CS330L Computer Architecture Lab
Lab 03: Introduction to RISC-V assembly (Decision Instructions)

CS Program
Habib University

Spring 2020

1 Introduction

In this lab we will learn about:

1. RISC-V pseudo instructions
2. the use of RISC-V decision instructions

2 Pseudo Instructions

RISC-V registers are named `x0` through `x31`. Except `x0`, which is hardwired to contain all zeros, the rest can be used to store arbitrary data. Still, a few conventions have been defined about the use of these registers and things run much smoother when everybody follows them. Table ?? provides a summary of conventional use of various registers.

RISC-V Registers			
Register	ABI Name	Description	Saved by
x0	zero	zero register(hardwired zero)	
x1	ra	return address	Caller
x2	sp	stack pointer	Callee
x3	gp	global pointer	
x4	tp	thread pointer	
x5-7	t0-2	temporary registers	Caller
x8	s0 / fp	saved register / frame pointer	Callee
x9	s1	saved register	Callee
x10-11	a0-1	function arguments / return values	Caller
x12-17	a2-7	function arguments	Caller
x18-27	s2-11	saved registers	Callee
x28-31	t3-6	temporary registers	Caller

The second column lists their pseudonyms which can be used in the assembly language as their original name equivalents. These pseudonyms are easier to remember and also give an inkling to the conventional use of the register.

Similarly, for ease of use, the assembler provides a series of pseudo instruction which are easier to remember and are more user friendly. The assembler translates these pseudo instructions into machine code for RISC-V equivalent instructions. Some of the pseudo instructions and their equivalent real RISC-V assembly instructions can be found on page 7 of this PDF at: `https:`

//compil-lyon.gitlabpages.inria.fr/compil-lyon/CAP1920_ENSL/riscv_isa.pdf. [1]

You can see from their syntax that some of them, i.e., `nop`, `li`, `mv`, etc., can be quite handy.

Advice. One advice is to follow one coding style only i.e., either use the pseudo instructions or use the original instructions with the original register names. Mixing them up will almost surely result in bugs. Same goes for register names.

3 Decision Instructions

Decision instructions help us manipulate the control flow of a program. Normally the processor would load an instruction, from the address stored in `PC`, from the RAM and execute it. While executing it would update the value of `PC` by adding 4 to it, i.e. `PC = PC+4;`, so that it now points to the next instruction in RAM. This process is repeated every clock cycle and the processor will proceed to continue executing instructions in a linear fashion forever, that is unless it runs out of RAM.

If we want to change this linear control flow, i.e. we want to execute some instruction other than the one next in line, we need to load the address of that instruction into the PC so that it becomes the next instruction to be executed. How do we know the address of a particular instruction? The assembler comes to our rescue here with the concept of `labels`.

3.1 Labels

```
  add x2, x3, x4
label1:
  sub x3, x0, x2    # label1 contain this instruction address
  addi x3, x3, 1

label2: sub x5, x6, x3  # label2 contain this instruction address
```

Listing 1: Labels

Labels are text strings, followed by a colon, that we can put before instructions. These strings can be used as a substitute for addresses of those instruction. The Listing 1 shows the use of labels. In this piece of code, `label1` and `label2` can be used instead of the addresses of the instructions that follow them. We will see next how to use these labels.

3.2 Conditional Branches

Branches are instructions which can be used to jump to a particular instruction by supplying its label. If the branch instruction performs the jump based on a particular condition then such branches are called conditional branches.

```
beq x22, x23, L1
bne x22, x23, L2
```

Listing 2: Labels

Two of the principal conditional branch instructions provided in the RISC-V ISA are called `beq` and `bne` as shown in Listing 2 where `L1` and `L2` both are labels as defined previously.

The instruction `beq` will jump to the instruction whose label is `L1` if the values in registers `x22` and `x23` are equal. That is to say that after the execution of this instruction the `PC` will

[1] Here as well: `https://github.com/riscv/riscv-asm-manual/blob/master/riscv-asm.md` `https://github.com/kvakil/venus/wiki/Pseudoinstructions`.

contain the address of the instruction whose label is `L1` and the processor will load that instruction for execution next instead of the one after the `beq` instruction. If however if the values in registers `x22` and `x23` are not equal, the jump will not happen and the `PC` will be updated with the value `PC+4` and the processor will continue with the execution of the next instruction in RAM.

The instruction `bne` behaves opposite to `beq`, i.e., it will jump to `L2` if the values in the two registers are unequal and continue with the next instruction in RAM if they are not.

Listings 3 and 4 show how these instructions can be used to implement `if-else` statements and `loops`.

```
  if (i == j)
    f = g + h;
  else
    f = g - h;
  // code after if/else goes here



  //assuming that variables f to j are in registers x19-x23
  bne x22, x23, Else
  add x19, x20, x21
  beq x0, x0, Exit  //un conditional jump
Else: sub x19, x20, x21

Exit: # the code after if/else goes here
```

Listing 3: If statement

```
  while (save[i] == k)
    i += 1;


  // assuming i and k in x22 and x24, and the base address of Save in x25
  Loop: slli x10, x22, 3 // Temp reg x10 = i * 8
      add x10, x10, x25 // x10 = address of save[i]
      ld x9, 0(x10) // Temp reg x9 = save[i]
      bne x9, x24, Exit // go to Exit if save[i] != k
      addi x22, x22, 1 // i = i + 1
      beq x0, x0, Loop // go to Loop
  Exit:

```

Listing 4: while loop

4 Task 1

4.1 Task 1a:

Type the codes in Listings 3 and 4 in the assembler editor and then in the simulator look at their binary:

- the `beq` and `bne` instructions are stored in the **SB format** with their 32-bit binary. Look it up and try to find out what values their different fields contain.
- Looking at the binary formats, find out how do the `labels` are stored in the 32-bit instruction. Compare the `label` values stored inside the instruction to the code and try to guess how does the assembler stores the jump addresses inside the instruction. (Hint: it uses a PC-relative scheme)

4.2 Task 1b:

The `switch` statement is similar to if/else statements. Write the equivalent RISC-V assembly code for the Listing **??**. Assume that the variables x, a, b, c are signed integers and stored in x20, x21, x22, x23 respectively.

```
switch (x) {
  case 1:
    a = b+c;
    break;
  case 2:
    a = b-c;
    break;
  case 3:
    a = b * 2;
    break;
  case 4:
    a = b / 2;
    break;
  default:
    a = 0;
}
```

Listing 5: task 1b.c

4.3 task 1c:

```
for(int i=0; i<10; i++)
  a[i] = i;

for(int i=0; i<10; i++)
  sum = sum+a[i];
```

Listing 6: Reduction Sum

Write the equivalent RISC-V assembly code for Listing 6. Assume that the variables `i` and `sum` are in `x22` and `x23`, while the array `a` located at address 0x200 is of 4-byte integers.

4.4 task 1d:

```
for(int i=0; i<10; i++)
  a[i] = i;

for(int i=0; i<10; i++){
  for( int j=i; j<10; j++){
    if (a[i] < a[j]){
      int temp = a[i];
      a[i] = a[j];
      a[j] = temp;
    }
  }
}
```

Listing 7: Bubble sort

Write the equivalent RISC-V assembly code for Listing 7. Assume that the variables `i`, `j`, and `temp` are in `x22`, `x23`, and `x5` while the array `a` located at address 0x200 is of 4-byte integers.

5 Rubric

StudnetID _____ Student Name _____

Rubric			
Task No.	LR2 Code/Design (50)	LR5 Result (20)	LR7 Viva (30)
Task 1a	00	05	05
Task 1b	15	05	10
Task 1c	10	05	05
Task 1d	25	05	10
Total	100		

Marks obtained			
Task No.	LR2 Code/Design (50)	LR5 Result (20)	LR7 Viva (30)
Task 1a			
Task 1b			
Task 2a			
Task 2b			
Total			