18-341: Logic Design and Verification

Homework 3: Random Things

Objective and Overview

The purpose of this assignment is to use the random number generation features of SystemVerilog. You will write a testbench, using random numbers for test vectors, that will test a memory controller (provided for you).

Schedule and Scoring

All homework assignments are equally weighted and will be scored out of 100%.

Start Date	3 Oct 2023
Due	11 Oct 2022 at 12:30pm

We will grade your homework based upon:

- **Fault Model (20%):** A discussion of what your fault model is. What do you think can go wrong and so what are you going to focus your testbench on testing for. Remember, you're not testing the memory — just the controller.
- **Testbench code (40%):** Your code, using random variables to generate all of the tests for the memory controller. Nicely written code, following the course coding standards.
- **Results (20%):** The output from running your testbench, showing the results for all four values of TBerrorSelect.
- **What's Wrong (20%):** For each error situation (TBerrorSelect), what is your educated guess as to what is wrong. Generally, the testbench won't diagnose the fault in great detail. Rather, it provides diagnostic information for a validation engineer (that's you in this exercise) to chase down the specific fault. For this assignment, you don't have to do much chasing. Just speculate, instead. We will grade you on how thoughtful your speculations are.

A Note about Collaboration

All 18-341 homework assignments are to be accomplished individually. All work must be your own.

Hints from others can be of great help, both to the hinter and the hintee. Thus, discussions and hints about the assignment are encouraged. However, the homework must be coded and written up individually (you may not show, nor view, any source code from other students). We may use automated tools to detect copying.

How to Turn In Your Solution

Use **`handin341`** to turn in your work. You should have:

- A nicely written report, in **`hw3.pdf`**, describing your fault model, what tests you decided to try (and why), and your guesses as to the cause of the fault.
- **`hw3.sv`**, your testbench code.
- **`hw3.sim`**, the output of vcs running your code.

Your Mission: Random Testing of the Memory Controller

The assignment is to develop a test bench for a memory subsystem that will receive packets of 8-bit data to be written into the memory. The packets are made up of several sequential bytes that include the following: a start byte indicator, a count, the base address for the data, the data, and a checksum. The memory subsystem then indicates when it is done (by asserting its **`done`** output) and if there was an error (by asserting its **`error`** output). See diagram at end, and details that follow. We provide the memory controller system, and you write the testbench.

The memory controller is in a protected file called **`/afs/ece/class/ece341/handout/hw3/mController.svp`**. The module to instantiate into your test bench file is **`loadMem`**, whose interface definition is:

```
module loadMem
 (input  logic [7:0] in,
  input  logic       clock, resetN,
  input  logic [1:0] TBerrorSelect,
  output logic       done, error);
```

The **`in`** input is an 8-bit bus on which an input "packet" to the memory controller is placed. The elements of the packet are defined as follows. Also, see the waveform at the end of this handout. Oh, and due to a quirk of the implementation, you should make sure you aren't sending **`X`** values into the **`in`** inputs after it comes out of reset.

`start` — indicated by in == 8'h01, a start packet indicator. Packet is arriving!

`count` — this is the number of data bytes that will be sent. The count includes only the data bytes.

`address` — this is the base address of where in memory to write the data bytes. The first data byte is written to this address, the second data byte is written to this address+1, etc.

`data bytes` — there could be between 0 and 19 data bytes sent, as specified by the previous count input, to be written into memory. Why 19? We don't want to look over reams of output. Why 0? Sometimes it is fun to troll the memory controller. (There won't be any data in this packet.)

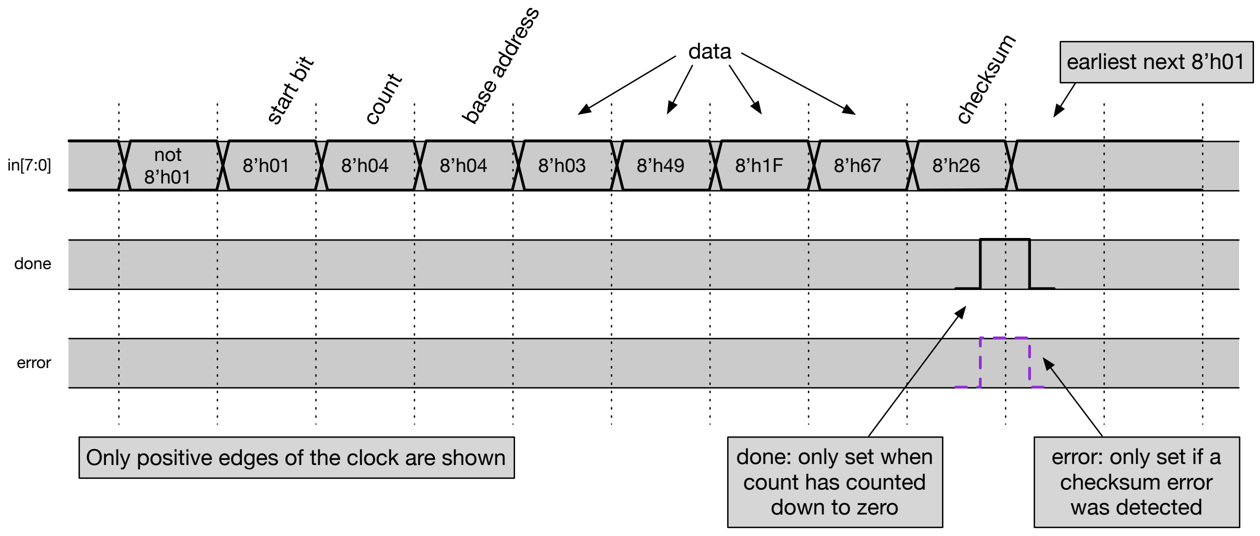
`checksum` — the value sent is the negative of the sum of the count+address+<all the data bytes>. When this value is added to the count+address+<all the data bytes> that the receiver calculates, the result modulus 256 should be 0. Anything else means there was an error in transmission (a "checksum error").

In the diagram shown, the packet being sent has a count of **`4`**, followed by the base address of **`4`**, followed by 4 data values (**`8'h03`**, **`8'h49`**, **`8'h1F`**, **`8'h67`**). Then the checksum follows which is -(4+4+3+49+1F+67). As a result of receiving this packet, mem[4] = 3, mem[5] = 49, mem[6] = 1F, and mem[7] = 67.

Before I talk about **`TBerrorSelect`**, let's look at the two outputs of the memory controller module.

`done` is asserted by the memory controller as shown in the diagram. When the checksum should be on the input, **`done`** is asserted. Being done is based on knowing how many data bytes are being sent.

`error` is asserted as shown in the diagram, but only if there was a checksum error. Notice that **`error`**, when it occurs, and **`done`** always appear at the same time.


start bit
count
base address
data
checksum
earliest next 8'h01
in[7:0]
not 8'h01
8'h01
8'h04
8'h04
8'h03
8'h49
8'h1F
8'h67
8'h26
done
error
Only positive edges of the clock are shown
done: only set when count has counted down to zero
error: only set if a checksum error was detected

An error? Yes, you might mess up some of the signals to make sure the controller is catching errors. After all, in real life stuff happens and you need to be able to detect when it does. The checksum is used by the memory controller to detect the errors.

The controller's **`TBerrorSelect`** input lets your testbench control the "faultiness" of the memory controller. You can set it to one of four values. 0 means that the memory controller is not faulty (as far as we know!). Values 1 - 3 will turn on a fault; see if your test bench can find them.

Assuming that you instantiated our **`loadmem`** module in your testbench with instance name **`fred`**, you can access the memory in the memory controller system as **`fred.M.M`**. It is defined in the memory controller as:

```
logic [7:0]   M[256];
```

You can read the memory to confirm the values in it, or write it to initialize it to some value (like all zeros).

What should your testbench do?

Your test bench should send a bunch of random packets of the form shown in the figure to see if it can find faults in the memory controller. You can assume that the memory itself is not faulty. So, the point here is to use the random features of SystemVerilog to do this. Do not use assertions — that's a later homework.

The idea of your test bench is not to diagnose the fault (It doesn't need to print "aha!, bit 2 of the adder is stuck at 1"), but to determine that there is a fault. In other words, some other values were incorrect or appeared at incorrect times. Just print some general indication of what value was wrong and what you expected.