Data Types & Constrained Random Assignment

The purpose of this assignment is to:

- Practice creating dynamic arrays, associative arrays, and queues
- Practice using $random & constrained randomization

1) Write a module to test dynamic array data type and its predefined methods. Run Questasim to make sure the display statements are working as expected.

   - declare two dynamic arrays dyn_arr1, dyn_arr2 of type int
   - initialize dyn_arr2 array elements with (9,1,8,3,4,4)
   - allocate six elements in array dyn_arr1
   - initialize array dyn_arr1 with index as its value
   - display dyn_arr1 and its size
     - Expected output: (0,1,2,3,4,5), 6
   - delete array dyn_arr1
   - reverse, sort, reverse sort and shuffle the array dyn_arr2 and display dyn_arr2 after using each method (refer to slide 71 in the notes)
     - Expected output: (4,4,3,8,1,9), (1,3,4,4,8,9), (9,8,4,4,3,1), <shuffled_array>

2) Write a module to test queue data type and its predefined methods. Run Questasim to make sure the display statements are working as expected.

   - Declare int j and a queue q of type int
   - initialize int j as 1 and queue q as (0, 2, 5)
   - insert int j at index 1 in queue q and display q
   - delete index 1 element from queue q and display q
   - push an element (7) in the front in queue q and display q
   - push an element (9) at the back in queue q and display q
   - pop an element from back of queue q into j, display q, and j
   - pop an element from front of queue q into j, display q, and j
   - reverse, sort, reverse sort and shuffle the queue and display q after using each method

3) The design to be tested is a synchronous single-port 8-bit x64K (512kBit) RAM. The RAM will read on the positive edge of the clock when input read =1 and write on the positive edge of the clock when input write = 1. Write enable signal has a higher priority than the read enable signal and both write and read data from the RAM is not allowed at the same time. Even parity will be calculated on data written to the RAM and placed in the 9th bit of the memory. The partially completed memory model is below (add the memory declaration)

```
module my_mem(
                    input clk,
                    input write,
                    input read,
                    input [7:0] data_in,
                    input [15:0] address,
                    output reg [7:0] data_out
);

  // Declare a 9-bit associative array using the logic data type & the key of int datatype
  <Put your declaration here>

  always @(posedge clk) begin
    if (write)
      mem_array[address] = {~^data_in, data_in};
    else if (read)
      data_out =  mem_array[address];
  end

endmodule
```

Your testbench will:

1. Perform 100 writes of random data to random addresses followed by 100 reads to the same addresses in reverse order. To do this:
    a. Use a local parameter TESTS with the value 100 to define the size of the following dynamic arrays
    b. Create a dynamic array of addresses called *address_array*
    c. Create a dynamic array of data in for the RAM called *data_to_write_array*
    d. Create an associative array of data expected to be read indexed by the address read called *data_read_expect_assoc*
    e. Create a queue of data read called *data_read_queue*
    f. Create a task called stimulus_gen that will prepare the data to be written as well as the address using a for loop for the dynamic arrays.

g. Create a task called “golden_model” that will fill the associative array by looping and filling the associative array with the expected values.
h. Drive the DUT with data to be written
i. Drive the data to read out the data, make your test-bench self-checking by calling a checking task that will compare the data read with the *data_read_expect_assoc* array. Maintain an error counter. Store the data read in the queue *data_read_queue.*

2. At the end of the test, print out the data read stored in the *data_read_queue* using while loop and pop_front method (search online how to do it). Also, print out the error and correct counter.

4) We will test the same ALU given in assignment 1 but using randomization and typedef enum.

Follow the following steps:

- Create typedef enum for the opcode at the top of the file before the class or the module
- Create a class to randomize all ALU inputs
  - Constraint the reset to be low most of the time
  - Use the typedef enum for the opcode variable
- Create the testbench module
  - Use the typedef enum for the opcode variable
  - Your testbench will use constrained randomization to drive the stimulus
  - Make the testbench self-checking
  - Monitor the output to display errors if occurred
  - Use a do file to run the testbench
  - Generate a code coverage report (100% design code coverage is expected. Less than 100% must be justified.)

Deliverables:

One .rar file having the following:

1. Design file
   a. If the design has bugs, then fix them otherwise upload the given design file.
2. Testbench file
3. Do file
4. Coverage report text file
1. PDF having the following
   a. **Clear and neat** QuestaSim waveform snippets showing the functionality of the design
   b. Branch, statement and toggle coverage report snippets with justification if you could not reach 100% coverage for the designs.
   c. Your PDF file must have this format <your_name>_Assignment2 for example Kareem_Waseem_Assignment2