University of California at Berkeley
College of Engineering
Department of Electrical Engineering and Computer Science

EECS150, Spring 2010

Discussion 3 Solution: More Verilog and some CMOS

Brandon Myers

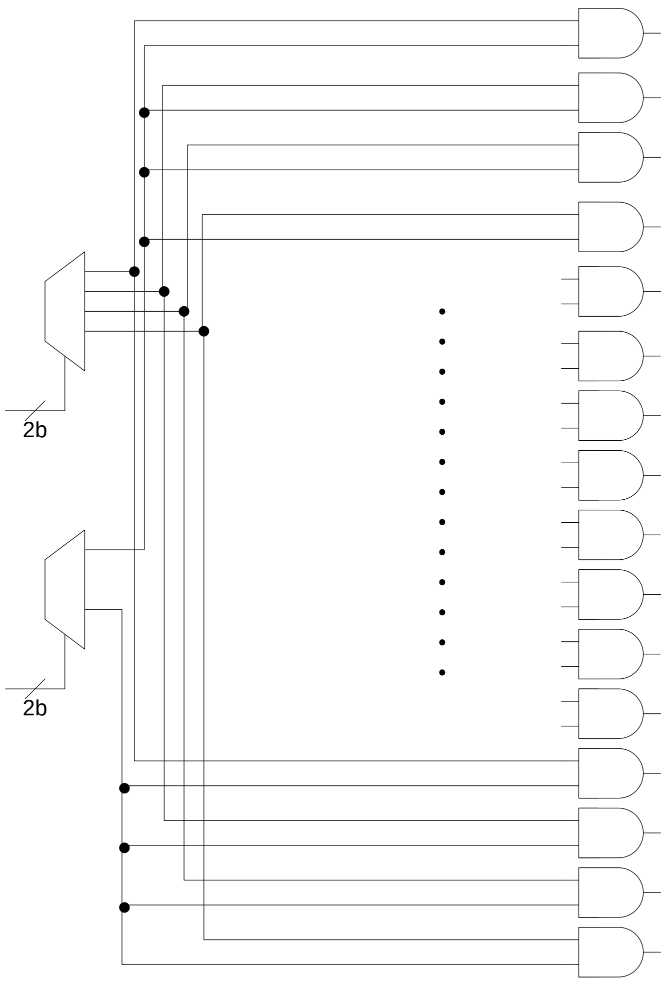
1. Write a Verilog module, Duplicate, that takes an input signal In and passes it thru to every bit of the output signal Out. Out is N bits wide; N is a parameter.

```
// using replication operator
module Duplicate(In, Out);
    parameter N = 8;
    input In;
    output [N-1:0] Out;

    assign Out = {N{In}};
endmodule

// using generate
module Duplicate(In, Out);
    parameter N = 8;
    input In;
    output [N-1:0] Out;

    genvar i;
    generate
    for (i=0; i<N; i=i+1) begin
        assign Out[i] = In;
    end
    endgenerate
endmodule
```

2. Write a Verilog module for a level sensitive latch, with interface In, Out, and C (“clock”). It must have a parameter Level, which if 1 makes the latch sensitive to high and if 0 makes the latch sensitive to low.

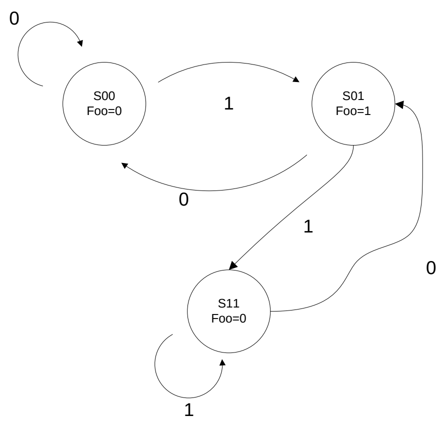
```
// using replication operator
module Latch(In, Out, C);
    parameter Level = 1;

    input In, C;
    output Out;

    always @ * begin
        if (C==Level) Out = In;
    end
endmodule
```

3. Design a 4:16 decoder out of 2:4 decoders and 2-input gates. Draw the circuit.



The 2:4 decoder asserts the output specified by the input, so it is a binary to one-hot translation. With two decoders, we have the 4 required inputs. Each decoder will be asserting exactly one of its 4 outputs; this is 4 output values per decoder. 4 values times 4 values gives 16 possible outputs. By ANDing every combination of outputs from the decoders, we get the 16 outputs (16-bit one-hot).

4. Consider the FSM state-transition diagram below.



Write a Verilog module to implement this FSM.

```
module Zombie(In, Foo, Clock);
    input In, Clock;
    output Foo;

    // state encodings
    localparam S00 = 2'b00,
               S01 = 2'b01,
               S11 = 2'b11;

    reg [1:0] State, Next_State;

    // state transition
    always @ (posedge Clock) begin
        State <= Next_State;
    end

    // next state logic
    always @ * begin
        case (State)
            S00 : begin
                if (In) Next_State = S01;
                else Next_State = S00;
            end
            S01 : begin
                if (In) Next_State = S11;
                else Next_State = S00;
            end
            S11 : begin
                if (In) Next_State = S11;
                else Next_State = S01;
            end
            default : Next_State = 2'bxx;
        endcase
    end

    // output logic
    assign Foo = (State == S01);
endmodule
```

5. A gray code counter is like a binary counter except with a different encoding, where every adjacent output changes just one bit. For example a sequence for a 2-bit gray code counter is
   00
   01
   11
   10
   Come up with a 3-bit gray code sequence and write a Verilog module, GrayCodeCounter, which is a gray code counter with this sequence.
   Any number could be adjacent to N other numbers in a gray code sequence. Using this observation, a systematic way to come up with a N-bit gray code is to draw an N-dimensional hypercube, label each corner in binary (in a certain way where each increase in dimension adds a digit, look it up if you are interested), and then just find a cycle visiting each node in the graph. Below is one 3-bit gray code sequence: 000
   001
   011
   111
   101
   100
   110
   010

   There are two basic ways to approach building this counter. One is to assign each number a state, and use our FSM methodology to write a state machine where the Next_State is always the next number in the sequence. The other way is a decoded binary counter. Specifically, a 3-bit binary counter is easy to build from a register and adder. The output of the counter goes through a decoder that turns the binary number into the corresponding number in the gray code seqeunce.

6. State machines and FPGAs.

   (a) What is the minimum number of bits of state required to implement an FSM with 6 states?
   $ceil(log_2 6) = 3$

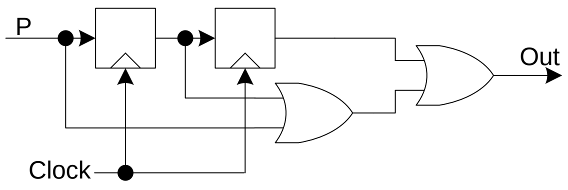
   (b) I wrote my FSM module in Verilog so that each of the 6 states had a unique binary encoding. But when I ran synthesis on the module, the RTL schematic shows 6 flipflops.

      i. What's going on?
      The synthesis tool chose a one-hot encoding of the states. Thus, each state is represented by one of the flipflops, rather than the combined state of the 3 flipflops in the binary encoding.

      ii. Why would the synthesis tool do this?
      The amount of state required for binary-encoded FSM grows as $log_2 N$, where N is the number of states. For one-hot encoded it grows as N. So, the one-hot version requires more resources. However, the tools would do this because now the next state logic consists of just some logic for each state saying whether the next state is or is not that state. This can map simply to a SLICE, where each next state logic and state is one LUT/flipflop pair. For small designs where available FPGA resources are not an issue, the tools may choose the one-hot encoded FSM. Of course, because of the log_2 versus

linear scaling of state element usage, for large FSMs, it would be best to choose the binary encoding.

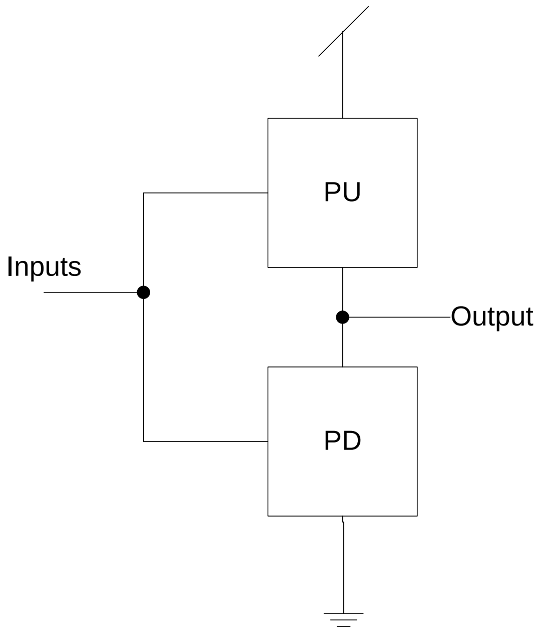
7. Write a Verilog module for the following circuit:



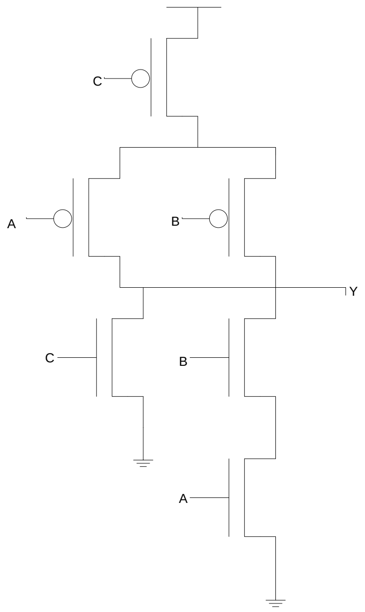
```
module PulseWidener(P, Out, Clock);
    input P, Clock;
    output Out;

    reg f1, f2;
    always @ (posedge Clock) begin
        f1 <= P;
        f2 <= f1;
    end

    assign Out = P | f1 | f2;
endmodule
```

This is one of the simplest ways to describe the circuit in Verilog. You could also describe it as an FSM, but this is sometimes, as in this case, unnecessary work.
Note: this circuit is an example of a Mealy state machine, whose output depends on both the current state and the current input.

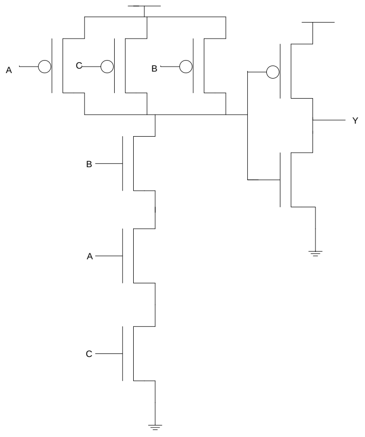
8. What is CMOS?
Complimentary Metal Oxide Semiconductor. The typical way to do CMOS logic is to have dual networks for pullup (using PMOS) and pulldown (using NMOS).



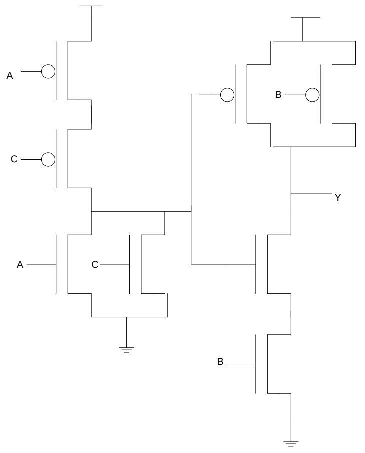
9. Give the boolean equation implemented by each of the CMOS circuits (input A,B,C; output Y).



Y= (AB+C)’



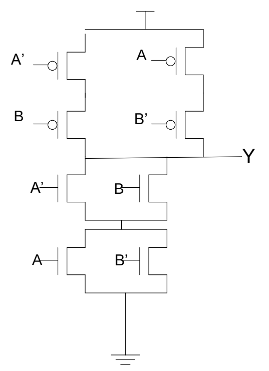
Y = ABC

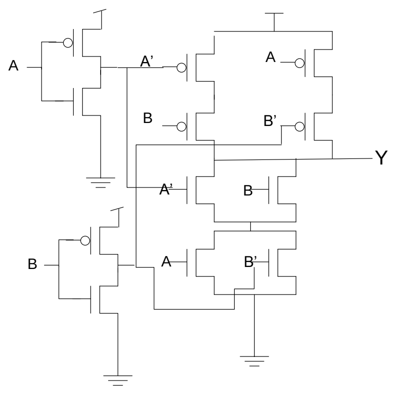


(B(A+C)')'
B' + A + C

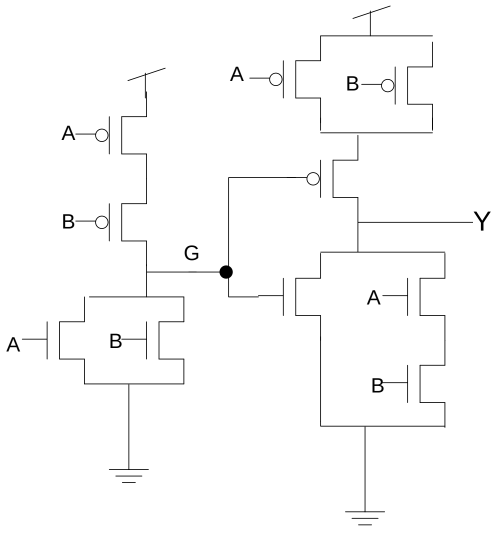
10. Design a 2-input XOR in CMOS. The equation for XOR is Y=AB'+A'B. Since this has both inverted and non-inverted inputs, there are at least two possible approaches for designing this in CMOS logic:

    (a) Treat each A,B,A',B' as separate inputs, and place inverters before the inverted inputs.





(b) Cascade cmos logic gates. We will manipulate the equation so part of it is like an input to another part.
AB'+A'B
((AB')'(A'B)')'
((A'+B)(A+B'))'
(A'A+A'B'+AB+BB')'
(A'B'+AB)' this is XOR == XNOR
let G = A'B' Since A'B' has all inverted inputs, we can implement it directly as a PMOS network.
(G + AB)' Since this is an equation all inverted, we can implement the inside as a NMOS network.



Here's two ways to go from boolean equation to CMOS implementation. Both involve building one of the networks then building the dual network. Recall that in the dual, all series become parallel and all parallel become series.

(a) Build the PMOS first. You can think of all the inputs as inverted versions going into NMOS to turn on the transistor. So, maninpulate the equation such that all inputs are inverted (i.e. A',B',...). Then implement this equation as PMOS network by doing series for AND and parallel for OR. Example: (A+BC)'
(A+BC)'
A'(BC)'
A'(B'+C')
So implement A series (B parallel C) using PMOS. Then the dual, A parallel (B series C), in NMOS.

(b) Build the NMOS first. A closed path through the NMOS network pulls output to 0. Therefore, when the equation produces a 1, we want it to produce a 0, so manipulate the equation so that all inputs are non-inverted (i.e. A,B,...) with the entire equation inverted. Example: A'+B'C'
A'+B'C'
A'+(B+C)'
(A(B+C))'
So implement A series (B parallel C) using NMOS. Then the dual, A parallel (B series C), in NMOS.