Advanced VLSI Verification Exercise #2 (Not for Submission)

Exercise goals:

1. Practice SystemVerilog syntax, for verification code
    a. In Part 1, you will implement some functions that can be found in verification models.
    b. In Part 2, you will implement a class for an input transaction as well as a task that will generate many such transactions, with the concept of randomization.

Exercise Setup

1. Following the previous exercise, make sure to have Euclide setup and open.
2. Assuming you've done **Setting up Our Work Environment** in the last exercise, no need to do it again.
3. Repeat the steps in **Setting up a Project** from the exercise setup guide, with the exercise number being 3.

Download exercise2.tar from the Moodle, which contains the skeleton for **Part 2 of the exercise.**

Unzip the archive and copy the file to your new project directory (or to the text editor of EDAPlayground), and in Euclide refresh the project (right click on ex_vlsi_<x> in **Project Explorer → Refresh**). Add it to build.cud.

For Part 1 of the exercise, please create a new file and follow the questions below.

Questions

<u>Part 1</u>

1. Create an unpacked array of 8 **int**, called **array_1**. Use a for loop OR a foreach loop to populate the array like so:
   Even index locations will have double the index number
   Odd index locations will have three times the index number.

2. In SystemVerilog, you can create new data types of commonly used arrays/other types.
   For example, if I am using a lot of 8-int arrays like in question 1, I would prefer defining them as a new type:

   ```
   typedef int int_arr[8]; // defining the new type

   int_arr result; // creating a variable of the new type
   ```

   Create the following new types:
   a. An unpacked array of 16 shortints, type name = si_16t
   b. A vector of 8 bits, type name = b_8t

c. A 2D unpacked array of 4 by 4 bytes, type name = B_4x4t
d. A vector of **VECTOR_WIDTH** (please define this parameter as 128) bits, type name = vector_t.

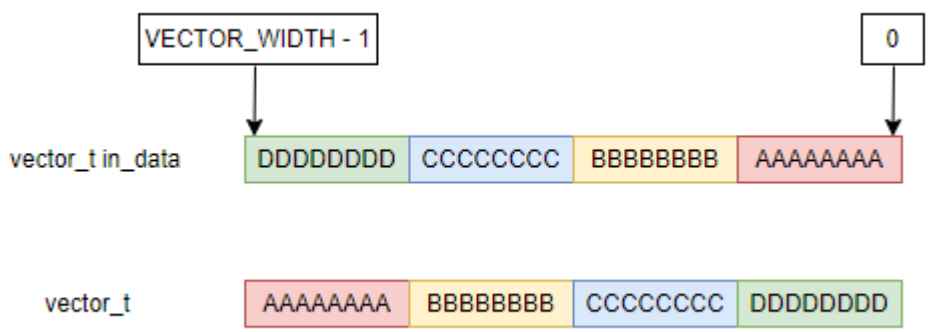
In the following questions, you might need to use the new types that you created.

3. A. Create a variable of the 4x4 byte array from Q2, called **primes**. Populate the array so it contains the first 16 prime numbers. Use direct assignment (`arr = '{.....}`)
B. Create a function called **is_prime**, that takes in a **byte** and returns a **bit**. If the input number is prime, the function should return 1 and otherwise 0.

4. Create a for loop where you iterate over the array from Q3. For each value, call your is_prime function from Q4. If the function returns 1, print "X is a prime" and otherwise "X is not a prime", where X is the value you are currently checking.

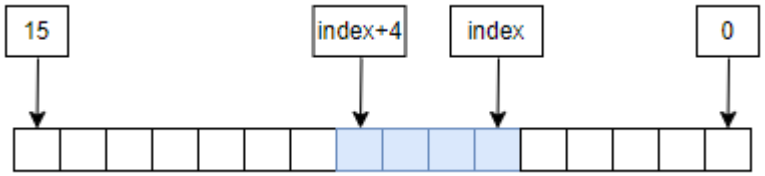
5. Create a function called **dot_prod**, that takes in two variables of the type '8 int array' (See Q2), and returns an int. This function should treat the int arrays as vectors and return their dot product.

6. Create a function called **rearrange_data**, that takes in a ***vector_t*** *in_data*(the typedef from Q2.d) and returns a ***vector_t.*** This function should reverse the input vector into the output vector, in chunks of **VECTOR_WIDTH/4**. To demonstrate:



The first row represents the input vector_t, where the LSB is on the right and MSB is on the left. As you can see, the different colors divide this packed array into 4 chunks. The goal is to re-arrange the data into the output vector_t as shown in the diagram.

Tip: in SystemVerilog there's a special syntax for accessing parts of a packed array if the accessing index is a variable. Let's say for example that we have a 16-bit packed array, and we want to access only 4 bits of that array, but starting from a location which is variable (not known from the start), as demonstrated below:



The following code demonstrated how to access the blue indices, either for writing (assigning a new value in those bit locations) or for reading (printing the value stored in that part of the vector)

```
bit [15:0] packed_array;
int index;
initial begin
    // assume index is computed somewhere else

    packed_array[index +: 4] = 4'b1101;
    // OR
    $display("data = 0x%h", packed_array[index +: 4]);
end
```

Part 2

For this part, we provide a skeleton file for you to add your code.
Download ex2.sv from Moodle and:

- If using EDAPlayground – copy its contents to testbench.sv
- If using Euclide – copy the file to your project directory ex_vlsi_2, and inside Euclide right click on the project explorer and refresh. Open the file in the editor.

Please only write code in the sections between Qx_START and Qx_END!

7. You're going to write a class representing an input item in a testbench used for verifying a computer memory system.

Address [31:0]	Data [7:0]	Parity [0:0]
0	hF4	1
1	'h90	0
2	'h00	0
3	'h46	1
...		
...		
...		
...		

In the diagram above you can see the structure of the memory. It has a 4GB address space (denoted by a 32-bit address), where each address points to a single memory **byte** (8 bits). In addition, there's one extra bit **parity**. Parity is 1 if the number of 1s in Data[7:0] is odd, otherwise it's 0.

Sometimes we call the input items 'transactions' since they represent a transfer of data, or in general – info. Our input item is described in a class called **transaction**.

- Add **randomizable** fields to the class – a 32-bit address and an 8-bit data.
- Add a **non-randomizable** field called parity – a single bit.

We want to randomize the address and data for every transaction, but not the parity. Think what would happen if we had randomized the parity as well.

In verification, we usually want to randomize the actual information sent to the design. In our case, that's the address and data. However, since most of the time we are verifying sequential systems, the **time** when we send our transactions is also important. To model randomization of time, we can add another **randomizable** field called **delay**, which will represent the number of clock cycles to wait between transactions.

- Add a 3-bit **randomizable** field called **delay**. We will use it to model delays of 0 clock cycles up to 7 clock cycles.

8. The skeleton file includes a signature for a **calc_parity** function. This function returns a bit (which is the parity value) and takes no arguments.
However, inside this function, you can use the class fields, like the address and data. Your job is to implement this function, so that it will calculate the parity for the **data** field and return the parity bit.

9. The skeleton file includes a signature for **new** function. This is a constructor. As we learned in recitation, we can customize a constructor to add any behavior that should happen when

creating the object. Your job is to customize the new function, by adding the two following behaviors to its implementation:

- Randomization – add a call to the built-in **randomize** function. This function will assign random values to the randomizable fields of our object. This function returns 1 if the randomization was successful, and 0 if it fails. While we don't expect it to fail, check that the return value is indeed 1.
- Assigning the parity – add a call to your function calc_parity and assign its return value to the **parity** field.

10. The skeleton file includes a signature for a **gen()** task and declares a variable of our transaction class.
    Your job is to implement this task, so that it runs an infinite loop that creates new transactions on the positive edge of the clock (provided to you as the **clk** variable). However, use the transaction **delay** field to create clock cycle gaps between consecutive transactions. Think what mechanism can help you achieve that.

    For example, if the transaction was randomized with **delay = 4**, then the next transaction should be created only after **5** clock cycles have passed.
    Note: delay = 0 means that two transactions are created on consecutive clock cycles. So you can think of delay = 0 as 'wait 1 clock cycle', delay = 1 as 'wait 2 clock cycles', and so on.

    To help you debug, you can add prints with $display. To print the current simulation time, use the built-in **$time()** function.