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

**Lab09 – Assembly Programming and Recursion**

COMP256 – Computing Abstractions

Dickinson College

Spring 2023

Prof. Grant Braught

**Name(s):**

**Introduction:**

There is a principle in computer design called “Hardware / Software Equivalence.” This principle states that once a sufficiently powerful machine is available any new operations that can be done in hardware can also be done in software and vice versa. This was evident in early machines where the machine hardware could add but not multiply. Multiplication was done using software routines that ran on the existing hardware of the time. Later, as the complexity of machines increased, the multiplication function was added to the hardware making it significantly faster. Computations using floating point numbers followed a similar path. Early PCs could only process integer operations in hardware and used software libraries for floating point operations. Of course, all general-purpose CPUs now include hardware support and corresponding machine language instructions for both multiplication and floating point arithmetic.

However, you may have noticed that the machine and assembly language that we’ve used in this course do not have a multiply instruction. In this lab, you will write a function in in assembly language that performs multiplication of non-negative integers. You will then use that function to write a recursive function that computes factorials. While these are relatively simple things to do in a high-level language like Java or Python, they are challenging in assembly language. Writing them will give you extra practice with assembly language programming and its function calling mechanisms. In addition, implementing a recursive function in assembly will give you some additional insight into how recursion works!

**Preliminaries:**

1. You will be using your Comp256Assembly container for this assignment. To get started:

* Start the Comp256Assembly container using Docker Desktop.
* Connect to the container using the TigerVNC Viewer.
* Use the terminal or File Manager (i.e. the file cabinet icon in the launcher at the bottom of the screen) to create a new directory named Lab07 within your home directory. You’ll save all of the files that you create for this lab into that directory.

There is nothing to turn in for this question. But you’ll need to have the Comp256Assembly container running to complete this lab.

**Using the .break Directive for Debugging:**

The .break directive is a debugging tool built into our assembler and machine simulator. When you run a program, if a .break directive is executed, the machine simulator will pause. You can then click Run again to resume the program. The execution will then continue either until the next .break is encountered or until a HALT instruction is executed.

2. Create the following program in a text file and save it into a file named BreakEx.asm in your Lab07 folder.

A: .word 2895

LOAD R0 A

.break

NOT R1 R0

.break

ADD R1 R1 #1

HALT

There is no answer required for this question. But be sure to name this program as indicated above. You will turn in all of your asm programs at the end of the lab.

3. Assemble and run the BreakEx.asm program. Then use it to answer the following questions that will help to illustrate how .break might be useful in debugging a program.

a. Before you click the Run button in the machine simulator, what base 10 value is stored in R1?

b. Click the Run button once. What instruction is displayed in the ASM text field in the machine simulator? Why?

c. What base 10 values are stored in R0 and R1 when the first .break is encountered?

d. Click the Run button a second time. What base 10 value is stored in R1 now?

e. Click the Run button a third time. What base 10 value is stored in R1 now?

4. Briefly explain how the .break instruction might be useful in debugging a program?

**Multiplication:**

Consider the following high-level language program that will multiply two non-negative integers:

main() {

read a;

read b;

c = mult(a,b)

print c;

}

mult(x,y) {

prod = 0;

for (int i=0; i<x; i++) {

prod = prod + y;

}

return prod;

}

5. Create a file named Mult.asm in your Lab07 directory and write an assembly language translation of the above program. Your implementation of main and mult must be implemented using the process described in class and as outlined on the following slide:



**Be sure to assemble and run your program in the machine simulator to test it.** You do not need to copy your code here. You will turn in all of your asm programs at the end of the lab.

6. Run and test your program. Briefly explain what you did to check that your program was computing the correct answer.

**Checking the mult Implementation:**

In addition to the testing you described in the prior question, it is important to also check that your mult function implements the function calling abstraction correctly. That is, you should be able to call mult from anywhere in a program without knowing its internal details (e.g. which general purpose registers it uses), and when you do call it, it should not have any unintended side effects (e.g. it should appear to not change any general purpose registers).

This section will ask you some questions to ensure that your mult function properly implements the function calling abstraction. Having a proper implementation of this abstraction will be essential to being able to complete the next section of the lab that uses mult to recursively compute a factorial.

7. A properly implemented function should be able to complete its computation based only on the values of its parameters. That is, it should not use any global variables (i.e. labels allocated with .word at the start of the program). Does the code in your mult function use any global variables? If no, skip this question. If yes, revise your mult function so that it no longer uses any global variables.

8. A function should also not have any unintended side-effect like changing the values in general purpose registers. In this question you will confirm that your mult function correctly saves and restores the registers that it uses.

a. List each general-purpose register (R0-R11) that is used by instructions within your mult function in the “Register” column in the table below. You’ll complete the other two columns in the following questions.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  | **Register** | **Value**  **Before CALL** | **Value**  **After CALL** |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

b. For each register that is used by mult add an immediate mode LOAD instruction to your main program after you PUSH the arguments and just before the CALL instruction. These LOAD instructions should place some known values into the registers. For example, if mult uses R2 then you might add the instruction LOAD R2 #22. You’ll check these after the call to mult to ensure that it properly saved and restored the values of the registers it uses.

c. Now you can use .break directives to ensure that your implementation of mult properly saves and restores the registers. Place a .break instruction immediately before and immediately after the CALL MULT instruction in your main program.

d. Click the Run button. Your program should run and stop at the .break just before the CALL. Record the value in each of the registers just before the CALL MULT instruction in the table above.

e. Click the Run button a second time. Your program should now run the mult function, return to main and stop at the .break just after the CALL. Record the value in each of the registers just after the CALL MULT instruction in the table above.

f. If mult properly saves and restores the values in the general purpose registers that it uses, the values of the registers in the table above should be the same before and after calling mult.

Does your mult function properly save and restore the registers that it uses? If yes, skip this question. If no, list the register(s) that were not being saved and restored properly and fix your code so that they are being saved and restored properly.

9. In the previous question you confirmed that your mult function correctly saves and restores the values in the general purpose registers that it uses.

Your mult function saved and restored these values using the stack. When properly implemented, the function calling abstraction ensures that a function manages the stack by ensuring that it is exactly the same after the call as it was before the call. This question will confirm that mult properly manages the stack.

As you know, the mult function should PUSH every register that it changes at the start and then POP those registers, and only those registers, just before RET. Thus, if mult properly manages the stack, the stack pointer should be the same just before and just after the call to mult.

a. Run your program with the .break statements again and complete the table below to check if mult properly manages the stack.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | **R13 (SP) Before CALL** | **R13 (SP) After CALL** |  |
|  |  |  |  |
|  |  |  |  |

b. Does your implementation of mult properly manage the stack? If yes, skip this question. If no, fix your implementation of mult and give the correct values in the table below.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | **R13 (SP) Before CALL** | **R13 (SP) After CALL** |  |
|  |  |  |  |
|  |  |  |  |

10. After completing the questions in this section you can be relatively sure that your mult function correctly implements the function calling abstraction. You can remove the .break directives and the LOAD instructions that you added for testing in question #8.

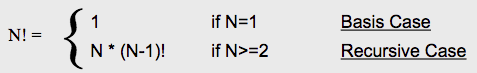
There is no answer required for this question. But your mult.asm file should not contain the .break instructions or the LOAD instructions added for testing in question #8.

**Recursive Factorials:**

Now that you have a working mult function, you’ll use it to compute factorials.

Recall that the factorial of a number N (usually written as N!) is the product of all of the values from N down to 1. For example, for N=5, we get 5! = 5 \* 4 \* 3 \* 2 \* 1 = 120.

There is also a nice recursive definition of factorial:



This definition essentially says that N! can be computes as N \* (N-1)!. For example, 5! = 5 \* 4!. So, if we can compute 4! we can then compute 5! simply by multiplying 4! by 5.

The following high-level language program uses the mult function to compute factorials using the recursive definition given above:

main() {

read a;

f = fact(a)

print f;

}

int fact(n) {

if (n == 1) {

return 1;

}

else {

t = fact(n-1);

t2 = mult(n,t);

return t2;

}

}

int mult(x,y) {

prod = 0;

for (int i=0; i<x; i++) {

prod = prod + y;

}

return prod;

}

11. Create a file named Fact.asm in your Lab07 directory and implement the above high-level language program in assembly language. You will need to copy your mult function from Mult.asm into Fact.asm so that fact can call it.

Note: Because this program will be making multiple recursive calls, each of which requires a new stack frame, it will require a larger stack. Your .stacksize directive should create a stack that is large enough to compute the value of 10! How big is that? A stack of 500 bytes should be sufficient.

tswm? (opposite of tldr… too short want more ;) … If you are curious where that 500 bytes came from, here is how it was estimated…. Each call to fact will require that its parameter (n) be pushed (4 bytes). fact will then push all of the registers that it uses to save and restore them. We can conservatively estimate that as 10 registers (40 bytes). The main program calls fact once, and then to compute 10!, fact will call itself 9 more times. So that is (1+9)\*(4+40) = 440 bytes. When fact calls mult to compute 2\*1 the stack will be as large as it gets. That call to mult requires that the 2 arguments (n,t) are pushed (8 bytes) and that mult PUSHes all of the registers that it needs to save and restore. Again, we’ll conservatively estimate that as 10 registers. So that call to mult will push another 8+40 = 48 bytes. That gives us 488 bytes total. Then just for convenience we’ll round that up to 500 bytes.

**Be sure to assemble and run your program in the machine simulator to test it.** You do not need to copy your code here. You will turn in all of your asm programs at the end of the lab.

If your program does not work correctly you can use .break directives to do some of the following things as a part of your debugging process:

* Test the base case:
  + Call fact with the argument 1 from main and check the result.
  + Ensure that the stack pointer and general-purpose registers are all restored after the call to fact.
* Test a small case:
  + Call fact with the argument 2 from main and check the result.
  + Ensure that the fact function is preserving and restoring the registers and the return address (R12) as is necessary. This is the most common error that occurs!
  + Check the value of n at the start of each call to fact (before the if) to be sure the calls are happening as you expect (i.e. n=2 then n=1).
  + Check the return value (t) from each call to fact in the else to be sure it is correct.
  + Check the return value (t2) from the call to mult in the else to be sure it is correct.

🏆 12. In your fact function it is possible to call mult in two different ways:

t2 = mult(n,t);

or

t2 = mult(t,n);

The difference is that the arguments to mult will be pushed to the stack in reverse order. Run your Fact.asm program and compute 7! several times each way. Which version of the call to mult results in a faster program? Briefly explain why?

**Faster Multiplication:**

**Optional Challenge:** Doing multiplication as repeated addition is straight forward. But there are much faster ways to perform multiplication using binary values. One such way is often called the “Russian Peasant Multiplication Algorithm”

* <https://iq.opengenus.org/russian-peasant-multiplication-algorithm/>

🏆 🏆 16. Make a copy of your Fact.asm program into a file named RPFact.asm and rewrite the mult function so that it performs multiplication using the Russian Peasant Multiplication Algorithm instead of repeated addition. Hint: Recall that the SHL and SHR assembly language instructions either multiply by 2 or do integer division by 2.

🏆 17. Does this algorithm improve the speed of the slower version of your fact function? Explain why in sufficient detail that a classmate would understand.

**Submitting the Lab:**

18. Submit this lab as follows:

a. Use the following commands in a terminal window in the container:

cd /home/student

tar -zcvf Lab07.tar.gz Lab07

b. Use the FireFox browser in the container to go to the course Moodle site and submit the Lab07.tar.gz file to the “Lab07 Code” assignment.

c. Convert this worksheet to a pdf and submit it to the “Lab07 Activity Sheet” assignment on Moodle.

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.