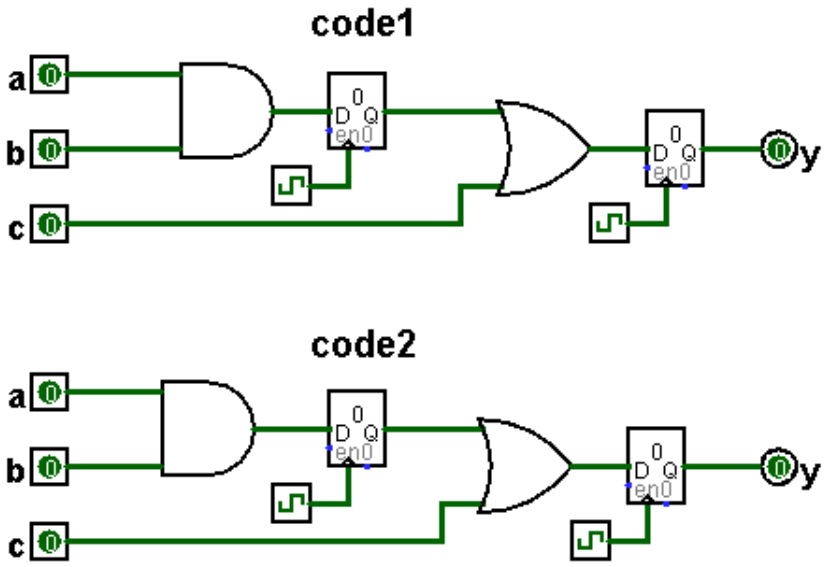
CS150 Fall 2012 — Solutions to Homework 2

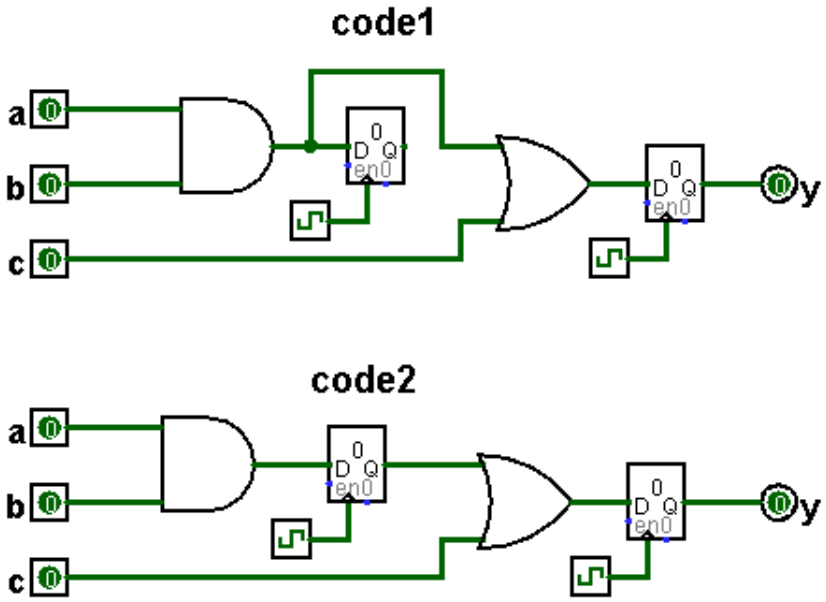
September 11, 2012

Problem 1, part 1: DDCA 4.46



With non-blocking assignments, the circuits do have the same functionality.

Problem 1, part 2: DDCA 4.47



With blocking assignments, the circuits do **not** have the same functionality. The left-hand side of each assignment is updated before the right-hand side of the next assignment is updated!

Problem 1, part 3: DDCA 4.48

a)
```
module latch (input            clk,
              input      [3:0] d,
              output reg [3:0] q);
      always @ (clk)
          if (clk) q <= d;
endmodule
```

The sensitivity list for the always block only has the clock listed. Therefore, the latch will only copy the value of d when it changes; since the output will be insensitive to changes in d while the clock is high. This block will actually function like a negative-edge triggered flip-flop. Changing the sensitivity to 'always @ (clk,d)' will fix the problem.

b)
```
module gates(input      [3:0] a, b,
             output reg [3:0] y1, y2, y3, y4, y5);
      always @ (a)
          begin
             y1 = a & b;
             y2 = a | b;
             y3 = a ^ b;
             y4 = ~(a & b);
             y5 = ~(a | b);
          end
endmodule
```

The logic in the always block depends on both a and b, while the sensitivity list only includes a. Modifying the sensitivity to 'always @ (a,b)' will fix the problem.

c)
```
module mux2(input        [3:0] d0, d1,
              input            s,
              output reg [3:0] y);
      always @ (posedge s)
          if (s) y <= d1;
          else   y <= d0;
endmodule
```

The sensitivity list does not include the data inputs; this means that the output will only change when the select input changes. Changing the selected data input will not alter the output while s remains constant. Also, the sensitivity list is only for the positive edge of s, which will cause the multiplexer to latch output d0, as (s) is low when used in a condition on a transition. Finally, the use of non-blocking assignments inside combinational logic breaks from convention established for CS150. Proper Verilog code would be:

```
module mux2(input        [3:0] d0, d1,
              input            s,
              output reg [3:0] y);
      always @ (s,d0,d1)
          if (s) y = d1;
          else   y = d0;
endmodule
```

d)
```
module twoflops(input        clk,
```

```
                    input      d0, d1,
                    output reg q0, q1);
        always @ (posedge clk)
            q1 = d1;
            q0 = d0;
endmodule
```

The always block contains two statements, but does not have a begin-end block. Also, the use of blocking assignments for synchronous elements breaks convention, even though this particular example does not need non-blocking assignments to function. Proper Verilog code is show below:

```
module twoflops(input        clk,
                    input      d0, d1,
                    output reg q0, q1);
        always @ (posedge clk)
            begin
               q1 <= d1;
               q0 <= d0;
            end
endmodule
```

e)
```
module FSM(input      clk,
            input      a,
            output reg out1,out2);

   reg state;

   // next state logic and register (sequential)
   always @ (posedge clk)
      if (state == 0) begin
         if (a) state <= 1;
      end else begin
         if (~a) state <= 0;
      end

   always @ (*) // output logic (combinational)
      if (state == 0) out1 = 1;
      else            out2 = 1;
endmodule
```

The output out1 is assigned only when state is 0, and the output out2 is assigned only when state is not 0; this results in latches being inferred on each output (not combinational). The always block can be fixed as follows:

```
   always @ (*) // output logic (combinational)
      begin
         out1 = (state == 0);
         out2 = ~(state == 0);
      end
```

f)
```
module priority (input      [3:0] a,
                 output reg [3:0] y);
```

```
   always @ (*)
      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;
endmodule
```

There is no "else" statement to cover the case in which every bit of a is zero. The always block can have this final case added as follows:

```
   always @ (*)
      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'b0000;
```

g)
```
module divideby3FSM (input  clk,
                     input  reset,
                     output out);

   reg [1:0] state, nextstate;

   parameter S0 = 2'b00;
   parameter S1 = 2'b01;
   parameter S2 = 2'b10;

   // State Register
   always @ (posedge clk, posedge reset)
      if (reset) state <= S0;
      else       state <= nextstate;

   // Next State Logic
   always @ (state)
      case (state)
        S0: nextstate = S1;
        S1: nextstate = S2;
         2: nextstate = S0;
      endcase

   // Output Logic
   assign out = (state == S2);
endmodule
```

The literal "2" is used for one of the cases of the case statement; this is inconsistent with the use of parameters for state encodings. Furthermore, since the encoding values are not localparams, they may be changed on instantion; this would cause the FSM to fail if S2 were altered from 2. Finally, there is no default to cover the value 2'b11, therefore inferring a latch on the output of the combination logic determining nextstate. The corrected code is below:

```
   ...
```

```
   localparam S0 = 2'b00;
   localparam S1 = 2'b01;
   localparam S2 = 2'b10;
   ...
   // Next State Logic
   always @ (state)
      case (state)
        S0: nextstate = S1;
        S1: nextstate = S2;
        default: nextstate = S0;
      endcase
   ...
```

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

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

Both tristate instances have 's' as an input; one should have the complement to prevent them from both driving the bus at the same time.

```
   ...
   assign not_s = ~s
   tristate t0(d0, not_s, y);
   tristate t1(d1, s, y);
   ...
```

i)
```
module floprsen (input            clk,
                 input            reset,
                 input            set,
                 input      [3:0] d,
                 output reg [3:0] q);

   always @ (posedge clk, posedge reset)
      if (reset) q <= 0;
      else       q <= d;

   always @ (set)
      if (set) q <= 1;
endmodule
```

Here, q is assigned in two different always blocks. In order to resolve the issue of both set and reset being high (which exists in hardware SR flops), we assign some priority to the inputs, as shown below:

```
   ...
   always @ (posedge clk, posedge reset, posedge set)
      if (reset) q <= 0;
      else begin
```

```
            if (set) q <= 1;
            else     q <= d;
         end
      ...
```

j)
```
module and3 (input      a, b, c,
             output reg y);

   reg tmp;

   always @ (a,b,c)
   begin
      tmp <= a & b;
      y   <= tmp & c;
   end
endmodule
```

Here, the use of non-blocking assignments causes y to use the value of tmp from before the execution of the first assignment. If tmp were in the sensitivity list, this would not be a problem; however, as tmp is not in the sensitivity list, the new value of tmp will not cause the change to propagate to y until a, b, or c changes. An idiomatic fix is:

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

Problem 2

a) The per-cost part for an FPGA implementation is \$10.

For 180nm CMOS:

$$5\,\text{mm}^2 * \frac{\$0.05}{\text{mm}^2} = \$0.25$$

For 22nm CMOS:

$$.5\,\text{mm}^2 * \frac{\$0.1}{\text{mm}^2} = \$0.05$$

b) The cost of the first unit for an FPGA is:

$$\$50,000 + \$10 = \$50,010$$

For 180nm CMOS:

$$\$250,000 + \$0.25 = \$250,000.25$$

For 22nm CMOS:

$$\$10,000,000 + \$0.05 = \$10,000,000.05$$

c) The unit cost of each specified volume for an FPGA is:

$$(\$50,000 + \$10 * 10^3)/10^3 = \$60.00 \text{ for one thousand}$$

$$(\$50,000 + \$10 * 10^6)/10^6 = \$10.05 \text{ for one million}$$

$$(\$50,000 + \$10 * 10^9)/10^9 = \$10.00005 \text{ for one billion}$$

For 180nm CMOS:

$$(\$250,000 + \$0.25 * 10^3)/10^3 = \$250.25 \text{ for one thousand}$$

$$(\$250,000 + \$0.25 * 10^6)/10^6 = \$0.50 \text{ for one million}$$

$$(\$250,000 + \$0.25 * 10^9)/10^9 = \$0.25025 \text{ for one billion}$$

For 22nm CMOS:

$$(\$10,000,000 + \$0.05 * 10^3)/10^3 = \$10,000.05 \text{ for one thousand}$$

$$(\$10,000,000 + \$0.05 * 10^6)/10^6 = \$10.05 \text{ for one million}$$

$$(\$10,000,000 + \$0.05 * 10^9)/10^9 = \$0.06 \text{ for one billion}$$

d) 180nm process technology costs the same as FPGA implementation at volume:

$$\lceil \frac{250,000 - 50,000}{\$10 - \$0.25} \rceil = 20,513\text{units}$$

e) 22nm process technology costs the same as FPGA implementation at volume:

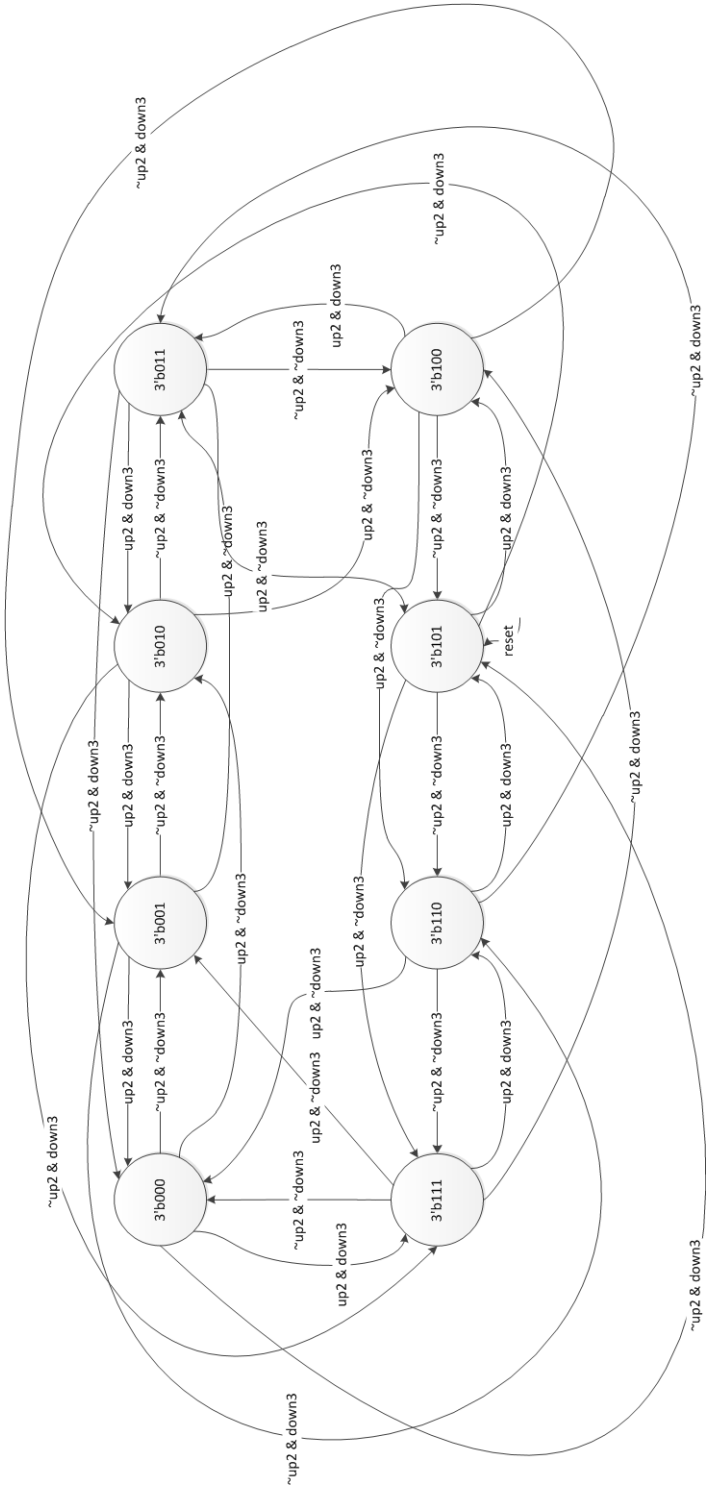
$$\lceil \frac{10,000,000 - 50,000}{\$10 - \$0.05} \rceil = 1,000,000\text{units}$$

22nm process technology costs the same as 180nm implementation at volume:

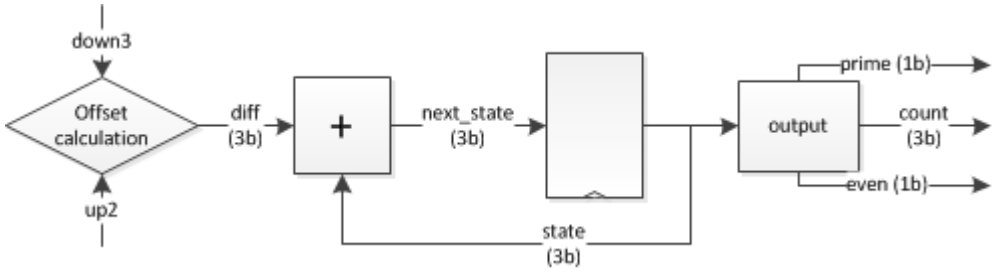
$$\lceil \frac{10,000,000 - 250,000}{\$0.25 - \$0.05} \rceil = 48,750,000\text{units}$$

Problem 3

a) State transition diagram:



b) Block diagram:



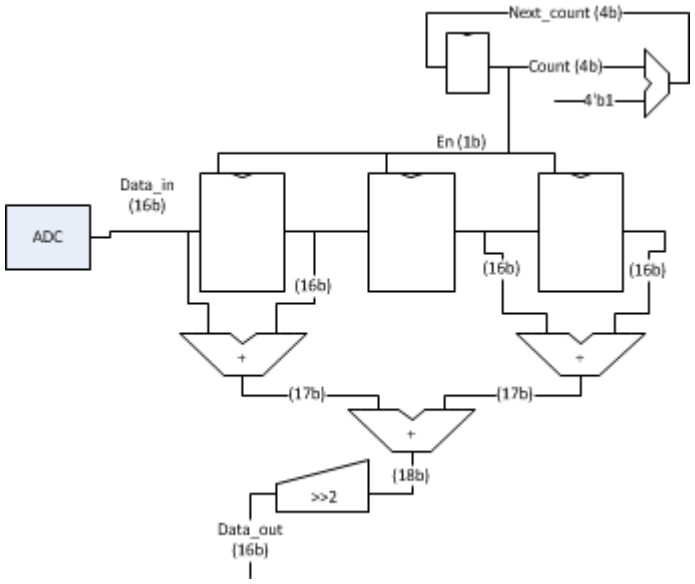
c) Verilog:

```
module counter (input        clk,
                input        up2, down3,
                output [2:0] count,
                output       even, odd);

    reg [2:0] state, next_state, diff;
    assign count = state;

    always @ (posedge clk)
       begin
          if (reset) state <= 3'd5;
          else state <= next_state;
       end

    always @ (*)
       begin
          case ({up2,down3})
             2'b11 : diff = 3'b111;
             2'b10 : diff = 3'b010;
             2'b01 : diff = 3'b100;
             default : diff = 3'b001;
          endcase
          next_state = state + diff;
       end

    assign even = ~state[0];
    assign prime = (state == 3'b10) || (state[0] && (state != 3'b1));

endmodule
```

Problem 4

a) For the 4-tap FIR moving average filter, no multiplier is needed. Division is accomplished through an arithmetic right-shift or part-select operator.

Block diagram:



b) Verilog:

```
module moving_avg_4tap (input         clk,
                        input  [15:0] data_in,
                        output [15:0] data_out);

    reg [3:0] enable_counter;
    reg [15:0] data_in_d1, data_in_d2, data_in_d3;
    reg [17:0] sum;

    always @ (posedge clk)
        enable_counter <= enable_counter + 4'b1;

    always @ (posedge clk)
        if (reset) enable_counter <= 4'b0;
        else
            if (enable_counter == 4'b0) begin
                data_in_d1 <= data_in;
                data_in_d2 <= data_in_d1;
                data_in_d3 <= data_in_d2;
            end

    always @ (*)
        sum = data_in + data_in_d1 + data_in_d2 + data_in_d3;

    assign data_out = sum[17:2]; // divide with part-select
endmodule
```