

# Passagem de argumentos na chamada de função e cuidados

```
section .text

global _start

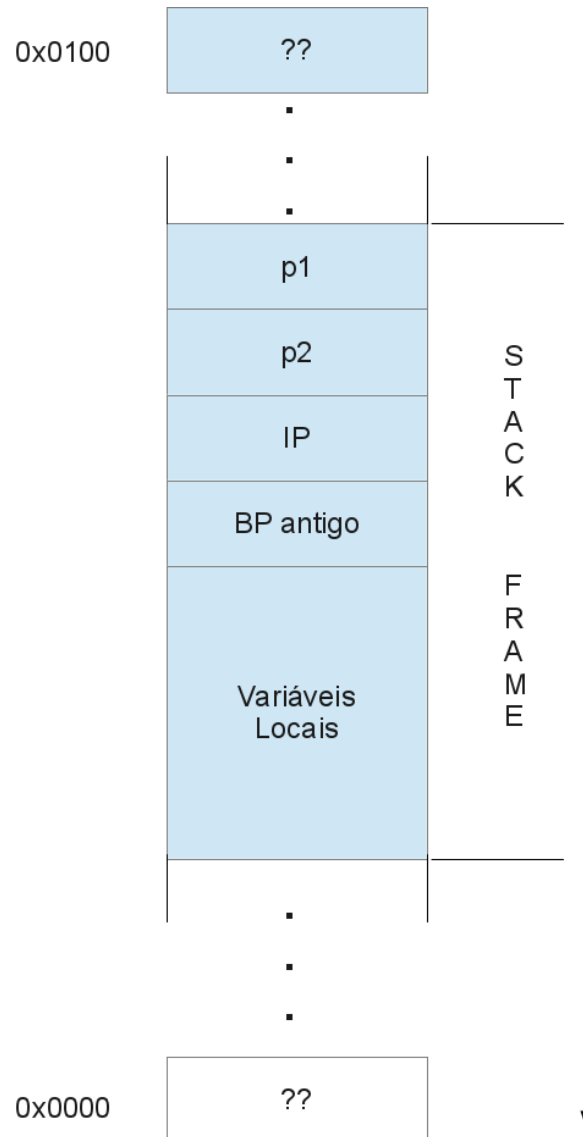
_start:

push 4      ;empilha 1º argumento
push 5      ;empilha 2º argumento
call soma   ;empilha IP de retorno e chama a função soma
mov rcx,rcx
;exit(0)
mov eax,1
mov ebx,0
int 0x80

soma:
;ao chegar aqui já guardou o RIP na pilha, -8 bytes
push rbp    ;salva rpb, -8 bytes
;usa-se o bp para guardar o valor de sp no stack frame
;o bp é a base para aceder os argumentos da pilha
mov rbp,rsp
mov rax,[rbp+24] ;1o arg
add rax,[rbp+16] ;2o arg
;caso precisasse o sp pode ser modificado á vontade
mov rsp,rbp
pop rbp     ;recupera bp antigo
ret        ;salta para o endereço de retorno
```

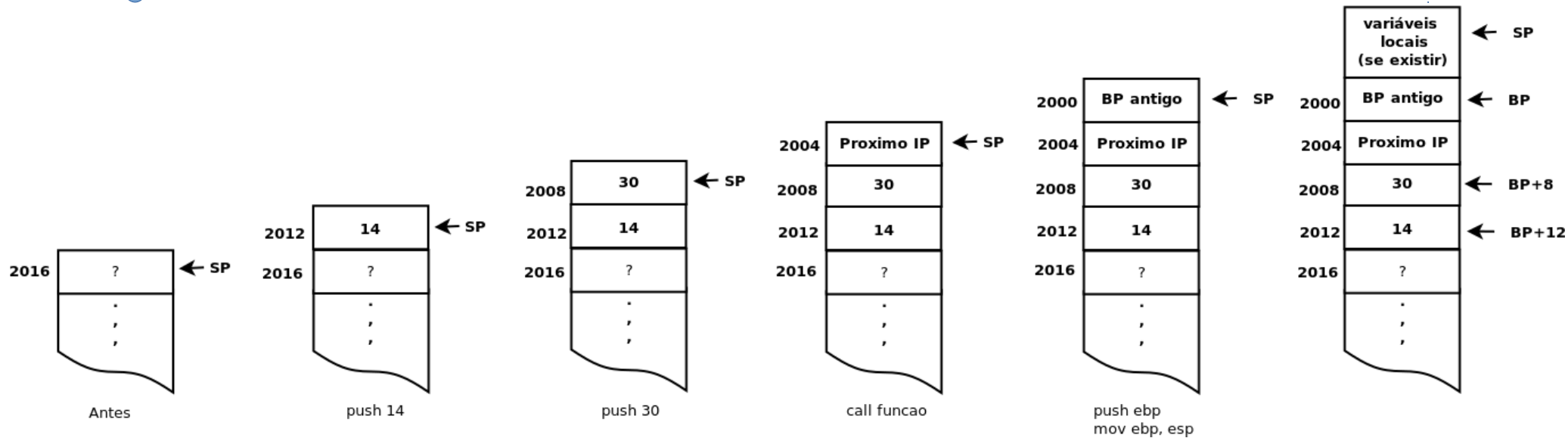
- No interior de uma função, BP é usado para guardar o valor de SP.
- Desta forma, BP pode ser usado para aceder aos argumentos da função, que estão na pilha (stack).
- Chama-se stack frame tudo que está armazenado desde os argumentos até onde SP aponta.
- BP é chamado de base pointer ou frame pointer porque contém o endereço da base do stack frame.
- BP não deve mudar de valor durante a execução da função chamada.
- BP deve ser restaurado ao valor original quando a função termina.
- A instrução RET deve ser usada para devolver o controle da execução de volta ao procedimento que chamou.
- **Como o processador sabe o endereço de retorno?**
  - Ele usa o endereço de retorno colocado na pilha como parte da execução da instrução CALL.
- É fundamental que o topo da pilha esteja apontando para o endereço de retorno na execução da instrução RET.

# Stack frame

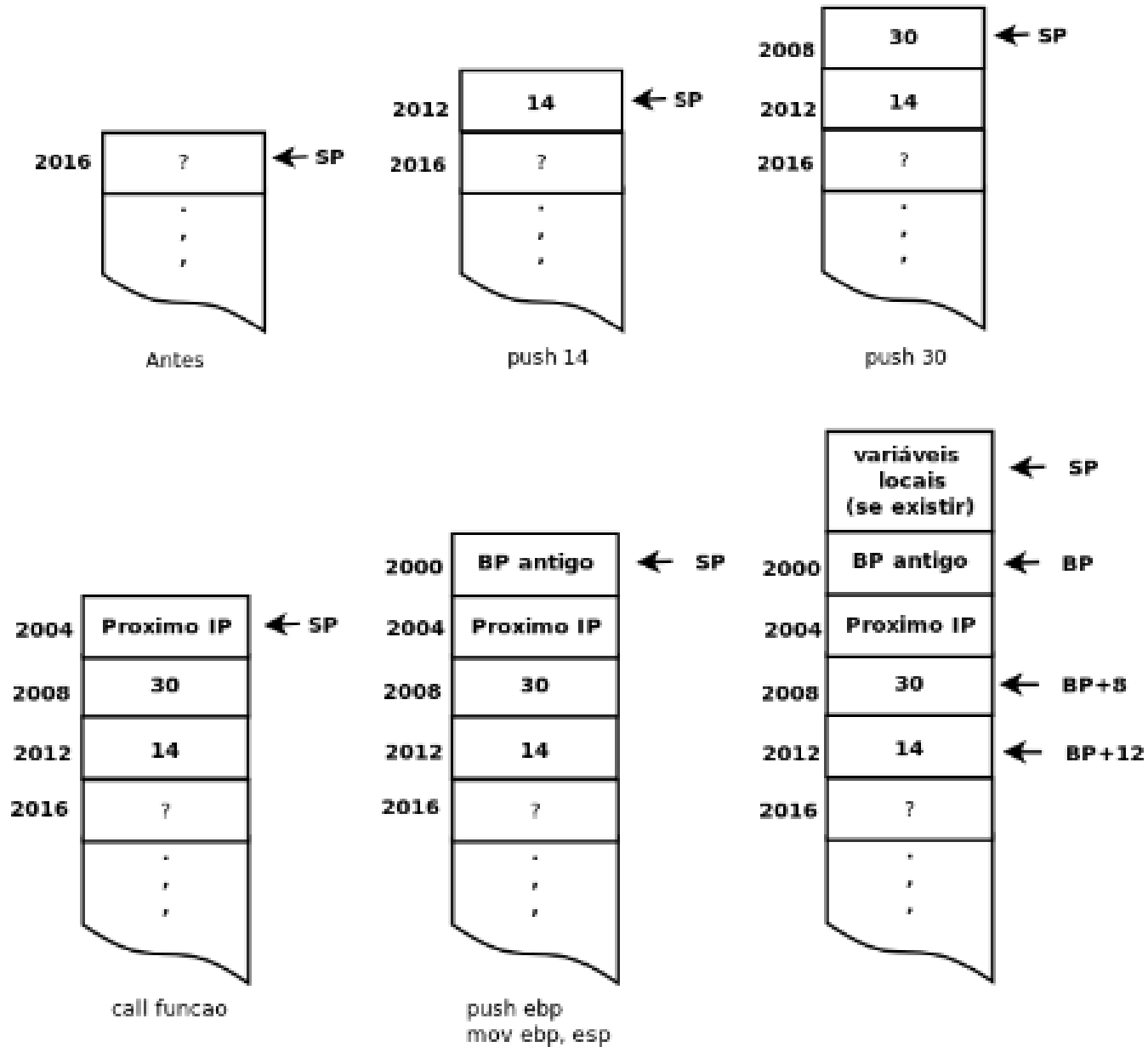


# Evolução da Pilha com passagem de argumentos na função

A pilha decresce o seu endereço, repare que começa no endereço 2016 e termina no 2000 (ou menos que 2000).



A pilha decresce o seu endereço, repare que começa no endereço 2016 e termina no 2000 (ou menos que 2000).



# Como seria no EDB?

The screenshot shows the EDB debugger interface with the following components:

- Assembly View:** Displays assembly instructions. The instruction `ret` at address `004000a5` is highlighted in yellow. A blue oval highlights the instruction `mov rbp, rsp` at address `00400099`. A red box highlights the instruction `push 4` at address `00400084`.
- Registers:** Shows the state of registers. The RIP register is highlighted with a red box and contains the value `0000000000400099`.
- Stack:** Shows the stack contents. The return address `0000000000400099` is highlighted with a red box. A blue oval highlights the instruction `mov rbp, rsp` in the assembly view, which corresponds to the return address on the stack.
- Annotations:**
  - A blue arrow points from the text "Código da função" to the assembly view.
  - A red arrow points from the text "Próximo IP após voltar da função" to the RIP register.
  - A red arrow points from the text "Próximo IP após voltar da função" to the return address on the stack.
  - A blue arrow points from the text "Próximo IP após voltar da função" to the instruction `mov rbp, rsp` in the assembly view.

# Comandos ENTER e LEAVE

- Digamos que queremos reservar 8 bytes na pilha para as variáveis locais da função:

```
push rbp
mov rbp, rsp
sub rsp,8; reserva 8 bytes
```

- Tal sequencia pode ser substituido por:

```
enter 8,0
```

- Para retornar ao IP do stack frame:

```
mov rsp,rbp
pop rbp
```

- Também pode ser substituído por:

```
leave
```

```
section .text

global _start

_start:

push 4           ;empilha 1º argumento
push 5           ;empilha 2º argumento
call soma       ;empilha IP de retorno e chama a função soma
mov rcx,rcx
;exit(0)
mov eax,1
mov ebx,0
int 0x80

soma:
enter 0,0
mov rax,[rbp+24] ;1o arg
add rax,[rbp+16] ;2o arg
;caso precisasse o sp pode ser modificado á vontade
leave
ret             ;salta para o endereço de retorno
```

# Comandos ENTER e LEAVE

- Vimos que o primeiro parâmetro de ENTER cria um stack frame e LEAVE destrói um stack frame.
- Mas e o segundo parâmetro de ENTER?
  - Linguagens como Pascal permitem funções aninhadas, conforme a figura ao lado.
  - Em um caso como este, Y tem acesso não só às suas próprias variáveis locais, mas também a todas as variáveis locais de X. Estas podem ser aninhadas a profundidade arbitrária, então você poderia ter um Z dentro de Y que tivesse acesso às suas próprias variáveis locais e também das variáveis de Y e de X. O segundo parâmetro a inserir especifica a profundidade de aninhamento, então X usaria `enter Sx, 0`, Y usaria `enter Sy, 1` e Z usaria `enter Sz, 2` (onde Sx, Sy e Sz significam o tamanho das variáveis locais para X, Y e Z, respectivamente).
  - Isso criaria uma cadeia de stack frames para que Z tivesse acesso a variáveis locais de Y e X, e assim por diante. Isso não é muito trivial se as funções são recursivas, então uma invocação de Z não pode simplesmente subir a pilha para os dois stack frames mais recentes - ele precisa saltar para stack frames de invocações anteriores de si mesmo e diretamente retornar aos stack frames função / procedimento pai, que é diferente do seu chamador no caso de recursão.
  - Essa complexidade é também a razão pela qual C e C++ *proíbem funções aninhadas*. Dada a presença de enter / leave, eles são bastante fáceis de suportar em processadores Intel, mas podem ser consideravelmente mais difíceis em muitos outros processadores que não possuem suporte direto. Isso, pelo menos, ajuda a explicar uma outra ... característica de enter - para o caso trivial que está sendo usado aqui (ou seja, `enter 0, 0`) é um pouco mais lento que o equivalente usando `push / mov`.

```
procedure X;  
  procedure Y;  
  begin  
    {...}  
  end;  
begin  
  {...}  
end;
```

# RET com parâmetro opcional

- Na figuras mostradas, foram empilhados dois argumentos com valores 14 e 30, supondo que cada empilhamento gastasse 4 bytes.
- Temos dois modos de retirar esses argumentos da pilha:
  - Após o retorno desempilhar esses argumentos com duas instruções pop.
  - Na função utilizar RET 8. Isso, levaria o SP a não mais considerar os argumentos como empilhados.
  - Tomar muito cuidado com isso.