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

**1. Perform a search on the Internet on the difference(s) between the terms procedures, functions, and methods.**

a named section of a program that performs a specific task. In this sense, a **function** is a type of procedure or routine. Some programming languages make a distinction between a **function**, which returns a value, and a procedure, which performs some operation but does not return a value.

In object-oriented **programming**, a **method** is a part of an object. A **method** allows the object to perform an action, whether this action is to modify itself or to return a value. An example is: public int getOne() { return 1; } This syntax is the same in different **programming** languages such as Java and C#.

In **computer programming**, a **procedure** is a set of coded instructions that tell a **computer** how to run a program or calculation. Depending on them **programming** language, a **procedure** may also be called a subroutine, subprogram or function.

**2. As we have learned in lectures, our compiled program is stored in the static part of the memory when it is being executed. When the CPU runs a program, it executes the statements according to the statement order (line numbers). Unless there is a branch (beq/bne) or jump (j/jal) statement, it will just execute the next statement.**

**3. Load proc1.s in MARS and study the MIPS program without assembling and running it. Write an equivalent program in C and name it proc1.c. You can treat m and n as variables declared in main.**

**4. In proc1.s, we use j SUM to ask the CPU to jump to the line with the label SUM and continue running the program from there. What line number is this? What does this line do?**

It is jumping to line 20, to SUM and it is adding m and n together and storing it in $v0(which was m)

**5. After function SUM is over, the program is supposed to return to the line after j SUM. In the code, jr $ra is used. Can we use j instead (we can create a label for that line)?**

No, because jr is the only R-format jump that takes you back to wherever $ra points to.

**6. Since the return address keeps changing depending on where SUM is called, we need to save the return address before SUM is called. At what line in proc1.s is the return address supposed to be saved? In what register is the address saved to? What is the value of address being saved here? Does this address value make sense?**

It is supposed to saved to adjacent line from jump. The address is saved at $ra. The value is being saved as 0x000000; address of zero does not exists

**7. Assemble the code and open the Execute tab. Here the program is listed in the Text Segment (as we have seen this in last lab). What happens when you try to run the program? This error is due to the invalid return address (program counter tells the CPU where to look for a statement).**

The program ends with an error due to invalid return address ($ra)

**8. Now, let’s correct the return address value. From the Text Segment window, what is the address of the statement that the program should return to from SUM?**

We should be back to 0x00400020 which points to line 14

**9. Modify the code so the correct return address is saved. Assemble it and take a look at the Text Segment again. (DO NOT execute it yet!) Double check the return address. Is it correct? What happened? You will know more about what happened here in later lectures.**

**addi $ra,$zero, 0x00400020**

It says something along the lines of successful reset

**10. What is the new return address? Modify your code, assemble, and run the program. What is the output of the program?**

This is what our address changes to **0x00400028**. The program gets dropped off the bottom because it is missing:

**li $v0, 10**

**Syscall**

Afterwards, we return 15.

**11. As you can see, saving the correct return address before each procedure call is tedious. It would be nice if the assembler can do it for us! Instead of using j to call a procedure, what operator should we use?**

JAL

**12. Modify the code so you don’t use j to call SUM. Make sure to comment out the line where the return address is saved.**

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

**1. Study proc2.c and trace the program. What will be the output if you run the program? Compile and run proc2.c in a terminal (or any IDE) and verify your answer.**

The output of our proc2.c is 38

**2. Load proc2.s in MARS. This is the MIPS version of proc2.c. Do not assemble and run this program, as there are errors due to the misuse of registers. Study the MAIN function and discuss with your partner about what it does (compare it with the C version).**

We are not following conventions, we have to return n unchanged, but it is changing with SUB.

It is all over the place because it is not using stack or the correct registers to appropriately execute the program.

**3. When MAIN calls SUM, SUM knows where to return to. Why? After SUM called SUB, what happens to the address returning to MAIN? Discuss with your partner about how you would resolve this problem. Do not attempt to fix it yet, as we have more problems to come.**

SUM knows to return to MAIN because the address of the next line is store in our return address register**($ra)**. After SUM calls SUB the address stored in $ra was replaced with the next line after the jump in SUM, losing our means to return to MAIN.

**4. The input argument (n, which stores x) in function SUM is used to call the next function, SUB, as well as being added to the return value. According to the register convention, the first argument of all function calls must be stored in $a0. From line 28 of proc2.s, the value in $a0 is no longer the same as the input argument of SUM (it has been changed to store input argument of SUB). We can resolve this problem by saving the original $a0 into a temporary register, but we may eventually run out of registers if our program is large. Discuss with your partner about how you would resolve this problem. Do not attempt to fix it yet.**

I can push the value of our argument register($a0) into the stack and then pop it to utilize the original value of it.

**5. Let’s take a look at line 25, what happens to the original value in $s0 from MAIN after this statement is executed? Is this a problem? Why? Discuss with your partner about how you would fix this problem. Do not attempt to fix it yet.**

It is a problem when you overwrite the $s0, you should tackle this problem by saving it in the stack area and calling them afterwards.

**6. Now we know that SUM needs to backup 3 values before calling SUB. Insert prologue and epilogue into the code so the program will run correctly. (Hint: study function SUB, as it contains not error)**

**Assignment 1, individual) Create proc3.s**

**Study the proc3.c and re-write the same program in MIPS with the following requirements:**

1. Local variables mapping:

a. main(): x -> $s0, y -> $s1

b. sum(): p -> $s0, q -> $s1

2. Input arguments mappings:

a. sum(): m -> $a0, n-> $a1

b. sub(): a -> $a0, b-> $a1

3. All return values from a function must be stored in V registers in ascending order (i.e. $v0, $v1).

4. Use of stack memory according to register convention.

Save your program as proc3.s