ECE 2372 Modern Digital Design July 21

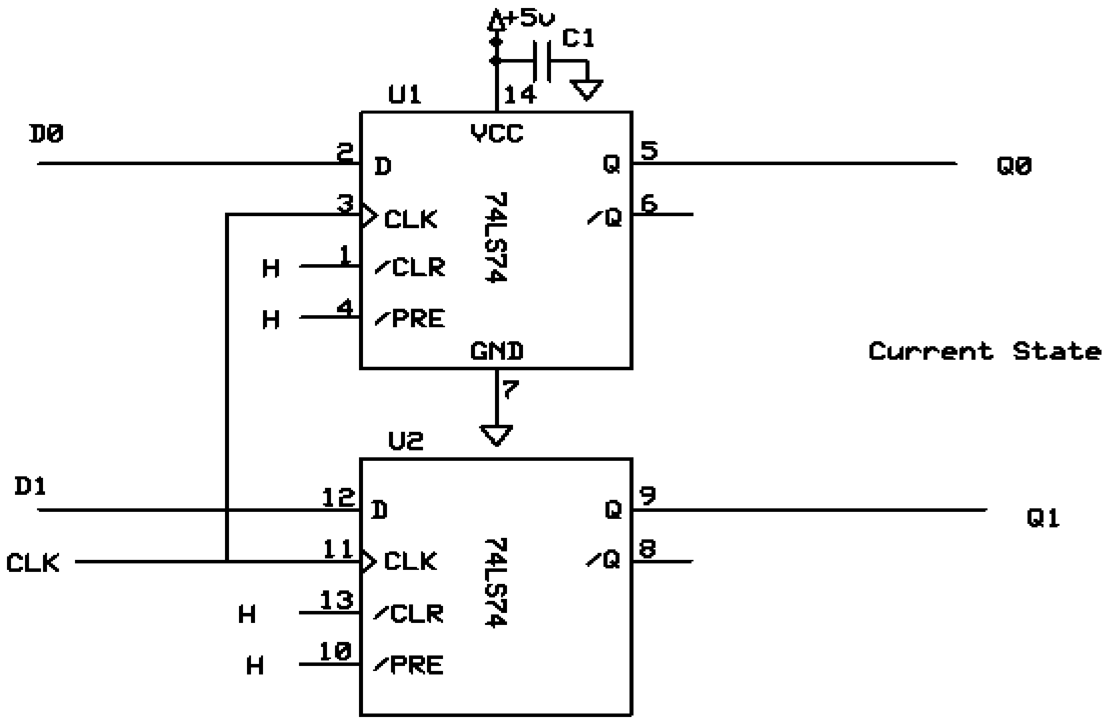
Reading assignment for today: Ch 4: 4-4, 4-5

Last time we introduced the idea of a memory element, a flip-flop, or latch that can store the state of one bit of information. We were able to form a basic S-R (Set-Reset) latch using two NOR gates connected to have feedback to "latch" a particular state. We provided Set and Reset inputs to easily change the state of the latch or flip-flop. We later saw some extensions of the basic S-R latch, to make it "clockable", that is where the data (on the D input) is latched or stored due to a clock event. The initial circuit showed this as a "level" clock, but edge-triggered flip-flops are preferred in actual practice. Several varieties of edge-triggered flip-flops are commercially available. One chip in particular that we should look at is the 74LS74. You can find the datasheet for this on the Mouser.com website. This particular component is a dual, that is there are two positive edge-triggered flip-flops in a single package. One disadvantage of the level triggered flip-flop is that the output will change if the input changes any time the "clock" "level" is in the active state. This problem is solved in the edge triggered flip-flop. The output only changes when an edge transition occurs and will be based on the input signals at the time the clock "event" (edge) occurred. This significantly improves reliability and is used in modern designs.

The datasheet for the 74LS74 is included with the other documentation today. Notice in the bottom right corner of the first page of the datasheet is a logic diagram of the internal structure of one section of the 74LS74 (it is a dual device, and has two sets of this logic inside the chip). Notice there appear to be 3 S-R latches internal to the flip-flop. Notice there is a Q and a ~Q output, a D (data) input, a clock input (CLK) and ~PRE and ~CLR inputs to preset the flip-flop or clear it to a particular initial state.

Scrolling on down to page 3, we see the detailed schematic diagram of the internal circuitry of transistors and resistors that form the flip-flop. Going back to the first page of the datasheet, look on the lefthand side, about halfway down the page and you will see a function table that describes the operation of this flip-flop. Notice in the first row of that table that we can use the ~PRE input signal to Preset the flip-flop into the state with Q = H and ~Q = L. We do this by setting ~PRE = L and ~CLR = H. Notice in the second row we can do the opposite. We can set Q = L and ~Q = H. Note in the third row that if ~PRE and ~CLR are both brought to L state, Q and ~Q both go H. This would be undefined – telling the chip to both CLR and PRESET at the same time. The most important part of the operation for our concern are rows 4 and 5. Notice that if D = H, upon a rising edge clock event (shown as an up arrow), Q is set to H and ~Q cleared to L. Notice in the 5th row, the opposite occurs, if D is L, upon a rising clock edge event, Q is cleared to L and ~Q set to H. And the last row merely shows the latched state. ~PRE and ~CLR are both H, D is in some unknown state, CLK is L, Q and ~Q will remain whatever they had been due to the last clock event and state of the D input at the time. The 74LS74 is a very handy little component. I first started working with those in industry in 1974. We don't use the 7400 series of logic gates as much now days as we did back then, but we still use them. Now days, we are much more likely to use microcontrollers rather than discrete logic gates to design and implement some digital function, but we still do use logic gates as well.

So, imagine we have two of the positive edge triggered D flip-flops just described. Such an arrangement is shown below:



The circuit above consists of two type 74LS74 positive edge triggered D flip-flops. These are wired to share a common clock. Notice that the D inputs are separate, one labeled D0 and the other one labeled D1. Also, note that the outputs Q are labeled as Q0 and Q1. Note that the inverses of Q0 and Q1 (~Q0 and ~Q1) are available as well, but are not explicitly brought out in the diagram above. This overall circuit as shown forms a two-bit memory. Each of the two D flip-flops forms a single one-bit memory cell, and the two combined forms a two-bit memory.

To load this memory with data, the appropriate data is applied to the two D inputs D0 and D1, and then the CLK signal is brought from a LOW state to a HIGH state (positive edge triggered). The data that was on the D0 and D1 inputs will be latched into the Q0 and Q1 outputs following the CLK event. Q0 and Q1 will remain in that output state until data on the D inputs is changed AND the CLK signal is once again brought from LOW to HIGH state to "clock in" the new data.

The above circuit forms a very convenient two-bit memory unit. This idea can be extended by merely adding more D flip-flops. For example, if we add two more D flip-flops to the above circuit, we will have a four-bit memory unit. With the two-bit memory unit shown above, there are 4 possible states. With the four-bit memory unit just described, we would have 16 possible states. Likewise, a three-bit memory unit such as above would have 8 possible states.

By combining an appropriate number of D flip-flops together with a common clock event for loading data into the memory, we can make a memory unit of an appropriate size to store information. For the moment, we will stick with fairly simple and small memory units.

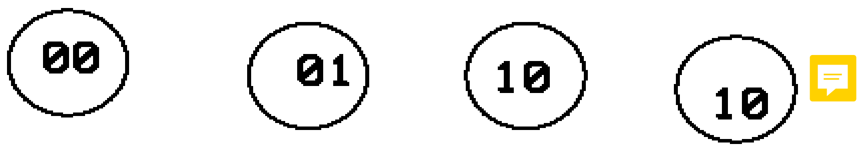
Interestingly, a lot of sequential problems can be solved with only a few states in memory. Hence, a small memory unit with only a few states can form the basis for solving some fairly complex problems.

In the diagram above of a two-bit state memory, notice that the output signals, Q0 and Q1 are labeled as "current state". The state of Q0 and Q1 represent the current state or present state of the two-bit memory. Every time we generate a rising edge clock event on the CLK, whatever data is present at D0 and D1 will be stored in the flip-flops and is available at Q0 and Q1 outputs.

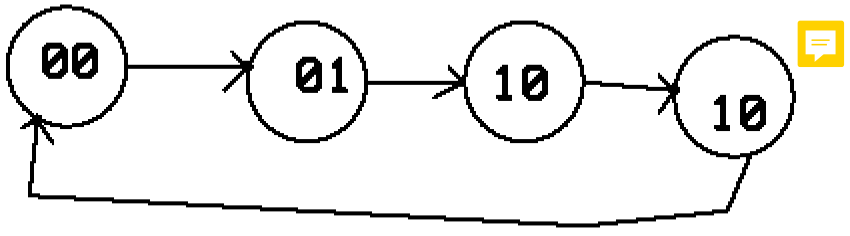
State

Q1	Q0
0	0
0	1
1	0
1	1

Those are all the possible states. We could put these into a state diagram as shown below:



These states are not shown to have any relationship between them, so it would make it more interesting if we set up a relationship among the states. How about a simple "count in sequence".



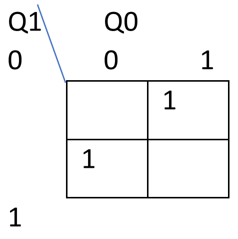
Here we can generate a Current State – Next State table to show the relationships between the states.

Current State		Next State	
Q1	Q0	D1	D0
0	0	0	1
0	1	1	0
1	0	1	1
1	1	0	0

We can now combine information from the above Current State – Next State table with the two-bit memory unit, and some combinational logic and create a machine which will in fact cycle among the states as shown, stepping to a new state with each new clock event.

We will need to solve for something called a Next State Decoder. That is simply some combinational logic circuitry which uses the Current State information and information from the Current State – Next State table or state diagram to generate D input information for the Next State. Since we have only two D inputs, we will need two K-maps, one for each D input. We only have two bits of state information, so only two input bits. We can set up K-maps to solve this. It is a simple enough case that we can observe that D1 input is simply Q1 XOR Q0. D0 is slightly more complicated.

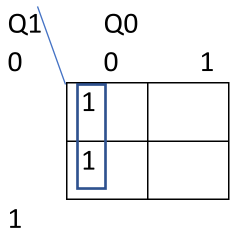
D1



D1 (next state) = F(Q1, Q0) = Q1~Q0 + ~Q1Q0

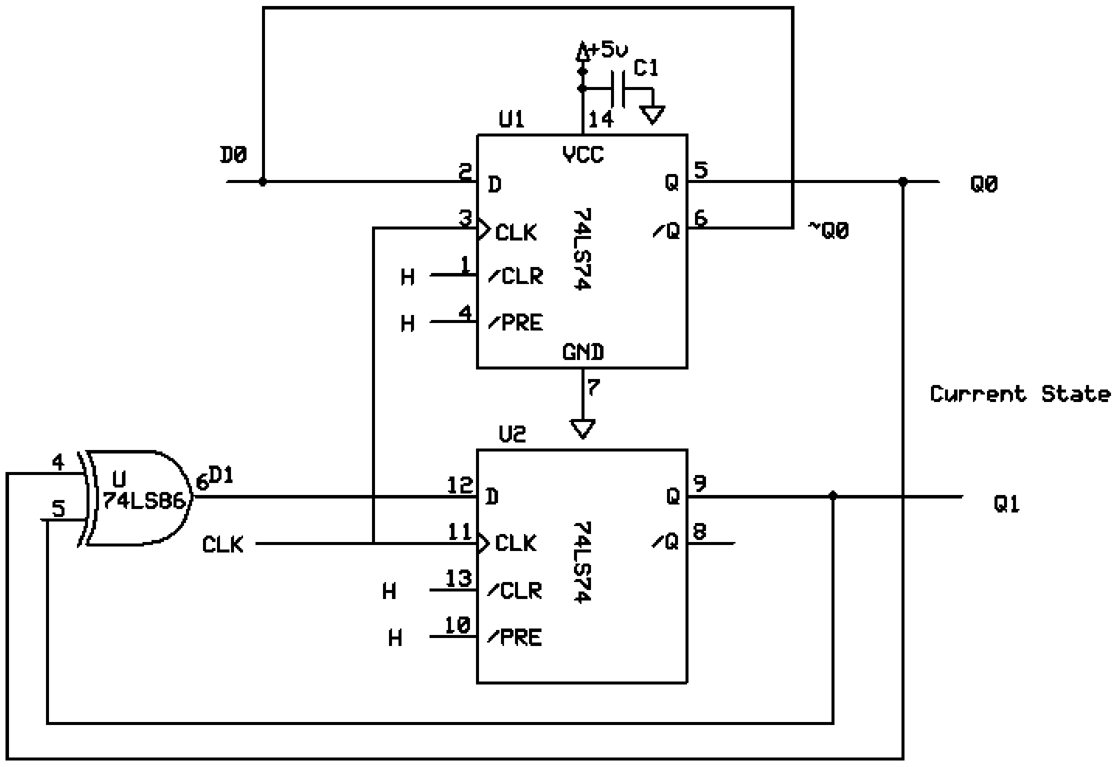
Now let's determine what is needed for D0.

D0



D0 (next state) = F(Q1, Q0) = ~Q0

Below is a complete diagram for a simple state machine (Finite State Machine) that will continuously go thru the sequence 00, 01, 10, 11, and back to 00 and repeat forever. We could also describe it as a FSM which goes thru the sequence 0,1,2,3 and repeats forever. In the world of digital circuit design, it may be more convenient to use the sequence 00, 01, 10, 11 since these are the actual bit patterns in the state memory.



At this point, you should understand enough about designing a simple FSM that you should give it a try. Refer back to the Gray Code table we encountered earlier. Design a FSM using a 3-bit memory that will repeat the 3-bit Gray Code sequence forever.