

Procedimentos

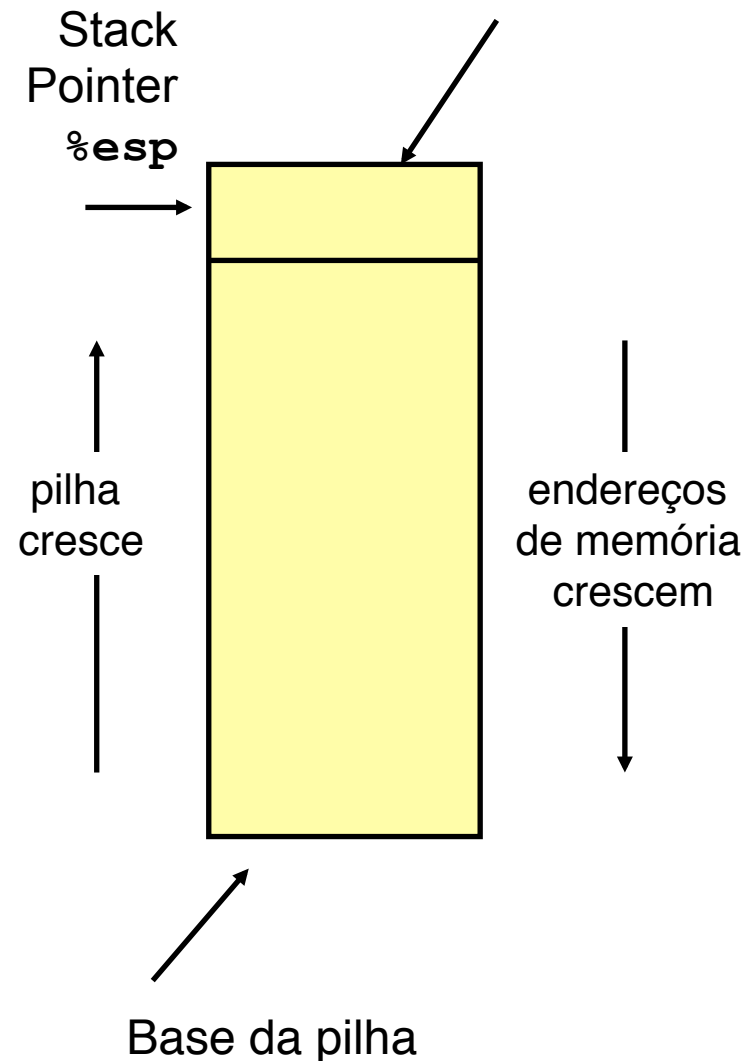
Chamada de funções e passagem de parâmetros

Procedimentos

- Ao chamarmos um procedimento/função passamos dados & controle de uma parte do código para outra
 - chamador ↔ chamado
 - parâmetros e valor de retorno
- A arquitetura IA32 (a maioria das arquiteturas) dá suporte à implementação de procedimentos com base em **instruções para transferência de controle** e no uso de uma **pilha**
 - memória auxiliar durante a execução de uma aplicação

Pilha de execução

- Registrador dedicado para apontar o topo da pilha: **%esp**
 - instruções para inserir (**push**) e remover (**pop**) elementos
- A pilha “cresce” em direção a endereços menores da memória
- A unidade de alocação é uma palavra (4 bytes)
 - para alocar espaço subtraímos 4 de %esp
 - para liberar espaço somamos 4 a %esp



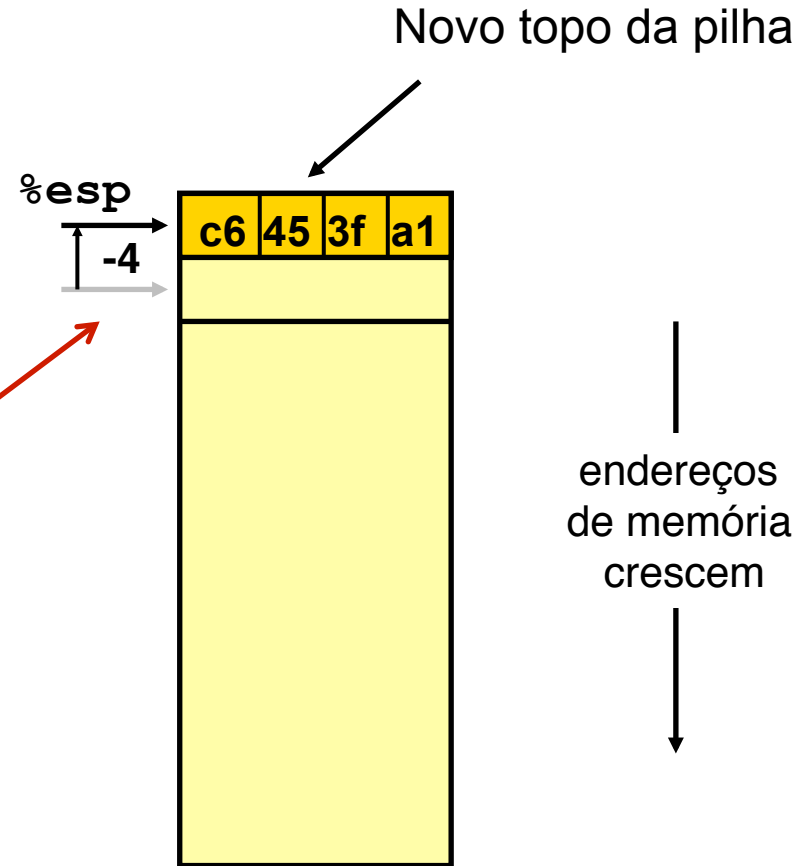
Operação push: empilha

- `pushl SRC` $\left\{ \begin{array}{l} \text{subl } \$4, \%esp \\ \text{movl } SRC, (\%esp) \end{array} \right.$

`%eax = 0xa13f45c6`

`pushl %eax`

Base da pilha

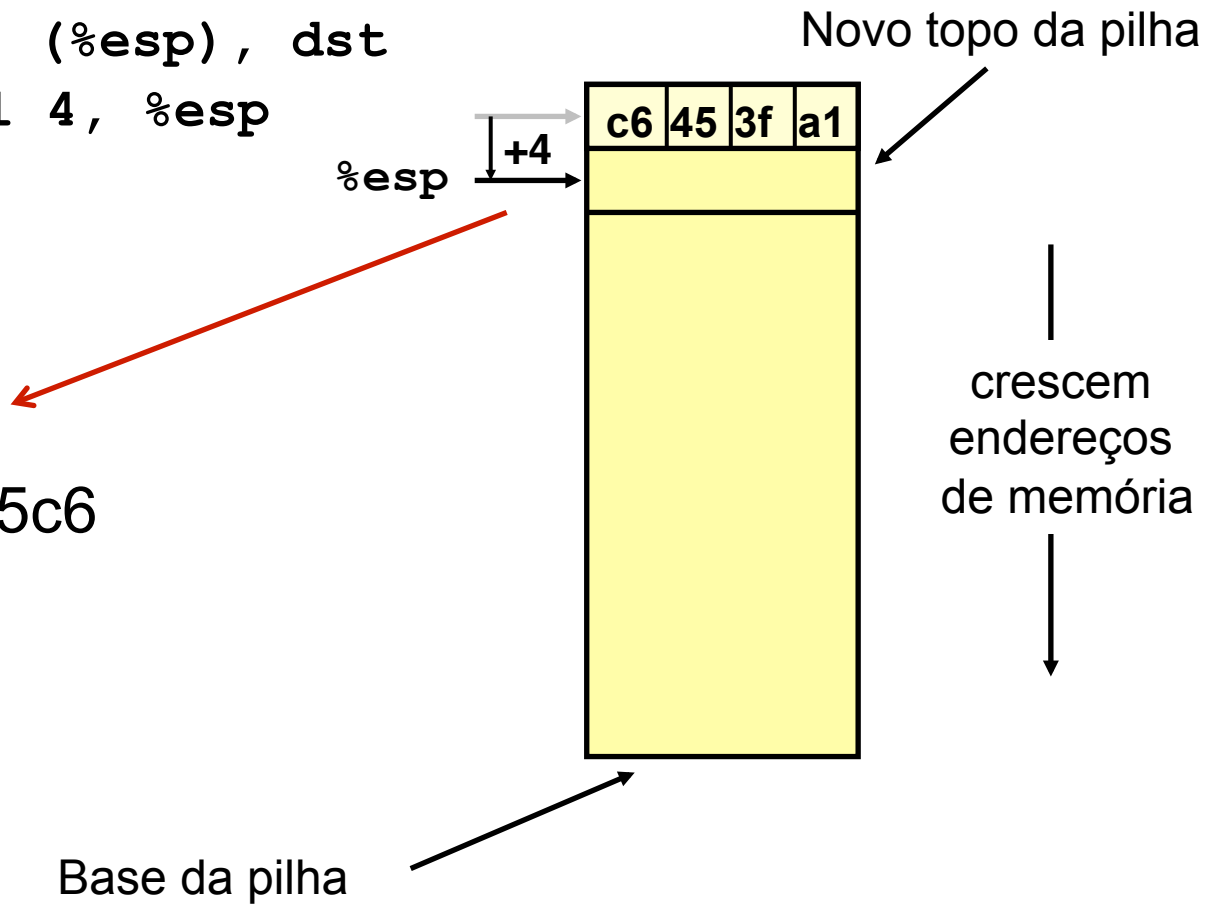


Operação pop: desempilha

- `popl dst` $\left\{ \begin{array}{l} \text{movl } (\%esp), \text{ dst} \\ \text{addl } 4, \%esp \end{array} \right.$

`popl %eax`

`%eax = 0xa13f45c6`



Transferência de Controle

- Duas instruções realizam a transferência de controle: **call label** e **ret**
 - é necessário guardar o **endereço de retorno** para que o controle possa voltar para esse endereço ao final da execução da função
 - esse endereço é o da instrução **seguinte** ao call !

```
        call  funcao1  
→ movl  %eax, (%ebx)
```

- O endereço de retorno é armazenado (pelo hw) na pilha

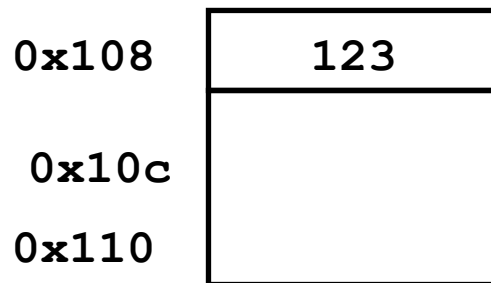
Chamada e Retorno

- A instrução **call** *label* transfere o controle para uma função
 - insere (push) o endereço de retorno na pilha antes de transferir o controle para a função
 - transferência é o desvio para o endereço de *label*
- A instrução **ret** retorna o controle ao chamador
 - retira (pop) o endereço da pilha e transfere o controle (desvia) para ele
- As duas instruções alteram, portanto, `%esp`

Exemplo de chamada

```
804854e: call 8048b90 <sum>
8048553: movl %eax, (%ebx)
```

Antes do call



`%esp`

0x108

`%eip`

0x804854e

Depois do call

0x104

0x8048553

0x108

123

0x10c

0x110

`%esp`

0x104

`%eip`

0x8048b90

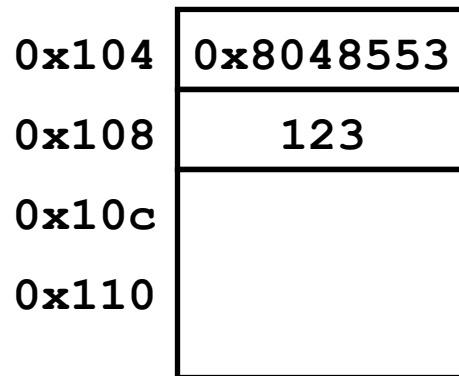
Pilha
(na memória)

Registradores
na CPU

Exemplo de Retorno

8048591: ret

Antes do ret



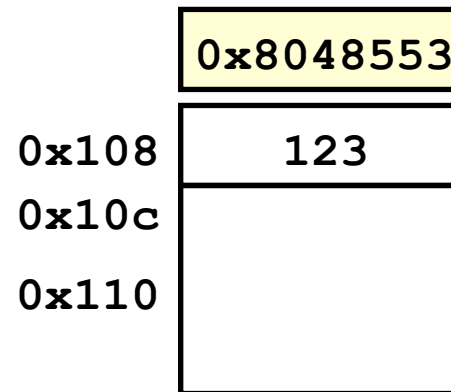
%esp

0x104

%eip

0x8048591

Depois do ret



%esp

0x108

%eip

0x8048553

Pilha
(na memória)

Registradores
na CPU

Valor de retorno e passagem de parâmetros

- instruções **call** e **ret** só transferem controle
 - não cuidam de passagem de valores!
 - em programa printnum, passagem de parâmetro em registrador...
 - não é adequada quando há muitos parâmetros!
- passagem de parâmetros pela pilha
- retorno de valor por %eax (inteiro ou endereço)

Passagem de parâmetros

- convenção em C: chamador empilha os parâmetros na ordem inversa em que aparecem na declaração do procedimento (i.e., da direita para a esquerda)

`int teste(int tam, int nums[])`

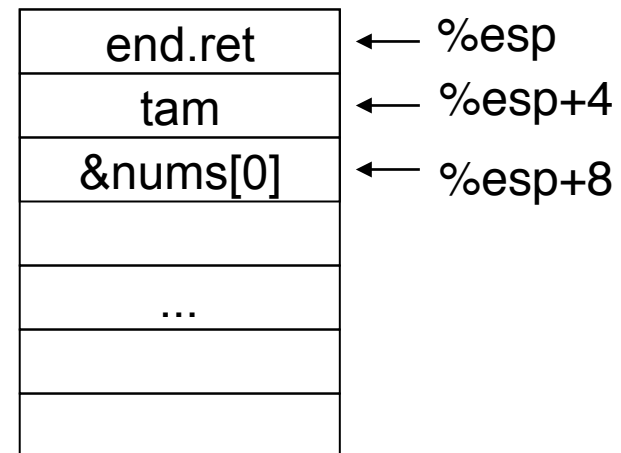


Empilhamento dos parâmetros

- No momento de ativação da função (após chamada):

```
int teste(int tam, int nums[])
```

- Por que não usar `%esp` para acessar os parâmetros?
 - durante a execução da função o valor se `%esp` pode variar!
- o registrador **`%ebp`** (***frame pointer***) é usado para acesso aos parâmetros
 - base do **registro de ativação** (a porção da pilha associada a uma chamada de função)



Acesso aos parâmetros

- Convenção (compilador C)

- Início de procedimento:

```
pushl %ebp → salva ebp-ant
movl  %esp, %ebp → novo RA
```

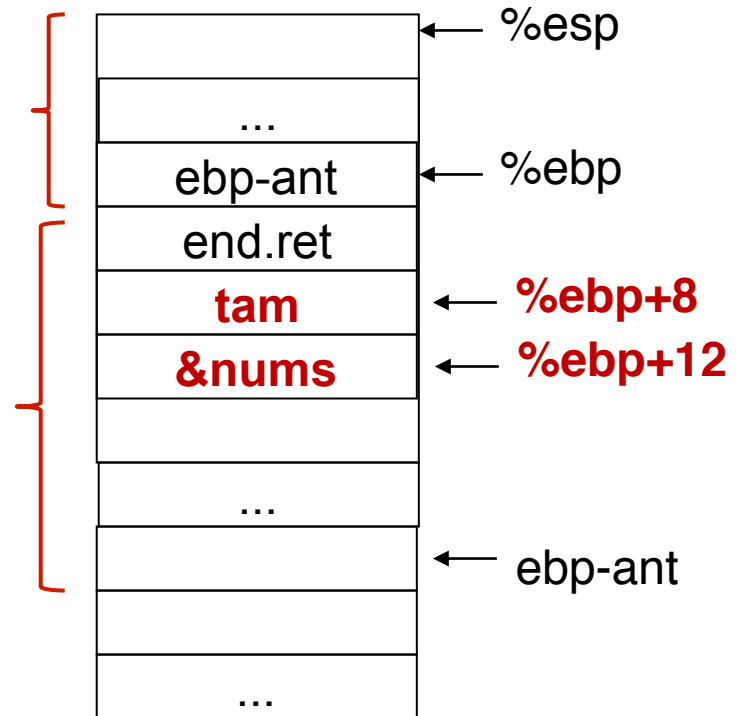
- Durante o procedimento:
parâmetros são acessados via

```
%ebp+<deslocamento>
```

- Fim de procedimento:

```
movl %ebp, %esp
popl %ebp → restaura ebp-ant
ret
```

Durante a execução da função

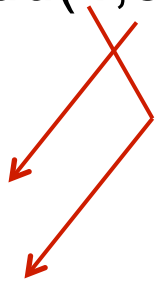


- pilha de execução também chamada de pilha de registros de ativação! 13

Exemplo

x = add(4,5)

```
...  
pushl $5  
pushl $4  
call add  
addl $8, %esp  
movl %eax, %ecx ;retorno  
...
```

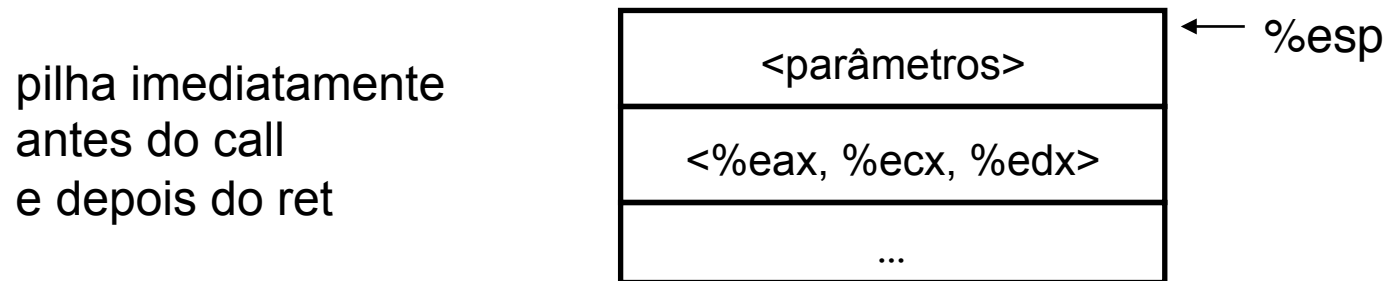


```
int add(int a, int b) {  
    return a+b;  
}
```

```
add:  
    push %ebp  
    movl %esp, %ebp  
    movl 8(%ebp), %eax /* a */  
    addl 12(%ebp), %eax /* b */  
    movl %ebp, %esp  
    popl %ebp  
    ret
```

Observações importantes

- É tarefa do chamador desempilhar os parâmetros após o retorno da função chamada
 - instrução **pop** ou soma de um valor a **%esp**



- Cada parâmetro ocupa uma palavra (4 bytes), mesmo que seu tipo ocupe um tamanho menor
 - instruções para estender a representação

Valores dos Registradores

- Funções usam registradores para armazenar valores (variáveis locais, temporários)
 - a “chamadora” (*caller*) e a “chamada” (*callee*)
- A pilha é usada para salvar o conteúdo de registradores
 - evita sua perda numa chamada de função
 - mas quem armazena: caller? callee?
- Cada sistema tem uma **convenção** para que
 - registradores em uso sejam "salvos" exatamente uma vez

Convenção de Salvamento de Registradores

- No Linux:
 - Caller (chamadora) deve salvar: **%eax**, **%ecx** e **%edx**
 - armazenar na pilha antes de preparar a chamada da função
 - Callee (chamada) deve salvar: **%ebx**, **%esi** e **%edi**
 - pode sobrescrever **%eax**, **%ecx** e **%edx**
 - porque a chamadora salvou
 - **%ebx**, **%esi** e **%edi** devem ser salvos na pilha, se forem usados
 - e restaurados antes do retorno!

Exemplo

```
/*  
%ecx usado pela chamadora  
%edx tem o valor do parâmetro  
*/  
...  
pushl %ecx ;salva reg  
pushl %edx ;parametro  
call g  
addl $4, %esp  
popl %ecx  
;o retorno da fç está em eax:  
movl %eax, (%ecx)  
...
```

```
g:  
push %ebp  
movl %esp, %ebp  
push %ebx  
movl 8(%ebp), %eax  
movl $1, %ebx  
...  
movl %ebx, %eax  
popl %ebx  
movl %ebp, %esp  
popl %ebp  
ret
```