COMP 4300 Computer Architecture

Project 3: Pipelined Datapath with Interlocks and Forwarding

Points Possible: 100

- Turned in via Canvas

No collaboration between students. Although you are allowed to share design documents among your group members, students should **NOT** share any project code with each other. Collaborations in any form of source-code sharing will be treated as a serious violation of the University's academic integrity code.

Requirements:

(1) Each student should **independently** accomplish this project assignment. You are allowed to discuss any design issue with your friends to solve the coding problems.
(2) You must submit your partially assembled code, the source code of your simulator (see Sections 2 and 3), a README file, and the values of C, IC, and speed-up (see Section 4).
(3) You must submit your partially assembled code, the source code of your simulator (see Sections 2 and 3), a README file through the **Canvas** system.
(4) Your README file must contain the values of total clock cycles, total instructions executed, and number of NOPs (see Section 5).
(5) You must test (i.e., compile and run) your simulator on **a Linux machine**. Specific compilers you have to use to test and compile your source code are given below:
    a. g++ for C++
    b. gcc for C

1. Introduction

This assignment is a serious step forward from your simulator for multi-cycle machines. Your will have to develop a simulator for a scalar pipelined architecture. It is believed that you are familiar with the five stage MIPS pipeline described in our previous lectures. However, you are allowed to study a different architecture by creating something of your own. Please keep the following **requirements** in mind:

(1) your pipeline must be a load-store, GPR-based 3-operand machine;
(2) your pipeline must be able to execute the subset of MIPS instructions specified below;
(3) your simulator must implement interlocks between pipe stages to recognize and resolve data hazards:

- through forwarding when feasible, including forwarding through the register file
- through NOPs otherwise

In this lab assignment, you are NOT required to implement branch or load delay slots. You must perform each part as described, answer all related questions; you need to submit the required four files (see Section 6) via the Canvas system. Please note that e-mail submissions will not be accepted.

Please refer to Section 1.1 of Project 1's description for instructions on how to make your code readable. Please refer to Section 1.2 of Project 1's description for some general tips for writing good code.

2. Building a Pipeline

We assume a simulator you will be building has a five-stage pipeline. The simulator is more complex than the previous ones you've developed because each instruction is only partially completed in each cycle, and because you have multiple things going on concurrently in the same clock cycle.

It should be noted that you must simulate concurrent behavior somehow. There are two models you could use: perhaps the most obvious one is a "push" model, where you fetch the next instruction and use that to push the behavior of all the downstream pipeline stages. Alternatively, you may use a "pull" model, where you complete the instruction in the WB stage, emptying that stage and pulling the upstream activities down the pipe. This also has the advantage of getting register file writes done early in the cycle, which is the key to forwarding through the register file.

Regardless of the model you will be using, the stages in your simple MIPS pipeline must behave as follows:

(1) The fetch stage is simple: just use the current value in the PC to index memory, and retrieve the instruction you find in the memory. Put it in a buffer (you could call it the IF/ID latch !) where it will be grabbed by decode in the next clock cycle. Note that you have to make sure that the decode stage doesn't get the instruction for cycle (N+1) by mistake.

(2) Decode reads zero, one or two values out of the register file and stores them in a buffer (the ID/EX latch ?) for use by the execute stage in the next clock cycle (we're not working directly with binary encoding of the instructions, so this stage can be a little smarter than the real hardware). If the current instruction is a branch, this stage needs to decide whether or not the branch is taken, and update the PC accordingly. Please keep in mind that this updated PC shouldn't be used until the next cycle, so if you're using the "pull" model you'll need to delay

the change of PC until then.

(3) Execute behaves as necessary, either completing an R-type instruction (such as an add) or doing an address calculation for a load or a store. The result could be put in an EX/MEM latch.
(4) The memory stage references data memory, performing a read or a write if the instruction in the stage is a load or a store, respectively. If the instruction in MEM is an R-type instruction, the MEM/WB latch should buffer the result that was being held in the EX/MEM latch.
(5) Write-back puts the results of an R-type instruction or a load into the register file. This value should come from the MEM/WB latch.

To run the example code we'll be working with, you need ADDI, B, BEQZ, BGE, BNE, LA, LB, LI, SUBI, and SYSCALL. xxx

3. Forwarding

If you take a look at the "Data hazards and forwarding" section of the textbook, you will see a large bubble labeled "Forwarding unit" sitting in the execute stage of the pipeline. This suggests that the appropriate place in your simulator to make forwarding decisions would be in the "execute" stage logic. The surrounding section of the textbook gives the details of what the forwarding logic needs to do. Make sure you can handle the additional complication detailed on p. A-32.

4. Branches and Loads

Please make sure there is always a NOP (no-op) following any branch or load instructions. If we "compile" things this way, the NOP will be fetched while the branch is in decode. The machine will head off in the correct direction in the cycle following the NOP.

Your simple five-stage MIPS pipeline resolves branches during decode, so any further speed-up has to come from doing branch prediction right at the front of the pipe, during fetch (before the machine even knows it has fetched a branch!). Correlating and multilevel predictors apply to machines with longer branch delays. They don't apply to this particular machine. However, there are a couple of cache-style mechanisms for improving the branch performance of the simple MIPS pipeline, and we may look into them further in the next Lab assignment. In this lab, just focus on implementing an execution model for the pipeline that can handle data hazards.

4.1 Suggestion

You should focus on getting the pipeline running hazard-free code first. Once the pipeline is behaving correctly, save a working version of your simulator before

moving on to add the forwarding logic. There will be significant points for the components built in Step 1 (see Section 2).

4.2 Improving the Simulator

In an effort to implement your simulator, please reflect on whether there is a way to structure it so that you could configure it as less-than-complete, in the sense that you'd be able to run four partial pipelines, as well as the full up machine:

- F only
- IF + ID
- IF + ID + EX
- IF + ID + EX + MEM
- IF + ID + EX + MEM + WB (the full machine)

It turns out that in more complex machines, the downstream stages are typically responsible for most of the performance loss (CPI increasing past 1). The ability to add stages successively and then measure CPI would pinpoint the stage(s) responsible for the largest incremental performance loss(es). If you see a nice way to structure your simulator so that you can run it with successively more complete pipelines, that would point to a way to gather such data. **Please note:** Although you are encouraged to implement this feature, you are NOT required to implement this. Rather, just let me know if you see a nice way to do so.

4.3 Detailed Syntax

```
ADD Rdest, Rsrc1, Rsrc2
ADDI  Rdest, Rsrc1, Imm
B  label
BEQZ  Rsrc1, label
BGE  Rsrc1, Rsrc2, label
BNE  Rsrc1, Rsrc2, label
LA  Rdest, label
LB  Rdest, offset(Rsrc1)
LI  Rdest, Imm
NOP
SUBI  Rdest, Rsrc1, Imm
SYSCALL
```

- **Rsrc1**, **Rsrc2**, and **Rdest** specify one of the general-purpose registers (**$0** through **$31**).
- **Imm** denotes a signed immediate (an integer).
- **label** denotes the address associated with a label in the .text or .data segment. This is encoded as a signed offset relative to the current value of the program counter, which will be one word past the address of the current instruction (i.e. the instruction referencing the label).

- **offset** denotes a signed offset (an immediate in the instruction) which is to be added to the value in the base register, **Rbase**. The result is the memory address from which a value is loaded into **Rdest**, or to which the value in **Rsrc1** is stored.

5. A Test Case

(1) You need to instrument your simulator to keep track of number of clock cycles of execution, total instruction count, and number of **NOP**s executed.

You should use a very simple test case (i.e, lab3a.s) to see whether your pipeline is working. You can invent other variations of this code to test things more thoroughly. As for the previous lab, you will need to hand-assemble it for execution.

(2) You should be able to take nearly the same example program (i.e, lab3b.s) you used for Lab 2 - but with **NOP**s inserted after each load or branch - and run it on your pipeline. This requires minimal changes to the assembly (some offsets will change due to the insertion of the **NOP**s), and still give you a fairly complex test case.

(3) Please try the vector addition example (i.e, lab3c.s) from the course pack, just to wring out the forwarding logic thoroughly.

One simulation-related detail: make sure your simulator runs long enough to push the last instruction in the program all the way to the end of the pipe! The simplest solution, of course, is just to add a bunch of **NOP**s to the end of the code (and that is acceptable), although there are certainly simple ways to do this more elegantly.

Your simulator should output total clock cycles, total instructions executed, and number of **NOP**s at the end of the run.

Please report the values of the total clock cycles, total instructions executed, and number of **NOP**s. (see **Grading Criteria 5)**

6. Deliverables

Please submit your program through Canvas (no e-mail submission is accepted).

6.1 A Single Compressed File

You must submit a single compressed file (e.g., file_name.tar.gz); the file name should be formatted as:

"project3.tgz"

Your compressed tarball (e.g., project3.tgz) should contain the three (3) items below:

(1) **lab3c.s** - It contains your partially assembled code.
(2) **pipeSim** - It should contain your simulator that simulates a pipelined data path with interlocks and forwarding.
(3) **README** - It should contain **compilation instructions**, and **instructions on how to use your program**. Your README file MUST contain **values of total clock cycles, total instructions executed, and number of NOPs** at the end of the run. In this README file, please **indicate your name and your student ID**. In addition, a discussion of any design issues you ran into while implementing this project and a description of how the program works (or any parts that don't work) is also appropriate content for the README file. It should contain **compilation instructions**, and **instructions on how to use your program**.

6.2 How to name and create your compressed file?
Create a single compressed (.tgz) file named "project.tgz"

To create a compressed tar.gz file from multiple files or/and folders, we need to run the tar command as follows.

```
tar vfzc project3.tgz <folder_project3>
```

where `<folder_project3>` is a folder that contains the three (3) items described in Section 6.1. `project3.tar.gz` is the single compressed file to be submitted via Canvas. For example, my single compressed file to be submitted can be created using the following command:

```
cd  comp4300
tar vfzc project3.tgz project3
```

where `project3` is a folder under your `comp4300` directory.

6.3 Notes:
1) Other format (e.g., pdf, doc, txt) will not be accepted.
2) No e-mail submission is accepted
3) You will **lose points** (at least 5 points and up to 10 points) if you do not submit a single compressed file and name your compressed file in the format described in this Section.
4) You **lose 15 points** if your README file does not contain values of total clock cycles, total instructions executed, and number of **NOPs** at the end of the run (see Section 5).
5) You will **lose 5 points** if you do not use the specific file names.
6) You will **lose 5 points** if you do not indicate your name and your student ID in your README file.
7) **You will lose points if your source code can not be compiled by g++, or gcc on a Linux machine**.

7. Grading Criteria

1) Partially assembled code for the loop example lab3c.s: 10%
2) A simulator for the pipelined machine pipeSim: 45% (**You will lose points if your source code can not be compiled by g++, or gcc on the Linux machine**)
3) Adhering to coding style: 15%
4) README file: 10%
5) Values of total clock cycles, total instructions executed, and number of **NOPs** at the end of the run (see Section 5): 15%.
6) Specific file names: 5%.

8. Late Submission Penalty

- Ten percent (10%) penalty per day for late submission. For example, an assignment submitted after the deadline but up to 1 day (24 hours) late can achieve a maximum of 90% of points allocated for the assignment. An assignment submitted after the deadline but up to 2 days (48 hours) late can achieve a maximum of 80% of points allocated for the assignment.
- Assignment submitted more than 3 days (72 hours) after the deadline will not be graded.

9. Rebuttal period

- You will be given a period of one week to read and respond to the comments and grades of your homework or project assignment. The TA may use this opportunity to address any concern and question you have. The TA also may ask for additional information from you regarding your homework or project.