```
###########################################################
#           cs315 Week 7
#
#   ->  Using stack to pass argument IN & OUT
#
###########################################################

Stack usage guideline (13 step process):

    1. allocate 1 words for $ra:
        addi $sp, $sp, -4

    2. store $ra on the stack:
        sw $ra, 0($sp)

    3. allocate words for all required backup register. If we want the value to sustain or stay in registers the back them up in stack
        addi $sp, $sp, -8

    4. store the values in backup. Backup registers $t0 & $t1 on the stack:
        sw $t0, 0($sp)
        sw $t1, 4($sp)

    5. allocate space for all the arguments (arguments in & argument out). e.g. 2 arguments IN, 3 arguments OUT, hence allocate 5 words:
                                                                                                  >>>>>>>>    2 + 3 = 5 words or 20 bytes
        addi $sp, $sp, -20

    6. store arguments IN on the stack:
        sw $t0, 0($sp)
        sw $t1, 4($sp)

    7. call the subprogram:
        jal subprogram

    8. read arguments OUT from stack:
        lw $t2, 8($sp)
        lw $t3, 12($sp)
        lw $t4, 16($sp)

    9. deallocate space form step 5:
        addi $sp, $sp, 20

    10. restore the backup values:
        lw $t0, 0($sp)
        lw $t1, 4($sp)

    11. deallocate space form step 3:
        addi $sp, $sp, 8

    12. restore $ra value:
        lw $ra, 0($sp)

   13. deallocate space from step 1:
        addi $sp, $sp, 4


    *** Do you see the symmetry above?
    | $ra  |        <-- stack pointer initially ($sp) before step 1         //      stack pointer after step 13
    | $t1  |        <-- backup register $t1 on the stack                    //      restore backups (i.e. register $t1)
    | $t0  |        <-- backup register $t1 on the stack                    //      restore backups (i.e. register $t0)
    | $t4  |        <-- load argument OUT from stack and store it in $t4
    | $t3  |        <-- load argument OUT from stack and store it in $t3
    | $t2  |        <-- load argument OUT from stack and store it in $t2
    | $t1  |        <-- store argument IN (i.e. $t1) onto the stack
    | $t0  |        <-- store argument IN (i.e. $t0) onto the stack
    |      |
    |______|
       ^
     stack
```

Important Notes:

1. $sp -> address on top of the stack
2. if we allocate memory on stack, then we need to deallocate it
3. to allocate add a negative value to $sp (the negative number is actually the number of bytes to allocate)
   ```
   addi $sp, $sp, -12  # allocate 3 words
   ```
4. to deallocate add a positive value to $sp (the positive number is actually the number of bytes to allocate)
   ```
   addi $sp, $sp, 12   # deallocate 3 words
   ```
5. everything over the $sp address is garbage
6. when subprogram start and when subprogram returns, the $sp needs to be the same
7. if subprogram itself allocate a space on stack, then it should deallocate that space
8. if subprogram itself does not allocate a space, then it should not deallocate a space
9. do not use backups as arguments
10. consider main as a subprogram
11. use of stack allows us to go into subprogram that more than 1 level deep (subprogram calls another subprogram or even recursive functions)

* DO NOT TAKE SHORTCUTS!
    * many shortcuts may be obvious, but taking them will introduce potential problems.
    * one of the most common sources of crashes in these programs is reading values from an incorrect stack offset. Trying to take shortcuts is primary reason this happens.
      In other words, trying to take shortcuts is the primary reason these programs crash.
    * DO NOT take shortcuts!

* Why is $ra on the stack? (steps 1, 2, 12, and 13)
    * executing a 'jal' command will overwrite $ra
    * when $ra is overwritten, a subprogram cannot return properly (it will 'return' to itself, causing an infinite loop)
    * putting $ra on the stack and restoring it after a 'jal' allows the subprogram to return properly

* Why are backups on the stack? (steps 3, 4, 10, and 11)
    * all subprograms share the same registers, and one subprogram may overwrite register values needed in another subprogram
    * putting backups on the stack and restoring them later allows registers to be overwritten without losing values

* Why are arguments on the stack? (steps 5, 6, 8, and 9)
    * there are a limited number of argument registers
    * use of the stack allows any number of arguments to be passed (limited only by available stack space)

* Review of some guidelines for stack allocation and deallocation
    * if a subprogram allocates some stack memory, it should deallocate that memory
    * if a subprogram did not allocate some stack memory, it should not deallocate that memory

* When a subprogram calls another, the subprogram doing the calling allocate ALL argument space (the called subprogram does not allocate its own IN or OUT arguments)
    * Reminder: The stack must be used in main. Failure to use the stack in main will lose points!

```
* Some common questions and their answers:
    1. Some registers need to be backed up, and their values are also used as arguments. May I restore values by reading them from the stack arguments?
        --> No. convention allows arguments to be changed on the stack. Attempting to restore values by reading them from the space for arguments could
            lead to incorrect values being restored.
            When values are both backups and arguments, put them on the stack twice: once as a backup and once as an argument. Using arguments as backups
            is a shortcut.  Do not take shortcuts.

    2. Subprogram A calls subprogram B. Some arguments passed IN to subprogram A need to be passed IN to subprogram B. May I overlap these arguments?
        --> No. $ra must be put on the stack before calling subprogram B. Putting $ra on the stack while leaving the arguments at the correct offsets
            for subprogram B will violate convention.
            The arguments into A must be put on the stack again as arguments into B. Overlapping arguments is a shortcut. Do not take shortcuts.

    3. Some arguments OUT of subprogram A are passed IN to subprogram B, which is called immediately after A returns. Their stack offsets overlap.
        May I leave the OUT arguments from A on the stack as IN arguments for B?
        --> No. This creates inflexible code and introduce much potential for offset problems. Any code inserted between the calls to A and B may require
            the stack use to be rewritten.
            Overlapping IN and OUT arguments is a shortcut. Do not take shortcuts.

    4. Are the answers to the questions above absolutely 'no'?
        --> No.

    5. there are three main blocks on the stack for subprogram calls: $ra, backups, and arguments. May I allocate all this space at once?
        --> Yes, but this complicates keeping track of offsets.  Allocating each block separately keeps the offsets straightforward.

    7. The posted lab/program solutions show $ra being taken off the stack and immediately being put back on the stack. Why is this done?
        --> this is the result of following the 13 step process without taking shortcuts.

    6.  Using the stack requires writing a lot of code. Is this normal?
        --> Yes, so write a lot of code and do not take shortcuts.


Generic example:
        calling a subprogram:

            steps:
                allocate space for $ra and then backup $ra:
                    addi $sp, $sp, -4
                    sw $ra, 0($sp)

                 allocate space for backups and then backup $t0, $t1:
                    addi $sp, $sp, -8
                    sw $t0, 0($sp)
                    sw $t1, 4($sp)

                allocate space for arguments IN and then pass IN arguments of $t0, $t1:
                    addi $sp, $sp, -16
                    sw $t0, 0($sp)
                    sw $t1, 4($sp)

                call subprogram:
                    jal subprogram

                get OUT arguments and store them in $t3, $t4 and then deallocate space for arguments IN:
                    lw $t3, 8($sp)
                    lw $t4, 12($sp)
                    addi $sp, $sp, 16
            restore backup $t0, $t1 and then deallocate space for backups:
                lw $t0, 0($sp)
                lw $t1, 4($sp)
                addi $sp, $sp, 8

            restore $ra and then deallocate space for $ra:
                lw $ra, 0($sp)
                addi $sp, $sp, 4
```

```
Example 1: call subprogA, passing two arguments IN and one OUT. No backups necessary.
    main:
        ...
        addi $sp, $sp, -4       # step 1
        sw $ra, 0($sp)          # step 2

        # no backups needed, nothing to do for steps 3 and 4    <<<<<<<  Assume values that are in temporally registers are already stored in static
                                                                <<<<<<<  variables (in .data section), so no need to back them up on the stack

        addi $sp, $sp, -12      # step 5 (two words IN, one word OUT)
        sw $t8, 0($sp)          # step 6, first argument IN
        sw $t9, 4($sp)          # step 6, second argument IN

        jal subprogA            # step 7

        lw $t7, 8($sp)          # step 8, only argument OUT
        addi $sp, $sp, 12       # step 9

        # no backups were stored, nothing to do for steps 10 and 11

        lw $ra, 0($sp)          # step 12
        addi $sp, $sp, 4        # step 13
        ...
```

```
Example 2: using IN and OUT arguments inside subprogA subprogram
    subprogA:
        lw $t0, 0($sp)          # load first argument
        lw $t1, 4($sp)          # load second argument

        ...
        # do some coding here.
        # does example 3
        # has value to return in $t3
        ...

        sw $t3, 8($sp)          # store OUT argument

        jr $ra
```

```
Example 3: calling subprogB, passing two arguments IN and receiving two arguments OUT, backing up three values
    subprogA:
        # continuing from example 2 ...
        addi $sp, $sp, -4       # step 1
        sw $ra, 0($sp)          # step 2
```

```
addi $sp, $sp, -12      # step 3
sw $t0, 0($sp)          # step 4, first backup
sw $t1, 4($sp)          # step 4, second backup
sw $t2, 8($sp)          # step 4, third backup

addi $sp, $sp, -16      # step 5 (two word IN, two words OUT)
sw $t0, 0($sp)          # step 6, first argument
sw $t1, 4($sp)          # step 6, second argument

jal subprogB            # step 7

lw $t4, 8($sp)          # step 8, first argument
lw $t5, 12($sp)         # step 8, second argument
addi $sp, $sp, 16       # step 9

lw $t0, 0($sp)          # step 10, first backup
lw $t1, 4($sp)          # step 10, second backup
lw $t2, 8($sp)          # step 10, third backup
addi $sp, $sp, 12       # step 11

lw $ra, 0($sp)          # step 12
addi $sp, $sp, 4        # step 13

add $t3, $t5, $t4       # add return values from subprogB
# ... back to example 2
```