Lab 5,6,7 - Pipeline Processor

In this lab you will build a pipelined version of the processor you developed in Lab 4. Labs 5 and 6 are only tested to ensure progress. Your processor should be complete by Lab 7.

The evaluation sheets for lab 5,6, and 7 are due in their respective lab sessions. The source submission of labs 5 and 6 will not graded but you are still required to submit your files every Friday. By the end of this lab you are expected to :

1. Create a block diagram of your entire design. No hand drawn, photocopied, or HDL Designer schematics will be accepted.

> NOTE
>
> 
>
> Any evaluation turned in without block diagram(s) of the design will receive a grade of 0.

2. Verify the operations of your CPU using the provided CPU wrapper and test bench.
3. Synthesize your design.
4. Verify the mapped version using the provided test bench.
5. Verify the mapped version on the FPGA.

1 Starting a New Project

You will now begin project 2. Issue the following commands to make the proper directory structure.

```
~> cd ece437
~/ece437> mkdir project2
~/ece437> cd project2
~/ece437/project2> dirset
```

Now you have a new project directory. Copy relevant VHDL files to your new source directory. It may be useful to copy your assembly files as well to test your processor.

2 Pipelining A Processor

Start by reading the sections on pipelining in the text book. You will find it useful to reuse most of your components from the single cycle processor.

You are responsible for implementing the ISA that is shown (excluding LL/SC, push, pop, nop, and org/chw/cfw) when you issue the command:

```
~/ece437/project2> asm -i
```

3 Provided VHDL Files

3.1 VarLatRAM.vhd and ram.vhd

The ram files (rami and ramd) are no longer useful. There is a new ram file provided `ram.vhd` This is your main memory. It contains 32k of memory 0 - 0x7FFC(0b0111111111111100).

> WARNING
>
> 
>
> You are only allowed to use one instance of ram in this project . Using rami and ramd from Lab 4 or creating two instances of the provided ram is strictly forbidden, and doing so will cause you to fail all tests on the grading script. In the real world, there is only on RAM for instructions and data (caches come later).

`VarLatRAM.vhd` is a wrapper for `ram.vhd`, and it simulates memory access latency as in real physical memory. The memory operations now take more cycles to complete.

Due to the multi-cycle latency of memory, you need to consider waiting for read data to come back from the memory (or write to be completed) before moving forward to the next instruction. You should design your pipeline properly to handle this latency (bubble, stall, etc.)

The halt (latency_override) signal going into the VarLatRAM must be driven by the halt signal that you have in your pipeline. This selects between the testbench address and your own dmem address (Lab 4 diagram). The purpose of this is to allow the testbench to dump memory more efficiently by disabling the access latency.

Be sure to read the code and comments in VarLatRAM.vhd so you understand how it works. It may be easier to start testing your processor with zero latency, but you will eventually need to test with different latency values.

> WARNING
>
> 
>
> Just because your design works with a particular latency does NOT mean it will work for others. Before submitting to the grading script, make sure your processor operates correctly for zero latency as well as several non-zero latencies. We can, and will, change the latency in your design before testing it.

NOTE

Any changes to VatLatRAM.vhd or ram.vhd (other than changing the LATENCY constant) must be approved by your TA.

WARNING

Again, don't name your top block "cpu"; the wrapper is named "cpu". Your top block must be named "mycpu" and your top level file needs to be named "mycpu.vhd".

DO NOT delete or comment out "hexout" and "dipin" signals from your cpu.vhd and tb_cpu.vhd or you may get a ZERO from the grading script.

3.2 New top-level files and new testbench

You will be using the files `tb_cpu.vhd`, `cpu.vhd`, and `cpuTest.vhd` as well as the files they depend on (`7segDecoder.vhd` and `cpuTest.pins`) and your own design `mycpu.vhd`.

The new testbench is designed to connect to the memory component directly from the given top level cpu.vhd. This means that your processor is now talking to memory through the top level portmap (notice more signals are in your top level now). The new testbench pre-loads meminit.hex into the RAM at the beginning of your simulation. Because of this, you do not have to synthesize your processor multiple times if you want to simluate with multiple asm files.

WARNING

Do NOT create a new instance of ram or VarLatRAM in your design as they have been port mapped in cpu.vhd. Thus, your design has to work with memory through the top level wrapper. In your Makefile, you should have something like

```
cpu.vhd : mycpu.vhd VarLatRAM.vhd
VarLatRAM.vhd : ram.vhd
```

NOTE

You can now take out the mux you implemented in your single cycle which arbitrates between dumpAddr and dmemAddr by halt since the new testbench takes care of the memory dump by talking directly to memory. If you look at the portmaps in cpu.vhd, you can see that you no longer need to connect dumpAddr from your processor.

3.3 icache.vhd and dcache.vhd

These are placeholders. They do nothing except pass values straight through. We provide these placeholders to facilitate easier cache integration in labs 8 and 9. Use of these files is optional but recommended.

4 Differences From The Book

4.1 Organization of the Processor and Memory

It will probably be useful to think of the processor and main memory as separate elements. This logical separation will prevent you from having to completely disassemble your design when caches are inserted.

4.2 Main Memory

The book assumes the existence of an icache and a dcache. We do not have this luxury yet (caches are developed in labs 8 and 9). You have only one main memory. This will cause a structural hazard (even with caches) when, in the same clock cycle, one instruction is accessing data from memory while a new instruction is being fetched. To solve this structural hazard, you must create an arbiter that selects the order of access to the memory when it gets two simultaneous requests (one instruction and one data). The arbiter should give the memory stage a higher priority over the instruction stage and the pipeline should stall to accommodate this hazard. This arbiter should sit between the main memory and the processor. In the future you will have the CPU connected to the caches, the caches connected to the arbiter, and the arbiter connected to the memory. From a processor view there will be two memory modules, but in actuality there is one memory with an arbiter managing the conflicts.

4.3 Write Enables

The book assumes no write enables for the pc and pipeline registers. It would be unwise for you to make the same assumption. They will prove useful for stalling when hazards are encountered.

4.4 Branches

Branches are not delayed in your design, meaning there is no branch delay slot. At minimum, you should use a predict not taken approach and squash bad instructions on taken branches. You are welcome to try other more complex prediction schemes.

4.5 The Halt Signal

The "halt" signal stops your processor from executing any new instructions. In the pipeline it takes a bit more effort to stop the execution. Note that when a halt is reached, the halt signal must remain high indefinitely so the test bench can dump the memory. The file cpu.vhd contains a mux to switch between your address to memory and the testbench address to memory on the halt signal.

IMPORTANT

It is essential that you are consistent with naming your files. Give all files the exact same name as the entity contained within the file. Also, make sure that your design is accurately ported to the cpu wrapper.

IMPORTANT

Put resets on all of the registers in your design even if they aren't required for proper logical operation. There are essential signals from your design the testbench uses, and it is possible that the testbench will not write the meminit.hex file into memory properly if there are undefined signals on its input. If this happens, simulating your mapped processor will require you to re-map it for each asm file.

5 Turning In

It is important that you make sure that your `Makefile` works, and that all dependencies are correct. Also make sure your cpu.vhd and tb_cpu.vhd are the same as given on the website, and that your own top level file is named mycpu.vhd. Type the following from your project2 directory:

```
~/ece437/project2> make clean
~/ece437/project2> make lab5
```

Now invoke vsim, load your design and run the simulation for the the varies asm files. Re-Diff the `memdump.hex` files to make sure you are submitting the working version of your design.

Lastly, from your `project2` directory, type:

```
~/ece437/project2> submit -p lab5
```

6 Hardware Test

As in Lab 4, you will be required to show proper operation of your pipelined processor on the FPGA by demonstrating proper output using Quartus.