Project 1

1. Overview

The first project is going to be a MIPS simulator. In general, you will be building a program that simulates the execution of an assembly language file. 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 $t0, $t1, $t2
```

This instruction tells the computer to add up the things stored in register $t1 and register $t2 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 codes 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, $t2
...
```

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 2.3.

2.3 How computer runs the program?

With the idea that all of the codes are stored in memory, and each has an address, we can now talk about how computers run these codes. Long story short, the computer runs the programs following the machine cycle.

2.3.1 Machine cycle

A shorter version of a machine cycle looks like this:

```
1. The computer loads the line of instruction PC is "pointing at".
2. The computer increment PC by 4 (think about why).
3. The computer runs the instruction loaded.
```

This goes on until the program terminates. PC in this context represents the "program counter". In other words, **PC is the "pointer" the computer maintains that stores the address of the next instruction to be executed**.

3. Project 1 details

3.1 Requirements

1. Your project 1 should be written in C/C++ only.
2. You will need to write your own makefile/cmake.
3. The testing environment is provided by Mr. Yifan Zhu and Mr. Guochao Xie. You need to 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.
4. The detailed list of the MIPS instructions you need to support will be announced later.

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
```

Remember that you are not assembling 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.

3.2.2 Details

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

1. Read file line by line.
2. Find the 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.
2. You need to scan through the file for the second time, line by line. This time, you need to identify which 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.

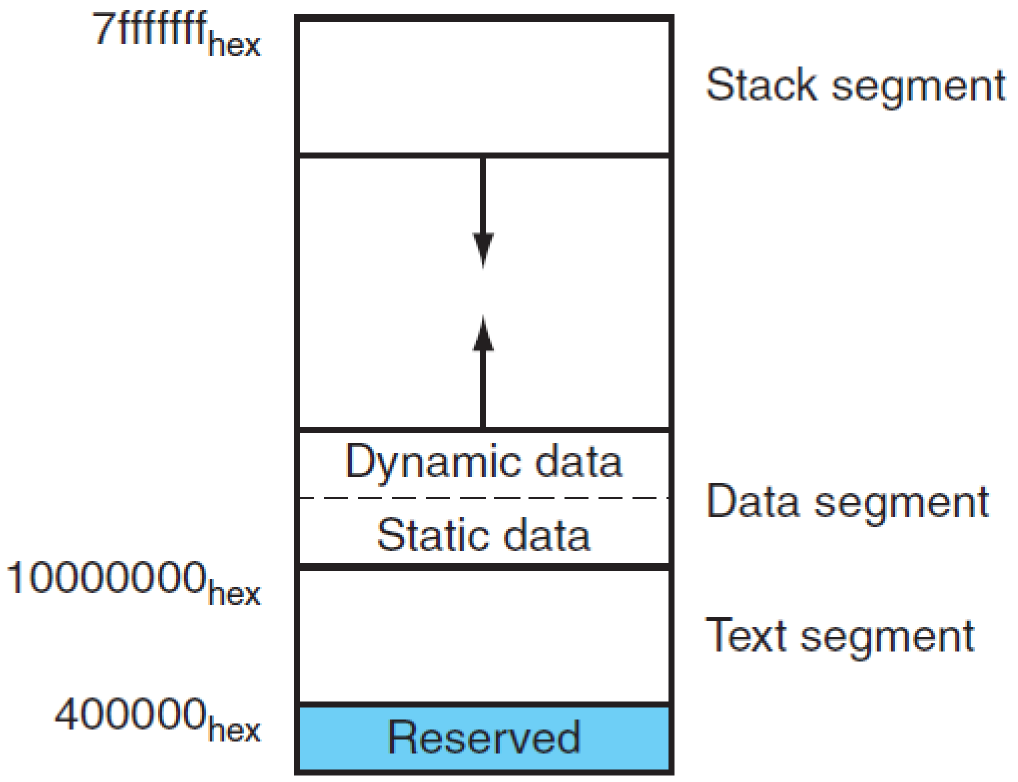
3.3 Simulator

3.3.1 Overview

This is the major part of your project 1. You need to have a full understanding of how computer executes programs, and how are things stored in memory. Your code will need to be capable of executing the MIPS code line by line.

3.3.2 Memory & register simulation

The first thing you will need to do is memory simulation. Think about your simulator as a mini-computer, that has its own main memory, CPU, etc. To simulate main memory, you need to dynamically allocate a block of memory with C/C++, with a size of 6MB. Here is a figure of what does a real computer memory look like.



Your simulated memory should also have these components. Also, since most of you are using a 64-bit computer, you need to "translate" the real address of your allocated memory to a 32-bit simulated address. Specifically:

1. Let's say you have the pointer named "real_mem" storing the real address of the block of memory allocated. The first thing you need to do is to map the value of "real_mem" to 400000_hex. Then the real address will have a 1-to-1 mapping relationship to the simulated address. For instance, if the address mentioned in the MIPS testing file is 500000_hex (such as lw, where we want to load the data storing on 500000_hex), then, you should access it at real address of: (real_mem + 500000_hex - 400000_hex).
2. The dynamically allocated 6MB memory block is pointing at the start of your text segment, and your text segment will be 1MB in size. The end of text segment will be at simulated address 400000_hex+1MB, or at address real_mem+1MB.
3. The static data segment will start at simulated address 500000_hex, or at real address (real_mem+1MB).
4. The dynamic data segment will start at wherever your static data section ends.
5. The stack segment will start at the highest address A00000_hex (real_mem+6MB), and it grows downwards (whenever you put things in there, the address decreases).

You should also simulate the registers. The registers should not be a part of your simulated memory. Recall that registers are in CPU. In this project, you are not accessing the real registers, however, you will allocate memory for the 32 general purpose registers, which are:

Register name	Number	Usage
$zero	0	constant 0
$at	1	reserved for assembler
$v0	2	expression evaluation and results of a function
$v1	3	expression evaluation and results of a function
$a0	4	argument 1
$a1	5	argument 2
$a2	6	argument 3
$a3	7	argument 4
$t0	8	temporary (not preserved across call)
$t1	9	temporary (not preserved across call)
$t2	10	temporary (not preserved across call)
$t3	11	temporary (not preserved across call)
$t4	12	temporary (not preserved across call)
$t5	13	temporary (not preserved across call)
$t6	14	temporary (not preserved across call)
$t7	15	temporary (not preserved across call)
$s0	16	saved temporary (preserved across call)
$s1	17	saved temporary (preserved across call)
$s2	18	saved temporary (preserved across call)
$s3	19	saved temporary (preserved across call)
$s4	20	saved temporary (preserved across call)
$s5	21	saved temporary (preserved across call)
$s6	22	saved temporary (preserved across call)
$s7	23	saved temporary (preserved across call)
$t8	24	temporary (not preserved across call)
$t9	25	temporary (not preserved across call)
$k0	26	reserved for OS kernel
$k1	27	reserved for OS kernel
$gp	28	pointer to global area
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address (used by function call)

Your code should initiate the registers as described by its functionality. For example, the stack pointer register, $sp, should always store the current stack top. You should initialize it with a value of 1000000_hex.

3.3.3 Putting things in the right place

Your simulator should take a MIPS file as input, and you should put everything in the right place before your simulation. After the simulated memory is ready, you will read a MIPS file, and:

1. Put the data in .data segment of MIPS file piece by piece in the static data segment. The whole block (4 bytes) is assigned to a piece of data even if it is not full. For example:

```
.data
str1: .asciiz "hello"
int1: .word 1

in memory:
| hell | o\0-- |  1  |
```

Here, each character of .asciiz type occupies 1 byte, so "hell" occupies the first block. The first two bytes of the second block is used by "o" and a terminating sign "\0", but the last two bytes of the second block is not used. However, when we put the next piece of data in the memory, we start a new block. The data type .word occupies 4 bytes, so the third block is assigned to this piece of data.

2. Assemble the .text segment of the MIPS file (section 3.2), and put the assembled machine code in the text segment of your simulated memory (the first line of code has the lowest address). The assembled machine code is 32 bits, which is 4 bytes. Thus, you can translate it to a decimal number and store it as an integer.

3.3.4 Start simulating

Your code should maintain a PC, which points to the first line of code in the simulated memory. Your code should have a major loop, simulating the machine cycle. Following the machine cycle, your code should be able to:

1. Go to your simulated memory to fetch a line of machine code stored at the address PC indicates.
2. PC=PC+4
3. From the machine code, be able to know what the instruction is and do the corresponding things.

The third step of the machine cycle requires you to write a C/C++ function for each instruction to do what it's supposed to. For example, for the `add` instruction, we can write in C:

```
void add (int* rs, int* rt, int* rd){
    *rd = *rs+ *rt;
}
```

In the third step of the machine cycle, when we read a line of machine code and the corresponding instruction is `add`, we can simply call the function.

4. Miscellaneous

4.1 Deadline

The deadline of this project is 3/21/2021 midnight.

4.2 Submission

You should put all of your source files in a folder and compress it in a **zip**. Name it with your **student ID**. Submit it through BB.

4.3 Grading

This project worth 30% of your total grade. The grading details of this project will be:

1. Assembling - 20%
2. Memory & register simulation - 30%
3. Proper MIPS execution - 40%

    Each unsuccessful MIPS instruction will result in 3% deduction.
4. Report - 10%

NOTICE: If your code does not support makefile/cmake, you will lose 50% automatic.

4.4 Report

1. The report of this project should be **no longer than 5 pages.**
2. In your report, you should write:
    1. Your big picture thoughts and ideas, showing us you really understand how are MIPS programs executed in computers. (This part should be the major part of your report.)
    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 structures did you define and how are they used.
3. In your report, you should not:
    1. Include too many screenshots your code.
    2. Copy and paste others' report.

4.5 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.