Name _____

Digital Design using HDLs
LSU EE 4755
Final Examination

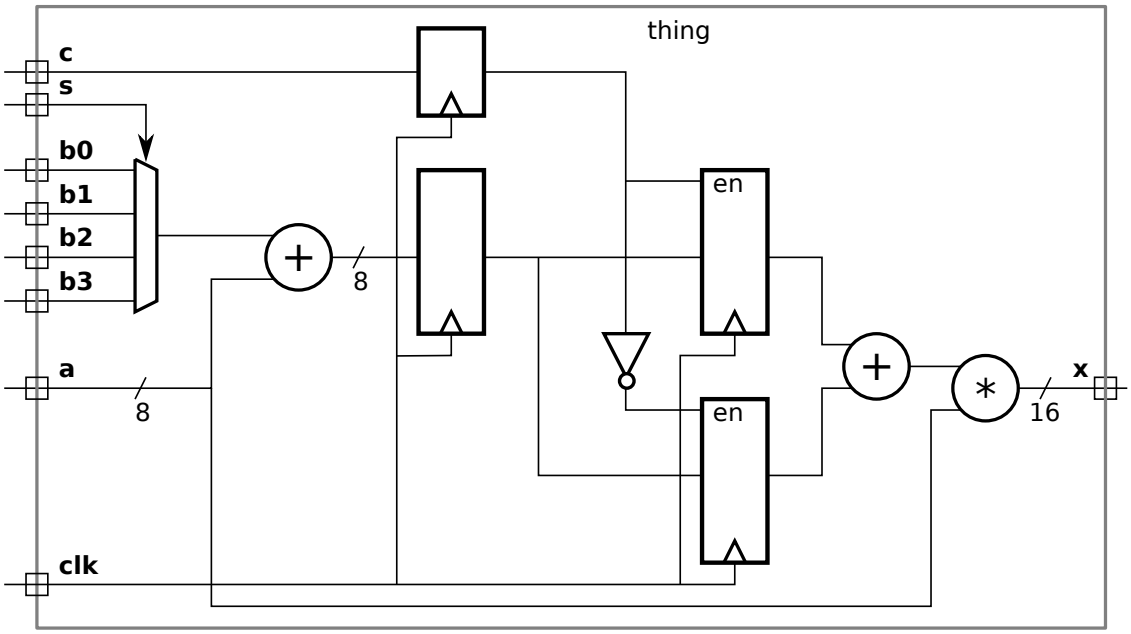
Saturday, 12 December 2015 12:30-14:30 CST

Problem 1	_____	(15 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(15 pts)
Problem 5	_____	(10 pts)
Problem 6	_____	(20 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [15 pts] Write a Verilog description of the hardware illustrated below.



☐ Verilog description of hardware including ☐ port declarations and ☐ port and other sizes.

Problem 2: [20 pts] The module below implements a simple memory module.

```
module smemory #( int size_lg = 4, int dbits = 8, int size = 1 << size_lg )
   ( output uwire [dbits-1:0] rd_data,
     input uwire [size_lg-1:0] wr_idx,  input uwire [dbits-1:0] wr_data,  input uwire write,
     input uwire [size_lg-1:0] rd_idx,  input uwire clk );

   logic [dbits-1:0] storage [size-1:0];

   always_ff @( posedge clk ) if ( write ) storage[wr_idx] = wr_data;

   assign rd_data = storage[rd_idx];

endmodule
```

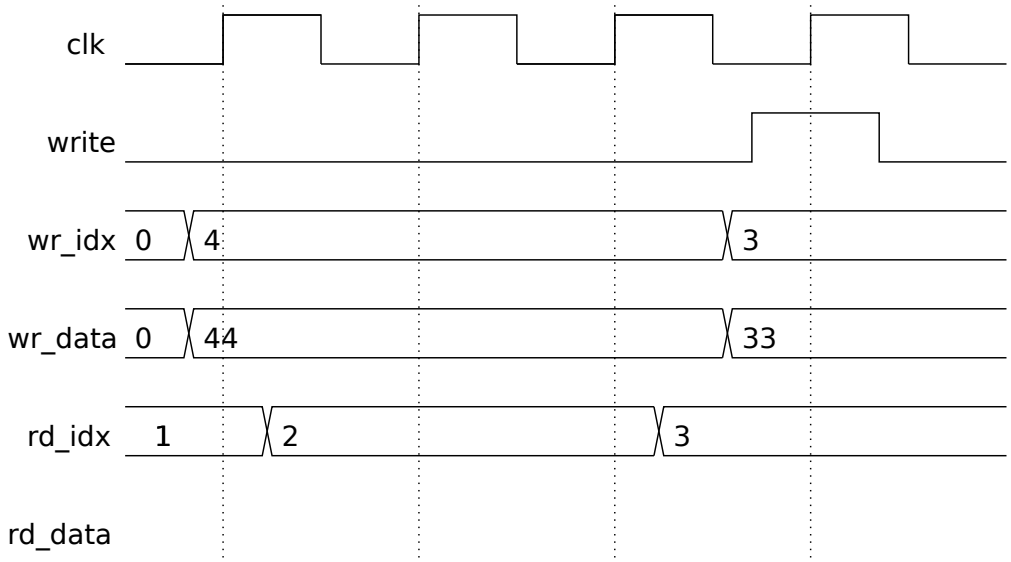
(*a*) Show the hardware that will be synthesized for this module when elaborated with `size_lg = 2`. Use registers, multiplexors, decoders, and basic gates. **Do not** use a memory module.

☐ Show synthesized hardware, including hardware for ☐ reading and ☐ writing.

Problem 2, continued: Appearing below is the module from the previous page.

```
module smemory #( int size_lg = 4, int dbits = 8, int size = 1 << size_lg )
   ( output uwire [dbits-1:0] rd_data,
     input uwire [size_lg-1:0] wr_idx,  input uwire [dbits-1:0] wr_data,  input uwire write,
     input uwire [size_lg-1:0] rd_idx,  input uwire clk );

   logic [dbits-1:0] storage [size-1:0];

   always_ff @( posedge clk ) if ( write ) storage[wr_idx] = wr_data;

   assign rd_data = storage[rd_idx];

endmodule
```

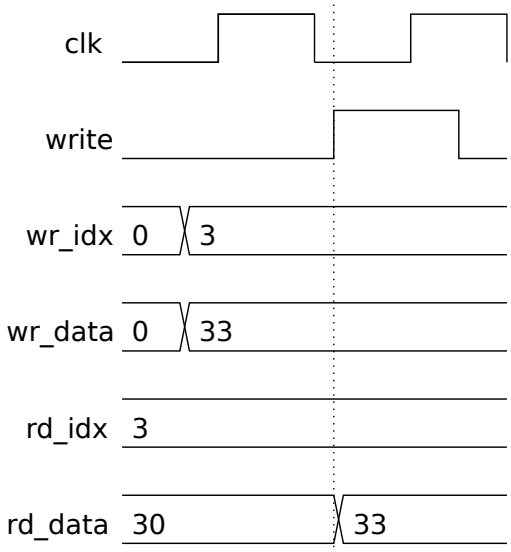
(*b*) Assume that initially location 1 (`storage[1]`) holds a 10, location 2 holds a 20, location 3 holds a 30, and so on. Complete the timing diagram below, consistent with this module.



☐ Complete `rd_data` row of timing diagram.

(*c*) Modify the module below (same as one on previous page) so that its behavior is consistent with the timing diagram to the right. That is, if the location being written is the same as the one being read the `rd_data` output shows the data on `wr_data`. If the locations don't match or nothing is being written the behavior is unchanged.

☐ Modify the module.




```
module smemory_bp #( int size_lg = 4, int dbits = 8, int size = 1 << size_lg )
 ( output uwire [dbits-1:0] rd_data,
   input uwire [size_lg-1:0] wr_idx,  input uwire [dbits-1:0] wr_data,  input uwire write,
   input uwire [size_lg-1:0] rd_idx,  input uwire clk );

   logic [dbits-1:0] storage [size-1:0];









   always_ff @( posedge clk ) if ( write ) storage[wr_idx] = wr_data;









   assign rd_data = storage[rd_idx];









endmodule
```

Problem 3: [20 pts] The module below and the similar one on the next page are like the memory module from the previous problem, except that their output is the sum of locations `rd_start`, `rd_start+1`, ..., `rd_start+rd_len-1`. Assume that `rd_start+rd_len <= size`.

```
module rsum_plan_a #( int sz_lg = 4, int ebits = 8, int size = 1 << sz_lg )
   ( output logic [ebits-1:0] sum,
     input [sz_lg-1:0] wr_idx,      input [ebits-1:0] wr_data,   input write,
     input [sz_lg-1:0] rd_start,    input [sz_lg-1:0] rd_len,    input clk    );

   logic [ebits-1:0] storage [size-1:0];

   // Don't show synthesized hardware for line below.
   always_ff @( posedge clk ) if ( write ) storage[wr_idx] = wr_data;

   // Plan A -- Show Synthesized Hardware for this Verilog
   always_comb begin
        sum = 0;
        for ( int i=0; i<size; i++ ) if ( i < rd_len ) sum += storage[ i + rd_start ];
     end

endmodule
```

(*a*) Show the hardware that will be synthesized for the `always_comb` block. Include basic optimizations, but don't optimize to the point where hardware is identical to Plan B (next page).

☐ Show not-too-optimized hardware for `sum`.

(*b*) Appearing below is Plan B for the module. Though we know it produces the same value for `sum` as Plan A, it might be synthesized into different hardware. Show the hardware synthesized for Plan B.

```
module rsum_plan_b #( int sz_lg = 4, int ebits = 8, int size = 1 << sz_lg )
   ( output logic [ebits-1:0] sum,
     input [sz_lg-1:0] wr_idx,      input [ebits-1:0] wr_data,   input write,
     input [sz_lg-1:0] rd_start,    input [sz_lg-1:0] rd_len,    input clk   );
   logic [ebits-1:0] storage [size-1:0];

   // Don't show synthesized hardware for line below.
   always_ff @( posedge clk ) if ( write ) storage[wr_idx] = wr_data;

   // Plan B -- Show Synthesized Hardware for this Verilog
   always_comb begin
        sum = 0;
        for ( int i=0; i<size; i++ )
          if ( i >= rd_start && i < rd_start + rd_len ) sum += storage[ i ];
     end
endmodule
```

☐ Show the hardware that will be synthesized for Plan B.

(*c*) Which one is better?

☐ Which is better, ◯ *Plan A* or ◯ *Plan B* .

☐ Explain, with a rough estimate of cost and timing.

Problem 4: [15 pts] Appearing below are excerpts based on the `cam_hash` module used in class, showing what we called the `hash_early` design. Recall that with the early hash design the hash function (in module `hash`) is computed before the positive clock edge while the lookup occurs after the positive edge. We assumed that the hash could be computed in about $\frac{1}{2}$ of our target clock period.

```
module cam_hash_exceprt
   ( output [dwid:1] out_data, output out_valid,        output ready,
     input [kwid:1] in_key,    input [dwid:1] in_data,
     input Cam_Command in_cmd, input clk);

   logic [kwid:1] b_key;
   logic [dwid:1] b_data;
   logic [hkey_size-1:0] b_hash;
   Cam_Command b_cmd;

   uwire [hkey_size-1:0] ohm_key_out;

   always_ff @( posedge clk ) begin
      b_key <= in_key;
      b_data <= in_data;
      b_cmd <= in_cmd;
      b_hash <= ohm_key_out;
   end

   hash #(kwid,num_sets_lg) our_hash_module( ohm_key_out, in_key );

   /// Hardware to find matching key below ...
```

(*a*) The early hash design requires that the external hardware has the right timing behavior. Show a timing diagram in which the timing behavior is correct for early hash, and one in which it is wrong. The "wrong" behavior should result in incorrect results using the early hash design, but correct results without the early hash design.

☐ Timing diagram showing ☐ correct and ☐ wrong behavior.

Problem 4, continued:

(*b*) Register `b_hash` saves the hashed version of `in_key`, and `b_key` holds the unhashed version. Why do we need the unhashed version?

☐ `b_key` is needed because ...

Problem 5: [10 pts] The Verilog below is part of a testbench (taken from `icomp.v`).

```
initial begin

   /// Watchdog – Stop simulation if it's taking too long.
   //
   fork begin

      automatic int cyc_limit = in_str.len() * 100;

      fork
         wait ( cycle_num == cyc_limit );
         wait ( tb_insert_done && tb_remove_done );
      join_any

      if ( cycle_num >= cyc_limit ) begin
         $write("Exceeded cycle limit, exiting.\n");
         $fatal(1);
      end

   end join_none

   // Below: Send data to module under test.
```

(*a*) Generically explain what a `fork` and `join` pair do (ignoring the code above).

☐ `fork` and `join` ...

(*b*) How would execution be effected if the last `join_none` were changed to `join_any`?

☐ Impact of changing `join_none` to `join_any` in code above.

(*c*) How would execution be effected if the inner `join_any` were changed to a `join_all`?

☐ Impact of changing `join_any` to `join_all` in code above.

Problem 6: [20 pts] Answer each question below.

(*a*) Suppose we would like our hardware to operate at a 1 GHz clock frequency. How do we tell the synthesis program? (The exact syntax is not important.)

☐ Method to tell synthesis program the clock frequency.

(*b*) The synthesis program will apply our target clock frequency to paths starting at launch points and ending at capture points. We could explicitly specify such points but if we don't it will use default launch and capture points. What are they?

☐ By default timing is computed for paths that start at:

☐ and end at:

(*c*) Suppose our target clock frequency is 1 GHz. What is the harm in telling the synthesis program to synthesize for 2 GHz? For 0.5 GHz?.

☐ Harm in specifying 2 GHz when we just need 1 GHz:

☐ Harm in specifying 0.5 GHz when we just need 1 GHz:

(*d*) The code below will inconsistently assign a variable. Explain why and fix the problem.

```
module short_ans( output logic [7:0] x, y, input [7:0] a, b, c, input clk);

   always @( posedge clk ) begin

      x = a + b;

   end

   always @( posedge clk ) begin

      y = x + c;

   end

endmodule
```

☐ Reason for inconsistent behavior:

☐ Fix problem.

(*e*) Describe the problem with the module below. How might it affect simulation?

```
module short_ans2( output logic [7:0] x, input [7:0] a, b, input reset);

   always_comb begin

      if ( reset ) x = a; else x = x + b;

   end

endmodule
```

☐ Problem with module.

☐ Impact on simulation.