COMP 4300 Computer Architecture

Project 2: Single vs. Multi-cycle Machines

Points Possible: 100

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. Submit your project 2 (four files) via the Canvas system (see Section 5).

1. Introduction

In project 1, you created two simulators - one for a stack-based machine and one for an accumulator-based machine. In this lab assignment you will be extending your accumulator-based machine into a General Purpose Register (GPR) machine that runs different instructions in different numbers of cycles (i.e. a multi-cycle machine). 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. More Registers

The starting point is your accumulator-based machine. It has only one register which is the accumulator, and that is referenced implicitly in all instructions. Now you need to add 32 general-purpose registers, which can be named $0 through $31. The more complex part is that you will need to parse the instruction stream to find out which registers are being used in each instruction. Let's assume that the only place in the instruction stream that a "$" will appear is right before a register number.

3. More Instructions

Your accumulator-based machine implemented **LOAD, STO, ADD, MULT** and **END**. We will have to change the instruction set a bit, to start to line it up with MIPS I. To run the example code we will be working with, you need **ADDI**, **B**, **BEQZ**, **BGE**, **BNE**, **LA**, **LB**, **LI**, **SUBI**, and **SYSCALL**. We add a table to the simulator which specifies the number of cycles required by each instruction, in each "functional unit" of the machine:

Instruction	Instruction Memory	Register file - read	ALU	Data Memory	Register file - write	Total
ADDI	2	1	2		1	6
B	2		2			4
BEQZ, BGE, BNE	2	1	2			5
LA	2		2		1	5
LB	2	1	2		1	6
LI	2				1	3
SUBI	2	1	2		1	6
SYSCALL	2	1	2	2	1	8

It will probably make Lab 3 simpler if you start thinking in terms of five separate functional units right now, and keep track of the time required by each, even if you don't step each instruction through the five functional units in sequence. Strictly speaking, though, for this lab all you need is the "Total" time for each instruction type.

Detailed syntax:

```
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
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.

4. Performance Comparison

Now you will have to evaluate how long it takes to execute an example program on your multi-cycle GPR-based machine, and compare that execution time to the time it would have taken on a "single-cycle" machine. Of course, the key word in all this is "**time**". Your multi-cycle machine will take more **cycles** to execute the program, but each cycle would be some fraction of the cycle time on the "matching" single-cycle machine. Why? Well, because the single-cycle machine must be clocked at the rate of the slowest (longest-running) instruction. In this example, **SYSCALL** takes 8 cycles on the multi-cycle implementation. As such, if the clock period (cycle time) on the multi-cycle machine is 1 ns, the cycle time on the single-cycle machine must be 8 ns so that **SYSCALL** will be able to complete in a single cycle.

Your task now is to take the example program written in MIPS assembler and convert it into a source file for your simulator. The instructions should stay the same, but you must lay out the code and data in memory and calculate the proper values for the offset and label values (noting carefully that label values will change depending on the address of the instruction where they're used !). You might want to think of this as a "partial assembly", where you are relocating the code and data to reside at chosen locations in memory.

As in Lab 1, your simulator will just be a big case statement inside a loop that reads instructions one by one and simulates the action(s) of each instruction. Of course, before dropping into the main loop you have to "load" your source code into memory and perform any necessary initializations of the data segment. Next you need to initialize the program counter, and start up the main loop.

This simulator requires what we call "instrumentation". As you execute the code, you need to keep track of the total number of instructions executed (**IC**), and the total

number of cycles spent in execution (**C**). The ratio **[8*IC] / C** is the speed-up of the multi-cycle implementation. Report your values for **IC**, **C**, and **[8*IC] / C.**
We've added branches to the instruction set, so thorough testing of both your simulator and your "partially assembled" code will be very important. The debugger is your friend - get to know and love it.

Recap: Simulator outline, with instrumentation for C and IC

```
Open source code file
Load source code and data into memory (using all-
powerful initialization routine)
Initialize PC
Initialize IC = 0
Initialize C = 0
user_mode = true;
while ( user_mode ) {

     read memory at PC
     increment PC
     increment IC
     case ADDI:    do something; add cycles
     for ADDI to C; break;
     case another_instruction_type:    do
     something; add cycles for this_type to C;
     break;
     etc.

}

Print summary statistics (including C and IC !)
and selected memory locations
```

5. Deliverables

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

5.1 A Single Compressed File

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

"project2.tar.gz"

Your compressed tarball should contain the four (4) items below:

1) **palindrome.s** - It contains your partially assembled code.
2) **gprSim** - It should contain your simulator for the multi-cycle machine.
3) **README** - It should contain **compilation instructions**, and **instructions on how to use your program**. 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**.
4) **result.txt** - It should contain the values you obtained for C, IC, and speed-up.

5.2 How to name and create your compressed file?

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

```
tar vfzc project1.tar.gz <folder_4300_p2_folder>
```

where `<folder_4300_p2_folder>` is a folder that contains the four (4) items described in Section 5.1. `project1.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:

```
tar vfzc project1.tar.gz ./comp4300/project2
```

where `./comp4300/project2` is a folder that contains the above seven items.

5.3 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.

6. Grading Criteria

1) Partially assembled code palindrome.s: 10%
2) A simulator for multi-cycle machines: 45%
3) Adhering to coding style: 15%
4) README file: 10%
5) Values obtained for C, IC, and speed-up: 15%.
6) Specific file names: 5%

7. Late Submission Penalty

- Twenty percent (20%) 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 80% 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 60% of points allocated for the assignment.
- Assignment submitted more than 3 days (72 hours) after the deadline will not be graded.

8. Rebuttal period

- You will be given a period of 72 hours 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.