New assembly instructions:

ble (branch if less or equal than)

ble x_{11}, x_{12}, offset

if $x_{11} < x_{12}$: PC ⟵ PC + offset

this means that I Jump by changing the program counter, adding a certain offset.

bge (branch if greater or equal than)

bge x_{11}, x_{12}, offset

if $x_{11} \geqslant x_{12}$: PC ⟵ PC + offset

Example program:

```
.data
    .word 12, 19
.text
    # We want to write a program that
      loads the smallest input number into the
      register a0. In this case we use the numbers
      12 and 19, but the program should work
      with any input.
```

Possible solution:

0x10010000 → 12

0x10010004 → 19

```
.data
    .word 12, 19
.text
```

```
lui  t0, 0x10010000
lw   a0, 0(t0) - 12
lw   a1, 4(t0) - 19
bge  a1, a0, salta
```

If I don't branch (Jump) here, it means that a_1 is smaller than a_0 ($a_1 < a_0$); so I can move a_1 into a_0 and STOP. In this case a_1 is > than a_0, so we Jump.

`addi a0, a1, 0` / `add a0, a1, zero` / `or a0, a1, zero` / `ori` ...

`mv a0, a1` — but instead of using one of the instructions above, we can use the pseudo instruction "mv".

I can move a_1 into a_0 by using one of these instructions.

salta:

Let's do another exercise, but this time using an array:

```
.data
    .word 8, 3, 4, -2, -9, 10, 11, 0, 5
.text  # We want to make a program
       that prints the smallest number
       of the array; to do this we
       have to use a loop and store
       in a register the value that
       the loop has encountered so far;
       by doing this, when the loop
       will finish, the smallest number
       of the array will be stored
       in that register.
       Let's use the register a0 to
       store this value, but:
       how can we initialize a0?
```

(8: this is the length of the array; 3: the array starts here)

the text continues below ↓

The smartest thing is to initialize it using the 1st element of the array (elegant way).

.text

lui t_0, 0x10010000 — We load the address of the length of the array in t_0.

lw t_1, 0(t_0) — We load in t_1 the effective length of the array.

lw a_0, 4(t_0) — We load in a_0 the 1st element of the array, so the number 3.

ori t_0, t_0, 0x08 — After this instruction t_0 will contain the address of the 2nd element of the array.

↑ destination register

this to say "take the content of t_0 and do the OR with 0x08".

→
```
10010000 ORi
00000008
--------
10010008
```

ATTENTION. In this case we used the ORi to make the sum, but this doesn't work always!

addi t_1, t_1, -1 — With this instruction we decrease the number stored in t_1, that is actually the effective length of the array.
We do this because the next instruction will start the loop and considering that the length

of the array is 8, and we need 7 loops in order to terminate the program, then we MUST decrease this number by 1 before even starting the loop.

ciclo: lw t_2, 0(t_0) — Here the loop starts and stores in t_2 the 2nd element of the array, so the number 4.

bge t_2, a_0, salta — Here we check if the content of t_2 is ≥ than the content of a_0. If it's ≥, we Jump; otherwise we execute the next instruction.

mv a_0, t_2 — With this instruction we move in a_0 the content of t_2; but this instruction depends on the previous one, so it will be executed only in some precise cases.

salta: addi t_0, t_0, 4 — With this instruction we load into t_0 the address of the 3rd element of the array, so the number -2.

addi t_1, t_1, -1 — With this instruction we decrease again the value of the remaining loops.

bne t_1, zero, ciclo — With this instruction we check when the loop MUST end.

li a_7, 1 — With this instruction we print the content stored into the register a_0. This printing system call always considers the content of the register a_0.

ecall

li a_7, 10 — With this instruction we close the program.

ecall

Let's introduce functions in assembly:

First, while using functions in assembly, we MUST save the "PC + 4" in order to be able to Jump back after calling a function.

We can do this by using 2 new assembly instructions:

Jal (Jump and link)

Jal ra, offset (ra: nickname of the register 2)

ra ← PC + 4

PC ← PC + offset — This finds the position of the function that we want to call.

(we replace this with the name of the function)

This instruction is used to call a function inside the MAIN function.

Jalr (Jump and link register)

Jalr zero, ra, offset

zero ← PC + 4

PC ← ra + offset (usually 0)

PC ← ra

the register zero can't be overwritten, so this never happen!!!

This instruction is used to go back to the main function after calling another function.

Functions' parameters:

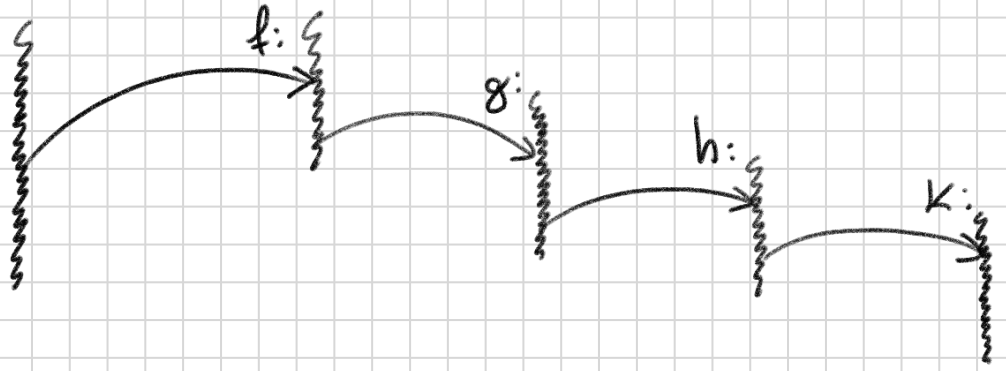
Traditionally the parameters are stored in the registers a_0 - a_7 if the parameters are ≤ 8.
The results of the functions are stored into the registers a_0 - a_1. (tradition)

Let's see an example:

Let's consider a function (piuuno) that returns the input parameter incremented by 1.

```
piuuno: addi a0, a0, 1
        Jalr zero, ra, 0
        Jal  ra, piuuno
        x
```

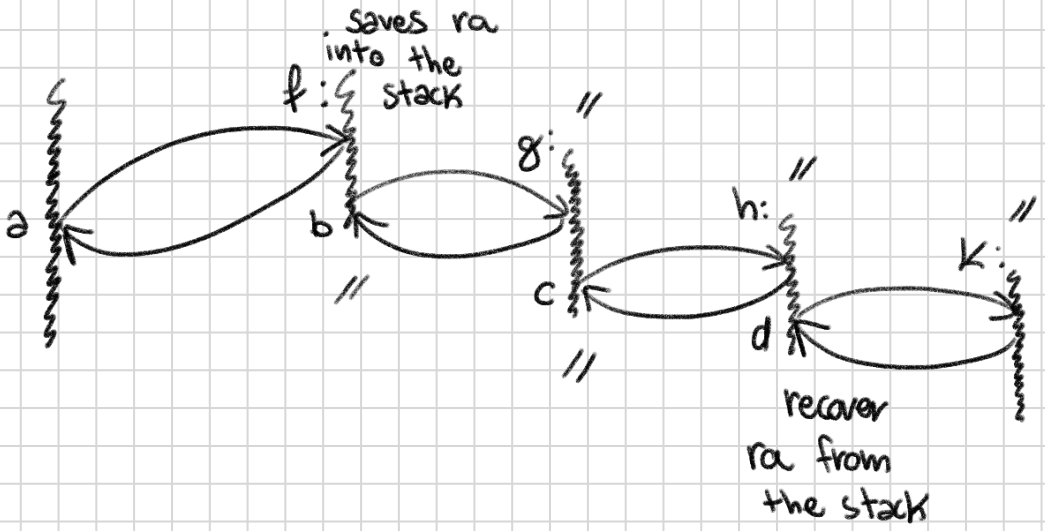
When we have to use/call a single function, then we use the method above; but when we have to use/call many functions, one after another, then everything becomes more complicated.



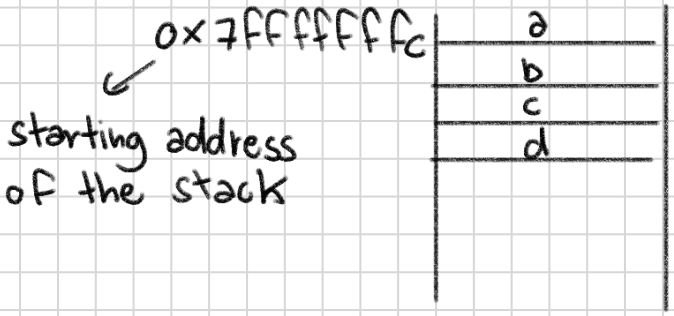
If we call many functions, we have to store in the memory the return address of each function, in this case we've to store the addresses of f, g, h and K.

To do this we use the stack data structure; we store their addresses in order and when we have to

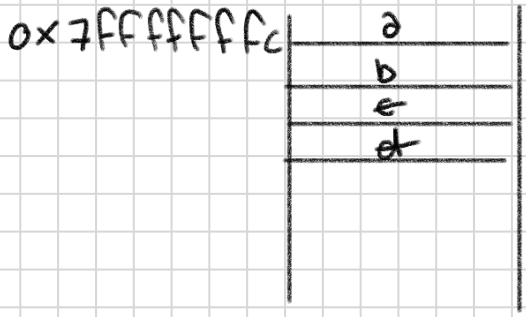
read the addreses to go back, we read them in reverse order.



The stack is a "place" into the RAM memory



After I recover ra from the stack, I have to delete it, step by step, until the stack will be empty.

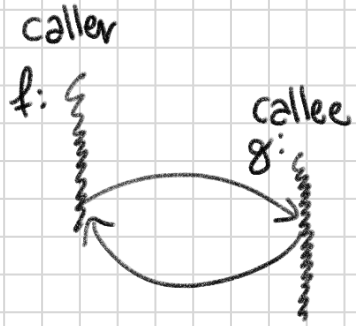


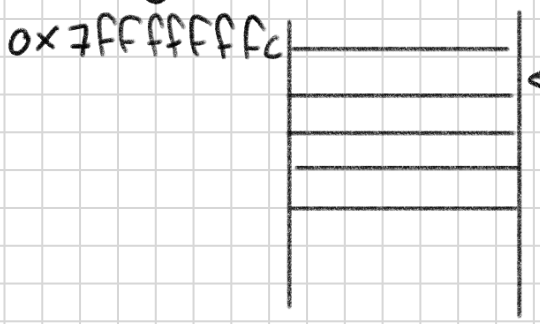
Another problem:

When we use different functions, these functions may overwrite the already used registers in the MAIN part of the program, so in order to avoid this, we establish a rule: the function "caller" saves its registers (t_0 - t_6)

in the STACK, and the "callee" function does the same but with the registers S_0 - S_{11}.

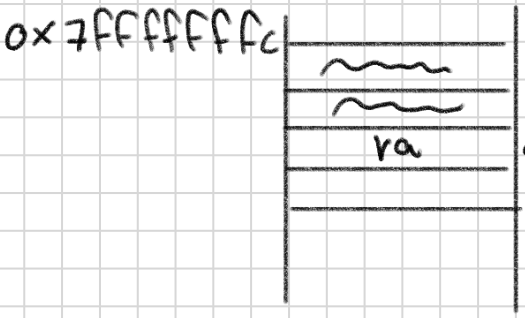
THE STACK IS REVERSED





← stack pointer (sp) (x_2)

Because we need to know the last register that we saved in the stack.

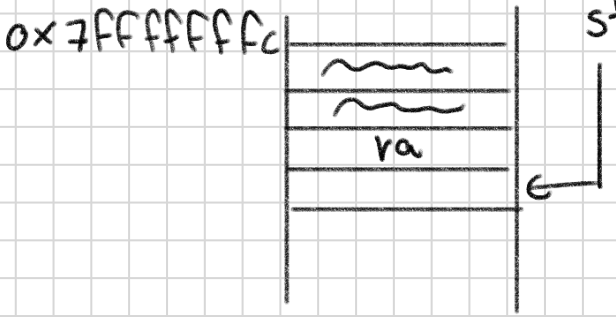


stack pointer (sp) (x_2)

```
addi sp, sp, -4
sw   ra, 0(sp)
```
} here I save the register into the stack

```
lw   ra, 0(sp)
addi sp, sp, 4
```
} here I load the register from the stack and I delete it.

Let's now suppose that we have to save two registers into the stack: ra and a_0.



stack pointer (sp) (x_2)

```
addi sp, sp, -8
sw   ra, 0(sp)
sw   a0, 4(sp)
```

```
lw ra, 0(sp)
lw a0, 4(sp)
addi sp, sp, 8
```

Recursive function examples:

Let's see a 1st recursive function that computes the factorial of an input.

$$f(n) \begin{cases} 1 \text{ if } n=1 \leftarrow \text{base case when } n=0 \\ n \cdot f(n-1) \end{cases}$$

we have to identify the base cases and the recursive cases:

- **base cases:** are the cases when we don't have to call the function again, in the function above, the base case is when n=0;

- **recursive cases:** we take as an assumption that the function works well with small numbers and then we use it with bigger numbers.

input=0 ↓

```
f:   bne  a0, zero, ric  # comparison
     li   a0, 1          # store 1 in a0
     Jalr zero, ra, 0    # return to MAIN
```

```
MAIN {
  var = 0
  factorial = f(0)
  print(factorial)
```

```
ric: addi sp, sp, -8     # book the word in the STACK
     sw   ra, 0(sp)      # store in the STACK the return address
     sw   a0, 4(sp)      # store in the STACK the initial input
     addi a0, a0, -1     # to do (n-1)
     Jal  ra, f          # recall the same function
     lw   t0, 4(sp)      # load in t0 the initial input
```

```
mul  a0, a0, t0 # product between a0 and t0
lw   ra , 0(sp) # store the return address of MAIN in ra
addi sp, sp, 8  # clean the stack
Jalr zero, ra, 0 # return to the MAIN
```

Let's see a 2nd recursive function that computes the sum of an array. (only crazy people do this) XD

sum
↓
S. (a [0 ... (n-1)])
↑ array

$$s(a) \begin{cases} 0 \text{ if } a \text{ is empty} \\ a[0] + s(a[1:(n-1)] \text{ otherwise} \end{cases}$$

```
.data
    .word 5 # array length
    .word 7,8,0,-2,1 #array
.text
START
    lui  t0, 0x10010
    ori  a0, t0, 0x04
    lw   a1, 0 (t0)
    Jal  ra , S

S:  bne  a1, zero, ric
    li   a0, 0
    Jalr zero, ra, 0
ric addi sp, sp, -8
    sw   ra, 0(sp)
    sw   a1, 4(sp)
    addi a0, a0, 4
    addi a1, a1, -1
    Jal  ra, s
    lw   t0, 4(sp)
    lw   t0, 0(t0)
    add  a0, a0, t0
    lw   ra, 0(sp)
    addi sp, sp, 8
    Jalr zero, ra, 0
```