Arquitectura de Computadores I

ABI - Application Binary Interface

Miguel Barão

ESCOLA DE CIÊNCIAS E TECNOLOGIA
DEPARTAMENTO DE INFORMÁTICA

Resumo

Application Binary Interface (ABI)

Chamadas de funções (ABI RISC-V)

Funções recursivas

Espaço de endereçamento

Application Binary Interface (ABI)

API vs ABI

API = Application Programming Interface

$\neq$

ABI = Application Binary Interface

Application Programming Interface (API)

Define, em alto nível, como se podem realizar as interacções entre programas e/ou bibliotecas de software:

- Que funções existem e o que fazem,
- Quais os parâmetros das funções,
- Quais as estruturas de dados,
- etc.

Application Programming Interface (API)

Define, em alto nível, como se podem realizar as interacções entre programas e/ou bibliotecas de software:

- Que funções existem e o que fazem,
- Quais os parâmetros das funções,
- Quais as estruturas de dados,
- etc.

Simplifica a programação porque o programador só precisa de saber como usar as funções/objectos/estruturas e não como estão implementados.

Podem existir várias implementações diferentes da mesma API.

Application Programming Interface (API)

Define, em alto nível, como se podem realizar as interacções entre programas e/ou bibliotecas de software:

- Que funções existem e o que fazem,
- Quais os parâmetros das funções,
- Quais as estruturas de dados,
- etc.

Simplifica a programação porque o programador só precisa de saber como usar as funções/objectos/estruturas e não como estão implementados.

Podem existir várias implementações diferentes da mesma API.

Uma API usa-se no código fonte em linguagens de alto nível.

Exemplos: libc, POSIX, Win32, JDBC, Twitter REST API.

Application Binary Interface (ABI)

Define como se processam as interacções em baixo nível (código máquina):

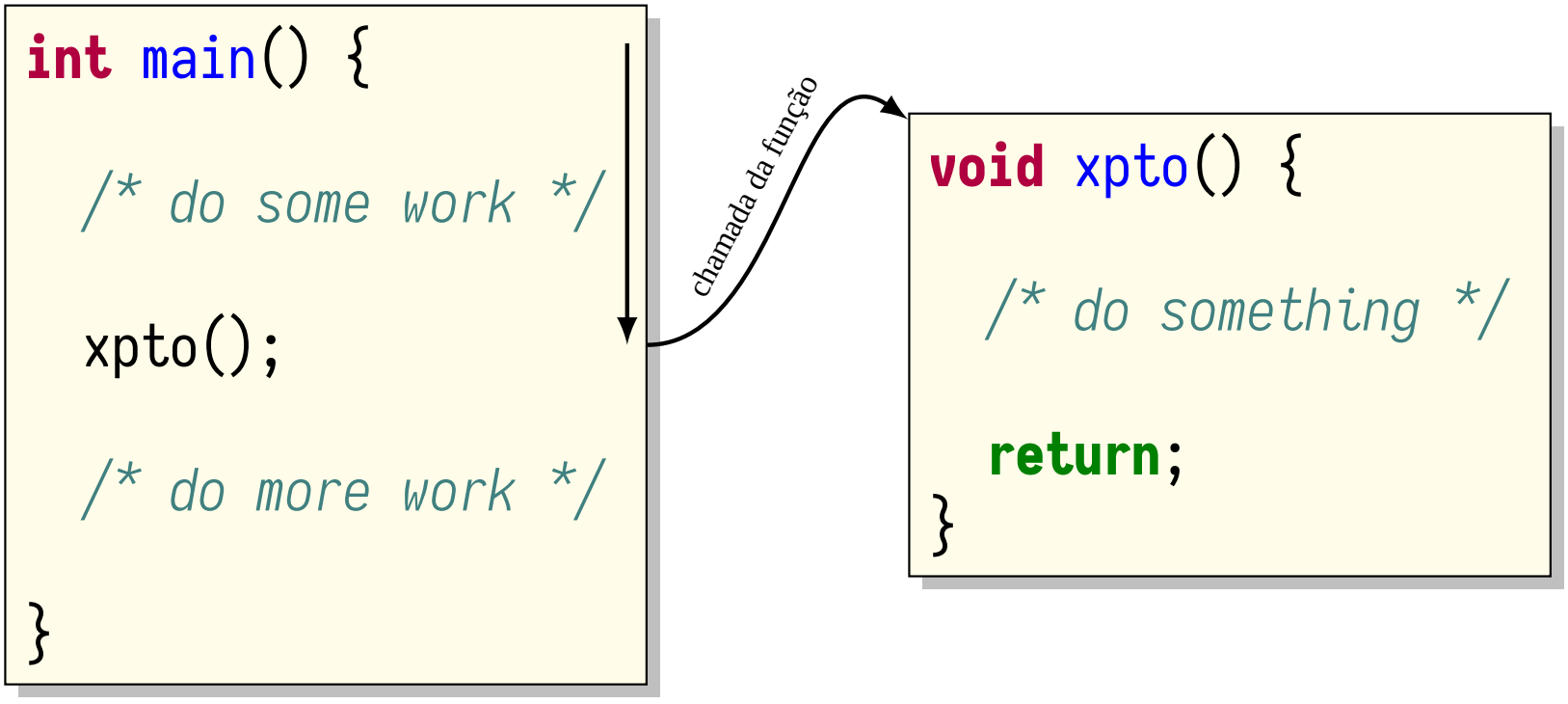
- Como se chamam funções,
- Como se passam argumentos para funções,
- Como se retorna o resultado de uma função,
- Tamanhos, layout e alinhamento dos tipos de dados básicos,
- etc.

Permite que programas com a mesma ABI possam interagir entre si (chamar funções e passar argumentos).

A adesão a uma ABI é tarefa do compilador.

Chamadas de funções (ABI RISC-V)

Chamada de funções em C



Função `main` em execução.
Chama a função `xpto` e espera que ela termine...

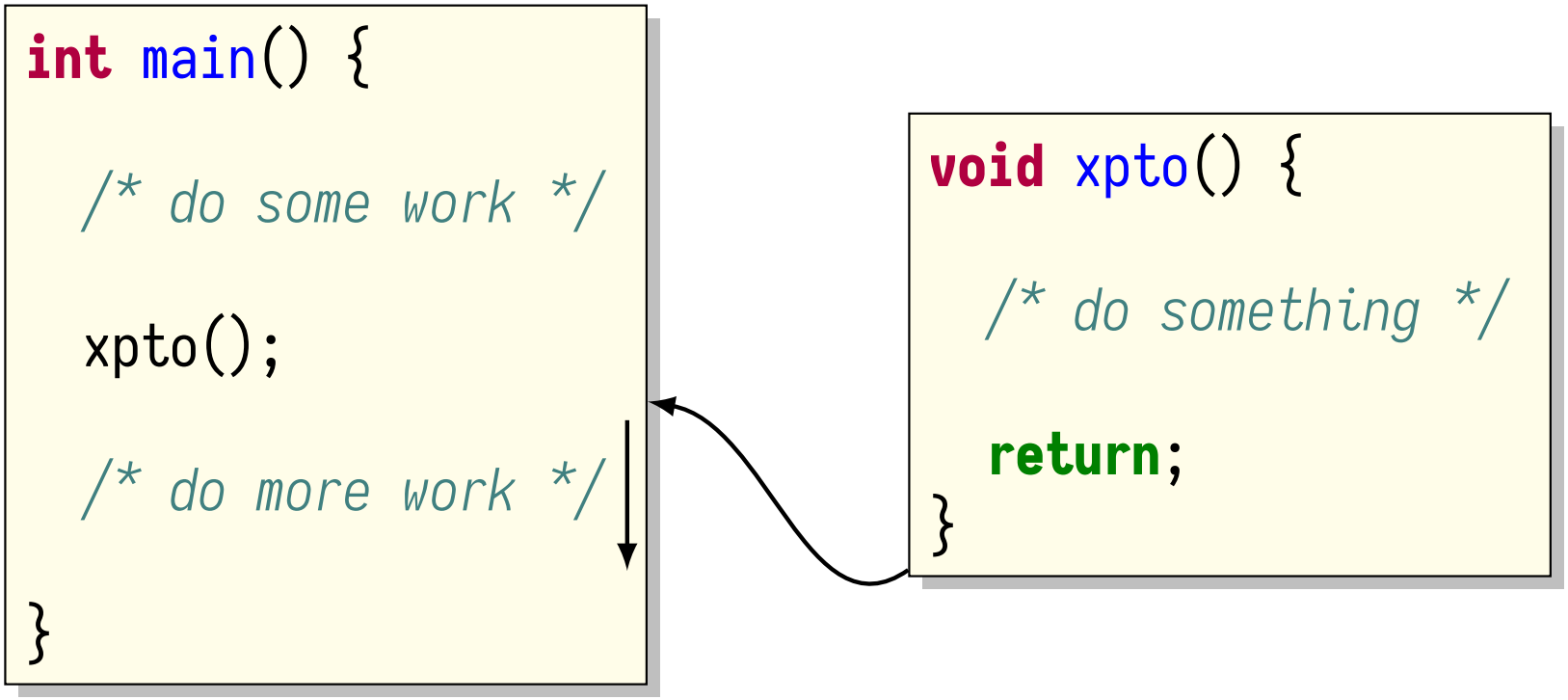
Chamada de funções em C

```c
int main() {

  /* do some work */

  xpto();

  /* do more work */

}
```

```c
void xpto() {

  /* do something */

  return;
}
```

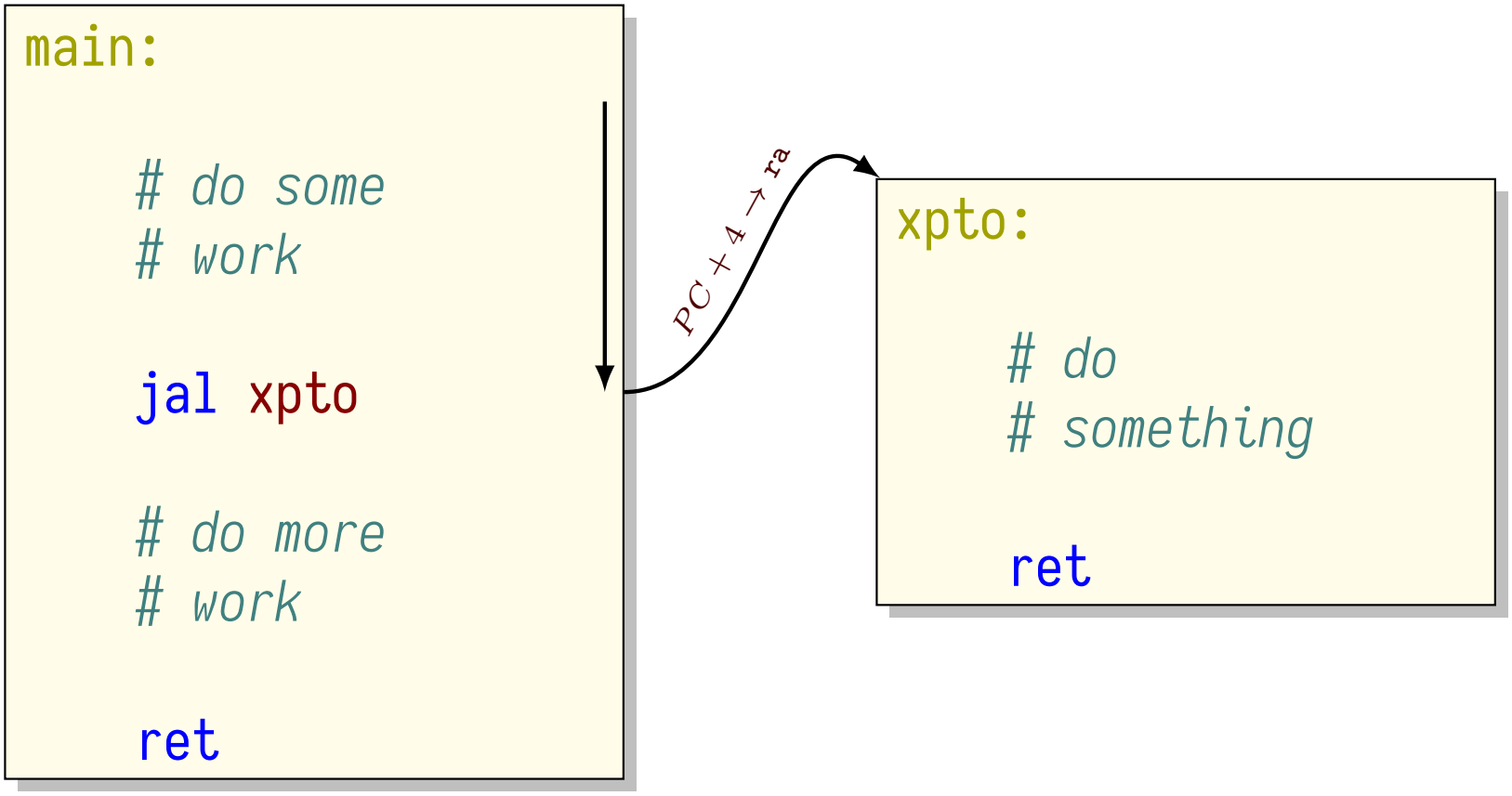
Função `xpto` em execução.
Estão as duas activas (não terminaram).

Chamada de funções em C



Função `xpto` termina e retorna à função inicial.
Função `main` continua execução.

Chamada de funções em RISC-V



Função `main` em execução.
Chama `xpto` e guarda *return address* no registo `ra`).

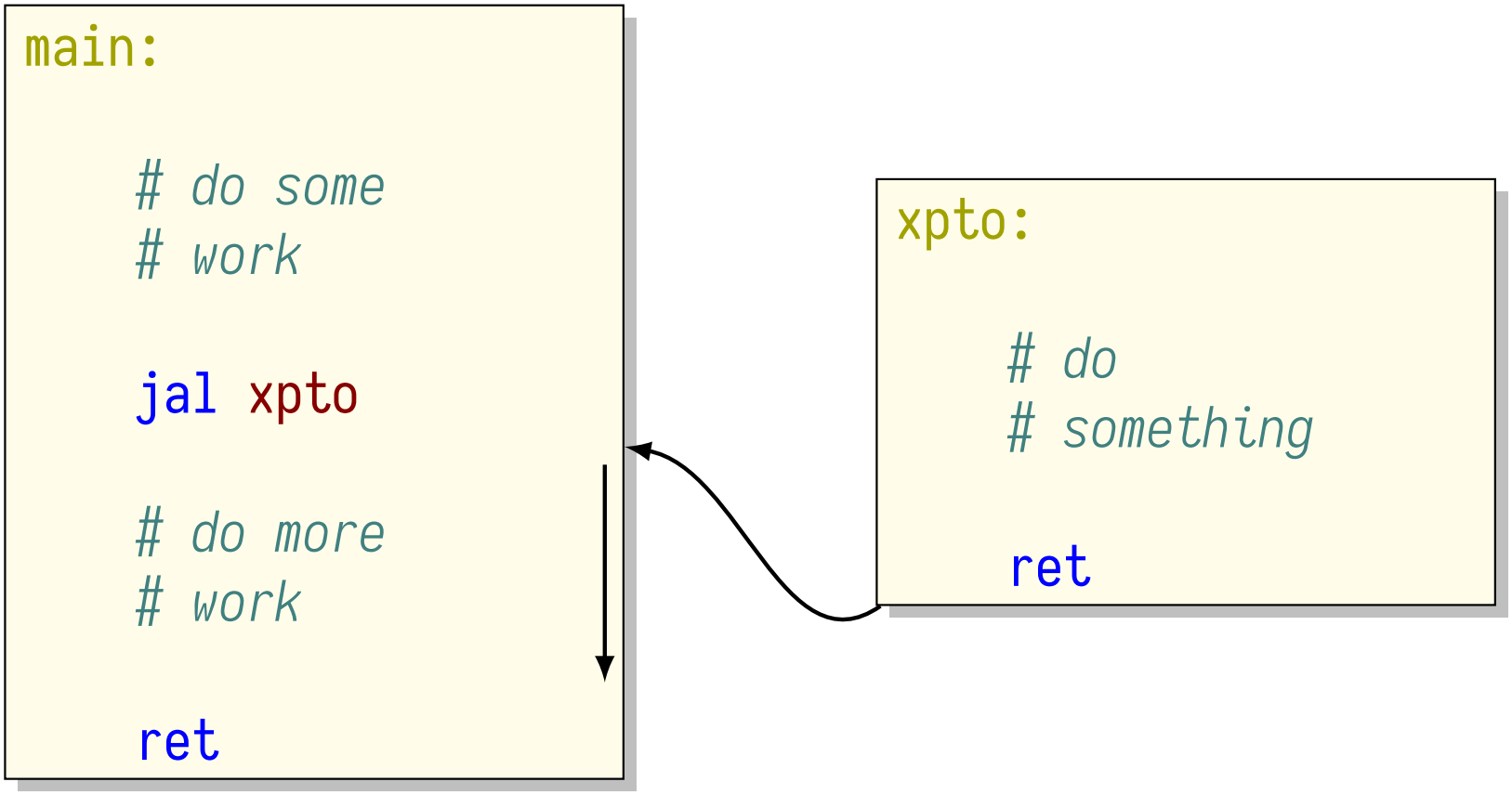
Chamada de funções em RISC-V

```
main:

    # do some
    # work

    jal xpto

    # do more
    # work

    ret
```

```
xpto:

    # do
    # something

    ret
```

Função xpto em execução.
Estão as duas activas (não terminaram).

Chamada de funções em RISC-V



Função `xpto` termina e salta para o *return address* guardado em `ra`.
Função `main` continua execução.

Passagem de argumentos para funções

Na chamada de uma função, os 8 primeiros argumentos são passados pelos registos

`a0, a1, a2, ..., a7`

Passagem de argumentos para funções

Na chamada de uma função, os 8 primeiros argumentos são passados pelos registos

`a0, a1, a2, ..., a7`

- 1 argumento: usa-se o `a0`.
- 2 argumentos: usa-se o `a0` e `a1`.
- 3 argumentos: usa-se o `a0`, `a1` e `a2`.
- etc...

Se forem mais de 8 argumentos, é necessário usar a memória.

Valor de retorno da função

Cada função só pode retornar 0 ou 1 valor.

- Valor até 32 bits (`int`, `char` ou endereço), usa-se o `a0`.
- Valor de 64 bits (`long long int`), usa-se `a0` e `a1`.
- Se não retorna nada (`void`), os registos não são necessários.

Exemplo 1 - chamada de uma função

Pretende-se executar a instrução

```
y = 2 * abs(x);
```

onde

- `abs()` é uma função,
- a variável `x` está no registo `t0`,
- a variável `y` está no registo `t1`,

Em Assembly:

```
mv a0, t0           # copia t0 -> a0 (argumento)
jal abs
slli t1, a0, 1      # multiplica por 2
```

Exemplo 2 - chamada de uma função

Pretende-se executar a instrução

```
y = abs(x) + x;
```

- `abs()` é uma função,
- a variável `x` está no registo `t0`,
- a variável `y` está no registo `t1`,

Em Assembly:

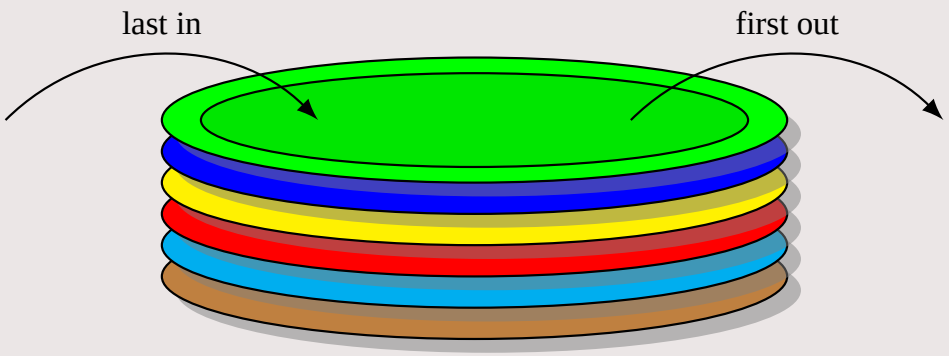
```
mv a0, t0          # copia t0 -> a0 (argumento)
jal abs
add t1, a0, t0     # t1 = abs(t0) + t0???
```

O que acontece se o registo `t0` for alterado dentro da função `abs`?
Nesse caso já não estamos a somar o valor original de `t0`!

A pilha (stack)

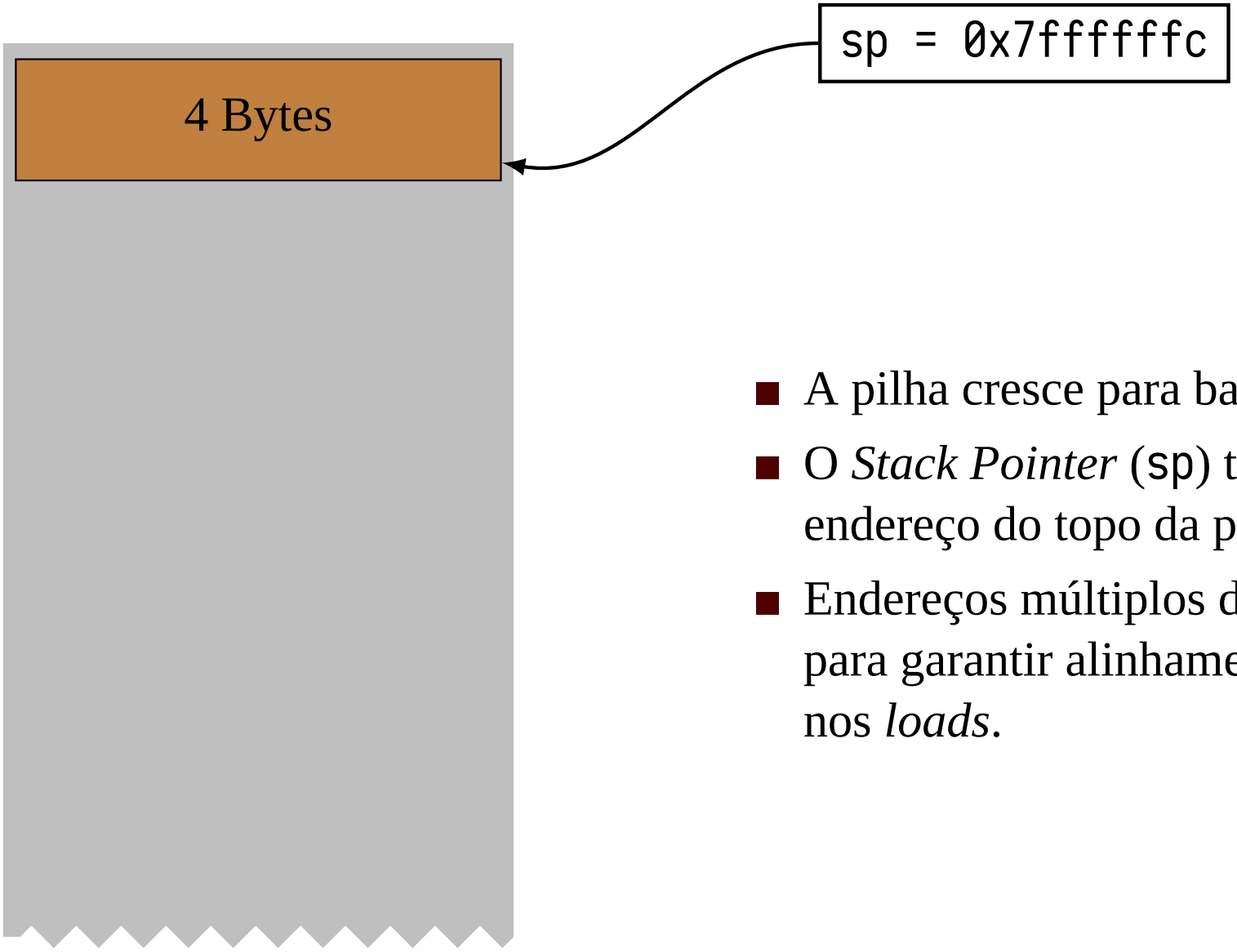
Definição (Pilha ou Stack)

É uma estrutura de dados LIFO (Last In First Out).



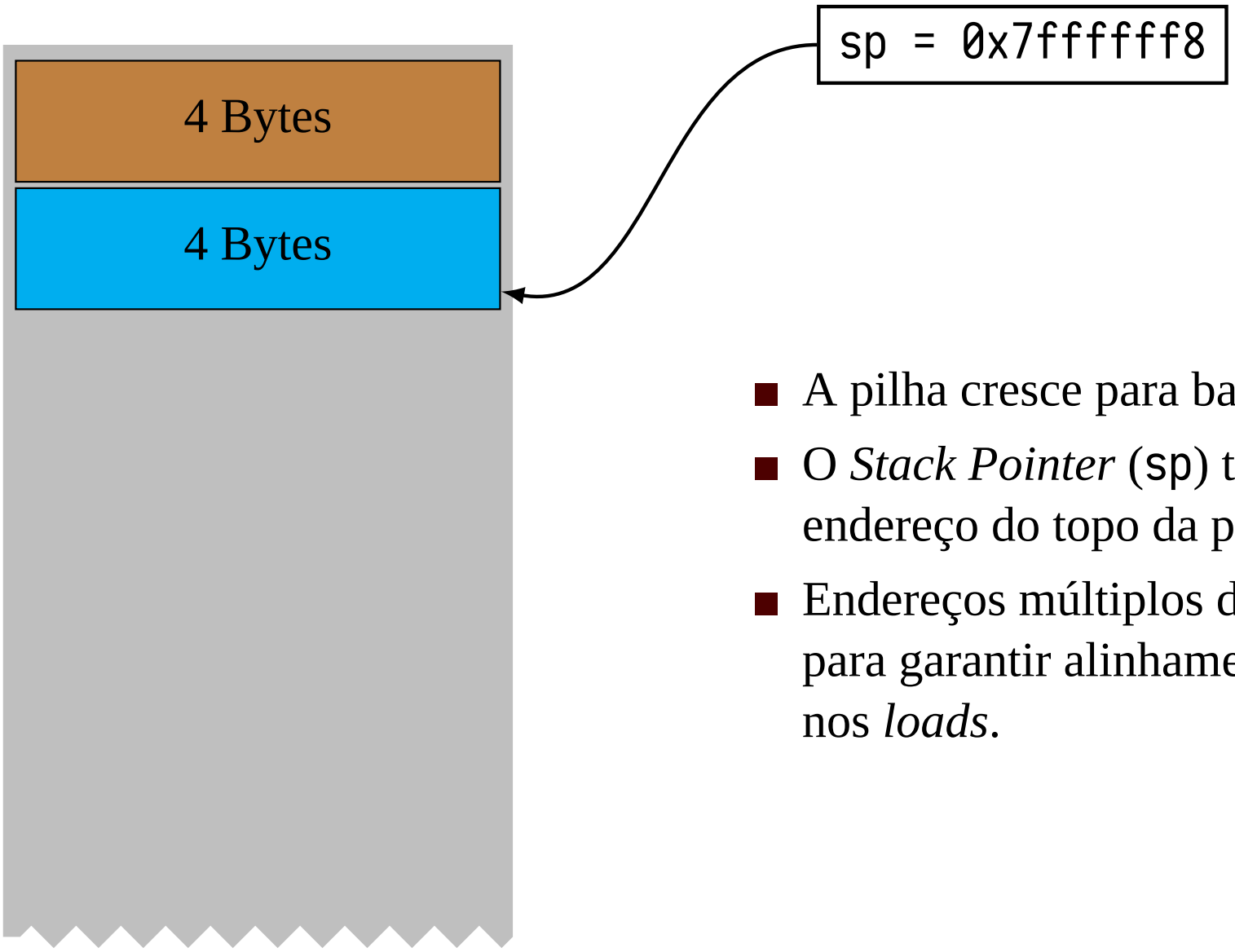
Funciona como uma pilha de pratos: o último a ser colocado na pilha é o primeiro a sair.

Implementação de uma pilha (*Stack Pointer*)



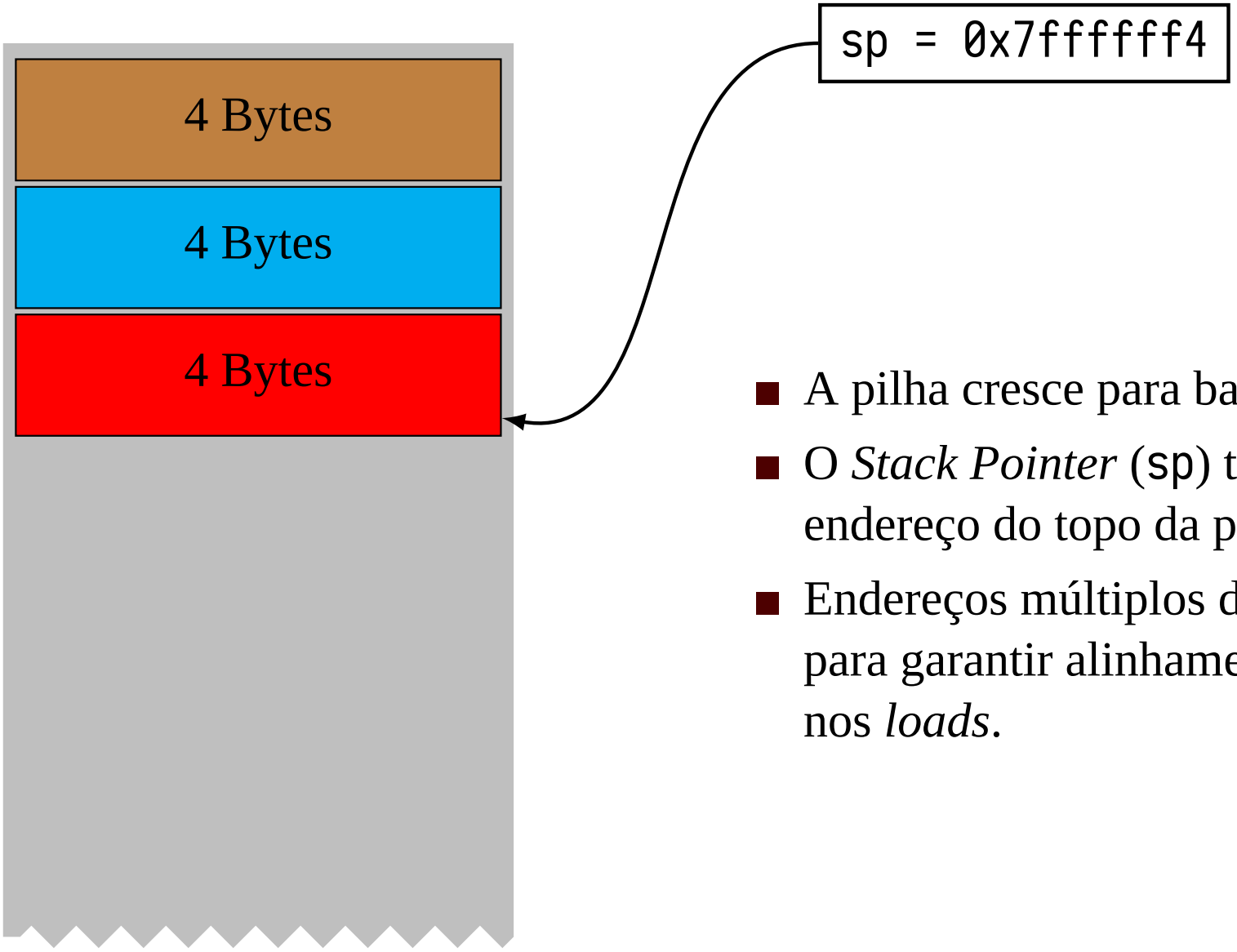
- A pilha cresce para baixo.
- O *Stack Pointer* (`sp`) tem o endereço do topo da pilha.
- Endereços múltiplos de 4 para garantir alinhamento nos *loads*.

Implementação de uma pilha (*Stack Pointer*)



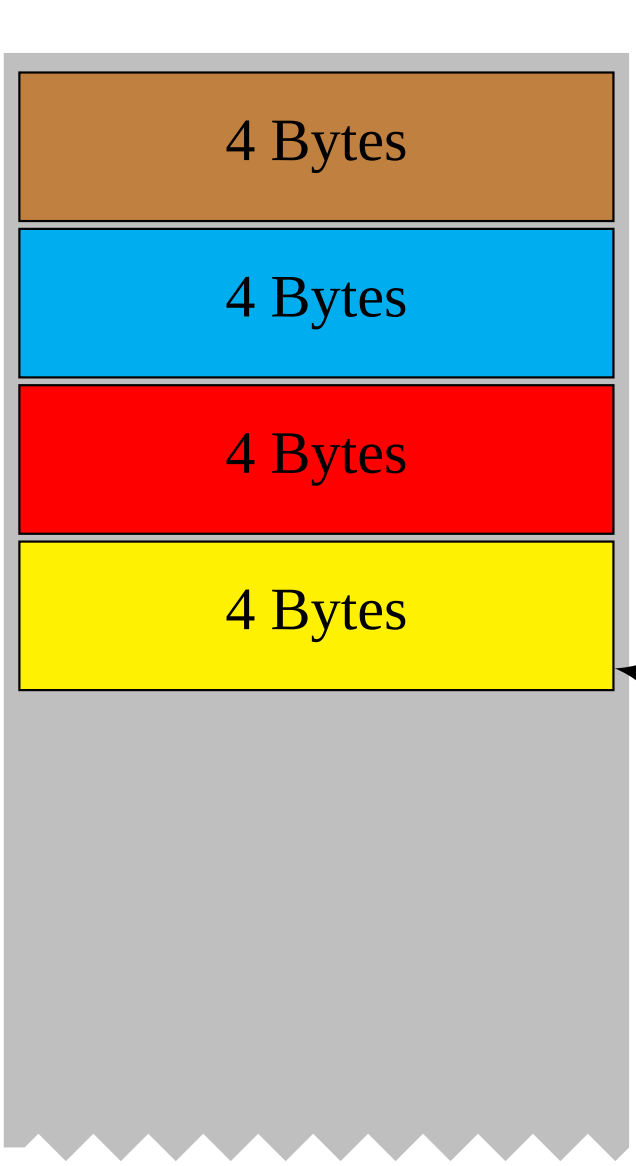
- A pilha cresce para baixo.
- O *Stack Pointer* (`sp`) tem o endereço do topo da pilha.
- Endereços múltiplos de 4 para garantir alinhamento nos *loads*.

Implementação de uma pilha (*Stack Pointer*)



- A pilha cresce para baixo.
- O *Stack Pointer* (`sp`) tem o endereço do topo da pilha.
- Endereços múltiplos de 4 para garantir alinhamento nos *loads*.

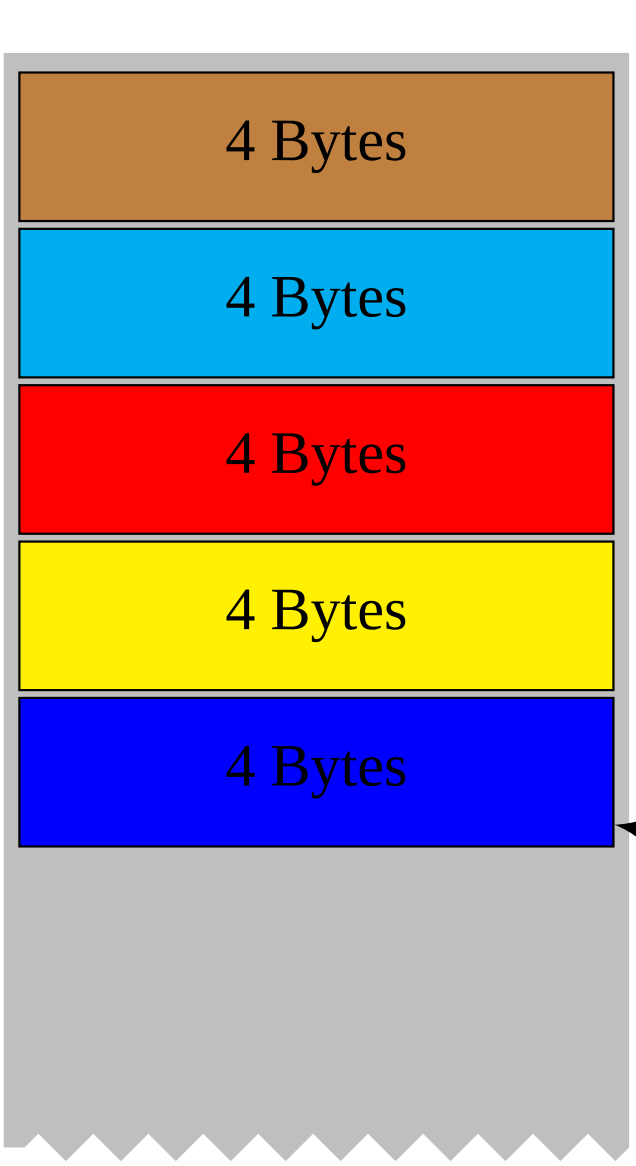
Implementação de uma pilha (*Stack Pointer*)



`sp = 0x7ffffff0`

- A pilha cresce para baixo.
- O *Stack Pointer* (`sp`) tem o endereço do topo da pilha.
- Endereços múltiplos de 4 para garantir alinhamento nos *loads*.

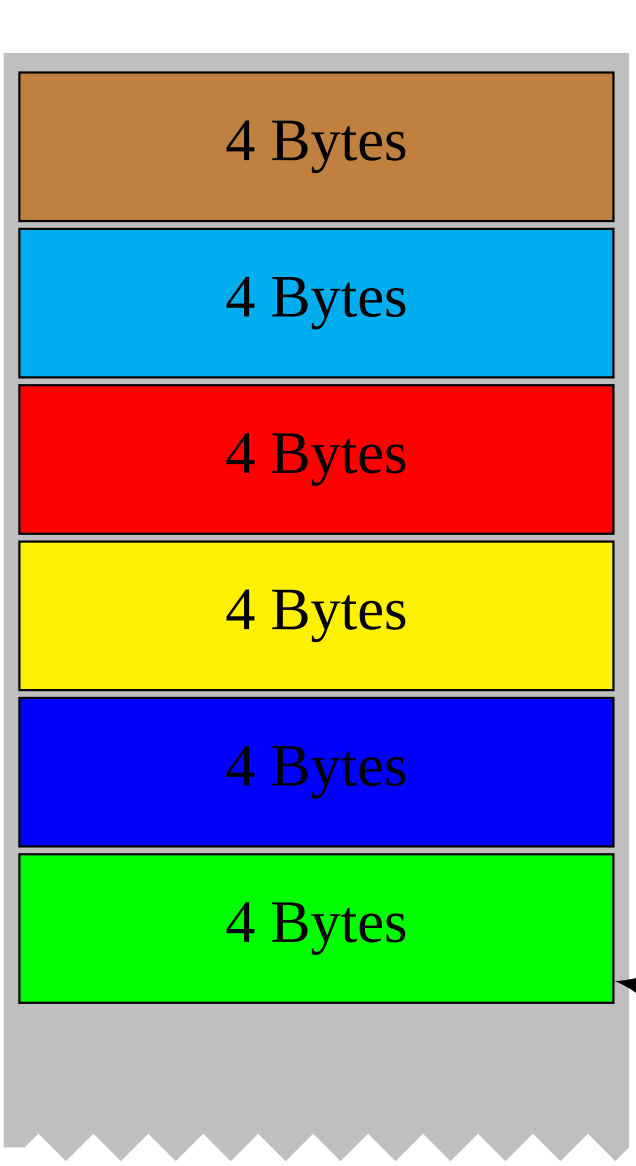
Implementação de uma pilha (*Stack Pointer*)



`sp = 0x7fffffec`

- A pilha cresce para baixo.
- O *Stack Pointer* (`sp`) tem o endereço do topo da pilha.
- Endereços múltiplos de 4 para garantir alinhamento nos *loads*.

Implementação de uma pilha (*Stack Pointer*)



- A pilha cresce para baixo.
- O *Stack Pointer* (`sp`) tem o endereço do topo da pilha.
- Endereços múltiplos de 4 para garantir alinhamento nos *loads*.

Exemplo 2 - chamada de uma função (versão corrigida)

```
y = abs(x) + x;
```

```
addi sp, sp, -4      # aloca espaco na pilha
sw t0, 0(sp)         # guarda t0 na pilha

mv a0, t0            # prepara argumento t0 -> a0
jal abs              # chama a funcao abs
                     # resultado retornado em a0

lw t0, 0(sp)         # recupera t0 original da pilha
add t1, a0, t0       # t1 = abs(t0) + t0

addi sp, sp, 4       # liberta espaco na pilha
```

Exemplo 3 - chamada de várias funções

```
y = f(x) + g(x);
```

```
addi sp, sp, -8      # aloca espaco na pilha (2 words)
sw t0, 4(sp)         # guarda t0 original na pilha

mv a0, t0            # copia t0 -> a0 (argumento)
jal f                # e chama funcao f
sw a0, 0(sp)         # guarda a0 = f(t0) na pilha
lw a0, 4(sp)         # recupera t0 original da pilha
jal g                # para usar como argumento na

lw t0, 0(sp)         # recupera resultado de f(t0)
add t1, t0, a0       # t1 = f(t0) + g(t0)
addi sp, sp, 8       # liberta espaco na pilha
```

Convenção de utilização dos registos

A utilização dos registos é especificada na ABI.

Relativamente às funções, os registos classificam-se em:

- Não preservados podem ser modificados livremente nas funções.
  `t0-t6, a0-a7, ra`.

Convenção de utilização dos registos

A utilização dos registos é especificada na ABI.

Relativamente às funções, os registos classificam-se em:

- Não preservados podem ser modificados livremente nas funções.
  `t0-t6, a0-a7, ra`.
- Preservados podem ser modificados, mas o valor original tem de ser reposto antes da função retornar.
  `s0-s11, sp`.

Convenção de utilização dos registos

A utilização dos registos é especificada na ABI.

Relativamente às funções, os registos classificam-se em:

- Não preservados podem ser modificados livremente nas funções.
  `t0-t6, a0-a7, ra`.
- Preservados podem ser modificados, mas o valor original tem de ser reposto antes da função retornar.
  `s0-s11, sp`.

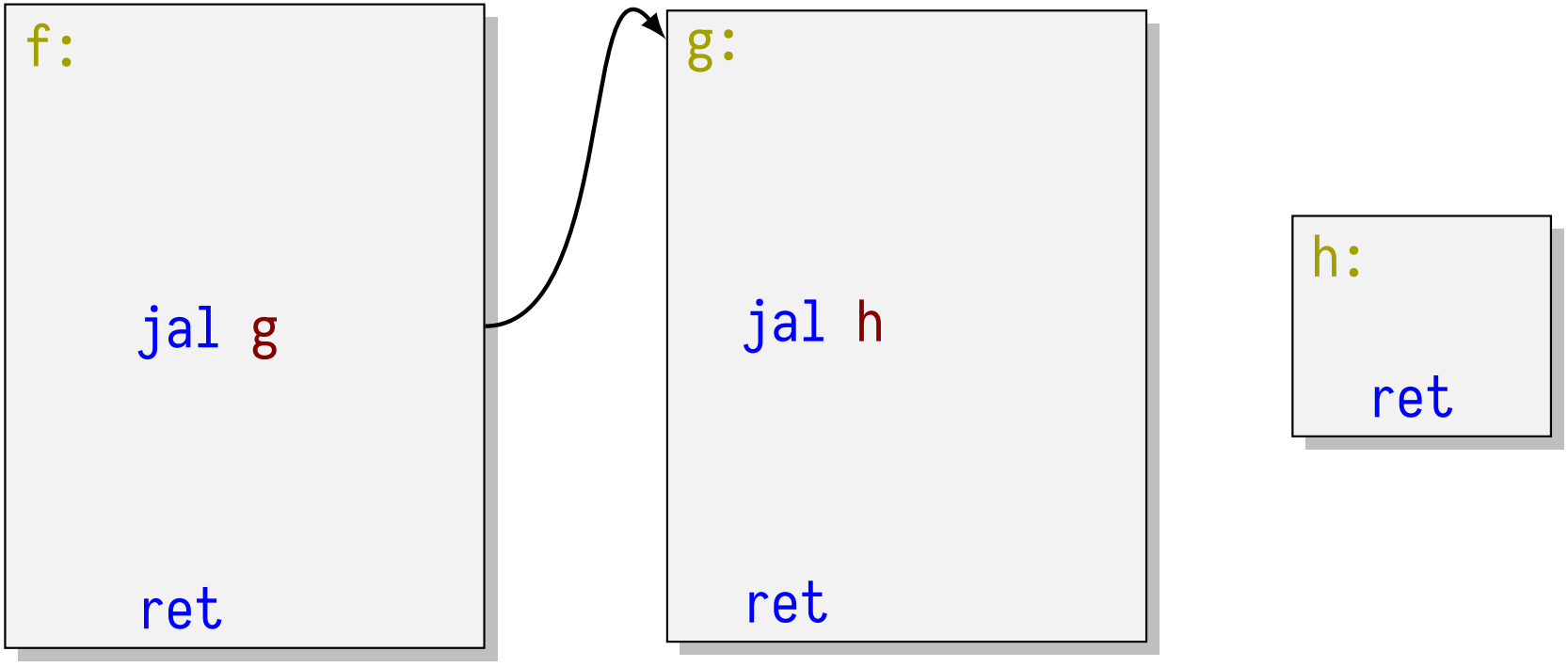
```
li s0, 123
li t0, 456

jal f     # f pode modificar t0 e s0 internamente

# s0 = 123,  f repõe o valor original antes de sair
# t0 = LIXO, f pode deixar o registo modificado
```

Exemplo: uma função que chama outra função

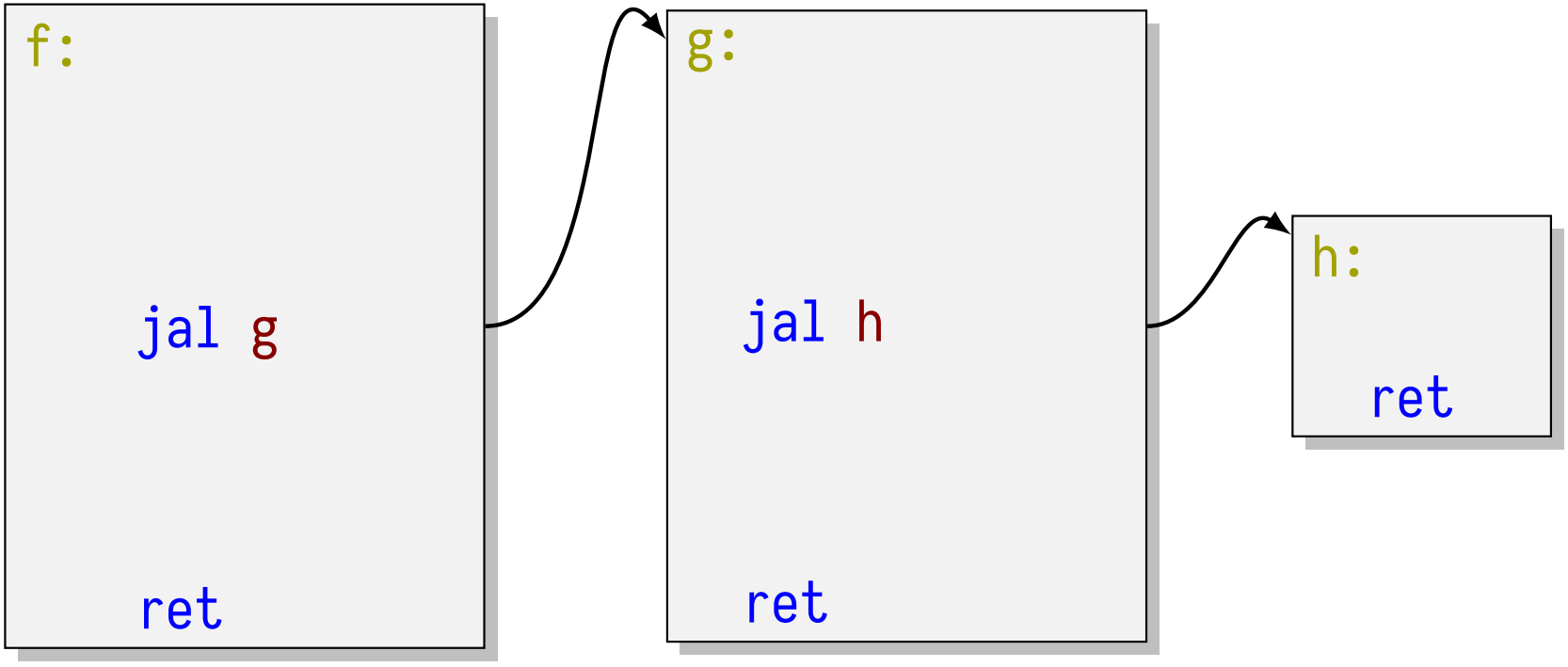
```c
int f(int x) {
    return g(x) + x;
}
```

```
f:  addi sp, sp, -4    # aloca espaco na pilha
    sw s0, 0(sp)       # temos de preservar s0

    mv s0, a0          # guardar a0 para usar depois
    jal g              # modifica regs nao preservados
    add a0, a0, s0     # a0 = g(x) + x

    lw s0, 0(sp)       # repor s0 (registo preservado)
    addi sp, sp, 4     # liberta espaco da pilha
    ret                # retorna da funcao???
```

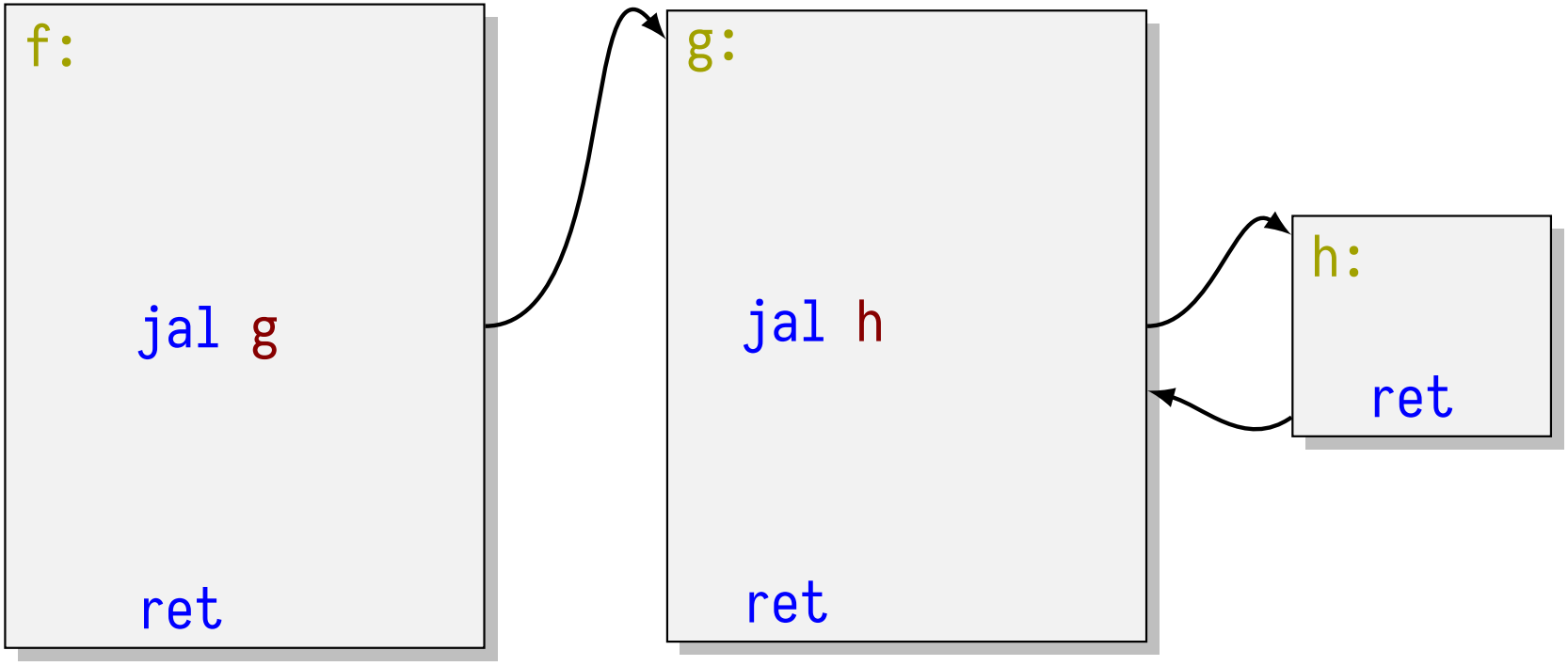
Funções que chamam funções



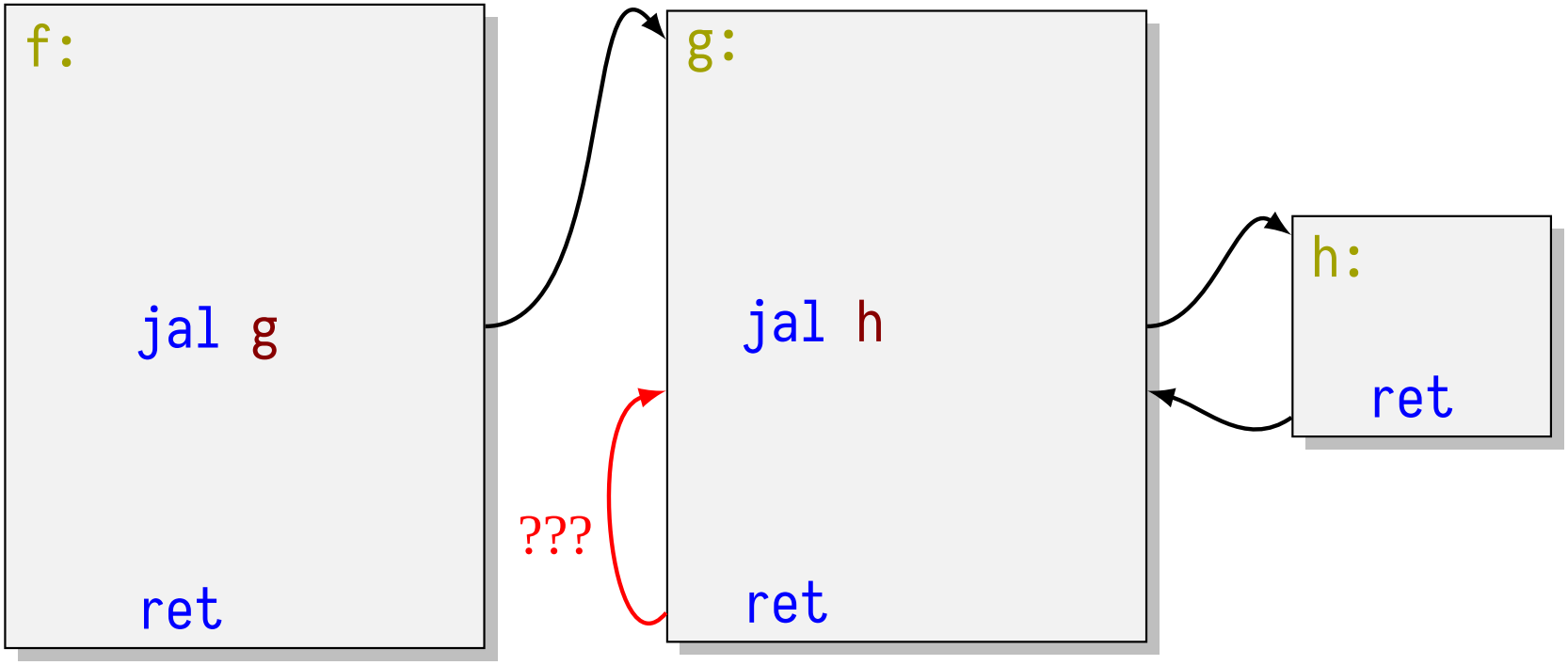
Funções que chamam funções



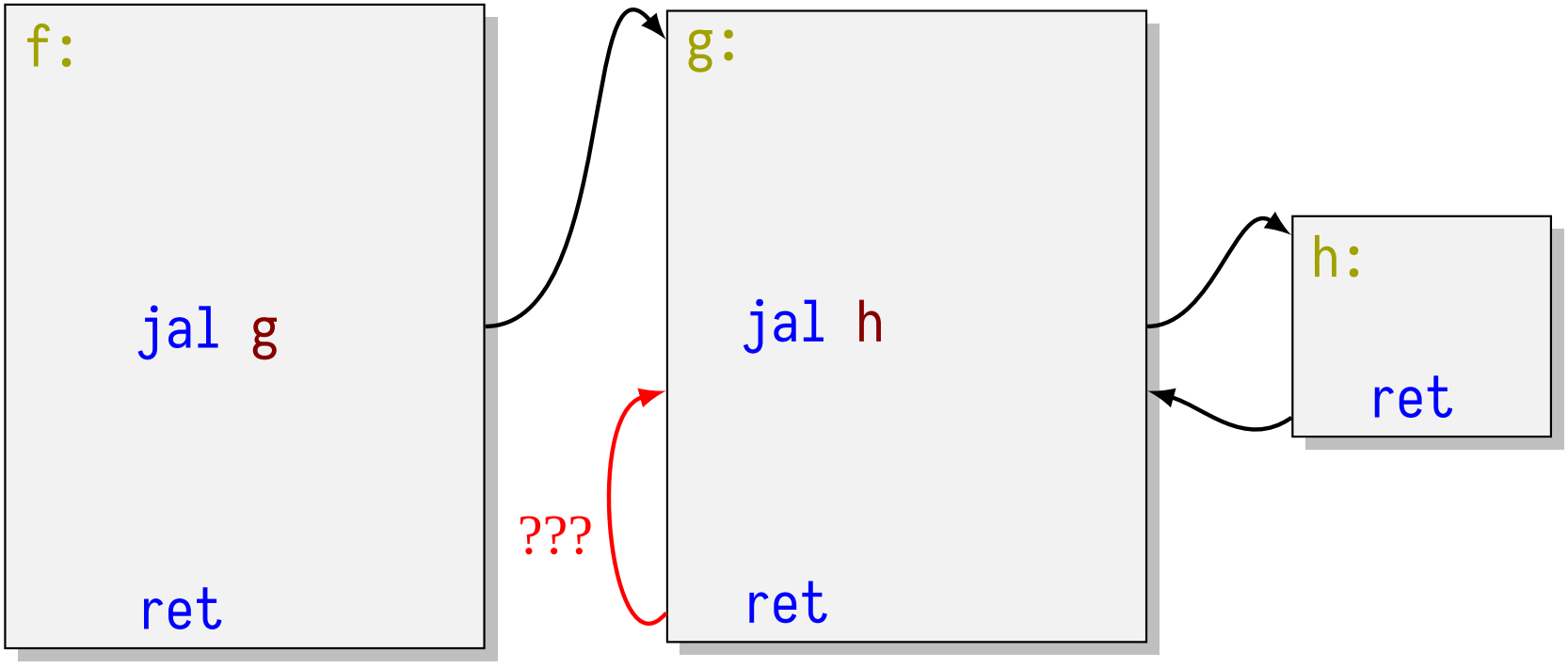
Funções que chamam funções



Funções que chamam funções

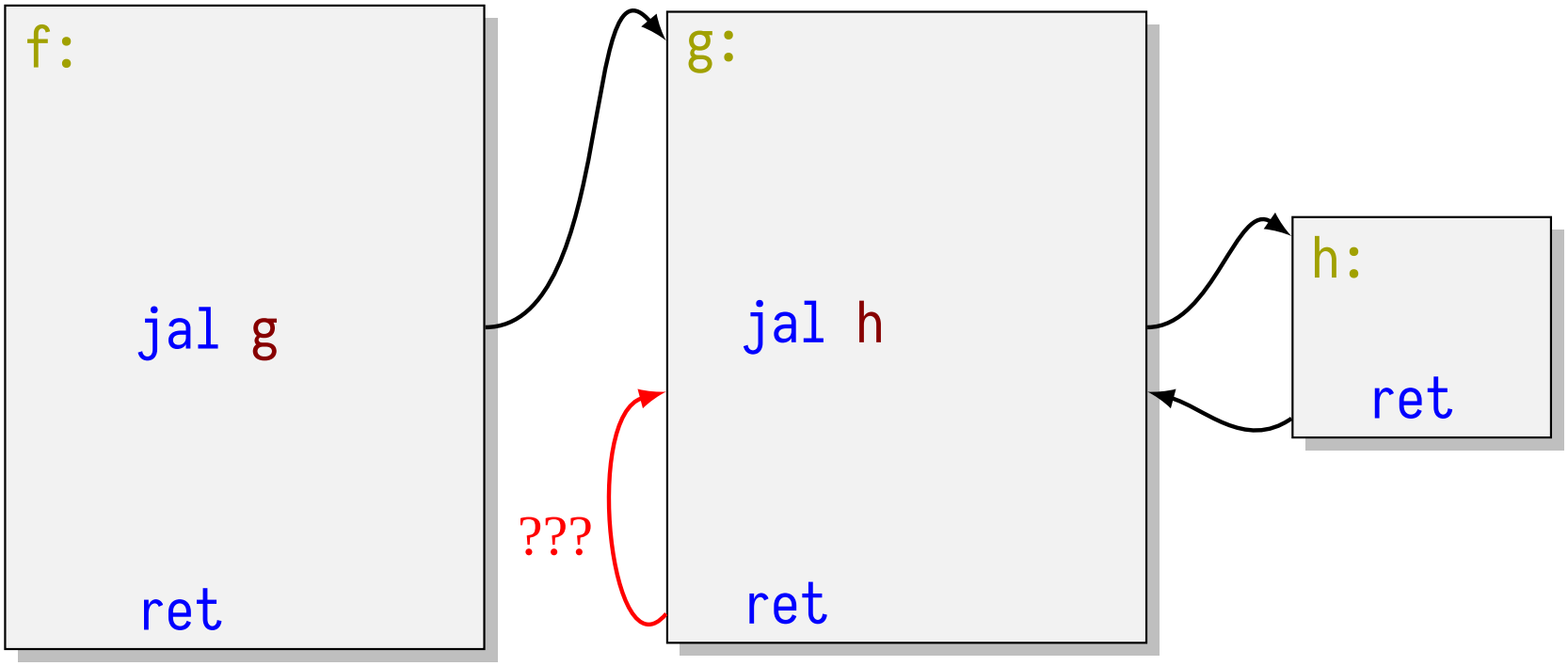


Funções que chamam funções



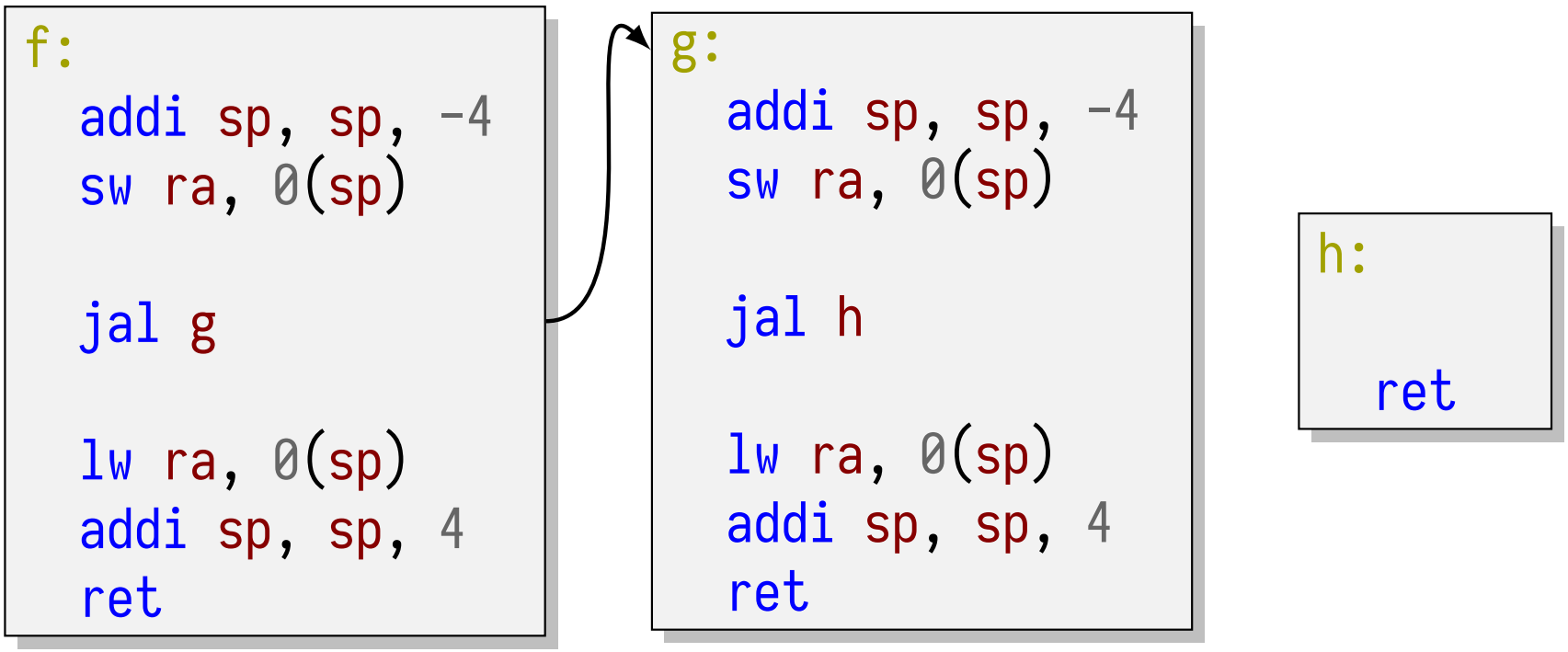
Como se resolve?

Funções que chamam funções

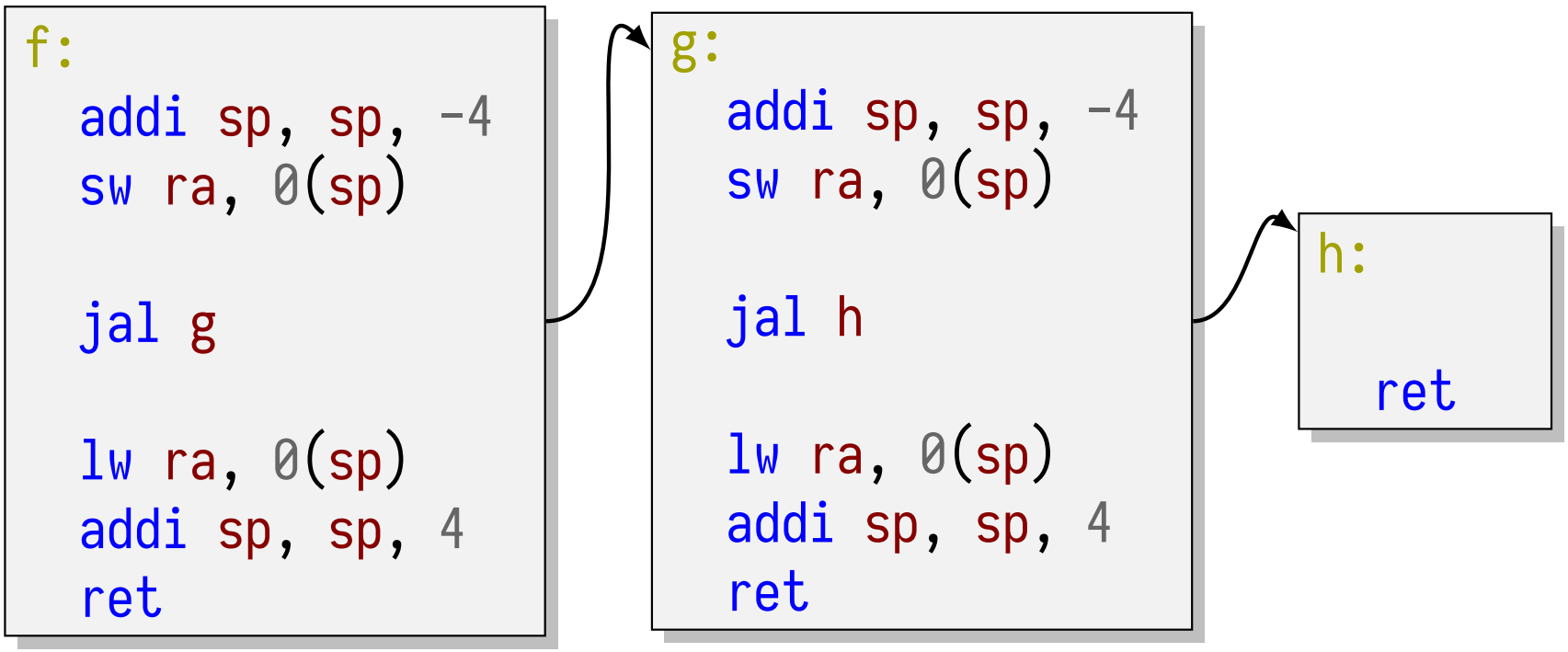


Como se resolve? As funções `f` e `g` têm de guardar os respectivos *return addresses* na pilha antes de chamar a função. Assim, podem retornar para o endereço correcto.

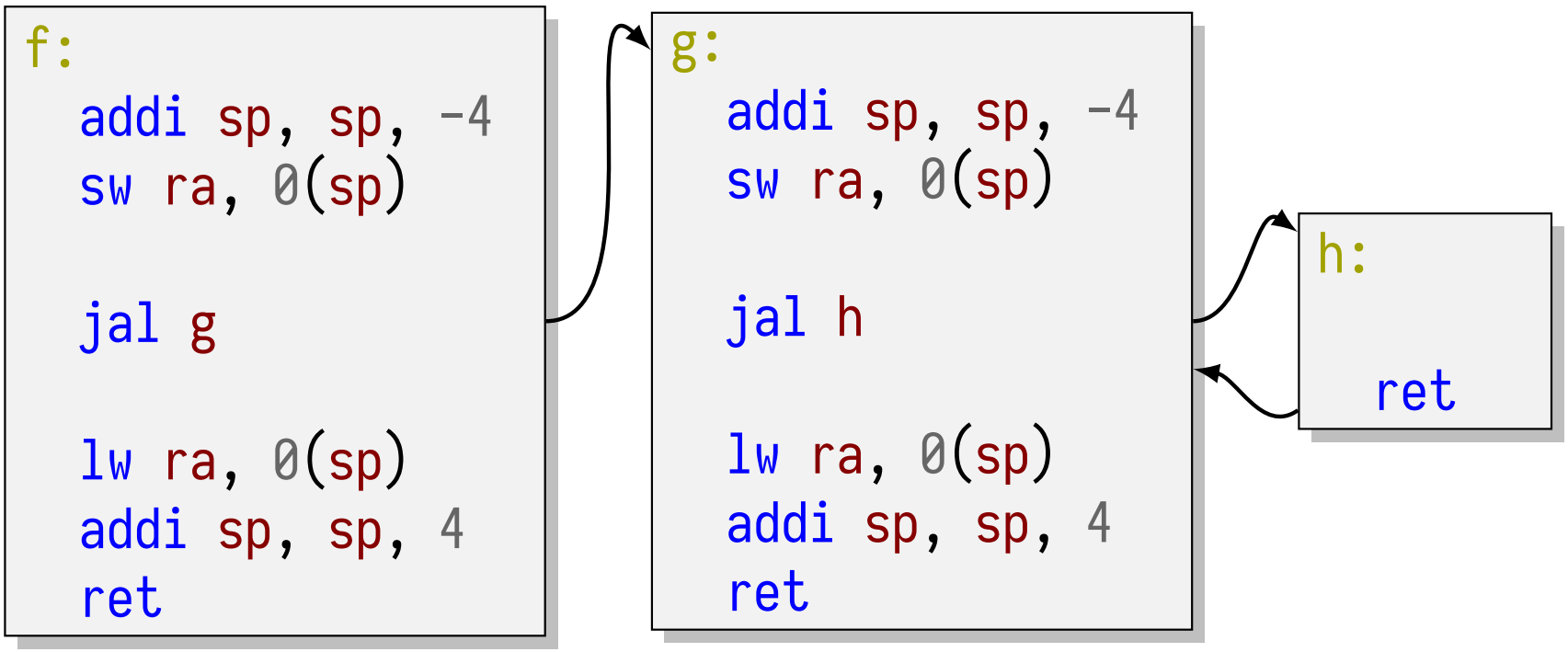
Funções que chamam funções



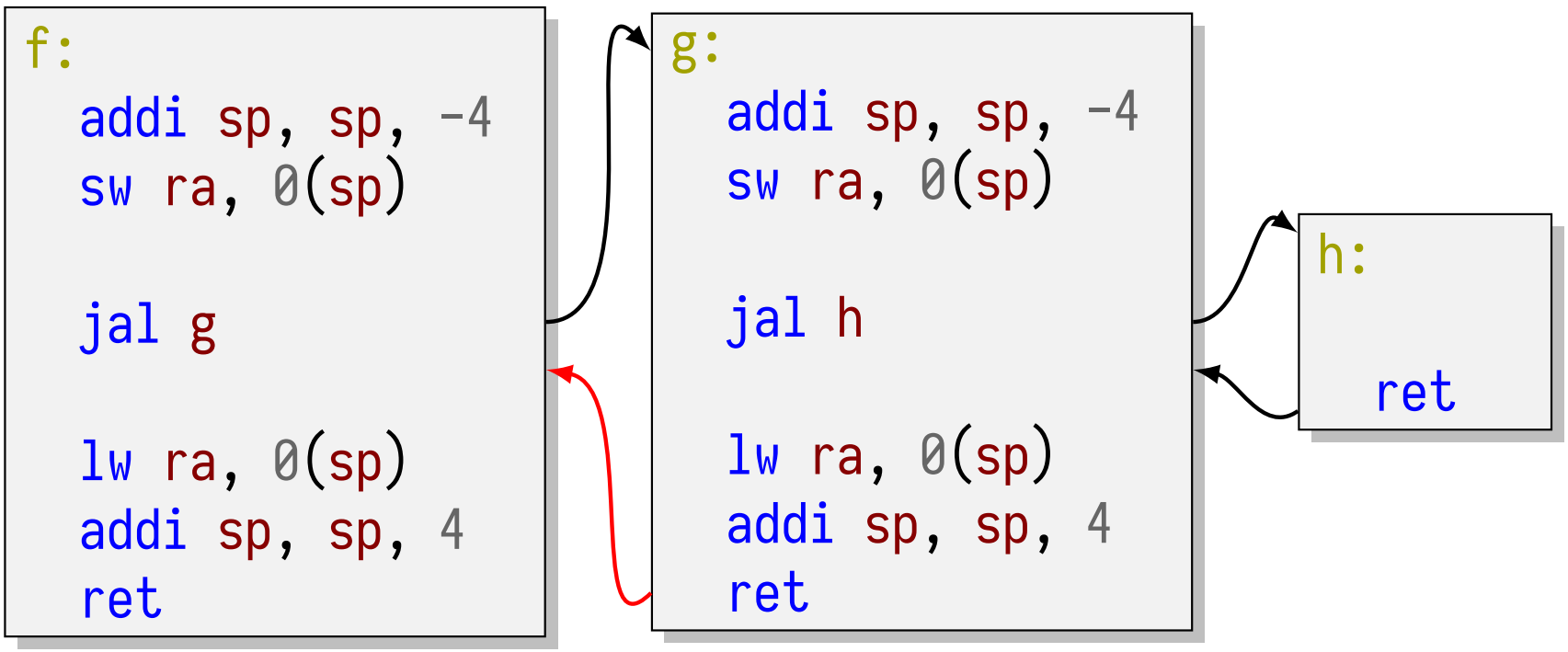
Funções que chamam funções



Funções que chamam funções



Funções que chamam funções



Funções recursivas

Funções recursivas

Definição (Recursividade)

É um método em que a solução para um problema envolve a solução de instâncias mais pequenas do mesmo problema.
Normalmente, a recursividade é implementada por uma função que se chama a ela própria.

```
int arr[] = {5, 0, -4, 12, 9, 2, 77, 54, 66, 82, -87};


int sum(int a[], int sz) {
    if (sz == 0)
        return 0;
    else
        return a[sz-1] + sum(a, sz-1);
}
```

Função recursiva

O que faz a seguinte função?

```
int xpto(int n) {
  if (n == 0)
    return 1;
  else
    return 2 * xpto(n - 1);
}
```

Função recursiva

O que faz a seguinte função? Calcula 2^n.

```
int xpto(int n) {
  if (n == 0)
    return 1;
  else
    return 2 * xpto(n - 1);
}
```

Função recursiva

```c
int xpto(int n) {
  if (n == 0)
    return 1;
  else
    return 2 * xpto(n - 1);
}
```

```asm
xpto:


        mv   t0, a0
# if
        li   a0, 1
        beq  t0, zero, RET
# else
        addi a0, t0, -1
        jal  xpto
        slli a0, a0, 1

RET:

        ret
```

Função recursiva

```c
int xpto(int n) {
  if (n == 0)
    return 1;
  else
    return 2 * xpto(n - 1);
}
```

Não funciona!
Ciclo infinito no `slli/ret` $\longrightarrow$

```
xpto:


        mv   t0, a0
# if
        li   a0, 1
        beq  t0, zero, RET
# else
        addi a0, t0, -1
        jal  xpto
        slli a0, a0, 1

RET:

        ret
```

Função recursiva - corrigida

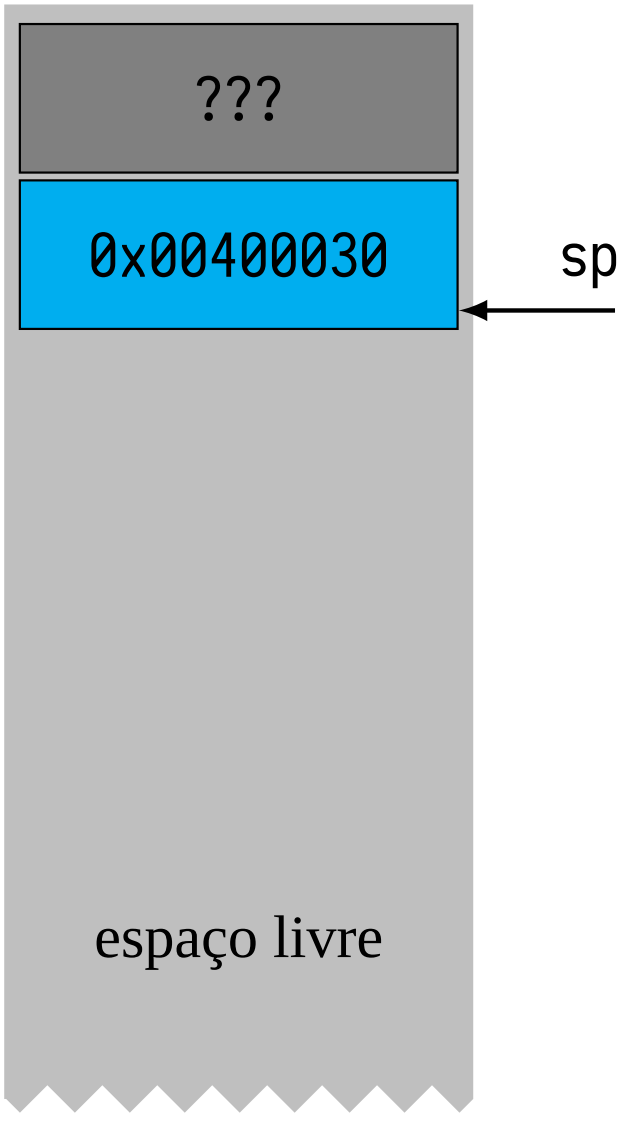
```c
int xpto(int n) {
  if (n == 0)
    return 1;
  else
    return 2 * xpto(n - 1);
}
```

```
xpto:
        addi sp, sp, -4
        sw   ra, 0(sp)
        mv   t0, a0
# if
        li   a0, 1
        beq  t0, zero, RET
# else
        addi a0, t0, -1
        jal  xpto
        slli a0, a0, 1

RET:    lw   ra, 0(sp)
        addi sp, sp, 4
        ret
```

Exemplo: calcular xpto(3)

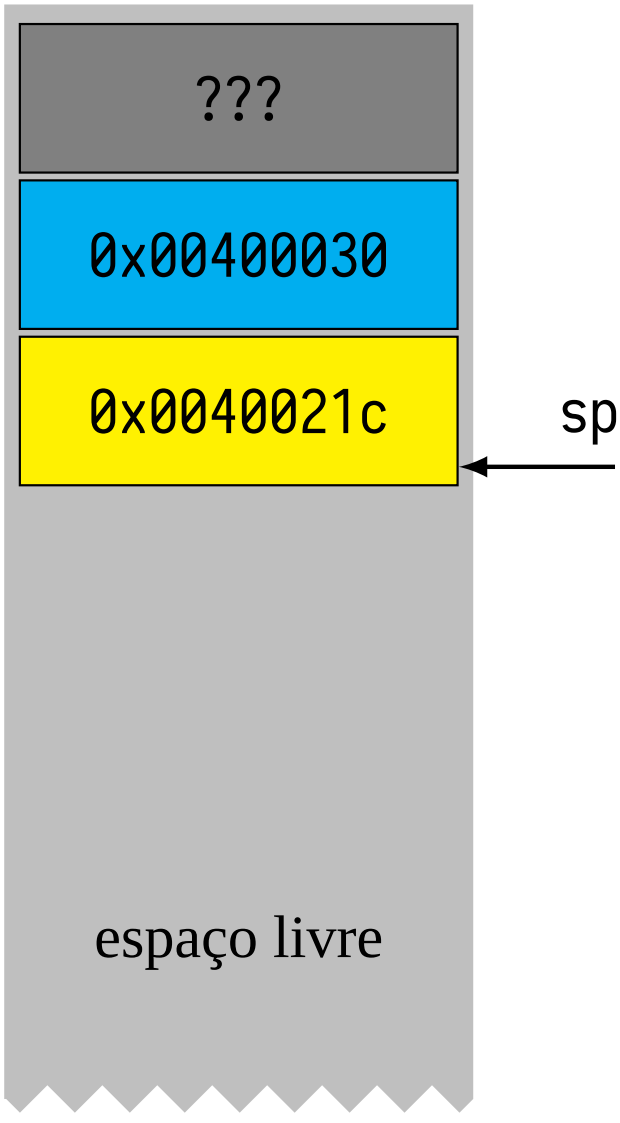

```
xpto:                   # 0x00400200
        addi sp, sp, -4
        sw   ra, 0(sp)
        mv   t0, a0
# if
        li   a0, 1
        beq  t0, zero, RETURN
# else
        addi a0, t0, -1
        jal  xpto
        slli a0, a0, 1    # +0x1c

RETURN: lw   ra, 0(sp)
        addi sp, sp, 4
        ret
```

Exemplo: calcular xpto(3)



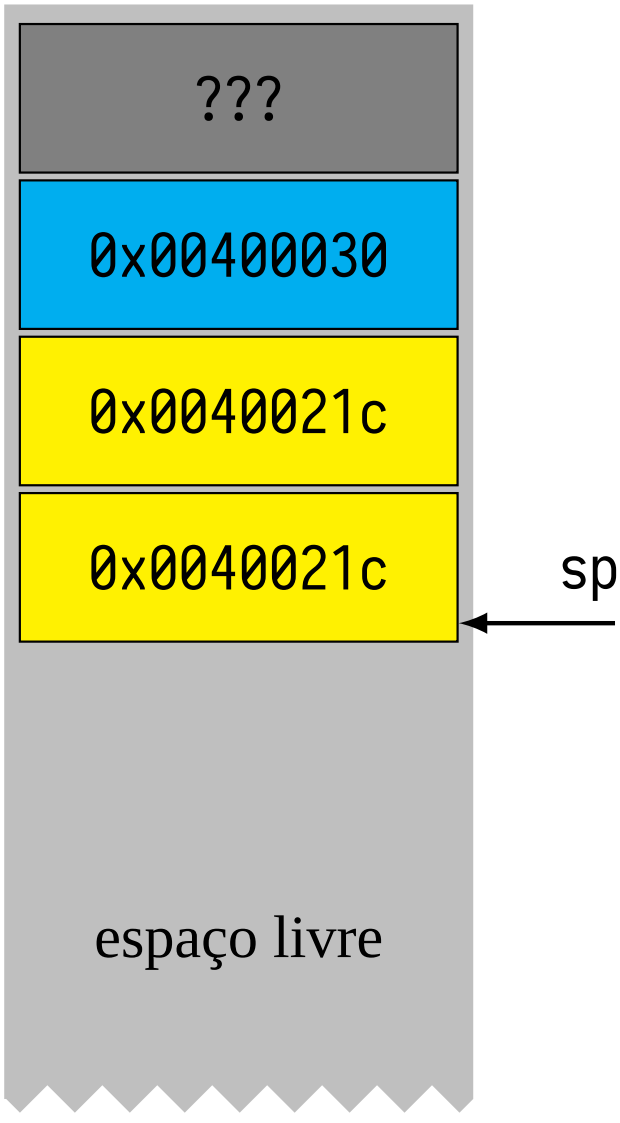

```
xpto:                   # 0x00400200
        addi sp, sp, -4
        sw   ra, 0(sp)
        mv   t0, a0
# if
        li   a0, 1
        beq  t0, zero, RETURN
# else
        addi a0, t0, -1
        jal  xpto
        slli a0, a0, 1    # +0x1c

RETURN: lw   ra, 0(sp)
        addi sp, sp, 4
        ret
```

Exemplo: calcular xpto(3)



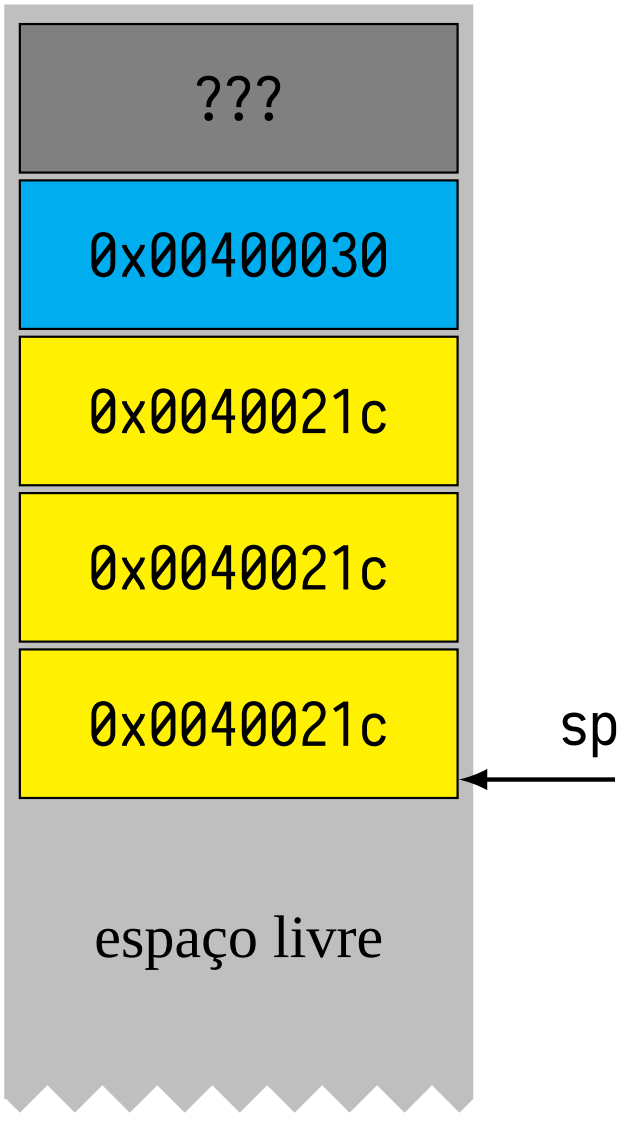

```
xpto:                   # 0x00400200
        addi sp, sp, -4
        sw   ra, 0(sp)
        mv   t0, a0
# if
        li   a0, 1
        beq  t0, zero, RETURN
# else
        addi a0, t0, -1
        jal  xpto
        slli a0, a0, 1    # +0x1c

RETURN: lw   ra, 0(sp)
        addi sp, sp, 4
        ret
```

Exemplo: calcular xpto(3)




```
xpto:                   # 0x00400200
        addi sp, sp, -4
        sw   ra, 0(sp)
        mv   t0, a0
# if
        li   a0, 1
        beq  t0, zero, RETURN
# else
        addi a0, t0, -1
        jal  xpto
        slli a0, a0, 1    # +0x1c

RETURN: lw   ra, 0(sp)
        addi sp, sp, 4
        ret
```

Exemplo: calcular xpto(3)




```
xpto:                   # 0x00400200
        addi sp, sp, -4
        sw   ra, 0(sp)
        mv   t0, a0
# if
        li   a0, 1
        beq  t0, zero, RETURN
# else
        addi a0, t0, -1
        jal  xpto
        slli a0, a0, 1    # +0x1c

RETURN: lw   ra, 0(sp)
        addi sp, sp, 4
        ret
```

Exemplo: soma(n)

```c
int soma(int n) {
  if (n == 0)
    return 0;
  else
    return n + soma(n - 1);
}
```

```asm
soma:
        addi sp, sp, -4
        sw   ra, 0(sp)
        mv   t0, a0
# if
        li   a0, 0
        beq  t0, zero, RET
# else
        addi a0, t0, -1
        jal  soma     # t0?
        add  a0, t0, a0
RET:    lw   ra, 0(sp)
        addi sp, sp, 4
        ret
```

Exemplo: soma(n)

```c
int soma(int n) {
  if (n == 0)
    return 0;
  else
    return n + soma(n - 1);
}
```

Não funciona!
Registo `t0` não preservado $\longrightarrow$

```
soma:
        addi sp, sp, -4
        sw   ra, 0(sp)
        mv   t0, a0
# if
        li   a0, 0
        beq  t0, zero, RET
# else
        addi a0, t0, -1
        jal  soma     # t0?
        add  a0, t0, a0
RET:    lw   ra, 0(sp)
        addi sp, sp, 4
        ret
```

Exemplo: soma(n) - correcção 1

```c
int soma(int n) {
  if (n == 0)
    return 0;
  else
    return n + soma(n - 1);
}
```

```
soma:
        addi sp, sp, -4
        sw   ra, 0(sp)
        mv   s0, a0
# if
        li   a0, 0
        beq  s0, zero, RET
# else
        addi a0, s0, -1
        jal  soma
        add  a0, s0, a0
RET:    lw   ra, 0(sp)
        addi sp, sp, 4
        ret
```

Exemplo: soma(n) - correcção 1

```c
int soma(int n) {
  if (n == 0)
    return 0;
  else
    return n + soma(n - 1);
}
```

Não funciona!
Registo `s0` devia ser preservado → mas não está guardado na pilha…

```
soma:
        addi sp, sp, -4
        sw   ra, 0(sp)
        mv   s0, a0
# if
        li   a0, 0
        beq  s0, zero, RET
# else
        addi a0, s0, -1
        jal  soma
        add  a0, s0, a0
RET:    lw   ra, 0(sp)
        addi sp, sp, 4
        ret
```

Exemplo: soma(n) - correcção 2

```c
int soma(int n) {
  if (n == 0)
    return 0;
  else
    return n + soma(n - 1);
}
```

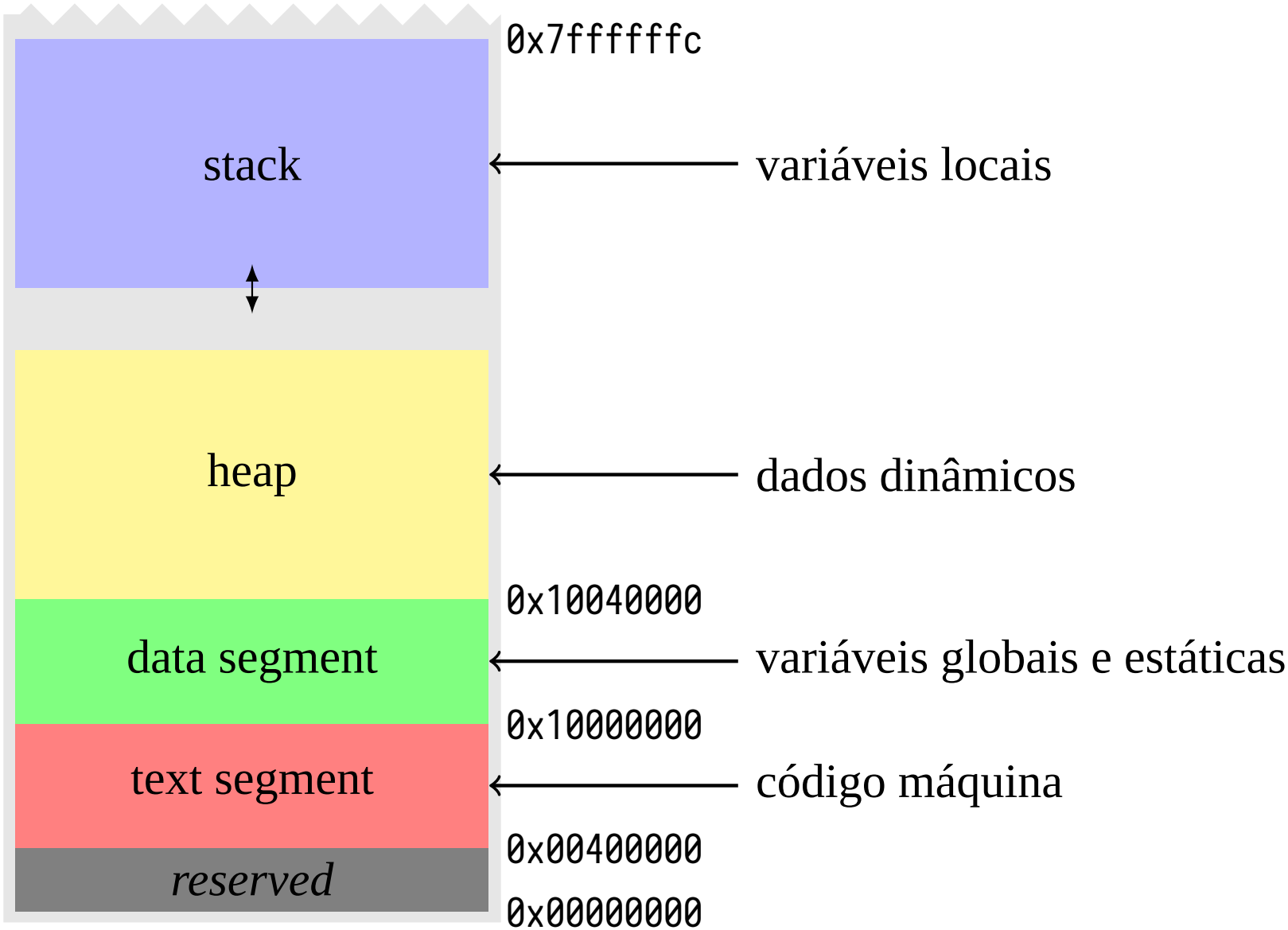
```
soma:
        addi sp, sp, -8
        sw   ra, 4(sp)
        sw   s0, 0(sp)
        mv   s0, a0
# if
        li   a0, 0
        beq  s0, zero, RET
# else
        addi a0, s0, -1
        jal  soma
        add  a0, s0, a0
RET:    lw   s0, 0(sp)
        lw   ra, 4(sp)
        addi sp, sp, 8
        ret
```

Espaço de endereçamento

Variáveis locais, globais e dinâmicas

```c
int n = 20;                                    /* global */

int *make_vec() {
  int *s;                                      /* local */
  s = (int *)malloc(n * sizeof(int)); /* dynamic */
  return s;
}

int main() {
    int *vec;                                  /* local */
    vec = make_vec();
    do_something_with(vec);
    free(vec);
    return 0;
}
```

Espaço de endereçamento



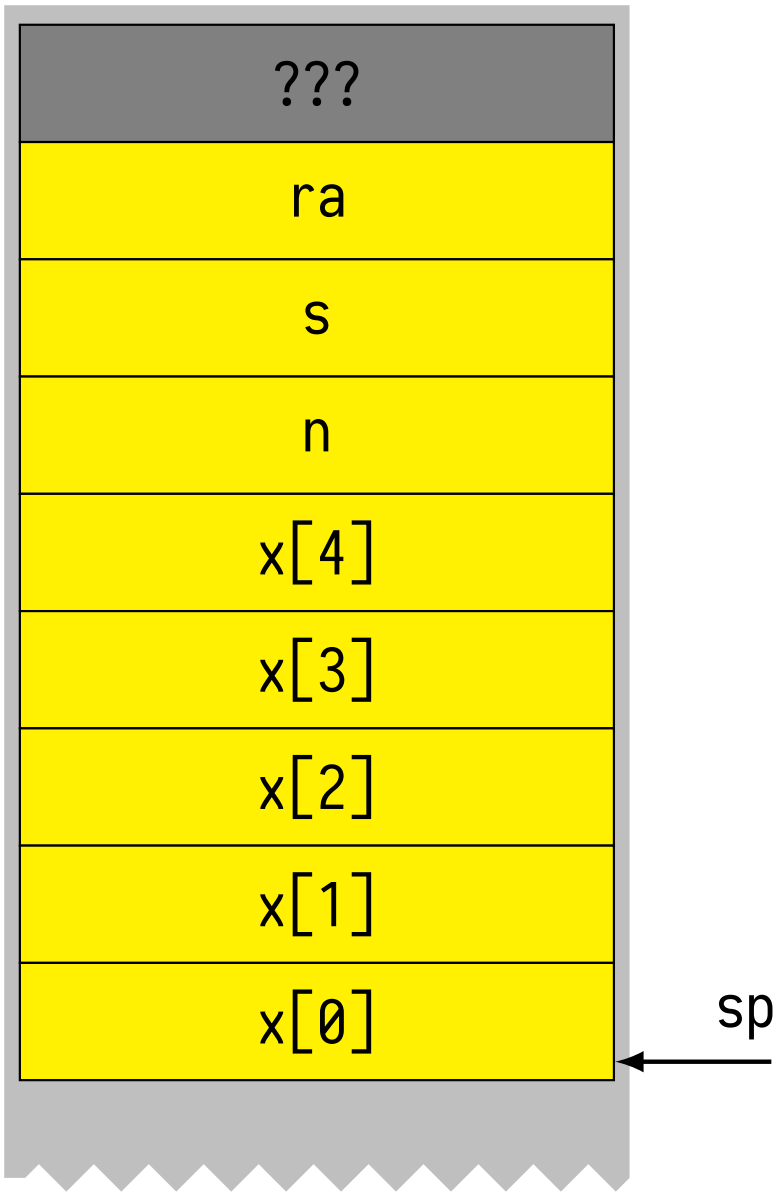
Varáveis locais

Em princípio, as variáveis locais são alocadas na pilha.

Mas um compilador pode optimizar certas variáveis para usar registos, ou o *immediate* das instruções caso a variável nunca seja modificada.

```
int my_function() {
  int n = 2;           /* unmodified, can be optimized */
  int x;               /* needs address, so use stack  */
  char s[] = "Ana";    /* copies string to stack       */
                       /* s unmodified can be optimized*/

  some_func(&x, s);
  return n;
}
```

Variáveis locais

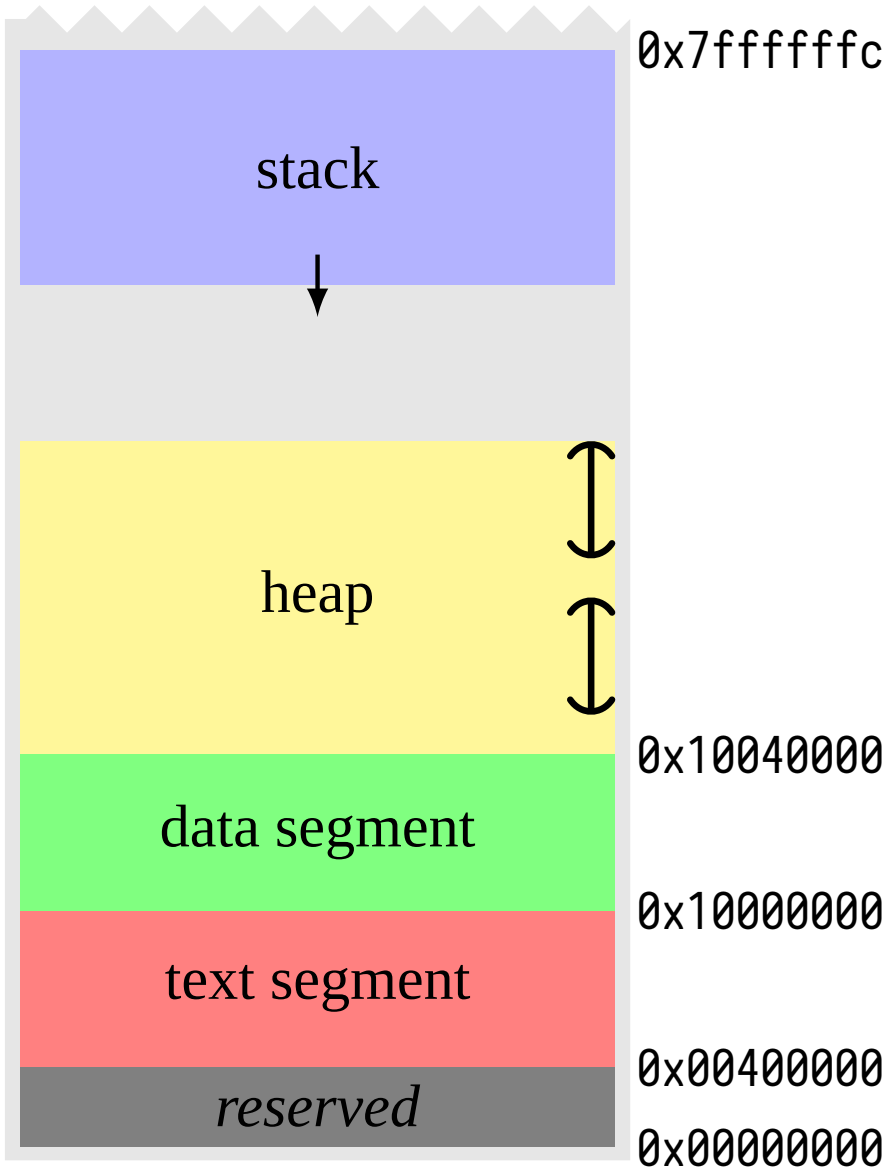



```c
int f(int a) {
  char *s;
  int n = 0;
  int x[5];
  /* ... */
}
```

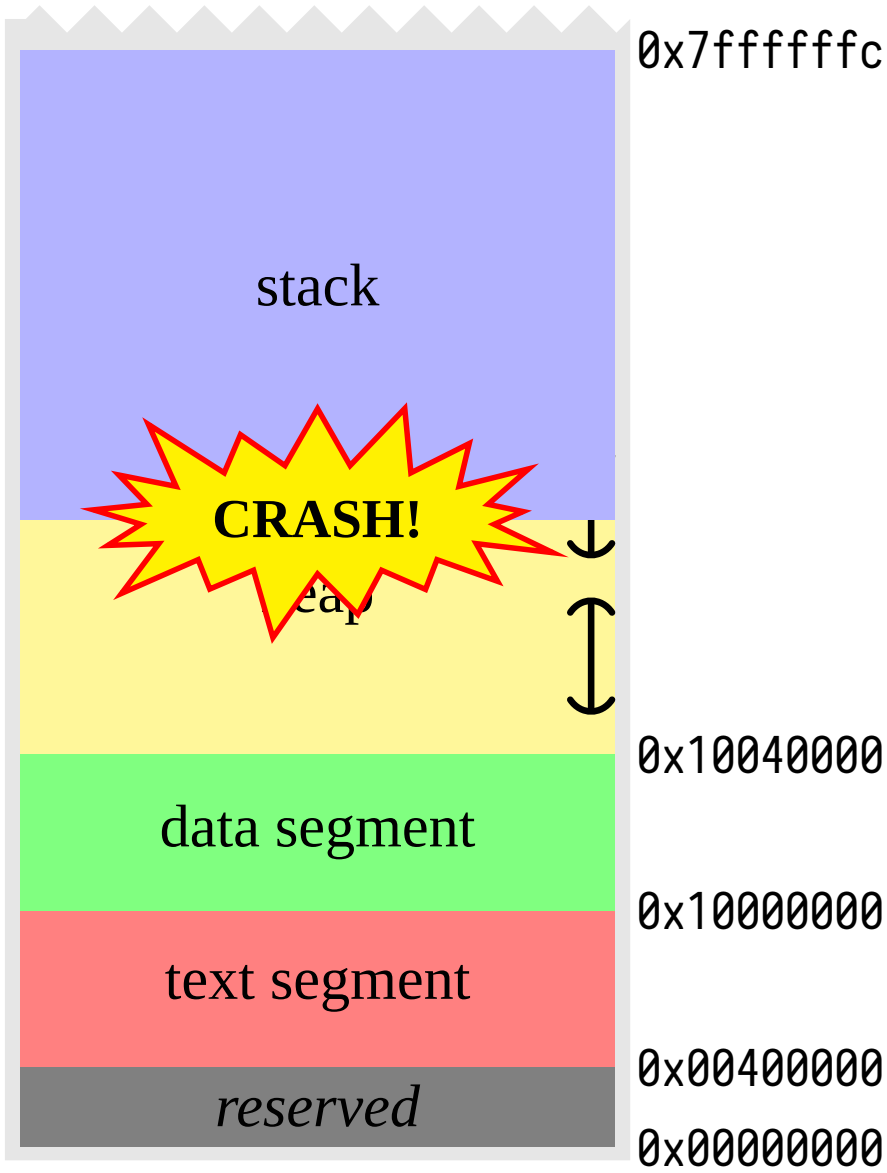
Para obter os valores:

x[2]	`lw t0, 8(sp)`
n	`lw t0, 20(sp)`
s	`lw t0, 24(sp)`

Espaço de endereçamento



Espaço de endereçamento



Data segment

O *data segment* contém as variáveis que necessitam de espaço alocado durante toda a execução do programa.

```
        .data
altura: .word 174

        .text
        lw t0, altura  # obter valor (pseudoinstrucao)
```

a pseudoinstrução é convertida pelo assembler em 2 instruções:

```
lui t0, %hi(altura)  # 20 bits mais signif.
lw  t0, %lo(t0)      # 12 bits menos signif.
```

onde `%hi()` e `%lo()` calculam os 20 e 12 bits mais e menos significativos a usar, respectivamente.

Global pointer

O acesso ao *data segment* requer 2 instruções, o que é ineficiente. O *global pointer* permite aceder com apenas uma instrução:

```
lw t0, -2048(gp)
```

- O registo gp contém um endereço fixo.
- O endereçamento é relativo ao gp.
- Permite acesso a uma região de 12KiB.
- Usada para dados pequenos e frequentes (words, endereços).
- Os restantes dados ficam em endereços mais altos e o acesso requer 2 instruções.

Dados dinâmicos

- Em C, os dados dinâmicos são geridos pelas funções `malloc()` e `free()`. Internamente, estas funções gerem um espaço de memória (*heap*) alocando e libertando parcelas de memória.
- A função `malloc` usava a chamada de sistema `sbrk` para aumentar a quantidade de memória alocada para o *data segment*. Este processo está descontinuado nos sistemas mais recentes, mas o RARS continua a simular esta chamada de sistema.
- Não existe a função `malloc` no simulador RARS (teriamos de implementar). Em vez disso fazemos a chamada de sistema `sbrk`, mas não é bem a mesma coisa…

```
# allocate 1024 bytes
li a7, 9      # system call service 9 (sbrk)
li a0, 1024   # number of bytes to allocate
ecall
# a0 = address of the new allocated memory
```