Project 2

1. Overview

The second project is going to be a MIPS simulator. In general, you will be building a program that simulates the execution of a binary file. There will be several input files of your program, including a MIPS file that contains MIPS assembly language code, which give you the static data infomation; a BIN file which contains the corresponding machine code; and a checkpoints file to help in debugging and grading your program. Further details will be given in the latter part of this instruction.

1.1 Readings:

Same as Project-1, 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. Moreover, project 2 is based on your first project, foundamental knowledge can be found in the material of Project-1, such as the MIPS Instruction List.

2. How computer runs the program?

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

2.1 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 2 details

3.1 Environment

1. Project 2 should be written in C/C++/Python 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. 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 Simulator

3.2.1 Overview

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. Your simulation **should support all instructions and data types in "MIPS Instruction Coding"**.

3.2.2 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++/python, with a size of 6MB. Here is a figure of what does a real computer memory look like.
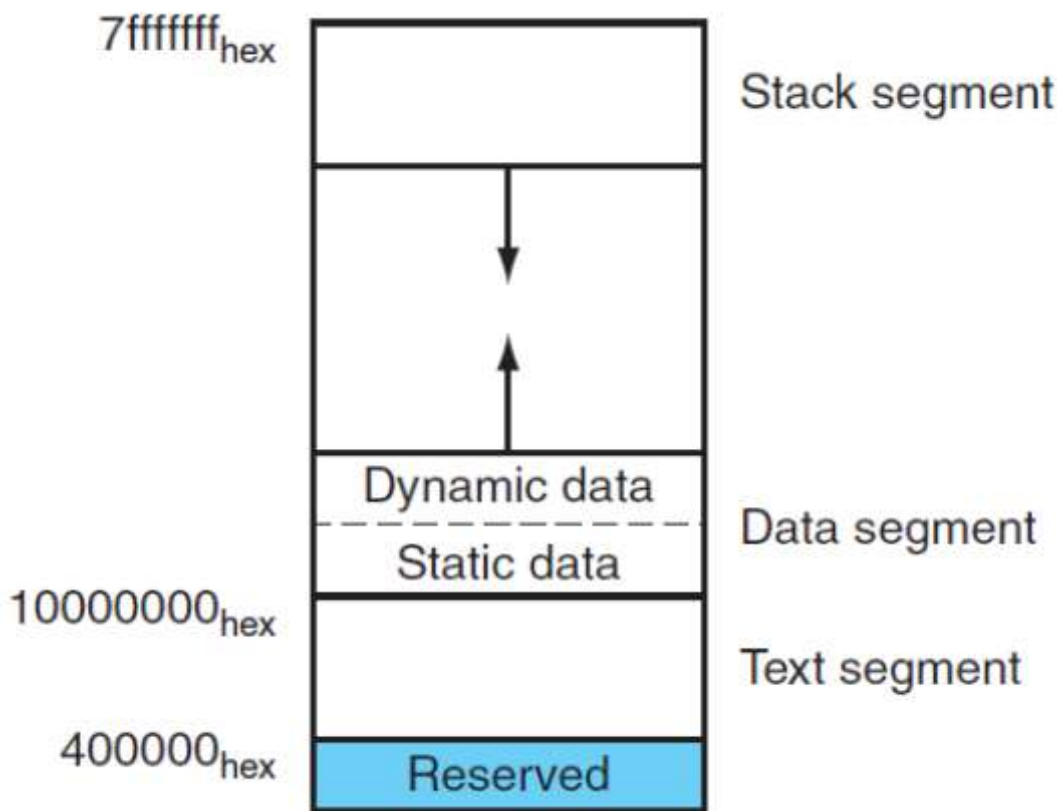
Your simulated memory should also have these components. Also, since most of you are using a 64-bit computer, you need to "translate" the real address of your allocated memory to a 32-bit simulated address. Specifically:

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 general purpose registers, the PC register, the HI register, and the LO register. The 32 general purpos registers 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.

3.2.3 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 (what you did in Project-1, we will also provide the correct machine code), 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.

3.2.4 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 function for each instruction to do what it's supposed to. For example, for the add instruction, we can write in C:

```
1  void add (int* rs, int* rt, int* rd){
2      *rd = *rs+ *rt;
3  }
```

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.

3.2.5 Inputs, tests and outputs

The command for running your program is given by:

```
./simulator test.asm test.txt test_checkpoints.txt test.in test.out
```

test.asm: An input test file of assembly codes, which is used for static data loading.

test.txt: An input file for assembled binary codes, which is used for binary code loading.

test_checkpoints.txt: An input file indicating where memory and registers' snapshots should be dumped. This file is just for testing the correctness of your program and grading. **See document `Checkpoints specifications.pdf` for more details**.

test.in: An input file storing inputs for some read-related I/O operations (read_int, read_char, read_string).

test.out: The name of the output file storing the outputs for print-related I/O operations.

4. Miscellaneous

4.1 Deadline

- **Due on: 23:59, 20 Mar 2022** (Late submission within 5 minitues is allowed without punishment)

4.2 Submission

- Your submission should contain source code, makefile/cmake and report. Name your compile output (excutable file) as **simulator(c/c++) or simulator.py(python)**. 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_2_118010001.zip**. The report should be submitted in the format of **pdf**, together with your source code. Format mismatch would cause grade deduction.
- 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**.

4.3 Grading

The grading details of this project will be:

1. Memory & register simulation - 20%
2. Proper MIPS execution - 70%
3. Report - 10%

4.4 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.

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.