ARCOS Group

uc3m | Universidad Carlos III de Madrid

Lesson 3 (III)

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:
    move $t0 $a0
    li $v0 1
b1: beq $t0 $zero f1
    mul $v0 $v0 $t0
    addi $t0 $t0 -1
    b b1
f1: jr $ra
…
li $a0 3
jal 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);
}
```

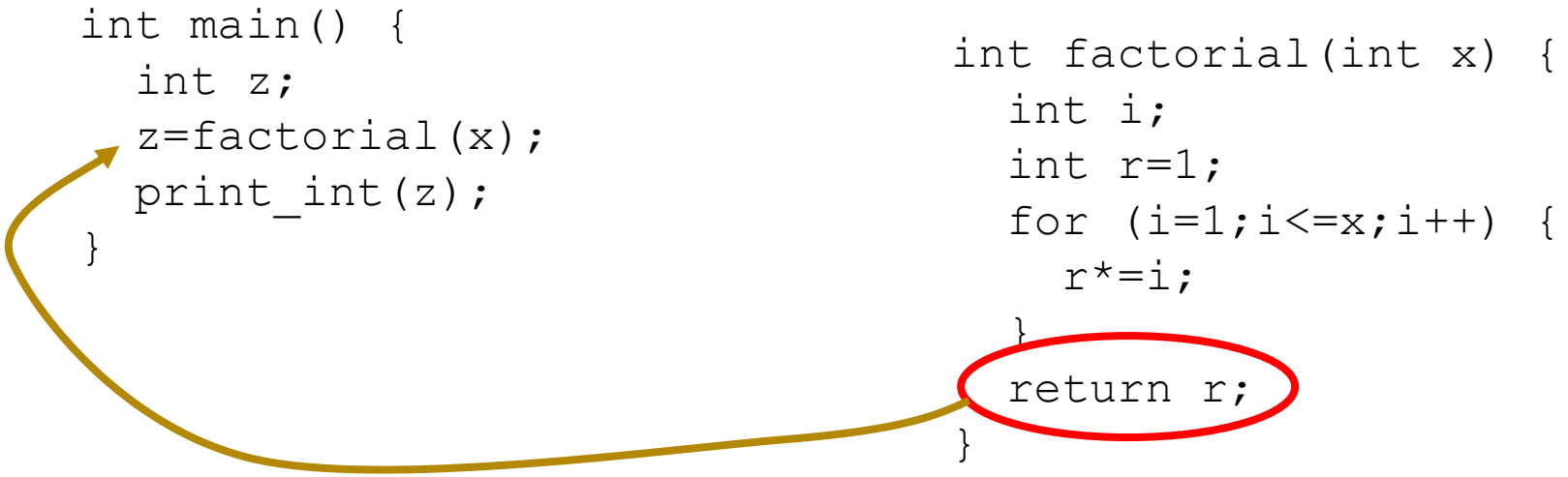
```
int factorial(int x) {
  int i;
  int r=1;
  for (i=1;i<=x;i++) {
    r*=i;
  }
  return r;
}
```

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

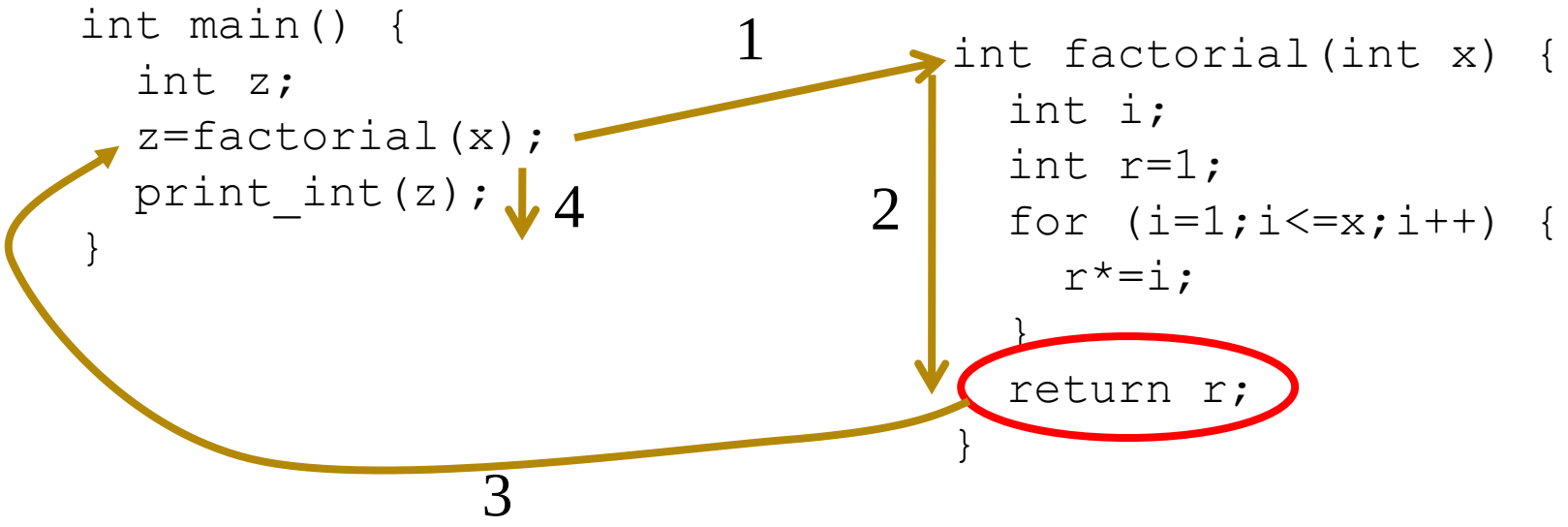


Functions in high level languages

```
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;
}
```

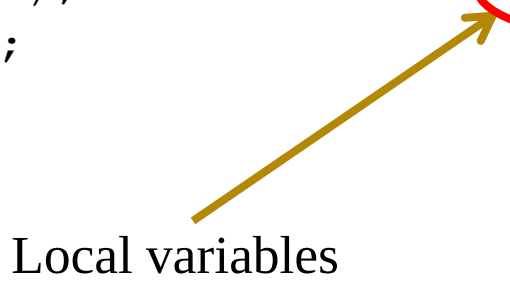
Functions in high level languages



Functions in high level languages

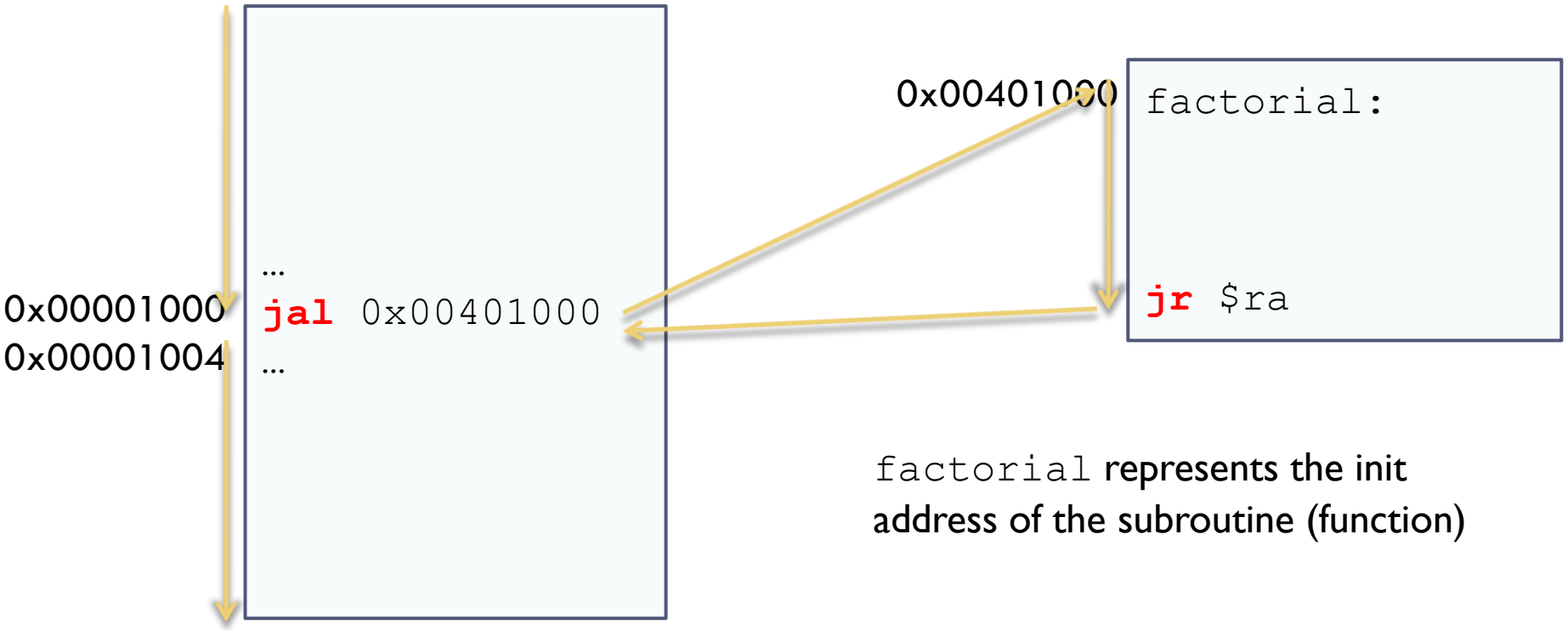
```
int main() {
  int z;
  z=factorial(x);
  print_int(z);
}
```

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



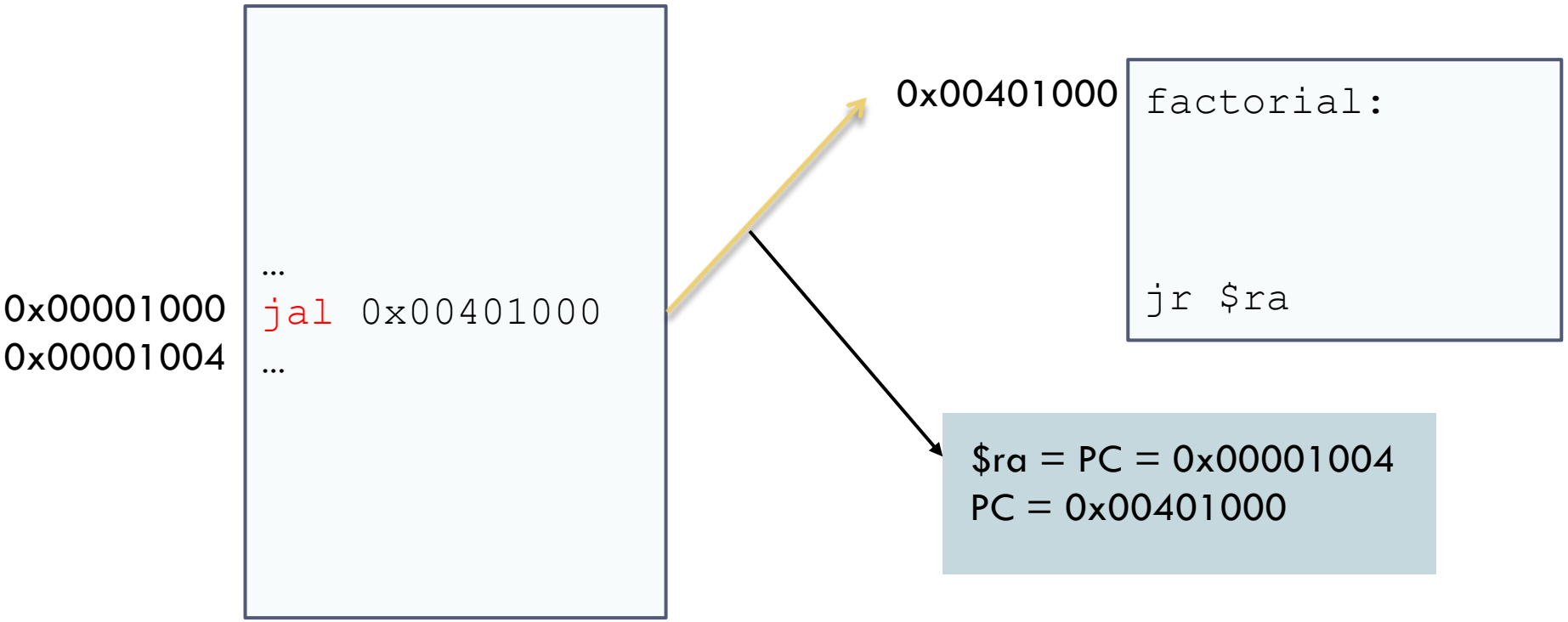
Function calls in MIPS

Function calls in MIPS (`jal` instruction)

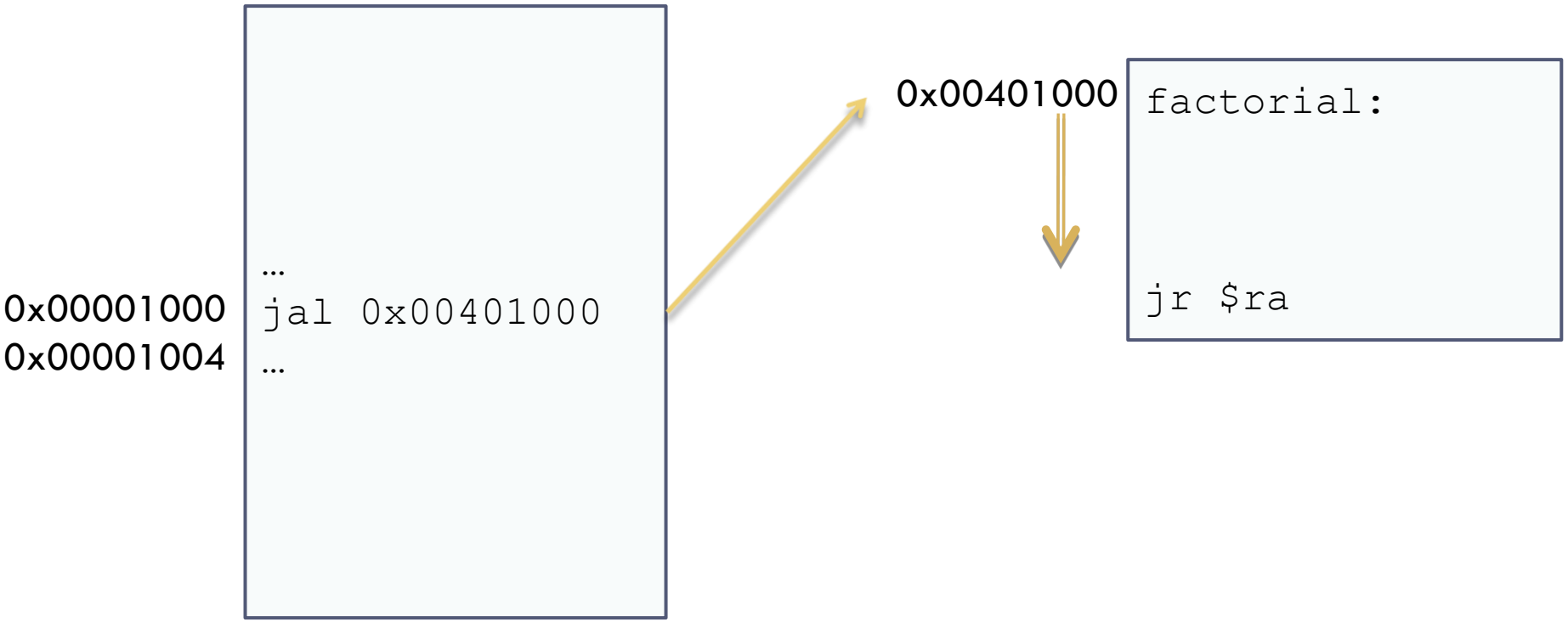


`factorial` represents the init address of the subroutine (function)

Function calls in MIPS

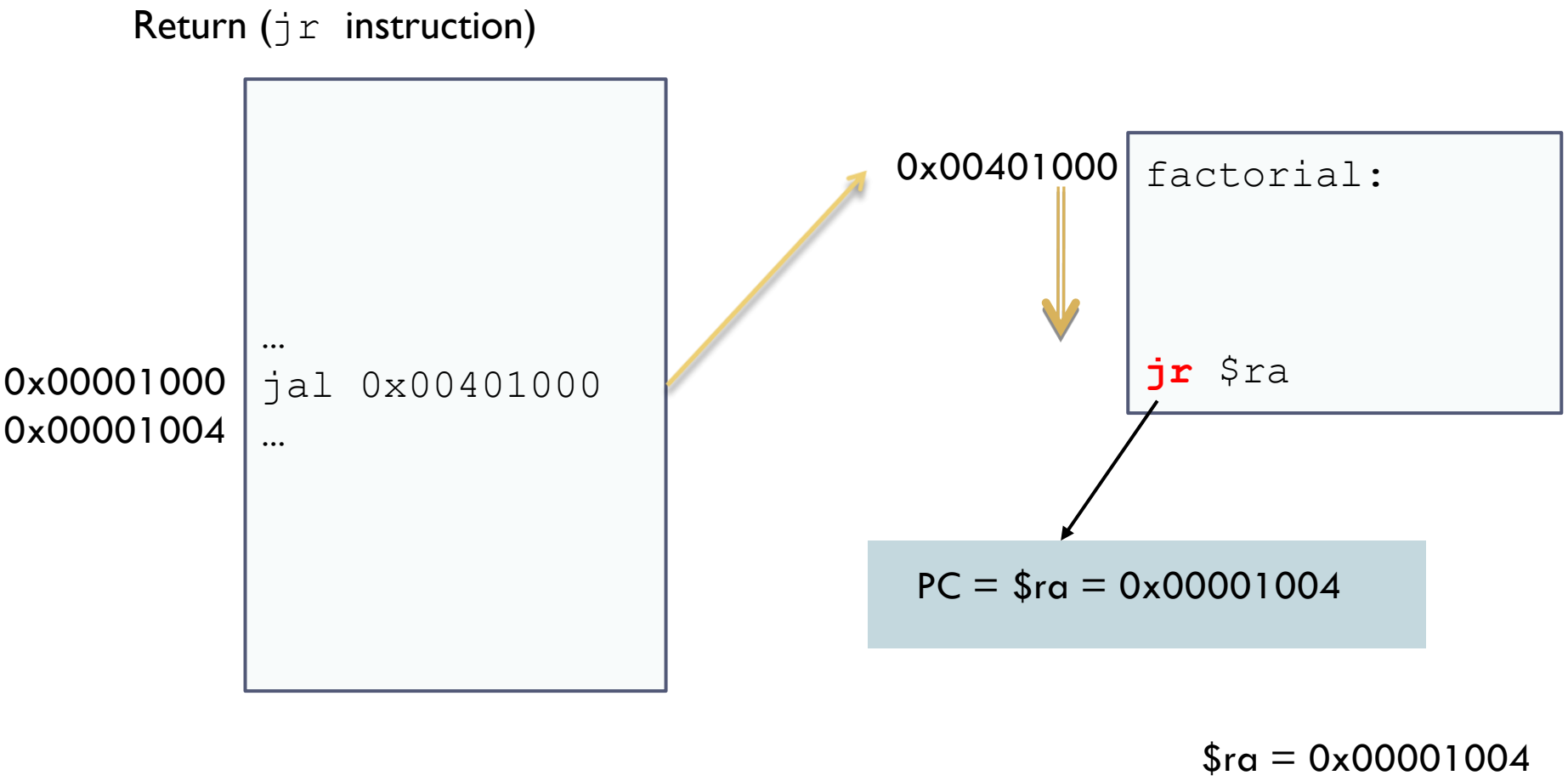


Function calls in MIPS

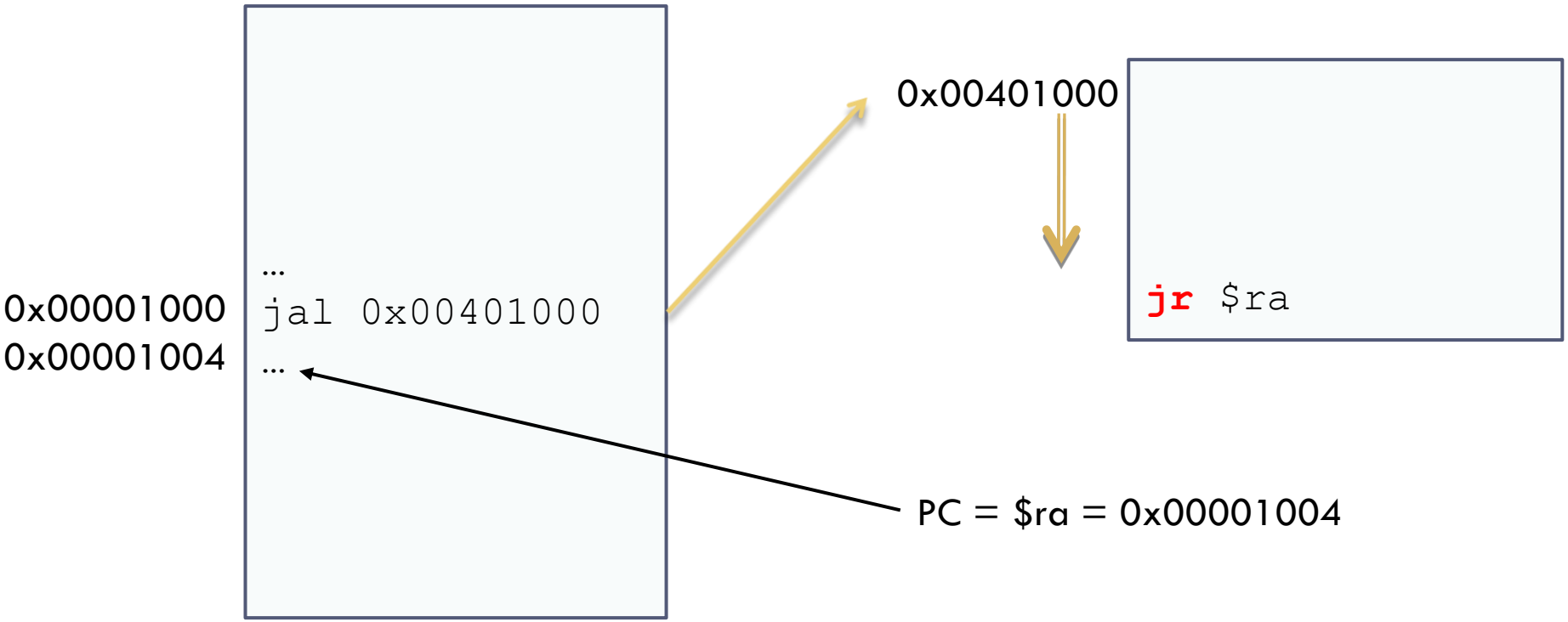


$ra = 0x00001004

Function calls in MIPS



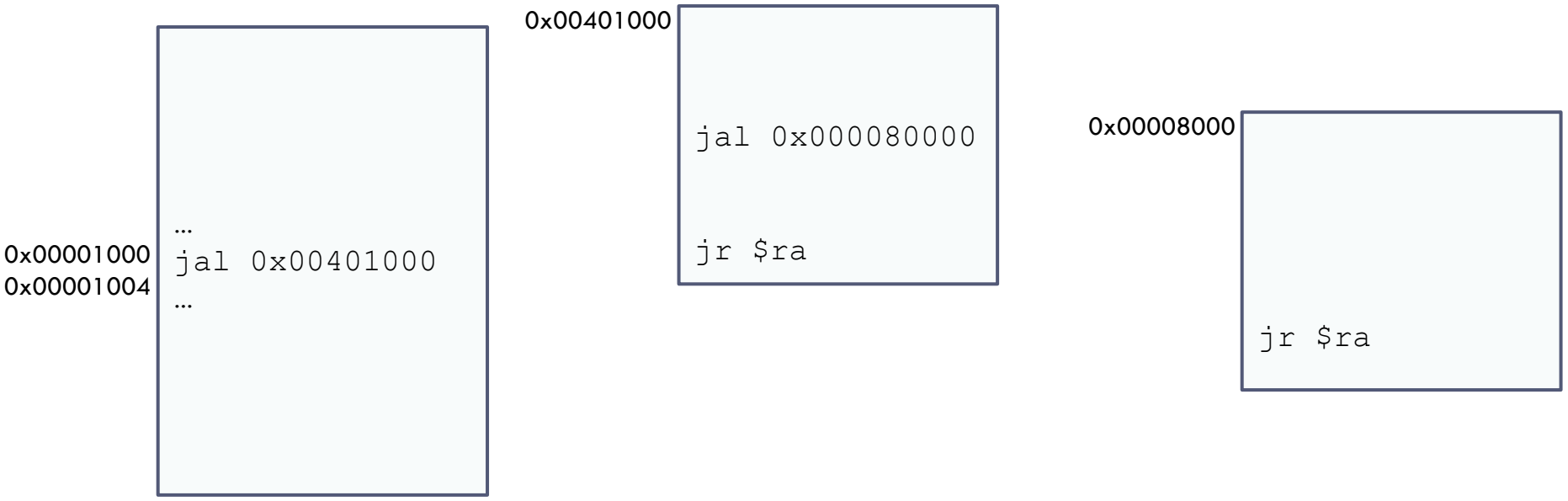
Function calls in MIPS



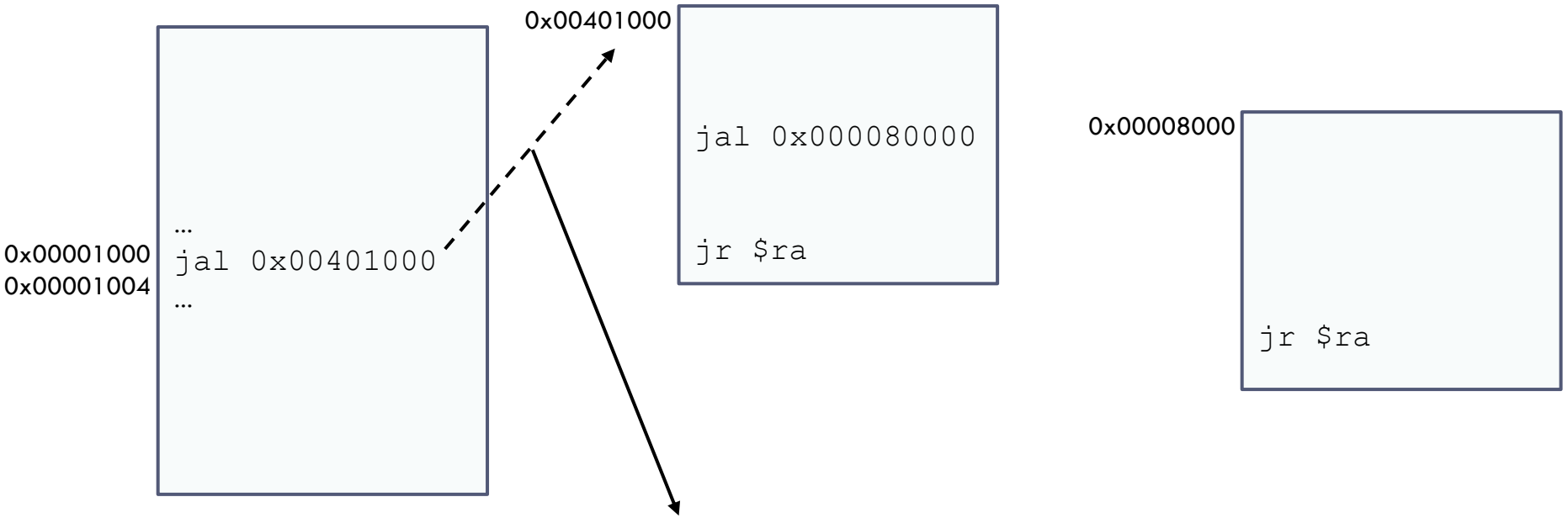
`jal/jr` instructions

- 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

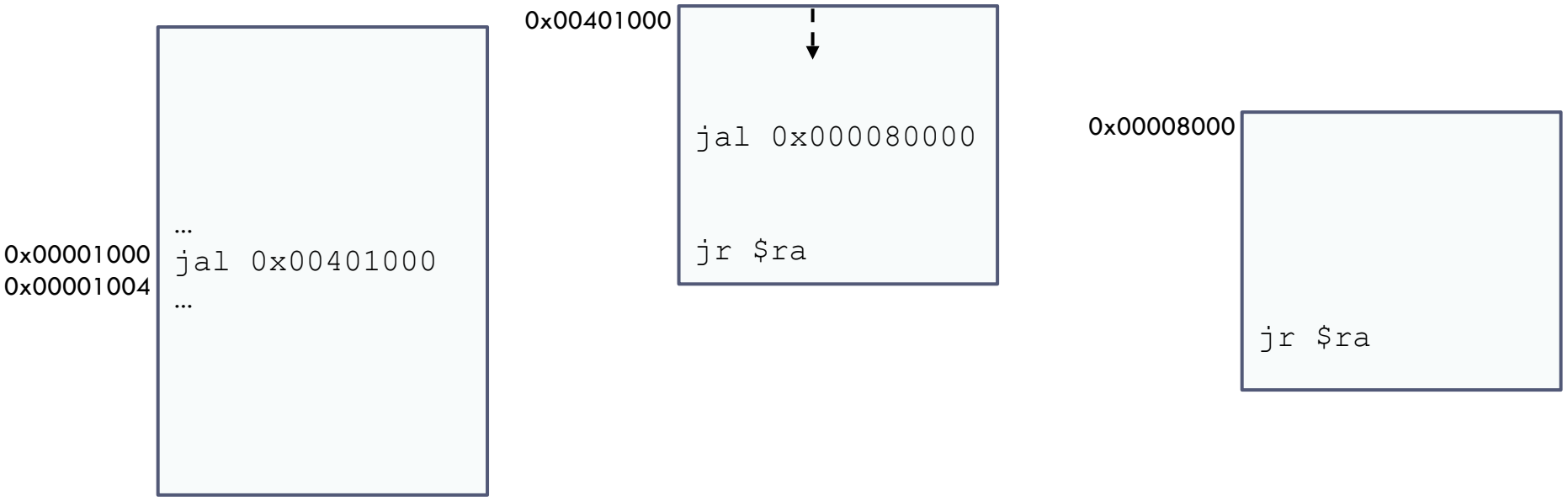


Nested calls



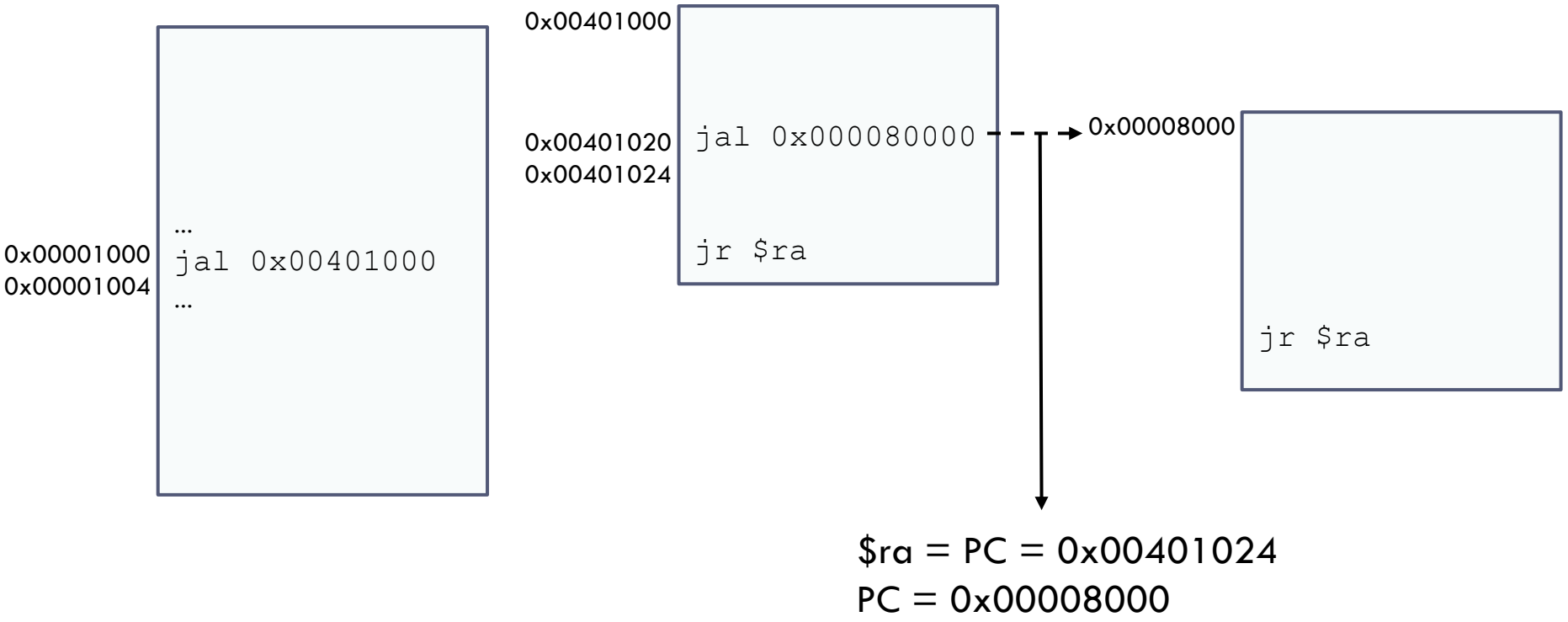
Return address $ra = PC = 0x00001004

Nested calls



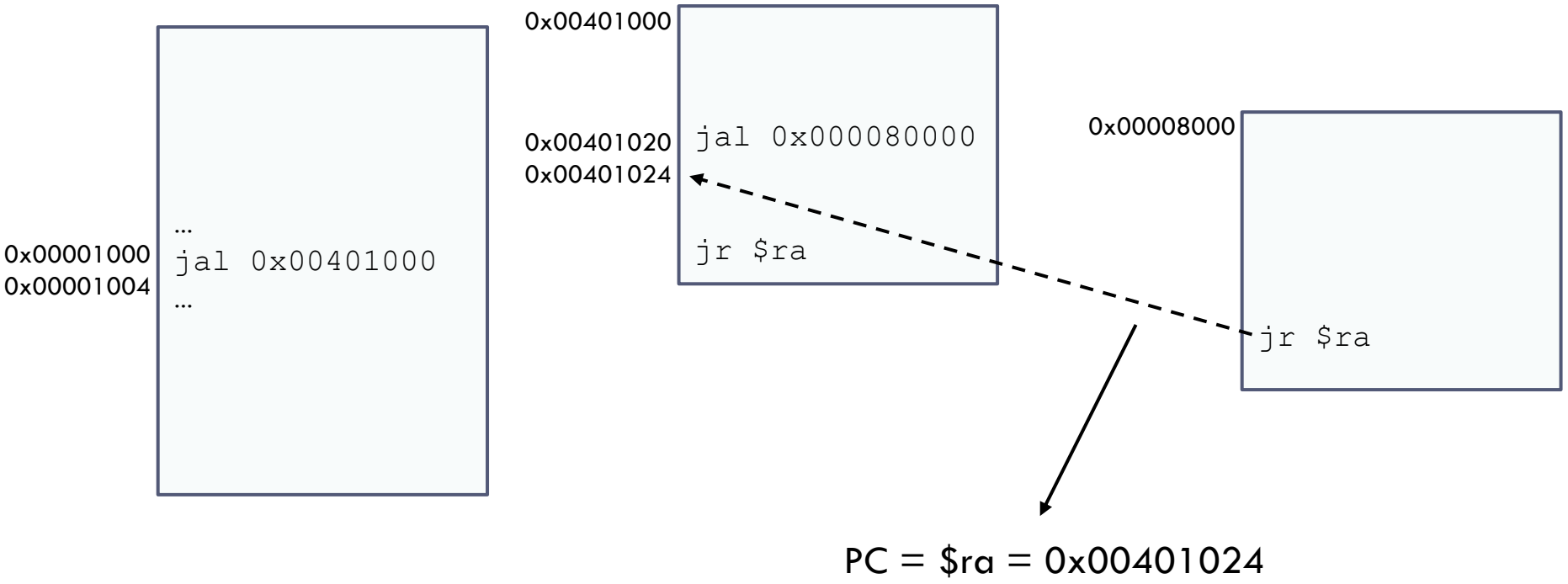
Return address $ra = PC = 0x00001004

Nested calls



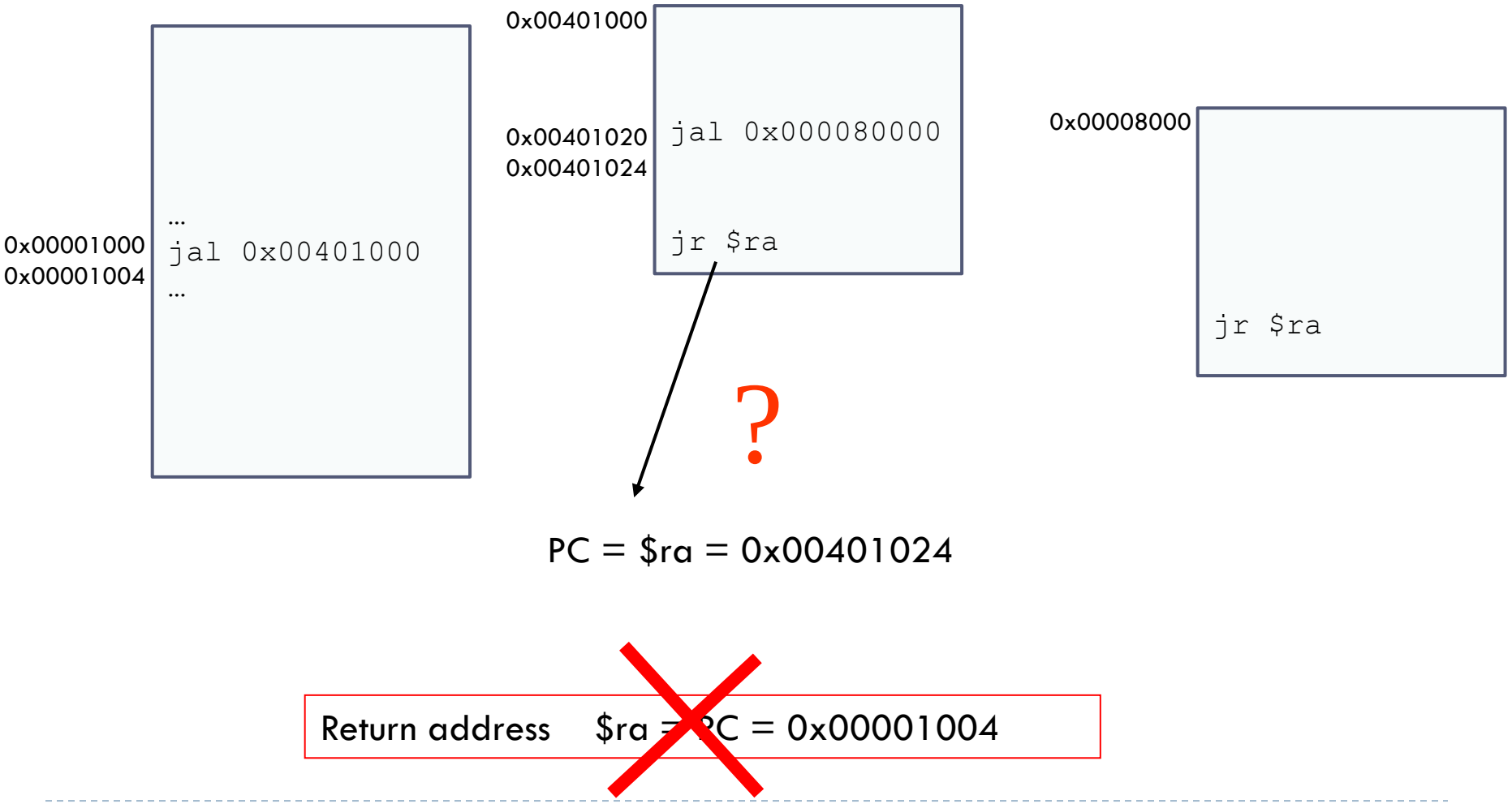
Return address $ra = PC = 0x00001004

Nested calls

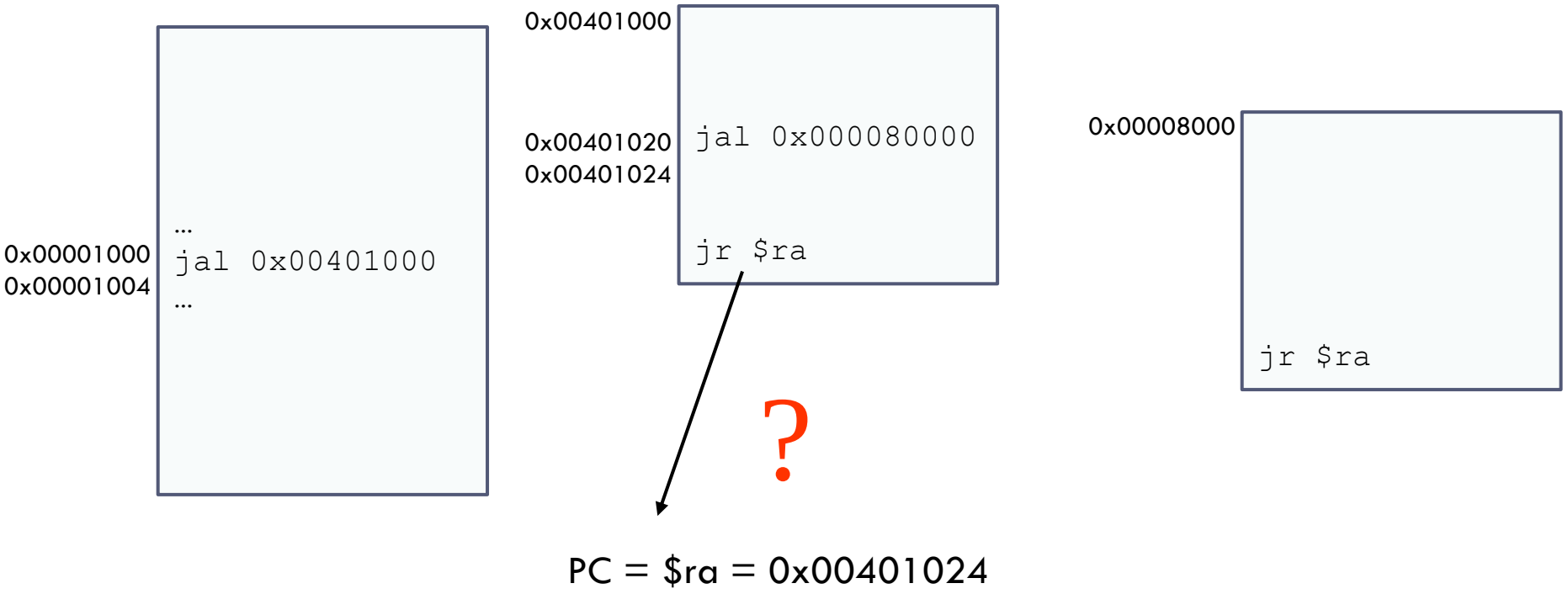


Return address $ra = PC = 0x00001004

Nested calls



Nested calls



PC = $ra = 0x00401024

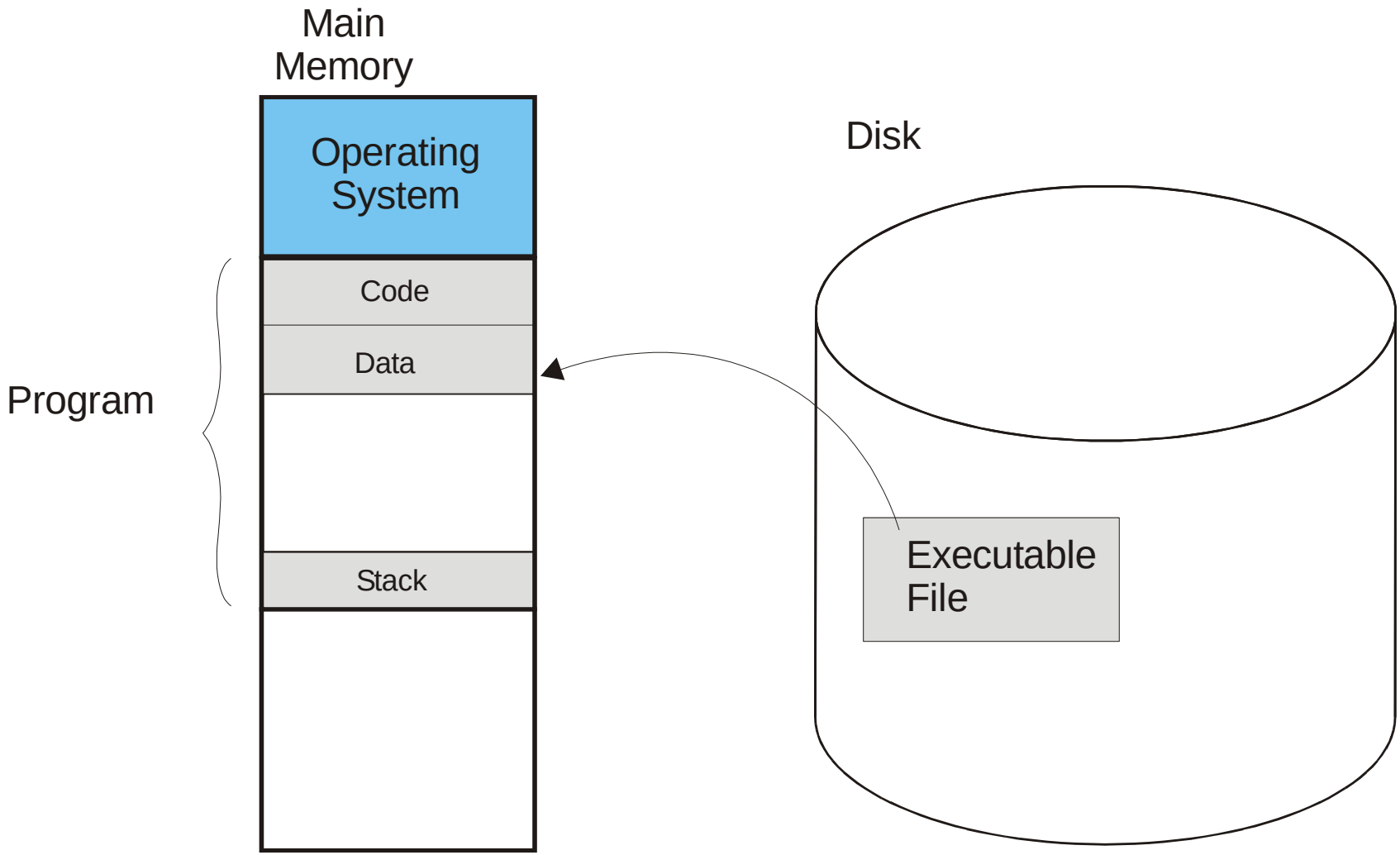
The return address is lost

Where to store the return address?

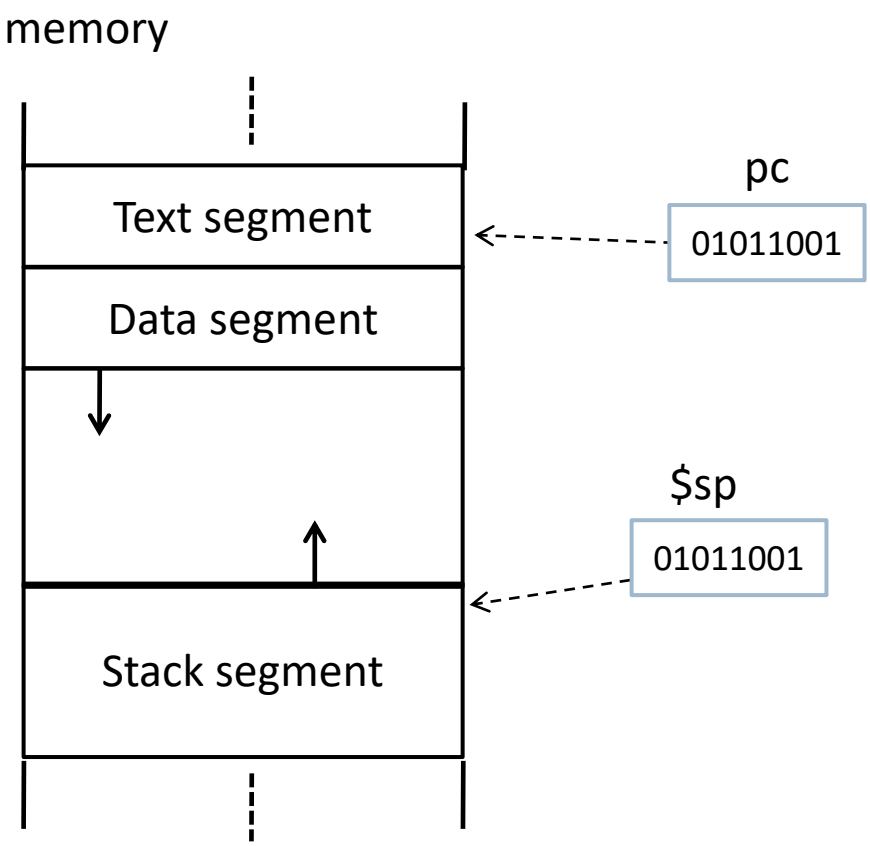
- Computers have two storage elements:
  - Registers
  - Memory

- 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



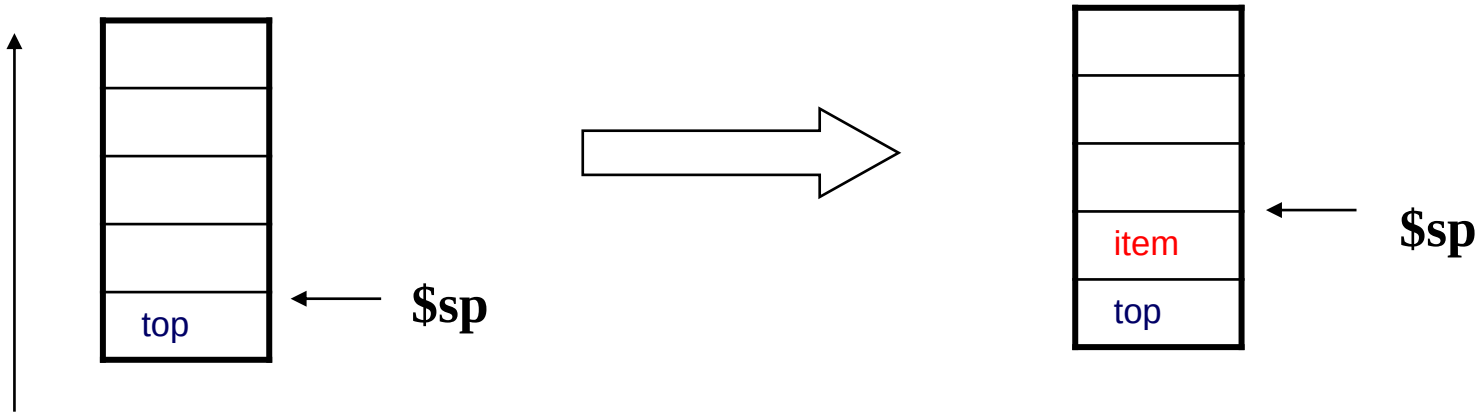
Memory map of a process



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

Stack

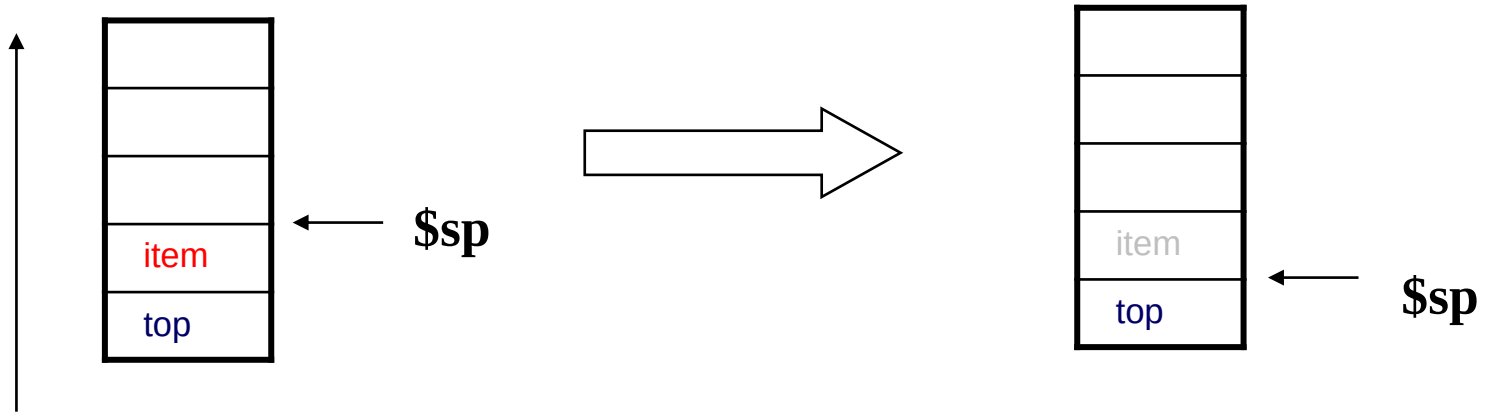
PUSH Reg Push an element in stack (item)



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

- MIPS 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

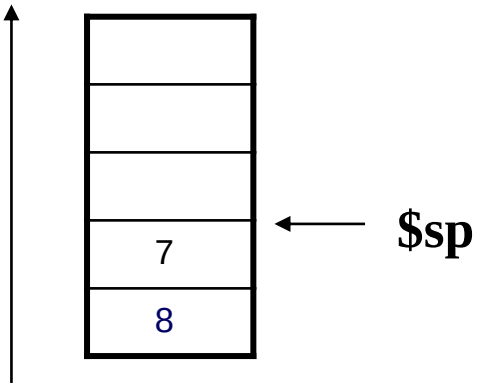
```
subu $sp, $sp, 4
sw   $t0, ($sp)
```

POP $t0

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

PUSH operation in MIPS

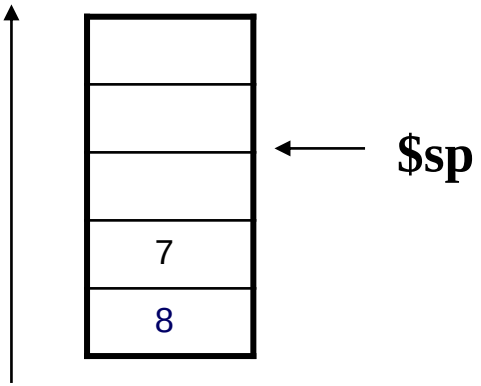
```
...
li   $t2,   9
subu $sp, $sp, 4
sw   $t2 ($sp)
...
```



- Initial state: stack pointer ($sp) points to the last element in the stack

PUSH operation in MIPS

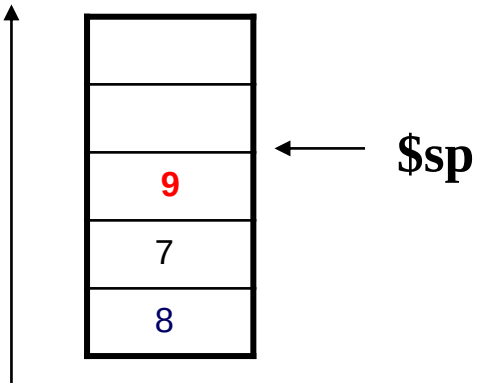
```
…
li   $t2,   9
subu $sp, $sp, 4
sw   $t2 ($sp)
…
```



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

PUSH operation in MIPS

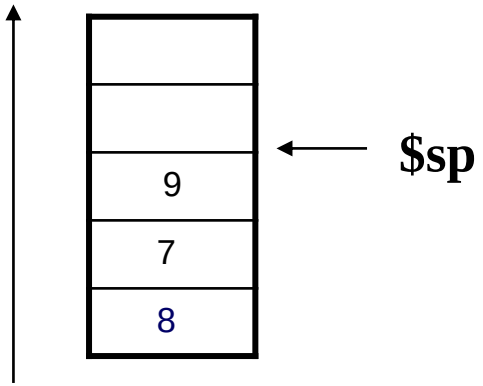
```
...
li   $t2,   9
subu $sp, $sp, 4
sw   $t2 ($sp)
...
```



- Insert the content of register $t2 in the stack

POP operation in MIPS

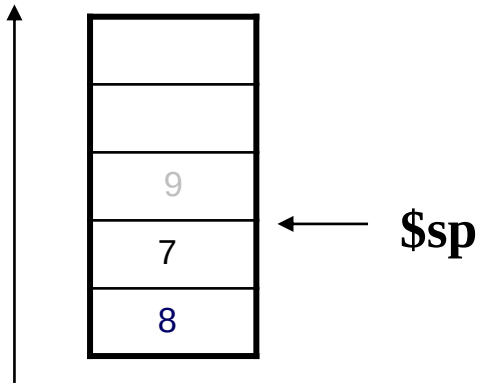
```
...
lw  $t2 ($sp)
addu $sp, $sp, 4
...
```



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

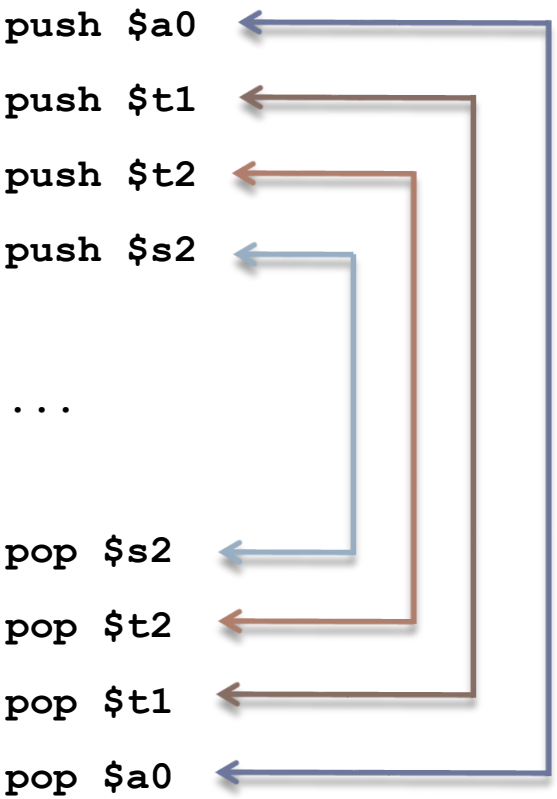
POP operation in MIPS

```
...
lw  $t2 ($sp)
addu $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.

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
```

```
addu $sp $sp -16
sw   $a0  ($sp)
sw   $t1  ($sp)
sw   $t2  ($sp)
sw   $s2  ($sp)
```

...

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

- Use of a convention (to be discussed in more detail) in MIPS:
  - Arguments/parameters are placed in $a0, $a1, $a2 and $a3.
    - If more than 4 parameters need to be carried, the first four in registers $a0, $a1, $a2 and $a3 and the rest on the stack.
  - The results are collected in $v0, $v1

- In the invocation of factorial: `z=factorial(5);`
  - One input parameter/argument: $a0
  - One result: $v0

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     # argument
          jal factorial  # invoke
          move $a0, $v0  # result
          # print_int(z)
          li $v0, 1
          syscall
          ...

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
           b    loop
      end: move $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
           b    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: sub  $sp, $sp, 8
           sw   $s0, 4($sp)
           sw   $s1, ($sp)
           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
           b    bucle
end:       move $v0, $s1  #result
           lw   $s1, ($sp)
           lw   $s0, 4($sp)
           add  $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     # first argument
li   $a1,  2     # second argument
jal  f1          # call
move $a0, $v0    # result
li   $v0, 1
syscall           # systam call
                  # to print an 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

    move  $a0, $a1
    jal   f2
    add   $v0, $s0, $v0

    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

    move  $a0, $a1
    jal   f2
    add   $v0, $s0, $v0

    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: addu  $sp, $sp, -8
    sw    $s0, 4($sp)
    sw    $ra,  ($sp)

    add   $s0, $a0, $a0

    move  $a0, $a1
    jal   f2
    add   $v0, $s0, $v0

    lw    $ra,  ($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, $a1, $a2 y $a3
  - If more than 4 parameters are need to be passed, the first four in registers $a0, $a1, $a2 and $a3 and the rest on the stack
  - Integer includes high-level datatypes such as char, int, etc.
- The float arguments are placed in $f12, $f13, $f14 y $f15
  - If more than 4 parameters need to be passed, the remainder in the stack
- The double arguments are placed in $f12-$f13 y $f14-$f15
  - If more than **2** parameters need to be passed, the remainder in the stack

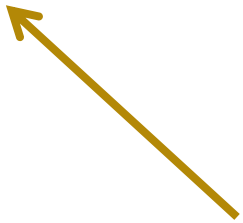
Simplified calling convention: return of results in MIPS

- Use $v0 and $v1 for integer type values
- Use $f0 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 MIPS

```
li      $t0, 8
li      $s0, 9

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



What are the values of the registers $t0 and $s0?

Simplified calling convention: registers in MIPS

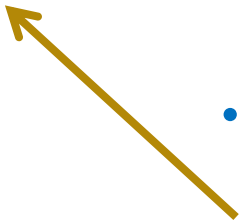
Register	Use	Preserving value
$v0-$v1	Results	No
$a0..$a3	Arguments	No
$t0..$t9	Temporary	No
$s0..$s7	Temporary to preserv	Yes
$sp	Stack pointer	Yes
$fp	Stack frame pointer	Yes
$ra	Return address	Yes

Simplified calling convention: registers in MIPS (floating point)

Register	Use	Preserving value
$f0-$f3	Results	No
$f4..$f11	Temporary	No
$f12-$f15	Arguments	No
$f16-$f19	Temporary	No
$f20-$f31	Temporary to preserv	Yes

Calling convention: registers in MIPS

```
li      $t0, 8
li      $s0, 9

li      $a0, 7    # argument
jal     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

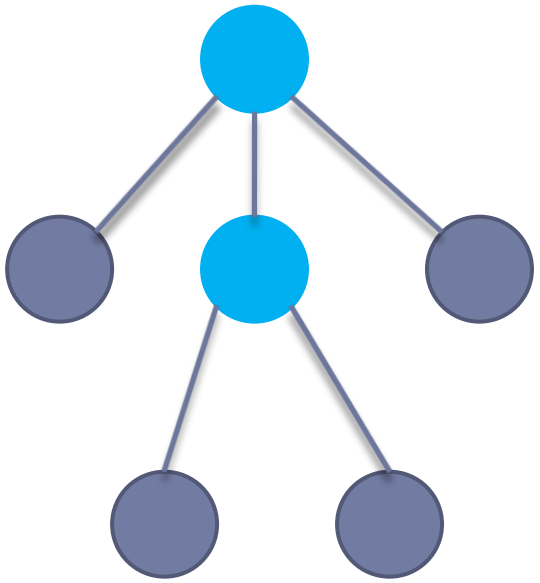
```
li      $t0, 8
li      $s0, 9

addu    $sp, $sp, -4
sw      $t0, ($sp)

li      $a0, 7      # argument
jal     función

lw      $t0, ($sp)
addu    $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

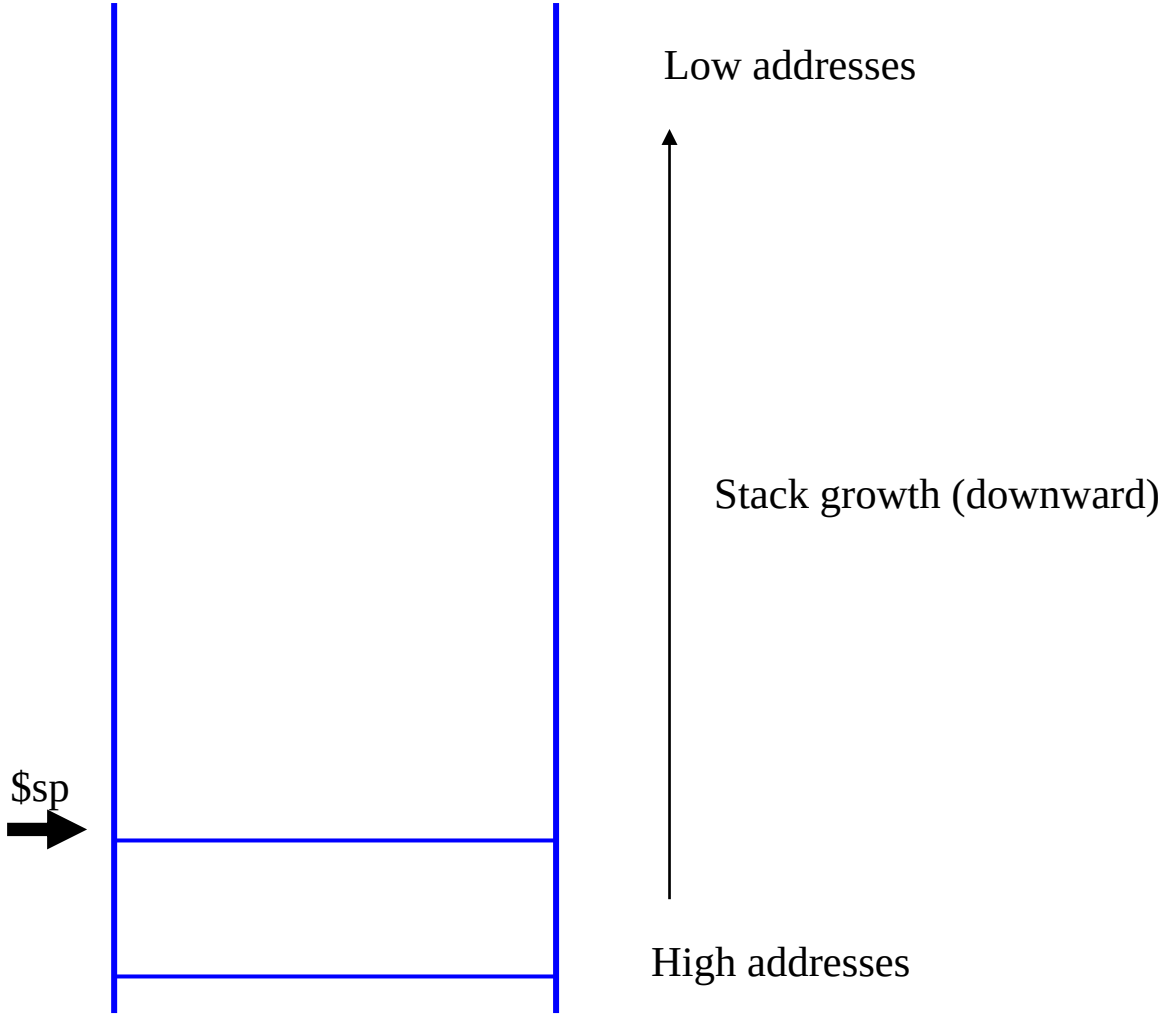
- The stack frame or activation register is the mechanism used by the compiler to activate functions in high-level languages.
- The stack frame is built on the stack by the calling procedure and the called procedure.
- The stack frame stores:
  - Parameters passed by the caller function
  - The stack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

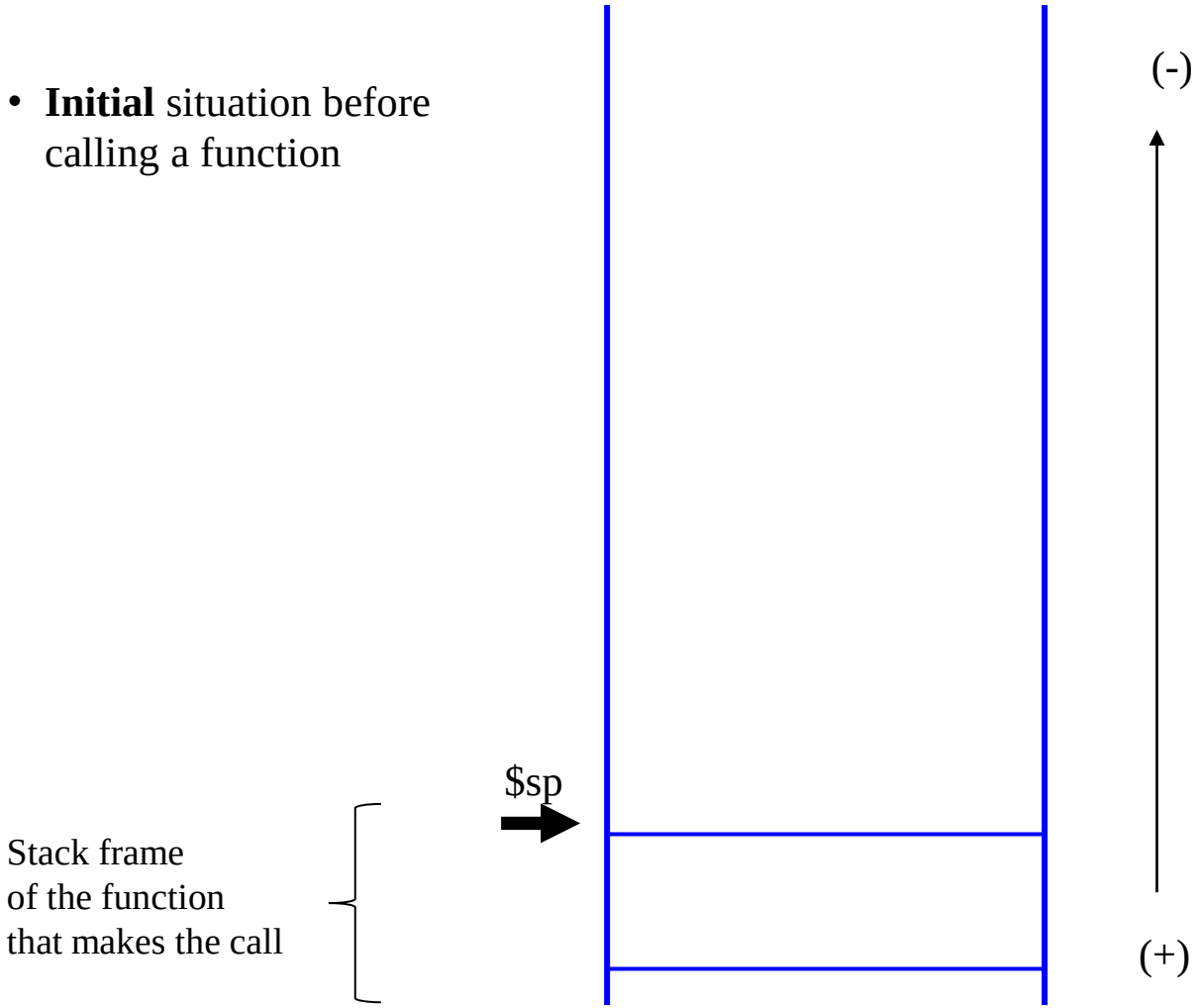
Construction of the stack frame
caller function

The MIPS convention will not be strictly followed for the sake of simplicity



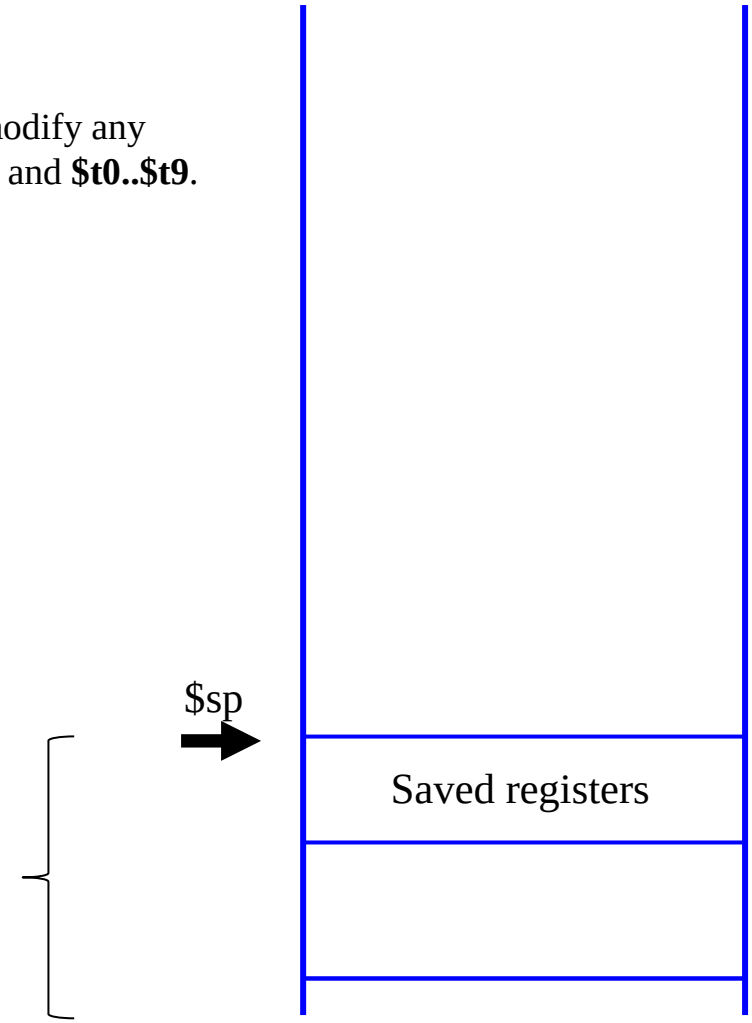
Construction of the stack frame
caller function

- **Initial** situation before calling a function



Construction of the stack frame
caller function

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



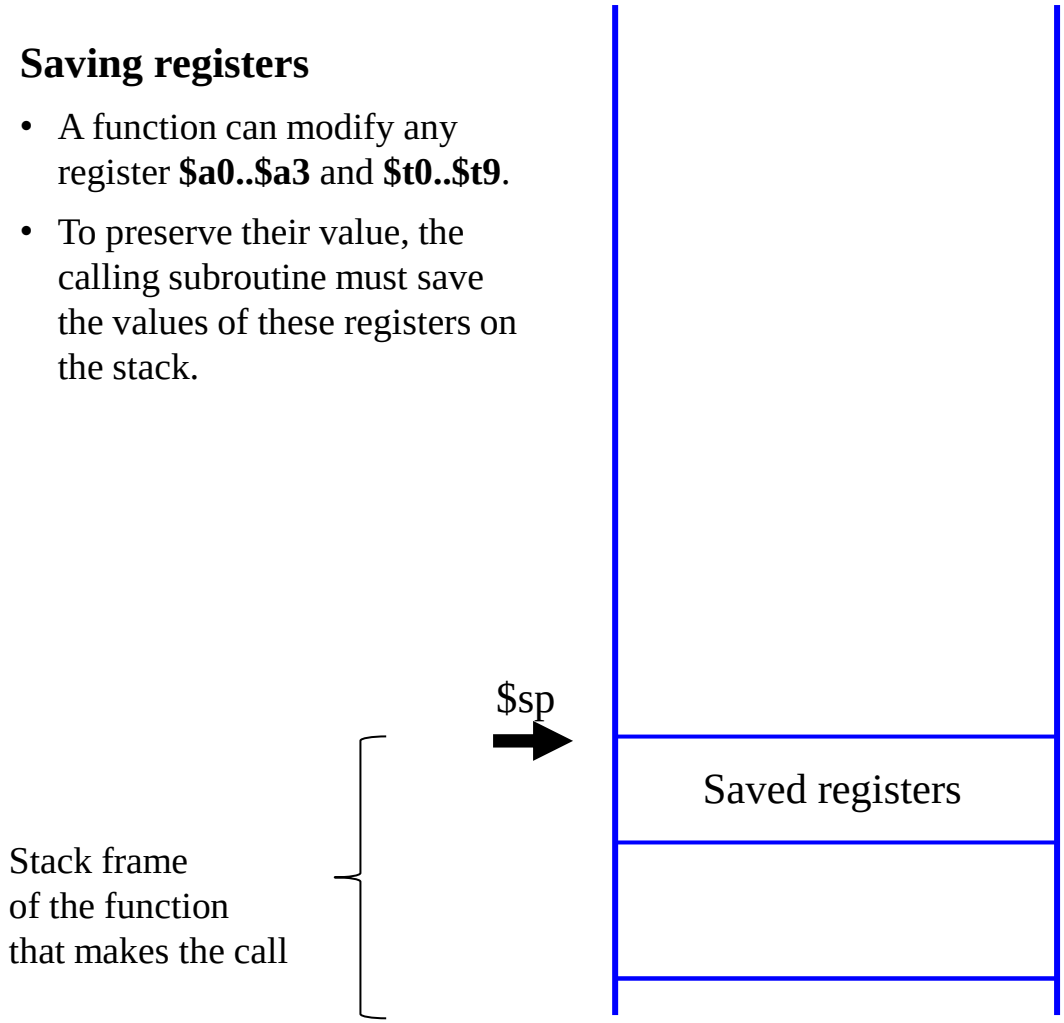
Example:

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

move $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.



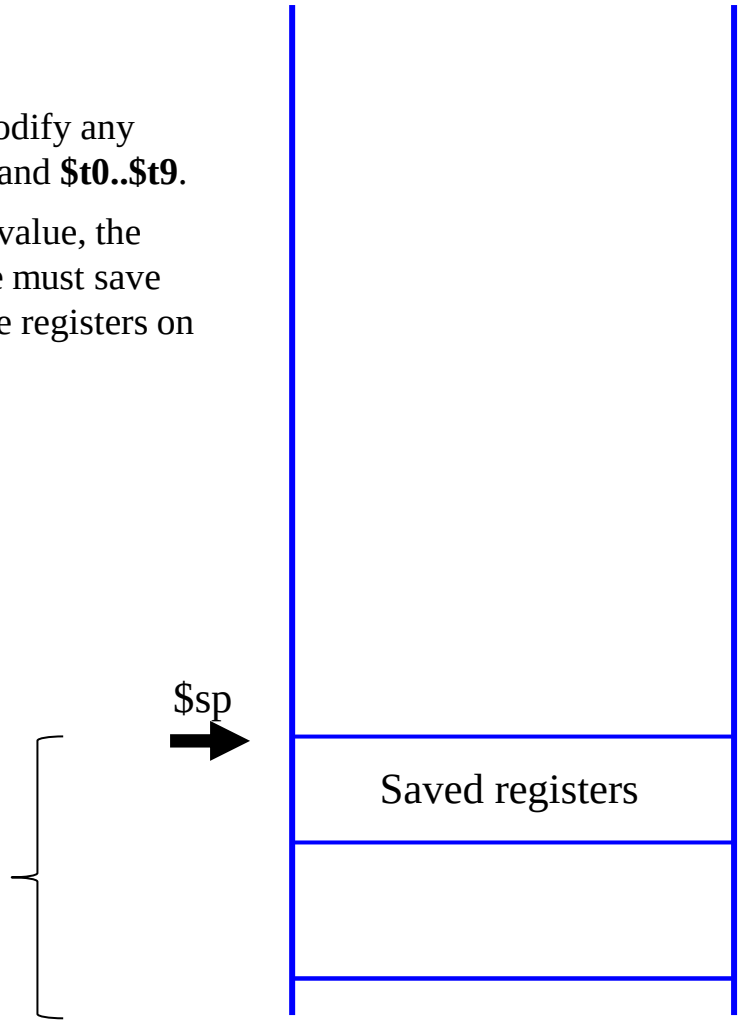
Example:

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

move $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.

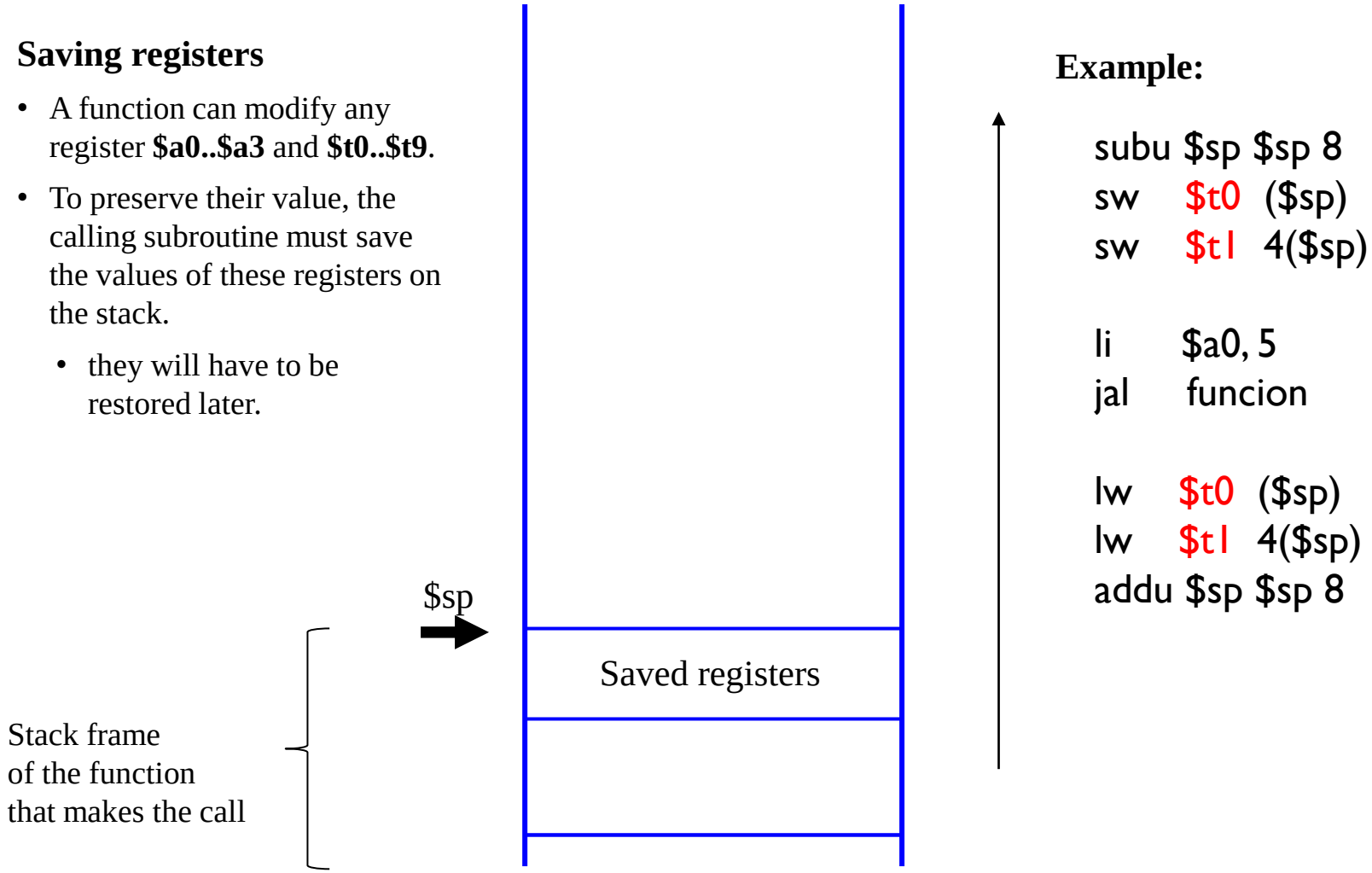


Example:

```
subu $sp $sp 8
sw   $t0  ($sp)
sw   $t1  4($sp)

li   $a0, 5
jal  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.

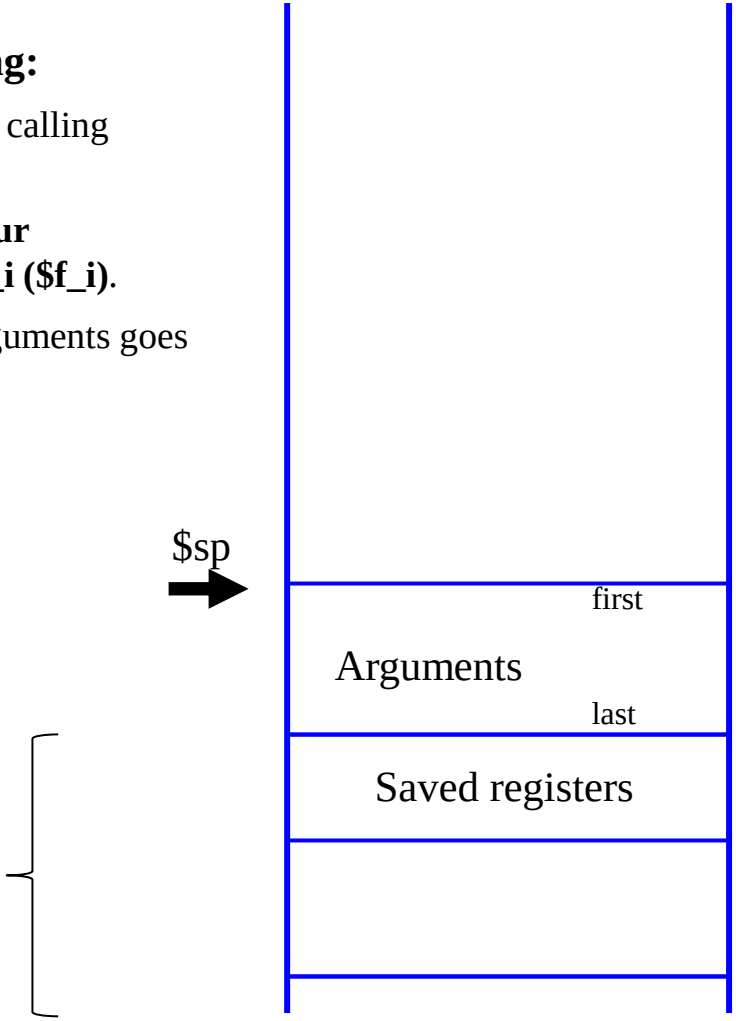


Example:

```
subu $sp $sp 8
sw   $t0  ($sp)
sw   $t1  4($sp)

li   $a0, 5
jal  funcion

lw   $t0  ($sp)
lw   $t1  4($sp)
addu $sp $sp 8
```

Construction of the stack frame
caller function

- **Argument passing:**
  - Before calling the calling procedure.
  - Leave the **first four arguments** in **$a_i ($f_i)**.
  - The rest of the arguments goes to the stack

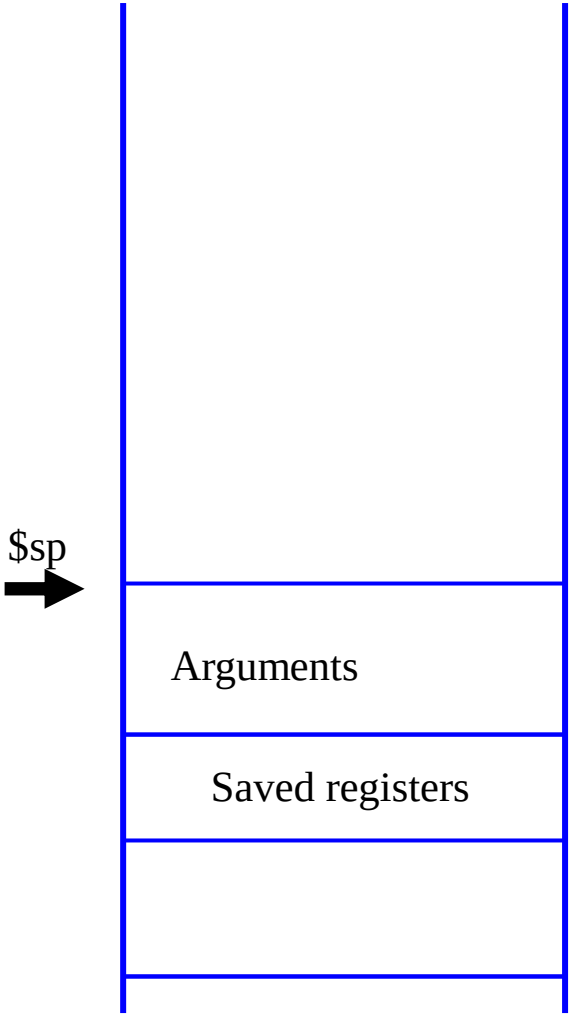


Example (6 arguments):

```
li    $a0, 1
li    $a1, 2
li    $a3, 3
li    $a4, 4

subu $sp, $sp, 8

li    $t0, 5
sw    $t0, 4($sp)

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

Construction of the stack frame
caller function

- **Function invocation:**
  - jal function



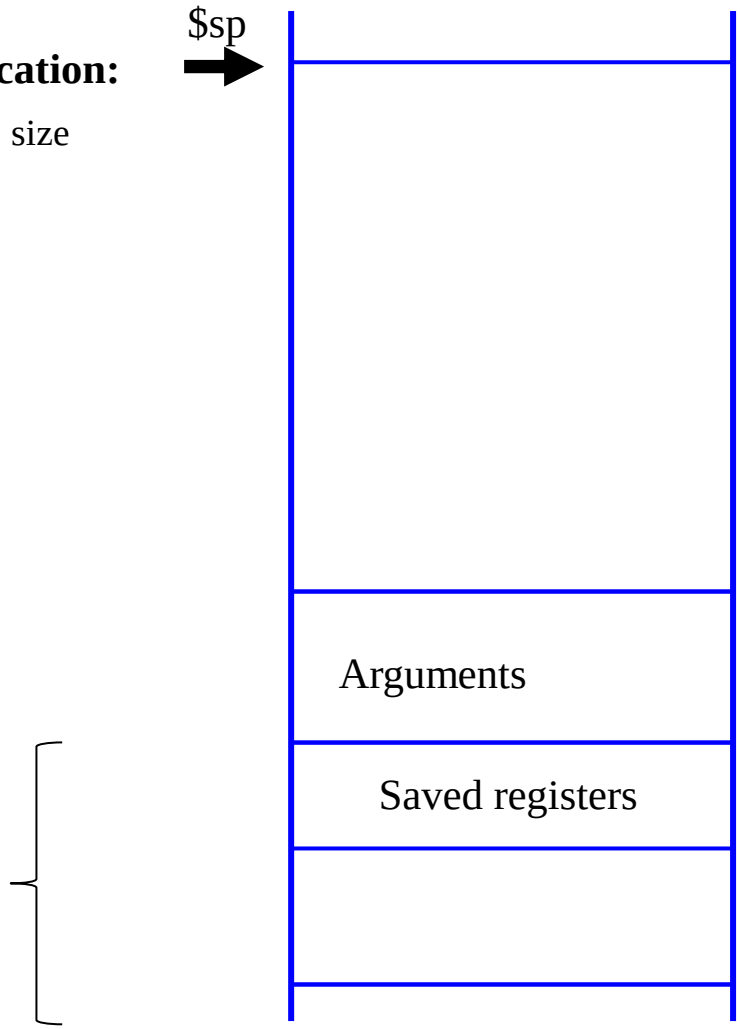
Stack frame
of the function
that makes the call

Example (6 arguments):

```
li      $a0, 1
li      $a1, 2
li      $a3, 3
li      $a4, 4

subu $sp, $sp, 8

li      $t0, 5
sw      $t0, 4($sp)

li      $t0, 6
sw      $t0, ($sp)

jal function
```

Construction of the stack frame
called function

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

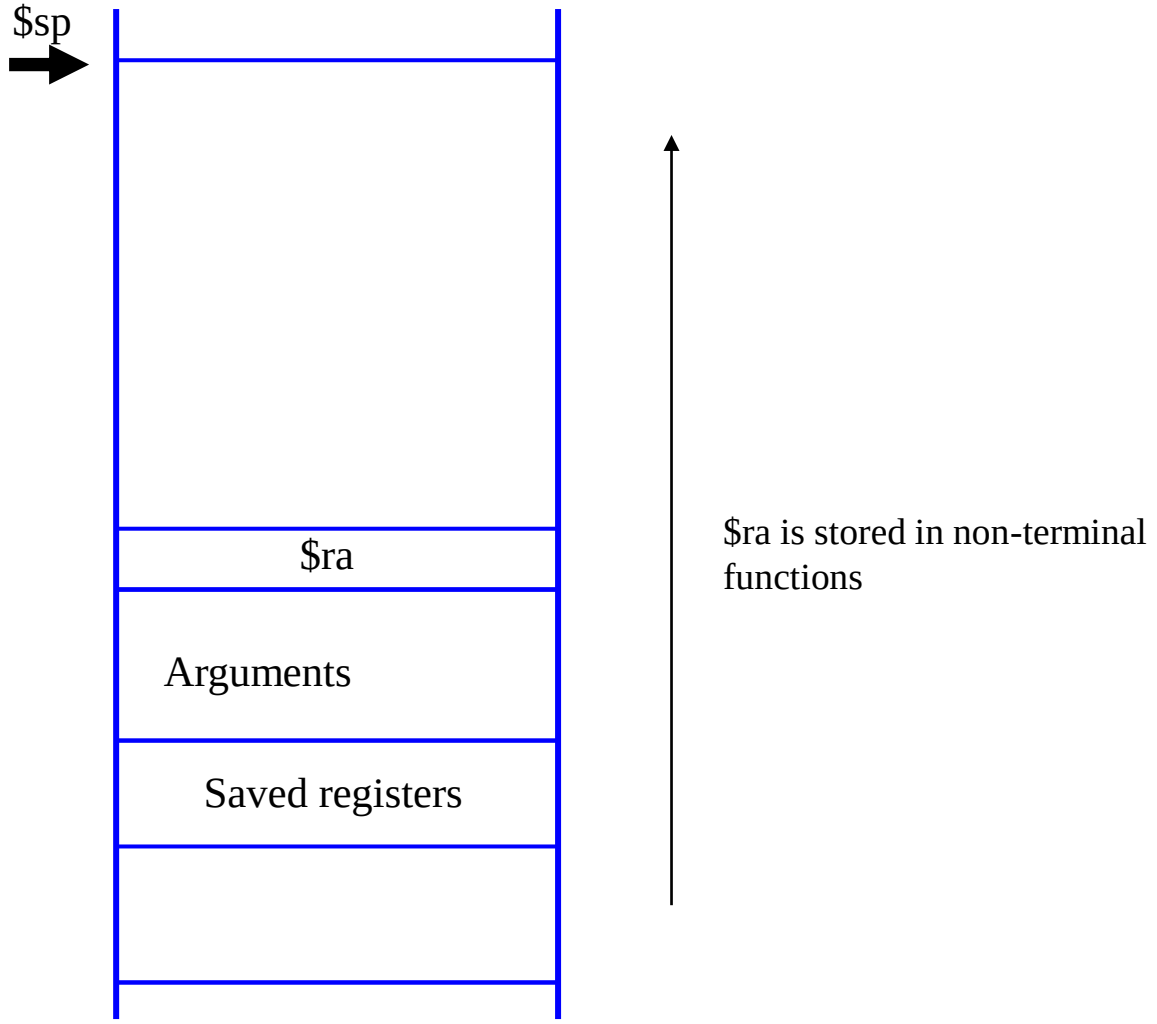


Example:

subu $sp $sp <fr.sz.>

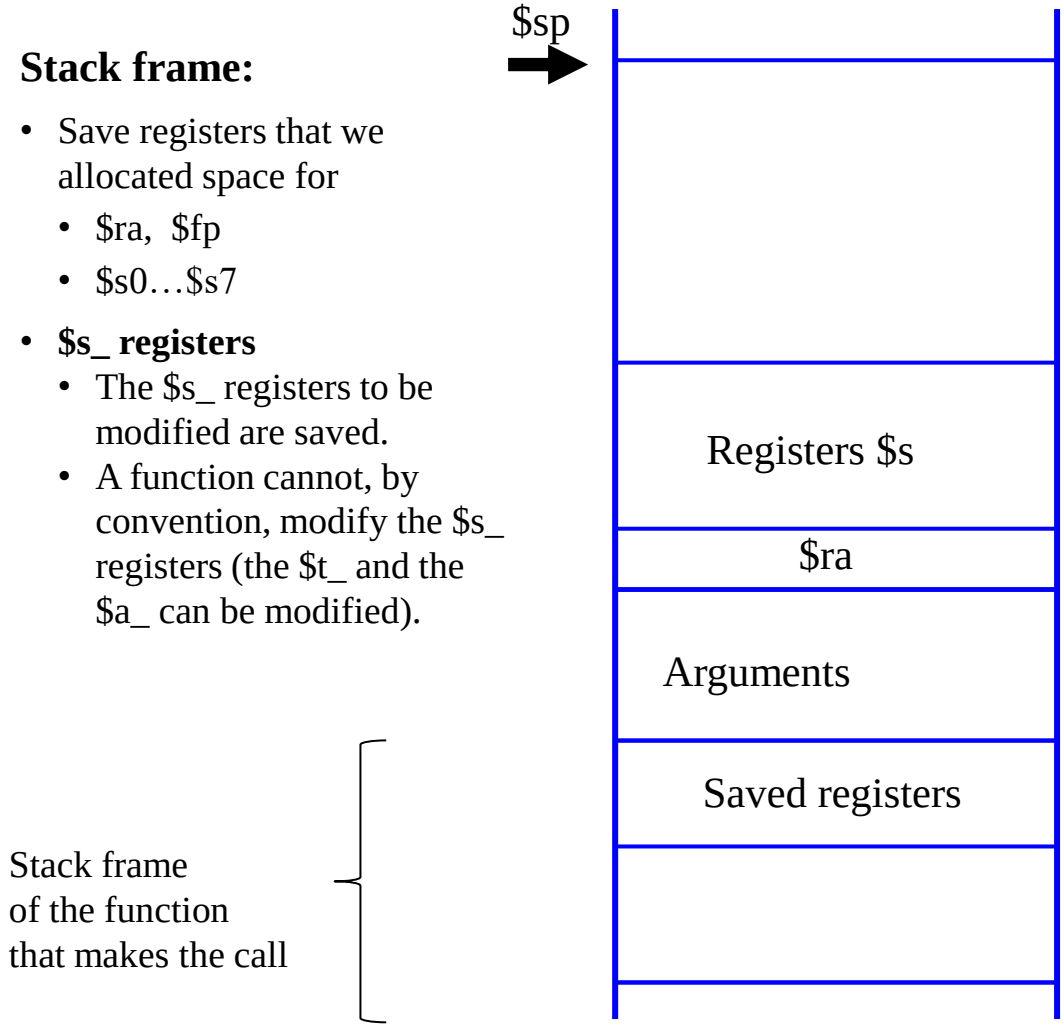
Construction of the stack frame
called function

- **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_ registers**
    - The $s_ registers to be modified are saved.
    - A function cannot, by convention, modify the $s_ registers (the $t_ and the $a_ can be modified).

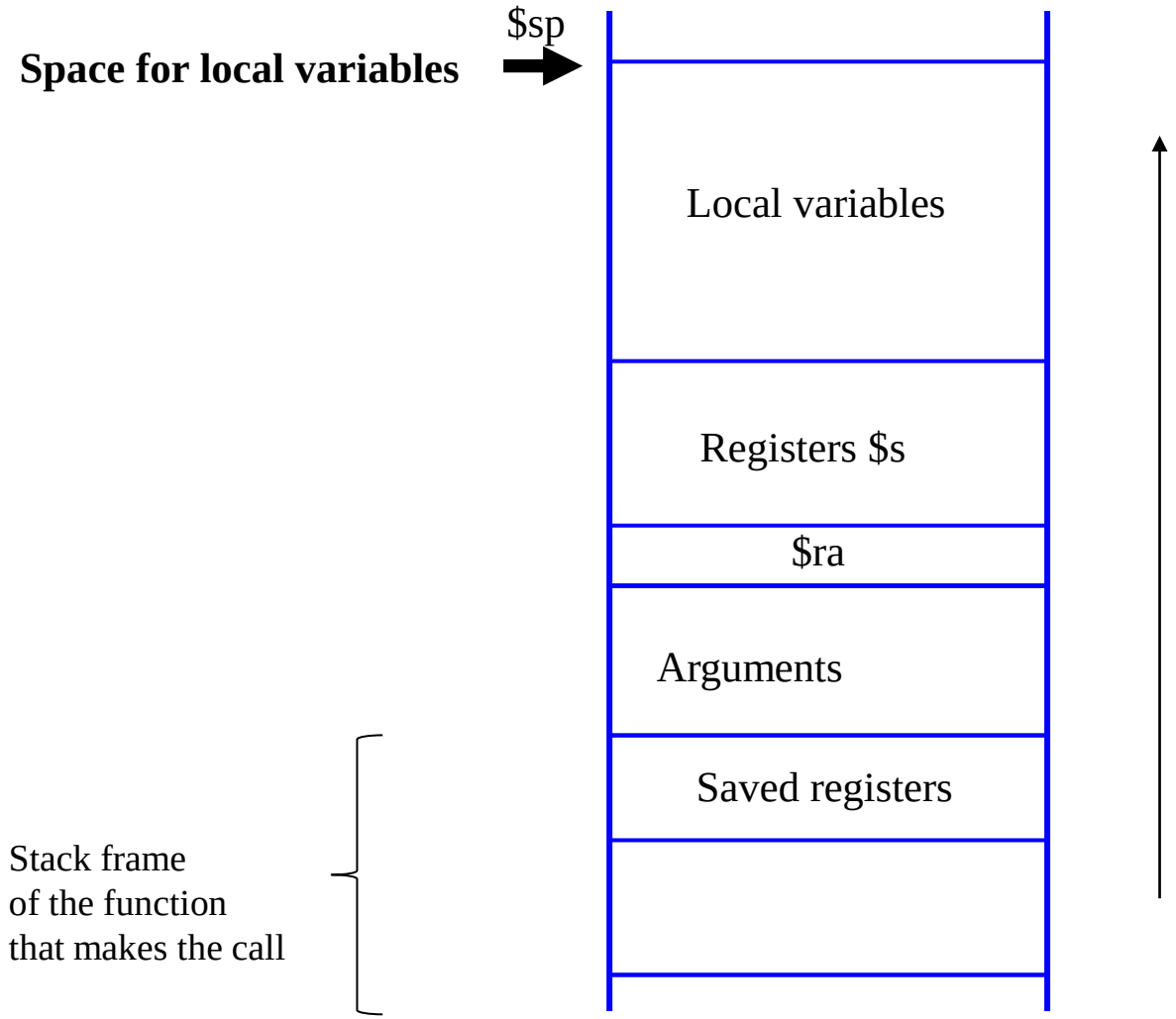


Example:

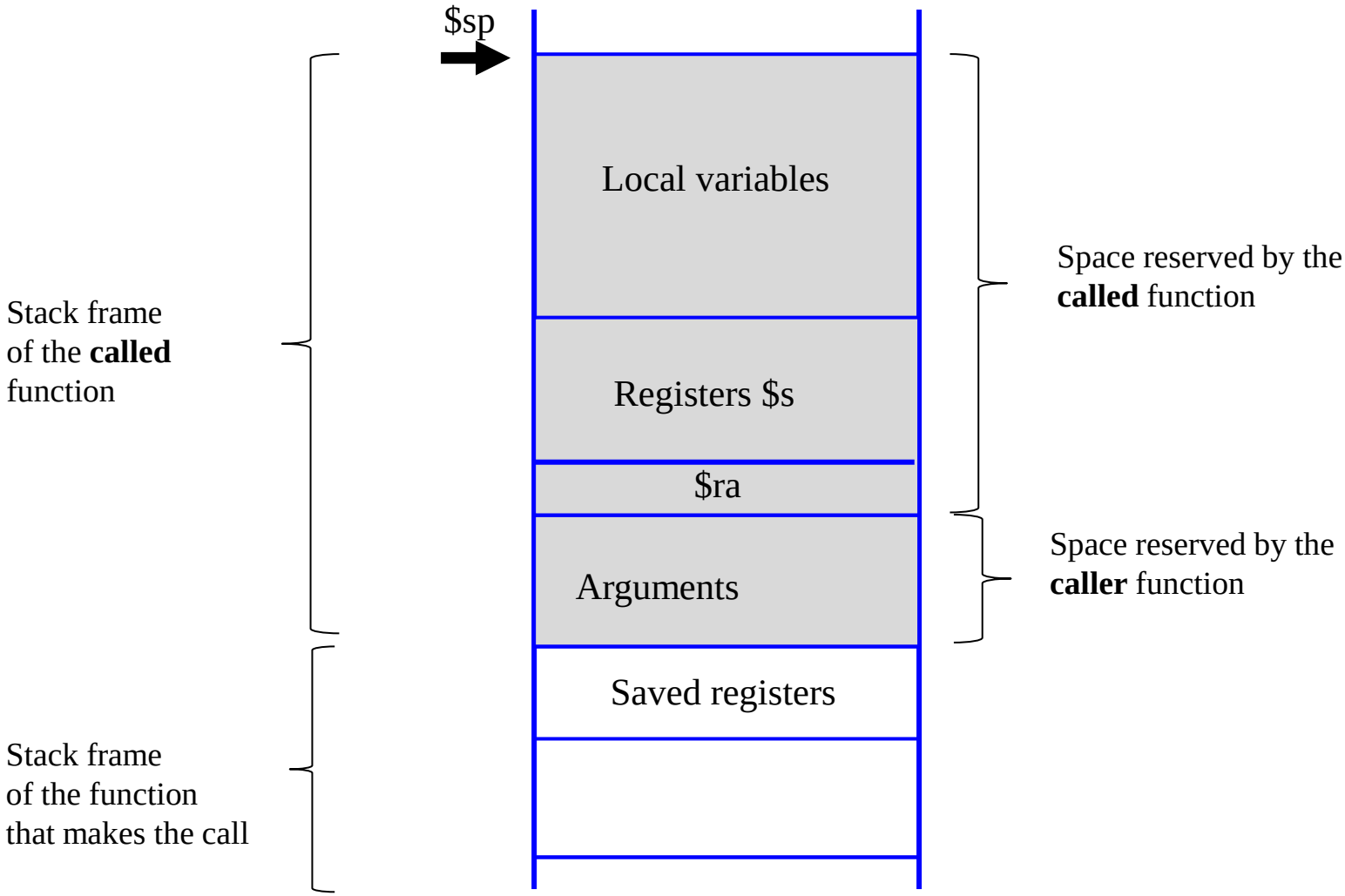
```
function:
   subu $sp $sp <fr.sz.>
   sw $ra <fr.sz-4>($sp)
   sw $s0 <fr.sz-8>($sp)
   sw $s1       <...>($sp)
   …
```

Construction of the stack frame
called function

- **Space for local variables**

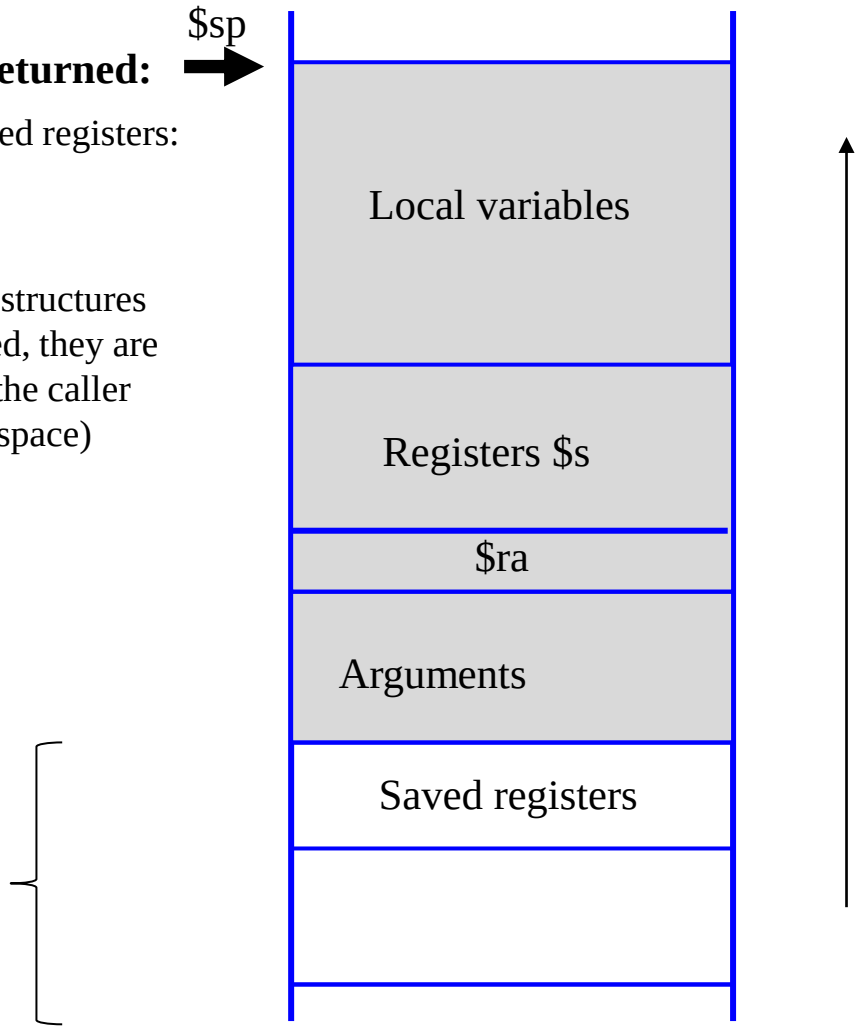


Stack frame construction



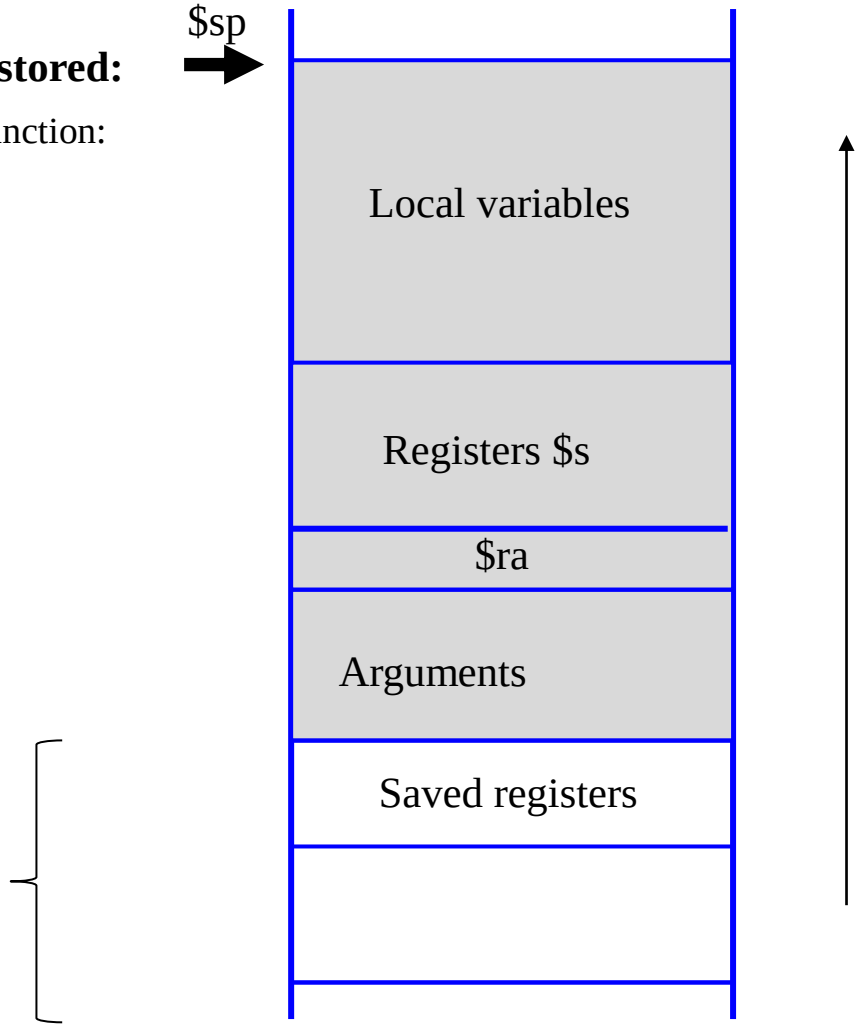
Subroutine termination
called function

- **The results are returned:**
  - Use the appropiated registers:
    - $v0, $v1
    - $f0
  - If more complex structures need to be returned, they are left on the stack (the caller must allocate the space)



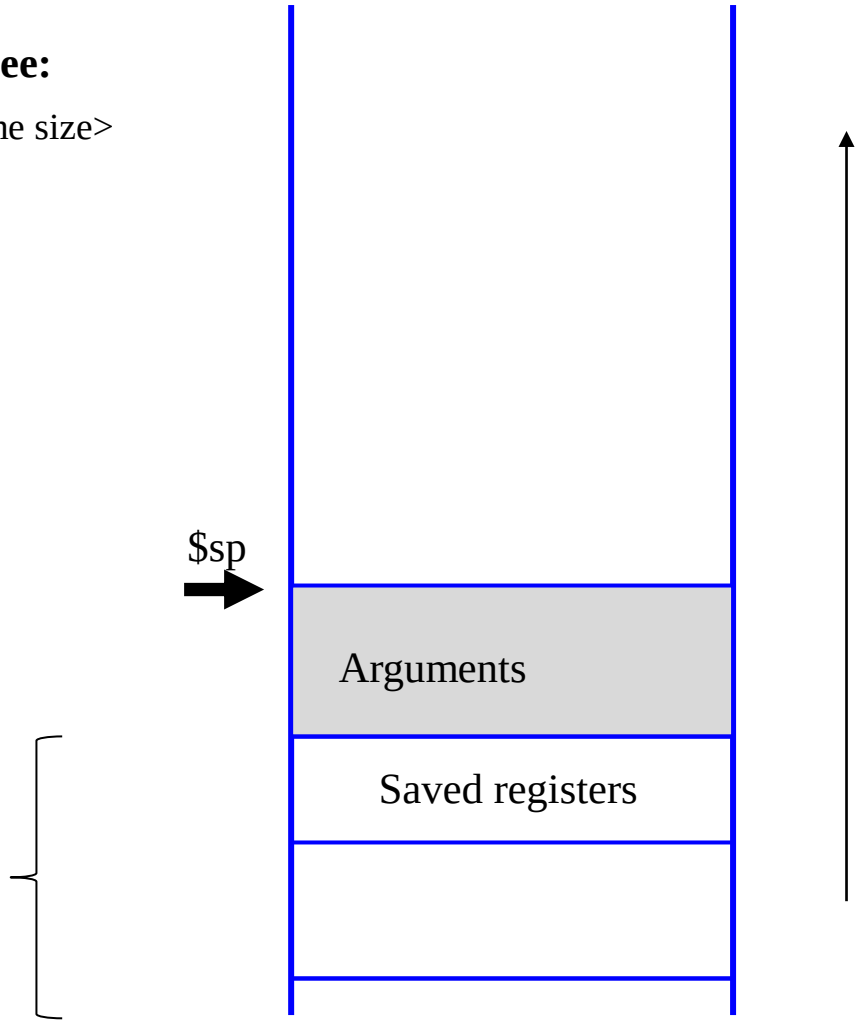
Subroutine termination
called function

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



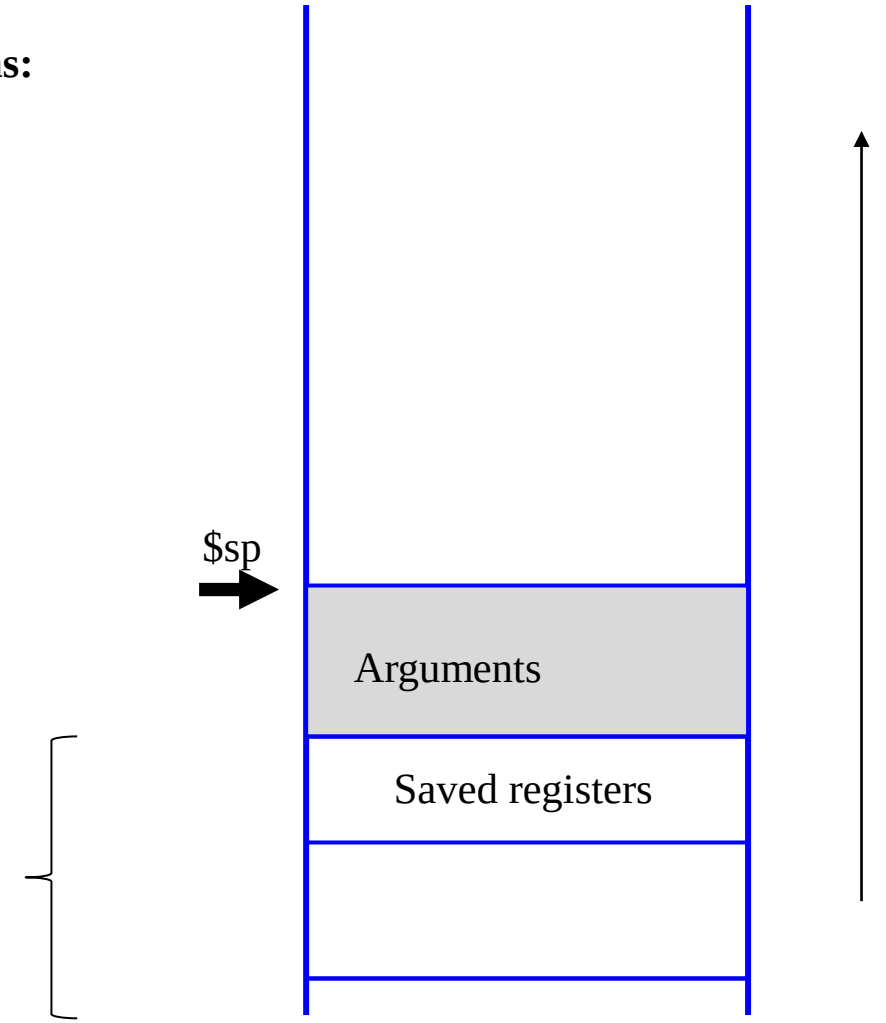
Subroutine termination
called function

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



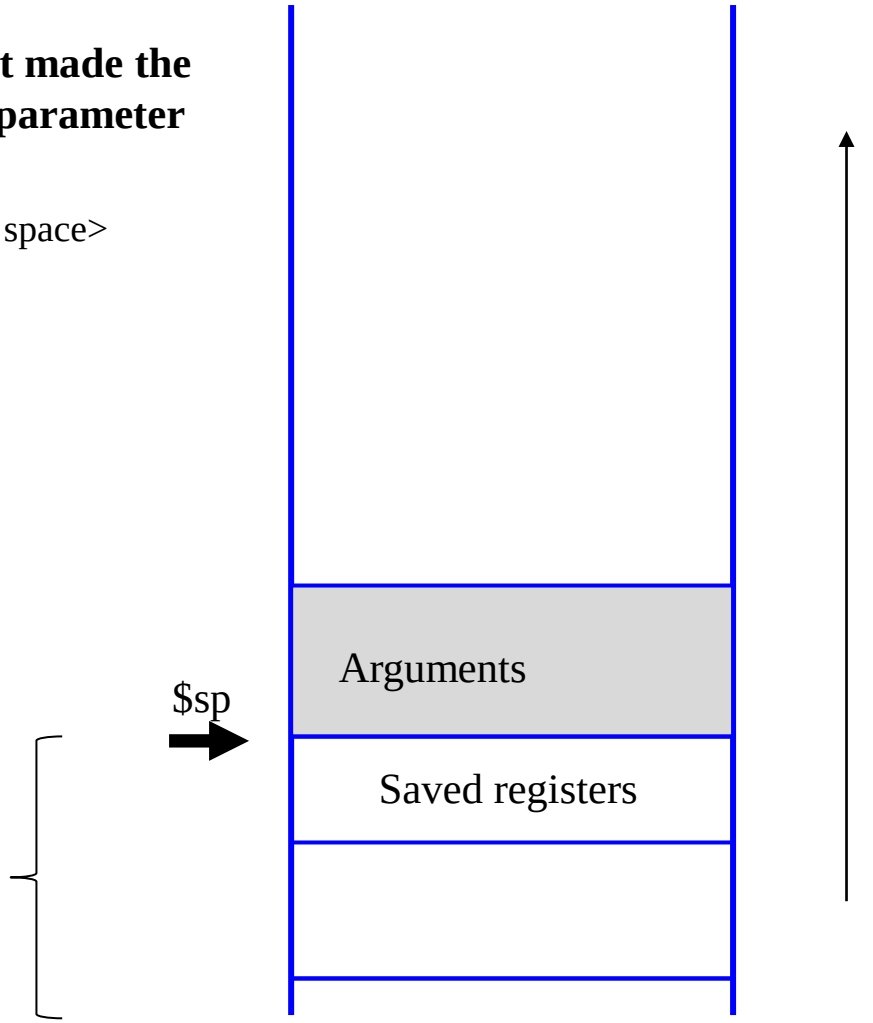
Subroutine termination
called function

- **Function returns:**
  - jr $ra



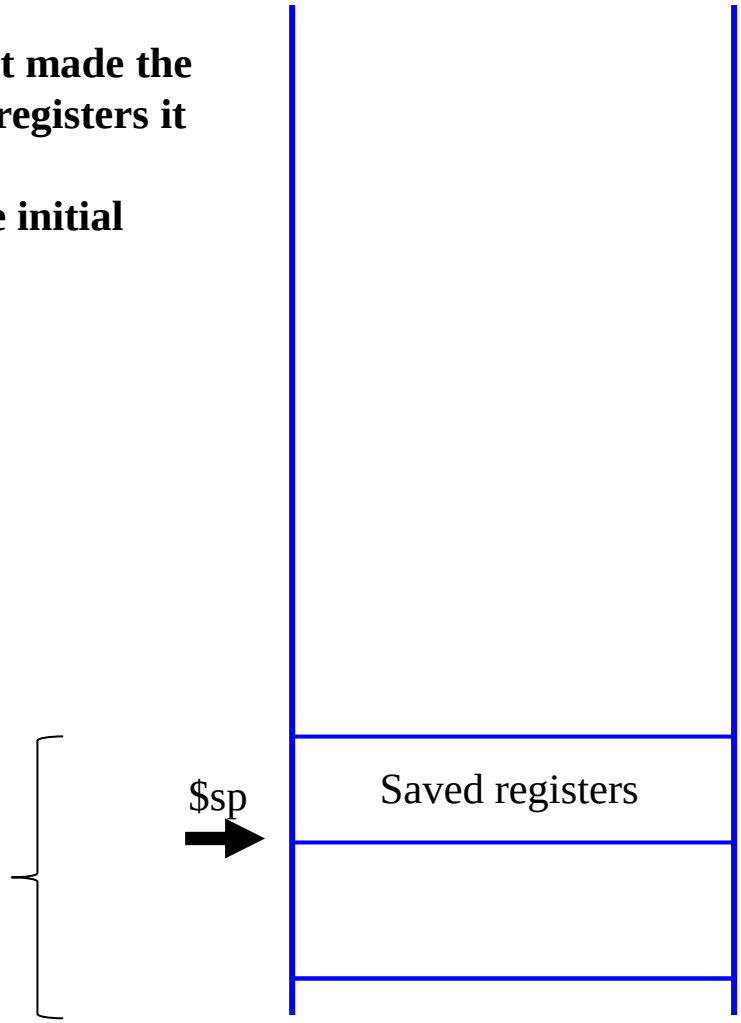
Subroutine termination
calle function

- **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**

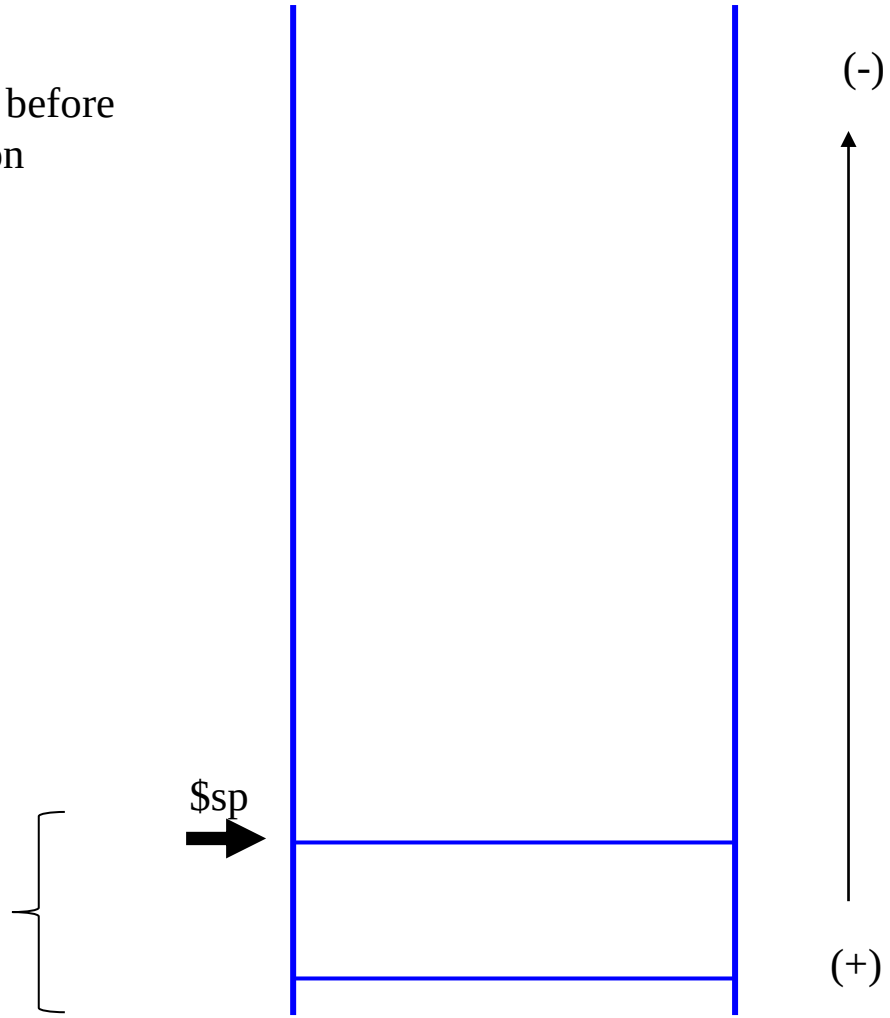


Example:

```
subu $sp $sp 8
sw   $t0  ($sp)
sw   $t1  4($sp)

li   $a0, 5
jal  funcion

lw   $t0  ($sp)
lw   $t1  4($sp)
addu $sp $sp 8
```

State after subroutine termination

- **Initial** situation before calling a function

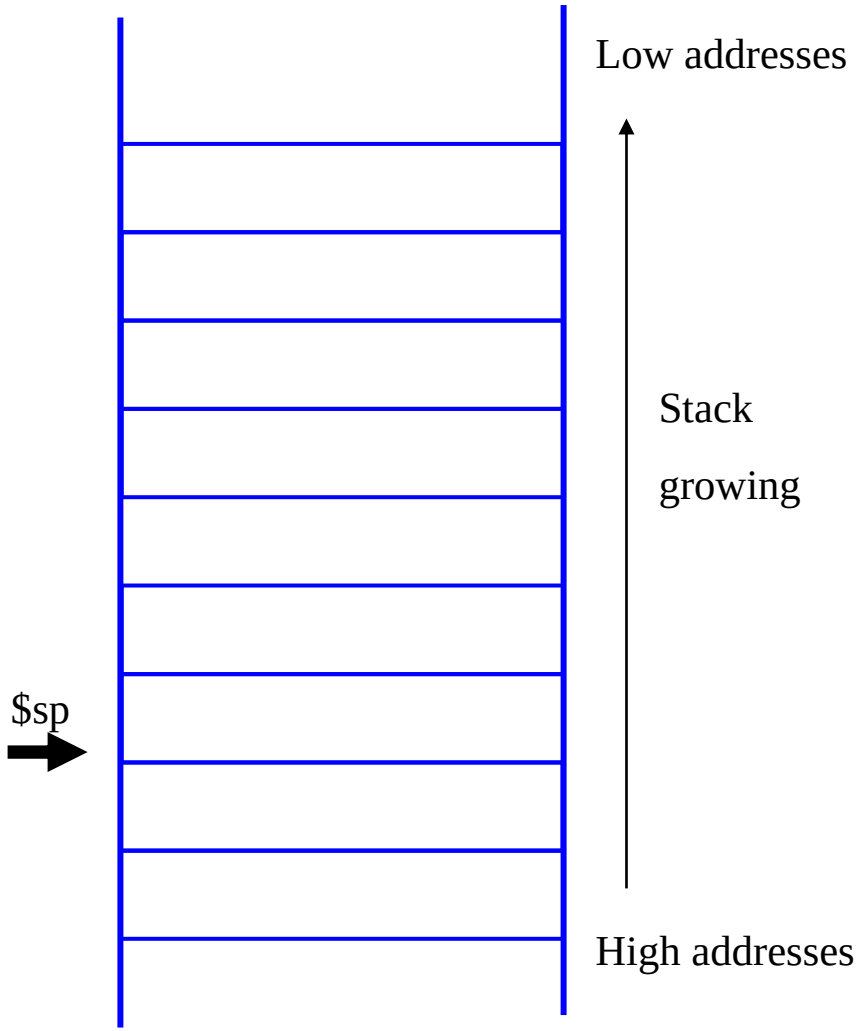


Stack frame
of the function
that makes the call

Access to parameters and local variables using the stack frame

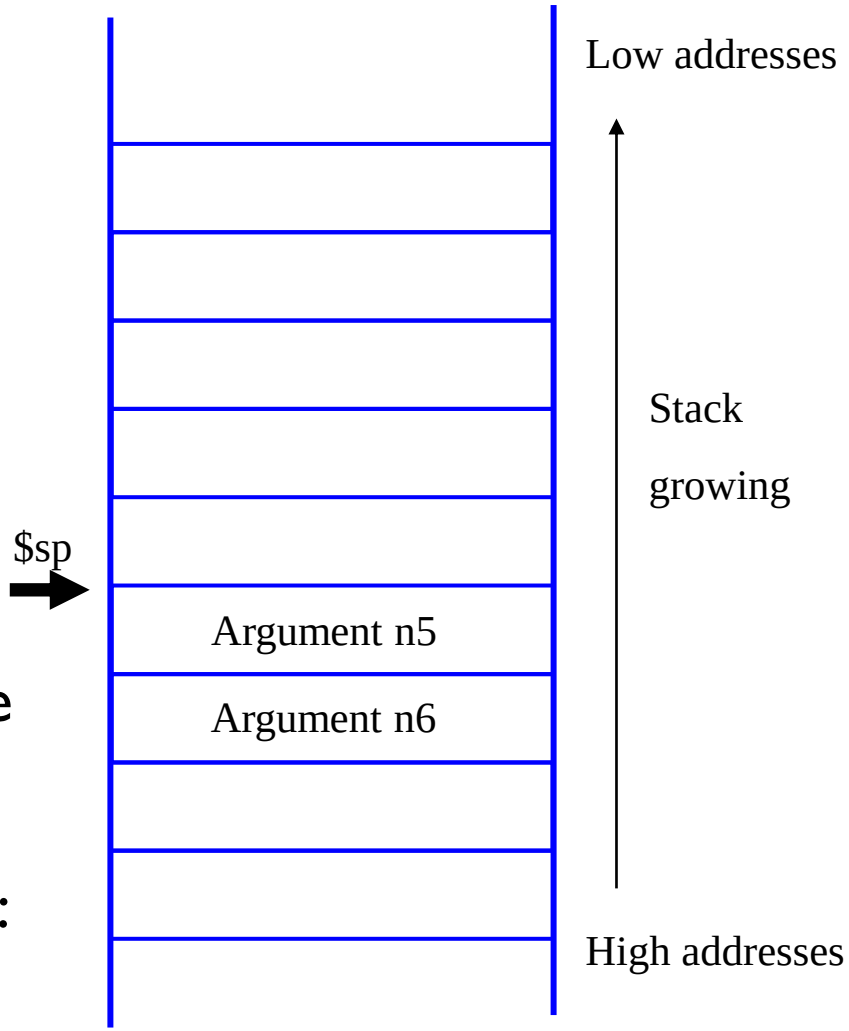
```
int f (int n1, n2, n3,
            n4, n5, n6)
{
   int v[4];
   int k;

   for (k= 0; k <3; k++) {
     v[i] = n1+n2+n3+n4+n5+n6;
   }
   return (v[1]);
}
```

- If a call to f(…) is made…



Access to parameters and local variables using the stack frame

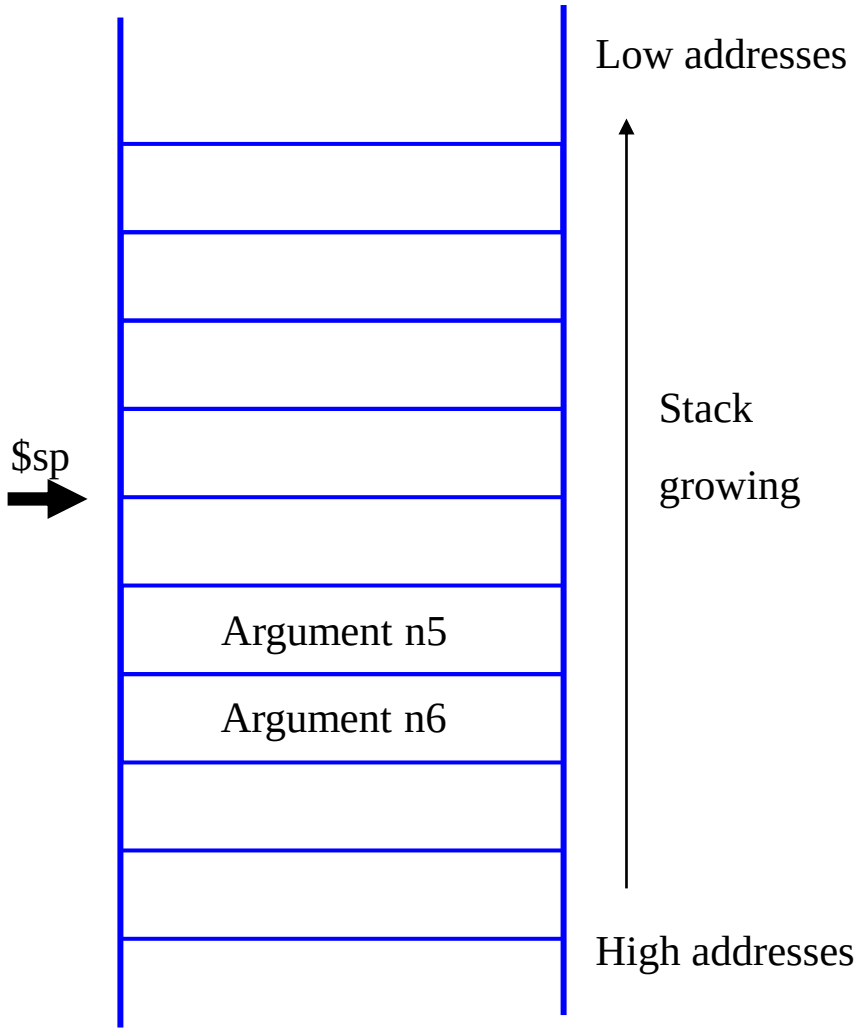
```
int f (int n1, n2, n3,
            n4, n5, n6)
{
   int v[4];
   int k;

   for (k= 0; k <3; k++) {
     v[i] = n1+n2+n3+n4+n5+n6;
   }
   return (v[1]);
}
```



- Arguments `n1`, `n2`, `n3` y `n4` are placed in:
  - `$a0`, `$a1`, `$a2`, `$a3`
- Arguments `n5`, `n6` are placed in:
  - The stack

Access to parameters and local variables using the stack frame

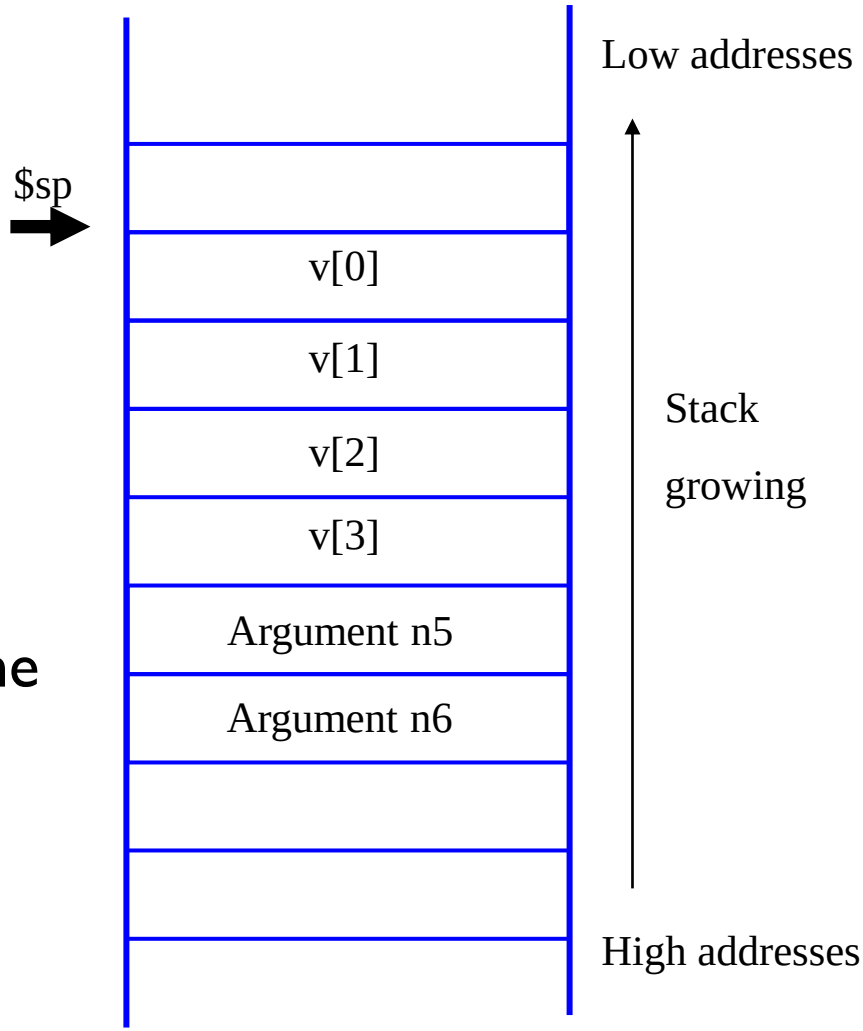
```
int f (int n1, n2, n3,
            n4, n5, n6)
{
   int v[4];
   int k;

   for (k= 0; k <3; k++) {
     v[i] = n1+n2+n3+n4+n5+n6;
   }
   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)
{
   int v[4];
   int k;

   for (k= 0; k <3; k++) {
     v[i] = n1+n2+n3+n4+n5+n6;
   }
   return (v[1]);
}
```

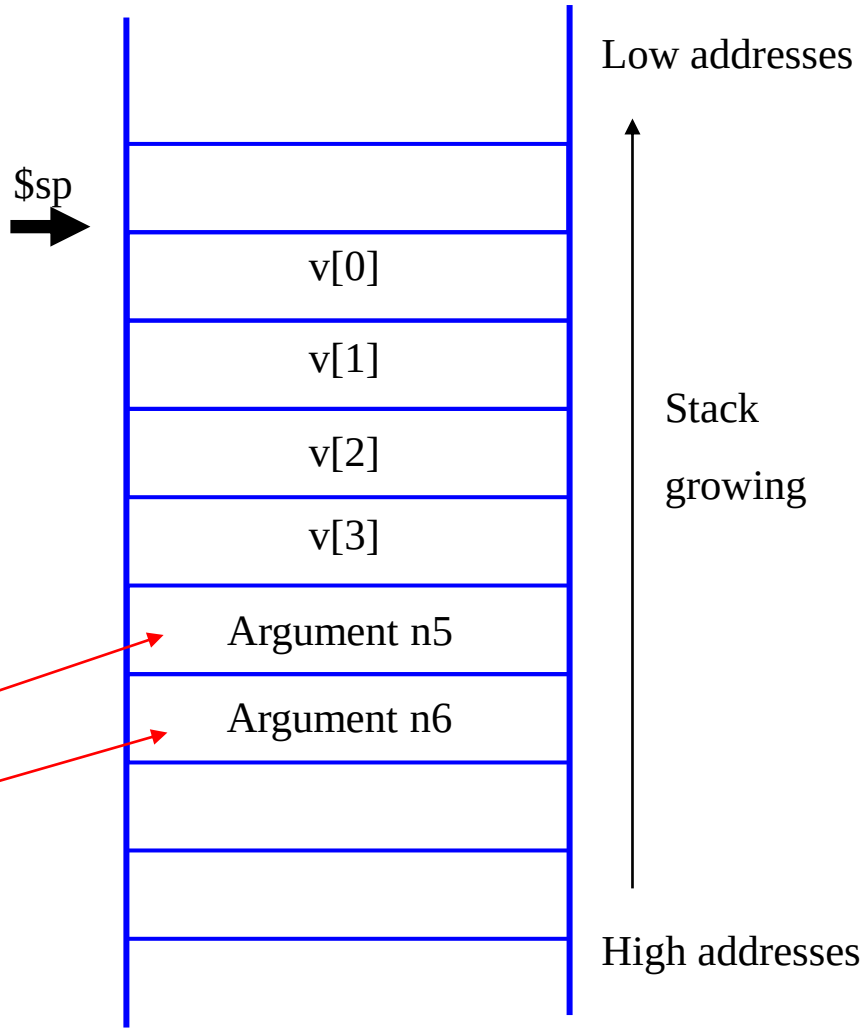


- f must reserve in the stack frame space for local variables that cannot be stored in registers (v in this example)
  - In this example f is not going to modify any register

Access to parameters and local variables using the stack frame

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

   for (k= 0; k <3; k++) {
     v[i] = n1+n2+n3+n4+n5+n6;
   }
   return (v[1]);
}
```

- The value of `n1` is in `$a0`
- The value of `n5` is in `-16($sp)`
- The value of `n6` is in `-20($sp)`
- The value of `v[3]` is in `-12($sp)`
- The value of `v[0]` is in `0($sp)`



Exercise

Code for the following call: `f(3, 4, 23, 12, 6, 7);`

Exercise (solution)

Code for the following call: `f(3, 4, 23, 12, 6, 7);`

```
li   $a0, 3
li   $a1, 4
li   $a2, 23
li   $a3, 12
addu $sp, $sp, -8
li   $t0, 6
sw   $t0, ($sp)
li   $t0, 7
sw   $t0, 4($sp)
jal  f
```

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
     . . .
```

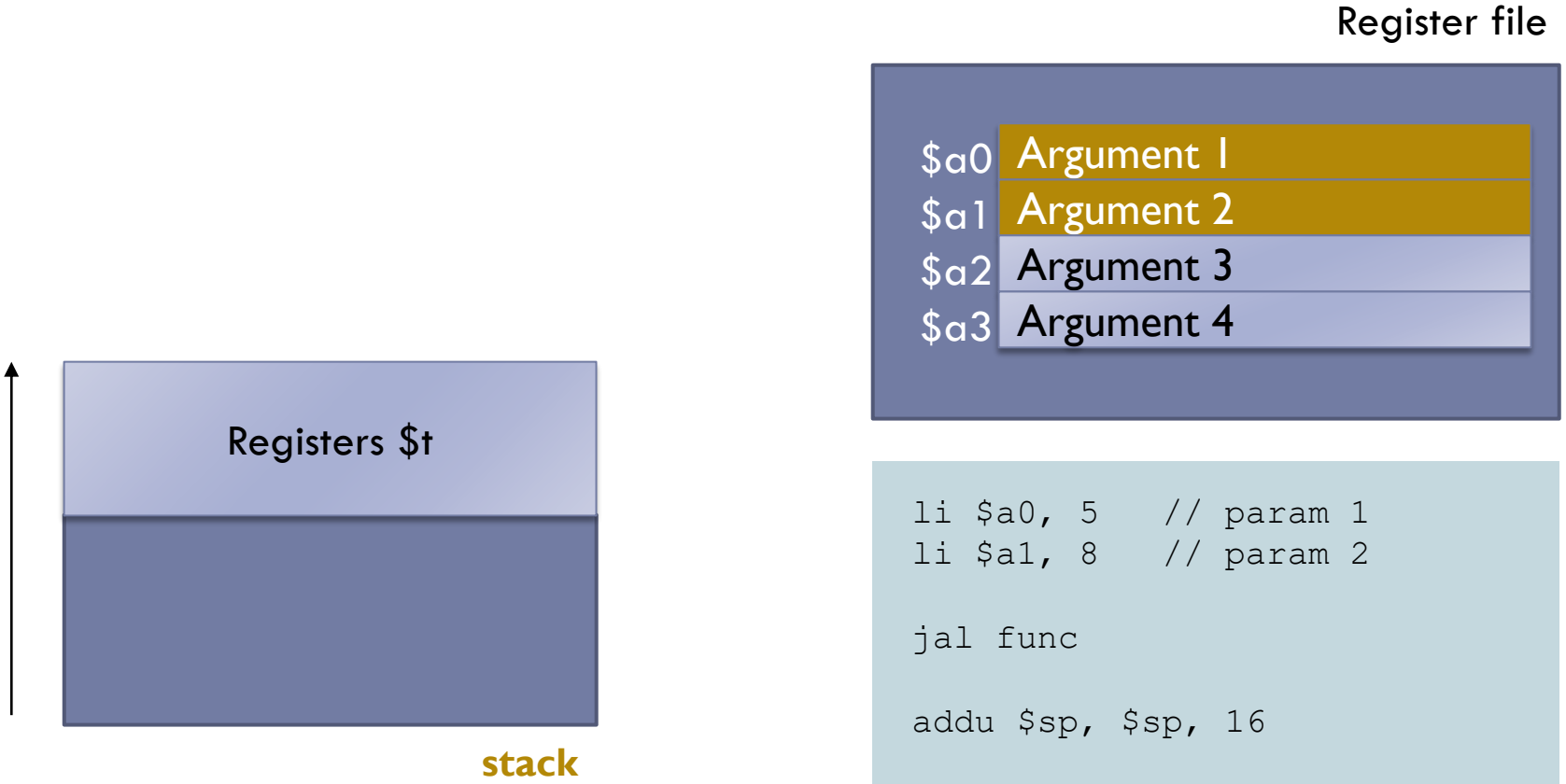
Exercise

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.

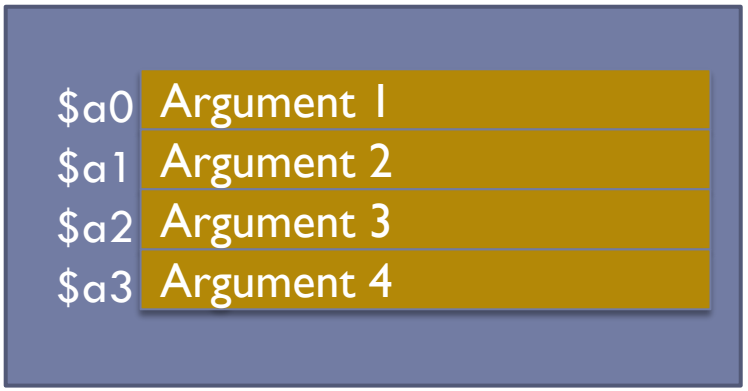
Passing 2 parameters

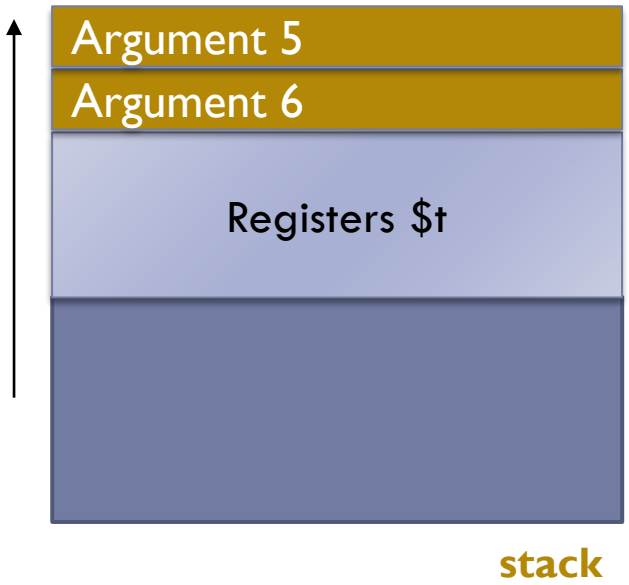



```
li $a0, 5   // param 1
li $a1, 8   // param 2

jal func

addu $sp, $sp, 16
```

Passing 6 parameters

Register file

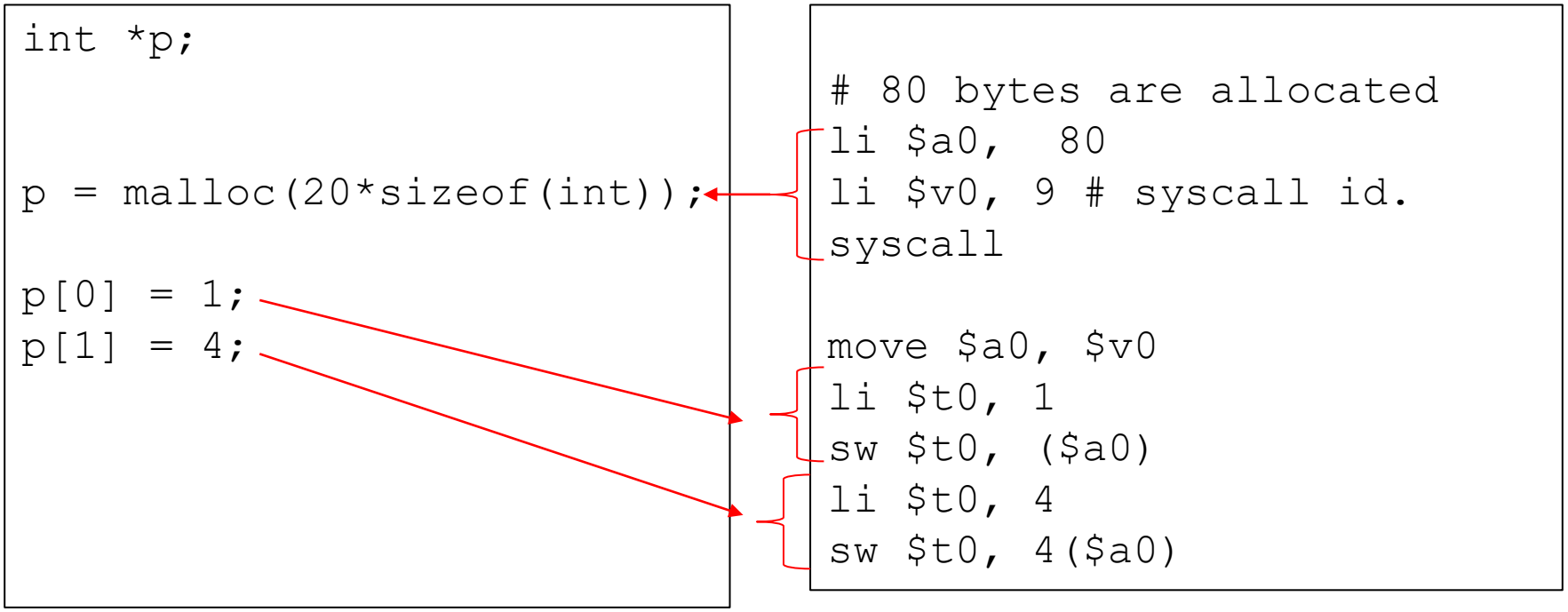





```
li $a0, 5          // param 1
li $a1, 8          // param 2
li $a2, 7          // param 3
li $a3, 9          // param 4

subu $sp, $sp, 8
li $t0, 10         // param 6
sw $t0, 4($sp)
li $t0, 7
s2 $t0, ($sp)      // param 5

jal func

addu $sp, $sp, 8
```

Dynamic memory allocation in CREATOR

- The system call sbrk() in CREATOR
  - $a0: number of bytes to allocate
  - $v0 = 9 (system call code)
  - Return in $v0 the address of the allocated memory block
  - In CREATOR there is not a free system call



Translation and execution of programs

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

Translation and execution steps (C program)