|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 🔑 **Essential** 🔑 | | | |  | 🏆 **Enhanced** 🏆 | | | |
|  |  |  |  |  |  |  |  |  |
| \_\_\_\_\_ | \_\_\_\_\_ | \_\_\_\_\_ | \_\_\_\_\_ |  | \_\_\_\_\_ | \_\_\_\_\_ | \_\_\_\_\_ | \_\_\_\_\_ |

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

**Lab09 – Assembly Programming and Recursion**

COMP256 – Computing Abstractions

Dickinson College

Spring 2022

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 integer multiplication. 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. From prior activities you should have a COMP256ASM folder containing the Assembler.jar and Machine.jar programs. If you no longer have that folder, recreate it and download the assembler and machine simulators using the following links:

* <https://github.com/dickinson-comp256/AsmMachine/raw/main/Assembler/bin/Assembler.jar>
* <https://github.com/dickinson-comp256/AsmMachine/raw/main/Machine/bin/Machine.jar>

There is nothing to turn in for this question. But you’ll need the assembler and the machine 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 COMP256ASM folder.

A: .word 2895

LOAD R0 A

.break

NOT R1 R0

.break

ADD R1 R1 #1

HALT

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 the program that you created in question #2 into the executable file BreakEx.ml. What command did you use to assemble the program?

🔑 4. Run the machine simulator with the executable program you just assembled. What command did you use to run the simulator?

🔑 5. The following questions 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?

6. 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 positive integers:

main () {

read x;

read y;

prod = 0;

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

prod = prod + y;

}

print prod;

}

🔑 7. Create a file named Mult1.asm in your COMP256ASM directory and write an assembly language translation of the above program.

**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.

🔑 8. Briefly describe how you tested your Mult1.asm program to ensure that it is correct.

**A Multiplication Function:**

Because multiplication might be used at many points in a program it would be better to encapsulate that functionality into a function as shown in the high-level code below:

main() {

read a;

read b;

c = mult(a,b)

print c;

}

int mult(x,y) {

prod = 0;

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

prod = prod + y;

}

}

🔑 9. Create a file named Mult2.asm in your COMP256ASM directory and adapt your code from question #7 to be the assembly language equivalent of the above program, including the function and the function call. You can refer back to classes 18 and 19 on Calling Functions and Implementing Functions for more information and for concrete examples.

**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.

**Checking the mult Implementation:**

In addition to the type of testing that you described in question #8, there is additional testing that you’ll need to do to ensure that the mult function is implemented correctly.

When a function is properly implemented we can easily use it as an abstraction. We should be able to call it from anywhere in a program without knowing its internal details, and when we do call it, it should not have any unintended side effects. This section will ask you some questions to ensure that your function is implemented properly. Having a proper implementation will be essential to being able to complete the next section of the lab that uses mult to recursively compute a factorial.

🔑 10. 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, indicate the global variable(s) that were used and revise your mult function so that it no longer uses them.

🔑 11. Recall that a function should not have any unintended side-effect like changing the values in general purpose registers. 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 columns in the following questions.

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

🔑 12. Let’s check to ensure that your implementation of mult properly saves and restores the registers that it uses so that they appear to be unchanged by its execution. Place a .break instruction immediately before and immediately after the call to MULT in your main program.

a. 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 before the CALL in the table in question #11.

b. Click the Run button a second time. Your program should now run and stop at the .break just after the CALL. Record the value in each of the registers before the CALL in the table in question #11.

c. Does mult properly save and restore the values in the general purpose registers that is 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.

🔑 13. In #11 and #12 you confirmed that your mult function correctly saves and restores the general purpose register values. To do this, your function uses the stack. If mult properly manages the stack it should pop every value that pushes, and no more. If it does so, the stack pointer just after mult returns should be exactly the same as just before it was called.

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** |  |
|  |  |  |  |
|  |  |  |  |

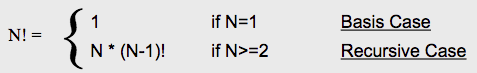
b. Does your implementation of mult properly manage the stack? If yes, skip this question. If no, fix your implementation of mult and give the correct values in the table below.

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

**Recursive Factorials:**

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

You should also recall that there is a nice recursive definition of factorial:



The following high-level language program uses the mult function from above to implement the recursive definition of factorial:

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;

}

🔑 14. Create a file named Fact.asm in your COMP256ASM directory and implement the above high-level language program in assembly language. You will need to copy your mult function from Mult2.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!

**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.
  + 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.
  + Ensure that the stack pointer and registers, including the return address, are the same before and after the recursive call to fact. The fact function not correctly preserving and restoring the registers or the return address is the most common error that occurs!
  + Check the return value (t2) from the call to mult in the else to be sure it is correct.

🏆 15. 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:**

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? Briefly explain why.

**Submitting the Lab:**

🔑 18. Export this document in pdf format. Copy the pdf version of this document and your Break.asm, Mult1.asm, Mult2.asm and Fact.asm files (and RPFact.asm if you did that part) into a COMP256-Lab9 directory and compress it to a zip file. Submit this zip file to the Lab09 assignment on the course 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.