



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	2
1.1	Compiler	2
1.2	Hardware	2
2	Testing	2
2.1	Automated	3
2.2	Manual	3
2.3	Results	4
	2.3.1 Automated	4
	2.3.2 Manual	6
3	Assumptions	7

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 erroneous expression or an unidentified token is encountered.
- The compiler checks the encodability 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.

```
1 $ sudo apt install spim
2 $ python3 -m pip install -r requirements.txt
3 $ 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.

- **TEST1:** arithmetic - testing validity of `add`, `mul`, `sub`
- **TEST2:** conditional - implementing if-then-else using `bne`, `beq` and jump statements
- **TEST3:** looping - implementing loops for counter and fibonacci numbers
- **TEST4:** stacks - storing and retrieving values from the stack
- **TEST5:** errors - extensive testing for errors with invalid register, labels, integers, out of bounds memory addressing, non-word aligned addresses and accessing kernel reserved registers

These tests are present in the tests/manual/ folder.

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.

MIPS Simulator

```
~/.../assignments/simulator master ● python3 tester.py -n 2 -m 10 -v
[o] Building program
[*] Generating test case (1/2)
    addi $8,$15,-12350
    addi $24,$20,-14731
    addi $20,$3,8868
    mul $7,$16,$2
    addi $15,$4,28801
    addi $22,$0,-17462
    addi $10,$21,-11185
    addi $17,$8,-7725
    addi $13,$8,-29998
    addi $22,$19,16364
[*] Extracting program output for : ./input/test0.in
[*] Fetching registers from SPIM
[+] ASM file cleared successfully
[+] Test Passed
-----
[*] Generating test case (2/2)
    addi $20,$12,27222
    mul $21,$13,$19
    add $15,$10,$17
    mul $7,$17,$7
    add $2,$2,$2
    sub $8,$14,$23
    addi $17,$24,13457
    addi $21,$21,10634
    addi $0,$0,-27265
    addi $7,$20,-29429
[*] Extracting program output for : ./input/test1.in
[*] Fetching registers from SPIM
[+] ASM file cleared successfully
[+] Test Passed
-----
[*] Passed 2/2 test cases
```

(a) 2 test cases each with 10 instructions

```
~/.../assignments/simulator master ● python3 tester.py -n 1 -m 20 -v
[-] Building program
[*] Generating test case (1/1)
    addi $13,$18,-938
    sub $17,$7,$14
    addi $15,$12,-7764
    addi $22,$3,-15471
    addi $19,$19,10034
    addi $13,$19,-22679
    addi $0,$21,-3629
    addi $19,$25,29269
    slt $10,$4,$16
    addi $24,$21,25787
    mul $16,$3,$14
    addi $24,$24,5904
    addi $10,$19,29145
    mul $11,$4,$0
    addi $4,$4,17217
    addi $14,$15,29400
    addi $18,$19,1872
    addi $10,$20,30450
    add $0,$14,$8
    sub $21,$7,$25
[*] Extracting program output for : ./input/test0.in
[*] Fetching registers from SPIM
[+] ASM file cleared successfully
[+] Test Passed
-----
[*] Passed 1/1 test cases
```

(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

```

> ~d/C/simulator on master x output/main tests/manual/errors/non_word_aligned_address
[+] Compiling file : tests/manual/errors/non_word_aligned_address
[+] Program compiled successfully !
[+] Executing program ...
[#] Executing current instruction - j 101
main: src/hardware.cpp:197: void Hardware::j(int): Assertion `("Unable to jump, The address
is not word aligned", jump % Hardware::BYTES == 0)' failed.
fish: "output/main tests/manual/errors..." terminated by signal SIGABRT (Abort)
> ~d/C/simulator on master x output/main tests/manual/errors/accessing_kernel_registers
[+] Compiling file : tests/manual/errors/accessing_kernel_registers
[+] Program compiled successfully !
[+] Executing program ...
[#] Executing current instruction - add $26, $0, $0
main: src/hardware.cpp:159: void Hardware::set_register(int, hd_t): Assertion `("Access Den
ied: Cannot modify a kernel reserved register: $26", dst != 26)' failed.
fish: "output/main tests/manual/errors..." terminated by signal SIGABRT (Abort)
> ~d/C/simulator on master x output/main tests/manual/errors/invalid_label 22:31:25
[+] Compiling file : tests/manual/errors/invalid_label
terminate called after throwing an instance of 'InvalidInstruction'
what(): asa p: is not an instruction or label
fish: "output/main tests/manual/errors..." terminated by signal SIGABRT (Abort)
> ~d/C/simulator on master x output/main tests/manual/errors/non_word_aligned_address
[+] Compiling file : tests/manual/errors/non_word_aligned_address
[+] Program compiled successfully !
[+] Executing program ...
[#] Executing current instruction - j 101
main: src/hardware.cpp:197: void Hardware::j(int): Assertion `("Unable to jump, The address
is not word aligned", jump % Hardware::BYTES == 0)' failed.
fish: "output/main tests/manual/errors..." terminated by signal SIGABRT (Abort)
> ~d/C/simulator on master x
22:32:05

```

(a) test cases causing different errors

```

[#] Executing current instruction - addi $0, $0, 0
[-] Warning: Modifying the $0 register has no effect
[#] Register contents -
0 : 000000000 16 : 000000000
1 : 0x0000000c 17 : 000000000
2 : 0x0000000c 18 : 000000000
3 : 000000000 19 : 000000000
4 : 0x00000064 20 : 000000000
5 : 000000000 21 : 000000000
6 : 000000000 22 : 000000000
7 : 000000000 23 : 000000000
8 : 000000000 24 : 000000000
9 : 000000000 25 : 000000000
10 : 000000000 26 : 000000000
11 : 000000000 27 : 000000000
12 : 000000000 28 : 000000000
13 : 000000000 29 : 0xffa1b50c
14 : 000000000 30 : 000000000
15 : 000000000 31 : 000000000
-----
[+] Program terminated

----- EXECUTION STATISTICS -----
[$] Frequency of instructions:
J : 1
BEQ : 1
ADDI : 4
[$] Clock cycles: 6
[$] Processor execution time: 0.0010270s
> ~d/C/simulator on master x
22:33:05

```

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

Figure 2: Automatic 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 hardware. 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 a space between the operand and its arguments. For example `add$4,$5,$6` is disallowed as there is no space between `add` and \$4