

LẬP TRÌNH HỆ THỐNG

ThS. Đỗ Thị Hương Lan
(landth@uit.edu.vn)



TRƯỜNG ĐH CÔNG NGHỆ THÔNG TIN - ĐHQG-HCM
KHOA MẠNG MÁY TÍNH & TRUYỀN THÔNG
FACULTY OF COMPUTER NETWORK AND COMMUNICATIONS

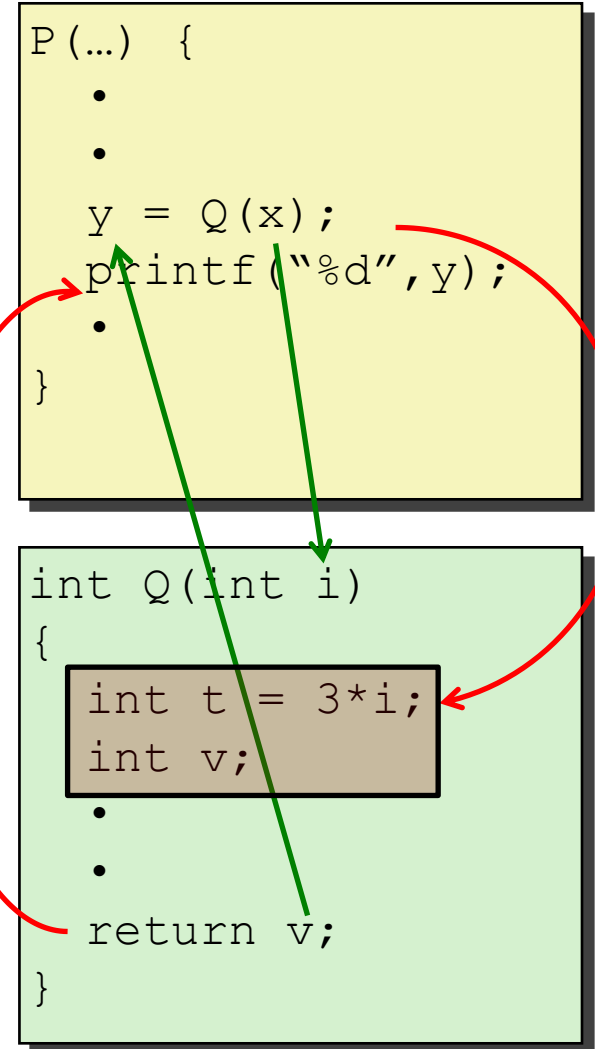
Tầng 8 - Tòa nhà E, trường ĐH Công nghệ Thông tin, ĐHQG-HCM
Điện thoại: (08)3 725 1993 (122)

Machine-level programming: Procedure (Hàm/Thủ tục)



Cơ chế gọi hàm/thủ tục (procedure)

- **1. Chuyển luồng**
 - Bắt đầu thực thi hàm được gọi
 - Trở về vị trí đã gọi hàm
- **2. Truyền dữ liệu**
 - Truyền tham số (arguments) cho hàm
 - Nhận giá trị trả về của hàm
- **3. Quản lý bộ nhớ**
 - Cấp phát bộ nhớ khi thực thi hàm
 - Thu hồi bộ nhớ khi thực thi xong
- **Tất cả đều thực hiện được ở mức máy tính!**
- **Hàm ở IA32 và x86-64 sẽ có một số khác biệt.**



Cơ chế gọi hàm/thủ tục (procedure)

```
int main()
{
    int result = func(5,6);
    return result;
}
```

```
int func(int x, int y)
{
    int sum = 0;
    sum = x + y;
    return sum;
}
```

main:

```
    pushl    %ebp
    movl     %esp, %ebp
    subl     $16, %esp
    pushl     $6
    pushl     $5
    call     func
    movl     %eax, -4(%ebp)
    movl     -4(%ebp), %eax
    leave
    ret
```

func:

```
    pushl    %ebp
    movl     %esp, %ebp
    subl     $16, %esp
    movl     $0, -4(%ebp)
    movl     8(%ebp), %edx
    movl     12(%ebp), %eax
    addl     %edx, %eax
    movl     %eax, -4(%ebp)
    movl     -4(%ebp), %eax
    leave
    ret
```

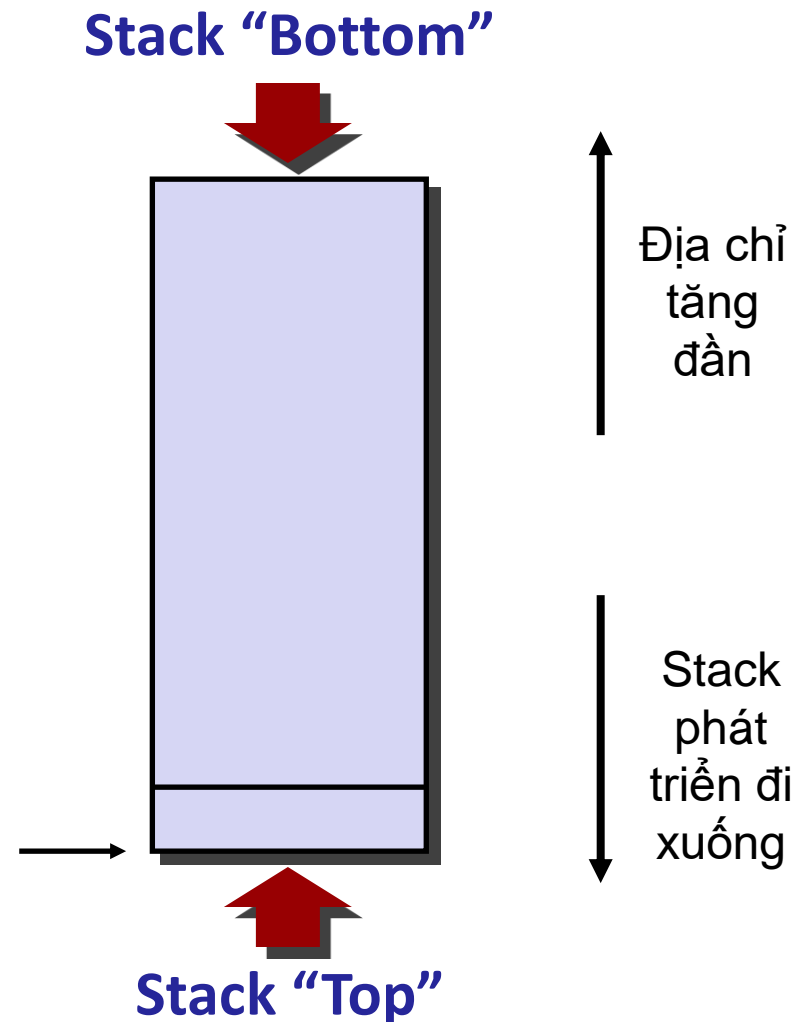
Nội dung

- Thủ tục (Procedures)
 - Cấu trúc stack
 - Gọi hàm trong IA32
 - Chuyển luồng
 - Truyền dữ liệu
 - Quản lý dữ liệu cục bộ
- Dịch ngược – Reverse Engineering

IA32 Stack

- Vùng nhớ được quản lý theo quy tắc ngăn xếp
 - First In Last Out
- Phát triển dần về phía địa chỉ thấp hơn
- Thanh ghi `%esp` chứa địa chỉ thấp nhất của stack
 - địa chỉ của “đỉnh” stack

Con trỏ stack
(Stack Pointer):
`%esp`

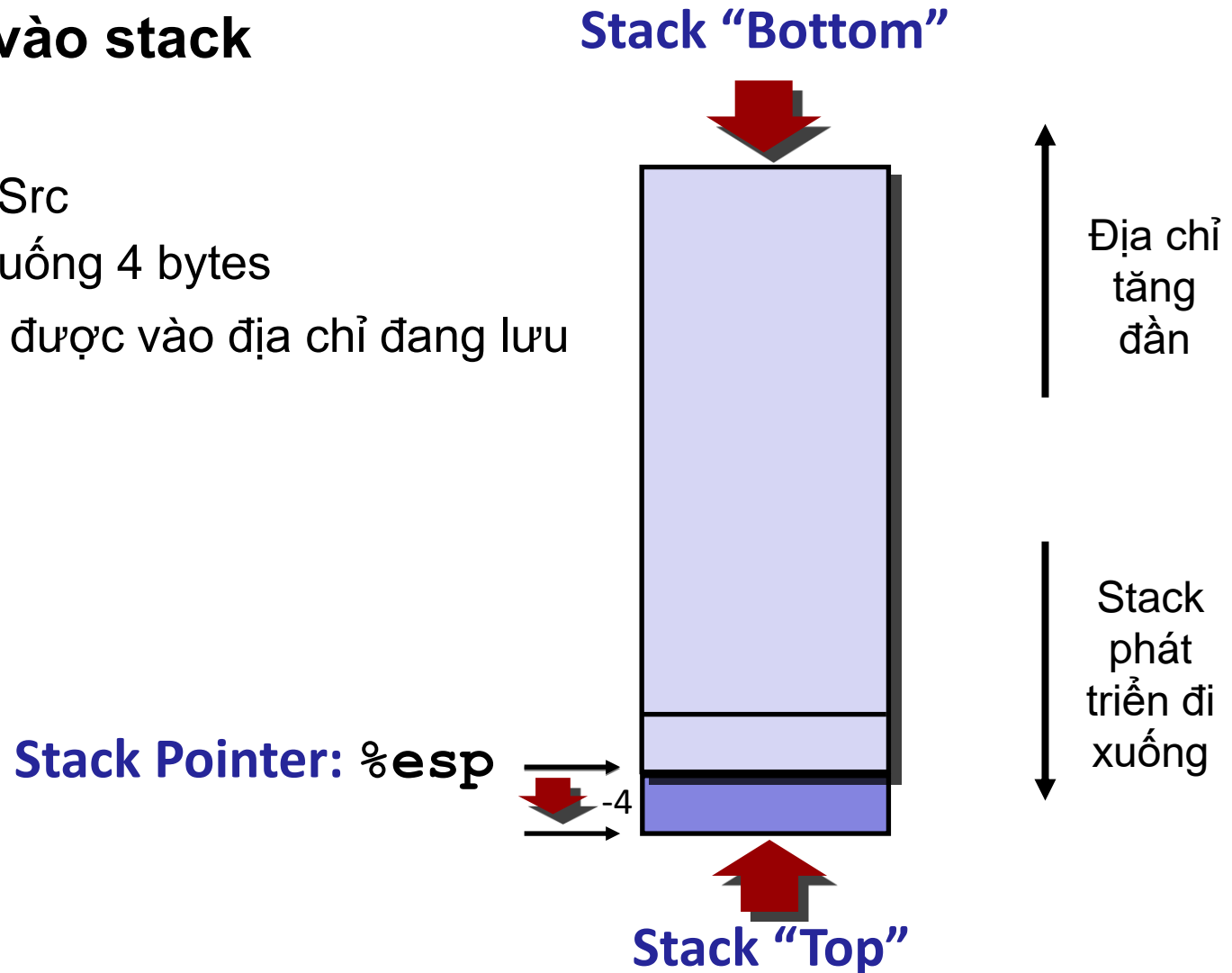


IA32 Stack: Push

■ Đẩy dữ liệu vào stack

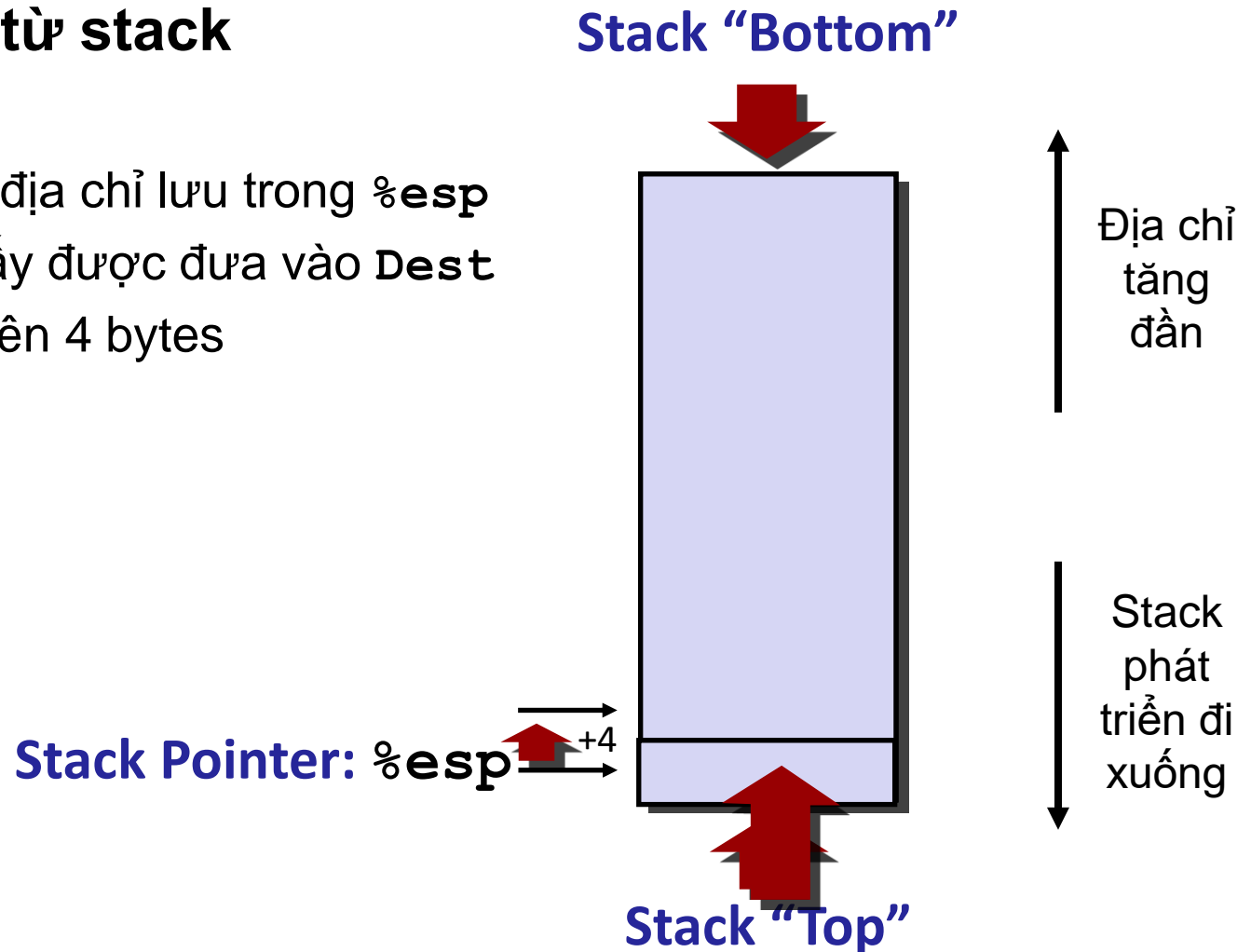
■ `pushl Src`

- Lấy giá trị từ `Src`
- Giảm `%esp` xuống 4 bytes
- Ghi giá trị lấy được vào địa chỉ đang lưu trong `%esp`



IA32 Stack: Pop

- Lấy dữ liệu từ stack
- `popl Dest`
 - Lấy giá trị ở địa chỉ lưu trong `%esp`
 - Đưa giá trị lấy được đưa vào `Dest`
 - Tăng `%esp` lên 4 bytes



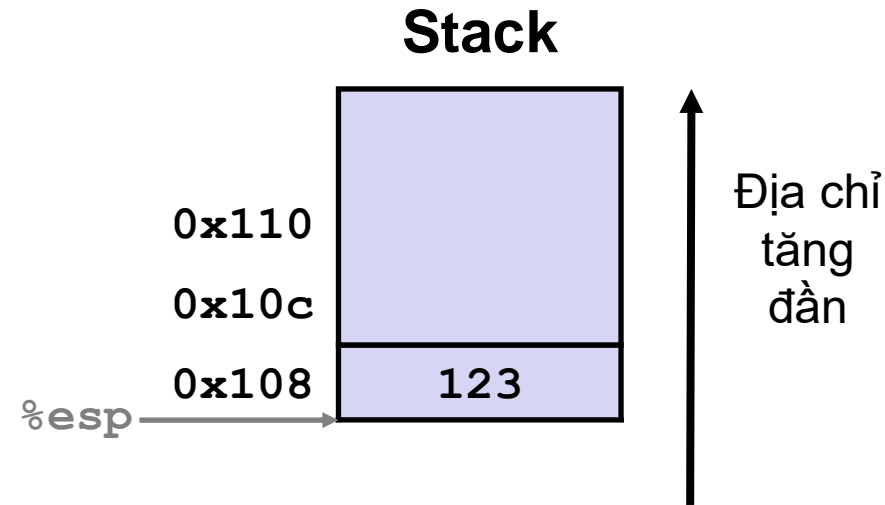
IA32 Stack: Push and Pop – Ví dụ

- `%esp = 0x108`
- `%eax = 0x1234`
- `%ebx = 0xABCD`

Các thanh ghi và stack thay đổi như thế nào khi thực hiện lần lượt các lệnh sau?

1. `push %eax`

2. `pop %ebx`



IA32 Stack: Push and Pop – Ví dụ 2

- `%esp = 0x108`
- `%eax = 0x104`
- `%ebx = 0xABCD`

Với các lệnh push dưới đây, giá trị bao nhiêu được đưa vào stack?

1. `push $0x100`

2. `push %eax`

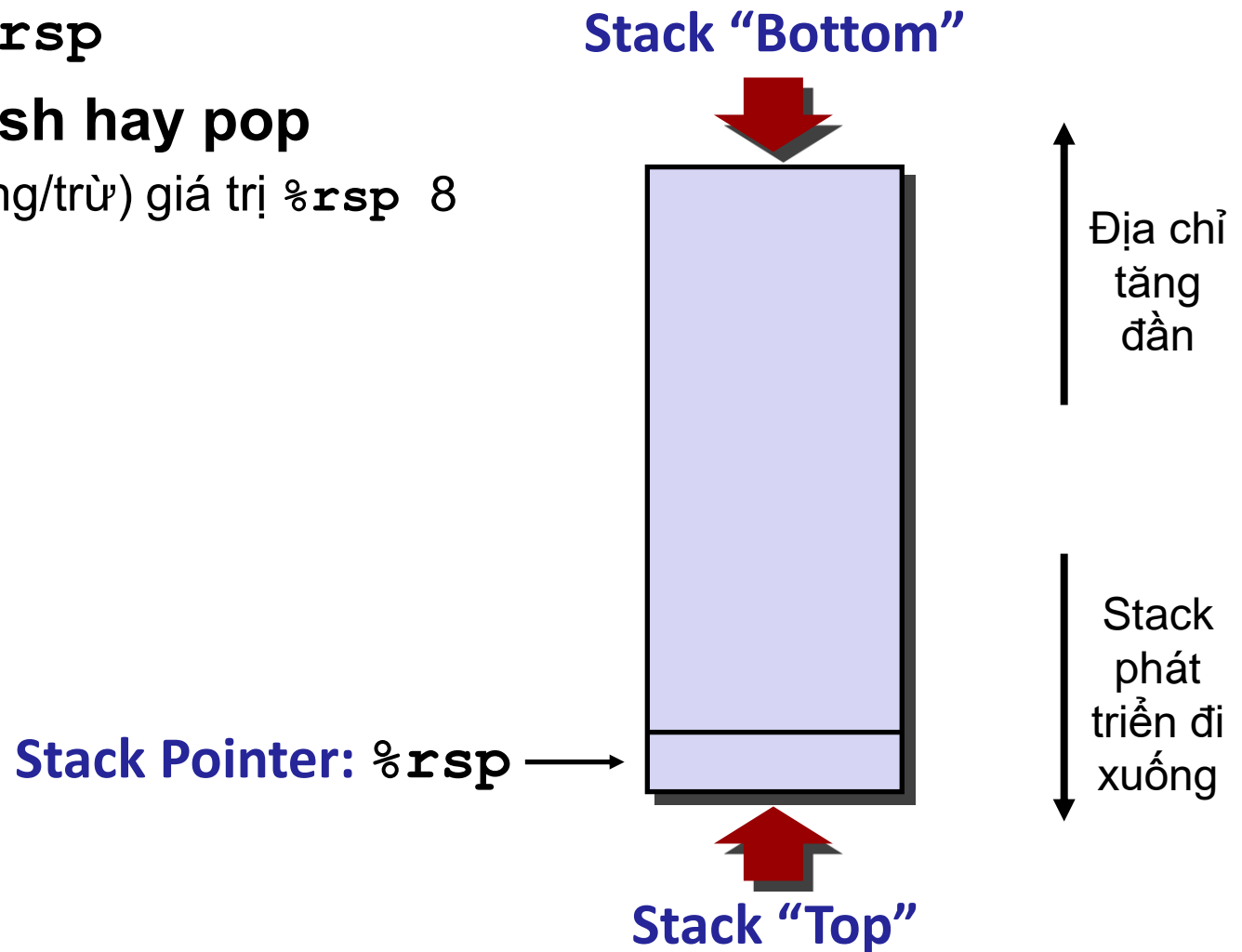
3. `push (%eax)`

4. `push 0x100`

Địa chỉ	Giá trị
0x108	0xF0
0x104	0xEF
0x100	0xAB

x86-64 Stack?

- Thanh ghi `%rsp`
- Các lệnh `push` hay `pop`
 - Thay đổi (cộng/trừ) giá trị `%rsp` 8 bytes



Nội dung

- **Thủ tục (Procedures)**
 - Cấu trúc stack
 - Gọi hàm trong IA32
 - Chuyển luồng
 - Truyền dữ liệu
 - Quản lý dữ liệu cục bộ
- Dịch ngược – Reverse Engineering

Chuyển luồng thực thi hàm

804854e: e8 3d 06 00 00 **call 8048b90** <main>
8048553: 50 **pushl %eax**

8048b90 <main>:

....

ret

Chuyển luồng thực thi hàm

- Mỗi hàm đều có địa chỉ bắt đầu, thường được gán *label*
- Stack hỗ trợ gọi hàm và trở về từ hàm
 - Gọi 1 hàm con – Procedure call
 - Trở về hàm mẹ từ hàm con – Procedure ret
- **Gọi hàm: `call label`**
 - Lưu địa chỉ trả về (return address) vào stack (push)
 - Nhảy đến `label` để thực thi
- **Trở về từ hàm: `ret`**
 - Lấy địa chỉ trả về ra từ stack (pop)
 - Nhảy đến địa chỉ lấy được để quay về hàm mẹ
- **Địa chỉ trả về (Return address):**
 - Địa chỉ câu lệnh assembly tiếp theo của hàm mẹ cần thực thi ngay phía sau lệnh `call` hàm con
 - Ví dụ trong mã assembly bên:
 - Địa chỉ trả về = 0x8048553

```
804854e: e8 3d 06 00 00  call 8048b90 <main>
8048553: 50                pushl %eax
```

Ví dụ: Gọi hàm và Trả về hàm

```
804854e:    e8 3d 06 00 00    call    8048b90 <main>
8048553:    50               pushl    %eax
```

Địa chỉ trả về? **0x8048553** → Lưu trong %eip (con trỏ lệnh)

call main ⇔ - Push địa chỉ trả về vào stack: **push \$0x8048553**
- Nhảy đến nhãn main: **jmp 8048b90**

↓
- Push địa chỉ trả về vào stack: **push %eip**
- Nhảy đến nhãn main: **jmp 8048b90**

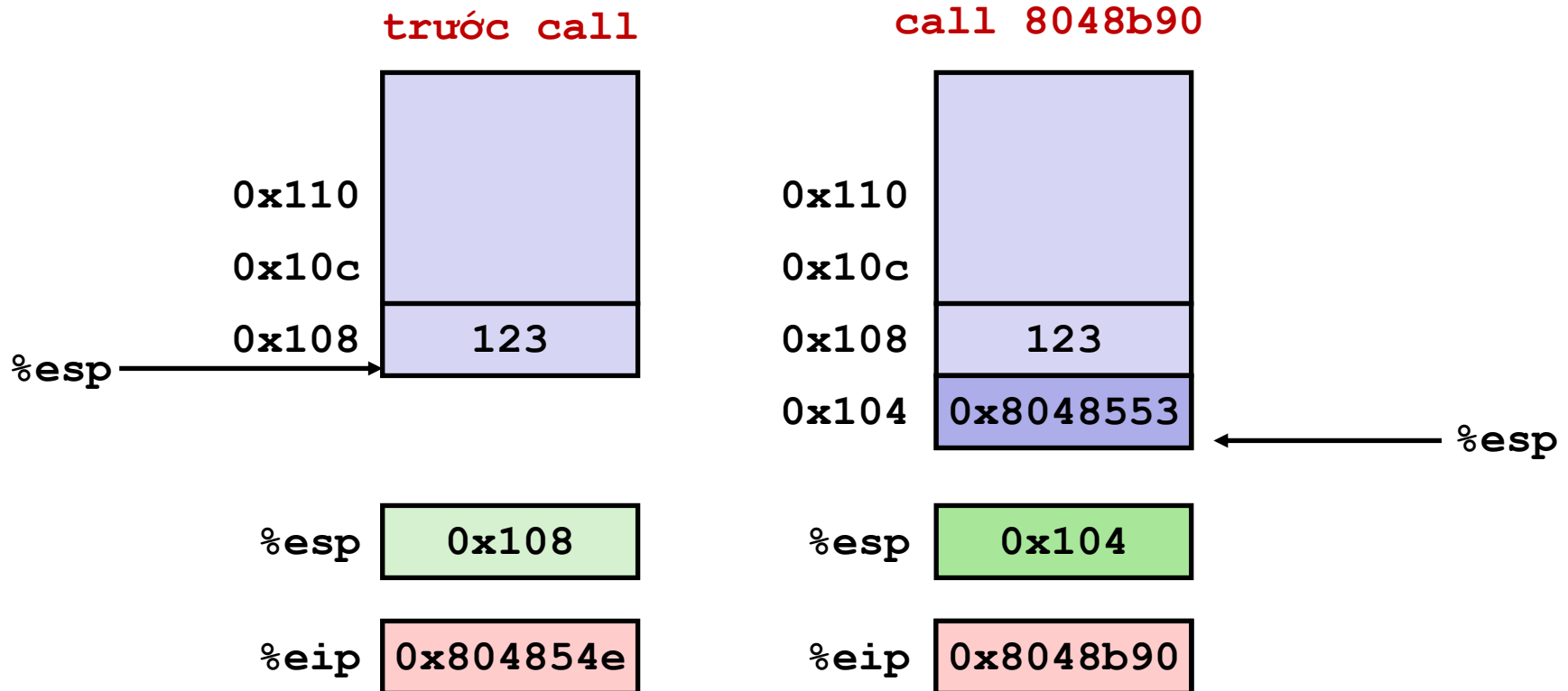
```
8048b90: main:
...
8048591:    c3               ret
```

ret ⇔ - Pop địa chỉ trả về vào stack: **pop %eip**
- Nhảy đến lệnh ở địa chỉ trả về: **jmp *%eip**

Ví dụ: Gọi hàm

804854e:	e8 3d 06 00 00	call	8048b90 <main>
8048553:	50	pushl	%eax

call 0x8048b90 = push %eip
 jmp 0x8048b90

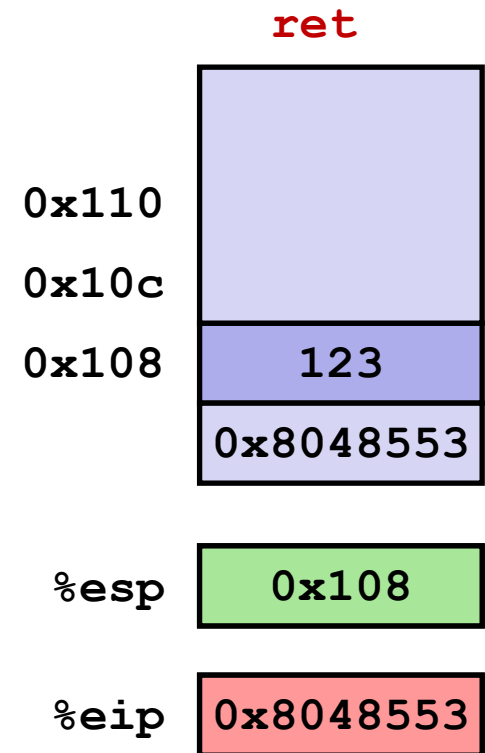
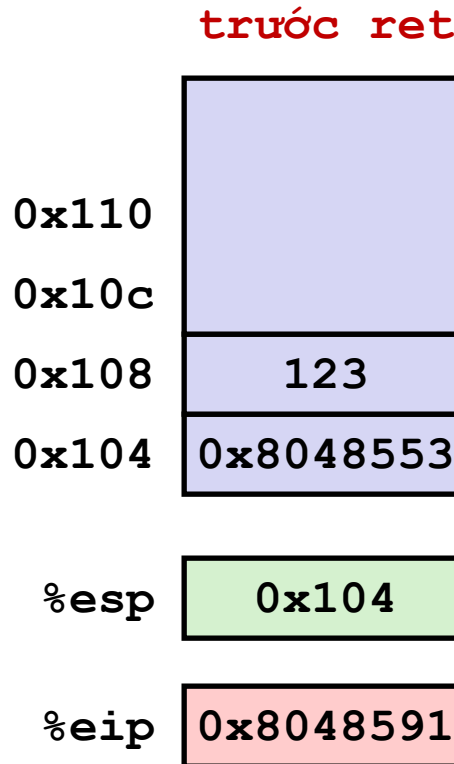


%eip: program counter

Ví dụ: Trả về hàm

```
8048591:    c3                ret
```

```
ret = pop %eip  
      jmp *%eip
```



%eip: program counter

Gọi và trả về hàm – Ví dụ

```
int main()
{
    int result = func(5,6);
    return result;
}
```

```
int func(int x, int y)
{
    int sum = 0;
    sum = x + y;
    return sum;
}
```

main:

```
    pushl    %ebp
    movl     %esp, %ebp
    subl     $16, %esp
    pushl     $6
    pushl     $5
    → call    func
    → movl     %eax, -4(%ebp)
    movl     -4(%ebp), %eax
    leave
    ret
```

return
addr

func:

```
    pushl    %ebp
    movl     %esp, %ebp
    subl     $16, %esp
    movl     $0, -4(%ebp)
    movl     8(%ebp), %edx
    movl     12(%ebp), %eax
    addl     %edx, %eax
    movl     %eax, -4(%ebp)
    movl     -4(%ebp), %eax
    leave
    ret
```

Hoạt động của hàm dựa trên stack

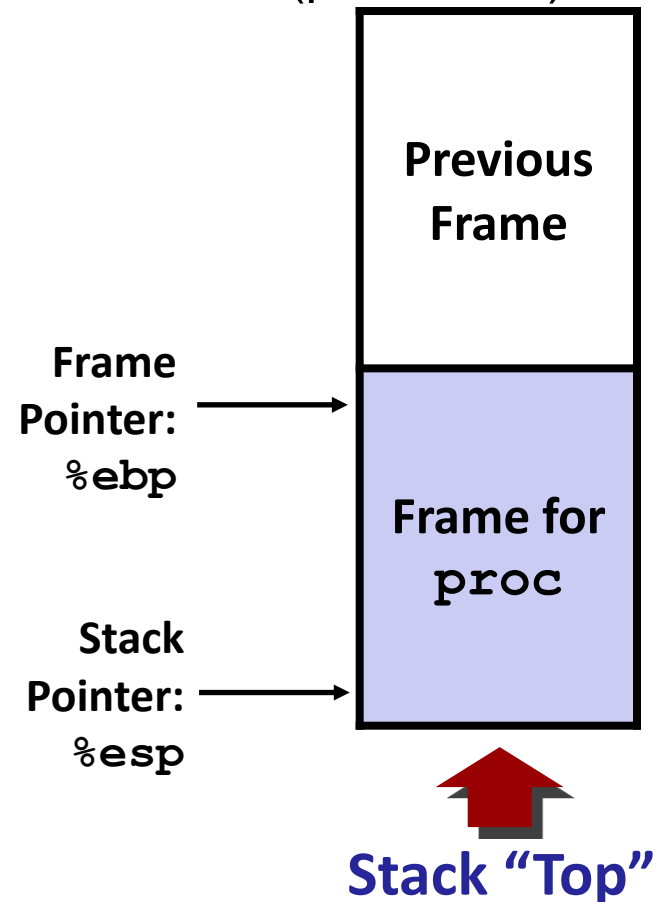
■ Stack được cấp phát bằng **Frames**

- 1 hàm (procedure) = 1 stack frame
- Hỗ trợ lưu trữ các thông tin dùng để gọi và trả về hàm (procedure)
 - Địa chỉ trả về
 - Các tham số (arguments)
 - Các biến cục bộ (local variables)

■ 1 Frame là vùng nhớ xác định bởi **%ebp** và **%esp**

- %ebp trỏ đến vị trí cố định
- %esp lưu động
- Thường truy xuất các dữ liệu trên stack dựa trên %ebp

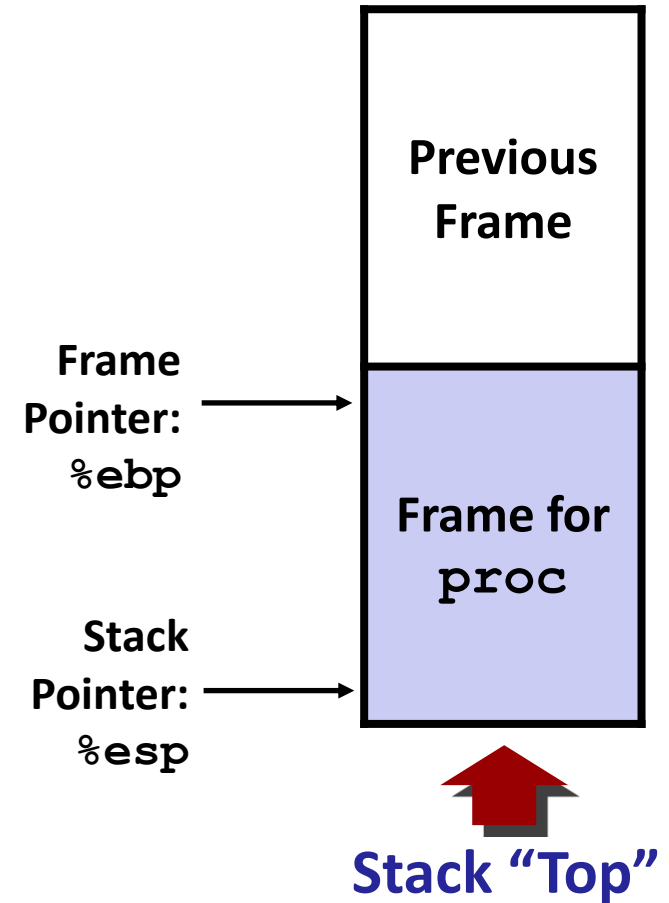
Ví dụ: `-4(%ebp)`



Stack Frames trong IA32

■ Quy tắc ngăn xếp

- Stack frame của 1 hàm tồn tại trong một khoảng thời gian từ lúc hàm được gọi đến lúc kết thúc.
 - Khi nào hàm được gọi thì stack frame của nó sẽ được tạo.
 - Khi kết thúc, stack frame sẽ được thu hồi.
- Hàm thực thi trước thì stack được cập nhật trước.
 - Stack frame cấp phát sau sẽ nằm ở các địa chỉ thấp hơn.
- Hàm kết thúc trước thì stack thu hồi trước.



Ví dụ chuỗi gọi hàm

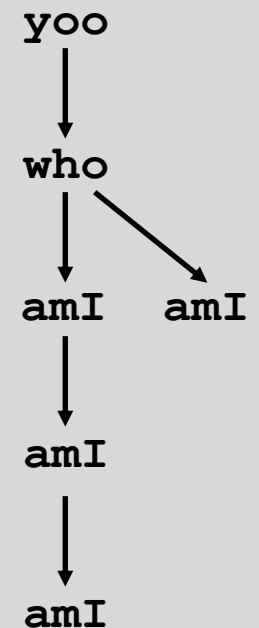
```
yoo (...)  
{  
  .  
  .  
  who ();  
  .  
  .  
}
```

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


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

Procedure **amI ()** is recursive

Example Call Chain

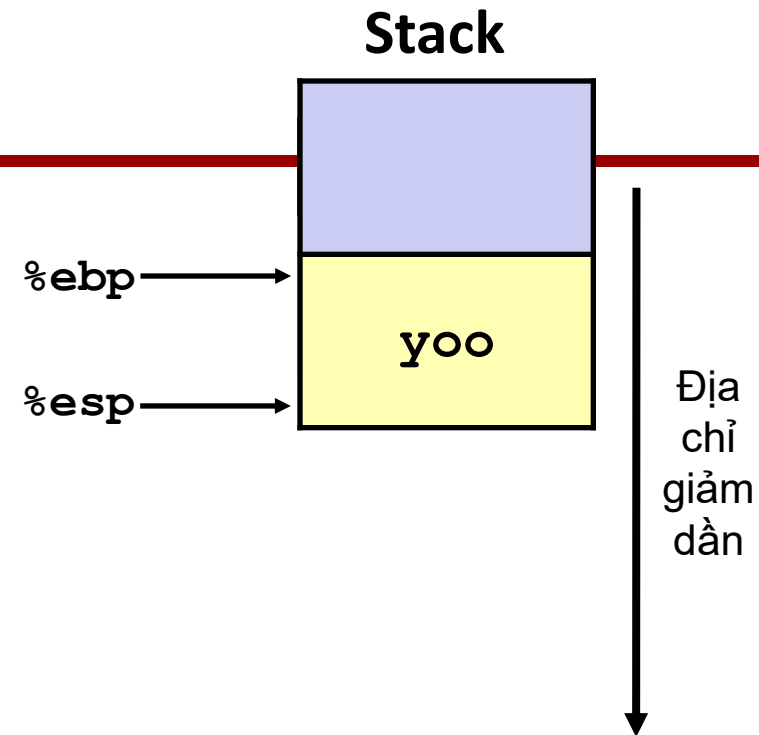


Ví dụ

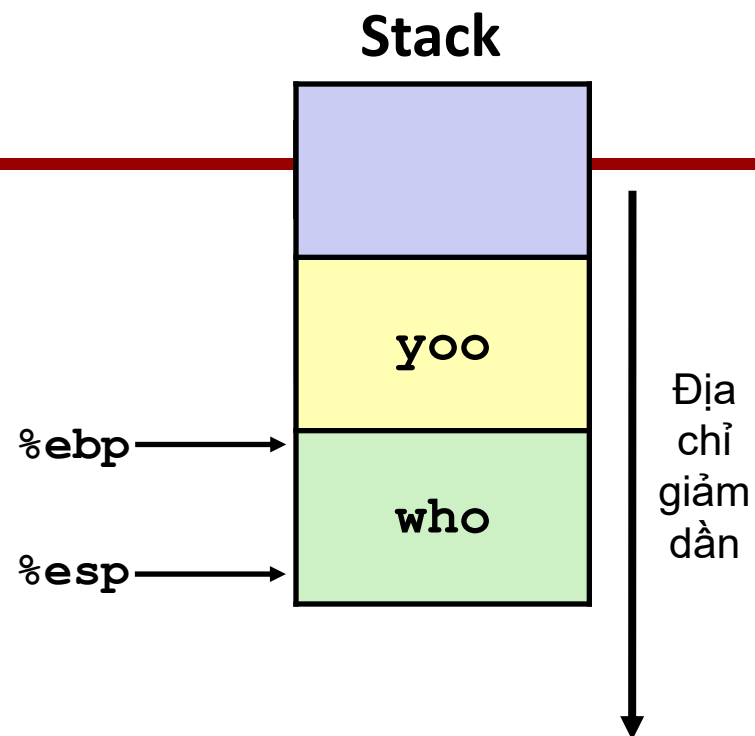
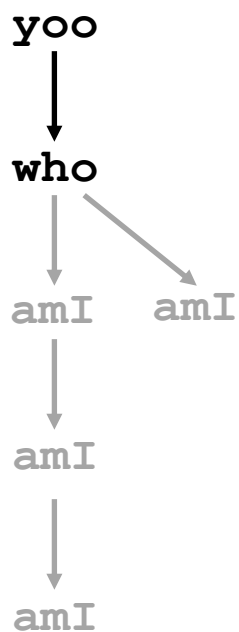
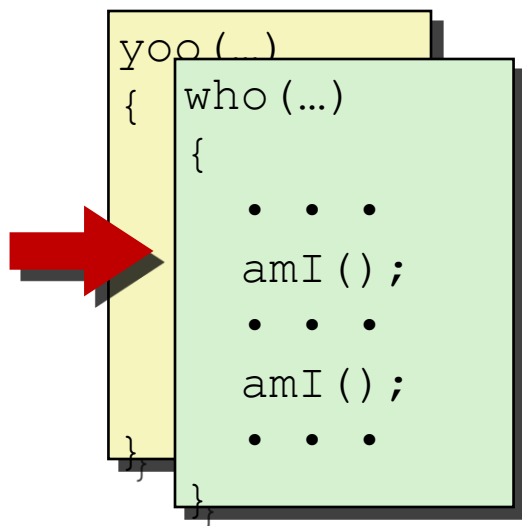


```
yoo (...)  
{  
  .  
  .  
  who ();  
  .  
  .  
}
```

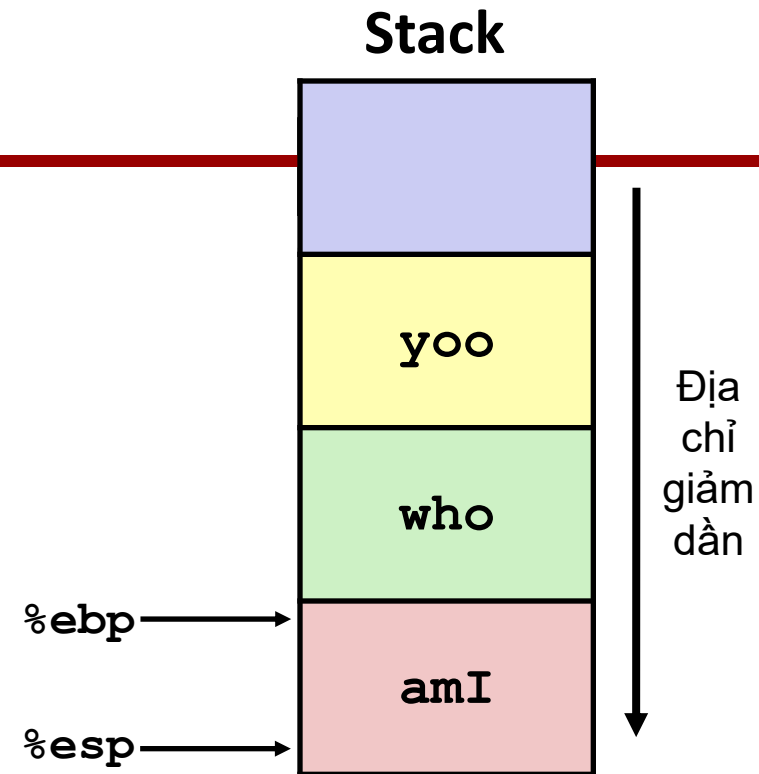
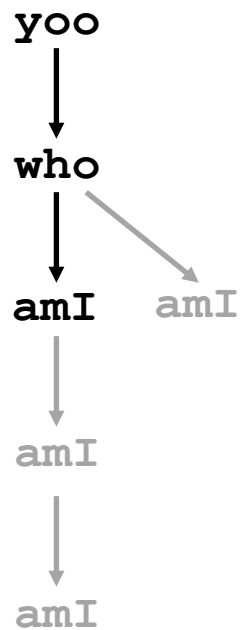
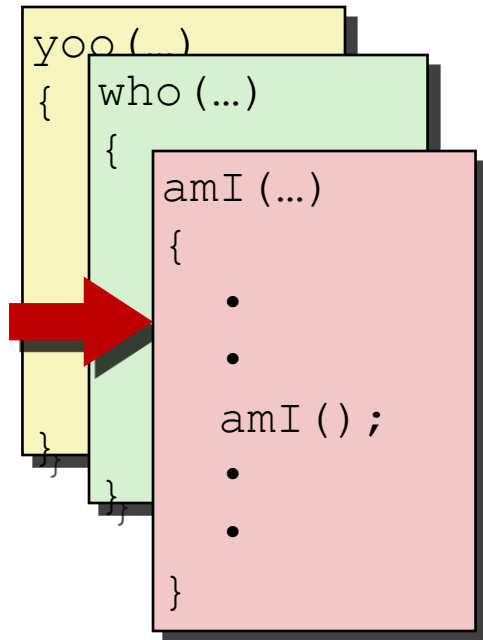
```
yoo  
  ↓  
who  
  ↓  ↘  
amI  amI  
  ↓  
amI  
  ↓  
amI
```



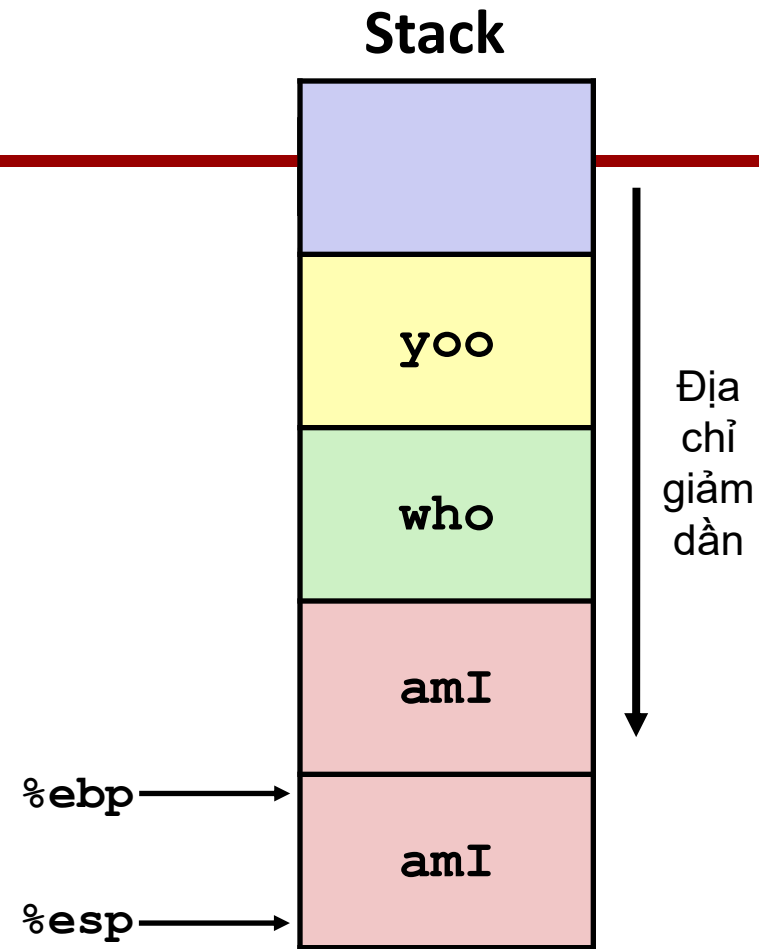
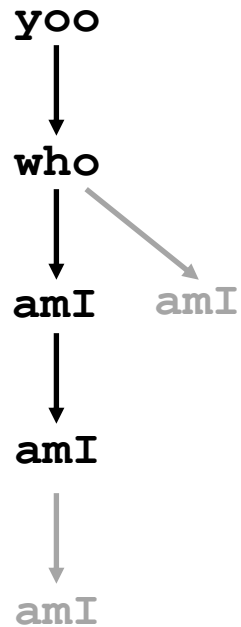
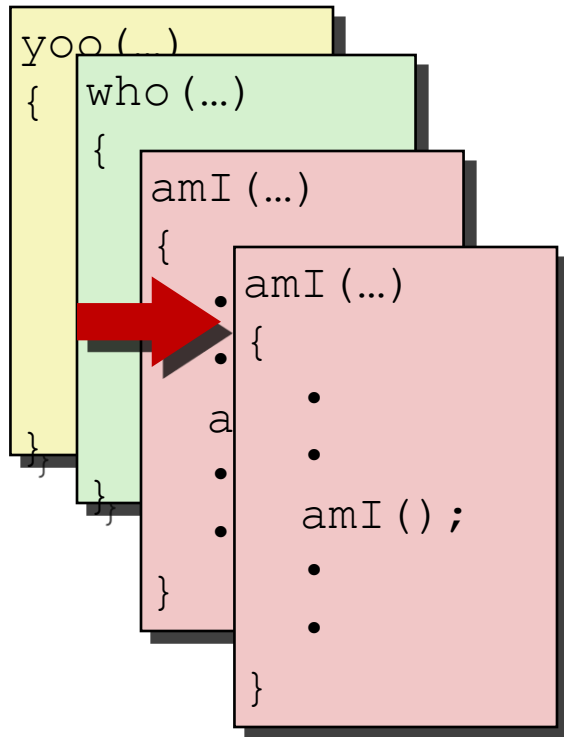
Ví dụ



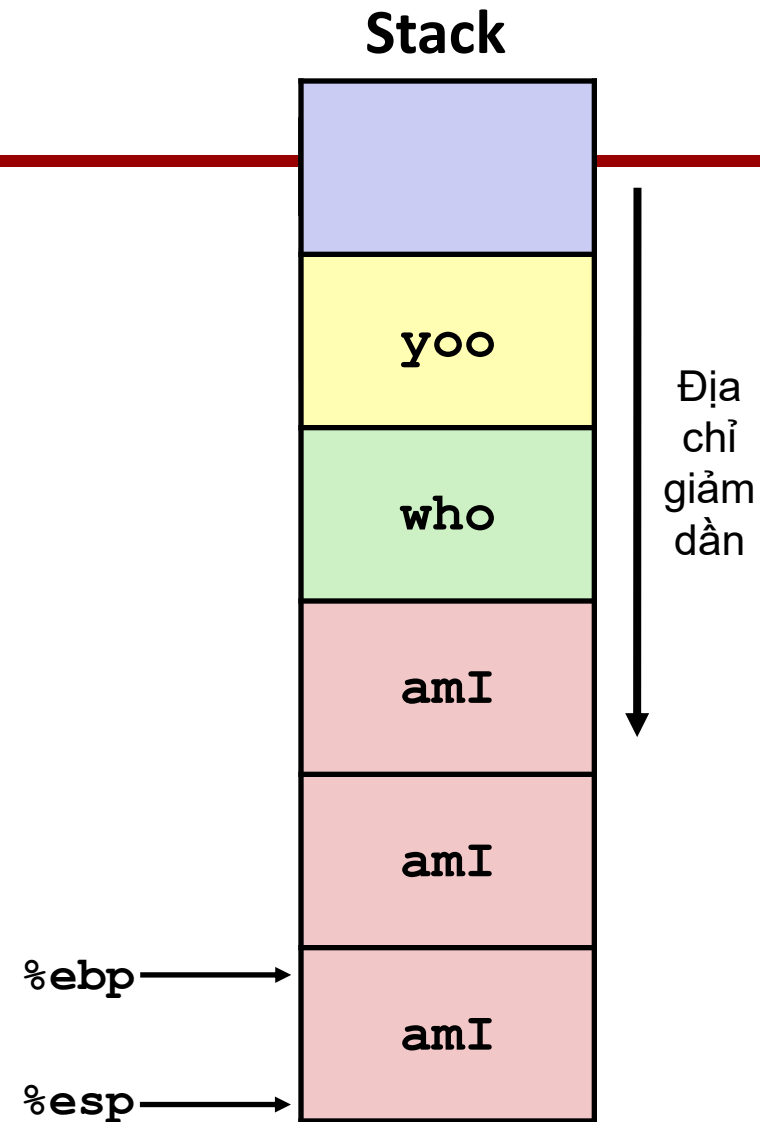
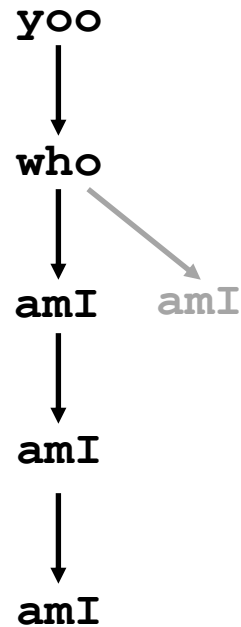
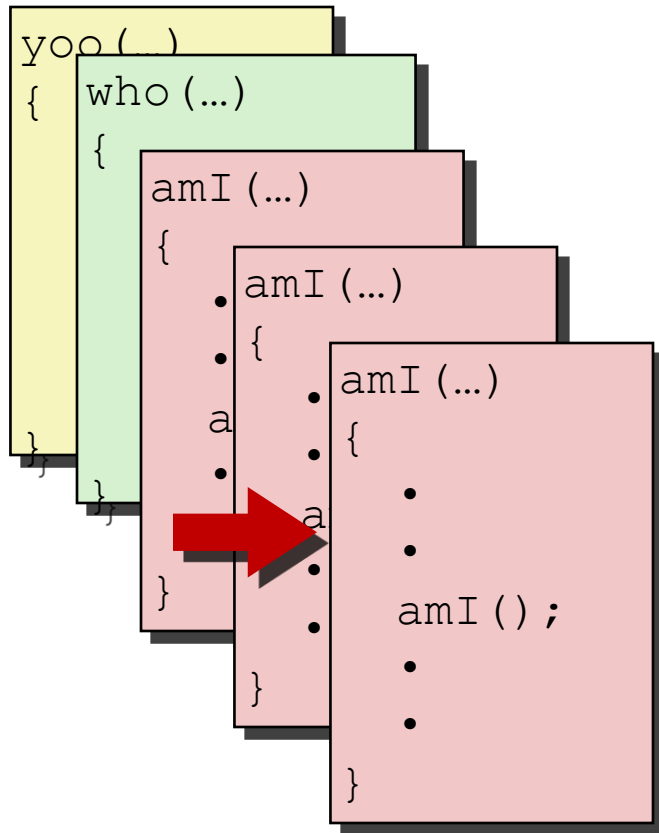
Ví dụ



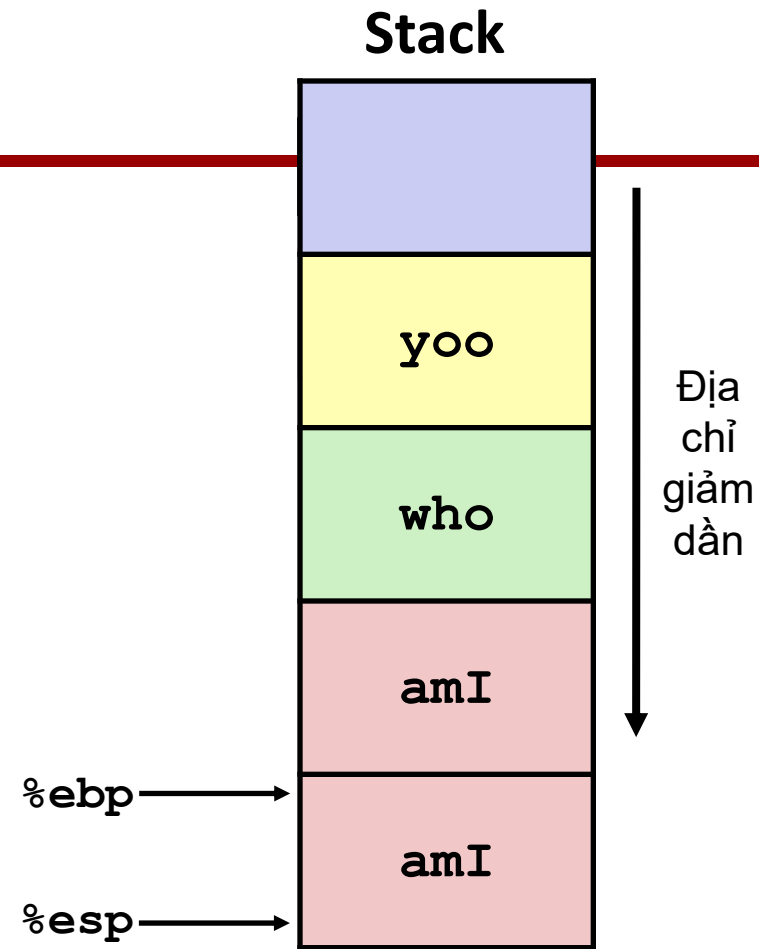
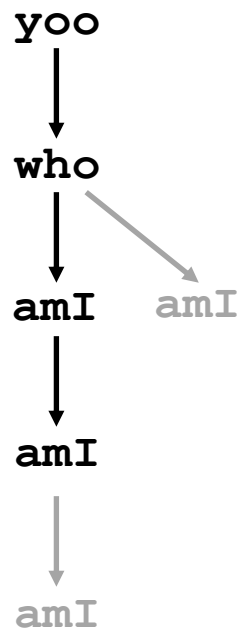
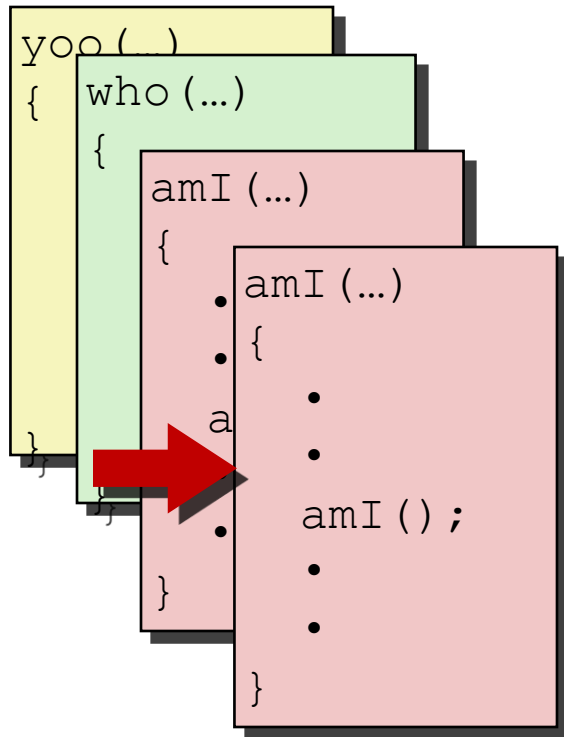
Ví dụ



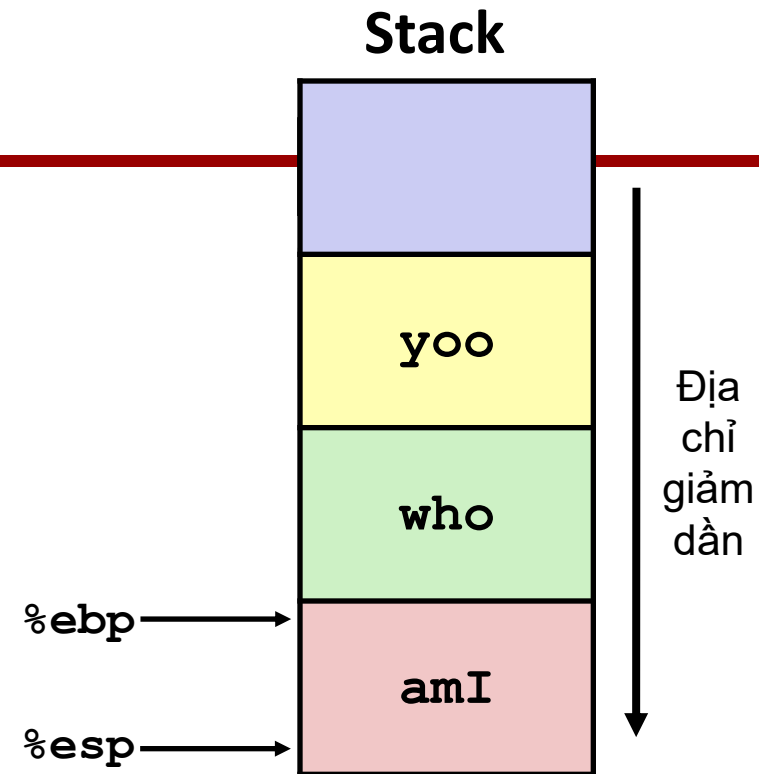
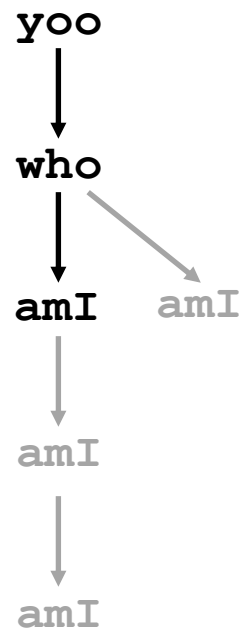
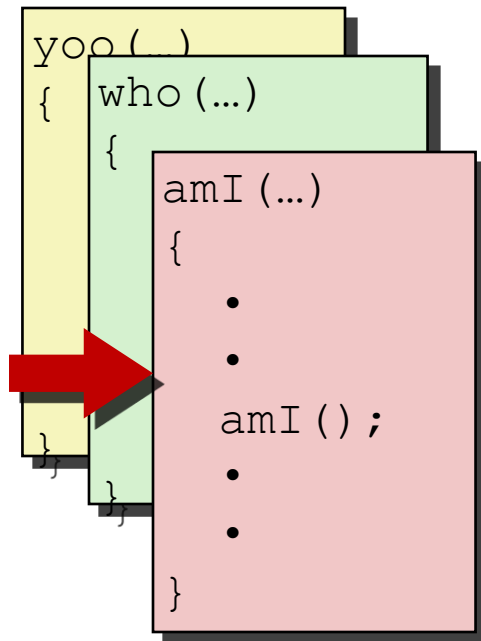
Ví dụ



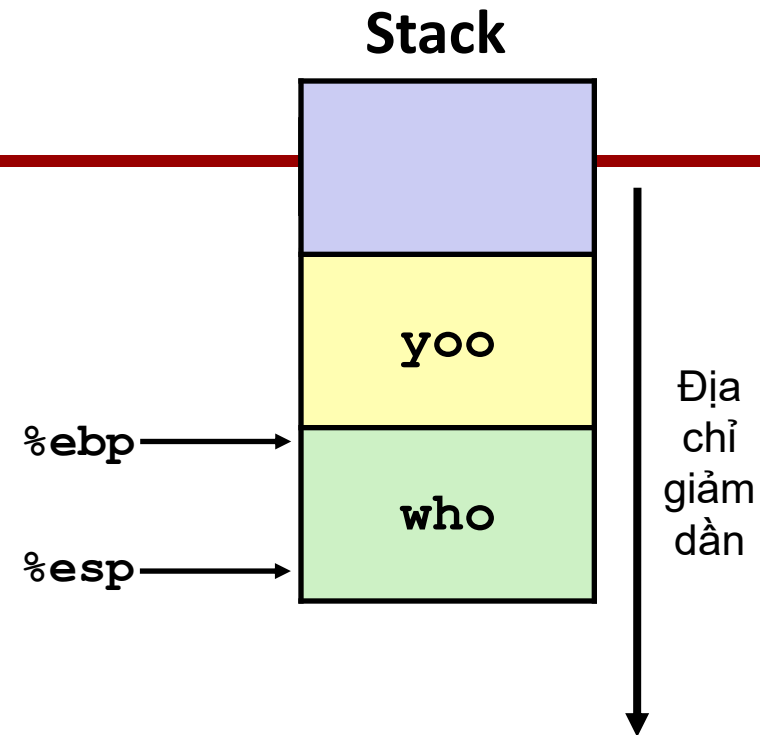
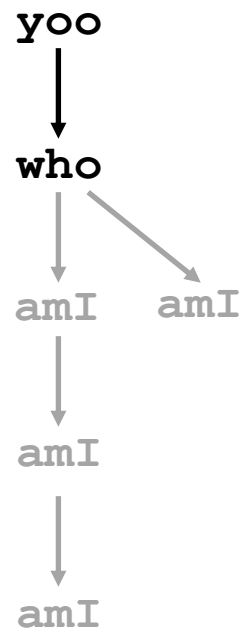
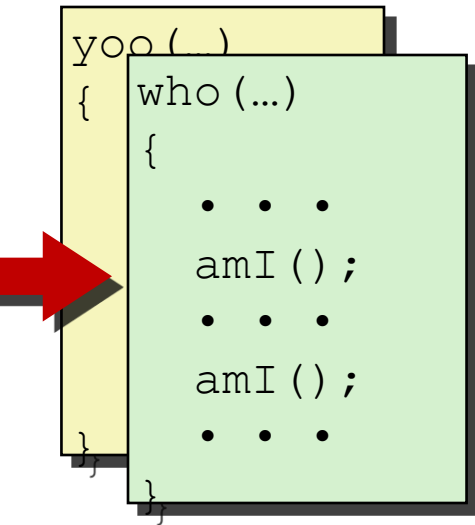
Ví dụ



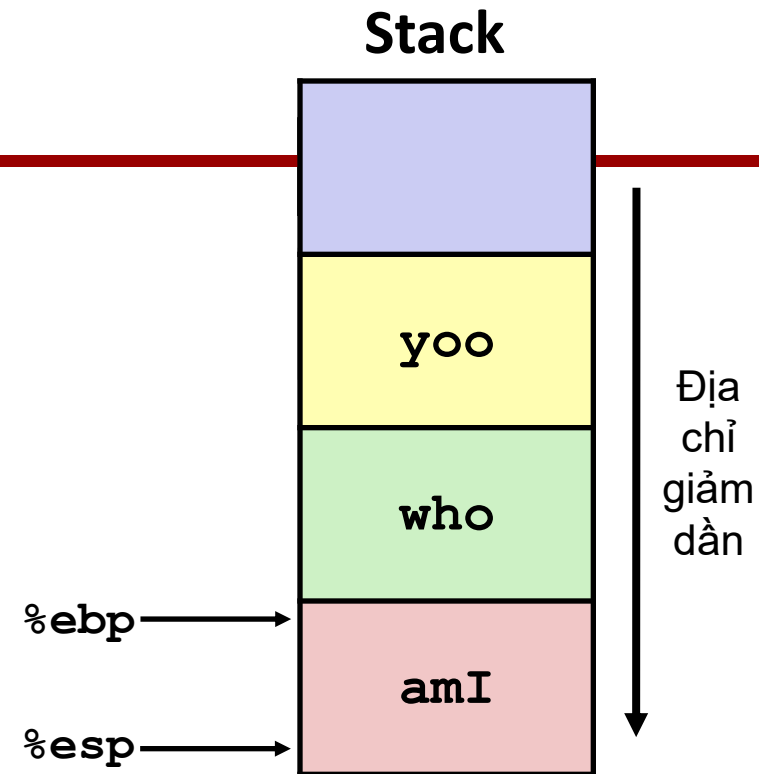
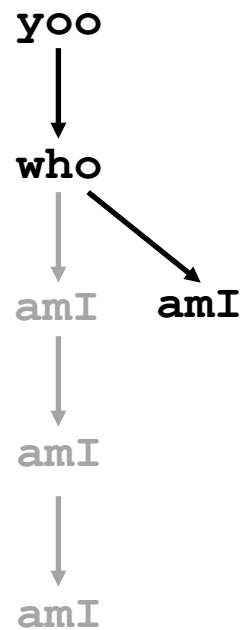
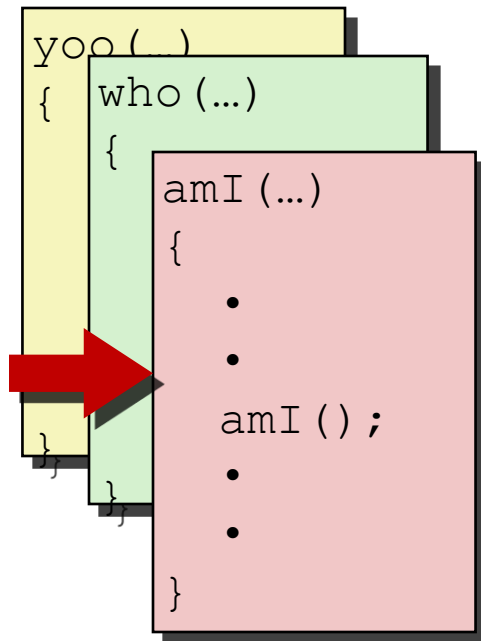
Ví dụ



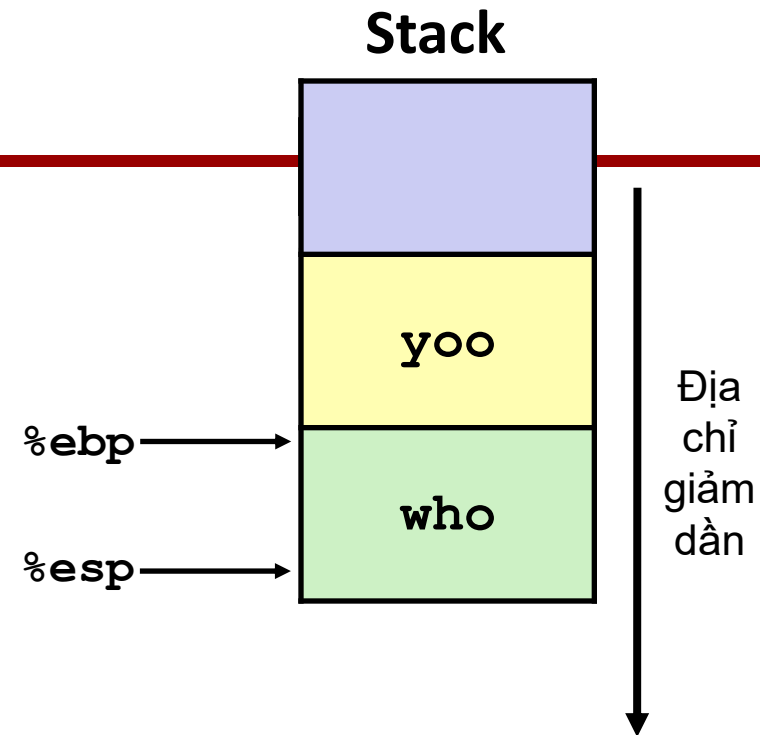
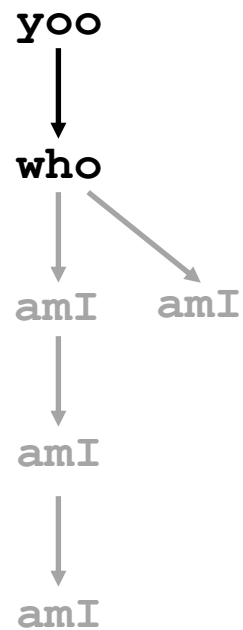
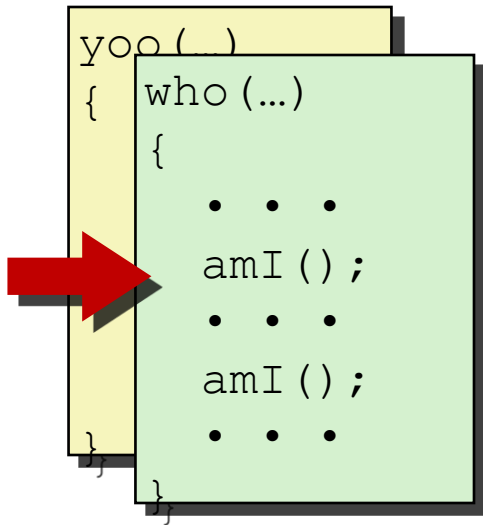
Ví dụ



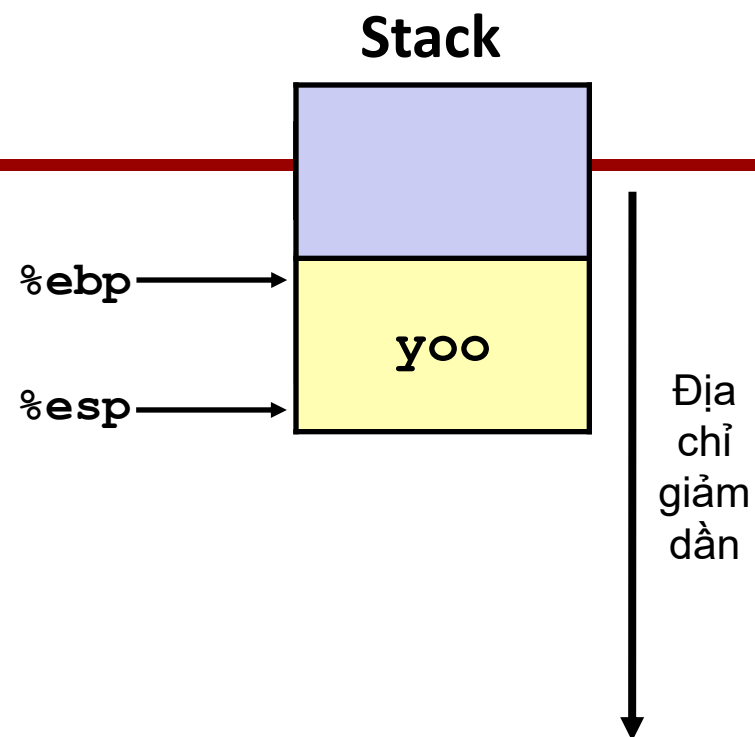
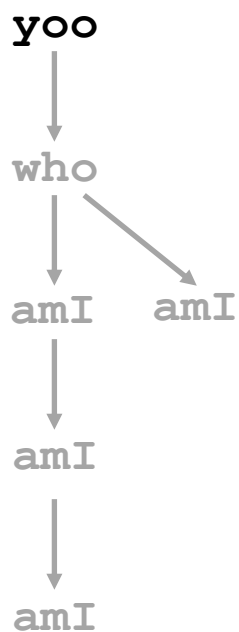
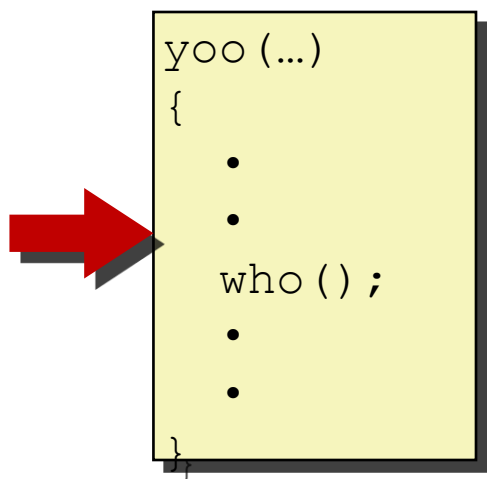
Ví dụ



Ví dụ



Ví dụ



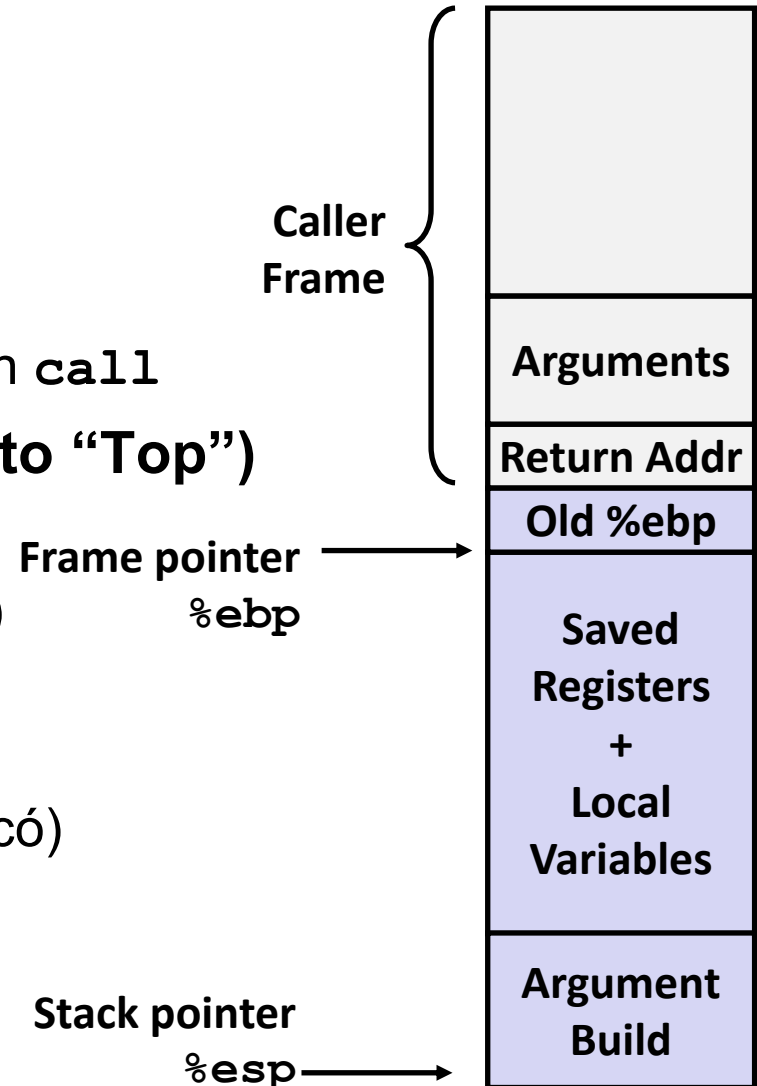
IA32 Stack Frame chứa thông tin gì?

■ Stack frame của hàm mẹ

- Các tham số cho hàm con
 - Đưa vào bằng các lệnh push/mov
- Địa chỉ trả về (Return address)
 - Tự động đẩy vào stack khi chạy lệnh `call`

■ Stack Frame của 1 hàm (“Bottom” to “Top”)

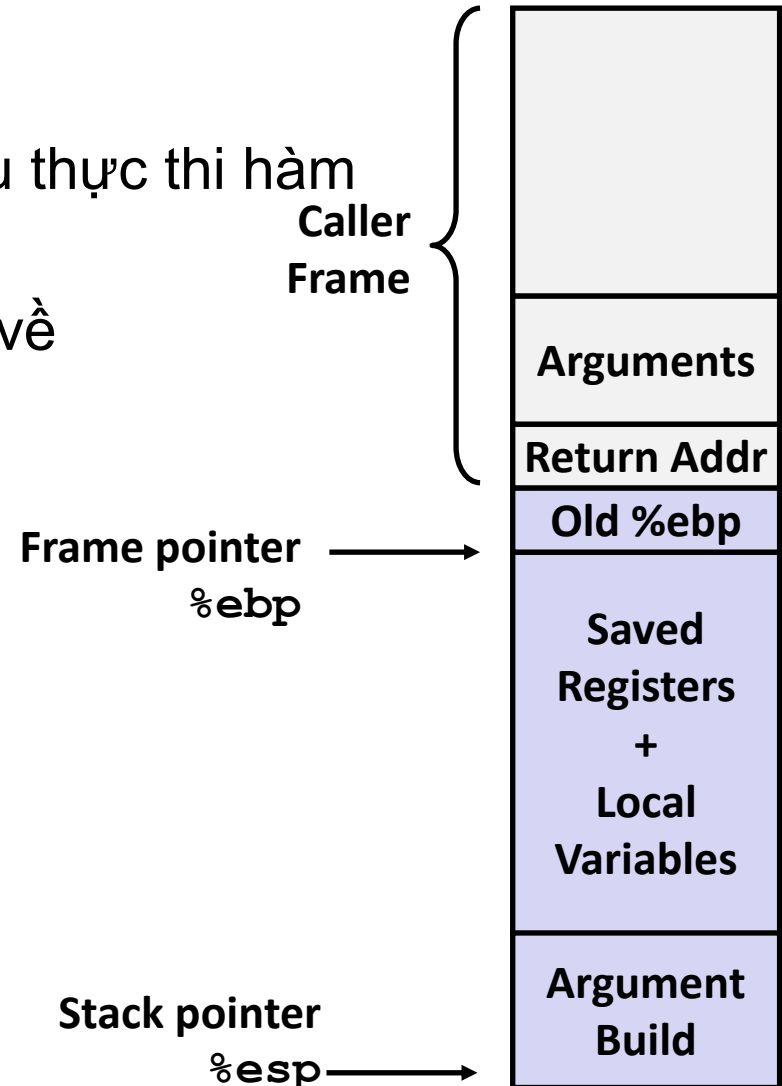
- Frame pointer của hàm mẹ (`%ebp`)
- Những thanh ghi được lưu lại (nếu có)
- Các biến cục bộ của hàm
- “Argument build”
Tham số cho các hàm muốn gọi (nếu có)



Quản lý IA32 Stack Frame

■ 2 hoạt động:

- **Cấp phát** không gian khi bắt đầu thực thi hàm
 - “Set-up” code trong assembly
- **Thu hồi** không gian khi hàm trả về
 - “Finish” code trong assembly



IA32 Stack frame - Set up & Finish

■ Stack Frame – Set up

- Thực hiện khi 1 hàm bắt đầu thực thi
- Lưu lại %ebp của hàm trước
- Thiết lập %ebp cho stack frame của nó
- Lưu lại các thanh ghi sẽ sử dụng trong hàm (nếu có)

swap:

```
pushl %ebp
movl  %esp, %ebp
pushl %ebx
```

} Set Up

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

■ Stack frame - Finish

- Thực hiện khi 1 hàm chuẩn bị trả về
- Khôi phục giá trị cũ của các thanh ghi đã sử dụng (nếu có)
- Khôi phục %ebp của hàm trước

```
popl  %ebx
popl  %ebp
ret
```

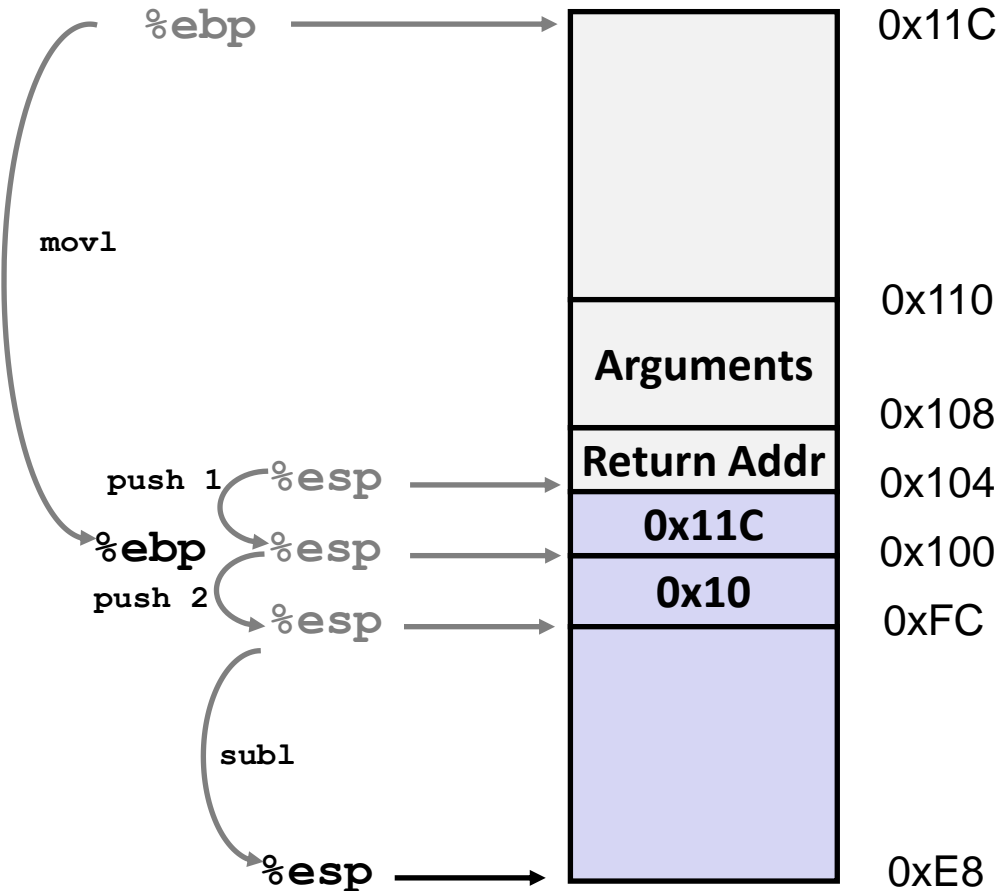
} Finish

Stack frame set up – Ví dụ

Stack sau khi thực hiện `call example`:

`%esp = 0x104`, `%ebp = 0x11C`

`%ebx = 0x10`



example:

```
pushl %ebp
movl %esp, %ebp
pushl %ebx
subl $20, %esp
```

Set Up

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

```
...
addl $20, %esp
popl %ebx
popl %ebp
ret
```

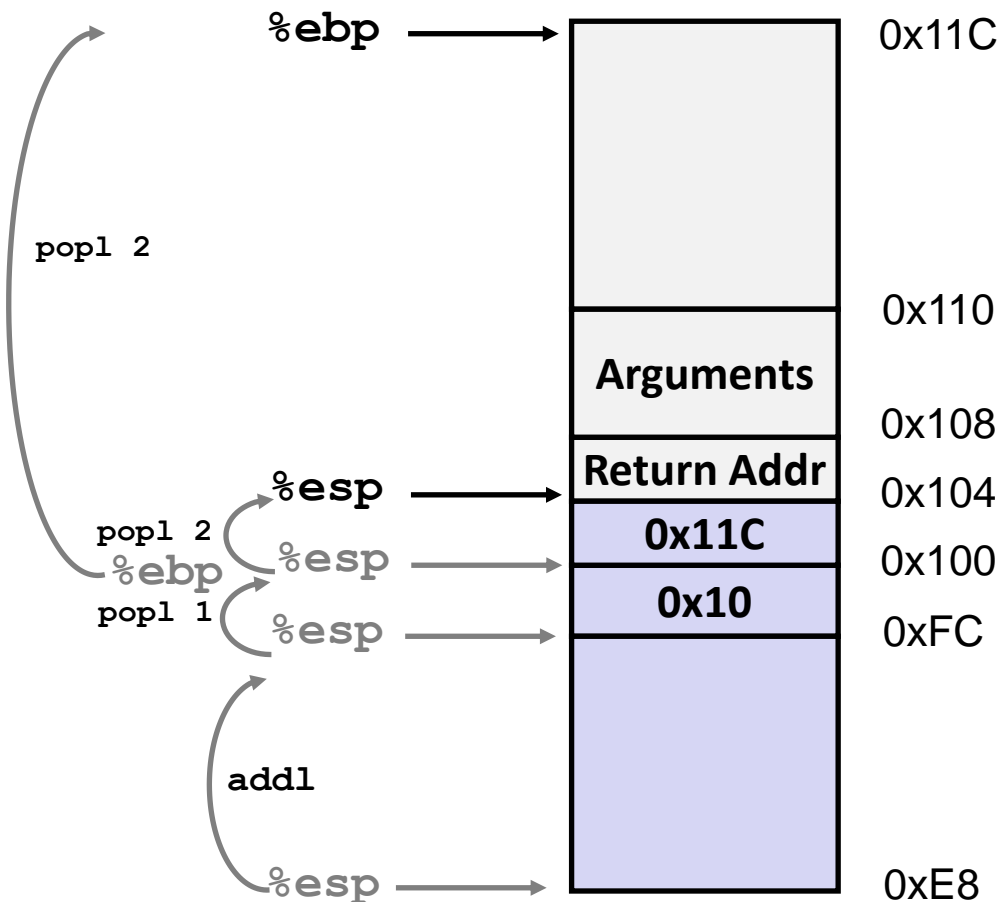
Finish

Stack sau set-up stack frame của **example**:

`%esp = 0xE8`, `%ebp = 0x100`

Stack frame Finish – Ví dụ

%ebx = 0x10



Stack sau finish stack frame của **example**:
%esp = 0x104, %ebp = 0x11C

example:

```
pushl %ebp
movl  %esp, %ebp
pushl %ebx
subl  $20, %esp
```

Set Up

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

...

```
addl  $20, %esp
popl  %ebx
popl  %ebp
ret
```

Finish

Stack frame set up & Finish – Ví dụ

```
int func(int x, int y)
{
    int sum = 0;
    sum = x + y;
    return sum;
}
```

func:

```
pushl    %ebp
movl     %esp, %ebp
subl     $16, %esp
```

} Set Up

```
movl     $0, -4(%ebp)
movl     8(%ebp), %edx
movl     12(%ebp), %eax
addl     %edx, %eax
movl     %eax, -4(%ebp)
movl     -4(%ebp), %eax
```

- Gán %esp = %ebp
- Pop %ebp từ stack

```
← leave
ret
```

} Finish

Nội dung

- **Thủ tục (Procedures)**
 - Cấu trúc stack
 - **Gọi hàm trong IA32**
 - Chuyển luồng
 - Truyền dữ liệu
 - Quản lý dữ liệu cục bộ
- Dịch ngược – Reverse Engineering

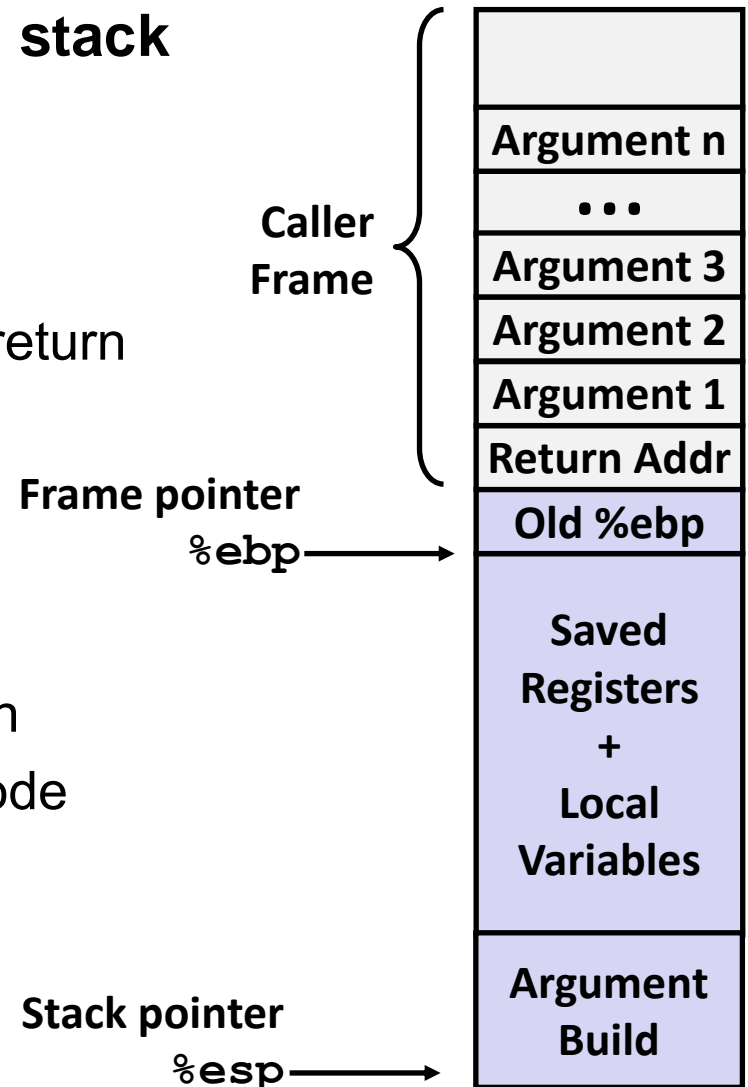
Truyền tham số trong Stack frame IA32

- **Hàm mẹ (caller) đưa tham số vào stack cho hàm con (callee)**

- Trước khi thực thi `call label`
 - Lệnh `push/mov`
 - Nằm ngay phía trước địa chỉ trả về (return address) trong stack
- Thứ tự: reverse order


- **Hàm con (callee) truy xuất tham số**

- Dựa trên vị trí so với `%ebp` của hàm con
 - `%ebp` sau khi hoàn thành “set up” code

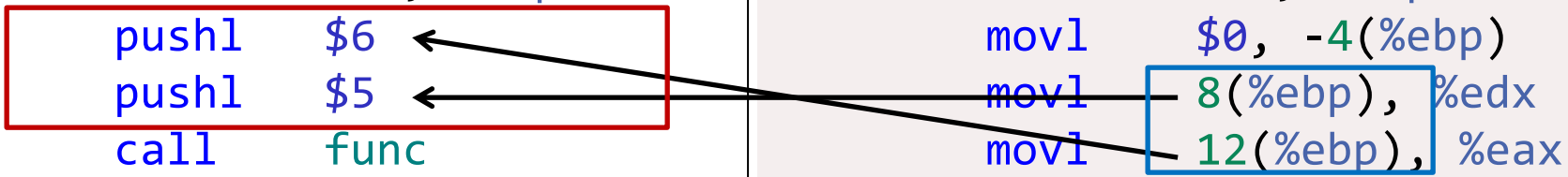


Truyền tham số cho hàm – Ví dụ 1

```
int main()  
{  
    int result = func(5,6);  
    return result;  
}  
  
int func(int x, int y)  
{  
    int sum = 0;  
    sum = x + y;  
    return sum;  
}
```



main:		func:	
pushl	%ebp	pushl	%ebp
movl	%esp, %ebp	movl	%esp, %ebp
subl	\$16, %esp	subl	\$16, %esp
pushl	\$6	movl	\$0, -4(%ebp)
pushl	\$5	movl	8(%ebp), %edx
call	func	movl	12(%ebp), %eax
addl	\$8, %esp	addl	%edx, %eax
movl	%eax, -4(%ebp)	movl	%eax, -4(%ebp)
movl	-4(%ebp), %eax	movl	-4(%ebp), %eax
leave		leave	
ret		ret	



Truyền tham số cho hàm: Ví dụ 2 - swap

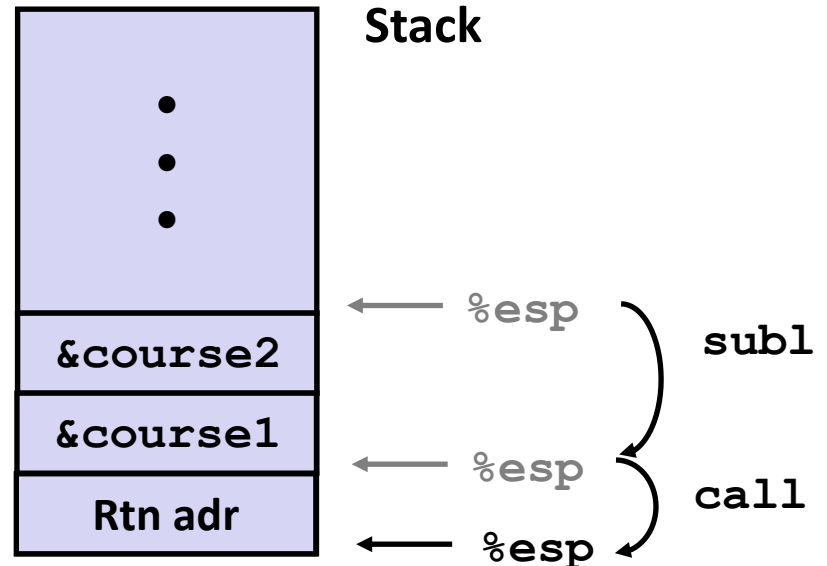
```
int course1 = 15213;
int course2 = 18243;

void call_swap() {
    swap(&course1, &course2);
}
```

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

Gọi swap từ hàm call_swap

```
call_swap:
    . . .
    subl    $8, %esp
    movl    $course2, 4(%esp)
    movl    $course1, (%esp)
    call    swap
    . . .
```



Truyền tham số: Ví dụ swap

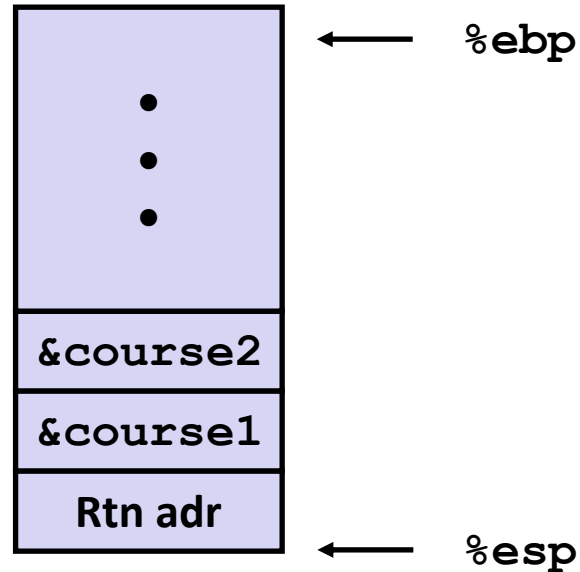
```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

swap:

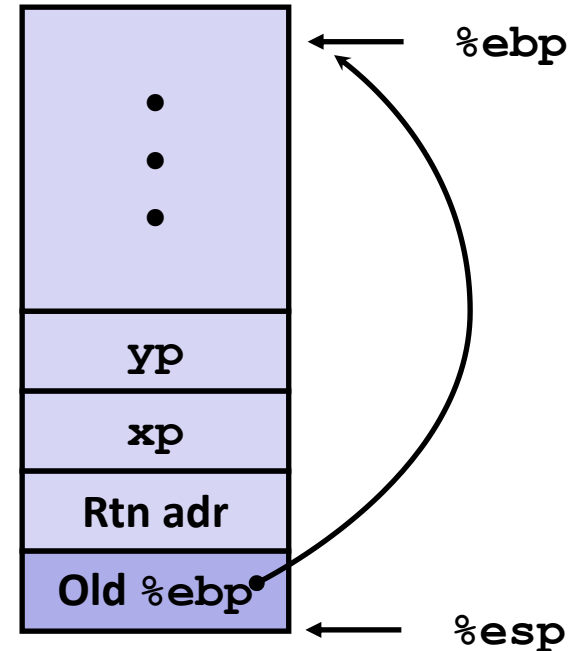
pushl %ebp	}	Set Up
movl %esp, %ebp		
pushl %ebx		
movl 8(%ebp), %edx	}	Body
movl 12(%ebp), %ecx		
movl (%edx), %ebx		
movl (%ecx), %eax		
movl %eax, (%edx)		
movl %ebx, (%ecx)		
popl %ebx	}	Finish
popl %ebp		
ret		

swap Setup #1

Entering Stack



Resulting Stack



swap:

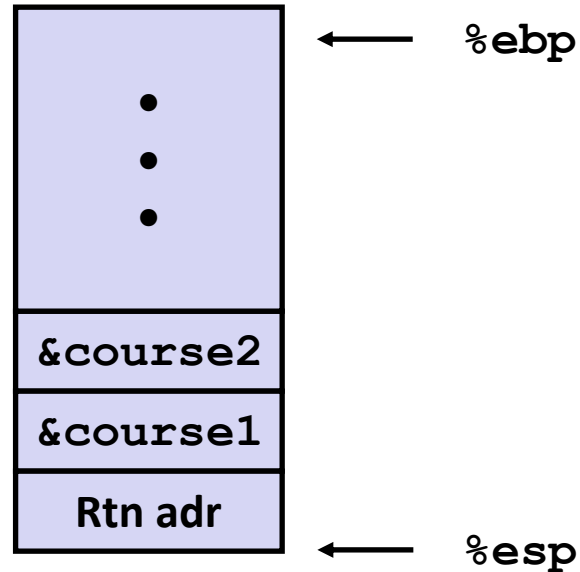
`pushl %ebp`

`movl %esp,%ebp`

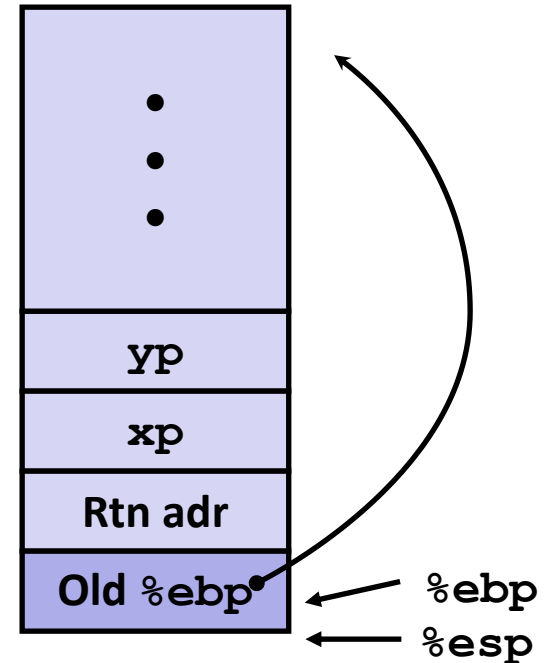
`pushl %ebx`

swap Setup #2

Entering Stack



Resulting Stack



`swap:`

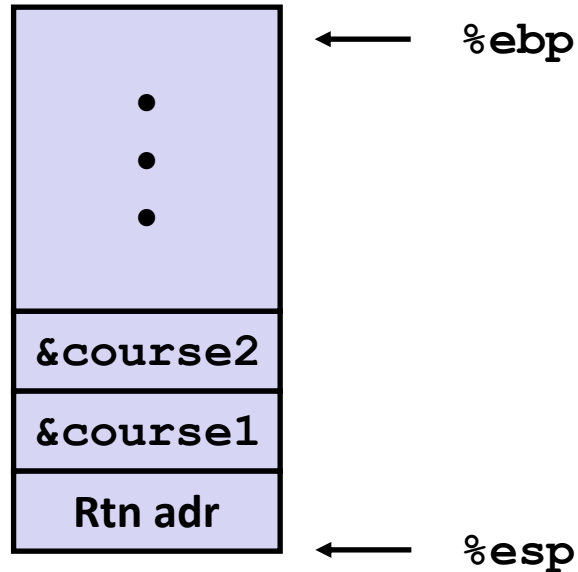
`pushl %ebp`

`movl %esp, %ebp`

`pushl %ebx`

swap Setup #3

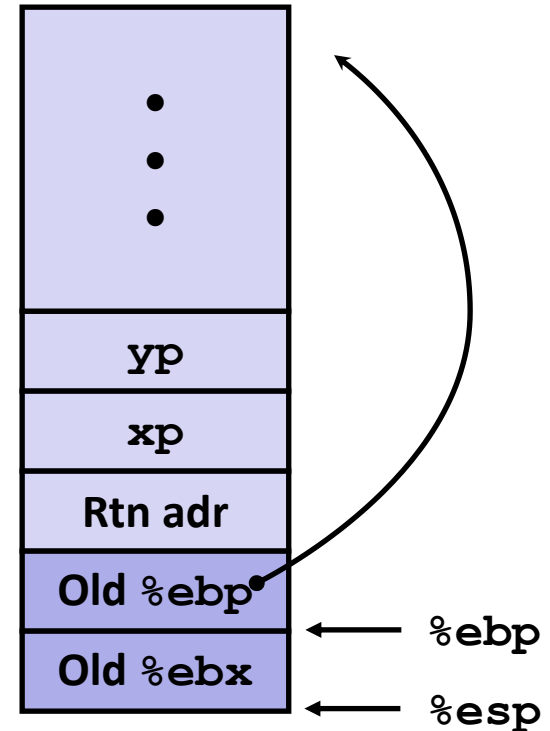
Entering Stack



`swap:`

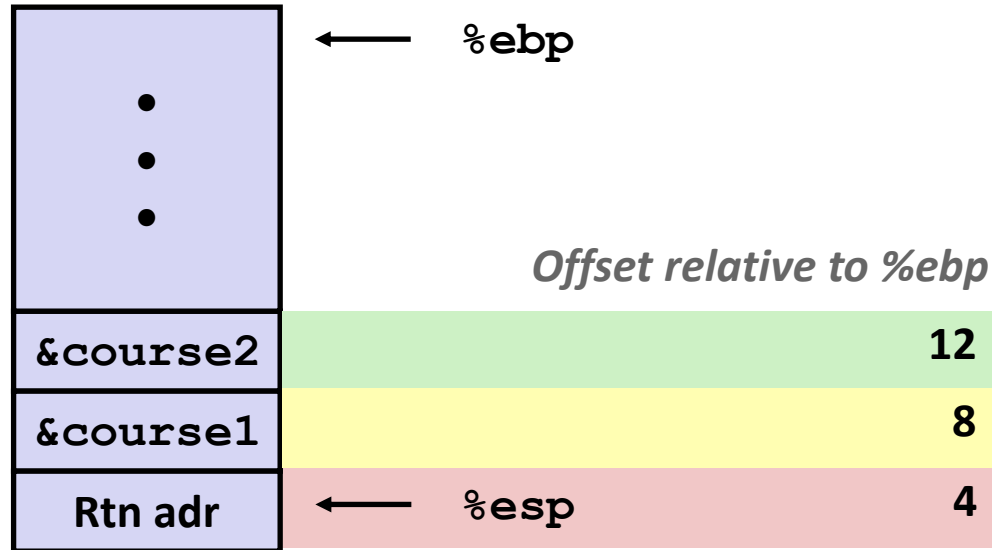
```
    pushl %ebp  
    movl %esp, %ebp  
    pushl %ebx
```

Resulting Stack

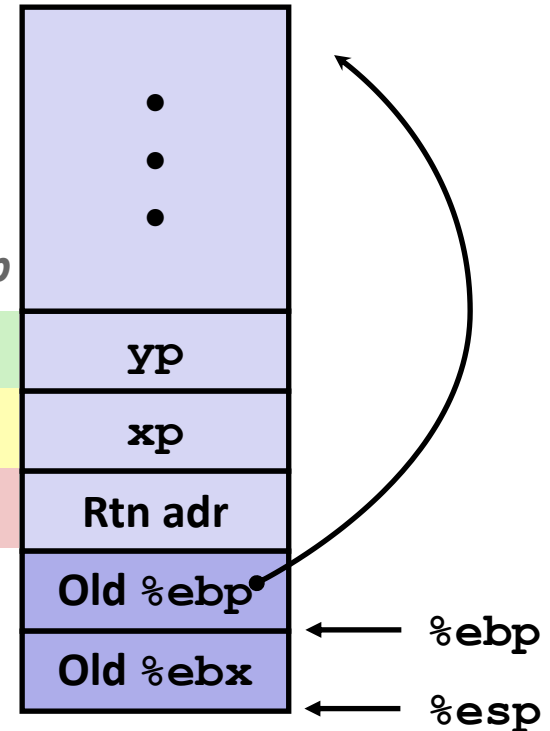


swap: Lấy các tham số

Entering Stack



Resulting Stack



```
movl 8(%ebp),%edx    # get xp
movl 12(%ebp),%ecx   # get yp
. . .
```

Giá trị trả về từ hàm

■ Hàm có trả về giá trị

- Trong C: qua lệnh **return x**.

■ Giá trị trả về của hàm trong assembly

- Thường lưu trong thanh ghi **%eax**

```
int Q(int i)
{
    int t = 3*i;
    int v[10];
    .
    .
    return v[t];
}
```

```
1  int func(int x, int y)
2  {
3      return x + y;
4  }
```



```
1  func:
2      pushl    %ebp
3      movl     %esp, %ebp
4      movl     8(%ebp), %edx
5      movl     12(%ebp), %eax
6      addl     %edx, %eax
7      popl     %ebp
8      ret
```


Giá trị trả về từ hàm – Ví dụ

```
int main()
{
    int result = func(5,6);
    return result;
}
```

```
int func(int x, int y)
{
    int sum = 0;
    sum = x + y;
    return sum;
}
```

main:

```
    pushl    %ebp
    movl     %esp, %ebp
    subl     $16, %esp
    pushl     $6
    pushl     $5
    call     func
    addl     $8, %esp
    movl     %eax, -4(%ebp)
    movl     -4(%ebp), %eax
    leave
    ret
```

func:

```
    pushl    %ebp
    movl     %esp, %ebp
    subl     $16, %esp
    movl     $0, -4(%ebp)
    movl     8(%ebp), %edx
    movl     12(%ebp), %eax
    addl     %edx, %eax
    movl     %eax, -4(%ebp)
    movl     -4(%ebp), %eax
    leave
    ret
```

Nội dung

- **Thủ tục (Procedures)**
 - Cấu trúc stack
 - **Gọi hàm trong IA32**
 - Chuyển luồng
 - Truyền dữ liệu
 - Quản lý dữ liệu cục bộ
- Dịch ngược – Reverse Engineering

Sử dụng thanh ghi cho trong hàm

- Giả sử yoo là hàm mẹ, gọi hàm who
- Có thể dùng thanh ghi để lưu trữ tạm?

```
yoo:  
  . . .  
  movl $15213, %edx  
  call who  
  addl %edx, %eax  
  . . .  
  ret
```

```
who:  
  . . .  
  movl 8(%ebp), %edx  
  addl $18243, %edx  
  . . .  
  ret
```

- Giá trị của thanh ghi `%edx` bị ghi đè trong hàm `who`
- Có thể gây ra vấn đề → cần lưu lại!

Quy ước lưu các thanh ghi

■ Giả sử yoo gọi who:

- yoo là hàm mẹ (caller)
- who là hàm con (callee)

■ Quy ước

- “Caller Save”
 - Hàm mẹ lưu lại các giá trị tạm thời trong stack frame của nó **trước khi gọi** hàm con
- “Callee Save”
 - Hàm con lưu lại các giá trị tạm thời trong stack của nó **trước khi sử dụng**

Sử dụng các thanh ghi IA32/Linux + Windows

■ **%eax, %edx, %ecx**

- Hàm mẹ lưu trước khi gọi nếu giá trị sẽ được sử dụng tiếp

■ **%eax**

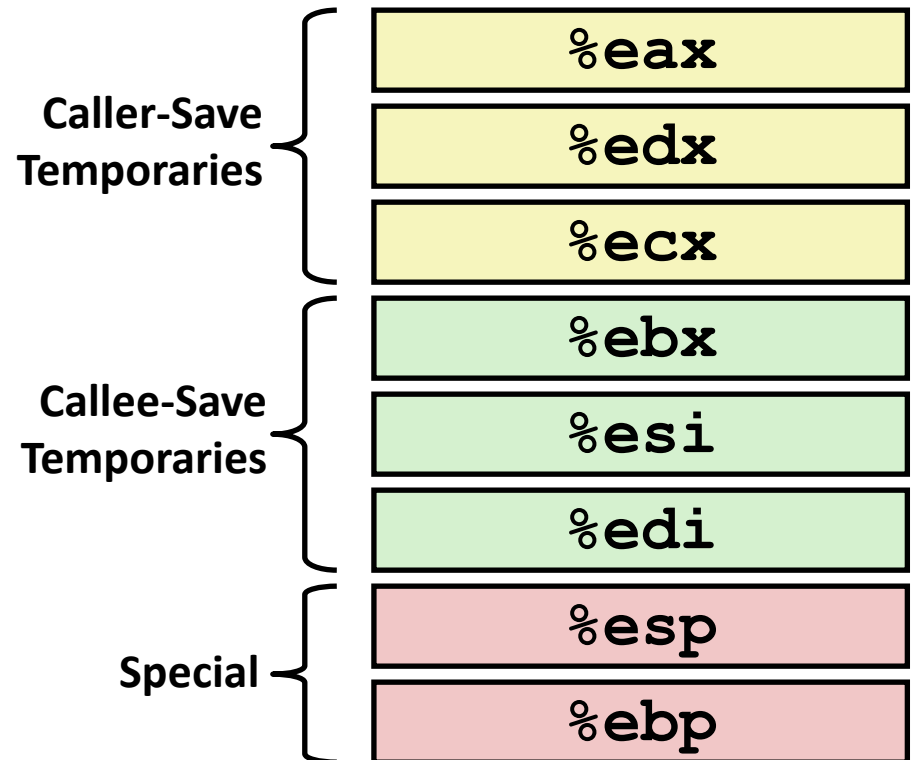
- được sử dụng để trả về giá trị số nguyên

■ **%ebx, %esi, %edi**

- Hàm con sẽ lưu nếu muốn sử dụng

■ **%esp, %ebp**

- Trường hợp đặc biệt cần hàm con lưu
- Khôi phục lại giá trị ban đầu trước khi thoát hàm



Khởi tạo biến cục bộ: Ví dụ

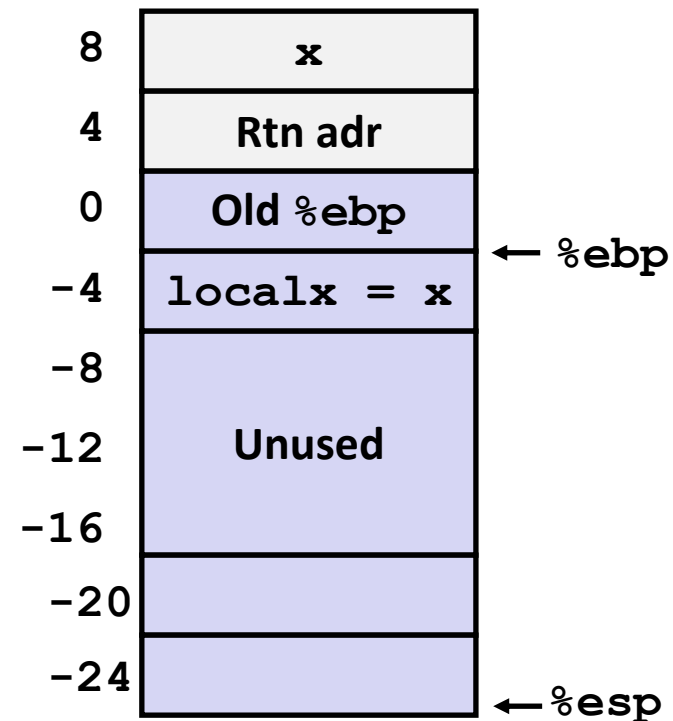
■ Biến cục bộ

- Cấp phát vùng nhớ trong stack để lưu các biến cục bộ của hàm
- Truy xuất dựa trên `%ebp`
 - Địa chỉ thấp hơn so với `%ebp`

```
int add3(int x) {  
    int localx = x;  
    incr(&localx, 3);  
    return localx;  
}
```

First part of add3

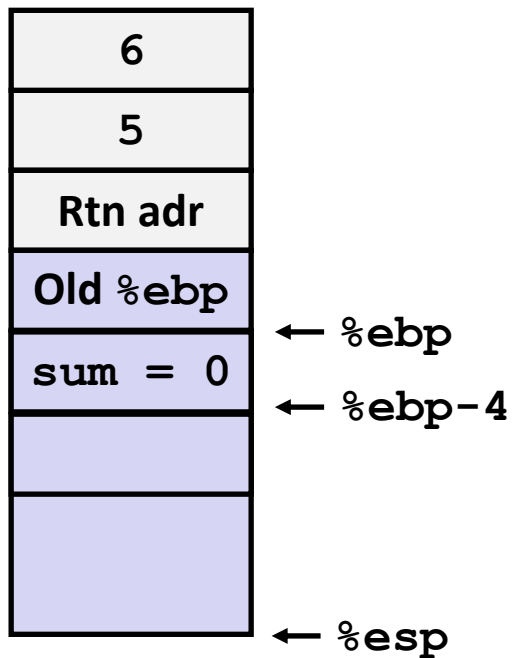
```
add3:  
    pushl %ebp  
    movl %esp, %ebp  
    subl $24, %esp      # Alloc. 24 bytes  
    movl 8(%ebp), %eax  
    movl %eax, -4(%ebp) # Set localx to x
```



Biến cục bộ – Ví dụ

```
int main()
{
    int result = func(5,6);
    return result;
}
```

```
int func(int x, int y)
{
    int sum = 0;
    sum = x + y;
    return sum;
}
```



func:

```
pushl    %ebp
movl     %esp, %ebp
subl     $16, %esp
movl     $0, -4(%ebp)
movl     8(%ebp), %edx
movl     12(%ebp), %eax
addl     %edx, %eax
movl     %eax, -4(%ebp)
movl     -4(%ebp), %eax
leave
ret
```

Gọi hàm (IA32): Tổng kết

- **Stack đóng vai trò quan trọng trong gọi/trả về hàm**
 - Lưu trữ địa chỉ trả về
 - Các tham số (trong stack frame hàm mẹ)
 - Có thể lưu các giá trị trong stack frame hoặc các thanh ghi
 - Giá trị trả về ở thanh ghi %eax

Bài tập gọi hàm 1

main:

```
pushl    %ebp
movl     %esp, %ebp
subl     $16, %esp
movl     $1, -4(%ebp)
movl     $2, -8(%ebp)
movl     $0, -12(%ebp)
pushl    -4(%ebp)
pushl    -8(%ebp)
call     function
addl     $8, %esp
movl     %eax, -12(%ebp)
movl     $0, %eax
leave
ret
```

function:

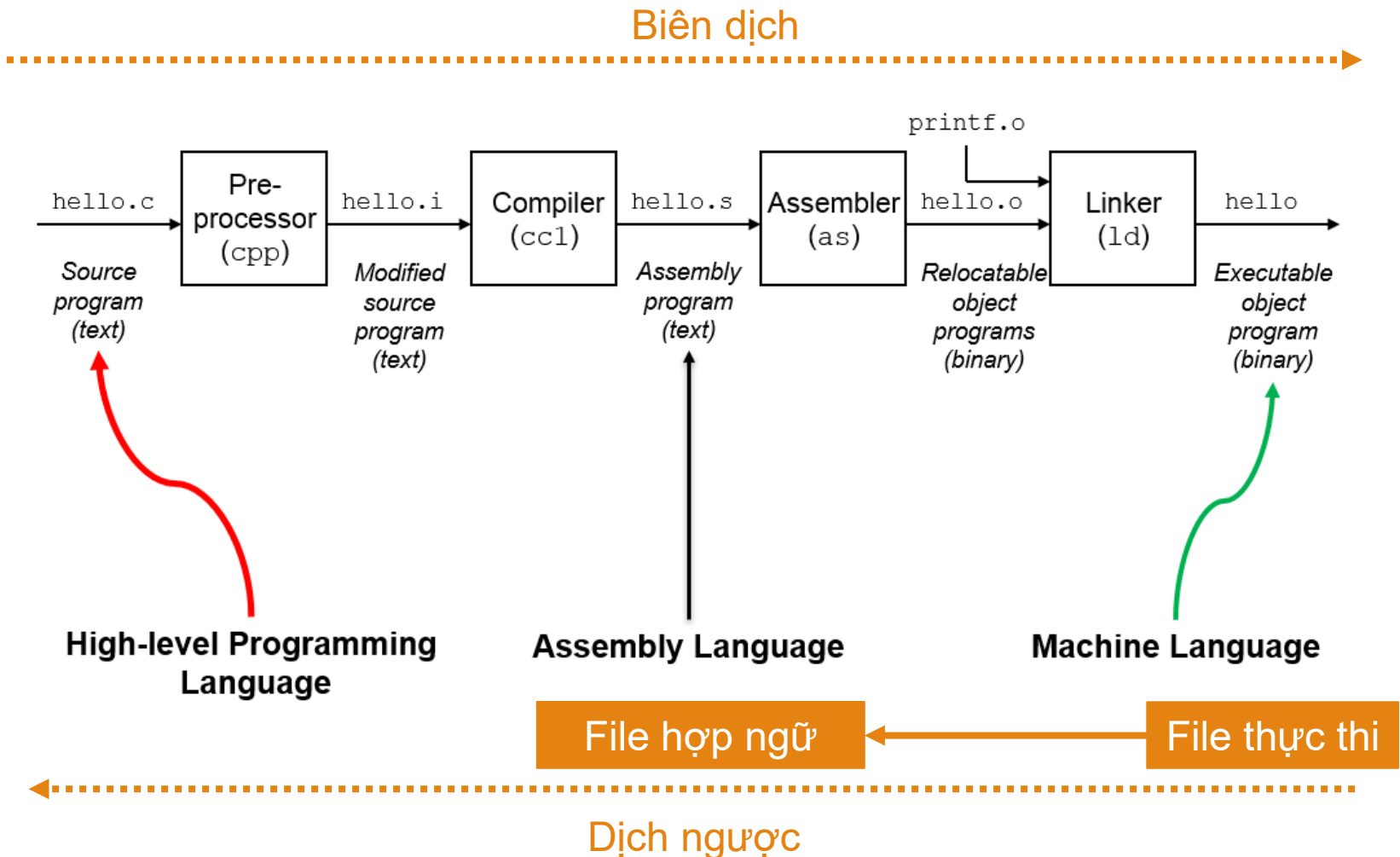
```
pushl    %ebp
movl     %esp, %ebp
subl     $16, %esp
movl     $10, -4(%ebp)    # a
movl     -4(%ebp), %edx
movl     8(%ebp), %eax    # x
addl     %eax, %edx
movl     12(%ebp), %eax   # y
imull    %edx, %eax
movl     %eax, -8(%ebp)
movl     -8(%ebp), %eax   # result
leave
ret
```

1. Hàm nào là caller/callee?
2. Mỗi hàm có bao nhiêu biến cục bộ? Giá trị như thế nào?
3. Hàm function nhận bao nhiêu tham số?
4. Hàm main đã truyền các tham số có giá trị cho function?
5. Hàm function làm gì? Với các giá trị tham số đã tìm thấy ở Câu 4, tìm giá trị được function trả về cho main?

Nội dung

- Thủ tục (Procedures)
 - Cấu trúc stack
 - Gọi hàm trong IA32
 - Chuyển luồng
 - Truyền dữ liệu
 - Quản lý dữ liệu cục bộ
- Dịch ngược – Reverse Engineering

Dịch ngược - Reverse Engineering?



Dịch ngược - Reverse Engineering?

■ Dịch ngược

- Từ một file thực thi (executable file) của chương trình, chuyển về dạng mã hợp ngữ (assembly) để đọc/hiểu hoạt động của nó.

```
55
89 e5
83 ec 10
c7 45 fc 00 00 00 00
8b 45 08
01 45 fc
8b 45 fc
c9
c4
```



File thực thi (binary)

RE

```
pushl    %ebp
movl     %esp, %ebp
subl     $16, %esp
movl     $0, -4(%ebp)
movl     8(%ebp), %eax
addl     %eax, -4(%ebp)
movl     -4(%ebp), %eax
leave
ret
```



File hợp ngữ (assembly)

Dịch ngược – Công cụ (1)

■ objdump – Xuất mã assembly của file thực thi

```
ubuntu@ubuntu:~$ objdump -d basic-reverse

basic-reverse:      file format elf32-i386

Disassembly of section .init:

0804841c <_init>:
804841c:      53                push    %ebx
804841d:      83 ec 08          sub     $0x8,%esp
8048420:      e8 0b 01 00 00    call   8048530 <__x86.get_pc_thunk.bx>
8048425:      81 c3 db 1b 00 00    add     $0x1bdb,%ebx
804842b:      8b 83 fc ff ff ff    mov     -0x4(%ebx),%eax
8048431:      85 c0             test    %eax,%eax
8048433:      74 05             je      804843a <_init+0x1e>
8048435:      e8 b6 00 00 00    call   80484f0 <__isoc99_scanf@plt+0x10>
804843a:      83 c4 08          add     $0x8,%esp
804843d:      5b               pop     %ebx
804843e:      c3               ret
```

- Command line
- Thường có trên Linux
- Định dạng assembly mặc định: AT&T
- **Chỉ hiển thị mã assembly**, không hỗ trợ chức năng phân tích

Dịch ngược – Công cụ (2)

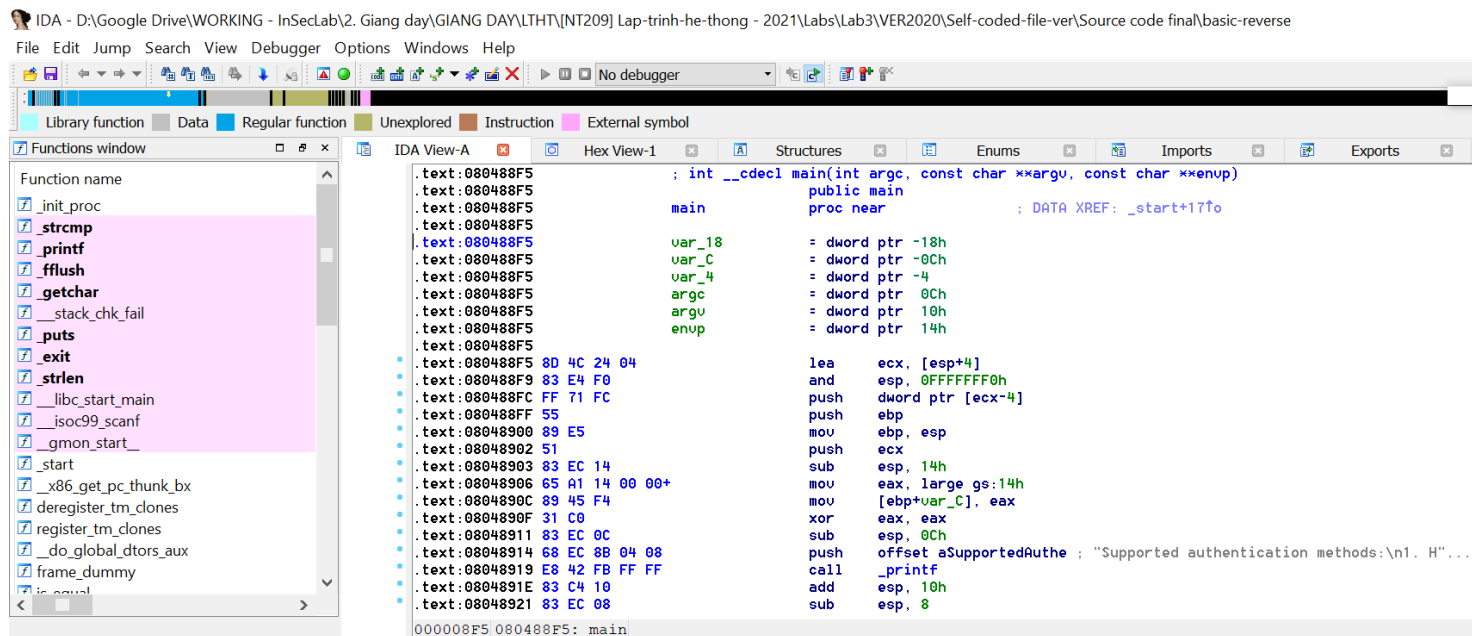
■ GDB Debugger (Phần 3.11 trong giáo trình chính)

```
ubuntu@ubuntu: ~$ gdb basic-reverse
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://www.gnu.org/licenses/gpl.html>
This is free software: you are free to copy and modify it under the terms of the GNU GPL.
There is NO WARRANTY, to the extent permitted by the license.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see
<http://www.gnu.org/software/gdb/bugs>.
Find the GDB manual and other documentation
<http://www.gnu.org/software/gdb/documentation>.
For help, type "help".
Type "apropos word" to search for commands.
Reading symbols from basic-reverse...
(gdb) disassemble
No frame selected.
Dump of assembler code for function main:
0x080488f5 <+0>:    lea    0x4(%esp),%ecx
0x080488f9 <+4>:    and    $0xffffffff0,%esp
0x080488fc <+7>:    pushl  -0x4(%ecx)
0x080488ff <+10>:   push   %ebp
0x08048900 <+11>:   mov    %esp,%ebp
0x08048902 <+13>:   push   %ecx
=> 0x08048903 <+14>:   sub    $0x14,%esp
0x08048906 <+17>:   mov    %gs:0x14,%eax
0x0804890c <+23>:   mov    %eax,-0xc(%ebp)
0x0804890f <+26>:   xor    %eax,%eax
0x08048911 <+28>:   sub    $0xc,%esp
0x08048914 <+31>:   push   $0x8048bec
0x08048919 <+36>:   call   0x8048460 <printf@plt>
0x0804891e <+41>:   add    $0x10,%esp
0x08048921 <+44>:   sub    $0x8,%esp
0x08048924 <+47>:   lea    -0x18(%ebp),%eax
0x08048927 <+50>:   push   %eax
```

- Command line
- Thường có trên Linux
- Định dạng assembly mặc định: AT&T
- **Có thể phân tích code ở dạng tĩnh và động (các chương trình cần chạy được trên hệ thống)**

Dịch ngược – Công cụ (3)

■ IDA Pro



- Có giao diện, nhiều cửa sổ cung cấp nhiều thông tin, có view mã giả
- Có thể chạy trên Windows
- Định dạng assembly: Intel
- Có thể phân tích code ở dạng tĩnh và động (các chương trình cần chạy được trên hệ thống)

Dịch ngược: Demo

- **File cần phân tích: `first_re_demo`**
 - File thực thi trên Linux 32 bit
 - Dạng command line
 - 1 hàm thực thi chính: **`main`**
 - Yêu cầu nhập 1 password.

Nội dung

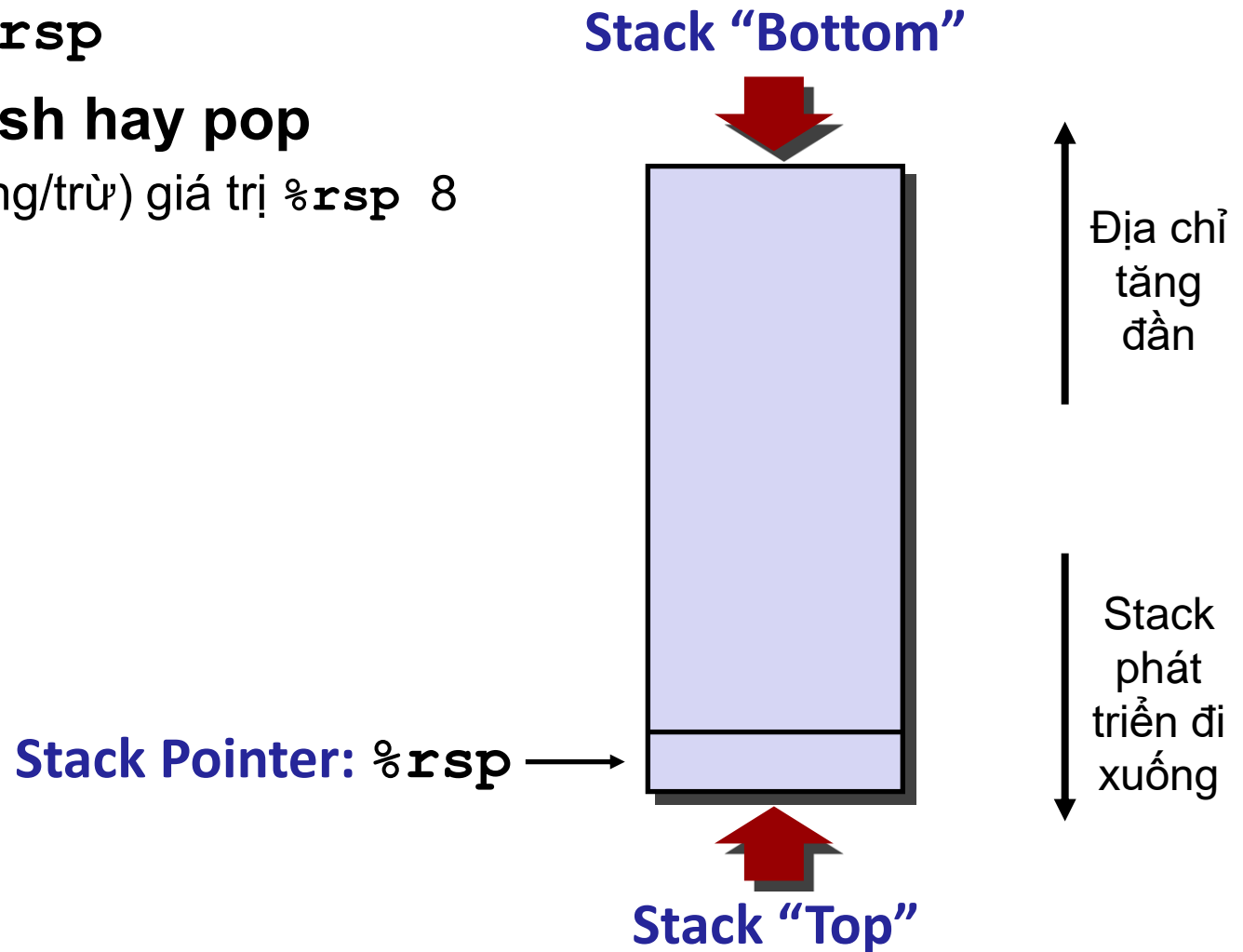
- **Thủ tục (Procedures)**
 - Cấu trúc stack
 - Gọi hàm trong IA32
 - Chuyển luồng
 - Truyền dữ liệu
 - Quản lý dữ liệu cục bộ
 - **Gọi hàm trong x86-64**
 - Minh hoạ hàm đệ quy
- Bài tập về hàm
- Dịch ngược – Reverse engineering

Điểm chung của hàm trong IA32 và x86-64

- Stack hỗ trợ việc gọi hàm
- Sử dụng lệnh **call**
 - Địa chỉ trả về (return address) được đưa vào stack
 - Địa chỉ câu lệnh assembly ngay sau lệnh **call**

x86-64 Stack?

- Thanh ghi `%rsp`
- Các lệnh `push` hay `pop`
 - Thay đổi (cộng/trừ) giá trị `%rsp` 8 bytes



Thanh ghi x86-64

%rax	%eax
%rbx	%ebx
%rcx	%ecx
%rdx	%edx
%rsi	%esi
%rdi	%edi
%rsp	%esp
%rbp	%ebp

%r8	%r8d
%r9	%r9d
%r10	%r10d
%r11	%r11d
%r12	%r12d
%r13	%r13d
%r14	%r14d
%r15	%r15d

- Số thanh ghi nhiều hơn gấp 2 lần
- Có thể truy xuất với các kích thước 8, 16, 32, 64 bits

Sử dụng các thanh ghi x86-64

- **Tham số được truyền cho hàm thông qua các thanh ghi**
 - Hỗ trợ truyền 6 tham số
 - Nếu nhiều hơn 6 tham số, các tham số còn lại sẽ truyền qua stack
 - Những thanh ghi này vẫn có thể dùng bình thường caller-saved
- **Tất cả tham chiếu đến giá trị trong stack frame đều qua *stack pointer***
 - Bỏ qua việc cập nhật giá trị `%ebp/%rbp` khi gọi hàm
- **Các thanh ghi khác**
 - 6 thanh ghi callee saved
 - 2 thanh ghi caller saved
 - 1 thanh ghi chứa giá trị trả về (cũng có thể sử dụng như caller saved)
 - 1 thanh ghi đặc biệt (stack pointer)

Truyền dữ liệu trong x86-64

■ Sử dụng các thanh ghi

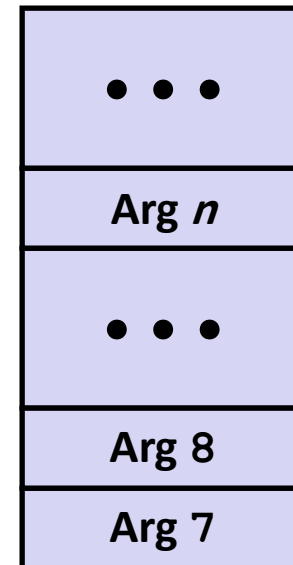
■ 6 tham số đầu tiên

<code>%rdi</code>
<code>%rsi</code>
<code>%rdx</code>
<code>%rcx</code>
<code>%r8</code>
<code>%r9</code>

■ Giá trị trả về

<code>%rax</code>

■ Stack



■ Chỉ cấp phát không gian trong stack khi cần thiết

Thanh ghi x86-64: Quy ước sử dụng

%rax	Return value
%rbx	Callee saved
%rcx	Argument #4
%rdx	Argument #3
%rsi	Argument #2
%rdi	Argument #1
%rsp	Stack pointer
%rbp	Callee saved

%r8	Argument #5
%r9	Argument #6
%r10	Caller saved
%r11	Caller Saved
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved

Sử dụng các thanh ghi x86-64 #1

■ **%rax**

- Giá trị trả về
- Hàm mẹ lưu lại (caller-saved)
- Có thể thay đổi trong hàm

■ **%rdi, ..., %r9**

- Tham số
- Hàm mẹ lưu lại (caller-saved)
- Có thể thay đổi trong hàm

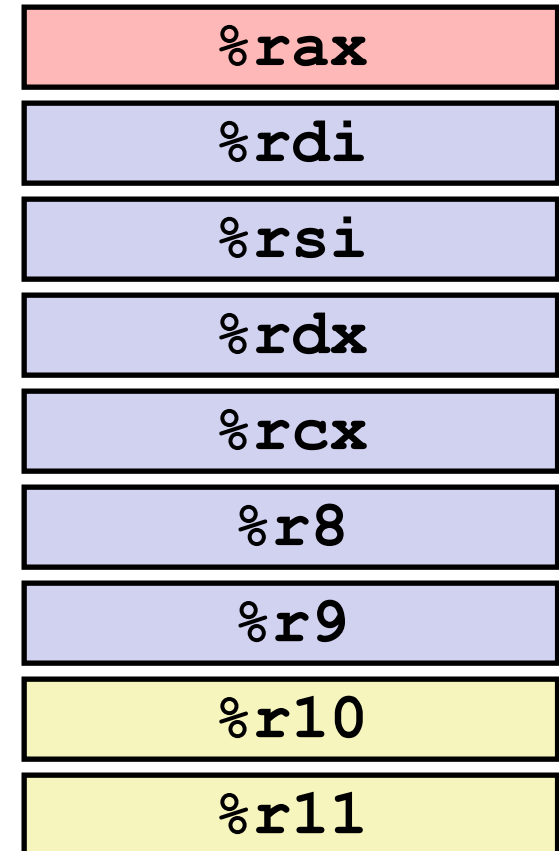
■ **%r10, %r11**

- Hàm mẹ lưu lại (caller-saved)
- Có thể thay đổi trong hàm

Return value

Arguments

Caller-saved
temporaries



Sử dụng các thanh ghi x86-64 #2

■ **%rbx, %r12, %r13, %r14**

- Hàm con lưu lại (callee-saved)
 - Hàm con cần lưu và khôi phục lại
- Callee-saved
Temporaries

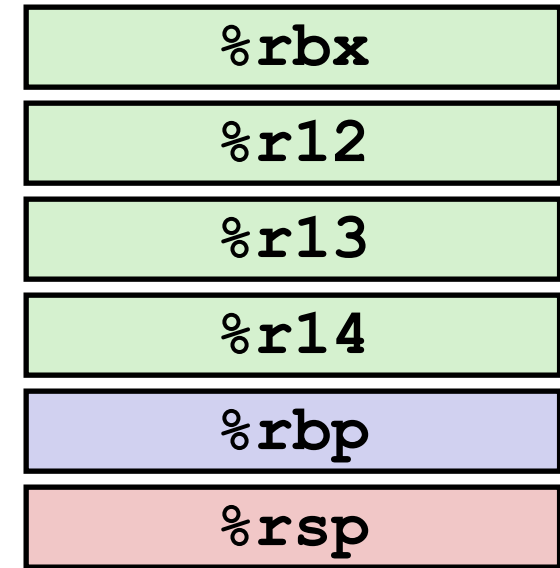
■ **%rbp**

- Hàm con lưu lại (callee-saved)
- Hàm con cần lưu và khôi phục lại
- Có thể dùng như frame pointer

■ **%rsp**

- Trường hợp đặc biệt của callee-saved
- Khôi phục lại giá trị ban đầu khi thoát hàm

Special



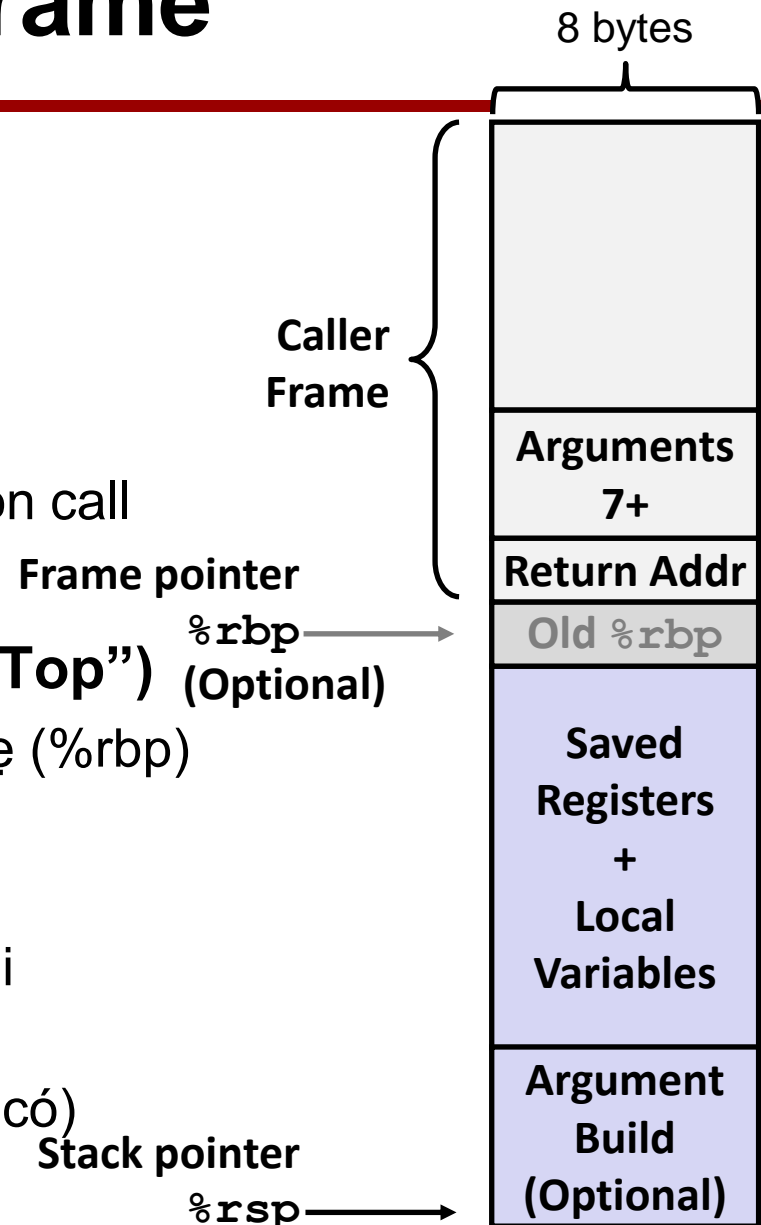
x86-64/Linux Stack Frame

■ Stack Frame của hàm mẹ

- Các tham số cho hàm con
 - +7??
- Địa chỉ trả về (Return address)
 - Được đẩy vào stack bằng instruction call

■ Stack Frame 1 hàm (“Bottom” to “Top”) (Optional)

- **(Optional)** Frame pointer của hàm mẹ (%rbp)
- Những thanh ghi được lưu lại
- Các biến cục bộ của hàm
Nếu không thể lưu trong các thanh ghi
- “Argument build”
Tham số cho các hàm muốn gọi (nếu có)



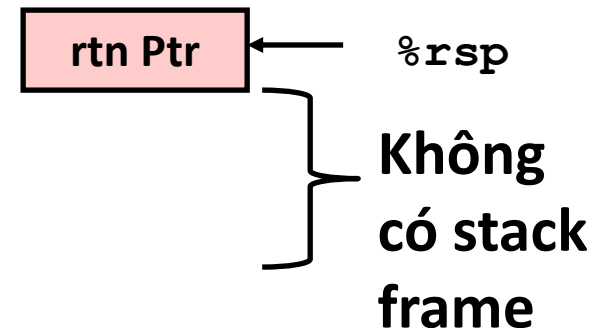
Ví dụ hàm trong x86-64: Long Swap

```
void swap_l(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

swap:

```
movq    (%rdi), %rdx
movq    (%rsi), %rax
movq    %rax, (%rdi)
movq    %rdx, (%rsi)
ret
```

- Tham số truyền qua thanh ghi
 - Tham số 1 (**xp**) trong **%rdi**, Tham số 2 (**yp**) trong **%rsi**
 - Các thanh ghi 64 bit
- Không cần các hoạt động trên stack (trừ **ret**)
- Hạn chế dùng stack
 - Có thể lưu tất cả thông tin trên thanh ghi



Ví dụ hàm trong x86_64: `incr`

```
long incr(long *p, long val) {  
    long x = *p;  
    long y = x + val;  
    *p = y;  
    return x;  
}
```

```
incr:  
    movq    (%rdi), %rax  
    addq    %rax, %rsi  
    movq    %rsi, (%rdi)  
    ret
```

Register	Use(s)
%rdi	Argument <code>p</code>
%rsi	Argument <code>val, y</code>
%rax	<code>x</code> , Return value

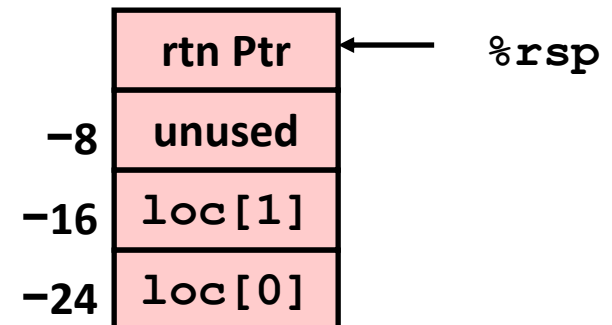
Biến cục bộ trong hàm x86_64 – VD 1

```
/* Swap, using local array */
void swap_a(long *xp, long *yp)
{
    volatile long loc[2];
    loc[0] = *xp;
    loc[1] = *yp;
    *xp = loc[1];
    *yp = loc[0];
}
```

```
swap_a:
    movq    (%rdi), %rax
    movq    %rax, -24(%rsp)
    movq    (%rsi), %rax
    movq    %rax, -16(%rsp)
    movq    -16(%rsp), %rax
    movq    %rax, (%rdi)
    movq    -24(%rsp), %rax
    movq    %rax, (%rsi)
    ret
```

■ Hạn chế thay đổi stack pointer (%rsp)

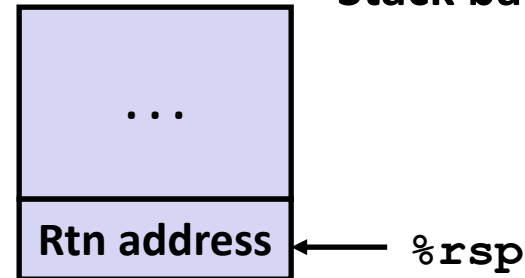
- Có thể lưu tất cả thông tin trong vùng nhớ gần stack pointer



Biến cục bộ trong hàm x86_64 – VD 2 #1

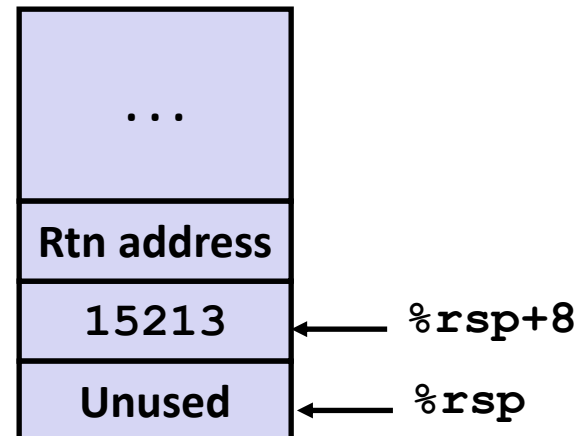
```
long call_incr() {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return v1+v2;  
}
```

Stack ban đầu



```
call_incr:  
    subq    $16, %rsp  
    movq    $15213, 8(%rsp)  
    movl    $3000, %esi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

Stack sau khi thay đổi

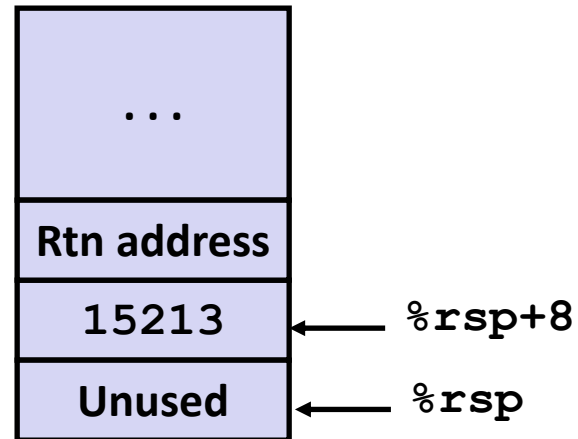


Biến cục bộ trong hàm x86_64 – VD2 #2

```
long call_incr() {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return v1+v2;  
}
```

```
call_incr:  
    subq    $16, %rsp  
    movq    $15213, 8(%rsp)  
    movl    $3000, %esi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

Stack

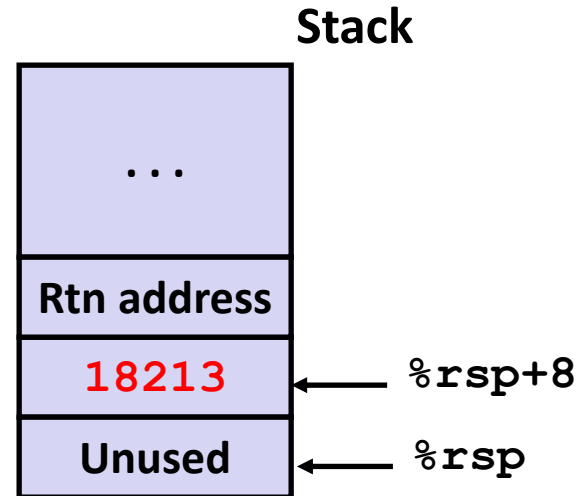


Register	Use(s)
$\%rdi$	&v1
$\%rsi$	3000

Biến cục bộ trong hàm x86_64 – VD2 #2

```
long call_incr() {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return v1+v2;  
}
```

```
call_incr:  
    subq    $16, %rsp  
    movq    $15213, 8(%rsp)  
    movl    $3000, %esi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

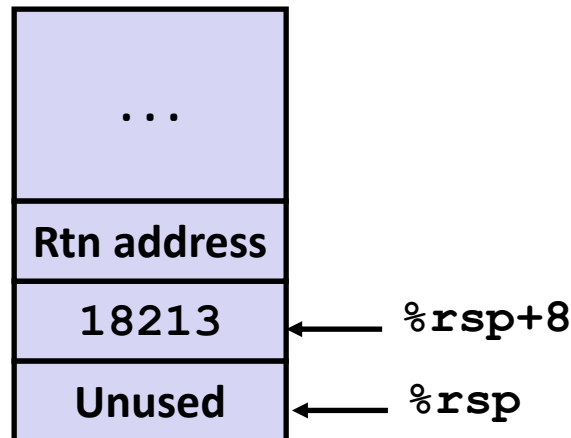


Register	Use(s)
%rdi	&v1
%rsi	3000

Biến cục bộ trong hàm x86_64 – VD2 #3

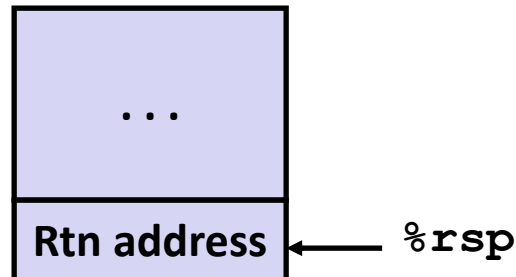
```
long call_incr() {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return v1+v2;  
}
```

```
call_incr:  
    subq    $16, %rsp  
    movq    $15213, 8(%rsp)  
    movl    $3000, %esi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```



Register	Use(s)
%rax	Return value

Stack sau khi cập nhật %rsp

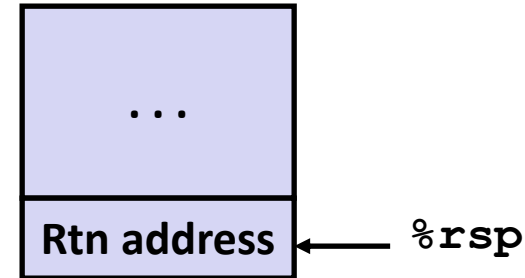


Biến cục bộ trong hàm x86_64 – VD2 #4

```
long call_incr() {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return v1+v2;  
}
```

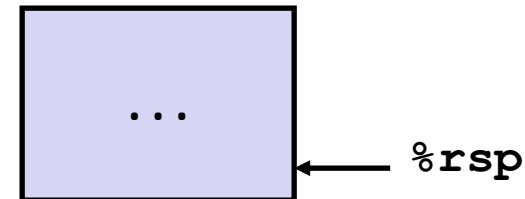
```
call_incr:  
    subq    $16, %rsp  
    movq    $15213, 8(%rsp)  
    movl    $3000, %esi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

Stack sau khi cập nhật %rsp



Register	Use(s)
%rax	Return value

Stack cuối cùng



x86-64 Stack Frame Example

```
long sum = 0;
/* Swap a[i] & a[i+1] */
void swap_ele_su
(long a[], int i)
{
    swap(&a[i], &a[i+1]);
    sum += (a[i]*a[i+1]);
}
```

- Lưu các giá trị `&a[i]` và `&a[i+1]` trong các thanh ghi callee save
- Cần set-up stack frame để lưu những thanh ghi này

```
swap_ele_su:
    movq    %rbx, -16(%rsp)
    movq    %rbp, -8(%rsp)
    subq    $16, %rsp
    movslq   %esi, %rax
    leaq    8(%rdi,%rax,8), %rbx
    leaq    (%rdi,%rax,8), %rbp
    movq    %rbx, %rsi
    movq    %rbp, %rdi
    call    swap
    movq    (%rbx), %rax
    imulq   (%rbp), %rax
    addq    %rax, sum(%rip) # global-scope
                                variable

    movq    (%rsp), %rbx
    movq    8(%rsp), %rbp
    addq    $16, %rsp
    ret
```

Hiểu x86-64 Stack Frame (1)

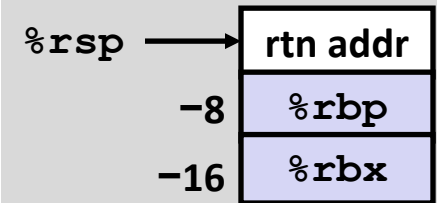
swap_ele_su:

movq	%rbx, -16(%rsp)	# Save %rbx
movq	%rbp, -8(%rsp)	# Save %rbp
subq	\$16, %rsp	# Allocate stack frame
movslq	%esi, %rax	# Extend I (4 -> 8 bytes)
leaq	8(%rdi, %rax, 8), %rbx	# &a[i+1] (callee save)
leaq	(%rdi, %rax, 8), %rbp	# &a[i] (callee save)
movq	%rbx, %rsi	# 2 nd argument
movq	%rbp, %rdi	# 1 st argument
call	swap	
movq	(%rbx), %rax	# Get a[i+1]
imulq	(%rbp), %rax	# Multiply by a[i]
addq	%rax, sum(%rip)	# Add to sum (global variable)
movq	(%rsp), %rbx	# Restore %rbx
movq	8(%rsp), %rbp	# Restore %rbp
addq	\$16, %rsp	# Deallocate frame
ret		

Hiểu x86-64 Stack Frame (2)

```
movq    %rbx, -16(%rsp)
movq    %rbp, -8(%rsp)
```

```
# Save %rbx
# Save %rbp
```



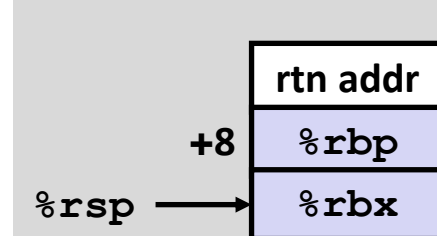
```
subq    $16, %rsp
```

```
# Allocate stack frame
```

● ● ●

```
movq    (%rsp), %rbx
movq    8(%rsp), %rbp
```

```
# Restore %rbx
# Restore %rbp
```



```
addq    $16, %rsp
```

```
# Deallocate frame
```

Đặc điểm thú vị của x86-64 Stack Frame

- **Cấp phát nguyên frame trong 1 lần**
 - Tất cả các truy xuất trên stack có thể dựa trên `%rsp`
 - Cấp phát bằng cách giảm giá trị stack pointer
- **Thu hồi dễ dàng**
 - Tăng giá trị của stack pointer
 - Không cần đến base/frame pointer

x86-64 Procedure: Tổng kết

- **Sử dụng nhiều thanh ghi**
 - Truyền tham số
 - Có nhiều thanh ghi nên có thể lưu nhiều biến tạm hơn
- **Hạn chế sử dụng stack**
 - Có khi không sử dụng
 - Cấp phát/thu hồi nguyên stack frame

Nội dung

- **Thủ tục (Procedures)**
 - Cấu trúc stack
 - Gọi hàm trong IA32
 - Chuyển luồng
 - Truyền dữ liệu
 - Quản lý dữ liệu cục bộ
 - Gọi hàm trong x86-64
 - Minh hoạ hàm đệ quy (tự tìm hiểu)
- Bài tập về hàm
- Dịch ngược – Reverse engineering

Hàm đệ quy

```
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}
```

■ Các thanh ghi

- **%eax, %edx** sử dụng mà không cần lưu lại trước
- **%ebx** sử dụng nhưng cần lưu lại lúc đầu và khôi phục lúc kết thúc

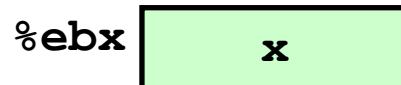
```
pcount_r:
    pushl %ebp
    movl %esp, %ebp
    pushl %ebx
    subl $4, %esp
    movl 8(%ebp), %ebx
    movl $0, %eax
    testl %ebx, %ebx
    je .L3
    movl %ebx, %eax
    shrl %eax
    movl %eax, (%esp)
    call pcount_r
    movl %ebx, %edx
    andl $1, %edx
    leal (%edx,%eax), %eax
.L3:
    addl $4, %esp
    popl %ebx
    popl %ebp
    ret
```

Hàm đệ quy #1

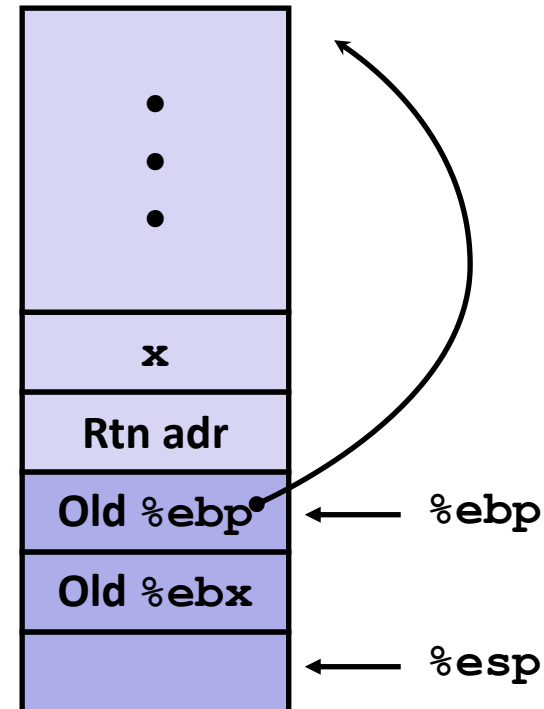
```
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}
```

■ Actions

- Lưu giá trị cũ của **%ebx** trên stack
- Cấp phát không gian cho các tham số của hàm đệ quy
- Lưu **x** tại **%ebx**



```
pcount_r:
    pushl %ebp
    movl  %esp, %ebp
    pushl %ebx
    subl  $4, %esp
    movl  8(%ebp), %ebx
    . . .
```



Hàm đệ quy #2

```
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}
```

```
    . . .
movl  $0, %eax
testl %ebx, %ebx
je    .L3
    . . .
.L3:
    . . .
ret
```

■ Actions

- Nếu $x == 0$, Trả về
 - Gán `%eax` bằng 0

`%ebx`

`x`

Hàm đệ quy #3

```
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}
```

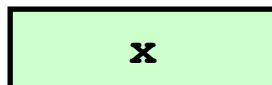
■ Actions

- Lưu $x \gg 1$ vào stack
- Gọi hàm đệ quy

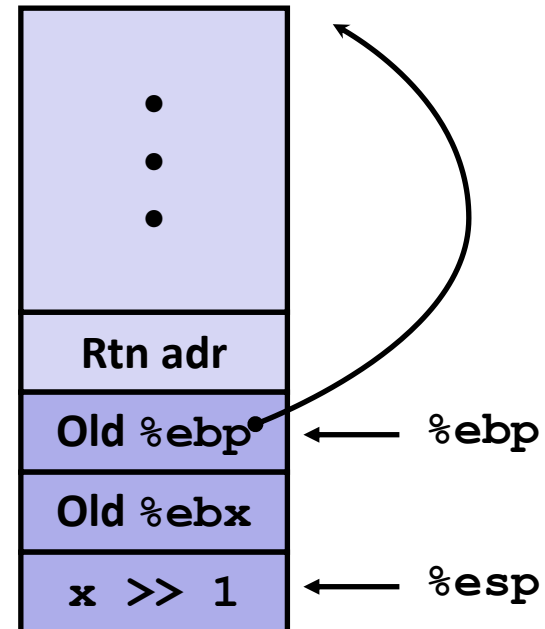
■ Tác động

- `%eax` được gán là giá trị trả về
- `%ebx` vẫn giữ giá trị của x

`%ebx`



```
• • •
movl    %ebx, %eax
shrl    %eax
movl    %eax, (%esp)
call    pcount_r
• • •
```



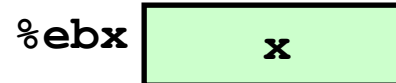
Hàm đệ quy #4

```
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}
```

```
• • •
movl    %ebx, %edx
andl    $1, %edx
leal    (%edx,%eax), %eax
• • •
```

■ Giả sử

- `%eax` giữ giá trị trả về của hàm đệ quy
- `%ebx` giữ `x`



■ Actions

- Tính $(x \& 1) +$ giá trị đã tính được

■ Ảnh hưởng

- `%eax` được gán bằng kết quả của hàm

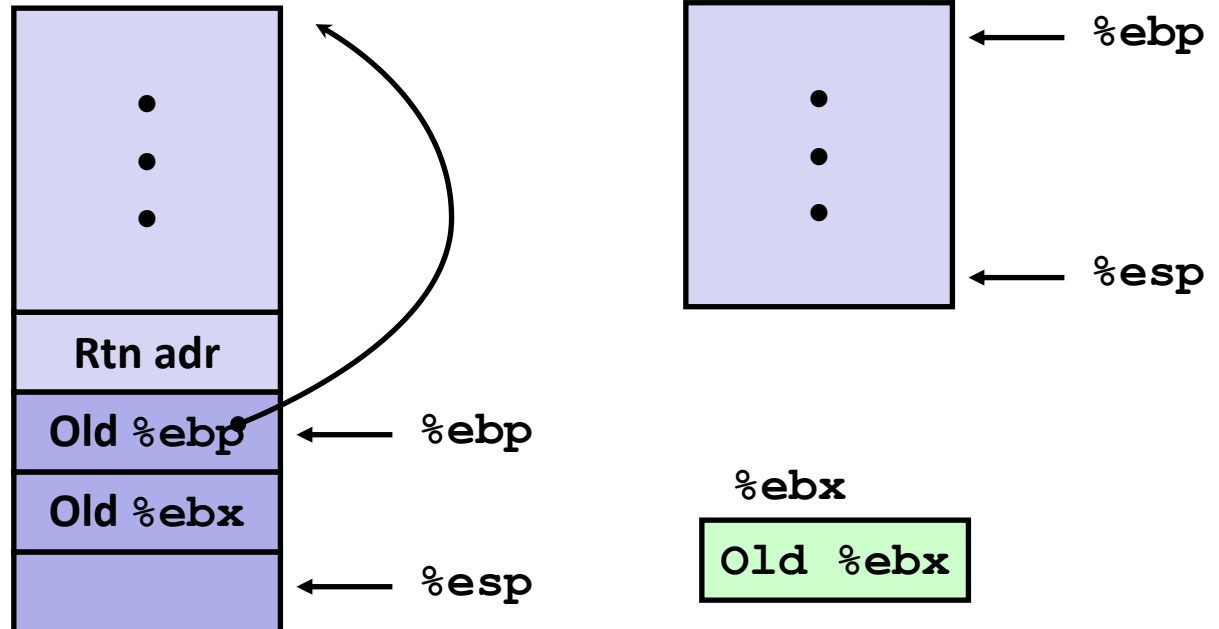
Hàm đệ quy #5

```
/* Recursive popcount */  
int pcount_r(unsigned x) {  
    if (x == 0)  
        return 0;  
    else return  
        (x & 1) + pcount_r(x >> 1);  
}
```

• • •
L3:
addl\$4, %esp
popl%ebx
popl%ebp
ret

■ Actions

- Khôi phục giá trị của **%ebx** và **%ebp**
- Khôi phục **%esp**



Hàm đệ quy (x86-64)

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

```
pcount_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    rep; ret
```

Hàm đệ quy (x86-64) – Trường hợp kết thúc

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

pcount_r:

```
movl    $0, %eax
testq   %rdi, %rdi
je      .L6
pushq   %rbx
movq    %rdi, %rbx
andl    $1, %ebx
shrq    %rdi
call    pcount_r
addq    %rbx, %rax
popq    %rbx
```

.L6:

```
rep; ret
```

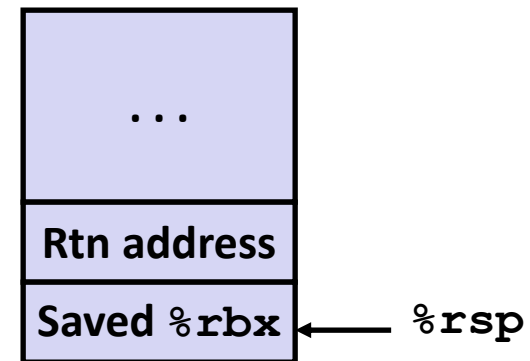
Register	Use(s)	Type
%rdi	x	Argument
%rax	Return value	Return value

Hàm đệ quy (x86-64) – Lưu thanh ghi

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

```
pcount_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    rep; ret
```

Register	Use(s)	Type
%rdi	x	Argument



Hàm đệ quy (x86-64) – Chuẩn bị gọi hàm

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

```
pcount_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    rep; ret
```

Register	Use(s)	Type
%rdi	x >> 1	Rec. argument
%rbx	x & 1	Callee-saved

Hàm đệ quy (x86-64) – Gọi hàm

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

```
pcount_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    rep; ret
```

Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Recursive call return value	

Hàm đệ quy (x86-64) – Kết quả hàm

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

```
pcount_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    rep; ret
```

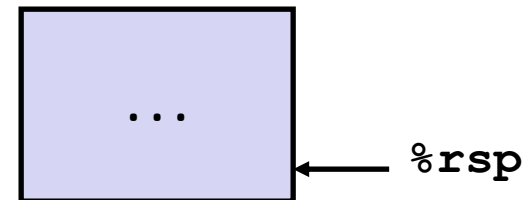
Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Return value	

Hàm đệ quy (x86-64) – Hoàn thành

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

```
pcount_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    rep; ret
```

Register	Use(s)	Type
%rax	Return value	Return value



Nội dung

- Thủ tục (Procedures)
 - Cấu trúc stack
 - Gọi hàm trong IA32
 - Chuyển luồng
 - Truyền dữ liệu
 - Quản lý dữ liệu cục bộ
 - Gọi hàm trong x86-64
 - Minh hoạ hàm đệ quy (tự tìm hiểu)
- Bài tập về hàm
- Dịch ngược – Reverse engineering

Nội dung

■ Thủ tục (Procedures)

- Cấu trúc stack
- Gọi hàm trong IA32
 - Chuyển luồng
 - Truyền dữ liệu
 - Quản lý dữ liệu cục bộ
- Gọi hàm trong x86-64
- Minh họa hàm đệ quy (tự tìm hiểu)
- Dịch ngược – Reverse engineering

Nội dung

■ Các chủ đề chính:

- 1) Biểu diễn các kiểu dữ liệu và các phép tính toán bit
- 2) Ngôn ngữ assembly
- 3) Điều khiển luồng trong C với assembly
- 4) Các thủ tục/hàm (procedure) trong C ở mức assembly
- 5) Biểu diễn mảng, cấu trúc dữ liệu trong C
- 6) Một số topic ATTT: reverse engineering, bufferoverflow
- 7) Phân cấp bộ nhớ, cache
- 8) Linking trong biên dịch file thực thi

■ Lab liên quan

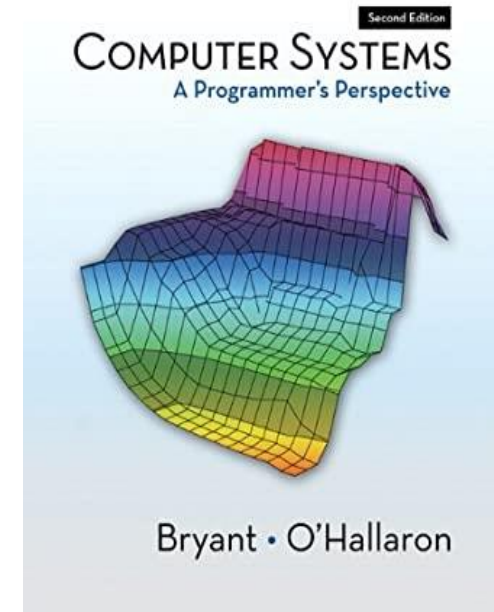
- | | |
|---|---|
| ▪ Lab 1: Nội dung <u>1</u> | ▪ Lab 4: Nội dung 1, <u>2</u> , 3, <u>4</u> , 5, <u>6</u> |
| ▪ Lab 2: Nội dung 1, <u>2</u> , <u>3</u> | ▪ Lab 5: Nội dung 1, <u>2</u> , 3, <u>4</u> , 5, <u>6</u> |
| ▪ Lab 3: Nội dung 1, <u>2</u> , <u>3</u> , <u>4</u> , <u>5</u> , <u>6</u> | ▪ Lab 6: Nội dung <u>1</u> , <u>2</u> , <u>3</u> , <u>4</u> , <u>5</u> , <u>6</u> |

Giáo trình

■ Giáo trình chính

Computer Systems: A Programmer's Perspective

- Second Edition (CS:APP2e), Pearson, 2010
- Randal E. Bryant, David R. O'Hallaron
- <http://csapp.cs.cmu.edu>



■ Tài liệu khác

- *The C Programming Language*, Second Edition, Prentice Hall, 1988
 - Brian Kernighan and Dennis Ritchie
- *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*, 1st Edition, 2008
 - Chris Eagle
- *Reversing: Secrets of Reverse Engineering*, 1st Edition, 2011
 - Eldad Eilam



**KEEP
CALM
AND
ENJOY YOUR
SEMESTER :)**