ECE 401 Project 1 Fall 2016

Description

In this project you are going to get familiar with the high level design of a five stage pipelined MIPS processor by studying an existing high-level description in Verilog. The code is provided has had certain important portions removed. Your job is scrutinizing the code in order to extract the component structure of the existing architecture, and adding the necessary components to enable it to run correctly.

Requirements

In this project you are asked to complete the following tasks:

- Extract all the existing components from the code and draw an illustrative diagram for the processor architecture.
- Find and complete the following missing portions of the architecture:
  - The PC does not get updated.
  - Branch and jump destination addresses do not get calculated.
  - Register writes do not occur.
  - Memory reads and writes do not work properly.
- Add forwarding/bypassing logic to the architecture and correspondingly update the diagram.
- Create a project report (details are enumerated below).

Other than the issues mentioned above, there are no bugs in the pipeline. Decoder.v and ALU.v are guaranteed to be complete and correct. Unless specifically stated in the comments, the bugs are caused by omission of needed code. All bit strings (eg: 6’b101010) used as matches in case blocks are correct (match what is specified by the comments).

To test your design, you will run the test programs provided for this project. For full credit, the test applications in the list below must all execute correctly. Substantial partial credit will be awarded for being able to get any test application to run correctly. Traditionally, *noio* is the easiest one to get working.

Applicaiton	Instruction Count	Cycle Count
noio	2081	2112
file	95215	95282
hello	95705	95760
class	97177	97232
sort	104924	104987
fact12	110820	110923
matrix	137286	137401
hanoi	201667	201842
ical	216479	216534
fib18	305749	305804

Note that the cycle count is always higher than the instruction count. Please explain this discrepancy in your reporter, along with the cycle and instruction counts you observed, if different.

Source Code

Download

The source code for this project can be found on blackboard. The tarball includes the necessary c++ files (collectively called sim main) that emulate the operating system and memory, and are responsible for driving your MIPS processor.

To decompress the source tarball, you can use the following command:

```
tar xvjf ece401-project1.tar.bz2
```

After decompressing the source, you will have a directory named ece401-project1. In this directory will be the following contents:

- **verilog/**

  This folder contains the verilog source files for your processor. If you need to add any additional verilog source files, please put them in this folder.

- **sim_main/**

  This folder contains the c++ source files for the simulator. You should not need to make any changes to them.

- **project1_instruction.pdf**

  A copy of these instructions

- **LWLandLWR.pdf**

  It explains the expected behavior of LWL and LWR. For more information, you can check mips-isa.pdf on the blackboard.

- **decoder.csv**

  A list of MIPS instructions recognized by the decoder, and the flags that are set for each instruction. These flags are used by the Instruction Decode stage. This file is generated from decoder.v; changes made to it do not propagate.

- **Makefile**

  A make-compatible build script used to compile your processor. (See the Compile section below for how to use this file.)

Verilator

You do not need to install Verilator on your system; the build system included with the product will download and compile a known-to-work version for you.

Test

To test your processor, you will need compiled MIPS binaries; sim_main accept System V ELFs. (ELFs are sued on Linux the way the Portable Executables are used on Windows.) You can download and extract the tests automatically by running the following commands:

```
cd ece401-project1     # get into ece401-project1 folder

make tests             # download and decompress the tests
```

Alternatively, you can download the tests from: *http://www.cs.rochester.edu/~swang/ece401/ece401-tests.tar.xz* and then run:

```
tar xvjf ece401-tests.tar.xz
```

You will then have a *tests* directory. Inside, there will be a *cpp* subfolder containing compiled versions of the c++ test applications. (These are the same test applications mentioned above in the requirements section.)

There will also be an *asm* subfolder containing various small tests intended to target specific issues that may be encountered while debugging your pipeline.

Each test will include the compiled ELF, a text file with the disassembly of the compiled version, and the source code used to generate the ELF.

Compile

After getting the source files and decompressing them, you should have a directory named *ece401-project1*. To compile the source, enter this directory (type *cd ece401-project1*) and then run *make*. If necessary, the build system will download and compile Verilator; it will then compile your processor. If the build process does not complete successfully, there will probably be some error messages from Verilator.

By default, Verilator will treat warnings as fatal errors, and refuse to finish the compile if there are any warnings. You can edit the *Makefile* to tell Verilator that it should ignore the warnings and keep going. (There are comments in the makefile to indicate what should be changed.)

Simulate

Once the build completes successfully, you should be able to run *./VMIPS*, which is the compiled MIPS simulator using your verilog code. To run a test, you will need to tell VMIPS which test application to load.

For example:

```
./VMIPS -f tests/cpp/noio     #run the noio test
```

VMIPS supports many options. You can get a complete list by running *./VMIPS -h*. Of particular interest will be the options below:

- **-d [number]**
  Number of cycles to simulate before halting. If, for example, you know that your simulation will run properly for 100 cycles, you can use this to speed through those first 100 cycles. If you do not want to stop after a certain number of cycles, set this to an extremely large positive number (1 million is good). After reaching the specified number of cycles, the simulator will drop into single-cycle mode.

- **-b [number]**
  A program counter to use as a breakpoint address (you can specify a hexadecimal number by prefixing it with “0x”, eg: “0x13A4B2F4”). When the Instruction Fetch stage requests a given address, and that address matches this address, the

simulator will drop into single-cycle mode, as if the cycle limit was reached. Once the cycle number specified by -d is reached, the simulator will pause, even if it hasnt reached this address. The simulator will also pause if the CPU attempts to execute an instruction at address 0x00000000.

Once the simulator reaches single-step mode, it will wait for further instruction. To step by a single cycle, just press enter. You can also provide another breakpoint address by specifying a new PC address. (As with the -b option, you can specify a hexadecimal address by prefixing it with “0x”.)

sim_main creates three output files when run:

- **stdout.txt**
  This contains anything that the test application wrote to standard output (file descriptor 1). *sim_main* mirrors text output here in addition to outputting it to the terminal so that you can review it. Unencumbered by verbose *$display* output. All test applications other than *noio* will contribute to this file.

- **stderr.txt**
  This contains anything that the test application wrote to standard error (file descriptor 2). Usually, attempts made to write to stderr are going to be due to bugs in the processor.

- **memwrite.txt**
  This will contain a list of reads and writes to main memory. Because *sim_main* evaluates memory accesses twice per clock, each request will usually be shown twice.

Turn-in

Your submission should be in the form of a tarball (.tar or .tar.bz2). You must include all files necessary to compile and run your processor, and the accompanying report (see below). To create the tarball, you can use a command like this:

```
cd ece401-project1

tar -cvjf usernames.tar.bz2 verilog/ report.pdf sim_main/ Makefile
```

The tarball you create should be submitted via blackboard. If you are working with a partner, the submission should specify who the partner is. Only one submission is needed per group.

You must provide a project report explaining what you did, and explaining how you did it. You should specify what is and is not working. If something does not work, try to explain why. If you have made any changes to sim_main, you must justify them in your report.

The report must be submitted in PDF format. While it is preferred for the processor diagram (mentioned in the requirements section above) to be in the same PDF as the report, it can be in its own file, and may be in PNG or JPEG format.

Advices

- Start the project early, otherwise you won’t be able to finish it before deadline.
- As said in lecture, draw an illustrative diagram for the design.
- Take advantage of the *asm* test programs to diagnose why the pipeline may be malfunctioning.
- Use *$display* to print out messages on the screen as needed.
- Add forwarding/bypassing logic to the processor.

Plagiarism

What you submit must be your own work (or that of your partner, if you are working in a two-person group). It is permissible to use general code snippets having nothing to do with MIPS or microprocessors (eg: bit twiddling methods or module templates) that are found online as long as you comment the code to attribute its source. You are also permitted (and encouraged) to discuss ideas with other groups, but you must not share code to avoid accidental appropriation.