CS330L Computer Architecture Lab
Lab 04: Introduction to RISC-V assembly (Jumps and Returns)

CS Program
Habib University

Spring 2020

1 Introduction

In this lab we will learn about:

1. RISC-V Jump and Return instructions
2. Handle the stack

2 Function calls

Functions, or procedures as they are sometimes called, are an important part of programming. They help avoid repeating the same code all over our programs. A function call would take the program flow to the start of the function code. That code will be executed and a return-from-function instruction would take the program flow back to where originally the function was called from and continue the execution from the next instruction after the function call.

Like all instruction sets, RISC-V provides special instructions to enable function calls and return from functions.

```
1 jal    x1, func_name
2 jalr   x0, 0(x1)
```

Listing 1: Function instructions

Listing 1 lists the `jal` and `jalr` instructions used for function calls and returns from functions respectively in RISC-V.

Executing `jal` instruction would make the program flow jump to `func_name` label in the code while at the same time storing the address of the next instruction `PC+4` in the register `x1`. The return address needs to be stored so that the program flow can be able to go back to where the function call was made in order to continue execution from there onwards once it has finished executing the function code. Since a function can be called from multiple locations in a program, without saving the return address we will not have a clue as to where to return to.

The `jalr` instruction is used to return from functions. The jump target address is provided in a register. The example shown in Listing 1, it will jump to the address stored in the register `x1`. Combining this with `jal` instruction we can see how the two instructions work in tandem to provided the functionality of calling a function and returning from it.

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

Table 1: RISC-V Registers

Another issue associated with function calls is passing the parameters and returning the result. As can be seen from Table 1, the RISC-V convention reserves the registers `x10-x17` for passing parameters to a function when calling it. The caller code would fill in these registers the parameters needed by a function before calling it. Once the function is called, it will access its parameters from these registers, use them, and then return the result in `x10` before calling the `jalr` instruction which would take the control flow to the caller who can then access the result of the function call by reading `x10`.

In case the registers `x10-x17` contains some values which will still be used by the caller code after the function call, the caller code needs to store these values as well as its own return address `x1` on the stack before overwriting these values with the ones needed by the callee. See section 2.8 in the book **??**.

3 Environment Calls

The Venus simulator supports some system calls, which it calls Environment Calls, as well. See the url `https://github.com/kvakil/venus/wiki/Environmental-Calls`. To execute an environment call we have to put its number in the register `a0(x10)` , any argument it might need in `a1(x11)` and then call the `ecall` instruction. The simulator will perform the system call specified in the `a0(x10)` register.

So far we have not had any use for the `Console Output` window in the simulator. Here it comes.

The RISC-V environment calls can be used to print values from registers or memory in the console window. For example, one of the supported system calls is `print_int` whose number is `1` and which will print any value stored in the register `a1(x11)` in the console output as an integer.

```
addi a0 x0 1        # print_int ecall
addi a1 x0 42       # integer 42
ecall
```

Listing 2: Print integer

The code in listing 2, taken from the above url, would print the integer `42` on the console output.

4 Task 1

```
int sum( int a, int b){
  return a+b;
}
```

Listing 3: Sum

Write a code for the function `sum` shown in listing 3. It takes two integers `a` and `b` as arguments and returns their sum. You should also write code that calls this function i.e. it sets up the two arguments `a` and `b` in registers `x10` and `x11` and then calls `sum()`. After calling `sum()`, it then retrieves the the returned result from the register `x10`. It can display the result using the `ecall` instruction.

5 Task 2:

```

void bubble (int *a, unsigned int len){

  if (a==NULL || len==0)
    return;

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

  return;
}
```

Listing 4: Bubble sort

Write the equivalent RISC-V assembly code for Listing 4. `a` and `len` get passed in `x10` and `x11` respectively.

6 Task 3

The nth triangular number is like a factorial but instead of a product we use a summation: `https://en.wikipedia.org/wiki/Triangular_number`.

```
int ntri (int num){
  if (num<=1)
    return 1;

  return num + ntri (num-1);
}
```

Listing 5: Nth Triangular number

Translate the above C function to calculate the Nth Triangular number into RISC-V assembly. You should also write the wrapper code, i.e., which calls this function with an argument and then displays the result returned by this function.

Since there's recursion going on, you will need to adjust the stack so that you store the return address as well as any useful variables of the the caller on the stack before calling the callee.

7 Rubric

StudnetID _____ Student Name _____

Rubric			
Task No.	LR2 Code/Design (50)	LR5 Result (20)	LR7 Viva (30)
Task 1	10	05	05
Task 2	25	05	10
Task 3	25	05	10
Total	100		

Marks obtained			
Task No.	LR2 Code/Design (60)	LR5 Result (15)	LR7 Viva (25)
Task 1			
Task 2			
Task 3			
Total			