Sequential Circuits

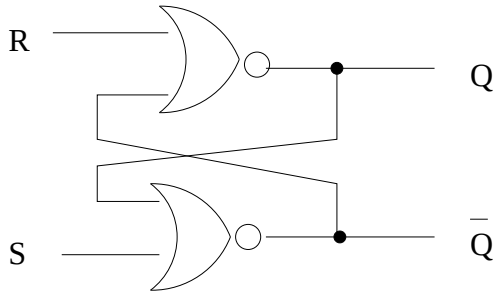
All of the circuits that I have discussed up to now are *combinational digital circuits*. For these circuits, each output is a logical combination of the inputs. We have seen that these circuits can do arithmetic and other operations. But these circuits are not powerful enough to build a general purpose computer.

A key element that is missing is memory. Consider two different ways of remembering something. Suppose I tell you a telephone number. One way for you to remember it would be to write it down, say on paper. Then later you could just look at the paper. Another way to remember the number which is good for short term, is just to repeat the number over and over to yourself.

You probably have some intuition about how a computer could write down a number. It could alter the state of some magnetic material, for example. What is probably less clear to you is how a computer could remember a number by repeating it to itself. Indeed this is the mechanism that is used in the central processor unit (CPU) as well as in the main memory (RAM). We will now turn to this technique.

RS latch (reset, set)

The basic way to design a circuit that tells itself a value over and over is as follows. Consider the following feedback circuit which is constructed from two NOR gates.[1] Such a circuit is called an RS latch. Just as gates are basic elements of combinational circuits, RS latches are basic elements of *sequential digital circuits*, and it is these circuits that implement memory.



On the left below is a truth table for a NOR gate, as a reminder. On the right are the values of Q and $\overline{Q}$ for various input combinations. It may seem strange that we label the two outputs as such, since if $R = S = 1$, then $Q = 1$ and $\overline{Q} = 1$ which is impossible. The reason we give the output these labels is that we will not allow $R = S = 1$. For the other three possible states of R, S, it is always the case that the outputs have complementary values.

A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

R	S	Q	$\overline{Q}$	
0	0	hold		← remember
0	1	1	0	←"set"
1	0	0	1	←"reset"
1	1	0	0	← not allowed

[1] In the lecture slides, I ramped up to this slowly by giving examples of feedback with single gates. Those details are omitted here.

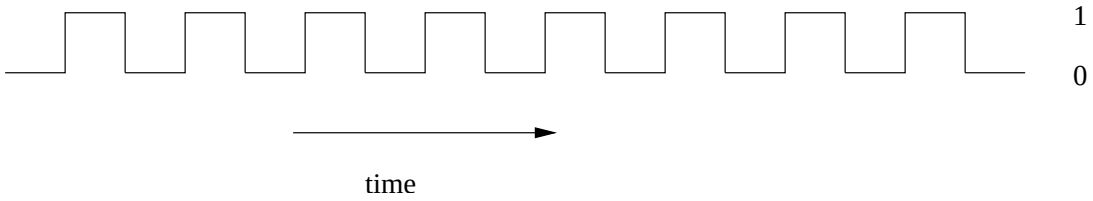
It is standard to use letters R and S and to call them "reset" and "set", respectively. The input $R = 1$ and $S = 0$ "resets" the output Q which traditionally means "gives it value 0". The input $R = 0$ and $S = 1$ "sets" the output Q which means "gives it the value 1". If you then put $R = 0$ and $S = 0$, you hold the value of Q *i.e.* you remember the value. Thus, an RS latch acts as a one-bit memory, holding the value Q as long as $R = 0, S = 0$. The feedbacks in the circuit are what I said before about repeating a number back to yourself to remember it.

The clock

All the circuits we've seen up to now can change their output values at any time their inputs change. You can imagine this has undesirable properties. If the outputs of one circuit are fed into other circuits, the exact timing and ordering of operations will be important. For example, consider the adder/subtractor we saw earlier. The sequence of carries takes some time, and we don't want the inputs to the adder to change (say, to add two new numbers) before all the carries have propagated and the result from the first sum is finished, and the result stored somewhere.

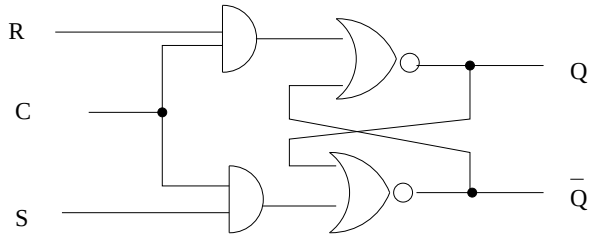
To prevent this problem, we need a mechanism of synchronizing the inputs and outputs of the circuits. Synchronization is achieved by a clock. A *clock* in a computer is a binary variable (a value on a wire) that alternates between O and 1 at regular rate. When you say that a processor has a clock speed of 3 GHz, you mean that the clock cycles between 0 and 1 at a rate of 3 billion times per second (GHz means "gigaHerz" where "giga" means billion.)

To understand how a clock signal is generated, you would need to take a course in electronics and learn about crystal oscillators. For COMP 273, we will simply assume there is such a clock signal is available and that it regularly oscillates between 0 and 1 as shown below. When we draw timing diagrams of the clock, we will make the ON interval of the same duration as the OFF interval.



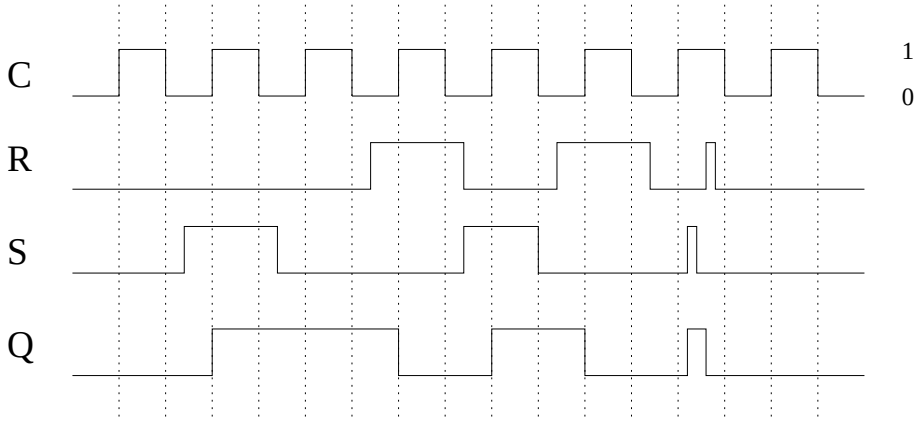
clocked RS latch

We can combine the clock and the RS latch to give us more control over when we write into the RS latch, namely we only write the clock is 1. The circuit below is called a *clocked RS latch*.



When $C = 0$, the circuit holds its value regardless of any changes in the R or S inputs. When $C = 1$, the circuit behaves as the RS latch shown above. Note that this does not yet avoid the $R = S = 1$ issue above.

Here is an example of how the output Q can vary as the R and S inputs vary. Note that transitions in Q can occur at the transition of C from 0 to 1 (in the case that S or R already has the value 1 at that time), or during the time that C=1, in the case that S or R becomes 1 while $C = 1$.

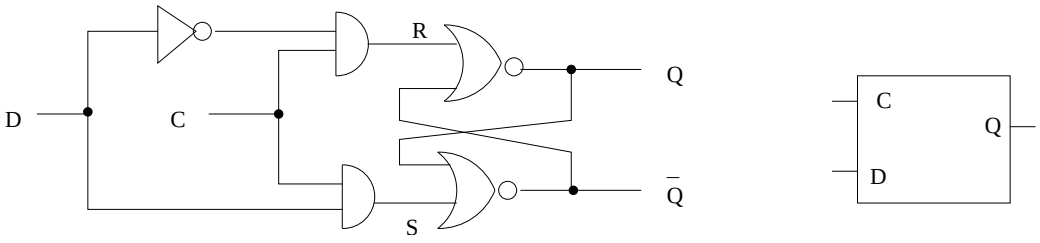


D latch (D for "data")

The RS latch allows us to write either a 1 or 0 value by inputing a 1 to S or R, respectively. The *D latch* allows us to write the value of a binary variable D. The detailed circuit is shown below on the left. The symbolic representation commonly used is shown on the right.

If $D = 1$, then we want to write 1 and so we make the S input be 1 (recall the clocked RS latch). If $D = 0$, then we want to write 0 and so we should make the R input be 1. This is done by feeding D into the S input and feeding the complement of D into the R input.

When $C = 0$, the outputs of the two AND gates is 0 and so Q holds its value, regardless of what is D. When $C = 1$, the value D is written into Q.



This *D latch* circuit is extremely important, so lets repeat the idea. When $C = 0$, the data input D does not get written to Q and the previous value of D is held. A write only occurs when $C = 1$, and in that case the current value of D is written and can be read. (Also notice that, by design,

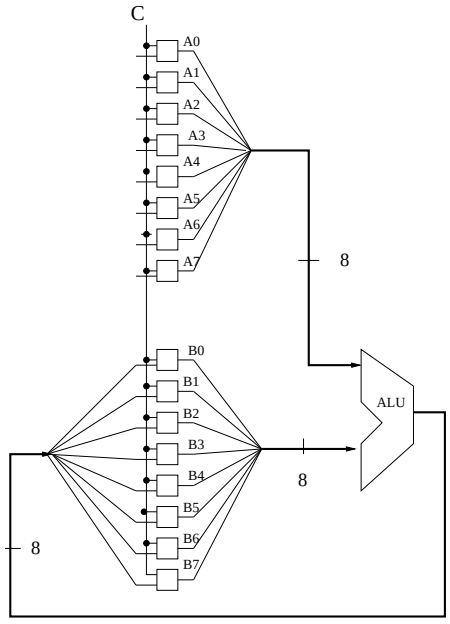
it is impossible for the R and S inputs to both be 1 here, so our earlier concern about these values being simultaneously 1 now goes away.)

The D latch provides some control on when we can write, but it still not good enough. When the computer's clock signal C is 1, data passes freely through the circuits. That is, Q takes the value D.

Let's consider an example of why this is problematic. Suppose we wish to implement an instruction such as

$$\texttt{x} := \texttt{x} + 27;$$

We could implement it by putting the value of `x` into a circuit made out of an array of sequential circuits. We will discuss the correct way of doing this over the next few lectures. For now, let's look at an *incorrect* way of doing it, namely by representing `x` and 27 as binary numbers and storing the bits of each number in an array of D latches. Say we have somehow managed to store 27 in the A array and `x` in the B array. (Each of the little boxes in the figure below is supposed to be a D latch. We when say "store a value" we mean that the Q outputs have that value.) We feed these two arrays into our ALU (arithmetic logic unit) and feed the answer back into the B array, in order to execute the above instruction. What happens?

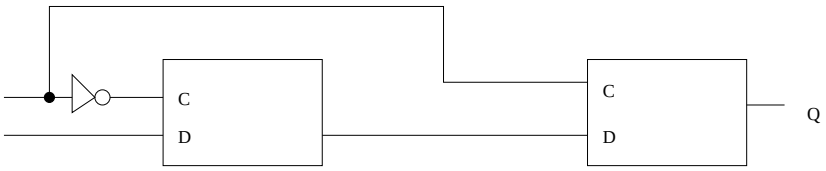


When $C = 1$, the sum bits S_i computed by the adder will write their values back into the B_i units. But as long as $C = 1$, these newly written values will go right through into the adder again. They will be summed once more with 27, and will loop back and be written into the B array again, etc. Moreover, the carry bits will take time to propagate the result and so we cannot even think of succession of additions. What a mess! When C becomes 0 again, and writes are no longer allowed, the values that are in the B units are not what we want.

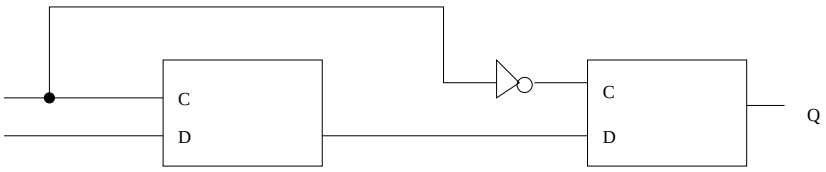
D flip-flop

We need a mechanism that prevents data from being read from a D latch at the same time as it is being written to a D latch. The mechanism that achieves this is called a *D flip-flop*. The two basic types of D flip-flop are shown below. In each case, we have a pair of D latches with the Q output value of the first being the D input of the second, and with the clock C being inverted for one D latch but not the other.

rising edge triggered



falling edge triggered

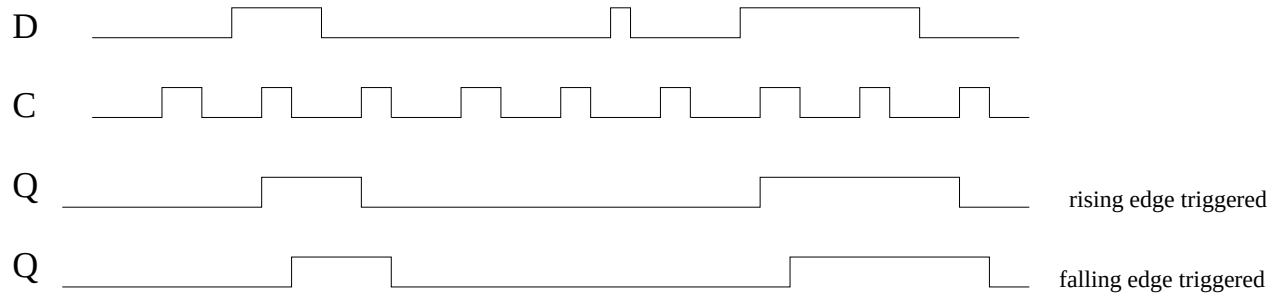


Consider the rising edge triggered case. When $C = 0$, the D input is written to the first D latch. The value is not written into the second D latch, however. When $C = 1$, the Q output from the first D latch is now written into the second D latch, but the D input to the first D latch can no longer be written because the clock has been inverted. We say "rising edge triggered" because the output Q of the second D latch can only change on the C transition from 0 to 1. (Note that once $C = 1$, Q cannot change any longer because Q is merely reading the value that had been stored in the first D latch when C was 0, and that value cannot change because the C input to the first D latch has been inverted to 0. Very clever!)

The reasoning is similar for the falling edge triggered D flip-flop. When C goes from 0 to 1, the first D latch is written to, but the second D latch holds its (previous) value. When C drops from 1 to 0 again, the value written to the first D latch is now written to the second D latch and can be read off from the output Q.

For this mechanism to guarentee the synchronization we desire, the interval of a clock pulse has to be long enough that the combinational circuits (such as adders, multiplexors, etc) between the flip-flops have finished computing their output values. In the example above for `x := x + 27`, each of the bit memory units in fact would be a D flip-flop rather than a D latch. The width of each clock pulse must be long enough for the ALU to compute the sum.

We have discussed how to represent numbers using bit strings. We have also seen combinational circuits that can add or subtract numbers by performing logical operations on these bit strings. Then we saw how sequential circuits (flip flops) could be used to store bit strings. Whereas the



values entering and leaving combinational circuits occurs as soon as the values are ready, writing into sequential circuits (flip flops) is synchronized by a clock. All sequential circuits use the same clock.

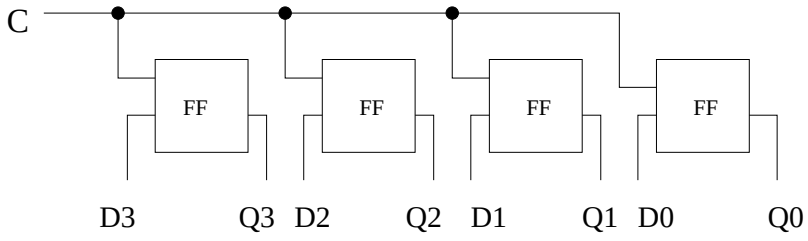
Thus, the picture that you should have in your head for the circuits in a computer is a graph of combinational and sequential circuits. Combinational circuits map input to output values without any control from a clock. The clock is used only to control when new values are written into the sequential circuits. Note that the clock cycle should be long enough that all combinational circuits have time to settle on values.

Registers

To perform an addition, you can imagine that the numbers/bits are read from a set of flip-flops (sequential circuit), go through an adder circuit (combinational circuit) and are written into another set of flip-flops, or even possibly the same flip flops. I gave a sketch of this above, and we'll see how this works in MIPS in coming lectures.

A set of flip-flops that holds the values of some variable is called a *register.* The key property of a register is that its flip-flops are written to *as a unit*. There is no way to write to the individual flip-flops in a register. On any clock cycle, you either write to every flip-flop in the register, or you write to none of the flip-flops in the register.

The sketch below shows a four bit register. Each of the square boxes represents one D flip-flop. [2] Each has a clock input C and data input D and a Q output. Next lecture we will examine several types of registers.



[2] I have written FF in each box to emphasize its a flip-flop, not a D latch. I will eventually redo this figure to make it consistent with the slides.