CS2292 Project Phase 1

Pranathi, Swetha and Sai Datta

February 16, 2022

- Develop a simulator on the lines of QtSPIM for RISC-V architecture.
- RISC-V, like MIPS, is an ISA based on Reduced Instruction Set Computer. It shares some similarity with MIPS including the same 5 pipeline stages. It originated at UC-Berkeley as an easy-to-extend ISA for academia and research, one of the key persons involved being our beloved David Patterson (you mighta heard). It has about 6 different instruction type formats, also specifying a design that works well with the hardware, and can be adapted for various architectures like vector-supported, in-order, out-of-order etc.
- RISC-V is the new black, or green shall we say! Intel and MIPS have jumped in on the RISC-V bandwagon. There is buzz about processors for AI computing and graphics! Read more to find out! [here] [here] and [here] !
- Find the RISC-V manual [here]. Be sure to read their commentary on design decisions (written in small font) for fun insights.

  Please note that you do NOT have to read the entire manual. It is **NOT** a textbook. Your simulator should support the following RISC-V instructions: ADD/SUB, BNE, JAL(jump), LW/SW, and an instruction of your choice. You might want to add in some immediate-type instructions too!
  Example usage:

```
LABEL1: add x0, x0, x1  #rd, rs1, rs2
        sub x0, x0, x1  #rd, rs1, rs2
        bne x0, x2, LABEL #rs1, rs2, offset (but label will
           suffice)
        sw x0, 0(x3) #rd, offset(rs1)
LABEL:  sub x0, x0, x2
        jal x0, LABEL1  #rd, offset (again, label will work)
```

  Note that rd and rs stand for destination and source registers. Comments start with #

  If you want your simulator to support functions, you can implement jal and jalr opcodes. Note that it is not a requirement for this phase.

An example of how jal and jalr might be used (not necessary that function calls work this way only)

```
func:
  inst 0: add x3,x4,x5
  inst 1: addi x3, x3, 1
  inst 2: jalr x1, x0, 0  #rd, rs1, off
main:
  inst 3: add x1, x1, x2
  inst 4: jal x0, -4 #rd, off
  inst 5: add x1, x2, x1
#at inst 4, pc is 4*4bytes = 16, offset is -4
#so x0 becomes pc+4 = 20.. which is address of inst 5, the next
   inst where pgrm has to come back after execution of func
#pc becomes 16 + -4*4 = 0... which is address of inst 0 which is
   the first inst of func, execution continues normally
#when it comes to inst 2: jalr... pc becomes x0 + 0*4 = 20 (
   notice NOT relative to pc).. so prgm comes to inst 5 and
   continues execution.
```

- Traditionally, for a 32-bit architecture, each instruction (of base type) is of 32 bits, and there are 32 registers(considering only integer supported) apart from PC. Each register also consists of 32 bits (4-byte word). If you decide to simulate an architecture different from the RISC-V, the register size, the number of registers, and the instruction width, you should have a design justification.
- The simulator should support atleast 4kB of memory.
- The simulator should read in an assembly file, execute the instructions, and in the end display the contents of the registers, and the memory.
- Features like single step execution, graphical interface or any other feature you can think of, is not compulsory, and is left to your choice.
- Code should be maintained using git, and uploaded on github.
- Any programming language can be used to develop the simulator.
- Your simulator should be able to run bubble sort written in assembly.
- If you are unable to complete the project, do prepare a document detailing what you tried, and what did not work etc. The document along with the incomplete code will be evaluated.
- Any kind of malpractice will fetch you a straight **F**. Malpractice also includes sharing your code with someone else!
- Individual members of a team will be evaluated based on their contribution.
- This is “your” project, “you” might add it in “your” CV.
- **Deadline**: March 6^{th} 11:59PM

- Have fun!