**Score: \_\_\_\_\_**

**LA5 – Calling Functions**

**Activities**

COMP256 – Computing Abstractions

Dickinson College

Spring 2023

Prof. Grant Braught

**Name:**

Today’s class extended the collection of high-level language features that we now know can be implemented in assembly language, and thus in machine language via an assembler. To the basic operations, standard control structures and arrays, we have added function calling. The ideas and techniques introduced today showed how we can call a function in assembly language and have it return a value to the calling code. This is accomplished with a combination of assembly instructions and special purpose registers. The CALL instruction jumps to the function and saves the return address. The function uses R14 to return a value to the calling code and the RET instruction uses the saved return address to return from the call.

We also learned about stacks and how they are used to pass arguments to a function. We saw the how to use the PUSH instruction to add arguments to the stack before calling the function and how to use the POP instructions to remove them after the function returns. For today, we have assumed that functions will be able to use the parameters that we put on the stack. The next topic will provide the details on how that works. These activities will provide you with experience making function calls, returning a value from a function, and passing arguments to a function.

Some of the activities at the end of this homework will have you begin to think about some of the challenges that can arise in implementing functions. Those exercises lay the foundation for the next class where we will finish off assembly language by seeing how to implement functions that use arguments and functions that call other functions.

**Assembly Instruction Reference:**

We have now seen all of the assembly language instructions for our assembler. A full reference to them is linked on the course schedule and you should refer to it as you work though the activities.

**Reserved Registers:**

Up until now we had been working exclusively with R0-R11 as the general-purpose registers. It was mentioned earlier that R12-R15 are reserved registers that are used for specific purposes in the machine. Today we saw the use of three of those reserved special purpose registers R12, R13 and R14.

🔑 1. For each of these registers, give a sentence or two of explanation of their purpose based on the examples from today’s class:

a. R12

b. R13

c. R14

**Calling a Function:**

🔑 2. Consider the following high-level language program:

main() {

n = nine()

print n;

}

int nine() {

p = 16;

r = 7;

s = p – r;

return s;

}

a. Give a translation of the nine function into assembly language. Use the label NINE for the first instruction of your function. Be sure to end the function using the RET instruction and to return the value of s in the correct register.

b. Give an assembly language translation of the main program that calls your nine function from part a. Notice that this function prints out the value that the function returns. Hint: Follow the similar example in the class slides.

c. Your solution to part b should contain a CALL instruction. If that CALL instruction were stored at memory address 132, what value will be placed into R12 by the CALL instruction when it is executed? Hint: Remember what the purpose of the PC (program counter) is and examine the meaning of the CALL instruction in the assembly language reference.

d. Explain in a few sentences when and how the value that is placed into R12 by the CALL instruction used. Hint: It is used at the end of the function.

**The Stack:**

The *stack* is a section of main memory that is set aside to be used to pass arguments to functions (and some other things we will see in the next class).

🔑 3. What directive allocates the main memory space for the stack?

🔑 4. Write a directive that allocates enough stack space for 512 bytes.

🔑 5. Write a directive that allocates enough stack space for 100 **integer values**. Hint: How many bytes are used to represent an integer on our machine?

🔑 6. Consider the assembly program that use a stack:

.stacksize 12 \* Enough for 3 ints.

LOAD R0 #5

LOAD R1 #10

LOAD R2 #20

PUSH R0

PUSH R1 \* LINE A

PUSH R2

POP R4 \* LINE B

POP R3

POP R5 \* LINE C

HALT

a. What values will be in R3, R4 and R5 when this program completes?

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | **Register** | **Value** |  |
|  | R3 |  |  |
|  | R4 |  |  |
|  | R5 |  |  |
|  |  |  |  |

b. Assume that R13, (i.e. the stack pointer), is initialized to the address 200 when this program begins. What value (i.e. address) will be in R13 at each of the points indicated in the table below? Hint: Trace the program like the example we did in class.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | **Point** | **R13 Value** |  |
|  | Just before LINE A executes |  |  |
|  | Just after LINE A executes |  |  |
|  | Just before LINE B executes |  |  |
|  | Just after LINE B executes |  |  |
|  | Just after LINE C executes |  |  |
|  |  |  |  |

**Passing arguments to a Function on the Stack:**

The high-level language function calling mechanism is an abstraction. For example, if we have the sum function from the class slides with the signature:

int sum(int a, int b)

Then we know we can call this function in a HLL by providing values for the parameters and saving the return value as shown here:

Read a;

Read b;

c = sum(a,b);

It is not necessary for us to know the internal details of the how the sum function works. We need only know that we have to provide 2 values as arguments and that the function will return the result that can be saved for later use.

The same is true in assembly language. We need to know what parameters the function requires so that we can push them onto the stack before we call it. We need to know that the function will return the value to us in R14. And we need to remember to remove the arguments from the stack when the function call is complete.

For example, the sum function can be called as we did in class:

LOAD R0 STDIN \* x

LOAD R1 STDIN \* y

PUSH R0 \* Pass x as a

PUSH R1 \* Pass y as b

CALL SUM

POP R15 \* Remove y from stack

POP R15 \* Remove x from stack

MOV R3 R14 \* get result.

🔑 7. Assume that an assembly language program begins as follows:

.stacksize 100

LOAD R3 STDIN \* x

LOAD R4 STDIN \* y

LOAD R5 STDIN \* z

**\* Your code goes here.**

a. Consider a function diff that computes the difference between its two parameters. This function might have the high-level language signature shown below:

int diff(int p, int q) // Compute p - q

Assume that an assembly language implementation of the diff function is given to you and that the label DIFF is the address of the first instruction in the function.

Give the assembly language statements that you would add to the above program to call the DIFF function passing x and y to the function as arguments and saving the return value in R8. Be sure to also include the POP statements that clean up the stack.

b. Consider a function flipSign that changes the sign of its parameter. For example flipSign(-7) would return 7 and flipSign(7) would return -7. This function might have the high-level language signature shown below:

int flipSign(int n) // change the sign of n

Assume that your are given the assembly language implementation of flipSign and the label FLIP is the address of the first instruction in the function.

Give the assembly language statements that you would add to the above program to call the flipSign function passing z to the function as an argument and saving the return value in R7. Be sure to also include the POP statements that clean up the stack.

c. Consider a function add3 that computes the sum of its three parameters. This function might have the high-level language signature shown below:

int add3(int a, int b, int c) // Compute a + b + c

Assume that you are given an assembly language implementation of add3 and that the label ADD3 is defined and is the address of the first instruction in the function.

Give the assembly language statements that you would add to the above program to call the ADD3 function passing x, y and z to the function as arguments and saving the return value in R9. Be sure to also include the POP statements that clean up the stack.

**Why Pass Arguments on the Stack?**

At first thought, it might seem like extra work to push the arguments to a function onto the stack and that there are less complicated solutions that would work just as well. The exercise in this section explores one of those possible solutions and highlights a reason why we pass parameters using a stack. The point is twofold, first to help us better understand function calling and second to motivate the need for the more complicated stack based approach for passing parameters.

One seemingly easier approach to parameter passing would be for the function to require that parameters be passed by placing argument values into specific registers (e.g. To call ADD3 put the arguments into R3, R4 and R5 and CALL ADD3). That would work, but it requires that every programmer to know which registers are used by every function that they call. That is not a very good abstraction!

8. Using the stack provides a better abstraction. To emphasize this point, write code that will use ADD3 to add the values in R9, R2 and R6.

9. Describe in a few sentences how using the stack to pass parameters to a function creates a better abstraction than an approach that would pass parameters through specific registers.

**Some Complications with Functions:**

We have the basics of function calling worked out. We push the arguments, call the function, clean up the stack (i.e. POP the arguments) and get the return value from R14. This works well, however there are some complications that can arise that we will need to deal with. The exercises in this section illustrate those complications. The purpose here, is to become aware of these complications. This awareness will motivate the techniques that we learn about in the next class that have been created to address them.

Function Side Effects:

When we call a function in a high-level language, we expect that it will not have any unexpected side effects (e.g. changing the value of a variable in the calling code).

10. Consider the high-level language program shown below:

main() {

a = 3;

n = two();

print n; // Line 1

print a; // Line 2

}

int two() {

a = 1;

b = a + a;

return a;

}

What output would you expect Line 1 and Line 2 to generate when this program is run? Hint: Remember the variable a in the function two is a local variable and is distinct from the variable a in main.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | **Line** | **Output** |  |
|  | Line1 |  |  |
|  | Line2 |  |  |
|  |  |  |  |

11. Now, consider the following assembly language translation of the high-level language program from question #10:

.stacksize 100

LOAD R1 #3 \* Use R1 as variable a

CALL TWO

STORE R14 STDOUT \* Line 1

STORE R1 STDOUT \* Line 2

HALT

TWO: LOAD R1 #1 \* Use R1 as local variable a

ADD R2 R1 R1 \* Use R2 as local variable b

MOV R2 R14 \* Return the result in R14.

RET

a. This program will produce the output:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Line** | **Output** |  |
|  | Line1 | **2** |  |
|  | Line2 | **1** |  |

Briefly explain why this implementation of TWO would give a different result than we expect based on our understanding of the HLL program in question #10?

b. One unsatisfactory solution to this complication would be to ensure that the MAIN program and the TWO function use different registers for their computations (i.e. that they do not both use R1). Why do you think that would that be an unsatisfactory solution? Hint: See question #9.

Nested Function Calls:

It is quite common for one function to make a call to another function, and for this to even go on further for several more calls to additional functions. This situation introduces another, more subtle, issue that we must address when implementing functions.

🏆 12. Consider the assembly language program below.

.stacksize 500

LOAD R0 #0

STORE R0 STDOUT \* Line 1

CALL AYE

STORE R0 STDOUT \* Line 5

HALT

AYE: LOAD R1 #1

STORE R1 STDOUT \* Line 2

CALL BEE

STORE R1 STDOUT \* Line 4

RET

BEE: LOAD R2 #2

STORE R2 STDOUT \* Line 3

RET

a. Without assembling and running this program, what output do you think would be generated by each of the indicated lines of this program?

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | **Line** | **Output** |  |
|  | Line 1 |  |  |
|  | Line 2 |  |  |
|  | Line 3 |  |  |
|  | Line 4 |  |  |
|  | Line 5 |  |  |
|  |  |  |  |

b. Assemble and run the above program. What output does it actually generate?

c. Which of the labeled lines (e.g. Line 1) is generating the repeated output that appears?

🏆 🏆 d. Briefly explain why this program behaves in the somewhat unexpected way that it does. Hint: What happens to the Return Address (R12) when the CALL BEE instruction is executed?

The issues raised in this section may seem particularly challenging! But, it turns out that having and using a stack for calling functions will help us address them all.

Optional: To help me improve and scope these activities for future semesters please consider providing the following feedback.

a. Approximately how much time did you spend on this activity outside of class time?

b. Please comment on any particular challenges you faced in completing this activity.