ECE 401 Project 2 Fall 2014

Description

In this project you are going to get familiar with designing a cache system for an in-order scalar MIPS processor. The code for in-order processor is provided. Your job is scrutinizing the code in order to extract the existing interfacing ports and adding the required cache components. You are free to use your platform from the previous project if you made it pass all the test programs.[1]

Requirements

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

- Compile and run the provided code for all the test programs and immediately report if you see any problem.
- Add separate L1 caches for instruction and data memories as described in the table below. Timing details are discussed further in their own section.

	Instruction Cache	***Data Cache***
Size	32KB	32KB
Associativity	Direct-mapped	2-way
Block size	32B	32B
Access Latency	1 cycle	1 cycle
Miss Penalty	10 cycles	10 cycles
Write Policies	N/A	write-back, write-allocate

- You can assume memory accesses resulting from cache misses by the instruction and data caches will not conflict with each other and can be serviced in parallel.
- Report all the issues regarding accessing to the memories that you solve in your project (and how).
- Assuming a memory access delay in the original processor (without caches) the same as miss penalty in the table above, compare the IPC for the entire test programs on both designs.
- Create a project report (details are enumerated below).

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

Application	***Instruction Count***
`noio`	2081
`file`	95215
`hello`	95705
`class`	97177
`sort`	104924
`fact12`	110820
`matrix`	137286
`hanoi`	201667
`ical`	216479
`fib18`	305749

Memory Timing

The timing details for the cache assume that data present in the cache can be retrieved in one cycle. This is the same access latency that was observed in Project 1; after a request is present (on `posedge CLK`), the data will be in the output register on the immediately following `posedge CLK`. This will be necessary to fetch an instruction on every cycle.

[1] If you do use your code from project 1, be sure to modify the Makefile to set `PROJECT=2`.

The ten cycle latency for cache misses can be construed as follows:

1. On the first cycle, the cache checks to see if the data is available. Since it isn't, it makes a request to the lower-level cache (L2 or main memory).
2. Return word 1 to cache On the second cycle, the lower-level cache finds the data and starts to return it. On a normal system, this would take longer than one cycle.
3. Return word 2 to cache Since the data bus is only one word (32 bits/4 bytes) wide, only one word is available on each cycle. To retrieve the entire line will take another seven cycles.
4. Return word 3 to cache
5. Return word 4 to cache
6. Return word 5 to cache
7. Return word 6 to cache
8. Return word 7 to cache
9. Return word 8 to cache
10. The cache line is now populated, and the cache services the original request.

`sim_main` will simulate the time needed to send 8 words by refusing to process block read/write requests sooner than every 8 cycles[2]. It is permissible to modify `sim_main` to match timing requirements of your cache, but you must still maintain the 10-cycle-per-miss latency. (You also must document all changes made to `sim_main` and the reasons for them.)

Note that writes to memory are not subject to the 10-cycle penalty. You can assume that this is because of an infinite write buffer between the cache and `sim_main`, or you can assume that waiting 20 cycles to read data because of the need to evict a dirty line is just too long.

Extra Credit

This lab includes the opportunity to earn up to 45 bonus points. (You can compare these relative to the project itself, which is worth 100 points.)

Twenty bonus points are available for implementing a shared L2 cache that is used by both L1 caches. It should be a 4MB cache that is 8-way set associative, with write-back and write-allocate policies. The block size is still 32 bytes, and the access latency is 3 cycles. Read requests that need to go to `sim_main` will still need 9 additional cycles (12 cycles total). It is up to you to code the appropriate delays into your cache.

An additional twenty bonus points are available for implementing cache optimizations such as early restart and critical word first.

Finally, five bonus points are available for implementing Load Linked and Store Conditional, and running `sim_main` with the `-l 0` flag (to test the `cpp` programs without the LL/SC emulation).

The bonus points specified here are also applicable to future projects, and will have the same value that they had for this project. However, you can only earn the bonus points for a given piece of extra credit once. (If you add an L2 to this project, and then also have an L2 in project 3, you only get 20 bonus points; not 40 bonus points.)

Also, as a general policy, if you can come up with something else that you would like to add to the processor for bonus points (eg: exceptions, floating point, ...), please talk to Isaac. There will probably be bonus points for doing it.

Downloads

Verilog source

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.

If you have trouble downloading the source tarball on an ECE machine, you can also use the following command to retrieve it:

[2]The code governing this is in `sim_main.cpp` in the function `mem_process_request`

```
cp /usr/ece/www/users/irichter/ece401/ece401-project2.tar.bz2 ~/
```

You should then find `ece401-project2.tar.bz2` in your home directory.

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

```
tar xvjf ece401-project2.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.

`project2_instructions.pdf`
: A copy of these instructions

`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.

Tests

To test your processor, you will need compiled MIPS binaries; `sim_main` accepts System V ELF[3]s. You can download and extract the tests automatically by running the following commands [4]:

```
#Enter the ece401-project2 folder
#If you are already in that folder, just
#skip to the next command.
cd ece401-project2

#Download and decompress the tests
make tests
```

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.

[3] ELFs are used on Linux the way the Portable Executables (EXEs) are used on Windows.

[4] Alternatively, you can download the tests from: `http://www.ece.rochester.edu/~irichter/ece401-tests.tar.xz` and then run `tar xvJf ece401-tests.tar.xz`

Compile

After getting the source files and decompressing them, you should have a directory named `ece401-project2`. To compile the source, enter this directory (type `cd ece401-project2`) 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.

`cachewrite.txt`
This will contain a list of reads and writes to the data cache. This is somewhat analogous to `memwrite.txt`, but is for requests that go to the caches. It makes use of the various `_2DC` and `_fDC` wires in `MIPS.v`, so you probably should be careful how they get modified. (To avoid compilation errors and ensure that the file contents are correct.)

Some Advice

You may find the following advice useful for doing this project.

1. Start the project early; right now is a good time. Otherwise you won't be able to finish it before deadline.
2. Draw a timing diagram for how the caches will operate.
3. Iterate on modification and verification of the design using the provided test programs.
4. Use `$display` to print out messages on the screen as needed.
5. Take advantage of the `asm` test programs to diagnose why the pipeline may be malfunctioning.
6. You may reduce the total of code you need to write by creating parameterized modules[5].
7. System calls will probably work best if caches are flushed first.
8. Both partners should contribute.

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:

```
#Enter the ece401-project2 folder
#If you are already in that folder, just
#skip to the next command.
cd ece401-project2

#Compress the Verilog files, your report,
#sim_main, and the Makefile
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.

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.

[5] See `http://www.asic-world.com/verilog/para_modules1.html` to learn how to parameterize a module.