CSC3050 Assignment 1

1. Overview

The first project is going to be a MIPS assembler. In general, you will be building a program that translate the assembly language file to an executable file (machine code). The input of your program will be a MIPS file that contains MIPS assembly language code. Examples will be given in the latter part of this instruction.

1.1 Readings

Please read Appendix A from the text book before you start writing your first project. Make sure you understand what each MIPS instruction is doing. All of the supplementary materials for this project can be found in Appendix A, such as the register numbers, instructions, and their machine code format.

2. MIPS

MIPS is an assembly language, which is a low level programming language that each line of code corresponds to one machine instruction.

2.1 Instruction

Machines cannot understand high level programming languages directly, such as C/C++/JAVA. High level programming languages are "translated" to machine instructions that machine can understand. Assembly languages, including MIPS we deal with, are the readable (although difficult) version of machine code. In assembly languages, one instruction tells the computer to do one thing exactly. For example, an instruction may look like this:

```
add $t3, $t1, $t2
```

This instruction tells the computer to add up the things stored in register $t1 and register $t and store the result in $t3. Here, registers are small chunks of memory in CPU used for program execution. The MIPS assembly language has three types of instruction in general: I-type, R-type, and J-type, each corresponds to a pattern of the 32 bits of machine code. Details can be found in Appendix A of the text book. The above instruction has the machine code:

```
00000001001010100100000000100000
```

It does not make sense at a glance, however, it follows a certain pattern. The add instruction is a R-instruction, so it follows the pattern of:

```
R-instruction:
    op      rs     rt       rd    shamt   funct
| 6bits | 5bits | 5bits | 5bits | 5bits | 6bits |
1. op: operation code, all zeros for R-instructions.
2. rs: the first register operand
3. rt: the second register operand
```

```
4. rd: the destination register
5. shamt: shift amount. 0 when N/A
6. funct: function code, which is used to identify which R-instruction this is.

The add instruction has the format:
add $rd, $rs, $rt

Therefore, for add $t0, $t1, $t2, we have:
000000 01001 01010 01000 00000 100000
```

Here, we go through how this instruction and its machine code corresponds to each other.

```
1. The first 6 bits for R-instruction are for operation code, which are all zeros
   for R-type.
2. The following three 5-bit slots are the register numbers specified in the
   instruction. "rs" and "rt" represents the first and second register operand in
   the instruction, and "rd" represents the destination register. Here, the register
   number of $t0, $t1, and $t2 are 8, 9, 10, respectively. These translate to 01000,
   01001, 01010 in binary, respectively. Putting them in the right place, we have
   the middle 15-bits.
3. "shamt" represents the shift amount, which are only in certain instructions
   (such as sll, srl), and 0's are filled when N/A. In add instruction, these 5 bits
   are zeros.
4. The last 6 bits are for function code. The function code for add is 32, which
   is 100000 in binary.
```

Machine code of other R-instructions are constructed through the same process. As for I- instructions and J-instructions, I will not go through an example. The formats of these two types of instructions are:

```
I-instruction:
    op      rs      rt        immediate
| 6bits | 5bits | 5bits |      16bits     |

1. op: the operation code that specifies which I-instruction this is.
2. rs: register that contains the base address
3. rt: the destination/source register (depends on the operation)
4. immediate: a numerical value or offset (depends on the operation)

J-instruction:
    op            address
| 6bits |         26bits        |

1. op: the operation code that specifies which J-instruction this is.
2. address: the address to jump to, usually associate with a label. Since the
   address of an instruction in the memory is always divisible by 4 (think about
   why), the last two bits are always zero, so the last two bits are dropped.
```

2.2 MIPS programs

Now you know what do MIPS instructions look like, but what does a MIPS program look like?
Here is a general format a MIPS program follows:

```
.data #static data go here
str1: .asciiz "hello world!\n"
.text #MIPS code goes here
main: add $t0, $t1, $t
...
```

In general, a MIPS program looks like this. All of the MIPS code goes under the .text section, and all of the static data in the program are under .data section. As you can see, for each piece of static data, we have a name to it, just like what we do in high level programming languages. "str1" is the name to that piece of data, .asciiz is the data type of it, and "hello world!\n" is the value.
There are many other types of data, you can find them in Appendix A.

For the code part, as you can see, we also have a "name" called "main". This is the label representing the line of code. Usually this is used for indicating the start of a loop, a function, or a procedure. Recall that all of these codes are stored somewhere in the memory, which means that each line of code has a specific address. To better understand this, see section 3.2.1

3. Project 1 details

3.1 Environment

1. Your project 1 should be written in C/C++/Pyhthon only.
2. For C/C++ users, you will need to write your own makefile/cmake, and make sure your program can execute without a problem on the VM/Docker they provided. You can access the testing environment through VM instruction on BB.If you would like to write your program in Visual Studio or other IDE, please test your programs on the virtual machine before submitting your final version (refer to 'Virtual Machine Setup'), yet you can develop your program in your own environment as normal. For ARM users who choose C/C++, you can follow this guide to set up Ubuntu and install Icarus Verilog, GNU C/C++ toolchains at specific versions(consistent with 'Virtual Machine Setup').

3.2 Assembler

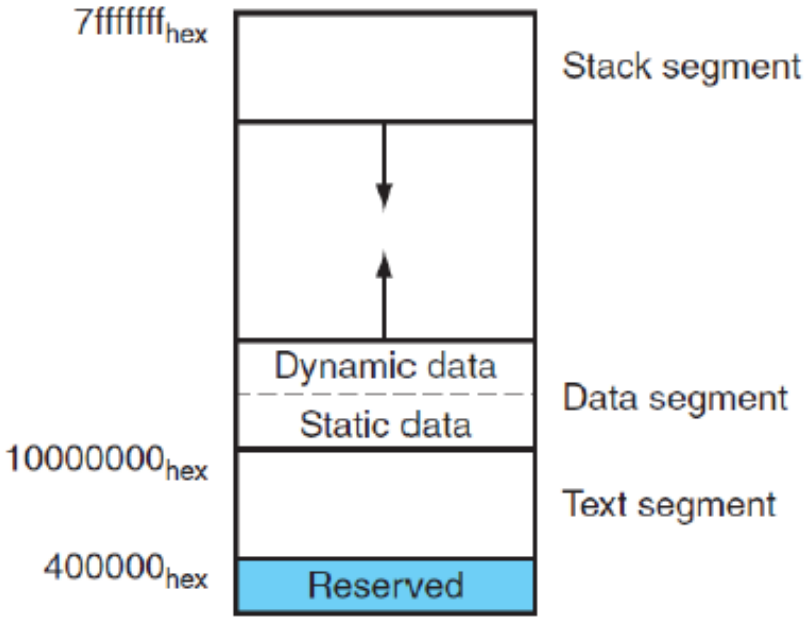
3.2.1 Overview

The first task you are doing is assembling the given MIPS code to their corresponding machine code. Here's a quick example of what you need to do:

```
# MIPS code:
.text
R: add $s0, $s1, $s2 #r instructions
addu $s0, $s1, $s2
sub $s0, $s1, $s2
subu $s0, $s1, $s2

Machine code:
00000010001100101000000000100000
00000010001100101000000000100001
00000010001100101000000000100010
00000010001100101000000000100011
```

Please note that do not assemble the .data section, nor the labels. The .data section will be loaded to your memory in your **simulation part**. The labels in your .text section will be translated to its corresponding address when needed. For example, when you see j R, you will put the address of the line of instruction label R is indicating. We assume that **addresses of instructions start from 0x400000** in this project.



3.2.2 Details

For a detailed list of instructions you need to support, please refer to "**MIPS Instruction Coding**" in BB.

This part of your program is easy but needs your patience. You need to:

1. Read files line by line.
2. Find segments of the MIPS file. (.data, .text)
3. Tokenize the line read in.
4. Discard useless information, such as comments.

Here are some ideas of how to implement this part:

1. You need to scan through the file for the first time. This time, discard all of the comments (following a

"#"). Remember you only need to deal with .text segment for assembling. Find all of the labels in the code, and store them with their corresponding address for later reference. This procedure should be included in **phase1.c/cpp/py**. Meanwhile, a proper data structure is needed to store that information. The program to maintain such a data structure is **labelTable.c/cpp/py**.

2. You need to scan through the file for the second time, line by line. This time, you need to identify which type of instruction the line is (R, I, J). According to the instruction type, you can assemble the line. For lines with the label, you can refer to the stored information for the label's address. This procedure should be included in **phase2.c/cpp/py**.
3. Do not write everything in one .c/cpp/py file.

3.3.3 Input, Test and Outout

1. ***tester.c*** This is where the main function locate. We will use this function to test your program and so can you. You can write your tester.c/cpp/py compatible with your programs.

2. ***testfile.asm, output.txt*** We provide two test cases in the form of .asm file. These are the test files you will look into. You can always create more test cases yourselves, just to make sure you are doing it correctly.

3. The **output** should be a .txt file which contains the machine code of the corresponding input .asm file.

4. Miscellaneous

4.1 Submission

- **Due on: 23:59, 6 Mar 2022** (Late submission within 5 minitues is allowed without punishment)
- Please note that, teaching assistants may ask you to explain the meaning of your program, to ensure that the codes are indeed written by yourself. Please also note that we would check whether your program is too similar to your fellow students' code using **plagiarism detectors** for all assignments.
- Your submission should contain source code(phase1, LabelTable, phase2, tester), makefile/cmake and report. Please compress all files in the file structure root folder into a single zip file and **name it using your student id as the code showing below and above, for example, Assignment_1_118010001.zip.** The report should be submitted in the format of **pdf**, together with your source code. Format mismatch would cause grade deduction. Here is the sample step for compress your code.

```
main@ubuntu:~/Desktop$ zip -q -r Assignment_1_<student_id>.zip
Assignment_1_<student_id>
main@ubuntu:~/Desktop$ ls
Assignment_1_<student_id>                  Assignment_1_<student_id>.zip
```

- Violation against each format requirements will lead to **5 demerit points**. (zip file, file name)
- C/C++ users need to include a **Makefile/cmake** in your folder, and make sure your code is able to compile and excute in the Ubuntu environment we provide, missing makefile/cmake will cause **5 demerit points**.

4.2 Grading

This project worth 15% of your total grade. The grading details of this project will be:

Phase1 of the project - 20%

labelTable - 20%

> Construct reasonable data structures - 10%

> Work correctly to find and save labels - 10%

Phase2 of the project - 50%

> Find corresponding instruction type - 15%

> Map correct machine code to instructions, registers, etc - 25%

> Correctly output a .txt file - 10%

Project report (**PDF**) -10%

> In your report, you should write:
>
> 1. Your big picture thoughts and ideas, showing us you really understand how a MIPS assembler works.
> 2. The high level implementation ideas. i.e. how you break down the problem into small problems, and the modules you implemented, etc.
> 3. The implementation details. i.e. what data structures did you define and how are they used.
>
> Figures such as work flow and corresponding assistant interpretation are expected.
>
> Please do not use screenshots of code blocks when presenting core code, which are usually regarded as informal.

> Other document formats like Word, TXT, etc will result in **5 demerit points**.

4.3 Honesty

We take your honesty seriously. **If you are caught copying others' code, you will get an automatic 0 in this project.** Please write your own code.