1) In class we used some Boolean identities involving NAND to prove that sending an arbitrary value $d$ to a $D$ latch causes the latch to converge at the state $Q=d$, regardless of whether $d$ is 0 or 1. In doing so we followed these rules:

- When a value changes, draw a box around it, and in the next row, recomputed all variables whose value depends on the changed variables.

- If no further changes in a row are possible, then the values have converged, and we place a check next to the row to show that the circuit has stabilized.

- Whenever we change an input variable, we copy down the previous values of the non-input variables into the row before determining whether their values change, on the next line.

<table>
<thead>
<tr>
<th>$D$</th>
<th>$EN$</th>
<th>$S$</th>
<th>$R$</th>
<th>$Q$</th>
<th>$Q'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t=0$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$t=10$</td>
<td>[d]</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$t=20$</td>
<td>d</td>
<td>[1]</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>[d']</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>d'</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>d'</td>
</tr>
</tbody>
</table>

Continue the table above by showing that if the $D$ input changes to $d'$ at $t=30$, that $Q$ will converge to $d'$.

2) We will show using Boolean algebra that the positive edge triggered $D$ flip-flop from the handout only changes $Q$ to the value of $D$ when the clock changes from 0 to 1. Starting with the initial inputs:

<table>
<thead>
<tr>
<th>$Clk$</th>
<th>$D$</th>
<th>$X$</th>
<th>$Y$</th>
<th>$S$</th>
<th>$R$</th>
<th>$Q$</th>
<th>$Q'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t=0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

a. At $t=10$, change $D$ to the value $d$, and show that $Q$ remains the same.

b. At $t=20$, change $Clk$ to 1, and show that $Q$ takes on the value $d$.

c. At $t=30$, change $d$ to $d'$ and show that $Q$ remains unaffected, but $S$ and $R$ both change back to 1.
3) We can construct state machines like the ones in Worksheet 6 by writing down a truth table for the state transition function. The inputs to the state transition function are the state variables $S_i$, and the outputs are the “next state” variables $N_i$. From there, we can use K-maps to produce simplified Boolean expressions for each output, and then implement them using gates.

Produce diagrams for these state machines:

a. Using two $D$ flip-flops and any additional gates necessary, construct a state machine that progresses from the state 00 to 01, 10, and finally to 11. Once in the state 11, your state machine should stay there indefinitely.

b. Using three $D$ flip-flops and any additional gates necessary, construct a state machine that progresses through all of the three-bit Gray codes, starting with 000 and continuing through 100 before starting over.

4) Here is a module dff implementing a $D$ flip-flop using behavioral Verilog*:

```verilog
module dff(clk, D, Q);
    input clk, D;
    output reg Q;
    initial
        Q <= 0;
    always @(posedge clk)
        Q <= D;
endmodule
```

a. Construct a Verilog module hw6_4a(clk, S) that implements the first state machine above by instantiating two $D$ flip-flops. Your module should take a single input (the clock), and it should assign its 2-bit output with the values of the flip-flops.

b. Similarly, construct a Verilog module named hw6_4b(clk, S) that implements the second state machine above by instantiating three $D$ flip-flops.

Submit both of your Verilog modules to the Blackboard site in a zipfile. The archive should contain two files named hw6_4a.v, and hw6_4b.v.

*Note that this module “cheats” by initializing its $Q$ value to 0. In general, it’s undesirable to use the initial keyword outside of a test bench. We’ll discuss why – and how to work around this – in class after break.