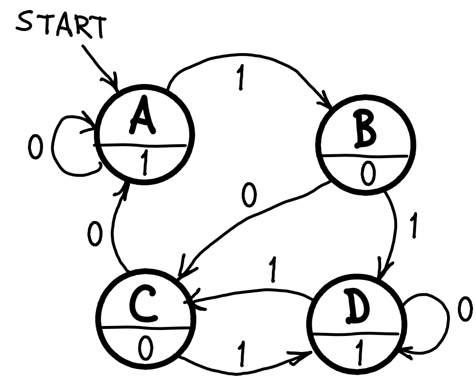
Lec-1-HW-2-FiniteStateMachines



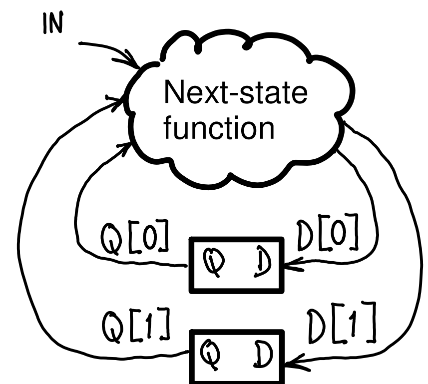
Build a finite state machine in Electric. Simulate it with verilog.

The machine is at right. It is a Moore machine: It's outputs are determined by which state it is in. E.g., if it is in state A, it outputs a 1. It has 1-bit input and 1-bit output. The output for each state is shown in the lower half of the state circle.

There are four states, so we need two 1-bit state elements to record which state we are in (two 1-bit D Flip Flops). The current state we will designate by the two bits, Q[1] Q[0] , or Q[1:0]. The next state is designated D[1:0]. The input is IN, the output is, OUT.

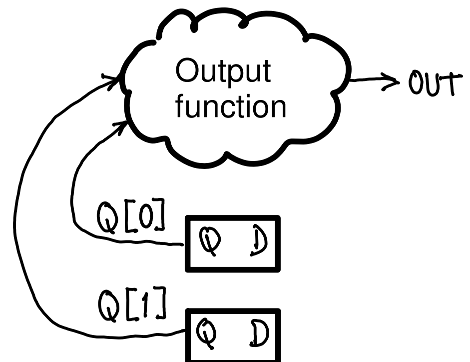
Here is the state encoding:

Q[1]	Q[0]	machine state
0	0	A
0	1	B
1	0	C
1	1	D



Q.1 Write down the next-state function for this machine. The first two rows are done for you.

Current State Q[1]	Current State Q[0]	IN	Next State D[1]	Next State D[0]
0	0	0	0	0
0	0	1	0	1

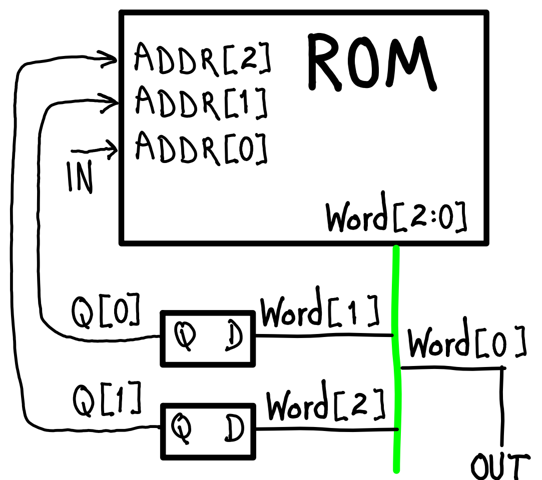


Q.2 Write down the output function. The first two rows are done for you. IN is "X" for "don't care".

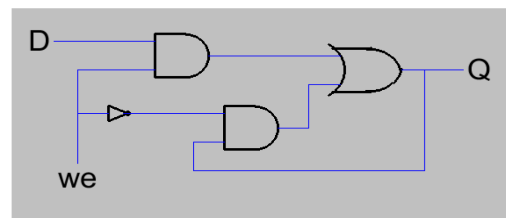
Current State Q[1]	Current State Q[0]	IN	Output OUT
0	0	X	1
0	1	X	0

Q.3 We will implement both of these functions in a Read Only Memory (ROM). The memory words will be three bits wide in this format: Word[2:0] = { D[1], D[0], OUT }. That is, the low-bit is the current state's output, OUT, and the high bits are the next state, D[1:0]. Because there are three bits of input to the next-state function, we need eight words of ROM. Write down the ROM content. Address is in this format: ADDR[2:0] = { Q[1], Q[0], IN }. The first two rows are done for you.

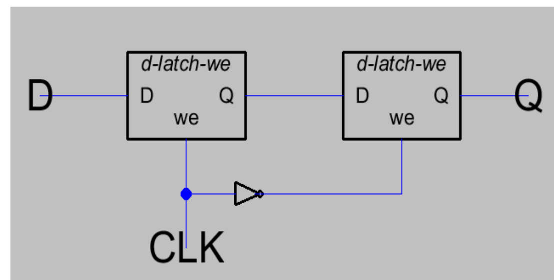
ADDR[2:0]	ROM content, WORD[2:0]		
000	001	current-state=00 IN=0	next-state=00 OUT=1
001	011	current-state=00 IN=1	next-state=01 OUT=1

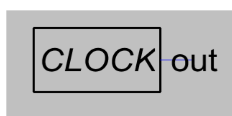


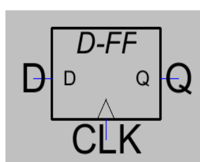
A library of parts (Lec-1-HW-2-parts.jelib) is provided for this assignment. You are to use these parts as basic building blocks. Do not use Electric's basic gates. The library includes a D latch w/ write enable (d-latch-we), shown at right. Two of these are used to implement a D Flip-Flop (D-FF) w/o write enable, also shown at right along with its icon. Our D-FF does not need a write-enable because it will always be written into on every clock tick.



A clock is also provided (CLOCK). The clock will be important because the clock period must allow for propagation delays. The basic gates implemented here (AND, OR, NOT) have signal delays. You will need to adjust the clock pulses to match the longest signal delay through your logic's feedback loop from the output of the state elements to their inputs.

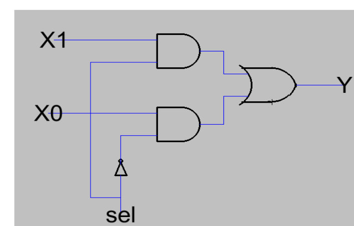


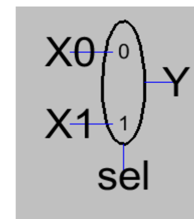


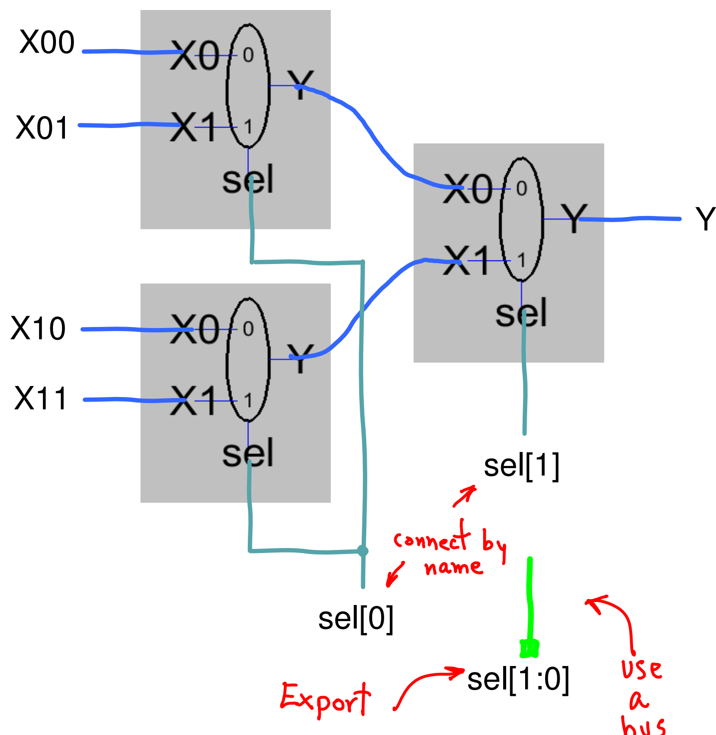


Q.4. Build a 1-bit, 8X1 MUX. A 1-bit, 2X1 MUX (MUX1_2X1) provided. Combine three to make a 1-bit, 4X1 MUX (call it MUX1_4X1). Next, use one MUX1_2X1 and two MUX1_4X1s to build a 1-bit, 8X1 MUX (MUX1_ 8X1).

If you prefer, you can also do this by building a 3-input AND gate from 2-input ANDs, and then use copies of it to produce all eight minterms. You would also need to build a tree of 2-input ORs to implement an 8-way OR.



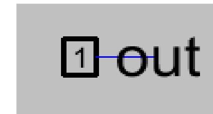




A note on icons.

When you create a cell, you then do View.MakeIconView, which creates the icon cell and its art work. These often do not look very good. Editing them requires just a couple of tricks: (1) Use Components.ArtWork instead of Components.Schematic. (2) To resize a box or relocate text, select the object, then Edit.Properties.ObjectProperties. Adjust x-y values as desired. (3) Exports can be in poor locations, but moving requires selecting one pin, move it, then select the other pin, and move it. Use arrows keys to move selected items. Also, rotate them by selecting the entire Export (both pins and wire), then use Edit.Rotate.

Q.5. Use two MUX1_8X1s to build your ROM. Tie the inputs to logic 0 or 1 as appropriate to implement the FSM next-state and output functions above. Two logic elements are provided in the library that output logic values 0 or 1 (logic_0 and logic_1). The icon for logic_1 is shown.

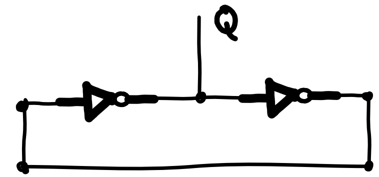


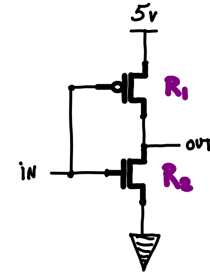
Q.7. Complete the implementation of the FSM by connecting your two D Flip-Flops to your ROM, providing exports for input and output. A test bench cell (test{sch}) is also provided. Drop an instance of your FSM into it. You may delete its current content. Provide verilog code to drive the input and display the output and any other "interesting" signals. You may need to name a wire or bus so you can refer to it in your testbench code. Provide input to drive the FSM through several interesting state-transition paths. Remember the clock cycle may need adjusting. Comment on the result.

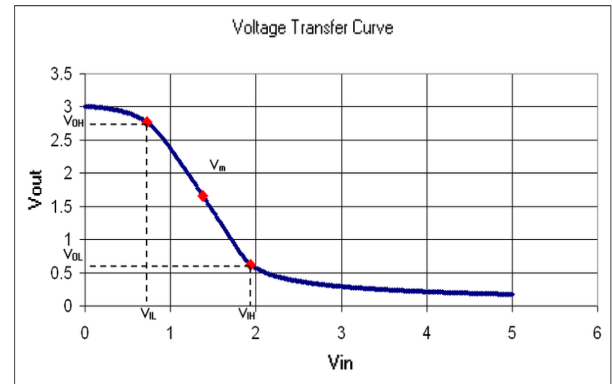
Q.8. The basic gates do not have initial values, and as a consequence it will take some time for the machine to establish its state. Usually at power on, the outputs of such physical devices settle to 0 or 1 quickly, but which one is more or less random. Consider implementing additional logic to set the two state elements to initial values (which should be 0 and 0 from the description of the FSM above, because it starts in state 00). Without going into detail, suggest a plausible proposal for doing this initialization. No verilog code can be used, just logic gates.

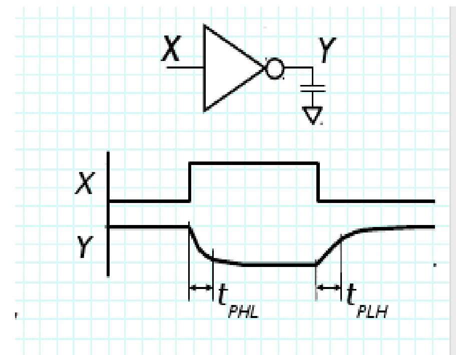
Q.9. The start-up problem (we do not know what state the Flip-Flops will settle into) has another interesting feature. We do not model the ramp-up of device outputs. That is, real outputs do not simply drop to 0 or rise to 1. Instead, they take time to get there going through intermediate values between 0 and 1. The response curve for a transistor was show in lecture. The output responds to the input with some delay, as well. In a feedback loop, this make create a "meta-stable" condition in which the rising output feeds back to the input through some logic and becomes a falling input. This causes the output to reverse and begin to fall. This can happen at a very fast rate so that the Flip-Flop never settles into one state or the other (see below). Comment on the effect of this type of situation on the FSM. What might you expect to see as output? Do you think it is possible to guarantee this will never happen?

The circuit at right is equivalent to a latch. Below it is the voltage output of a single inverter, and the fall and rise times of an inverter (T_PHL and T_PLH). Suppose one inverter powers on. Its input voltage will be something random. Suppose both output voltages are random (~ 1.5 v). Both inverters' inputs are thus also 1.5v. At 1.5v input, each inverter will start to ramp its output up (say 1.7v). But at 1.7v input, both outputs' response is about 1.3v. And at 1.3v input both start to ramp up to 1.7v again. This can oscillate indefinitely, but usually one inverter will get ahead of the other fairly soon and drive the state into one of the two stable states. This is the start up problem.









What to turn in: **Check in your work** to your branch. **Submit in class on paper**:
Name, course, year, HW assignment title
Answers to questions, comments on project files and simulations, difficulties, and suggestions.