Name ______________________

Digital Design Using Verilog

EE 4702-1

Final Examination

8 May 2000, 7:30–9:30 CDT

Problem 1	______	(20 pts)
Problem 2	______	(20 pts)
Problem 3	______	(20 pts)
Problem 4	______	(20 pts)
Problem 5	______	(20 pts)
Exam Total	______	(100 pts)

Alias ______________________

Good Luck!

Problem 1: The modules below are supposed to describe combinational logic that rearranges bits. The output of module `rearrange`, below, is a rearranged version of its input `a`; input `op` determines how the bits are rearranged. Module `rerearrange` uses two instances of `rearrange` to reverse and then left shift its inputs. Unfortunately, the modules are not quite ready for tape out because both contain errors.

Find and fix the following kinds of errors. (Points may be deducted if correct Verilog is identified as having errors.) (20 pts)

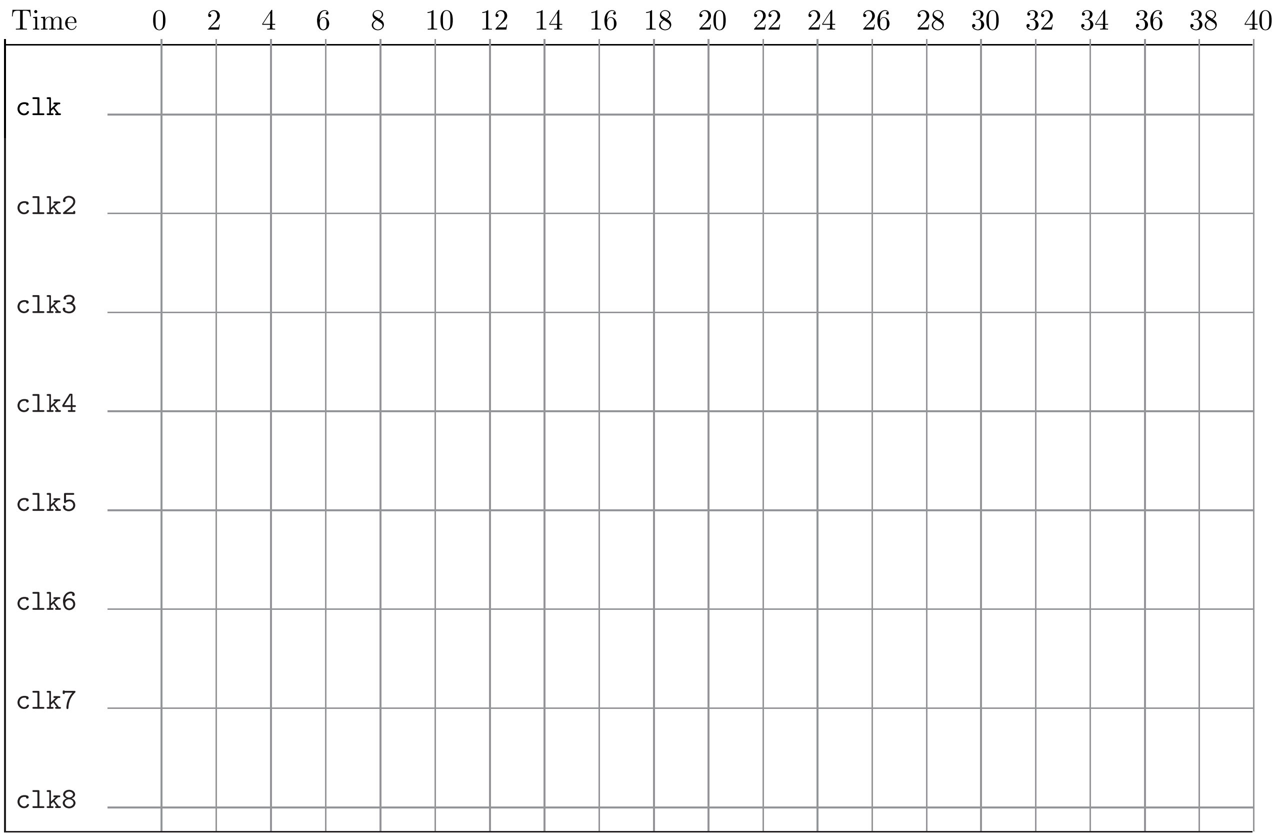
- A: One compile error. (Modelsim will not compile it.)
- B: One load error or warning. (Modelsim will compile it but will issue a warning or error message when loading it.)
- C: Three errors that result in incorrect output. The code will simulate but the output, if any, will be incorrect.

Lines with the comment `// Okay` do not have errors. None of the errors are typographical or are due to syntactic minutiæ such as missing semicolons.

```
module rerearrange(y,a);
   input a;             output y;
   wire [7:0] a;        reg [7:0]  y;      wire [0:7] temp;

   wire       operation;
   assign     operation = e1.op_reverse;
   rearrange e1(temp,a,operation);

   assign     operation = e1.op_left_shift;
   rearrange e2(y,temp,operation);
endmodule

module rearrange(x,a,op);
   input      a, op;      output     x;
   wire [7:0] a;          wire [1:0] op;
   reg [7:0]  x;          reg [2:0]  ptr, ptr_plus_one;

   parameter  op_reverse     = 0;  // Reverse order of bits.            // Okay
   parameter  op_identity    = 1;  // No change.                        // Okay
   parameter  op_left_shift  = 2;  // Circular (end-around) left shift. // Okay
   parameter  op_right_shift = 3;  // Circular (end-around) right shift.// Okay

   always @( a ) for(ptr=0; ptr<8; ptr=ptr+1) begin
        ptr_plus_one = ptr + 1;                                         // Okay
        case( op )
          op_reverse:     x[ptr]          = a[7-ptr];                   // Okay
          op_identity:    x[ptr]          = a[ptr];                     // Okay
          op_right_shift: x[ptr]          = a[ptr_plus_one];            // Okay
          op_left_shift:  x[ptr_plus_one] = a[ptr];                     // Okay
        endcase
     end
endmodule
```

Problem 2: Using the grid show the register values for the first 40 time units of execution of the module below. (20 pts)

```
module clocks();
   reg clk, clk2, clk3, clk4, clk5, clk6, clk7, clk8;
   initial begin
      clk  = 0; clk2 = 0; clk3 = 0; clk4 = 0;
      clk5 = 0; clk6 = 0; clk7 = 0; clk8 = 0;
   end

   always #8 clk = ~clk;
   always @( clk ) #4 clk2 = ~clk2;
   always @( clk ) clk3 <= #10 clk;
   always @( posedge clk ) clk4 = ~clk4;
   always #2 forever #8 clk5 = ~clk5;
   always wait( clk ) #3 clk6 = ~clk6;
   always @( clk | clk4 ) clk7 = ~clk7;
   always @( clk or clk4 ) clk8 = ~clk8;

endmodule
```



Problem 3: Draw a schematic of the hardware Leonardo will synthesize for the following Verilog code examples. These should approximate the RTL schematic, showing the hardware before optimization and technology mapping. If flip flops are used, indicate if they are level triggered or edge triggered. Otherwise, don't worry about using the precisely correct gate or symbol, as long as it's functionally correct.

(*a*) Show an approximate RTL schematic for the module below. What form is the description in? *Hint: think about what form the code is in.* (6 pts)

```
module mod_a(x,y,a,b,c);
   input a,b,c;
   output x,y;
   wire [7:0] b, c;
   reg [8:0]  x, y;

   always @( a or b or c ) begin
      if( a ) begin
         x = b + c;
         y = b - c;
      end else begin
         x = b - c;
      end
   end
endmodule
```

Problem 3, continued: (*b*) Show an approximate RTL schematic for the module below. What form is the description in? *Hint: think about what form the code is in.* (6 pts)

```
module mod_b(x,y,d,e,f,g,h);
   input d,e,f,g,h;
   output x,y;
   reg    x,y;

   always @( posedge d or negedge e or posedge f )
     if( d ) begin
        x = 0;
        y = 1;
     end else if ( f ) begin
        x = 1;
     end else begin
        if( g ) x = h;
        y = h;
     end

endmodule
```

Problem 3, continued: (*c*) Show an approximate RTL schematic for the module below. Assume that the synthesis program will not infer that this module performs magnitude comparison. Use symbols $\boxed{<}$ and $\boxed{>}$ for bit comparison. (8 pts)

```
module compare(gt, lt, a, b);
   input a, b;
   output gt, lt;
   wire [2:0] a, b;
   reg        gt, lt;
   integer    i;

   always @( a or b ) begin
      gt = 0; lt = 0;
      for(i=2; i>=0; i=i-1) if( !gt && !lt ) begin
         if( a[i] < b[i] ) lt = 1;
         if( a[i] > b[i] ) gt = 1;
      end
   end

endmodule
```

Problem 4: The incomplete code below, `compare_ism`, is for a magnitude comparison module (similar to the one in the previous problem, except it's sequential).

When input `start` is set to 1, output `valid` goes to zero and the module computes `lt` and `gt`. When `lt` and `gt` are set to their proper values `valid` is set to one. The module is to compare one bit position per cycle of input `clk`. Output `valid` should go to one as soon as possible.

Complete the module so that it is in the form of an implicit state machine, synthesizable by Leonardo. The solution can be based on the combinational module `compare`, below. Don't forget signals `start` and `valid`. (20 pts) *Hint: The solution is very similar to the combinational module.* For partial credit ignore synthesizability but follow other specifications.

```
module compare(gt, lt, a, b);    // Synthesizable combinational implementation.
   input a, b;              output gt, lt;
   wire [31:0] a, b;
   reg        gt, lt;       integer    i;

   always @( a or b ) begin
      gt = 0; lt = 0;
      for(i=31; i>=0; i=i-1) if( !gt && !lt ) begin
         if( a[i] < b[i] ) lt = 1;
         if( a[i] > b[i] ) gt = 1;
      end
   end
endmodule

// Implicit state machine implementation.
module compare_ism(gt, lt, valid, a, b, start, clk);
   input a, b, start, clk;              output gt, lt, valid;
   wire [31:0] a, b;                    reg    gt, lt, valid;
   wire        start, clk;              integer i;









         if( a[i] < b[i] ) lt = 1;  // Part of solution.
         if( a[i] > b[i] ) gt = 1;









endmodule
```

Problem 5: Answer each question below.

(*a*) Complete the module below so that it will stop simulation (using the system task $stop) if there is no change in signal **`heartbeat`** for 1000 simulator time units. There might be many changes in **`heartbeat`**, but the first time **`heartbeat`** remains unchanged for 1000 simulator time units simulation should be stopped. *Hint: use a fork. Also, the answer is short.* (5 pts)

```
module watchdog(heartbeat);
   input heartbeat;
   wire heartbeat;









endmodule // watchdog
```

(*b*) What is a critical path? At what point in the design flow can one first find out about critical paths? (5 pts) (*c*) Provide an example case statement in which the directive `exemplar case_parallel` is needed. What is its effect? (5 pts)

(*d*) The module below is supposed to zero the middle 3 bits of its input. It's rejected by the compiler (the "b=" line), identify and fix the problem. (5 pts)

```
module whatswrong(a,b);
   input a;       output b;
   wire [8:0] a;  wire [8:0] b;

   assign b = {a[8:6],0,a[2:0]};

endmodule
```