Name ______________________________

Digital Design using HDLs
EE 4755
Final Examination

Monday, 8 December 2014   10:00-12:00 CST

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: [20 pts] The encode module below, based on Homework 4, is used to convert a decimal value to binary one ASCII digit at a time. Input val_prev is the binary value so far, and output val_next is the binary value after using ASCII character ascii_char. If ascii_char isn't a numeric digit non_digit is set to 1 and val_next is set to zero. There is also an overflow output.

```
module encode
  #( int width = 32 )
   (output logic [width-1:0] val_next,
    output logic overflow,                output uwire non_digit,
    input uwire [7:0] ascii_char,         input uwire [width-1:0] val_prev);

   logic [width+3:0] val_curr;            logic [3:0] high_bits, bin_char;

   assign non_digit = ascii_char < Char_0 || ascii_char > Char_9;

   always_comb begin
      bin_char = ascii_char - Char_0;
      val_curr = 10 * val_prev + bin_char;
      high_bits = val_curr >> width;
      if ( non_digit ) begin    overflow = 0; val_next = 0;       end
      else              begin
        overflow = high_bits != 0;
        val_next = val_curr;
      end
   end
endmodule
```

(*a*) Show the hardware that will be synthesized for this module. Take into account optimizations (see the next subproblem).

☐ Synthesized hardware.

(*b*) Indicate how many units such as adders, multipliers, shifters, and multiplexors will actually be present in the optimized hardware. The count should be based on the units that are present after optimization, not on the hardware first inferred from the Verilog.

☐ Number of adders. ☐ Number of multipliers. ☐ Number of shifters. ☐ Number of multiplexors.

Problem 2: [20 pts] Appearing below is another encode module, this one has a new input `radix`, which indicates the radix (base) of the number to be converted. When completed the module should function like the module from the previous problem, except that the digits form a radix-`radix` number. For example, if `radix` were 10 it would operate like the previous module. If `radix` were 8 the digits would be octal, etc.

(*a*) Modify the module so that it takes into account the radix. Assume that `radix` can be any value from 2 to 16. Note that for a radix of 16 the valid digits are 0-9 and A-F (only consider upper case).

☐ Modify the module to generate the correct `non_digit` output.

☐ Modify the module to update `val_next` correctly given the radix.

```
typedef enum {Char_0 = 48, Char_9 = 57, Char_A = 65, Char_F = 70} Chars;
module encode_radix #( int width = 32 )
   (output logic [width-1:0] val_next,
    output logic overflow,                     output uwire non_digit,
    input uwire [7:0] ascii_char,              input uwire [width-1:0] val_prev,
    input uwire [4:0] radix);

   logic [width+3:0] val_curr;

   logic [3:0] high_bits, bin_char;






















   always_comb begin

      val_curr =

      high_bits = val_curr >> width;
      if ( non_digit ) begin
         overflow = 0;
         val_next = 0;
      end else begin
         overflow = high_bits != 0;
         val_next = val_curr;
      end
   end
endmodule
```

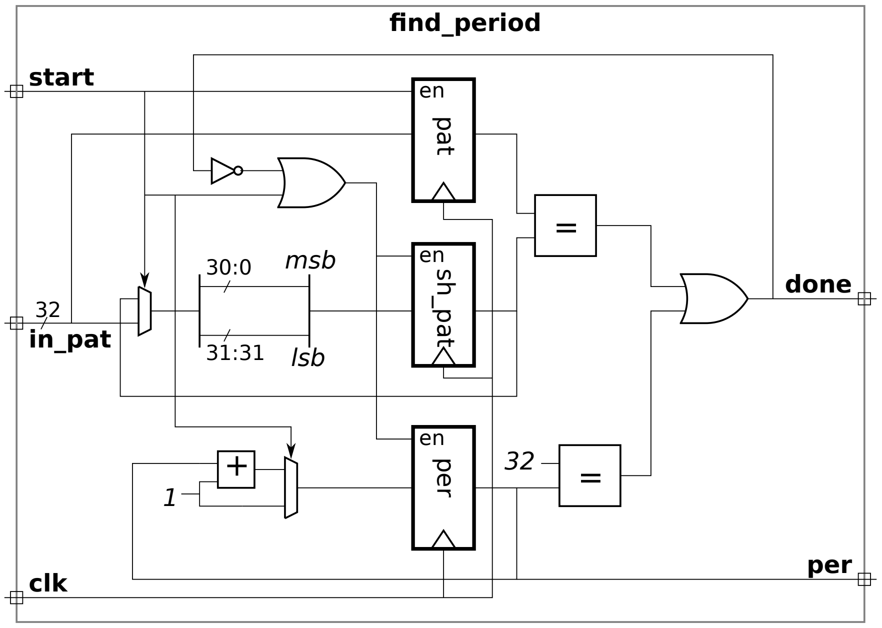
Problem 2, continued:

(*b*) Suppose that module `encode_radix` (from the previous part) were to be used in a larger design in which the values of `radix` could only be 2, 8, 10, and 16. Also suppose that the synthesis program can't figure out that `radix` is limited to these values. Why would the cost be higher than necessary, and how could `encode_radix` be modified to get the lower cost hardware?

☐ Explain why the cost will be higher than is necessary.

☐ Show the changes to `encode_radix` so that the synthesis program *will* generate the lower cost design. The port definitions **cannot** be changed.

Problem 3: [20 pts] Appearing to the right is hardware and a corresponding Verilog module. The module is incomplete, finish it. *Hint: The hardware includes an end-around shift, that's the part with the msb/lsb labels.*



☐ Add sizes and other information to port declarations.

☐ Finish the Verilog code.

```
module find_period
   (output                 per,       output                   done,

    input                  in_pat,

    input                  start,     input clk);
```

```
endmodule
```

Problem 4: [20 pts] The Verilog below is the key lookup part of the simple CAM module used in class.

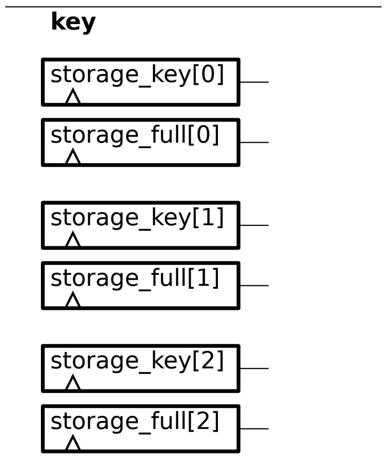
```
logic [dwid:1] storage_data [ssize];
logic [kwid:1] storage_key [ssize];
logic [ssize-1:0] storage_full;

always_comb begin
   mmatch = 0;   midx = 0;
   for ( int i=0; i<ssize; i++ )
     if ( storage_full[i] && storage_key[i] == key ) begin mmatch = 1; midx = i; end
end

assign out_data = storage_data[midx];
```

(*a*) Starting with the registers and `key` shown below, sketch the hardware synthesized for this code without optimization. The hardware should produce values for `mmatch` and `midx` (but not `out_data`). Do so for `ssize=3`. In class we often showed part of this as a box labeled "priority encoder" (or "pri" for short), in this problem actually show the hardware.

☐ Synthesized hardware for `ssize = 3` to generate ☐ `mmatch` and ☐ `midx`.



(*b*) Assume that the cost of an a-bit comparison unit is a, and its delay is also a. Assume that the cost of an a-input, b-bit multiplexor is ab and the delay is 1. Compute the cost and delay of the logic used to compute `midx` in terms of `ssize` (use s in your formulas) and `kwid` (use k in your formulas). As with the previous part, do this for the unoptimized hardware. Remember to solve this for an arbitrary value of `ssize` (s), not for $s = 3$.

☐ Cost in terms of s and k:

☐ Delay in terms of s and k:

Problem 4, continued: Appearing below is a variation on the key lookup from the CAM module. Instead of finding a matching key it finds the largest stored key that is $\leq$ to the lookup key. Note that this version doesn't include `storage_full`.

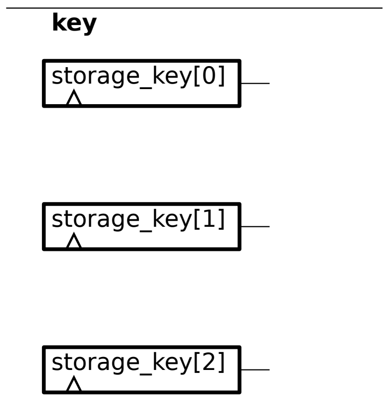
```
logic [dwid:1] storage_data [ssize];
logic [kwid:1] storage_key [ssize];

always_comb begin
   midx = 0;  bkey = 0;
   for ( int i=0; i<ssize; i++ )
     if ( storage_key[i] >= bkey && storage_key[i] <= key ) // READ THIS LINE CAREFULLY
       begin   midx = i;  bkey = storage_key[i];   end
end

assign out_data = storage_data[midx];
```

(c) Sketch the hardware for `ssize=3`.

☐ Sketch the synthesized hardware needed to generate `bkey`.



(d) Compute the cost and performance in terms of `ssize` (use s) and the key size (use k). As before a k-bit comparison unit (equality or magnitude) costs k and has a delay of k and an a-input, b-bit mux costs ab and has a delay of 1. *Hint: There's a big difference.*

☐ Cost in terms of s and k:

☐ Delay in terms of s and k:

Problem 5: [20 pts] Answer each question below.

(*a*) The module below is supposed to count from 0 to `max` (inclusive), then return to zero. Strictly speaking it does, but there are problems, including the fact that it's not synthesizable. Fix the problems.

```
module counter #(int max = 3)(output logic [7:0] count, input uwire clk);

   always @( posedge clk ) begin

      count <= count + 1;

   end

   always @* begin

      if ( count == max ) count <= 0;

   end

endmodule
```

☐ Why isn't the module synthesizable?

☐ Fix the problem.

(*b*) There is a problem with the module below due to the way that `a` is declared.

```
module sa1(output uwire a, input uwire c, d);

   always_comb begin

      a = c & d;

   end

endmodule
```

☐ Fix the problem **by changing** the declaration of `a`.

☐ Fix the problem **without changing** the declaration of `a`.

(*c*) Describe a situation in which using `always_comb` has a benefit over using `always @*`.

☐ Situation where `always_comb` helps.

(*d*) The module below is supposed to be computing $x^2 + y^2$.

```
module sa2(output logic [63:0] sos, input uwire [63:0] x, y);

   logic [63:0] a1, b1, a2, b2;
   uwire [63:0]  p, s;

   fpmul f1(p,a1,b1);
   fpadd f2(s,a2,b2);

   always @* begin

      // Compute x^2.
      a1 = x;  b1 = x;
      #1;
      sos = p;

      // Compute y^2.
      a1 = y;  b1 = y;
      #1;

      // Compute x^2 + y^2.
      a2 = p;  b2 = sos;
      #1;
      sos = s;

   end

endmodule
```

☐ Explain why the module is not synthesizable.

☐ Fix the problem.