CS 152 Lab 1: RISC-V Assembly

Due: 11:59 PM, Feb 4

Overview

For this lab, you will write RISC-V assembly code to add a sequence of numbers together according to some conditions, and analyze its performance. You will submit both your code and a short report answering some questions. (See "What to Submit)

RISC-V Emulator

It is suggested to use the provided lightweight command-line RV32I ISA emulator for this lab. The emulator and its instructions can be found here: https://github.com/sangwoojun/rv32emulator

Please take some time to read the instructions given in the github page, as the emulator has helper functionality such as breakpoints, single-stepping, and memory dump. It also has many limitations, such as not allowing comments at the end of lines, and words separated by whitespace instead of commas.

The use of this emulator is simply for your convenience, and you are welcome to use other, more elaborate emulators such as Spike (https://github.com/riscv/riscv-isa-sim), or Ripes (https://github.com/mortbopet/Ripes).

Problem Description and Skeleton Code

Skeleton code is provided in the rv32emulator repository, located at: example_questions/reduction.s.

You will finish the implementation of the function labeled "solve", which takes three arguments via "a0", "a1", and "a2". The length of the array to add is given in "a0", and "a1" holds the address of the input array to add. Register "a2" holds the "mask" array. Values in the input array should only be added if the corresponding location in the mask array holds a nonzero value.

Once all valid numbers are added, call the "submit" function using jal, giving the total sum via register a0. Note that you must correctly manage the "ra" register, as calling jal for the submit function will overwrite the "ra" value set when the "solve" function was called. This is why the "submit" function call is commented out in the provided version. Simply un-commenting this line without any other changes will cause the code to go into an infinite loop.

Once the execution reaches the special "Halt and Catch Fire" instruction used to mark the natural end of the program, it will print the state of the emulated machine at that point, and exit. But before that, the assembly code in the skeleton code will compare the submitted answer with the pre-calculated correct one, and emit a return code. The return code will be zero if the answer is correct, and 1 if not.

Figure 1 shows an example execution of the skeleton code with incorrect results. Note the two red rectangles. The first one shows the "c" command, which tells the emulator to continue execution until a breakpoint is met, or until the end of the program. The second one shows the return code emitted by the skeleton code, meaning the submitted answer is incorrect.

```
[redacted]:~/rv32emulator$ ./obj/emulator ./example_questions/reduction.s
Parsing input file

Next: lui x02, 0x00000010
[inst:      1 pc:      0, src line    4]
>>c
[System output]: 0x1


----------

Reached Halt and Catch Fire instruction!
inst:     24 pc:     88 src line: 24
x00:0x00000000 x01:0x00000028 x02:0x00010000 x03:0x00000000 x04:0x00000000 x05:0x00000001 x06:0x00000000 x07:0x00000000
x08:0x00000000 x09:0x00000000 x10:0x00000010 x11:0x00002004 x12:0x00002014 x13:0x00000000 x14:0x00000000 x15:0x00000000
x16:0x00000000 x17:0x00000000 x18:0x00000000 x19:0x00000000 x20:0x00000000 x21:0x00000000 x22:0x00000000 x23:0x00000000
x24:0x00000000 x25:0x00000000 x26:0x00000000 x27:0x00000000 x28:0x00000000 x29:0x00010000 x30:0x00000000 x31:0x00000000
Execution done!
[redacted]:~/rv32emulator$
```

Figure 1: Example execution with incorrect results.

What to Submit

1. Your modified reduction.s. Please be mindful of the comment that tells you the parts of the code that can be modified!
2. A short report (pdf, txt, odf, rtf, doc/docx) answering the questions below:
   (a) Use the breakpoint functionality described in the repository's README, and measure how many instructions were executed inside the "solve" function. How many instructions did it take to add 16 numbers? Is it good? Can you think of ways to improve this just in software?
   Note: Every time a breakpoint is met, it will print the total number of instructions executed so far, along with the current PC address, and the current line number. See the yellow box in Figure 1. You can set the breakpoint at the first line of "solve", and "ret".
   (b) Thinking back on the variety of ISA designs we talked about so far, what ISA changes do you think can improve the numbers we saw in (a)? Please include a short justification. There is no single correct answer to this question.