**ECE 260: Fundamentals of Computer Engineering – Lab #5  
MIPS Procedures**

**Name \_\_\_\_\_\_\_Le Quient Lewis\_\_\_\_ Lab Partner(s) \_\_\_\_\_\_\_\_N/A\_\_\_\_\_\_\_\_\_\_**

**1. Introduction**: This lab provides students with practice implementing MIPS procedures using the *jal* and *jr* instructions, modifying the stack, and utilizing registers **$a0**-**$a3** and **$v0**-**$v1**.

**2. Background**: In lecture, we discussed the conventions for implementing procedures in MIPS assembly. Two new instructions were introduced *jal* (Jump and Link) and *jr* (Jump Register). The *jal* instruction sets the program counter (PC) to the memory location of the called procedure and saves a return address back to the current location in the Return Address register (**$ra**). The *jr* instruction is an unconditional jump that jumps to any location specified within a register. Typically, the *jr* instruction is used in conjunction with the **$ra** register, but any register can be used.

To create a procedure there are two main roles, a CALLER and a CALLEE. When implementing and calling a procedure, both the caller and the callee must follow several conventions. These conventions are copied below from the lecture notes:

* The **CALLER** will:
  + Save all temp registers that it wants to survive subsequent procedure calls into its stack frame  
    (**$t0**-**$t9**, **$a0**-**$a3**, and **$v0**-**$v1**)
  + Pass the first 4 arguments to a CALLEE in registers **$a0**-**$a3** — save subsequent arguments on stack, in reverse order
  + Call CALLEE procedure, using a *jal* instruction which places the return address in register **$ra**
    - If this CALLER is also a CALLEE, you must save **$ra** before using *jal*
  + Access CALLEE procedure’s return values in registers **$v0**-**$v1** after CALLEE returns
  + Restore all temp registers that were saved prior to calling CALLEE
    - Be sure to grab return value from CALLEE prior to restoring any saved **$v0**-**$v1** from stack or you will overwrite the CALLEE’s return value
* If needed the **CALLEE** will:
  + Allocate a stack frame with space for saved registers, local variables, and spilled args
  + Save any “preserved” registers that it will use/overwrite: **$ra**, **$sp**, **$fp**, **$gp**, **$s0**-**$s7**
  + If CALLEE has local variables -or- needs access to args on the stack, save CALLER’s frame pointer and set **$fp** to 1st entry of CALLEE’s stack
  + EXECUTE procedure
  + Place return values in **$v0**-**$v1**
  + Restore saved registers including those that were preserved for CALLER
  + Restore **$sp** to its original value
  + Return to CALLER with *jr* **$ra**

Recall that, by convention, certain registers must be preserved across procedure calls. These registers are indicated in the MIPS Green Sheet and were presented in lecture. Further, recall that any registers that are not preserved cannot be assumed to hold the same value after a procedure returns, i.e. the value of **$t0** may be different after a procedure call.

This lab will help you further understand procedures and practice their implementation within MIPS.

**3. Writing Procedures**

**3.1 Writing Leaf Procedures**

1) In MARS, open the file called lab05\_part1.asm. In this file, you will need to write an assembly program to implement the following equation: . To do so, you will first need to implement the two parameterized procedures as described below.

2) Implement a *multiply* procedure that accepts two arguments (**$a0** and **$a1**), multiplies those arguments, and returns the result in **$v0**. Inside your *multiply* procedure, use the native *mul* instruction to multiply the input arguments. Note that the native *mul* instruction is not included on the MIPS Green Sheet Reference – see the Mars Help file. If you use the *mult* instruction (as opposed to the *mul* instruction) you will need to copy your result from the LO register using the *mflo* instruction. Make your life easy and just use the *mul* instruction.

3) Implement a *divide* procedure that accepts two arguments (**$a0** and **$a1**), divides **$a0** by **$a1**, and returns the result in **$v0**. Inside your *divide* procedure, use the *div* pseudo-instruction to perform the division operation. Note that the *div* pseudo-instruction is not included on the MIPS Green Sheet Reference – see the Mars Help file.

4) Under the “*ece260\_main*” label, implement the rest of the assembly program required to implement the equation . For the variables defined in the equation, store their values in the registers specified in Table 1. You can populate registers **$s0**, **$s1**, and **$s2** using the *li* instruction in the “*initialize*” section of the provided code. Of course, your implementation should use the *jal* instruction to call your newly created *multiply* and *divide* procedures.

Table 1: Register allocation for Part 3.1

|  |  |
| --- | --- |
| **Variable Name** | **Register** |
| X | **$s0** |
| Y | **$s1** |
| Z | **$s2** |
| F | **$s3** |

Using the initial values for X, Y, and Z in the Table 2, fill in the expected output values based on the equation, and the actual output value that your MIPS code produces.

Table 2: Expected and measured result for Problem 1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **X** | **Y** | **Z** | **Expected Value for F** | **Computed Value for F** |
| **3** | **4** | **5** | 0 | 0 |
| **15** | **4** | **5** | 2 | 2 |
| **100** | **5** | **3** | 20 | 20 |

**Test Your Code**

You should test your code with different initial values to ensure that it satisfies the requirements described above. Then, test your code using the supplied unit tests. To run the supplied unit tests, open a Cygwin shell and type the following:

cd h:  
cd ECE260  
cd Lab05\_MIPS\_Procedures  
make test\_part1

You will see output that indicates if your code passed or failed the included unit tests. If your code did NOT pass the units tests, address any errors and try running the unit tests again.

**3.2 Writing Nested Procedures**

Procedures that do not call other procedures are called “leaf” procedures. Procedures that do call other procedures are called “nested” procedures. Figure 1 shows the C code for a “*power*” procedure with a nested “*multiply*” procedure. In this part, you will need to write a MIPS assembly program to implement those procedures. The “*power*” procedure iteratively computes the value . You should use the “*multiply*” procedure that you already wrote for the previous part of this lab assignment. **Do NOT** call the *mul* instruction directly from within your “*power*” procedure.

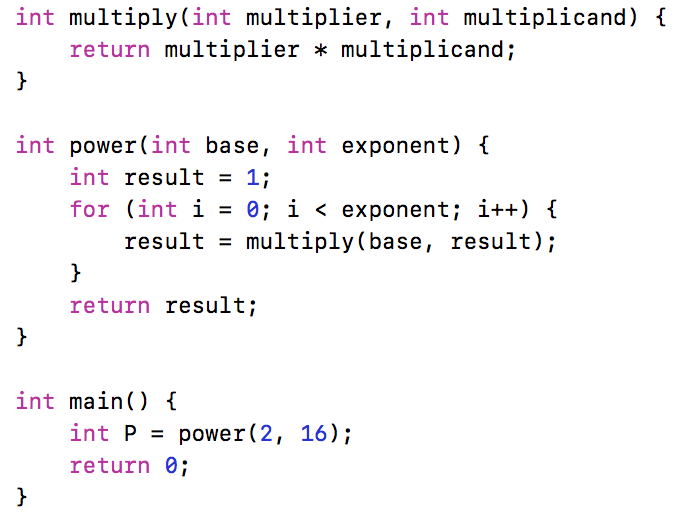
****

Figure 1: Nested Procedures

1) In MARS, open the file called lab05\_part2.asm. In this file, you will need to write an assembly program to implement the C code shown in Figure 1. You’ll see that registers **$s0** and **$s1** are populated with the *base* and *exponent* values respectively. Change these values as necessary to test your code.

2) Start by recreating your “*multiply*” procedure from the previous part of this lab. Recall that a procedure must put its return value into registers **$v0**-**$v1** to pass the value back to the caller. Implement the “*power*” procedure, doing all multiplication by calling the “*multiply*” procedure.

**Do NOT** use any **$tX** registers in your “*power*” procedure or in your “*multiply*” procedure. In your “*power*” procedure, use register **$s0** to store the *result* variable. Use register **$s1** to store the iteration variable *i*. You may need additional **$sX** registers. Be sure to preserve any and all **$sX** registers that you use.

3) After your “*power*” procedure returns to “*ece260\_main*”, move the final return value into the **$s2** register. The provided unit tests will look for your final value there.

**Test Your Code**

You should test your code with different initial values to ensure that it satisfies the requirements described above. Then, test your code using the supplied unit tests. To run the supplied unit tests, open a Cygwin shell and type the following:

cd h:  
cd ECE260  
cd Lab05\_MIPS\_Procedures  
make test\_part2

You will see output that indicates if your code passed or failed the included unit tests. If your code did NOT pass the units tests, address any errors and try running the unit tests again.

**3.3 Writing Nested Recursive Procedures**

A popular recursive procedure is the procedure to compute values of the Fibonacci sequence. If you’re not familiar with the Fibonacci sequence, [Google it](https://www.google.com/#q=Fibonacci+Sequence). The C code for a recursive Fibonacci procedure is shown in Figure 2. In this part, you will need to write a MIPS assembly program to implement that procedure.

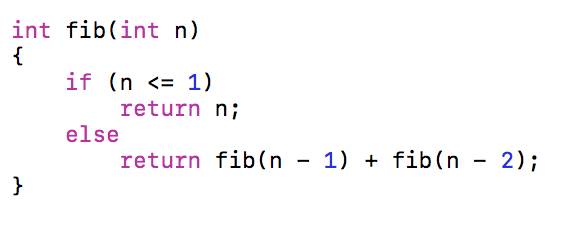


Figure 2: Recursive Fibonacci Procedure

1) In MARS, open the file called lab05\_part3.asm. In this file, you will need to write an assembly program to implement the Fibonacci procedure as shown in Figure 2. The Fibonacci procedure computes the nth number in the Fibonacci sequence. Several test values and their expected output values are shown in Table 3. Use those values to determine if your solution is correct. Remember, registers **$a0**-**$a3** are not preserved across function calls, requiring that you save registers **$a0**-**$a3** to the stack if they will be used after a procedure call. Also, recall that a procedure must put its return value into registers **$v0**-**$v1** to pass the value back to the caller.

2) After your “*fib*” procedure returns to “*ece260\_main*”, move the final return value into the **$s1** register. The provided unit tests will look for your final value there.

3) Use Table 3 to compare the output of your MIPS program to the expected output values for the Fibonacci procedure. A quick Google search will provide you with additional values in the Fibonacci sequence so you can do additional testing. Do not use input values for n that are greater than 40 – doing so will result in a computation that takes longer than you’re willing to wait.

Table 3: Expected Output Values from Fibonacci Procedure

|  |  |
| --- | --- |
| **n** | **Expected Output Value** |
| **0** | 0 |
| **1** | 1 |
| **5** | 5 |
| **11** | 89 |
| **20** | 6765 |

**Test Your Code**

You should test your code with different initial values to ensure that it satisfies the requirements described above. Then, test your code using the supplied unit tests. To run the supplied unit tests, open a Cygwin shell and type the following:

cd h:  
cd ECE260  
cd Lab05\_MIPS\_Procedures  
make test\_part3

You will see output that indicates if your code passed or failed the included unit tests. If your code did NOT pass the units tests, address any errors and try running the unit tests again.

**4. Submission**

When you have finished your lab, demo your program for your instructor. Write your answers to the above questions electronically in this document. To submit your lab assignment, make sure your files have all been saved (*including this file*). In a Cygwin window type the commands:

cd h:  
cd ECE260  
cd Lab05\_MIPS\_Procedures  
make submit

Enter your Marmoset username and password (which you should have received by email). Note that your password will not be echoed to the screen. Make sure that after you enter your username and password, you see a message indicating that the submission was successful. Log into [Marmoset](https://cs.ycp.edu/marmoset/login) via the web to check the files you submitted to ensure they are correct.

**DO NOT MANUALLY ZIP YOUR PROJECT AND SUBMIT IT TO MARMOSET.  
YOU MUST USE THE make submit COMMAND.**