Computer Architecture

DATA: 11/03/2022

INSTRUCTIONS

- **BLT** (BRANCH LESS THAN $<$)

  ex: blt x11, x12, offset → if x11 < x15 :

  PC ← PC+ offset

- **BGE** (BRANCH GREATER OR EQUAL THAN $\geq$)

  ex: if x11 $\geq$ x12 :

  PC ← PC+ offset

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

```
.data
  .word 12, 19

.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
.word 8,3,4,-2,-9,10,11,0,5
.text
        lui t0, 0x10010     # address of 8 inside t0
        lw t1, 0(t0)        # loads 8 inside t1
        lw a0, 4(t0)        # loads 1st element of array (3) in a0
        ori t0, t0, 0x08    # "sums" 8 to t0 in order to increment later
        addi t1, t1, -1     # decrement t1 (contains array length)
ciclo   lw t2, 0(t0)        # loads 2nd element of array (4) in t2
        bge t2, a0, salta   # compares t2 with a0
          mv a0, t2         # t2 moved in a0
salta:  addi t0, t0, 4      # increments address to get next element
        addi t1, t1, -1     # decrement length
        bne t1, zero, ciclo # compares t0 with R[0]
        li a7, 1            # print integer
        ecall               # system call
        li a7, 10           # exit
        ecall
```

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`

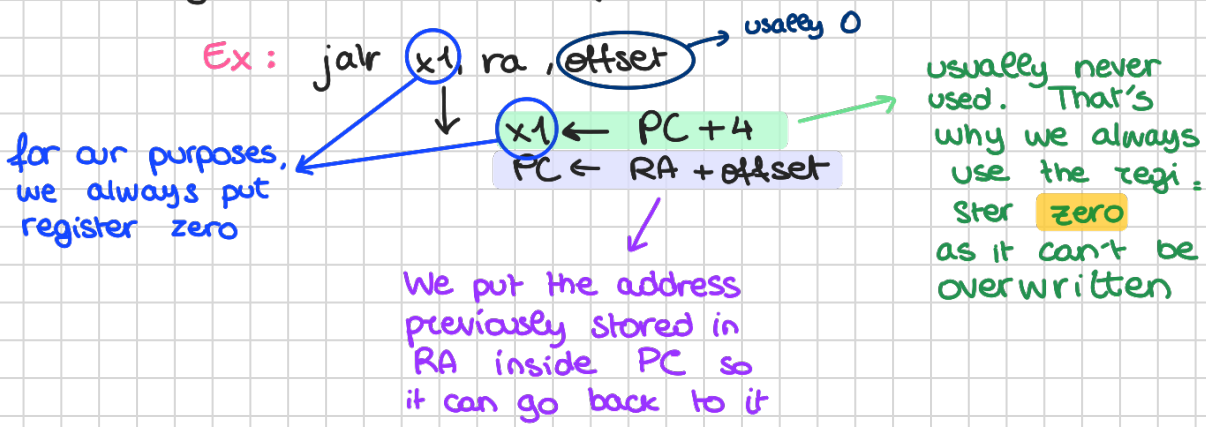
↓

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

`PC ← PC + offset` — this jumps to the function to execute

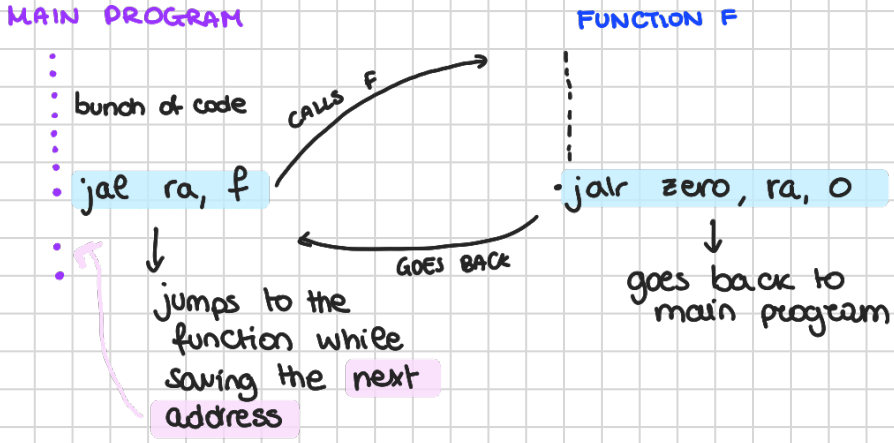
We usually put the name of the function in RARS simulator. The program itself understands HOW much to jump.

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:



- 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
- jal ra, piuuno → example of calling a function in your main program.

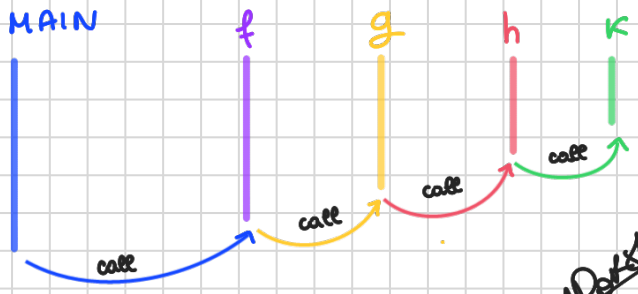
There's a case where we call a function **f** which calls another function **g**. The problem would be:

- **HOW TO GO BACK TO YOUR MAIN ?**

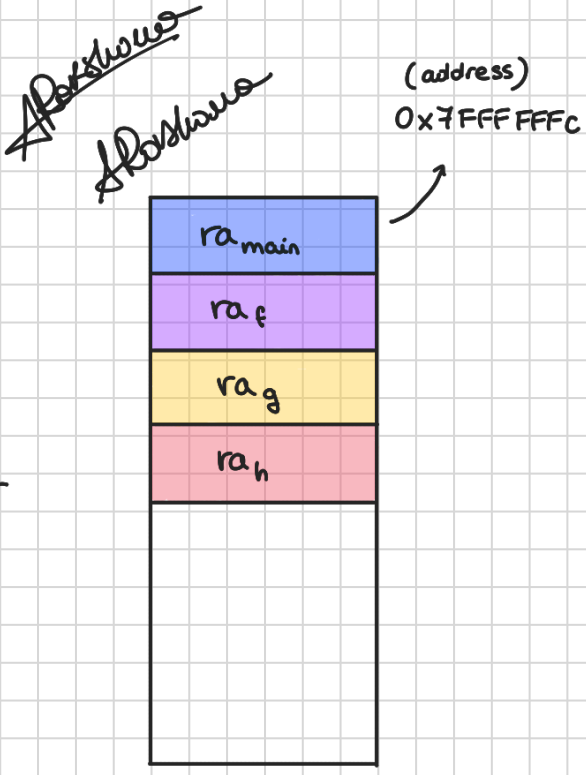
  Let's say you jumped from main → f → g.
  From main → f, you store in "ra" the return address of main.
  From f → g, you store in "ra" the return address of "f".
  So from g, you read "ra" and go back to f but then, how would you go back to main if "ra" has been overwritten ?

- **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.



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

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

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

↓

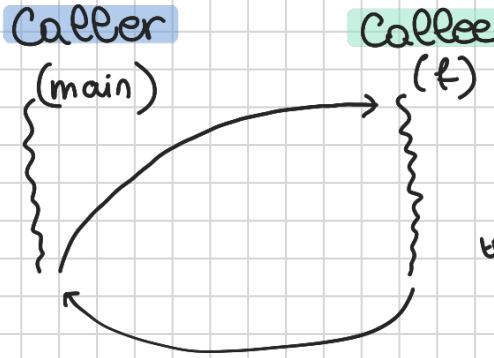
That's why we save registers in the stack.

To make life easier:
- t registers are saved by Caller
- s registers are saved by Callee

t0 to t6

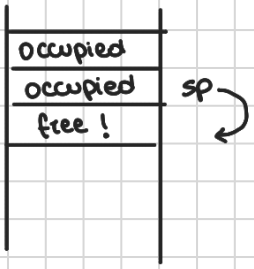
s0 to s11

1.39.21



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

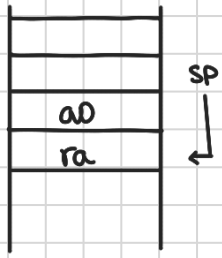
SP - Stack Pointer (register x2)




```
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 dont 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

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

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

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

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

- text

```
lui t0, 0x10010
ori a0, t0, 0x04
lw a1, 0(t0)
jal ra, sum
```

```
sum: bne a1, zero, ric
     li a0, 0
     jalr zero, ra, 0
ric: addi sp, sp, -8     # to save ra and 1st element
     sw ra, 0(sp)
     sw a1, 4(sp)        # address of 1st element
     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
```