Name Solution

Digital Design using HDLs
LSU EE 4755
Final Examination

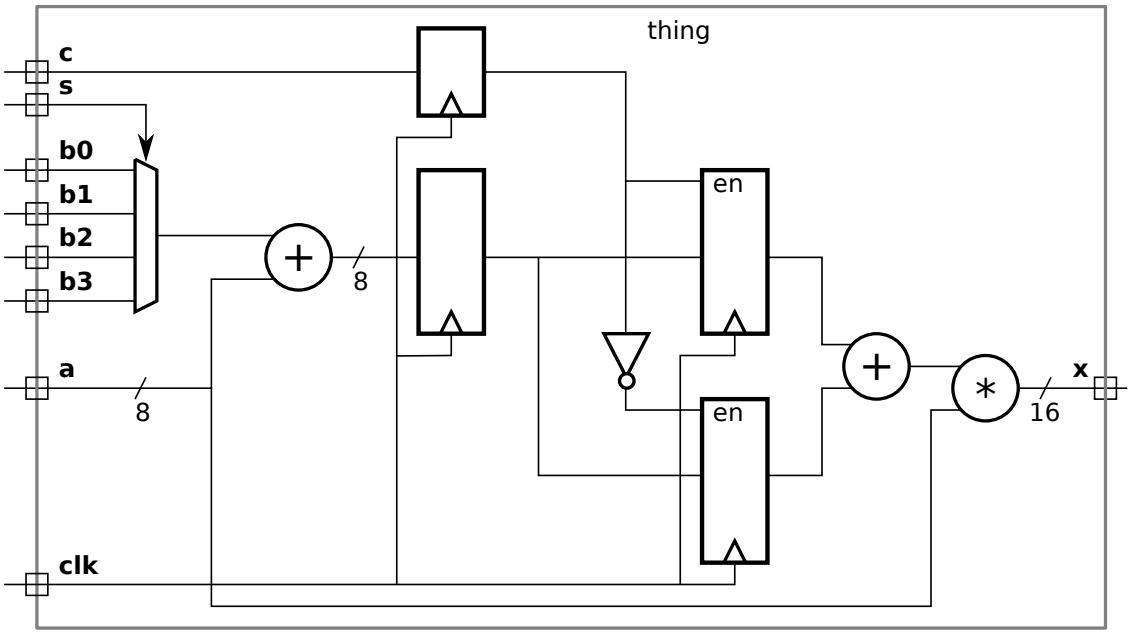
Saturday, 12 December 2015 12:30-14:30 CST

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 Not Synthesizable

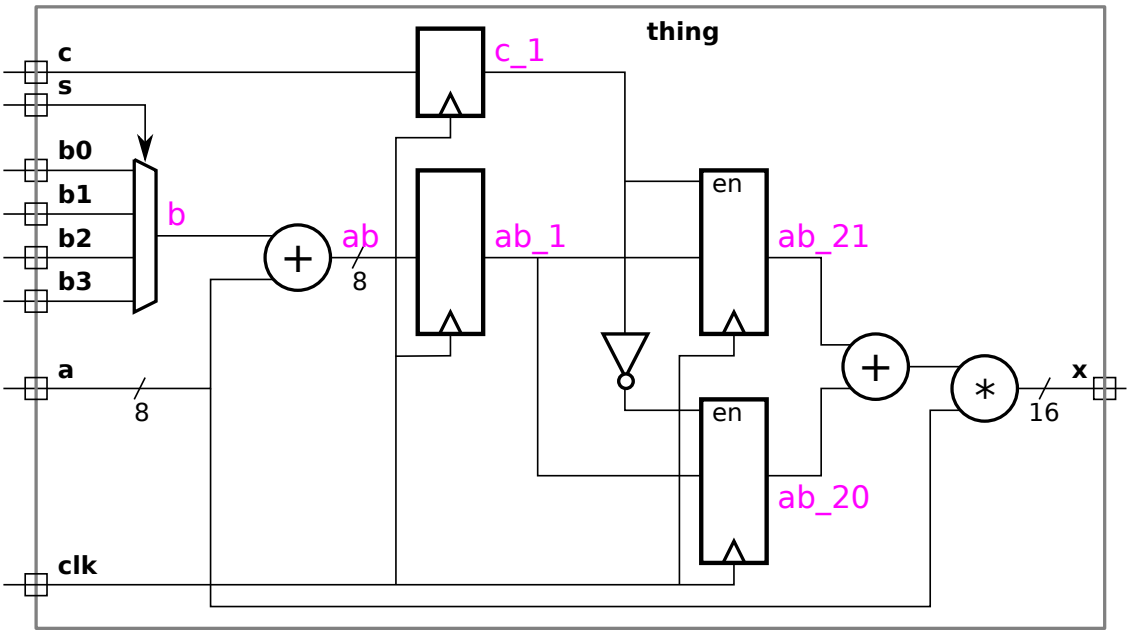
Good Luck!

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



SOLUTION ON NEXT PAGE

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



The solution appears below. Names for wires that were unlabeled in the problem appear in purple. (That is, the purple labels are part of the solution.) Note the use of `case`/`endcase` for the mux. Though using an `if`/`else` chain or the conditional operator, `?:`, would be correct, they are more tedious and prone to error and so it's worth taking the trouble to remember to use `case`.

```
module thing( output uwire [15:0] x,  input uwire c,       input uwire [1:0] s,
              input uwire [7:0] b0, b1, b2, b3, a,         input wuire clk );

   logic [7:0] b, ab, ab_1, ab_20, ab_21;
   logic       c_1;

   always_comb begin
      case ( s )
        0: b = b0;
        1: b = b1;
        2: b = b2;
        3: b = b3;
      endcase
      ab = a + b;
   end

   always_ff @( posedge clk ) begin
      c_1 <= c;     // Note: Delayed assignment, so if(c_1) uses prior value.
      ab_1 <= ab;   // Delayed assignment here too.
      if ( c_1 ) ab_21 <= ab_1; else ab_20 <= ab_1;
   end

   assign       x = a * ( ab_20 + ab_21 );
endmodule
```

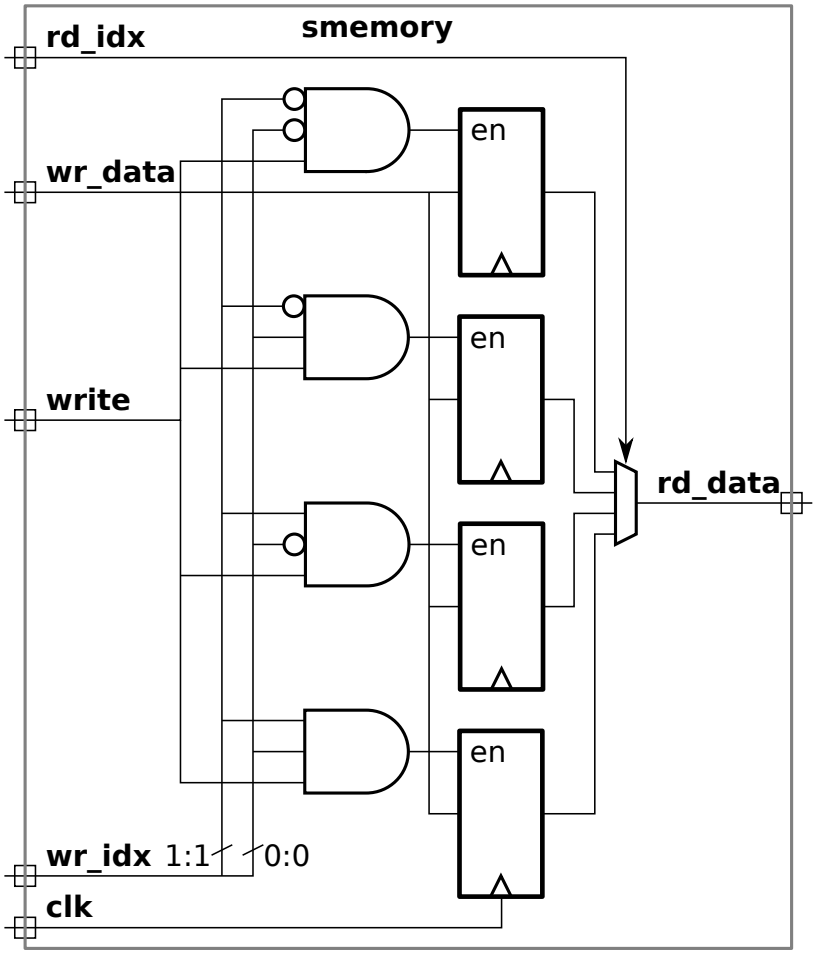
Problem 2: [20 pts] The module below implements a simple memory module.

```
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.

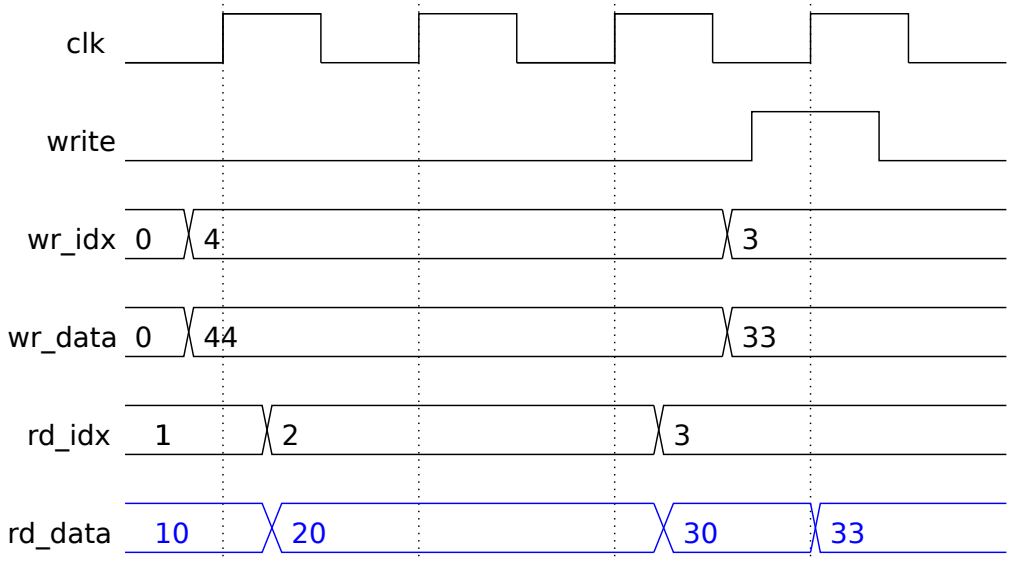
Solution appears below.



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
```

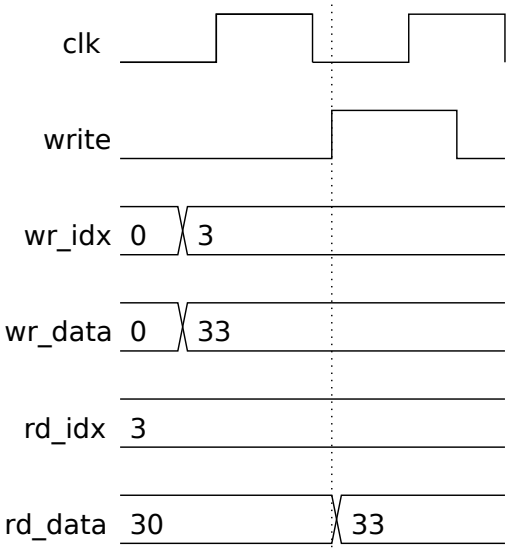
(*b*) Assume that initially location 1 (`storage[1]`) holds a 10, location 2 holds a 20, location 3 holds a 30, and so on. Complete the timing diagram below, consistent with this module.



☑ Complete `rd_data` row of timing diagram.

Solution appears above in blue.

(*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.

Solution appears below. The original line is commented out for reference. Otherwise, cluttering your code with commented out lines is bad style. Instead, learn how to diff your working copy with the latest committed version and be able to do so in < 500 ms.

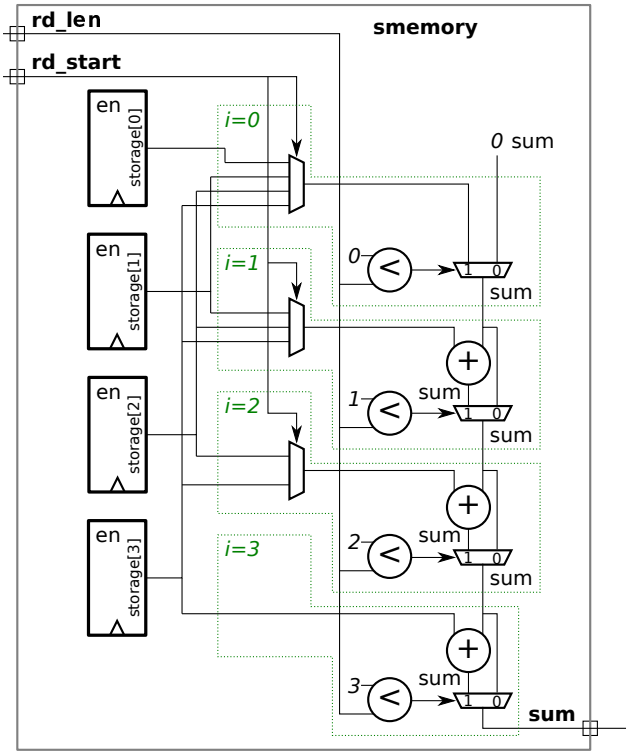
```
module smemory_bp #( int size_lg = 4, int dbits = 8, int size = 1 << size_lg )
 ( output uwire [dbits-1:0] rd_data,
   input uwire [size_lg-1:0] wr_idx,  input uwire [dbits-1:0] wr_data,  input uwire write,
   input uwire [size_lg-1:0] rd_idx,  input uwire clk );

   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];
   // SOLUTION
   assign rd_data = write && rd_idx == wr_idx  ?  wr_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`.

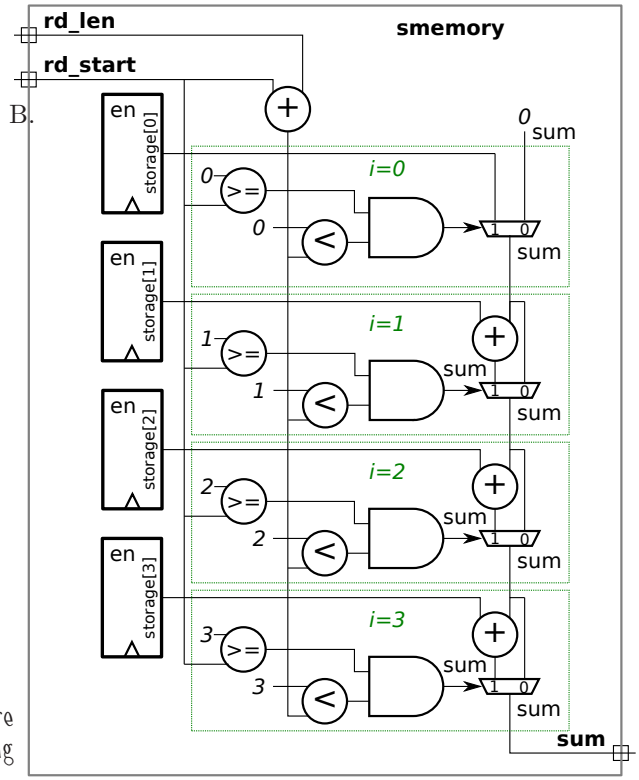


(*b*) Appearing below is Plan B for the module. Though we know it produces the same value for `sum` as Plan A, it might be synthesized into different hardware. Show the hardware synthesized for Plan B.

```
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.

Solution appears to the right.



(*c*) Which one is better?

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

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

Short Answer: The cost of the multiplexors makes Plan A more expensive than Plan B when `ebits` is greater than 1. The timing is about the same.

Detailed answer: Plan A contains three more multiplexors than Plan B, the total number of additional multiplexor inputs is $3+2+1 = 6$, and each of these is `ebits` wide, for a cost of $6 \times 3 \times e = 18e$ units, where e is `ebits`. The logic in Plan B that's not in Plan A includes four AND gates, a 2-bit adder and three fixed comparison units. Assume that the cost of a BFA is 10 units. Since the inputs to the adder are 2-bit quantities and since a carry-out is needed, the cost is 20 units. (The adder output must be three bits to do the comparison `i<rd_start+rd_len`.) Assume that the $\geq$ fixed comparison units cost 3 units each (draw a truth table). The total cost of logic in Plan B not in plan A is then $4 + 20 + 3 \times 3 = 33$ units. So Plan B is less expensive whenever the storage element size, `ebits`, is greater than 1 bit, which presumably is most of the time.

The path to the select signal for the $i = 0$ mux in Plan B passes through an adder (albeit a small one), a comparison, and an AND gate. In contrast, signal arrive at the data inputs to the corresponding multiplexor at a delay of about 4 units. Therefore Plan B is a little bit slower based on this simple analysis.

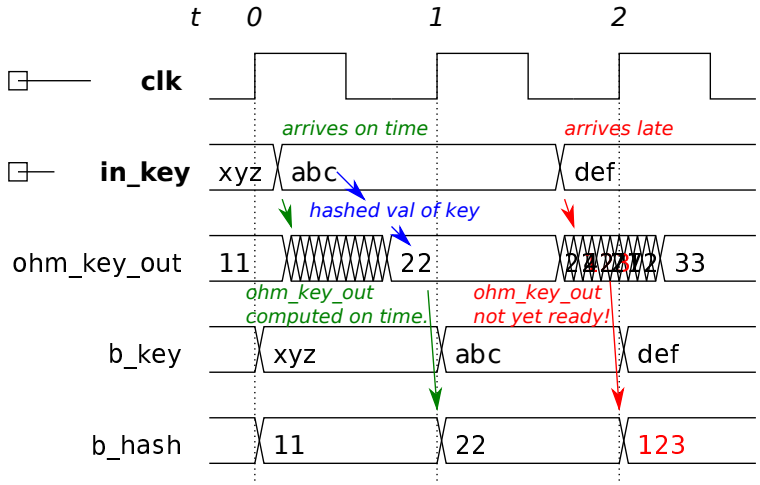
Problem 4: [15 pts] Appearing below are excerpts based on the `cam_hash` module used in class, showing what we called the `hash_early` design. Recall that with the early hash design the hash function (in module `hash`) is computed before the positive clock edge while the lookup occurs after the positive 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.

Solution appears to the right. In the early hash design the value on port `in_key` must arrive in the first half of the clock cycle (before the negative edge). That is what happens for input `abc` and so `hash` which computes `ohm_key_out` has enough time to finish. The correct hash, 22 is clocked into register `b_hash`. In contrast, key `def` arrives late, and so when the next positive clock edge arrive `ohm_key_out` has not stabilized and so some arbitrary value is clocked into `b_hash`. Notice that `b_key` gets the correct value in both cases, because register gets its input directly from input port `in_key`.



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 ...

The number of bits in the hash of a key is less than the key itself, therefore two keys can have the same hash. The unhashed version of the key is needed to check whether the key matches the key for item at the hashed location.

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` ...

Each statement executes with its own thread of control, meaning that delays and other timing controls in one does not affect the progress of the other. The statement after the `join` does not execute until all threads inside the `fork`/`join` finish.

(*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.

Execution would never reach the `//Below` statement. With the `join_none`, execution proceeds to the `//Below` statement without delay. Code after the `//Below` statement tests modules and will set `tb_insert_done` and `tb_remove_done` when tests are finished. But with `join_none` changed to `join_any` the `//Below` statement will not be executed until the first `fork` finishes. That first `fork` finishes when either the cycle limit is exceeded or all modules have been tested, whichever comes first. But with `join_none` changed to `join_any` module tests won't have started and so the cycle limit will be exceeded. Note that if the cycle limit is exceeded the code exits with a fatal error, and so the `//Below` statement will never be reached.

(*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.

The testbench will always report that the cycle limit was exceeded, even if all tests were completed.

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.

Short Answer: `define_clk -name ee4755 -period 1000 myclkport`.

Details: In Cadence Encounter use the command `define_clk -name NAME -period PERIOD PORTS`. To set the clock frequency to 1 GHz set the period argument to 1000, which is the clock period in picoseconds: $10^{12}\frac{1}{10^9} = 1000$. Argument `PORTS` is set to the name of the clock ports and `NAME` is a name by which this clock can be referred to in subsequent commands.

(*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: register outputs.

☑ and end at: register inputs.

Notice that the default launch and capture points **do not** include module inputs and outputs. Those have to be added with `external_delay` commands.

(*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:

The resulting design will work correctly, but may be more expensive than had we specified 1 GHz.

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

The synthesized hardware may not work at 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:

Because the value of `x` used in the second `always` block may be before the `a+b` assignment, or after.

☑ Fix problem.

One way is to put the two statements in the same block. That's shown below. Another possibility is to use nonblocking assignment.

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

   always @( posedge clk ) begin
      x = a + b;
      y = x + c;
   end

endmodule
```

(*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.

Wire `x` is both an input and an output of the `always_comb`. So each change in `x` would trigger another execution of the block. To fix it a clock is needed to control when `x` is incremented.