## Part I: Objective Questions

Section A: True/False (30 Points Total)
Read each of the following statements carefully. If the entire statement is true, circle T, otherwise circle F. Each correct response is worth two points.

T F 1. Verilog is case sensitive.

T F 2. Verilog synthesizers treat the white space ‘ ‘ and carriage returns differently.

T F 3. “beginmodule” and “endmodule” are reserved words in Verilog.

T F 4. “2’b1x == 2’b1x;” has a “true” return value.

T F 5. The semantics of an “&” operator depends on the number of operands.

T F 6. An “if” statement must always be inside of an “always” block.

T F 7. Verilog may be written at the Behavioral, Structural, Gate, Switch, and Transistor levels.

T F 8. The use of a tick timescale (‘timescale) enables code to be synthesized with the specified delays.

T F 9. Verilog permits module ports to be unconnected.

T F 10. The use of the “initial” keyword is to model circuit behavior at time 0 and possibly beyond.

T F 11. The left argument of a statement in an initial or always block may be a wire.

T F 12. One must always declare the data type of a signal before using that signal within a design.

T F 13. It is NOT necessary for a task to have inputs and outputs.

T F 14. Use of Blocking Assignments is preferable to Non-Blocking Assignments because race conditions are less likely to occur.

T F 15. Verilog supports both one- and two-dimensional arrays of registers, integers, nets or times.

Section B: Multiple Choice (30 Points Total)

For each of the following statements, write the letter corresponding to the best answer in the space provided. Each correct response is worth three points.

1. **__E__** Consider the following choices below. To comment Verilog Code, one may use:

   I A "Double-slash" // for a single-line comment.
   II Multiple "Double-slashes" (one per line) for a multiple-line comment.
   III A "Block-comment" /* */ for a single-line comment.
   IV A "Block-comment" /* */ for a multiple-line comment.

   A. I and II
   B. I and IV
   C. II and III
   D. I, II, and IV
   E. I, II, III and IV

2. **__D__** Which of the following is (are) equivalent to logic level 1?

   A. 1
   B. 1'b1
   C. 1 'b 1
   D. A and B
   E. A, B and C

3. **__D__** If X is a ten bit number, which of the following will left extend X[9] three times?

   A. {(X[9], X[9], X[9]), X}
   B. (X[9], X[9], X[9], X)
   C. {3(X[9]), X[9]}
   D. {3{X[9]}, X[9]}

4. **__A__** Which of the following is true about parameters?

   A. The default size of a parameter in most synthesizers is the size of an integer, 32 bits.
   B. Parameters enable Verilog code to be compatible with VHDL.
   C. Parameters cannot accept a default value.
   D. All of the above.
   E. None of the above.

5. **__B__** Which of the following best describes a user-defined primitive (UDP)?

   A. A module which can have several outputs.
   B. A combinatorial or sequential piece of logic described through the use of a truth table.
   C. A module with variable parameters.
   D. All of the above.
   E. A and B.

6. **__D__** Which of the following is true about the always block?

   A. There can be exactly one always block in a design.
   B. There can be exactly one always block in a module.
   C. Execution of an always block occurs exactly once per simulation run.
   D. An always block may be used to generate a periodic signal.

7. **__B__** Which of the following is a *difference* between a Function and a Task?

   A. A Function can call another function; a Task cannot.
   B. A Function cannot call a task; a Task can call another task..
   C. A Function has one or more inputs; a Task has no inputs.
   D. A Function argument may be an output; a Task's argument may only be an input.

8. **__C___** Which of the following is not true about operators?

   A. A logical "or" is performed by writing "C = A || B;"
   B. '!' performs logical negation while '~' performs bitwise negation.
   C. The two types of "or" operators are "logical" and "bitwise."
   A. The "shift right" (>>) operator inserts zeros on the left end of its argument.

9. __**C**__ In synthesis, what will happen when a signal that is needed in a sensitivity list is not included?

   A. An error will be generated and the code cannot be synthesized.
   B. A warning message will be generated and the code will be synthesized but the resulting netlist will not provide the desired results.
   C. The synthesis tool will ignore the sensitivity list since all objects that are read as part of a procedural assignment statement are considered to be sensitive.
   D. There will be no effect on the design and pre-synthesis simulation will be consistent with post-synthesis simulation.

10. __**C**__ Which of the following is used for Verilog-based synthesis tools?

   A. Intra-statement delay statements can be synthesized, but inter-statement delays cannot.
   B. Inter-statement delay statements can be synthesized, but intra-statement delays cannot.
   C. Initial values on wires are almost always ignored.
   D. Synthesized results are identical for “if” and “case” statements

## Section C: Short answer (40 Points Total)

Provide a short answer to the following. Complete sentences are not required. The number of points given for a correct response is indicated in parenthesis. Partial credit will be given for partially correct answers.

1. What is the difference between implicit (or positional) association and explicit (or named) association? (5 points)

   Implicit association connects ports with signals based on the order in which the signals are instantiated. Explicit association connects ports with signals based on the direct named connections specified in the instantiation.

2. Explain the difference in functionality between the "==" and "===" operators. (3 points)

   The "==" operator may only be used as a comparator between two non-'x' or non-'z' values. If the "==" operator is applied to arguments which contain at least one 'x' or 'z', a false value will be returned. Use "===" for explicitly comparing 'x' or 'z' values without interpretation of these values.

3. Why is it better to use a case statement over a nested if-else? (5 points)

   Use of a case statement will synthesize code which results in a multiplexer. Use of a nested if-else, on the other hand, will synthesize code which results in a multiple logic-level if-and structure. By using a single logic-level implementation (multiplexer) as opposed to a multiple logic-level implementation (if-and structure), timing improvements are often made in the design.

4. How do you guard against inferring latches? (vs FF) (5 points)

   To prevent latches from being inferred, ensure that all possible situations in a case or if-else statement are included within the code. If a value is not to be changed then a simple statement such as q = q; will be sufficient.

5. Write a Verilog module for a CMOS inverter. (5 points)

```
module INV(A, Abar);
  input A;
  output Abar;
  pmos PMOS1 (Abar, 1'b1, A);
  nmos NMOS1 (Abar, 1'b0, A);
endmodule
```

6. Give an instance where it would be better to instantiate logic rather than infer it. (2 points)

Answers will vary.

7. Give an instance where it would be better to infer logic rather than instantiate it. (2 points)

Answers will vary.

8. List Five Verilog Gate Primitives. (5 points)

Answers will vary. Some examples are:

and
nand
or
nor
xor
xnor
buf
not

9. Suppose we assign A and B as shown in the code below. What are the waveforms of A and B? If it is not possible to determine the waveforms, what additional information do we need? Explain your reasoning. Assume that the value of A is 0, and the value of B is 1 when the code below is executed. (8 points)

```
#5 A = 1;
#5 B = 0;
```

These are blocking assignments which is evidenced by the use of the '=' rather than the '<='.

However, if the statements are in a parallel (fork-join) block, then the waveforms show both A and B transitioning at time unit 5. If on the other hand, the statements are in a sequential (begin-end) block, then A transitions at time 5, and B at time 10.

## Part II: Free-Response Questions

Section A: Critical Thinking Design Methodology (60 points)
This portion of the exam involves creating several Verilog designs. Five lower-level modules are written and then two of these are combined to create a top-level design.

For each module, write a complete Verilog program with correct declaration, etc. Write your answers below the module description. A completely correct solution will receive the number of points indicated in parentheses. Partially correct solutions will receive partial credit.

---

Module 1: Gray Counter (10 points)
In this module, create a three-bit gray counter with positive edge reset. When reset, the count value becomes "000". Recall that a Gray Counter changes only one-bit at a time. For example, a 2-bit Gray counter has a count sequence 00, 01, 11, 10 corresponding to a decimal count values of 0, 1, 2, 3 respectively. In the Gray count sequence, only one bit changes between adjacent count values, and the right-most bit is changed as long as it does not result in a code word that has been visited earlier.

The following are the ports of the module:

| | |
|---|---|
| CLK | 1-bit clock input, all actions on positive edge |
| RESET | 1-bit reset, causes reset on positive edge |
| GRAY_OUT | 3-bit result |

---

```
module gray_cnt (
        clk,
        reset,
        gray_out);

input clk;
input reset;

output [2:0] gray_out;

reg [2:0] gray_out;

always @ (posedge clk or posedge reset)
if (reset)
        gray_out = 3'b000;
else
        case (gray_out)
                3'h0:   gray_out = 3'h1;
                3'h1:   gray_out = 3'h3;
                3'h3:   gray_out = 3'h2;
                3'h2:   gray_out = 3'h6;
                3'h6:   gray_out = 3'h7;
```

```
                3'h7:    gray_out = 3'h5;
                3'h5:    gray_out = 3'h4;
                3'h4:    gray_out = 3'h0;
        default: gray_out = 3'h0;
        endcase

endmodule
```

Module 2: Parallel-in, Serial-out Shift Register. (10 points)

In this module, create a register that shifts data in parallel but shift data out serially, MSB first. The following are the ports of the module:

| | |
|---|---|
| CLK | 1-bit clock, **all** operations must be on the rising edge |
| LD | 1-bit input, when high, PI is loaded into the shift register |
| SHIFT | 1-bit shift enable input, when high, contents of shift register are shifted out on to the serial output Q |
| PI | 8-bit parallel data input |
| Q | 1-bit serial output |

---

```
module piso_shift (
        clk,
        ld,
        shift,
        pi,
        q);

input clk;
input ld;
input shift;
input [7:0] pi;
output q;

reg q;
reg [7:0] shifter;

always @ (posedge clk)
if (ld)
        shifter = pi;
else if (shift)
        begin
                q = shifter[7];
                shifter = {shifter[6:0], 1'b0};
        end

endmodule
```

Module 3: Up-Down, Loadable Counter. (10 points)

In this module, create a counter that counts in both the up and down directions. The preset is to be set to decimal value 3. That is, upon asserting the reset signal low, the register value should be reset to 3. Also, upon asserting the load signal LD, the register value should be set to the input value DIN. Both the reset and the load are synchronous and the module should count on the rising edge of the clock. The following are the ports of the module:

| | |
|---|---|
| CLK | 1-bit clock input, all actions performed on rising edge |
| RESET_N | 1-bit preset (synchronous) |
| UP_DNN | 1-bit input (if '1', then count up, if '0', then count down) |
| LD | 1-bit load enable input, loads synchronized with CLK rising edge |
| DIN | 3-bit input data for loading counter value |
| Q | 3-bit result |

---

```
module up_dn_ld_cnt (
        clk,
        reset_n,
        up_dnn,
        ld,
        din,
        q);

input clk;
input reset_n;
input up_dnn;
input ld;
input [2:0] din;
output [2:0] q;

reg [2:0] q;

always @ (posedge clk or negedge reset_n)
if (~reset_n)
        q = 3'b011;
else
        if (ld)
                q = din;
        else if (up_dnn)
                q = q + 1;
        else
                q = q - 1;

endmodule
```

Module 4: Magnitude Comparator. (10 points)
In this module, two inputs are treated as signed, 2's complement numbers and compared. The output should be a 1 if the AD_IN is greater than or equal to VS_IN. The following are the ports of the module:

| | |
|---|---|
| AD_IN | 8-bit data input |
| VS_IN | 5-bit input |
| GTE | 1-bit output |

---

```
module signed_add_decoder (
        ad_in,
        vs_in,
        gte);

input [7:0] ad_in;
input [4:0] vs_in;
output gte;

assign gte =  (ad_in >= { 3{vs_in[4]}, vs_in}) ? 1'b1 : 1'b0;

endmodule
```

Module 5: Multiplexer. (10 points)

In this module, create a byte-wide 8 to 1 multiplexer. In this case, the value on the 3-bit select line will route 1 of 8 inputs to the output. This module is purely combinatorial. The following are the ports of the module:

| | |
|---|---|
| SEL | 3-bit select line |
| D0, D1, D2, D3, D4, D5, D6, and D7 | 8-bit data inputs |
| O | 8-bit output |

---

```
module mux_8 (
        sel,
        d0,
        d1,
        d2,
        d3,
        d4,
        d5,
        d6,
        d7,
        o);

input [2:0] sel;
input [7:0] d0;
input [7:0] d1;
input [7:0] d2;
input [7:0] d3;
input [7:0] d4;
input [7:0] d5;
input [7:0] d6;
input [7:0] d7;
output [7:0] o;
reg o;

always
        case (sel)
                3'b000 : o = d0;
                3'b001 : o = d1;
                3'b010 : o = d2;
                3'b011 : o = d3;
                3'b100 : o = d4;
                3'b101 : o = d5;
                3'b110 : o = d6;
                3'b111 : o = d7;
        endcase
endmodule
```

Top-Level Design (10 points)

For this design, combine Gray Counter (Module 1) with the Multiplexer (Module 5) to create a circuit such that the output of the Gray counter controls the select lines of the multiplexer. The top-level design has the following port definitions:

| | |
|---|---|
| CLK | 1-bit clock |
| RESET | 1-bit reset line |
| OUT_DATA | 8-bit data output |
| IN0, IN1, IN2, … IN7 | 8-bit data inputs |

---

```
module top (clk, reset, out_data, in0, in1, in2, in3, in4, in5, in6, in7);
        input clk, reset;
        input [7:0] in0;
        input [7:0] in1;
        input [7:0] in2;
        input [7:0] in3;
        input [7:0] in4;
        input [7:0] in5;
        input [7:0] in6;
        input [7:0] in7;
        output [7:0] out_data;

        wire [7:0] GRAYOUT;

        mux_8 MUX8_INST1 (GRAYOUT, in0, in1, in2, in3, in4, in5, in6, in7, out_data);
        gray_cnt GRAYCNT_INST1 (CLK, RESET, GRAYOUT);
endmodule
```

Section B: Debugging and Error-Checking (40 points)

The following Verilog code is supposed to describe a circuit that has an 8-bit data input, an 8-bit data output, plus two additional single-bit outputs. The over_flow output is strictly combinatorial. You may assume that the operations to produce data_out, carry_out and over_flow are correct (i.e. the functional specification for these values is correct). However, syntax errors for these statements do exist.

Examine the code below and identify any errors that would prevent this code from compiling and synthesizing properly. These errors may include syntax errors, logical design errors or needed lines that are missing from the code. Indicate the modifications you would make to the code to make it work. Include comments beside these modifications or the portion of the code that is incorrect. There are 20 errors in the code below.

---

```
module bad_module (clk,                    (1) Removed "is" after module
                   reset,
                   data_in,                (2) Removed [7:0]
                   carry_in,               (3) Added carry_in
                   data_out,               (4) Changed ";" to ","
                   carry_out,              (5) Changed "carry out" to "carry_out"
                   over_flow               (6) No comma after "over_flow"
                   );

input clk;
input reset;
input carry_in;
input [7:0] data_in;                       (7) Added [7:0]
output [7:0] data_out;                     (8) Added [7:0]
output carry_out;
output over_flow;


reg [7:0] data_out;
reg carry_out;                             (9) Changed "over_flow" to "carry_out"

always @ (posedge clk or posedge reset)    (10) Changed "|" to "or"
if (reset)                                 (11) Added Parentheses around "reset"
                                           (12) Removed "then"
        begin                              (13) Added "begin"
        data_out = 0;
        carry_out = 0;
        end                                (14) Added "end"
```

```
else
        begin                                   (15) Added “begin”
        data_out = (data_in * 2) + carry_in;
        carry_out = ~& data_in[7:6];            (16) Added “_” between “carry” & “out”
                                                (17) Changed “7|6” to “7:6”
end

assign over_flow = data_out[7];                 (18) Changed “<=” to “=”
                                                (19) Changed “out7” to “out[7]”

endmodule                                       (20) Removed space between “end” &
                                                “module”
```