ECE 571- Fall 2021
Homework #3

Submit your deliverables to your Homework #3 dropbox no later than 10:00 PM on Sunday, 21-Nov-2021. You have a 23-hour, 59 minute "grace period" to submit with no late deduction. After that your submission score will be reduced by 2 pts/day, with the absolute deadline being 3:00 PM on Wed, 24-Nov. No submissions for this assignment will be accepted after that. We will only grade the submission (and apply the late penalty) with the latest date stamp.

HDL Coding requirement(s)

Source code should be structured and commented with meaningful variables. Include a header at the top of each file listing the author and a description. You may use the following template:

```
////////////////////////////////////////////////////////////
// <filename>.sv - <one line description>
//
// Author:  <your name> (<your email address>)
// Date:    <date you created the code>
//
// Description:
// ------------
// <text description of what function the module performs>
////////////////////////////////////////////////////////////
```

Submission Guidelines:

Make a directory named ece571f21/<odinID>/hw3 for your source code and transcript. Add the code to display your simulation directory and a greeting and terminating message in the transcript to your testbench. The `tb_add_sub.sv` testbench provided in homework #2 includes this code. For a class size this large (about 40 students) we want to make sure we match the test results from your simulation with the code you simulated. Before you roll your eyes, let me assure you that I am speaking from experience.

IMPORTANT: Create a single `.zip` or `.rar` file containing your source code files, design report and transcript(s) showing that your implementation(s) simulate correctly. Name the file `<your odin ID> _hw3.zip (ex: rkravitz_hw3.zip)`. We will deduct points if your submission is incomplete or not in a single .zip file. You're `.zip` file can be organized more simply and should contain folders for each problem with an `hdl` directory containing all the SystemVerilog code you wrote or modified for the problem.

Deliverables:

You submission should include the following files in one folder:

- Source code for your memory controller module.

- Source code for your processor module.
- Source code for any SystemVerilog modules, packages, etc. that you changed.
- Transcript(s) of your successful memory controller test
- A 3 - 5-page design report (`.pdf`) describing your memory controller and processor (testbench). Provide technical details (flowchart, short code snippets with explanations, etc.) to help us understand your testbench code and results. Doing this should provide some practice for the Theory of Operation document you will write for your final project.

Problem Statement:

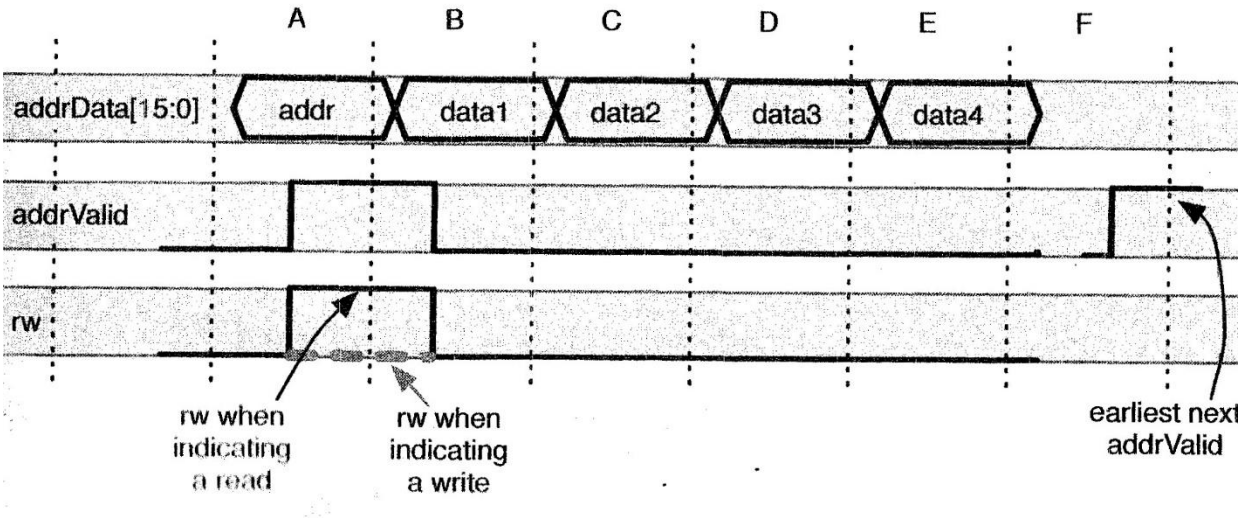
Original concept by Donald Thomas (Exercise 6.4, Logic Design and Verification Using SystemVerilog). Extensively reworked by Roy Kravitz. Revised by Roy Kravitz (several times) to simplify the architecture. This version is the simplest version, yet…but I still think you'll find it challenging and (I hope) fun to do.

For this homework assignment you are going to design, implement and test a SystemVerilog model for a tri-state bus-based processor/memory system. We will use a SystemVerilog interface to encapsulate the main bus between your processor (testbench) and the memory controller. You will connect your processor model to your memory controller through this interface.

You may collaborate with one other member of the class on this assignment to architect a solution, create test cases, etc. but your implementation must be your own. Please list your collaborator in the headers for each of the source code files you collaborated on.

The Memory Transaction Protocol

The timing diagram and descriptions below shows how the processor <-> memory bus works:



During the first cycle of a memory access the CPU drives the `AddrValid` and `rw` signals. The `AddrData` bus is implemented as a tristate bus because the `AddrData` bus is driven by both the CPU and the memory controller.

Let's do an example of a memory read. The processor (testbench) drives `AddrData` bus with a memory address and asserts `AddrValid` and `rw` (`rw` is asserted high on a read). This is state A on the timing diagram. At the next clock edge, the memory controller saves the address (the base address) and accesses the memory, returning the data from `mem[base address]` in state B. In state C the memory controller returns the memory data from `mem[base address + 1]`. In state D the memory controller returns the memory data from `mem[base address + 2]`. In state E the memory controller returns the memory data from `mem[base address + 3]`. At this point the bus

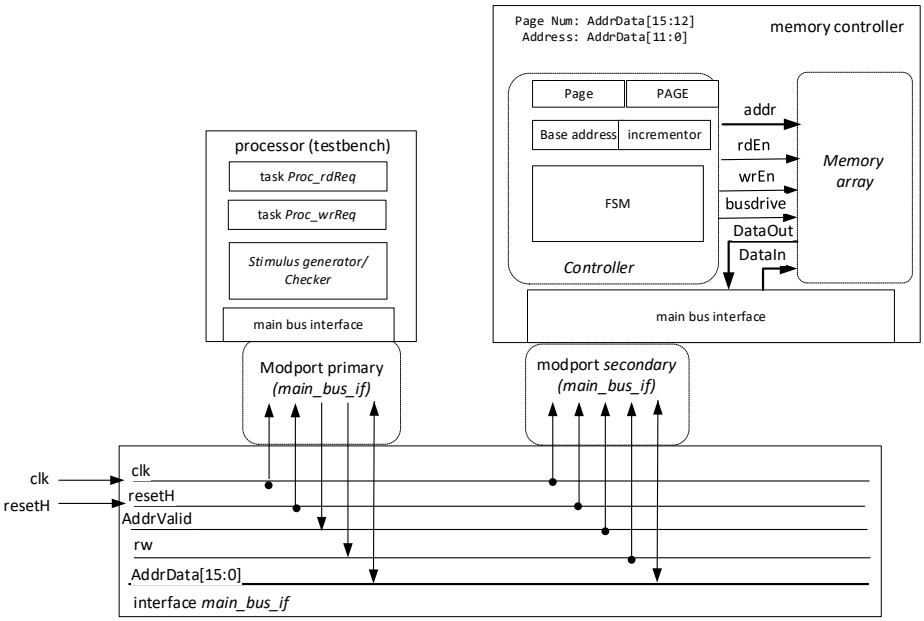
read is complete. The next bus transfer could start as early as state F with `AddrValid`, `rw`, and the `AddrData` bus being driven by the processor.

If the access is a write to the memory, `rw` is deasserted (low) when `AddrValid` is asserted (state A) and the processor drives the data item to write to the memory in states B-E. The memory controller writes the data items to the memory as it receives them. That is, the memory controller does not buffer the data items and then write them into the memory some time later; the data are written to `mem[base address]` to `mem[base address + 3]` on consecutive clock cycles.

Your memory controller should only respond to accesses to its `PAGE`. By default the memory is 4096 data items deep (lower 12-bits of the address), the `PAGE` portion of the address is the top 4 bits. Read or write accesses that extend beyond the 12-bit address should wraparound to 0. For example, a Read access to 0xFFE would return the memory data at 0xFFE, 0xFFF, 0x000, and 0x001.

The Target system:

The target system for this project can be described with the following block diagram:



The components of this block diagram are:

Interfaces:

- `main_bus_if.sv`: This interface contains the connections between the processor and memory controller. There are no methods (task and/or functions) included in this interface. The interface does, however, provide two modports. The `primary` modport is responsible for initiating bus transactions from the processor (testbench). The `secondary` modport is responsible for responding to the bus transactions with (in this case) the memory data. The SystemVerilog source code for this interface is included in the release.

Modules:

- `processor.sv` (testbench): This module is the Primary on the bus and originates read and write requests to the memory array through the `mainbus_if` interface. It implements the testbench for the system, and, as such, should create and make use of a `CPU_RdReq` and `CPU_WrReq` tasks to test your memory controller. You will write the source code for this module. You may use the starter code provided in the release or write your own module.
- `memory_controller.sv` (the memory array and the controller). The memory controller should use a PAGE parameter to specify the page (the upper 4-bits of the memory address) that it responds to. Accesses to pages other PAGE should be ignored. The memory controller can use a FSM to implement the main bus protocol and generate signals to access the memory array integrated into the module. It also includes registers for the base address and page and an incrementor to provide base address, base address +1, base address +2 and base address +3 to the memory array. You will design and implement this module. I have included some starter code in the release, or you can write your own module.
- `tc_mc_top.sv`: This module is the top-level module for the system. The module instantiates the interface, the memory controller, and the processor (testbench). It also generates the clock and has an initial block to toggle reset and get the system running. The module has been included in the release, but you may have to do some minor editing to match the hierarchy of the memory-related interfaces.

Packages:

- `mc_defs.sv`: This package includes the configuration parameters that I used in my implementation. I have included this file in the release – feel free to use it (or not) or change it.

Assignment:

A. (30 pts) Write a SystemVerilog model for the memory controller. The memory controller should communicate with the processor module as `Main Bus` Secondary. Make use of the SystemVerilog constructs we have discussed in class (package, typedef, enum, etc.) as appropriate.

B. (30 pts) Write a SystemVerilog model for a "processor." The processor should communicate as a `Main_bus` Primary. It would be prudent to try a few test cases that access memory locations outside of your selected page to check that your memory subsystem does not respond to memory accesses outside of the selected page. You can observe the results using `$display()` (or `$monitor()` or `$strobe()`) and manually check the results or you can make an auto-checking testbench, perhaps by implementing its own memory and comparing the results with the data returned by the memory controller. This could be a good opportunity to experiment with constrained randomization if you'd like but that is not required. It could also be a good time to write a `program` block instead of making it a module.

C. (15 pts) Simulate the processor/memory system. Save a transcript showing that your memory subsystem is working. Include the transcript with your deliverables. I included a copy of my test results and a waveform showing a write and read operation.

D. (15 pts) Write a 3-to-5-page design report describing your memory controller and processor modules. Provide technical details (flowchart, short code snippets with explanations, etc.) to help us understand your testbench code and results. Doing this should provide some practice for the Theory of Operations document you will need to include in your final project deliverables.

<finis>