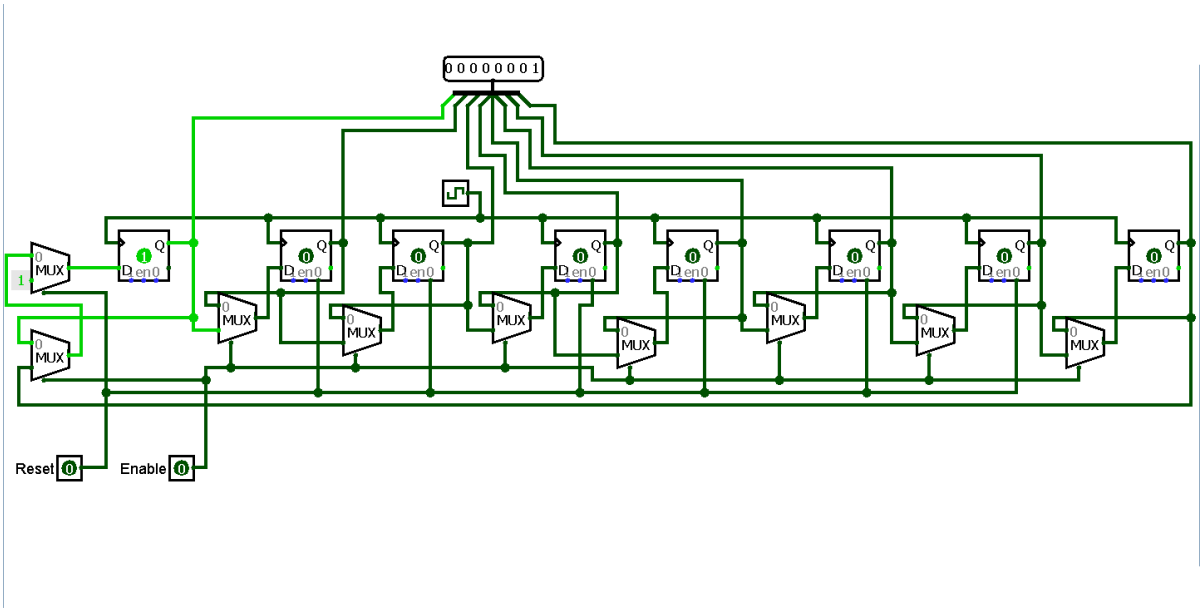
University of California at Berkeley
College of Engineering
Department of Electrical Engineering and Computer Science

EECS150, Spring 2010

Homework 3 Solutions: Verilog and Sequential Logic

Keep in mind that problems requiring writing a Verilog module may have many solutions. You should compare your answers with those below and perhaps even run the Synthesis tool on them to see what schematics they produce. Two correct behavioral modules describing the same functionality may or may not produce the same circuit.

1. One-hot counter



The above approach is to chain 8 flipflops together, reset the circuit so only the first flipflop holds a 1, and each cycle, pass the values along the chain. The following Verilog implements this design.

```
module OneHotCounter (Reset, Enable, Clock, Out);
        input wire Reset, Enable, Clock;
        output reg [7:0] Out;

        always @ (posedge Clock) begin
```

```
                if (Reset) Out <= 8'b00000001;
                else if (Enable) begin // pass reg i-1 value to reg i
                        Out[0] <= Out[7];
                        Out[1] <= Out[0];
                        Out[2] <= Out[1];
                        Out[3] <= Out[2];
                        Out[4] <= Out[3];
                        Out[5] <= Out[4];
                        Out[6] <= Out[5];
                        Out[7] <= Out[6];
                end
        end
endmodule
```

Alternatively, we could create a binary counter, then decode its output into one-hot.

```
module OneHotCounter (Reset, Enable, Clock, Out);
        input wire Reset, Enable, Clock;
        output reg [7:0] Out;

        // binary counter
        reg [2:0] count;
        always @ (posedge Clock) begin
                if (Reset) count <= 3'd0;
                else if (Enable) count <= count + 1;
        end

        // decode
        always @ * begin
                case(count)
                        2'd0 : Out = 8'h01;
                        2'd1 : Out = 8'h02;
                        2'd2 : Out = 8'h04;
                        2'd3 : Out = 8'h08;
                        2'd4 : Out = 8'h10;
                        2'd5 : Out = 8'h20;
                        2'd6 : Out = 8'h40;
                        2'd7 : Out = 8'h80;
                endcase
        end
endmodule
```

2. Write a Verilog module for a 2:4 decoder circuit

   (a) using an always block

```
module Decoder2_4 (Select, Out0, Out1, Out2, Out3);
```

```
        input wire [1:0] Select;
        output reg Out0, Out1, Out2, Out3;

        always @ * begin
                case(Select)
                        2'b00 : {Out3, Out2, Out1, Out0} = 4'b0001;
                        2'b01 : {Out3, Out2, Out1, Out0} = 4'b0010;
                        2'b10 : {Out3, Out2, Out1, Out0} = 4'b0100;
                        2'b11 : {Out3, Out2, Out1, Out0} = 4'b1000;
                endcase
        end
endmodule
```

(b) using continuous assignment

```
module Decoder2_4 (Select, Out0, Out1, Out2, Out3);
        input wire [1:0] Select;
        output wire Out0, Out1, Out2, Out3;

        assign Out3 = (Select==2'b11);
        assign Out2 = (Select==2'b10);
        assign Out1 = (Select==2'b01);
        assign Out0 = (Select==2'b00);
endmodule
```

3. DDCA 4.19 We want the position of the highest order bit of A that is a 1. We use priority logic to check each of the bits of A, starting at the highest order bit. It is called priority logic because the circuit acts behaves like it considers certain bits before others. Think about when if-else statements in Verilog will create priority logic and when they will not.

```
module PriorityEncoder83 (A, Y, NONE);
        input wire [7:0] A;
        output reg [2:0] Y;
        output reg NONE;

        always @ (*) begin
                NONE = 0;
                if      (A[7]) Y = 3'b111;
                else if (A[6]) Y = 3'b110;
                else if (A[5]) Y = 3'b101;
                else if (A[4]) Y = 3'b100;
                else if (A[3]) Y = 3'b011;
                else if (A[2]) Y = 3'b010;
                else if (A[1]) Y = 3'b001;
                else if (A[0]) Y = 3'b000;
                else begin
                        Y = 3'b000;
```

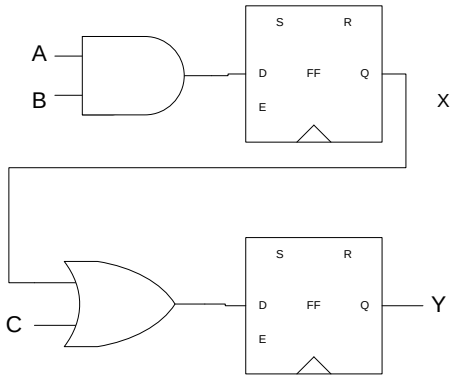
```
                NONE = 1'b1;
            end
        end
    endmodule
```

Note that in the **else** clause Y is set. If a signal is assigned in only some but not all paths through an always@* block, then a latch will be generated. The latch stores the old value for cases when the signal is not assigned. This is usually not the intended behavior.

Think of the circuits that could implement the PriorityEncoder. On one hand you might have a cascading circuit where each stage passes the output thru if a previous stage set it or else checks if the corresponding bit is high to set the output. On the other hand, you might instead have a 'priority mux' that selects based on the most significant bit that is high. This is a simple truth table that can be implemented without needing a stage of logic for every if statment.

extra question: How many 6LUTs could our PrioirtyEncoder83 fit into?

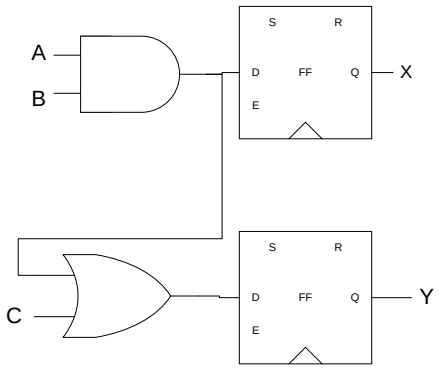
4. DDCA 4.46 The two modules have the same function and both imply the same hardware.



Both modules use the non-blocking assignment (<=) in the always@(posedge clk) block. Non-blocking assignment means that all assignments happen at the same time and so ordering does not matter. In both, x gets the value of a & b and y gets the value of the old x ORed with c. Remember that always@(posedge clk) is used for inferring state elements that capture the value on their input at the rising edge of clk. This is why x and y are outputs of flipflops.

5. DDCA 4.47

   (a) Code1 now has a different behavior and implies different hardware than that in 4.46.



Blocking assignments (=) are procedural, which means that the assignments happen in the order they appear in the always block. A simulator (like Modelsim) that is simulating this this Verilog module would assign the variables in this order, like execution of a programming language. However, remember that if we are synthesizing hardware from behavioral Verilog (i.e. with always blocks), the hardware will just have the behavior of the Verilog. When the always@(posedge clk) block is "executed", x is assigned to a & b, then y is assigned to x OR c. Since the y assignment comes second, the x used in x OR c is the new x, so effectively what y is getting is (a & b) OR c, as can be seen in the diagram. This is probably not the intended hardware, particularly since the value of internal reg x is not even used. Since, using blocking assignment = in an always@(posedge clk) can be easy to make mistakes with (and hard to comprehend), it is best to strictly use non-blocking in always@(posedge clk) and blocking in always@*. Even more importantly, never mix (=) and (<=) in the same always block; this is just asking for trouble and confusion. If you find that you think you need to break these rules while writing a Verilog module, then you should probably look for another way to write your code that obeys the rules.

(b) Code2 happens to have the same behavior and imply the same hardware as the modules in 4.46. This is just because the order of the assignments presents no dependencies. When done in this order, y gets x OR c, then x gets a & b. So we use "old" value of x for the y assignment, just like with the non-blocking (<=) assignment.

How <= and = result in different circuits can be a subtle point to grasp; however, if you are still confused trace carefully through each of the 4 modules to understand the behavior. Trace this behavior in the respective circuit diagrams. For blocking assignments, you can try substituting the new value of a variable into where it is used later in the always block (as we did for x in 4.47 code1).

6. DDCA 4.48 excluding part h.

(a) Recall the behavior of a level-sensitive latch is that when clk is low, the latch is "opaque" and q will hold its value, unaffected by d. When clk is high, the latch is "transparent" and the latch will continously capture the value of d (i.e. q gets d). (Contrast this with a flipflop that only captures d at the rising edge of the clk). Next, recall that an always block is "executed" whenever one of the signals in its sensitivity list changes. Here only clk is in the list, so the block is executed only when clk changes. So if clk is high, q should be

getting the value of d, but even if d is changing, the always block will not execute in to assign it to q. Clearly the incorrect behavior for a latch. (Interestingly, as it is written, this circuit operates as a flipflop, because when clk goes high-to-low (a falling edge), the always block executes, but no assignment occurs because clk is low. When clk goes low-to-high (a rising edge), the always block executes again, and the assignment occurs because clk is high.) Fix: include both clk and d in the sensitivity list. Even better just use always@*, which is a catch-all for all signals in the always block.

(b) By the title of the module, "gates", we can assume this is just trying to assign the output to the output of 2-input (a and b) gates. Again, the problem is that a is included in the sensitivity list of the always, but b is not, even though assignments depend on both signals. Fix: include b in the sensitivity list or just use *.

(c) Since the always block triggers on posedge s, s is only ever going to be high when the if(s) check occurs, so y will never be assigned to d0. In addition, a mux is just combinational logic and should not be created using posedge because changes to the output occur at both transitions (posedge and negedge). Also, since d1 and d0 are not in the sensitivity list, neither will be assigned to s except when s changes. Fix:

```
always @ * begin
        if (s) y = d1;
        else   y = d0;
end
```

Note that keeping the <= would have still given correct behavior, but it is best practice to use = inside always@* blocks.

(d) In always @ (posedge clk) we should only use non-blocking assignments (<=).

(e) The code does not state what happens to out1 when state!=0 or what happens to out2 when state==0. Therefore, the tools will assume that out1 holds its value when state!=0 and out2 holds its value when state==0. This is not likely to be the intended behavior since the module is an FSM, whose outputs should be a combinational logic function of the state and input. Here, the output gets latched and actually each out signal would remain 1 after it first becomes 1. To avoid generating unwanted latches, make sure that in your always@* blocks, every path through the block assigns all signals that can be assigned in the block. Consider adding default assignments at the top of the block or just cover all possibilities in each if-clause. Fix: two solutions shown for the always block

```
always @ * begin
        out1 = 0;
        out2 = 0;
        if (state==0) out1 = 1;
        else          out2 = 1;
end
```

OR

```
always @ * begin
        if (state==0) begin
                out1 = 1;
```

```
                out2 = 0;
        end else begin
                out1 = 0;
                out2 = 1;
        end
end
```

(f) Ignoring the different apostrophes in the constants, the problem is that the if-statements do not cover the case where a equals 4'b0000. Even if you know that the input a will never be 4'b0000, the tools do not necessarily know that and will likely generate a latch for y to cover the case that a is 4'b0000. Again, use either a default assignment at the top of the block or a default case, which for if-statements is "else". The default can be a don't care, x, if you do not care what the value is in the extra case. Fix:

```
always @ * begin
        y = 4'bxxxx;
        if (a[3]) y = 4'b1000;
        else if (a[2]) y = 4'b0100;
        else if (a[1]) y = 4'b0010;
        else if (a[0]) y = 4'b0001;
end
```

OR

```
always @ * begin
        if (a[3]) y = 4'b1000;
        else if (a[2]) y = 4'b0100;
        else if (a[1]) y = 4'b0010;
        else if (a[0]) y = 4'b0001;
        else           y = 4'bxxxx;
end
```

(g) The problem here is that FSM module state encodings should, in general, be private to the module implementation. This means use localparam instead of parameter for state encodings. The state encodings could actually be parameters as long as the implementation does not depend on the chosen encoding and the instantiator sets the parameters to be the right bit width and all different. However, in this case, it would not work because the implementation does depend on the encoding. The last case line is "2", which is the encoding for S2 (2'b10), but now makes the case dependent on S2 being 2'b10. In addition to this, the case statement has no default behavior for the case where state is 2'b11. This will generate a latch for nextstate to cover the case of 2'b11. When coding a state machine, make sure that if your states don't use all the possible encodings for the bit width of state (in this case there are 2 bits of state but only 3 not 4 states), then make a default case in logic. Fix: change state encodings to be localparam. Also add the extra case and change "2" to "S2".

```
always @ * begin
        case(state)
                S0 : nextstate = S1;
```

```
                S1 : nextstate = S2;
                S2 : nextstate = S0;
                default : nextstate = 2'bxx;
        endcase
end
```

OR

```
always @ * begin
        nextstate = 2'bxx;
        case(state)
                S0 : nextstate = S1;
                S1 : nextstate = S2;
                S2 : nextstate = S0;
        endcase
end
```

(h) We did not assign h, but if you are interested, read on. The definition for the tristate module and an explanation is on DDCA page 180. It has the following truth table.

a	en	y
0	0	z
0	1	0
1	0	z
1	1	1

So if en (enable) is high, the tri-state passes the input to the output. If en is low, then the tri-state does not drive the output. This allows two tri-state outputs to attach to the same wire, with the condition that their enable signals are never high at the same time (otherwise the wire has two drivers at once!). Therein lies the problem with the code in part (h). The same signal s is used for both tri-states and both tri-states output on y. The correct code should be:

```
module mux2tri(input [3:0] d0, d1,
        input s,
        output [3:0] y);

        tristate t0 (d0, ~s, y);
        tristate t1 (d1, s, y);
endmodule
```

(i) Assuming that the designer intended to have an asynchronous set and reset (note that asynchronous reset and set are sometimes called clear and preset, respectively), the problem is that when reset and set go high at the same time, which takes precedence. The code is ambiguous; assignments to q must be mutually exclusive, as are the two assignments in the first always block. We will take the convention that reset has priority over set, to demonstrate our fix. Note also the posedge on reset is not necessary, since we only do something special if reset is high anyway. Fix:

```
module floprsen(input clk,
```

```
                input reset,
                input set,
                input [3:0] d,
                output [3:0] q);

        always @(posedge clk, reset, set) begin
                if (reset) q <= 0;
                else if (set) q <= 1;
                else q <= d;
        end
endmodule
```

In addition to the priority of set and reset, if the designer intended to make a flipflop with synchronous set and reset, then we have to make the always block only sensitive to posedge clk, not reset and set. This way, the reset or set happens only at the rising edge of the clock.

(j) Non-blocking (<=) assignments are used, so the second assignment will use the previous tmp value not the one that just got the current AND of a and b. Also, even though it is necessary here because the always only needs to trigger on a, b, or c, we might as well use always@*. Fix:

```
always @(*) begin
        tmp = a & b;
        y = tmp & c;
end
```

7. DDCA 4.27

```
module TrafficLight (Ta, Tb, La, Lb, Clock, Reset);
        input wire Ta, Tb;
        output wire [1:0] La, Lb;

        reg [1:0] state, next_state;

        // state encodings
        localparam      S0 = 2'b00,
                        S1 = 2'b01,
                        S2 = 2'b10,
                        S3 = 2'b11;

        // color encodings
        localparam      GREEN = 2'b00,
                        YELLOW = 2'b01,
                        RED = 2'b10;

        // state
        always @(posedge Clock) begin
```

```
        if (Reset) next_state <= S0;
        else  state <= next_state;
    end

    // next_state logic
    always @ (*) begin
        case (state)
            S0 : begin
                    if (Ta) next_state = S0;
                    else next_state = S1;
                 end
            S1 : next_state = S2;
            S2 : begin
                    if (Tb) next_state = S2;
                    else next_state = S3;
                 end
            S3 : next_state = S0;
        endcase
    end

    // output logic (separate for clarity)
    always @ (*) begin
        case (state)
            S0 : begin
                    La = GREEN;
                    Lb = RED;
                 end
            S1 : begin
                    La = YELLOW;
                    Lb = RED;
                 end
            S2 : begin
                    La = RED;
                    Lb = GREEN;
                 end
            S3 : begin
                    La = RED;
                    Lb = YELLOW;
                 end
        endcase
    end
endmodule
```

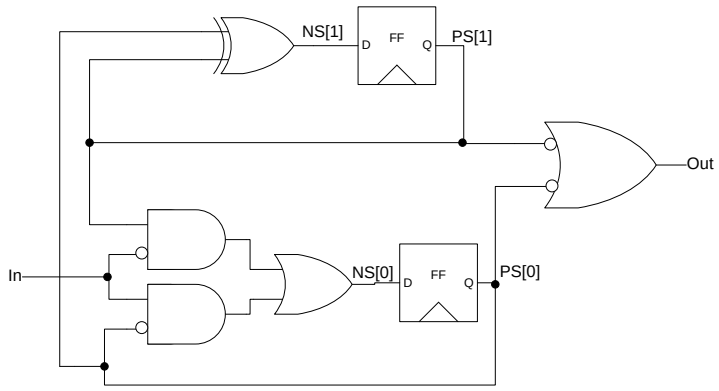
Notice that encodings are used for the color outputs. This is useful because you can later change things more easily. Also, you don't have to think too hard: just assign the output based on the

state (as shown in the state bubbles of the FSM diagram). You can let the tools do the logic simplification for you.

The tools should reduce the high-level output assignments shown above to gates that correspond to:

```
assign La[1] = state[1];
assign La[0] = ~state[1] & state[0];
assign Lb[1] = ~state[1];
assign Lb[0] = &state;
```

8. Consider the following finite state machine (FSM) circuit:



(a) Write a Verilog description of the circuit using continuous assignment for the NS and OUT signals.

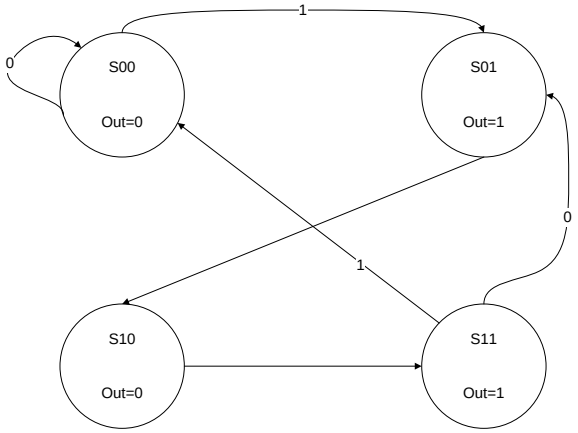
```
module FooCircuit(Clock, Reset, In, Out)
        input Clock, Reset, In;
        output Out;

        reg [1:0] PS, NS;

        always @ (posedge Clock) begin
                if (Reset) PS <= 2'b00;
                else PS <= NS;
        end

        assign NS[0] = (In & ~PS[0]) | (~In & PS[1]);
        assign NS[1] = PS[1] ^ PS[0];
        assign Out = ~PS[1] | ~PS[0];

endmodule
```

(b) Draw a state transition diagram describing the behavior of the circuit. Within each state bubble indicate the bit encoding for that state. Remember to label the arcs with input values and state with output values.

Write out the truth table to see what the transitions are.

PS[1]	PS[0]	In	NS[1]	NS[0]
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	0
1	0	0	1	1
1	0	1	1	1
1	1	0	0	1
1	1	1	0	0

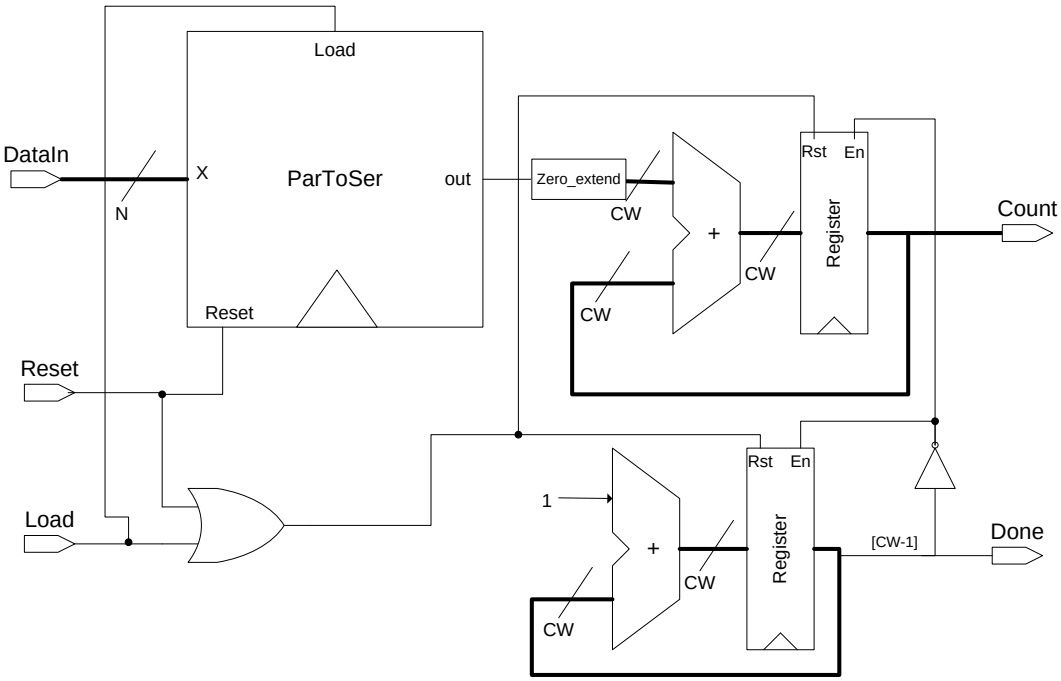
Using this we look at each state PS[1], PS[0] and make an arc to state NS[1], NS[0] for each value of In. Below is the resulting state transition diagram. Arcs with no label for the input mean that In==1 and In==0 both go the same path (i.e. the input does not matter from that state).



9. A "population count" (PopCount)

   (a) If the input DataIn is N bits, how many bits must the Count output be? This is "CW" in the figure. For an N bit input there could be between 0 and N high (1) bits. This means Count can take on N+1 values. So we need log2(N+1) bits. Since this can be a fraction, we will take the ceiling. ceil(log2(N+1)).

   (b) Draw your PopCount circuit for N=4. (shown for N) The below shows one possible design for PopCount. It uses a parallel-to-serial converter to take in the N-bit input and shift it out one bit each cycle. On each cycle the bit coming out of the converter is added to a running total, so if it is 1 the count is increased and if 0 then the count stays the same. A counter with the same width is used to assert the Done signal after the N cycles. When Done is high, the counters are disabled so that the Count output stays valid until the next load happens.



(c) Write the Verilog for the PopCount, making N a parameter.

```
module PopCount(Clock, Reset, Load, Done, DataIn, Count);
        parameter N = 32;  // 32 is just an arbitrary default for N
        parameter CW = ceil(log2(N+1));
        // ceil and log2 not available in Verilog, but there is an
        // integer `log2 macro available in Const.v.  If you ever
        // use it in your Verilog code make sure to look at Const.v
    // to see exactly what it does

        input Clock, Reset, Load;
        input [N-1:0] DataIn;
        output Done;
        output reg [CW-1:0] Count;

        // infer a counter that stops when Done is high
        reg [CW-1:0] DoneCount;
        always @ (posedge Clock) begin
                if (Reset|Load) DoneCount <= 0;
                else if (~Done) DoneCount <= DoneCount+1;
        end

        // Done when DoneCount reaches N
        assign Done = (DoneCount==N);

        // serial bit to count during the cycle
```

```
    wire bitout;

    // We want to load the input in parallel but feed it
// to the adder serially one bit per clock cycle.
    // We will use the parallel-to-serial converter
    // from "Lecture #6 CAD Tools (Synthesis)"
    // (with a Reset signal).
    ParToSer #(.N(N)) (
          ptos (.LD(Load),
                              .Reset(Reset),
                              .X(DataIn),
                              .out(bitout),
                              .CLK(Clock));

    // zero extend the bit to be the width of the
// adder to add to current count if bitout is
// 0 then will add zero, if bitout is a 1 then
// we will be adding 1 (incrementing the count)
    wire [CW-1:0] zeroExtendedBit;
    assign zeroExtendedBit = {{(CW-1){1'b0}}, bitout};

    // add the bit to the stored count so far
    wire [CW-1:0] addresult;
    assign addresult = zeroExtendedBit + Count;

    // register for Count; gets disabled when Done
    // is high to hold the output until the next
    // Load happens
    always @ (posedge Clock) begin
          if (Reset|Load) Count <= 0;
          else if (~Done) Count <= addresult;
    end
endmodule
```

Keep in mind that for the N-cycle PopCount, many Verilog implementations are possible. For example, instead of zero-extending the bit and adding it to the current Count, we could have instead implemented an incrementer like in Lab3, where if the bit==1 then increment and if b==0 keep the current value. It is likely that this would synthesize to the same hardware since the add one is like increment enable. Also, keep in mind, that other implementations of general PopCount are possible. You could count all the bits purely combinationally using a tree of adders and carry-save addition. We will cover design components, like different types of adders, later in the semester.