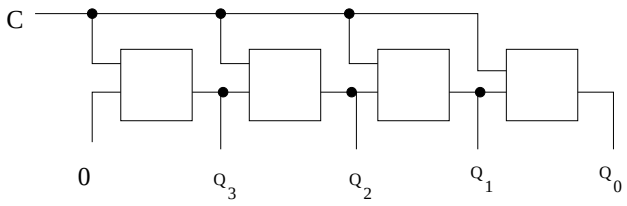
Shift registers

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 a 0. This would be used for dividing a positive number by 2. Of course, you can design the circuit to fill it with whatever you want (1, D, or even wraparound from Q_0, etc). *In the lecture slides, I gave a slight variation on this idea.*



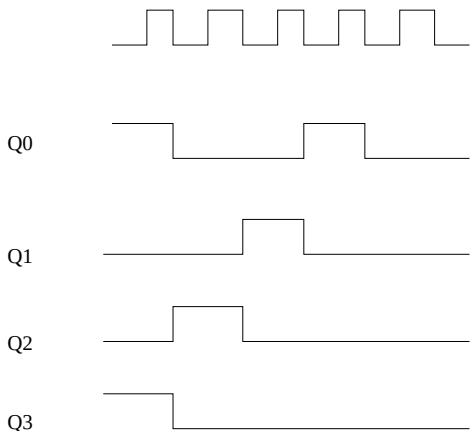
Note that for "right" and "left" to be meaningful, we need to use the common convention that the least significant bit (that is, Q_0) is on the right.

Examples

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

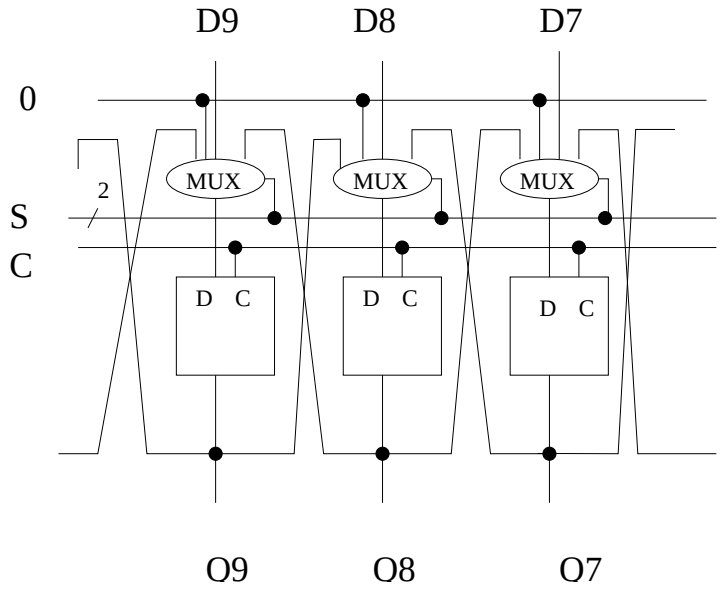
$$(Q_3, Q_2, Q_1, Q_0) \;=\; (1, 0, 0, 1)$$

and assume the flip-flops in the register are falling-edge triggered. [ASIDE: the 1 values in fact could never enter this register. You would need the more general register shown in the slides (or the register below) to get 1's in there.]



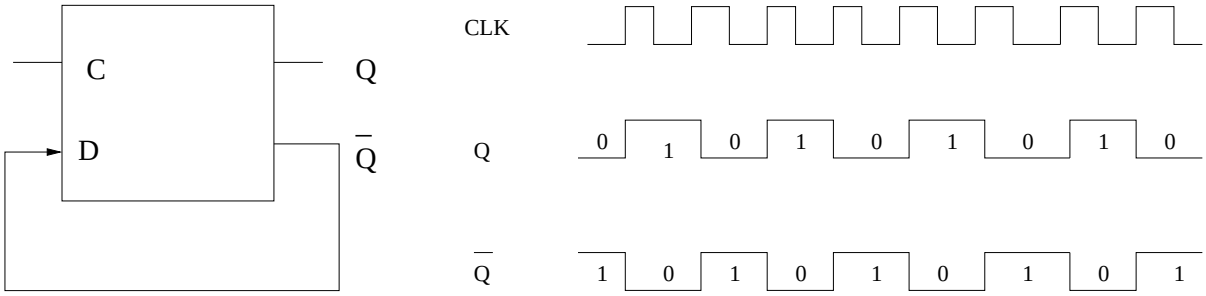
Let's look at a more general shift register that can either shift left or right, and that can also be cleared (all bits set to zero) and that can also be written to as a unit (write to all bits in one

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.



T flip-flop (toggle)

The circuit below shows a modified rising edge triggered D flip-flop, such that the data input D comes from the complement of the stored value Q. The timing diagram over several clock pulses is also shown. We see that the Q value switches from 0 to 1 or 1 to 0 one time over a complete clock cycle. Such a flip-flop is called a T flip-flop where T is for *toggle*.

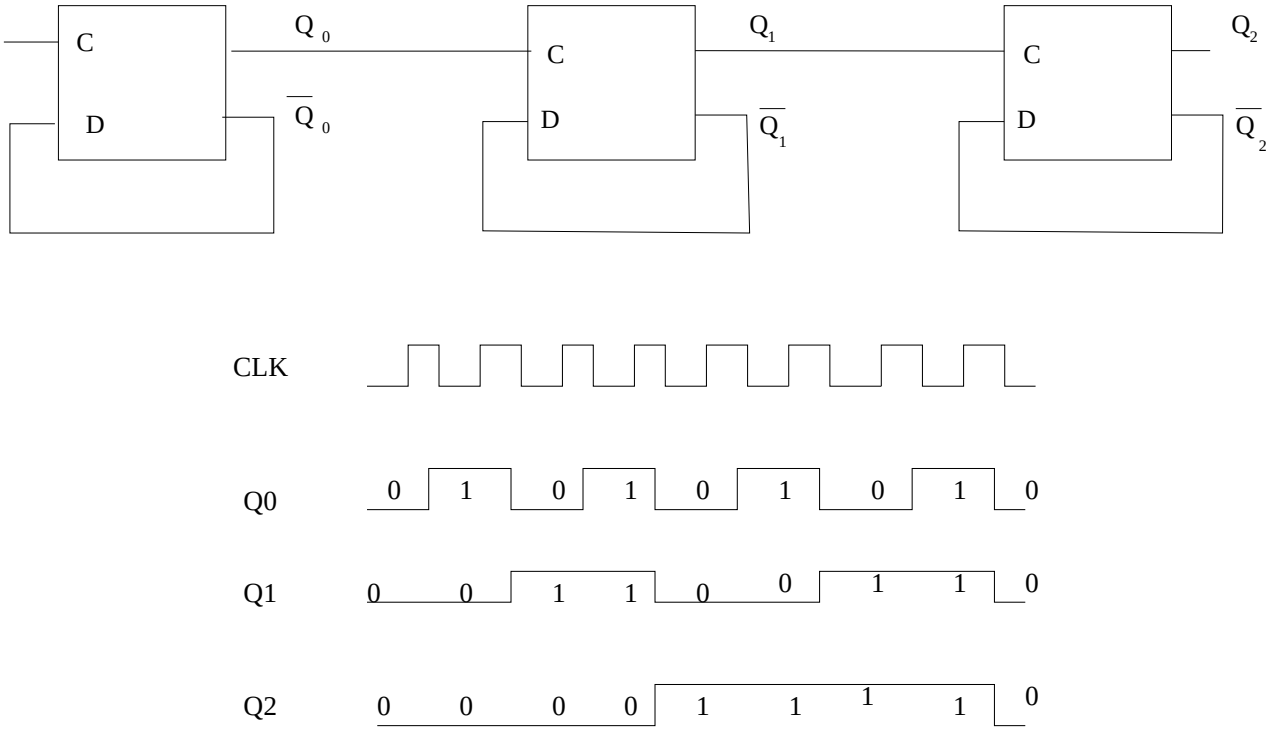


The T flip flop changes its value at half the rate of its clock input. Whether it changes value at the rise or fall of the clock input depends on whether it is a rising edge or falling edge triggered flip-flop.

Counters and timers

One important way in which T flip flops are used is as counters. Consider the sequence of numbers 000,001,010,011,100,101,110,111 which is just the numbers from 0 to 7. How can we implement a set of flip flops automatically counts clock cycles (without using an adder circuit) ?

Suppose we use a register such that the values in the flip flops are are labelled $Q_{n-1} \ldots Q_2 Q_1 Q_0$, where these values are the bits of a counter. In the figure below, the least significant bit is on the left, to make the diagram more easy to read. Suppose *falling edge* triggered flip-flops. Note that Q_0 (on left) toggles its value once per clock cycle. Also observe that Q_{i+1} toggles its value whenever bit Q_i changes from 1 to 0 (falling edge).



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. We call this a *timer* since it decreases in value (similar to the timer on your stove). *See the slides for the timing diagram.*

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 in two registers, feed the two values into an adder (ALU), and then write the result into a register. *All of this is done in one clock cycle.*

In this course, we 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.

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. We select a register by inputting the 5 bit code of the desired register into a huge multiplexor. Rather than drawing all 32 bits for each Q output leaving each register, we just draw one line from each register and say that the line (wire) 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.

If we are adding two numbers, 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. The two multiplexors select the two 32 bit values, and feed them into a 32 bit ALU which contains the adder circuit. Each of the two multiplexors needs to be told which register to select, and for this each needs 5 bits (since $2^5 = 32$). Thus a 5-bit coded number must be fed into each of two multiplexors. In the figure below, these are called ReadReg1 and ReadReg2.

Writing into a register

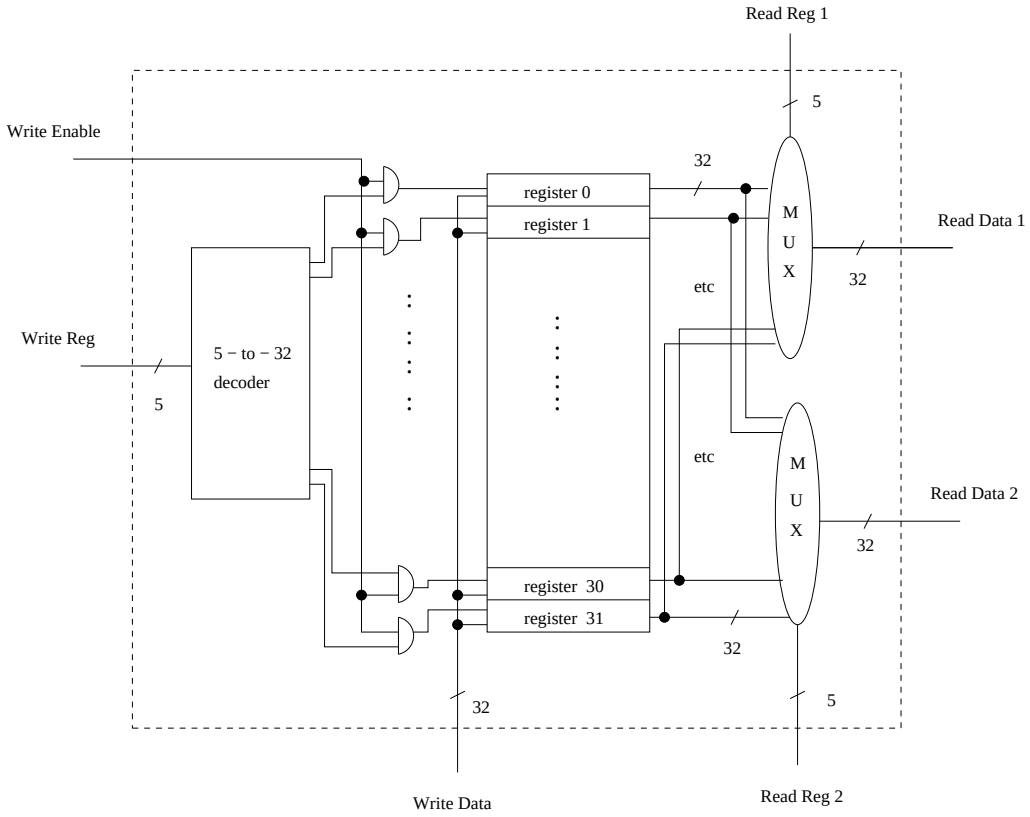
When we add two numbers, we then want to store the sum. We store the sum in a register. For example, if we were computing `x = x+y`, we would store the result in the same register as one of the operands. Or, if we were computing `y := x + z`, then we would write the sum into a different register than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

To write into a register, we take a 32 data wires (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. In MIPS, we have 32 registers available and we only want to write into one of them. How do we specify which? The idea is to use the clock input to each flipflop. Instead of just inputing the raw clock signal, we AND each of the true clock signals with the output of a decoder (see figure on next page) such that the decoder specifies which register we want to write to. We thus need to feed the decoder a 5 bit code for the register number that we want to write to. According to the scheme just described, we write to a register at each clock pulse. The register that is written to is the one specified by the 5 bit number which I am calling WriteReg which stands for "write register" (see figure).

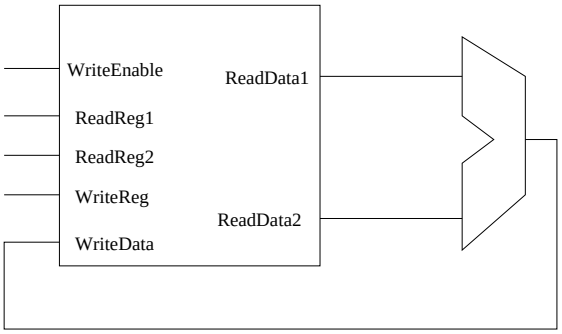
As we will see later, there may be many situations in which we do not want to write to a register. For this reason, we have given the clock input a more general name here: WriteEnable. WriteEnable would be the output of some other circuit which depends on the computer's clock, but also on other variables.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

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

where the Y_i are other variables. (This will make more sense in a few weeks when we discuss data paths.)



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



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)

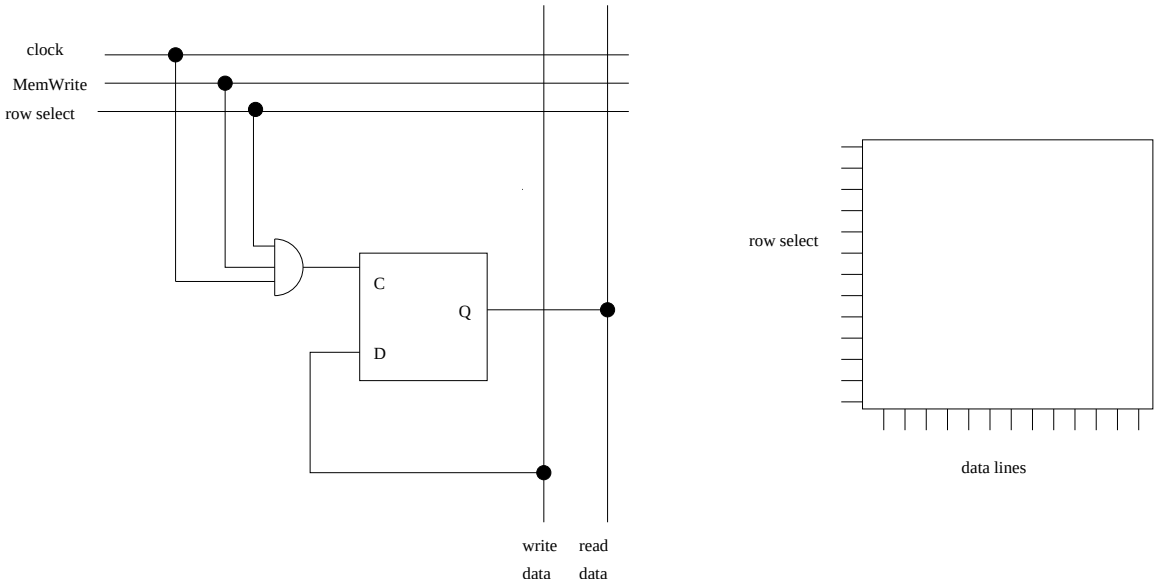
Notice that the register array uses multiplexors which choose between 32 different 32 bit numbers. Think of this as 32 different one bit multiplexors, each of which are choosing between 32 bits. We only need one decoder, but we still need 32^2 AND gates.

The above design is feasible for small memories such as an array of registers. But it won't work for larger memories. The reason is that, for each bit in the memory, you would need a separate wire coming out of the square array. As you increase the size (or density) 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 $4n$). Let's look at an alternative way to solve the problem in this case.

Suppose 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 a column. The same line would be used for all 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 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.)

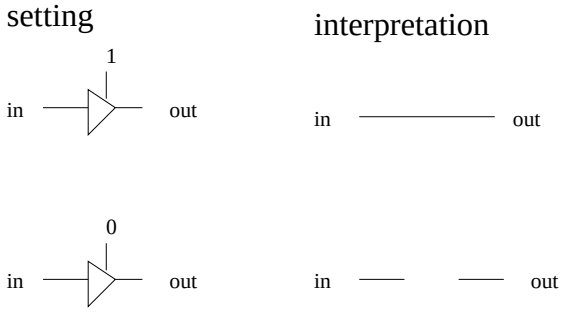
In this design, we can write to all the flipflops in a row by selecting that row (turn on only the row select control for that row). Each flipflop within that row has a value determined by the write data line, and there are N such write data lines – one for each column.

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 value on any given line. We could put a multiplexor in there, but note this is basically going back to the previous solution above.

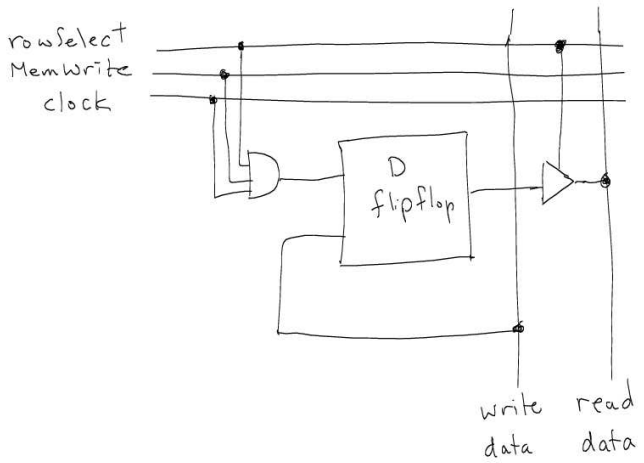


The tri-state buffer

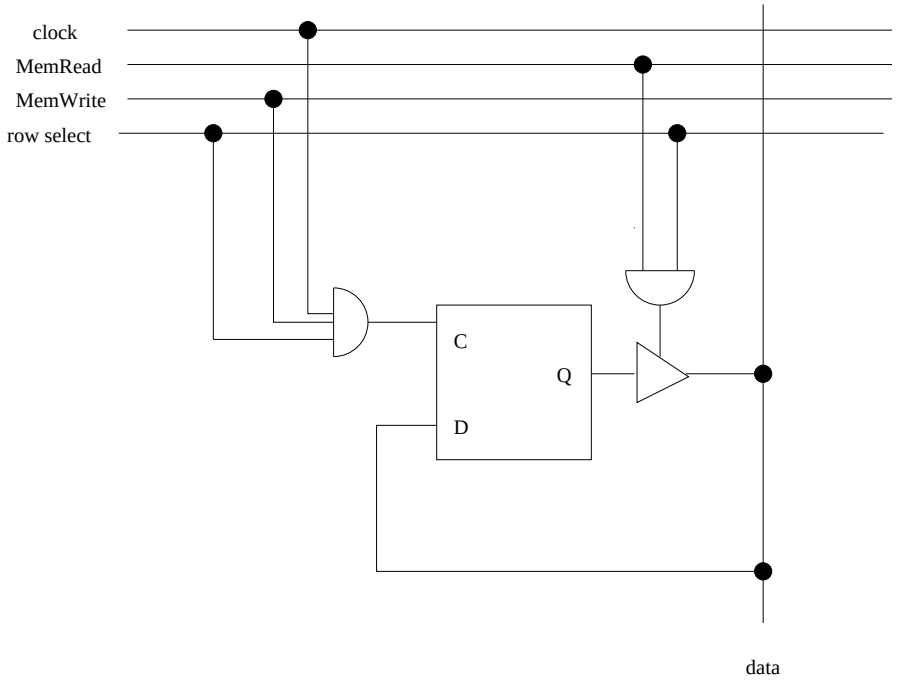
Engineers long ago solved this sort of problem by creating a new gate called a *tri-state* gate (more commonly called a tri-state buffer. A tri-state buffer has a data input and data output, as well as an enable input. If enable=1, then the data output is equal to the data input. If enable=0, then there is no data output. Saying "there is no data output" is different from saying there is a 0 output. If enable=0, it is as if we are disconnecting the wire (temporarily). [ASIDE: how the electronics of this works is well beyond the scope of this course.]



To read or write to a flipflop in a single row, that row must be selected. The rowselect control must be 1. Moreover, the Memread control must be 1. As long as only a single row is selected, the tristate buffers ensure that only one cell in each column is putting a value on the read data line. That is, if MemRead = 1 and rowselect = 1, then the tri-state buffer for that cell is enabled and the bit stored in that cell is put onto the readdata line. Since all cells within a column are connected to the same readdata line, only one row is selected. For all other rows, the rowselect control is 0, and so all cells in other rows are disconnected from the readdata column.



The figure 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.

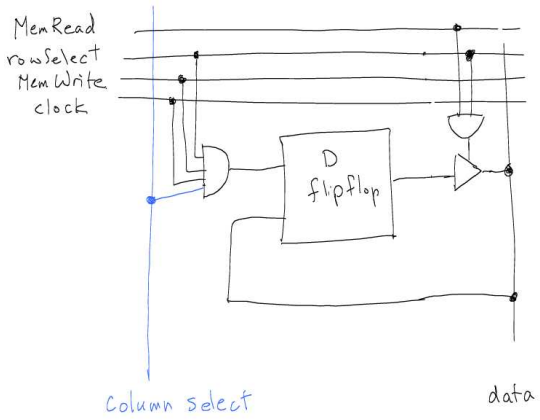


RAM

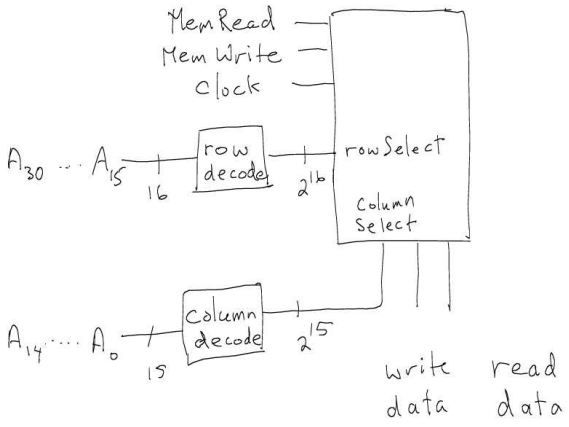
In the above discussion, we considered an $N \times N$ array of D flip-flops and we were reading or writing all N flip-flops in some row. This was motivated by the register application, where we group a set of bits into a unit and operate in the same way on all bits in that unit. We next consider a different 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 column of the N columsn 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 you might want to be aware that there are other slightly different ways of getting transister based circuits to store one bit memories.]



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



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