**What**: Mid-Term Exam #1 - Solution

**Details**: Please answer the following questions to the best of your ability. If you need to make any assumptions or you do something that is not explicitly mentioned in the question, please note it on your paper and the rationale for your answer (Failure to do so will result in missing points).

One last note, since you will be NOT be hand-writing the answers (you MUST use a computer/word-processor), your neatness matters. So, if you do a poor job with your answer, you will receive no points!!

**Question 1 (50 points possible)**: Given the following snippet of Verilog code, fill out the simulation schedule on the next page (only go to 35ns).

```verbatim
`timescale 1ns/1ns

module tb_c2cross ( );
reg clk, rst_n, C;
reg [7:0] r_count;

// Use of initial statement to generate clock
initial
begin
    clk = 0;
    rst_n = 0;
    # 5 rst_n = 1;
    forever
        begin
            # 5 clk = 1;
            # 5 clk = 0;
        end
end

// Always block to generate synchronous transactions
always@(negedge clk or negedge rst_n)
    if (~rst_n)r_count <= 8'd0;
    else begin r_count <= r_count + 8'd1;
always@
    #5 C = ~|r_count
endmodule
```
**Question 2 (20 of 120 points possible):** Write a System Verilog Assertion that describes the following waveform. Your assertion must cover the sequence of expression evaluation points shown below using the $O$ symbols. (This assertion should make use of some type of implication operator such that it only is activated when request is detected on the next clock edge.)

```
// I am looking for one of the following:
request |-> (##1 (grant & request & !done) ##2 (!grant & done));
request |==> (grant & request & !done) ##2 (!grant & done);
```

**Bonus Question #2 (5 points):** The done going high at the EXACT same time grant goes low is considered what kind of design technique? Combinational Feedback Loop
Question 2 (5 points possible) (2 minutes): What does the $root syntactical element do in SystemVerilog? $root refers to the top level of the simulation environment (I am looking for a simple answer here).

But, to expand upon this: There might be multiple modules at the same level, e.g. parallel and not instantiated but referenced. An example of this might be a waveform dumping module which is in parallel with the test bench (not instantiated in the test-bench). The $root syntax will allow us to access both the waveform dumping module and the test-bench of our device-under-test.

Question 3 (5 points possible): How do we check whether randomization was successful when using the SystemVerilog rand data-type? The randomization call will return a value that indicates whether it was successful or not. A value of 1 indicates success while a value of 0 indicates failure.

Question 4 (5 Points): Dynamic arrays are useful for dealing with contiguous collections of variables whose number changes dynamically (e.g. int array[]). What SystemVerilog data-type dynamically sizes based on the index used (sometimes used as hash value)? Anything along the lines of an associative array will be accepted.

Question 5 (5 Points): What does "void" mean in the return value of a function’s specification? "void" in the return value of a function’s specification means that the function does not have a return value.

Question 6 (5 Points): What is coverage driven verification? Coverage driven verification relies on coverage metrics to determine whether or not we have successfully hit all lines, statements, and branches in our design. It is not necessarily concerned with whether or not our design is functionally correct. It is concerned with how much our test cases exercise our design. It assumes that our tests will be intelligently written to cover the features we are interested in. And, by looking at what lines of our code we have exercised, we can measure the effectiveness of our testing effort. This has been proven to be a somewhat flawed approach.

What is functional verification? Later on it was determined that a more valid approach to validation would be to make sure the features of the design actually work. This was deemed as functional verification. This can be achieved by writing directed tests that verify specific features. It can also be achieved by assertions that monitor the devices behavior and ensure that it follows the design’s specification. One final approach is to create a transaction level model that generates all possible sets of traffic described in the devices’ feature list, or specification, and randomized upon those features (functional constraint based randomization).

Question 7 (5 Points): Explain pass by value and how it differs from pass by reference? Pass by value can be describe as passing a copy of a primitive data-type, e.g. an integer, an array of bits, or a bit itself through a function/task call or a module interface. If the copy sent is modified, the original element will not be updated (it will not be changed if the module or task it is passed to changes it). Pass by reference only passes the address of the object through a module interface or task/function call. This means if the module or task/function call modifies the value, the original object will also be changed.

Question 8 (5 Points): What is the difference between "program" block and "module"? The program block is used to identify test code. What makes it unique is that it is given its own scheduling region called “reactive”.

Question 9 (15 Points): What is the different between "virtual" functions and normal functions and methods. Write a small example to show the difference (it cannot be something from our lectures): "virtual" methods are those that override an already existing method in a parent class and add functionality to
```plaintext
class BaseClass
    virtual function int add(int a, int b)
        return a + b;
    endfunction

    function int meaning_of_life( )
        $display("original_method[meaning_of_life] The meaning of life is 42");
        return 42;
    endfunction
endclass

class MyClass extends BaseClass
    function int meaning_of_life(int v)
        $display("overloaded_method[meaning_of_life] The meaning of life is ", v);
        return v;
    endfunction
endclass
```
**Question 10 (30 Points):** Create a Verilog based task or function that mimics the following waveform called `do_calc`. Assume that the clock is provided by another piece of code and you are responsible for driving the values of START, IN_PROGRESS, C and DONE. The values must be visible externally while they are toggling.

**Note:** The values A, B and C are integers (make "A" and "B" inputs, and "C" is an output).
Question 11 (50 points): Given the following module "code_coverage", write the necessary code to create 100% coverage for each of these groups:

**Statement Coverage/Condition/Expression:**

```verilog
module code_coverage
#define(parameter REG_WIDTH = 8)
(output reg [REG_WIDTH - 1:00] A,
 input wire DIRECTION,
 output reg ODD, EVEN,
 input [REG_WIDTH - 1:00] D);
always @(posedge CLK or posedge RESET)
if (RESET == 'b1)  
  A <= 'd0;
else if (a)  
  A <= DIRECTION ? A + 1 : A - 1;
always@(*)
  if (A[0])  
    EVEN = 1'b0;
else  
    EVEN = 1'b1;
always@(*)
  if (A[0])  
    ODD = 1'b1;
else  
    ODD = 1'b0;
endmodule
```
Toggle Coverage (note, for this part name the variable(s) that need to be toggled):

Note: Toggle coverage verifies how many times variables and nets toggled. Toggle coverage could be as simple as the ratio of nodes toggled to the total number of nodes.

<table>
<thead>
<tr>
<th>Initial Value</th>
<th>Final Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>X or Z</td>
<td>1 or H</td>
</tr>
<tr>
<td>X or Z</td>
<td>0 or L</td>
</tr>
<tr>
<td>1 or H</td>
<td>X or Z</td>
</tr>
<tr>
<td>0 or L</td>
<td>X or Z</td>
</tr>
</tbody>
</table>

Above example shows the signal changes from one level to another. All types of transitions mentioned above are not necessary. Only 1->0 and 0->1 are important. Toggle coverage will show which signals did not change states. Toggle coverage will not consider zero-delay glitches. This is very useful in gate level simulation.

// We are only concerned about all bits of A toggling, ODD, EVEN, CLK, RESET.
initial
begin
    // Since we have no control over the counter, we have no choice but to wait
    // for all bits in the counter to toggle.
    forever
        begin
            clk = #10 ~clk;
            begin
                reset = 'b0;
                #50 reset <= 'b1;
            end
            direction = 0;
            // Realistically, the direction of the count is arbitrary.
            #100 direction = 'b1;
            // We will wait until ALL bits have toggled. We do this by allowing the
            // counter to go through ALL possible values. Realistically, we only have
            // to go half way and then issue a reset. But, that would require a great
            // deal more explanation than I would like to do right now.
            repeat(REG_WIDTH << 1) @(posedge clk);
        end
end
