- Estrutura de uma função
  - código
    - corpo da função
      - implementa a funcionalidade especificada
    - gestão da função
      - garante o correcto funcionamento da estrutura
  - contexto
    - variáveis globais (em memória)
    - parâmetros (em registo ou *stack*)
    - variáveis locais (em registo ou *stack*)
    - info de suporte à gestão (endereço de retorno, ...)

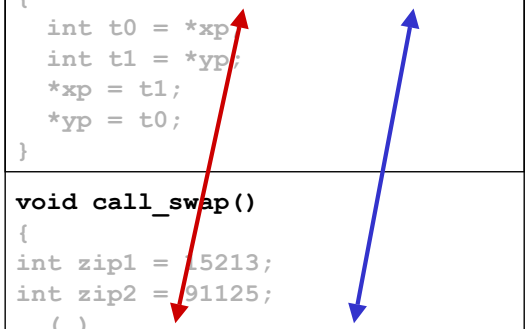
- Análise do contexto de uma função
  - variáveis globais
    - localização definida pelo *linker* / *loader*
  - parâmetros; propriedades:
    - designação independente (chamadora/chamada)
    - deve suportar aninhamento e recursividade
    - localização ideal: em registo, se os houver; mas...
    - localização no IA32: na memória (*stack*)
  - variáveis locais; propriedades:
    - visíveis apenas durante a execução da função
    - deve suportar aninhamento e recursividade
    - localização ideal: em registo, se os houver; mas...
    - localização no IA32: na memória (*stack*)
  - info de suporte à gestão ...

- Análise do código de gestão de uma função
  - invocação e retorno
    - instrução de salto, mas com salvaguarda do end. retorno
      - em registo (RISC; aninhamento / recursividade ?)
      - em memória/*stack* (IA32; aninhamento / recursividade ?)
  - invocação e retorno
    - instrução de salto para o endereço de retorno
  - salvaguarda & recuperação de registos (na *stack*)
    - antes/após a invocação ? (nenhum/ alguns/ todos ? RISC/IA32 ?)
    - antes/após o retorno? (nenhum/ alguns/ todos ? RISC/IA32 ?)
  - gestão do contexto (em *stack*)
    - actualização/recuperação do *frame pointer* (IA32...)
    - reserva/libertação de espaço para variáveis locais

- Análise de exemplos
  - revisão do exemplo `swap`
    - análise das fases: inicialização, corpo, término
    - análise dos contextos (IA32)
    - evolução dos contextos na *stack* (IA32)
    - implementação IA32 *versus* MIPS
  - evolução de um exemplo: `Fibonacci`
    - análise de uma compilação do **`gcc`**
  - aninhamento e recursividade
    - evolução dos contextos na *stack*

Designação independente dos parâmetros

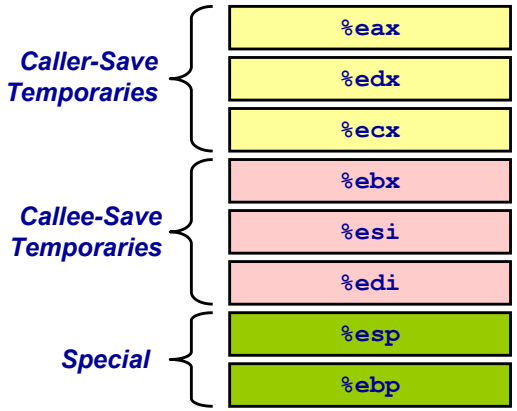
```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}

void call_swap()
{
int zip1 = 15213;
int zip2 = 91125;
  (…)
  swap(&zip1, &zip2);
  (…)
}
```



IA32/Linux Register Usage

(baseado no Bryant, Class07, F'02)

- Registos de Inteiros
  - Dois especiais
    `%ebp, %esp`
  - Três do tipo *callee-save*
    `%ebx, %esi, %edi`
    - valores anteriores colocados na *stack* antes de usar
  - Três do tipo *caller-save*
    `%eax, %edx, %ecx`
    - responsabilidade de salva-guarda da função chamadora
  - Nota: valor de retorno da função em `%eax`



Using Simple Addressing Modes

(Bryant, Class05, F'02)

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    pushl %ebp          } Set Up
    movl %esp,%ebp
    pushl %ebx

    movl 12(%ebp),%ecx  } Body
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)

    movl -4(%ebp),%ebx  } Finish
    movl %ebp,%esp
    popl %ebp
    ret
```


Análise dos contextos em `swap`

(baseado no Bryant, Class07, F'02)

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}

void call_swap()
{
int zip1 = 15213;
int zip2 = 91125;
  (…)
  swap(&zip1, &zip2);
  (…)
}
```

- em `call_swap`
- na invocação de `swap`
- na execução de `swap`
- de volta a `call_swap`

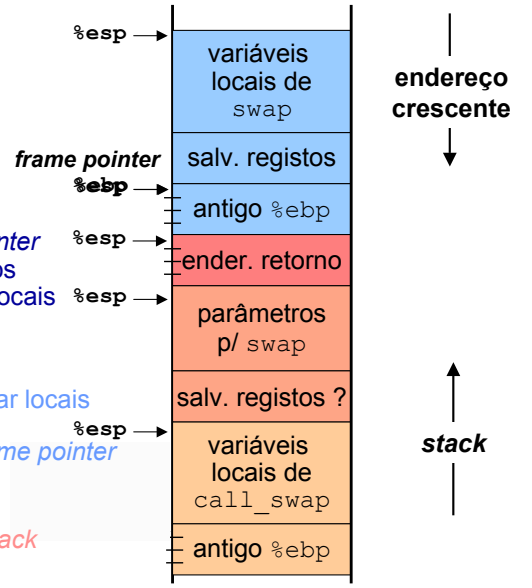
Que contextos (IA32)?

- passagem de parâmetros
  - via *stack*
- espaço para variáveis locais
  - na *stack*
- info de suporte à gestão (*stack*)
  - endereço de retorno
  - apontador para a *stack frame*
  - salvaguarda de registos

Construção do contexto na stack, IA32

call_swap **swap**

1. Antes de invocar `swap`
2. Preparação p/ invocar `swap`
   - •salvaguardar registos?
   - •passagem de parâmetros
3. Invocar `swap`
   - •e guardar endereço de retorno
   1. Início de `swap`
      - •actualizar *frame pointer*
      - •salvaguardar registos
      - •reservar espaço p/ locais
   2. Corpo de `swap`
   3. Término de `swap` ...
      - •libertar espaço de var locais
      - •recuperar registos
      - •recuperar antigo *frame pointer*
      - •voltar a `call_swap`
4. Terminar invocação de `swap`...
   - •libertar espaço de parâmetros na *stack*
   - •recuperar registos?

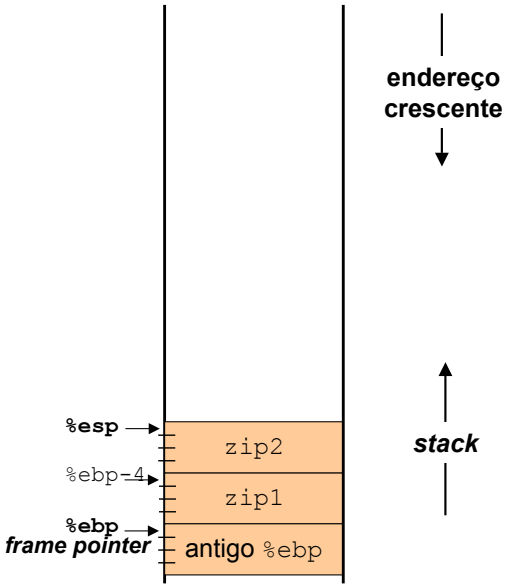


Evolução da stack no IA32 (1)

call_swap

1. Antes de invocar `swap`

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}

void call_swap()
{
int zip1 = 15213;
int zip2 = 91125;
  (…)
  swap(&zip1, &zip2);
  (…)
}
```

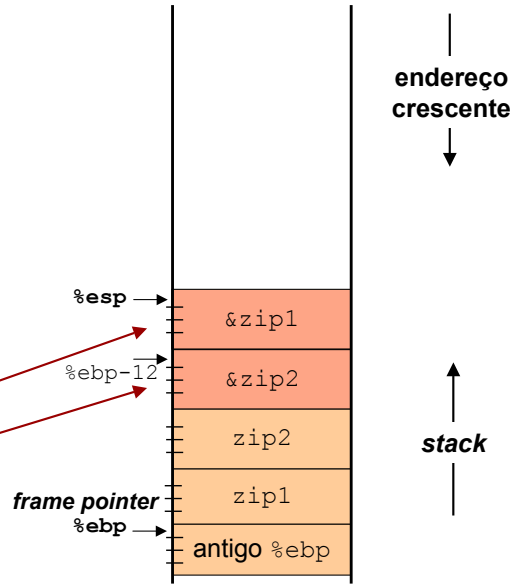


Evolução da stack no IA32 (2)

call_swap

2. Preparação p/ invocar `swap`
   - •salvaguardar registos?...**não**...
   - •passagem de parâmetros

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}

void call_swap()
{
int zip1 = 15213;
int zip2 = 91125;
  (…)
  swap(&zip1, &zip2);
  (…)
}
```

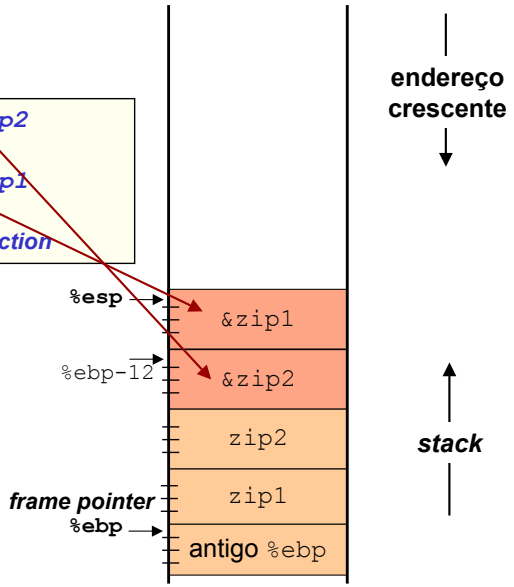


Evolução da stack no IA32 (3)

call_swap

2. Preparação p/ invocar `swap`
   - •salvaguardar registos?...**não**...
   - •passagem de parâmetros

```
leal  -8(%ebp),%eax   Compute &zip2
pushl %eax            Push &zip2
leal  -4(%ebp),%eax   Compute &zip1
pushl %eax            Push &zip1
call  swap            Call swap function
```



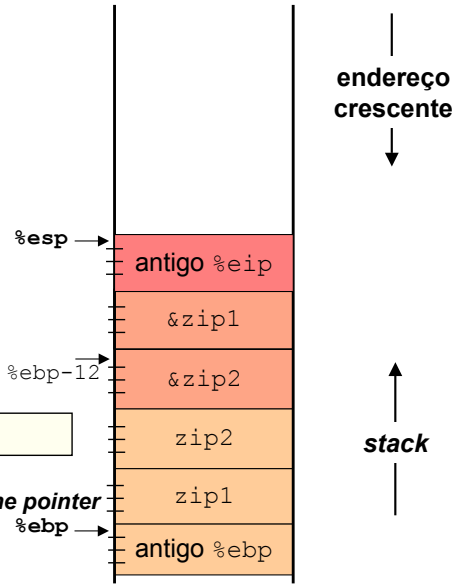
Evolução da stack no IA32 (4)

call_swap

3. Invocar swap
- e guardar endereço de retorno

```
void call_swap()
{
int zip1 = 15213;
int zip2 = 91125;
  (…)
  swap(&zip1, &zip2);
  (…)
}
```

```
call  swap          Call swap function
```



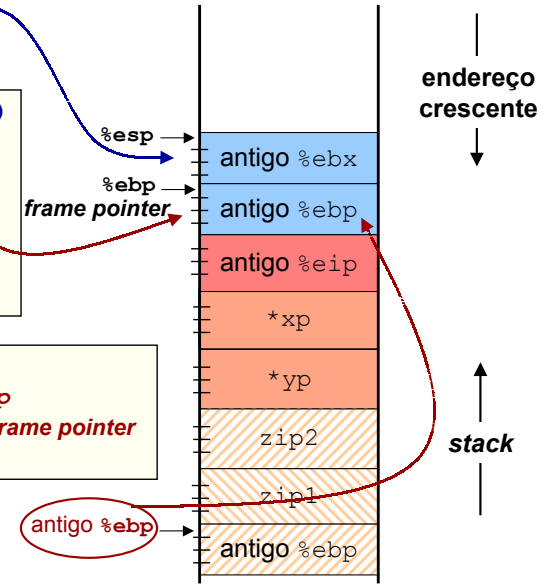
Evolução da stack no IA32 (5)

swap

1. Início de swap
- actualizar *frame pointer*
- salvaguardar registos
- reservar espaço p/ locais...**não**...

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
  pushl  %ebp        Save old %ebp
  movl   %esp,%ebp   Set %ebp as frame pointer
  pushl  %ebx        Save %ebx
```



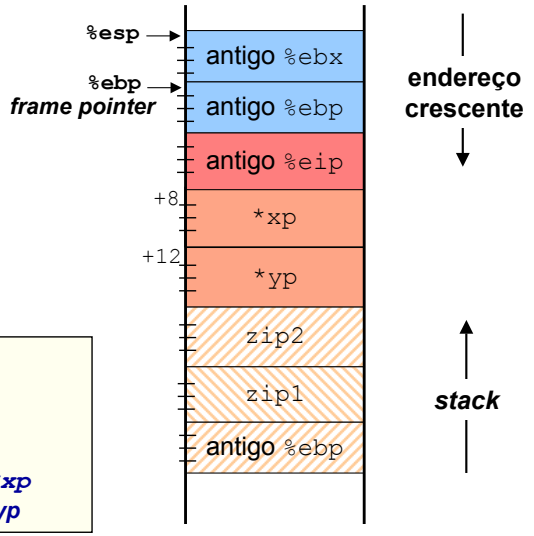
Evolução da stack no IA32 (6)

swap

2. Corpo de swap

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
movl  12(%ebp),%ecx   Get yp
movl  8(%ebp),%edx    Get xp
movl  (%ecx),%eax     Get y
movl  (%edx),%ebx     Get x
movl  %eax,(%edx)     Store y at *xp
movl  %ebx,(%ecx)     Store x at *yp
```



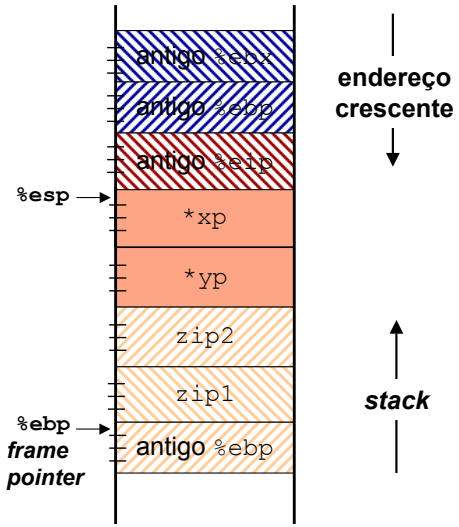
Evolução da stack no IA32 (7)

swap

3. Término de swap ...
- libertar espaço de var locais...**não**...
- recuperar registos
- recuperar antigo *frame pointer*
- voltar a call_swap

```
void swap(int *xp, int *yp)
{
  (…)
}
```

```
popl %ebx          Restore %ebx
movl %ebp,%esp     Restore %esp
popl %ebp          Restore %ebp
ret                Return to caller
                   ou
popl %ebx          Restore %ebx
leave              Restore %esp, %ebp
ret                Return to caller
```



Evolução da stack no IA32 (8)

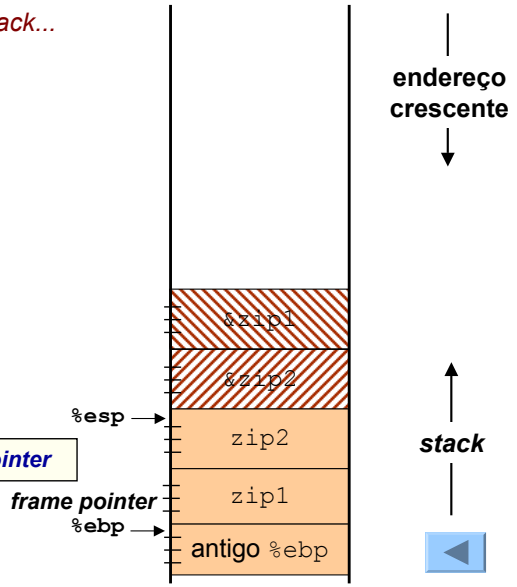
call_swap

4. Terminar invocação de swap . . .

- libertar espaço de parâmetros na *stack*...
- recuperar registos?...**não**...

```
void call_swap()
{
int zip1 = 15213;
int zip2 = 91125;
  (…)
  swap(&zip1, &zip2);
  (…)
}
```

```
addl  $8,(%esp)     Update stack pointer
```



Funções em assembly: IA32 versus MIPS (1)

- Principais diferenças
  - na organização dos registos
    - IA32: poucos registos genéricos
      - variáveis e parâmetros normalmente na *stack*
    - MIPS(RISC): 32 registos genéricos
      - registos para variáveis locais
      - registos para passagem de parâmetros para funções
      - registo para endereço de retorno
  - consequências:
    - menor utilização da *stack*
    - RISC potencialmente mais eficiente

Funções em assembly: IA32 versus MIPS (2)

IA32

```
_swap:
     pushl     %ebp
     movl      %esp, %ebp
     pushl     %ebx
     movl      8(%ebp), %edx
     movl      12(%ebp), %ecx
     movl      (%edx), %ebx
     movl      (%ecx), %eax
     movl      %eax, (%edx)
     movl      %ebx, (%ecx)
     popl      %ebx
     popl      %ebp
     ret
_call_swap:
     pushl     %ebp
     movl      %esp, %ebp
     subl      $24, %esp
     movl      $15213, -4(%ebp)
     movl      $91125, -8(%ebp)
     leal      -4(%ebp), %eax
     movl      %eax, (%esp)
     leal      -8(%ebp), %eax
     movl      %eax, 4(%esp)
     call      _swap
     movl      %ebp, %esp
     popl      %ebp
     ret
```

MIPS

```
swap:
   lw          $v1,0($a0)
   lw          $v0,0($a1)
   sw          $v0,0($a0)
   sw          $v1,0($a1)
   j           $31

call_swap:
   subu        $sp,$sp,32
   sw          $ra,24($sp)
   li          $v0,15213
   sw          $v0,16($sp)
   li          $v0,0x10000
   ori         $v0,$v0,0x63f5
   sw          $v0,20($sp)
   addu        $a0,$sp,16   # &zip1= sp+16
   addu        $a1,$sp,20   # &zip2= sp+20
   jal         swap
   lw          $ra,24($sp)
   addu        $sp,$sp,32
   j           $ra
```


Funções em assembly: IA32 versus MIPS (3)

call_swap

1. Invocar swap

- salvaguardar registos
- passagem de parâmetros
- chamar rotina e guardar endereço de retorno

IA32

```
                          No registers to save
leal  -4(%ebp),%eax       Compute &zip2
pushl %eax                Push &zip2
leal  -8(%ebp),%eax       Compute &zip1
pushl %eax                Push &zip1
call  swap                Call swap function
```

Acessos à *stack*

MIPS

```
sw    $ra,24($sp)         Save reg with return addr
addu  $a0,$sp,16          Compute and load &zip1
addu  $a1,$sp,20          Compute and load &zip2
jal   swap                Call swap function
```

Funções em assembly: IA32 versus MIPS (4)

swap

1. Inicializar `swap`
- actualizar *frame pointer*
- salvaguardar registos
- reservar espaço p/ locais

IA32

```
swap:
  pushl %ebp          Save old %ebp
  movl  %esp,%ebp     Set %ebp as frame pointer
  pushl %ebx          Save %ebx
                      No need to save locals
```

Acessos à *stack*

MIPS

```
                      No frame pointer to update
                      No registers to save
                      No need to save locals
```


Funções em assembly: IA32 versus MIPS (5)

swap

2. Corpo de `swap` ...

IA32

```
movl   12(%ebp),%ecx     Get yp
movl   8(%ebp),%edx      Get xp
movl   (%ecx),%eax       Get y
movl   (%edx),%ebx       Get x
movl   %eax,(%edx)       Store y at *xp
movl   %ebx,(%ecx)       Store x at *yp
```

Acessos à memória (todas...)

MIPS

```
lw    $v1,0($a0)     Get x
lw    $v0,0($a1)     Get y
sw    $v0,0($a0)     Store y at *xp
sw    $v1,0($a1)     Store x at *yp
```


Funções em assembly: IA32 versus MIPS (6)

swap

3. Término de `swap` ...
- libertar espaço de var locais
- recuperar registos
- recuperar antigo *frame pointer*
- voltar a `call_swap`

IA32

```
                      No need to restore locals
popl %ebx             Restore %ebx
movl %ebp,%esp        Restore %esp
popl %ebp             Restore %ebp
ret                   Return to caller
```

Acessos à *stack*

MIPS

```
                      No need to restore locals
                      No need to restore reg's
                      No need to restore frame pointer
j   $31               Return to caller
```


Funções em assembly: IA32 versus MIPS (7)

call_swap

2. Terminar invocação de `swap`...
- libertar espaço de parâmetros na *stack*...
- recuperar registos

IA32

```
addl  $8,(%esp)      Update stack pointer
                     No registers to restore
```

Acessos à *stack*

MIPS

```
                     No need to free space in stack
lw    $ra,24($sp)    Restore reg with return addr
```

A série de Fibonacci (1)

do-while

```
int fib_dw(int n)
{
  int i = 0;
  int val = 0;
  int nval = 1;

  do {
    int t = val + nval;
    val = nval;
    nval = t;
    i++;
  } while (i<n);

    return val;
}
```

for

```
int fib_f(int n)
{
  int i;
  int val = 1;
  int nval = 1;

 for (i=1; i<n; i++) {
    int t = val + nval;
    val = nval;
    nval = t;
}

    return val;
}
```

while

```
int fib_w(int n)
{
  int i = 1;
  int val = 1;
  int nval = 1;

 while (i<n) {
    int t = val + nval;
    val = nval;
    nval = t;
    i++;
  }

    return val;
}
```

recursive function

```
int fib_rec (int n)
{
   int prev_val, val;
   if (n<=2)
       return (1);
   prev_val = fib_rec (n-2);
   val = fib_rec (n-1);
   return (prev_val+val);
}
```


A série de Fibonacci (2)

recursive function

```
int fib_rec (int n)
{
   int prev_val, val;
   if (n<=2)
       return (1);
   prev_val = fib_rec (n-2);
   val = fib_rec (n-1);
   return (prev_val+val);
}
```

```
_fib_rec:
   pushl  %ebp
   movl   %esp, %ebp          Actualiza frame pointer

   subl   $12, %esp           Reserva espaço na stack para 3 int's
   movl   %ebx, -8(%ebp)      Salvaguarda os 2 reg's que vão ser usados;
   movl   %esi, -4(%ebp)          de notar a forma de usar a stack...

   movl   8(%ebp), %esi
```


A série de Fibonacci (3)

recursive function

```
int fib_rec (int n)
{
   int prev_val, val;
   if (n<=2)
       return (1);
   prev_val = fib_rec (n-2);
   val = fib_rec (n-1);
   return (prev_val+val);
}
```

```
   ...
   movl   %esi, -4(%ebp)
   movl   8(%ebp), %esi       Coloca o parâmetro n em %esi
   movl   $1, %eax            Coloca já o valor de retorno em %eax
   cmpl   $2, %esi            n<=2 ?
   jle    L1                  Se sim, salta para o fim
   leal   -2(%esi), %eax      Se não, ...
   ...
L1:
   movl   -8(%ebp), %ebx
```


A série de Fibonacci (4)

recursive function

```
int fib_rec (int n)
{
   int prev_val, val;
   if (n<=2)
       return (1);
   prev_val = fib_rec (n-2);
   val = fib_rec (n-1);
   return (prev_val+val);
}
```

```
   ...
   jle    L1                  Se sim, salta para o fim
   leal   -2(%esi), %eax      Calcula n-2, e...
   movl   %eax, (%esp)        ... coloca-o no topo da stack (parâmetro)
   call   _fib_rec            Chama a função fib_rec e ...
   movl   %eax, %ebx          ... guarda o valor de prev_val em %ebx
   leal   -1(%esi), %eax
   ...
```

A série de Fibonacci (5)

recursive function

```
int fib_rec (int n)
{
   int prev_val, val;
   if (n<=2)
      return (1);
   prev_val = fib_rec (n-2);
   val = fib_rec (n-1);
   return (prev_val+val);
}
```

```
   ...
   movl  %eax, %ebx
   leal  -1(%esi), %eax      Calcula n-1, e...
   movl  %eax, (%esp)        ... coloca-o no topo da stack (parâmetro)
   call  _fib_rec            Chama de novo a função fib_rec
   leal  (%eax,%ebx), %eax
   ...
```


A série de Fibonacci (6)

recursive function

```
int fib_rec (int n)
{
   int prev_val, val;
   if (n<=2)
      return (1);
   prev_val = fib_rec (n-2);
   val = fib_rec (n-1);
   return (prev_val+val);
}
```

```
   ...
   call  _fib_rec
   leal  (%eax,%ebx), %eax   Calcula e coloca em %eax o valor de retorno
L1:
   movl  -8(%ebp), %ebx
   movl  -4(%ebp), %esi      Recupera o valor dos 2 reg's usados
   movl  %ebp, %esp          Actualiza o valor do stack pointer
   popl  %ebp                Recupera o anterior valor do frame pointer
   ret
```


Call Chain Example (1)

(Bryant, Class07, F'02)

Code Structure

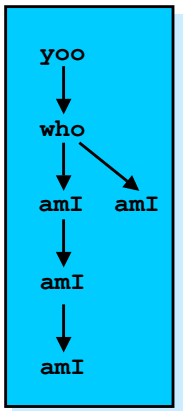
```
yoo(...)
{
   •
   •
   who();
   •
   •
}
```

```
who(...)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

```
amI(...)
{
   •
   •
   amI();
   •
   •
}
```

– Procedure `amI` recursive

Call Chain



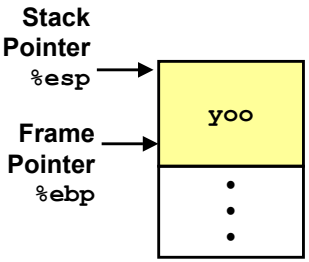
Call Chain Example (2)

(baseado no Bryant, Class07, F'02)

```
yoo(...)
{
   •
   •
   who();
   •
   •
}
```

Call Chain

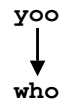
yoo

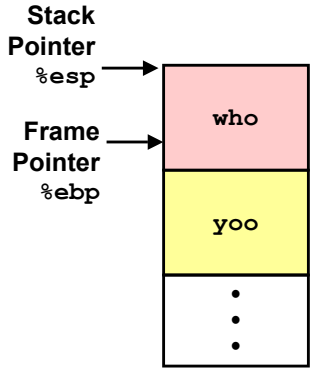


Call Chain Example (3)
(baseado no Bryant, Class07, F'02)

```
who(…)
{
  • • •
  amI();
  • • •
  amI();
  • • •
}
```

Call Chain

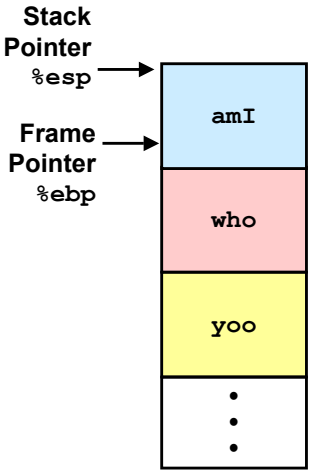




Call Chain Example (4)
(baseado no Bryant, Class07, F'02)

```
amI(…)
{
  •
  •
  amI();
  •
  •
}
```

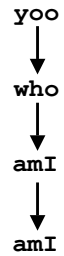
Call Chain

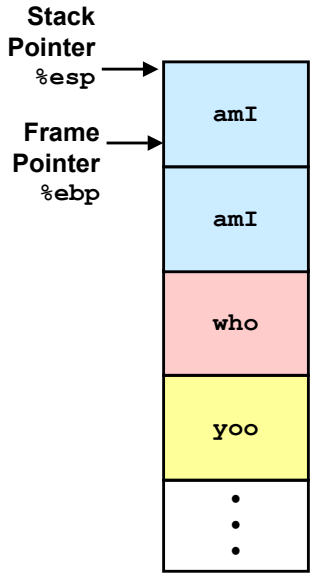


Call Chain Example (5)
(baseado no Bryant, Class07, F'02)

```
amI(…)
{
  •
  •
  amI();
  •
  •
}
```

Call Chain

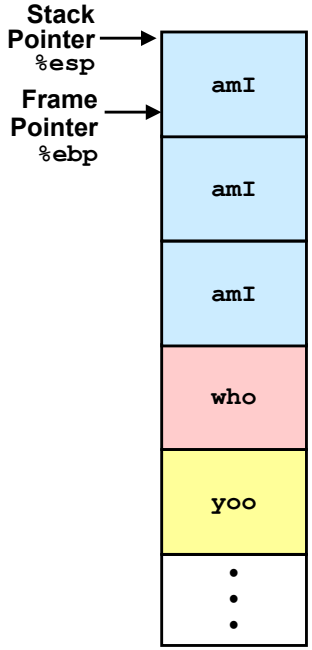




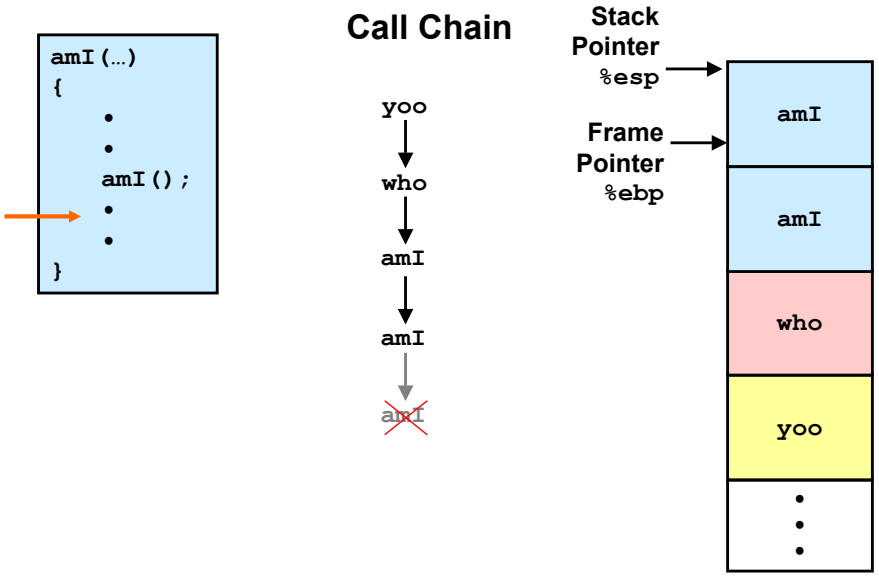
Call Chain Example (6)
(baseado no Bryant, Class07, F'02)

```
amI(…)
{
  •
  •
  amI();
  •
  •
}
```

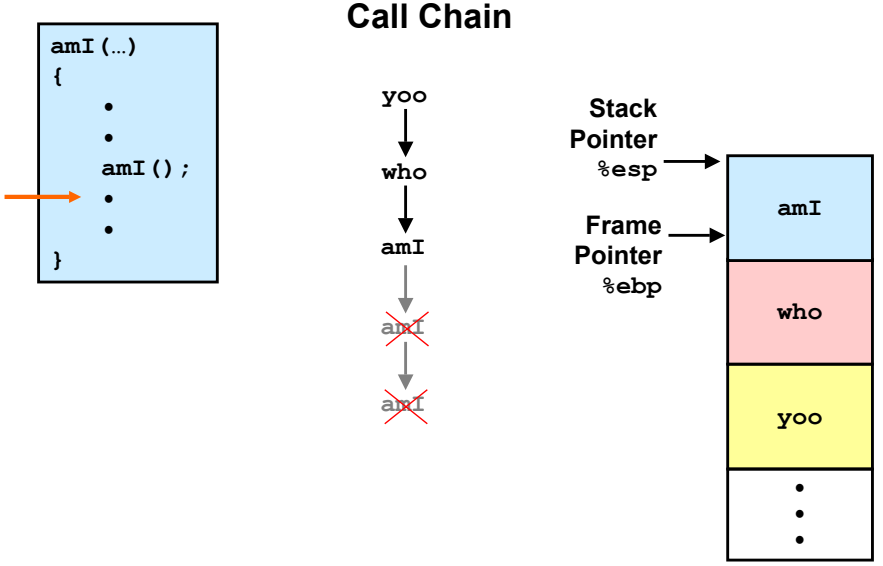
Call Chain



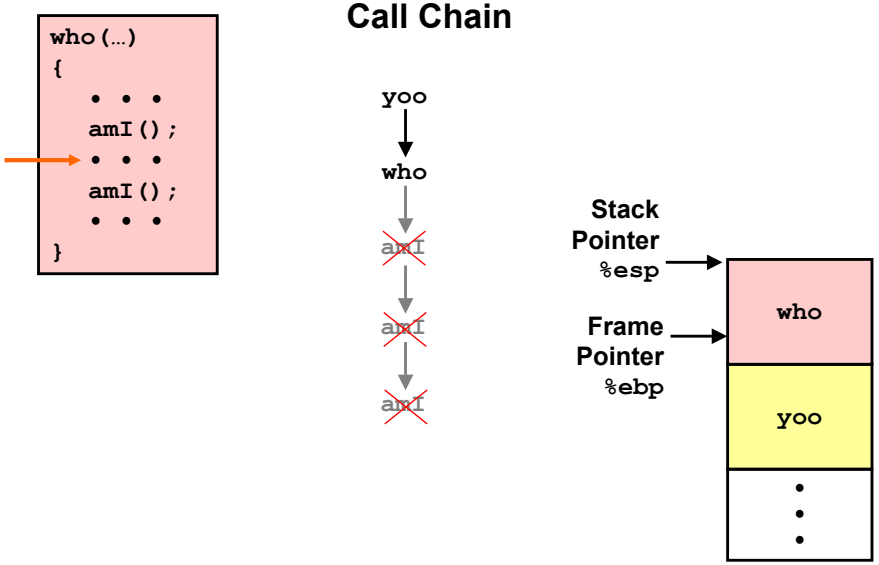
Call Chain Example (7)
(baseado no Bryant, Class07, F'02)



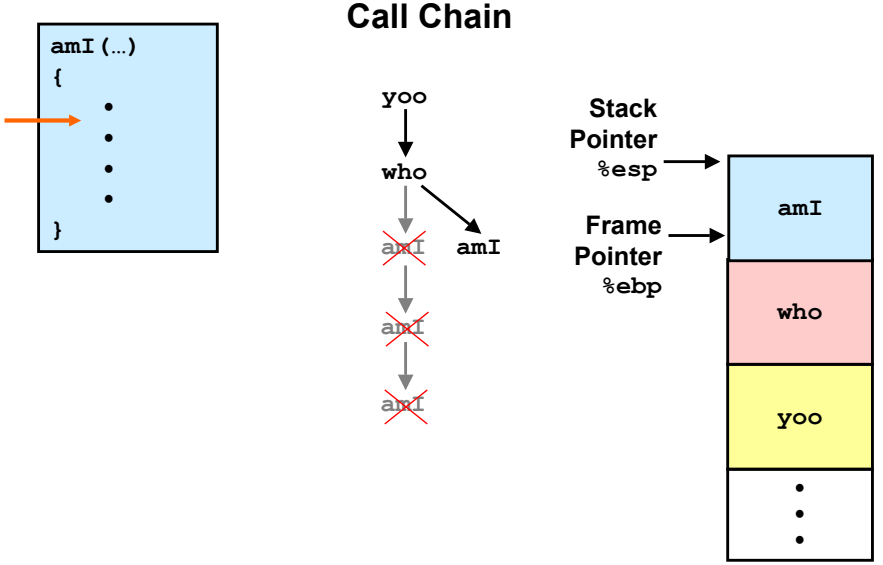
Call Chain Example (8)
(baseado no Bryant, Class07, F'02)



Call Chain Example (9)
(baseado no Bryant, Class07, F'02)



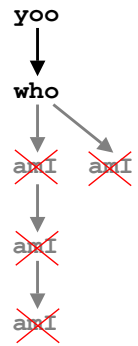
Call Chain Example (10)
(baseado no Bryant, Class07, F'02)

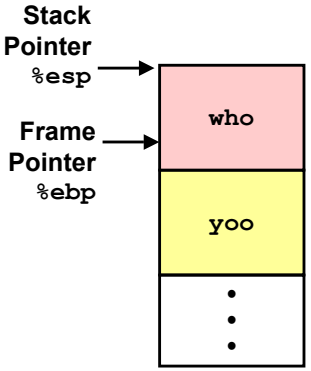


Call Chain Example (11)
(baseado no Bryant, Class07, F'02)

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

Call Chain





Call Chain Example (12)
(baseado no Bryant, Class07, F'02)

```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

Call Chain