CSC 3210 Computer Organization and programming

LAB 7

Macros and Subroutines

INSTRUCTIONS FOR LAB 7

Tasks to be done in this lab:

- Define macros to make programming in assembly language easier
- Define subroutines to make programming in assembly language easier
- Communicate values to macros and/or subroutines
- Learn how a reference to a value at a memory location (e.g. "la a1, sum") is different from a reference to an immediate value (e.g. li).

ASSIGNMENT FOR LAB 7

- There are 3 parts in this lab.
- Execute all three of them.
- Answer the questions that are mentioned in bold.

INTRODUCTION

- Have you noticed how we often need to repeat the same code again and again?
- For example, we might want to print something out. To print a value, we might print a string as well as the data value, both defined in the data section.

```
.data
helloworld:      .string "Hello World\n"
int1: .word 0x23
sum:  .word 0
```

- Then in the code section, we refer to the string and the value to print as follows.

```
# Move the sum value to s1
 la    x7, sum
 lw    s1, 0(x7)
 # Now print the sum out
 li    a0, 1       # print an integer
 mv    a1, s1      # int value to print
 ecall
```

INTRODUCTION

- Thus to print something out, we use a set of commands like those above. Then we might want to print another value, so we repeat the same code with a minor variation.
- Later, we print a third value, and repeat the same code again with a minor variation.
- **Isn't there a way to make this easier?**
- There are a couple of ways, and that is the subject of this lab. First, we will examine macros in part1. Then we will define and use a subroutine in part 2.
- By the way, the value "0x23" in the line "int1: .word 0x23" specifies that "int1" should have the value hexadecimal 23.
- As a side note, so far we have used "j loop" at the end of the code, to jump back to the beginning and execute it again. We can also end the program. The following commands exit the program.

```
li    a0, 10
ecall
```

INTRODUCTION

- A variation of the commands to exit the program is the following.
- It exits the program and returns a value.
- Shell scripts and C programs do this, and it is a good way to signal to the caller that the program exited normally (0) or not (some other value).

```
li    a0, 17
li    a1, 0    # 0 for everything is OK
ecall
```

- With one of these exit calls as the last part of the program, you can now press the "Continue" button to run the entire program at once.

PART-1

- **This part will use RARS (described below).**
- A macro is a kind of short-hand notation that the assembler (technically, the pre-processor) will process. This exists in higher level languages, too, such as the "#define" directive in C. It works like a smart find-and-replace. When the macro is found, it is "expanded" into whatever the programmer indicated. This is perhaps best explained with an example.
- Consider this example.

```
.macro print_NL
   li   a0, 11
   li   a1, 10
   ecall
.endm
```

- The ".macro" and ".endm" delineate where this macro begins and ends. The macro has the name "print_NL", and no arguments. If you examine the code within the macro, you should recognize it as the commands that we have used to print a newline character. Some macros allow arguments such as "%x" or "%1".
- However, the simulator that we use does not appear to support macros. Perhaps this will change in the future; the above example is based on the RISC-V Assembly Programmer's Manual..

PART-1

- You may notice that sources for RISC-V assembly may have slightly different syntaxes and details. One may say to use .asciz in place of .string, or to use a7 instead of a1 for the ecall argument.
- Yet another RISC-V simulator, called the RISC-V Assembler and Runtime Simulator (RARS), implements macros with a slightly different syntax.
- The next 5 lines are modified from an example at robertwinkler.com. It is licensed under the Creative Commons BY-NC-SA 4.0. You will likely find it helpful for the lab portion of our course.

```
.macro print_string(%x)
   li    a7, 4
   la    a0, %x
   ecall
.end_macro
```

- See Robert Winkler, RISC-V Assembly Programmming, version 1.0.3, 2024-11-30, available at the above link.
- The only detail left to notice is that the macro contains "%x", which correspond to the argument.
- To use the macro, we put the macro's name, followed by the label of the format string and the memory location to print.

```
   print_string(helloworld)
```

PART-1

- What the assembler will do is "expand" this to the code defined in the macro, substituting "helloworld" for "%x". Instead of typing out several lines to print the string, we just specify the one line with the macro name.
- Think of it like a global find-and-replace operation. Whether you use the macro or type out the equivalent lines, the result is the same.
- If we have several strings to print, our code might look like this.

  ```
  print_string(string1)
  print_string(string2)
  print_string(string3)
  ```

- It should be easy to see that working with macros can save the programmer a lot of time and energy. It can also make the code easier to debug, since it contains a regular pattern. That is, imagine if we do not use a macro and instead type the commands for printing a string several times.
- **And suppose that there is a subtle mistake in one of the commands, like using a1 where we need a7. Would you be able to spot the difference?**

PART-1

For this part of the lab, you will examine some RISC-V code with a macro defined, and simulate it with RARS on the SNOWBALL server.

- Log in to your SNOWBALL account
- Copy the file **"/home/mweeks/macro_example.s"** to your account.
- You can do this by entering **cp /home/mweeks/macro_example.s ~/Lab7_pt1.s**, which will copy it to your home directory. Notice that ".s" is lower-case, which appears to be the convention for this simulator. It should work with either ".s" or ".S", however.
- Simulate this. To make this easy for you, RARS is already under the "/home/mweeks" directory.
  java -jar /home/mweeks/rars1_6.jar Lab7_pt1.s
- Enter **echo $?** at the prompt. This should return the value 42, which is what the example says to return on exit.
- Edit the program to have it print "Types of apples", and then the names of 3 different types of apples, each preceded by " " (as a character), and each name on its own line. You can encode newline characters in the strings, but do not include the preceding space in the strings.
- Show that this works.

PART-2

- **This part will use VSCode/Venus.**
- Just as macros may rely on the pre-processor, we can use the pre-processor for other things. You may have noticed that when using "ecall" we can print different things, but must remember the correct number to put into a0. To help with this, we can define a value associated with a name, and the pre-processor replaces each instance of the name with the value.

```
#define  PRINT_STR  4

#  ... more code

li    a0, PRINT_STR     # print a string
la    a1, str2          # load address of str2
ecall
```

- Now, the code is slightly more readable. It will be easier for us to remember "PRINT_STR" instead of "4". Using all-caps is a convention for a value that cannot be changed by the program, such as a constant.

PART-2

- Notice how the "#define" is an exception to the way that the "#" character is normally processed. The pre-processor uses "#" in a different way, and recognizes "#define" as something that it will handle.
- Copy your code from the last lab, and replace any instances of printing a string with PRINT_STR as defined above.
- Likewise, define "PRINT_INT", "PRINT_CHAR", "NEWLINE", and "SPACE", with the corresponding values.
- Call the result "lab7_pt2.S".
- Also, include the exit with a return code instructions from above. Make the return code whatever the last two digits of your student ID are.
- Show that this works as expected.

PART-3

- **This part will use VSCode/Venus.**
- Another option for making repetitive code easier to use is the subroutine. Like a macro, a subroutine defines code that you can use again and again.
- However, a subroutine is a function, similar to a method in OOP. When you want to use a subroutine, you issue a call command to it, like the following.

```
call mysubroutine
```

- Notice that this is similar to "ecall", because "ecall" is a like a special version of "call". When we use "call", we have to specify what function (subroutine) to call. The subroutine must be defined in the code section.
- A "main" function in the C language includes a "return" instruction as the last command. This returns control to whatever called the program, such as the OS shell. A subroutine is no different: it must end with a return instruction.
- When the computer calls the subroutine, it must remember where to come back to. In some assembly languages, it does this by pushing the current Program Counter (PC) on the stack, then setting the PC to the subroutine's address.

PART-3

- When the CPU gets to the return instruction, it pops the address from the stack and puts that in the PC. RISC-V is a bit different: it uses a register (x1) to remember where to return.
- The call command is a pseudo-instruction for jal x1, offset, while ret is a pseudo-instruction for jalr x0, 0(x1). As a programmer, you will likely see that "call" and "ret" are easy to understand and remember.
- We can create a subroutine for printing an integer, specifically the one stored at "sum", and call it like this.

  call print_int

- By the way, do not expect a subroutine to preserve your a and t register values. But this raises a question: how does it know what value to print?
- We would have to communicate the value somehow. A possible solution is to have a specific data value, defined with a label in the data section, then move the value to that location before calling "print_int". While this would work, you (the programmer) would need to remember which label to use for the move.
- Another solution is to use a register, such as a0. Move the value to print to a0, then call the subroutine. This helps with efficiency, especially if the value needs to be in a register in the subroutine.

PART-3

- The subroutine should be located after the main function's return. The program should look like this:

```
# ... code goes here
# Put value to print in register a0, if it is not there already
        la    t1, sum
        lw    a0, 0(t1)
        call  print_int
        # ... more code
        # exit
        li     a0, 17
        li     a1, 0          # 0 for everything is OK
        ecall
    print_int:
        # Instructions to print an int value go here
        mv    a1, a0       # int value to print
        li    a0, 1          # print an integer
        ecall
        ret
```

- Copy your code from the previous lab (or part 1), and replace any instances of printing a string (with ecall) with a subroutine as defined above. Call the result "lab7_pt3.S". As with all labs, show the code, and show that it runs.

QUESTIONS

1. **In Part 1, you may have observed that the "print_char" macro does this:**

```
li    a7, 11
la    a1, %x
lb    a0, 0(a1)
ecall
```

   Why not use this instead? Explain.

```
li    a7, 11
la    a0, %x
ecall
```

2. **In Part 1, how many total lines are in the program? How many lines would there be if you did not use a macro? Explain how you get your answers.**
3. **Why do you think the RARS and VSCode/Venus environments use different registers and values for ecall?**
4. **With a subroutine call as part of the program, how do the "step over" and "step into" buttons (for Venus) behave differently?**

QUESTIONS

5. In Part 3, the last commands are:

   ```
   ecall
   ret
   ```

   Why do we need "ret" when we use "ecall" to exit the program? Explain.

6. Why does a subroutine need a ret instruction, but a macro does not?

7. When you call a subroutine, how do you know if the registers will have the same values after it returns?

8. Suppose that it is important that your program remembers the value in register a0 after a subroutine call. What can you do outside of the subroutine to remember a0's value?

9. Suppose that you write a subroutine that other people might use. Your subroutine uses (i.e. changes) the a1 register. When someone else uses your subroutine, they may have something important in a1. What can you do inside of the subroutine so that a1's value is the same upon return as it was when the subroutine started?

QUESTIONS

10. Does using a macro make a difference for the problem of remembering register values?
11. The macro "print_char(char1)" works fine when you invoke it, where "char1" is defined in the data section. Suppose that you have the value in register a0 already, but you do not have it in memory, and use "print_char". Does it work? Why or why not?
12. Suppose that you have the value in register a0 already, but you do not have it in memory. If you use a subroutine call such as "call print_int", does it work? Why or why not?
13. Which is approach (macro versus subroutine) is likely to generate a larger executable program, and why?

IMPORTANT NOTE:

Remember that we will grade your lab report so it is vital to turn that in. The other files (your code, a text version of any log file, etc.) are to document your work in case we need more information.