CISS 360: Computer Systems and Assembly Language
Project 2

Objectives
The objective is to write a MIPS interpreter in C/C++.

Submission instruction:
You will do a demo of your program during the week before finals. There will be a peer review of your program. The best projects will be showcased on Thursday or Friday of the finals week. If selected, you will present your work to the CS student body and the CS faculty members. This will hopefully generate some interest and discussion on your work and move you in the direction of your senior capstone project. Possible areas of further work related to this project includes programming languages and interpreters/compilers.

Description

As an interpreter, the software offers the benefit of an interpreter which allows us to enter one MIPS/SPIM instruction at a time. Each time an instruction is entered, the program executes the instruction and provides error or warning messages immediately. Like any interpreter, your software must provide this instant response for each instruction to help in writing and experimenting a MIPS program. The program however also allows the user to run a complete MIPS program. Therefore one can view the software as capable of running in two modes: an interactive mode when the user enters one instruction at a time and a non-interactive or batch mode when the user run a complete MIPS program using the software.

If the statement is a MIPS/SPIM instruction, the instruction is placed at the location in memory specified by PC, executed interactively, and PC is advanced. For instance the following is a sample execution of a simulator:

```
MySPIM simulator 0.1
? - help

TEXT:0x400000 >
TEXT:0x400000 >
TEXT:0x400000 > li $s0, 1
TEXT:0x400008 >
```

(`li` is a pseudoinstruction that consists of two instructions.) Note that you need not follow any specific user interface in this handout; you may design your own. The important thing is the feature list of your software.

Of course the software allows you to switch between entering data into the text segment and the data segment.

The software should also allow you to view the state of your simulation of MIPS. This should include the registers, the memory, and the labels with their values. For instance the following is an example of a display of the data segment and the labels:

```
================================================================
DATA SEGMENT
================================================================
  addr (int)|  addr (hex)| value (int)| value (hex)| value (char)
------------+------------+------------+------------+-------------
   268435456|    10000000|  1214606444| 48 65 6c 6c|  H e l l
   268435460|    10000004|  1864390511| 6f 20 57 6f|  o   W o
   268435464|    10000008|  1919706145| 72 6c 64 21|  r l d !
   268435468|    1000000c|   172849440|  a 4d 79 20| \n M y
   268435472|    10000010|  1851878757| 6e 61 6d 65|  n a m e
   268435476|    10000014|   543781664| 20 69 73 20|    i s
   268435480|    10000018|  1397770573| 53 50 49 4d|  S P I M
   268435484|    1000001c|   554303592| 21  a  0 68|  !\n
```

```
================================================================
Labels
================================================================
   268435456|  PROMPT
   268435464|     TAB
   268435466| NEWLINE
   268435468|   SPACE
   268435470|     END
```

A minor thing to note is that for the display of the data segment, the above simulator displays the newline character as the characters `'\\'` and `'n'` rather than a newline character. Therefore a two-character-space was allocated for the display of a single character.

Note that it's probably a good idea to distinguish between the input prompt for performing a syscall from your simulator's prompt. For instance if a syscall is made to request for an integer input from the user, the prompt can change to something similar to this:

```
CONSOLE INTEGER INPUT>
```

The following is a very important point: Note that not all instructions can be executed immediately. For instance if an instruction uses a label that has yet to be defined, then execution of the program should halt temporarily. The user can still enter instructions and the program continues execution once the missing label is supplied.

Feature list

- The user can enter SPIM instruction and data at the prompt.
- The user can load a SPIM program.
- The simulator executes instruction and data input whenever possible. If an invalid instruction was entered the program displays a warning that should help in debugging the instruction.
- The user can view the state of the MIPS including the text, data segment and labels.
- The user can re-execute the program that was interactively entered from the first instruction.
- The user can save the program entered into a file.
- [OPTIONAL] The user inserts breakpoints and singlestep/multi-step through the program.
- [OPTIONAL] The user can set environment variables such as setting the verbosity of the simulator.

Implementation Notes

The following are some implementation hints and some ideas. It is not the intention of this section that you must implement your software using the ideas stated here. In other words you can design your own data structures and algorithms.

Instructions

An assembly instruction (as a string) can be parsed and executed immediately. For instance if instruction is entered by the user (as a string):

```
add $t0, $s1, $s2
```

your program should (more or less) produce these substrings:

```
add
$t0
$s1
$s2
```

With these substrings you can immediately execute this code (assuming you have unsigned int variables `t0, s1, s2:`

```
t0 = s1 + s2
```

Of course you need to keep the string "add $t0, $s1, $s2" entered by the user for later re-execution and for saving to a program file. In summary your software must be able to do this:

```
if string entered is "add $t0, $s1, $s2"
    t0 = s1 + s2
```

Another way to do this is to translate the assembly code into machine code and execute the machine code. The benefit of doing this is that your simulator can then execute *any* valid MIPS machine code. (There are actually other details to worry about.) This is the basic principle behind some virtual machines that can run code built for different architecture and platforms. Each machine instruction is made up of 32 bits that represents an instruction. Therefore for each word, you need to extract the bits representing the actual opcode (i.e. the type of instruction), the registers, etc. For example the instruction

```
add $t0, $s1, $s2
```

is translated into

000000 10001 10010 01000 00000 100000

i.e. as a MIPS machine (not assembly) instruction is

00000010001100100100000000100000

For instance the first 6 bits 000000 is the opcode for `add`. The other groups of bits also having special meanings. For instance 10001 is 17 and $s1 is also known as $17. Therefore to execute this machine code, your program would need to extract the first 6 bits, 5 bits, etc. from

00000010001100100100000000100000

to

000000 10001 10010 01000 00000 100000

in order to execute this instruction..

Note that there is a ***third*** way of executing a MIPS program. You can display the machine code to the user as the MIPS assembly code is entered but your program can have its ***own*** representation of the instruction. The reason for doing this is that while MIPS machine instruction is optimized for hardware; your simulator is software. If you prefer speed to conserving memory, then you can keep a format with the opcode and registers already separated from a single word. If you do this, then the above machine code (as a word):

00000010001100100100000000100000

can be stored in your program as 6 unsigned integers (or unsigned char):

000000 10001 10010 01000 00000 100000

or

0 17 18 8 0 32

An even better option is to store the string entered by the user, the machine code, and the decoded parts of the machine code, i.e., your data structure stores this:

"`add $t0, $s1, $s2`", 00000010001100100100000000100000, (0,17,18,8,0,32)

Advice: I strongly suggest you start with the first format (store program as assembly code) to get a feel for your simulator software before moving on to generate machine code.

Registers

It is obviously better to use an array of unsigned integers for all your registers than to use individual unsigned integer variables. If the array is called reg, then the pseudocode

```
if string entered is "add $t0, $s1, $s2"
    t0 = s1 + s2
```

becomes:

```
if string entered is "add $t0, $s1, $s2"
    reg[8] = reg[17] + reg[18]
```

assuming that the reg[17] corresponds to register $s1, reg[18] to $s2, and reg[8] to $t0. Rather than inventing the wheel, you might as well use the numbering of the registers in MIPS; refer to your lecture notes or our textbook. In fact in MIPS register $s1 is also known as $17.

Memory

It is probably better to implement your memory as an array of unsigned integers than an array of unsigned chars. But I leave this up to you.

Pseudo-Instructions

Note that pseudoinstructions can be expanded into a sequence of instructions. Therefore if you only need to write your software to execute instructions and to translate pseudo-instructions into instructions. Note however that you can nonetheless execute pseudo-instructions in your interpreter directly. You can choose which version to implement.

Controlling Flow of Execution

For a real CPU, the address of the next instruction to be executed is stored in the register called the program counter. This contains an address. Therefore to control the execution of a program your program simply needs to fetch the code to execute (for instance a machine code) using the PC.

First assume that you want to store the instructions entered by the user as machine code in memory. The PC has to be advanced to the address of the next instruction after the fetch operation. Therefore if your memory (the array of unsigned int) is called mem and your PC is modeled by the unsigned int variable PC, the code executing your code looks like this:

```
bool run = true;
while (run)
{
      unsigned int instr = fetch(mem, PC);

      PC += 4;           // remember that you have an array of unsigned
                         // ints. Therefore fetch() returns mem[PC/4].
                         // If you modeled your memory with an array
                         // of bytes (i.e. unsigned char) then
                         // the word returned is a concatenation of the
                         // bits in
                         // mem[PC], mem[PC+1], mem[PC+2], mem[PC+3]
      ...
}
```

This is then followed by decoding the instruction `inst` to get the opcode, parameters, etc.

Remember that another way to simulate MIPS is to simply execute the instruction directly from the assembly code. For instance (see above):

```
      if string entered is “add $t0, $s1, $s2“
          t0 = s1 + s2
```

This tells you that you can store your program as an array of strings as your program. For instance:

```
      "li $s1, 2"
      "li $s2, 3"
      "add $t0, $s1, $s2"
```

Of course since these are not machine code, you cannot put them into your MIPS memory. Hence you cannot really assign addresses to them. One way is to simply use the array of your instructions as “addresses”.

Submission

You will do a demo of your program during the week before finals. There will be a peer review of your program. The best projects will be showcased on Thursday or Friday of the finals week. If selected, you will present your work to the CS student body and the CS faculty members. This will hopefully generate some interest and discussion on your work and move you in the direction of your senior capstone project.

Note that there is no dress code for the demo of your program. On the day of your presentation, you dress for a group of programmers or software engineers.

Once it's your turn to demo your program, you must begin your talk/demo within 10 seconds. In other words, you have 10 seconds to setup your demo.

During the demo of your program, you will explain how to use your program and do a short demo. After that, the audience can participate in the following ways:

- The audience can ask you enter specific MIPS assembly code.
- The audience can ask you for the algorithm to perform a specific task.
- The audience can ask you to show your code.
- The audience can ask you for a list of features that were not implemented.
- For a group project the audience can ask questions about team collaboration (example: the distribution of workload, fights that you had, etc.)
- For extra credit project: The audience can ask if any extra credit features were implemented.

Note that the audience includes the “instructor” as well as other invited faculty members (not likely).

You need not have any presentation slides but it will probably impress your audience. Furthermore it will help you in your presentation when questions were asked. Missing a point in your algorithm when the audience ask for clarification will give the wrong kind of impression. (i.e. “Does this guy really know what he's talking about?”)

In particular, slides containing top-down decomposition of your functions and pseudo-code for specific (and non-trivial) algorithms are things you might want to include. (Note that preparing these slides ***during*** the development of your program will actually help you plan for your project too.)

Grading

The first rule for the grading of this project is very simple:

- If your program crashes during the demo you will get a zero.

If your program does not crash during the presentation, you will then be graded in two different ways.

- Points are assigned to your ***presentation***.
- Points are assigned to ***features completed***. The following is a description of the program in various stages of completion. In the following list "The program does X" is the same as "The program does X correctly".

    1. The user can enter code which does not involve the data segment, syscall, branching, or jumps. If the code is incorrect, a meaningful error message is displayed. Otherwise the code is executed. The user can view the registers. At this point you only need to show the values of the registers as integers. At this point your simulator allows the user to work onto with registers (load-immediate, move, addition, subtract). The addition and subtraction should immediate add and sub. Of course you need to allow the user to quit your program.
    2. As above except that the user can all arithmetic operations including multiplication and subtraction.
    3. As above except that your software displays register values as integers, unsigned integers, hexadecimals, and characters.
    4. The user can load a program. (This has to be implemented early in order to aid in testing.) The software runs each instruction as it is read from the file.
    5. The user can view all the instructions entered (only the valid ones are remembered by your simulator); your program need not show the machine code.
    6. As above except that when the user enters an instruction, the machine code and decoded parts of the machine code is shown. Also during a re-execution of the program, the assembly code, machine code, and decoded parts of the machine code is shown.
    7. As above except that user can enter code which does not involve the syscall, branching, or jumps and the user can view the registers and the data segment. The At this point the user can put data into the data segment.
    8. As above except that the user can enter code which does not involve branching or jumps. The user can enter code which does not involve branching and jump instructions is executed in your simulator.
    9. As above except that the user can enter MIPS assembly code including branching and jumps.
    10. The user can save the program to a file.

As a guide, a completed simulator must execute the programs in our assignments up to function calls.

(Therefore your simulator need not perform instructions for the math co-processor, etc.)

The following are other useful features to consider for future improvements:

- The user inserts breakpoints and singlestep/multi-step through the program.
- The user can set environment variables such as the verbosity of the simulator, prevent interactive execution, prevent display of trace, etc.
- The user can modify the values of the registers and memory.
- The simulator can view the call stack.
- The simulator allows the user to check for stack corruption.
- The simulator has different syscalls for file i/o, graphics, network calls, etc.

Some Advice

You can only use the standard libraries that come with the programming language. And you do want to make full use of these libraries. Do not re-invent wheels unnecessarily. You are not permitted to use code outside the standard libraries of your language.

SUGGESTIONS/HINTS PART 1

It's clear that there are some basic data conversion.

For instance you will need to convert an unsigned integer (the bits in a register) to octals, hexadecimals, etc. Each 8 bits in an int is also converted to characters according to the ASCII code table. It will probably be useful convert the data backward too. So you should have a separate header and cpp for such data manipulation/conversion. Test these functions thoroughly. Here are some you want to write RIGHT AWAY.

- Function that accepts unsigned int and returns hexadecimal string
- Function that accepts unsigned int and returns 4 characters
- Function that accepts hexadecimal string and return an unsigned int
- Function that accepts 4 characters and return an unsigned int

Clearly there are also some string operations. Quickly read up on the std::string class. You can use the C-string library too if you like. Clearly you must write a function that accepts a user input and go through the different formats of MIPS instructions and break up the MIPS pseudoinstruction/struction into parts. For instance from “add $s0, $s1, $s2”, you must be able to obtain “add”, “$s0”, “$s1”, $s2”. You might want to write a function that splits up a string into substrings by commas and spaces.

Following MIPS' ISA, there are 32 registers. Sure, they have names, but to the chip, they are identified by numbers. Register names are just for the convenience of assembly programming. With that in mind, the registers form an array of 32 unsigned int. The set of registers in the CPU is frequently called a register file. You might want to create a class called RegisterFile. Or you can just use a plain array of 32 unsigned ints. You should have a simple function to translate register name to register number.

- Class to represent register file. The implementation of the class should probably use an array of 32 unsigned integers
- Function to convert register name to register number.

SUGGESTIONS PART 2

As for the memory, you already know that the memory is an array of bytes. You may assume that the memory you are simulating has a 32-bit address space. In other words the first address is 0 and the last address is 2^32 – 1. This means that the memory is an array of 2^32 chars. But if you try to declare such a variable in your program, you will see that the memory required is too too big.

Now note that most of the cells is filled with 0s.

One option is to std::map which implements a container a key-value pairs somewhat like a Python dictionary. (It's typically structured as trees, not hashtables.) Here's an example:

```
#include <map>

int main()
{
   std::map<unsigned int, unsigned char> mem;
   std::cout << mem[0] << '\n';
   mem[0] = 42;
   std::cout << mem[0] << '\n';
}
```

You will find that you can simulate a larger memory with the above.

Google for STL map tutorials.

Otherwise, just make sure you don't use all the memory addresses from 0 up to 2^32 – 1 when you run a sample SPIM program.