Review of combinational logic

Combinational logic is memory-less. The output of a combinational logic circuit is determined purely by its current inputs. This is in contrast to sequential logic, in which the output depends on both current inputs and current state (Mealy Machine) or only the current state (Moore Machine), and in both Machines where the current state depends on prior inputs. In this document we will generally use the asterisk \( * \) as AND, the plus \( + \) as OR, a single tick \( ' \) as negation, and a circled-plus \( \oplus \) as XOR. It is generally safe to assume that both a signal (e.g. \( A \)) and its complement (\( A' \)) are available as inputs to a circuit.

Q1. Consider the logic function \( F = (A+B')*(A+C) \)
   a. Generate the truth table for this function.
   b. Implement \( F \) using three 2-input NOR gates (do not simplify the expression).

Q2. Consider a 2-to-1 MUX as drawn on the right.
   a. Generate the truth table for this circuit.
   b. Implement \( Z = f(A,B,Se) \) using three 2-input NAND gates.

Q3. Consider a full-adder as drawn on the right.
   a. Generate the truth table for this circuit.
   b. Using standard 2-input gates (AND, OR, XOR) and the NOT gate, draw a circuit which implements the same function (has the same truth table).

Answer the following questions about implementing combinational logic in Verilog. You will need to refer to the combinational logic tutorial (linked online from the course homepage).

Q4. Provide a single-line assign statement which implements the following
   a. \( Y=(A*B')+((B*C'))+(A*C) \)
   b. A 4-to-1 MUX. Use \( A, B, C, D, S0, \) and \( S1 \) as the inputs and \( Z \) as the output.
Q5. Using the adder in the tutorial as a template, implement
   a. A module implementing a 4-to-1 MUX called MUX41 which takes six bits of input
      (A,B,C,D,S0,S1) and generates one output (Z).
   b. A module implementing an 8-to-1 MUX called MUX81 which takes 11 bits of input
      (A[7:0], S[2:0]) and generates one output (Z). It should do all its work by instantiating
      the 4-to-1 MUX of part (a) as needed.

   In addition to assign statements, combinational logic can be implemented in an “always @*” block.

Q6. Redo problem Q4.a using an always @* block rather than an assign statement.

Review of Sequential logic

Sequential logic is built around storage elements. There are a number of
different types of storage elements (D, T, and JK flip-flops, for example), but digital
designers tend to use only one: the D flip-flop or DFF. A DFF has two inputs—“D” and
“clock”—and one output—“Q”. The value of D is copied to Q on a rising edge of the
clock (when clock transitions from 0 to 1). Furthermore, the value of D cannot change
too soon before or too soon after the rising edge of the clock. Finally it is common to have the value “Q”
and its inverse (Q’) as outputs. A DFF is typically drawn as shown on the right (the triangle is the clock
input). Nearly every storage unit you will encounter is built out of D flip-flops.

Q7. Neatly complete the following timing diagram for a D flip-flop. Indicate that the value is
    unknown before the first rising edge in any way you see fit.

   ![Timing Diagram](http://cpuville.com/register.htm)

   It is commonly the case that we wish to store larger values than just a
   single bit. In such cases, we generally use flip-flops in parallel and call them
   registers. The following diagram shows a typical 4-bit register (though it
   uses a slightly different symbol for a D flip-flop; CP means clock-positive).

   ![4-bit Register](http://cpuville.com/register.htm)

   A counter is a device that that counts how many times a certain event

Q8. Using D flip-flops, and AND, OR and NOT gates, as building
    blocks, draw a 2-bit modulo counter. That is, on each rising edge
    the counter’s output value is incremented by 1, but where 3
    wraps around to 0, so it is a modulo-4 counter (the counting
    sequence is: 0→1→2→3→repeat). You do not need to worry
    about reset.

   ![2-bit Modulo Counter](http://cpuville.com/register.htm)
Refer to the Verilog sequential logic overview linked from the course homepage for these questions.

**Q9.** Draw the state transition diagram which is implemented by the code on the next page. Your diagram should resemble figure 2a from the tutorial. You need not include reset.

**Q10.** Define the term “setup time” and the term “hold time” with respect to a D flip-flop. Your answer should include a (simple) timing diagram which illustrates your point.
module fsm(clk, in, reset, out);

input clk, in, reset;
output [3:0] out;

reg [3:0] out;
reg [1:0] state;
reg [1:0] next_state;

parameter zero=2'd0, one=2'd1, two=2'd2, three=2'd3;

always @*
    begin
        case (state)
            zero:
                begin
                    out = 4'b0000;
                    next_state = one;
                end
            one:
                begin
                    out = 4'b0001;
                    if(in)
                        next_state = two;
                    else
                        next_state = one;
                end
            two:
                begin
                    out = 4'b0010;
                    if(in)
                        next_state = two;
                    else
                        next_state = three;
                end
            three:
                begin
                    out = 4'b0100;
                    if(in)
                        next_state = zero;
                    else
                        next_state = two;
                end
            default:
                begin
                    out = 4'b0000;
                    next_state = zero;
                end
        endcase
    end

always @(posedge clk)
    begin
        if (reset)
            state <= zero;
        else
            state <= next_state;
    end
endmodule