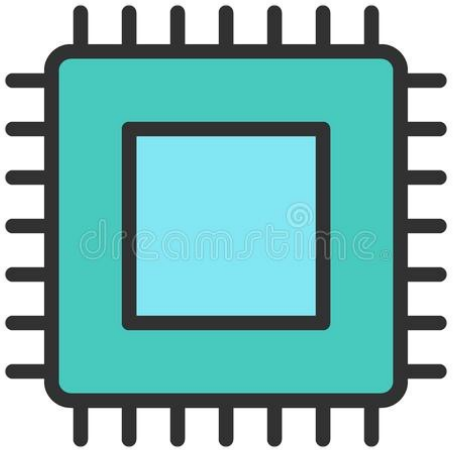
Programação em Assembly: aspetos avançados



TÉCNICO LISBOA

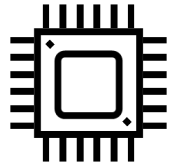
Descubra o bug!

```
valor           EQU   76H       ; Valor cujo número de bits a 1 é para ser contado
mascaraInicial  EQU   01H       ; 0000 0001 em binário (máscara inicial)
mascaraFinal    EQU   80H       ; 1000 0000 em binário (máscara final)

; Utilização dos registos:
; R0 – auxiliar (valores intermédios)
; R1 – contador de bits a 1
; R2 – máscara

inicio:     MOV   R1, 0             ; Inicializa o contador de bits com zero
            MOV   R2, mascaraInicial ; Inicializa valor da máscara
teste:      MOV   R0, valor         ; Cópia do valor
~~teste:~~  AND   R0, R2            ; Isola o bit que se quer ver se é 1
            JZ    proximo           ; Se o bit for zero, passa à máscara seguinte
            ADD   R1, 1             ; O bit é 1, incrementa o valor do contador
proximo:    MOV   R0, mascaraFinal
            CMP   R2, R0            ; Compara com a máscara final
            JZ    fim               ; Se forem iguais, já terminou
            SHL   R2, 1             ; Desloca bit da máscara para a esquerda
            JMP   teste             ; Vai fazer mais um teste com a nova máscara
fim:        JMP   fim               ; Fim do programa
```

Conclusão: programar em Assembly é difícil e propenso a error; é muito baixo nível.

Do baixo para o alto para o baixo nível

Correspondência com as linguagens de alto nível (C)



- Em C:

```
a = 2;   // variáveis. O compilador escolhe se as coloca ...
b = a;   // ... em registos ou na memória
```

- Em *assembly*, em **registos**:

```
MOV     R1, 2           ; a = 2 (atribuição)
MOV     R2, R1          ; b = a (atribuição)
```

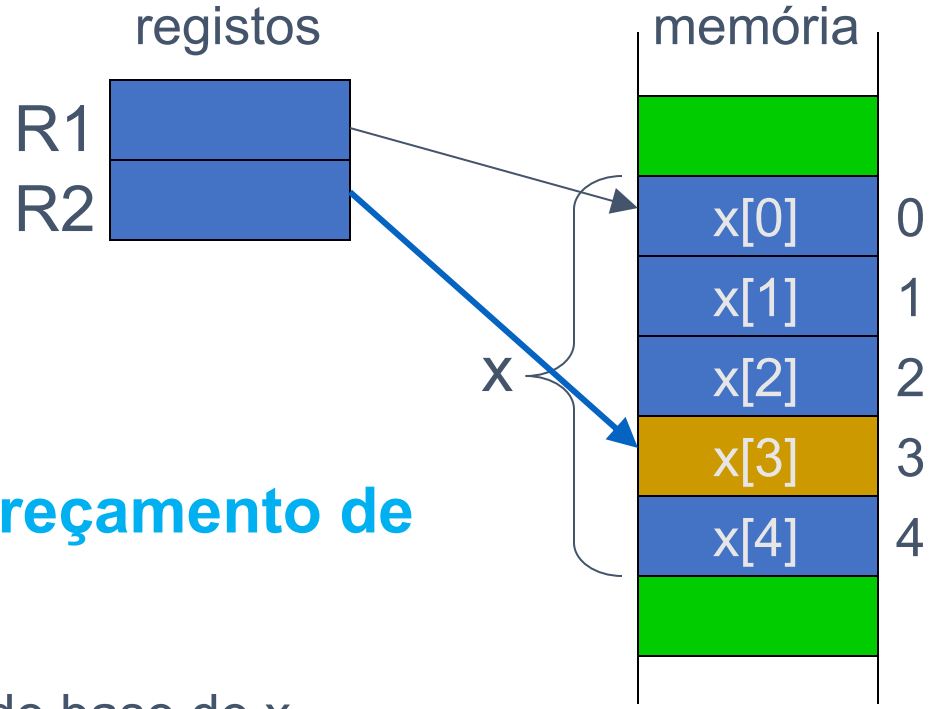
- Em *assembly*, em **memória**:

```
MOV     R1, 2
MOV     [A], R1         ; a = 2 (escrita na memória). A é o endereço da variável a
MOV     R1, [A]         ; lê a da memória para um registo. B é o endereço da variável b
MOV     [B], R1         ; b = a (escrita na memória)
```

Vetores (*arrays*) em assembly

- Em C:

```
int x[5];
x[3] = x[3] + 4;
```



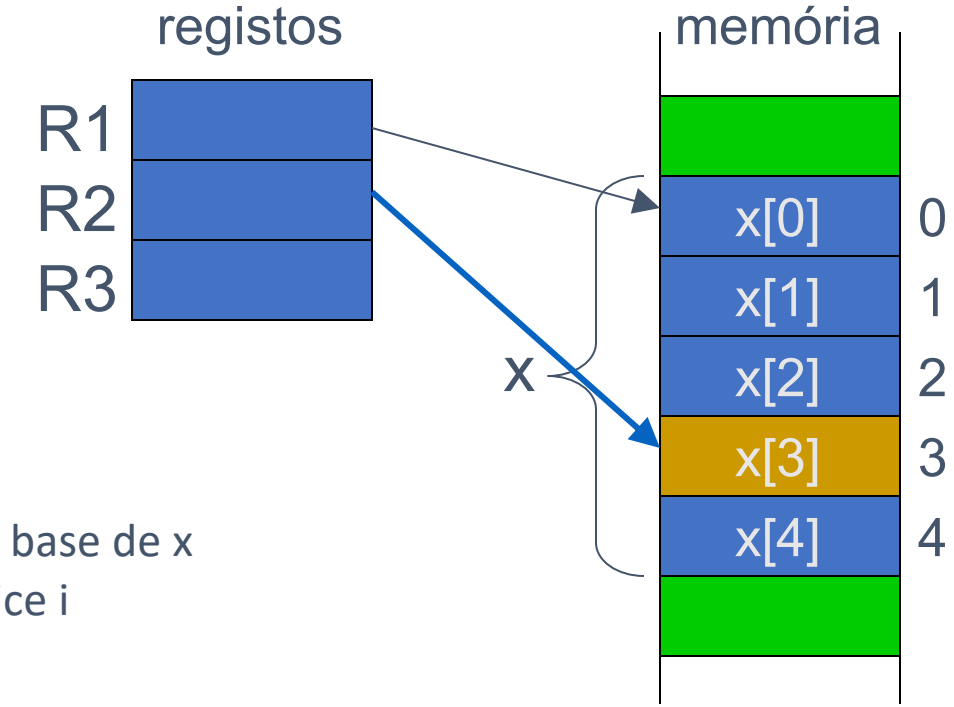
- Em *assembly* (atenção ao **endereçamento de byte**!):

```
MOV     R1, X          ; endereço de base de x
MOV     R2, [R1+6]     ; x[3]
ADD     R2, 4          ; x[3] + 4
MOV     [R1+6], R2     ; x[3] = x[3] + 4
```

Vetores com índice variável

- Em C:

```
int x[5];    int i;
for (i=0; i != 5 ;i++)
    x[i] = x[i] + 4;
```



- Em *assembly*:

```
     MOV    R1, X          ; endereço de base de x
     MOV    R3, 0          ; inicializa índice i
L1:  MOV    R2, [R1+R3]    ; x[i]
     ADD    R2, 4          ; x[i] + 4
     MOV    [R1+R3], R2    ; x[i] = x[i] + 4
     ADD    R3, 2          ; i++ (i+=2 para usar o i como índice)
     MOV    R4, 10         ; 5 * 2 (5 elementos, mas 10 bytes)
     CMP    R3, R4         ; i != 5 (10 em endereço)
     JNZ    L1             ; volta para trás enquanto i != 5
     . . .                 ; instruções a seguir ao for
```

Instrução *if*

- Em C:

  if (*expressão-booleana*) /* 0 ≡ false, != 0 ≡ true */

  { *instruções* }

- Em *assembly*:

```
        expressão       ; calcula expressão (afeta bit de estado Z)
        JZ OUT          ; se expressão booleana for falsa, não executa instruções
        instruções      ; código das instruções dentro do if
OUT:    ...             ; instrução a seguir ao if
```

Instrução *if-else*

- Em C:

  if (*expressão-booleana*)

  { *instruções 1* } else { *instruções 2*}

- Em *assembly*:

```
        expressão       ; calcula expressão (afeta bit de estado Z)
        JZ      ALT     ; se expressão booleana for falsa, salta
        instruções 1    ; código das instruções dentro do if
        JMP     OUT     ; não pode executar instruções 2
ALT:    instruções 2    ; código das instruções da cláusula else
OUT:    ...             ; instrução a seguir ao if
```

Expressões booleanas no *if*

- Para fazer:

  if (*a* < *b*)

    { *instruções* }

- O compilador pode fazer:

```
        CMP     Ra,Rb       ; afeta bit de estado N
        JGE     OUT         ; se a >= b, bit de estado N estará a 0
                            ; ou então: JNN       OUT
        instruções          ; código das instruções dentro do if
OUT:    ...                 ; instrução a seguir ao if
```

- O PEPE tem instruções para suportar os vários casos relacionais possíveis (<, <=, >, >=, =, <>)

Ciclos (iteração)

- Destinam-se a executar um bloco de código **um certo número de vezes**.
- Fixo, ou incondicional (*for*)

```
for (i=0; i < N; i++;)
  { instruções }
```

- Condicional (*while* e *do-while*)

```
while (expressão)
  { instruções }
```

```
do
   { instruções }
while (expressão);
```

Ciclos incondicionais (*for*)

- Em C:

```
for (i=0; i < N; i++)
  { instruções }
```

- Em *assembly* (assumindo que *i* está no registo R1):

```
        MOV     R1, 0     ; inicializa variável de índice (i = 0;)
LOOP:   MOV     R2, N
        CMP     R1, R2    ; i < N?
        JGE     OUT       ; se i >= N, já terminou e vai embora
        instruções        ; código das instruções dentro do for
        ADD     R1, 1     ; i++
        JMP     LOOP      ; próxima iteração
OUT:    ...               ; instrução a seguir ao for
```

Ciclos condicionais (*while*)

- Em C:

  while (*expressão*)

    { *instruções* }

- Em *assembly*:

```
LOOP:   expressão       ; código para calcular a expressão
        JZ      OUT     ; se expressão for falsa, sai do ciclo
        instruções      ; código das instruções dentro do while
        JMP     LOOP    ; próxima iteração (avalia expressão de novo)
OUT:    ...             ; instrução a seguir ao while
```

Ciclos condicionais (*do-while*)

- Em C:

```
do
   { instruções }
while (expressão);
```

- Em *assembly*:

```
LOOP:   instruções              ; instruções dentro do do-while
        expressão               ; instruções para calcular expressão
        JNZ         LOOP        ; se expressão for verdadeira, continua no ciclo
OUT:    ...                     ; instrução a seguir ao do-while
```

Bubblesort

Custo da abstração vetor/array.

PROGRAMA EM C

```
#define  N  4

int seq [N] = {10, 5, 6, 2};



main () {
   int houveTroca ;
   int i;
   int auxiliar;


   do
      houveTroca = false;
      i = 0;

      while (i < N-1)
      {


         if (seq[i] > seq[i+1])
         {









            auxiliar = seq[i];
            seq[i] = seq[i+1];
            seq[i+1] = auxiliar;
            houveTroca = true;
         }
         i = i + 1;
      }
   while (houveTroca);

}
```

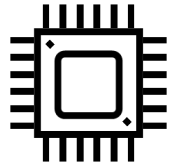
PROGRAMA EM *ASSEMBLY*

```
N      EQU   4  ; #define  N  4
PLACE 1000H     ; localiza bloco de dados
seq:  WORD  10 ; int seq [N] = {10, 5, 6, 2};
      WORD  5
      WORD  6
      WORD  2
; R0 - base do vetor seq
; R1 - variável houveTroca
; R2 - variável i
; R3 - variável auxiliar
PLACE 0000H       ; localiza bloco de código
main: MOV   R0, seq  ; base do vetor
ronda:
      MOV   R1, 0    ; houveTroca = false;
      MOV   R2, 0    ; i = 0;
iteração:
      MOV   R7, N    ; N
      SUB   R7, 1    ; N-1
      CMP   R2, R7   ; i < N-1 ?
      JGE   teste    ; se não, acabou o ciclo
      MOV   R9, R2       ; obtém cópia de i
      SHL   R9, 1        ; 2*i (ender. em bytes)
      MOV   R7, [R0+R9] ; seq[i]
      MOV   R10, R2      ; obtém cópia de i
      ADD   R10, 1       ; i+1
      SHL   R10, 1       ; 2*(i+1)
      MOV   R8, [R0+R10]   ; seq[i+1]
      CMP   R7, R8       ; seq[i] > seq[i+1] ?
      JLE   próximo      ; não, passa ao próximo
      MOV   R3, R7       ; auxiliar = seq [i];
      MOV   [R0+R9], R8 ; seq[i] = seq[i+1];
      MOV   [R0+R10], R3 ; seq[i+1]=auxiliar;
      MOV   R1, 1        ; houveTroca = true;
próximo:
      ADD   R2, 1        ; i = i + 1;
      JMP   iteração     ; próxima iteração
teste:CMP   R1, 0        ; houveTroca = falso?
      JNZ   ronda        ; mais uma ronda
fim:  JMP   fim          ; acaba aqui
```

TÉCNICO LISBOA

Apontadores

Apontadores



- Usados para **referência a blocos de memória** criados dinamicamente (e.g., **malloc()**)
- Usados para fazer **passagem de parâmetros por referência** a funções

Bubblesort

Usar ponteiros tem custo muito menor.

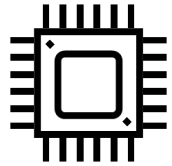
PROGRAMA EM C

```
#define  N  4

int seq [N] = {10, 5, 6, 2};


main () {
   int * num;
   int houveTroca ;
   int i;
   int auxiliar;
   do

      num = seq;
      houveTroca = false;
      i = 0;

      while (i < N-1)
      {


         if (*num > *(num+1))
         {


            auxiliar = *num;
            *num = *(num+1);
            *(num+1) = auxiliar;
            houveTroca=true;
         }
         i = i + 1;
         num = num + 1;
      }
   while (houveTroca);
```

PROGRAMA EM *ASSEMBLY*

```
N      EQU   4  ; #define  N  4
PLACE  1000H    ; localiza bloco de dados
seq:   WORD  10 ; int seq [N] = {10, 5, 6, 2};
       WORD  5
       WORD  6
       WORD  2
; R0 - variável num (apontador)
; R1 - variável houveTroca
; R2 - variável i
; R3 - variável auxiliar
PLACE  0000H      ; localiza bloco de código
ronda:
       MOV   R0, seq  ; num = seq;
       MOV   R1, 0    ; houveTroca = false;
       MOV   R2, 0    ; i = 0;
iteração:
       MOV   R7, N    ; N
       SUB   R7, 1    ; N-1
       CMP   R2, R7   ; i < N-1 ?
       JGE   teste    ; se não, o ciclo acabou
       MOV   R7, [R0]    ; *num
       MOV   R8, [R0+2]  ; *(num+1)
       CMP   R7, R8      ; *num > *(num+1)?
       JLE   próximo  ; não, passa ao próximo
       MOV   R3, R7      ; auxiliar = *num;
       MOV   [R0], R8    ; *num = *(num+1);
       MOV   [R0+2], R3  ; *(num+1) = auxiliar;
       MOV   R1, 1       ; houveTroca = true;
próximo:
       ADD   R2, 1       ; i = i + 1;
       ADD   R0, 2       ; num = num + 1;
       JMP   iteração    ; próxima iteração
teste: CMP   R1, 0       ; houveTroca = falso?
       JNZ   ronda       ; mais uma ronda
fim:   JMP   fim         ; acaba aqui
```

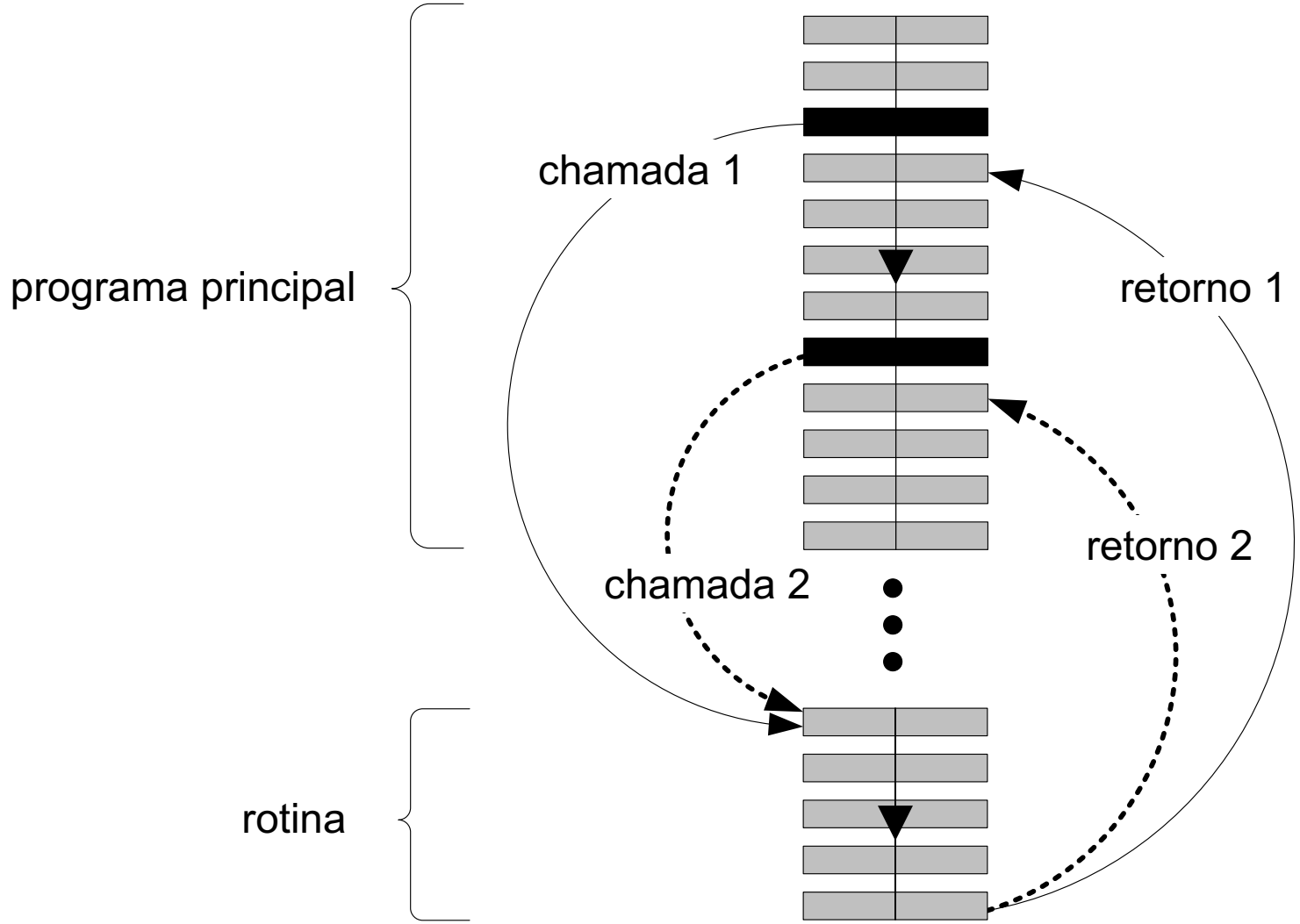
Conclusão



- A introdução de abstrações introduz muitas vezes um **custo** ao nível de desempenho
- Há por isso um **compromisso** entre abstrações amigáveis para o programador e eficiência da computação
- O compilador ideal consegue o **compromisso ótimo** entre estas duas variáveis
  - Por isso é fundamental ter um conhecimento profundo da **arquitetura dos computadores**!

Rotinas

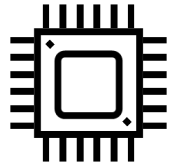
Uso de rotinas



Questões fundamentais

- Como é que se chama uma rotina?
- Como é que a rotina sabe para onde retornar quando termina?
- Como evitar que a rotina “estrague” os valores dos registos no programa principal?
- Como é se efetua a passagem de parâmetros?

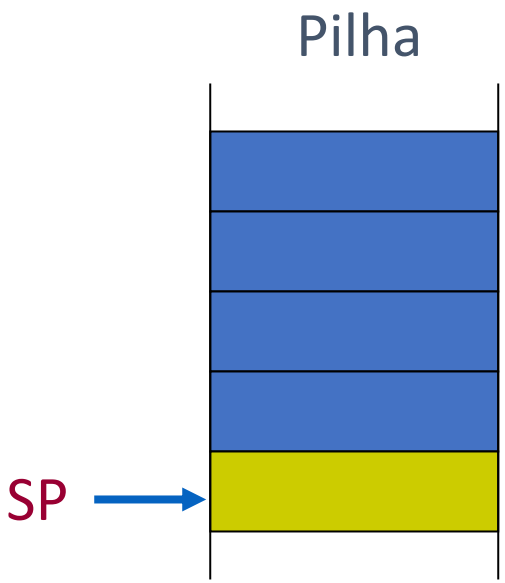
Questões fundamentais

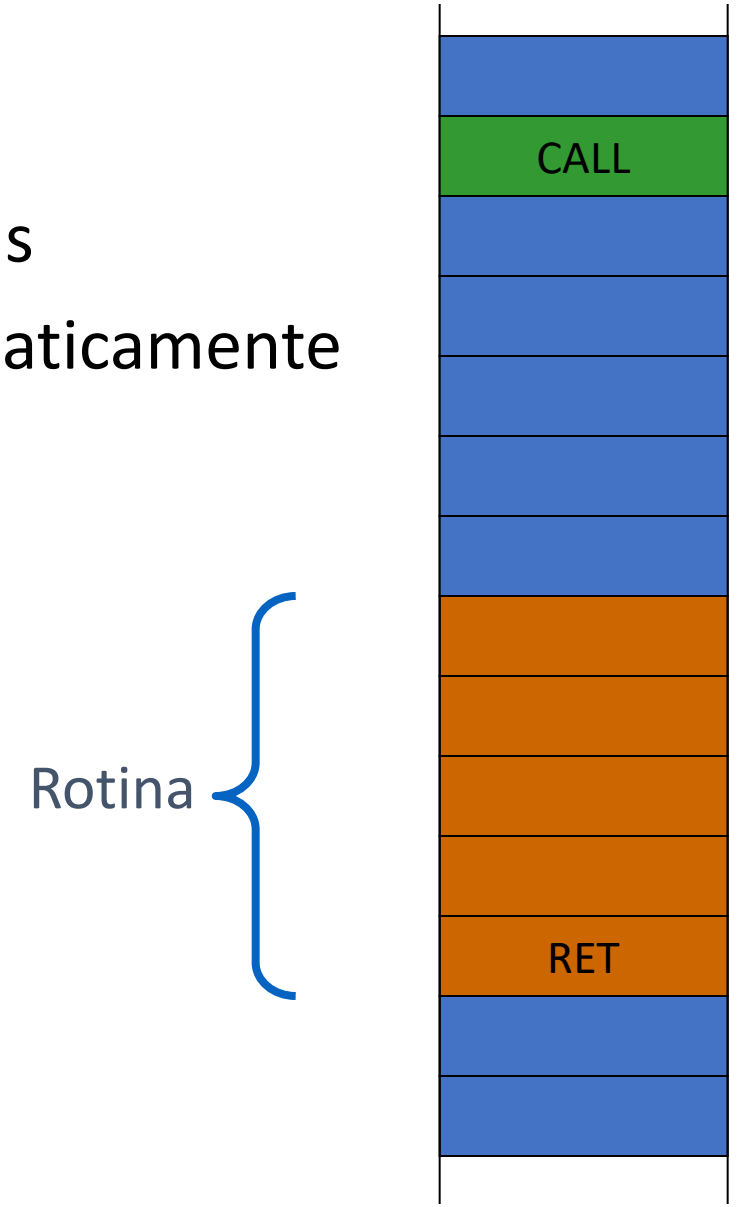


- **Como é que se chama uma rotina?**
- **Como é que a rotina sabe para onde retornar quando termina?**
- Como evitar que a rotina “estrague” os valores dos registos no programa principal?
- Como é se efetua a passagem de parâmetros?

Chamada/retorno

- As rotinas não sabem de onde são chamadas
- O **par CALL-RET** resolve esta questão automaticamente





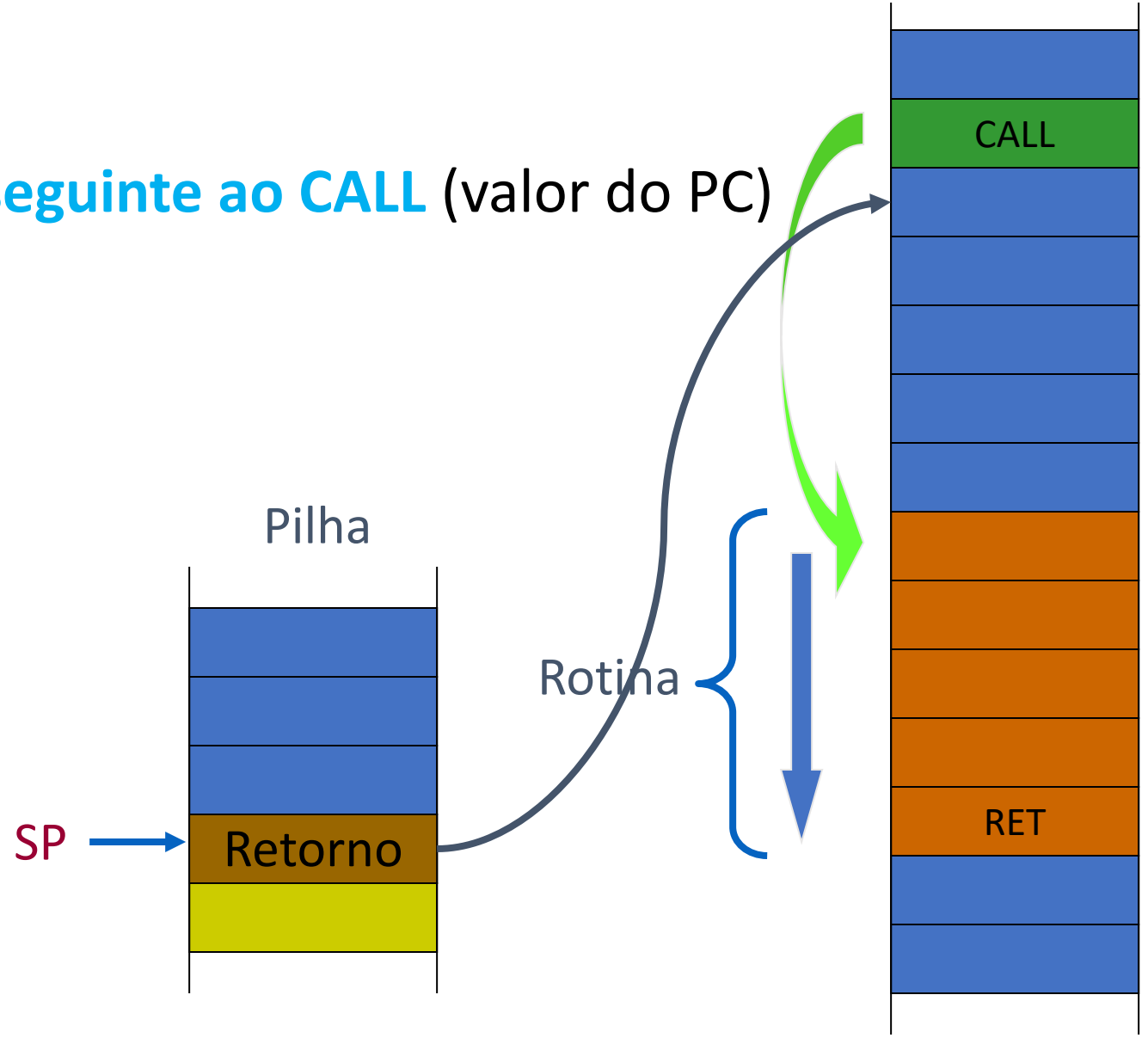
Chamada/retorno

- **A pilha** memoriza o **endereço seguinte ao CALL** (valor do PC)

  SP ← SP - 2

  M[SP] ← PC

  PC ← *endereço da rotina*



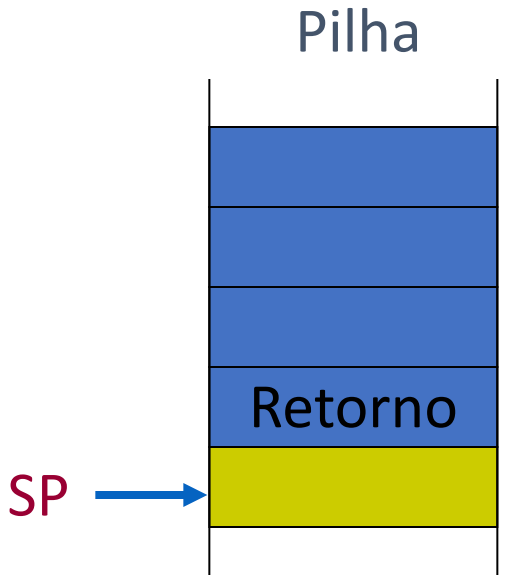
Chamada/retorno

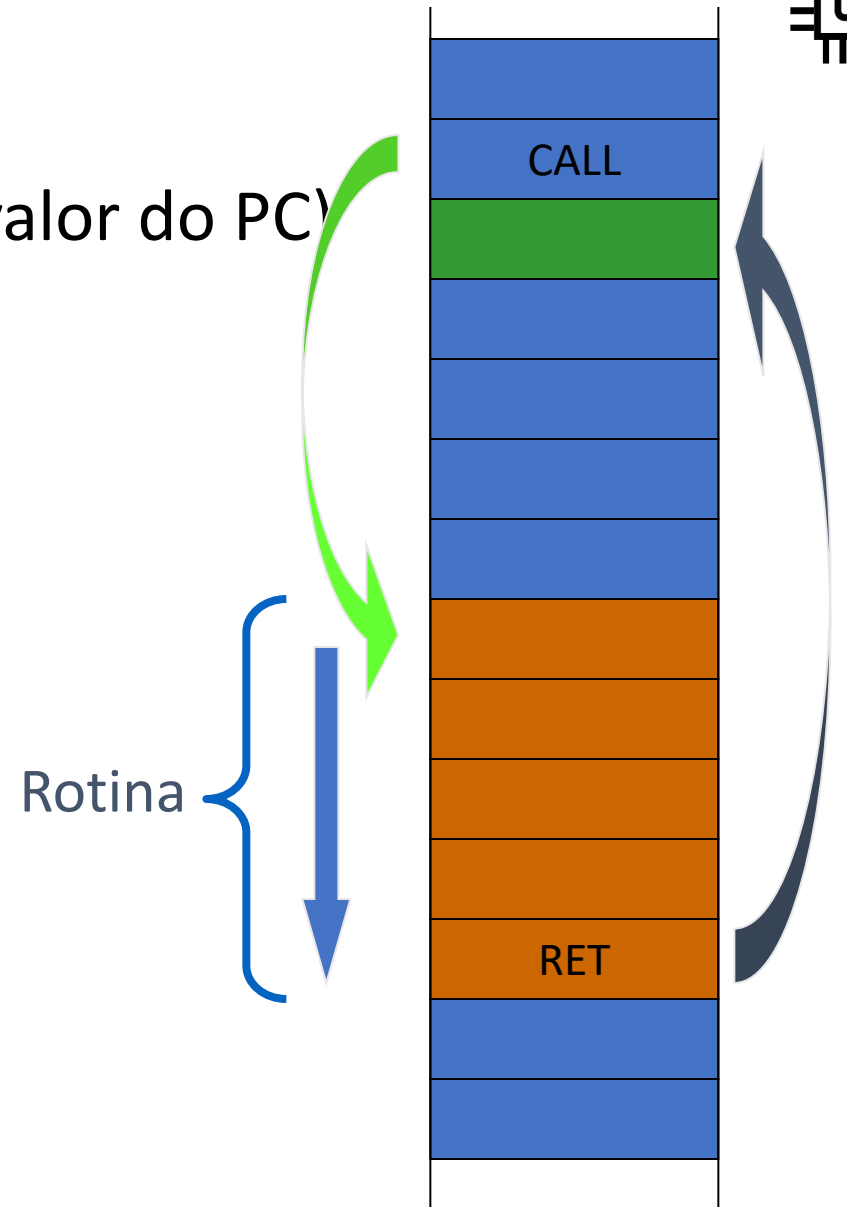
- **A pilha** memoriza o **endereço seguinte ao CALL** (valor do PC)

  PC ← M[SP]

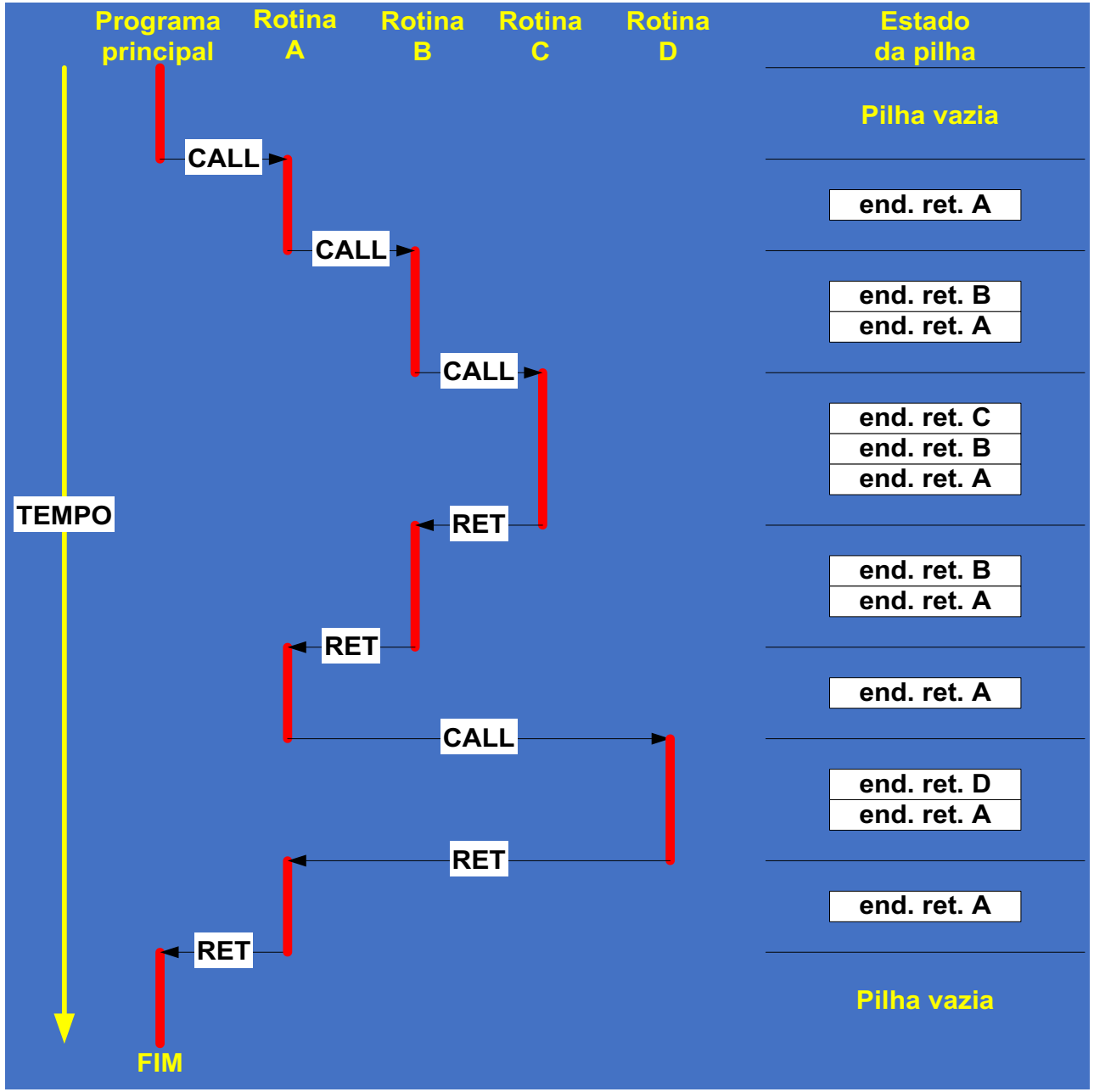
  SP ← SP + 2

- RET usa esse endereço para retornar





Rotinas e a pilha



TÉCNICO LISBOA

Instruções de chamada/retorno

Instruções	Descrição	Comentários
JMP K	PC ← PC + K	Salto sem retorno
CALL K	SP ← SP - 2 M[SP] ← PC PC ← PC + K	Ajusta SP Guarda endereço de retorno na pilha Salta para a rotina
RET	PC ← M[SP] SP ← SP + 2	Recupera endereço de retorno Ajusta SP

- Tem de se reservar espaço para a pilha (STACK)
- O SP (Stack Pointer) tem de ser inicializado (com o endereço a seguir à área da pilha)
- Exemplo: **rotinas.asm**

Questões fundamentais

- Como é que se chama uma rotina?
- Como é que a rotina sabe para onde retornar quando termina?
- **Como evitar que a rotina "estrague" os valores dos registos no programa principal?**
- Como é se efetua a passagem de parâmetros?

Salvaguarda de registos

- Uma rotina nunca sabe de onde é chamada
- Se usar **registos**, tem de:
  - **salvá-los (na pilha)** antes de os usar
  - **restaurá-los pela ordem inversa** antes de retornar

```
. . .
PUSH  R1  ; salva R1
PUSH  R2  ; salva R2
. . .     ; código da rotina que altera R1 e R2
POP   R2  ; restaura R2
POP   R1  ; restaura R1
RET       ; já pode retornar
```

Guardar registos na pilha

Instruções		Descrição	Comentários
PUSH	Rs	SP ← SP - 2 M[SP] ← Rs	SP aponta sempre para a última posição ocupada (topo)
POP	Rd	Rd ← M[SP] SP ← SP + 2	POP não destrói os valores lidos da pilha

- Não esquecer – **antes de usar a pilha** tem de:
  - Verificar o tamanho máximo previsível para a pilha e **reservar espaço suficiente** (geralmente, 100H palavras é suficiente)
  - **Inicializar o SP com o endereço seguinte à área reservada** para a pilha

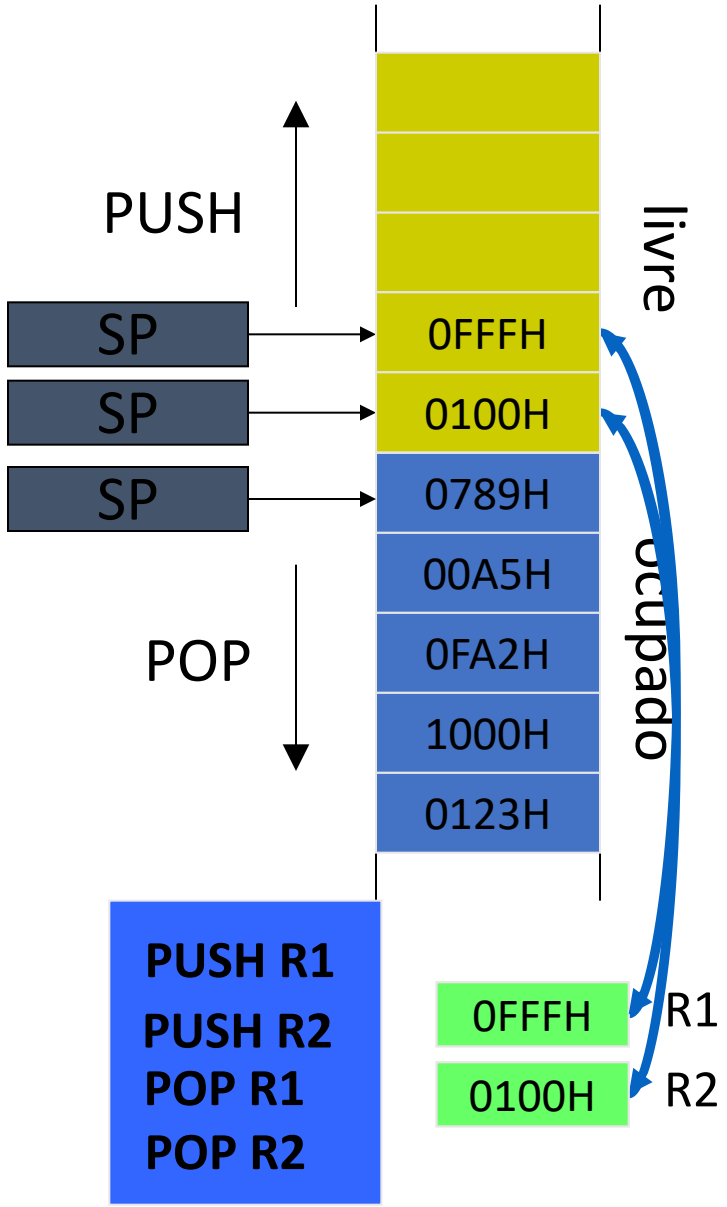
Pilha (*stack*)

- SP aponta para a última posição ocupada da pilha (topo da pilha)

PUSH R_i: SP←SP-2; M[SP]←R_i

POP R_i: R_i←M[SP]; SP←SP+2

- Exemplos:
  - **rotinas-push-pop.asm**
  - **maximo.asm**

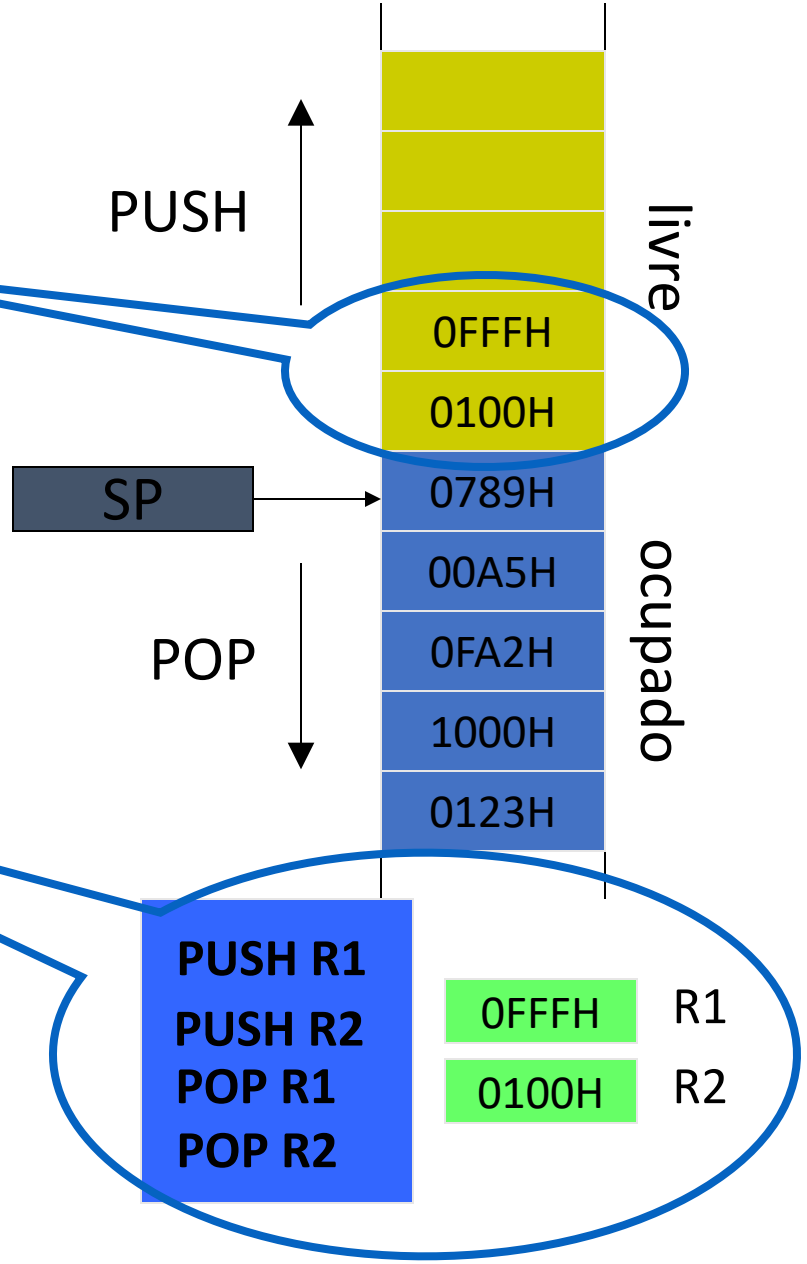


Pilha (*stack*)

O POP não apaga os valores, apenas os deixa na zona livre.

Os POPs têm de ser feitos pela ordem inversa dos PUSHes, senão os valores vêm trocados!

Exemplo: **push-pop-bug.asm**



Instruções CALL e RET

- A instrução CALL *rotina* equivale **conceptualmente** a:

```
PUSH    PC      ; guarda o endereço da instrução
                ; que vem a seguir ao CALL
JMP     rotina  ; transfere controlo para a rotina
```

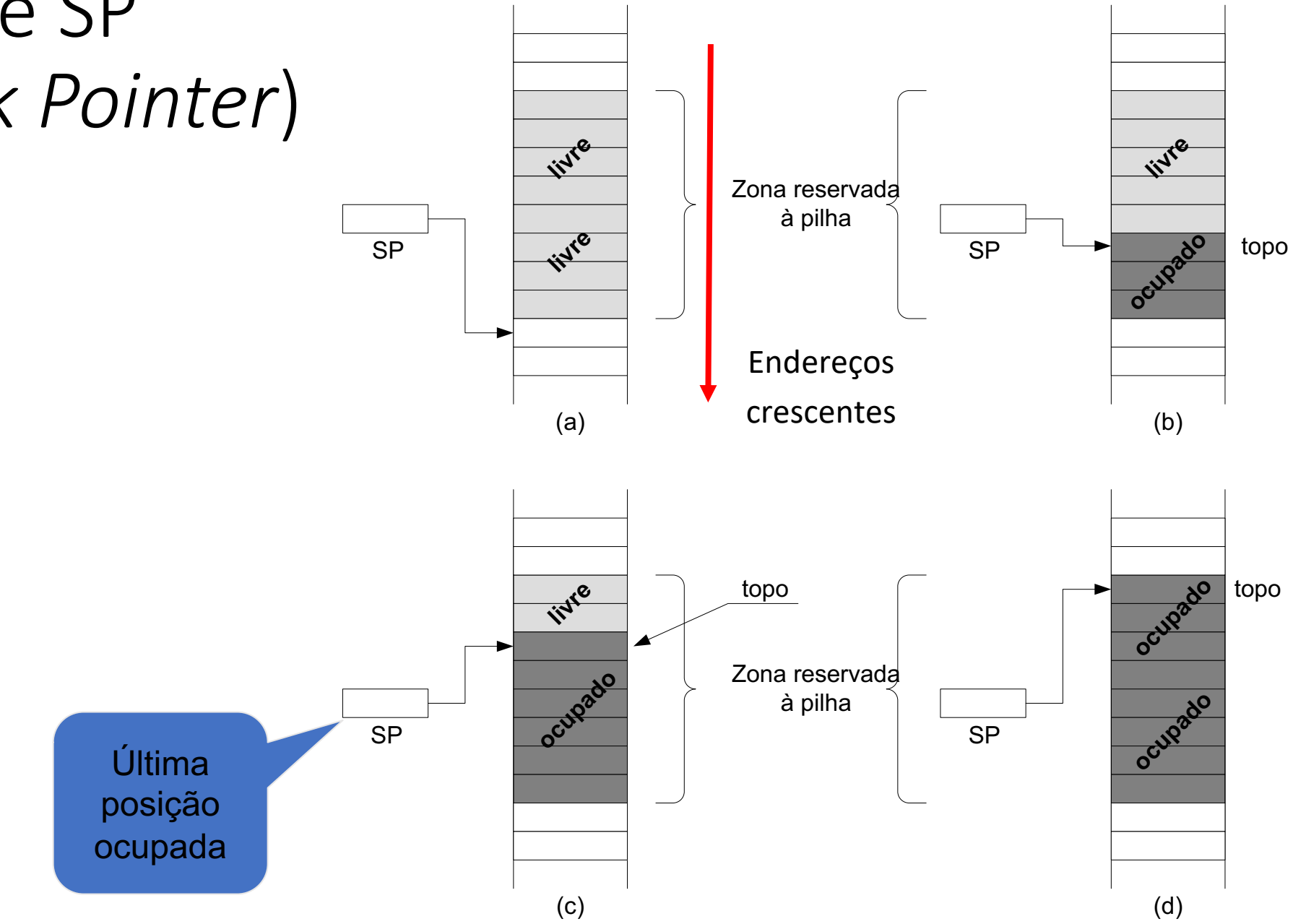
- A instrução RET equivale **conceptualmente** a :

```
POP     PC      ; retira da pilha o endereço da instrução
                ; para onde deve retornar e salta para lá
```

- Quando se chamam muitas rotinas, o mecanismo **LIFO da pilha** implica que a ordem de retorno seja **inversa** à ordem de chamada

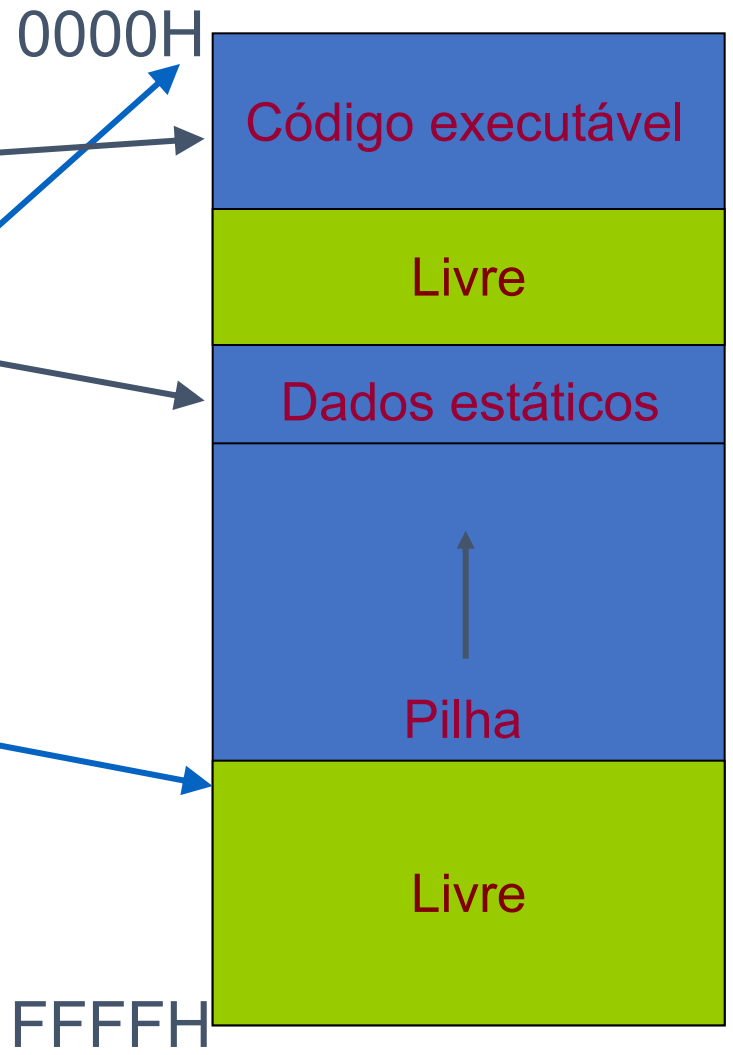
*NOTA: Isto é só conceptual, as instruções PUSH **PC** e POP **PC** não existem.

Pilha e SP (*Stack Pointer*)



Mapa de endereçamento

- **PLACE** permite localizar:
  - Blocos de código
  - Dados estáticos (variáveis criadas com WORD)
- No PEPE, o PC é inicializado com 0000H
- A pilha é localizada através da **inicialização do SP**



Recursividade

- Tal como nas linguagens de alto nível, a **recursividade** também funciona em assembly
- A rotina **pode chamar-se a ela própria** (vai guardando o endereço de retorno e eventuais registos PUSHed)
- Tem de haver uma **condição de terminação** que permita regressar das N instâncias de chamada da rotina
- A "**profundidade**" da pilha atingida depende do caso concreto (cada chamada recursiva substitui uma iteração na versão iterativa do algoritmo)
- Exemplos:
  - **fatorial-iterativo.asm**
  - **fatorial-recursivo.asm**

Questões fundamentais

- Como é que se chama uma rotina?
- Como é que a rotina sabe para onde retornar quando termina?
- Como evitar que a rotina “estrague” os valores dos registos no programa principal?
- **Como é se efetua a passagem de parâmetros?**

Passagem de valores

- Invocação: z = soma (x, y)
- Quem chamar a função tem de colocar os parâmetros **num local combinado** com a função.
- Idem para o valor de retorno

Por registos

```
MOV     R1, x
MOV     R2, y
CALL    soma
; resultado em R3
```

Por memória (pilha)

```
MOV     R1, x       ; 1º operando
PUSH    R1
MOV     R2, y       ; 2º operando
PUSH    R2
CALL    soma
POP     R3
MOV     R4, z       ; endereço resultado
MOV     [R4], R3
```

Passagem de valores por registos

```
MOV     R1, x
MOV     R2, y
CALL    soma
; resultado em R3
```

- Vantagens
  - **Eficiente** (registos)
  - Registo de saída de uma função pode ser logo o registo de entrada noutra (não é preciso copiar o valor)
- Desvantagens
  - **Menos geral** (número de registos limitado, não suporta recursividade)
  - Estraga registos (pode ser preciso guardá-los na pilha)

Passagem de valores pela pilha

```
MOV     R1, x       ; 1º operando
PUSH    R1
MOV     R2, y       ; 2º operando
PUSH    R2
CALL    soma
POP     R3
MOV     R4, z       ; endereço resultado
MOV     [R4], R3
```

```
soma:   POP     R2
        POP     R1
        MOV     R2,[R2]
        ADD     R2,[R1]
        PUSH    R2
```

```
soma:   MOV     R2,[SP+2]
        MOV     R1,[SP+4]
        MOV     R2,[R2]
        ADD     R2,[R1]
        MOV     [SP+2],R2
```

- Vantagens
  - **Genérico** (dá para qualquer número de parâmetros)
  - **Recursividade fácil** (já se usa a pilha)
- Desvantagens
  - **Pouco eficiente** (muitos acessos à memória)
  - É **preciso cuidado** com os PUSHes e POPs (tem de se "consumir" os parâmetros e os valores de retorno)

Exemplo de rotina

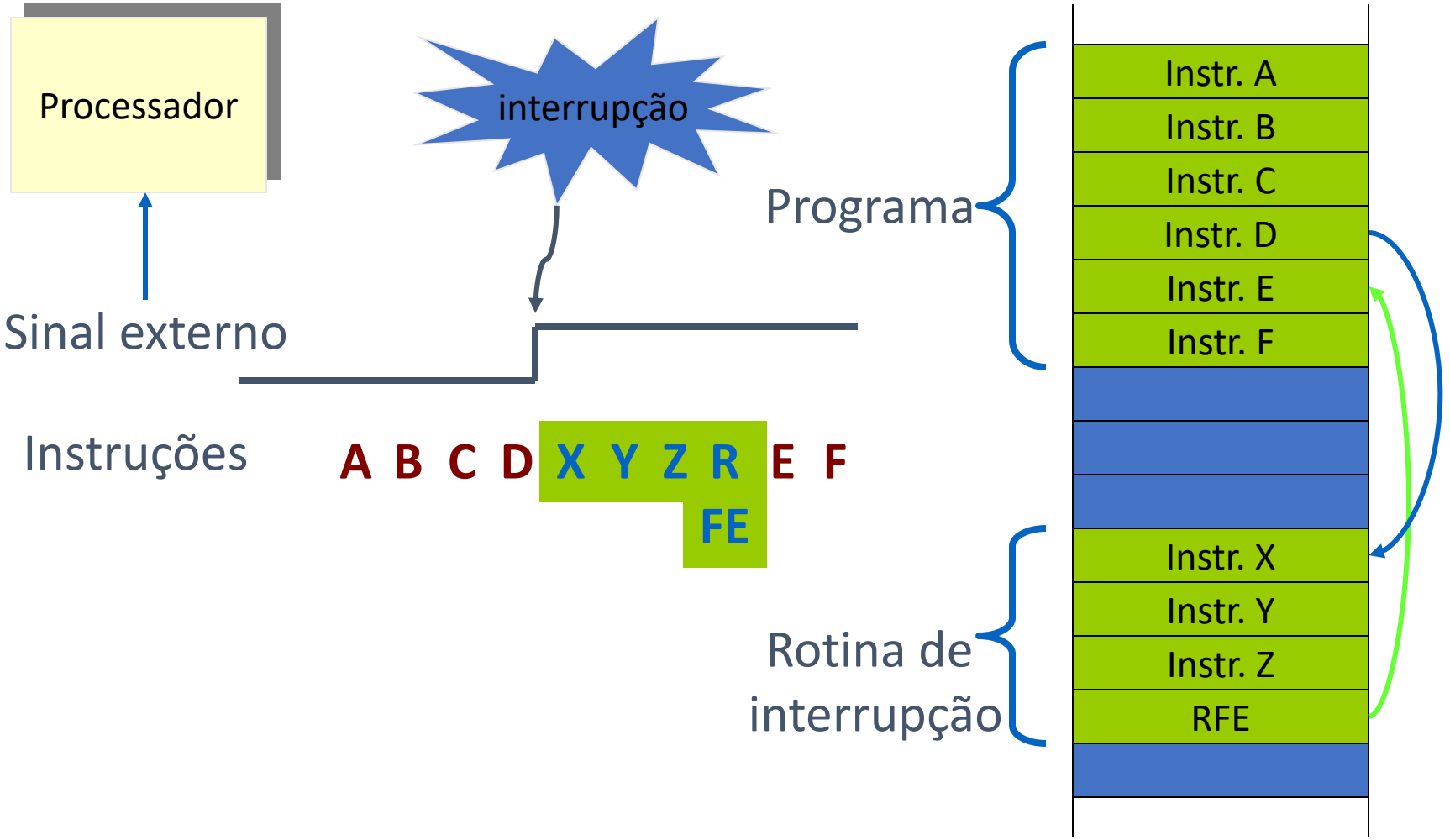
```
PLACE           1000H
inicio_pilha:   TABLE   100H            ; reserva espaço para a pilha (256 words)
SP_inicial:

PLACE           0
inicio:         MOV     SP, SP_inicial ; inicializa SP
                MOV     R1, 4          ; argumento
                CALL    FACT           ; chama rotina
fim:                    JMP            fim

;*************************************************************
;   Descrição: Calcula o factorial de um número (n!)
;   Entradas:      R1 - Parâmetro (valor n)
;   Saídas:        R2 - Factorial de n (n!)
;*************************************************************
fact:           PUSH    R1             ; é boa política a rotina guardar os registos que vai alterar...
                CMP     R1, 1          ; n válido?
                JLE     n_LE_1         ; vai tratar casos n <= 1
ok:             MOV     R2, R1         ; inicializa resultado com n
ciclo:          SUB     R1, 1          ; n - 1
                JZ      sai            ; se R1 já era 1, acabou
                MUL     R2, R1         ; resultado = resultado * n-1
                JMP     ciclo          ; vai acumulando
n_LE_1 :        MOV     R2, 1          ; n! = 1 ( se n<=1)
sai:            POP     R1             ; ...e deixá-los com o valor inicial
                RET
```

Exceções e interrupções

Exceções e interrupções

- Exceção - qualquer evento que pode ocorrer **de forma inesperada** para o programa que está a correr
  - Não é prático estar sempre a testar se algo aconteceu
  - Como reagir e tratar o evento, logo que ele ocorra?
- Solução: **interromper** o programa normal e **invocar uma rotina** de tratamento da exceção.
- As exceções podem ser causadas:
  - Pelo próprio programa (divisão por zero, falta de página, acesso à memória desalinhado, etc.). São **síncronas** em relação ao programa.
  - Pela ativação de um **pino externo** (**interrupções**). São **assíncronas** face ao programa, sendo imprevisível a instrução em que ocorrem.

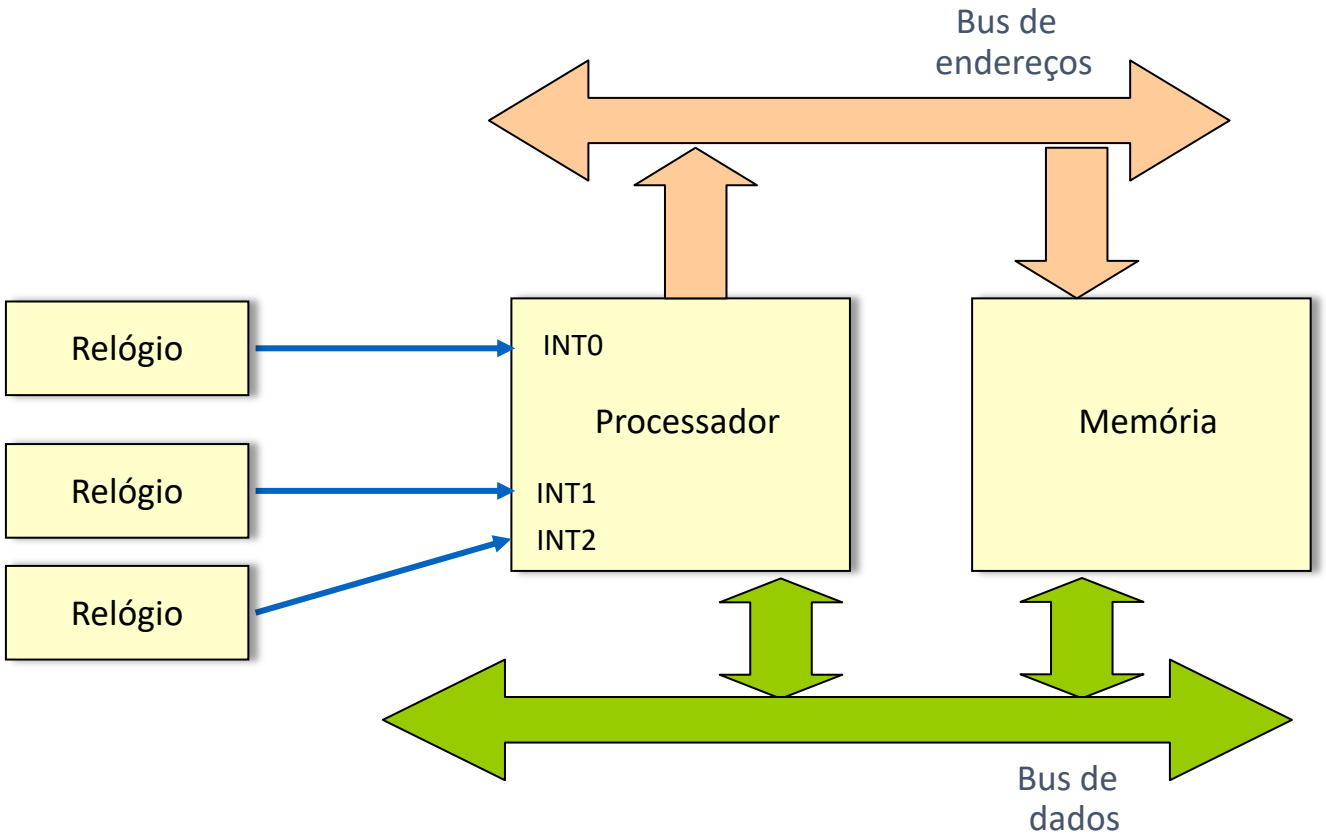
Mecanismo das interrupções



O programa nem se "apercebe" da interrupção

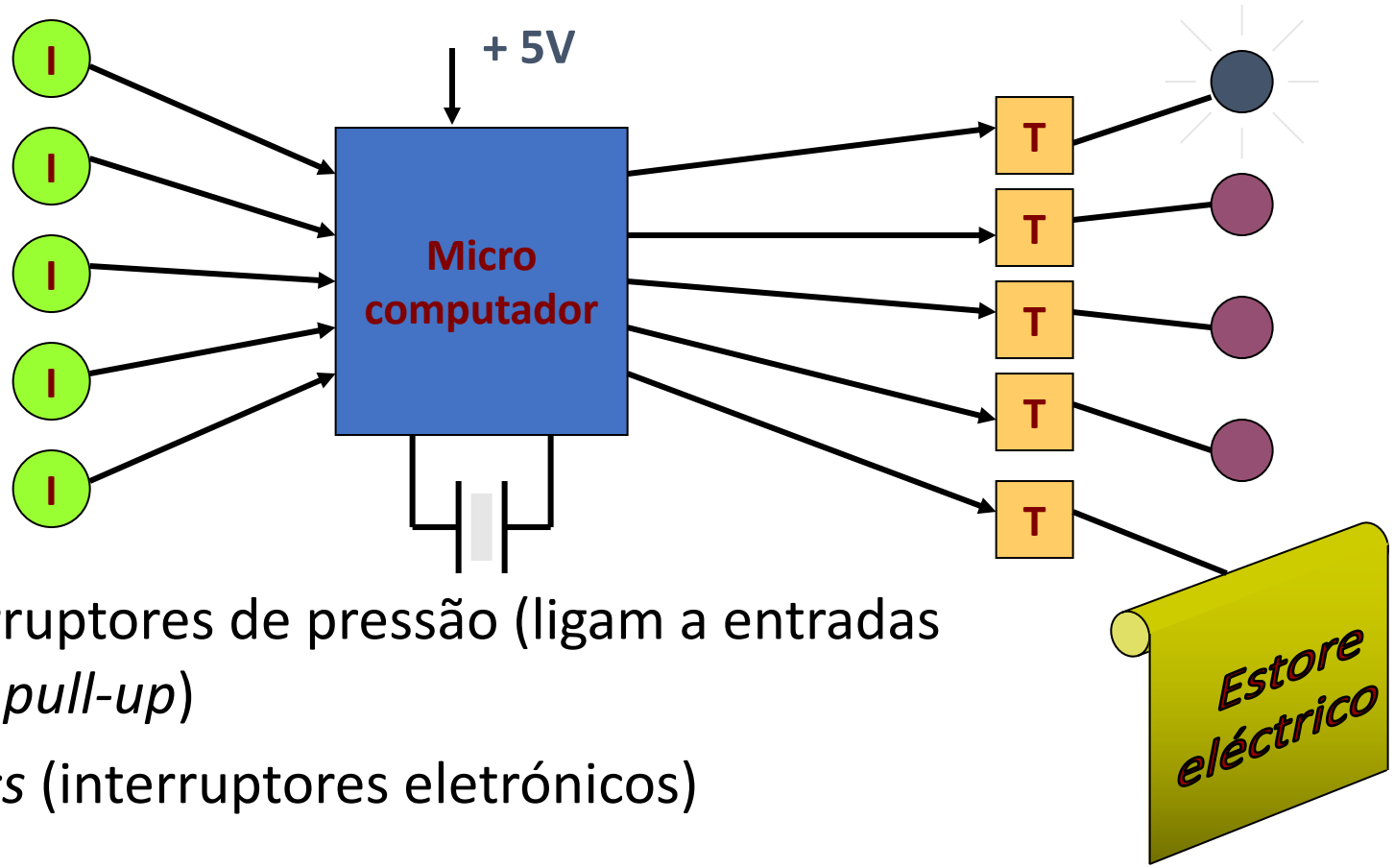
Interrupções

- O processador tem vários **pinos de interrupção**
- Podem ligar a relógios (temporizadores), botões, sensores, etc.



Exemplo de aplicação

- Aplicação: controlo de uma casa (domótica)

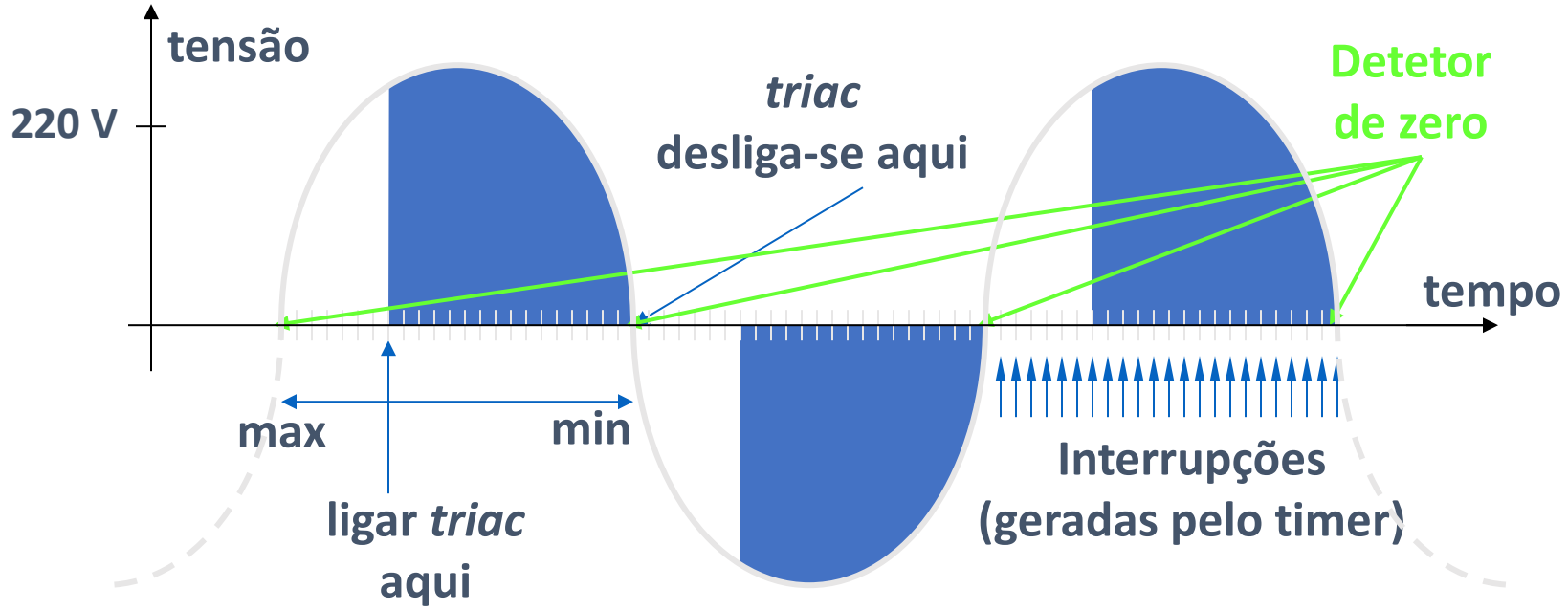


I – Interruptores de pressão (ligam a entradas com *pull-up*)

T – *Triacs* (interruptores eletrónicos)

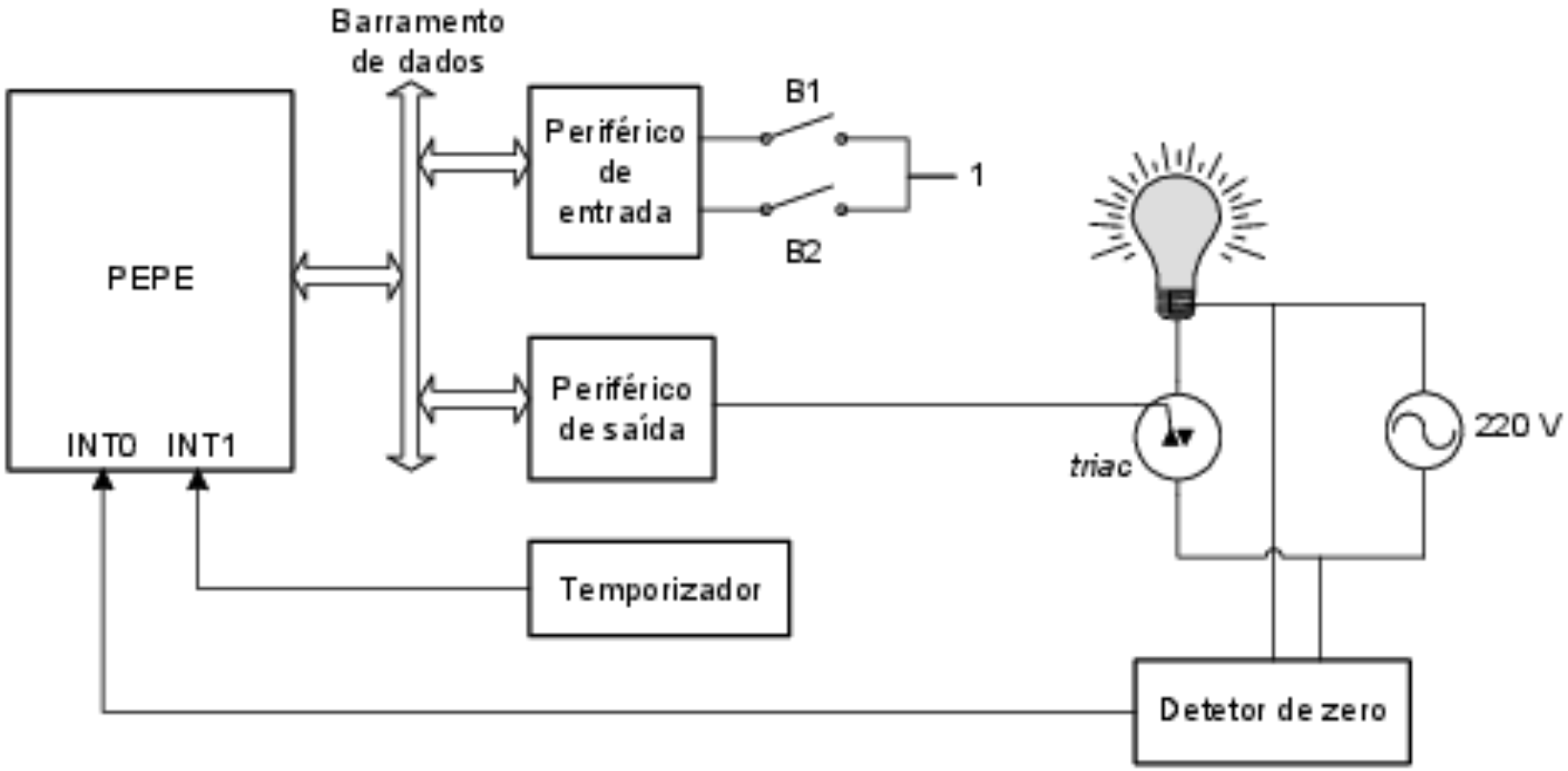
Controlo de tempo real

- Controlo da intensidade luminosa das lâmpadas:

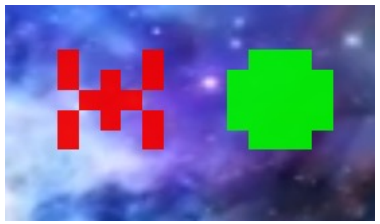


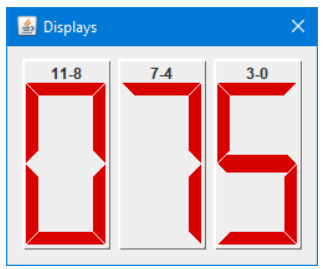
  - Subir ou descer a intensidade luminosa é mudar o ponto de ativação do *triac* (número de interrupções ocorridas desde a deteção de zero da sinusoide)

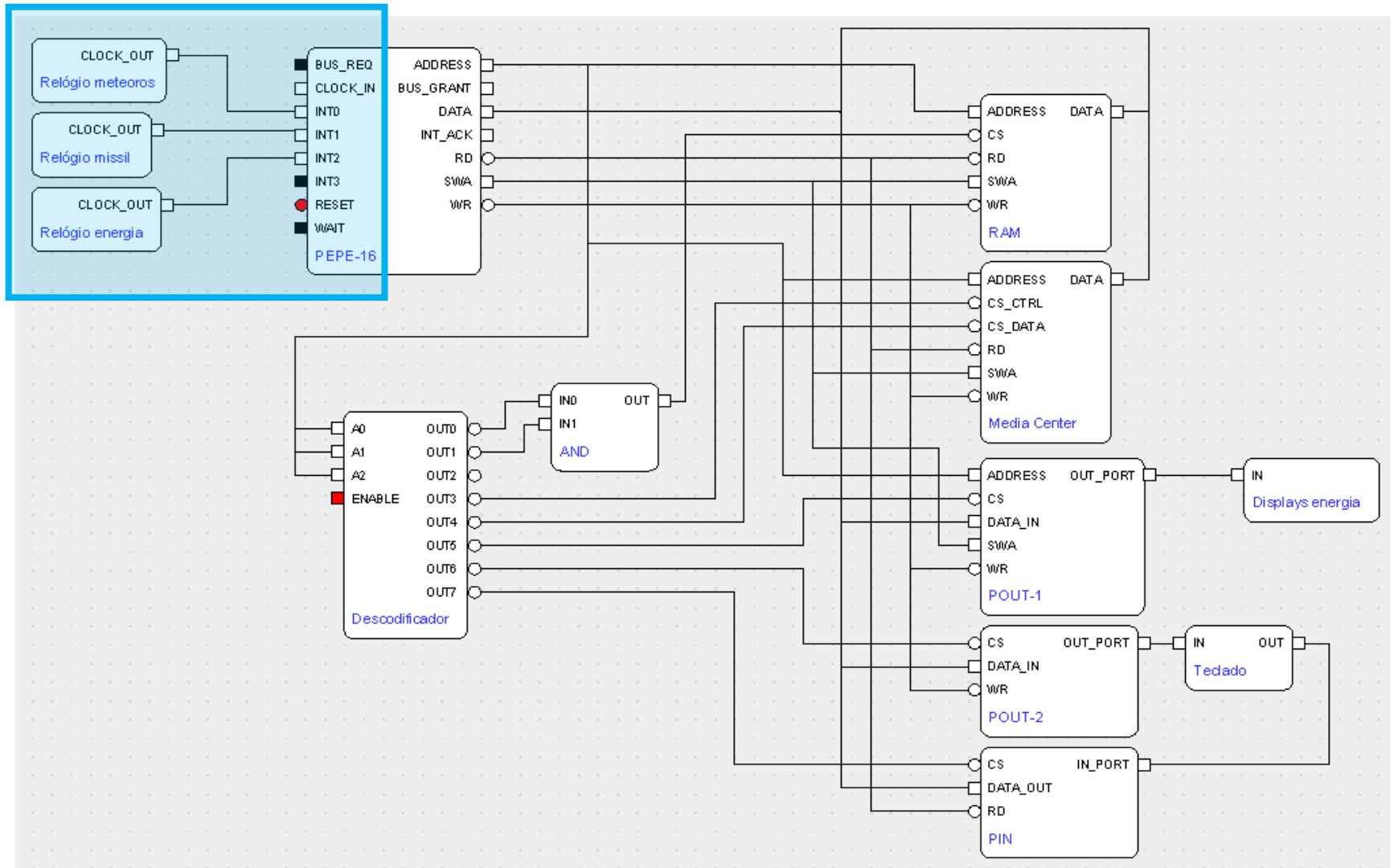
Circuito de controlo da intensidade de uma lâmpada



Interrupções no projeto







TÉCNICO LISBOA

Rotinas de interrupção

- Invocáveis em qualquer ponto do programa quando um **sinal externo**:
  - muda de valor (flanco) – de 0 para 1, o mais comum, ou de 1 para 0 – ou
  - tem um dado valor (nível) – ou 1 ou 0
- **Não podem alterar nada** do estado do processador (nem mesmo os bits de estado)
- A invocação da rotina de interrupção **guarda automaticamente na pilha** (equivalente a dois PUSHs):
  - Endereço de retorno (endereço da próxima instrução na altura em que a interrupção aconteceu)
  - Registo dos bits de estado
- A instrução **RFE (*Return From Exception*) faz o equivalente a dois POPs** pela ordem inversa (repondo os bits de estado e fazendo o retorno).
  - RET e RFE não são equivalentes!
- Se a rotina de interrupção alterar qualquer registo, tem de o guardar primeiro na pilha e restaurá-lo antes do RFE
  - Como é **boa prática** fazer em qualquer rotina

Exemplo de rotina de interrupção

```
DISPLAYS   EQU 0A000H      ; endereço dos displays de 7 segmentos (periférico POUT-1)

           PLACE  0100H
pilha:     STACK 10H       ; espaço reservado para a pilha
SP_inicial:

tab:       WORD rot0       ; tabela de interrupções (neste caso só tem uma WORD)

           PLACE  0        ; o código tem de começar em 0000H
           MOV    SP, SP_inicial  ; inicializa SP
           MOV    BTE, tab        ; inicializa BTE
           MOV    R0, 0           ; inicializa contador (variável global)
           EI0                    ; permite interrupção 0
           EI                     ; permite interrupções (geral)
fim:       JMP    fim             ; fica à espera

rot0:                      ; rotina que trata da interrupção 0 (incrementa contador)
           PUSH   R1
           ADD    R0, 1           ; incrementa contador (var global não é preciso salvaguardar)
           MOV    R1, DISPLAYS    ; endereço do periférico
           MOVB   [R1], R0        ; atualiza display
           POP    R1
           RFE                    ; regressa da interrupção
```

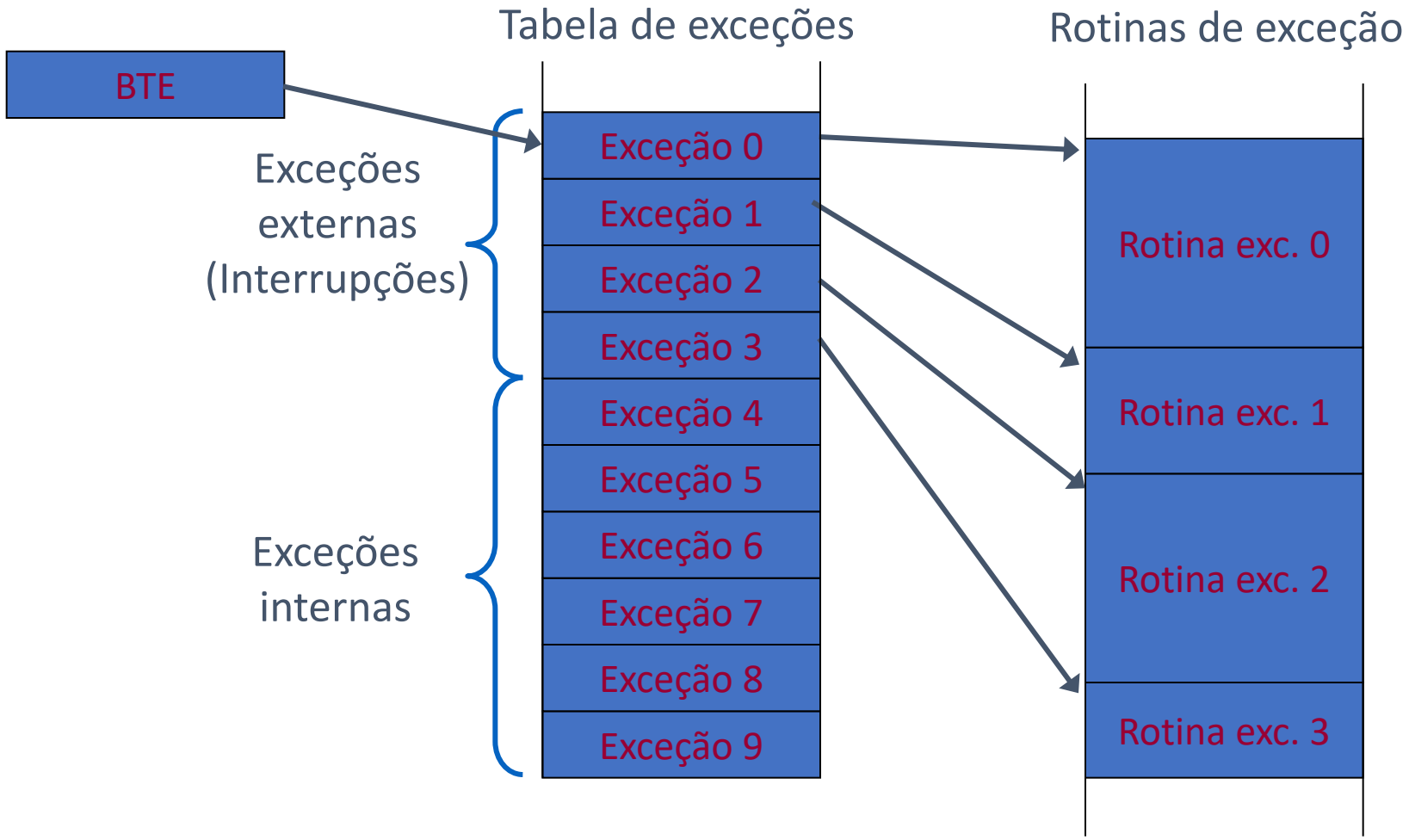
interrupção-simples.asm

Várias rotinas de interrupção

```
        ...                     ; dados, pilha

tab:    WORD rot0               ; tabela de interrupções
        WORD rot1               ; cada endereço tem de ficar na posição na tabela
        WORD rot2               ; correspondente ao respetivo número de interrupção (0 a 2)

        PLACE  0                ; o código tem de começar em 0000H
        MOV    SP, SP_inicial   ; inicializa SP
        MOV    BTE, tab         ; inicializa BTE
        ...
        EI0                     ; permite interrupção 0
        EI1                     ; permite interrupção 1
        EI2                     ; permite interrupção 2
        EI                      ; permite interrupções (geral)
fim:    JMP    fim              ; fica à espera

rot0:   ...                     ; rotina que trata da interrupção 0
        RFE                     ; regressa da interrupção
rot1:   ...                     ; rotina que trata da interrupção 1
        RFE                     ; regressa da interrupção
rot2:   ...                     ; rotina que trata da interrupção 2
        RFE                     ; regressa da interrupção
```

interrupções-várias.asm

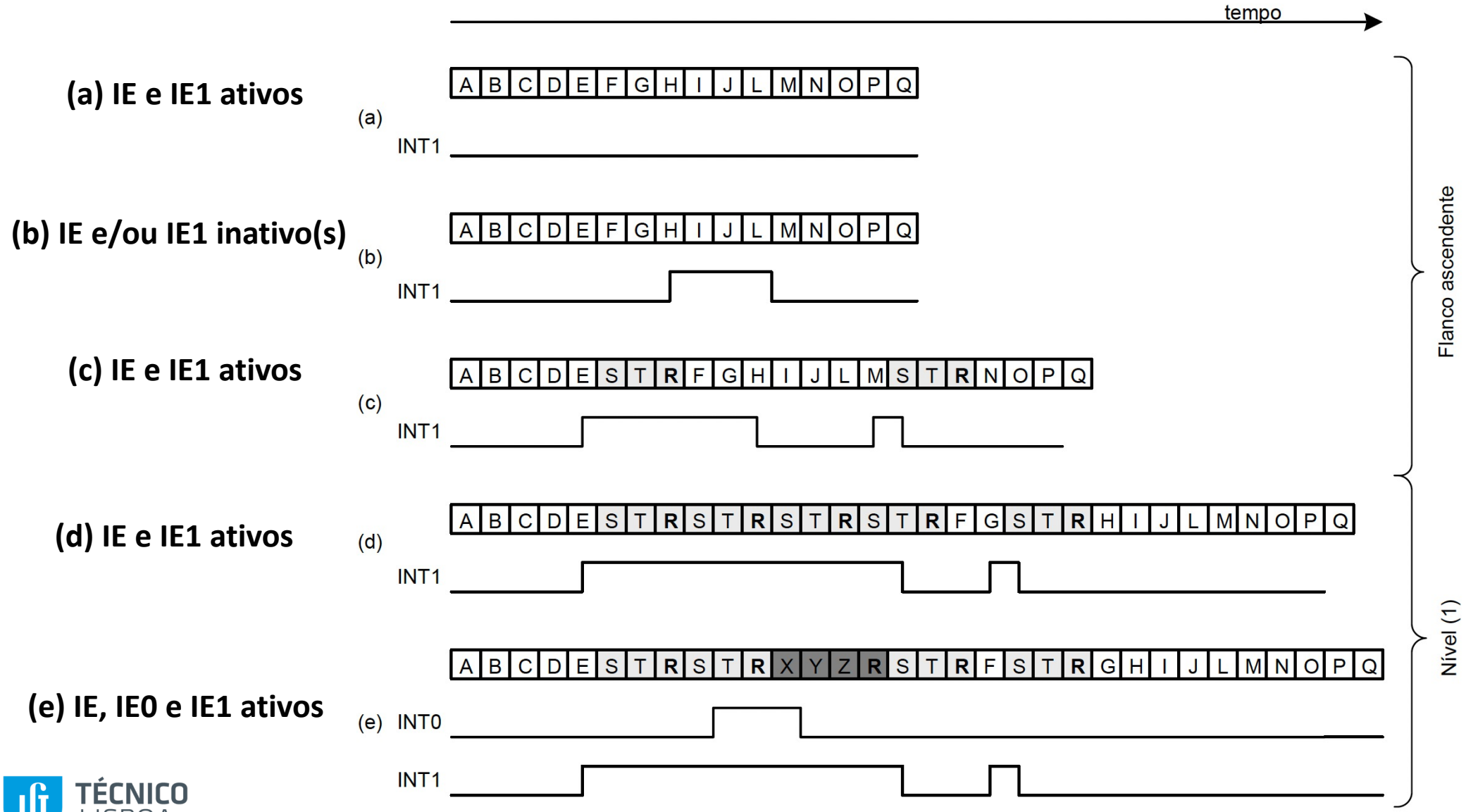
Tabela de exceções



Bit de estado IE

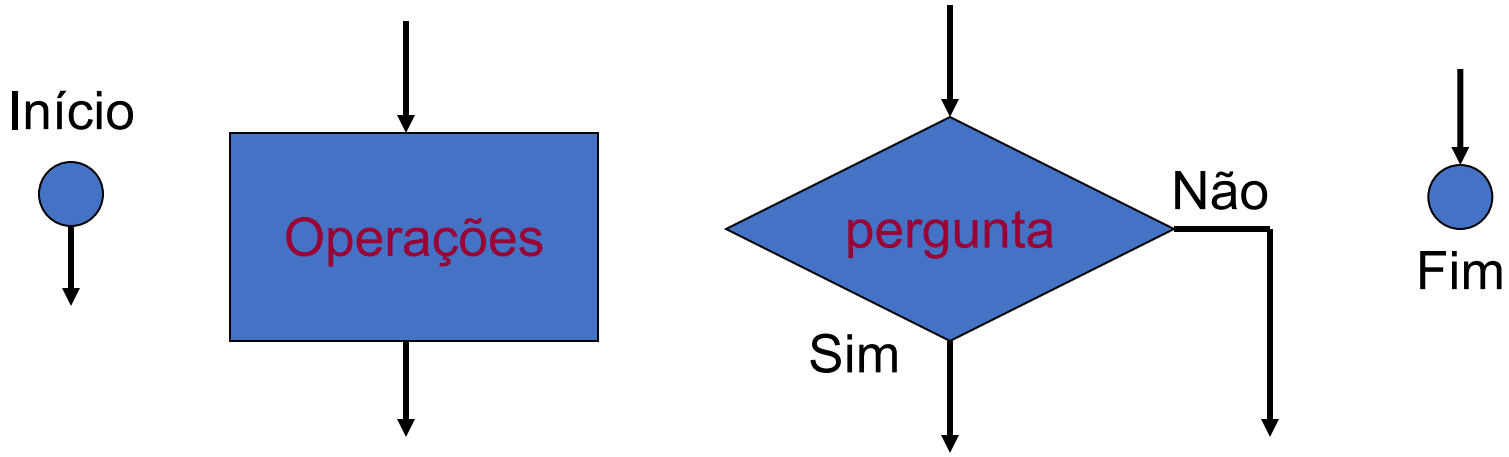
- Um programa pode estar a executar operações críticas que **não devem ser interrompidas**
- Por isso, existe um **bit de estado (IE) que quando está a 0 impede o processador** de atender interrupções.
- **Para manipular** este bit existem duas instruções:
  - **EI** (*Enable Interrupts*). Faz IE ← 1
  - **DI** (*Disable Interrupts*). Faz IE ← 0
- A própria rotina de interrupção pode ser crítica e não permitir interrupções a ela própria. Por isso, IE é colocado a 0 **automaticamente** quando uma interrupção é atendida.
- O bit IE é **automaticamente reposto no RFE** (porque o registo das flags é reposto)

Funcionamento das interrupções

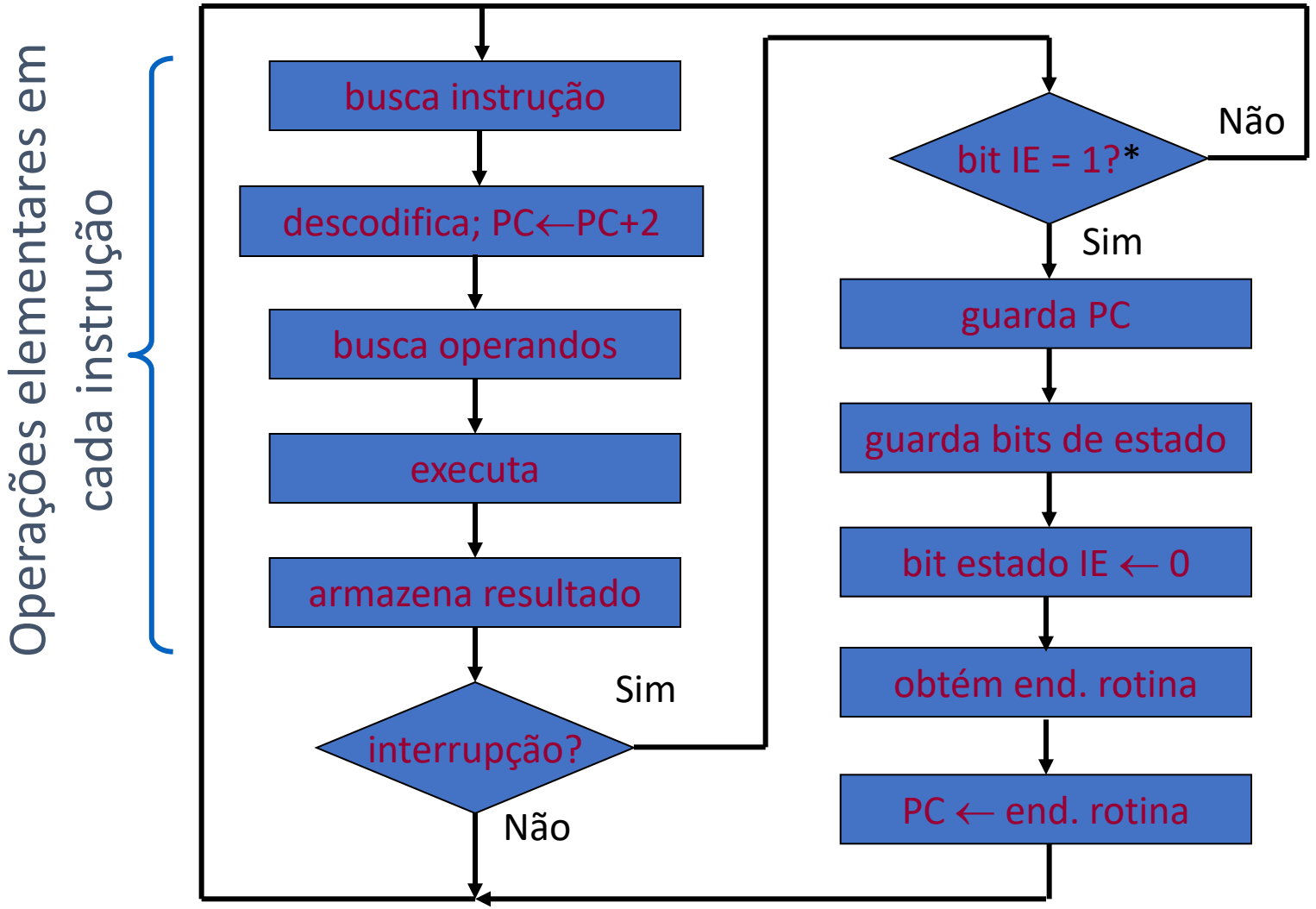


Fluxograma

- Notação gráfica para especificar o comportamento de uma rotina
- Construções fundamentais:

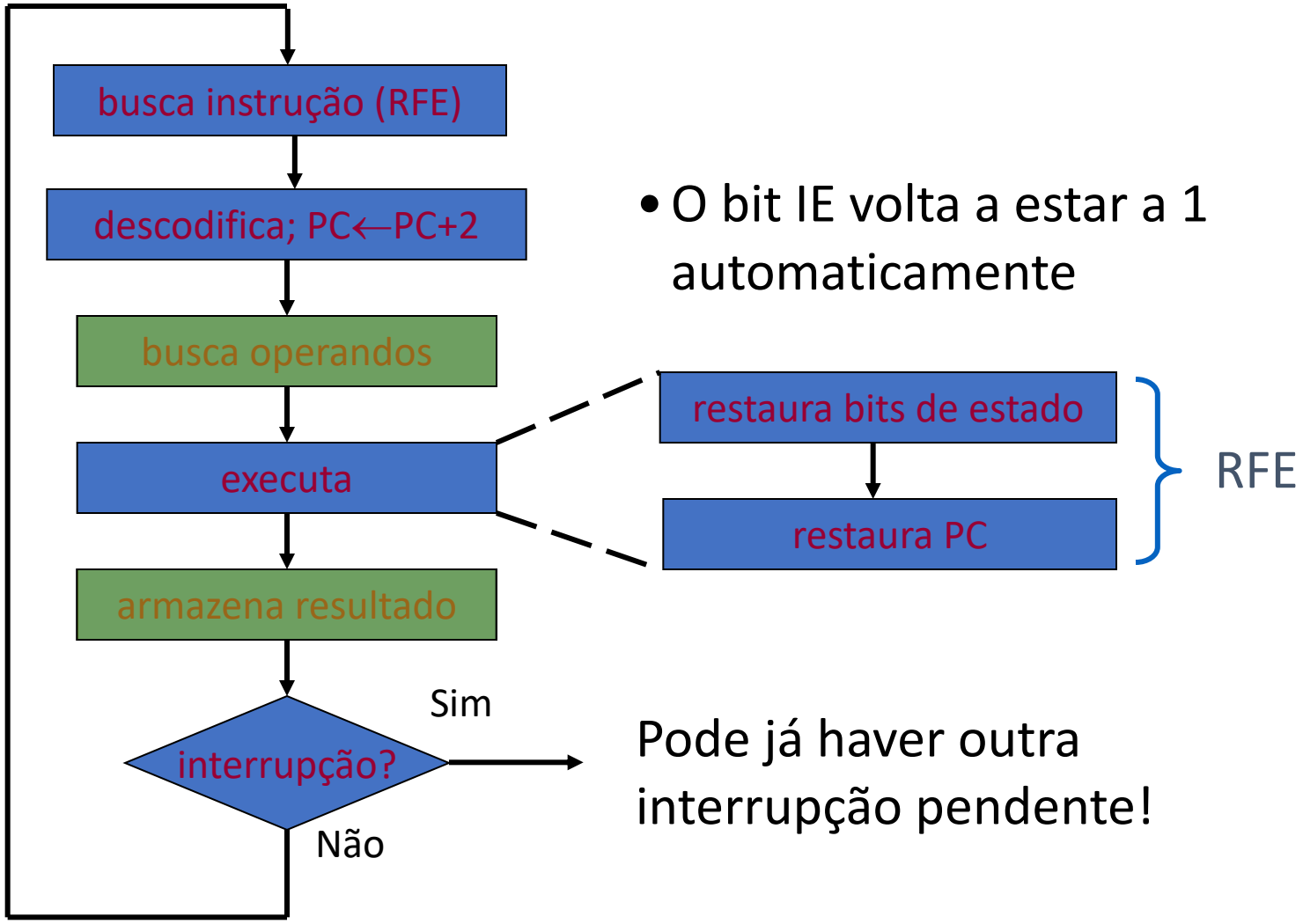


Tratamento de interrupções



*versão simplificada, deveria incluir IE0, IE1, IE2

Retorno de interrupções (RFE)



- O bit IE volta a estar a 1 automaticamente

Pode já haver outra interrupção pendente!

As rotinas de interrupção têm de **terminar com RFE**

Rotina de interrupção típica

```
rotina_int:                     ; DI automático, bit IE fica a 0 (não responde às interrupções)
        push    registos        ; guarda registos que a  rotina vá usar
        instruções críticas     ; sequência de instruções que não pode ser interrompida
        EI                      ; permite interrupções (se necessário)
        instruções não críticas ; sequência de instruções que pode ser interrompida
        pop     registos        ; restaura registos (pela ordem inversa)
        RFE                     ; retorna da rotina (EI automático)
```

- As rotinas de interrupção são mecanismos de baixo nível. Devem ser **muito curtas** e **por regra** não devem permitir ser interrompidas (mas há essa possibilidade)
- O seu papel é **essencialmente assinalar a ocorrência** da interrupção, desencadeando ações a executar por software de mais alto nível (e não diretamente por elas)

Rotina de interrupção (nível)

```
        . . .                       ; dados, pilha

tab:    WORD rot0                   ; tabela de interrupções (neste caso só tem uma WORD)

        PLACE   0                   ; o código tem de começar em 0000H
        MOV     SP, SP_inicial      ; inicializa SP
        MOV     BTE, tab            ; inicializa BTE
        MOV     R0, 2
        MOV     RCN, R0              ; interrupção 0 sensível ao nível 1 (e não ao flanco)
                                     ; RCN (Registo de Controlo do Núcleo)
        MOV     R0, 0               ; inicializa contador
        EI0                         ; permite interrupção 0
        EI                          ; permite interrupções (geral)
fim:    JMP     fim                 ; fica à espera

rot0:                               ; rotina que trata da interrupção 0 (incrementa contador)
        PUSH    R1
        ADD     R0, 1               ; incrementa contador
        MOV     R1, DISPLAYS        ; endereço do periférico
        MOVB    [R1], R0            ; atualiza display
        POP     R1
        RFE                         ; regressa da interrupção
```

interrupção-nível.asm

Programação concorrente