Name Solution

Computer Organization
EE 3755
Practice Midterm Examination

23 October 2001

Verilog solution code in `http://www.ece.lsu.edu/ee3755/2001f/mtp.html`

Problem 1		(10 pts)
Problem 2		(30 pts)
Problem 3		(10 pts)
Problem 4		(10 pts)
Problem 5		(10 pts)
Problem 6		(10 pts)
Problem 7		(10 pts)
Problem 8		(10 pts)
Exam Total		(100 pts)

Alias always @(posedge)

Good Luck!

Problem 1: Add Verilog code to the module below for the `carry` signals and `sum[3]` using the generate and propagate signals. *Hint: This is straight from the notes.* (10 pts)

```
 // Solution
module cla_3sol(sum,a,b);
   input [2:0] a, b;
   output [3:0] sum;

   wire [2:0]   g, p, carry;

   assign       carry[0] = 1'b0;
   assign       carry[1] = g[0];
   assign       carry[2] = g[0] & p[1] | g[1];
   assign       sum[3] = g[0] & p[1] & p[2] | g[1] & p[2] | g[2];

   cla_slice s0(sum[0],g[0],p[0],a[0],b[0],carry[0]);
   cla_slice s1(sum[1],g[1],p[1],a[1],b[1],carry[1]);
   cla_slice s2(sum[2],g[2],p[2],a[2],b[2],carry[2]);

endmodule
```

Problem 2: Complete the module below so that it determines whether its input, a floating point number in IEEE 754 single format, is positive, zero, negative, and whether it is an integer. Output `pos` is `1` if the input is positive, `neg` is `1` if it's negative, etc. The solution can ignore special values ($\pm\infty$, NaN, subnormals, etc.) (30 pts)

```
// Solution
module fp_flags(pos,zero,neg,int,single);
   input [31:0] single;
   output       pos, zero, neg, int;

   reg          int;
   reg          sign;
   reg [7:0]    exp;
   reg [22:0]   frac;

   reg          zero, pos, neg;

   reg [5:0] loc;
   integer   i;
   reg       found;

   always @( single ) begin

      sign = single[31];
      exp = single[30:23];
      frac = single[22:0];

      zero = !single[30:0];
      pos = !sign && !zero;
      neg = sign && !zero;

      found = 0;
      for(i=0; i<23; i=i+1) if( !found && frac[i] ) begin loc = i; found = 1; end
      if( !found ) loc = 23;

      int = zero || loc + exp >= 150;

   end

endmodule
```

Problem 3: The `for` loop in the code below looks harmless but is actually an infinite loop. Why? *Hint: It has to do with the way* `i` *is declared.* (10 pts)

Because `i` is only five bits and so it's impossible to represent 32. When `i` is 31 the statement `i=i+1` will set `i` to zero (the overflow bit is ignored). Since 32 can't be represented `i<32` is always true and so there's no way out of the `for` loop.

```
module iloop(z,a);
   input [31:0] a;
   output       z;

   reg [4:0] i;
   reg       s, z;

   initial begin
      s = 0;
      for(i=0; i<32; i=i+1) s = s | a[i];
      z = !s;
   end

endmodule
```

Problem 4: Consider the adder modules below.(10 pts)

(*a*) What kind of adders are these? Ripple adders.

(*b*) How do the speed of the two adders compare? The first one is much faster since it computes its sum in one cycle. The second one requires 32 cycles.

(*c*) Compare the amount of hardware that the adders will synthesize into. How is the second adder penny wise and £ foolish?

Because of the loop, the first adder will be synthesized with 32 binary full adders (connected to form a ripple adder). The second will consist only of a single binary full adder; at each cycle a different bit of the input is directed into the BFA using a multiplexor for `a` and `b`. The cost of the multiplexors will rival, if not exceed the cost of the BFAs so the cost of the two modules is comparable.

```
module add_1(sum,a,b,clk);
   input [31:0] a,b;   input        clk;   output       sum;
   reg [31:0]   sum;   integer      i;     reg          carry;

   always @( posedge clk )
     begin
        carry = 0;

        for(i=0; i<31; i=i+1) begin

           sum[i] = ~a[i] & ~b[i] &  carry |
                    ~a[i] &  b[i] & ~carry |
                     a[i] & ~b[i] & ~carry |
                     a[i] &  b[i] &  carry;

           carry = a[i] & b[i] | b[i] & carry | a[i] & carry;

        end
   end

endmodule

module add_2(sum,a,b,clk);
   input [31:0] a,b;   input        clk;   output       sum;
   reg [31:0]   sum;   integer      i;     reg          carry;

   always @( posedge clk )
     begin
        i = i + 1;
        if( i == 32 ) begin carry = 0; i = 0; end

        sum[i] = ~a[i] & ~b[i] &  carry |
                 ~a[i] &  b[i] & ~carry |
                  a[i] & ~b[i] & ~carry |
                  a[i] &  b[i] &  carry;
```

```
        carry = a[i] & b[i] | b[i] & carry | a[i] & carry;
    end

endmodule
```

Problem 5: Consider the module below. (10 pts)

```
module prefix_xor_4(x,a);
   input [3:0] a;
   output [3:0] x;

   assign       x[0] = a[0];

   xor x1(x[1],a[0],a[1]);
   xor x2(x[2],x[1],a[2]);
   xor x3(x[3],x[2],a[3]);

endmodule
```

(*a*) Suppose that each gate has a delay of one unit. How long would it take to compute the result?

Three units. (The critical path is: `a[0]->x[1]->x[2]->x[3]`, which goes through all three XOR gates.)

(*b*) Suppose during a run of the simulator on the code above new inputs arrived at $t = 100$. At what simulated time would the results be available? *Hint: The first part is intentionally misleading.*

At $t = 100$. There are no delays so simulated time is not advanced.

(*c*) How would timing obtained after synthesis relate to the times used to solve the first two parts?

If the code were backannotated with timing information, then the timing would be based on models of actual components chosen by the synthesis program. If would be greater than zero, because the time for the first part was in unspecified "units" there is no way to compare it to an actual time obtained in a post-synthesis run.

Problem 6: In the module below fill in the values for c, whether the corresponding addition overflowed, and fix the last assignment. (10 pts)

```
// Solution
module sums();

   reg [3:0] a, b, c;
   reg [5:0] d;

   initial begin

      a = 4'b0101;  b = 4'b0001;  c = a + b;

      // Unsigned decimal: c = 6          Overflow? No

      // Signed decimal:   c = 6          Overflow? No


      a = -6; b = 4'b0001;  c = a + b;

      // Unsigned decimal: c = 11         Overflow? No

      // Signed decimal:   c = -5         Overflow? No


      a = 4'b1101; b = 4'b1100;  c = a + b;

      // Unsigned decimal: c =  9         Overflow? Yes (finally)

      // Signed decimal:   c =  -7        Overflow? No


      // Suppose c and d are used for signed quantities.
      // Fix the assignment below.
      // Original assignment: d = c;
      d = {c[3],c[3],c};  // Fixed assignment with sign extension.

   end

endmodule
```

Problem 7: Convert the module below to an explicit structural form. (10 pts)

```
module to_str(x,s,a,b);
   input [1:0] s;
   input       a, b;
   output      x;

   assign x = s == 2 ? a : b;

endmodule
```

```
// Solution
module is_structural(x,s,a,b);
   input [1:0] s;
   input       a, b;
   output      x;

   wire        a_path, b_path;
   wire        s_eq_2, s_ne_2;
   wire        not_s_0;

   or o1(x,a_path,b_path);

   and a1(s_eq_2,s[1],not_s_0);
   not n2(s_ne_2,s_eq_2);

   not n1(not_s_0,s[0]);

   and a2(a_path,s_eq_2,a);
   and a3(b_path,s_ne_2,b);

endmodule
```

Problem 8: Show the longhand steps needed to multiply $00100111_2 \times 00100111_2$ using radix-4 Booth recoding. (10 pts)

```
 -00100111 = 11011001

      00100111 = 39
    | 00100111 = 39
      ^^^^^^^^
11111111011001
000001001110
1110110010
00100111
^^^^^^^^^^^^^
00010111110001 = 1521
```