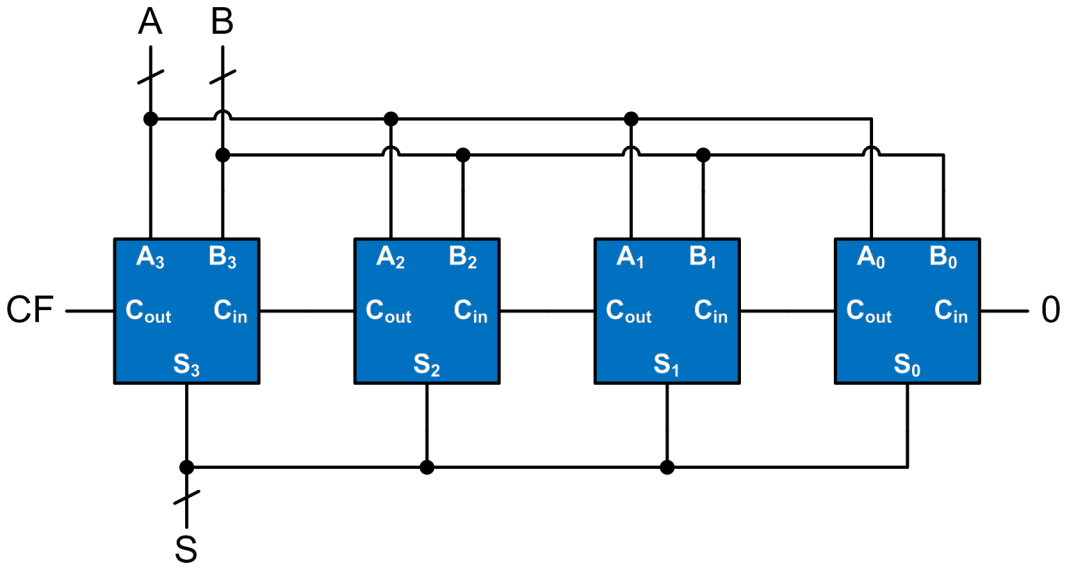
Arithmetic/Logic Units

In this section we won't go through the complete development of an Arithmetic/Logic Unit (ALU), but it is nice to at least illustrate the concept of how an ALU works internally. Specifically, ALUs have two data inputs, one data output, and a number of control inputs that specify what operation is performed. These control inputs frequently drive internal components like multiplexers and demultiplexers to control where logic signals are routed.

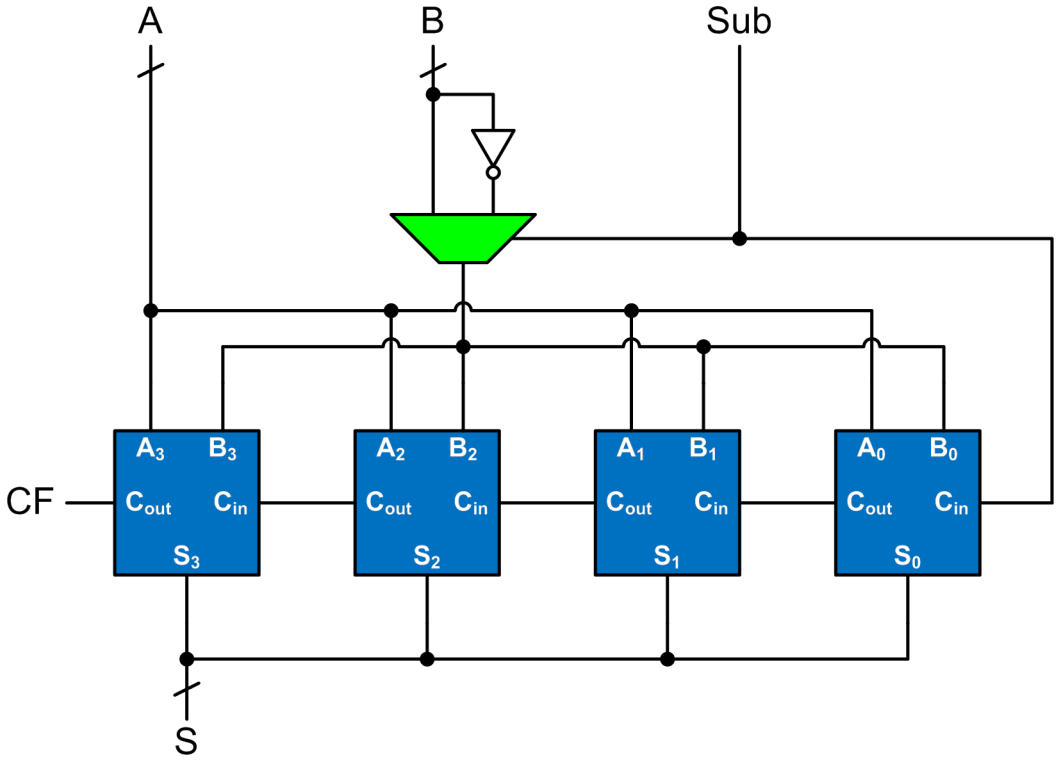
In this section we will look at a very simple example. We previously showed how a full adder can be built from logic gates; we will represent this adder as a simple box:



Using the full adder, it is very easy to build an n-bit adder. For example, a 4-bit adder:



The next question is, how can we use this adder and a small amount of additional logic to build an adder/subtractor, where we can switch between addition and subtraction with a simple signal? This is actually remarkably easy to accomplish with the above circuit, if we note that A – B = A + (-B). All we need to do is negate B before we feed it into the adder. This also turns out to be easy, because two's complement negation can be implemented as bitwise inversion, then adding one to the inverted result. Using this simple approach, we can extend the 4-bit adder in the following way:



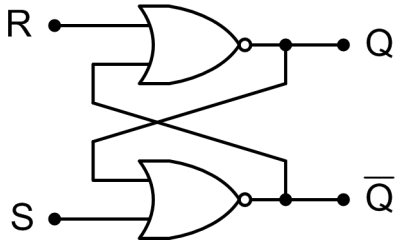
Now, if Sub is 0 then we do the same thing as the previous diagram: B is fed through to the adder, and the initial carry-in is 0. This is simple addition.

However, if Sub is 1 then we do two additional things: we invert B before sending it into the adder, and we set the initial carry-in to 1. In combination, these effectively negate B, so that the adder is performing A + (-B), or A – B. Now we have a simple configurable adder/subtractor! If Sub = 0 then we are performing addition, but if Sub = 1 we are doing subtraction.

Real ALUs greatly expand this idea, including an arithmetic subunit and a logic subunit, each of which is able to perform a wide variety of operations based on the control inputs. This is how we are able to construct the very core of the computer, the configurable logic that actually performs our computations.

Memories

The previous circuits have largely been straightforward, but the mechanism for constructing memory is a subtle one. The fundamental concept behind constructing a memory circuit is to have some kind of feedback loop in the circuit. For example, here is one of the most basic kinds of memory circuits, the **RS-latch**:



These logic gates are called NOR (short for "not-or") gates, and produce a result that is simply the inverse of the OR-gate. The output of each gate is connected to one of the inputs of the other gate, creating a feedback loop that maintains the current state of the circuit. Because we are using gates, we can also control the value of the output based on our control inputs R and S, which stand for "Reset" and "Set." Q is the current state of the latch.

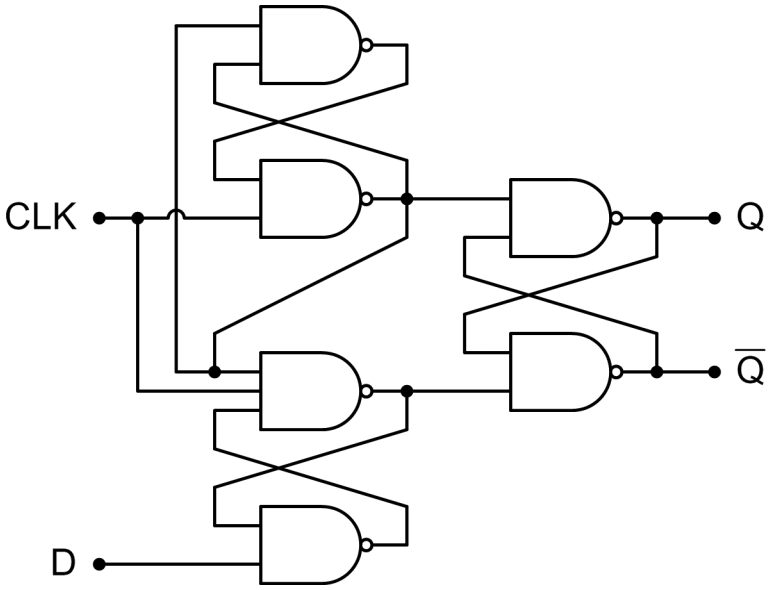
(If you are not familiar with this notation, names with a bar over them mean that the signal is inverted; i.e. false means "active" and true means "inactive." Thus, the Q without a bar over it is the current data value of the RS-latch, and the Q with a bar over it is the *inverse* of the data value.)

The behavior of the RS-latch is as follows:

S (set)	R (reset)	Behavior
0	0	Maintain current state
0	1	Set Q = 0
1	0	Set Q = 1
1	1	*Disallowed!*

While this simple circuit clearly illustrates the concept behind memory cells constructed from feedback loops, it is rather complicated to control, and state transitions are not governed by a clock signal. This is generally true of all **latches** (and there are a variety of latches). Thus, another category of memory circuits are also commonly used, **flip-flops**, which are controlled by a clock signal.

Here is an example of a **D flip-flop**, which takes a single data input, and stores the current value of the data input when the clock transitions from 0 to 1:



As before, Q is the current value stored in the D flip-flop. When the flip-flop is clocked (i.e. the CLK signal goes from 0 to 1), the current value of D is stored into the flip-flop. You might notice that this circuit has strong similarities with the latch circuit; flip-flops are built from latches that are connected to detect changes in the clock signal, so that the input will be stored only at the proper time.

As a final note, real memories tend to be much simpler than these circuits outlined above; these are rather expensive ways to construct memory cells. (Memory cells in SRAM are essentially two inverters connected in a feedback loop.) But, the basic principle is the same: a logic circuit is constructed that has a feedback-loop between two gates, causing the current state of the signal (be it true or false) to be reinforced until it is altered by external inputs.