Name Solution

Digital Design Using Verilog

EE 4702-1

Final Examination

9 May 2001 7:30-9:30 CDT

Problem 1		(15 pts)
Problem 2		(18 pts)
Problem 3		(17 pts)
Problem 4		(18 pts)
Problem 5		(12 pts)
Problem 6		(20 pts)
Exam Total		(100 pts)

Alias Not Synthesizable

Good Luck!

Problem 1: The module below is in an explicit structural form.

(*a*) Re-write the module in behavioral form. The delays can be assumed to be pipeline delays. (10 pts)

(*b*) What is the difference between pipeline and inertial delays? Which kind of delay is used in your solution to the problem above? (5 pts)

In a pipeline delay of duration t units each signal change will appear t units later, regardless of other changes that occur in the interim. The delays in nonblocking delayed assignments, such as `a <= #3 b;`, are pipeline delays. In an inertial delay of duration t units a signal change (from an old to a new value) only appears if the new value does not change for t units (until the change is visible). Delays on gates and wires, such as `and #3 a1(x,a,b);`, are inertial delays.

```
module expl_str(x,y,a,b,c);
   input a, b, c;
   output x, y;
   wire   a, b, c, x, y;
   wire   na, nb, nc, t3, t5, t6;

   not n1(na,a);
   not n2(nb,b);
   not n3(nc,c);
   and #1 a1(t3,na,b,c);
   and a2(t5,a,nb,c);
   and a3(t6,a,b,nc);
   or o1(x,t3,t6);
   or #3 o2(y,a,t5);

endmodule


// Solution
module behav(x,y,a,b,c);
   input  a, b, c;
   output x, y;
   wire   a, b, c;
   reg    x, y;
   reg    t3;

   // The delays in expl_str are inertial delays, the delays here
   // are pipeline delays.

   // Note that t3 is delayed but t6 is not.

   always @( a or b or c ) t3 <= #1 !a & b & c;

   always @( a or b or c or t3 ) x = t3  a & b & !c;

   // Code below can be simplified to y <= #3 a;
   always @( a or b or c ) y <= #3 a  a & !b & c;

endmodule
```

Problem 2: The module below sets output `rot` to the number of times that input `a` must be rotated (end-around shifted) to obtain the value on input `b`, or to 32 if `a` is not a rotated version of `b`.

(*a*) Write a testbench module that tests `rots` with input pairs `a`=0,`b`=0; `a`=0,`b`=1; `a`=0,`b`=2; and `a`=0,`b`=3. (The `rot` output should be zero for the first pair and 32 for the others.) The testbench should include an integer `err` and set it to the number of incorrect outputs.

It is important that the testbench makes correct use of `ready` and `start`. (Part of the problem is determining just what is "correct use.") The testbench should use `ready` rather than assumed timing. Also, test only a single instance of `rots` and don't forget the clock. (18 pts)

```
module rots(ready, rot, start, a, b, clk);
   input a, b, start, clk;        output ready, rot;
   reg         ready;             wire [31:0] a, b;
   reg [5:0]   rot;               wire        start, clk;
   reg [31:0]  acpy;
   initial rot = 0;
   always @( posedge clk ) begin
      ready = 1;   while ( !start ) @( posedge clk );
      ready = 0;   while (  start ) @( posedge clk );
      rot = 0;  acpy = a;
      while (  acpy != b  &&  rot < 32  ) @( posedge clk ) begin
         acpy = { acpy[30:0], acpy[31] };
         if ( acpy == a ) rot = 32; else rot = rot + 1;
      end
   end
endmodule
```

```
module testrot();
   reg [31:0] b;               wire       rdy;
   reg        start, clk;      wire [5:0] r;
   integer    i, err;

   rots myrots(rdy, r, start, 32d'0, b, clk);

   always #1 clk = !clk;

   initial begin
      err = 0;  start = 0;  clk = 0;
      wait(rdy);
      for (i=0; i<4; i=i+1) begin
         b = i;
         start = 1; wait(!rdy);
         start = 0; wait( rdy);
         if ( !b && r ) err = err + 1;
         if (  b && r != 32 ) err = err + 1;
      end
      $display("Error count: %d",err); $stop;
   end
endmodule
```

Problem 3: Convert the `rots` module (repeated below) to synthesizable Form 2 (edge-triggered flip-flops). Do not change the ports or what it does. In particular, `ready` and `start` must be used the same way. Ignore reset. (17 pts)

```
module rots(ready, rot, start, a, b, clk);
   input a, b, start, clk;        output ready, rot;
   reg         ready;             wire [31:0] a, b;
   reg [5:0]   rot;               wire        start, clk;
   reg [31:0]  acpy;
   initial rot = 0;
   always @( posedge clk ) begin
      ready = 1;   while ( !start ) @( posedge clk );
      ready = 0;   while (  start ) @( posedge clk );
      rot = 0;  acpy = a;
      while (  acpy != b  &&  rot < 32  ) @( posedge clk ) begin
         acpy = { acpy[30:0], acpy[31] };
         if ( acpy == a ) rot = 32; else rot = rot + 1;
      end
   end
endmodule
```

Solution on next page.

```
module rots(ready, rot, start, a, b, clk);
   input a, b, start, clk;
   output ready, rot;  // Don't forget port types and other declarations.









         acpy = { acpy[30:0], acpy[31] };
         if ( acpy == a ) rot = 32; else rot = rot + 1;





endmodule
```

```
 /// Solution 1, using named states and assuming little about start.

module rots(ready, rot, start, a, b, clk);
   input a, b, start, clk;
   output ready, rot;

   wire [31:0] a, b;
   reg [5:0]   rot;
   wire        start, clk;
   reg [31:0]  acpy;
   reg [1:0]   state;

   parameter   st_ready = 2'b01;
   parameter   st_wait  = 2'b00;
   parameter   st_go    = 2'b10;

   wire        ready = state[0];

   initial begin rot = 0; state = st_ready; end

   always @( posedge clk )
     case ( state )
       st_ready:
         if ( start ) state = st_wait;
       st_wait:
         if ( !start ) begin  rot = 0;  acpy = a;  state = st_go;  end
       st_go:
         if (  acpy != b  &&  rot < 32  ) begin
            acpy = { acpy[30:0], acpy[31] };
            if ( acpy == a ) begin rot = 32; state = st_ready; end
            else rot = rot + 1;
         end else begin
            state = st_ready;
         end
     endcase
endmodule
```

```
 /// Solution 2, basing state on ready and assumed behavior of start.

module rots(ready, rot, start, a, b, clk);
   input a, b, start, clk;
   output ready, rot;

   reg         ready;
   wire [31:0] a, b;
   reg [5:0]   rot;
   wire        start, clk;
   reg [31:0]  acpy;

   initial begin rot = 0; ready = 1; end

   always @( posedge clk )
     case ( {ready,start} )
       {2'b10}:; // Wait for start to go to one.
       // Unlike original module, gets value of "a" when start goes
       // to 1, not when start goes to zero. (This is where behavior assumed.)
       {2'b11}: begin ready = 0; acpy = a; rot = 0; end
       {2'b01}:; // Wait for start to go to zero.
       {2'b00}:
         if (  acpy != b  &&  rot < 32  ) begin
            acpy = { acpy[30:0], acpy[31] };
            if ( acpy == a ) begin rot = 32;  ready = 1; end
            else rot = rot + 1;
         end else begin
            ready = 1;
         end
     endcase
endmodule
```

Problem 4: Two synthesizable descriptions appear below.

(*a*) In what synthesizable form is the Verilog description below? (2 pts)

Form 1: combinational logic, level triggered.

(*b*) Draw a schematic showing the approximate RTL-level description generated by a synthesis program like Leonardo. (7 pts)

```
module whatsyna(x, y, z, a, b, op);
   input a, b, op;
   output x, y, z;
   wire [7:0] a, b;
   wire [1:0] op;
   reg [7:0]  x, y, z;

   always @( op or a or b ) begin

      if ( a == 0 ) y = b;

      if ( a < b ) z = a; else z = b;

      case ( op )
        0: x = a + b;
        1: x = a;
        2: x = b;
      endcase

   end

endmodule
```

If you're an LSU ECE student in a Verilog-related course ask for a complete solution. For now: Output `y` is connected to a level-triggered flip-flop with enable input `a==0` and data input `b`.

Output `z` is connected to a two-input mux, controlled by `a < b`.

Output `x` is connected to a level triggered flip-flop enabled by `op != 3` . The data input is a mux controlled by `op`.

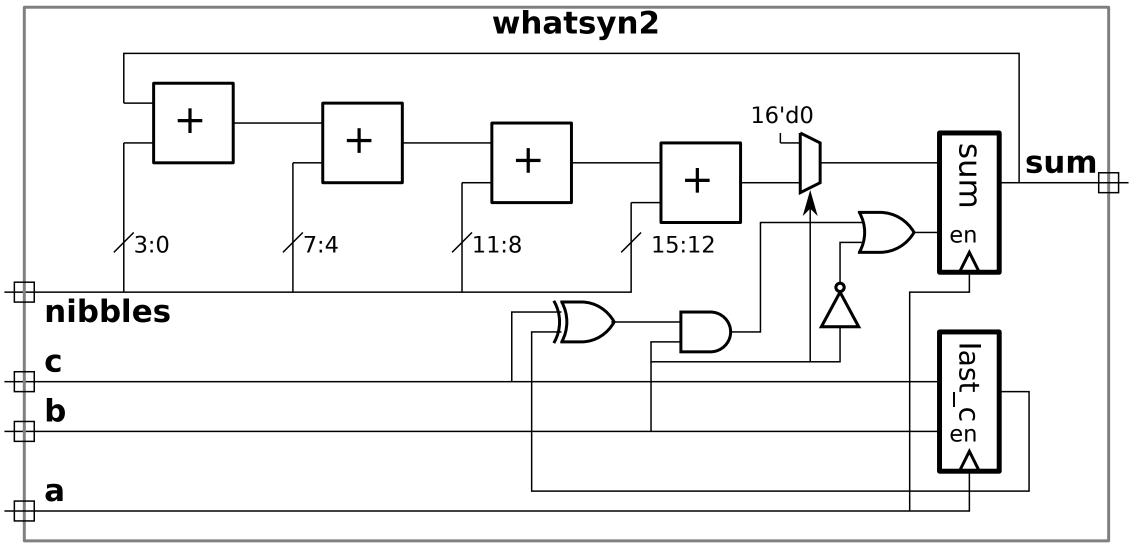
Problem 4, continued:

(*c*) (2 pts) In what synthesizable form is the Verilog description below? Form 2: Edge triggered logic.

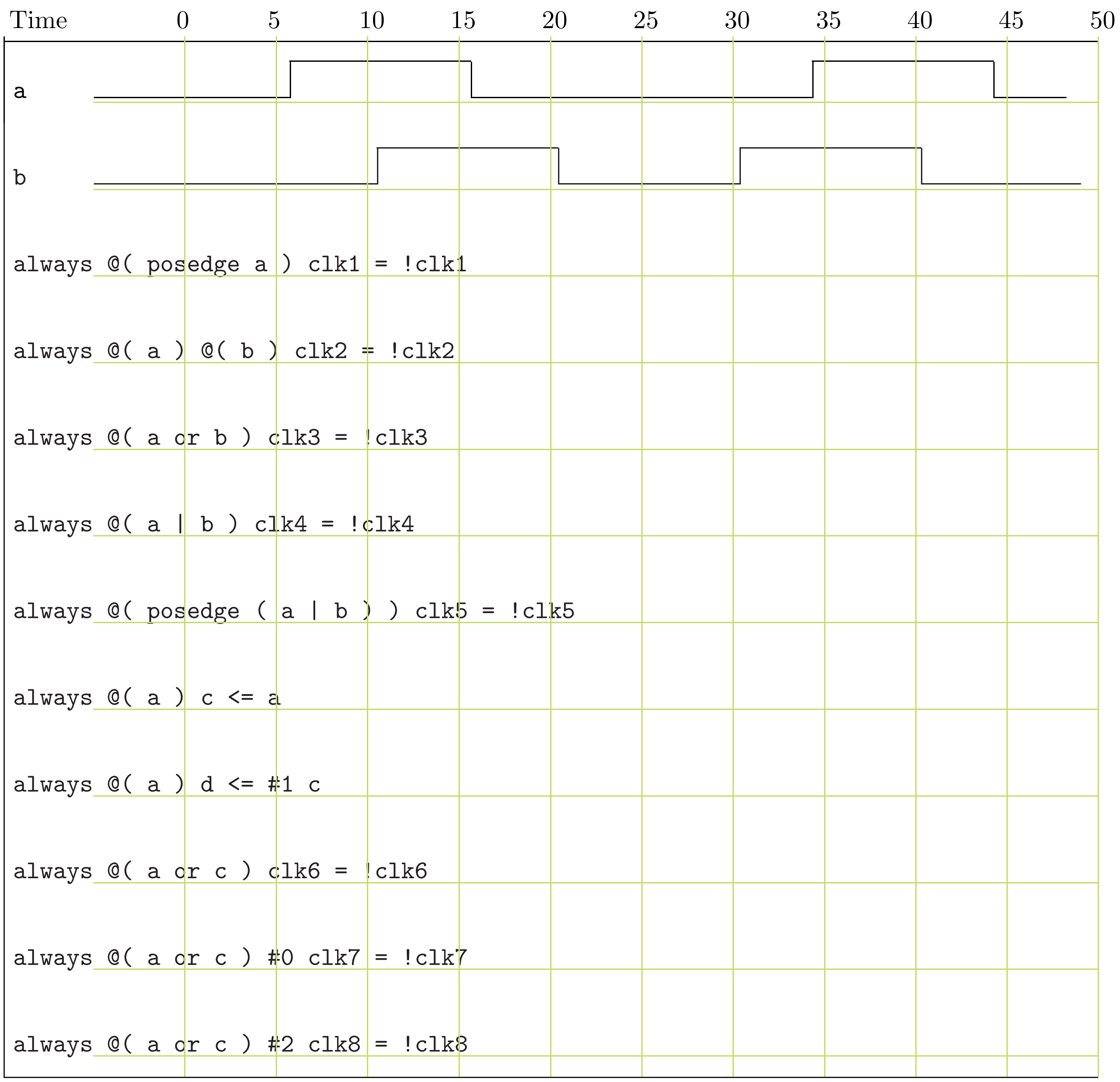
(*d*) (7 pts) Draw a schematic showing the approximate RTL-level description generated by a synthesis program like Leonardo. *Grading Note: In the 2001 version the event control was* `posedge a or negedge b`.

```
module whatsyn2(output [6:0] sum, input [15:0] nibbles, a, b, c);
   logic [15:0] n2;
   logic        last_c;

   always @( posedge a )
     if ( !b ) begin
        sum = 0;
     end else begin

        if ( c != last_c ) begin
           n2 = nibbles;
           for ( int i=0; i < 4; i++ ) begin
              sum = sum + n2[3:0];
              n2 = n2 >> 4;
           end
        end
        last_c = c;

     end
endmodule
```

Solution appears below. Output `sum` is driven by an edge-triggered register clocked by `a`. The nibbles input is connected to a cascade of four adders, the first adder connected to `sum`, the others each connected to a different four bits of `nibbles`. Note that no logic is synthesized for the shift operator, that only determines bit numbers for the adder inputs. The `sum` register is reset by `!b` and enabled by `b` and `c` $\oplus$ `last_c`.



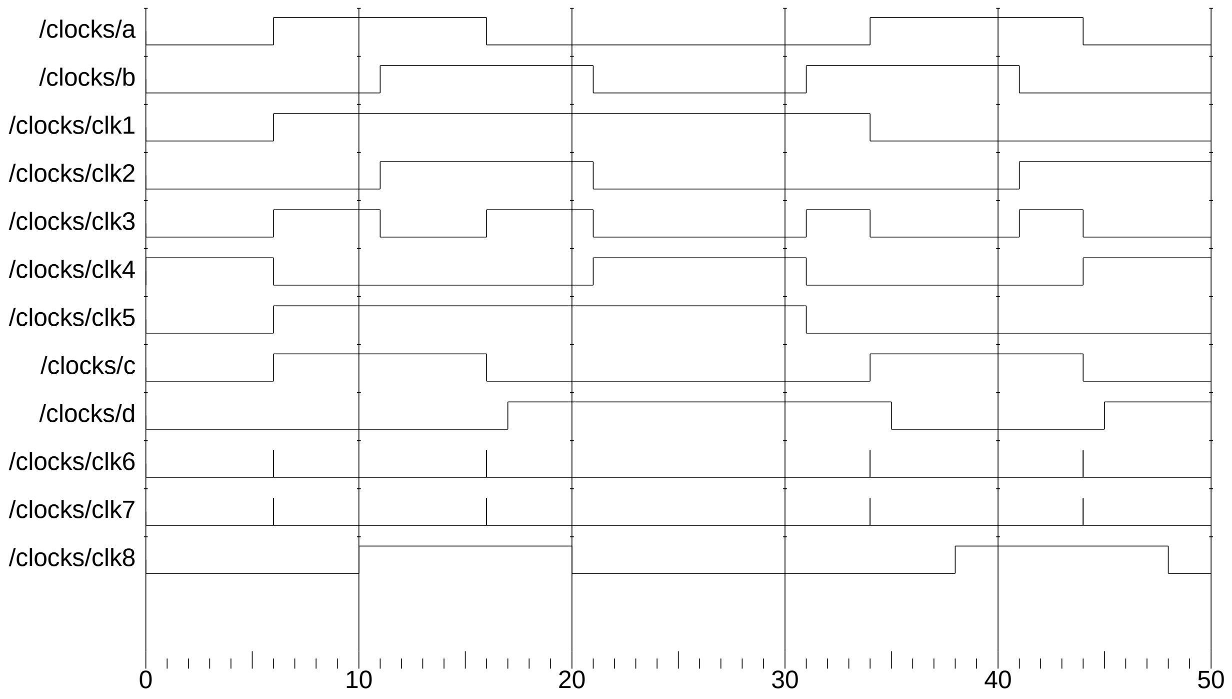
Problem 5: In the diagram below `c`, `d`, and identifiers starting with `clk` are all initialized to zero. Complete the timing diagram. (12 pts)



Time
0
5
10
15
20
25
30
35
40
45
50
a
b
always @(posedge a) clk1 = !clk1
always @(a) @(b) clk2 = !clk2
always @(a or b) clk3 = !clk3
always @(a | b) clk4 = !clk4
always @(posedge (a | b)) clk5 = !clk5
always @(a) c <= a
always @(a) d <= #1 c
always @(a or c) clk6 = !clk6
always @(a or c) #0 clk7 = !clk7
always @(a or c) #2 clk8 = !clk8

Solution on next page.

Solution to Problem 5:



Problem 6: Answer each question below.

(*a*) The code below, based on the Homework 3 solution, simulates properly before synthesis but in the post-synthesis simulation the testbench reports an incorrect beep time.

What goes wrong? Fix the problem without modifying the code below the indicated line. *Hint: The beep can start (and stop) at a slightly different time than the code below.* (5 pts)

```
module beepprob(beep, clk);
   input clk;
   output beep;

   // Code from exam: assign beep = | beep_timer;
   // Solution: Set beep on negative edge, after beep_timer computed.
   reg       beep;
   always @( negedge clk )  beep =  beep_timer;


   // DO NOT MODIFY CODE BELOW THIS LINE.
   always @( posedge clk ) begin
      // Lots of stuff;

      if ( beep_timer ) beep_timer = beep_timer - 1;

   end

endmodule
```

(*b*) Describe something that a parameter can be used for that an ordinary input port cannot and something that an input port can be used for that a parameter cannot. (5 pts)

Of course, the two are completely different things. Parameters can be used to set the size of vectors, an input value could not do that. An input can change, parameters are constant.

(*c*) What is the difference between `case`, `casex`, and `casez`? (5 pts)

In a `case` statement there must be a bitwise match, including unknowns and high impedance values, between the case expression and a case item. In a `casex` statement an unknown value acts as a wildcard matching any bit in the corresponding position, `casez` is similar with high impedance acting as the wildcard.

(*d*) Explain how each of the three statements below behave differently with unknown values. In particular, explain what has to be unknown and how the results of each statement is different. (5 pts)

```
m1 = a > b ? c : d;

if ( a > b ) m2 = c; else m2 = d;

case ( a > b )
  1: m3 = c;
  default: m3 = d;
endcase
```

The three behave identically if `a > b` is not unknown. If it is unknown the `m1` statement assigns a bitwise combination of `c` and `d`. (For the bit positions where `c` and `d` hold the same value `m1` is set to that value, in positions where `c` and `d` differ the corresponding position `m1` is set to unknown.)

If `a > b` is unknown `d` is assigned to `m2` and `m3`.