Computer Architecture - CS2323. Autumn 2024

Lab-4 (RISC-V Simulator)

This assignment extends your previous assignment where you generated the hex code from a given assembly code. Now, you should support simulating the given assembly code. Parsing of the input assembly code, etc. should already be implemented in the previous assignment. You need to implement some structures in your code that can model registers and memory structures to simulate the execution of RISC-V code. Your implementation (in C/C++ code) should functionally update the registers and memory as per the given instruction and support functionalities as defined in this document.

Instructions on Input:

1. An input RISC-V code will be provided through a text file.
2. The register names in the input program can be specified as x0, x1, … x31 or using their calling convention-based aliases like a0, t0, s0, … or a mix of these. e.g., add t2, x3, s1
3. Input code can be assumed to be syntactically correct.
4. If it helps, you may assume that there is only one space between the instruction and the first operand. Also, only one space between “,” and the second operand, and so on. Similarly, one colon after the label and then one space.
5. Input code will be reading and writing valid memory addresses only.
6. The program starts with the first character in each line
7. There is only one instruction in each line
8. There are no blank lines in the input file
9. You should verify your code’s correctness using various example hex files from RIPES or any other RISC-V simulator.
10. Optional: A line starting with a semicolon (;) can be treated as a comment and omitted. Similarly, anything after a semicolon (;) within a line can be omitted.
11. .data and .dword can be used to initialize some portion of memory

Functionalities to be Implemented:

The executable from your code shall run in an infinite loop, waiting for one of the following commands:

Part 1: (60% weightage)

1. **load <filename>:** Loads the file containing RISC-V assembly code. (all registers and memory get initialized to the default value.)
2. **run:** Execute a given RISC-V code and update registers, memory, etc. It runs the given RISC-V code till the end.
3. **regs:** Print the values of all registers in hex format. (64-bit registers).
4. **exit:** Exit the simulator and print a meaningful message to gracefully exit the program.

Part 2: (20% weightage)

1. **mem <addr> <count>:** Print *count* memory locations starting from address *addr* in the data section. (assume Little Endian format).
2. **step:** Run one instruction and print the “Executed <instruction>; PC=<address>”

Part 3: (20% weightage)

1. **show-stack:** Prints the stack information, and the line elements are pushed onto and popped from the stack frame. The stack frame is only updated on function invocations.
2. **break <line>:** Sets a mark to stop the code execution once the line is reached, preserving registers and memory state.
3. **del break <line>:** Deletes the breakpoint at the specified line. If no breakpoint is present, a meaningful error message can be printed.

Instructions on Simulator:

1. Your code will run in an infinite loop, waiting for instructions as described above.
2. On every instruction, update the corresponding memory addresses and registers.
3. On every instruction run, it should print the value of PC on the terminal and the corresponding instruction (e.g., Executed instruction add x8, x9, x5; PC=0x00001000). If run till a breakpoint, also print ”Execution stopped at breakpoint”
4. The text section spans from 0x0 -> 0x10000. Each instruction is 4 bytes.
5. The data section starts at 0x10000. The stack starts at 0x50000 and grows downward.
6. The simulator must wait for the command even if input code execution is complete.
7. Registers and data memory are initialized to 0 by default.
8. Step does not update any register/memory after the last instruction and prints “Nothing to step”
9. After the output of each command, leave an empty line.
10. By default, **show-stack** shows main at the top of the stack. As function calls occur, it is pushed to bottom. main is popped out of call stack only at the end of execution. Basically, main label won’t necessarily be specified.
11. The line number shown with labels is the last line number executed. So initially, program is at main:0
12. Up to 5 breakpoints can be set by the user on the assembly code.

Submission instructions:

The assignment is to be done in groups of two. Submit your code and a Makefile which would compile your code and generate an executable named riscv_asm.
Prepare a short report on your coding approach and what all you did for testing your code to be correct. ONLY ONE MEMBER OF A GROUP SHOULD MAKE THE SUBMISSION.
Submit a zip file (Lab4_CSYYBTECHZZZZZ_CSYYBTECHXXXXX.zip) containing the following:

1. Source files of your code
2. Makefile - A makefile that would compile your code and generate an executable named riscv_sim
3. README - Containing information about various files in the directory, usage instructions, etc.
4. Test Cases, if you tried out specific input files
5. report.pdf - a short report on your implementation/coding approach and what all you did for testing your code to be correct.

Demo: We may later conduct a demo where you will show the functioning of your code to the TAs and accordingly be assigned marks.

Note: Submissions are subject to similarity check and hence submit only your own work. Any similarity will be strictly dealt with.
Your submissions will be evaluated on Ubuntu 20.04 machines and hence you must check them to be working on such a setup. If you do not have those machines, please visit the B-524 lab.

Examples:

1. **input.s**

```
addi x5, x5, 0
lui x4, 0x10
addi x5, x5, -1
sd x5, 2(x4)
addi x5, x5, 1
```

Command Line Interface:

```
load input.txt
step
Executed: addi x5, x5, 0 ; PC = 0x00000000

step
Executed: lui x4, 0x10 ; PC = 0x00000004

step
Executed: addi x5, x5, -1; PC = 0x00000008

step
Executed: sd x5, 2(x4); PC = 0x0000000c

mem 0x10000 4
Memory[0x10000] = 0x0
Memory[0x10001] = 0x0
Memory[0x10002] = 0xFF
Memory[0x10003] = 0xFF

regs
Registers:
x0  = 0x0
x1  = 0x0
x2  = 0x0
```

```
x3  = 0x0
x4  = 0x10000
x5  = 0xFFFFFFFFFFFFFFFF
x6  = 0x0
x7  = 0x0
x8  = 0x0
x9  = 0x0
x10 = 0x0
x11 = 0x0
x12 = 0x0
x13 = 0x0
x14 = 0x0
x15 = 0x0
x16 = 0x0
x17 = 0x0
x18 = 0x0
x19 = 0x0
x20 = 0x0
x21 = 0x0
x22 = 0x0
x23 = 0x0
x24 = 0x0
x25 = 0x0
x26 = 0x0
x27 = 0x0
x28 = 0x0
x29 = 0x0
x30 = 0x0
x31 = 0x0

run
Executed addi x5, x5, 1; PC=0x00000010

step
Nothing to step

exit
Exited the simulator
```

2. **Input.s:**

```
main: addi x10, x0, 2
      jal x1, fact
      beq x0, x0, exit
```

```
fact: addi sp, sp, -16
      sd x1, 8(sp)
      sd x10, 0(sp)
      addi x5, x10, -1
      blt x0, x5, L1
      addi x10, x0, 1
      addi sp, sp, 16
      jalr x0, 0(x1)
L1:   addi x10, x10, -1
      jal x1, fact
      addi x6, x10, 0
      ld x10, 0(sp)
      ld x1, 8(sp)
      addi sp, sp, 16
      addi x20, x0, 1
mul:  add x10, x10, x10
      addi x20, x20, 1
      bne x20, x6, mul
      jalr x0, 0(x1)
exit: add x0, x0, x0
```

Command Line Interface:

```
load input.s

break 13
Breakpoint set at line 13

show-stack
main:0

run
Executed addi x10, x0, 2; PC=0x00000000
Executed jal x1, fact; PC=0x00000004
Executed addi sp, sp, -16; PC=0x0000000c
Executed sd x1, 8(sp); PC=0x00000010
Executed sd x10, 0(sp); PC=0x00000014
Executed addi x5, x10, -1; PC=0x00000018
Executed blt x0, x5, L1; PC=0x0000001c
Executed addi x10, x10, -1; PC=0x000002c
Execution stopped at breakpoint
```

```
show-stack
Call Stack:
main:2
fact:13

step
Executed jal x1, fact; PC=0x00000030

run
Executed addi sp, sp, -16; PC=0x0000000c
Executed sd x1, 8(sp); PC=0x00000010
Executed sd x10, 0(sp); PC=0x00000014
Executed addi x5, x10, -1; PC=0x00000018
Executed blt x0, x5, L1; PC=0x0000001c
Executed addi x10, x10, -1; PC=0x000002c

step
Executed jal x1, fact; PC=0x00000030


show-stack
Call Stack:
main:2
fact:13
fact:13

run
Executed addi sp, sp, -16; PC=0x0000000c
Executed sd x1, 8(sp); PC=0x00000010
Executed sd x10, 0(sp); PC=0x00000014
Executed addi x5, x10, -1; PC=0x00000018
Executed blt x0, x5, L1; PC=0x0000001c
Executed addi x10, x0, 1; PC=0x00000020
Executed addi sp, sp, 16; PC=0x00000024
Executed jalr x0, 0(x1); PC=0x00000028
Executed addi x6, x10, 0; PC=0x00000034
Executed ld x10, 0(sp); PC=0x00000038
Executed ld x1, 8(sp); PC=0x0000003c
Executed addi sp, sp, 16; PC=0x0000040
Executed addi x20, x0, 1; PC=0x0000044
Executed add x10, x10, x10; PC=0x00000048
Executed addi x20, x20, 1; PC=0x0000004c
Executed bne x20, x6, mul; PC=0x00000050
Executed jalr x0, 0(x1); PC=0x00000054
Executed beq x0, x0, exit; PC=0x00000008
```

```
Executed add x0, x0, x0; PC=0x00000058

show-stack
Empty Call Stack: Execution complete
```