

INSTRUCTIONS

● BLT (BRANCH LESS THAN <)

ex: blt x11, x12, offset → if $x11 < x15$:
 $PC \leftarrow PC + offset$

● BGE (BRANCH GREATER OR EQUAL THAN ≥)

ex: if $x11 \geq x12$:
 $PC \leftarrow PC + offset$

PROGRAM #1 (Load the smaller of the two words into a0)

. data

- word 12, 1a

. text

```

lui t0, 0x10010      # loading address of 12
lw a0, 0(t0)          # loading 12 in a0
lw a1, 4(t0)          # loading 9 in a1
bge a1, a0, salta    # if a1 ≥ a0 skip, otherwise
addi a0, a1, 0         # a1 shifted into a0

```

salta:



We could use a pseudo-instruction to move the content of one register into another.
We will use "MV" (move). This "pseudo-instruction" will choose on its own the instruction to implement, let that be add, or etc. It's mainly used because it makes life easier for us human being.

PROGRAM #2 (print the smallest n° in the array)

. data

length array
Word 8, 3, 4, -2, -9, 10, 11, 0, 5

. text

```
lui t0, 0x10000
lw t1, 0(t0)
lw a0, 4(t0)
ori t0, t0, 0x08
addi t1, t1, -1
ciclo: lw t2, 0(t0)
bge t2, a0, salta
mv a0, t2
salta: addi t0, t0, 4
addi t1, t1, -1
bne t1, zero, ciclo
li a7, 1
ecall
li a7, 10
ecall
```

address of 8 inside t0
loads 8 inside t1
loads 1st element of array (3) in a0
"sums" 8 to t0 in order to increment later
decrement t1 (contains array length)
loads 2nd element of array (4) in t2
compares t2 with a0
t2 moved in a0
increments address to get next element
decrement length
compares t0 with R[0]
print integer
system call
} exit

FUNCTIONS IN ASSEMBLY

When we call a function inside our main code, we need to know the address of where the function should go back once its done executing its code.
For this reason, we have to store the next PC address :

- registers (faster so this is the go-to choice)
- memory

By doing so, we will overwrite the PC (Program Counter) and add 4 (PC+4).

We will introduce two new instructions :

JAL → JUMP AND LINK. This helps us to call the function inside the main AND store the next address to come back to.

Ex: jalr ra, offset

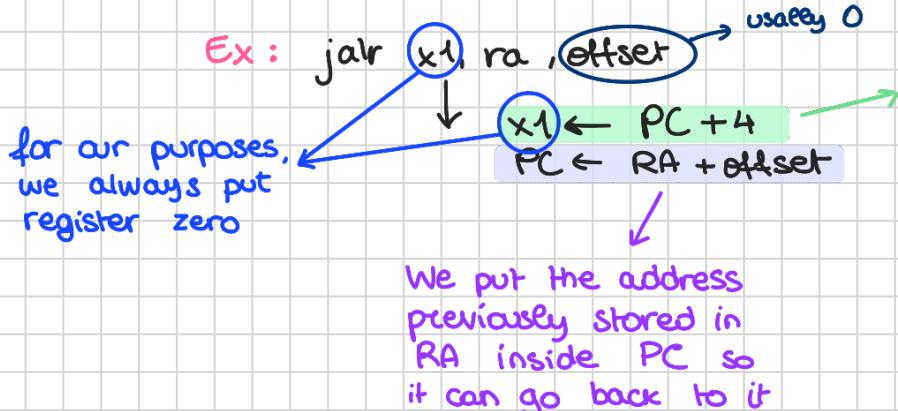
↓

ra ← PC + 4 we store the next address to come back to
PC ← PC + offset

we usually put the name of the function in RARS simulator.
The program itself understands HOW much to jump.

this jumps to the function to execute

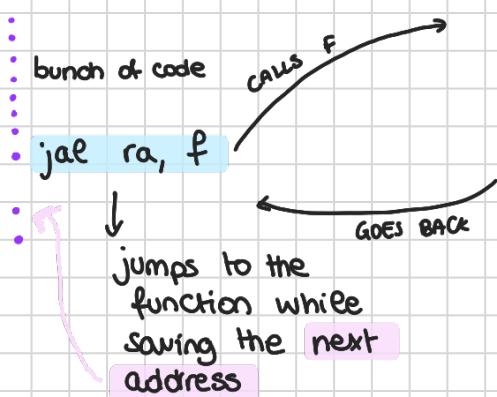
JALR → JUMP AND LINK REGISTER. This is used to go back from the function to the main



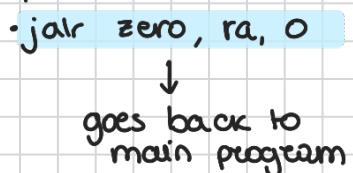
NOTE: The register 2, also known as "ra", always contains the next address to go back.

FUNCTIONS SPECIFICS :

MAIN PROGRAM



FUNCTION F



- Parameters of a function are stored from register a0 to a7 (maximum of 8)
- Results of a function are stored in either a0 or a1

EXAMPLE:

sum 1 to a register given in input

- piuuno : `addi a0, a0, 1`
`jalr zero, ra, 0` } FUNCTION USUALLY DEFINED AFTER PROGRAM'S ECALL
- `jalr ra, piuuno` → example of calling a function in your main program.

There's a case where we call a function **f** which calls another function **g**. The problem would be:

- HOW TO GO BACK TO YOUR MAIN ?

Let's say you jumped from **main** → **f** → **g**.

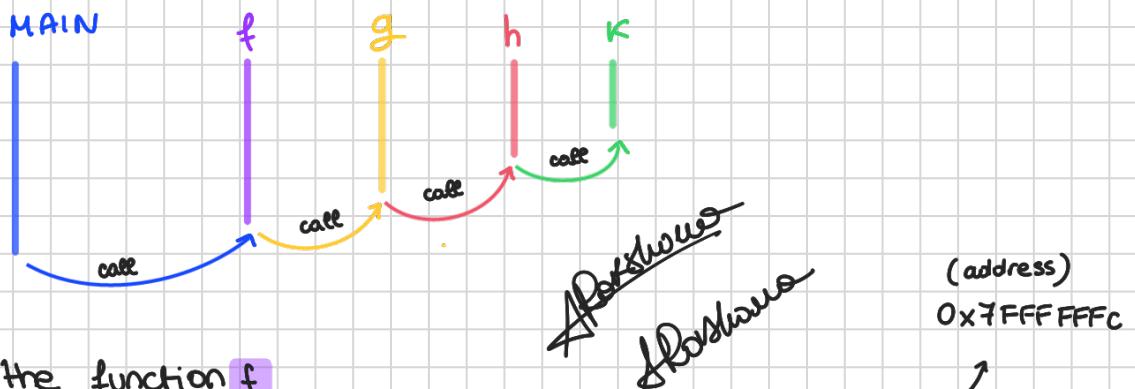
From **main** → **f**, you store in **"ra"** the return address of **main**.

From **f** → **g**, you store in **"ra"** the return address of **f**.

So from **g**, you read **"ra"** and go back to **f** but then, how would you go back to main if "ra" has been overwritten?

- WE WILL USE THE STACK → this way we can keep storing return addresses without overwriting.

The stack is placed into the memory.



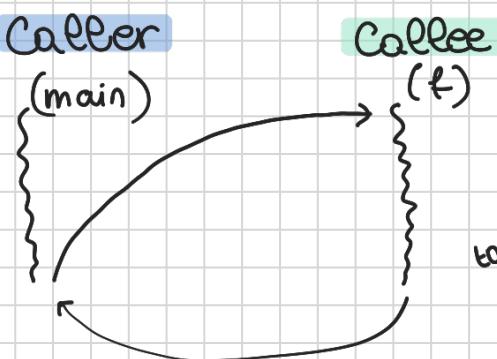
- We call the function **f**
- **F** will store **ra** in stack
- **=** calls another function **"g"**
- **g** stores **ra** of **f** in the stack
- **=** calls another function **"h"**
- **h** stores **ra** of **g** in the stack.
- **=** calls another function **"k"**
- **K** stores **ra** of **h** in the stack
- **=** doesn't have to call another function so it will read "**ra_h**" and go back to **h**.
- **h** will read "**ra_g**" and go back to **g**.
- **g** will recover "**ra_f**" and go to **f**
- **f** will recover "**ra_m**" and go back to the **main** function.

NOTE: whenever a function uses an address saved in stack, it deletes it as well

we keep storing and will use it from last to first.

ANOTHER PROBLEM →

If we call a function, what if it uses registers that already contain some info



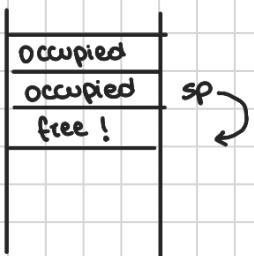
That's why we save registers in the stack.

To make life easier:

t0 to t6 ← t registers are saved by Caller
s0 to s11 ← s registers are saved by Callee

1.39.21

It's important to know the last one you saved. So we use a



SP - Stack Pointer (register x2)

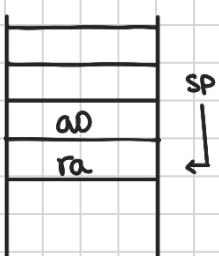
addi sp, sp, -4
sw ra, 0(sp)

We make a spot for ra (-4)
Save ra in stack

lw ra, 0(sp)
addi sp, sp, 4

Load address from stack to ra
delete the spot

HIGHEST TO LOW



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

We make two spots for a0 & ra
Save ra in the stack
" " a0 " " "

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

Load address in ra
" " " " a0
delete the two used spots

RECURSIVE FUNCTION (example: factorial)

$$f(n) = \begin{cases} 0 & \text{if } n=0 \text{ (BASE CASE)} \\ n \cdot f(n-1) & \text{otherwise (RECURSIVE CASES)} \end{cases}$$

FACTORIAL

① IDENTIFY BASE CASES
(when you don't have to call function again)

② IDENTIFY RECURSIVE CASES

f: bne a0, zero, ric
li a0, 1
jalr zero, ra, 0
ric: addi sp, sp, -8
sw ra, 0(sp)
sw a0, 4(sp)

conditional
+ This is done if input=0
save ra in stack
save "n" in stack

```

add. a0, a0, -1      # to get "n-1"
jal ra, f             # 0x0040101c
lw t0, 4(sp)          a0 will contain n-1! (so we call f)
mul a0, a0, t0        # we take n from the stack
lw ra, 0(sp)          # we take ra (since it was overwritten) to pc
add. sp, sp, 8         # we give back the slots to stack
jaer zero, ra, 0       # go back to caller

```

MULTIPY ←

PROGRAM FACTORIAL

MAIN FUNCTION

```

0x00400000 li a0, 5
jal ra, f
li a7, 1
ecall

```

5
0x00400008
4
0x00401020
3
0x00401020
2
0x00401020
1
0x00401020

PROGRAM (sum array)

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

• data

- Word 5
- word 7, 8, 0, -2, 1

(we load an array to parameter by giving length and first element)

• Text

```

lui t0, 0x10010
ori a0, t0, 0x04
lw at, 0(t0)
jal ra, sum

```

sum: bne at, zero, ric

li a0, 0

jalr zero, ra, 0

ric: add. sp, sp, -8 # to save ra and 1st element

sw ra, 0(sp)

sw at, 4(sp) # address of 1st element

add. a0, a0, 4

add. at, at, -1

jalr 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