Programming Project: riscv-simulator

In this project, you will be familiar with how the assembler work and how the RISC-V instruction set is implemented. Through this project, you should learn the principles and techniques used in implementing a processor. Your assignment is to models an assembler and an unpipelined processor for a small RISC-V instruction set.

1 Files

There are 6 files: *asm.c, riscv-small.c, Makefile, simple.s, simple.out, sim_simple.output*.

1) *asm.c*: an assembler that you will design for the reduced RISC-V ISA (more about the assembler and the ISA later).
2) *riscv-small.c*: a fully functional unpipelined simulator which also you will design.
3) *Makefile*: a unix make file which will produce the binaries *asm* (assembler) and *sim* (simulator) from the source files, *asm.c* and *riscv-small.c*, respectively.
4) *simple.s*: an example assembly program
5) *simple.out*: the assemble code file of assembling *simple.s* through your assembler *asm*
6) *sim_simple.output*: the result of running *simple.out* through your simulator *sim*

2 A Scaled-Down RISC-V ISA

You will be simulating the RISC-V ISA and implement a 32-bit architecture. Also, to keep your simulator simple, you will only be required to support a scaled-down version of the RISC-V ISA consisting of 10 instructions. Then, these instructions, along with their encoding, are given in Table 1.

name	*format type*	*instruction format*					
beq	*SB-type*	*imm[12|10:5]*	*rs2*	*rs1*	*000*	*imm[4:1|11]*	*1100011*
lw	*I-type*	*imm[11:0]*		*rs1*	*010*	*rd*	*0000011*
sw	*S-type*	*imm[11:5]*	*rs2*	*rs1*	*010*	*imm[4:0]*	*0100011*
addi	*I-type*	*imm[11:0]*		*rs1*	*000*	*rd*	*0010011*
add	*R-type*	*0000000*	*rs2*	*rs1*	*000*	*rd*	*0110011*
sub	*R-type*	*0100000*	*rs2*	*rs1*	*000*	*rd*	*0110011*
sll	*R-type*	*0000000*	*rs2*	*rs1*	*001*	*rd*	*0110011*
srl	*R-type*	*0000000*	*rs2*	*rs1*	*101*	*rd*	*0110011*
or	*R-type*	*0000000*	*rs2*	*rs1*	*110*	*rd*	*0110011*
and	*R-type*	*0000000*	*rs2*	*rs1*	*111*	*rd*	*0110011*

Table 1: instruction encoding for a reducing RISC-V ISA

3 asm: an assembler for the reduced RISC-V ISA

In this project, you should creat an assembler, *asm.c*, that you use to assemble programs for your simulator. The format for assembly programs is very simple. A valid assembly program is an ASCII file in which each line of the file represents a single instruction, or a data constant. The format for a line of assembly code is:

Label <tab> instruction <tab> field0 <tab> field1 <tab> field2 <tab> comments

The leftmost field on a line is the label field that indicates a symbolic address. Valid labels contain a maximum of 6 characters and can consist of letters and numbers. The label is optional (the tab following the label field is not). After the optional label is a tab. Then follows the instruction field, where the instruction can be any of the assembly-language mnemonics listed inTable1. After another tab comes a series of fields. All fields are given as decimal numbers. The number of fields depends on the instruction. The following describes the instructions and how they are specified in assembly code:

lw	*rd*	*rs1*	*imm*	*Reg[rd] <- Mem[Reg[rs1] + imm]*
sw	*rs2*	*rs1*	*imm*	*Mem[Reg[rs1] + imm] <- Reg[rs2]*
beq	*rs1*	*rs2*	*imm*	*if (Reg[rs1] == Reg[rs2]) PC <- PC+imm*
addi	*rd*	*rs1*	*imm*	*Reg[rd] <- Reg[rs1] + imm*
add	*rd*	*rs1*	*rs2*	*Reg[rd] <- Reg[rs1] + Reg[rs2]*
sub	*rd*	*rs1*	*rs2*	*Reg[rd] <- Reg[rs1] - Reg[rs2]*
sll	*rd*	*rs1*	*rs2*	*Reg[rd] <- Reg[rs1] << Reg[rs2]*
srl	*rd*	*rs1*	*rs2*	*Reg[rd] <- Reg[rs1] >> Reg[rs2]*
and	*rd*	*rs1*	*rs2*	*Reg[rd] <- Reg[rs1] & Reg[rs2]*
or	*rd*	*rs1*	*rs2*	*Reg[rd] <- Reg[rs1] | Reg[rs2]*
halt		*stop simulation*		

Note that in the case of the beq instruction, PC-relative addressing is used (and again, your simulator should not perform the left-shift when computing the PC-relative branch target). For the *lw*, *sw*, and *beq* instructions, the *imm* field can either be a decimal value, or a label can be used. In the case of a label, the assembler performs a different action depending on whether the instruction is a *lw / sw* instruction, or a *beq* instruction. For *lw* and *sw* instructions, the assembler inserts the absolute address corresponding to the label. For *beq* instructions, the assembler computes a PC-relative offset with respect to the label.

After the last field is another tab, then any comments. The comments end at the end of the line.

In addition to instructions, lines of assembly code can also include directives for the assembler. The only directive we will use is *.fill*. The *.fill* directive tells the assembler to put a number into the place where the instruction would normally be stored. The *.fill* directive uses one field, which can be either a numeric value or a symbolic address. For example, "*.fill 32*" puts the value 32 where the instruction would normally be stored. In the following example, "*.fill 50*" will store the value 50:

```
lw    21    0     op1  reg[21] <- op1
lw    22    0     op2  reg[22] <- op2
```

```
        lw    23    0     op3   reg[23] <- op3
        add   24    21    22    reg[24] <- reg[21] + reg[22]
        sub   25    24    23    reg[25] <- reg[24] - reg[23]
        sw    25    0     answer    reg[25] -> answer
donehalt
op1   .fill   50
op2   .fill   30
op3   .fill   20
answer    .fill   0
```

Try taking the above example and running the assembler on it. Enter the above assembly code into a file called "*simple.s*". Then type "*asm simple.s simple.out*". The assembler will generate a file "*simple.out*" which should contain:

```
(address 0x0):      01c02a83
(address 0x4):      02002b03
(address 0x8):      02402b83
(address 0xc):      016a8c33
(address 0x10):     417c0cb3
(address 0x14):     03902423
(address 0x18):     0000003f
(address 0x1c):     00000032
(address 0x20):     0000001e
(address 0x24):     00000014
(address 0x28):     00000000
```

The assembler assumes that all programs will be loaded into memory beginning at address 0x0.

As mentioned above, your simulator must produce assembly code. In addition, your assembler must also use the following loop code (which we have provided). And in the source code, we have given you a example of how assemble *add* instruction:

```
if (!strcmp(opcode, "add")) {
        num = (ADD_FUNC7 << FUNCT7_SHIFT) | (atoi(arg2) << RS2_SHIFT) | (atoi(arg1) <<
RS1_SHIFT) | ADD_FUNC3<<FUNCT3_SHIFT | (atoi(arg0) << RD_SHIFT) | REG_REG_OP;
}
/* STUDENT CODE START HERE. */

/* STUDENT CODE END. */
else if (!strcmp(opcode, "halt")) {
        num = HALT_OP;
}
```

When you start to creat your assembler, you will insert the necessary code between the

comments that perform the functions in asm.c, but do not change any other part of this loop. Especially, be careful not to change the output format of *printState*.

4 Simulator

As described above, you will design a simulator that simulates the same ISA. Before creating this simulator, you should also look at the code in *riscv-small.c*. The code is very simple, but you should make sure you understand it before moving on to building the simulator. **When you start to creat your simulator, you will insert the necessary code between the comments that perform the functions in *riscv-small.c*. Especially, be careful not to change the output format of *printState*.**

If you haven finished designing, build the simulator by typing "*make sim*" in the directory where you've copied the files. This will produce an executable *sim* which you can run. Try running this simulator on the program we have provided, *simple.s*. Be sure to assemble the program before running the simulator. The sequence of commands you should give are:

make sim
./asm mult.s mult.out
./sim mult.out

When you run the simulator, a ton of messages will spew onto your terminal. The simulator we have provided dumps the state of the machine (including the contents of registers and memory). You can examine this output to see what the simulator is doing. To put these messages in a file so that you can actually read them, redirect the output of the simulator to a file by saying "*./sim mult.out > sim.output*".

5 Assignment

Your assignment is to build an assembler and a simulator. We have provided some code to get you started in *asm.c* and *riscv-small.c*. This code serves two purposes: it will make your life a bit easier since you won't have to write the whole simulator from scratch, and it is meant to enforce some coding disciplines that ensure you actually simulate the details of the processor.

6 Grading

We will grade your programming assignment based on correct functionality. In other words, for any valid reduced RISC-V ISA program, your simulator should accurately simulate the program and produce the correct values at each cycle. We will be using our own programs to validate your simulator, so make sure you you're your simulator on several test cases (In other words, you should write a few extra assembly programs to test your simulator. Recommended to test your code under Linux operation system.).

7 Academic Honesty

This is a one-person project. Do not allow any other student to see any of your code. You may however discuss the assignment in general terms, with the other students.

8 Submission

To submit your solution, make a directory with the same name as your student id (317010xxxx), copy your **report** and **all the code files (all of your *.c* and *.h*)** to that directory and tar and compress the directory, naming the tar file with the student id.
The final step is to upload the compressed tar file *317010xxxx.tgz* to Learning in ZJU (http://courses.zju.edu.cn/) before the deadline.

9 Deadline

2019.11.28 24:00
If you have any question, please contact TA.