Sequential Circuits

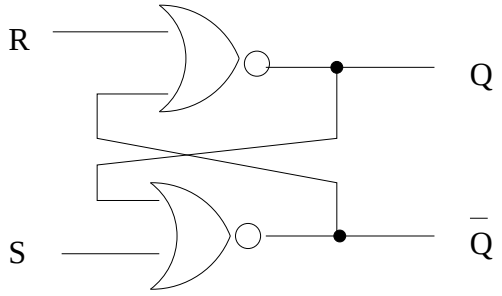
All of the circuits that I have discussed up to now are *combinational digital circuits*. For these circuits, each output is a logical combination of the inputs. We have seen that these circuits can do arithmetic and other operations. But these circuits are not powerful enough to build a general purpose 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 less clear to you is how a computer could remember a number by repeating it to itself. Indeed this is the mechanism that is often used. 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 a feedback circuit which is constructed from two NOR gates. Such a circuit is called an RS latch.



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 $Q = 0$ and $\overline{Q} = 0$ which is impossible, and indeed in the slides I used the symbol Q' instead of $\overline{Q}$ when introducing the RS latch. The reason it is ok to give the output labels Q and $\overline{Q}$ is that eventually we will not allow $R = S = 1$.

The way RS latches are used is that at most one of R and S have value 1. When exactly one of them has value 1, it determines Q and $\overline{Q}$. When both have value 0, then Q and $\overline{Q}$ keep the values that they had when one of R and S last had the value 1. See slides for illustration.

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

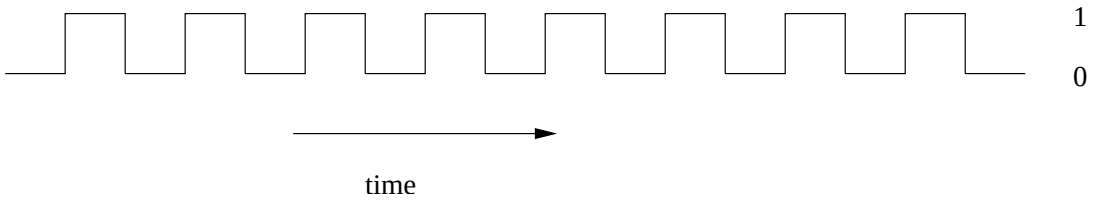
It is standard to use letters R and S and to call them "reset" and "set", respectively. The input $R = 1$ and $S = 0$ "resets" the output Q which traditionally means "gives it value 0". The input $R = 0$ and $S = 1$ "sets" the output Q which means "gives it the value 1". If you then put $R = 0$ and $S = 0$, you hold the value of Q *i.e.* you remember the value. Thus, an RS latch acts as a one-bit memory, holding (locking, latching shut) 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.

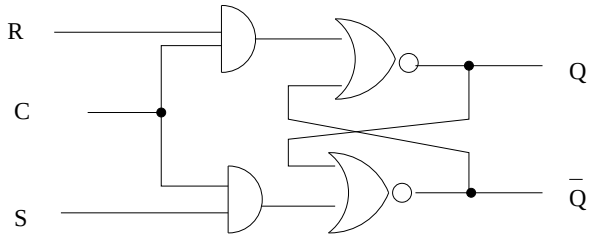
To prevent this problem, we need a mechanism of synchronizing the inputs and outputs of the circuits. Synchronization is achieved by a clock. A *clock* in a computer is a signal (a value on a wire) that alternates between O and 1 at regular rate. When you say that a processor has a clock speed of 3 GHz, 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 such a clock signal is available and that it regularly oscillates between 0 and 1 as shown below. The ON interval has the same duration as the OFF interval.



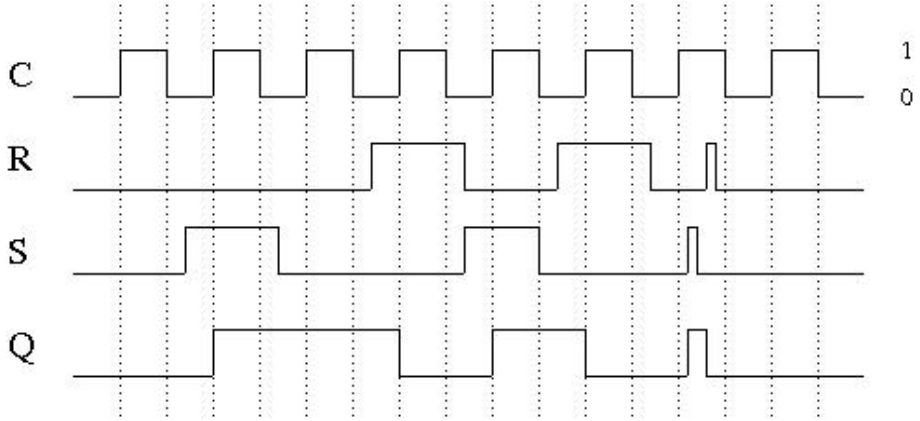
clocked RS latch

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 when 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 $R = S = 1$ situation which I mentioned 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 $C = 1$. One tricky case occurs during the second to last clock pulse. During that pulse, there are two brief events to note: first S goes to 1 temporarily, and then R goes to 1 temporarily. Note what happens to Q. When S goes to 1, Q goes to 1, and when S drops back to 0, Q stays on (memory). Q falls to 0 a short time later namely when R goes to 1 since then Q is reset.

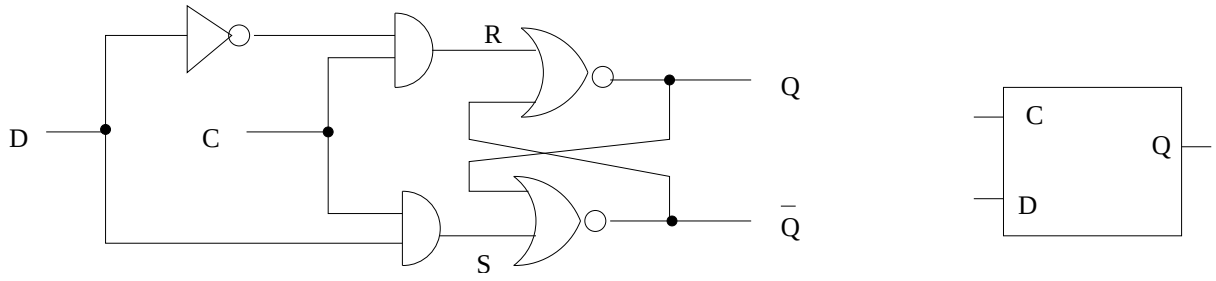


D latch (D for "data")

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 (latches) its value, regardless of what is D. When $C = 1$, the latch is open and the value of D is written into Q.

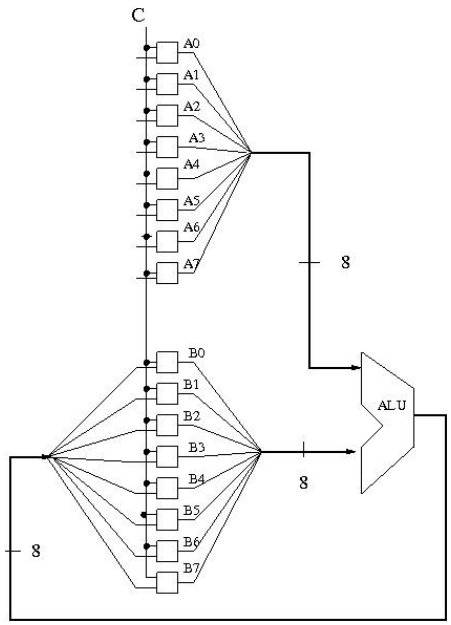


This *D latch* circuit is extremely important, so lets repeat the idea. When $C = 0$, the data input D does not get written to Q and the previous value of D is held. A write only occurs when $C = 1$, and in that case the current value of D is written and can be read. Also notice that, by design, it is 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. Let's consider an example of why this is problematic. Suppose we wish to implement an instruction such as

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

We could implement it by putting the values of `x` and `y` into circuits 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 `y` as binary numbers and storing the bits of each number in an array of D latches. Say we have somehow managed to store `y` 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?

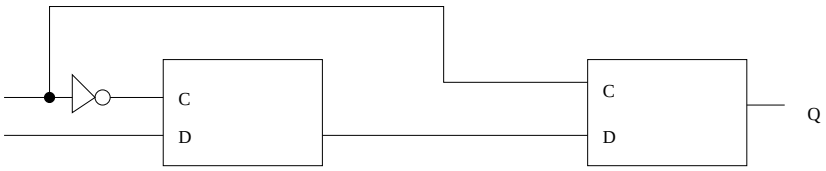


When $C = 1$, the sum bits S_i computed by the adder will write their values back into the B_i units. But as long as $C = 1$, these newly written values will go right through into the adder again. They will be summed once more with `y`, and will loop back and be written into the B array again, etc. Moreover, 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.

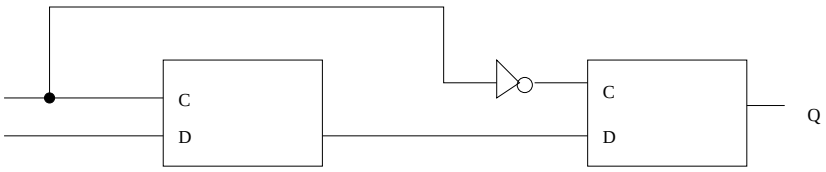
D flip-flop

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



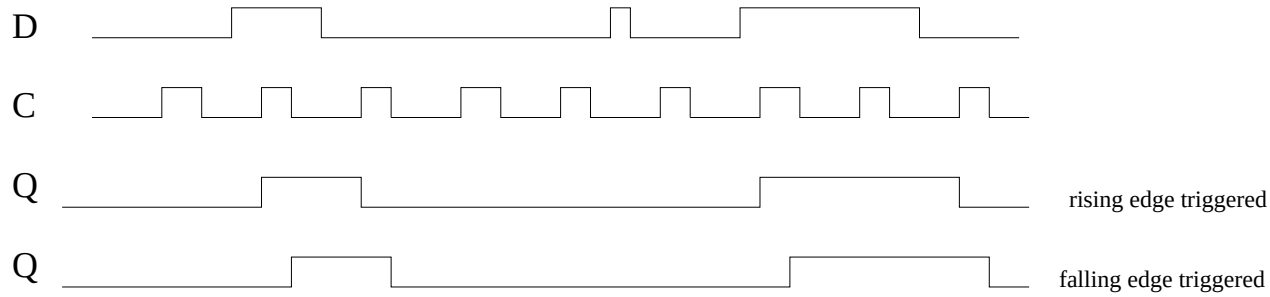
falling edge triggered



Here I discuss just the falling edge one. (The argument of what happens for the rising edge one is very similar.) When C goes from 0 to 1, the first D latch is written to, but the second D latch holds its previous value because the inverted clock input to the second D latch goes from 1 to 0. When C drops from 1 to 0, the value that had just been written to the first D latch now is written to the second D latch whose clock input rises from 0 to 1 and this data value appears in the output Q. Note that when C drops from 1 to 0, the value that had just been written into the first D latch is held even though the input to the first D latch might have changed. The reason the previous value is held is that C is 0. We say this flipflop is falling edge triggered because the Q value changes when the clock falls from 1 to 0.

For this mechanism to guarantee 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 flip-flops have finished computing their output values. Consider again the example above for `x := x + y`. Each of the bit memory units in fact would be a D flip flop rather than a D latch. We would now have synchronization. During a clock pulse (C=1), the adder circuit would compute `x + y` and at the end of the clock pulse when C falls from 1 to 0, then result would be written into `x`.

Below is an example of timing diagram for D and the value stored in a flip flop. I have included both the rising edge case and falling edge case.

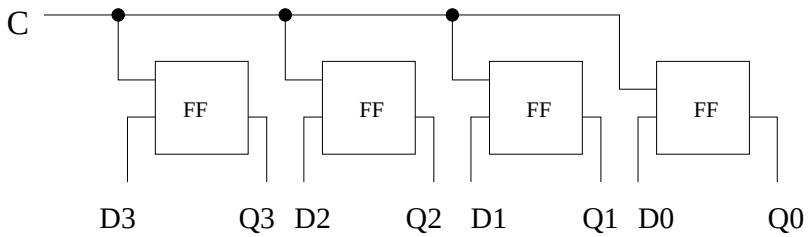


Registers

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 computer 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 selectively 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.

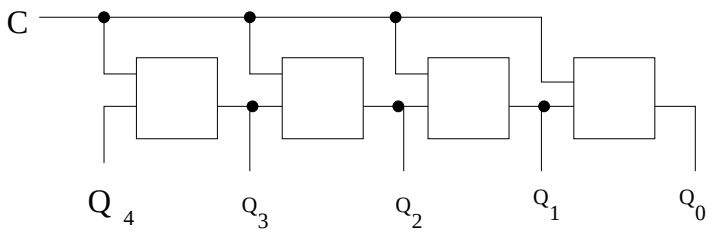
The sketch below shows a four bit register. Each of the square boxes represents one D flip-flop. Here I have written FF in each box to emphasize its a flip-flop, not a D latch. (In the slides, I didn't write "FF", and in future I won't write "FF".) Each flip flop has a clock input C and data input D and a Q output.



Shift registers

A *shift register* is a special register that can shift bits either to the left or right. We have seen that shifting bits of an unsigned number to the left multiplies that number by 2, and shifting bits to the right divides the number by 2. You can imagine it might be useful to have specialized circuits for computing such shifts.

The figure below shows a four bit *shift right* register. When we shift right, we need to specify the D input for the leftmost flip-flop. In the figure below, the leftmost bits is filled with the value of variable Q_4. You can design the circuit to fill it with whatever you want (0, 1, or even wraparound from Q_0, etc) and select one of these with a multiplexor.



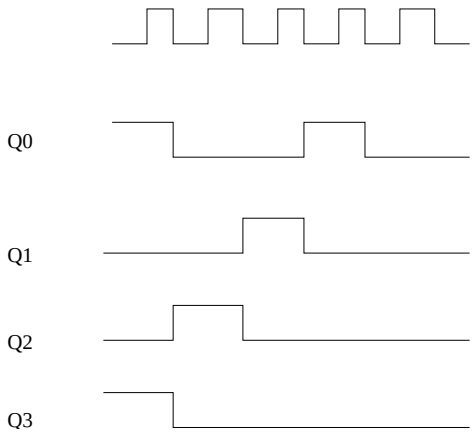
Note that for "right" and "left" to be meaningful, we need to use the common convention that the least significant bit (that is, Q_0) is on the right.

Examples

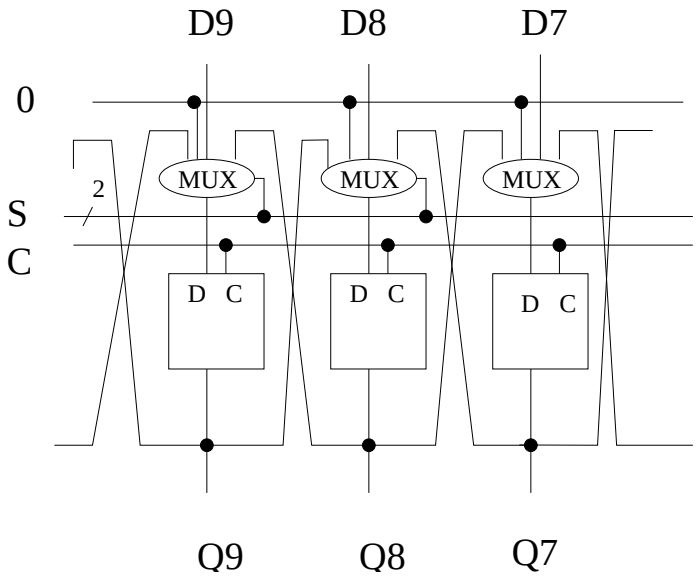
Show a timing diagram of the contents of the above register. Assume the initial values in the register are

$$(Q_3, Q_2, Q_1, Q_0) \;=\; (1, 0, 0, 1)$$

and assume the flip-flops in the register are falling-edge triggered. Also assume that, over the close pulses shown, the value of Q_4 is 0. (In general, Q_4 is not always 0, but it makes the timing diagram easier to understand if we think of it as 0. Essentially we are just following the initial Q_3 value of 1 as this 1 value moves through Q_2 to Q_1 to Q_0.



Let's look at a more general shift register that can either shift left or right, and that can also be cleared (all bits set to zero) and that can also be written to as a unit (write to all bits in one clock cycle). Notice that we have tilted each flip-flop on its side in order to simplify the figure. Also notice that we need a 2 bit selector to specify which of the four operations you want to perform. The same selector is applied to each multiplexor. I have only shown bits 7, 8, 9 but you can imagine extending the figure to other bits.


D9
D8
D7
0
MUX
MUX
MUX
2
S
C
D C
D C
D C
Q9
Q8
Q7