Lab 3: Pipelined ARM

Assigned: Wed., 2/5; Due: **Fri., 2/21** (Midnight)

Instructor: Onur Mutlu
TAs: Rachata Ausavarungnirun, Varun Kohli, Paraj Tyle, Xiao Bo Zhao

"It is comparable to a pipeline which, once filled, has a large output rate no matter what its length. The same is true here. Once the flow is started, the execution rate of the instructions is high in spite of the large number of stages through which they progress."
Chapter 14. The Central Processing Unit. Erich Bloch.
Planning a Computer System: Project Stretch. IBM. McGraw-Hill, 1962.

1. Introduction

In Lab 2, you implemented single-cycle ARM machine. In this lab, you will implement a *pipelined* ARM machine. The pipeline must consist of five stages as discussed in class and covered in a textbook (Patterson and Hennessy).

Now that multiple instructions can be "in-flight" at the same time, you must detect *dependencies* between instructions and handle them correctly. In this lab, we will consider only data dependencies and ignore control dependencies. Specifically, you will implement *two* pipelined MIPS machines that handle data dependences in two different ways: *stalling* and *forwarding*. Both MIPS machines must be correct and synthesizable.

Warning. This lab is difficult. We strongly encourage you to get started early.

Extra Credit. "Top" students will receive extra credit (up to 20%) for implementing the highest performing MIPS machines. Only correct and synthesizable implementations will be eligible.

2. Specifications of the MIPS Machine

2.1. Architecture

- **Instruction Set.** The machine supports all ARM instructions specified in Lab 1, *excluding* those related to control-flow: `B`, `BL`. As shown in the following table, there are 23 ARM instructions that the machine supports.

ADC	ADD	AND	~~B~~	BIC
~~BL~~	CMN	CMP	EOR	LDR
LDRB	MLA	MOV	MUL	MVN
ORR	RSB	RSC	SBC	STR
STRB	SUB	TEQ	TST	SWI

- **System Call Instruction.** Terminates the program (same as Lab 2). In addition, for debugging and grading purposes, you must ensure that the contents of the register file are dumped out in the **same format** as Lab 2.
- **Exceptions.** No support (same as Lab 2).

2.2. Microarchitecture

Unlike a single-cycle microarchitecture, a pipelined microarchitecture divides the "work" required to execute an instruction across multiple cycles. Each cycle corresponds to a *stage* within a pipeline.

The major advantage of a pipelined microarchitecture is that it can execute multiple instructions in parallel: multiple instructions can be in the pipeline at the same time, albeit at different stages.

Pipeline Stages.

For this lab, you must implement the following *five-stage pipeline*. Please ensure that there are exactly five stages. Please ensure that each stage does exactly what it is supposed to do (no more, no less). For example, as long as the memory is accessed only during the `MEM` stage, you are allowed to generate control signals (or perform other bookkeeping activities) for other stages. Later on, for extra credit, you will be allowed to design your own custom pipeline if you want.

Stage	Specification
1. IF	Instruction fetch
2. ID	Instruction decode and register file read
3. EX	Execution or memory address calculation
4. MEM	Memory access
5. WB	Writeback to register file

Handling Data Dependences.

For this lab, you must implement *two* five-stage pipelines, each of which handles data dependences in a different way. You will submit both of these implementations *separately*.

1. **Stalling.** When a data dependence is detected, simply prevent later instructions from entering/progressing through the pipeline. This leads to idle pipeline stages referred to as "bubbles". *For this lab, your implementation must stall only when necessary.*
2. **Forwarding.** When a data dependence is detected, allow an earlier instruction to send data directly to a later instruction even *before* the data has been written back into the register file. *For this lab, you must forward data into the* ***end of the decode stage****. Your implementation is still allowed to stall, but only when stalling cannot be prevented by forwarding data.*

3. Lab Resources

3.1. Source Code

The source code is exactly the same as Lab 2, except for one difference. We do not provide a `src/` folder. Instead, please use your own `src/` that you completed for Lab 2. Lab 2 source code is available at: `/afs/ece/class/ece447/labs/lab2`. Do NOT modify any files or folders unless explicitly specified in the list below.

- `Makefile`
- `src/` (**NOT PROVIDED; PLEASE USE YOUR OWN FOLDER FROM LAB 2**)
- `447src/` – Supplementary source code.
  - `regfile.v`: An array of multi-ported registers.
  - `arm_mem.v`: The memory module.
  - `testbench.v`: The testbench.
- `447inputs/` – Example test inputs.
- `inputs/` (**MODIFIABLE**) – You are allowed to add your own test inputs.
- `outputs/` (**MODIFIABLE**) – You can direct your outputs to here.

- `447ncsim/` – Config. file for the `ncsim` tool.
- `ncsim/` (**MODIFIABLE**) – Ignore. If you really want, you can implement a customized config. file.
- `dc/` (**MODIFIABLE**) – Config. files for the `DC` tool. You have to add any new source files in here, according to dependencies.
- `447util/` – Testing script for simulations.

3.2. Software Tools

We provide software tools for compiling, simulating, and synthesizing your ARM machine. In order to use the tools, remotely log into an ECE server (e.g., `ece[000-008].ece.cmu.edu` from off-campus and `ece[009-031].campus.ece.cmu.local` from on-campus) and execute the following command in your bash shell. The command will modify your shell's environment variables to set up the correct AFS paths to the tool binaries as well as the licenses for using them. *We highly recommend doing the lab on the ECE servers.*

```
$ source /afs/ece/class/ece447/bin/setup447
```

3.3. Makefile

We provide a `Makefile` that automates the tedious process of compiling, simulating, verifying, and synthesizing your (System)Verilog implementation. Typing `make` without any targets or options will invoke the help screen. **Please read this help screen carefully!**

```
$ make
```

4. Getting Started & Tips

4.1. Getting Started

1. Review the material on pipelined machines from the lecture notes and the textbook (Patterson and Hennessy).
2. Sketch a high-level block diagram of how you want to organize your pipelined machine. Refine this block diagram as you make progress on your implementation. This block diagram will be collected.
3. Copy your `lab2/` folder to a Lab 3 (`lab3/`) directory.
4. Implement just enough of the pipeline so that the machine can execute the following stream of instructions where **there are no back-to-back instructions**: `ADD`, `NOP`, `NOP`, `NOP`, `NOP`, `ADD`, `NOP`, `NOP`, `NOP`, `NOP`, ...
   - While there is no `NOP` machine instruction defined in the ARM ISA *per se*, you can use an instruction that basically does nothing and does not interfere with the other instructions in the pipeline (like 0+0).This is not required but would help debug. Your machine should then "execute" this instruction by doing nothing except for incrementing the program counter. This is a convenient way of forcing an artificial bubble into your pipeline.
   - Create a test input called `inputs/addiu_nop.s` which is the same as the example `addiu.s`, except that four `nop` assembly instructions are inserted after every assembly instruction. Simulate this test input to check whether your rudimentary pipeline is correct.
5. Implement just enough of the pipeline so that the machine can execute the following stream of instructions **where there are no data dependences**: `ADD`, `ADD`, `ADD`, ...

- Create a test input called `inputs/addiu_addiu.s`, which is the same as the example `addiu.s`, except that no register is ever read by a later instruction if the register had been written to by an earlier instruction (i.e., no RAW data dependences). Simulate this test input to check whether your rudimentary pipeline is correct.

6. Augment your pipeline with logic for *detecting data dependence* and *stalling* so that the machine can execute the following stream of instructions **where there are data dependences**: `ADD`, `ADD`, `ADD`, ...
   - Simulate the example `addiu.s` to check whether your rudimentary pipeline is correct.
7. Once you have *completely* implemented the stalling version of your pipeline, use it a basis for implementing the forwarding version. Remember, you will submit *both* versions.

4.2. Tips

- **Read this handout in detail.**
- **Please ask questions to the TAs using Piazza.** (The link is available on the course website.)
- When you encounter a technical problem, please read the logs/reports generated by the software tools in the `outputs/` folder.
- Your (System)Verilog code will require many wires. Please adopt a consistent scheme for naming them.
- The system call instruction should terminate the program only when all other preceding instructions have completed execution.
- When coding in (System)Verilog, we recommend having a separate module for each pipeline stage (maybe even in separate files). This way, it is easy to draw a separate block diagram for each module.
- Make sure your (System)Verilog implementation is synthesizable.
- For debugging and grading purposes, the system call instruction must dump out the contents of the register file in the same format as Lab 2. Since the `$display()` Verilog function is not synthesizable, you must include `synthesis translate_off` and `synthesis translate_on` (in commented form) before and after the code block that includes `$display()`. Please refer to the source code from Lab 2 to see how it is done.
- When synthesizing your implementation, check for warnings that mention the word "latch". This is most likely due to a bug in your (System)Verilog code (e.g., incomplete `case` or `if` statements).
- The cycle time of your implementation is determined by the critical path of the slowest stage. While a short cycle time is not necessary to achieve a perfect score on this lab, you should still try to keep your cycle time as short as possible. Keep in mind that cycle time is not the whole story. Forwarding paths may actually increase your cycle time, but also improve overall performance (by reducing stalls and decreasing CPI). Finally, its important to make your processor work first, and then make it work fast. *Premature optimization is the root of all evil.*

5. Submission

5.1. Block Diagram

Please submit two *hardcopies* of the *computer-drawn* diagram of your MIPS machine: one for the stalling version and the other for the forwarding version. The TAs will collect these diagrams during the Lab Sections and you will be responsible for explaining your design decisions to the TAs.

The diagrams should be at the same level of detail as you saw in the textbook and lecture notes.

All major structures (e.g., registers, muxes) should be drawn, as well as boxes for the various control logic blocks. Label all wires with their names and widths. We suggest using different colors (or line styles) to differentiate control- and data-path wires. Putting in an extra effort to keep this diagram neat and clean will definitely pay off. It is okay to utilize plenty of white space and to span multiple sheets of paper. We recommend using Inkscape (cross-platform), Adobe Illustrator (Windows/Mac), or Microsoft Visio (Windows/Mac).

5.2. Lab Section Checkoff

So that the TAs can check you off, please come to any of the lab sections *before* Sat., 3/1. **Note that you must be checked off for both versions of your implementation (stalling and forwarding) and also the extra credit if you choose to submit it.** You can get them checked off separately in different lab sections or all in one sitting. Please come *early* during the lab section. During the Lab Section, the TAs may ask you:

- to answer questions about your implementations,
- to simulate your implementation using various test inputs (some of which you may have not seen before),
- to synthesize your implementation.

5.3. Source Code

Make sure that your source code is readable and documented. Please submit the lab by executing the following commands. **Make sure you format your folders as specified. We will penalize wrong submissions.**

Stalling Version.

```
$ cp -r src /afs/ece/class/ece447/handin/lab3/andrewID/stalling
$ cp -r inputs /afs/ece/class/ece447/handin/lab3/andrewID/stalling
```

Forwarding Version.

```
$ cp -r src /afs/ece/class/ece447/handin/lab3/andrewID/forwarding
$ cp -r inputs /afs/ece/class/ece447/handin/lab3/andrewID/forwarding
```

5.4. README

In addition, please submit two `README.txt` files. To submit these files, execute the following command.

```
$ cp README.txt /afs/ece/class/ece447/handin/lab3/andrewID/stalling/README.txt
$ cp README.txt /afs/ece/class/ece447/handin/lab3/andrewID/forwarding/README.txt
```

The `README.txt` file must contain the following two pieces of information.

1. A high-level description of your design.
2. A high-level description of the critical-path of your synthesized implementation.

It may also contain information about any additional aspect of your lab.

5.5. Late Days

We will write-lock the handin directories at midnight on the due date. For late submissions, please send an email to `447-instructors@ece.cmu.edu` with tarballs of what you would have submitted to the handin directory.

```
$ tar cvzf lab3_stalling_andrewID.tar.gz src inputs README.txt
$ tar cvzf lab3_forwarding_andrewID.tar.gz src inputs README.txt
```

Remember, you have only 5 late lab days for the entire semester (applies to only lab submissions, not to homeworks, not to anything else). If we receive the tarball within 24 hours, we will deduct 1 late lab day. If we receive the tarball within 24 to 48 hours, we will deduct 2 late lab days... During this time, you may send updated versions of the tarballs (but try not to send too many). However, once a tarball is received, it will immediately invalidate a previous tarball you may have sent. You may not take it back. We will take your very last tarball to calculate how many late lab days to deduct. If we don't hear from you at all, we won't deduct any late lab days, but you will receive a 0 score for the lab.

6. Extra Credit

For extra credit on this lab, we will hold a performance competition. Among all implementations that are correct and synthesizable, the "top"[1] students that have the lowest execution time for an undisclosed set of test inputs will receive up to 20% additional credit *for this lab* (equivalent to 1% additional credit for the course) as well as "prizes" (at the discretion of the instructor). You may choose to submit the stalling or the forwarding version of the pipeline without making any effort to optimize it further.

All of the guidelines for Lab 3 specified in this handout also apply to the extra credit, except for the following differences.

- As long as it is correct and synthesizable, **there are no restrictions on the pipeline.** It may have an arbitrary number of stages and an arbitrary way of handling data dependences.
- To make the competition more interesting, you are **not required to implement the multiplier module** which is likely to be on the critical path no matter what. We will not run the extra credit with any test input that includes multiplication-related instructions. However, optimizing multiplication is also a valid way of improving your speed.
- The README.txt must include the following two *additional* pieces of information.
  1. How you have defined your pipeline stages (if different from the reference) and whether you included multiplication.
  2. Description of the effort you invested (if any) to improve performance.
- Submission path: `/afs/ece/class/ece447/handin/lab3/andrewID/extra`
- Tarball (for late submissions): `lab3_extra_andrewID.tar.gz`

For late submissions, we must receive all three tarballs (stalling, forwarding, extra credit) on the same day. If not, we will deduct the lab late days for whichever tarball we received last.

[1] The instructor reserves all rights for the precise definition of the word "top".