Digital Design Using Verilog
EE 4702-1
Midterm Examination
5 April 2000  8:40-9:30 CDT

Problem 1 (40 pts)
Problem 2 (60 pts)
Exam Total (100 pts)

Good Luck!
Problem 1: Complete the Verilog description (below) of a FIFO-like module which has a 3-bit data input, \texttt{in}; a 7-bit output, \texttt{out}; 1-bit inputs \texttt{inclk} and \texttt{outclk}; and 1-bit outputs \texttt{full} and \texttt{empty}. The module operates like a FIFO (first in, first out) except that the width of the data input and output ports are different: it reads data 3 bits at a time (on a positive edge of \texttt{inclk}) and outputs 7 bits at a time (consisting of data from two input words plus one bit of a third). Unless the module has less than 3 bits of space left, on a positive edge of \texttt{inclk} the value on \texttt{in} is stored. The oldest 7 bits stored by the module always appear on output \texttt{out}. On a positive edge of \texttt{outclk} the oldest 7 bits are removed and the output displays the next 7 bits. Output \texttt{full} is 1 if the module cannot accept another 3 bits of input and is 0 otherwise; output \texttt{empty} is 1 if the module is empty and is 0 otherwise. Parameter \texttt{storage} is the total number of bits stored by the module. An example of the module operating is shown in the timing diagram below. (40 pts)

\begin{verbatim}
module width_change(out,full,empty,outclk,in,inclk);
    input outclk, in, inclk;
    output out, full, empty;

    parameter storage = 20;

    wire [6:0] out; // Can change to reg for solution.
    wire [2:0] in;
    wire inclk, outclk;
    wire full, empty; // Can change to reg for solution.

    reg [storage-1:0] sto; // Storage for data.
    integer amt; // Number of occupied bits in sto.

    // USE THE NEXT PAGE FOR THE SOLUTION.

endmodule // width_change
\end{verbatim}
module width_change(out,full,empty,outclk,in,inclk);
  input outclk, in, inclk;
  output out, full, empty;

  parameter storage = 20;

  wire [6:0] out; // Can change to reg for solution.
  wire [2:0] in;
  wire inclk, outclk;
  wire full, empty; // Can change to reg for solution. // Can change to reg for solution.

  reg [storage-1:0] sto; // Storage for data.
  integer amt; // Number of occupied bits in sto.

  // Solution goes here.
  initial begin amt = 0; sto = 0; end

  assign full = amt + 3 > storage;
  assign empty = amt == 0;
  assign out = sto[7-1:0];

  always @( posedge outclk )
    if( amt >= 7 ) begin
      sto = sto >> 7;
      amt =amt - 7;
    end

  always @( posedge inclk )
    if( !full ) begin
      sto = sto | in << amo;
      amo = amo + 3;
    end

endmodule // width_change
Problem 2: Answer each question below.

(a) Describe something that a function can do (or be used for) that a task cannot. Describe something that a task can do (or be used for) that a function cannot. (10 pts)

A function can be used in an expression (but a task cannot). A task can include delays, but a function cannot.

(b) Convert the following behavioral code to explicit structural code. (10 pts)

module btos(x, a, b);
    input a, b;
    output x;
    wire a, b;
    reg x;

    always @( a or b ) if( a ) x = b; else x = ~b;
endmodule // btos

If you don’t see the logical function performed, draw a truth table. The function, \( x = \overline{a} \oplus b \), can be performed by a primitive gate (xnor), a solution consisting of several other gates realizing the same function would also receive full credit.

module explicit(x, a, b);
    input a, b;
    output x;
    wire a, b;
    wire x; // Wire, not reg.

    xnor (x,a,b);
endmodule
(c) Show the changes (values and times) to a and b in the module below. (10 pts)

module assig();
  reg [15:0] a, b;
  initial
    begin
      a = 1;
      b = 2;
      #1;
      a <= b;
      b <= a;
      #1;
      a <= b + 10;
      b <= #5 b + 20;
      #1;
      b = #1 3;
      b <= 4;
      b <= #2 5;
      b <= #10 6;
      b = 7;
      #20;
    end
endmodule

Note that b = #1 3; is a blocking assignment. The condition is evaluated immediately (since it's 3 here evaluation time doesn't matter) and the assignment is done after the delay. Following statements are executed after the assignment. The solution is plotted below.
(d) Show the changes (values and times) to x in the module below using the timing diagram provided. (10 pts)

module events1();
    wire a, b, c, d;
    reg [2:0] x;
    reg [3:0] i;
    assign {d,c,b,a} = i;
    initial begin
        i = 0;
        forever #10 i = i + 1;
    end
    always begin
        #15;
        @( a );
        x = 1;
        @( posedge a ) x = 2;
        @( a or b ) x = 3;
        @( a | b | c | d ) x = 4;
        wait( a | b ) x = 5;
        wait( a ) x = 6;
        wait( ~a ) x = 7;
    end // always begin
endmodule // events1

An event control, @( foo ), delays execution until foo changes. A wait statement, wait( foo ), delays execution until foo is nonzero (or true).

The solution appears below.
(e) Show the changes (values and times) to \texttt{aa} in the module below. (10 pts)

\begin{verbatim}
module d();
    reg a;
    wire aa;

    and #(2,3) (aa,a,1);

    initial begin
        a = 0;
        # 10;
        a = 1;
        # 10;
        a = 0;
        # 10;
        a = 1;
        # 1;
        a = 0;
        # 10;
    end
endmodule // d
\end{verbatim}

Solution:
(ʃ) Complete module **after** so that it does the same thing as **before**. All procedural code in module **after** must go in the one initial process. The solution must use **fork** and **join**. Structural code **cannot** be added. (10 pts)

```verilog
module before(asum,bsum,out,a,ainp,b,binp,c);
  output asum, bsum, out;
  input  a, ainp, b, binp, c;

  reg [9:0] asum, bsum, out;
  wire [9:0] ainp, binp;
  wire    a,b,c;

  always @( a ) asum = asum + ainp;
  always @( b ) bsum = bsum + binp;
  always @( posedge c ) out = asum + bsum;
endmodule

module after(asum,bsum,out,a,ainp,b,binp,c);
  output asum, bsum, out;
  input  a, ainp, b, binp, c;

  reg [9:0] asum, bsum, out;
  wire [9:0] ainp, binp;
  wire    a,b,c;

  // ALL code must go in the initial process below.
  initial begin

  // Solution:
  fork
    forever @( a ) asum = asum + ainp;
    forever @( b ) bsum = bsum + binp;
    forever @( posedge c ) out = asum + bsum;
  join

  end // initial
endmodule
```