CSC3050 Assignment 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 DEBUG 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, fundamental knowledge can be found in the material of Project-1, such as the **MIPS Instruction List**.

1.2 Environment

1. Project 2 should be written in C/C++/Python only.
2. Since this project will use Linux functions, you are recommended to use virtual machine of Ubuntu 20.04. Please test your programs on the virtual machine before submitting your final version.
3. For C/C++ users, you will need to write your own makefile/cmake, and make sure your program can be compiled and executed.

1.3 Basic knowledge: how computer runs programs?

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.

1.3.1 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.

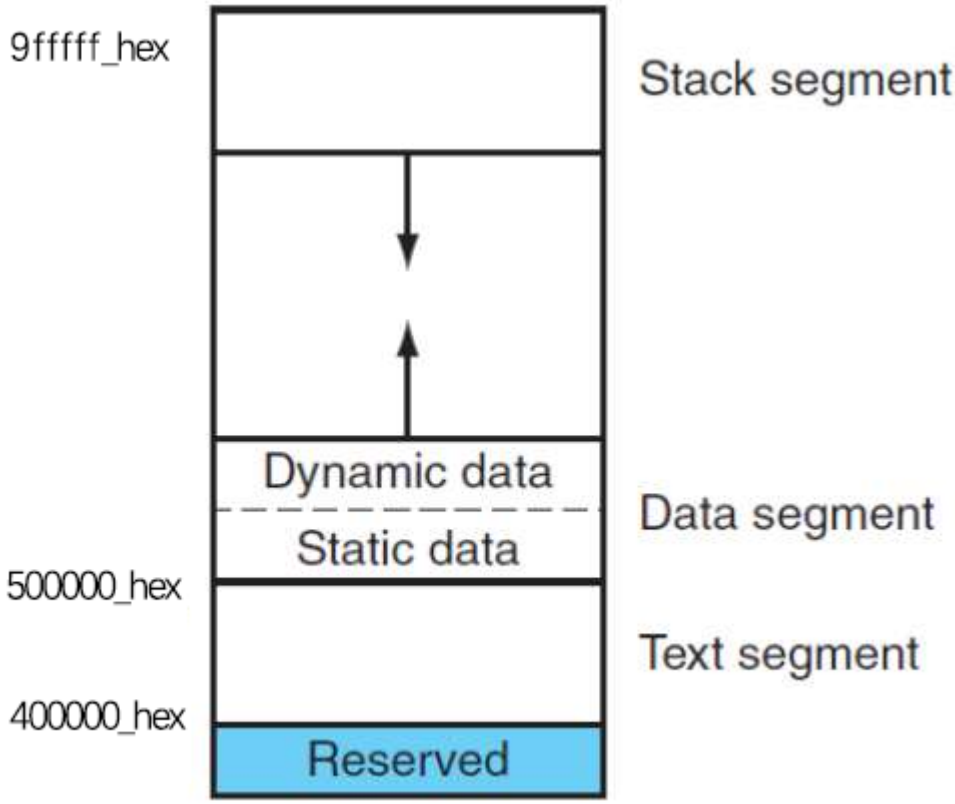
2. Project 2 details

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 LIST.pdf”. To be specific, your works can be divided into four parts: memory simulation, register simulation, initialization, and simulating.

2.2 Memory 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.



You need to map the real address of your allocated memory to a 32-bit simulated address.

To be clear, let's say you have a pointer named "real_mem" storing the real address of the block of memory allocated, which is the starting address of the block. Then the pointer "real_mem" should represent 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).

Let me use c++ to further explain the concept. For example, to allocate 6MB in c++, one can use `char memory_simu[0x600000];` because 6 MB = 6*1024 *1024 B. Denote the address we want to access as `address_acc` (which is obviously between 0x400000 and 0xA00000). Denote the pointer `real_mem = 0x000000`. Then the mapping functions can be `memory_simu[real_mem + address_acc - 0x400000]`.

About the specific segments:

1. 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).
2. The **static data segment** will start at simulated address 500000_hex, or at real address (real_mem+1MB).
3. The **dynamic data segment** will start at wherever your static data section ends.
4. 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).

2.3 Register simulation

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 purpose 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)

Register Name	Number	Usage
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 A00000_hex.

2.4 Initialization: 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.

2.5 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++/Python 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.

2.6 Detail specification

2.6.1 Tips in memory and register simulation

- For simplicity, you just need to initialize `$fp` with the same value as that initialized in `$sp`, and you can just initialize `$gp` with the address 32KB above the beginning of the static data section (that is, 0x508000).(for detailed reasons, you may refer to the textbook pages 102-106 and appendix A.5)
- For PC, you should keep PC value to be the first instruction that is not executed yet.(e.g. when you just finish executing the instruction at 0x400004 and dump the register layout out, you should keep PC=0x400008 in your dumped binary file)

2.6.2 Tips in syscalls

- You only need to support the syscalls 1, 4, 5, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17. (Details are in MIPS Instruction List.pdf)
- For syscalls 10, 13, 14, 15, 16, 17, you should simulate by **directly invoking the Linux APIs** (some of them have been discussed in tutorial 4) with the parameters given in the "Arguments" column of the system service table in "MIPS Instruction List".You can just regard syscall 10's behavior to be a normal exit (with status code 0).
- For syscall 9, you need to simulate the program break in your allocated 6MB memory, and make sure to return a pointer to the location in dynamic data so that we can put stuff in.
- For syscalls 5, 8, 12, you just need to read from .in file one line at a time. For syscalls 1, 4, 11,you can just print the argument in the .out file one line at a time.
- You don't need to consider negative numbers for syscall(sbrk).
- All system calls have been covered in the test cases released, so you may check to see their usages in testing.

2.6.3 Tips in initialization and simulating

- You don't need to consider overflow exception in instruction add.
- You don't need to consider negative numbers for these instructions(div divu mult multu)
- For the data types you need to support, you don't need to consider non-integer numbers(such as float or double) in `.word`/`.byte`/`.half`.
- You can just ignore the labels in `.data`.
- To simplify the problem, only `.ascii` and `.asciiz` use Big-Endian, and `.word` `.half` `.byte` use the Little-endian for storage in memory.

2.7 Inputs, tests, and outputs

The command for running your program is:

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

Or in Python, we may run your program with the following command:

```
python simulator.py 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.

2.8 Report guide

The report of this project should **be no longer than 6 pages**, and you should not include too many screenshots of your code. In your report, you should write:

1. Your big picture thoughts and ideas, show us you really understand how are MIPS programs are executed in computers.
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. Miscellaneous

3.1 Deadline

- **Due on: 23:59, 29 Oct 2023** (Late submission within 5 minitues is allowed without punishment)

3.2 Submission

- 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, 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_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**.

3.3 Grading

The grading details of this project will be:

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

3.4 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.