Simple Calculator

32190984 Isu Kim

April 6, 2023

1 Index

2 Introduction

Professor Nam gave us a homework to implement a simple MIPS-like simulator. His expectations on this homework were as it follows:

- Support 10 registers.
- Support following instructions:
  - `ADD`: Add two registers.
  - `SUB`: Subtract two registers.
  - `MUL`: Multiply two registers.
  - `DIV`: Divide a register by another register.
  - `MOV`: Move a constant or register value to a register.
  - `LW`: Load a constant into a register.
  - `SW`: Print out a register value by `STDOUT`.
  - `RST`: Reset all register values.
  - `JMP`: Jump into a specific line of code.
  - `BEQ`: Compare two constants or registers and jump if they are equal.
  - `BNE`: Compare two constants or registers and jump if they are not equal.
  - `SLT`: Compare if first constant or register is smaller than the another.
- Read file named `input.txt` and execute it line by line.
- Print out state changes of registers at the end of each instructions.
- Halt when reaching `EOF`.
- Handle exceptions "gracefully".

For implementing the program, I am going to use `C`. Not only did professor recommend `C` over any other languages, but it is quite simple to implement compared to any other languages. Also, from now on, we are going to call this project as `mipsim`.

In order to implement those requirements mentioned up above, we need to design the program before we actually implement it. All the considerations on implementation of this project will be discussed in the later section.

3 Background

The most important concept that we need to keep in mind when designing this program is the concept of "stored program" introduced by *John von Neumann* and "Turing machine" by *Alan Mathison Turing*.

3.1 Turing Machine

In 1936, Alan Turing proposed an import concept named "Turing machine". Since this is not an automata class, we are not going to discuss deeply into the subject. Otherwise, it will be too long to be included in this document. Turing machine consists of an infinite tape divided into small squares, a head that can read the write symbols on that tape and a set of rules for determining the next state of the machine based on the current state and the symbol under the head. Which looks like following figure:

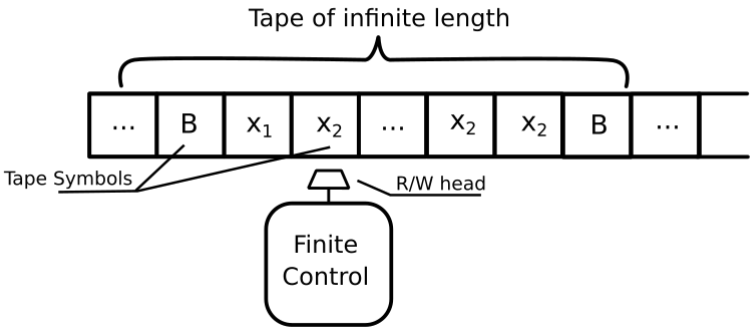

Figure 1: Turing Machine Concept - https://iq.opengenus.org/content/images/2019/07/TurigMachine.png

Head moves from one square to the next square and executes sets of instructions in sequence. We can think of heads as PC(Program Counter) in MIPS assembly, or IP(Instruction Pointer) in x86 architectures. In a Turing machine, the head reads and writes symbols on the tape and moves along the tape according to a set of rules that determine the next state of the machine. The position of the head on the tape represents the current state of the machine, and the tape contains both the data being processed and the instructions for the machine.

3.2 Stored Program

In the mid-1940s, John von Neumann and his friends introduced a concept named "Stored Program". This was a way to design computers that could execute instructions stored in the memory. The basic idea was to store both data and instructions in the same memory, and to use a program counter to keep track of the address of the next instruction to be executed.

3.3 Summary

Since this is a computer architecture class, I thought that those two concepts were worth mentioning before we start designing and implementing the program. In order to implement `mipsim`, I tried to implement it in a way that as much mimics the workflow of computers. For specific considerations on implementations, we are going to discuss them in the next section.

4 Design & Implementation

4.1 Background

Of course, there might be more core design principles when it comes to designing a program, there were two major considerations for this program:

- **Stability**: The program shall be stable, meaning that the program must have "graceful" error handling rather than `SEGSEGV`.
- **Extensibility**: The program shall be easy to add some extra features in the future, such as new instructions and new registers.

In order for making those two principles alive, the program was designed in two major components:

- **Assembler**: A component that reads file from user and makes it into a instruction that will be performed by the simulator in the future.
- **Simulator**: A component that actually executes instructions that were generated by assembler accordingly.

There were more minor components that were involved in this program to run smooth. Besides those requirements from Professor Nam, I have added some more requirements that will make this program perform better:

- **Error Detection**: When the input file that the program was processing had an error, the program is required to tell user at which line there seems to be an error. For now, this only detects syntax errors instead of logical errors like infinite loops.
- **Command Line Interface**: For ease of use for users, a simple command line interface was supported. Such includes specifying an input file, specifying an output file.

4.2 Design Components

4.2.1 Assembler

As the requirements of `HW1.pdf` suggests, the user will input assembly-like instructions inside a file. In order for the program to execute instructions from string, the program first needs to parse those strings into instructions. The assemble was designed to perform this task. In short, the assembler will read the file provided by the user line by line and store them as an array of instructions. Those instructions will be stored inside the memory and will be executed later by simulator.

One might ask, "why not execute string instructions line by line instead of storing them inside a memory?". Well, there are three main reasons for current design: reading all file and storing them as instructions:

- **Mimic Stored Program**: Since we are studying computer architecture class, I just wanted to make this program mimic *stored program* concept introduced by *John von Neumann*.
- **Speed and Optimization**: Since the program will read input file, parse string and execute instruction accordingly, the speed of executing a single instruction compared

to stored instructions will be far slower due to the fact that there will be more IO operations involved. Not only does it require more IO operations, but there will be redundancy issues. For example, when a specific line of input is executed multiple times due to operations like `jmp` or `beq`, the program will perform the same operation to parse string into instructions multiple times. Therefore, for speed and optimization, stored instructions method was preferred.

- **Error Detection**: Since one of the requirements that this program has to offer were error detection, with assembler it will be easier to implement syntax error detection rather than line by line execution. Since the assembler will read all the lines before executing and check whether there is an error or not, the program will be more stable since it will have little chance of executing some unverified code that might have potential of bugs and errors.

4.2.2 Simulator

Implementing a simulator is rather a simple process compared to assembler. The simulator will take current context of simulation and execute the user's instruction that was requested. To mimic real computer operations, the simulator works with a simple variable named `pc`. Also, the simulator was designed with as much error handling as possible. For details on limitations on simulator, please check later section for more information.

4.3 Implementation

4.3.1 Type Definitions

Before starting to talk about details on implementations, we will now talk about type definitions first. There are some data types that were defined in the program for implementing the simulator. Those definitions can be found under `common.h`.

The first is the user's instructions and parameters. Those are tightly coupled together for future use.

```
struct param_t {
    uint32_t value;
    uint8_t type; // Distinguish registers from constants.
};

struct user_instruction_t {
    uint8_t instruction;
    struct param_t param[3];
    uint8_t param_count;
};
```

Code Snippet 1: Definition of `struct user_instruction_t` and `struct param_t`

`struct user_instruction_t` are stored inside an array to mimic *stored program* concept. This array will be used later inside the simulator.

The second is the context. Since the simulator needs to keep track of current contexts, such as

PC and register states, the data type was introduced.

```
struct context_t {
    char input_file[MAX_STRING];
    char output_file[MAX_STRING];
    FILE *in_fp;
    FILE *out_fp;
    uint32_t pc;
    uint32_t registers[REGISTER_COUNT];
    uint16_t used_map;
    uint32_t last_instruction;
    struct user_instruction_t user_instructions[MAX_INSTRUCTION_COUNT];
};
```

Code Snippet 2: Definition of `struct context_t`

`struct context_t` store major information such as input file, output file, register status with their values and user's instructions. In order to keep track of used registers, this program uses `uint16_t` to store whether a specific register was used or not. LSB represents 0th register's status, if marked 1 this means that the register was being used. Since there are only 10 registers available, `uint16_t` was considered enough to represent all registers.

The last type definition is the instruction. In order to save the syntax and function pointers of a specific instruction, this data type was introduced.

```
struct instruction_t {
    enum TokenType syntax[MAX_TOKEN_COUNT];
    uint8_t token_count;
    int (*function)(struct context_t *, struct user_instruction_t);
    char name[MAX_NAME_LEN];
};
```

Code Snippet 3: Definition of `struct instruction_t`

At the start of the program, all available instructions are turned into an array of `struct instruction_t`. Then with this instructions, the program checks predefined syntax. Also, for implementing "polymorphism", function pointer was used to call the function that actually executes the instruction.

With those newly defined data types, the program performs syntax checking, assembling and simulating operations.

4.3.2 Overview

This program works in following sequence with give input file:

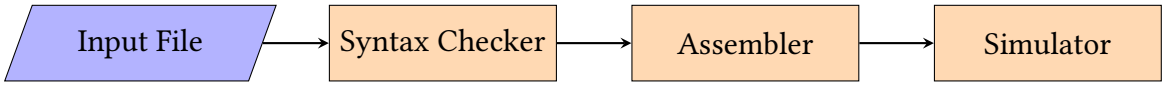

Figure 2: Syntax Checker Flowchart

From now on, we are going to talk about each sequences one by one. Actually, syntax checking and assembling is taken place at the same time. However, for ease of description, we are going to talk about each topics one by one.

4.3.3 Syntax Checker

Syntax checker verifies syntax of user's input file. The program will read line by line from the user's input file and store it as a `string`. Once the string was stored, it will check for syntax and verify if the input was valid or not. Again, keep in mind that syntax checker is actually implemented with the assembler in the code. For easy explanation, we are going to separate them in this document. The syntax checker works in following sequence as flowchart shows below:

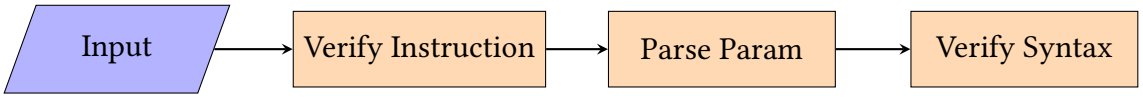

Figure 3: Syntax Checker Flowchart

1. Syntax checker will verify the instruction. For example, if the user's input was `LW r0 0x5`, this step will verify that the instruction was given as `lw`.
2. Syntax checker will parse parameters. This will distinguish a constant from register. For example, if `r0` was given, it will parse it as register. On the other hand, `0x5` will be parsed as a constant. Also, in this step, each parameters will be stored as `struct param_t` type to indicate the type of parameter.
3. Syntax checker will verify syntax. In this step, the program will check if the types of each parameters that were parsed from previous step was in a valid syntax. Details on this step will be discussed right below.

Now, it is time to discuss about how syntax checker verifies if a syntax is valid or not. When the program loads, the program will automatically generate a predefined syntax of each instructions. Then it will compare each given parameter's types with the predefined syntax. To implement this, as we all know, there are three types of arguments for each different instructions:

- **Register**: The register information.
- **Constant**: The constant value which are represented as hexadecimal.
- **Both**: Either register or constant works.

As Professor Nam requested, we can represent syntax of each instructions like following table:

Name	Syntax	Token Count
add	Register, Register, Register	3
sub	Register, Register, Register	3
mul	Register, Register, Register	3
div	Register, Register, Register	3
mov	Register, Both	2
lw	Register, Constant	2
sw	Register	1
rst		0
jmp	Constant	1
beq	Both, Both, Constant	3
bne	Both, Both, Constant	3
slt	Register, Both, Both	3

Figure 4: Syntax Table of Instructions

For ease of implementation, two instructions were implemented in a slightly modified way:

- `rst`: Originally required format of `rst $register STDOUT`. However, in order to implement the program a bit easier, this program omits `STDOUT` since there is just only one option.
- `slt`: Originally the first argument was fixed as `$r0`, however this program supports other registers as well.

As we construct this table, it will be easier for us to implement the syntax checking feature. With this table, by comparing each instruction's type with the predefined syntax, we can achieve syntax checking.

Let's check an example input below to see how this works.

```
LW r0 0x5 # Load 0x5 to register 0
```

Code Snippet 4: Example Input

Before the program checks the syntax, it will drop everything after `#` which are comments. In the first step, syntax checker will identify `LW` as instruction `lw`. Once it identifies the user's instruction, it will then parse each passed arguments. Since the code had `r0` and `0x5`, each will be parsed as `Register` and a `Constant`. Then, it will check syntax. Since we have a predefined syntax like below:

```
{ .syntax = {Register, Constant}, ... .name = "lw"},
```

Code Snippet 5: Syntax Definition of `LW`

The full code can be found under `generate_instructions` of `main.c`. Since the user's given syntax was in the correct sequence, `Register, Context` it will pass syntax check. Once the syntax was valid, the program will then move onto the next step: assembling.

4.3.4 Assembler

As mentioned earlier, while the syntax checker verifies for the syntax, it also generates instruction at the same time. The generated instruction will be stored as a `struct user_instruction_t` type for future use. For example, if we had a instruction like Code Snippet 4, the program will store instruction as following figure:

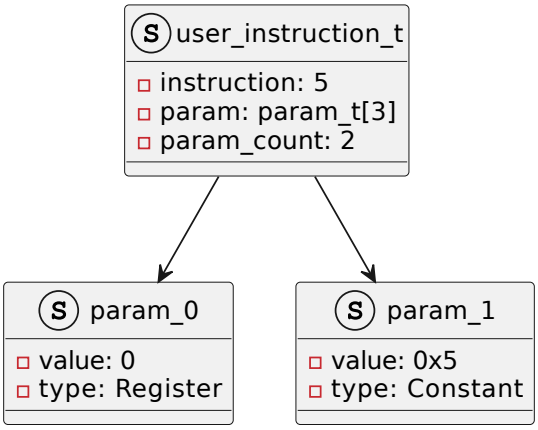

Figure 5: Example of Storing Instruction

The figure is for your understanding. Some details were omitted for better explanation. Also, instruction 5 means that this was an `lw` instruction. The table of each instructions are as it follows:

Entry	Enum Value
Add	0
Sub	1
Mul	2
Div	3
Mov	4
Lw	5
Sw	6
Rst	7
Jmp	8
Beq	9
Bne	10
Slt	11

Figure 6: Instruction ID Table

Then each instructions will be stored in memory as an array for simulation in the future. Once the assembler finishes converting a user input text into a `struct user_instruction_t` type data, it is now ready to be executed.

4.3.5 Simulator

Since syntax checking and assembling were finished, it is now time to start simulating each instructions. In each execution of instructions, there are 4 requirements that shall be satisfied:

1. **Manage PC**: Since the simulator needs to direct the next instruction's address, in our case which is index, the simulator needs to manage PC before executing the instruction.
2. **Execute**: The simulator must execute user's given instruction. Each instruction will be actually executed by functions accessed by function pointers.
3. **Output**: One of the requirements of this program was to show internal state of registers when being used. Also if we had some `STDOUT` related jobs, the program must print it out in the screen as well.
4. **Dump**: The program stores each execution results into a file. Therefore, it shall dump data into the file when executing a single instruction.

Simulator simulates the user instruction with following simple algorithm.

Algorithm 1 Simulator Algorithm

```
procedure EXECUTE(ctx)
    while ctx.pc ≠ ctx.lastInstruction do
        ctx.pc ←ctx.pc + 1                          ▷ Increment PC
        Execute instruction
        if Error Found then
            Quit
        end if
        if  thenctx.pc > ctx.lastInstruction        ▷ When PC was jumped too far
            Quit
        end if
        Print register stats and dump info file
    end while
```

Each user instructions are executed by function pointers that are loaded in the memory when the program starts. For example with Code Snippet 4, which performs `lw` instruction, the simulator will first set PC to the next instruction. After that, it will call callback function for actually executing the instruction. The `lw` instruction uses `sim_lw` function to execute instruction. Therefore, following code will be executed using function pointer:

```
int sim_lw(struct context_t *ctx, struct user_instruction_t instruction) {
    uint32_t constant = instruction.param[1].value;
    uint32_t *dst = &ctx->registers[instruction.param[0].value];
    *dst =  constant;

    mark_registers(ctx, instruction);
    return 0;
}
```

Code Snippet 6: Callback Function for LW

After simulator executes `sim_lw`, it will check for errors and verify if the execution context is correct. For example some operations like `jmp`, `bne` and `beq` does not check if the address that it is jumping to is valid or not. Therefore, in verification step, the simulator will check if the simulation is being performed well or not. Also the simulator will print data out to the user screen and store output to file as well. The simulator will continue execution until it hits the last instruction. Once it hits the end of the instructions, it will quit execution gracefully.

With simple algorithm, simulator can execute user instructions until it meets the last instruction in designated file.

4.3.6 Summary

In short, `mipsim` has three major parts: syntax checker, assembler and simulator. Syntax checker checks and verifies user's input with predefined syntax. At the same time, assembler converts each strings of user's lines into `struct user_instruction_t` which will be used later. Once syntax checker and assembler finishes their job, the simulator starts executing stored instructions one by one. If you would like to see the actions in code, please check `simulator.c`, `syntax.c` and `io.c` for more information. All functions and definitions have rich documentation, therefore it will be easy to understand what is going on with the code.

5 User Guide Environments

5.1 Environment

The environment that this program was checked running is as it follows:

- **OS**: Ubuntu 22.04.01 LTS
- **CPU**: Intel(R) Core(TM) i9-7940X CPU @ 3.10GHz
- **RAM**: 64GB
- **Compiler**: GCC 11.3.0, Make 4.3
- **C**: C99

Please be aware that with some environments, the program might not be able to execute. Also, with the code was zipped with version `Zip 3.0`.

5.2 User Guide

5.2.1 Unzipping

Once you have downloaded the attached `.zip` file, you first need to unzip the file using following command.

```
$ unzip FILE_NAME.zip
```

Code Snippet 7: Unzip Command Example

Please change the `FILE_NAME.zip` to the file that I have attached accordingly.

5.2.2 Building

Once you have unzipped the attatched file, there will be a directory named `mipsim`. Navigate to the directory `mipsim` by using `cd`. `mipsim` supports `make` with following recipes:

- `debug`: For building debug version executable program with `-DDEBUG` option.
- `all`: For building executable program.
- `clean`: For cleaning up all `.o` files and generated executable program.

For example, if you were to build yourself a version for end-user case, use following command:

```
$ make
gcc -O2 -Wall -std=gnu99 -o context.o -c context.c
gcc -O2 -Wall -std=gnu99 -o io.o -c io.c
gcc -O2 -Wall -std=gnu99 -o simulator.o -c simulator.c
gcc -O2 -Wall -std=gnu99 -o syntax.o -c syntax.c
gcc -O2 -Wall -std=gnu99 -o main.o -c main.c
gcc -o mipsim context.o io.o simulator.o syntax.o main.o
```

Code Snippet 8: Example Build Output

This will generate object files ending with `.o` as well as an executable file named `mipsim`. To clean up your workspace, use `clean` recipe for removing all generated files.

5.2.3 User Guide

`mipsim` offers simple command line interfaces. You can use following command to get a glimpse of each options:

```
$ ./mipsim  --help
...
                    A Simple MIPS Calculator
                            32190984 Isu Kim
Usage: mipsim [options]
Options
  -h, --help            Print this help message and exit
  -i, --input           Specify an input file to execute
  -o, --output          Specify an output file to dump log into
```

Code Snippet 9: Example Help Output

The options are:

- `help`: For printing out the help message.
- `input`: For specifying an input file. This is required parameter.
- `output`: For specifying the output file. This will write execution results to the specified file. If not specified, the file will default to `out.txt`.

For example, an example execution command will be:

```
$ ./mipsim  -i input.txt -o output.txt
...
                    A Simple MIPS Calculator
                            32190984 Isu Kim
[INFO] Input File: input.txt
[INFO] Output File: output.txt
```

Code Snippet 10: Example Command Line Interface Output

This will execute an file named `input.txt` and store results to `output.txt`.

As defined in `common.h`, the maximum length of input is 1024 characters. File names that are longer than 1024 characters will have unexpected behaviors.

5.2.4 Available Instructions

Name	Syntax	Token Count	Available	Callback Function
add	Register, Register, Register	3	O	`sim_add`
sub	Register, Register, Register	3	O	`sim_sub`
mul	Register, Register, Register	3	O	`sim_mul`
div	Register, Register, Register	3	O	`sim_div`
mov	Register, Both	2	O	`sim_mov`
lw	Register, Constant	2	O	`sim_lw`
sw	Register	1	O	`sim_sw`
rst		0	O	`sim_rst`
jmp	Constant	1	O	`sim_jmp`
beq	Both, Both, Constant	3	O	`sim_beq`
bne	Both, Both, Constant	3	O	`sim_bne`
slt	Register, Both, Both	3	O	`sim_slt`

Figure 7: Table of Available Instructions

Each implementation of callback functions can be found under `simulator.c`. Since mentioning all the functions one by one seems redundant, we are not going to discuss about each functions. Each functions have quite rich documentation, therefore it will be better to check the source code.

5.2.5 Demonstrations

In order to verify if `mipsim` works properly or not, the list below is the codes that were tested working on `mipsim`. Also codes are included in the attachments, so you can try them yourself.

Figure 8: Example Execution Screenshot

The screenshot is an example executing `mipsim`. Since this picture was difficult for me to recognize the results, I will be copying the execution results into this document instead of attaching some screenshots.

All the codes the examples mentioned above can be found under the directory `inputs`. You can test the following codes out yourself to check `mipsim`:

- `input/1.txt`: Basic execution from page 5 of `HW1.pdf`.
- `input/2.txt`: Basic execution from page 6 of `HW1.pdf`.
- `input/3.txt`: Basic execution from far left side code page 7 of `HW1.pdf`.
- `input/4.txt`: Basic execution from far right side code page 7 of `HW1.pdf`.
- `input/e1.txt`: Error execution with 3 syntax errors.
- `input/e2.txt`: Error execution with wrong addressing.
- `input/l1.txt`: Loop execution for 10 times. Prints out 4, 8, 12, ... 40.

```
LW r0 0xF # The second operand is a number
LW r1 0x4
ADD r2 r0 r1
MOV r0 r2
SW r0 STDOUT # STDOUT is a signal to print out $r0
RST # Reset all registers
LW r0 0x2
LW r1 0x4
MUL r2 r0 r1
MOV r0 r2
SW r0 STDOUT
```

Code Snippet 11: Example Code `input/1.txt`

Also for demonstration of dump file, we are going to store result data into `./outputs/1.txt`. Therefore, an example execution will be as it follows:

```
$ ./mipsim  -i inputs/1.txt -o output/1.txt
...
[INFO] Input File: inputs/1.txt
[INFO] Output File: output/1.txt
[SUCCESS] inputs/1.txt has no syntax errors.
[INFO] Starting simulator
[PC 1] Instruction : lw  r0  0xf
        Register Status: r0: 0xf,
[PC 2] Instruction : lw  r1  0x4
        Register Status: r0: 0xf, r1: 0x4,
[PC 3] Instruction : add  r2  r0  r1
        Register Status: r0: 0xf, r1: 0x4, r2: 0x13,
[PC 4] Instruction : mov  r0  r2
        Register Status: r0: 0x13, r1: 0x4, r2: 0x13,
```

```
[PC 5] Instruction : sw  r0
        >> STDOUT: 19
        Register Status: r0: 0x13, r1: 0x4, r2: 0x13,
[PC 6] Instruction : rst
        Register Status:
[PC 7] Instruction : lw  r0  0x2
        Register Status: r0: 0x2,
[PC 8] Instruction : lw  r1  0x4
        Register Status: r0: 0x2, r1: 0x4,
[PC 9] Instruction : mul  r2  r0  r1
        Register Status: r0: 0x2, r1: 0x4, r2: 0x8,
[PC 10] Instruction : mov  r0  r2
        Register Status: r0: 0x8, r1: 0x4, r2: 0x8,
[PC 11] Instruction : sw  r0
        >> STDOUT: 8
        Register Status: r0: 0x8, r1: 0x4, r2: 0x8,
[INFO] mipsim terminated successfully!
```

Code Snippet 12: Example Execution Output for `input/1.txt`

Since the code did not have any errors, `mipsim` executes the code without any issues. In `output/1.txt`, following dump is stored:

```
$ cat output/1.txt
Instruction : lw  r0  0xf
        Registers: r0: 0xf,
Instruction : lw  r1  0x4
        Registers: r0: 0xf, r1: 0x4,
Instruction : add  r2  r0  r1
        Registers: r0: 0xf, r1: 0x4, r2: 0x13,
Instruction : mov  r0  r2
        Registers: r0: 0x13, r1: 0x4, r2: 0x13,
Instruction : sw  r0
        Registers: r0: 0x13, r1: 0x4, r2: 0x13,
Instruction : rst
        Registers:
Instruction : lw  r0  0x2
        Registers: r0: 0x2,
Instruction : lw  r1  0x4
        Registers: r0: 0x2, r1: 0x4,
Instruction : mul  r2  r0  r1
        Registers: r0: 0x2, r1: 0x4, r2: 0x8,
Instruction : mov  r0  r2
        Registers: r0: 0x8, r1: 0x4, r2: 0x8,
Instruction : sw  r0
        Registers: r0: 0x8, r1: 0x4, r2: 0x8,
```

Code Snippet 13: Example Execution Dump for `input/1.txt`

Following output is for demonstrating the `input/2.txt`'s execution.

```
$ ./mipsim  -i inputs/2.txt -o output/2.txt
...
[INFO] Input File: inputs/2.txt
[INFO] Output File: output/2.txt
[SUCCESS] inputs/2.txt has no syntax errors.
[INFO] Starting simulator
[PC 1] Instruction : lw  r0  0x5
        Register Status: r0: 0x5,
[PC 2] Instruction : lw  r1  0xf
        Register Status: r0: 0x5, r1: 0xf,
[PC 3] Instruction : add  r2  r0  r1
        Register Status: r0: 0x5, r1: 0xf, r2: 0x14,
[PC 4] Instruction : jmp  0x9
        Register Status: r0: 0x5, r1: 0xf, r2: 0x14,
[PC 9] Instruction : add  r3  r1  r2
        Register Status: r0: 0x5, r1: 0xf, r2: 0x14, r3: 0x23,
[PC 10] Instruction : mov  r0  r3
        Register Status: r0: 0x23, r1: 0xf, r2: 0x14, r3: 0x23,
[PC 11] Instruction : sw  r0
        >> STDOUT: 35
        Register Status: r0: 0x23, r1: 0xf, r2: 0x14, r3: 0x23,
[INFO] mipsim terminated successfully!
```

Code Snippet 14: Example Execution Output for `input/2.txt`

Following output is for demonstrating the `input/3.txt`'s execution.

```
$ ./mipsim  -i inputs/3.txt -o output/3.txt
...
[INFO] Input File: inputs/3.txt
[INFO] Output File: output/3.txt
[SUCCESS] inputs/3.txt has no syntax errors.
[INFO] Starting simulator
[PC 1] Instruction : lw  r0  0x5
        Register Status: r0: 0x5,
[PC 2] Instruction : lw  r1  0xa
        Register Status: r0: 0x5, r1: 0xa,
[PC 3] Instruction : bne  r0  r1  0x6
        Register Status: r0: 0x5, r1: 0xa,
[PC 6] Instruction : mov  r0  0x1
        Register Status: r0: 0xa, r1: 0xa,
[PC 7] Instruction : sw  r0
        >> STDOUT: 10
        Register Status: r0: 0xa, r1: 0xa,
```

```
[INFO] mipsim terminated successfully!
```

Code Snippet 15: Example Execution Output for `input/3.txt`

Following output is for demonstrating the `input/4.txt`'s execution.

```
$ ./mipsim  -i inputs/4.txt -o output/4.txt
...
[INFO] Input File: inputs/4.txt
[INFO] Output File: output/4.txt
[SUCCESS] inputs/4.txt has no syntax errors.
[INFO] Starting simulator
[PC 1] Instruction : lw  r0  0x5
        Register Status: r0: 0x5,
[PC 2] Instruction : lw  r1  0xa
        Register Status: r0: 0x5, r1: 0xa,
[PC 3] Instruction : slt  r2  r0  r1
        Register Status: r0: 0x5, r1: 0xa, r2: 0x1,
[PC 4] Instruction : bne  r0  0x0  0x7
        Register Status: r0: 0x5, r1: 0xa, r2: 0x1,
[PC 7] Instruction : mov  r0  0x1
        Register Status: r0: 0xa, r1: 0xa, r2: 0x1,
[PC 8] Instruction : sw  r0
        >> STDOUT: 10
        Register Status: r0: 0xa, r1: 0xa, r2: 0x1,
[INFO] mipsim terminated successfully!
```

Code Snippet 16: Example Execution Output for `input/4.txt`

As you can see, the executions were performed without any issues. Now, let's test some cases with errors. Let's use a code that has three errors.

```
LW r0 r2 # Second needs to be constant
LW r1 0x4
ADD r2 r0 0x3 # Third needs to be register
MOV r0 r2
SW r0 STDOUT
RST
LW r0 0x2
LW r1 0x4
MUL 0x4 r0 r1 # First needs to be register
MOV r0 r2
SW r0 STDOUT
```

Code Snippet 17: Example Erroneous Code `input/e1.txt`

The code has three syntax errors. When we execute the code with `mipsim`, the result is as it

follows:

```
$ ./mipsim -i ./inputs/e1.txt
...
[INFO] Output file not specified, defaults to out.txt
[INFO] Input File: ./inputs/e1.txt
[INFO] Output File: out.txt
 - Syntax Error @ ln1
        invalid argument type: r2
        expected constant but got register
 - Syntax Error @ ln3
        invalid argument type: 0x3
        expected register but got constant
 - Syntax Error @ ln9
        invalid argument type: 0x4
        expected register but got constant
[ERROR] Found syntax error, exiting mipsim.
```

Code Snippet 18: Example Erroneous Code Execution - `input/e1.txt`

`mipsim` catches three errors with reason. Since `mipsim` found errors in syntax checking and assembling phase, it will not start simulator.

Let's take a look into another case with a code that is correct in syntax however addresses unavailable address.

```
LW r0 0x5
LW r1 0xF
ADD r2 r0 r1
JMP 0xF  # Line 15 does not exist
RST
LW r0 0x2
LW r1 0x4
MUL r2 r0 r0
ADD r3 r1 r2
MOV r0 r3
SW r0 STDOUT
```

Code Snippet 19: Example Erroneous Code `input/e2.txt`

Since this was a code that was correct in syntax, syntax checker and assembler will consider this code valid. Therefore, the simulator will start operating. However, since the addressing was invalid, simulator will quit execution as soon as it hits the `jmp` instruction. An example output is like as it follows:

```
$ ./mipsim -i ./cmake-build-debug/inputs/e2.txt
...
```

```
[INFO] Output file not specified, defaults to out.txt
[INFO] Input File: ./cmake-build-debug/inputs/e2.txt
[INFO] Output File: out.txt
[SUCCESS] ./cmake-build-debug/inputs/e2.txt has no syntax errors.
[INFO] Starting simulator
[PC 1] Instruction : lw  r0  0x5
        Register Status: r0: 0x5,
[PC 2] Instruction : lw  r1  0xf
        Register Status: r0: 0x5, r1: 0xf,
[PC 3] Instruction : add  r2  r0  r1
        Register Status: r0: 0x5, r1: 0xf, r2: 0x14,
[PC 4] Instruction : jmp  0xf
Invalid addressing while executing instruction @ ln15
```

Code Snippet 20: Example Erroneous Code Execution - `input/e2.txt`

`mipsim` catches that the code instruction was using wrong address, therefore terminating program. Let's take a look at an example with loop using codes:

```
LW r1 0x4  # Actually a constant 4
LW r2 0x1  # Actually a constant 1
LW r3 0x0  # i = 0
ADD r3 r3 r2  # i++
MUL r4 r1 r3  # a = i * 4
SW r4 STDOUT  # print a, shall print out 4, 8 ~ 40
SLT r0 r3 0xa  # i < 10
BEQ r0 0x1 0x4  # loop
```

Code Snippet 21: Example Loop Code `input/l1.txt`

When we execute the program, `mipsim` prints out numbers 4, 8, 12, ... 40 correctly like below:

```
$ ./mipsim -i ./cmake-build-debug/inputs/l1.txt
...
[INFO] Output file not specified, defaults to out.txt
[INFO] Input File: ./cmake-build-debug/inputs/l1.txt
[INFO] Output File: out.txt
[SUCCESS] ./cmake-build-debug/inputs/l1.txt has no syntax errors.
[INFO] Starting simulator
[PC 1] Instruction : lw  r1  0x4
        Register Status: r1: 0x4,
[PC 2] Instruction : lw  r2  0x1
        Register Status: r1: 0x4, r2: 0x1,
[PC 3] Instruction : lw  r3  0x0
        Register Status: r1: 0x4, r2: 0x1, r3: 0x0,
[PC 4] Instruction : add  r3  r3  r2
        Register Status: r1: 0x4, r2: 0x1, r3: 0x1,
```

```
[PC 5] Instruction : mul  r4  r1  r3
        Register Status: r1: 0x4, r2: 0x1, r3: 0x1, r4: 0x4,
[PC 6] Instruction : sw  r4
        >> STDOUT: 4
...
        >> STDOUT: 8
...
         >> STDOUT: 40
Register Status: r0: 0x1, r1: 0x4, r2: 0x1, r3: 0xa, r4: 0x28,
[PC 7] Instruction : slt  r0  r3  0xa
        Register Status: r0: 0x0, r1: 0x4, r2: 0x1, r3: 0xa, r4: 0x28,
[PC 8] Instruction : beq  r0  0x1  0x4
        Register Status: r0: 0x0, r1: 0x4, r2: 0x1, r3: 0xa, r4: 0x28,
[INFO] mipsim terminated successfully!
```

Code Snippet 22: Example Loop Code Execution - `input/l1.txt`

As the result shows, `mipsim` executes loop for 10 times and terminates successfully.

`mipsim` does not have ability to detect infinite loops. Therefore, be extra cautious when writing loop codes.

5.3 Limitations and Extra Features

`mipsim` was designed to implement as much as the original `HW1.pdf` requested. However, in some operations, the requirements were slightly changed due to limitations and for better experience. Please check following list for changes in instructions.

- `sw`: The instruction will actually ignore token `STDOUT`. To reduce complexity in the code, the instruction was designed to print out the reigster value that was given as first argument. Therefore, the syntax checker will make some expressions without `STDOUT` as valid.
- `sll`: The instruction originally was required to set `r0` as the result of comparing two registers or constants. However, for extra features, `mipsim` supports other registers to be used as well. Meaning that `sll r2 10 20` will store `r2` a value of 1.

Besides instructions, there are limitations of the overall program:

- **Maximum Input**: The maximum string length that `mipsim` can handle is 1024 bytes. Meaning that if you have a single line that exceeds 1024 characters, `mipsim` will drop the remaining part.
- **Maximum Instructions**: In order to minimize memory overhead, `mipsim` offers maximum 1024 user instructions. Meaning that if you instructions more than 1024 instructions, `mipsim` will not be able to execute instructions correctly.
- **Register Size**: Each registers are stored as a `uint32_t` variable. Meaning that each registers are 32 bit. `mipsim` will not trap nor notify the user about register values overflowing.

- **Constant Size**: When you use a constant such as `0xFF`, the maximum value that `mipsim` can store is 32 bits.
- **Character Set**: `mipsim` only supports `ASCII` characters, saving input fie using other character set such as `UNICODE` will have unexpected behaviors.

6 Conclusion

Implementing `mipsim` was quite a fun project. Since we all learn the concept of "stored program" and "Turing machine", it was quite interesting to make a program that mimics the characteristics of those concepts. Also, when implementing `mipsim`, the following list represents the encountered difficulties:

- **Parsing String**: C has some powerful library named `string.h` for dealing with strings. However, compared to other languages which supports easy use of string related functions, using strings in `C` is quite difficult. For example, using `strtok` compared to `.split()` method from Python is quite challenging.
- **Designing**: Designing `mipsim` to become extensible was quite challenging process. There were lots of considerations when implementing `mipsim` since the program itself was quite simple. However, keeping design principles throughout the project was quite difficult process.
- **Environment**: Professor Nam told us that if he could not create an environment himself by this manual, he will not grade the project. In order for this project to be graded, I considered using from `Docker` to using `Vagrant`. However, since the code is quite simple and does not require external libraries, it was preferred to use basic `C` and `make` commands.

While implementing this project, there were some features that I thought that might come in handy when adding more instructions:

- **Memory**: Since current `lw` and `sw` does not store extra data in memory, it was considered to be a good idea to construct a virtual memory.
- **Branches**: Since current code uses direct line count for `jmp`, `beq` and `bne`, it would have been better if we add a branch feature. Meaning that instead of using `jmp 0x10`, using `jmp label` will jump us to the corresponding line as well.

Also, while designing and implementing `mipsim`, there were some things that I was able to recall and review:

- `C`: Since the last time that I wrote a genuine `C` code was last semester, yes there were some codes like `eBPF` which doesn't count, it was nice to recall some techniques and important concepts in `C`. Such includes: function pointers and string parsing.
- **Designing**: Designing a program and defining requirements before implementing a program was a good practice in this project. Avoiding to code before designing the program saved lots of time.

I pretty much enjoyed implementing `mipsim` and would like to add those features mentioned up above in the future.