INDIAN INSTITUTE OF TECHNOLOGY DELHI

COL216

Report

Assignment – 3

MIPS Simulator

Abstract

A simulator is a software that emulates the actions of an entity without actually using the entity. Here we attempt to create a cross platform MIPS simulator that emulates all the hardware instructions supported by MIPS. This simulator takes as input a MIPS assembly program that translates it into instructions executed by MIPS.

Harsh Agrawal 2019CS10431

Saptarshi Dasgupta 2019CS50447

Contents

1 Approach

We implemented a simulator in `C++` for executing MIPS instructions from input assembly code. We divided our program into two modules a `compiler` and a `hardware` module. These modules perform different functions as outlined below

1.1 Compiler

- This module takes as input the assembly program and parses the tokens according to the syntax specifications of MIPS assembly programs.
- This module assembles the program into a sequence of instructions that can be executed according to the hardware specifications.
- It also generates appropriate syntax errors if an erreneous expression or an unidentified token is encountered.
- The compiler checks the encodibility of instructions into 4 bytes (without actually encoding it) using the size of operands involved.

1.2 Hardware

- This module emulates a subset of instructions provided by MIPS. They are `add, mul, sub, slt, addi, bne, beq, j, lw, sw`.
- The module maintains a record of all the register and memory values and modifies them according to the instruction specifications.
- It generates appropriate exceptions on performing prohibited actions like out of bounds memory access and reserved register access.

2 Testing

We employed both automated and manual modes of testing. The automated mode helped establish the correctness of the arithmetic operations like `add, sub, mul, slt, addi` while the manual mode helped us evaluate the correctness for `lw, sw, bne, beq, j` instructions.

2.1 Automated

We wrote a `python3` script that generates random MIPS assembly statements for the arithmetic instructions `add, mul, sub, slt, addi`. The script executes these instructions using our program and stores the register values after each step. The same program is then fed into the `spim` simulator and the register values are queried and stored. The two sets of register values are checked against each other for each `ASM` statement in the test case. All communication with the executed process occurs through unix pipes and a `PTY` interface. To install the dependencies, enter the following commands.

```
$ sudo apt install spim
$ python3 -m pip install -r requirements.txt
$ python3 tester.py -n 20 -m 30
```

The last command will run 20 test cases with 30 instructions in each test case and print a test log with the result of each test case. The generated test cases are saved in a the `./output` directory by default but can be changed by supplying the `-o, --output` command line flag to the python script

Note: The test cases generated by this method is saved in the `./tests/automated` directory for reference

2.2 Manual

We have also included some manual `ASM` files in the `./tests/manual` directory. These test cases are meant to test the correctness of `lw, sw, beq, bne, j` instructions. These tests are present in the `./tests/manual/` folder.

- `arithmetic`: This test case checks the correctness of `add, mul, sub`. We have also handled overflows and raised an arithmetic overflow exception in such a case.
- `conditional`: `if-then-else` implemented using `bne, beq ,j`. These cases test the branching conditions in an assemblt program and ensure that the execution jumps to a logically correct statement.
- `looping`: Implementing loops for counter and fibonacci numbers. We have implemented loops using jump statements and matched the output

with the expected output. The fibonacci numbers and a simple counter are the most intuitive examples.

- `stacks`: We implemented stacks using the `lw` and `sw` instructions. The push and pop methods were implemented and the expected values were checked.

- `errors`: Extensive testing for errors with invalid register, labels, integers, out of bounds memory addressing, non-word aligned adresses and accessing kernel reserved registers

2.3 Results

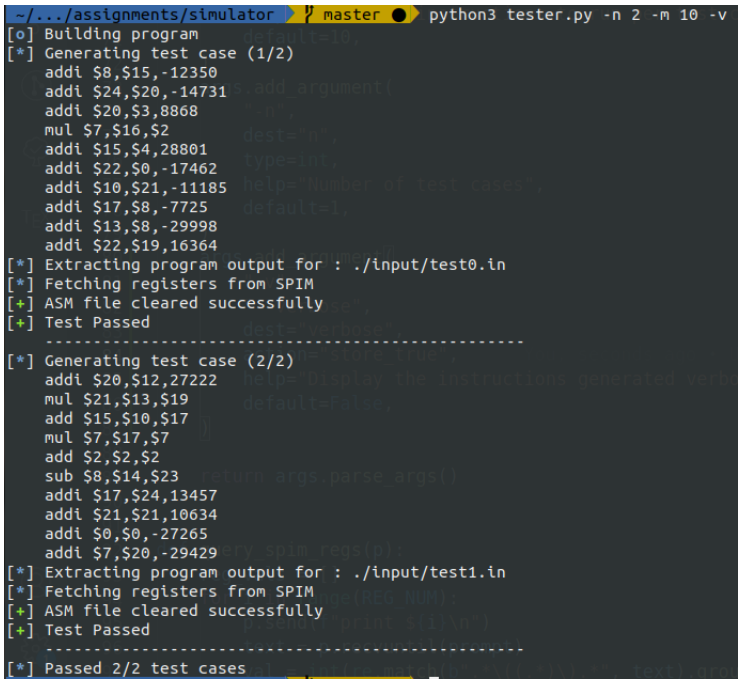
We evaluated the results of these tests in both the automated process and the manual process. Based on our testing we conclude the following:

2.3.1 Automated

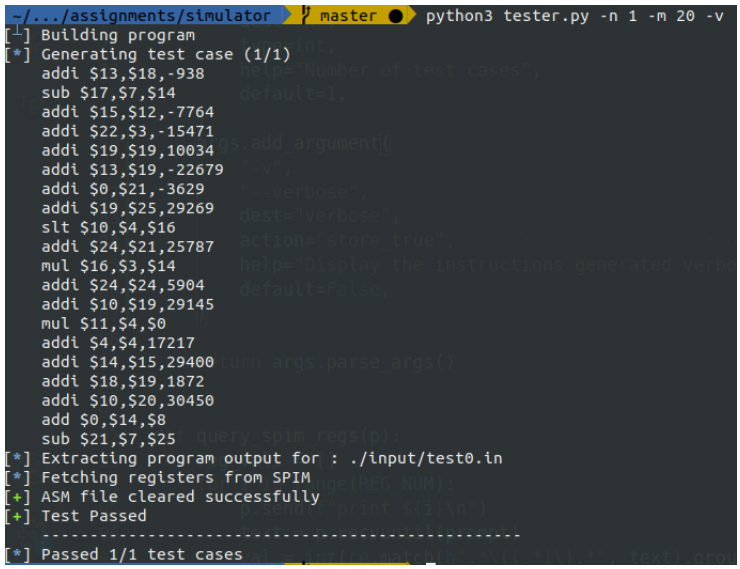
The register contents of `spim` and our simulator were compared and all the test cases passed as expected. The snapshots of the results are attached below.

Since our approach allows us to test for arbitrarily large test cases we were able to extensively test our implementation. The results showed that our implementation is robust as it passed all sorts of test cases.

We could not match our output with `spim` output in cases where integer overflow occured. This is because spim automatically handles arithmetic overflows whereas our program raises an arithmetic overflow exception. Since our simulator aims to emulate the hardware as closely as possible we decided to not handle the exception by default.



(a) 2 test cases each with 10 instructions

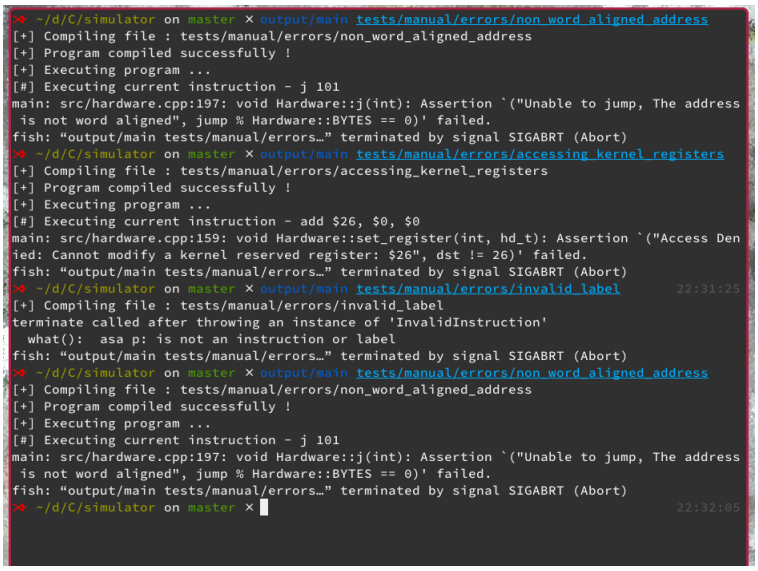


(b) 1 test case with 20 instructions

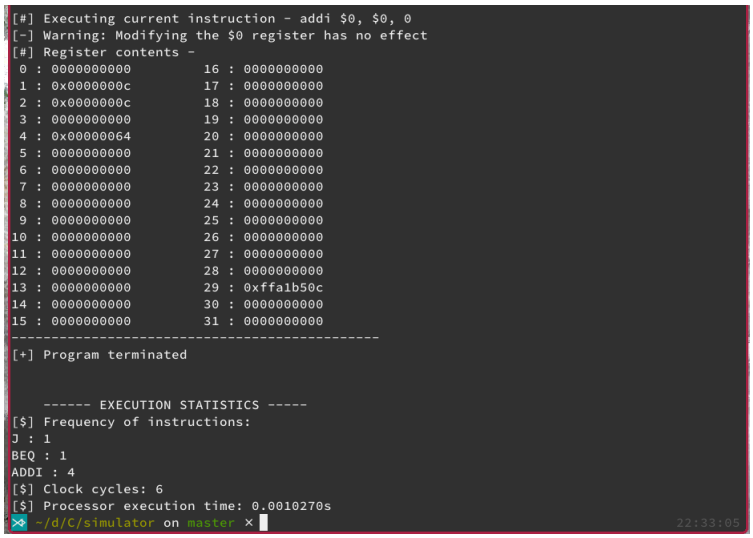
Figure 1: Automatic evaluation

2.3.2 Manual

We tested for all the test cases described above. Some of the outputs are shown below



(a) test cases causing different errors



(b) 1 test case with if-then-else implemented using beq

Figure 2: Manual evaluation

3 Assumptions

- We have assumed that the maximum memory available to any user program is 2^{20} bytes. This includes the memory required for storing instructions and the stack memory.
- We have assumed 32 registers for MIPS each of them storing 32 bits. We have hardwired the register `$0` to the value 0 as is the case in MIPS hardware. We have also restricted use of kernel reserved registers `$26, $27`.
- We have assumed that exactly 1 clock cycle is required for each of the instruction to display the execution statistics.
- We have enforced tight syntax rules to enforce good coding practice. We disallow certain instructions like (a) `lw $5, ($29)` and (b) `lw $5, 40231`. The first instruction skips supplying the offset to the register and the second one involves raw memory access. We have used actual memory addresses allocated by the OS to initialize the stack pointer as will be the case for any program running on an actual harware. This makes our implementation more scalable and robust as compared to using indexes from $0 - 2^{20}$ to reference memory which will most likely be a reserved address and lead to segmentation fault.
- Our implementation also enforces using numeric registers instead of their named counterparts. For example `$5` is allowed but its named counterpart `$a1` is disallowed
- We have enforced the use of a whitespace between the operand and its arguments. For example `add$4,$5,$6` is disallowed as there is no whitespace between `add` and `$4`
- We have implemented branching instructions like `beq, bne, j` to take only labels and instruction numbers as the last argument.