CSE 140L Exam

Prof. Tajana Simunic Rosing

Winter 2010

- Do not start the exam until you are told to.
- Turn off any cell phones or pagers.
- Write your name and PID at the top of every page. Do not separate the pages.
- This is a closed-book, closed-notes, no-calculator exam. You may only refer to one 8 ½” x 11” page of your own handwritten notes.
- Do not look at anyone else’s exam. Do not talk to anyone but an exam proctor during the exam.
- If you have a question, raise your hand and an exam proctor will come to you.
- You have 50 minutes to finish the exam. When the time is finished, you must stop writing.
- Write your answers in the space provided.
- To get the most partial credit, clearly and neatly show all steps of your work.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>15 points</td>
</tr>
<tr>
<td>2.</td>
<td>15 points</td>
</tr>
<tr>
<td>3.</td>
<td>20 points</td>
</tr>
<tr>
<td>4.</td>
<td>15 points</td>
</tr>
<tr>
<td>5.</td>
<td>25 points</td>
</tr>
<tr>
<td>6.</td>
<td>10 points</td>
</tr>
<tr>
<td>Total</td>
<td>100 pts</td>
</tr>
</tbody>
</table>
PROBLEM 1. Answer True (T) or False (F) for the following questions in the space provided.

<table>
<thead>
<tr>
<th>Part</th>
<th>Question</th>
<th>T/F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Tri-state gates can implement the same functionality as a Mux.</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>Non-blocking assignment completes the left side of the assignment before continuing to the next statement.</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>It is better to use non-blocking assignments when modeling both sequential and combinational logic</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>A flip-flop can enter a metastable state if its inputs change too close to the triggering clock.</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>Hold time should be longer than the setup time when cascading flip-flops.</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>DRAM technology is compatible with SRAM technology.</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>The above circuit works without any violations.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>The delay of CLK_BUF (in 7) affects the maximum frequency of operation</td>
<td>F</td>
</tr>
<tr>
<td>9</td>
<td>Clock-skew problem arises when clock signal is delayed in its distribution through the logic circuit.</td>
<td>T</td>
</tr>
<tr>
<td>10</td>
<td>A state diagram with 17 states can be implemented with at least 5 flip flops</td>
<td>T</td>
</tr>
<tr>
<td>11</td>
<td>One-Hot encoding trades the size of combinational logic for the number of flip-flops.</td>
<td>T</td>
</tr>
<tr>
<td>12</td>
<td>Increase in the number of instructions in IROM of lab4 does not require any changes to the program_counter module.</td>
<td>F</td>
</tr>
<tr>
<td>13</td>
<td>A synchronizer is a circuit which guarantees that metastability will never occur</td>
<td>F</td>
</tr>
<tr>
<td>14</td>
<td>SRAM requires a refresh</td>
<td>F</td>
</tr>
<tr>
<td>15</td>
<td>FPGA with a DSP is a programmable logic device (PLD)</td>
<td>T</td>
</tr>
</tbody>
</table>
PROBLEM 2

Four D-FFs produce outputs A, B, C, & D which are then input into combinational logic element \( Y = A'B'C + A'BD + BC' + A'B'D + A'BD \) and stored into another D-FF. Assume \( RnCg = 2\text{ns} \), wire delay is 1\text{ns} per wire, D-FF propagation delay is 3\text{ns} and setup time is 1\text{ns}.

a) Show the transistor level implementation for \( Y \) that results in the maximum operating frequency. What is the smallest clock period this circuit can operate at?

Maximum delay of the transistor circuit = 
3 \times (R_nC_g) = 6 \text{ ns}

Wire delay on either side of the circuit = 
1\text{ns} + 1 \text{ns} = 2 \text{ns}

Propagation delay = 3 \text{ ns}

Setup time = 1\text{ns}

Total = 12 \text{ ns}

The circuit can operate at a clock period of 12 \text{ ns}
b) Using the data provided in part a), and the smallest clock period obtained for the transistor level implementation you derived, find out what should the D-FF hold time be if clock skew is 0.5ns?

Sol:

The hold time of the D-FF should be less than the propagation delay, wire delay and the delay of the transistor circuit

\[ = 3 \text{ ns} + 6 \text{ ns} + 1 \text{ ns} + 1 \text{ ns} = 11 \text{ ns} \]

Clock skew = 0.5 ns =>
Propagation delay + Delay of transistor circuit + Wire delays \( \geq \) Hold time – Clock skew
The hold time = 11 ns + 0.5 ns = 11.5 ns
PROBLEM 3

Design a synchronous 4-bit up / down counter with parallel load capability and asynchronous reset using Verilog. The contents of the counter are Cnt[3:0]. Asynchronous reset signal is aRst. Two bit function selector (Fin[1:0]) determines which of the following functions will be implemented:

- 00 Hold (register contents remain unchanged)
- 01 Increment
- 10 Decrement
- 11 load from external inputs Din[3:0]

Sol:

```verilog
module mid2register (clock,aRst,Din,Fin,Cnt);
    input clock,aRst;
    input [3:0]Din;
    input [1:0]Fin;
    output [3:0]Cnt;
    reg [3:0]Cnt;

    always @ (negedge aRst or posedge clock)
    begin
        if (!aRst)
            Cnt <= 0;
        else
            case(Fin)
                1 : Cnt <= Cnt + 1;
                2 : Cnt <= Cnt - 1;
                3 : Cnt <= Din;
                default : Cnt <= Cnt;
            endcase
    end
endmodule
```
PROBLEM 4
Structural description of a module called fib is provided below, where mux is a 32-bit wide multiplexor, adder is a 32-bit unsigned adder and register holds 32-bit D-FF values:

```verilog
// standard 32-bit wide multiplexor
module mux (IN0, IN1, select, OUT);
    input [31:0] IN0, IN1;
    input select;
    output [31:0] S;
    output [31:0] OUT;
endmodule

// 32 bit positive edge triggered reg
module register (D, clk, Q);
    input [31:0] D;
    input clk;
    output [31:0] Q;
endmodule

// unsigned adder
module adder (A, B, R);
    input [31:0] A, B;
    input clk;
    output [31:0] R;
endmodule

module fib (reset, clk, Y);
    input reset, clk;
    output [31:0] Y;
    wire [31:0] A, B, C, D, E, F;
    assign E = 1;
    assign F = 0;
    mux m1 (Y, E, reset, A);
    mux m2 (B, F, reset, C);
    register r1 (A, clk, B);
    register r2 (C, clk, D);
    adder add (B, D, Y);
endmodule
```

a) Draw the circuit diagram of fib module; use block level representation for the adder, register or multiplexer.
Sol:

[Diagram of fib module]
b) Write the behavioral description for the above fib module, including all the submodules.

Sol:

module fib (reset, clk, Y);
    input reset, clk;
    output [31:0] Y;
    reg [31:0] B, D;
    assign Y = D + B;

    always @ (posedge clk) begin
        if (reset)
            B <= 1;
        else
            B <= Y;
    end

    always @ (posedge clk) begin
        if (reset)
            D <= 0;
        else
            D <= B;
    end
endmodule
PROBLEM 5
[25 points]

The CPU of Lab 4 has the following instruction set:

<table>
<thead>
<tr>
<th>Command</th>
<th>Opcode</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
</table>
| Initialize| 0000   | 0000 | Initialize all registers: R1=R2=R3=R4=R5=0000
|           |        |      | And comparison output. |
| Move R1   | 0001   | aaaa | Put aaaa value in register R1 |
| Move R2   | 0010   | aaaa | Put aaaa value in register R2 |
| Move R3_R1| 0011   | XXXX | Take previous result in R3 and store into register R1 |
| Add       | 0100   | 0001 | Add R1 and R2 and store result in R3. Store overflow in R4. |
|           |        | 0101 | Add R5 and R2 and store result in R3. Store overflow in R4. |
| Multiply  | 0101   | XXXX | Multiply R1 and R2, put the least significant 4-bit result in R3 and MSB in R4 |
| Compare   | 0110   | 0001 | Compare the contents of R1 and R2. The flag bit is true iff R1≤R2 |
|           |        | 0101 | Compare the contents of R5 and R2. The flag bit is true iff R5≤R2 |
| Branch    | 0111   | aaaa | Jump to instruction aaaa if the compare flag bit is true |
| Divide    | 1000   | XXXX | Divide R1 and R2, put the quotient in R4 and remainder in R3 |
| Subtract  | 1001   | 0001 | Subtract R2 from R1 (R1-R2) and store result in R3. Overflow should be in R4. |
|           |        | 0101 | Subtract R2 from R5 (R5-R2) and store result in R3. Overflow should be in R4. |
| Move R3_R2| 1010   | XXXX | Take previous result in R3 and store into register R2 |
| Load R1   | 1011   | Memory address | Load memory value to R1 |
| Move R3_R5| 1100   | XXXX | Take previous result in R3 and store it in R5 |
| Store R3  | 1101   | Memory address | Store R3 to memory location. |
| Move R5   | 1110   | aaaa | Put aaaa value in register R5 |
| LDR1ID    | 1111   | XXXX | Load R1 Indirect. R1 gets memory value whose address is stored in R5 |

All module definitions (for your reference):

```
module program_counter(clk, enable, reset, load, data, q);
module instruction_decoder( inst, pc_en, r1_en, r1_sel, r2_en, r2_sel, reset, r3_en,
    comp_en, result_sel, r4_en, ldr1_en, str3_en, ram_en, ram_we,
    r5_en, ldr1id_en, r5_sel);
module regfile( clk, instant, dp_result, r1_en, r1_sel, r2_en, r2_sel, reset, r3_en, comp_in,
    comp_en, r1, r2, r3, comp_out, r4_en, r4, ldr1_en, str3_en, ram_data_in,
    ram_data_out, r5, r5_en, ldr1id_en, r5_sel);
module datapath(r1, r2, r5, op, comp, dp_out, clk, instant);
module data_RAM(ram_en, ram_clk, ram_we, data_addr, ram_data_in, ram_data_out );
module instruction_ROM(addr, inst);
```
You need to change 3 instructions as follows.

Replace the “Store R3” instruction with a “STR3ID” instruction that writes the data in R3 to a memory address obtained by summing current values in R2 and R5 (Assume the sum will not take more than 4 bits). The instruction format is:

Store R3, (R2+R5).

i.e. [R2+R5] → R3.

Replace the “LDR1ID” instruction with “LDR1ID_rel” instruction that does the following. It adds the signed immediate value to R5, takes the sum as address and loads data from that memory location to R1 (Assume the sum will not take more than 4 bits). The instruction format is:

Load R1, #( R5).

i.e. R1 ← [# + R5].

Replace the absolute branch “BR” instruction with a “relative branch (BR_rel)” instruction. For this take the signed immediate value from the [3:0] instruction.

(i) List all the module names that need to be redesigned to accommodate these changes.

Sol:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Module Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR3ID</td>
<td>cpu_top, instruction_decoder, regfile</td>
</tr>
<tr>
<td>LDR1ID_rel</td>
<td>cpu_top, instruction_decoder, regfile</td>
</tr>
<tr>
<td>BR_rel</td>
<td>program_counter</td>
</tr>
</tbody>
</table>
(ii). Write Verilog code for implementing the changes. (Write code for only those modules that are affected by these changes). This should include the module names, new (or changed) ports/wires and the new module statements implementing the changes (not the entire module).

Sol:
```
module cpu_top(cpu_clk, clear, inst_addr, instruction,r1,r2,r3,r4,r5,dp_result,comp_out);
  wire str3id_en, ldr1id_rel_en;
  assign ram_addr =  
      (str3id_en)?(r2+r5):(ldr1id_rel_en)?((instruction [3:0] + r5): (instruction[3:0]));

instruction_decoder id0(… ldr1id_rel_en(ldr1id_rel_en), .str3id_en(str3id_en));
regfile reg0(… ldr1id_rel_en(ldr1id_rel_en),str3id_en(str3id_en));
endmodule

module instruction_decoder( …ldr1id_rel_en, str3id_en);
  output str3id_en, ldr1id_rel_en;
  reg str3id_en, ldr1id_rel_en;
  parameter STR3ID =4'b1101;       //[r2 + r5] → r3
  parameter LDR1ID_rel=4'b1111;   // R1 ←[# + R5].

STR3ID: begin
  … str3id_en =1; ldr1id_rel_en=0;  //remove str3_en and ldr1id_en
end
LDR1ID: begin
  … str3id_en =0; ldr1id_rel_en=1;  //remove str3_en and ldr1id_en
end
endmodule

module regfile(… ldr1id_rel_en, str3id_en);
  input str3id_en, ldr1id_rel_en;

  if(r3_en & str3id_en) begin
    ram_data_in = r3;
  end
  if(r1_en) begin
    if (ldr1id_rel_en) begin
      r1 = ram_data_out;
    end
  end
endmodule

module program_counter(clk, enable, reset, load, data, q);
  input clk, enable, reset, load;
  input [3:0] data;
  output [4:0] q;
  reg [4:0] q;
```
always @(posedge clk) begin
  if(reset)
    q = 0;
  else if(load)
    q = q + data;
  else if(enable)
    q = q+1;
end
endmodule
PROBLEM 6
[10 points]

Design a positive edge detector using behavioral Verilog. A positive edge detector produces a single-cycle pulse each time its input goes high. (Assume you have a clock signal (clk) available).

Sol:

module l2p_conv(clk, in, out);
    input clk, in;
    output out;
    reg q,
        always @(posedge clk) begin
            q <= in;
        end
    assign out = in & (~q);
endmodule