Programação em Sistemas Computacionais PSC

Assembly

Arquitectura IA32

Centro de Cálculo
Instituto Superior de Engenharia de Lisboa

Pedro Pereira palex@cc.isel.ipl.pt

Exemplo com jump

Sabendo que:

- **A variável x está no registo edx**

incMod10.c

```
int x;
…
++x;
if ( x==10 )
  x=0;
…
```

```
  …
  add   edx, 1
  cmp   edx, 10
  jne   .L1
  xor   edx, edx
.L1:
  …
```

```
++x;
zf = x==10;
if(!zf) goto L1;
x = 0;
L1:
```

-O2

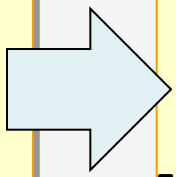
```
  …
  xor   eax, eax
  add   edx, 1
  cmp   edx, 10
  sete  al
  dec   eax
  and   edx, eax
  …
```

?

```
eax = 0;
++x;
zf = x==10;
al = zf;
--eax;
x &= eax;
```

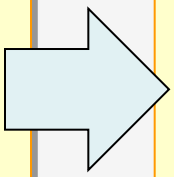
If-else

```
...
if (E)
  A;
...
```



```
...
t = E;
if (!t)
 goto Lelse;
A;
Lelse:
...
```

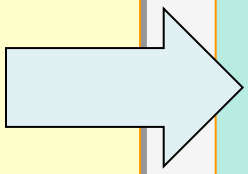
```
...
if (E)
  A;
else
  B;
...
```



```
...
t = E;
if (t)
 goto Lthen;
B;
goto Lend;
Lthen:
A;
Lend:
...
```

absdiff.c

```
int absdiff(int x, int y) {
  if (x<y)
    return y-x;
  else
    return x-y;
}
```

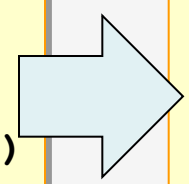


```
  …
 mov edx, [ebp+8]
 mov eax, [ebp+12]
 cmp edx, eax
 jl  .L6
 sub edx, eax
 mov eax, edx
 jmp .L7
.L6:
 sub eax, edx
.L7:
  …
```

```
edx <- x
eax <- y
compare x y
if (x<y) goto Lthen;
y -= x;
eax = y;
goto Lend;
Lthen:
eax -= y;
Lend:
```

Loops (do-while)

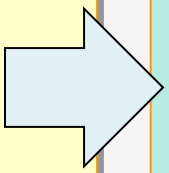
```
...
do
  A;
while(E)
...
```



```
 ...
Loop:
 A;
 t = E;
 if (t)
  goto Loop;
...
```

fibn.c

```
int fibn(int n) {
  int val=0, nval=1, r;
  do {
    r = val + nval;
    val = nval;
    nval = r;
  } while (--n);
  return val;
}
```



```
 …
 mov   edx, [ebp+8]
 mov   ebx, 0
 mov   eax, 1
.L3:
 lea   ecx, [eax+ebx]
 mov   ebx, eax
 mov   eax, ecx
 dec   edx
 jnz   .L3
 mov   eax, ebx
 …
```

```
 ecx <- r
 edx <- n
 ebx <- val
 eax <- nval
Loop:
 r = val + nval;
 val = nval;
 nval = r;
 --n;
 if (n!=0) goto Loop;
```

Loops (while e for)

```
...
while(E)
  A;
...
```

```
 ...
Loop:
 t = E;
 if (!t)
  goto End;
 A;
 goto Loop;
End:
 ...
```

Evitar 2 jmps dentro do loop

```
 ...
 t = E;
 if (!t)
  goto End;
Loop:
 A;
 t = E;
 if (t)
   goto Loop;
End:
 ...
```

```
...
for(I;E;N)
  A;
...
```

```
 ...
 I;
Loop:
 t = E;
 if (!t)
  goto End;
 A;
 N;
 goto Loop;
End:
 ...
```

Evitar 2 jmps dentro do loop

```
 ...
 I;
 t = E;
 if (!t)
  goto End;
Loop:
 A;
 N;
 t = E;
 if (t)
   goto Loop;
End:
 ...
```

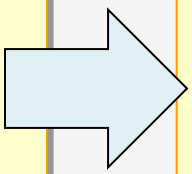
Switchs

switch.c

```
int switch_ex(int x) {
  int res=x;

  switch(x) {

  case 100:
    res*=13; break;
  case 102:
    res+=10; /*next*/
  case 103:
    res+=11; break;
  case 104:
  case 106:
    res*=res; break;
  default: res=0;
  }
  return res;
}
```



```
Code *jt[] = {  /* Not in C */
  L100, Ldef, L102, L103,
  L104, Ldef, L104 };
int switch_ex_jt(int x) {
  int res=x;
  unsigned xi=x-100;
  if (xi>6) goto Ldef;
  goto jt[xi]; /* Not in C */
L100:
  res*=13; goto Lend;
L102:
  res+=10; /*next*/
L103:
  res+=11; goto Lend;
L104:
  res*=res; goto Lend;
Ldef:
  res=0;
Lend:
  return res;
}
```

Switchs

```
Code *jt[] = { /* Not in C */
  L100, Ldef, L102, L103,
  L104, Ldef, L104 };
int switch_ex_jt(int x) {
  int res=x;
  unsigned xi=x-100;
  if (xi>6) goto Ldef;
  goto jt[xi]; /* Not in C */
L100:
  res*=13; goto Lend;
L102:
  res+=10; /*next*/
L103:
  res+=11; goto Lend;
L104:
  res*=res; goto Lend;
Ldef:
  res=0;
Lend:
  return res;
}
```

```
.section .rodata
.align 4
.L7:
 .long  .L3
 .long  .L2
 .long  .L4
 .long  .L5
 .long  .L6
 .long  .L2
 .long  .L6
```

```
eax <- res
edx <- xi
```

```
  …
  lea   edx, [eax-100]
  cmp   edx, 6
  ja    .L2
  jmp   [.L7+edx*4]
.L3:
  imul  eax, 13
  jmp .L1
.L4:
  add   eax, 10
.L5:
  add   eax, 11
  jmp .L1
.L6:
  imul  eax, eax
  jmp .L1
.L2:
  xor   eax, eax
.L1:
  …
```

Call & Return

- **Chamada a função**

  Coloca no topo do stack (push) o endereço de retorno (da instrução a seguir ao call) e a execução continua no endereço indicado.

  – Directa: call L

  ```
  call .func
  ```

  ```
  push (ip+N)
  jmp L
  ```

  Endereço indicado com valor imediato.

  – Indirecta: call S

  ```
  call [ebx+10]
  ```

  Endereço armazenado no registo

  ou posição de memória indicada

- **Retorno da chamada**

  A execução continua no o endereço de retorno retirado do topo do stack (pop)

  – ret

  ```
  ret
  ```

  ```
  pop (L)
  jmp L
  ```

  O valor a retornar fica em eax

Exemplo de chamada simples

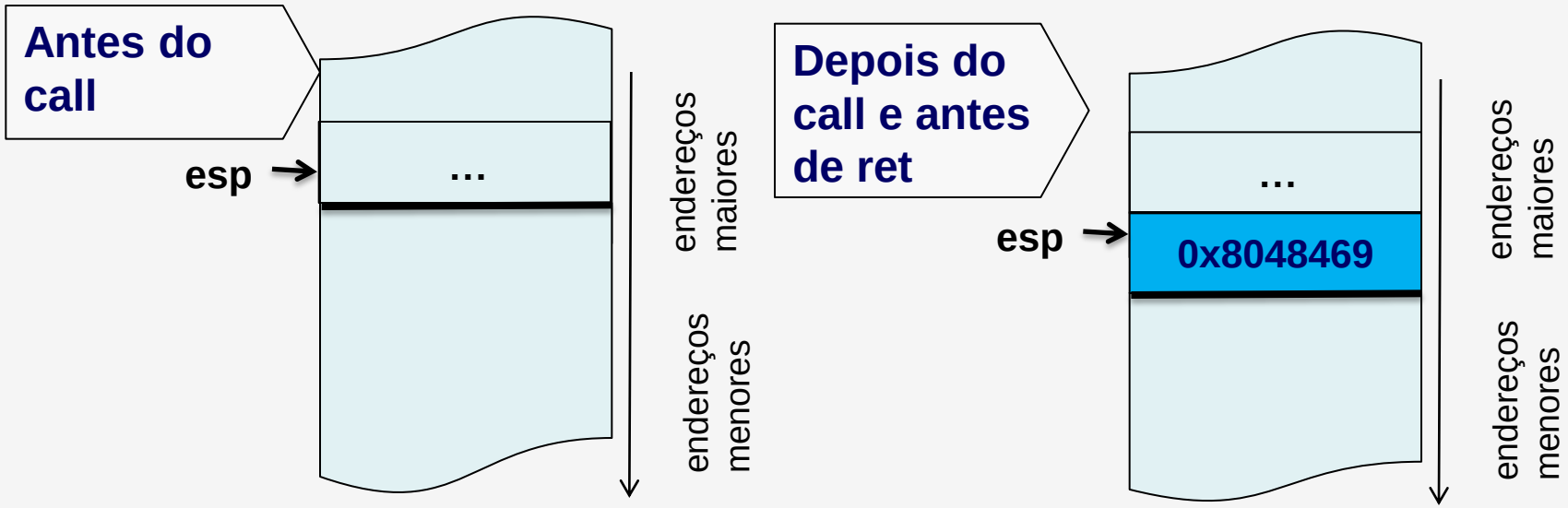
```
int func() {
  return 10;
}
```

```
08048444 <func>:
8048444: b8 0a 00 00 00   mov  eax,0xa
8048449: c3               ret

 …
8048464: e8 db ff ff ff   call 8048444 <func>
8048469: 01 c3            add  ebx,eax
 …
```

```
  …
  x+=func();
  …
```

ebx <- x



Passagem de parâmetros e retorno

- **Parâmetros empilham-se no stack antes do endereço de retorno**
- **Da direita para a esquerda (o 1º parâmetro é o último a empilhar)**
- **Parâmetros menores que 32 bits ocupam também 4 bytes no stack**
- **O retorno fica em al, ax ou eax conforme o tipo (<=32 bits)**
- **O chamador desempilha os parâmetros (ajusta o stack)**

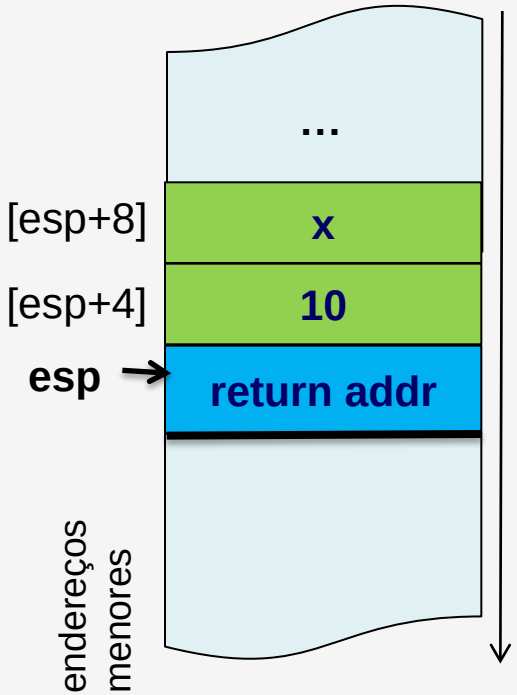
```
int sum(int a, int b) {
   return a+b;
}
```

```
   …
   x+=sum(10,x);
   …
```

ebx <- x

```
sum:
 mov eax, [esp+4]
 add eax, [esp+8]
 ret

…
 push ebx
 push 10
 call sum
 add  esp, 8
 add  ebx, eax
…
```



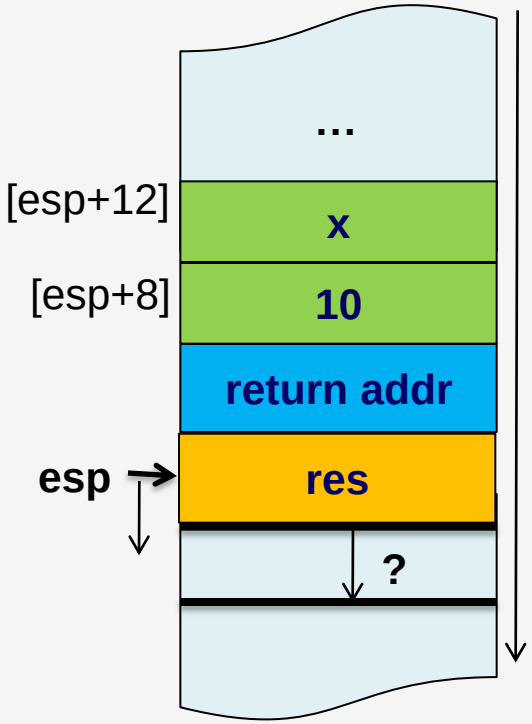
Variáveis locais

- **As variáveis locais também são alojadas em stack (depois do endereço de retorno)**
  - Mas assim, o offset para acesso aos parâmetros depende da dimensão das variáveis locais.
  - O registo esp pode variar durante a execução da função (push,pop…) ?

```
int sum(int a, int b) {
   int res=a;
   res +=b;
   …
   return res;
}
```

```
   …
   x+=sum(10,x);
   …
```

```
sum:
 sub esp, 4
 mov eax, [esp+8]
 mov [esp], eax
 add eax, [esp+12]
 mov [esp], eax
 …
 mov eax, [esp]
 add esp, 4
 ret
```



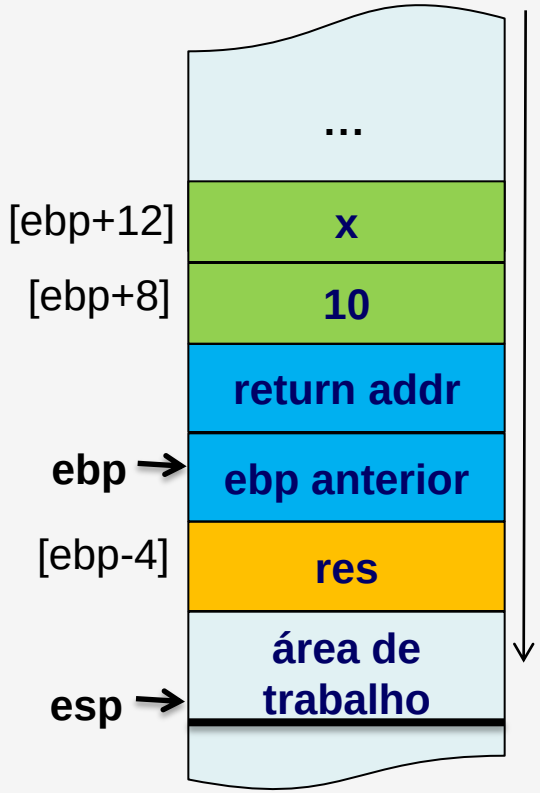
Utilização de base pointer (ebp)

- **Para o acesso aos parâmetros e variáveis locais não ficar dependente de esp usa-se o registo ebp**
  - Mas é necessário preservar o valor anterior de ebp.

```
int sum(int a, int b) {
  int res=a;
  res +=b;
  …
  return res;
}
```

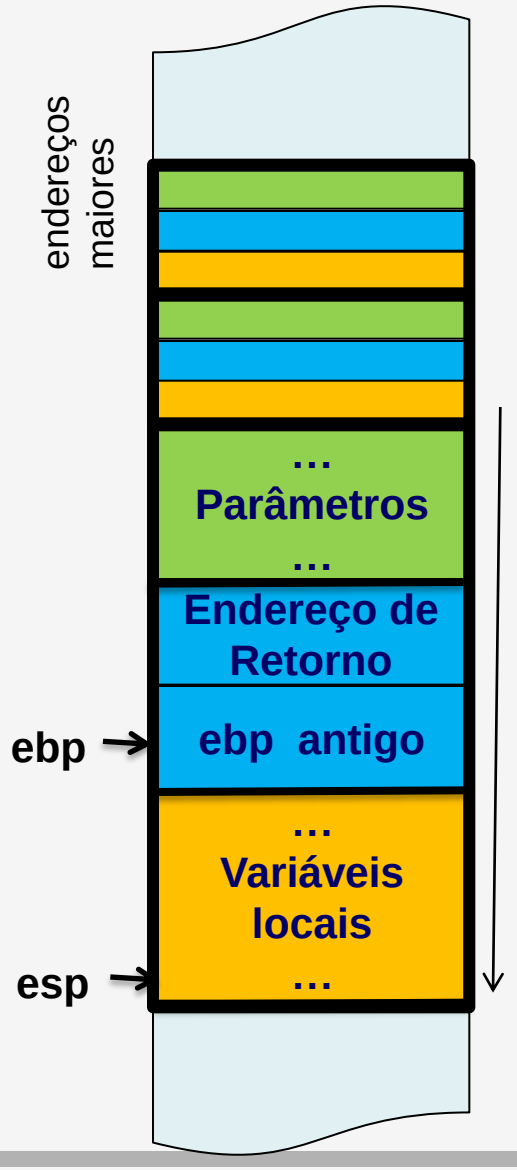
```
  …
  x+=sum(10,x);
  …
```

```
sum:
  push  ebp
  mov   ebp, esp
  sub   esp, ???
  mov   eax, [ebp+8]
  mov   [ebp-4], eax
  add   eax, [ebp+12]
  mov   [ebp-4], eax
  …
  mov   eax, [ebp-4]
  mov   esp, ebp
  pop   ebp
  ret
```



stack frame

- **Chamada**
  1. Passagem de parâmetros
     - O chamador empilha os parâmetros da direita para a esquerda
  2. Chamada
     - O endereço de retorno é empilhado pela instrução `call`
  3. Acertar `ebp` (*base pointer*) e alojar as variáveis locais
     - A função chamada empilha o valor de `ebp`
     - copia o valor de `esp` para `ebp`
     - reserva espaço para as variáveis locais ajustando o valor de `esp`
- **Retorno**
  4. Valor de retorno
     - A função chamada deixa em `eax` o valor de retorno
  5. Libertar espaço das variáveis locais e repor `ebp`
     - A função chamada ajusta o `esp`
     - repõe o valor de `ebp` desempilhando-o
  6. Retornar ao chamador
     - A execução passa para o chamador pela instrução `ret`
  7. Libertar espaço dos parâmetros
     - O chamador ajusta o `esp`



stack frame (2)

chamador

```
void p() {
  int x;
  ...
  x += sum(10,x);
  ...
}
```

```
p:
  ...
  mov   eax, [ebp-4]
1 push  eax
  push  10
2 call  sum
.L1:
7 add   esp, 8
  add   [ebp-4], eax
  ...
```

leave

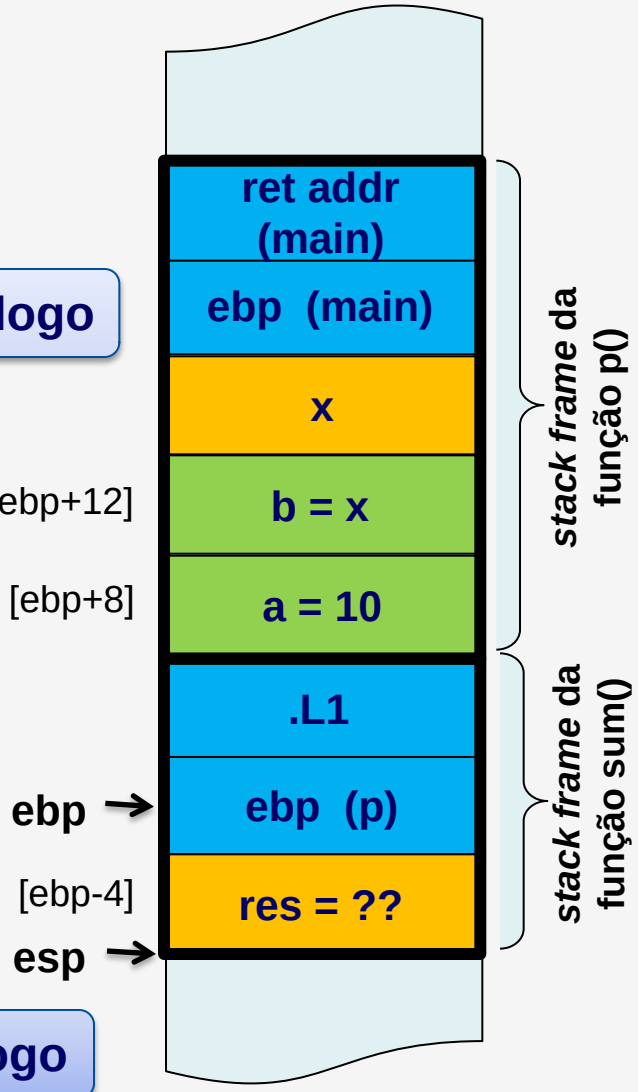
chamado

```
int sum(int a, int b) {
  int res = a;
  res +=b;
  return res;
}
```

Prólogo

```
sum:
3 push  ebp
  mov   ebp, esp
  sub   esp, 4
  mov   eax, [ebp+8]
  mov   [ebp-4], eax
  add   eax, [ebp+12]
  mov   [ebp-4], eax
4 mov   eax, [ebp-4]
5 mov   esp, ebp
  pop   ebp
6 ret
```

Epílogo



Convenção de utilização de registos

- **Salvos pelo chamador: eax, edx, ecx**
  - A função chamada pode usar livremente
- **Salvos pela função chamada: ebx, esi, edi**
  - Se os quiser usar, a função chamada terá que preservar os seus valores no stack
- **O registo eax é usado para o retorno**
  - Se a dimensão do retorno for inferior ou igual a 32 bits
- **Os registos esp e ebp são usados na satck frame**

Ligação entre C e Assembly (Chamada a partir de C)

tswap.c

```
#include <stdio.h>

/* Função para fazer em assembly */
int swap_add(int *x, int *y);

int main() {
  int a = 512;
  int b = 1024;
  int sum;
  sum = swap_add(&a,&b);
  printf("a=%d, b=%d, sum=%d\n",a,b,sum);
  return 0;
}
```

Ligação entre C e Assembly (função em assembly)

swapadd.s

```
  .intel_syntax noprefix
  .globl swap_add
swap_add :
   push ebp
   mov  ebp, esp

   push ebx
   mov  edx, [ebp+8]
   mov  ecx, [ebp+12]
   mov  ebx, [edx]
   mov  eax, [ecx]
   mov  [edx], eax
   mov  [ecx], ebx
   add  eax, ebx
   pop  ebx

   leave
   ret
```

```
int swap_add(int *xp, int *yp) {
   int x = *xp;
   int y = *yp;
   *xp = y;
   *yp = x;
   x += y;
   return x;
}
```

gcc tswap.c swapadd.s –o tswap