The Chinese University of Hong Kong, Shenzhen

CSC3050

Computer Architecture

Assignment 2 Report

Author: Yang Yuzhe
Student ID: 121090684

October 31, 2023

Contents

1 Environment

`cat /etc/*-release` output:

```
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=22.04
DISTRIB_CODENAME=jammy
DISTRIB_DESCRIPTION="Ubuntu 22.04.3 LTS"
PRETTY_NAME="Ubuntu 22.04.3 LTS"
NAME="Ubuntu"
VERSION_ID="22.04"
VERSION="22.04.3 LTS (Jammy Jellyfish)"
VERSION_CODENAME=jammy
ID=ubuntu
ID_LIKE=debian
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies
    /privacy-policy"
UBUNTU_CODENAME=jammy
```

`python3 --version` output:

```
Python 3.10.12
```

2 Overview

The MIPS Simulator is a program designed to emulate the behavior of a MIPS processor. It is capable of executing MIPS assembly instructions and provides a platform for debugging and analyzing MIPS programs. The simulator is implemented in Python and supports a subset of MIPS instructions. The main part of the MIPS Simulator is implemented in `simulator.py` with a function libaray named `lib.py`.

HOW TO RUN:

Navigate to the source code directory and run the following command:

```
python3 simulator.py test.asm test.txt test_checkpoints.txt test.in
    test.out
```

Or you can run the test shell file in the `test_bash` directory:

```
bash test.sh
```

3 Design

3.1 Architecture

The `simulator.py` follows a simple architecture comprising a MIPS-like memory, registers, and an instruction execution unit. It utilizes a bytearray to represent the memory and an array for registers. The program counter keeps track of the current instruction being executed.

When the simulator start running, it will first initialize the memory, register and the output file information by analyzing all the static data. Then it will start to execute the instructions one by one and it will update the memory and register according to the corresponding instructions. When the simulator reaches the end of the program, it will print the final memory and register information to the output file. In addition, there has a `flag` array in order to record the `return value` and `exit` information. The simulator will stop if it receive the `exit` signal and generate the output file finally.

3.2 Memory & Registers

The memory is represented as a bytearray in Python, providing a contiguous block of addressable memory.

```
mem = bytearray(MEMORY_SIZE)
reg = [0] * (32 + 3)
```

This bytearray is divided into two segments: the text segment and the data segment. The text segment stores machine code instructions, while the data segment holds static data.

The register file is implemented as an array, simulating the registers of a MIPS processor. Each register is represented by an element in the array. The array allows for easy access and manipulation of register values during instruction execution.

These two array will output as a `.bin` file when the program encounters to the checkpoints.

3.3 Instruction Execution

Instructions are executed through a set of functions corresponding to each instruction type, which stores in the `lib.py`. The simulator uses a switch-case-like structure to determine the operation to be performed based on the opcode. Control flow instructions, arithmetic operations, and memory access instructions are implemented.

3.4 Static Data

`data_handler`

This function processes an assembly file containing a data segment `(.data)`. It opens the specified file, reads it line by line, and parses the data based on the specified data types `(.asciiz, .ascii, .word, .byte, .half)`. The parsed data is stored in the global variable `mem` and updates the global variable `STATIC_DATA`, which tracks the position of data in memory.

`text_seg`

This function handles an assembly file containing a text segment `.text`. It opens the given file, reads it line by line, and stores the machine code of each instruction (in binary representation) in specific locations in the global variable `mem`. Simultaneously, it updates the global variable `machine_code_size` to track the size of the machine code.

4 Implementation

4.1 Static Data

`data_handler`

This function processes an assembly file containing a data segment `(.data)`. It opens the specified file, reads it line by line, and parses the data based on the specified data types `(.asciiz, .ascii, .word, .byte, .half)`. The parsed data is stored in the global variable `mem` and updates the global variable `STATIC_DATA`, which tracks the position of data in memory.

`text_seg`

This function handles an assembly file containing a text segment `.text`. It opens the given file, reads it line by line, and stores the machine code of each instruction (in binary representation) in specific locations in the global variable `mem`. Simultaneously, it updates the global variable `machine_code_size` to track the size of the machine code.

4.2 Instruction Execution Functions

Each MIPS instruction has a corresponding function implemented in Python. These functions emulate the behavior of the MIPS instructions, updating registers and memory accordingly. Debugging statements have been included to print the details of each instruction execution.

4.3 System Calls

The simulator comprehensively supports MIPS system calls, enabling interaction with the simulated environment. The implemented system calls cover a range of functionalities, such as reading and writing integers, strings, and characters. Additionally, the simulator supports file operations, including opening and closing files, as well as program termination.

File Operations

For the file opening syscall `_open`, the filename is obtained from register `_a0`, and the Python `os.open` function is employed to acquire a file descriptor, which is stored in the register `_a0`. To read from a file `_read`, the file descriptor, buffer address, and length are fetched from specific registers. Subsequently, the program uses `os.read` to retrieve data from the file and stores it in the designated memory buffer. For writing to a file `_write`, the file descriptor, buffer address, and length are similarly extracted from registers. The program then reads data from memory and utilizes `os.write` to write it to the file. Closing a file `_close` involves retrieving the file descriptor and using `os.close` to close the file. In summary, these operations make use of Python's file handling functions, facilitating interaction with the simulated environment through system calls.

Print

For printing integers: `_print_int`, the value is obtained from register `_a0` and directly printed. String printing: `_print_string` extracts the address of the string from register `_a0`, and the characters are retrieved from memory and printed until a null terminator is encountered. Character printing: `_print_char` obtains the character from register `_a0` and prints it.

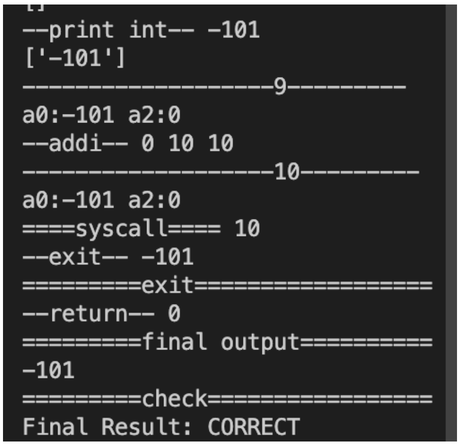
4.4 Checkpoints

During the process of executing the instructions, the simulator will check the the current line to the checkpoints. If the current line is the checkpoint, the simulator will output the memory and register information to the `.bin` file.
In order to debugging the program and make the registers information more readable, I write `reg.py` to convert the `.bin` file to the a `.txt` file, which contains all the registers' value and it makes me easier to check the current registers' value with the correct value.
Here is one of the test shell files I wrote to test the program:

```
cd "/home/parallels/toby_dev/csc3050_toby/CSC3050_P2"
clear
python3 simulator.py Example_test_cases/many/many.asm Example_test_cases/many/many.txt Example_test_cases/many/many_checkpts.txt Example_test_cases/many/many.in many.out

values=(0 23 97)

for a in "${values[@]}"
do
    cmp register_$a.bin Example_test_cases/many/correct_dump/register_$a.bin
    if [ $? -eq 1 ]
    then
        hexdump register_$a.bin > register_$a.txt
        hexdump Example_test_cases/many/correct_dump/register_$a.bin > correct_register_$a.txt
        python3 reg.py Example_test_cases/many/correct_dump/register_$a.bin register_$a.bin $a
        cmp true_reg_$a.txt my_$a.txt
    fi

    cmp memory_$a.bin Example_test_cases/many/correct_dump/memory_$a.bin
    if [ $? -eq 1 ]
    then
        hexdump memory_$a.bin > memory_$a.txt
        hexdump Example_test_cases/many/correct_dump/memory_$a.bin > correct_memory_$a.txt
    fi
done

cmp many.out Example_test_cases/many/many_correct.out
if [ $? -eq 1 ]
then
    echo "Final Result: FAIL"
else
```

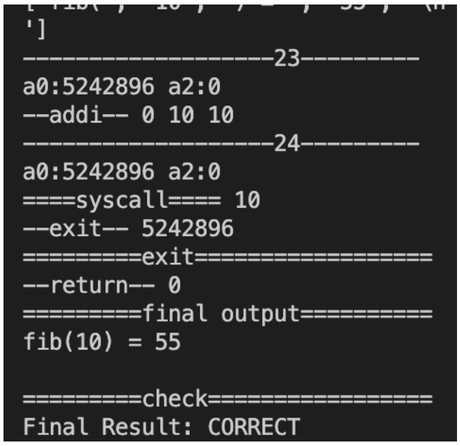
```
    echo "Final Result: CORRECT"
fi
```

5 Program Output



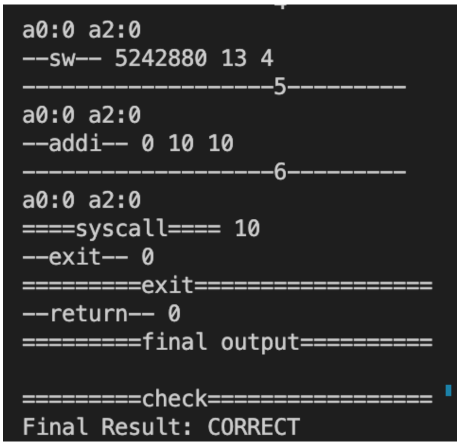
```
--print int-- -101
['-101']
--------------------9----------
a0:-101 a2:0
--addi-- 0 10 10
--------------------10----------
a0:-101 a2:0
====syscall==== 10
--exit-- -101
=========exit==================
--return-- 0
=========final output==========
-101
=========check==================
Final Result: CORRECT
```

(a) a-plus-b



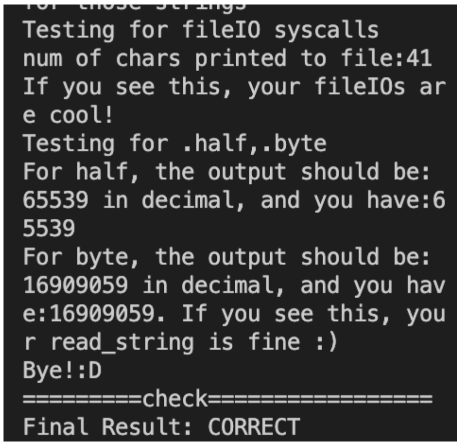
```
']
--------------------23----------
a0:5242896 a2:0
--addi-- 0 10 10
--------------------24----------
a0:5242896 a2:0
====syscall==== 10
--exit-- 5242896
=========exit==================
--return-- 0
=========final output==========
fib(10) = 55

=========check==================
Final Result: CORRECT
```

(b) fib



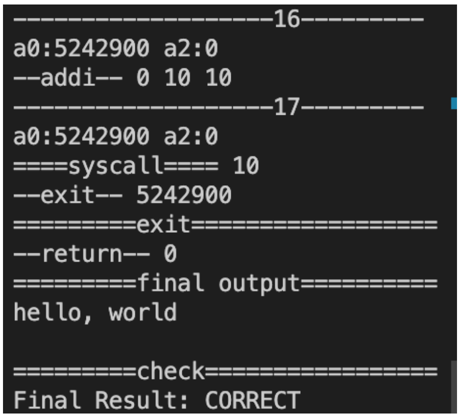
```
a0:0 a2:0
--sw-- 5242880 13 4
--------------------5----------
a0:0 a2:0
--addi-- 0 10 10
--------------------6----------
a0:0 a2:0
====syscall==== 10
--exit-- 0
=========exit==================
--return-- 0
=========final output==========

=========check==================
Final Result: CORRECT
```

(c) lw_sw



```
Testing for fileIO syscalls
num of chars printed to file:41
If you see this, your fileIOs ar
e cool!
Testing for .half,.byte
For half, the output should be:
65539 in decimal, and you have:6
5539
For byte, the output should be:
16909059 in decimal, and you hav
e:16909059. If you see this, you
r read_string is fine :)
Bye!:D
=========check==================
Final Result: CORRECT
```

(d) many



```
--------------------16----------
a0:5242900 a2:0
--addi-- 0 10 10
--------------------17----------
a0:5242900 a2:0
====syscall==== 10
--exit-- 5242900
=========exit==================
--return-- 0
=========final output==========
hello, world

=========check==================
Final Result: CORRECT
```

(e) memcpy

Figure 1: Program Output