Lab 11: Functions in RISC-V Assembly

Digital Design and Computer Architecture: RISC-V Edition (Harris & Harris, Elsevier © 2021)

Objective

The purpose of this lab is to write functions using RISC-V assembly language.

1. RISC-V Assembly Functions

Write the following RISC-V assembly functions, and then call them as described below. Simulate each function (and a call to the function) using the Venus simulator.

Program 1: Divisible by 9?
This is the same program that you wrote in Lab 10 but this time you will write it as a function. Write a RISC-V assembly function that determines if the value in its input argument (an unsigned integer) is divisible by 9. The function prototype looks like this:

```
int div9(unsigned int val);
```

If the input argument (`val`) is divisible by 9, the function should return 1. If not, it should return 0. Be sure to comment each line of code (using #). Beyond the instructions needed for a function call (jal and jr ra), **you may use only the following subset of instructions:** `add`, `addi`, `sub`, `beq`, `blt`, `bge`, `bne`, and `j`. Name your text file **div9Function.s**.

Test your function: Now test the function by calling it several times. Test at least five cases, but submit only these cases (i.e., the RISC-V assembly version of this function calls):

```
div9(15);
div9(81);
```

Submit screenshots of `a0` after each of these function calls, and submit your code that includes your function (`div9`) and the the two function calls above (in assembly).

Hint: Remember to use breakpoints in the Venus RISC-V simulator by clicking on a line of assembly code in the Simulator tab. Then click Run, and the program will run until it reaches each line set as a breakpoint. This makes it so that you don't have to tediously step through code that you already know works.

Program 2: Bubble sort
This is very similar to the program that you wrote in Lab 10 but this time you will write it as a function. Write a bubble sort function that sorts a variable-sized array called `sortarray` from smallest to largest. The function prototype looks as follows:

```
void bubbleSort(int sortarray[], int size);
```

The `int sortarray[]` argument gives the address of `sortarray[0]`, i.e., the base address of `sortarray`. The `size` argument gives the number of elements in the array. For example, if `sortarray` has 15 elements, `size` is 15.

Name your text file **bubblesortFunction.s**.

Test your function: Write initialization code (and add it to bubblesortFunction.s) that stores the following values to a 15-element array starting at address 0x400:

```
sortarray[15]={-15,42,73,19,-8,24,16,-2,99,-78,-21,23,-88,49,-101};
```

Then call the `bubbleSort` function using `sortarray`'s base address and a size of 15. Submit a screenshot of memory contents from 0x400 to 0x438 after the `bubbleSort` function completes.

Test your `bubbleSort` function with other arrays and array sizes as well, but you only need to submit the results of the above call.

Program 3: Greatest Common Divisor
Write a RISC-V assembly function that finds the *greatest common divisor* of two unsigned input arguments, *a* and *b*, according to the Euclidean algorithm. Here is some additional information about the Euclidean algorithm: https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/the-euclidean-algorithm. You can also simply google "Euclidean algorithm".

The function prototype looks like this:

```
unsigned int gcd(unsigned int a, unsigned int b);
```

The function should return the greatest common denominator. For example,

- A call of `gcd(25, 15)` should return 5.
- A call of `gcd(64, 96)` should return 32.
- A call of `gcd(71, 9)` should return 1.

To complete this exercise:

- First sketch high-level code (it can be pseudocode) of your algorithm. Name your text file **gcd.c**. You do not need to run or test this pseudocode.
- Write the `gcd` function. Use at least `s0`, a preserved register, within your function and correctly deal with saving/restoring registers on the stack. Name your text file **gcd.s**.

Test your function: Test your function by making the following calls:

```
gcd(25, 15)
gcd(64, 96)
```

Take a screenshot of `a0`, the return register, after each of the function calls. Make additional calls (at least five others) to your `gcd` function to test it, but you need only include the results of the two function calls above in your submission.

2. Recursive Function Call Example

Recursive functions call themself. These functions tend to confuse people, especially when they encounter them for the first time. To help counter this confusion, an effective strategy is to act as if the function is calling a separate function (not itself) at first.

Consider the following recursive function (also given in the textbook):

```
int factorial(int n) {
  if (n <= 1)
    return 1;
  else
    return (n * factorial(n - 1));
}
```

This function performs the factorial operation. For example, if n (the input argument) were 5, the function would return 120, because 5! = 5*4*3*2*1 = 120. So, consider what happens when factorial is called with the argument n = 5:

1. The else condition is true, so the function returns: (5 * factorial(4))

The tricky part is, we don't yet know what factorial(4) is, so we continue with the recursive function calls until the program reaches a stopping point (i.e., when `n` = 1). So,

2. factorial(4) is called: the else condition is true, so the function returns: (4 * factorial(3))
3. factorial(3) is called: the else condition is true, so the function returns: (3 * factorial(2))
4. factorial(2) is called: the else condition is true, so the function returns: (2 * factorial(1))
5. factorial(1) is called: the if condition is true, so the function returns: (1)

So, now that the stopping point is reached, the program rolls back up the recursive calls and can calculate the return values:

5. factorial(1) is called: the if condition is true, so the function returns: (1)
4. factorial(2) is called: the else condition is true, so the function returns: (2*factorial(1)) = 2*1 = 2
3. factorial(3) is called: the else condition is true, so the function returns: (3*factorial(2)) = 3*2 = 6
2. factorial(4) is called: the else condition is true, so the function returns: (4*factorial(3)) = 4*6 = 24
1. The else condition is true, so the function returns: (5*factorial(4)) = 5*24= 120

So, the return value of `factorial(5)` is 120. Note that recursive functions must have a stopping condition – that is, a point when recursion stops. Otherwise, the program will never complete.

We convert the recursive function call to RISC-V assembly, first, by acting as if the call is to another function. After this first pass, we make a second look of the code to account for any unintended effects of recursion by using the stack.

So, step 1 of converting high-level recursive function call into RISC-V assembly (ignoring the stack and acting as if the call to factorial is to another function (instead of itself):

```
// factorial in C

int factorial(int n) {

  if (n <= 1)
    return 1;

  else
    return (n * factorial(n - 1));
}
```

```
# factorial in RISC-V assembly

factorial:
  addi t0, zero, 1   # temporary = 1
  bgt  a0, t0, else  # if n>1, go to else
  addi a0, zero, 1   # otherwise, return 1
  jr   ra            # return
else:
  addi a0, a0, -1    # n = n-1
  jal  factorial     # recursive call: factorial(n-1)
  mul  a0, ?, a0     # after call to factorial(n-1) returns,
                     # return: n * factorial(n-1)
                     # problem: n (a0) was overwritten, so
                     # we must save n on stack before the
                     # call
  jr   ra            # return
```

Now we take another pass at the code, accounting for the stack and any issues with either calling another function or recursion. In this case, the issues are: `ra` (a preserved register) is overwritten, and the function needs to use `n` after the call to `factorial`). So, step 2:

```
// factorial in C

int factorial(int n) {



  if (n <= 1)
    return 1;




  else
    return (n * factorial(n - 1));
}
```

```
# factorial in RISC-V assembly

factorial:
  addi sp, sp, -8    # make room for a0, ra
  sw   a0, 4(sp)
  sw   ra, 0(sp)
  addi t0, zero, 1   # temporary = 1
  bgt  a0, t0, else  # if n>1, go to else
  addi a0, zero, 1   # otherwise, return 1
  addi sp, sp, 8     # restore sp
  jr   ra            # return
else:
  addi a0, a0, -1    # n = n-1
  jal  factorial     # recursive call: factorial(n-1)
  lw   t1, 4(sp)     # restore n into t1
```

	`lw   ra, 0(sp)      # restore ra`
	`addi sp, sp, 8       # restore sp`
	`mul  a0, t1, a0      # a0 = n*factorial(n-1)`
	`jr   ra              # return`

Notice that we used `t1`, a temporary register to restore `n` (i.e., originally `a0`) into from the stack. Temporary registers may be overwritten without having to restore them before the function returns.

You can (and should) also simulate / walk through your RISC-V assembly code to make sure it functions as you expect. This is similar to what we did with the high-level code above.

3. Write Recursive Function Call

Now, write a recursive function that performs the following algorithm:

Consider *N* people in a room. Suppose each person can greet each other only once. You are to calculate how many greetings occur.

For example, suppose 4 people are in the room (i.e., *N* = 4). The following occurs:

Person 4: Greets 3 people (persons, 3, 2, and 1).
Person 3: Greets 2 people (persons 2 and 1). Note: this person already greeted person 4.
Person 2: Greets 1 person (person 1). Note: this person already greeted persons 3 and 4.
Person 1: Greets no additional people (has already greeted everyone).

So, when *N* = 4, the number of people greeted is 3 + 2 + 1 = 6.

First, write a recursive function in C (or other high-level programming language) that implements the greeting algorithm above. Name the file **greet.c**. The function prototype in C is as follows:

```
unsigned int greet(unsigned int n);
```

The input argument `n` is the number of people in the room, and the function should calculate and return how many greetings occurred. While several methods exist for solving this problem, your function must use recursion (in a meaningful way).

You need not compile or test your high-level code, but test do test it by hand – as we showed for the factorial function.

Now, convert your function from Part 2 to RISC-V assembly. Name the program **greet.s**. Be sure to handle the stack, including saving and restoring registers to/from the stack, correctly.

4. Simulate Recursive Function Call

Run and simulate your RISC-V assembly version of the `greet` function by copy/pasting your program code into the Venus simulator and simulating your program. You may use additional tests, but also test your program by writing RISC-V assembly code that makes the following two function calls:

```
greet(7);
greet(10);
```

Show the contents of `a0` (using a screenshot in the Venus simulator) after each of the previous function calls returns.

What to Turn In

1. Please indicate how many hours you spent on this lab. This will be helpful for calibrating the workload for next time the course is taught.
2. Your code: div9Function.s, bubblesortFunction.s, gcd.s, greet.c, and greet.s.
3. Screenshots of `a0` from the Venus simulator after making the following function calls:
   ```
   div9(15);
   div9(81);
   gcd(25, 15);
   gcd(64, 96);
   gcd(71, 9);
   greet(7);
   greet(10);
   ```
4. A screenshot of memory addresses 0x400 - 0x438 when `bubblesort()` is called with the following 15-element array, `sortarray`, as an argument, with the array's base address being at 0x400. Be sure to include code that initializes the array before the function call.

   ```
   sortarray[15]={-15,42,73,19,-8,24,16,-2,99,-78,-21,23,-88,49,-101};
   ```

Please indicate any bugs you found in this lab manual, or any suggestions you would have to improve the lab.