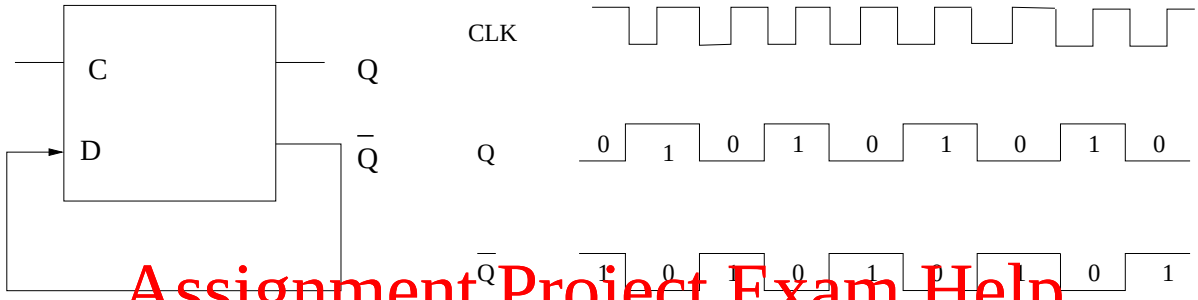
Today I will finish off our discussion of registers, and then move on to discuss larger memories.

T flip-flop (toggle)

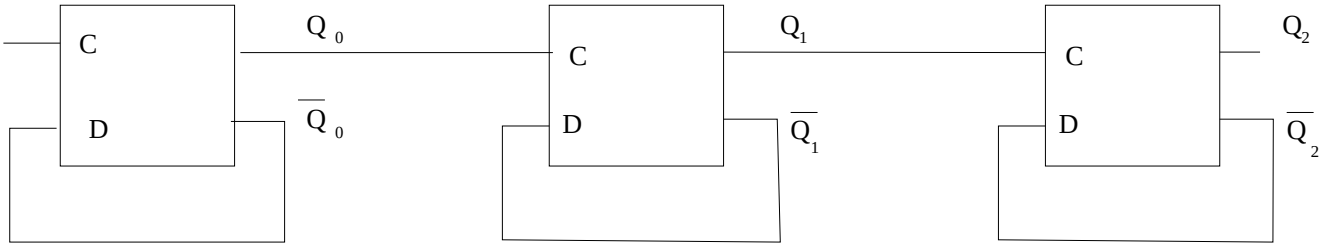
The circuit below on the left shows a D flip-flop, such that the data input D comes from the complement of the stored value Q. At every clock cycle, the Q value changes, namely it takes the value $\overline{Q}$. If the D flip flop is falling edge triggered, then the value changes when the clock goes from 1 to 0 which is the case shown on the right below. If the D flip flop is rising edge triggered, then the value of Q changes when the clock rises from 0 to 1.



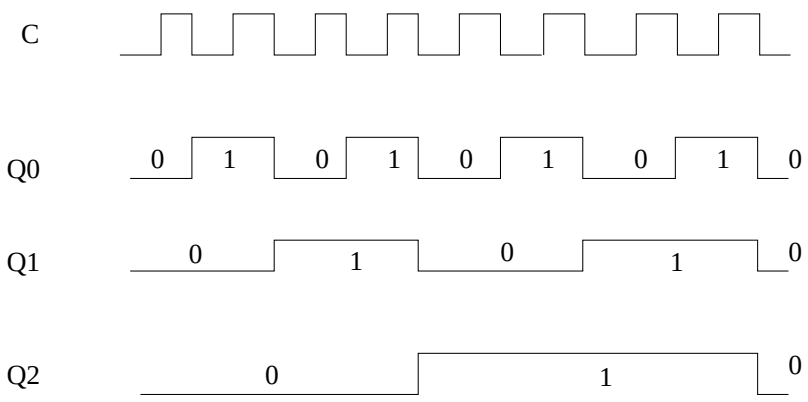
Such a flip-flop is called a T flip-flop where T is for *toggle*. Note that the T flip flop changes its value at half the rate of its clock input.

Counters and timers

Suppose we make a register out of T flip flops as shown below. The values in the flip flops are are labelled $Q_{n-1} \ldots Q_2 Q_1 Q_0$. In the figure, the least significant bit is on the left, to make the diagram more easy to read. Note that we are using the Q_i value as a clock input for flip flop $i+1$. So, Q_0 (on left) toggles its value once per clock cycle, and more generally Q_{i+1} toggles its value whenever bit Q_i changes from 1 to 0 (falling edge).

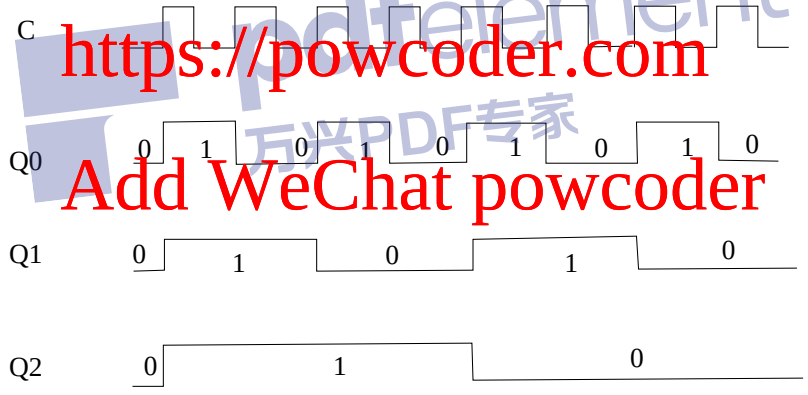


The timing diagram for this circuit is shown below. Suppose the flip flops are *falling edge* triggered. The values of $Q_2 Q_1 Q_0$ represent binary numbers and, for each successive falling edge of the clock, the triplet encodes a binary number. By inspection we can see that the sequence of numbers is 000, 001, 010, 011, 100, 101, 110, 111 which is just the numbers from 0 to 7. Thus we have a counter circuit!



What would happen if we were to use *rising edge* triggered flip-flops instead? Q_{i+1} would toggle its value whenever Q_i rises from 0 to 1. You can verify for yourself that this produces a counter which counts down (and which wraps around from 000 to 111). We call this a *timer* since it decreases in value. This is the circuit that is used for the timer on your microwave oven or wristwatch.

[ASIDE: in the lecture, someone asked why there is an initial jump from 000 to 111. See if you can answer that question by thinking about what happens on the first rising edge of the C.]

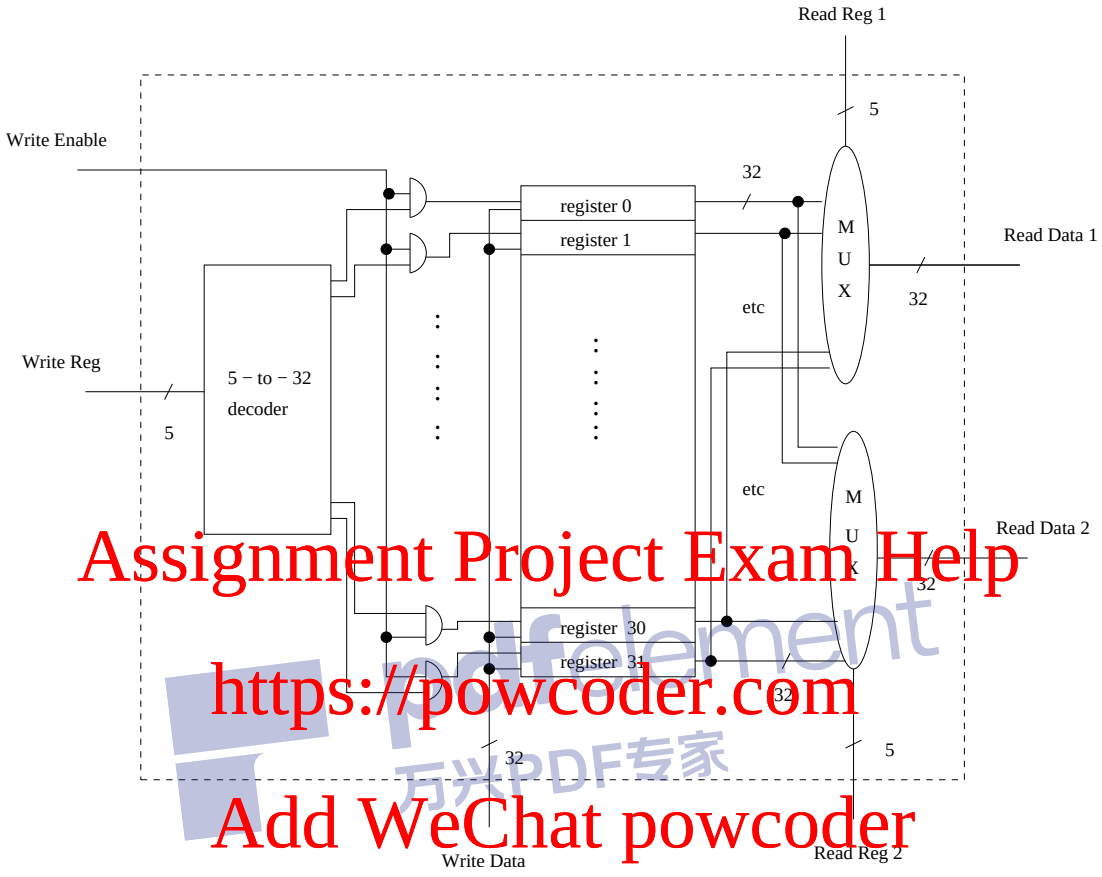


Register array

One common way in which we use registers is to hold the values of variables in a program. Suppose we wish to add two variables and store the result in another variable *e.g.* `x := y+z`, or `x := x+z`. Suppose the value of each of the variables is stored in a register. To perform the addition, we must read the values from two registers, feed the two values into an adder (ALU), and then write the result into a register. We will do all of this is done in one clock cycle.

A few lectures from now, we will use the MIPS processor to illustrate how such things are done. In MIPS, the registers that are used in your programs are grouped into an array. There are 32 registers in the array – each register stores 32 bits. There is no significance to the fact that the number of registers is the same as the number of bits in each register.

We next discuss how to read from and write to these 32 registers. We will discuss the various elements in the following diagram.



Reading from registers

Suppose we wanted to read the data stored in one of the 32 registers, and feed it into some circuit, say an adder. We can access this data by taking the 32 bit outputs from all 32 registers and feeding them into a multiplexor (selector). We select a register by inputting the 5 bit code of the desired register into this multiplexor ($2^5 = 32$). For example, if we want register 17 then we input bits 10001. We call this 5 bit code ReadReg1.

Rather than drawing all 32 lines for the output of each of the 32 registers, we just draw one line from each register and indicate that the line is 32 bits wide. Remember that you cannot access the individual bits of a register. You have to access all 32 bits at once. So, we might as well just draw one line for each register.

If we are adding two numbers – or more generally operating on two bit strings contained in two registers – then we need to read from two registers, not just one. (The two registers may be the same, such as `y = x + x`.) To access two registers simultaneously, we need two 32 bit multiplexors as in the figure above. The two multiplexors select the two 32 bits, and feed them into a 32 bit ALU which contains the adder circuit *which is not shown in the figure above, but see the figure on the next page*. Each of the two multiplexors needs to be told which register to select, and for this each

needs 5 bits (again $2^5 = 32$). Thus a 5-bit coded number must be fed into each of two multiplexors. In the figure, these are called ReadReg1 and ReadReg2. The 32 bit values that are read out are labelled ReadData1 and ReadData2.

Writing into a register

When we add two numbers, we then want to store the sum. We store the sum in a register. For example, if we were computing `x = x+y`, we would store the result in the same register as one of the operands. Or, if we were computing `y := x + z`, then we would write the sum into a different register than the two operand registers.

To write into a register, we take a 32 data result of the operation (e.g. the sum bits S_i which are output from the adder) and we feed these into the D inputs of the flipflops of the desired register. We specify which register to write to using a 5 bits, labelled WriteReg in the figure. Note that the WriteReg signal is decoded into $2^5 = 32$ lines, one of which is 1 and the rest are 0. e.g. if WriteReg has the value 13 i.e. 01101, then the 13th line is 1 and the rest are 0.

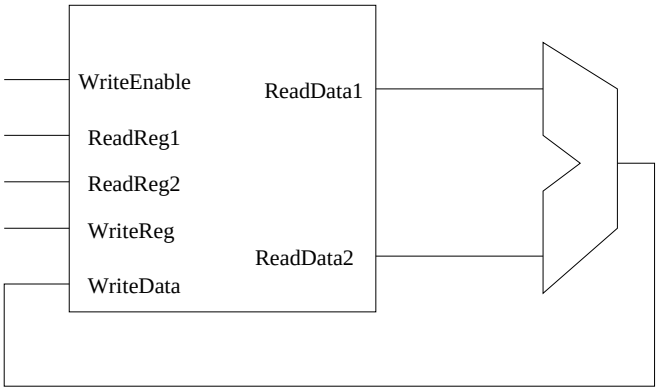
We do not necessarily want to write into a register: we will see lots of examples later when we learn how MIPS works. We can control when we write into a register by requiring that certain conditions are met. Firstly, we only write into the one selected register. Second, we only write when we have some signal – call it WriteEnable – that says it is time to write.

As we will see later, there may be many situations in which we do not want to write to a register. For this reason, we have given the clock input a more general name here: WriteEnable. WriteEnable would be the output of some other circuit which depends on the computer's clock, but also on other variables. Think of

$$\text{WriteEnable} = C \cdot Y_1 \cdot Y_2 \ldots$$

where the Y_i are other variables. Sometimes I will not bother writing the clock explicitly to avoid clutter in the diagram. [**ASIDE: Unfortunately I forgot to mention this important point in the lecture. I have added a slide mentioning it and also added a slide showing the figure above which includes the read and write circuits together.**]

While the details in the above figures are important to understand, sometimes it is easier not to clutter your head with the details and instead just think of the register array as the contents of a "black box."



Finally, it is natural to classify the inputs and outputs into three types:

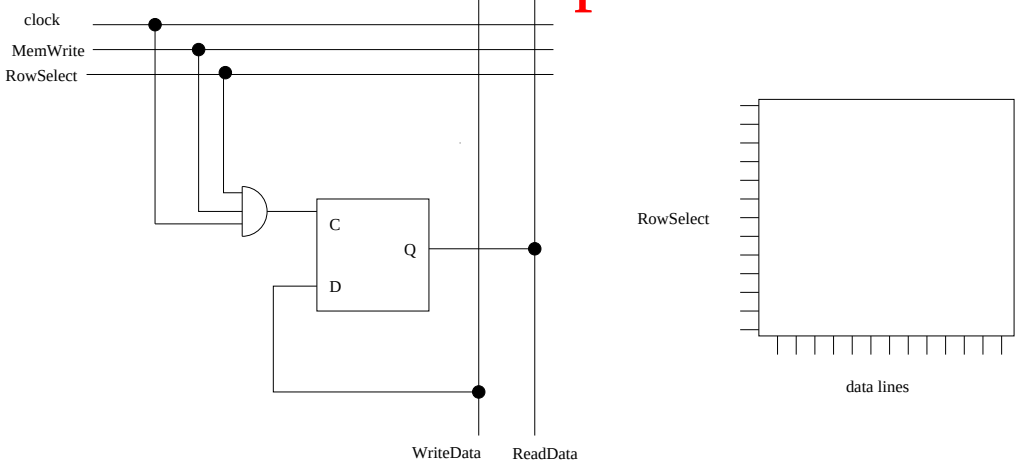
1. *addresses*: ReadReg1 (5 bits), ReadReg2 (5 bits), WriteReg (5 bits)
2. *data*: WriteData (32 bits), ReadData1 (32 bits), ReadData2 (32 bits)
3. *control*: WriteEnable (1 bit)

Later when we look more generally at how the computer works, this three way classification will help us think about what the various signals do.

Let's next turn to a slightly different design for memories. The above design is feasible for small memories such as a register array. But it won't work for larger memories. The reason is that for each flip flop (bit) in the memory, you would need a separate wire coming out of the square array. As you increase the size of the square array, the number of flip-flops in the array rises at a much faster rate than the length of the perimeter around the array (roughly n^2 versus $4n$). There simply will not be enough room for so many wires. Let's look at some alternative approaches.

We have an $N \times N$ array of flip-flops. For each flip-flop, we connect the Q output to a line that runs vertically down an entire column. The same line would be used for all N flipflops in each column. Similarly, suppose we were to connect the D input for each flipflop to a single line that vertically runs up a column. Again, the same line would be used for all N flipflops in each column. (There is no physical difference between down and up. I just use those words so you can imagine which way the data moves.)

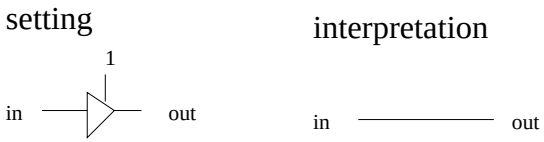
In this design, we can write to all the flipflops in a row by selecting that row (turn on only the RowSelect control for that row). Each flipflop within that row has a value determined by the write data line, and there are N such write data lines – one for each column.



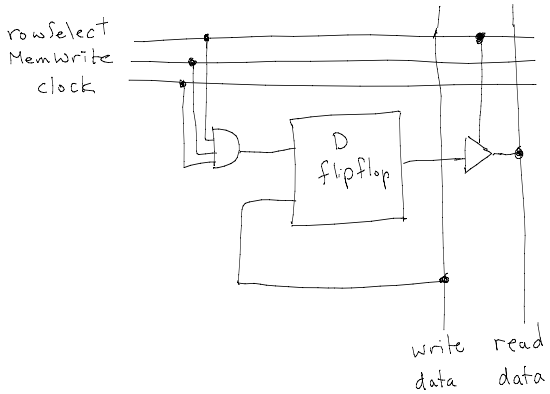
This design seems very nice (at first glance) because we need only a small number of lines for each row and for each column. This design would allow us to scale up to much larger memories. There is a problem with this design, however. For any column, all of the Q outputs of the cells in that column are attached to the read data line. This won't work. We can only allow one flipflop to put its Q value on any given line. If we tried to put a multiplexor in there, that would be going back to the previous solution which didn't work either because there are too many wires needed.

The tri-state gate (or "tristate buffer")

The problem can be solved by using another new type of gate called a *tri-state* gate (more commonly called a tri-state buffer). A tri-state gate has a data input and data output, as well as an enable input. If enable=1, then the data output is equal to the data input. If enable=0, then there is no data output. (Saying "there is no data output" is different from saying there is a value 0 output.) If enable=0, it is as if we are disconnecting the wire (temporarily). How the electronics of this works is well beyond the scope of this course, and frankly know how it works would not help you understanding the behavior just described (just like knowing how the motor of your electric toothbrush works will not help you to use it). Note that it is called tri-state because the output can be 0 or 1, or neither. For more, see `https://en.wikipedia.org/wiki/Three-state_logic`.

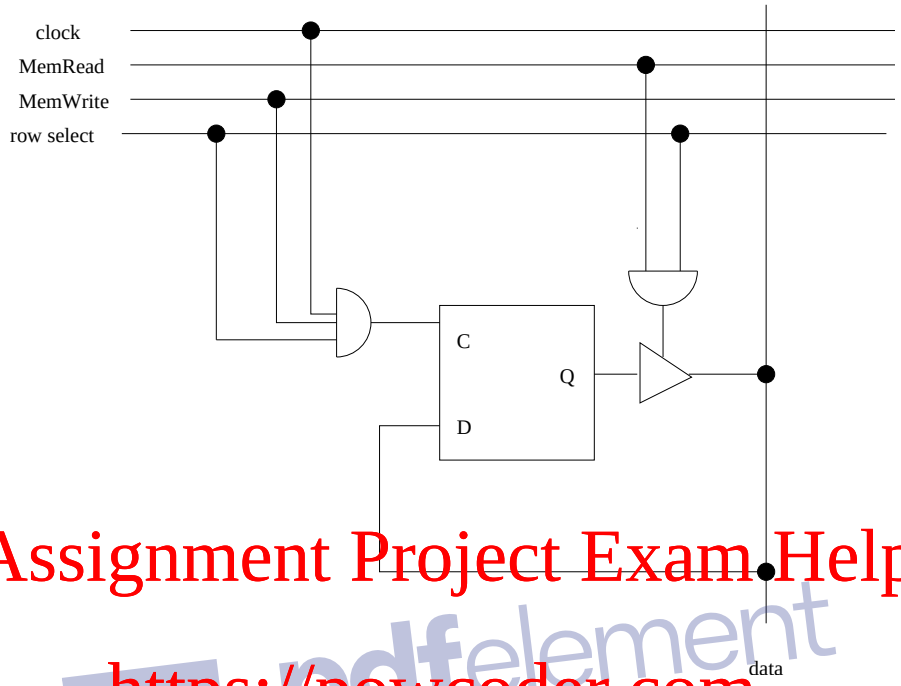


The figure below is similar to the one on the previous page, except that now a tri-state gate is used. Previously, to write to a flipflop in a single row, that row must be selected, the RowSelect control must be 1, and the MemWrite (and clock) control must be 1. Now, we also use the RowSelect control to open the tristate gate and allow the flip flop shown to put its signal on the ReadData line. Since only a single row is selected, the tristate gates ensure that only one cell in each column is putting a value on the ReadData line.



The figure below is a slightly different design in which the read and write data lines in each column are combined into a single line. Now we can either read into a flip-flop or write from a flip-flop *but not both*. Both the MemRead control and the RowSelect control are used to control

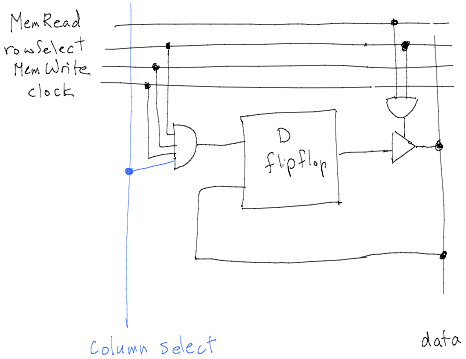
the tristate buffer. As before, the MemWrite control determines (along with clock and RowSelect) whether the value on the data line is written into the flip-flop.



RAM

In the above discussion, we considered an $N \times N$ array of D flip-flops and we were reading or writing all N flip-flops in some row. This was motivated by the register application, where we group a set of bits into a unit and operate in the same way on all bits in that unit. We next consider a different situation in which we allow ourselves either to read from or write to a single flip-flop in the $N \times N$ array.

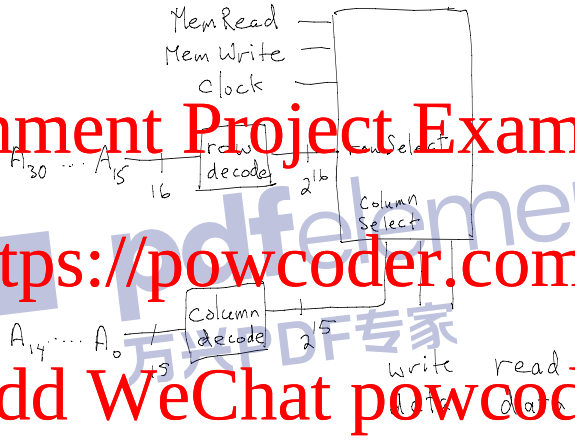
To select a column, we need another control line which we call ColumnSelect. This control is fed into the same clock input for the flip-flop along with the other controls which must be 1 in order for a write to occur. Only one of the N columns will have its control signal with value 1. The rest will have value 0.



In the lecture, I discussed an example of the sort of RAM you could buy and put into your computer. The example showed 8 chips of RAM which had 2 GB. Since each byte is 8 bits, it follows that there are 2^{31} bits on each chip. Let's think about this as a $2^{15} \times 2^{16}$ array of one bit memories.

[ASIDE: there are many different electronic technologies used for RAM. We won't go into the details. Its fine if you just assume they are the flip-flops we have been discussing – but be aware that there are many ways of getting transister-based circuits to store one bit memories.]

The figure below shows where the RowSelect and ColumnSelect controls come from. Keep in mind that these signals are 2^{16} and 2^{15} lines. Exactly one of the RowSelect and one of the ColumnSelect lines has the value 1 at any time. In the figure below, the one out of 2^{31} bits we are indexing (or addressing) is coded as a binary number $A_{30}A_{29}\cdots A_1A_0$. The 16 bits $A_{30}A_{29}\cdots A_{15}$ code the row number and the 15 bits $A_{14}\cdots A_0$ code the column number.



We will return to these RAM designs later in the course.