ARCOS Group

uc3m | Universidad Carlos III de Madrid

L3: Fundamentals of assembler programming (4)
Computer Structure

Bachelor in Computer Science and Engineering
Bachelor in Applied Mathematics and Computing
Dual Bachelor in Computer Science and Engineering and Business Administration

Contents

- Basic concepts on assembly programming
- RISC-V 32 assembly language, memory model and data representation
- Instruction formats and addressing modes
- Procedure calls and stack convention
  - How do you call a function/subroutine?
  - Where is the return address stored in non-terminal routines?
  - What is the parameter passing convention?
  - What is the register use agreement?
  - What are the local variables like?

Contents

- Basic concepts on assembly programming
- RISC-V 32 assembly language, memory model and data representation
- Instruction formats and addressing modes
- Procedure calls and stack convention
  - How do you call a function/subroutine?
  - Where is the return address stored in non-terminal routines?
  - What is the parameter passing convention?
  - What is the register use agreement?
  - What are the local variables like?

Procedures and functions

```
int factorial(int x) {
  int i;
  int r=1;
  for (i=1;i<=x;i++) {
    r*=i;
  }
  return r;
}
…
r1 = factorial(3) ;
…
```

- A high-level function (procedure, method, subroutine) is a subprogram that performs a specific task when invoked.
  - Receives input arguments or parameters
  - Returns some result

Félix García Carballeira, Alejandro Calderón Mateos

Functions in a high-level language
Steps in the execution of a function

```
int main() {
  int z;
1 x=3;
  z=factorial(x);
  print_int(z);
}
```

```
int factorial(int x) {
  int i;
  int r=1;
  for (i=1;i<=x;i++) {
    r*=i;
  }
  return r;
}
```

1. Place the parameters in a place where they can be accessed by the function
2. Transfer the flow control to the function
3. Acquire storage resources needed for the function
4. Perform the desired task
5. Store the result where the calling function can access it
6. Return control to the point of origin

Functions in a high-level language

Steps in the execution of a function

```
int main() {
  int z;
  x=3;
  z=factorial(x);
  print_int(z);
}
```

2

```
int factorial(int x) {
  int i;
  int r=1;
  for (i=1;i<=x;i++) {
    r*=i;
  }
  return r;
}
```

1. Place the parameters in a place where they can be accessed by the function
2. Transfer the flow control to the function
3. Acquire storage resources needed for the function
4. Perform the desired task
5. Store the result where the calling function can access it
6. Return control to the point of origin

Functions in a high-level language

Steps in the execution of a function

```
int main() {
  int z;
  x=3;
  z=factorial(x);
  print_int(z);
}
```

3

```
int factorial(int x) {
  int i;
  int r=1;
  for (i=1;i<=x;i++) {
    r*=i;
  }
  return r;
}
```

1. Place the parameters in a place where they can be accessed by the function
2. Transfer the flow control to the function
3. Acquire storage resources needed for the function
4. Perform the desired task
5. Store the result where the calling function can access it
6. Return control to the point of origin

Functions in a high-level language
Steps in the execution of a function

```
int main() {
  int z;
  x=3;
  z=factorial(x);
  print_int(z);
}
```

```
int factorial(int x) {
  int i;
  int r=1;
  for (i=1;i<=x;i++) {
    r*=i;
  }
  return r;
}
```

Local variables

1. Place the parameters in a place where they can be accessed by the function
2. Transfer the flow control to the function
3. Acquire storage resources needed for the function
4. Perform the desired task
5. Store the result where the calling function can access it
6. Return control to the point of origin

Functions in a high-level language

Steps in the execution of a function

```
int main() {
  int z;
  x=3;
  z=factorial(x);
  print_int(z);
}
```

4

```
int factorial(int x) {
  int i;
  int r=1;
  for (i=1;i<=x;i++) {
    r*=i;
  }
  return r;
}
```

1. Place the parameters in a place where they can be accessed by the function
2. Transfer the flow control to the function
3. Acquire storage resources needed for the function
4. Perform the desired task
5. Store the result where the calling function can access it.
6. Return control to the point of origin

Functions in a high-level language

Steps in the execution of a function

```
int main() {
  int z;
  x=3;
  z=factorial(x);
  print_int(z);
}
```

```
int factorial(int x) {
  int i;
  int r=1;
  for (i=1;i<=x;i++) {
    r*=i;
  }
  return r;
}
```

5

1. Place the parameters in a place where they can be accessed by the function
2. Transfer the flow control to the function
3. Acquire storage resources needed for the function
4. Perform the desired task
5. Store the result where the calling function can access it.
6. Return control to the point of origin

Félix García Carballeira, Alejandro Calderón Mateos

Functions in a high-level language

Steps in the execution of a function

```
int main() {
  int z;
  x=3;
  z=factorial(x);
  print_int(z);
}
```

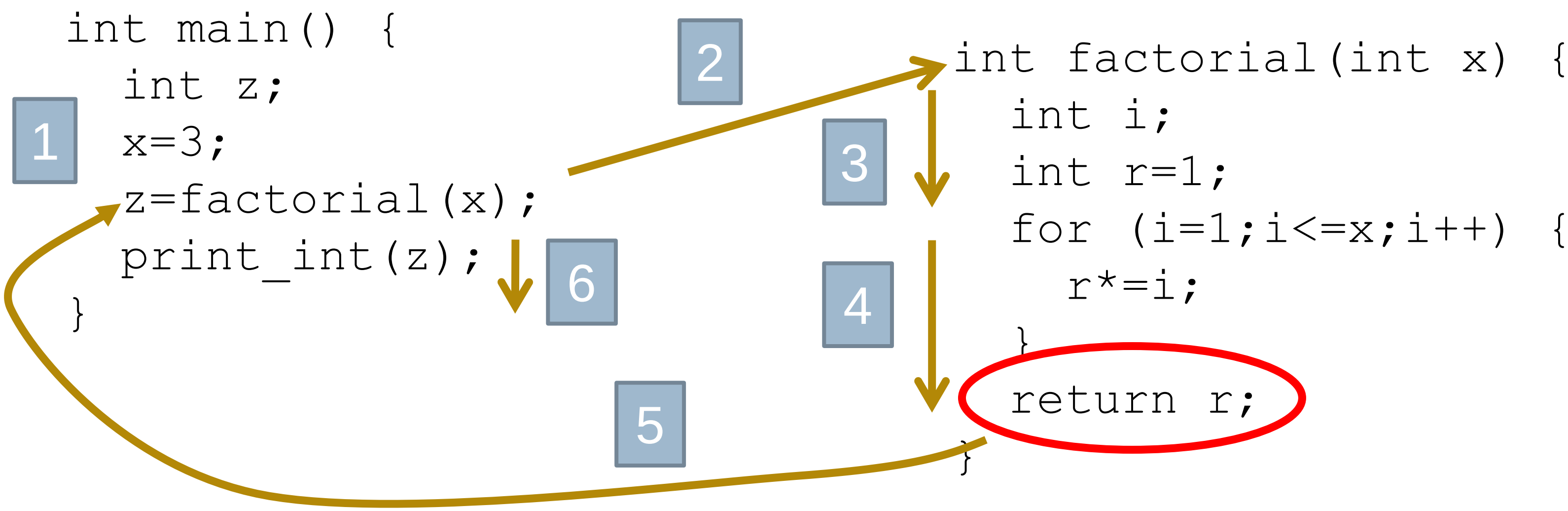
6

```
int factorial(int x) {
  int i;
  int r=1;
  for (i=1;i<=x;i++) {
    r*=i;
  }
  return r;
}
```

1. Place the parameters in a place where they can be accessed by the function
2. Transfer the flow control to the function
3. Acquire storage resources needed for the function
4. Perform the desired task
5. Store the result where the calling function can access it.
6. Return control to the point of origin

Steps in the execution of a function de alto nivel

summary



1. Place the parameters in a place where they can be accessed by the function
2. Transfer the flow control to the function
3. Acquire storage resources needed for the function
4. Perform the desired task
5. Store the result where the calling function can access it.
6. Return control to the point of origin

Procedures and functions

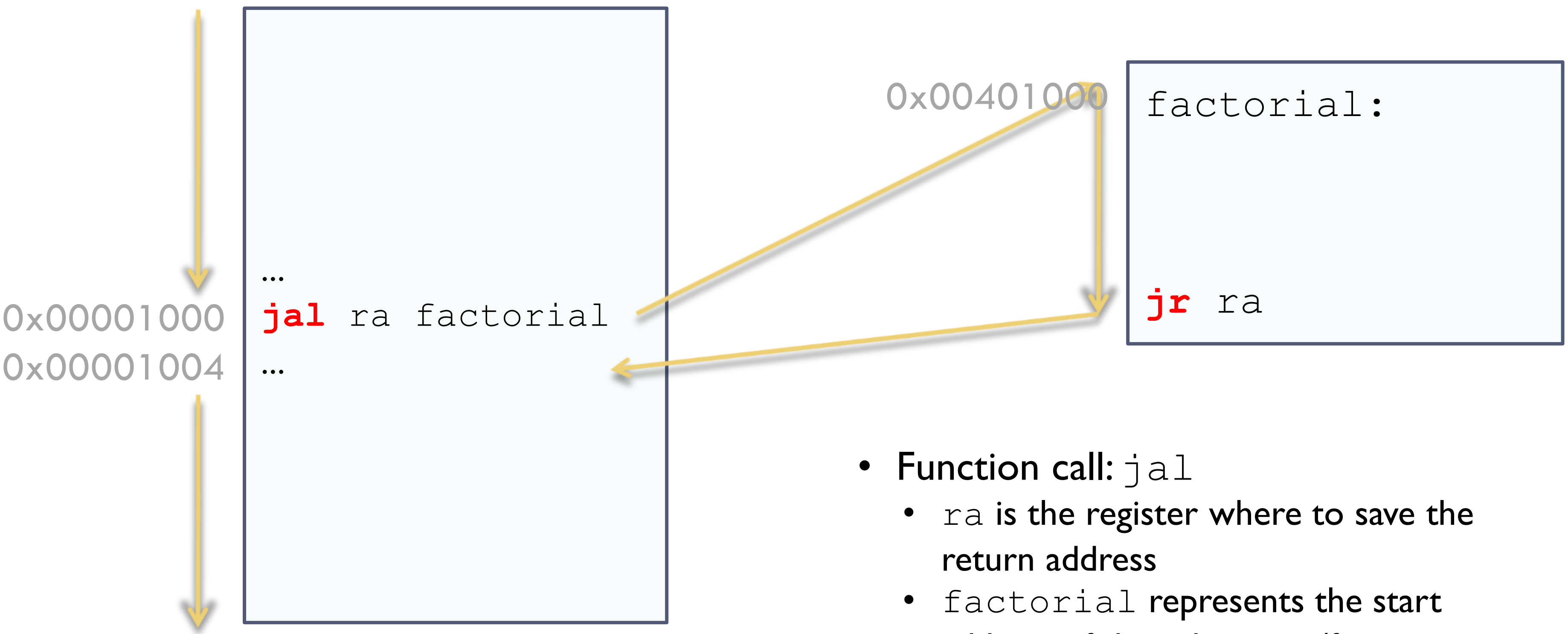
```
int factorial(int x) {
  int i;
  int r=1;
  for (i=1;i<=x;i++) {
    r*=i;
  }
  return r;
}
…
r1 = factorial(3) ;
…
```

- A high-level function (procedure, method, subroutine) is a subprogram that performs a specific task when invoked.
  - Receives input arguments or parameters
  - Returns some result

```
factorial:
    mv   t0 a0
    li   v0 1
b1: beq  t0 zero f1
    mul  v0 v0 t0
    addi t0 t0 -1
    j     b1
f1: jr ra
…
li   a0 3
jal ra factorial
…
```

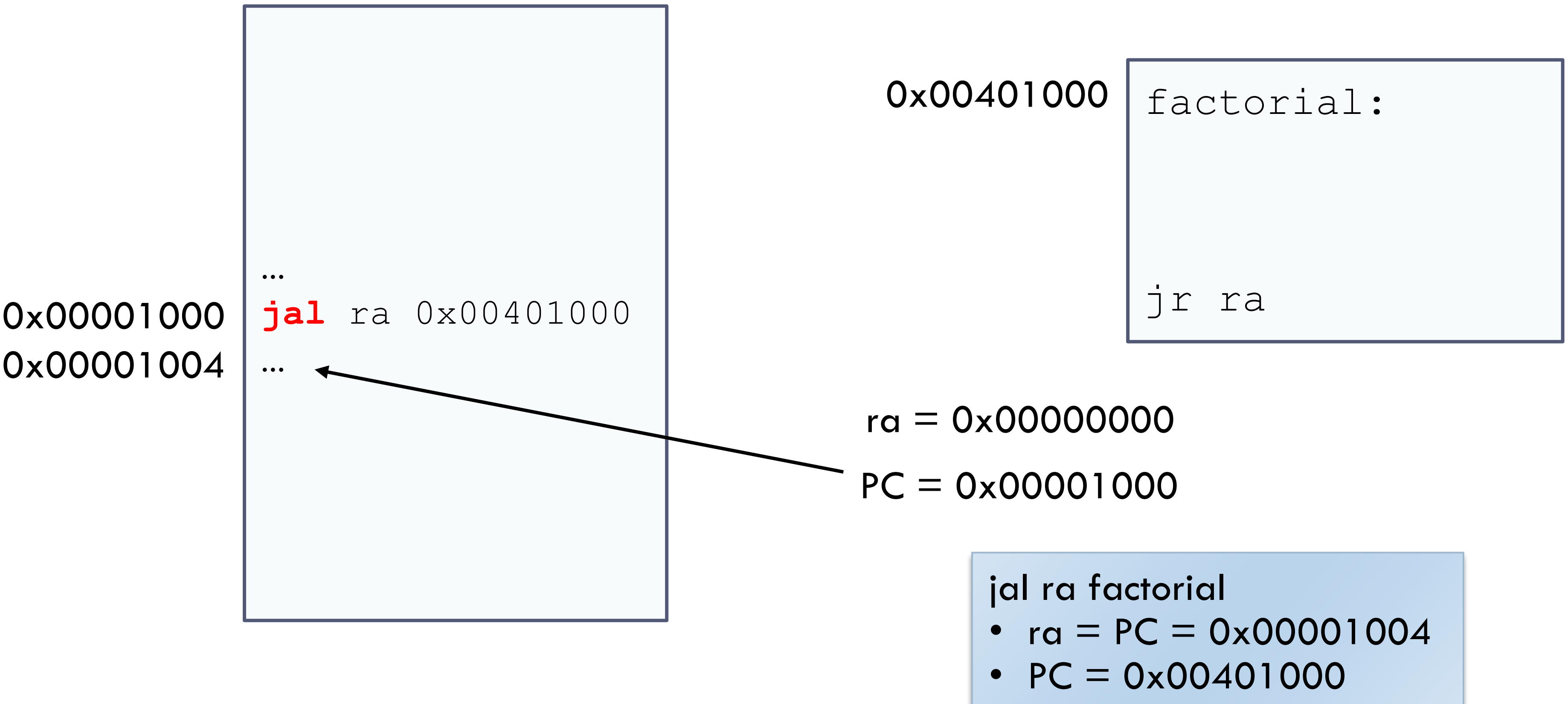
- In assembler, a function (subroutine) is associated with a label in the first instruction of the function
  - Symbolic name that denotes its starting address.
  - Memory address where the first instruction (of function) is located

Function calls in RISC-V

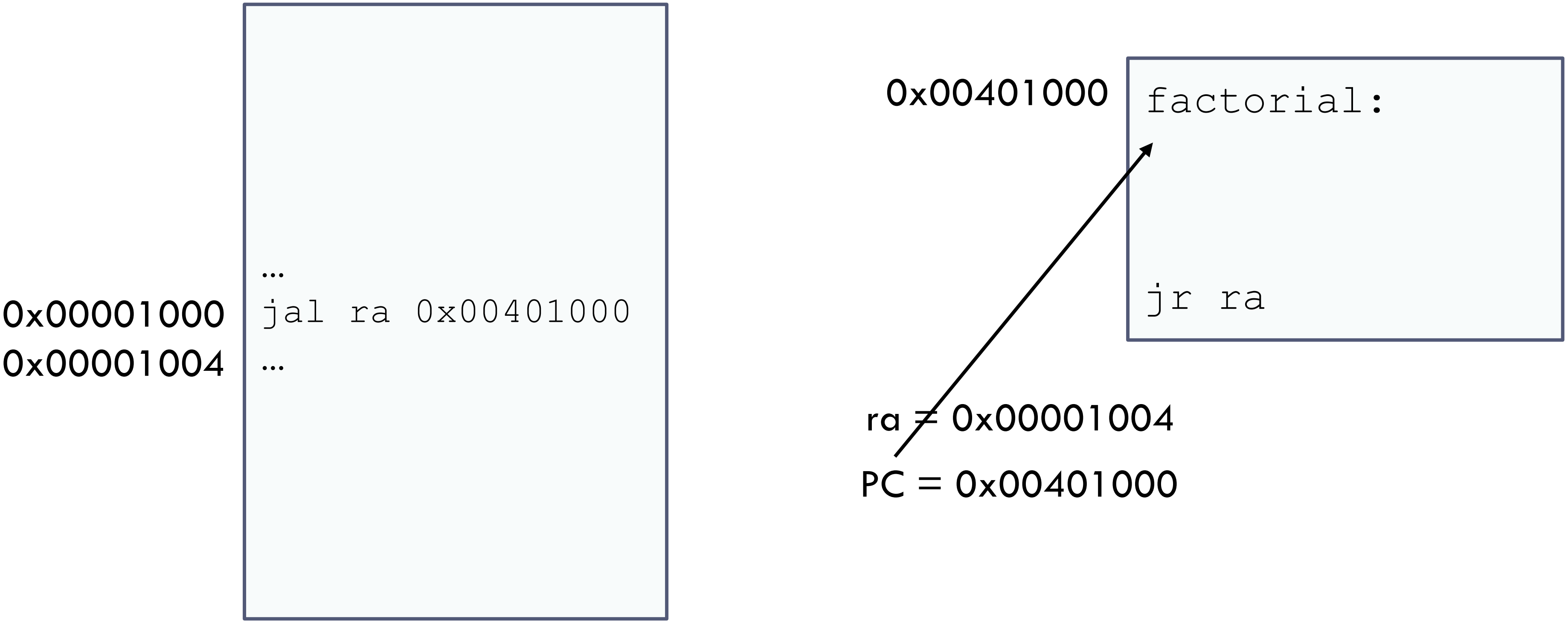


- Function call: `jal`
  - `ra` is the register where to save the return address
  - `factorial` represents the start address of the subroutine/function
- Return from function: `jr`

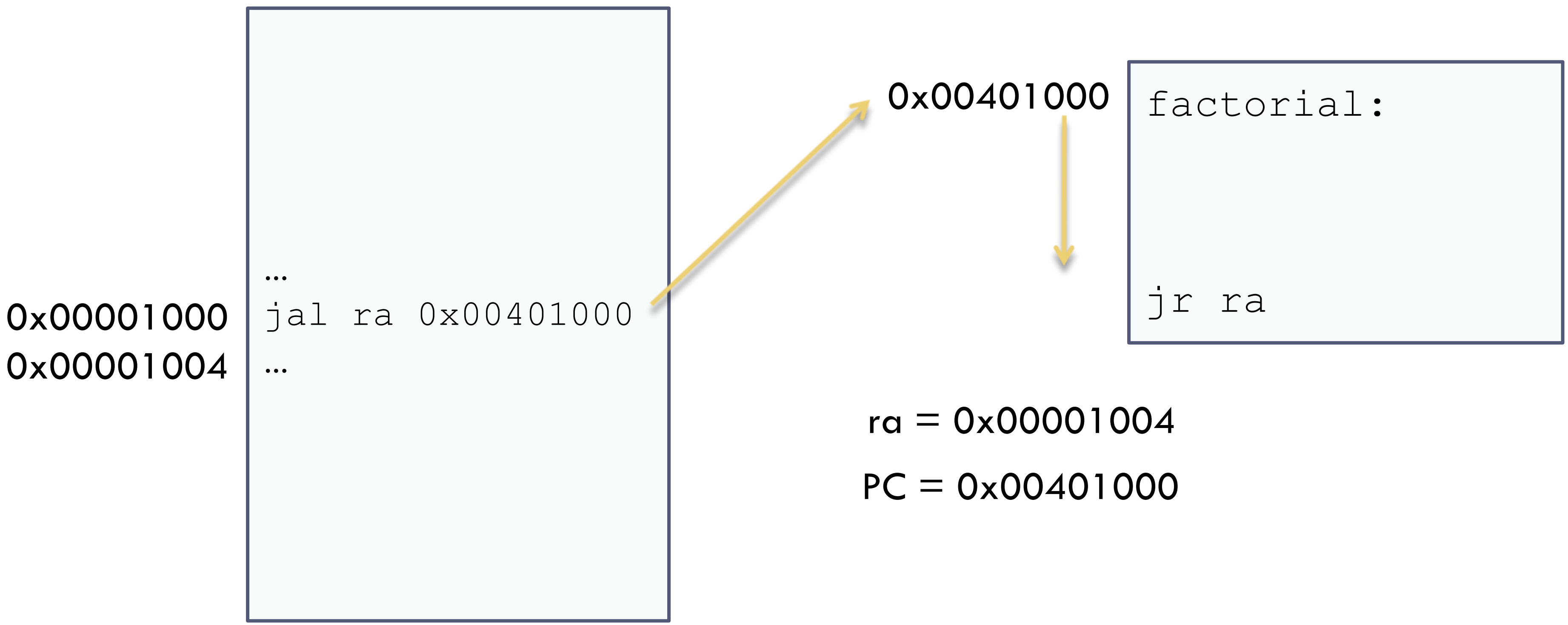
Function calls in RISC-V



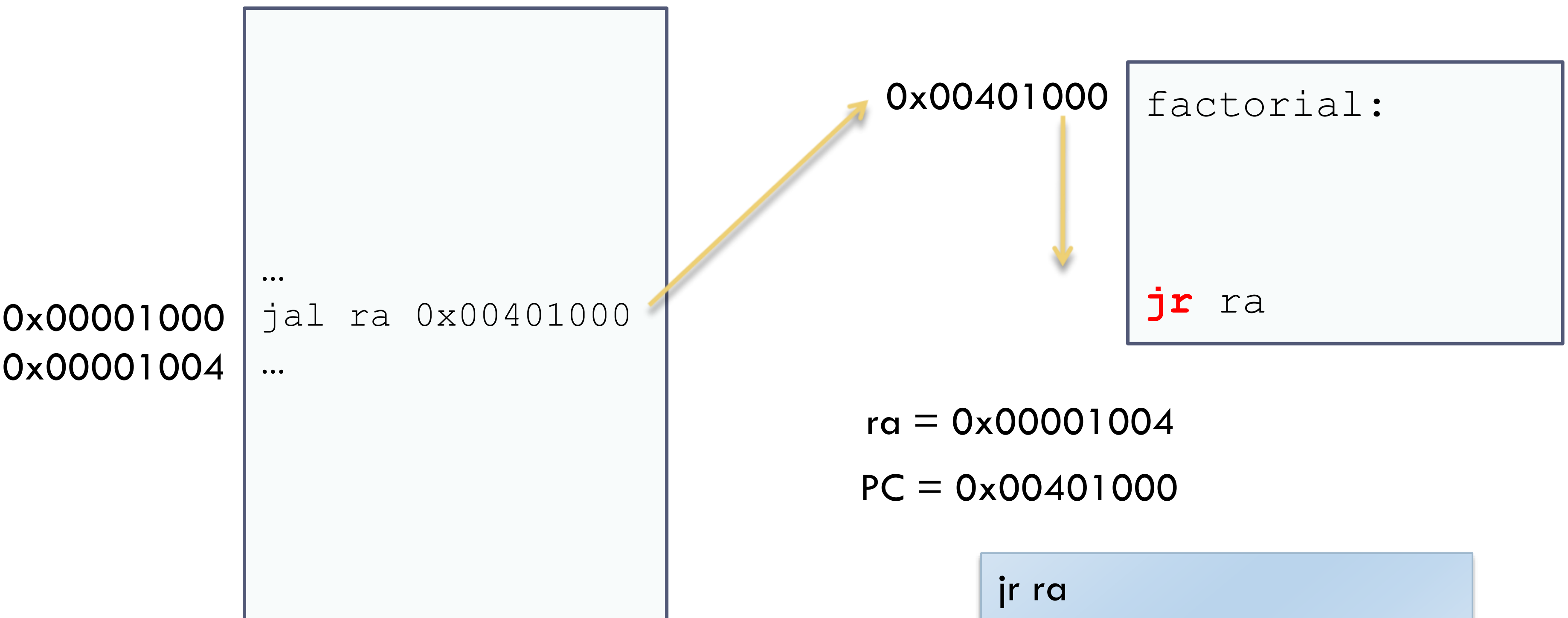
Function calls in RISC-V



Function calls in RISC-V



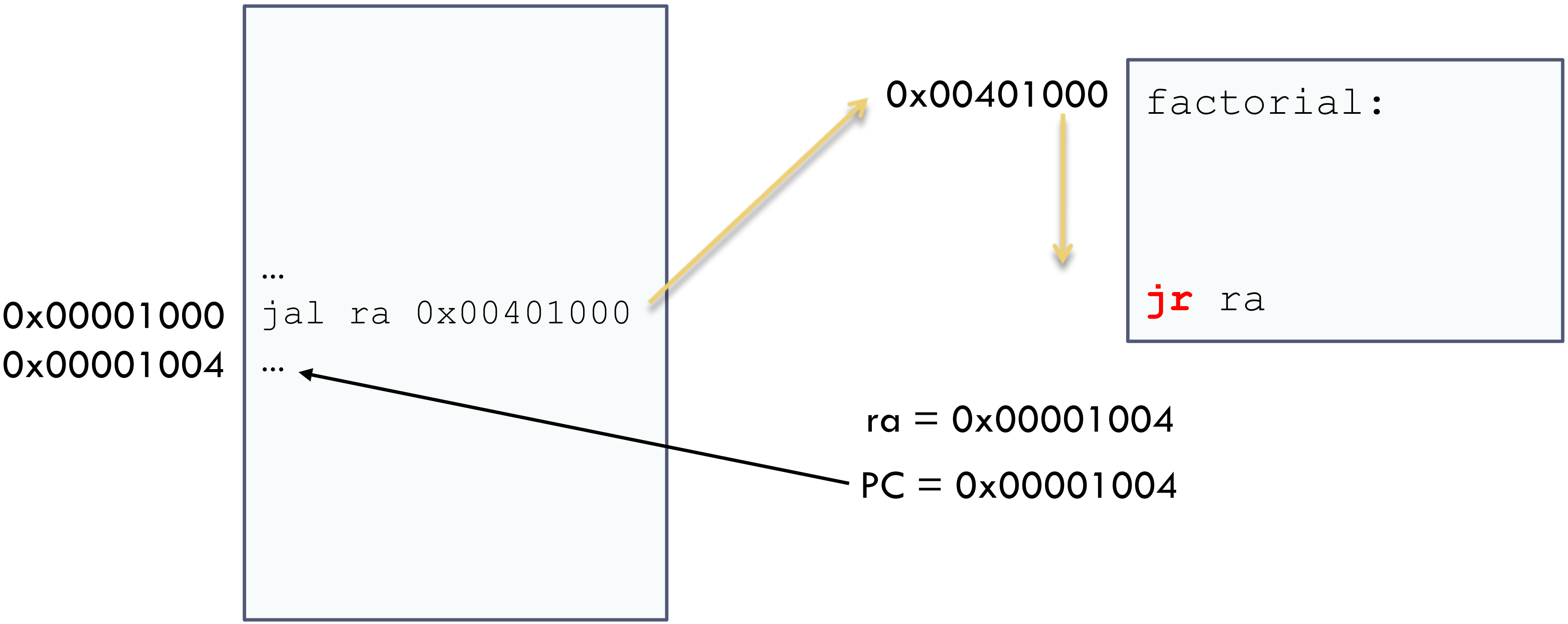
Function calls in RISC-V



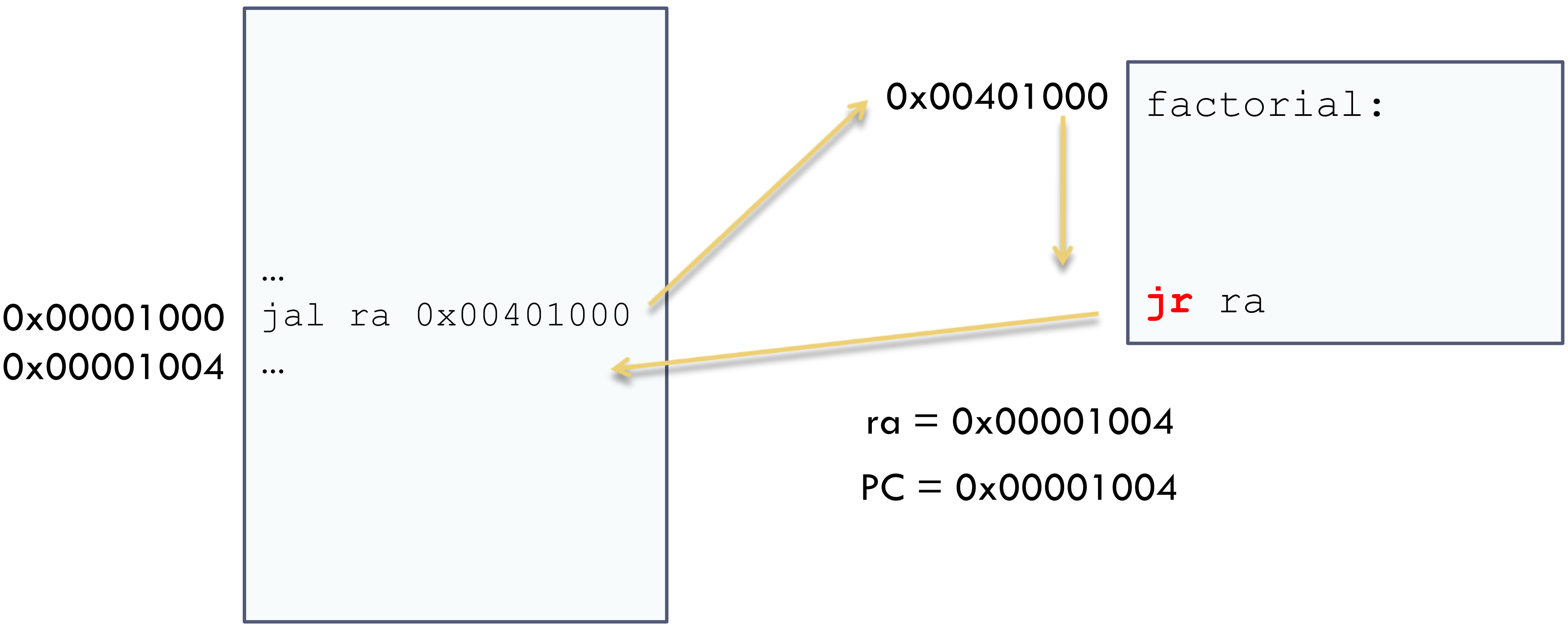
jr ra

- PC = ra = 0x00001004

Function calls in RISC-V



Function calls in RISC-V



`jal/jr` instructions

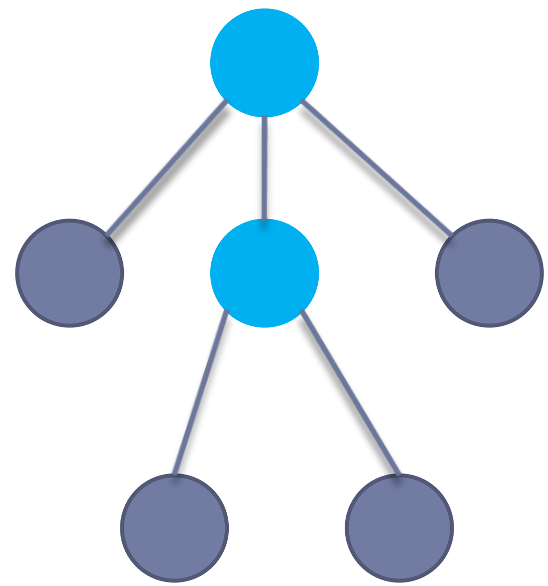
Subroutines / Functions		
`jal reg2, label`	`reg2 = PC` `PC   = label`	• Loads the contents of PC into the reg2 register. When executing the jal instruction PC points to the first byte of the following instruction. • Calculates and loads into PC the memory address that the label represents. The next instruction to be executed will be the one pointed by PC.
`jr  reg1`	`PC = reg1`	• Saves to PC the value stored in the reg1 registry.

Contents

- Basic concepts on assembly programming
- RISC-V 32 assembly language, memory model and data representation
- Instruction formats and addressing modes
- Procedure calls and stack convention
  - How do you call a function/subroutine?
  - Where is the return address stored in non-terminal routines?
  - What is the parameter passing convention?
  - What is the register use agreement?
  - What are the local variables like?

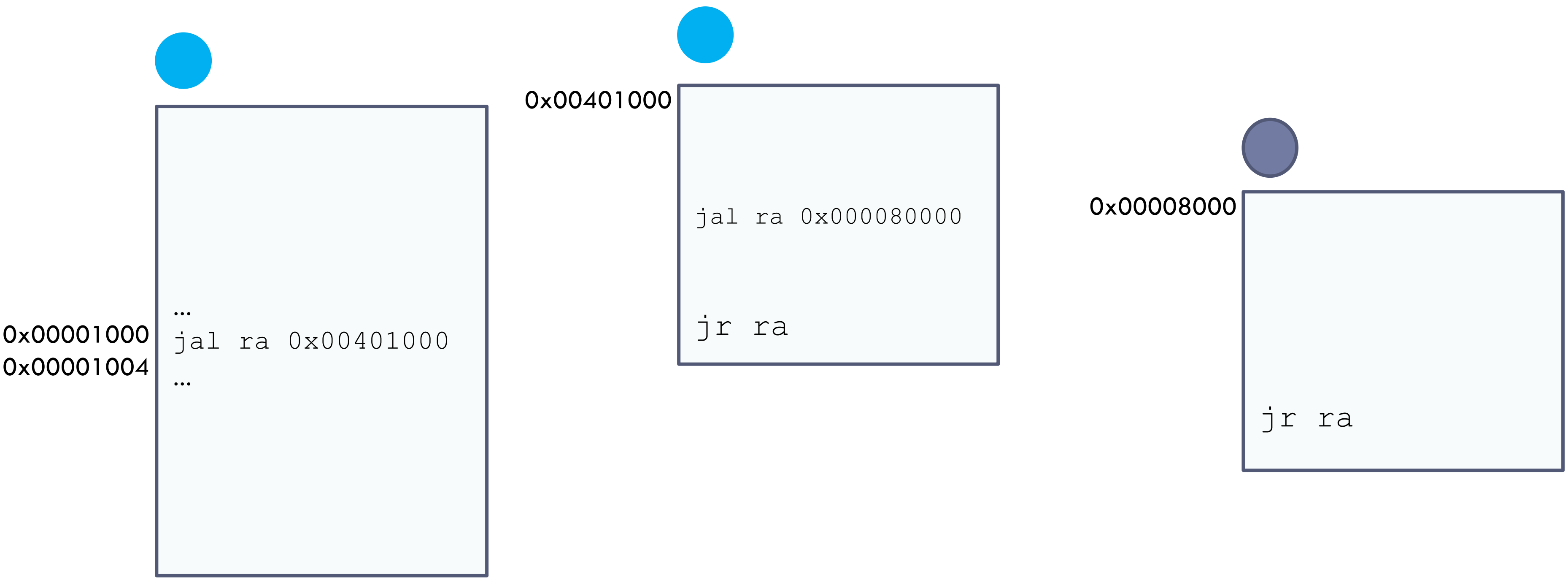
Types of subroutines

- Terminal subroutine.
  - **Does not** invoke any other subroutine.
- Non-terminal subroutine.
  - If you invoke any other subroutine.

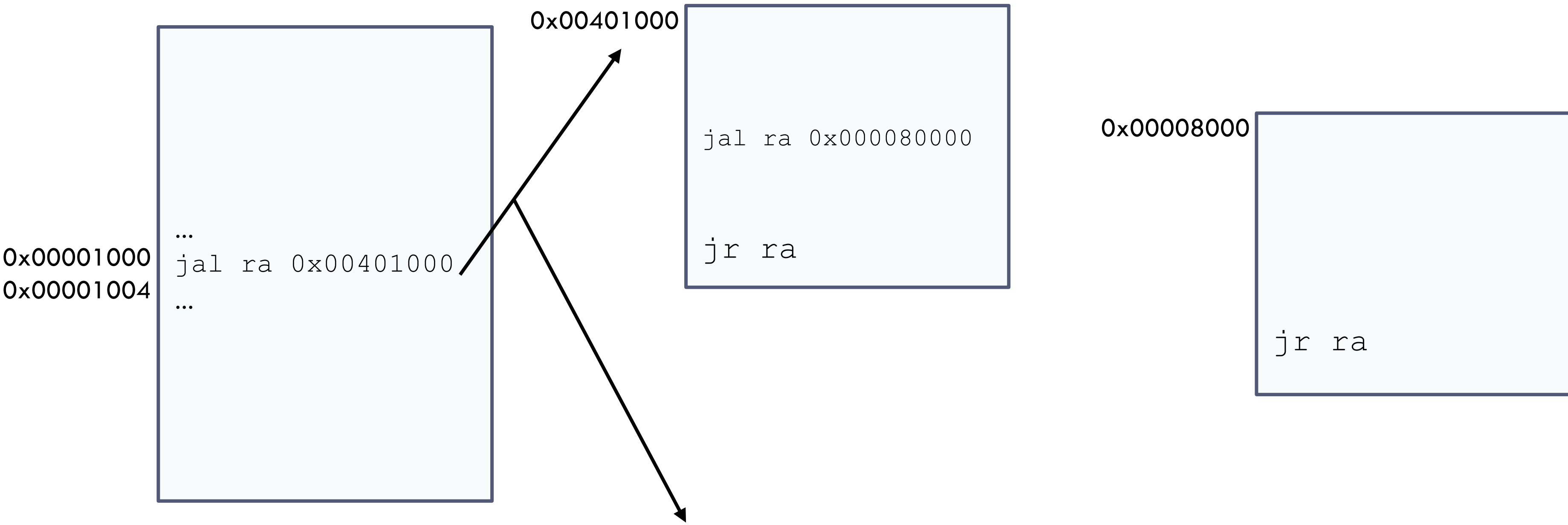


Félix García Carballeira, Alejandro Calderón Mateos

Problem in non-terminal subroutines



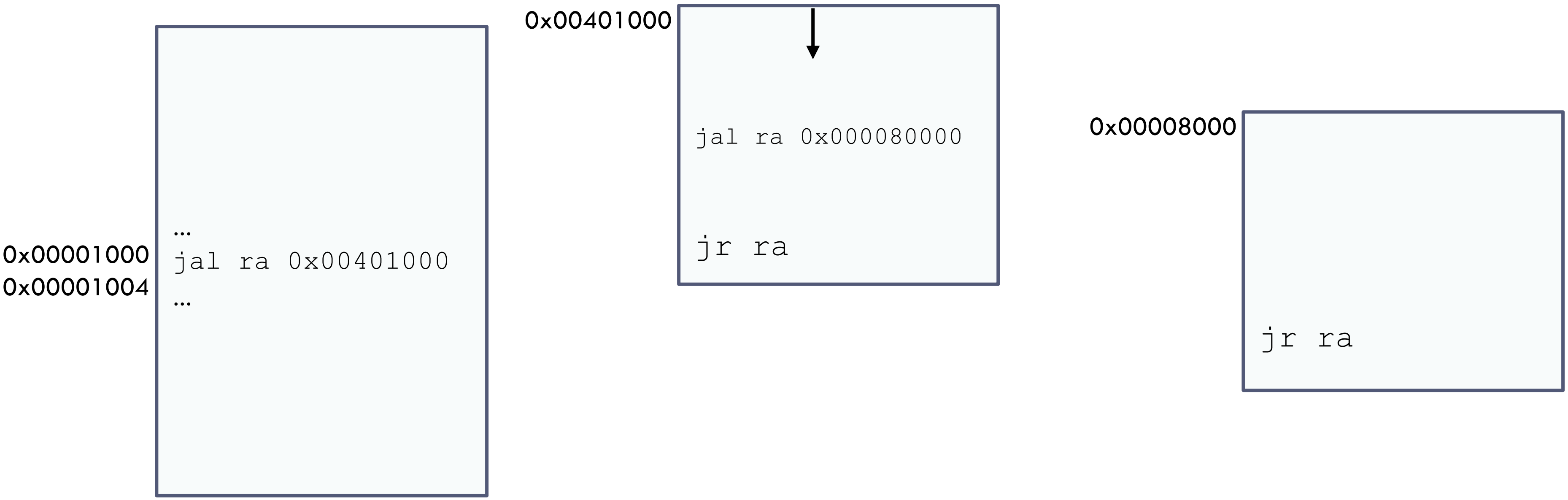
Problem in non-terminal subroutines



ra = PC = 0x00001004
PC = 0x00401000

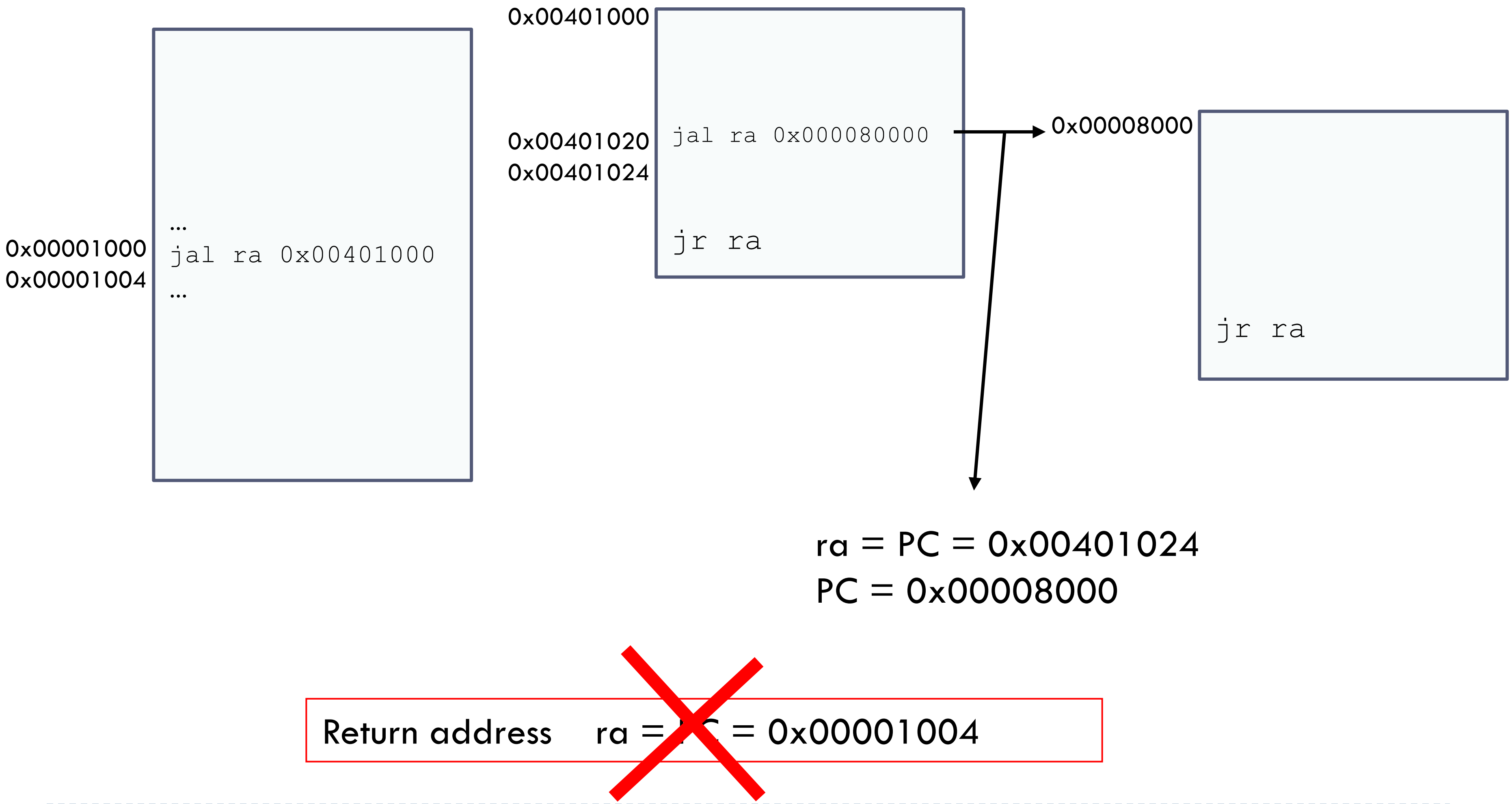
Return address ra = PC = 0x00001004

Problem in non-terminal subroutines

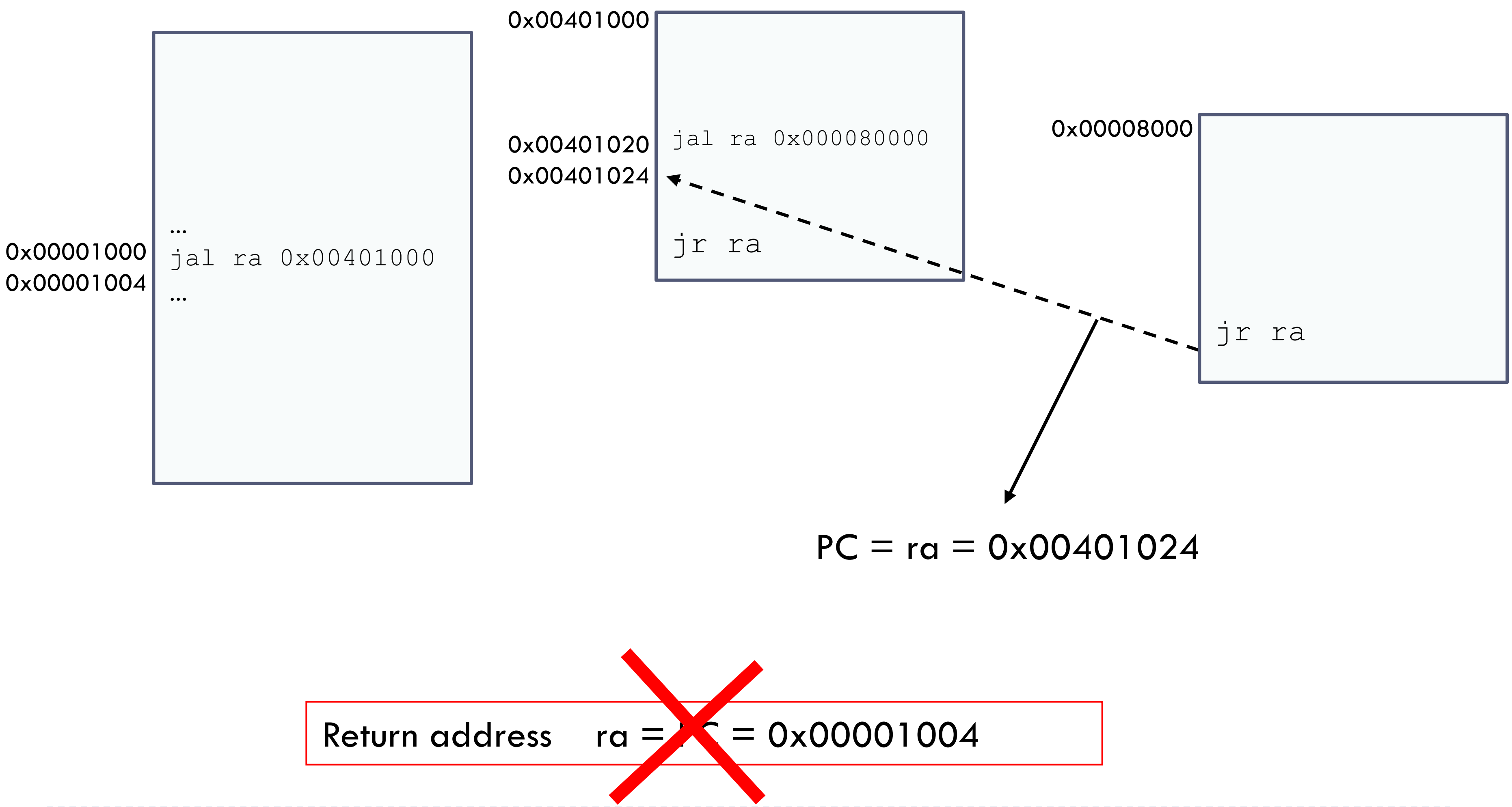


Return address ra = PC = 0x00001004

Problem in non-terminal subroutines

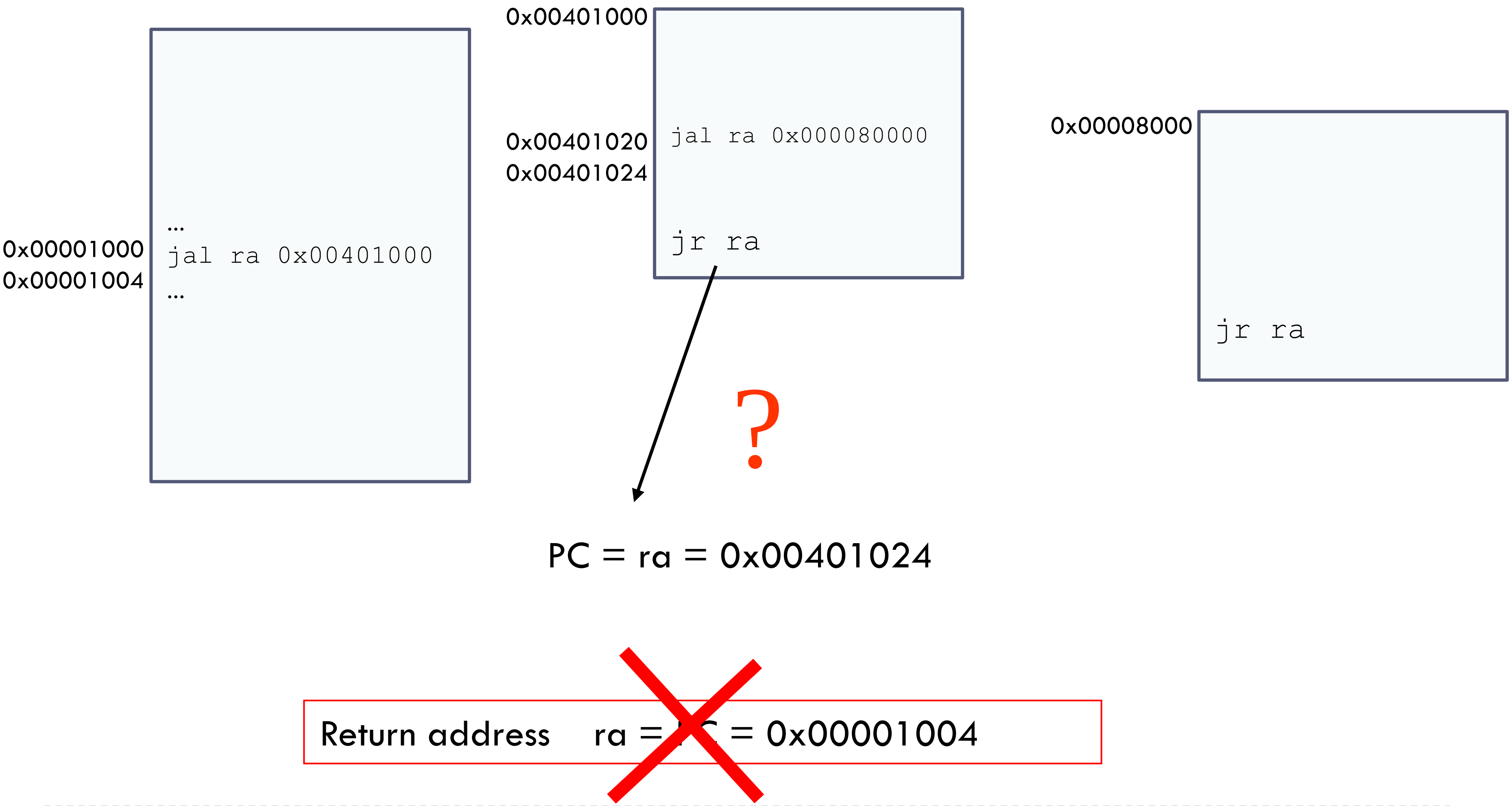


Problem in non-terminal subroutines

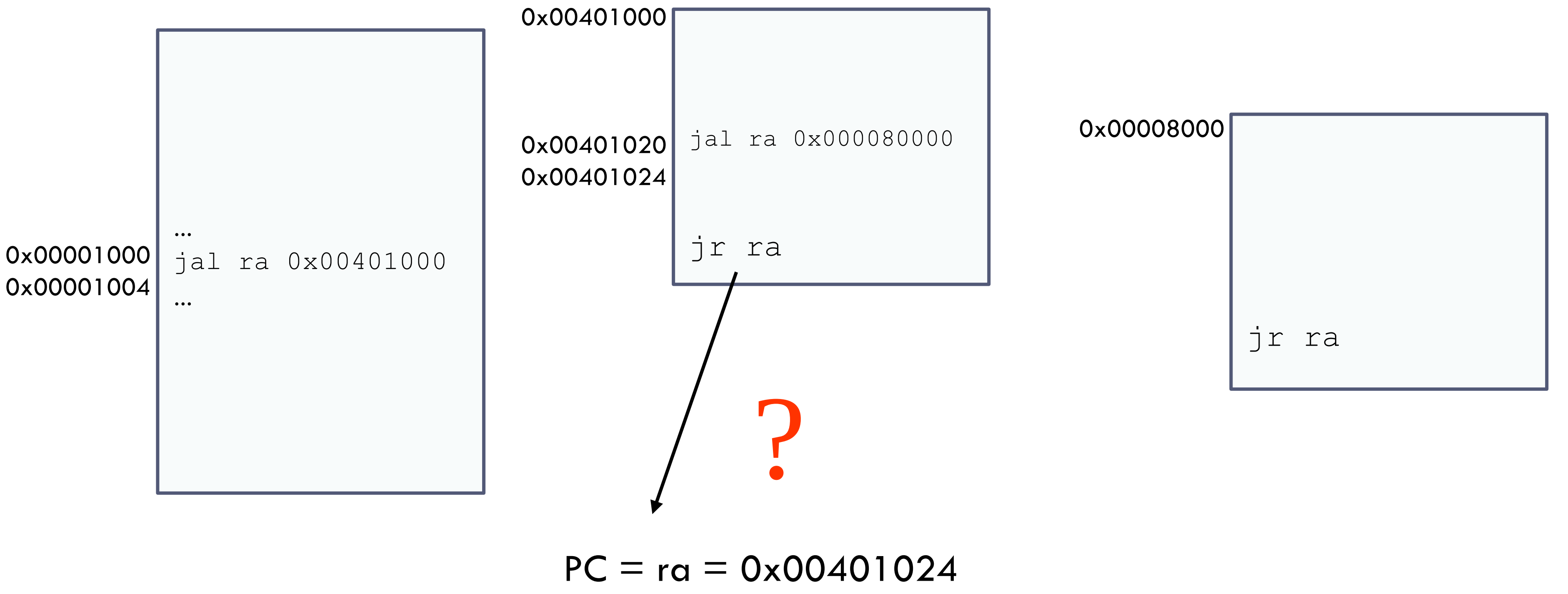


Félix García Carballeira, Alejandro Calderón Mateos

Problem in non-terminal subroutines



Problem in non-terminal subroutines



Return address has been lost

Where to store the return address?

- Computers have two storage elements:
  - Registers
  - Memory

- Registers: The number of registers is limited, so registers cannot be used (e.g.: recursive calls)
- Memory: Return addresses are stored in main memory
  - In a program area called stack

Stack, jal y jr…

IMPORTANT!

```
no_terminal:

    addi sp sp -4
    sw   ra 0(sp)

    li   t0, 8
    li   s0, 9

    ...

    jal  ra, función

    ...

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

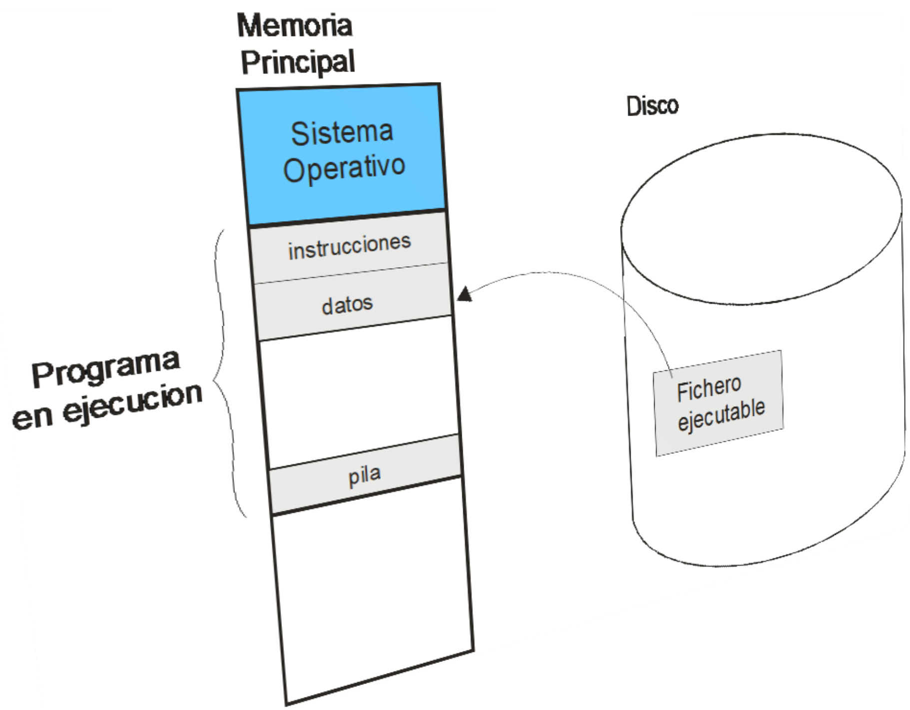
Ra is saved in the stack at the beginning

The value before "jr ra" is restored.

Program execution

Reminder

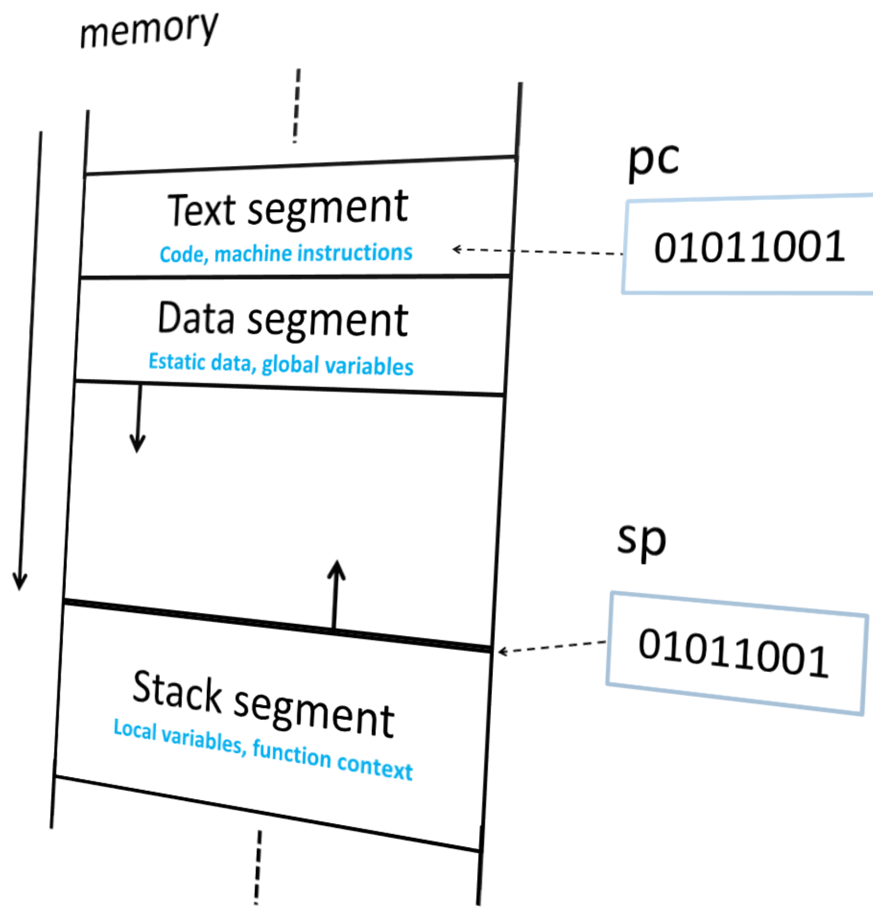
```
no_terminal:

    addi sp sp -4
    sw   ra 0(sp)

    li   t0, 8
    li   s0, 9

    ...

    jal  ra, función

    ...

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



Program execution

Reminder

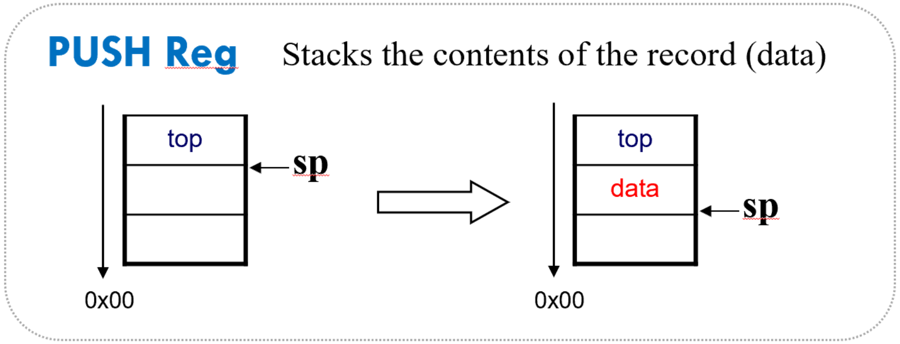
```
no_terminal:

    addi sp sp -4
    sw   ra 0(sp)

    li   t0, 8
    li   s0, 9

    ...

    jal  ra, función

    ...

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

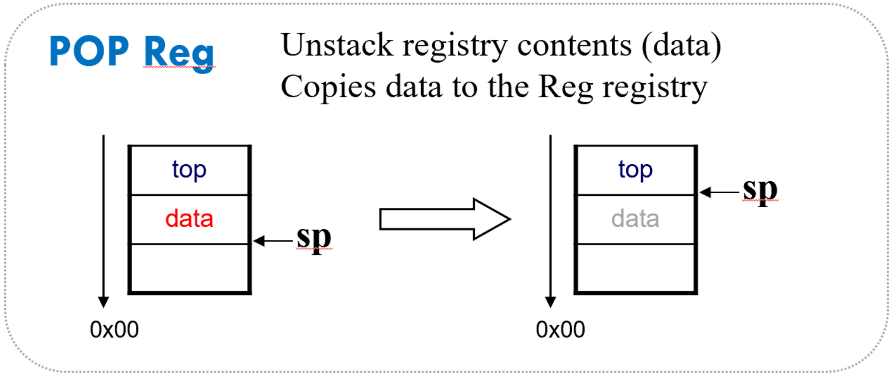


Program execution

Reminder

```
no_terminal:

    addi sp sp -4
    sw   ra 0(sp)

    li   t0, 8
    li   s0, 9

    ...

    jal  ra, función

    ...

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

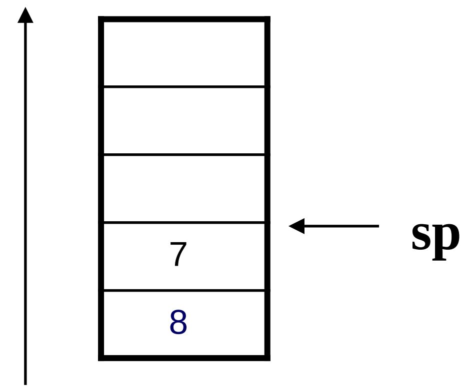




PUSH operation in RISC-V

Reminder

```
…
li   t2,  9
addi sp, sp, -4
sw   t2 0(sp)
…
```

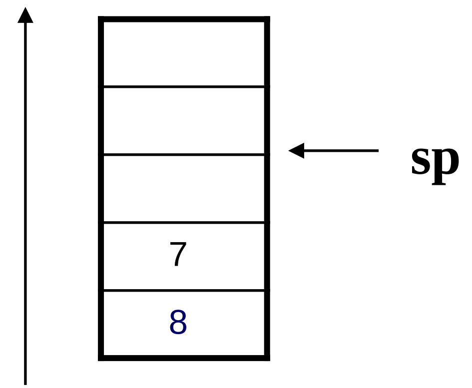


- Initial state:
  - The stack pointer register (sp) points to the last element at the top of the stack
  - The t2 register holds the value of 9

PUSH operation in RISC-V

Reminder

```
…
li    t2,  9
addi  sp,  sp,  -4
sw    t2  0(sp)
…
```

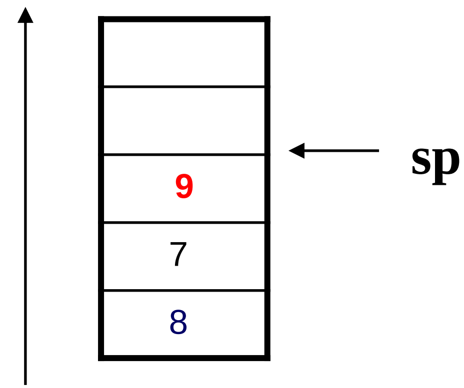


- Subtract 4 to stack pointer to insert a new word in the stack
  - addi sp, sp, -4

PUSH operation in RISC-V

Reminder

```
…
li    t2,  9
addi sp, sp, -4
sw    t2  0(sp)
…
```

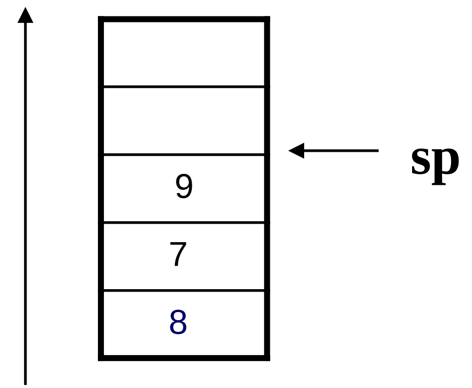


- The contents of register t2 are inserted at the top of the stack:
  - sw t2 0(sp)

POP operation in RISC-V_{32}

Reminder

```
…
lw   t2 0(sp)
addi sp, sp, 4
…
```

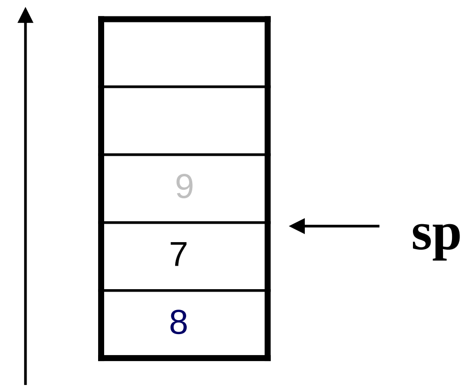


- The data stored at the top of the stack (9) is copied to t2.
  - lw t2 0(sp)

Reminder

POP operation in RISC-V

```
…
lw    t2 0(sp)
addi sp, sp, 4
…
```



- The sp register is updated to point to the new top of the stack.
  - addi sp, sp, 4
- The unstacked data (9) is still in memory but will be overwritten in future PUSH (or similar memory access) operation

Stack: use of consecutive push and pop

a) unstacking in reverse order of stacking

```
push a0
push t1
push t2
push s2

...



pop s2
pop t2
pop t1
pop a0
```

Stack: use of consecutive push and pop

b) it is possible to add sums by operation or to join sums together

```
push a0
push t1
push t2
push s2
```

...

```
pop s2
pop t2
pop t1
pop a0
```

```
addi sp sp -4
sw   a0   0(sp)
addi sp sp -4
sw   t1   0(sp)
addi sp sp -4
sw   t2   0(sp)
addi sp sp -4
sw   s2   0(sp)
```

...

```
lw s2 0(sp)
addi sp sp 4
lw t2 0(sp)
addi sp sp 4
lw t1 0(sp)
addi sp sp 4
lw a0 0(sp)
addi sp sp 4
```

Stack: use of consecutive push and pop

b) it is possible to add sums by operation or to join sums together

```
push a0
push t1
push t2
push s2
```

```
...
```

```
pop s2
pop t2
pop t1
pop a0
```

```
addi sp sp -16
sw   a0  12(sp)
sw   t1   8(sp)
sw   t2   4(sp)
sw   s2   0(sp)
```

```
...
```

```
lw   s2   0(sp)
lw   t2   4(sp)
lw   t1   8(sp)
lw   a0  12(sp)
addi sp  sp  16
```

Contents

- Basic concepts on assembly programming
- RISC-V 32 assembly language, memory model and data representation
- Instruction formats and addressing modes
- Procedure calls and stack convention
  - How do you call a function/subroutine?
  - Where is the return address stored in non-terminal routines?
  - What is the parameter passing convention?
  - What is the register use agreement?
  - What are the local variables like?

Parameters and registers agreement

```
no_terminal:

    addi sp sp -4
    sw   ra 0(sp)

    li   t0, 8
    li   s0, 9

    ...

    jal  ra, función


    ...

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

In which registers are the parameters passed and the results returned?

Parameter passing agreement

- When programming in assembler, a convention is defined that specifies how arguments are passed and how registers are treated.
- Compilers define this convention for a given architecture.
- A simplified version of the conventions used by compilers will be used in this course.

Simplified agreement (RISC-V)

IMPORTANT!

- **Parameter passing**
  - Integer parameters (char, int) are passed in **a0 … a7**
    - If you need to pass more than eight parameters,
      first eight parameters in a0 … a7 and the rest in the stack
  - Float parameters are passed in **fa0 … fa7**
    - If you need to pass more than eight parameters, the rest in the stack
  - Double parameters are passed in **fa0 … fa7**
    - If you need to pass more than eight parameters, the rest in the stack
- **Return** on **results**
  - **a0** and **a1** are used for integer type values.
  - **fa0** and **fa1** are used for float and double values.
  - In case of structures or complex values, they must be left on the stack. The space is reserved by the calling function (caller).

Contents

- Basic concepts on assembly programming
- RISC-V 32 assembly language, memory model and data representation
- Instruction formats and addressing modes
- Procedure calls and stack convention
  - How do you call a function/subroutine?
  - Where is the return address stored in non-terminal routines?
  - What is the parameter passing convention?
  - What is the register use agreement?
  - What are the local variables like?

Parameters and registers agreement

```
no_terminal:

    addi sp sp -4
    sw   ra 0(sp)

    li   t0, 8
    li   s0, 9

    ...

    jal  ra, función

    ...

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

What are the values of the t0 and s0 registers on return?

Convention for using registers (RISC-V)

IMPORTANT!

Name	Usage	Preserving value
zero	Constant 0	No
ra	Return address (subrutines)	**Yes**
sp	Stack pointer	**Yes**
gp	Global pointer	No
tp	Thread pointer	No
t0 … t6	Temporal	No
s0/fp	Temporal / Frame pointer	**Yes**
s1 … s11	Temporal	**Yes**
a0 … a7	Argumento de entrada para rutinas	No

Name	Usage	Preserving value
ft0 … ft11	Temporals	No
fs0 … fs11	Temporals to save	**Yes**
fa0 … fa1	Arguments/return	No
fa2 … fa7	Arguments	No

Register agreement

```
li      t0, 8
li      s0, 9

li      a0, 7   # parameter
jal     ra, función

...
```

According to the agreement:

- `s0` will still be 9,
- there is no guarantee that `t0` is 8
- nor that `a0` is 7 after the execution of function.

  - If we want `t0` to continue to be 8, it must be saved on the stack before each function call

Félix García Carballeira, Alejandro Calderón Mateos

Register agreement

```
li      t0, 8
li      s0, 9

addi    sp, sp, -4
sw      t0, 0(sp)

li      a0, 7   # parameter
jal     ra, función

lw      t0, 0(sp)
addi    sp, sp, 4

...
```

← It is saved in the stack before the call

← Value is restored after calling

Parameters and registers agreement

summary

```
no_terminal:

      li  s0, 9
      li  t0, 8

      li  a0, 7    # parameter
      jal ra, función

      jr ra
```

Parameters and registers agreement

summary

ra	DO preserve
sp	
s0 … s11	

```
no_terminal:

      addi sp sp -8
      sw   ra 0(sp)
      sw   s0 4(sp)

   li  s0, 9
   li  t0, 8

   li  a0, 7   # parameter
   jal ra, función

      lw   s0 4(sp)
      lw   ra, 0(sp)
      addi sp, sp, 8

   jr ra
```

Parameters and registers agreement

summary

```
no_terminal:

        addi sp sp -8
        sw   ra 0(sp)
        sw   s0 4(sp)

     li  s0, 9
     li  t0, 8

        addi    sp, sp, -4
        sw      t0, 0(sp)

     li  a0, 7    # parameter
     jal ra, función

        lw      t0, 0(sp)
        addi    sp, sp, 4

        lw   s0 4(sp)
        lw   ra, 0(sp)
        addi sp, sp, 8

     jr ra
```

DO preserve
ra
sp
s0 … s11

DONT preserve
gp
tp
t0 … t6
a0 … a7

ADVANCE

Parameters and registers agreement

summary

```
no_terminal:

     addi sp sp -12
     sw   ra 0(sp)
     sw   s0 4(sp)

  li  s0, 9
  li  t0, 8

      addi    sp, sp, -4
      sw      t0, 0(sp)

  li  a0, 7   # parameter
  jal ra, función

      lw      t0, 0(sp)
      addi    sp, sp, 4

     lw   s0 4(sp)
     lw   ra, 0(sp)
     addi sp, sp, 12

  jr ra
```

- Local variables in stack.
- Example reserving 4 bytes (an integer)

DO preserve
ra
sp
s0 … s11

DONT preserve
gp
tp
t0 … t6
a0 … a7

Example

(1) Suppose a high-level language code

```
int main() {
  int z;
  z=factorial(5);
  print_int(z);
  .
  .
  .
}
```

```
int factorial(int x) {
  int i;
  int r=1;
  for (i=1;i<=x;i++) {
    r*=i;
  }
  return r;
}
```

Example

(2) Analyze how to pass the arguments

- arguments/parameters are placed in a0 … a7
  - `z=factorial(5)` has an input parameter in a0
- Results are collected in a0, a1
  - `z=factorial(5)` returns a result in a0
- If you need to pass more than eight parameters,
  (1) the first eight in registers a0…a7 and
  (2) rest on the stack
  - No more than eight parameters are required

Example

(3) Translate to assembly language

```
int main() {
  int z;
  z=factorial(5);
  print_int(z);
  . .
}

int factorial(int x) {
  int i;
  int r=1;
  for (i=1;i<=x;i++) {
       r*=i;
  }
  return r;
}
```

```
main:     # factorial(5)
          li   a0,  5        # arg.
          jal ra factorial   # invoke
          mv   a0 a0         # result
          # print_int(z)
          li a7, 1
          ecall
          ...

factorial: li    s1, 1    #s1 for r
           li    s0, 1    #s0 for i
    loop1: bgt   s0, a0, end1
           mul   s1, s1, s0
           addi  s0, s0, 1
           j     loop1
     end1: mv    a0, s1   #result
           jr    ra
```

- The parameter is passed in a0
- The result is returned in a0

Example

(4) Analyze the registers modified (1/2)

```c
int main() {
  int z;
  z=factorial(5);
  print_int(z);

int factorial(int x) {
  int i;
  int r=1;
  for (i=1;i<=x;i++) {
      r*=i;
  }
  return r;
}
```

```
main:      # factorial(5)
           li  a0,  5          # arg.
           jal ra factorial    # invoke
           mv  a0 a0           # result
           # print_int(z)
           li a7, 1
           ecall
           ...

factorial: li   s1, 1    #s1 for r
           li   s0, 1    #s0 for i
    loop1: bgt  s0, a0, end1
           mul  s1, s1, s0
           addi s0, s0, 1
           j    loop1
     end1: mv   a0, s1   #result
           jr   ra
```

- main is non-terminal (there is a jal... that calls another subroutine).
- It is therefore modified ra

Example

(4) Analyze the registers modified (2/2)

```
int ma
  int
  z=fa
  prin
  . .
}
```

- The factorial function uses (modifies) registers s0, s1, s2 and s3.
- If these registers are modified within the function, it could affect the function that made the call (the main function).
- Therefore, the factorial function must save the value of these registers on the stack at the beginning and restore them at the end.

```
int factorial(int x) {
  int i;
  int r=1;
  for (i=1;i<=x;i++) {
       r*=i;
  }
  return r;
}
```

```
...
factorial: li    s1, 1   #s1 for r
           li    s0, 1   #s0 for i
    loop1: bgt   s0, a0, end1
           mul   s1, s1, s0
           addi  s0, s0, 1
           j     loop1
     end1: mv    a0, s1  #result
           jr    ra
```

Example

(5) Store registers in stack (1/2)

```
int main() {
   int z;
   z=factorial(5);
```

- It is necessary to save ra.
  - The main routine is non-terminal
- Neither s0…s11 nor t0…t6 should be saved

```
int factorial(int x) {
   int i;
   int r=1;
   for (i=1;i<=x;i++) {
         r*=i;
   }
   return r;
}
```

```
main:        addi sp sp -4
             sw ra 0(sp)
             # factorial(5)
             li  a0,  5          # arg.
             jal ra factorial    # invoke
             mv  a0 a0           # result
             # print_int(z)
             li a7, 1
             ecall
             ...
             lw ra 0(sp)
             add sp sp 4
             jr ra

factorial:  addi   sp, sp, -8
            sw     s0, 4(sp)
            sw     s1, 0(sp)
            li     s1, 1    # s1 para r
            li     s0, 1    # s0 para i
    loop1:  bgt    s0, a0, end1
            mul    s1, s1, s0
            addi   s0, s0, 1
            j      loop1
    end1:   mv     a0, s1   # result
            lw     s1, 0(sp)
            lw     s0, 4(sp)
            addi   sp, sp, 8
            jr     ra
```

Example

(5) Store registers in stack (2/2)

```
int main() {
  int z;
  z=factorial(5);
  print_int(z);
  . . .
}
```

```
main:       addi sp sp -4
            sw ra 0(sp)
            # factorial(5)
            li  a0,  5          # arg.
            jal ra factorial    # invoke
            mv  a0 a0           # result
            # print_int(z)
            li a7, 1
            ecall
            ...
            lw ra 0(sp)
            add sp sp 4
            jr ra
```

- It is not necessary to save ra.
  - The main routine is terminal
- s0, s1 stored on stack (they are modified)
  - Not necessary if using t0 and t1

```
      r*=i;
  }
  return r;
}
```

```
rial:  addi   sp, sp, -8
       sw     s0, 4(sp)
       sw     s1, 0(sp)
       li     s1, 1   # s1 para r
       li     s0, 1   # s0 para i
op1:   bgt    s0, a0, end1
       mul    s1, s1, s0
       addi   s0, s0, 1
       j      loop1
end1:  mv     a0, s1  # result
       lw     s1, 0(sp)
       lw     s0, 4(sp)
       addi   sp, sp, 8
       jr     ra
```

Example 2

```
int main()
{
   int z;

   z=f1(5, 2);

   pint(z);
}
```

```
int f1(int a, int b)
{
   int r;

   r = a+a+f2(b);
   return r;
}
```

```
int f2(int c)
{
    int s;

    s = c * c * c;
    return s;
}
```

Example 2. Body of main (1/3)

```c
int main()
{
  int z;

  z=f1(5, 2);

  pint(z);
}
```

```
main:
li    a0,  5     # first argument
li    a1,  2     # second argument
jal   ra, f1     # call
                 # result (a0)

li    a7, 1
ecall             # system call
                  # to print int


jr ra
```

Example 2. Analysis of main (2/3)

```
int main()
{
  int z;

  z=f1(5, 2);

  pint(z);
}
```

```
main:
li    a0,   5        # first argument
li    a1,   2        # second argument
jal   ra,  f1        # call
                     # result (a0)

li    a7, 1
ecall                 # system call
                      # to print int
```

- The parameters are passed in a0 and a1.
- The result is returned in a0
- Non-terminal routine (calls another routine)

Example 2. Adjustment of main (3/3)

```
int main()
{
  int z;

  z=f1(5, 2);

  pint(z);
}
```

```
main:
addi sp sp -4
sw ra 0(sp)

li   a0,  5      # first argument
li   a1,  2      # second argument
jal  ra, f1      # call
                 # result (a0)
li   a7, 1
ecall             # system call
                  # to print int

lw ra 0(sp)
addi sp sp 4
jr ra
```

Example 2. Body of f1 (1/3)

```
int f1 (int a, int b)
{
  int r;

  r = a + a + f2(b);
  return r;
}
```

```
f1: add   s0, a0, a0

    mv    a0, a1
    jal   ra f2
    add   a0, s0, a0

    jr    ra
```

```
int f2(int c)
{
   int s;

   s = c * c * c;
   return s;
}
```

Example 2. Analysis of f1 (2/3)

```
int f1 (int a, int b)
{
  int r;

  r = a + a + f2(b);
  return r;
}
```

```
f1: add   s0, a0, a0

    mv    a0, a1
    jal   ra f2
    add   a0, s0, a0

    jr    ra
```

```
int f2(int c)
{
   int s;

   s = c * c * c;
   return s;
}
```

- f1 modifies s0 and ra, therefore they are saved on the stack.
- The register ra is modified in the instruction "jal ra f2".
- The register a0 is modified by passing the argument to f2, but by convention the function f1 does not have to save it on the stack only if it uses it after making the call to f2

Example 2. Body of f1 storing in the stack the registers that are modified (3/3)

```
int f1 (int a, int b)
{
  int r;

  r = a + a + f2(b);
  return r;
}
```

```
int f2(int c)
{
   int s;

   s = c * c * c;
   return s;
}
```

```
f1: addi   sp, sp, -8
    sw     s0, 4(sp)
    sw     ra, 0(sp)

    add    s0, a0, a0
    mv     a0, a1
    jal    ra f2
    add    a0, s0, a0

    lw      ra, 0(sp)
    lw      s0, 4(sp)
    addu    sp, sp, 8

    jr     ra
```


Félix García Carballeira, Alejandro Calderón Mateos

Example 2. Body and analysis of f2

```
int f1 (int a, int b)
{
  int r;

  r = a + a + f2(b);
  return r;
}
```

```
int f2(int c)
{
   int s;

   s = c * c * c;
   return s;
}
```

```
f2: mul t0, a0, a0
    mul a0, t0, a0
    jr  ra
```

- The function f2 does not modify the register ra because it does not call any other function.
- The register t0 does not need to be stored because its value is not to be preserved according to convention.

Contents

- Basic concepts on assembly programming
- RISC-V 32 assembly language, memory model and data representation
- Instruction formats and addressing modes
- Procedure calls and stack convention
  - How do you call a function/subroutine?
  - Where is the return address stored in non-terminal routines?
  - What is the parameter passing convention?
  - What is the register use agreement?
  - What are the local variables like? (activation log)

Activation log
stack frame

- The stack frame or activation register is the mechanism used by the compiler to activate functions in high-level languages.
- The stack frame is built on the stack by the calling procedure/function and the called procedure/function.

Frame pointer

- The stack frame stores:
  - The **parameters entered** by the calling procedure, **if necessary**.
  - The **registers saved** by the function (including the `ra` register in case of non-terminal procedures/functions).
  - **Local variables**.

Félix García Carballeira, Alejandro Calderón Mateos

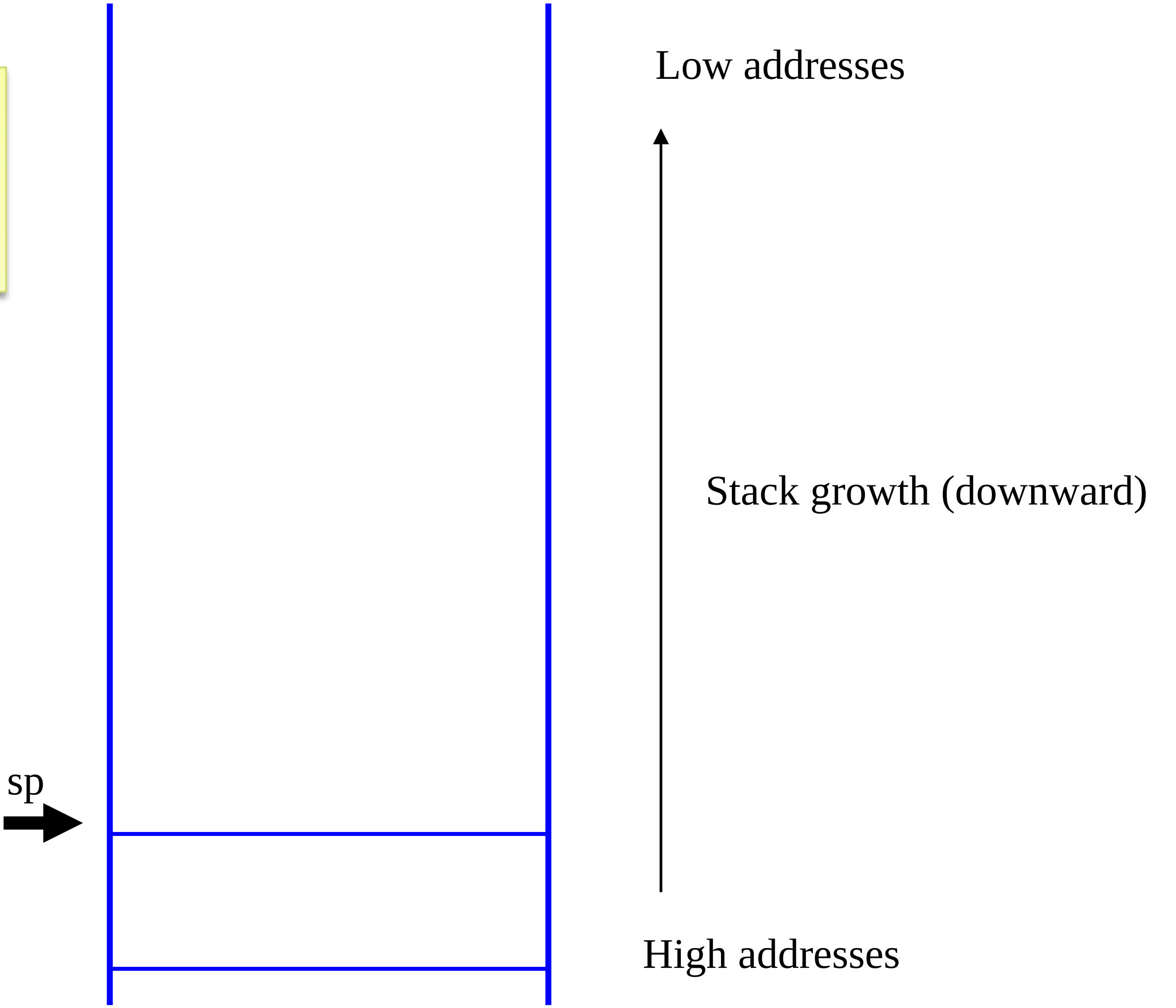
General function call steps
simplified version

Caller function	Calle function
Save the registers not preserved across the call (t_x, a_x, ..)	
Parameter passing + (if needed) allocation of space for values to be returned	
Make de call (jal)	
	Stacking frame allocation
	Save registers (ra, s_x)
	Function execution
	Restoring saved values
	Copy values to be returned in the space reserved by the caller
	Stack frame release (calle part)
	Return from function (jr ra)
Get returned values	
Restoration of saved registers, freeing the reserved stack space	

Félix García Carballeira, Alejandro Calderón Mateos

Construction of the stack frame
calle**r** function

The RISC-V convention will not be strictly followed for the sake of simplicity

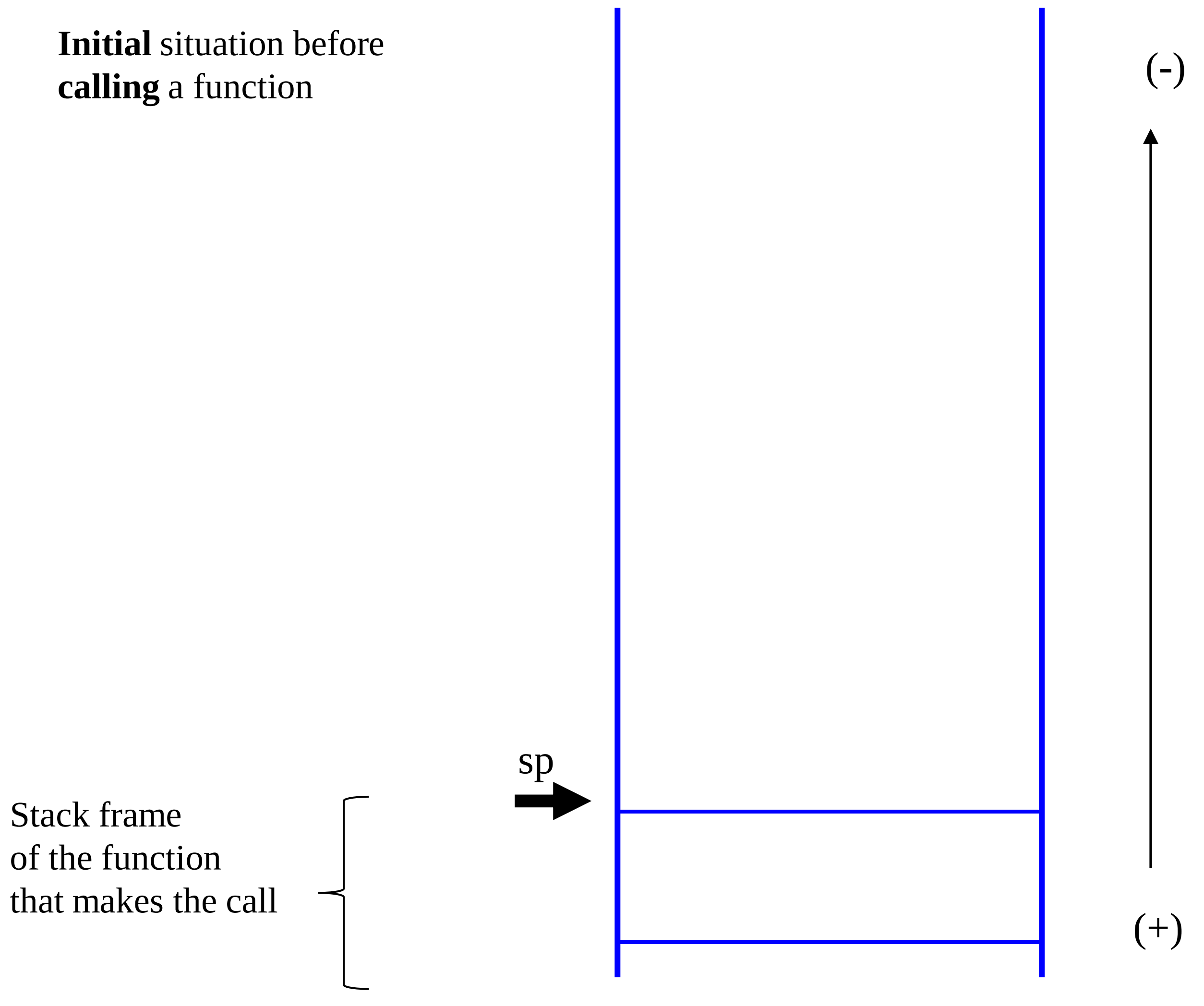


Félix García Carballeira, Alejandro Calderón Mateos

Construction of the stack frame

caller function

Initial situation before
calling a function

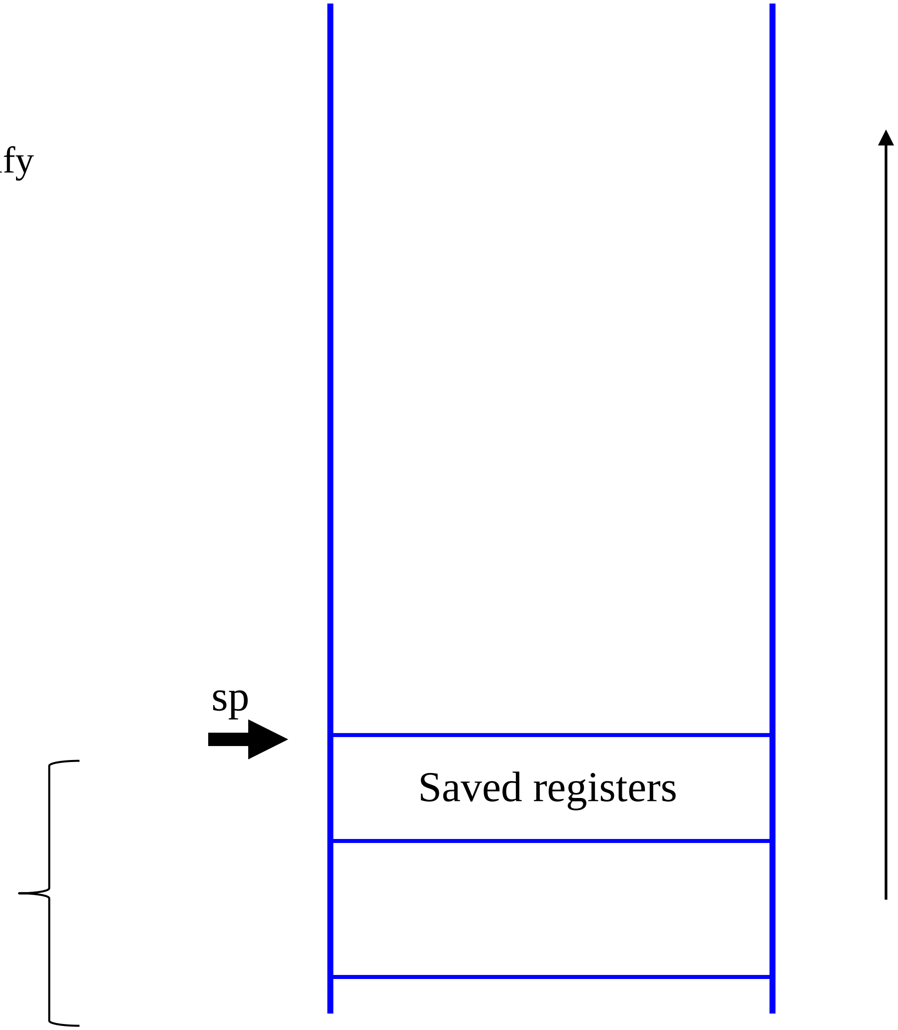


Construction of the stack frame
caller function

Saving registers

A function can modify any register $\mathbf{a}_x$ y $\mathbf{t}_x$

Stack frame of the function that makes the call



Example:

```
li    t0, 4
li    t1, 8
li    a0, 5
jal   ra, funcion

mv  s2,  t0
```

What is the value of t0 and t1?

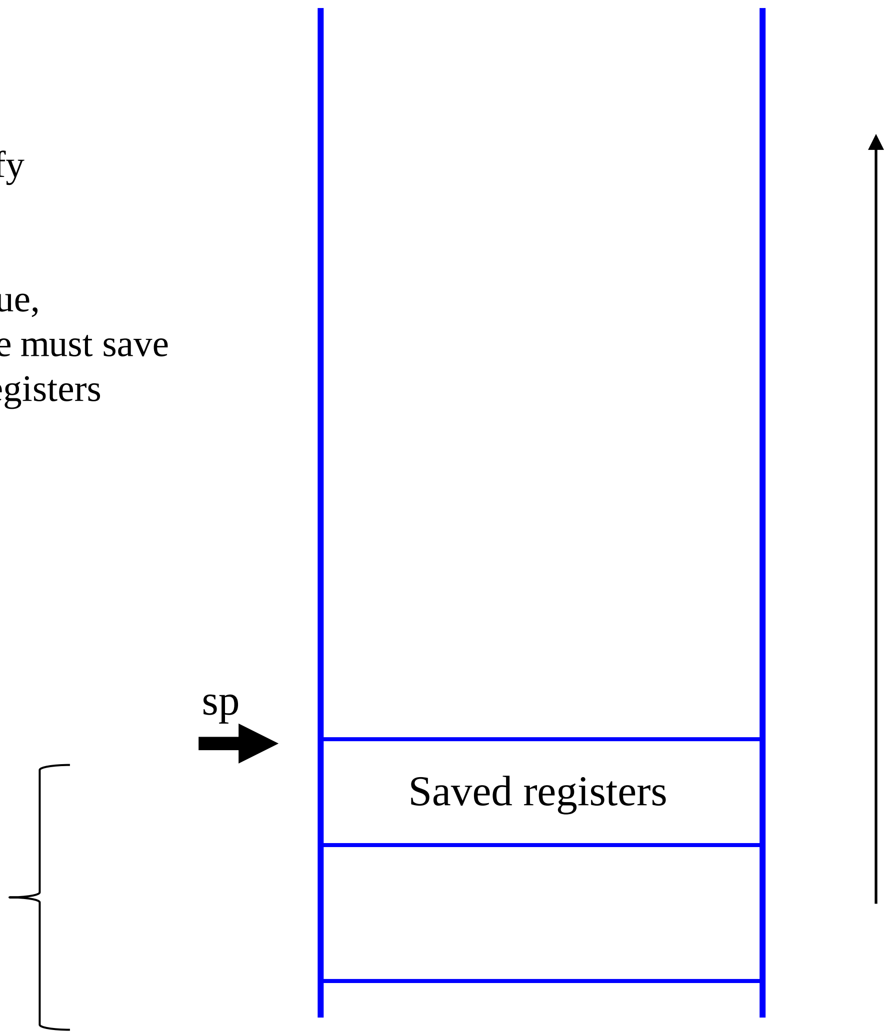
Construction of the stack frame
calle**r** function

Saving registers

A function can modify any register $\mathbf{a_x}$ y $\mathbf{t_x}$

To preserve their value, the calling subroutine must save the values of these registers on the stack

Stack frame of the function that makes the call



Example:

```
li    t0, 4
li    t1, 8
li    a0, 5
jal   ra, funcion

mv  s2,  t0
```

What is the value of t0 and t1?

Félix García Carballeira, Alejandro Calderón Mateos

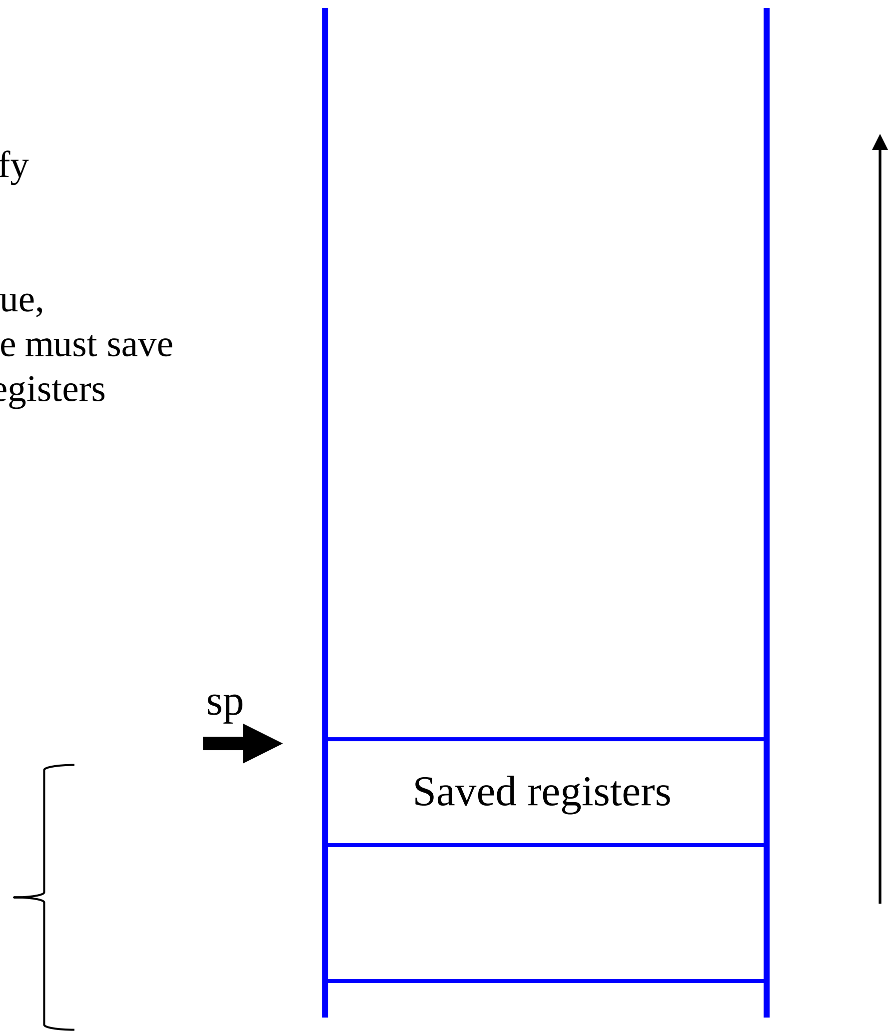
Construction of the stack frame
calle**r** function

Saving registers

A function can modify any register $\mathbf{a_x}$ y $\mathbf{t_x}$

To preserve their value, the calling subroutine must save the values of these registers on the stack

Stack frame of the function that makes the call



Example:

```
sub   sp sp 8
sw    t0  0(sp)
sw    t1  4(sp)

li     a0,  5
jal    ra, funcion
```

Construction of the stack frame
caller function

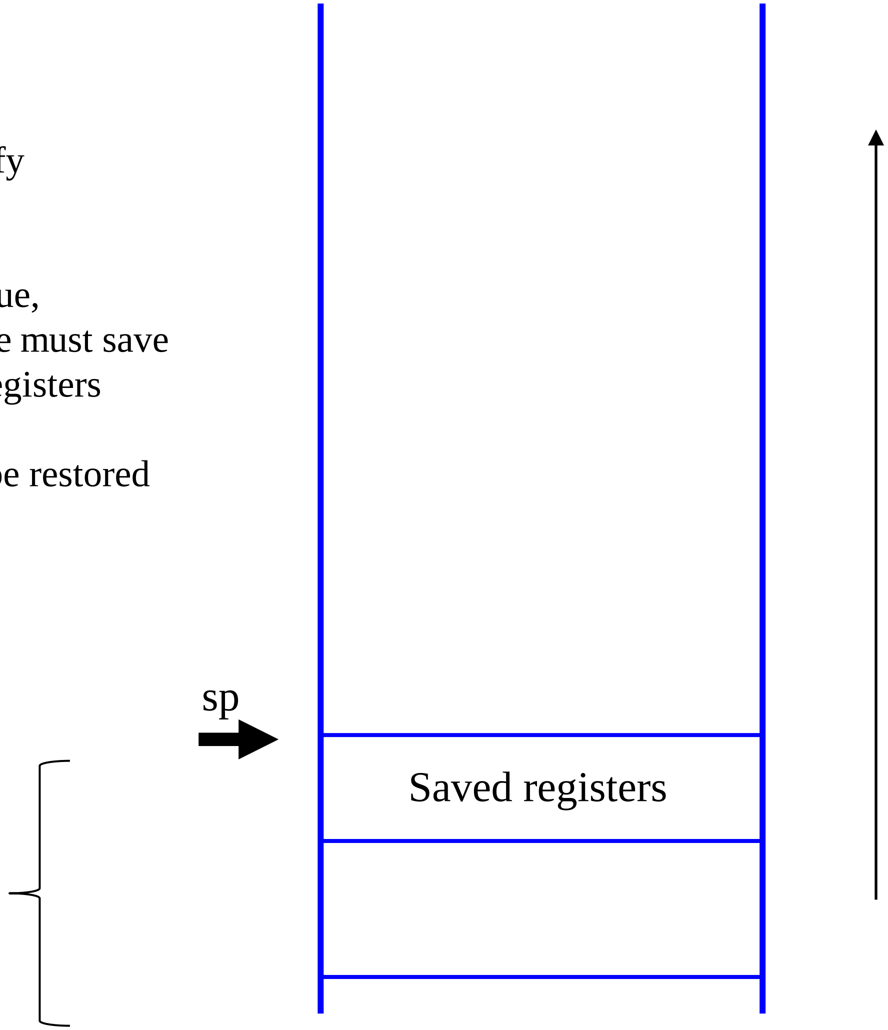
Saving registers

A function can modify any register **a_x** y **t_x**

To preserve their value, the calling subroutine must save the values of these registers on the stack

- they will have to be restored later.

Stack frame of the function that makes the call



Example:

```
sub   sp sp 8
sw    t0  0(sp)
sw    t1  4(sp)

li    a0,  5
jal   ra, funcion

lw    t0  0(sp)
lw    t1  4(sp)
add   sp sp 8
```

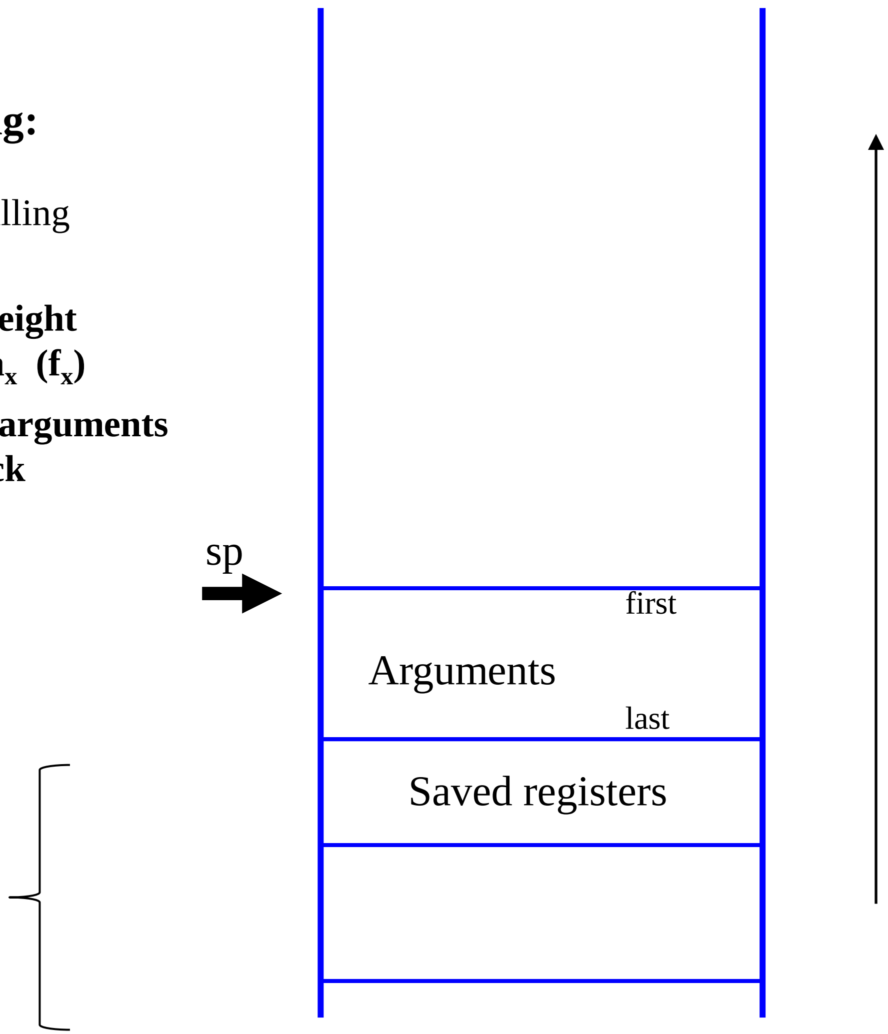
Construction of the stack frame
caller function

Example (10 arguments):

Argument passing:

Before calling the calling procedure:

- Leave the **first eight arguments in a_x (f_x)**
- The **rest** of the **arguments** goes to **the stack**



Stack frame of the function that makes the call

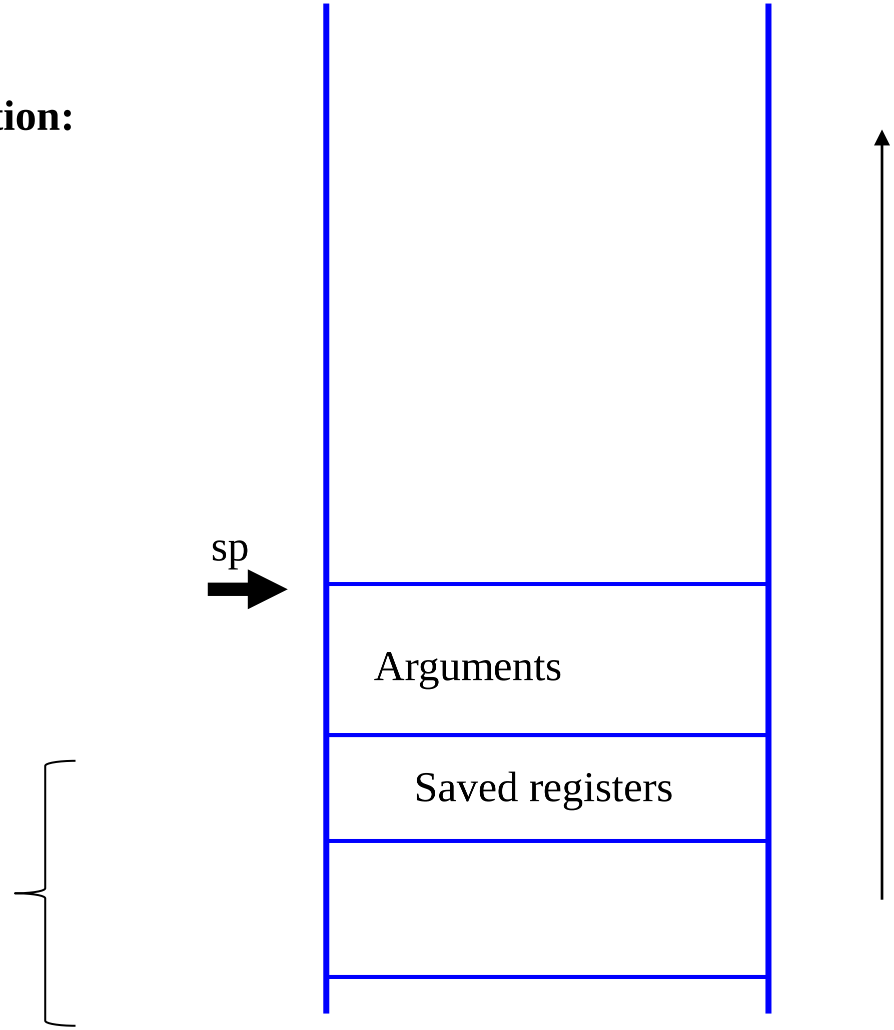
```
li    a0, 1
li    a1, 2
li    a2, 3
li    a3, 4
li    a4, 5
li    a5, 6
li    a6, 7
li    a7, 8

addi sp, sp, -8

li    t0, 10
sw    t0, 4(sp)

li    t0, 9
sw    t0, 0(sp)
```


Félix García Carballeira, Alejandro Calderón Mateos

Construction of the stack frame
caller function

Function invocation:

jal ra subrutina



Stack frame
of the function
that makes the call

```
li      a0, 1
li      a1, 2
li      a2, 3
li      a3, 4
li      a4, 5
li      a5, 6
li      a6, 7
li      a7, 8

addi sp, sp, -8

li     t0,  10
sw     t0,  4(sp)

li     t0,   9
sw     t0,   0(sp)

jal ra subrutina
```

Construction of the stack frame called function

Stack frame allocation:

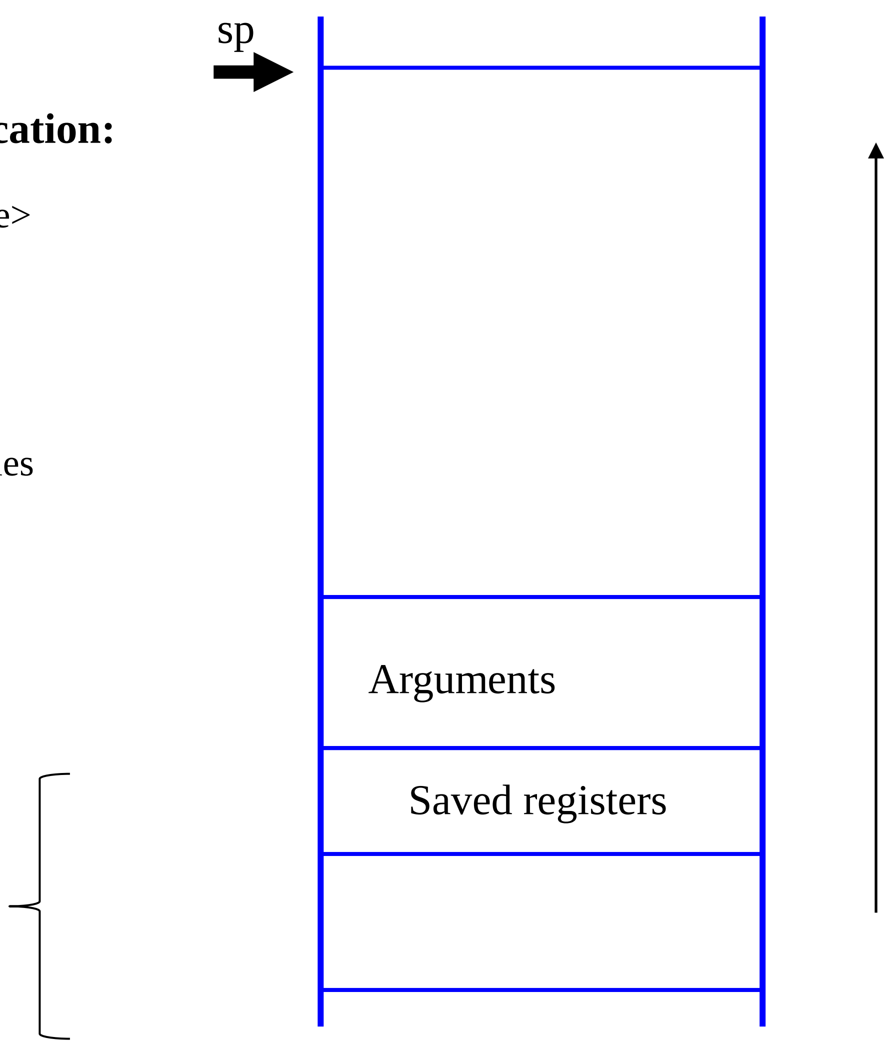
sp = sp – <frame size>

Espacio para:

- ra
- s0…s7
- Local variables

Example:

```
function:
   subu sp sp <fr.sz.>
```



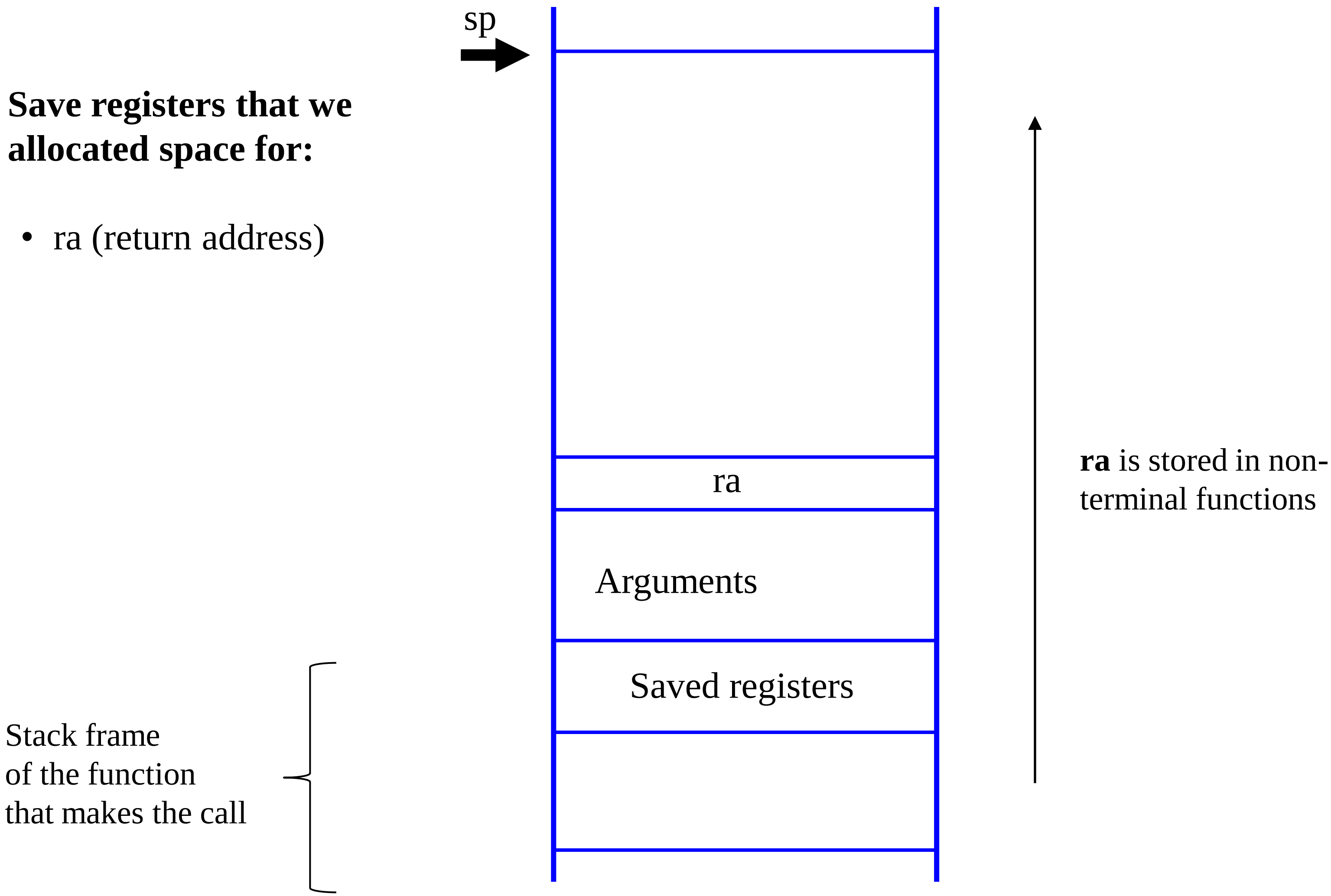
Stack frame
of the function
that makes the call

Félix García Carballeira, Alejandro Calderón Mateos

Construction of the stack frame
calle**d** function

Save registers that we allocated space for:

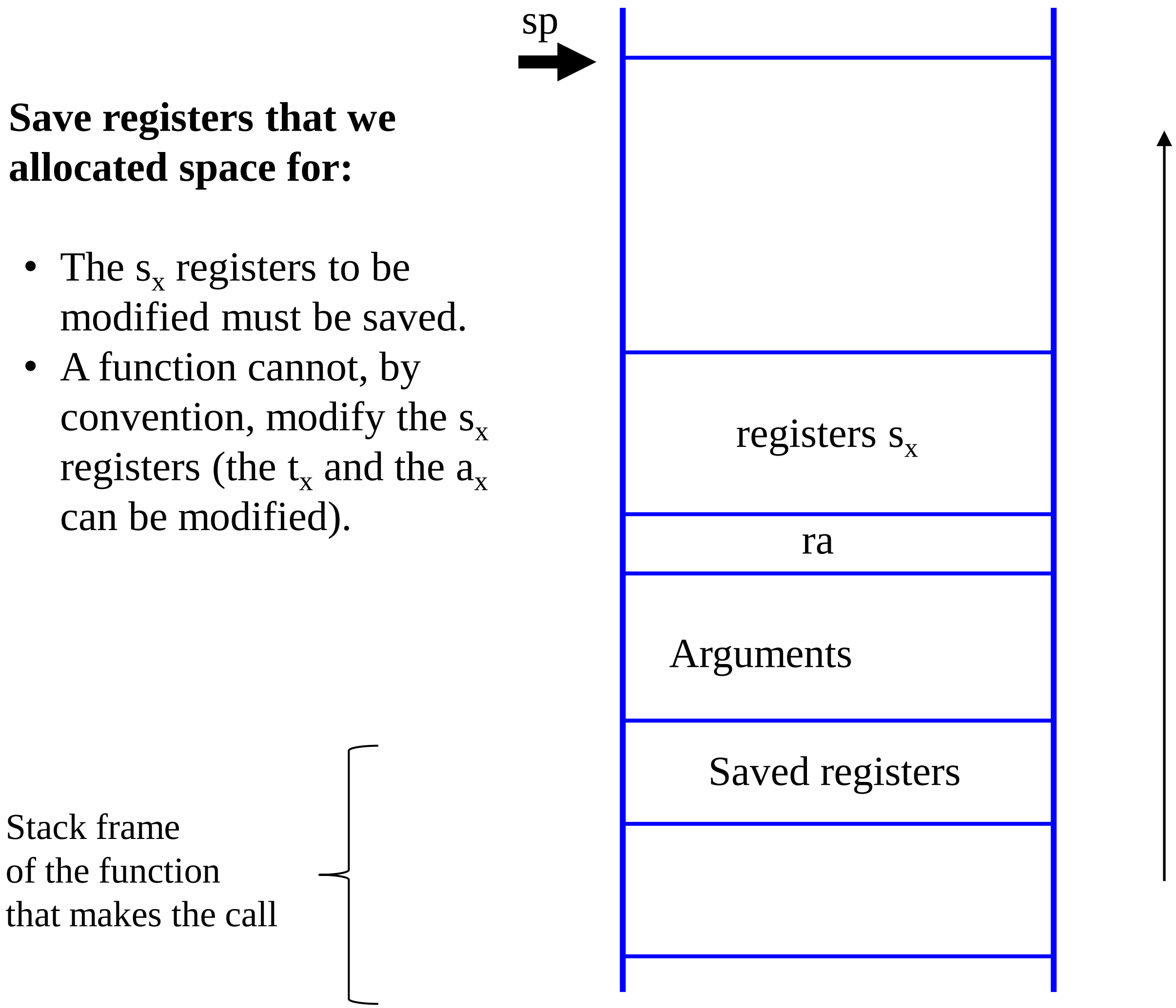
- ra (return address)



Construction of the stack frame called function

Save registers that we allocated space for:

- The s_x registers to be modified must be saved.
- A function cannot, by convention, modify the s_x registers (the t_x and the a_x can be modified).

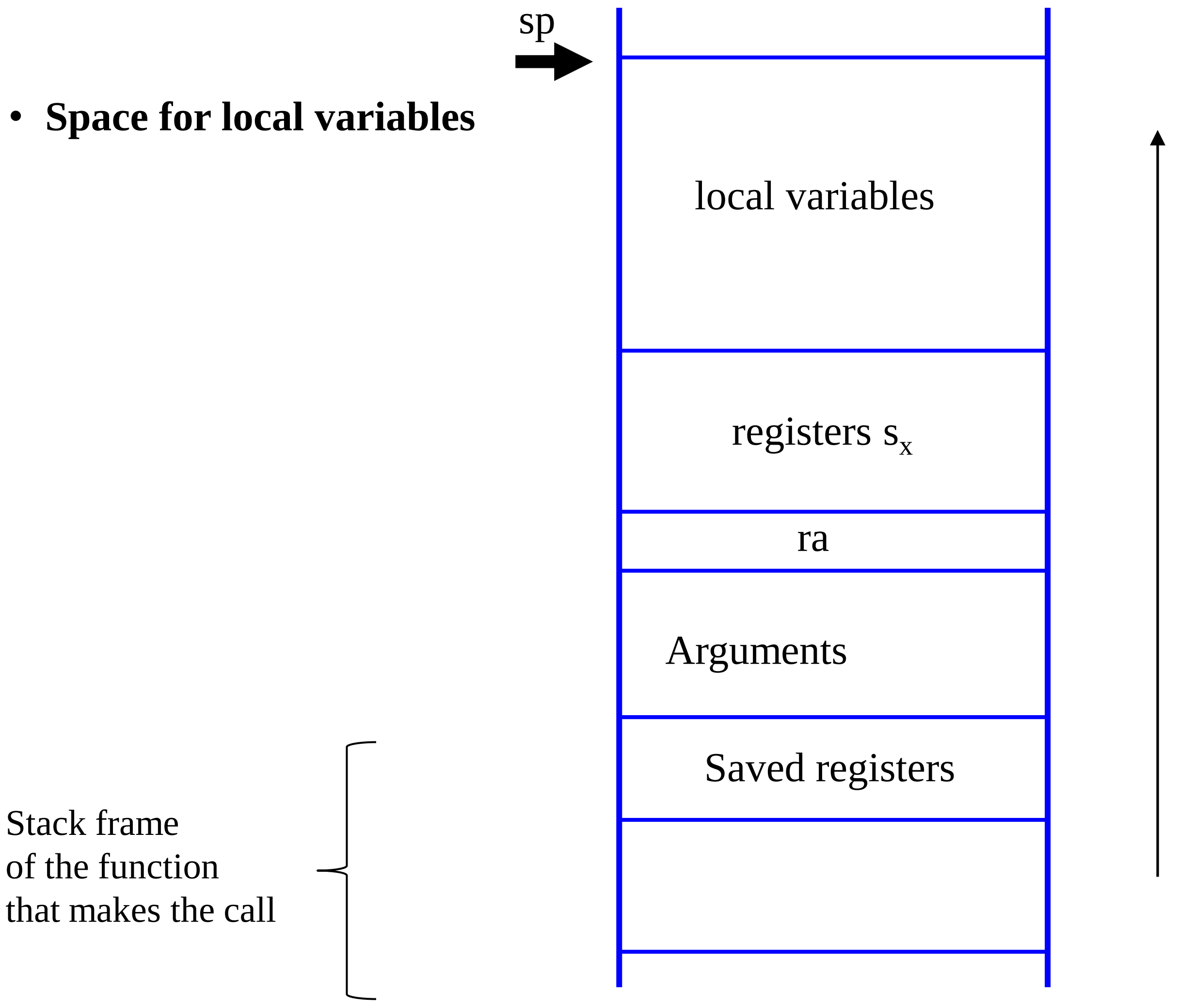


Example:

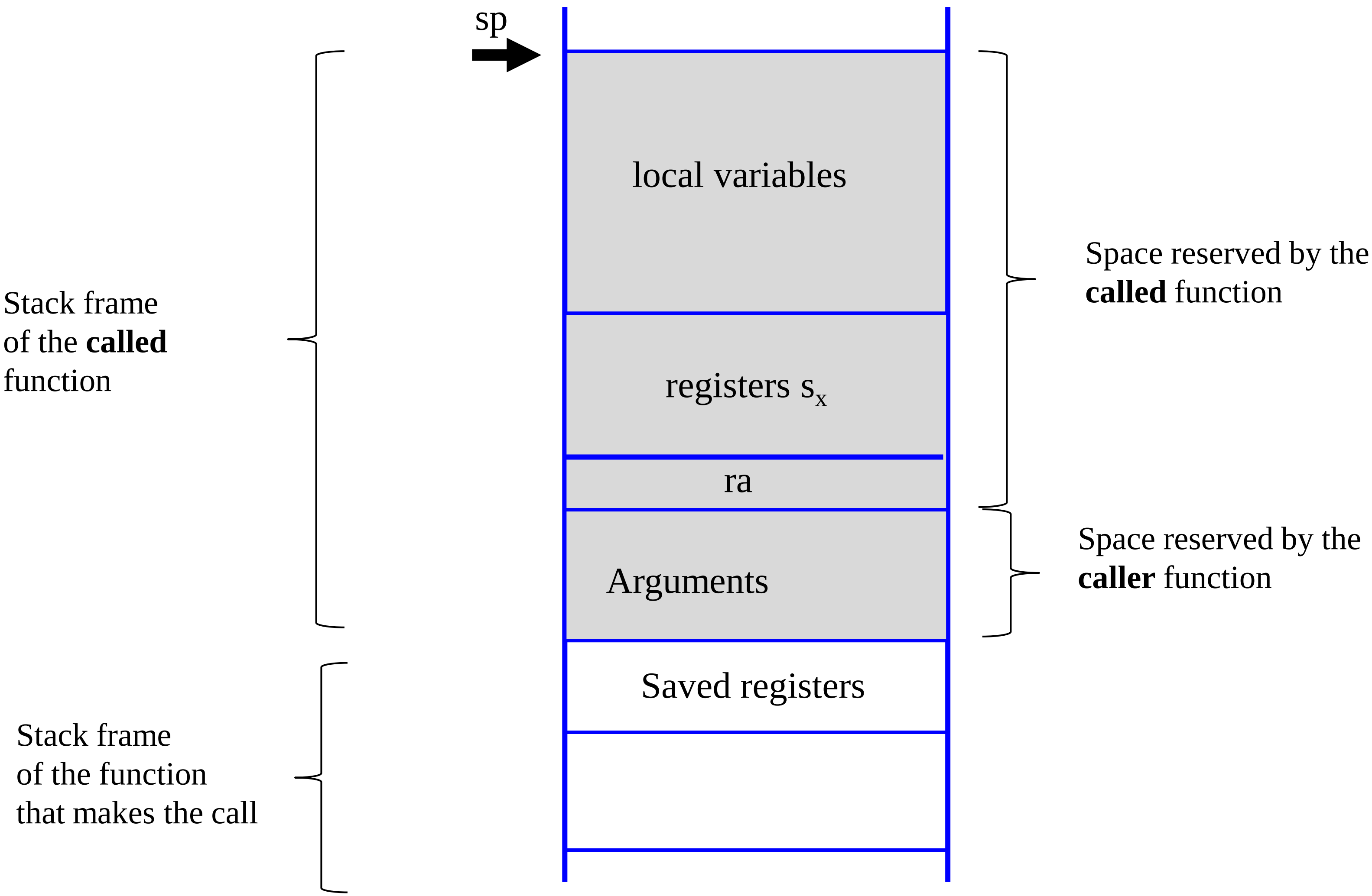
```
function:
   subu sp sp <fr.sz.>
   sw ra <fr.sz-4>(sp)
   sw s0 <fr.sz-8>(sp)
   sw s1       <...>(sp)
   …
```

Construction of the stack frame
calle**d** function

- **Space for local variables**



Stack frame construction

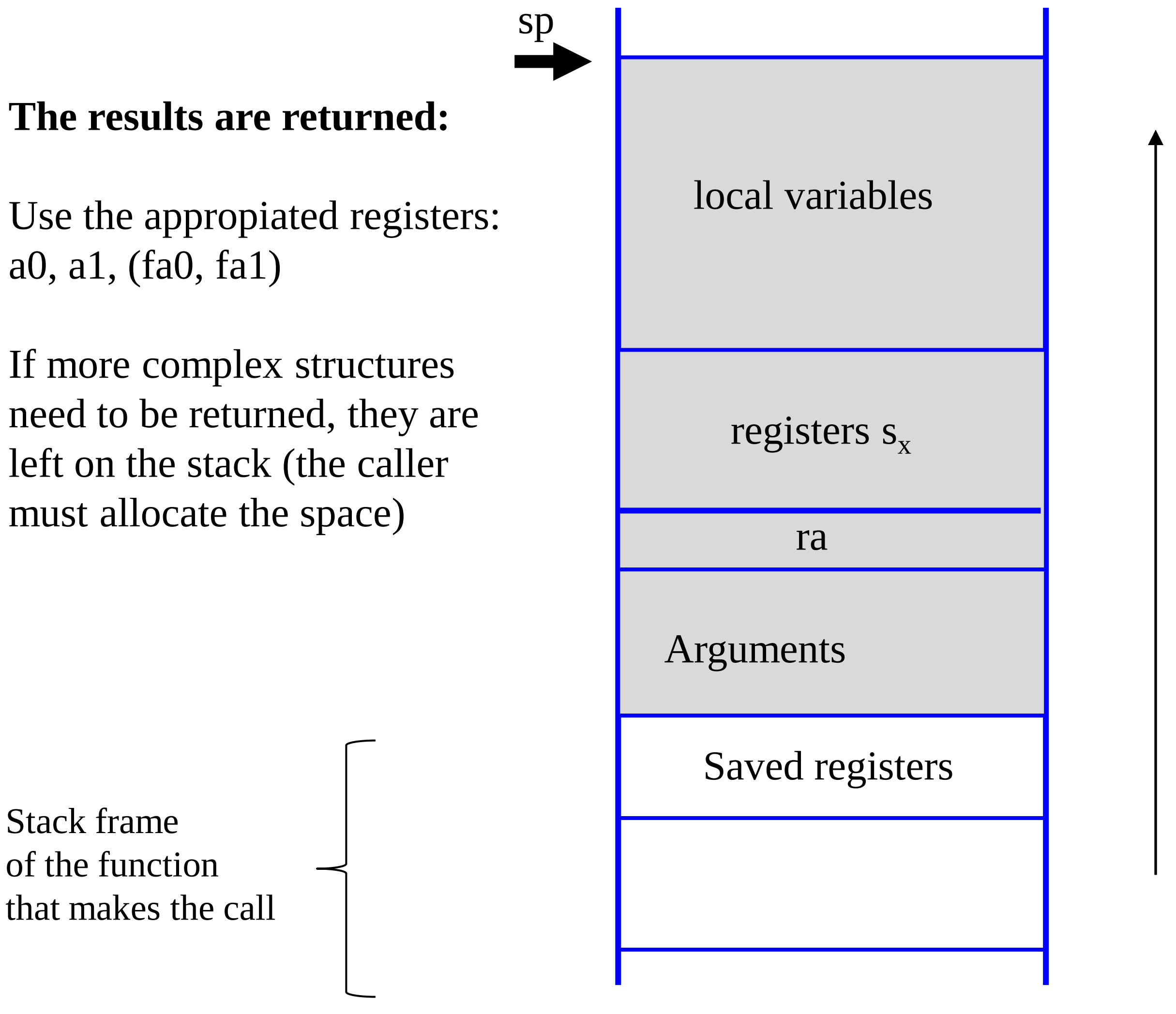


Subroutine termination
calle**d** function

The results are returned:

Use the appropiated registers: a0, a1, (fa0, fa1)

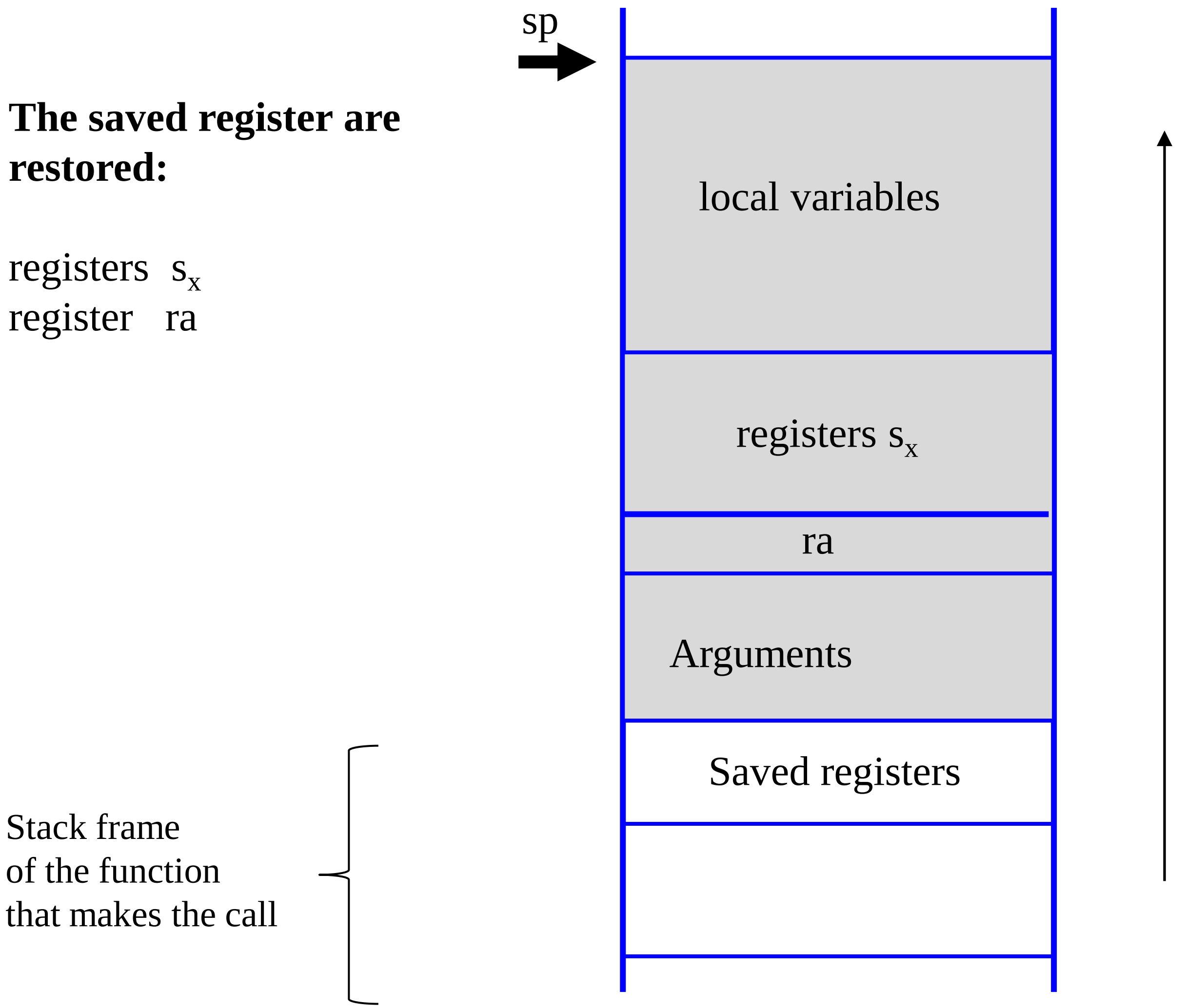
If more complex structures need to be returned, they are left on the stack (the caller must allocate the space)



Subroutine termination
calle**d** function

The saved register are restored:

registers s_x
register ra

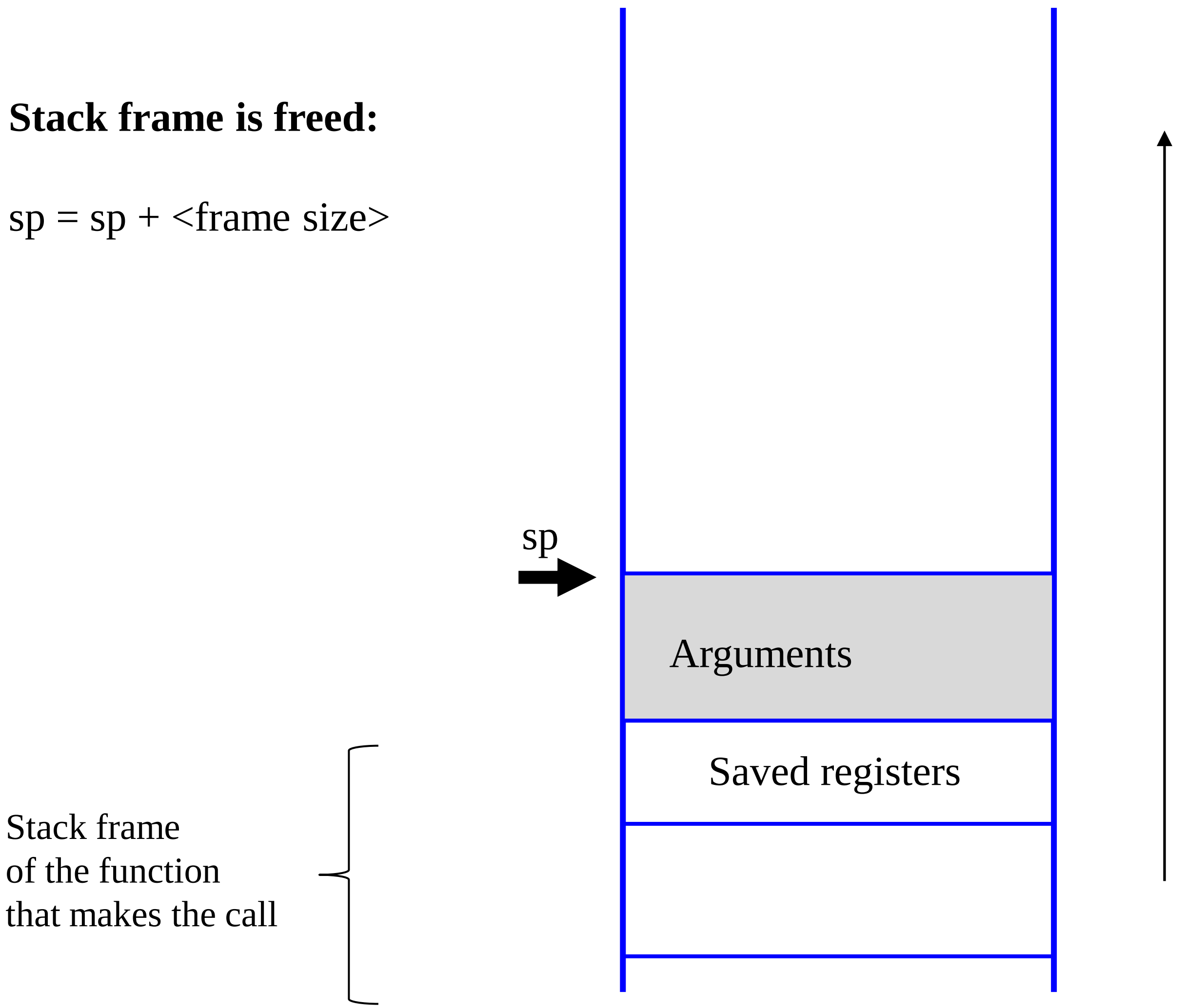


Félix García Carballeira, Alejandro Calderón Mateos

Subroutine termination
calle**d** function

Stack frame is freed:

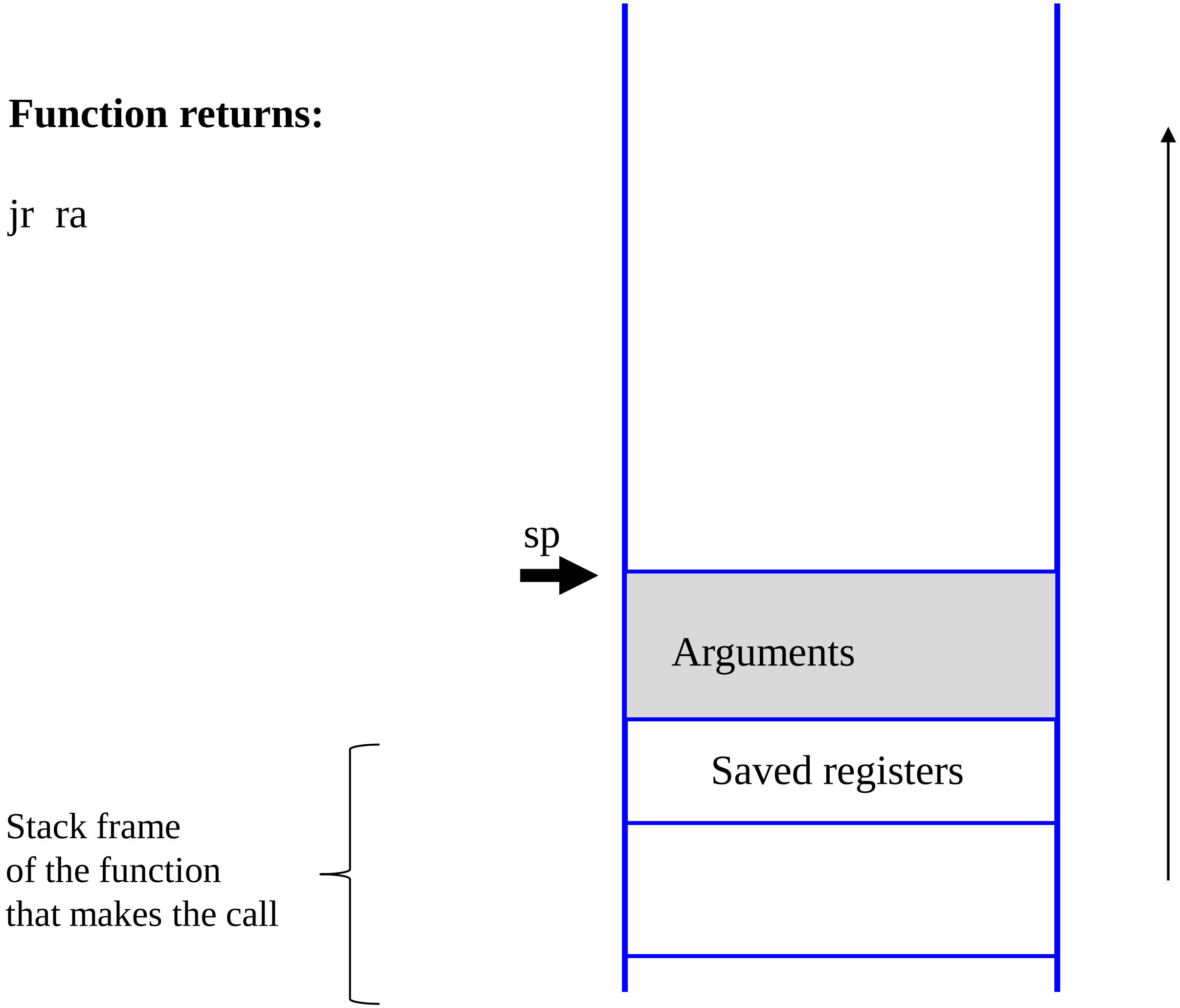
sp = sp + <frame size>



Subroutine termination
calle**d** function

Function returns:

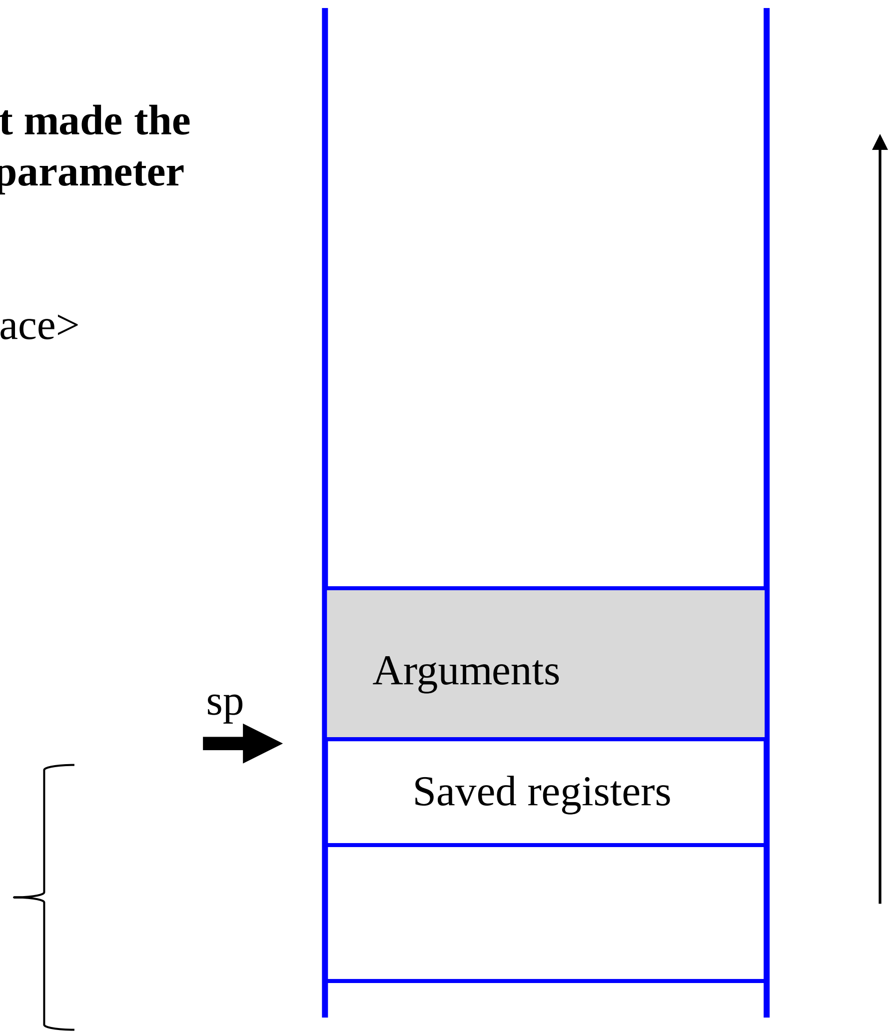
jr ra



Subroutine termination
calle function

The function that made the call frees up the parameter space:

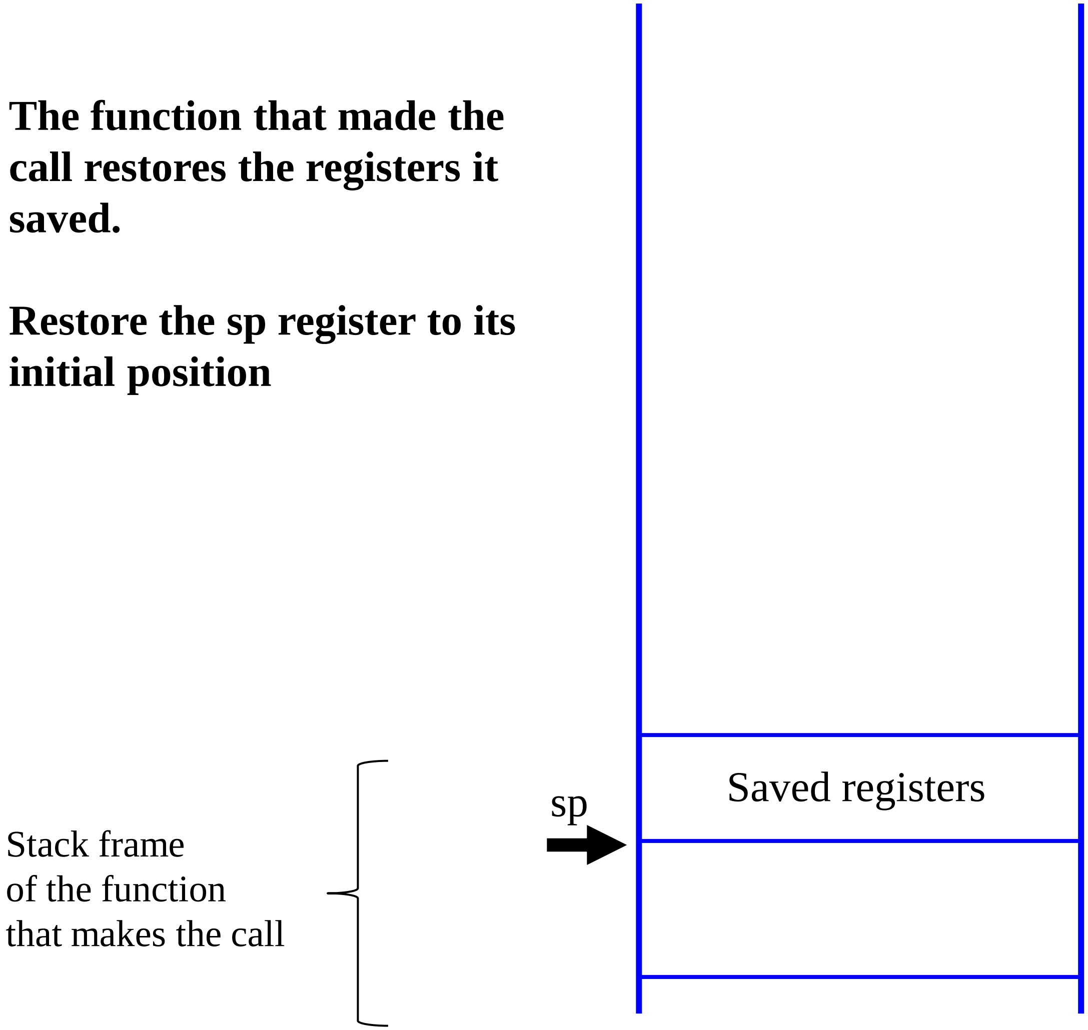
sp = sp + <arg. space>



Subroutine termination
calle function

The function that made the call restores the registers it saved.

Restore the sp register to its initial position

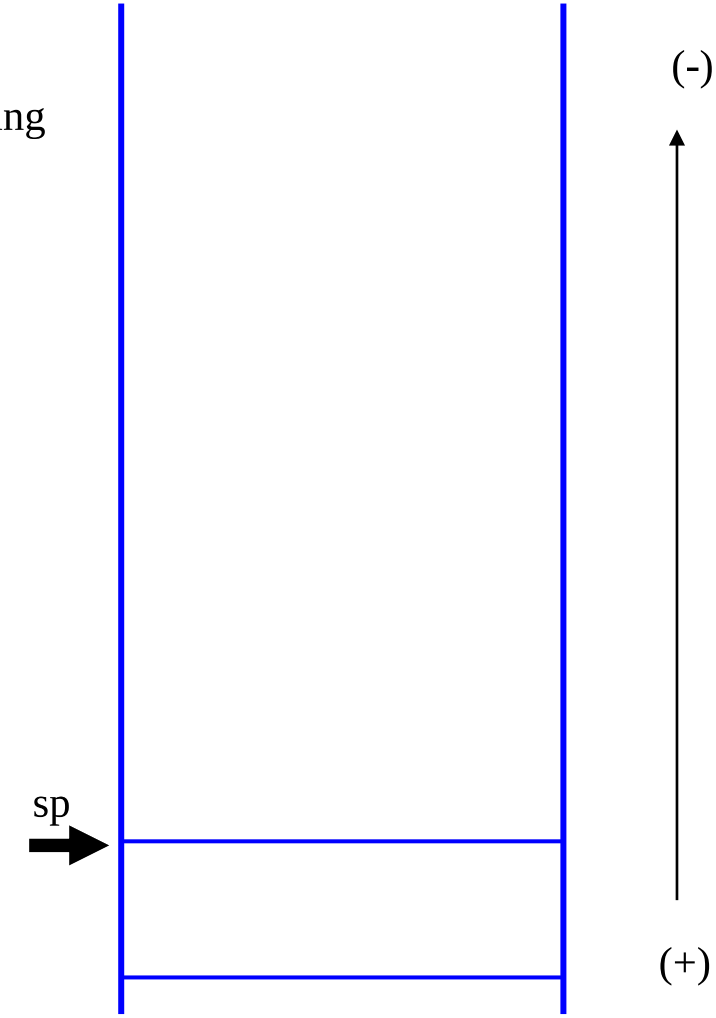


Example:

```
addi  sp sp -8
sw    t0  0(sp)
sw    t1  4(sp)

li    a0, 5
jal   ra, funcion

lw    t0  0(sp)
lw    t1  4(sp)
add   sp sp 8
```

State after subroutine termination

- **Initial** situation before calling a function



Local variables in registers

- **Whenever possible, local variables (int, double, char, …) are stored in registers**
  - If registers cannot be used (there are not enough), the stack is used

```
int f(….)
{
  int i, j, k;

  i = 0;
  j = 1;
  k= i + j;
 . . .
}
```

```
f:   . . .
     li   t0, 0
     li   t1, 1
     add t2, t0, t1
     . . .
```

ARCOS Group

uc3m | Universidad Carlos III de Madrid

L3: Fundamentals of assembler programming (4)
Computer Structure

Bachelor in Computer Science and Engineering
Bachelor in Applied Mathematics and Computing
Dual Bachelor in Computer Science and Engineering and Business Administration