Introduction

- Assignment #2
- Simple Verilog Examples
  - Combinational logic
  - Sequential logic (registers, clk)
- Verilog CPU simulation
- Verilog Blocking vs. Non-blocking
- Xilinx project CPU1

Please put your cellphone on vibrate
Assignment 2

1. Implement your CPU in Verilog in Xilinx.
2. Demo your Verilog CPU simulation & synthesis result to TA.
3. Due next Friday 6pm (8/21)
4. Grading:
   a. Verilog Completion (no-syntax error) 40%
   b. Simulation 40%
   c. Synthesis 20%
• Verilog is case sensitive
• All keywords are in lower case
• Module is the basic design unit
• Module contains ports (input/output) and logical assignments
module mux2to1(s, a, b, y);
  input s;
  input a;
  input b;
  output y;

  // This is a comment
  assign y = (b & s) | (a & ~s);
endmodule
module mux2to1(s, a, b, y);
    input s;
    input a;
    input b;
    output y;

    wire na, nb;

    assign nb = b & s;
    assign na = a & ~s;
    assign y = na | nb;
endmodule
module mux2to1(s, a, b, y);
    input s;
    input a;
    input b;
    output y;

    // conditional statement
    assign y = s ? b : a;
endmodule
wire na;

// wire is shorted together
assign na = b & s;
assign na = a & ~s;
module mux2to1(s, a, b, y);
  input s;
  input a;
  input b;
  output y;

  reg y;

  always @(a or b or s)
  begin
    if (s)
      y = b;
    else
      y = a;
  end

endmodule
module mux2to1(s, a, b, y);
    input s;
    input a;
    input b;
    output y;

    reg y;

    always @(a or b or s)
    begin
        y = (b & s) | (a & ~s);
    end
endmodule

module mux2to1(s, a, b, y);
    input s;
    input a;
    input b;
    output y;

    reg y;

    always @(a or b or s)
    begin
        y = (b & s) | (a & ~s);
    end
endmodule
module mux2to1(s, a, b, y);
    input s;
    input a;
    input b;
    output y;

    reg y, na, nb;

    always @*
    begin
        nb = b & s;
        na = a & ~s;
        y = na | nb;
    end

endmodule
always (ld or clr or d or q_old)
begin
  q = q_old;
  if (ld)
    q = d;
  if (clr)
    q = 0;
end

- No else statement
- clr takes precedence
always (ld or clr or d or q_old)
begin
  q = q_old;
  if (clr)
    q = d;
  if (ld)
    q = 0;
end

- No else statement
- ld will take precedence
Common mistake

// correct
always (ld or d or q_old)
begin
  q = q_old;
  if (ld)
    q = d;
end

// mistake
always (ld or d)
begin
  if (ld)
    q = d;
end

- no assignment made to q when ld = 0
module mux2to1_4bit(s, a, b, y);
    input s;
    input [3:0] a, b;
    output [3:0] y;

    mux2to1 u3 (.a(a[3]), .b(b[3]), .s(s), .y(y[3]));
    mux2to1 u2 (.a(a[2]), .b(b[2]), .s(s), .y(y[2]));
    mux2to1 u1 (.a(a[1]), .b(b[1]), .s(s), .y(y[1]));
    mux2to1 u0 (.a(a[0]), .b(b[0]), .s(s), .y(y[0]));
endmodule

module mux2to1(s, a, b, y);
    input s;
    input a;
    input b;
    output y;

    // conditional statement
    assign y = s ? b : a;
endmodule
module mux2to1_4bit(s, a, b, y);
    input s;
    input [3:0] a, b;
    output [3:0] y;

    assign y = (b & s) | (a & ~s);
endmodule

Combinational logic for buses
module mux4to1_4bit(s, a, b, c, d, y);
    input [1:0] s;
    input [3:0] a, b, c, d;
    output[3:0] y;

    reg [3:0] y;

    always @*
    begin
        if (s == 2'b00)
            y = a;
        else if (s == 2'b01)
            y = b;
        else if (s == 2'b10)
            y = c;
        else
            y = d;
    end

endmodule
4-bit 4:1 Mux - case

module mux4to1_4bit(s, a, b, c, d, y);
  input [1:0] s;
  input [3:0] a, b, c, d;
  output[3:0] y;

  reg [3:0] y;

  always @*
  begin
    case (s)
      2'b00 : y = a;
      2'b01 : y = b;
      2'b10 : y = c;
      default: y = d;
    endcase
  end

endmodule
module add4bit(a, b, s);
  input [3:0] a, b;
  output [3:0] s;
  assign s = a + b;
endmodule

4-bit adder

NO carry in
NO carry out
module add4bit(ci, a, b, s, co);
    input ci;
    input [3:0] a, b;
    output [3:0] s;
    output co;

    wire [4:0] y; // not used in always block

    assign y = {1'b0, a} + {1'b0, b} + {4'b0, ci};
    assign s = y[3:0]; // 5-bits
    assign co = y[4]; // 5-bits

endmodule

\[
\begin{array}{cccc}
    a_3 & a_2 & a_1 & a_0 \\
    + & b_3 & b_2 & b_1 & b_0 \\
\hline
    c_0 & s_3 & s_2 & s_1 & s_0
\end{array}
\]
D Flip flop

module latch(d, clk, q);
    input d, clk;
    output q;

    always @(posedge clk)
    begin
        q <= d;
    end

endmodule
reg qa, qb, qc;
always @(posedge clk)
begin
    qa <= a;
    qb <= qa;
    qc <= qb;
end

reg qa, qb, qc;
always @(posedge clk)
begin
    qa <= a;
end
always @(posedge clk)
begin
    qb <= qa;
end
always @(posedge clk)
begin
    qc <= qb;
end
module reg8bit (clk, r, d, ld, q);

    input clk, r, ld;
    input [7:0] d;
    output [7:0] q;

    always @(posedge clk)
    begin
        if (!r)
            q <= 8'b0;
        else
            begin
                if (ld)
                    q <= d;
                end
            end
    end

endmodule
module UpDownCounter (control, clk, counter);

    input control, clk;
    output [1:0] counter;

    always @(posedge clk)
    begin
        if (control)
            counter <= counter + 1;
        else
            counter <= counter - 1;
    end
endmodule
module MooreFSM(A, clk, Z);

    input A, clk;
    output Z;
    reg Z;

    parameter s0=0, s1=1, s2=2, s3=3;
    reg [0:1] MooreState;

    always @(posedge clk)
    begin
        case (MooreState)
            s0: MooreState <= (!A) ? s0: s2;
            s1: MooreState <= (!A) ? s0: s2;
            s2: MooreState <= (!A) ? s2: s3;
            s3: MooreState <= (!A) ? s1: s3;
        endcase
    end

    always @(MooreState)
    begin
        case (MooreState)
            s0: Z = 1;
            s1: Z = 0;
            s2: Z = 0;
            s3: Z = 1;
        endcase
    end

endmodule
**Blocking vs. non-blocking**

// blocking assignments
always @(posedge clk)
begin
  q1 = d; \_\_ creates DFF
  q2 = q1; \_\_ then q2 sets q 1
end

// *non-blocking assignments
always @(posedge clk)
begin
  q1 <= d; \_\_ \_\_ Execute at the same clk cycle
  q2 <= q1;
end

*Non-blocking area only assigned to the LHS targets AFTER the always block completes.*
1. Use blocking assignments ("=") in always blocks that are meant to represent combinational logic.
2. Use non-blocking assignments ("<=") in always blocks that are meant to represent sequential logic.
3. Do not mix blocking & non-blocking assignments in the same always block.
4. In a block with complex if/else statements, ensure all outputs are assigned default values at the beginning.
5. Do not make assignments to the same output from multiple always blocks.
Warning messages

1. "Input X1 is unused"
2. "Output X1 is stuck at VDD or GND"
3. "Output X1 and X2 shared the same net"
4. "Output X1 has no driver" (empty circuit)
• Go through Xilinx project
  ○ Create new project
  ○ Write Verilog code for full-adder
  ○ Write Verilog code for 4-bit adder
  ○ Write test code
  ○ Run simulation
  ○ Synthesize the circuit

module fullAdder(A, B, C, SUM, COUT);
  input A;
  input B;
  input C;
  output SUM;
  output COUT;

  assign SUM = A ^ B ^ C;
  assign COUT = (A & B) | (B & C) | (A & C);
endmodule

module FourBitAdder(A, B, SUM, COUT);
  input [3:0] A;
  input [3:0] B;
  output [3:0] SUM;
  output COUT;

  fullAdder a0 (A[0], B[0], 1'b0, SUM[0], w1);
  fullAdder a1 (A[1], B[1], w1, SUM[1], w2);
  fullAdder a2 (A[2], B[2], w2, SUM[2], w3);
  fullAdder a3 (A[3], B[3], w3, SUM[3], COUT);
endmodule

module fullAdderTest_v;
  // Inputs
  reg A;
  reg B;
  reg C;

  // Outputs
  wire SUM;
  wire COUT;

  // Instantiate the Unit Under Test (UUT)
  fullAdder uut (       .A(A),       .B(B),       .C(C),       .SUM(SUM),       .COUT(COUT)   );

  initial begin
    // Initialize Inputs
    A = 0;
    B = 0;
    C = 0;

    // Wait 100 ns for global reset to finish
    #100;
  end

module FourBitAdderTest_v;
  // Inputs
  reg [3:0] A;
  reg [3:0] B;

  // Outputs
  wire [3:0] SUM;
  wire COUT;

  // Instantiate the Unit Under Test (UUT)
  FourBitAdder uut (      .A(A),      .B(B),      .SUM(SUM),      .COUT(COUT) );

  integer i, j;
  initial begin
    for (i=0; i<16; i=i+1)
      for (j=0; j<16; j=j+1)
        begin
          A = i;
          B = j;
          #10;
        end
// Add stimulus here
A = 0;
B = 0;
C = 1;
#100;

A = 0;
B = 1;
C = 0;
#100;

A = 0;
B = 1;
C = 1;
#100;

A = 1;
B = 0;
C = 0;
#100;

A = 1;
B = 0;
C = 1;
#100;

A = 1;
B = 1;
C = 0;
#100;

A = 1;
B = 1;
C = 1;
#100;

end

endmodule
• Show CPU1 in Xilinx
  ○ Verilog Code (Behavioral)
  ○ Simulation Result (timing chart)
  ○ Synthesis (Hardware)
Reminder

You may now turn on your cellphone ringer