**TPS (Think-Pair-Share) activity 1 Paired with the classmate sitting next to you and do the following tasks (20 minutes):**

**1. Load fib.s in MARS and study the code. The is the same program we worked on in Lab #6.**

**2. Recall that fib.s calculates the 13th Fibonacci number (n = 13). Now let’s make this program more generic so it will calculate the nth Fibonacci number, where n can be any number from a user input.**

We can utilize:

li $v0, 5 #User input

syscall

move $t3, $v0 #store the user input in $t3

To read user input and store it in the same memory location that n occupies.

**3. From Lab #6, we have learned how to print out statements in MIPS. Insert instructions in fib.s so that the program will print out “Please enter a number:” at the beginning of the program to prompt user for input.**

The following instruction will print out our desired message:

li $v0, 4 #Print out "Please enter the value of:"

la $a0, strNum

syscall

**4. In the program, $t3 was used to store the value of n. Now, let’s read in a user input and save that value into $t3. Do a search in the MARS documentations to find out how to use syscall to read an INTEGER from a user. Again, you must store the value into $t3.**

I already answered this in my question 2:

li $v0, 5 #User input

syscall

move $t3, $v0 #store the user input in $t3

Where our return register $v0 stores the value and saves it into our temporary register $t3.

**5. Since the program now reads a number from a user, do we need to declare n in the .data segment of the program? How about the la and lw instructions regarding n? Comment out those instructions so they won’t mess up your program.**

Since we are reading input from the user and directly storing it into our temporary register $t3, we do not have to declare our n in the .data segment of our program. As for the load address we don’t need it because it obtained the address of n and load immediate is useless since we are directly obtaining the input from our syscall.

**6. Assemble the program and test it with different numbers to see if it runs correctly. (You can use the original fib.s to verify your results.)**

(insert answer here)

**(Exercise) Recursive functions**

**In Lab #7, we’ve understood how register convention can help us manage registers in procedures. Let’s find out how we can follow register convention in recursive functions.**

**TPS (Think-Pair-Share) activity 2 Paired with the same classmate and answer the following questions (30 minutes):**

**1. Study recursion.c and trace the program. Without running the program, what will be the output if 5 is entered? Compile and run recursion.c in a terminal (or any IDE) and verify your answer.**

If we enter a 5 in recursion.c in my calculation I will obtain 29. When 5 is entered in the .c file we will obtain 29 as well.

**2. Load recursion.s in MARS. This is the MIPS version of recursion.c. Do not**

**assemble and run this program, as the program is incomplete. Study the MAIN function and discuss with your partner about what it does (compare it with the C version). A lot of instructions are missing, and we will fill them out in the following steps.**

As it currently stands, our MIPS main function only pushes space into the stack, then it jump and links the address from the main to recursion. Assuming that it eventually returns to main, it then has a jump function to end.

**3. Since the recursion.c prompts to a user for input, insert instructions in**

**recursion.s so the program will prompt the same statement to a user.**

We first declare in our .data segment of the code:

strInt: .asciiz "\n Please enter an integer: \n"

And then we print out our string with:

li $v0, 4 #print out the string

la $a0, strInt #load the address of our string into argument register($a0)

syscall #print

**4. Insert statements for the program to read in a value from a user. What register should we use to store that value? (Hint: you will use it as the argument for recursion function call.)**

Instruction to read the input of the user. Store into return register $v0.

li $v0, 5

syscall #user input.

move $a0, $v0

We store into our argument register, $a0, because that will be our argument for our recursion function.

**5. Next, the main function calls recursion with the correct input argument. After returning from recursion, we need to print out the returned value. What**

**register do we expect the returned value to be stored in? However, the syscall**

**for printing out a value is also using the same register. What can we do?**

We expect the return value to be store in our return register, namely $v0. What we can do in this situation is to store our correct return value into our next return register, $v1. In order to do this, we first need to store word(sw) the value of $v0 into the stack, when we return to main we can load word(lw) the value of $v0, into $v1, in that way freeing our first return register for the syscall function.

**6. Based on your answer from #5, insert the correction instructions to print out the returned value before jumping to the end of program.**

addi $sp, $sp, -4 #create space in the stack

sw $v0, 4($sp) #push The value of $v0 into the stack

lw $t0, 4($sp) #pop the top of the stack into $t0

li $v0, 1 #instruction for integer print function

add $a0, $t0,$zero #use our $t0 for the printable

syscall #print

**7. Now, let’s complete the recursion function. The stack pointer was moved to**

**create extra storage for the function. How many integer values are reserved in**

**this storage? What is the first thing to be stored in this stack frame? Insert a**

**statement to accomplish this.**

There are three integer values reserved in storage. The first thing to store would be our return address register($ra).

**8. Based on the branch statement under label recursion, update the returning**

**value. Again, you must use the correct register to store the returning value.**

li $v0, 1 #instance where the value is -1 so we return 1

**9. Based on the branch statement under label not\_minus\_one, update the**

**returning value. Again, you must use the correct register to store the returning**

**Value.**

li $v0, 3 #instance where we have a zero and we return 3

**10. When the input argument is not 0 or -1, the program will call recursion 2**

**times. This happens in the code under label not\_zero. Why do we need to save $a0 into the stack?**

To prepare for the next value and not lose our previous values.

**11. Insert a statement to update the input argument for the next recursion call.**

addi $a0, $a0, -2

**12. After returning from the last recursion, the program is about to call the next recursion. However, the last recursion came back with a returned value. What will happen to if we call recursion right away? Insert statements to prevent this from happening.**

sw $v0, 8($sp) #Top of stack

**13. Now the program is ready to call recursion again. Insert statements to update the next input argument.**

lw $a0, 4($sp) #restore the value so you won’t have (m-2)-1

addi $a0, $a0, -1 #now you have m-1

**14. After returning from the second recursion call, insert statements to update the final value to be returned to main.**

lw $a0, 4($sp) #Reset as part of the convention

lw $t0, 8($sp) #load to temporary register $t0

add $v0, $t0, $v0 #add everything to our

**15. Before returning to main, a value needs to be retrieved so the program can return to the correct location of the code. What is this value? Insert a statement under the label end\_recur to retrieve this value.**

lw $ra, 0($sp) # return the original ra to main

addi $sp, $sp, 12 # Pop stack frame

**(Assignment 1, individual) Create recursion1.s**

**Study recursion1.c and translate the same program in MIPS following register convention. You can compare the output of your MIPS program with that of recursion1.c. Save your program as recursion1.s**