ARCOS Group

uc3m | Universidad Carlos III de Madrid

Lesson 3 (IV)
Fundamentals of assembler programming

Computer Structure

Bachelor in Computer Science and Engineering

Contents

- Basic concepts on assembly programming
- MIPS32 assembly language, memory model and data representation
- Instruction formats and addressing modes
- Procedure calls and stack convention

Functions

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

```
factorial:
    mv   t0 a0
    li   v0 1
b1: beq  t0 zero f1
    mul  v0 v0 t0
    addi t0 t0 -1
    beq  x0 x0 b1
f1: jr ra
…
li   a0 3
jal ra factorial
…
```

- 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
- 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

Steps in the execution of a function

- Pass the input parameters (arguments) to the function
- Transfer the flow control to the function
- Acquire storage resources needed for the function
- Make the task
- Save the results
- Return to the previous point of control

Functions in high level languages

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

Functions in high level languages

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

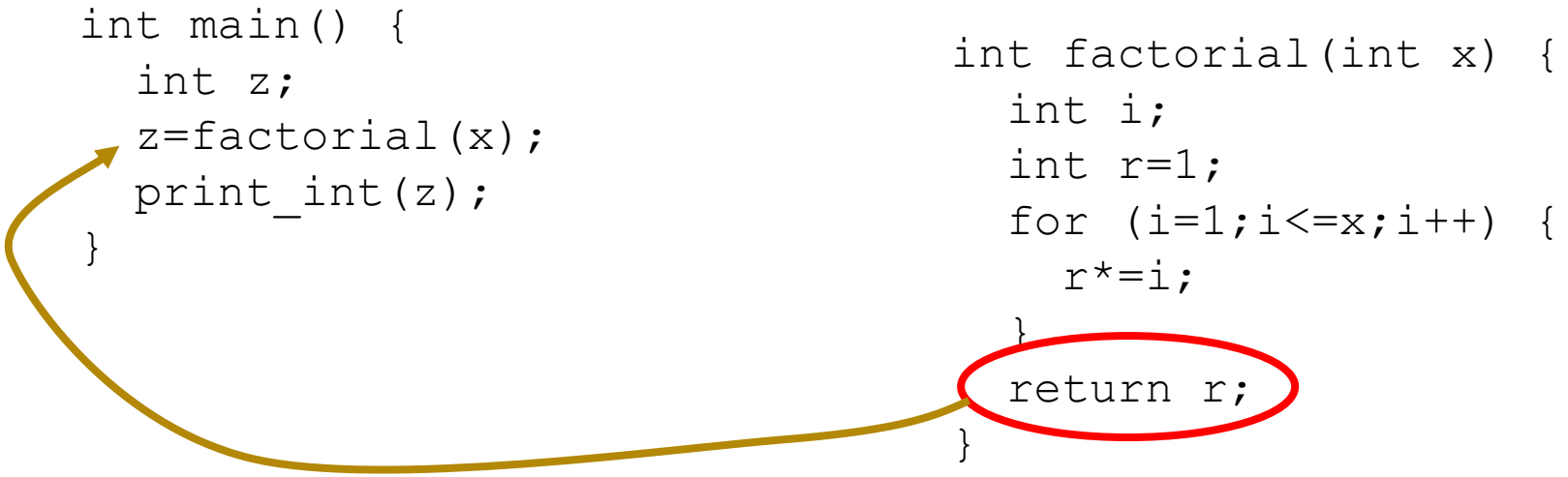
```
int factorial(int x) {
  int i;
  int r=1;
  for (i=1;i<=x;i++) {
    r*=i;
  }
  return r;
}
```

Functions in high level languages

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

Functions in high level languages

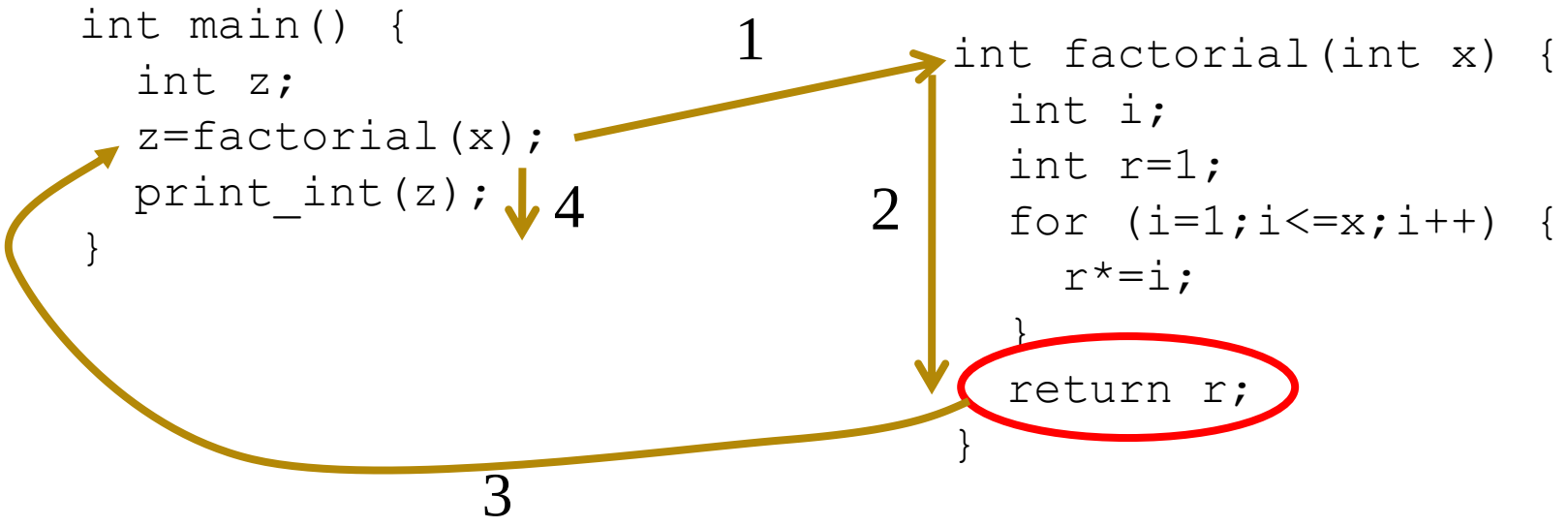


Functions in high level languages

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

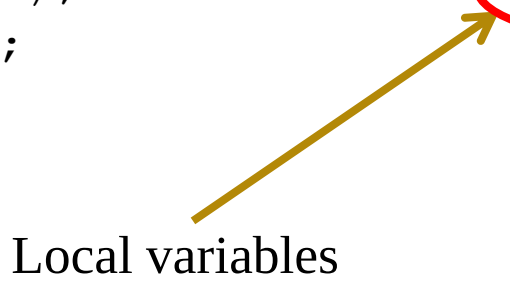
Functions in high level languages



Functions in high level languages

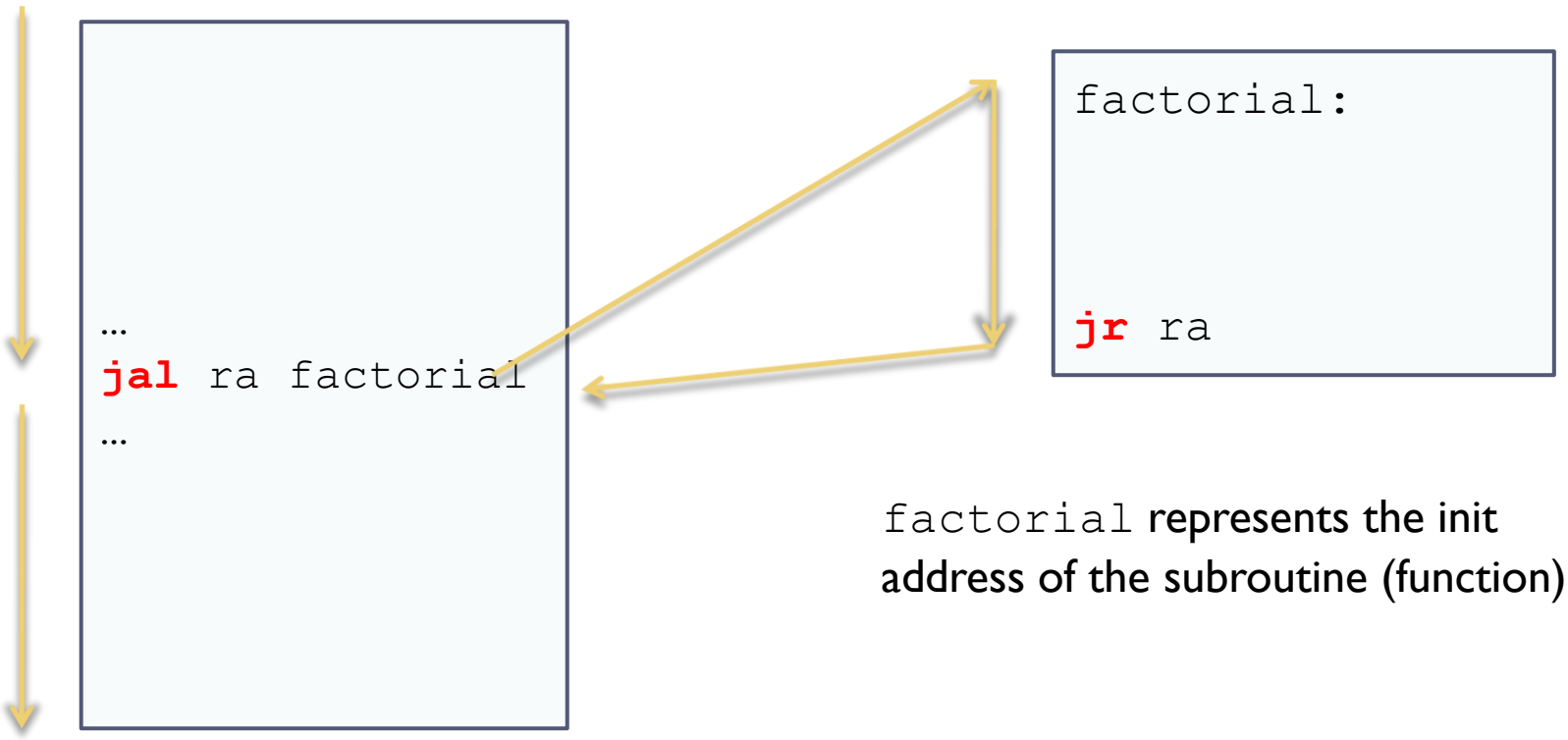
```
int main() {
  int z;
  z=factorial(x);
  print_int(z);
}
```

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



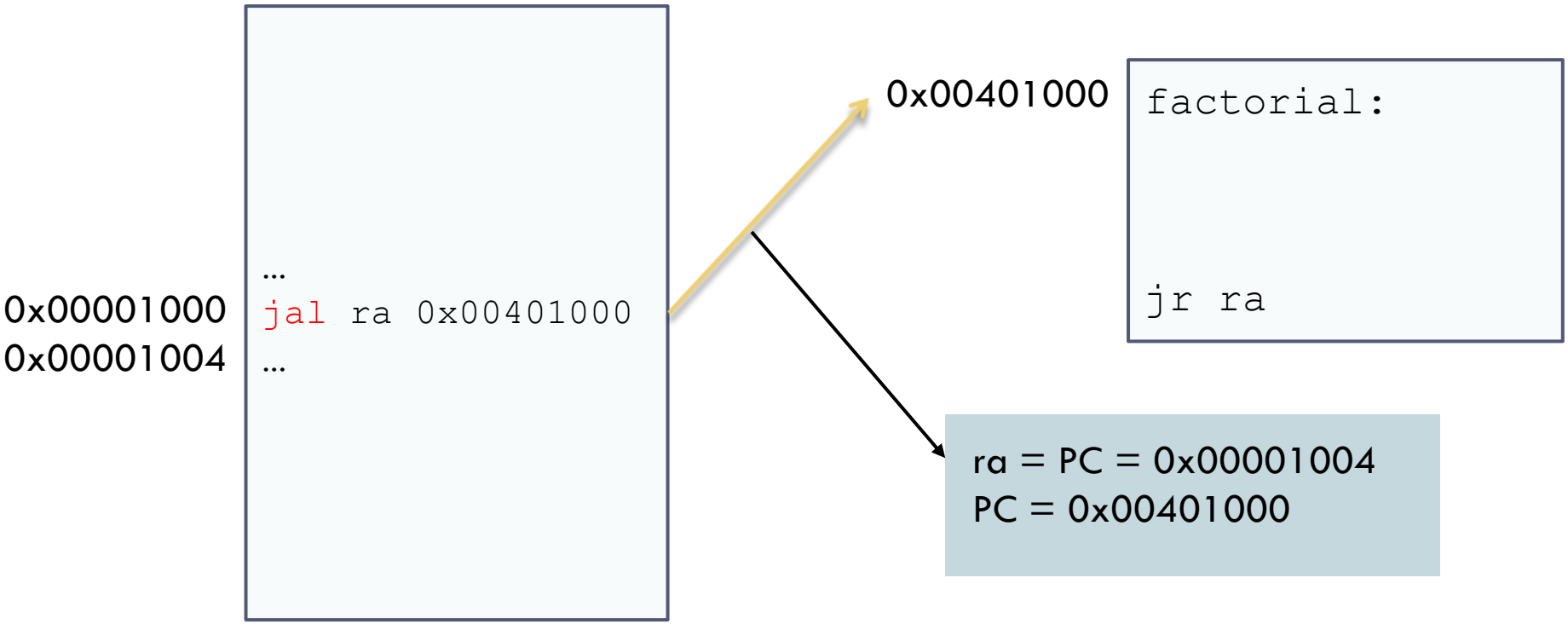
Function calls in RISC-V

Function calls in RISC-V (`jal` instruction)

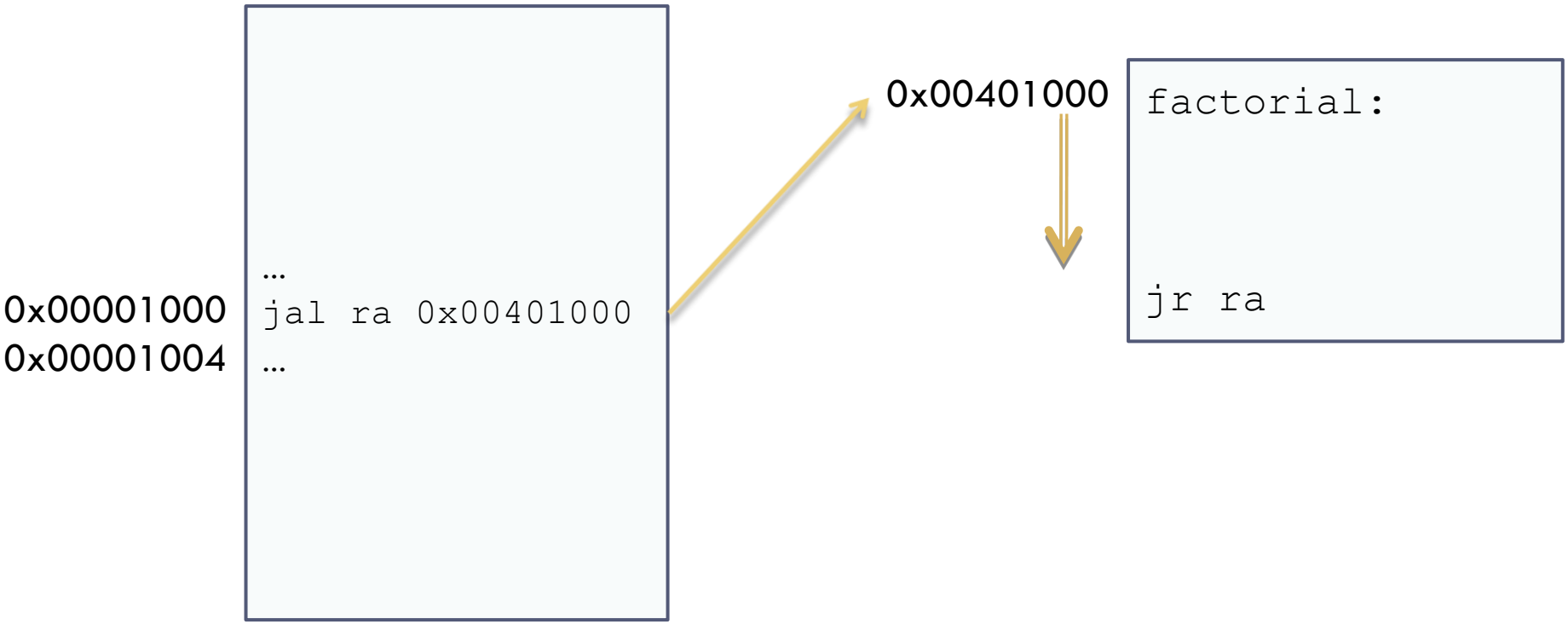


`factorial` represents the init address of the subroutine (function)

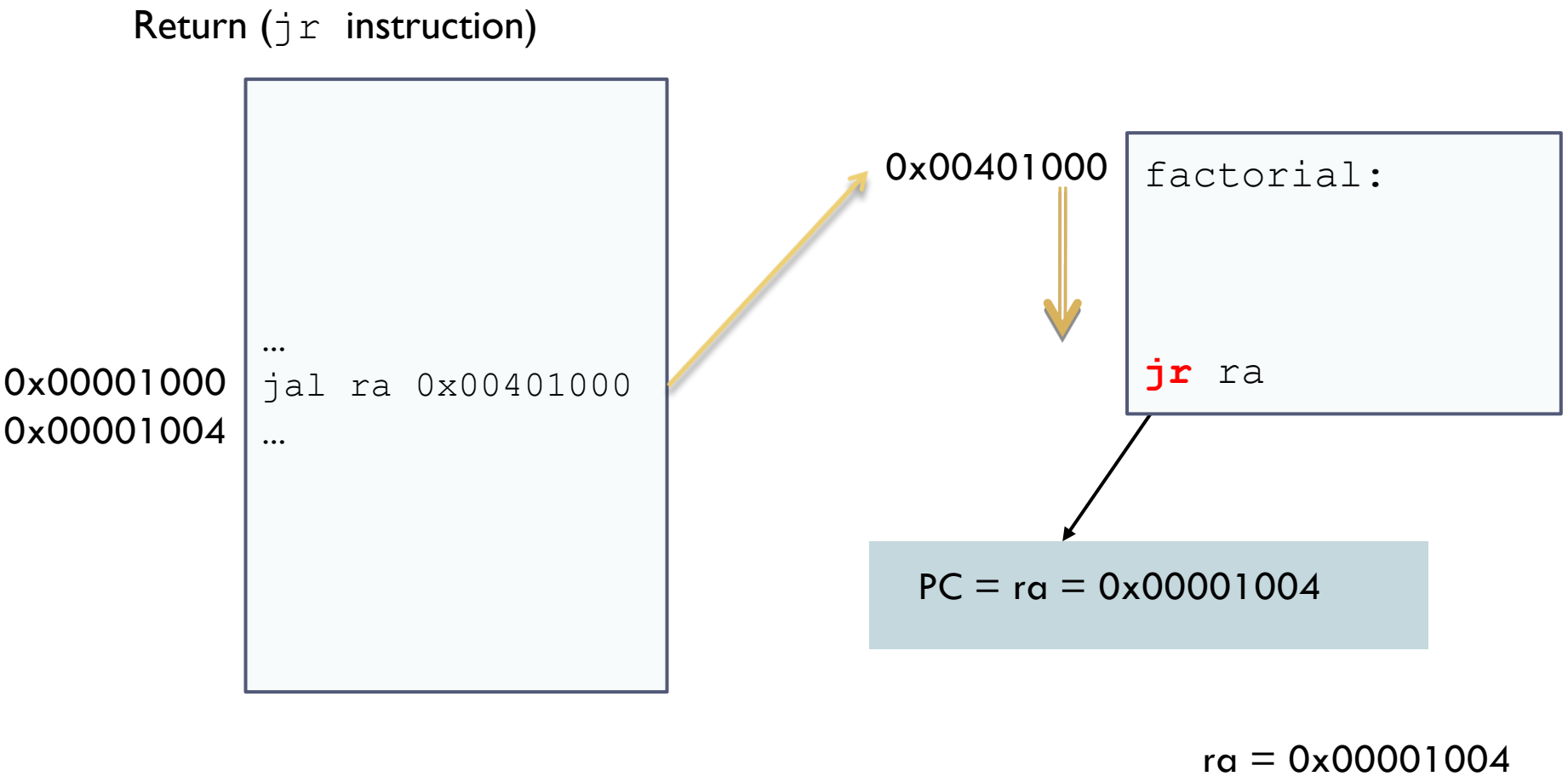
Function calls in RISC-V



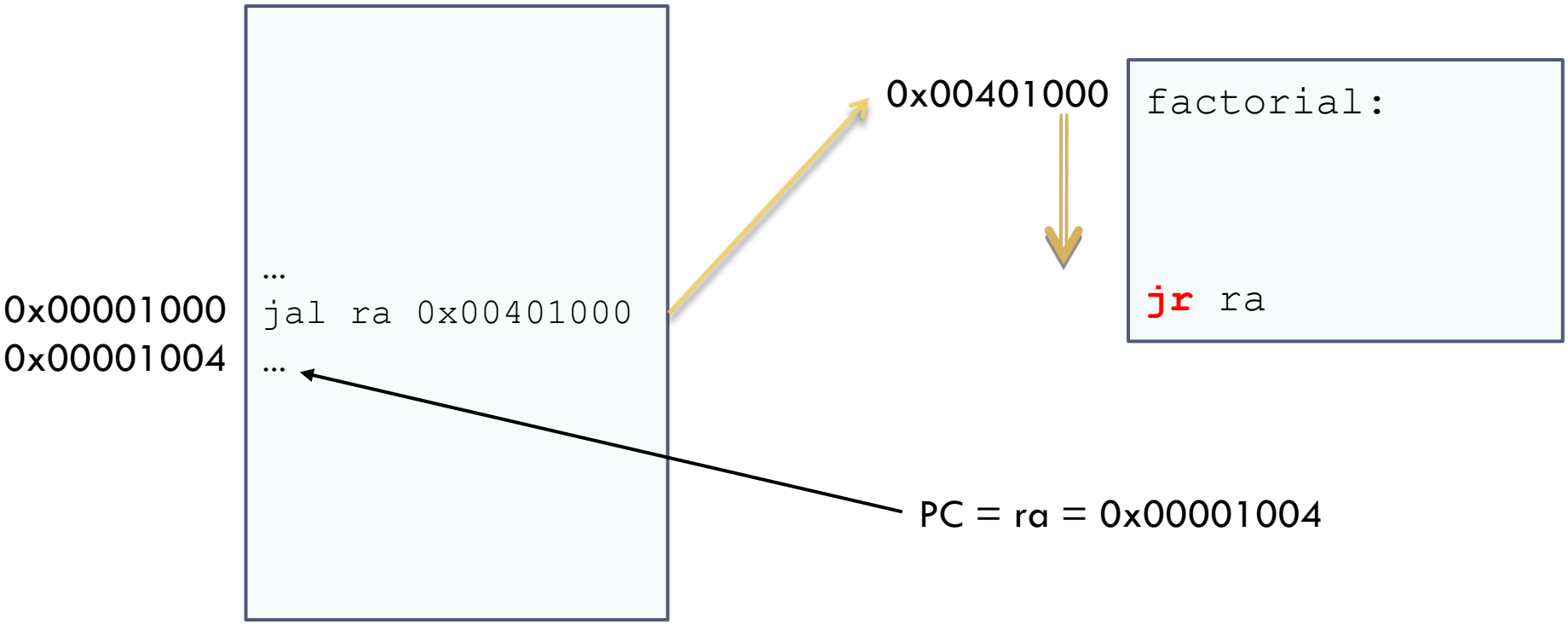
Function calls in RISC-V



Function calls in MIPS



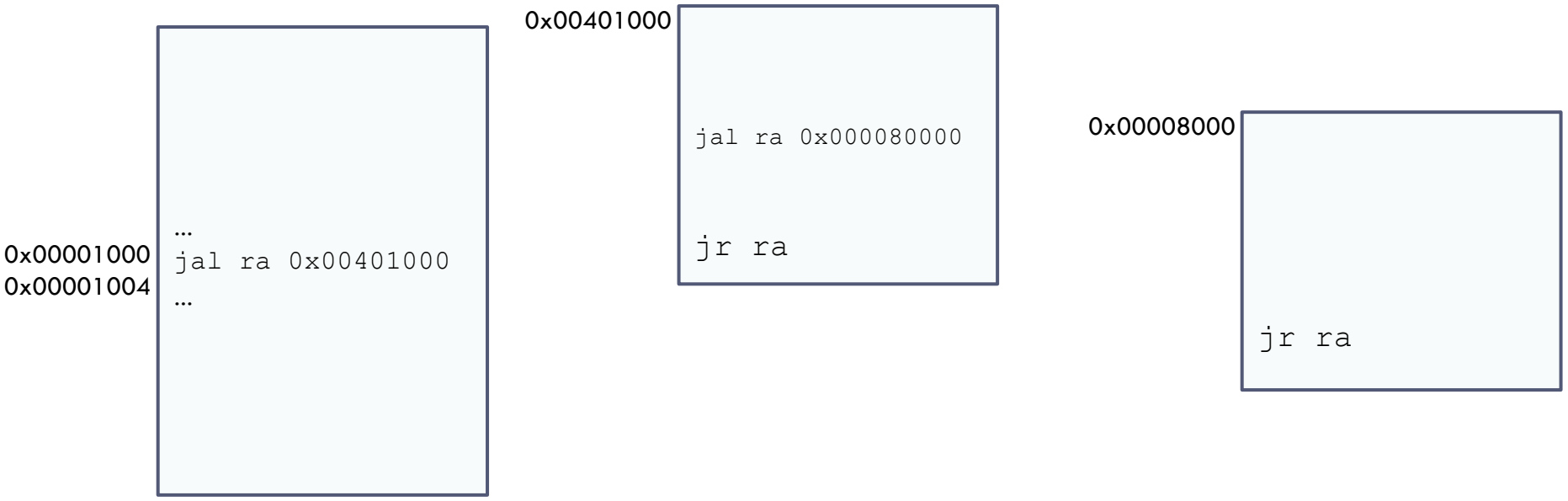
Function calls in MIPS



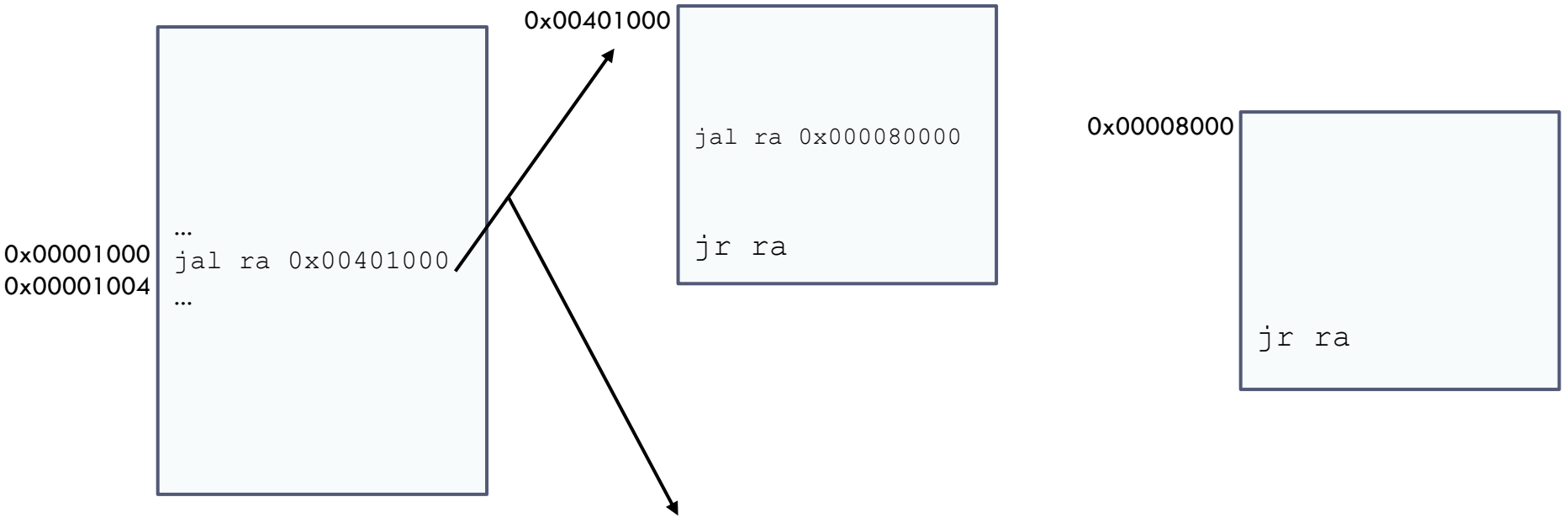
`jal/jr` instructions

- What is the behavior of `jal` instruction?
  - ra ← PC
  - PC ← initial address of the function
- What is the behavior of `jr` instruction?
  - PC ← ra

Nested calls

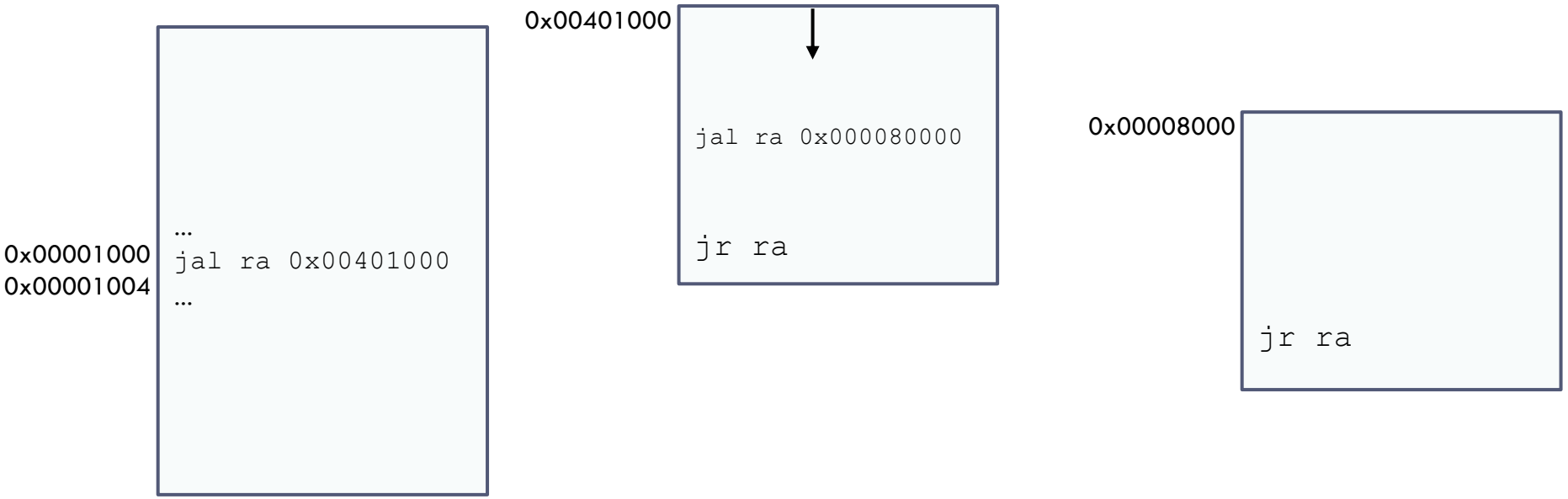


Nested calls



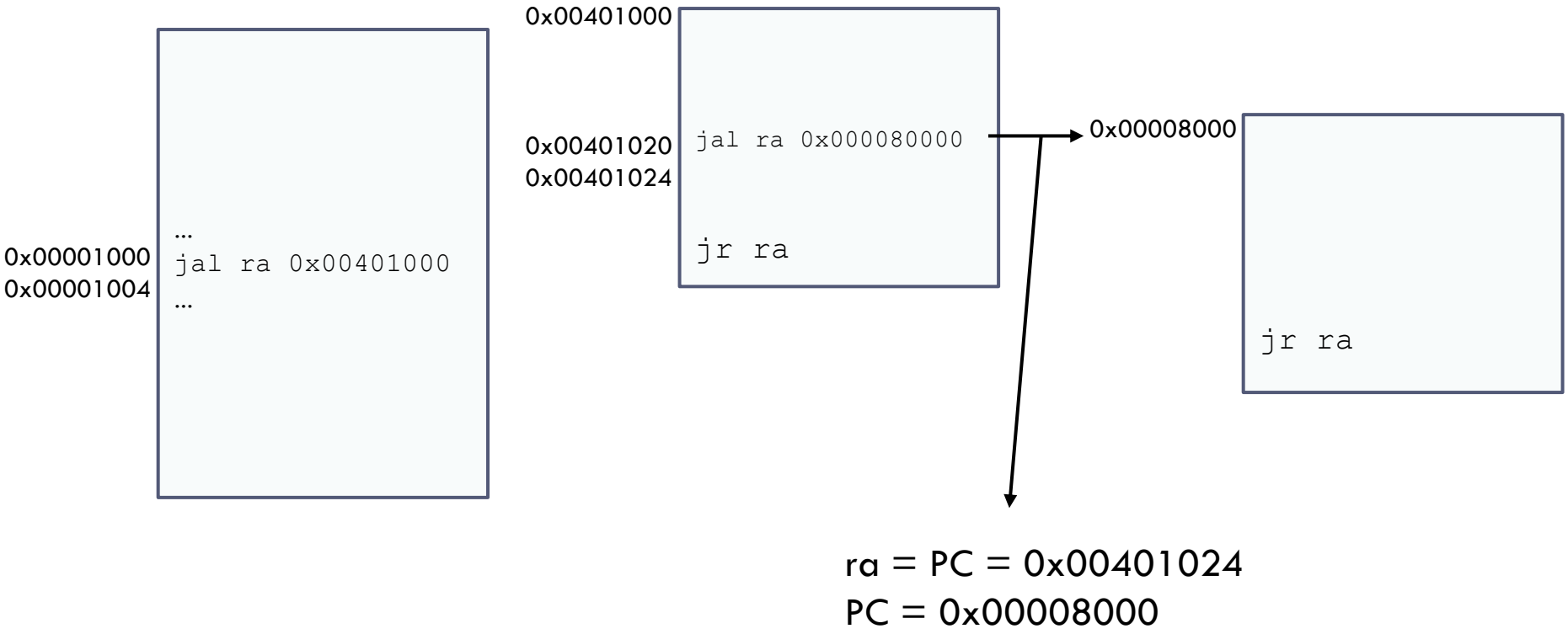
Return address ra = PC = 0x00001004

Nested calls

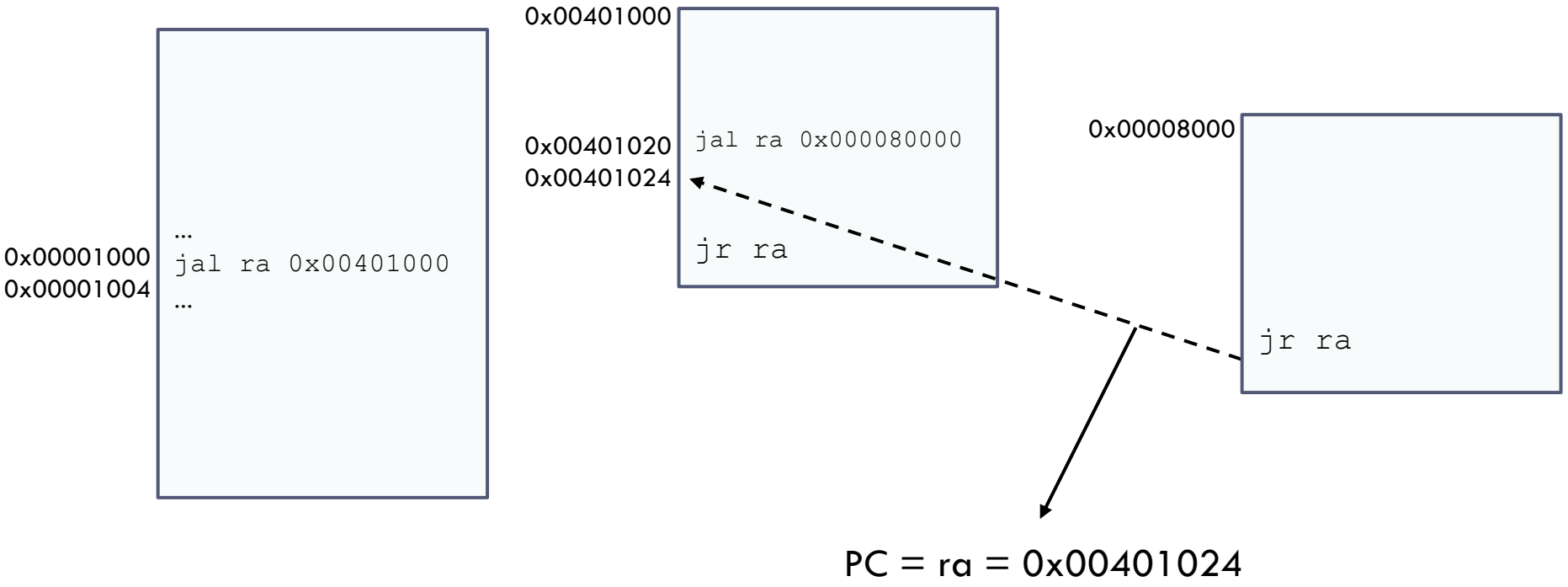


Return address ra = PC = 0x00001004

Nested calls

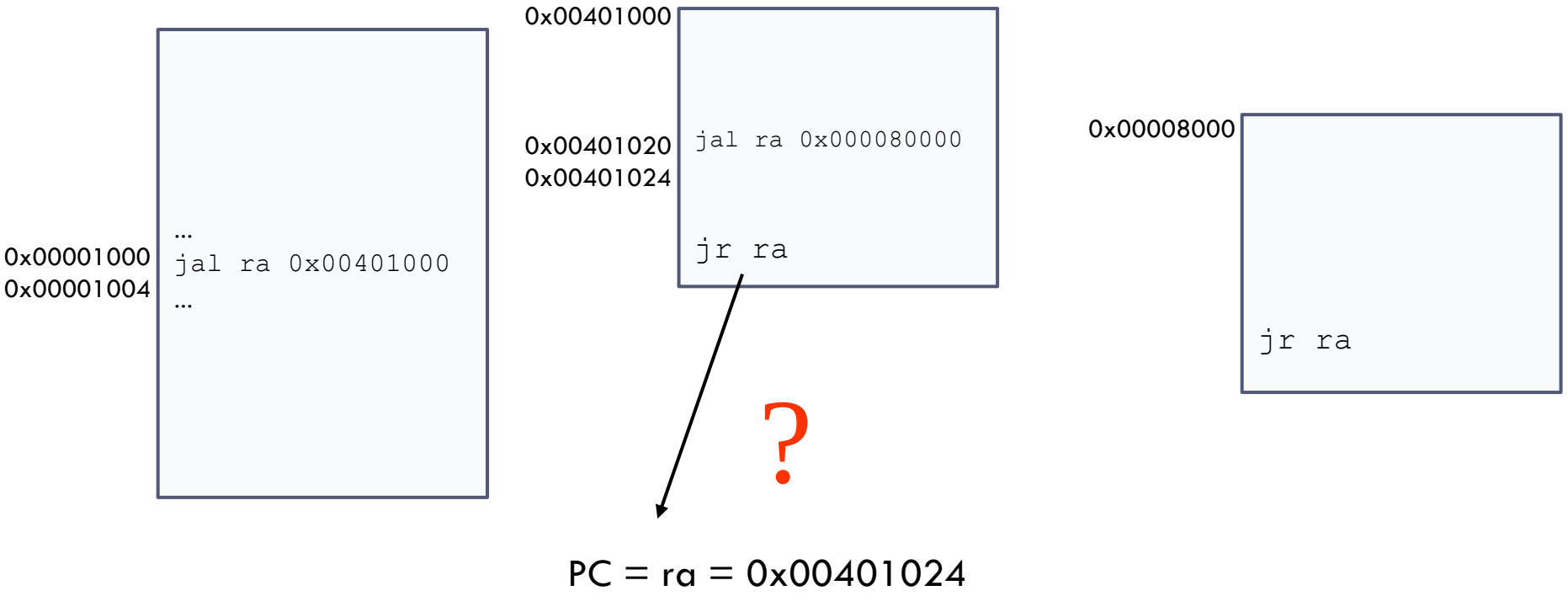


Nested calls



Return address $ra = PC = 0x00001004

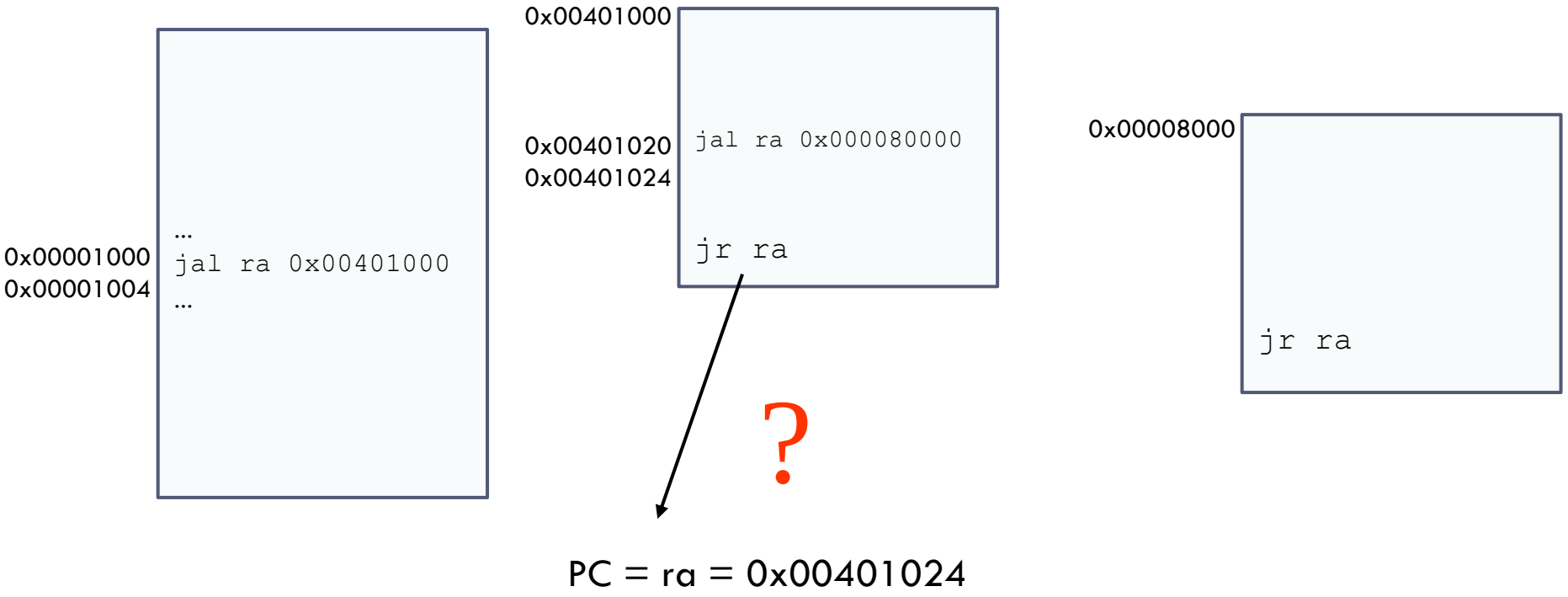
Nested calls



PC = ra = 0x00401024

Return address $ra = PC = 0x00001004

Nested calls



PC = ra = 0x00401024

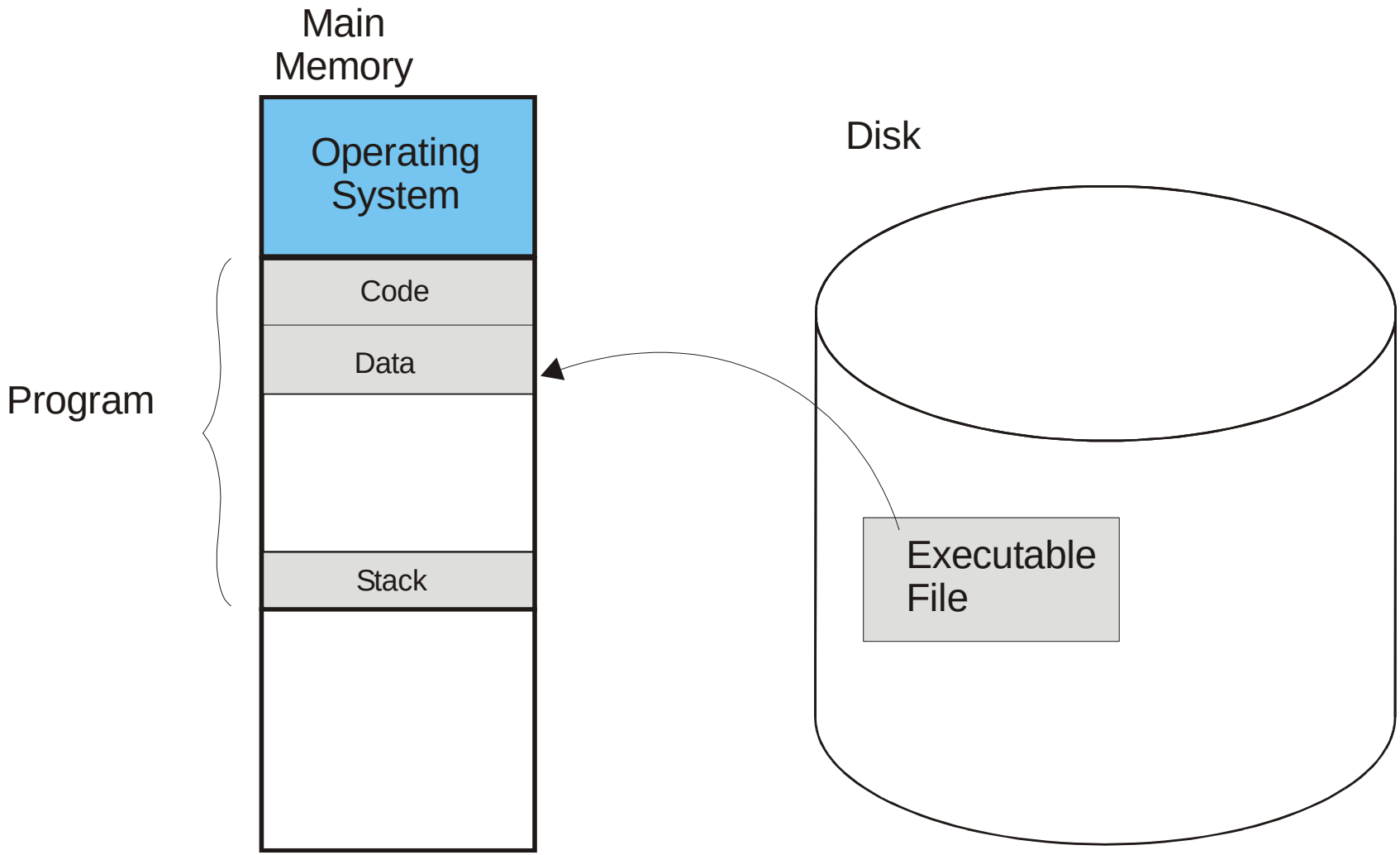
The return address is lost

Where to store the return address?

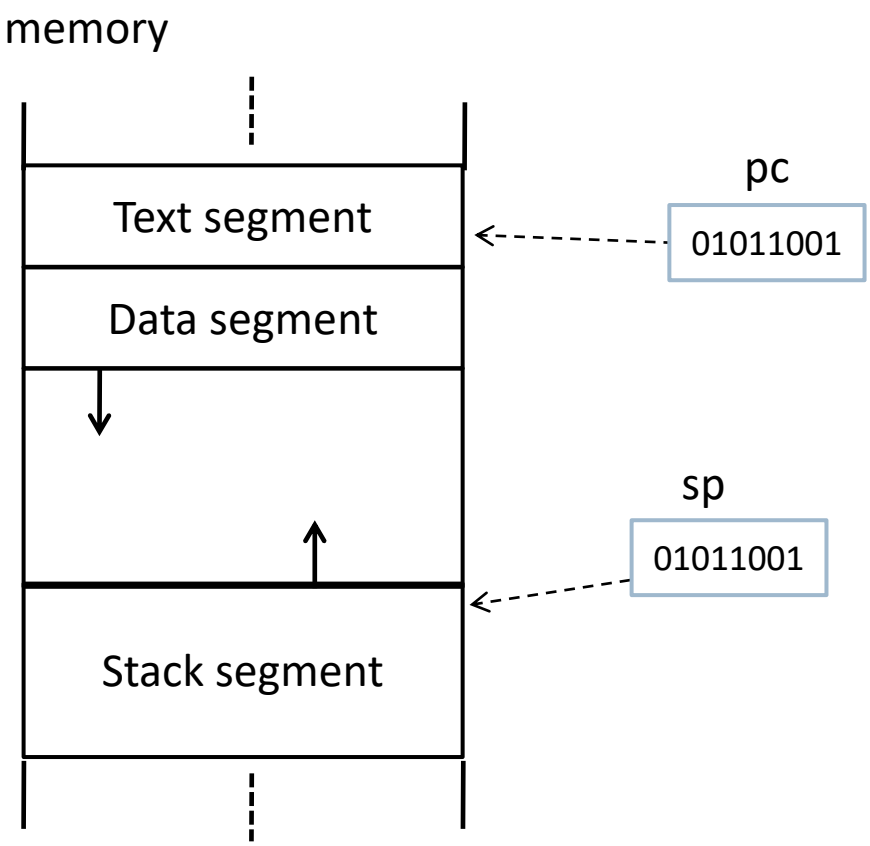
- Computers have two storage elements:
  - Registers
  - Memory

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

Program execution



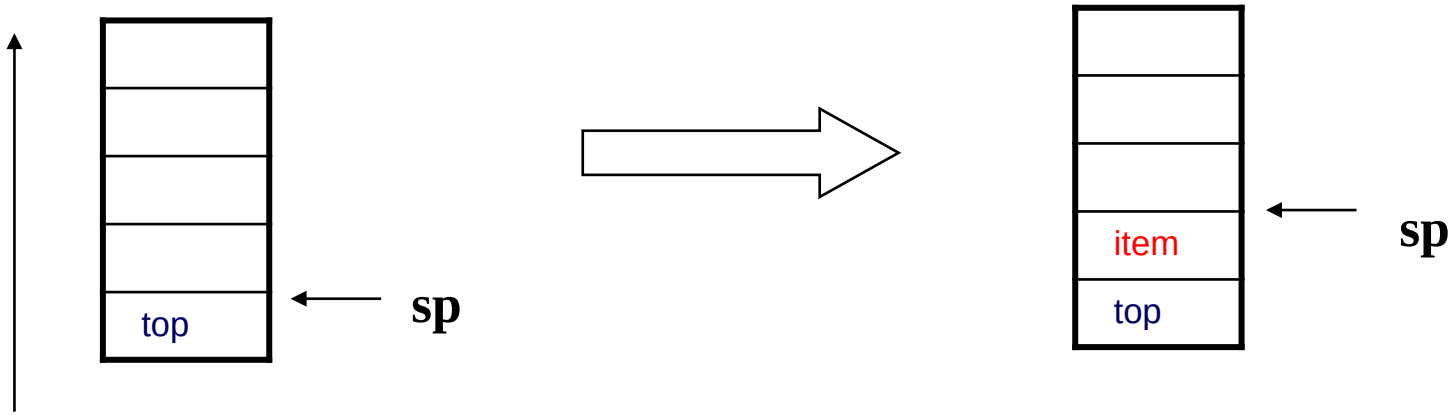
Memory map of a process



- User programs are divided in segments:
  - Text segment (code)
    - Machine instructions
  - Data segment
    - Static data, global variables
  - Stack segment
    - Local variables
    - Function context

Stack

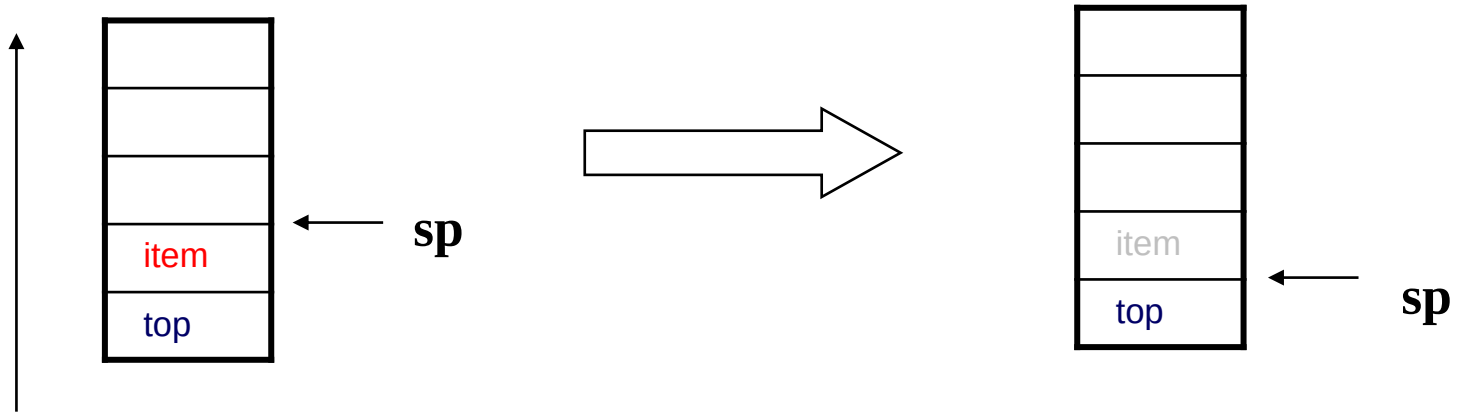
PUSH Reg Push an element in stack (item)



Stack grows to lower memory addresses

Stack

POP Reg Pop last element and copy value in a register



Stack grows to lower memory addresses

Before to start

- RISC-V does not have PUSH or POP instructions
- Stack pointer (`sp`) is used to manage the stack
  - We assume that stack pointer points to the last element in the stack

PUSH $t0

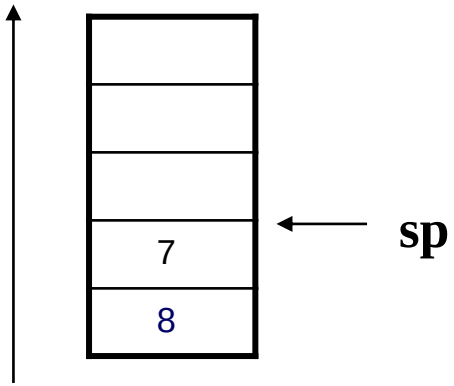
```
addi sp, sp, -4
sw   t0, 0(sp)
```

POP $t0

```
lw   t0, 0(sp)
addu sp, sp, 4
```

PUSH operation in RISC-V

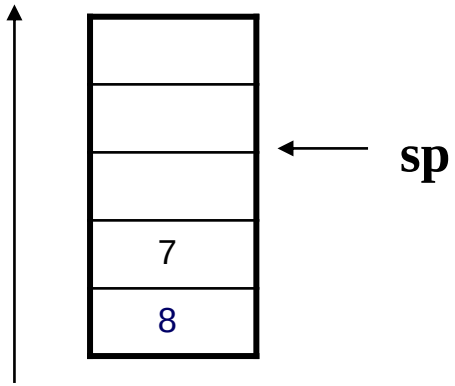
```
...
li   t2,  9
subu sp, sp, 4
sw   t2 0(sp)
...
```



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

PUSH operation in RISC-V

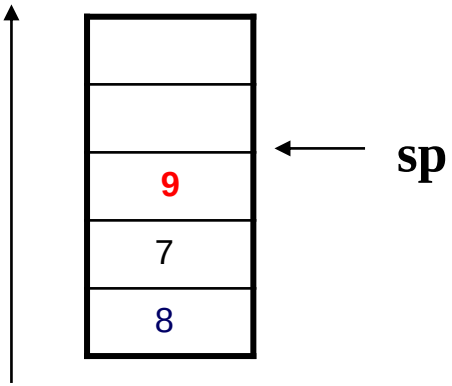
```
…
li    t2,  9
subu sp, sp, 4
sw    t2 0(sp)
…
```



- PUSH (1/2): Subtract 4 to stack pointer to insert a new word in the stack

PUSH operation in RISC-V

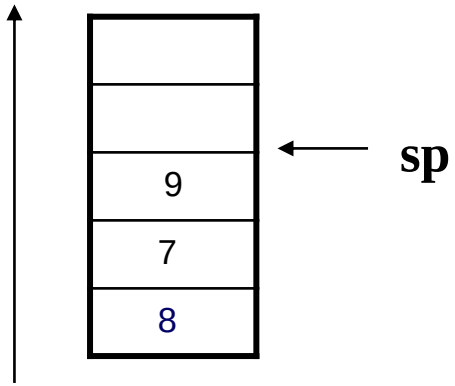
```
…
li   t2,  9
subu sp, sp, 4
sw   t2 0(sp)
…
```



- PUSH (2/2): Insert the content of register t2 on the stack

POP operation in RISC-V_{32}

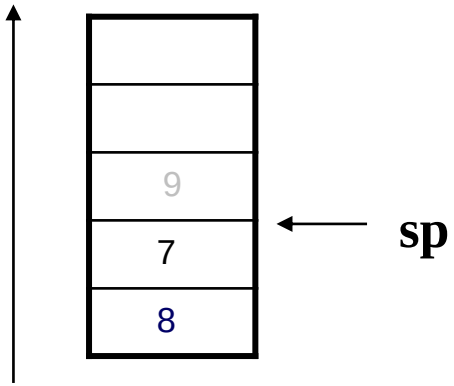
```
...
lw   t2 0(sp)
addi sp, sp, 4
...
```



- Copy in t2 the value of the first element of the stack (9)

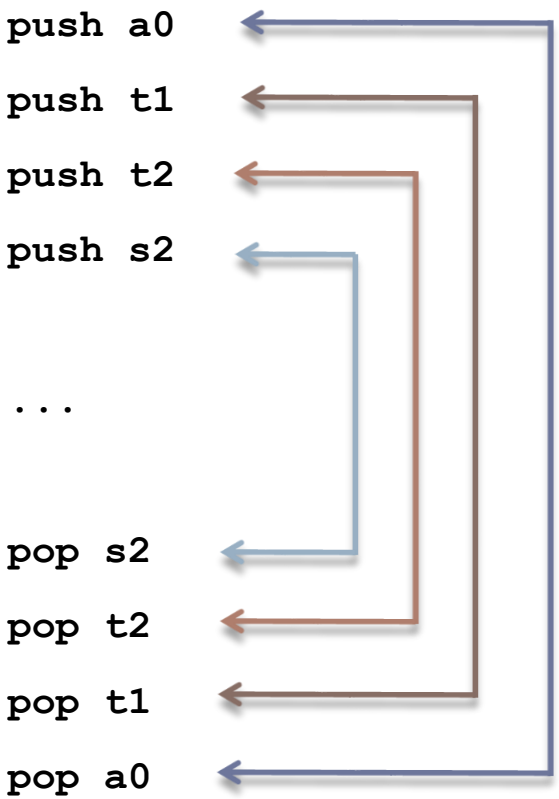
POP operation in RISC-V_{32}

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



- Update the stack pointer to point to the new top.
- The data (9) continues in memory but will be overwritten in a future PUSH operations (or similar store instruction).

Stack
Consecutive PUSH and POP



Stack
Consecutive PUSH and POP

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

...

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

```
addu sp sp -4
sw   a0  (sp)
addu sp sp -4
sw   t1  (sp)
addu sp sp -4
sw   t2  (sp)
addu sp sp -4
sw   s2  (sp)
```

...

```
lw s2 (sp)
addu sp sp 4
lw t2 (sp)
addu sp sp 4
lw t1 (sp)
addu sp sp 4
lw a0 (sp)
addu sp sp 4
```

Stack
Consecutive PUSH and POP

```
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    (sp)
```

...

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

Example 1
(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 1
(2) Analyze how to pass the arguments

- The arguments in RISC-V are placed in a0 … a7
- The results in RISC-V are collected in a0, a1
  - It will be seen in more detail later
- If more than 8 parameters need to be carried, the first eight in registers a0 … a7 and the rest on the stack (leaving room in the stack for the first eight)

- In the invocation `z=factorial(5);`
  - One input parameter/argument: a0
  - One result in a0

Example 1
(3) Translate to assembly language

Input parameter in a0
Result in v0

```
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 v0           # result
          # print_int(z)
          li a7, 1
          ecall
          ...

factorial: li   s1, 1   #s1 for r
           li   s0, 1   #s0 for i
     loop: bgt  s0, a0, end
           mul  s1, s1, s0
           addi s0, s0, 1
           beq  x0, x0, loop
      end: mv   v0, s1  #result
           jr   ra
```

Example 1
(4) Analyze the registers modified

```
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
     loop: bgt  s0, a0, end
           mul  s1, s1, s0
           addi s0, s0, 1
           beq  x0, x0, loop
      end: move v0, s1  #result
           jr   ra
```

- The function uses (modifies) registers $s0 and $s1
- If this registers are modified, the caller function (main) can be affected
- Then, factorial function must store this registers in the stack at the beginning and restore them at the end

Example 1
(5) Store registers in stack

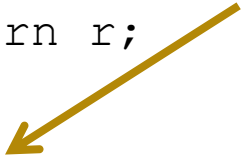
```
int factorial(int x) {
  int i;
  int r=1;
  for (i=1;i<=x;i++) {
       r*=i;
  }
  return r;
}
```

```
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
bucle:     bgt    s0, a0, fin
           mul    s1, s1, s0
           addi   s0, s0, 1
           beq    x0, x0, bucle
fin:       mv     a0, s1   # resultado
           lw     s1, 0(sp)
           lw     s0, 4(sp)
           addi   sp, sp, 8
           jr     ra
```

- Is not necessary to store $ra in stack, function is terminal
- Registers $s0 and $s1 are stored in the stack because are modified
  - If we had used $t0 and $t1, it would not have need to copy $t* in the stack (because temporary registers are not saved)

Example 2

```
int main()
{
  int z;

  z=f1(5, 2);
  print_int(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: call

```
int main()
{
  int z;

  z=f1(5, 2);
  print_int(z);
}

…
```

```
li   a0,  5
li   a1,  2
jal  f1

li   a7, 1
ecall
```

```
# primer argumento
# segundo argumento
# llamada
# resultado (a0)

 # llamada para
 # imprimir un int
```

- Parameters are passed in $a0 and $a1
- Result is returned in $v0

Example 2: function f1

…

```
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: analyze registers modified in f1

…

```
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, then store them in the stack
- Register $ra is modified in instruction jal f2
- Register $a0 is modified to pass the argument to function f2, but f1 by convention does not need to keep its value on stack unless f1 needs the value after call f2

Example 2: storing registers in the stack

…

```
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
```

Example 2: function 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 v0, t0, a0
    jr  ra
```

- Function f2 does not modify register $ra (is terminal)
- Register $t0 is not stored in stack because this is a temporal register, and its value does not need be preserved

Simplified calling convention: argument to functions

- The integer arguments are placed in a0 … a7
  - If more than 8 parameters are need to be passed, the first eight in registers a0…a7 and the rest on the stack
  - Integer includes high-level datatypes such as char, int, etc.
- The float arguments are placed in f10 … f17
  - If more than 8 parameters need to be passed, the remainder on the stack
- The double arguments are placed in f10 … f17
  - Registers are packed two by two (f10-f11, f12-f13, …)
  - If more than **4** parameters need to be passed, the remainder in the stack

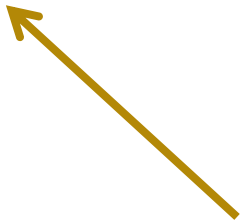
Simplified calling convention: return of results in RISC-V

- Use a0 and a1 for integer type values
- Use f0 and f1 for float type values
- Use f0-f1 for double type values

- In case of complex structures/values:
  - They must be left in the stack.
  - The space is reserved by the function that makes the call

Calling convention: registers in RISC-V

```
li      t0, 8
li      s0, 9

li      a0, 7   # argument
jal     ra funcion
```



What are the values of the registers t0 and s0?

Simplified calling convention: registers in RISC-V

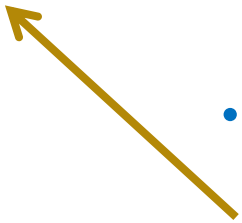
Register name	Register number	Use	Preserving value
zero	0	Constant 0	No
ra	1	Return address	**Yes**
sp	2	Stack pointer	**Yes**
gp	3	Global area pointer	No
tp	4	Thread pointer	No
t0...t2	5…7	Temporary	No
s0/fp	8	Temporary / Stack frame pointer	**Yes**
s1	9	Temporary to be preserved	**Yes**
a0...a7	10…17	Arguments	No
s2… s11	18…27	Temporary to be preserved	**Yes**
t3...t6	28…31	Temporary	No

Simplified calling convention: registers in RISC-V (floating point)

Nombre registro	Uso	Preservar el valor
f0 … f7	Temporary	No
f8 … f9	Temporary to be preserved	**Yes**
f10 … f11	Arguments/return	No
f12 … f17	Arguments	No
f18 … f27	Temporary to be preserved	**Yes**
f28 … f31	Temporary	No

Calling convention: registers in RISC-V

```
li      t0, 8
li      s0, 9

li      a0, 7   # argument
jal     ra, funcion
```



- According to the convention, s0 will still be 9, but there is no guarantee that t0 will kept its 8 or a0 7.
- If we want t0 to continue to be 8, it must be saved on the stack before calling the function.

Calling convention: registers in MIPS

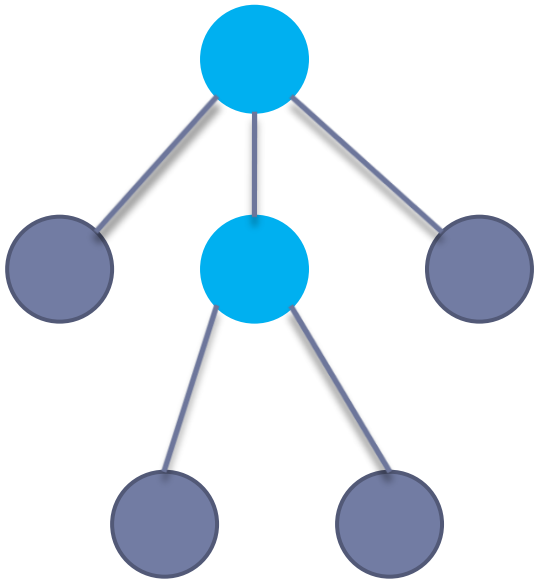
```
li      t0, 8
li      s0, 9

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

li      a0, 7   # parámetro
jal     ra, función

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

It is saved in the stack before the call…

… and the value is recovered after

Types of functions

- Terminal function.
  - Does not call other functions.
- Not terminal function.
  - Call other functions.



Activation of functions
stack frame

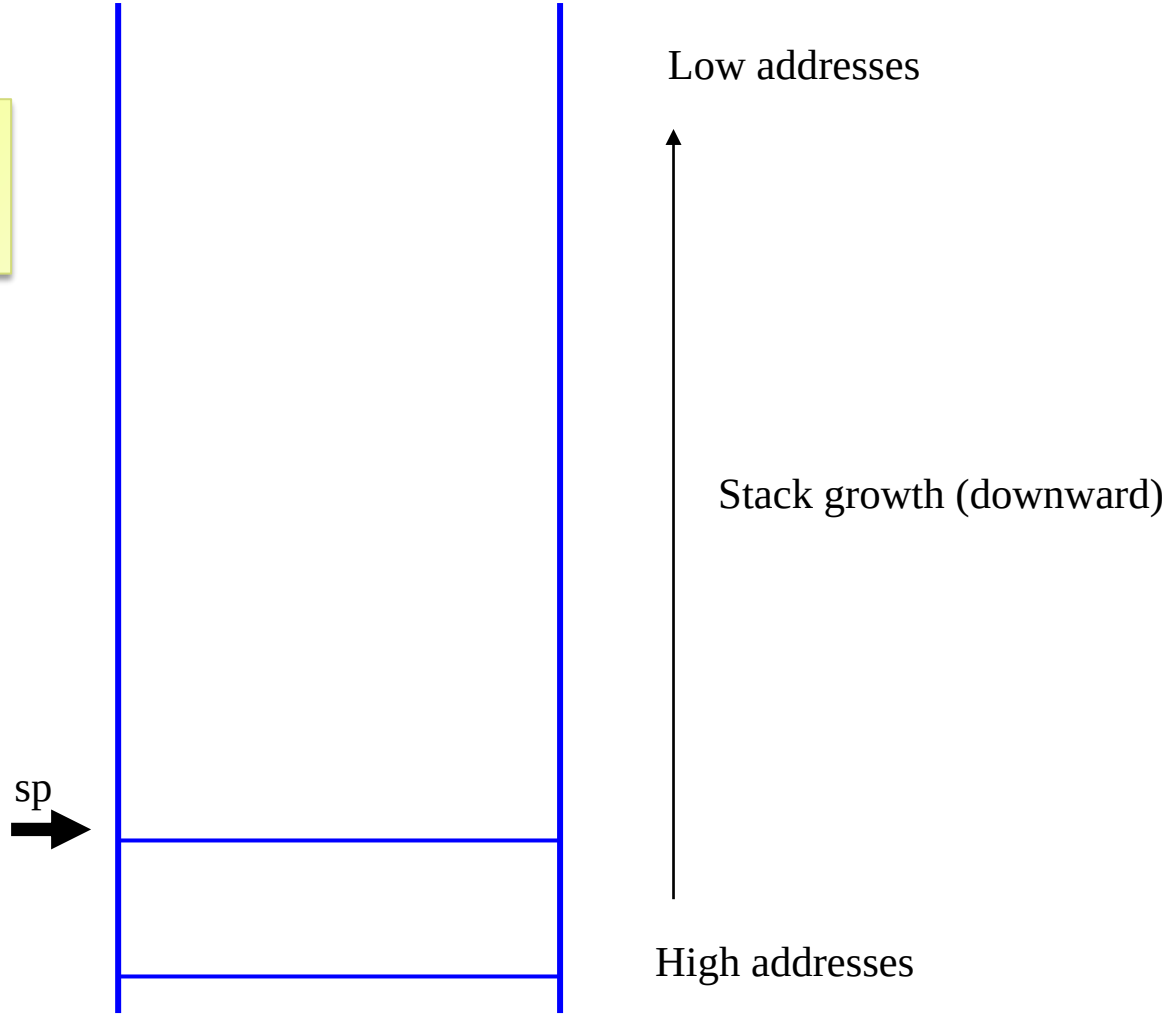
- The stack frame or activation register is the mechanism used by the compiler to activate functions in high-level languages.
- The stack frame is built on the stack by the calling procedure and the called procedure.
- The stack frame stores:
  - Parameters passed by the caller function
  - The stack frame pointer of the caller function
  - Registers saved by the procedure (`ra` in not terminal function)
  - Local variables

General function call steps (simplified version)

Caller function	Calle function
Save the registers not preserved across the call (t_, a_, ...)	
Parameter passing + (if needed) allocation of space for values to be returned	
Make de call (jal)	
	Stacking frame reservation
	Save registers (ra, , s_)
	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 records, freeing the reserved stack space	

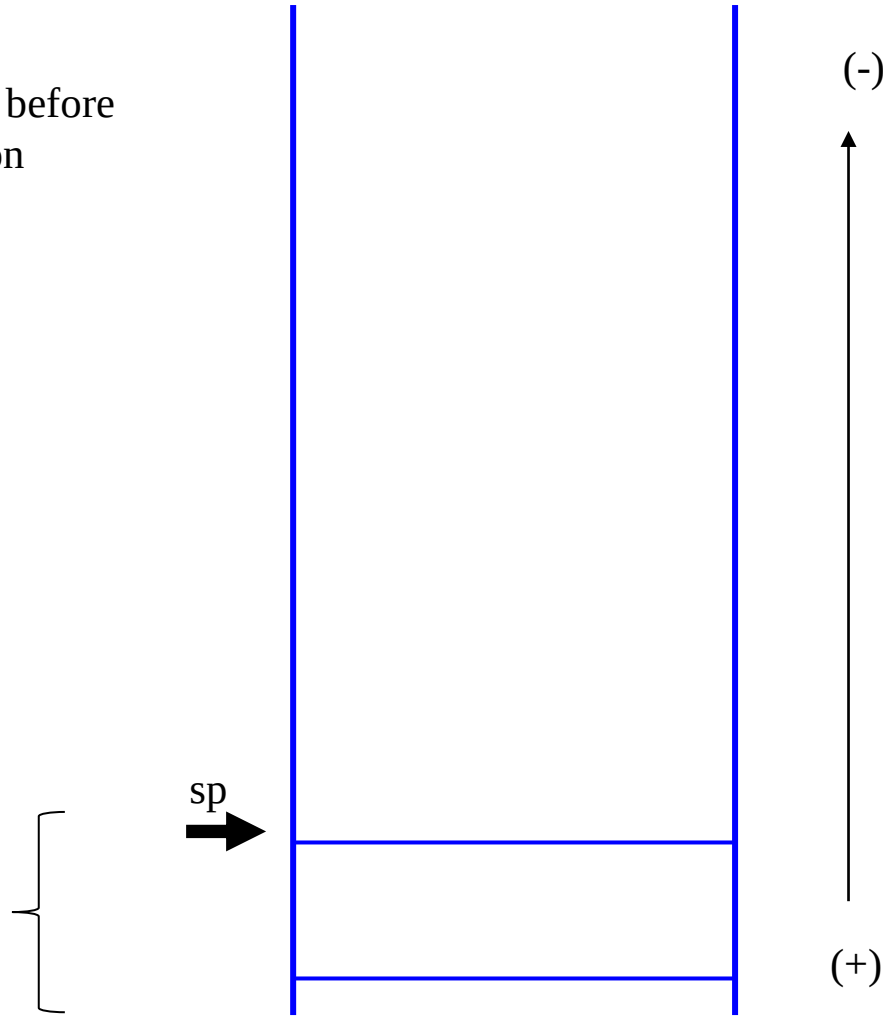
Construction of the stack frame
caller function

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



Construction of the stack frame
caller function

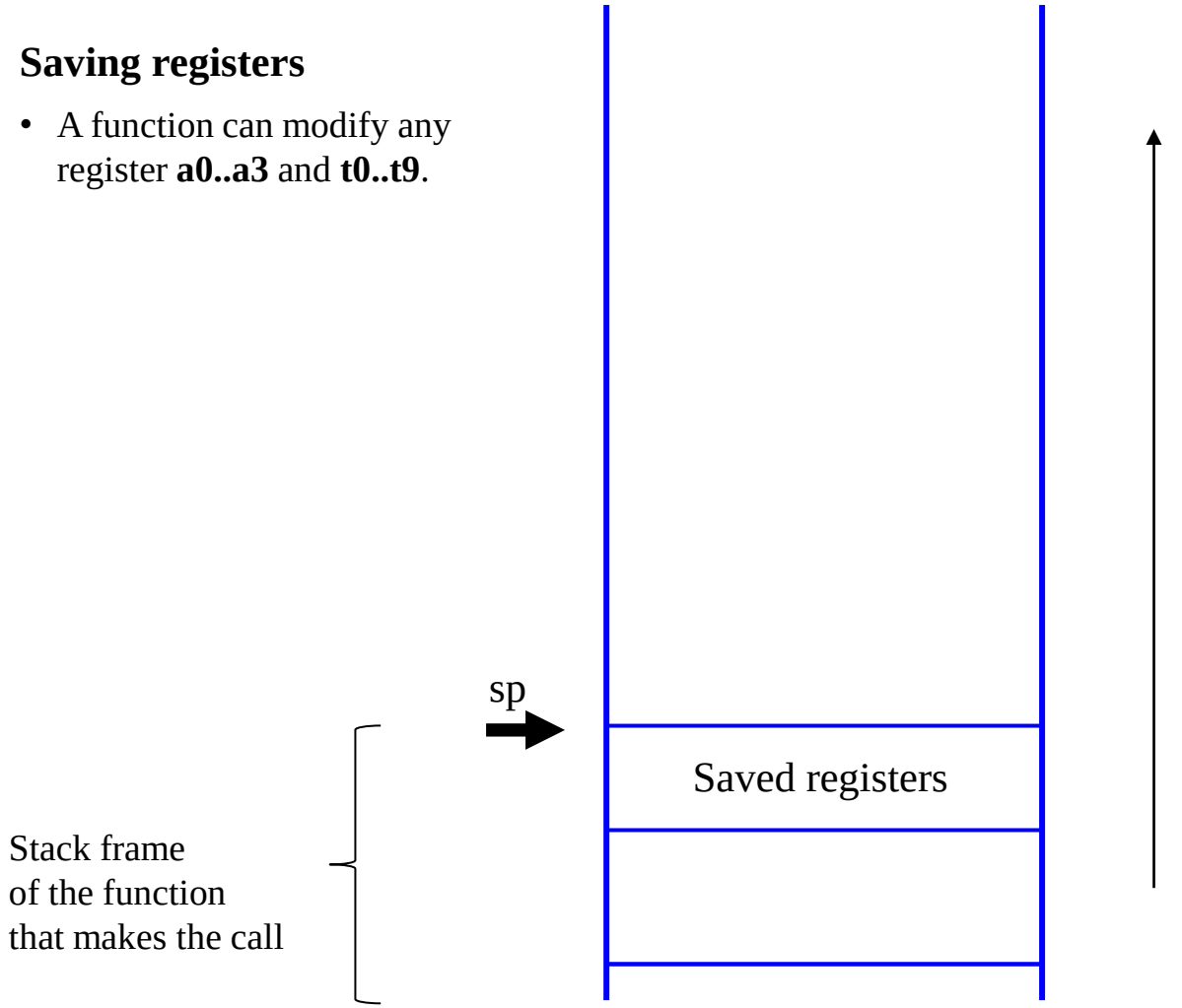
- **Initial** situation before calling a function



Stack frame
of the function
that makes the call

Construction of the stack frame
caller function

- **Saving registers**
  - A function can modify any register **a0..a3** and **t0..t9**.



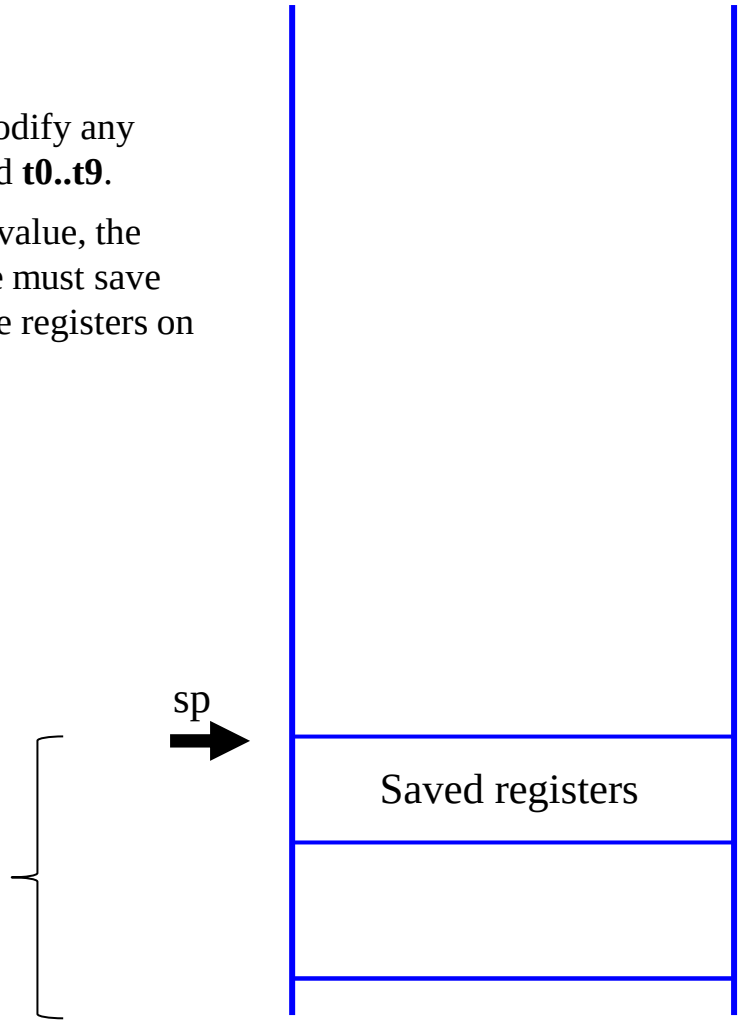
Example:

```
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
caller function

- **Saving registers**
  - A function can modify any register **a0..a3** and **t0..t9**.
  - To preserve their value, the calling subroutine must save the values of these registers on the stack.



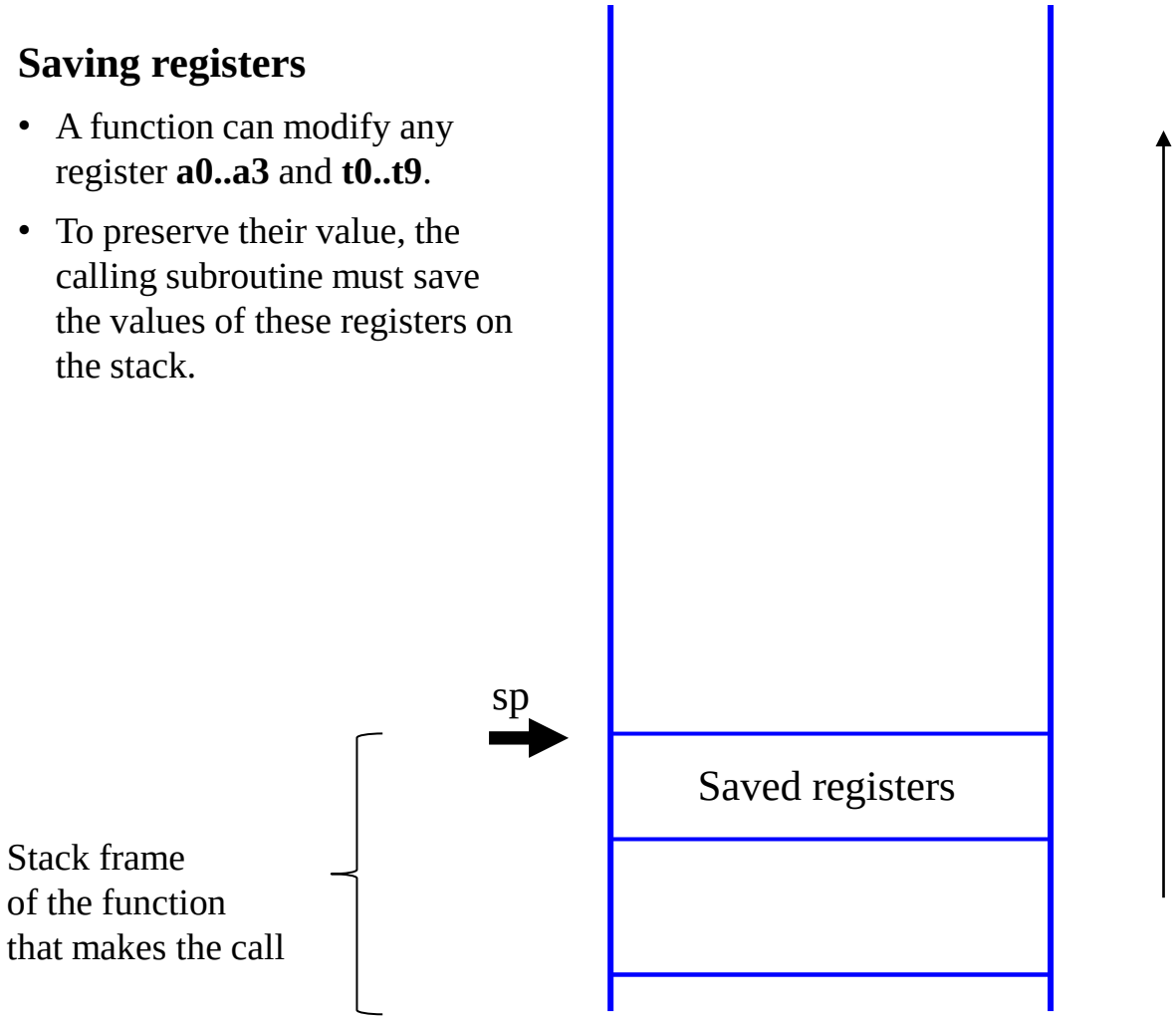
Example:

```
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
caller function

- **Saving registers**
  - A function can modify any register **a0..a3** and **t0..t9**.
  - To preserve their value, the calling subroutine must save the values of these registers on the stack.

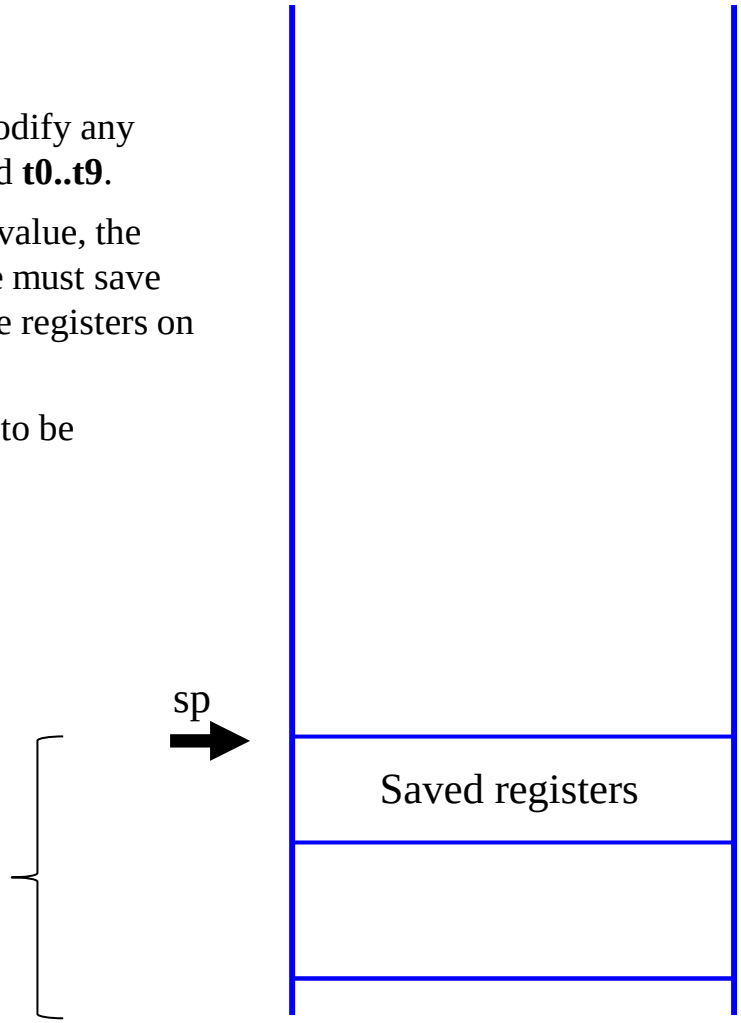


Example:

```
subu 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 **a0..a3** and **t0..t9**.
  - To preserve their value, the calling subroutine must save the values of these registers on the stack.
    - they will have to be restored later.

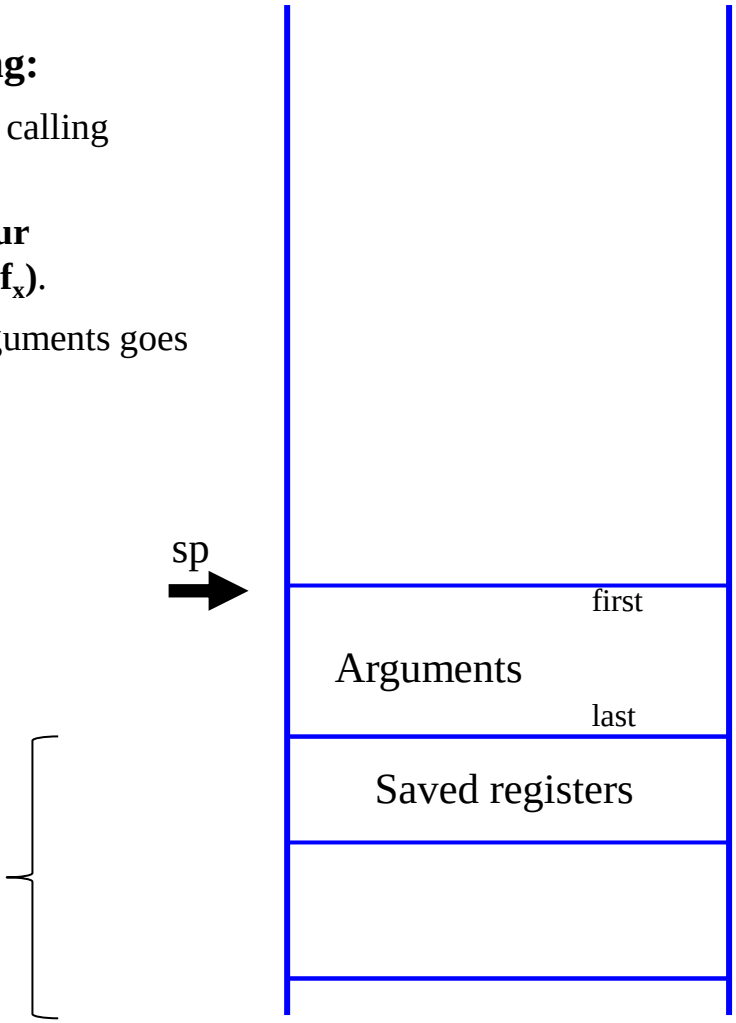


Example:

```
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

- **Argument passing:**
  - Before calling the calling procedure.
  - Leave the **first four arguments** in $\mathbf{a_x}$ **($\mathbf{f_x}$)**.
  - The rest of the arguments goes to the stack

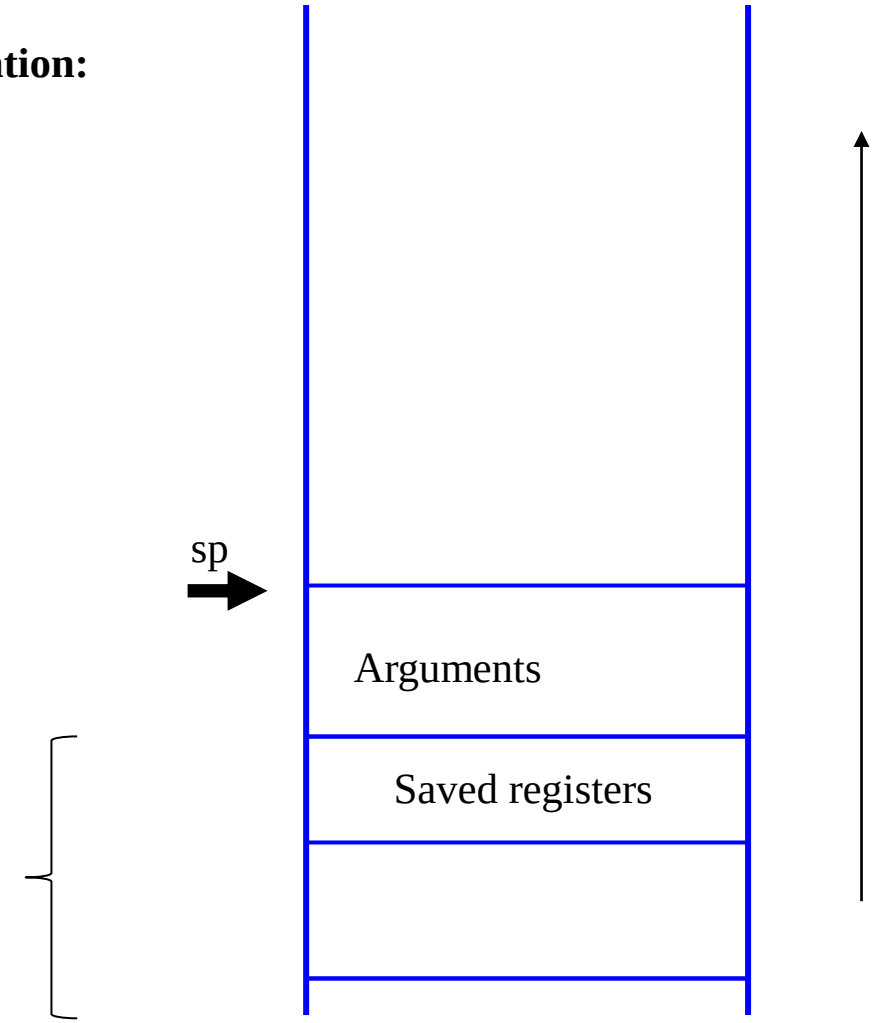


Example (10 arguments):

```
li      a0, 1
li      a1, 2
li      a3, 3
li      a4, 4
li      a5, 1
li      a6, 2
li      a7, 3

addi sp, sp, -8

li      t0, 5
sw      t0, 4(sp)

li      t0, 6
sw      t0,  0(sp)
```

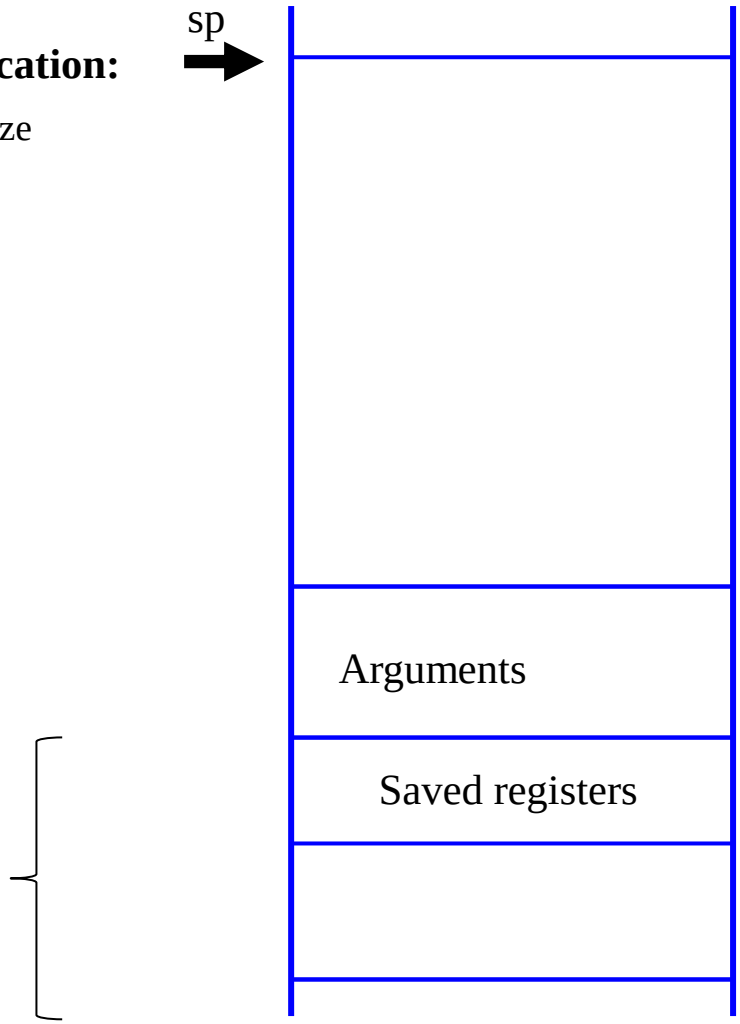
Construction of the stack frame
caller function

- **Function invocation:**
  - jal ra function



Construction of the stack frame
called function

- **Stack frame allocation:**
  - sp = sp – frame size
  - **Space for**:
    - ra, fp
    - s0…s7
    - Local variables

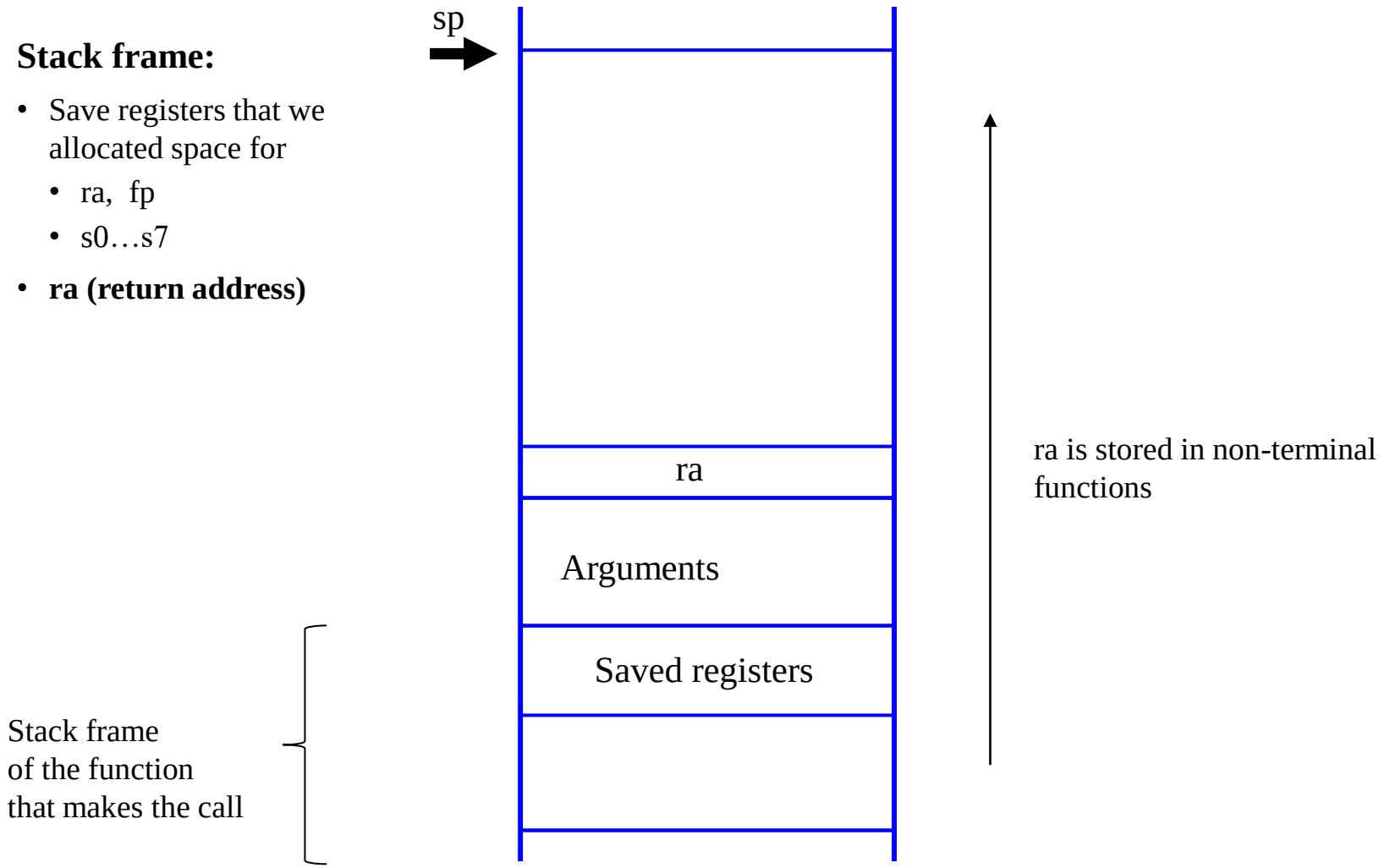


Example:

addi sp sp -<fr.sz.>

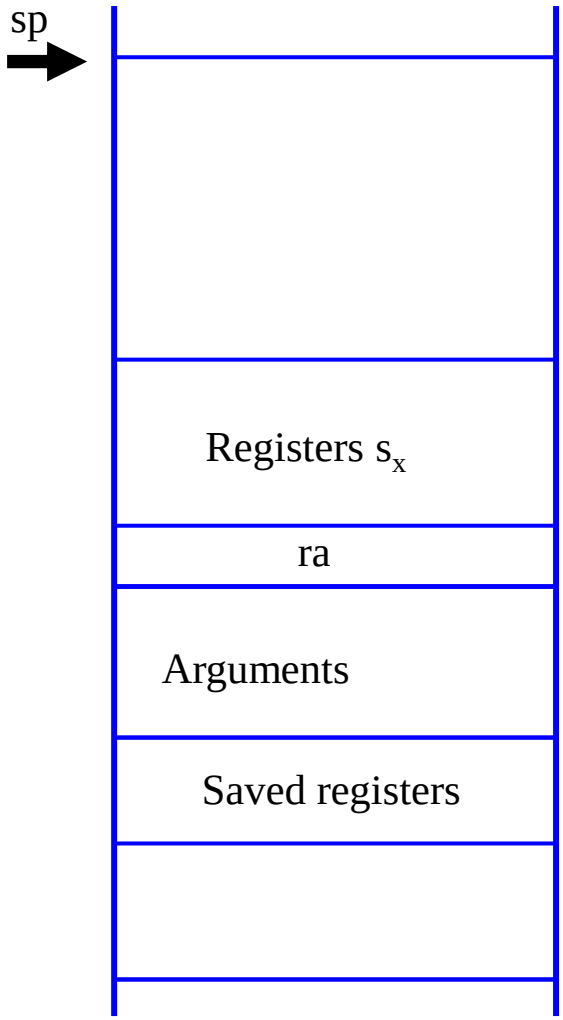
Construction of the stack frame
called function

- **Stack frame:**
  - Save registers that we allocated space for
    - ra, fp
    - s0…s7
  - **ra (return address)**



Construction of the stack frame
called function

- **Stack frame:**
  - Save registers that we allocated space for
    - ra, fp
    - s0…s7
  - **s_x registers**
    - The s_x registers to be modified are saved.
    - A function cannot, by convention, modify the s_x registers (the t_x and the a_x can be modified).

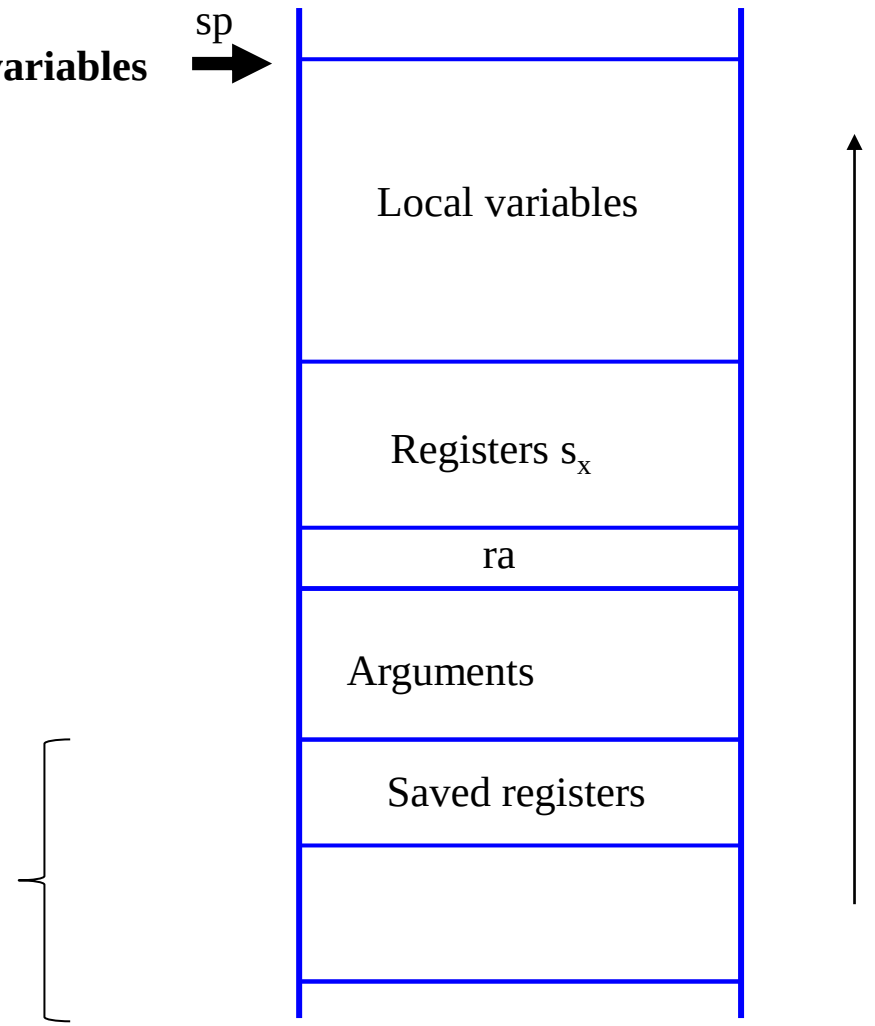


Example:

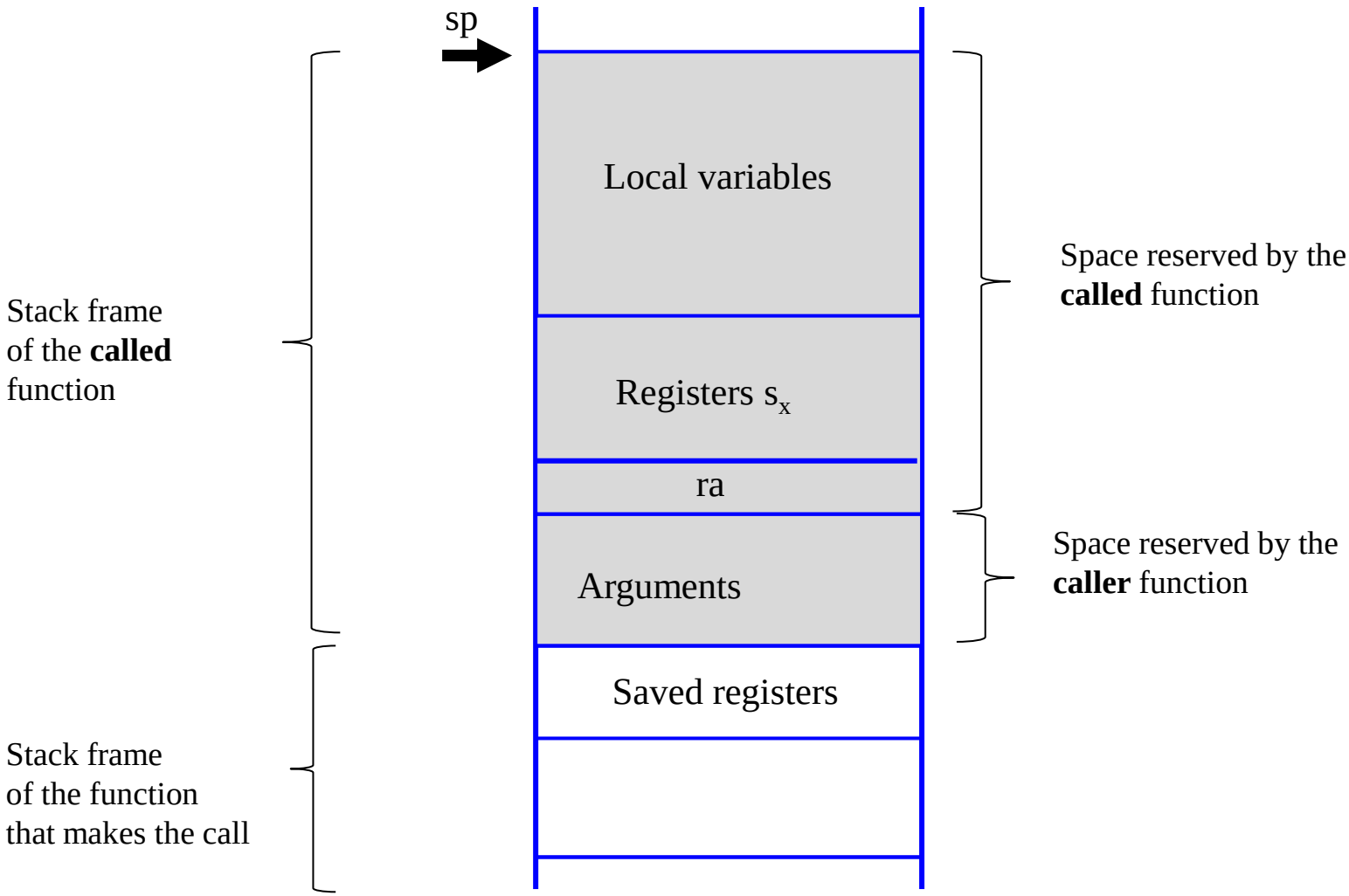
```
function:
   addi sp sp -<fr.sz.>
   sw ra <fr.sz-4>(sp)
   sw s0 <fr.sz-8>(sp)
   sw s1      <...>(sp)
   …
```

Construction of the stack frame
called function

- **Space for local variables**

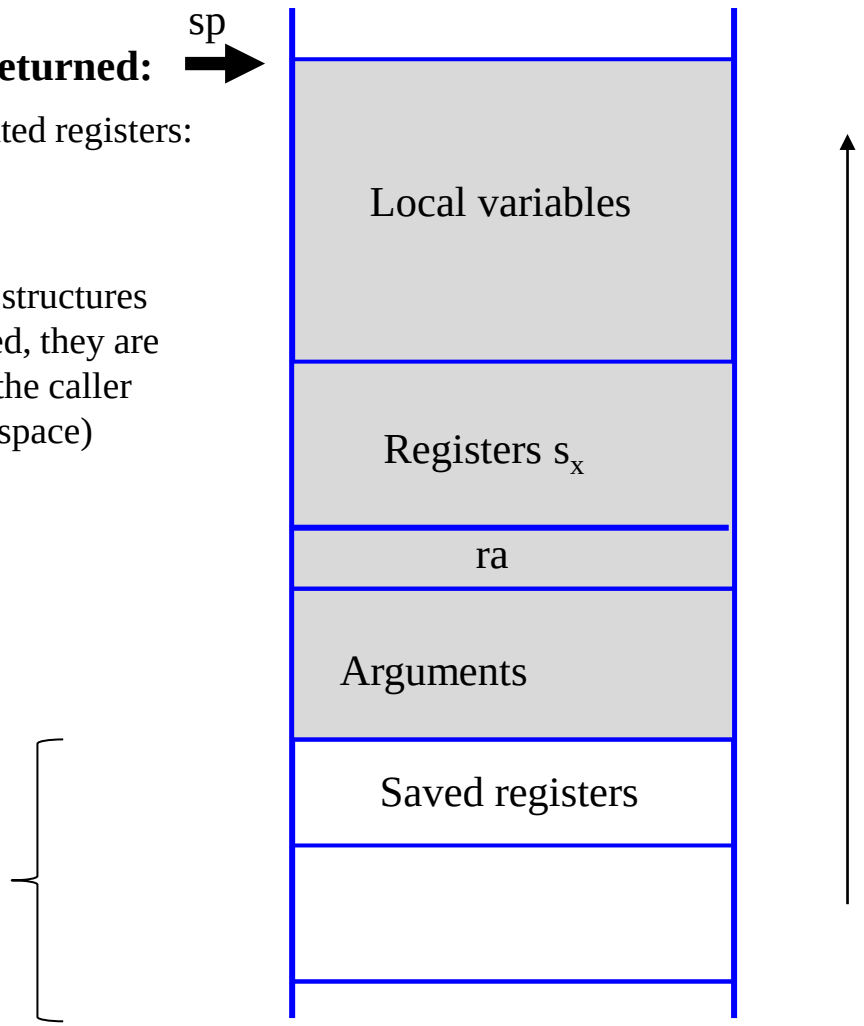


Stack frame construction



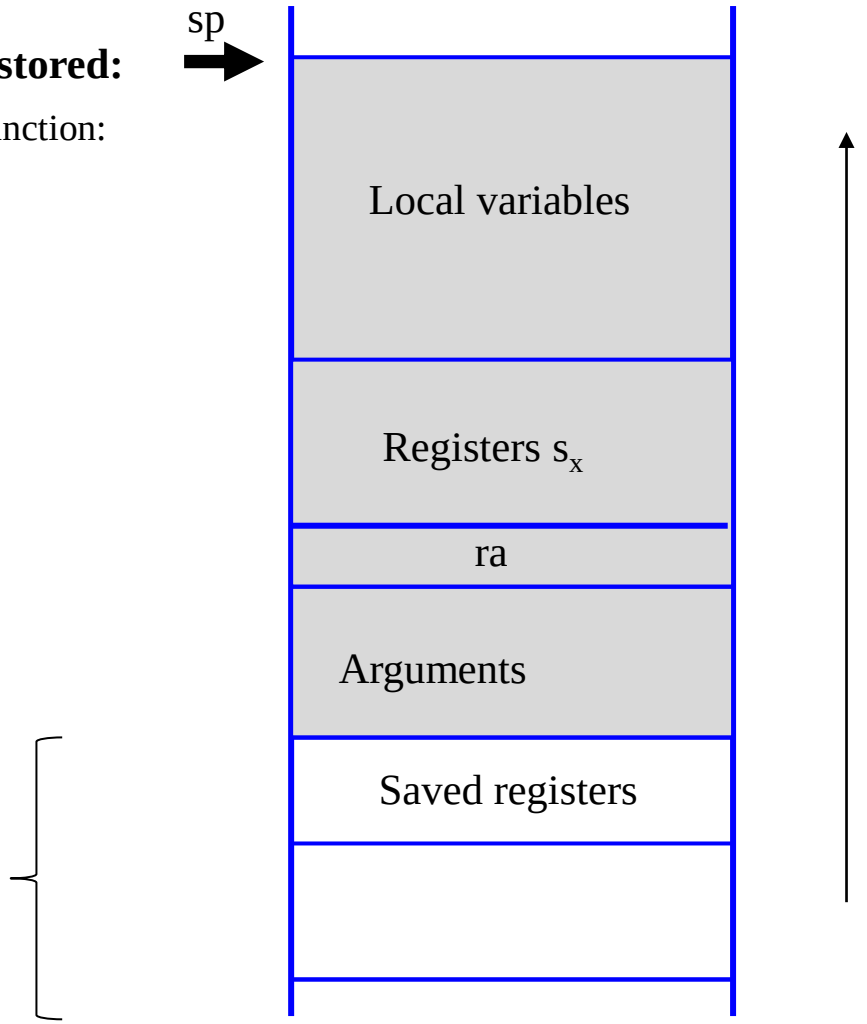
Subroutine termination
called function

- **The results are returned:**
  - Use the appropriated registers:
    - a0, a1
    - f10, f11
  - If more complex structures need to be returned, they are left on the stack (the caller must allocate the space)



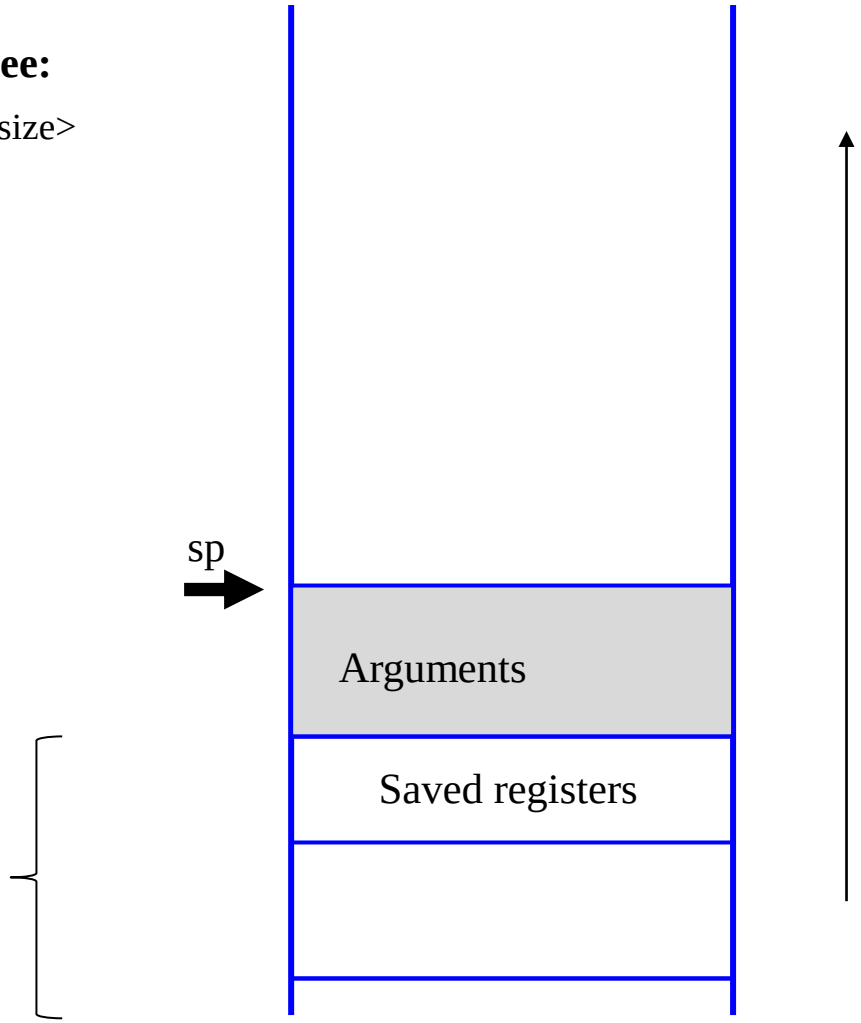
Subroutine termination
called function

- **The saved are restored:**
  - Saved in called function:
    - s_x
    - ra



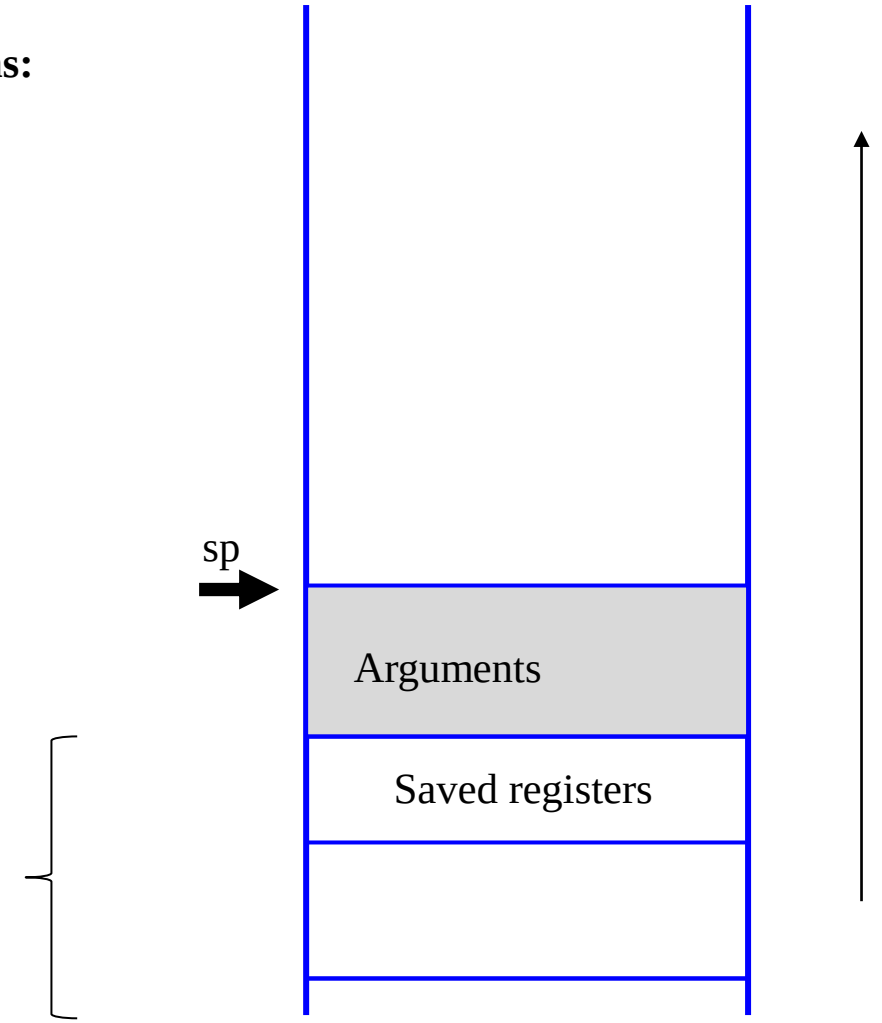
Subroutine termination
called function

- **Stack frame is free:**
  - sp = sp + <frame size>



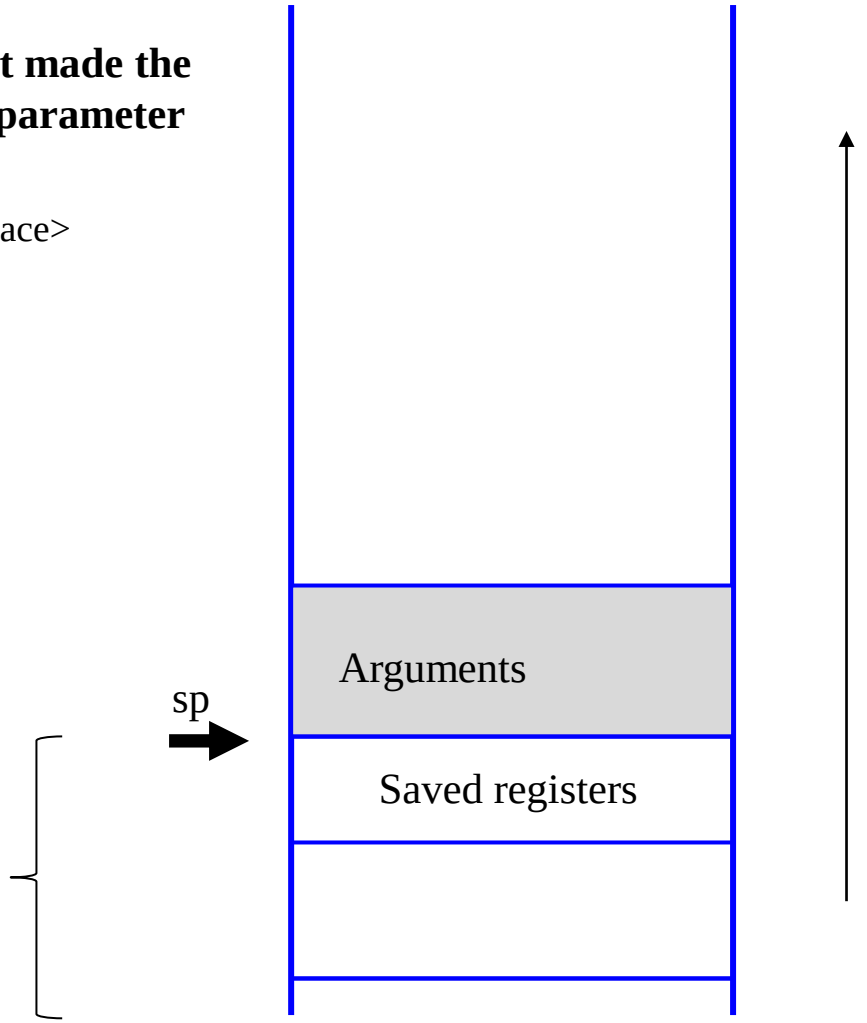
Subroutine termination
called function

- **Function returns:**
  - jr ra



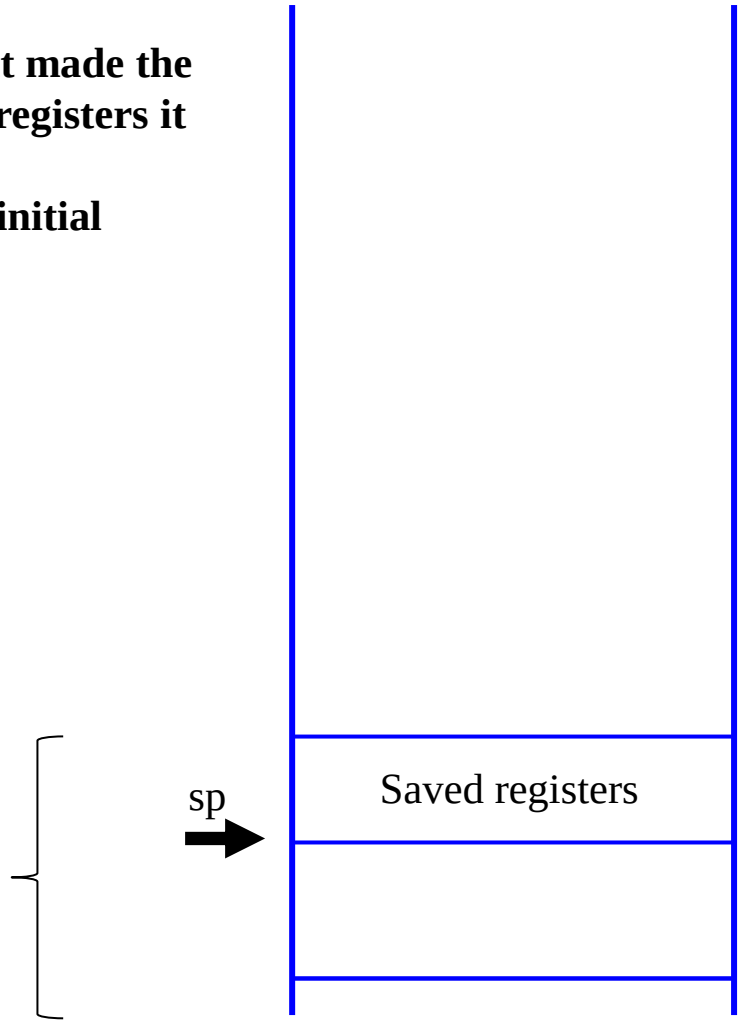
Subroutine termination
calle function

- **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.**
- **Adjust sp to the initial position**

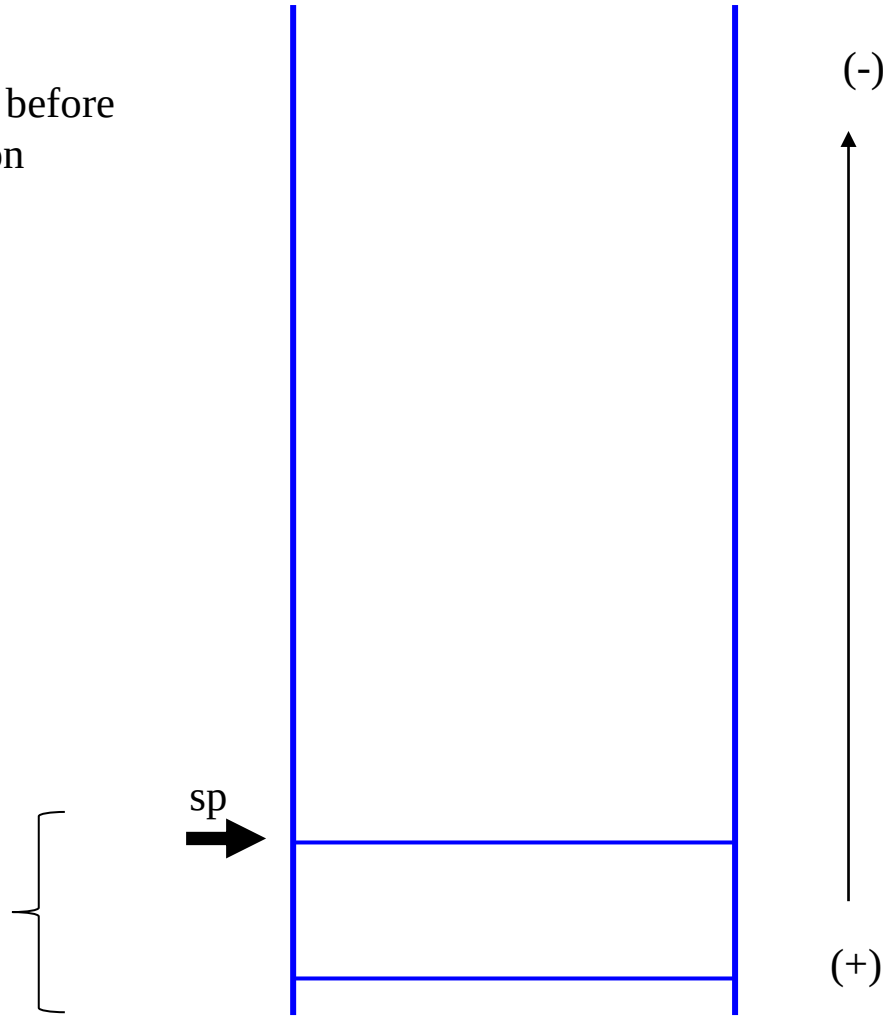


Example:

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

li    a0, 5
jal   funcion

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

State after subroutine termination

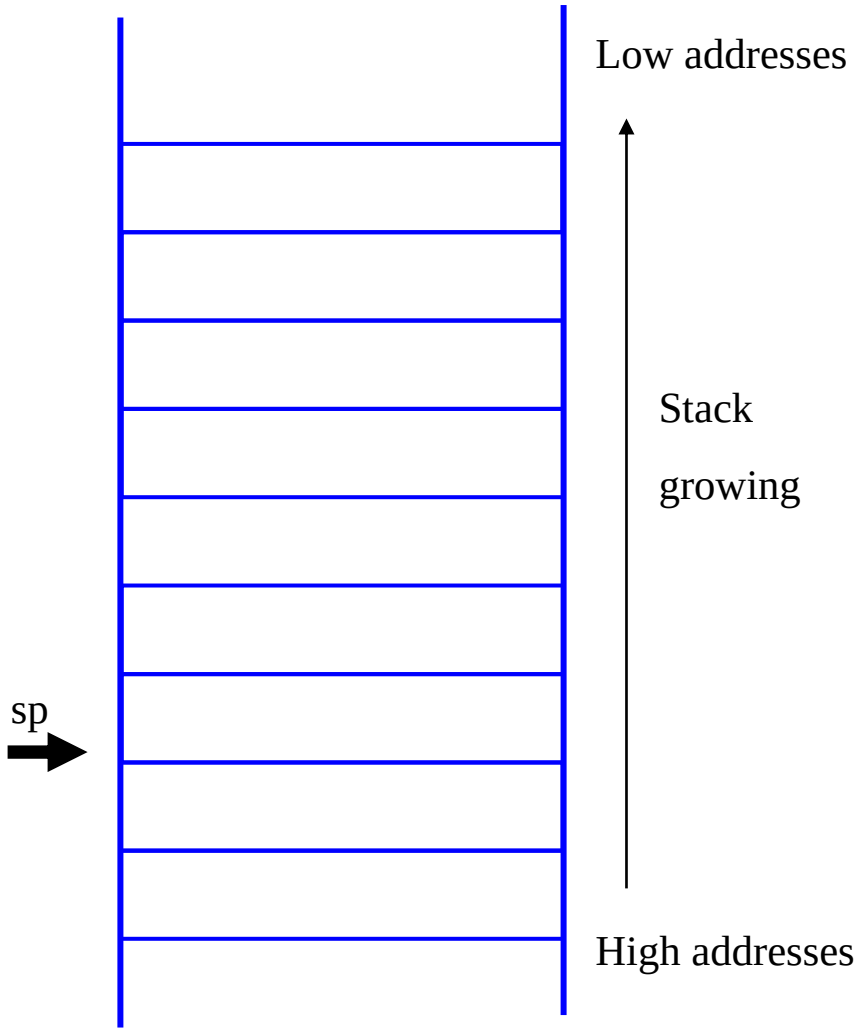
- **Initial** situation before calling a function



Stack frame
of the function
that makes the call

Access to parameters and local variables using the stack frame

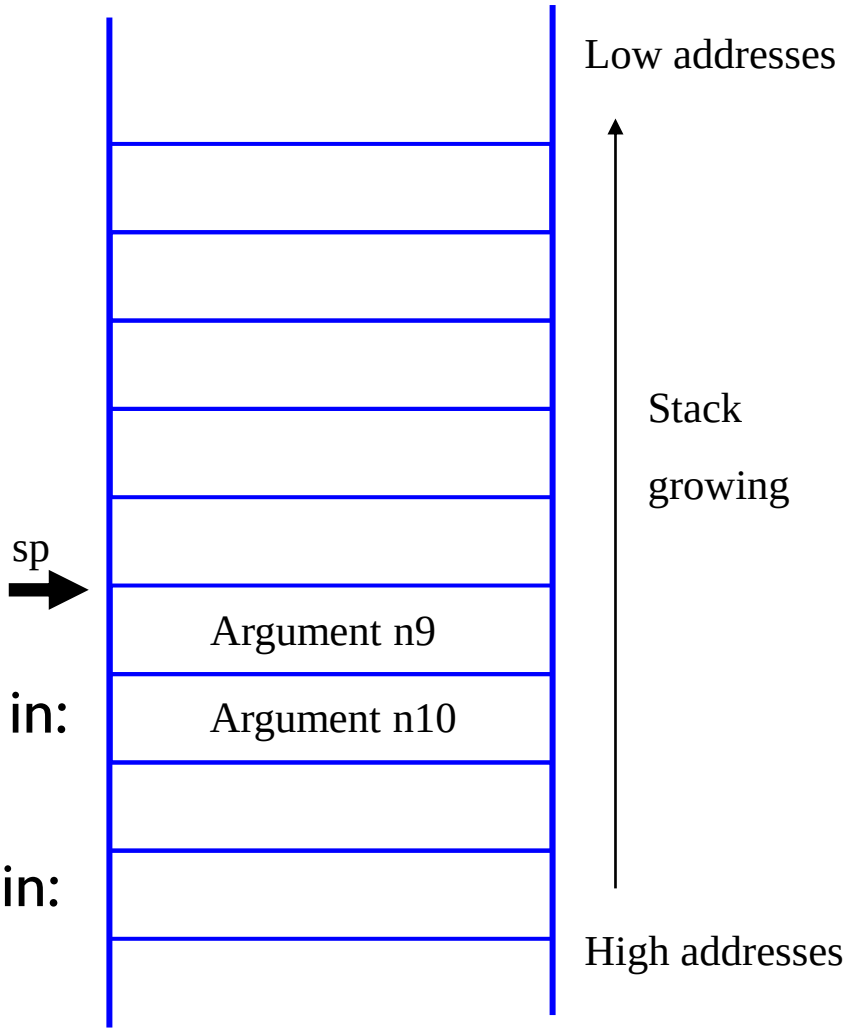
```
int f (int n1, n2, n3, n4, n5
           n6, n7, n8, n9, n10)
{
   int v[4];
   int k;

   k = n1+n2+n3+n4+n5+n6+n7+n8+n9+n10;
   for (k= 0; k <3; k++)
        v[i]=k;

   return (v[1]);
}
```



- If a call to f(…) is made…

Access to parameters and local variables using the stack frame

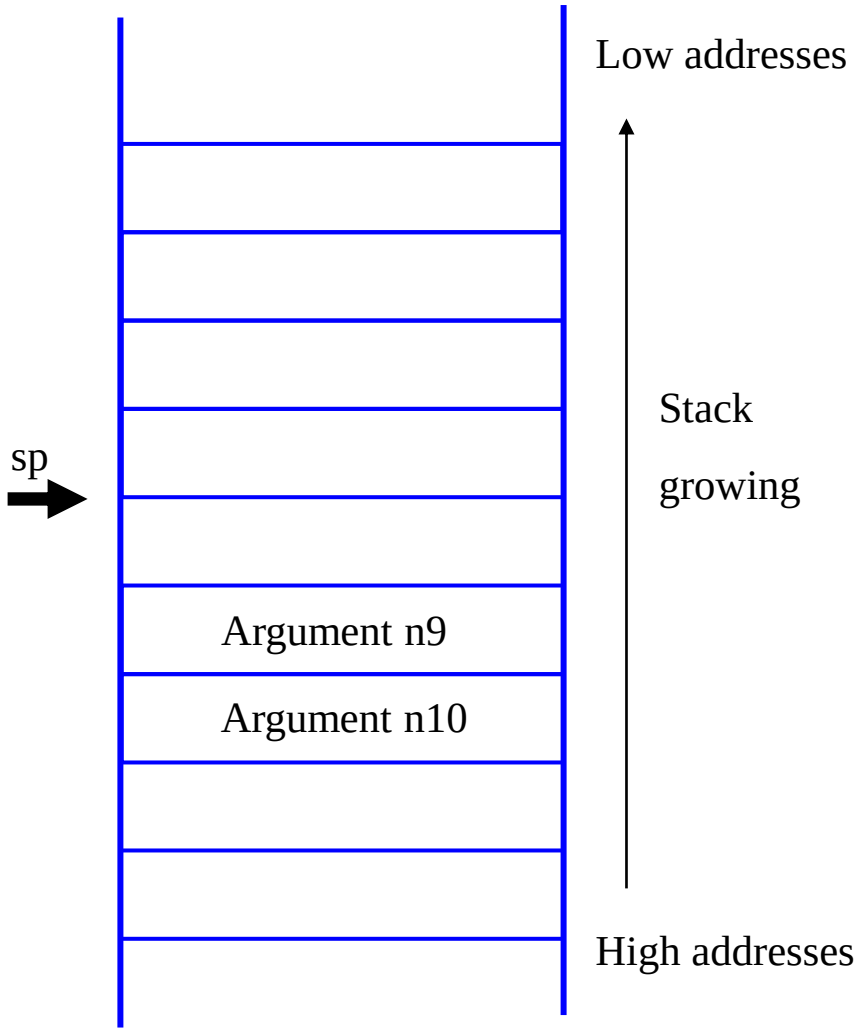
```
int f (int n1, n2, n3, n4, n5
           n6, n7, n8, n9, n10)
{
   int v[4];
   int k;

   k = n1+n2+n3+n4+n5+n6+n7+n8+n9+n10;
   for (k= 0; k <3; k++)
        v[i]=k;

   return (v[1]);
}
```



- Arguments `n1...n8` are placed in:
  - `a0...a3`
- Arguments `n9, n10` are placed in:
  - The stack

Access to parameters and local variables using the stack frame

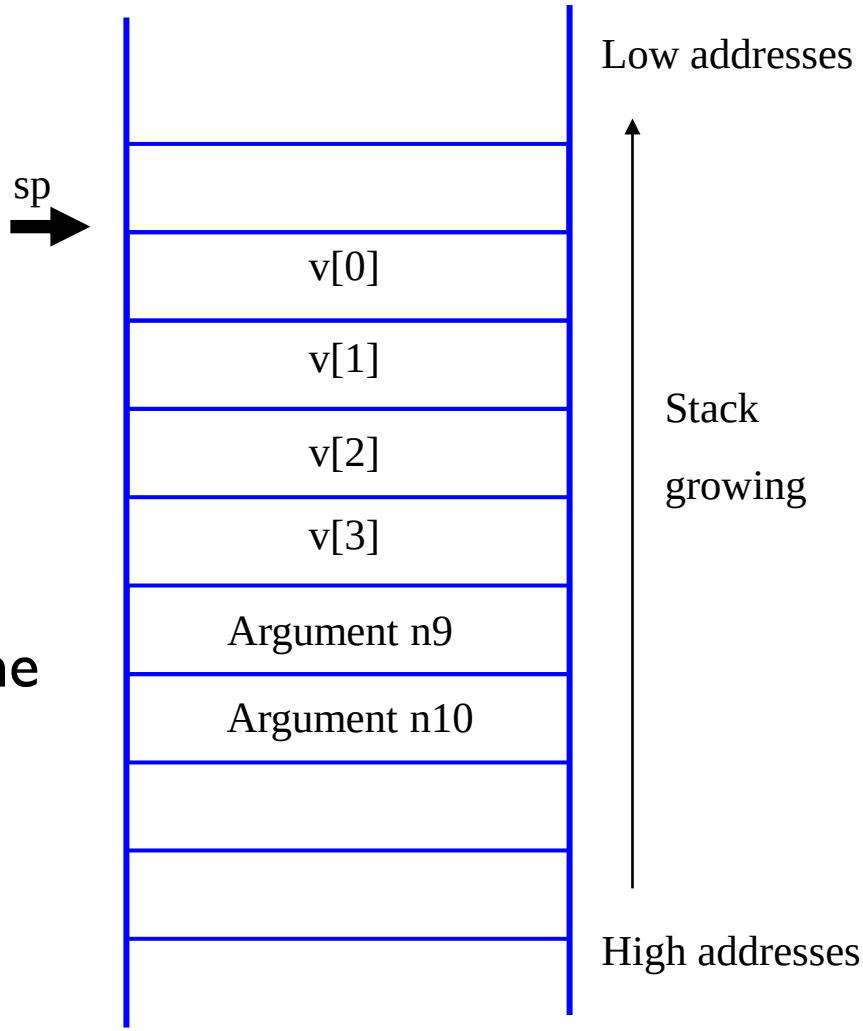
```
int f (int n1, n2, n3, n4, n5
           n6, n7, n8, n9, n10)
{
   int v[4];
   int k;

   k = n1+n2+n3+n4+n5+n6+n7+n8+n9+n10;
   for (k= 0; k <3; k++)
        v[i]=k;

   return (v[1]);
}
```



- Once the function has been invoked, f(…) must:
  - Save a copy of the registers to be preserved
    - Not ra because f(…) is terminal

Access to parameters and local variables using the stack frame

```
int f (int n1, n2, n3, n4, n5
           n6, n7, n8, n9, n10)
{
   int v[4];
   int k;

   k = n1+n2+n3+n4+n5+n6+n7+n8+n9+n10;
   for (k= 0; k <3; k++)
        v[i]=k;

   return (v[1]);
}
```

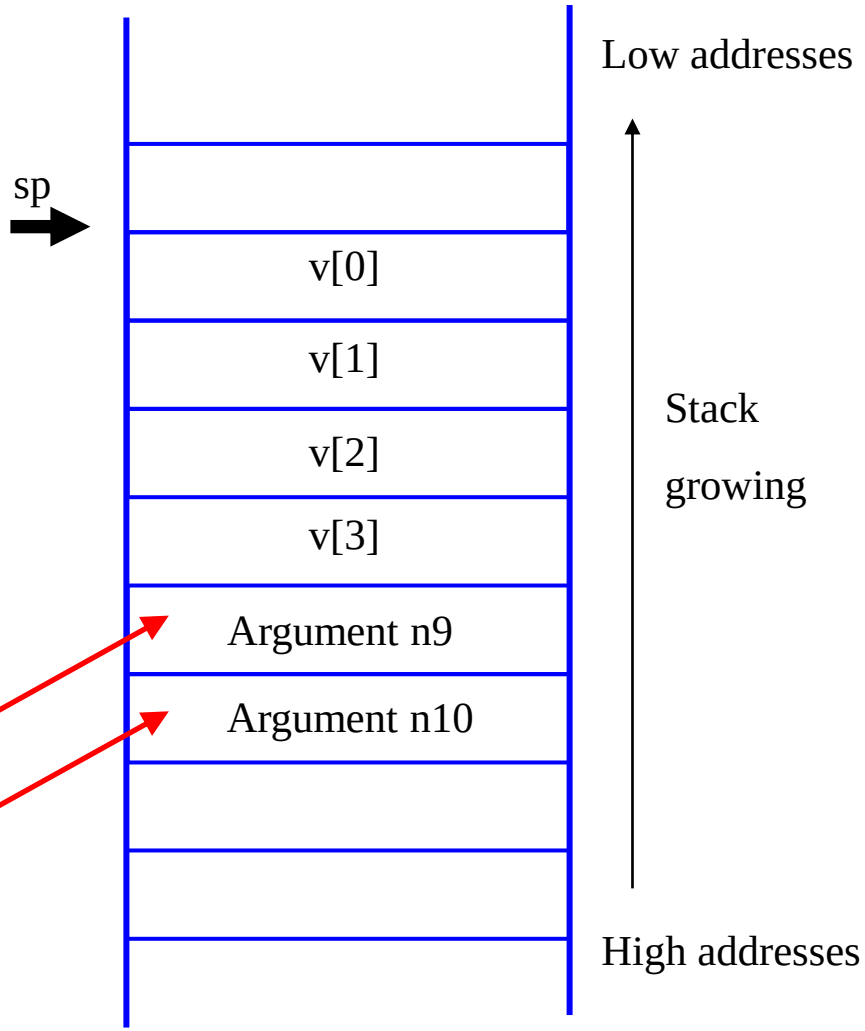


- f must reserve in the stack frame space for local variables that cannot be stored in registers (v in this example)
  - In this example f is not going to modify any register

Access to parameters and local variables using the stack frame

```
int f (int n1, n2, n3, n4, n5
           n6, n7, n8, n9, n10)
{
   int v[4];
   int k;

   k = n1+n2+n3+n4+n5+n6+n7+n8+n9+n10;
   for (k= 0; k <3; k++)
        v[i]=k;

   return (v[1]);
}
```

- The value of `n1` is in `a0`
- The value of `n9` is in `16(sp)`
- The value of `n10` is in `20(sp)`
- The value of `v[3]` is in `12(sp)`
- The value of `v[0]` is in `0(sp)`



Exercise: code to invoke to

f(`int n1, n2, n3, n4, n5, n6, n7, n8, n9, n10`)

Code for: `f(3, 4, 23, 12, 6, 7, 7, 8, 9, 10);`

Exercise (solution)

Para la llamada: `f(3, 4, 23, 12, 6, 7, 7, 8, 9, 10);`

```
li    a0, 3
li    a1, 4
...
li    a7, 8
addi sp, sp, -8
li    t0, 9
sw    t0, 0(sp)
li    t0, 10
sw    t0, 4(sp)
jal   ra, f
addi sp, sp, 8          # se libera la pila
```

First four placed on $ai registers

The rest placed on the stack

Local variables in registers

- Whenever is 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
     . . .
```

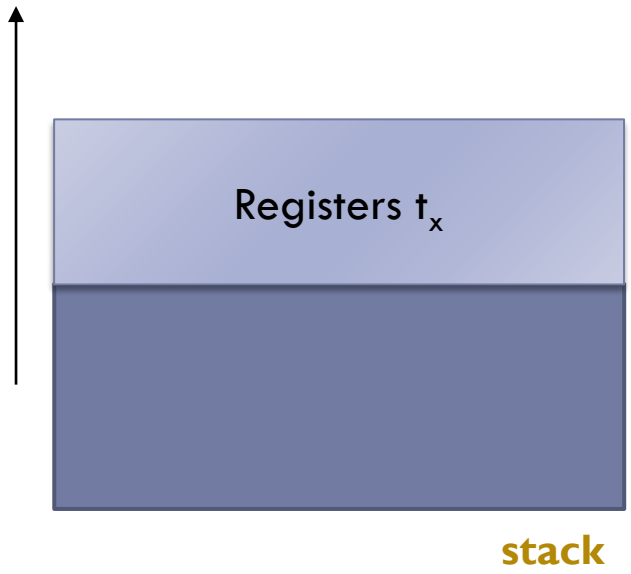
Exercise

Consider a function named `func` that receives three parameters of type integer and returns a result of type integer, and consider the following data segment fragment:

```
.data
        a: .word 5
        b: .word 7
        c: .word 9

.text
```

Indicate the code necessary to call the above function passing as parameters the values of the memory locations a, b and c. Once the function has been called, the value returned by the function must be printed.

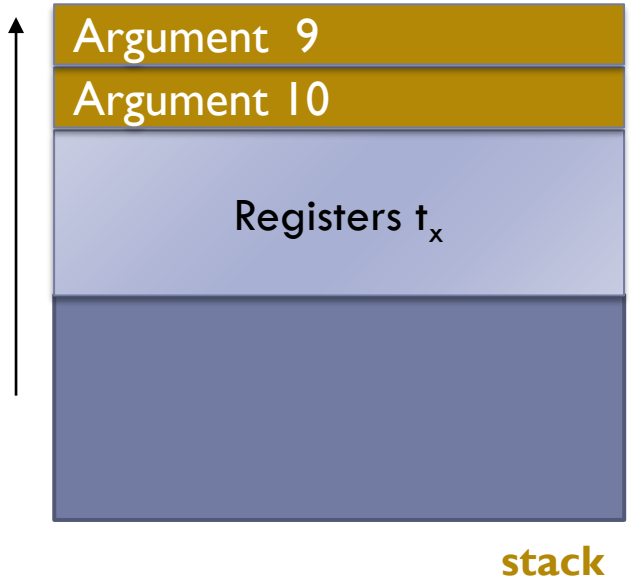
Passing 2 parameters

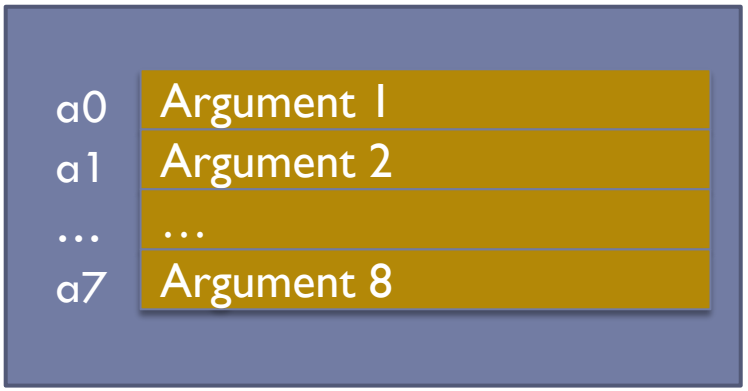


Register file


```
li a0, 5    # param 1
li a1, 8    # param 2

jal ra, func

addi sp, sp, 16
```

Passing 10 parameters






```
li a0, 5      # param 1
li a1, 8      # param 2
…
li a7, 9      # param 8

add sp, sp, -8
li t0, 10     # param 10
sw t0, 4(sp)
li t0, 7
s2 t0, 0(sp) # param 9

jal ra, func

add sp, sp, 8
```

Dynamic memory allocation in CREATOR

- The system call sbrk() in RISC-V
  - a0: number of bytes to allocate
  - a7 = 9 (system call code)
  - Return in v0 the address of the allocated memory block
  - In some cases to make free you have to use sbrk with a negative number

```
int *p;


p = malloc(20*sizeof(int));

p[0] = 1;
p[1] = 4;
```

```
# 80 bytes are allocated
li a0,  80
li a0, 9 # syscall id.
ecall

mv a0, v0
li t0, 1
sw t0, 0(a0)
li t0, 4
sw t0, 4(a0)
```

Translation and execution of programs

- Elements involved in the translation and execution of programs:
  - Compiler
  - Assembler
  - Linker
  - Loader

Translation and execution steps (C program)