COMP 4300 Computer Architecture

Project 1: Instruction Set Architectures

Points Possible: 100

No collaboration among groups. Students in one group should NOT share any project code with another group. Collaborations in any form among groups will be treated as a serious violation of the University's academic integrity code.

Requirements:

1. Each group should **independently** accomplish this project assignment. You are allowed to discuss with your friends to solve the coding problems.
2. You must submit your corrected code for the sample program – `quadratic_eval.s`.
3. The coding exercise consists of 3 subtasks. (see Sections 3-5)
   You need to submit the both the written portion and the programming portion via the Canvas system using a single compressed file (see Section 6.1).

Objectives:

- To improve your code reading skills
- To learn how to build simple computer simulators
- To enhance your corporation skills
- To design data structures for stack and accumulator-based machines
- To build a solid foundation for project 2

1. Introduction

This lab assignment aims to re-acquaint you with the MIPS assembly language. First, you will have to simulate two simple machines. Next, you are required to design binary encodings for two simple instruction set architectures. After designing these encodings and comparing the results with the MIPS Instruction Set Architecture (ISA), you will get a better understanding of the issues of ISA. Last, but not least, you need to write an assembler to automatically translate the source codes to binary, and rewrite the simulators to use ("execute") the binary format.

1.1 Make Your Code Readable

It is very important for you to write well-documented and readable code in this programming assignment. The reason for making your code clear and readable is two-fold. First, subsequent labs will make use of the components developed in this lab.

Second, it will be a lot easier for the TA to grade your programming assignments if you provide well-commented code.

Since there exist a variety of ways to organize and document your code, you are allowed to make use of any particular coding style for this programming assignment. It is believed that reading other people's code is a way of learning how to writing readable code. Importantly, when you write code, please pay attention to comments which are used to explain what is going on in your simulated systems.

1.2 Here Are Some General Tips For Writing Better Code

- A little time spent thinking up better names for variables can make debugging a lot easier. Use descriptive names for variables and procedures.
- Group related items together, whether they are variable declarations, lines of code, or functions.
- Watch out for uninitialized variables.
- Split large functions that span multiple pages. Break large functions down! Keep functions simple.
- Always prefer legibility over elegance or conciseness. Note that brevity is often the enemy of legibility.
- Code that is sparsely commented is hard to maintain. Comments should describe the programmer's intent, not the actual mechanics of the code. A comment which says "Find a free disk block" is much more informative than one that says "Find first non-zero element of array."
- Backing up your code as you work is difficult to remember to do sometimes. As soon as your code works, back it up. You always should be able to revert to working code if you accidentally paint yourself into a corner during a "bad day."

2. Begin Your Assignment

2.1 MIPS Assembly

First, download the article - "Assemblers, Linkers, and the SPIM Simulator", and read Section A.9 on pp. A38-A47 to get familiar with the SPIM simulator. Second, download and save the sample program. This program is intended to evaluate a quadratic. Third, since the sample program contains a few bugs, you will need to find them and correct the code so that it can be properly executed. Leave the bugs in your corrected source, but of course, comment them out. You must add comments to explain what your corrections do. **You will need to submit your corrected code.**

2.2 An Example of a Simulated Machine

Throughout our laboratory assignments, we will use xspim for our example of a simulated machine with general purpose registers that uses loads and stores to access memory. xspim is a self-contained simulator that will run MIPS32 assembly language

programs. It reads and executes assembly language programs written for this processor. It also provides a simple debugger and minimal set of operating system services. xspim does NOT execute binary (compiled) programs. Detailed information about xspim can be found at http://www.cs.wisc.edu/~larus/spim.html

Downloading SPIM

Platform	Program	Form	File
Unix or Linux system Mac OS X	*spim* *xspim*	Source code	http://www.cs.wisc.edu/~larus/SPIM/spim.tar.Z or http://www.cs.wisc.edu/~larus/SPIM/spim.tar.gz
Linux	*spim* *xspim*	Binary RPM for Fedora	http://www.cs.wisc.edu/cbi/downloads/
Microsoft Windows (Windows NT, 2000, XP) (spim 7.0 and later versions no longer run on Windows 95/98. Use version 6.5 or earlier.)	*spim* *PCSpim*	Executable	http://www.cs.wisc.edu/~larus/SPIM/pcspim.zip
		Source code	http://www.cs.wisc.edu/~larus/SPIM/pcspim_src.zip

3. Develop Two Simple Simulated Machines

Now that you are in a position to develop two simulated machines: a stack-based machine, and an accumulator-based machine. **You must use either C or C++ in a Linux machine.**

3.1. Basic Components and operations

First, you need to think about the basic components and operations in any machine. Whether it's stack, accumulator or GPR-based (General Purpose Register), a program must be stored in memory. Hence, you should simulate a memory system: given an address, it will need to return the contents of that address. Please note that the contents are either instructions or some data. You could use an array, and then the

address would just be the array index. However, I'd like you to use the MIPS memory model and 32-bit addresses in your code, so that really isn't a practical solution.

Other better choices include:

1. using one array for each memory area (user text, kernel text, user data, the stack, and kernel data). The base address of each area would then be a constant that needs to be added to the index of each array element to form the actual address.
2. using some form of linked list or heap, if you think code might need to be scattered throughout memory (in which case the array approach would be very wasteful).
3. hashing, which may be a good choice because it is a primitive in some languages.

Your first module, then, should be something that takes an address and returns the contents of that address. Of course, that only takes care of reads (loads). You also need to take care of writes (stores), and you also need some sort of all-powerful initialization routine that simulates the action of the operating system when it loads all the relevant areas of memory prior to handing execution off to a user program. So that's actually three modules to start with. The third module (the "loader") will need some sort of file format to read from - for source code files I'd suggest you use a layout similar to MIPS assembly source files. We'll get to binaries later...

3.2. Load, Read, and Write Memory

Now that you can load, read and write memory. You will have to create two separate simulators - one stack-based, and the other accumulator-based.

Let's start with the stack-based machine. The instructions you'll need for your quadratic evaluation code are **PUSH**, **POP**, **ADD** and **MULT**. Something like **END** will be handy too.

1. **PUSH x** will read the contents of memory at address **x** and push the value onto the stack.
2. **POP y** will pop one word off the top of the stack and write the value to memory at address **y**.
3. **ADD** will remove the top two words from the stack, add them, and leave the result on the top of the stack.
4. **MULT** behaves similarly.
5. **END** just signals the end of the code. We could think about implementing a return, but that's overkill for now.

Neither **ADD** nor **MULT** have explicit operands - both operate implicitly on the top of the stack.

Your simulator will just be a big case statement inside a loop that reads source code

statements one by one and simulates the action(s) of each instruction. There will be one "case" for each type of instruction, so for the stack-based machine you will have a loop containing five cases. Of course, before dropping into the loop you have to "load" your source code into memory and perform any necessary initializations of the data areas. Then you need to give the starting address of the code to the case loop, and start it up. (*Hey, that sounds like you need to have something like a program counter, right? Plus some way of automatically incrementing it... note that we're not handling branches in this lab*.)

3.3. Simulator Outline

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

        read memory at PC
        increment PC
        case PUSH:    do something; break;
        case POP:    do something; break;
        case ADD:    do something; break;
        case MULT:    do something; break;
        case END:    user_mode = false; break;

}
Print summary statistics and selected memory locations
```

Once you have this simulator ready, write the code for the quadratic evaluator running on the stack-based machine.

That's all there is to the stack-based machine. Um - except for the stack, right? You should know how to build that.

The accumulator-based machine is very similar - the only real change will be the specifics of the cases in the main loop. And of course, the source code that you write for your "machine" to run will be quite different. You only have a single "register" in the CPU, which is of course the accumulator. You can load a value into the accumulator from memory, and store the current value back to memory. The arithmetic operations all have the accumulator as one implicit operand, and an explicit memory location as the other operand. I suppose it would be possible to build an accumulator-based machine where the destination of an arithmetic operation is a specified location in memory, but for this lab let's make the destination implicitly the accumulator. All this means you'll have the following instructions:

1. **LOAD x** will read the contents of memory at address **x** and put the value into the accumulator.
2. **STO y** will write the contents of the accumulator to memory at address **y**.
3. **ADD x** will read the contents of memory at address **x**, add the value to the accumulator, and leave the result in the accumulator.
4. **MULT x** behaves similarly.
5. **END** just signals the end of the code.

A README file will have to be submitted along with your source code. Your README file must include **compilation instructions**, and **instructions on how to use your program**. 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.

4. Binary Instruction Encoding

It should be noted that having ASCII-encoded instructions in memory is not realistic. Even in case when very large semiconductor memories are available, one of the key problems is that memory cycle times are much slower than CPU cycle times - and getting slower (in relative terms). One way to solve this problem is to encode instructions and data as densely as possible - so that means we need dense, binary encodings for the instruction sets of both simulated machines.

We've only been using five instructions, so you could potentially use a 3-bit opcode. In order to have an apples-to-apples comparison with the code density of the MIPS ISA, I'll add one "extraneous" condition: your instruction set encoding must be able to accommodate 140 instructions. Other than that, the conditions your encodings must satisfy are:

1. addresses in main memory are 32-bit quantities. That means any address fields in your instructions must be 32 bits long, unless you can find some other way to specify a reasonable range of addresses (at minimum, you must be able to get to the base addresses of all five regions in the MIPS memory model: user text, kernel text, user data, kernel data and the stack).
2. you must invent one new instruction for each architecture that manages an immediate value, either pushing it onto the top of the stack in the stack-based machine, or loading it into the accumulator of the accumulator-based machine.

There is no programming in this part of the lab assignment. Your task is to invent separate binary encodings for the stack-based and accumulator-based machines. Then you have to write your quadratic evaluators in binary (please use hexadecimal notation though!). Finally, record the number of bytes required for your programs (both code and data) and compare that to the size of the MIPS program (not including the trap handler, of course).

5. Using the Binary Encodings in the Simulations

Now you have two simulators that take ASCII input, and you have also developed the rules for translating that ASCII input into binary. The next step would be to **write the assemblers** to automatically translate the source codes to binary, and **rewrite the simulators** to use ("execute") the binary format. That actually wouldn't be too tough, as we only have five instructions to deal with.

The more important thing that this brings us face-to-face with is the need to define the layout of the file containing an executable. This issue doesn't become "visible" until you start grappling with the use of binary files containing executables, because ASCII files have lots of clues that make them easy to parse (**.text** and **.data**, for example). If you pick up something that is just a sequence of anonymous bits, though, how would you know what's code and what's data, for example?

The answer is that you need a defined format. Recent versions of Linux use something called ELF (executable and linking format). You can find lots of information about ELF using google. If you take a peek at the specification, the thing that should jump out at you is that there is something called an "ELF Header", and this header is always the first thing in an ELF file. The header specifies the type of file and the target architecture, specifies the position in the file of other headers and sections, and gives the location of the first executable instruction in the file.

You are not required to use binaries in this course. It just seemed that the process of working through the lab and building the simulators brings us to a point where it's easy to motivate something that isn't often discussed (i.e. the need for defined formats for executables), and that as a result you might see quite easily what the issue is, and how important it is.

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:

"<group ID>_project1.tar.gz"

For example, if I am a member of group 6, mine would read “group06 _project1.tar.gz”).

Your compressed tarball (e.g., group06_project1.tar.gz) should contain the eight (8) items below:

1) **quadratic_eval.s -** It contains your debugged / commented code (see Section 1)
2) **stackSim.c or stackSim.cpp** - It is the source code of your simulator for the stack-based machine.
3) **stackCode** - It should contain your code for the quadratic evaluator, to run on your stack simulator.
4) **accumSim.c or accumSim.cpp** - It is the source code of your simulator for the accumulator-based machine.
5) **accumCode** - It contains the code for the quadratic evaluator to run on it.
6) **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.
7) **encoding.pdf** You must hand in hardcopy with the answers for the questions described in Section 4. This will consist of:
    - diagrams giving your binary format for each of the instructions on the two machines,
    - the assembled versions of both programs in a two-column format, with the source code in the left-hand column and the binary translation in the right-hand column (binary version in hexadecimal, please), and
    - the size in bytes of the binary versions of your stack and accumulator quadratic evaluation codes, and the size in bytes of the user code plus user data segments of the MIPS version of the quadratic evaluator.
8) **collaboration.doc** You must briefly describe how the two members in your group collaborate on this project. You also need to summarize the contributions of each member.

6.2 How to name and create your compressed file?

Create a single compressed (.tar.gz) file named "<group ID>_project1.tar.gz" (for example, mine might read "group06_project1.tar.gz")

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

```
tar -fcz <firstname_lastname>_project1.tar.gz <folder_4300_p1_folder>
```

where `<folder_4300_p1_folder>` is a folder that contains the seven (7) items described in Section 6.1. `<group ID>_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 -fcz group06_project1.tar.gz ./comp4300/project1
```

where `./comp4300/project1` is a folder that contains the above seven items.

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 subsection.

7. Grading Criteria

1) Debugged / commented code: 15%
2) Stack-based machine: 25%
3) Accumulator-based machine: 25%
4) Binary Instruction Encoding: 15%
5) Adhering to coding style and README file: 10%.
6) Collaboration document: 10%

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 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.