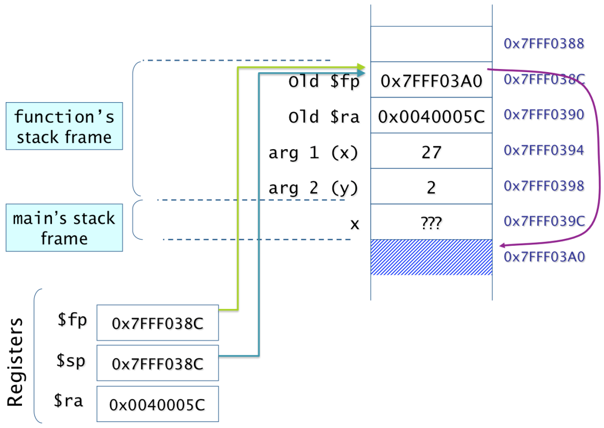
# FIT1008: Introduction to Computer Science (FIT2085: for Engineers)
## Tutorial 4 Solutions
### Semester 1, 2019

## Exercise 1

A possible memory diagram corresponding to the point right before the if-then is executed, and assuming the value of **$sp** before the **function** begins is 0**x7FFF03A0**, is on the right. By this point, **main** has created local variable **x**, but has not given it a value yet. This is why I left its value as ???, since different languages would do different things (set it to None, to Null, leave it unset, etc). If you have used anything like Null, None or \0, that would be OK. Note that if **x** was assigned to an array, the value of **x** would be the address in the heap where the actual array is stored (the array is not copied into the stack).




Once the local variable is allocated, the value of the two arguments (27 for the first one, and 2 for the second one) are pushed onto the stack. Then **main** calles the **function**, and the **functions** starts by saving the **$fp** and the **$ra**. The value of the **$fp** must be whatever address is that of local variable **x** + 4 (in this case, 0**x7FFF03A0**). That of the **$ra** can be any address pointing to the text segment, since it points to the next MIPS instruction to be executed once the **function** is finished.

The **function** then copies the **$sp** into the **$fp**. Since the value of **$sp** is at this point 0**x7FFF038C**, the **$fp** gets the same value and, as always, ends up pointing to the saved copy of itself. Since it has no local variables, the **function** prepares to execute the if-then, leaving the stack as shown by the figure.

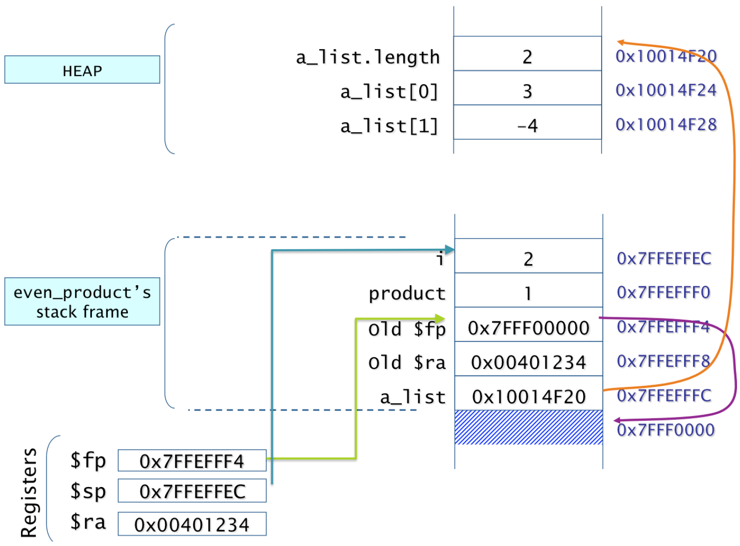
Given the memory diagram provided above, the MIPS code could be as follows:

```
    .data
nline:    .asciiz  "\n"

    .text
main: # setup frame pointer for main and make space for locals
    addi $fp, $sp, 0          # copy $sp to $fp
    addi $sp, $sp, -4         # make space for one local variable: x

    # time for n = function(27,2)
    # first pass parameters
    addi $sp, $sp, -8         # make space for 2 arguments
    addi $t0, $0, 27          # get 27 in a register
    sw $t0, 0($sp)            # pass n as arg1
    addi $t0, $0, 2           # get 2 in a register
    sw $t0, 4($sp)            # pass 2 as arg2

    # then call
    jal function              # call function

    # then return
    sw $v0, -4($fp)           # set x to the return value
    addi $sp, $sp, 8          # remove space for two args on stack

    # print x and then a new line (as printed by Python)
    lw $a0, -4($fp)           # $a0 = x
    addi $v0, $0, 1           # set syscall 1
    syscall                   # print x
    # print nline
    la $a0, nline             # load address of nline into $a0
    addi $v0, $0, 4           # set syscall to 4
    syscall                   # print newline
```

```

      # exit main
   addi $v0, $0, 10          # set syscall to 10
   syscall                   # exit

   # now for the code of the function
function: # save $ra and $fp and update $fp
   addi $sp, $sp, -8         # make space for $fp and $ra
   sw $fp, 0($sp)            # store $fp on stack
   sw $ra, 4($sp)            # store $ra on stack
   addi $fp, $sp, 0          # copy $sp to $fp

   # if x > = y
   lw $t0, 8($fp)            # $t0 = x
   lw $t1, 12($fp)           # $t1 = y
   slt $t2, $t0, $t1         # if x < y (not x >= y) then $t2 = 1, else $t2 = 0
   bne $t2, $0, endif        # If not x>=y goto endif

   # then branch: compute x -y  and return it
   lw $t0, 8($fp)            # $t0 = x
   lw $t1, 12($fp)           # $t1 = y
   sub $v0, $t0, $t1         # return value is x - y
   # returning
   lw $fp, 0($sp)            # restore $fp
   lw $ra, 4($sp)            # restore $ra
   addi $sp, $sp, 8          # remove space on stack for $fp and $ra
   jr $ra                    # return to address pointed to by $ra
   # code for the else part

endif:   add $v0, $0, $0     # return value is 0
   # returning
   lw $fp, 0($sp)            # restore $fp
   lw $ra, 4($sp)            # restore $ra
   addi $sp, $sp, 8          # remove space on stack for $fp and $ra
   jr $ra                    # return to address pointed to by $ra
```

# Exercise 2

A possible memory diagram corresponding to the point right before the while loop is executed, assuming the argument list is [3,-4], and the value of **$sp** before the **function** begins is 0**x7FFF03A0**, is as follows:




Noe that you could put the local variables **i** and **product** in any order you want, say, reversed with **i** closes to the **$fp** and **product** at the top of the stack.

# Exercise 3

1. Caller: saves temporary registers by pushing their values on to the stack
   Rationale: required to avoid the callee overwritting the values of registers that the caller intends to use after the call (thus, required for optimizing compilers who are capable of reusing registers values accross function calls). This step must be performed by the caller since the callee has no idea of which registers are to be reused. Of course, the convention could have the callee saving all $s registers. But that would be a waste of space.
2. Caller: prepares the arguments by pushing them on to the stack
   Rationale: needed to allow for a function to have more than a fixed (4 in MIPS) number of parameters. The caller must do that since it is the only one who knows which values the arguments have.
3. Caller: calls the function using the `jal` instruction
   Rationale: needed to be able to call a function **AND** be able to come back and continue executing once the function is finished. The caller must do this, since the callee has no idea who called it (i.e., can be called from many different points).
4. Callee: saves `$ra` by pushing its value on the stack
   Rationale: it allows the callee to itself call other functions without losing track of its own return address. The callee must do this since the $ra is only written after the execution control has already been transfered to the callee.
5. Callee: saves `$fp` by pushing its value on the stack
   Rationale: it allows the callee to itself call other functions without losing track of its own frame pointer. Both the callee and the caller could have done this (in fact, some call/return conventions don't even use the $fp).
6. Callee: copies `$sp` to `$fp`
   Rationale: marks the new frame. Again, both the callee and the caller could have done this.
7. Callee: allocates local variables by reserving enough space onto the stack
   Rationale: required to allow a function to have its own local variables. Only the callee knows the number and type of its local variables, so it must do it itself.
8. Callee: if there is a return value, stores it in `$v0`
   Rationale: required to allow a function to return a value. Only the callee knows its return value, so it must do it itself.
9. Callee: deallocates local variables by popping the previously pushed space
   Rationale: required to regain memory once a function is finished. Only the callee knows the space allocated to its local variables, so it must do it itself.
10. Callee: restores `$fp` by popping its saved value off the stack
    Rationale: same as for its storing. Again, both the callee and the caller could have done this.
11. Callee: restores `$ra` by popping its saved value off the stack
    Rationale: same as for its storing. Only the callee can do this, since the value is needed in $ra in order for control to return to the caller.
12. Callee: returns using the `jr $ra` instruction
    Rationale: required to be able to return control to the caller. Obviously, only the callee can do this.
13. Caller: clears the function arguments by popping their allocated space off the stack
    Rationale: needed to regain memory once a function is finished. This can be done by either of them (the callee also knows the amount of space taken by the arguments). It is just cleaner to have the same one pospping them.
14. Caller: restores temporary registers by popping their values off the stack
    Rationale: required to restore the values of the registers that the caller intends to use after the call. Again, this step must be performed by the caller.
15. Caller: uses the return value `$v0` if necessary
    Rationale: required to allow the caller to use the value returned by the function. Of course, it can only be performed by the caller.

## Exercise 4

Given the memory diagram shown in Exercise 2 above, the MIPS code could be as follows:

```
    .text
even_product: # save $fp and $ra, and update $fp
    addi $sp, $sp, -8        # make space for $fp and $ra
    sw $fp, 0($sp)           # store $fp on stack
    sw $ra, 4($sp)           # store $ra on stack
    addi $fp, $sp, 0         # copy $sp to $fp

    # allocate 2 local variables (product and i)
    addi $sp, $sp, -8        # move $sp by 8 bytes
    # setup product=1
    addi $t0, $0, 1          # $t0 = 1
    sw $t0, -4($fp)          # product = 1

    # setup i=length of array
    lw $t0, 8($fp)           # $t0 = &(the start of a_list)
    lw $t1, ($t0)            # $t0 = length of a_list
    sw $t1, -8($fp)          # i = length of a_list

loop:# time for the loop
    #check condition (while i != len(the_list)
    lw $t0, -8($fp)          # $t0 = i
    beq $t0, $0, endloop     # if i == 0 go to endloop, otherwise keep on executing the
        loop

    # if i % 2 == 0
    lw $t0, -8($fp)         # $t0 = i
    addi $t1, $0, 2          # $t1 = 2
    div $t0, $t1             # i // 2, since we want the reminder, we cannot use sra
    mfhi $t2                 # $t2 = i % 2
    bne $t2, $0, else        # if i % 2 is not 0), go to else

    # product = product*a_list[i]
    # get a_list[i] in $t4
    lw $t0, -8($fp)         # $t0 = i
    sll $t1, $t0, 2          # $t1 = 4*i
    lw $t2, 8($fp)           # $t2 = &(start of the_list)
    add $t3, $t1, $t2        # $t3 = &(start of the_list) + 4*i
    lw $t4, 4($t3)           # $t4 = the_list[i]

    lw $t0, -4($fp)          # $t0 = product
    mul $t1, $t4, $t0        # product * a_list[i]
    sw $t1, -4($fp)          # product = product * a_list[i]

else:#out of the if-then, time to decrement i
    lw $t0, -8($fp)         # $t0 = i
    addi $t0, $t0, -1        # $t0 = i - 1
    sw $t0, -8($fp)         # i  = i - 1

    # back to loop
    j loop

endloop:# time to  return
    lw $v0, -4($fp)          # set return value $v0 = product
    # deallocate 2 local variables
    addi $sp, $sp, 8

    # restore $fp and $ra
    lw   $fp, 0($sp)          # restore $fp
    lw   $ra, 4($sp)          # restore $ra
    addi $sp, $sp, 8         # remove space on stack for $fp and $ra
    jr   $ra                  # return to address pointed to by $ra
```

How many of you realised the Python code was incorrect? One way of fiing would be to set `i` to the length minus 1, and the condition to `i>=0`. As you can see, you can still correctly translate it. It will just work as badly as the Python code was designed.

## Exercise 5

(a) It would be easy to think that the order is not necessary, and it is simply a convention; that it could happily be reversed, intertwined, etc, as long as everyone followed the same convention.
This is true execept for functions which take a variable number of parameters (such as formatter.format). These functions determine how many parameters they have actually been called with by examining the first parameter (in the case of format, by counting % characters). If the first parameter is at a fixed place, like `8($fp)`, it can be found, and scanned for % signs. If it is at an unknown offset from `$fp`, which will be the case if the order of parameters is reversed, the function won't be able to locate it and establish how many parameters there are in total (and thus avoid trampling on some other function's stack frame).

(b) `4($fp)` contains the value of the `$ra` register set by instruction `jal` right before transferring control to the function. It will only be needed to restore the `$ra` before `jr` is executed, and thus be able to return to the caller of the function. And even this is usually done by accessing `4($sp)` rather than `4($fp)`.