MIPS Dynamic Instruction Decode

CS 370 Project 1 – Fall 2017
Due: October 23, 2017 at 11:00pm

1 Introduction

Please read this entire document. It is lengthy and contains many details – but most of these details are meant to make writing your program easier. Save yourself some headache and read this first.

This is the first project for CS 370. The purpose of the project is to give you an introduction to processor modeling and simulation. Also, this project gives you ample practice encoding and decoding instructions. The statistics gathered in this project can also give you a sense of what actual program behavior is common in real benchmarks.

Your program will need to read a stream of *dynamic* instructions, along with their instruction addresses (PCs). Your program will act like the decode logic of the MIPS processor, determining what type of instruction is being executed, and what operands are used, etc. This instruction and operand information is then used to keep various statistics that will be reported at the end of the your program.

Programs like this are typically referred to as "trace-driven simulations" because the program has already been run, and you are post-processing statistics based on the trace (executed sequence) of instructions through the program. In this style of processor modeling, you *do not* need to worry about what values each register have, what the results of instructions are, etc. The alternative to trace-driven simulation is "execution-driven simulation", which is very similar to emulators that you might already be familiar with (for instance for old game systems, or Android/iOS app development emulators). Execution-driven simulations take a program binary as an input, and the simulation must fetch, decode, and execute instructions by stepping through the program just like the actual processor would.

This project is to be done individually. Submitted projects will be passed through a code comparison tool to compare your code to other students (this semester and previous). Any evidence of copied project source code will be reported to the Student Life Office as an academic integrity violation. Be cautious when helping friends with the assignment – discussing your general approach for the various aspects of this assignment is okay, but sharing code is not.

You are free to write your project code in any programming language that you prefer **as long as you program runs without modification in linux** which is the system that will be used for grading your projects. The due date for this project is **Monday, October 23rd, 2017 at 11:00pm**. Projects must be zipped into a single file, and submitted on D2L. Late projects cannot be accepted. It is in your best interest to start this project as early as possible. All of the material that you need to know to complete this project has already been given in lectures.

2 Project Specification

In this project, you must write a program that reads an input file that consists of a sequence of dynamic instructions. Your program must decode those instructions, and report several statistics. Since we are only concerned with statistics that require decoding the instructions, you do not have to worry about modeling the execution of the instructions (i.e. when you encounter an `add` instruction, you do not need to read register values and add them like the processor would do). The instructions are encoded in the 32 bit MIPS-I instruction format, as specified in the course textbook. Details on each instruction appear in the MIPS Reference Data sheet.

For this project, each dynamic instruction is also paired with its instruction address (the value of the PC). Thus, the input to your program will be a text file with <PC, instruction>pairs. The file will contain the dynamic instruction stream – this represents a program that has already been run. Any given static instruction might be skipped by branch or jump instructions, or might repeat due to loops. This means that there may be gaps in addresses (PCs skipped by a branch) and some addresses may occur more than once (PCs inside of loop bodies). Your program should collect the statistics of all of the dynamic instructions.

Table 1 lists all of the statistics that your program must collect, along with a detailed description of each statistic. Note that the register statistics need to be collected *per-register* (i.e. $r0 through $r31) even though the register statistics are only described once in Table 1.

We will restrict the instructions that can possibly appear in the input file to only the integer instructions found on the front of the MIPS Reference Data sheet. Thus, instructions that have the format FI-type or FR-type will never appear in the input file. We will also *exclude* the following integer instructions:

- `mult` and `multu`
- `div` and `divu`
- `mfhi` and `mflo`
- `lwc1`, `ldc1`, `swc1`, and `sdc1`
- `mfc0`
- `ll` and `sc`

all other integer instructions could appear in the input file. Don't forget the `sra` instruction, it *will* be included – you might miss it because it appears in the right hand column of the MIPS Reference Data sheet among several instructions that won't be included.

There are no performance requirements for your program. Although, while grading, I might think that your program is stuck in an infinite loop if it is taking an inordinate length of time compared to other students programs. There is no need to print status of your program while it is executing, although no points will be deducted if your program prints to the screen. The input to your program will be a plain text file, described in the next section. And likewise, the output should be written to a file (instead of printing to the screen). The format of the output file is also described in its own section, after the input file description.

2.1 Input Format

The input will be a plain ASCII text file, with one <PC, instruction>tuple per line. Both the instruction address (PC) and the instruction will be in hexadecimal without any prefix or postfix to denote hex (i.e. there will be no "0x" before the PC or instruction). Only the integer instructions will be used, no floating point types or registers will be used, as outlined in the previous section. The input file will never end with a branch or jump instruction.

The input file will always be named `trace.txt`. The input file will always be in the same directory as your program executable before starting your program. This means that it is safe for you to hard-code the input file name into your program. There is no fixed upper limit on the number of instructions in the input file, with the exception that it is guaranteed that the input will chosen such that it never causes any single statistic to overflow a 32 bit integer. This means that it is safe for you to use 32 bit integers for your statistic variables. There will always be at least two instructions in the input trace file.

Listing 1 shows a small example input file, exactly as it would appear when used as an input to your program. The file will always be in ASCII text (i.e. one byte characters). Notice that the text file has no header row, and a single space between the PC (first column), and MIPS encoded instruction (second column). A copy of this input file can be obtained from D2L with the file name `project1-example-trace.txt`. If you use this file for testing, you should be sure to rename it to the required file name `trace.txt` before running your program.

This example input file shown in Listing 1 is not sufficient for testing your program. It only includes a small loop that iterates three times. It only uses a handful of opcodes and register operands. You should write your own test input files that stress every instruction, register, etc. You will want to be sure to come up with test input files that exercise each statistic that is required for this project. I will post the expected output file for this example on D2L.

Table 1: List of all statistics that your program must collect.

Statistic	**Output File Identifier**	**Description**
Total Dynamic Instructions	insts	The total number of dynamic instructions from the input file. This is equivalent to the number of lines in the input file.
Number of R-type Instructions	r-type	The total number of dynamic instructions that are R-type.
Number of I-type Instructions	i-type	The total number of dynamic instructions that are I-type.
Number of J-type Instructions	j-type	The total number of dynamic instructions that are J-type. The sum of R-type, I-type, and J-type should equal the total number of dynamic instructions.
Number of forward taken branches	fwd-taken	The total number of dynamic instructions that 1) are a branch or jump, 2) is taken, and 3) the next instruction executed has a target address that is greater than the current address +4 (i.e. an instruction after the next sequential instruction). All three conditions must be met to increment this statistic. Note that jumps are always taken.
Number of backward taken branches	bkw-taken	The total number of dynamic instructions that 1) are a branch or jump, 2) is taken, and 3) the next instruction executed has a target address that is less than the current address +4 (i.e. an instruction before the next sequential instruction). All three conditions must be met to increment this statistic. Note that jumps are always taken.
Number of not-taken branches	not-taken	The total number of dynamic instructions that 1) are a branch, and 2) is not taken, defined as having the next instruction address executed that is exactly the current instruction address +4. Note that jumps cannot be not-taken, so a jump instruction will never cause this statistic to be incremented.
Number of loads	loads	The total number of dynamic instructions that are load instructions. This includes lbu, lhu, ll, and lw. Note that lui is not a load, despite the name, since it does not access main memory. Also, the floating point instructions are not included in this assignment, and thus lwc1 and ldc1 will never appear in the dynamic instruction input, even though they are I-type instructions.
Number of stores	stores	The total number of dynamic instructions that are store instructions. This does not include the swc1 or sdc1 instructions, since floating point instructions are not included in this assignment.
Number of register reads	reg-<x>	The number of times that register <x>has been used as a source register operand. Some R-type instructions have two source register operands, so two statistics should be incremented for each source register. When reporting statistics, the number of reads for each register will appear in the first column after the reg-<x> identifier.
Number of register writes		The number of times that register <x>has been used as a destination operand. Note that some instructions use the rd field to denote a destination register operand, whereas other instructions may use the rt field. When reporting statistics, the number of writes for each register will appear in the second column after the reg-<x>identifier.

```
00401038 00134880
0040103c 01364820
00401040 8d280000
00401044 15150002
00401048 22730001
0040104c 0810040e
00401038 00134880
0040103c 01364820
00401040 8d280000
00401044 15150002
00401048 22730001
0040104c 0810040e
00401038 00134880
0040103c 01364820
00401040 8d280000
00401044 15150002
00401050 ad2f0000
```

Listing 1: Example input file.

2.2 Output Format

The output should be an ASCII plain-text file with the exact format as shown in Listing 2. The statistics gathered by your program should replace each of the angle-bracketed items in the output (and you should not print the angle brackets). The output file generated by your program should always have the exact name `statistics.txt`. When running your program, I will always be sure that there is not already an existing file with the same file name in your program directory before running your program. This means that it is safe to hard-code the output file name in your program.

Your output file should *exactly* match the format in Listing 2: no extra spaces, no extra line breaks, etc. All of the registers should be reported even if they were never read or written by any instruction in the trace. In that case, the statistic should be 0. The register numbers should be in decimal. There should be a single space between the number of reads and the number of writes for a given register. All of the reported statistics should be integers, in decimal, and should not include the angle-brackets.

```
insts: <num-insts>
r-type: <num-r-type>
i-type: <num-i-type>
j-type: <num-j-type>
fwd-taken: <num-fwd-taken>
bkw-taken: <num-bkw-taken>
not-taken: <num-not-taken>
loads: <num-loads>
stores: <num-stores>
reg-0: <num-reads> <num-writes>
reg-1: <num-reads> <num-writes>
reg-2: <num-reads> <num-writes>
reg-3: <num-reads> <num-writes>
reg-4: <num-reads> <num-writes>
reg-5: <num-reads> <num-writes>
reg-6: <num-reads> <num-writes>
reg-7: <num-reads> <num-writes>
reg-8: <num-reads> <num-writes>
reg-9: <num-reads> <num-writes>
reg-10: <num-reads> <num-writes>
reg-11: <num-reads> <num-writes>
reg-12: <num-reads> <num-writes>
reg-13: <num-reads> <num-writes>
reg-14: <num-reads> <num-writes>
reg-15: <num-reads> <num-writes>
reg-16: <num-reads> <num-writes>
reg-17: <num-reads> <num-writes>
reg-18: <num-reads> <num-writes>
reg-19: <num-reads> <num-writes>
```

```
reg-20: <num-reads> <num-writes>
reg-21: <num-reads> <num-writes>
reg-22: <num-reads> <num-writes>
reg-23: <num-reads> <num-writes>
reg-24: <num-reads> <num-writes>
reg-25: <num-reads> <num-writes>
reg-26: <num-reads> <num-writes>
reg-27: <num-reads> <num-writes>
reg-28: <num-reads> <num-writes>
reg-29: <num-reads> <num-writes>
reg-30: <num-reads> <num-writes>
reg-31: <num-reads> <num-writes>
```

Listing 2: Output file format.

Listing 3 shows some of the output that is expected when using the example input file shown in Listing 1. To save space in this document, not all registers have been shown in this example. The full output file can be downloaded from D2L, file name `project1-example-statistics.txt`. If you use these example input and output files to test your program, you should be able to use the linux `diff` tool to see if your output file exactly matches the file downloaded from D2L.

```
insts: 17
r-type: 6
i-type: 9
j-type: 2
fwd-taken: 1
bkw-taken: 2
not-taken: 2
loads: 3
stores: 1
...
reg-8: 3 3
reg-9: 7 6
...
reg-19: 5 2
reg-20: 0 0
reg-21: 3 0
reg-22: 3 0
...
```

Listing 3: Partial example output file that matches the example input file from Listing 1.

The code shown in Listing 4 shows the assembly code loop that matches the example input file from Listing 1. In the example input, the loop body iterates three times before exiting.

```
loop:
        sll $t1, $s3, 2
        add $t1, $t1, $s6
        lw $t0, 0($t1)
        bne $t0, $s5, exit
        addi $s3, $s3, 1
        j loop
exit:
        sw $t7, 0($t1)
```

Listing 4: The assembly code that was used to generate the trace in Listing 1.

3 Hints

You should read the PC and instruction into an integer data type, that way you can perform bitwise operations on the value read from the file to extract bit fields. If you use the Java programming language, you may want to look into the `Long.parseLong()` method (more hints on this below). **You should not parse the instructions as Strings** – to do so is an affront to coding humanity. Instead, convert the input values to an integer type first, and learn how to use bitwise operators. If you use Java, you can see variables printed in hex using `System.out.println(Long.toHexString(myLongVariable));`

For most statistics, you can simply look at one instruction at a time. You can use bitwise operators to extract the bits of each instruction. For example, to extract the opcode field, you can right-shift the instruction value by 26, and then use a `switch` statement or `if/else if` blocks to execute the code that is specific for that particular opcode.

Remember that instructions and addresses are all 32 bits. In programming languages other than Java, you can use 32 bit integers in your program. You will want to be sure to use unsigned integers (in C/C++) so that when you right-shift, you don't get 1's in the upper positions (i.e. a negative number) due to sign extension. Java, however, does not have `unsigned` data types – so you will need to use 64 bit `long` data types instead, if you write your project in Java. Tisk-tisk, Java for such a glaring omission. Also, in Java you can use the `>>>` operator to right-shift a `long` without sign-extending.

For fields other than the opcode, once you right-shift to align the bits that you want into the lowest bit positions, you will still have the upper bits (part of the next field). You can get rid of these bits by recognizing the nifty properties of AND: ANDing with 0 will always result in 0, and ANDing with 1 will always depend on the number you're ANDing with. So, if you bitwise AND your shifted instruction with a value that has 0s in the positions that you want to get rid of, and 1s in the places that you want to keep, the result will be only those bits that you want. This is called *bit masking*.

If your programming language of choice does not have bitwise operators, you are not completely out of luck. You can cleverly use integer division by powers of two to do bit shifts to the right. You can also use the modulo operator with powers of two to extract bits. But in that event you might want to seriously consider using a real programming language.

For branch and jump instructions, you need to know not only the current instruction PC, but also the PC of the next executed instruction. If the instruction is a branch and the next instruction PC is PC+4, then you know that the branch was *not-taken*. If the instruction is a branch or jump and next instruction PC is less than PC+4, then the branch/jump was *taken*, and it is backward target. If the next instruction PC is greater than PC+4, then the branch/jump is *taken* and it is a forward target.

If you have not written any programs involving reading or writing files within your program, you can search with terms like "java file i/o tutorial" or "c++ file i/o examples" to see how files are handled in your programming language of choice. It is your responsibility to figure out how to read and write files within your program.

Most programming languages have a way to read input that is in hexadecimal. For example, in C/C++, you can use something like:

```
fscanf(fd,"%x",&foo);
```

to read a hexadecimal value from a file opened in the file descriptor `fd` into an `unsigned int` named `foo`. In Java, you can have something like:

```
foo = Long.parseLong(hexInput, 16);
```

where `foo` is a `long` and `hexInput` is a `String`.

If you are stuck, stop by office hours.

4 Language, Running Your Program

The preferred language for this project is C or C++. However, you can write your program in any programming language that you are comfortable with **as long as the I can run your program on a standard linux command line**. I will not install any IDE development environment (like Eclipse or Visual Studio), so please get your program to work from the command line if you use a language other than C/C++ or Java.

You might want to consider including a README text file with your program source code if you use a language other than C/C++ or Java. The README should contain the steps that I should take in order to compile and run your program.

An example input file its matching expected output file is posted on D2L.

5 Submission and Grading

You should submit your project **source code**, zipped into a single file, to D2L. The dropbox is named "Project 1". The project deadline is October 23rd, at 11:00pm. Late projects can only be accepted with a university-approved excused absence.

The program should compile without errors and run without infinite loops, segmentation faults, unhandled exceptions, or any other crashes. The output format should exactly match the specification, if not, a deduction of 20% will apply. Your program should always read from an input file named `trace.txt`, and the output should always be written to a file named `statistics.txt`. Any other file names will incur a 20% penalty. The values reported by your program must exactly match the expected output for given the input trace for full credit.

There will be three input trace files tested, with varying complexity. Each output file will be equally weighted, and are worth 20% each. If any value is erroneous, points will be deducted based on the scale of the difference between the expected value and the value returned by your program. For example, if the expected number of total instructions is 2571, you will have a higher grade if your program reports 2565 than if your program reports 582.

You do not need to submit any trace input files or statistic output files. They will not be used for grading, and if you do include any input or output files, they will be replaced when testing anyway.