NAMES: ____

COMP256 – Computing Abstractions
Dickinson College
LAB #6
Assembly Language Functions and Recursion

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 more transistors became available and 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 floating point arithmetic.

You may have noticed that the machine and assembly language that we've been using in this course does not have a multiply instruction. In this lab, you will create a software routine for multiplication add then use it to perform some computations. This will give you additional practice with assembly language programming and its function calling mechanisms.

Preliminaries:

1. If you do not already have one create a COMP256-Machine folder in you Documents folder and download the Assembler.jar and Machine.jar from the course homepage to that folder.

The .break Directive

The `.break` directive is a debugging tool. When a `.break` directive is executed, it causes the machine simulator to pause. This gives you the opportunity to examine the contents of the registers and memory to ensure that the program is operating correctly. Complete the following exercises to get a feel for how the `.break` directive works.

2. Create the following program in a text file and save it into a file named `BreakEx.asm` in your COMP256-Machine folder.

```
A:    .word 143
B:    .word 287

      LOAD R0 A
      LOAD R1 B

      .break

      OR R0 R0 R1

      .break

      AND R1 R1 R0

      HALT
```

3. Assemble the program that you created in question #2 into the executable file `BreakEx.exe`. What command did you use?

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

5. Before you click the Run button in the machine simulator, what values are stored in `R0` and `R1`?

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

7. What binary values are stored in `R0` and `R1` when the first `.break` is encountered?

8. Click the Run button again. What binary value is stored in R0 now?

9. Click the Run button again. What binary value is stored in R2 now?

Multiplication:

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

```
read x;
read y;

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

print prod;
```

10. Create a file named `Mult1.asm` in your lab directory and write an assembly language equivalent to the above program. Be sure to assemble and run your program in the machine simulator to test it.

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;
  }
}
```

11. Create a file named `Mult2.asm` in your lab directory and adapt your code from question #10 to be the assembly language equivalent to the above program. Recall that the calling code (i.e. `main` here) must:

- Create a stack.
- Push the parameter values onto the stack.
- Call the `mult` function.
- Retrieve the return value.

Also recall that the code for the called function (i.e. the `mult` function here) must:

- Preserve the return address and any other registers it modifies.
- Retrieve the parameter values.
- Perform the computation.
- Set the return value.
- Restore the registers and return address.
- Clean up the stack.
- Return to the calling code.

You should refer to the class slides on Function Calls for more information and a concrete example of what `main` and `mult` must do. Be sure to assemble and run your program in the machine simulator to test it.

12. What registers are used by your `main` program?

13. What registers are used by your `mult` function?

14. When writing functions, it is important to ensure that the stack and the registers are being maintained properly. The `.break` directive can help with this. Add `.break` directives to your program at the following points:

   a. Just before `main` pushes the parameter values onto the stack.
   b. Just after `main` pushes the parameter values onto the stack but before it calls `mult`.
   c. Just after `mult` starts and before it pushes any values.
   d. Just after `mult` preserves the return address and registers.
   e. Just after `mult` sets the return value, but before it restores the return address and registers.
   f. Just before `mult` returns.

15. Run your program and answer the following questions.

   I. What is the value of the stack pointer at point a?

   II. What is the value of the stack pointer at point b? Why that value?

   III. At point b, what values are in the registers used `main` and by `mult`?

   IV. What is the value of the stack pointer at point c?

V. What is the value of the stack pointer at point d? Why that value?

VI. At point e, what values are in the registers used `main` and by `mult`?

VII. At point f, what values are in the registers used `main` and by `mult`?

VIII. What is the value of the stack pointer at point f?

14. Are the values in the registers used `main` and by `mult` the same at point b and point f? If not, there is a problem in your program. Be sure that the `mult` function is preserving and restoring all of the registers that it modifies. If there was a problem, fix it and explain what the problem was.

15. Is the value of the stack pointer the same at points a and f? If not, there is a problem in your program. Be sure that the `mult` function is removing everything from the stack that it puts on the stack as well as the parameter values pushed by `main`. If there was a problem, fix it and explain what the problem was.

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:

$$N! = \begin{cases} 1 & \text{if } N=1 \qquad \underline{\text{Basis Case}} \\ N * (N-1)! & \text{if } N>=2 \qquad \underline{\text{Recursive Case}} \end{cases}$$

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;
  }
```

16. Create a file named `Fact.asm` in your lab directory and implement the above high-level language program in assembly language. The fact function will need to perform all of the same steps as any function (preserving / restoring the registers that it changes, etc…). Note that `fact` will also make calls to `fact` and to `mult`. You will need to copy your `mult` function from `Mult2.asm` into `Fact.asm` so `fact` can call it. You should also remove all of the `.break` directives from `mult`.

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!

17. Assemble and test your program. If your program does not work correctly you can use `.break` directives to do some of the following things:

- Test the base case:
  - Call `fact` with 1 from `main` and check the result.
  - Ensure that the stack pointer and registers are all restored after the call to `fact`.

- Test a small case:
  - Call `fact` with 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 are the same before and after the recursive call to `fact` (not correctly preserving and restoring the registers or the return address is probably the most common error).
  - Check the return value (`t2`) from the call to `mult` in the `else` to be sure it is correct.

18. Use your program to compute the value of 7! and give that value here.

19. In your `Fact.asm` program, change the call to `mult` from :

```
t2 = mult(n,t);
```

to

```
t2 = mult(t,n);
```

I.e. reverse the order in which you push the arguments. Compute the value of 7! Does the program run faster or slower? Briefly explain why?

Optional Faster Multiplication:

This section is optional and not counted as part of the lab score. But given what you just saw in question 19 it is pretty interesting!

20. As it turns out there are much faster ways to perform multiplication using binary values. Read about what is known as the "Russian Peasant Multiplication Algorithm"

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

Recall that the SHL and SHR assembly/machine language instructions either multiply by 2 or do integer division by 2. Thus, these instructions can be used to implement Russian Peasant Multiplication in software.

21. Make a copy of your `Fact.asm` program into a file named `RPFact.asm` and rewrite the `mult` function to use the Russian Peasant Multiplication Algorithm.

22. Does this algorithm improve the speed of the slower version of your `fact` function? Briefly explain why.

Submit the Code:

23. Compress your COMP256-Lab6 directory to a zip file and submit it to the Lab6 assignment on the course Moodle. One submission per lab pair is sufficient.