ECE 571- Fall 2020
Homework #3

Create a single .zip or .rar file containing your source code files and transcripts showing that your implementations simulates correctly. Name the file <yourname>_hw3.zip (ex: rkravitz_hw3.zip).

Submit your deliverables to your Homework #3 dropbox no later than 10:00 PM on Sunday 22-Nov-2020. We will only grade the submission with the latest date stamp.

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>
////////////////////////////////////////////////////////////
```

Deliverables:

You submission should include the following files in one folder:

- Source code for your memory subsystem including your memory subsystem testbench for unit testing your memory subsystem code.
- Source code for your CPU/testbench and any other files you wrote. You do not need to include the `.sv` files included in the release unless you change them.
- Transcripts of your successful simulations of both the memory subsystem and the entire CPU <-> memory system.
- A 3 - 5 page design report (.pdf) describing your test strategy and 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:

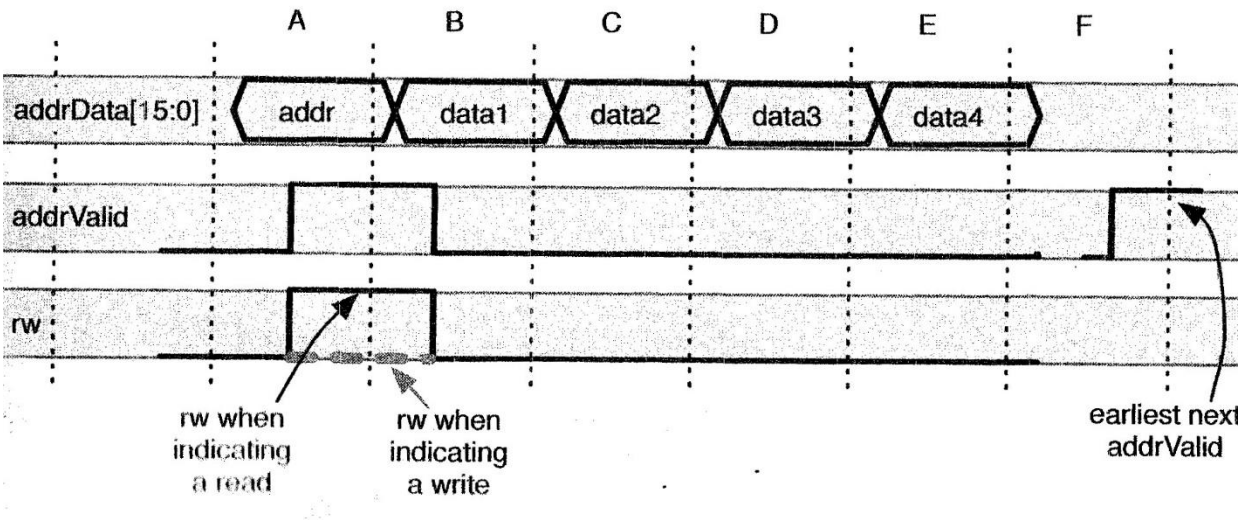
Original concept by Donald Thomas (Exercise 6.4, Logic Design and Verification Using SystemVerilog). Extensively reworked by Roy Kravitz. Revised by Roy Kravitz (several times) to simplify the architecture.

For this homework assignment you are going to design, implement and test a SystemVerilog model for a bus-based CPU <-> Memory system. We will be using SystemVerilog interfaces to abstract the processor and memory subsystems.

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 CPU <-> Memory bus works over time:



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 CPU drives `AddrData` 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 CPU.

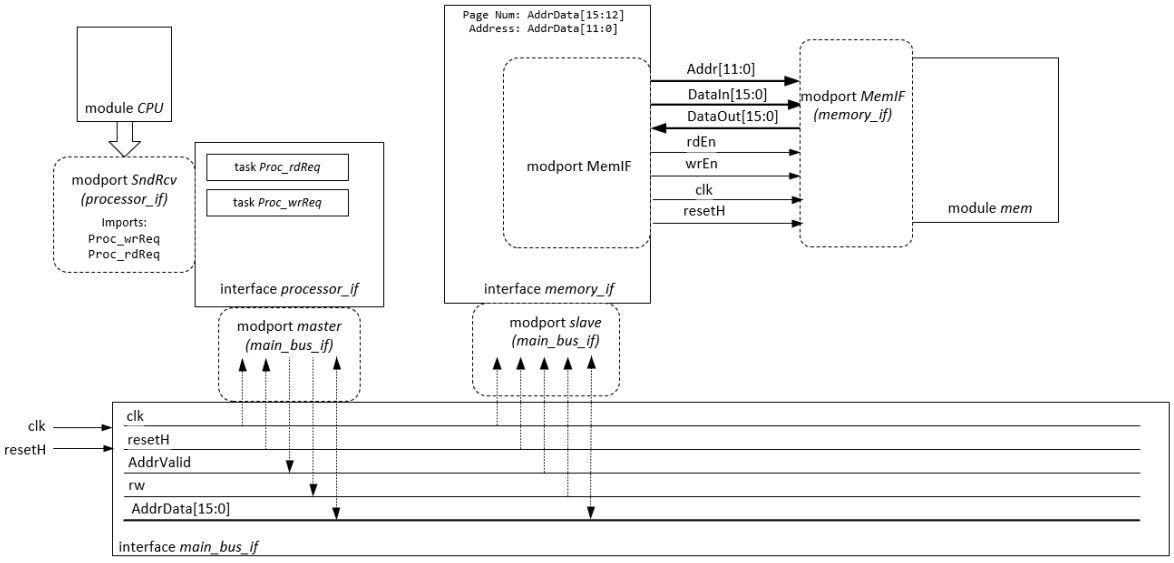
If the access is a write to the memory, `rw` is deasserted (low) when `AddrValid` is asserted (state A) and

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 all of the data items and then write them into the memory some time later. The data items 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 wrap around 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:` This interface contains the connections between the CPU and memory subsystems. There are no methods (task and/or functions) included in this interface. The interface does, however, provide two modports. The `master` modport is responsible for initiating bus transactions from the CPU/testbench. The `slave` modport is responsible for responding to the bus transactions initiated by the master. The SystemVerilog source code for this interface is included in the release.

- `processor_if`: This interface is responsible for initiating bus transactions (memory reads and writes) from the CPU/testbench model across the main bus to the memory interface. The SystemVerilog source code for this interface is included in the release.

The processor interface provides a transaction-level interface to the CPU in the form of two tasks:

- ```
  task Proc_rdReq
  // performs a 4 word memory read to "page" starting at "baseaddr"
  // the data from memory is returned in the 64-bit packed array
  // (four 16-bit words) "data"
  (
     input     bit [3:0]          page,
     input     bit [11:0]         baseaddr,
     output    bit [DBUFWIDTH-1:0] data
  );
  ```

- ```
  task Proc_wrReq
  // performs a 4 word memory write to "page" starting at "baseaddr"
  // the data to be written to memory is passed to the task in the
  // 64-bit (four 16-bit words) packed array "data"
  (
     input  bit [3:0]             page,
     input  bit [11:0]            baseaddr,
     input  bit [DBUFWIDTH-1 : 0]  data
  );
  ```

The two tasks are exported from the interface in the `SndRcv` modport. Here is the suggested signature for the interface and the `SndRcv` modport definition:

```
interface processor_if
(
   main_bus_if.master  M     // interface to processor is a master
);

modport SndRcv
(
   import Proc_rdReq,
   import Proc_wrReq
);
```

- `memory_if`: This interface is responsible for handling bus transactions originating in the `processor_if` interface. It performs the role of the memory controller in this system. The interface drives the ports of the memory array through a modport `MemIF` in response to requests from the CPU. When `rw` is asserted high the memory interface initiates a read operation to the memory and return the memory data on the `AddrData` bus. When `rw` is deasserted (low) the memory interface initiates a write operation to the memory starting at the address on `AddrData` and writing four consecutive locations with the data appearing on the `AddrData` bus (one word per cycle) after the address.

  The memory interface should use a PAGE parameter to specify the page (the upper 4-bits of the memory address) that the memory controller responds to. Accesses to pages other PAGE should be ignored. You will design and implement this module. *Hint: you may find the `processor_if` code useful as a reference. We have also included the source code for Thomas' three-and-out example in the release as an example of a producer-consumer application.*

Here is the definition for the interface and the modport to the memory array:

```
// global definitions, parameters, etc.
import mcDefs::*;

interface memory_if
#(
   parameter logic  [3:0]  PAGE = 4'h2
)
(
   main_bus_if.slave   MBUS      // memory controller is a slave
);

// interface to the memory array
modport MemIF
(
   input     Addr,
   input     DataIn,
   input     rdEn,
   input     wrEn,
   output    DataOut,

   // passed from main_bus to memory array
   // eliminates the need for memory array to be on main_bus
   input     clk,
   input     resetH
);
```

Modules:

- CPU/testbench: This module implements the Master on the bus and originates read and write requests to the memory array through the `memory_if` interface. It implements the testbench for the system, and, as such, should make use of the `Proc_rdReq` and `Proc_wrReq` tasks implemented in `processor_if`. You will write the source code for this module.

- MEM: This module implements the memory array. This module has been included in the release.

- TOP: This module is the top level module for the system. The module instantiates the interfaces, the memory array and the 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.

Assignment:

A. (20 pts) Write SystemVerilog model for the memory subsystem (`memory_if`). Connect this interface to the memory array and the main bus (`main_bus_if`). Make use of the SystemVerilog constructs we have discussed in class (package, typedef, enum, etc.) as appropriate.

B. (15 pts) Create a top level module for the memory subsystem and write a testbench to perform unit-level testing on the memory subsystem. This testbench does not have to be complicated; it is adequate to test a few memory read and memory write operations to the selected page by applying test vectors that drive the pins of the main bus. 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.

C. (10 pts) Simulate your memory subsystem including your testbench. Save a transcript showing that your memory subsystem is working at the unit level. In QuestaSim you can save a transcript using the GUI by clicking in the transcript pane and selecting File/Save Transcript As… Include the transcript with your deliverables.

D. (25 pts) Design and implement your CPU/Testbench and integrate it with the main bus and processor interfaces and your memory subsystem. Include test cases to check the functionality of the entire system. Your CPU/testbench should generate a transcript showing that the CPU <-> memory subsystem bus is working correctly. This may be accomplished by including self-checking code in your testbench. One way to do this (and you do not need to follow this approach in your implementation) is to include a memory array in the testbench that is written and read along with the bus. I wrote two tasks in my testbench. My `CPU_Write()` task does a memory write bus access to the system memory using the `Proc_wrReq()` task in the `processor_if` and also writes the data the testbench memory if the access is to the correct page. My `CPU_Read()` task performs a memory read operation across the bus using the `Proc_rdReq()` task in the `processor_if` and compares the results returned from the memory to the contents of the testbench memory. The testbench displays a message with the comparison results after each memory access. A transcript showing the results of my CPU/Testbench is included in the release. You do not need to model my testbench or my approach to the problem. The transcript is just an example of would be a reasonable test set.

E. (20 pts) Simulate your implementation. Save a transcript and include it in your deliverables.

F. (10 pts) Write a 3 to 5 page design report describing your test strategy and 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 Operations document you will need to include in your final project deliverables.

<finis>