

# 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
- Bài tập về hàm
- Dịch ngược – Reverse engineering

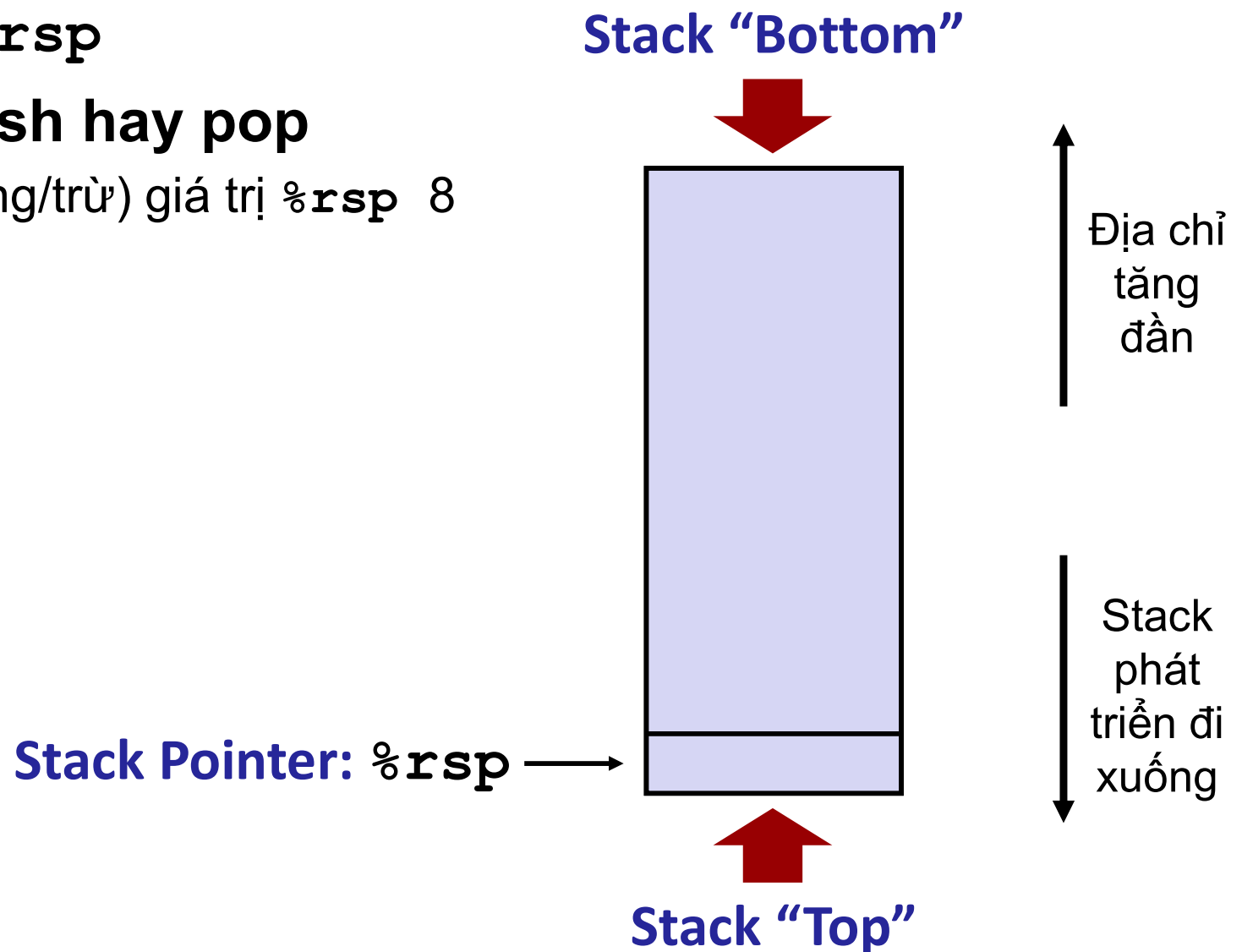
# Điểm chung của hàm trong IA32 và x86-64

---

- 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

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

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

- 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

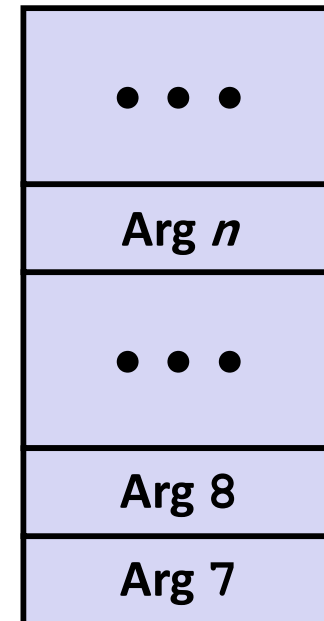
### ■ 6 tham số đầu tiên

<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

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

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

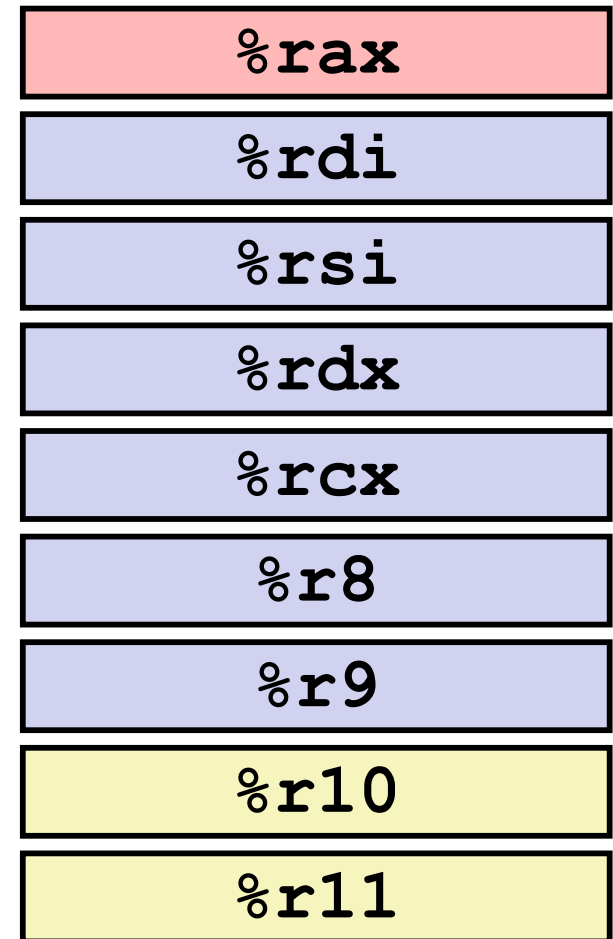
## ■ **%r10, %r11**

- 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 Temporaries

Callee-saved

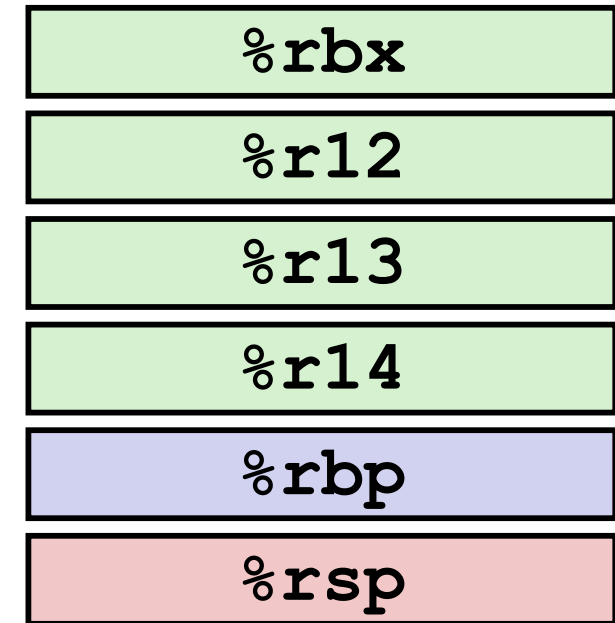
## ■ **%rbp**

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

Special

## ■ **%rsp**

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



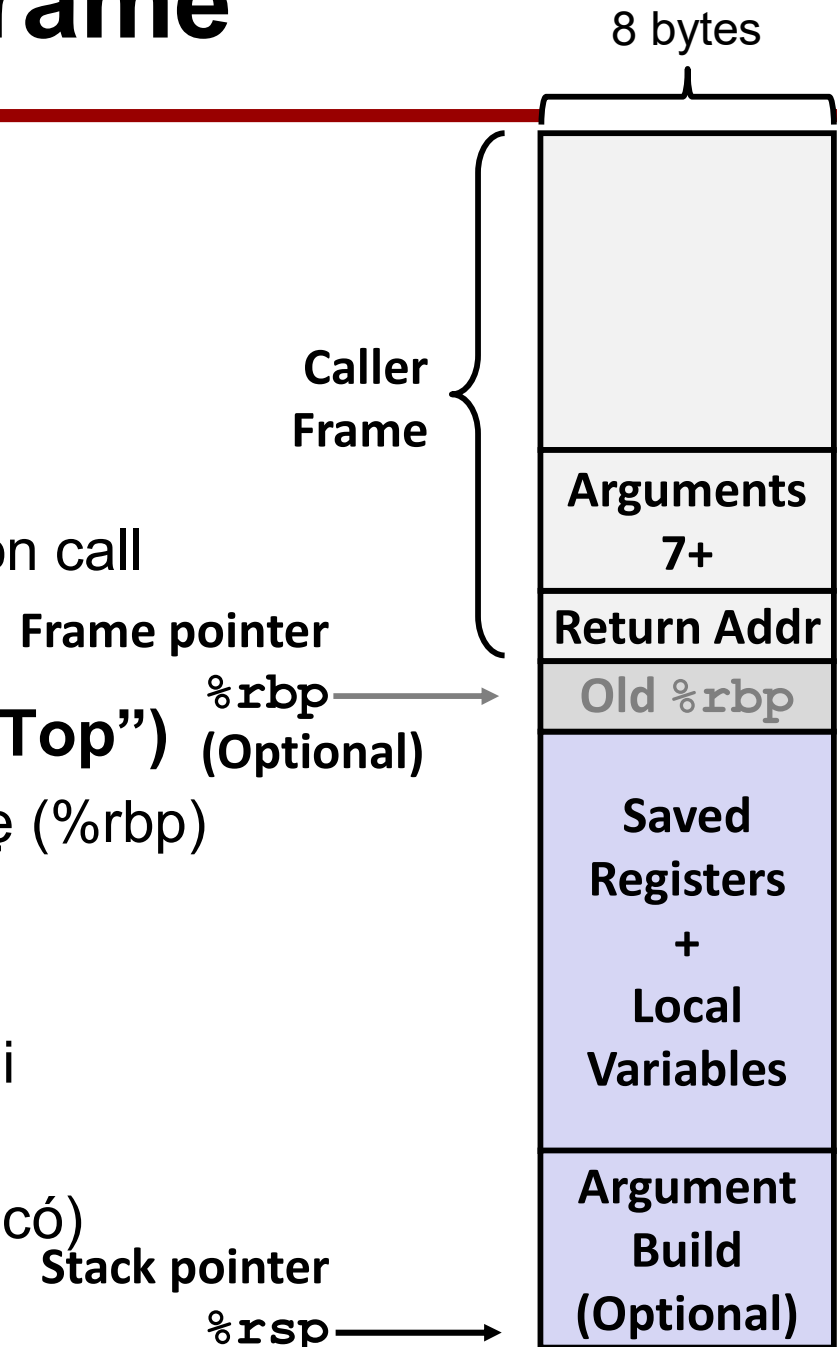
# x86-64/Linux Stack Frame

## ■ 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

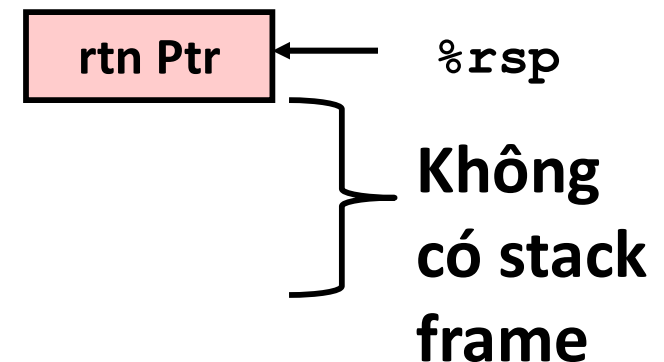
- Tham số 1 (**xp**) trong **%rdi**, Tham số 2 (**yp**) trong **%rsi**

- 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</code> , <code>y</code>
%rax	<code>x</code> , Return value

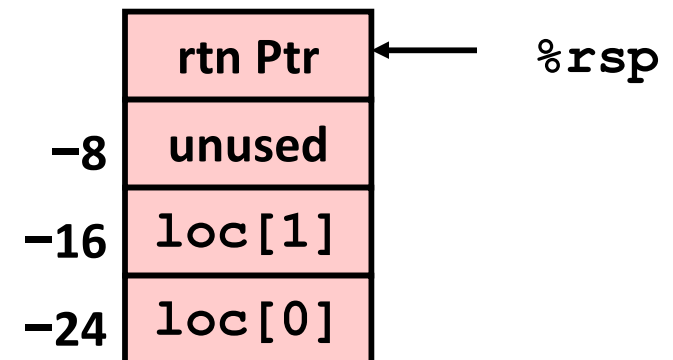
# Biến cục bộ trong hàm x86\_64 – VD 1

```
/* 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)

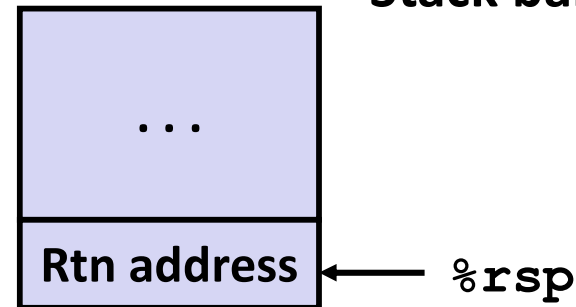
- Có thể lưu tất cả thông tin trong vùng nhớ nhỏ gần stack pointer



# Biến cục bộ trong hàm x86\_64 – VD 2 #1

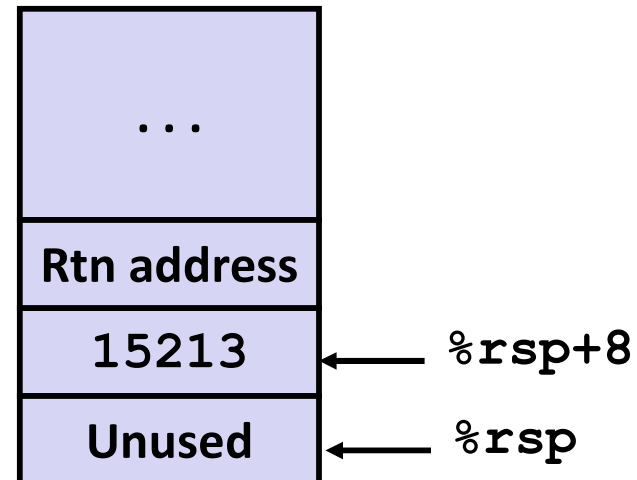
```
long call_incr() {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return v1+v2;  
}
```

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
```

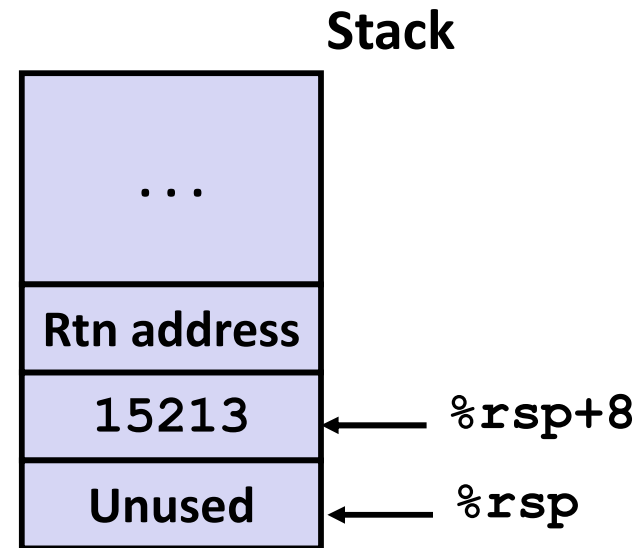
Stack sau khi thay đổi



# 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
```

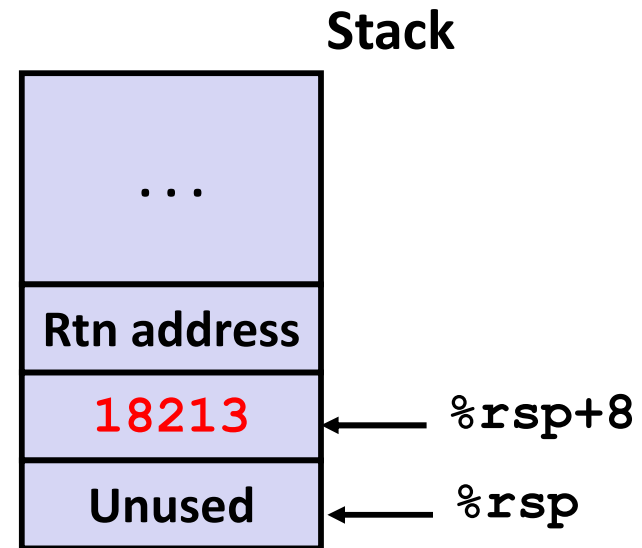


Register	Use(s)
%rdi	&v1
%rsi	3000

# 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
```



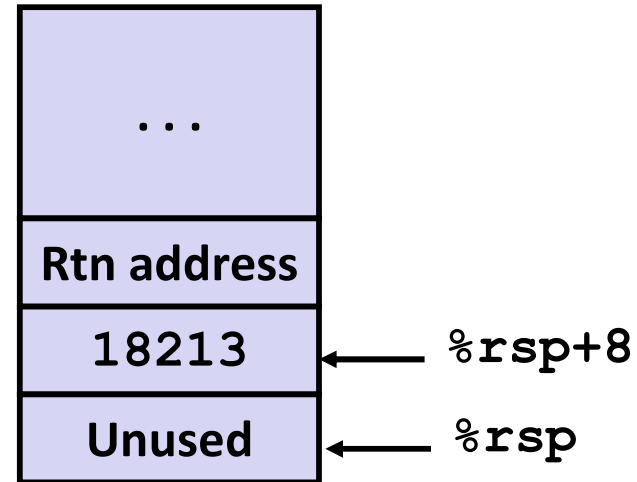
Register	Use(s)
%rdi	&v1
%rsi	3000



# Biến cục bộ trong hàm x86\_64 – VD2 #3

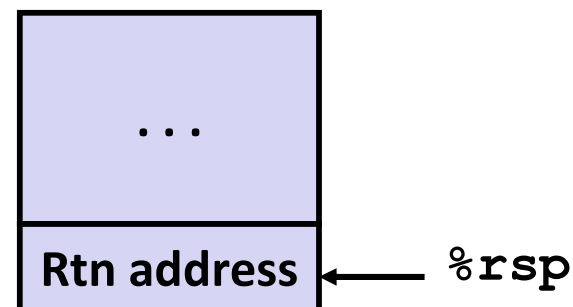
```
long call_incr() {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return v1+v2;  
}
```

```
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)
<code>%rax</code>	Return value

Stack sau khi cập nhật `%rsp`

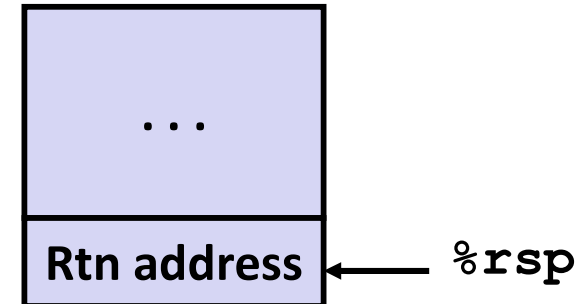


# Biến cục bộ trong hàm x86\_64 – VD2 #4

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

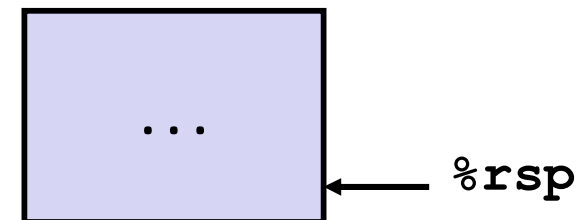
```
call_incr:  
    subq    $16, %rsp  
    movq    $15213, 8(%rsp)  
    movl    $3000, %esi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

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)

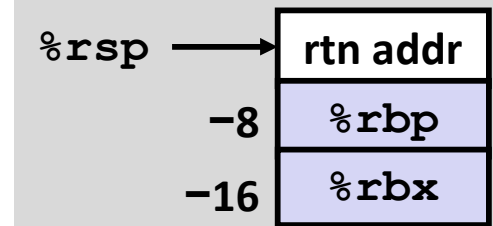
swap\_ele\_su:

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

# Hiểu x86-64 Stack Frame (2)

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

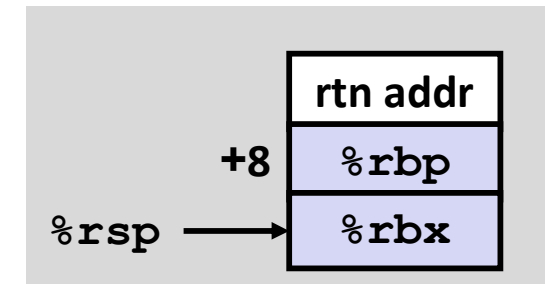
```
# Save %rbx
# Save %rbp
```



```
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 họa 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

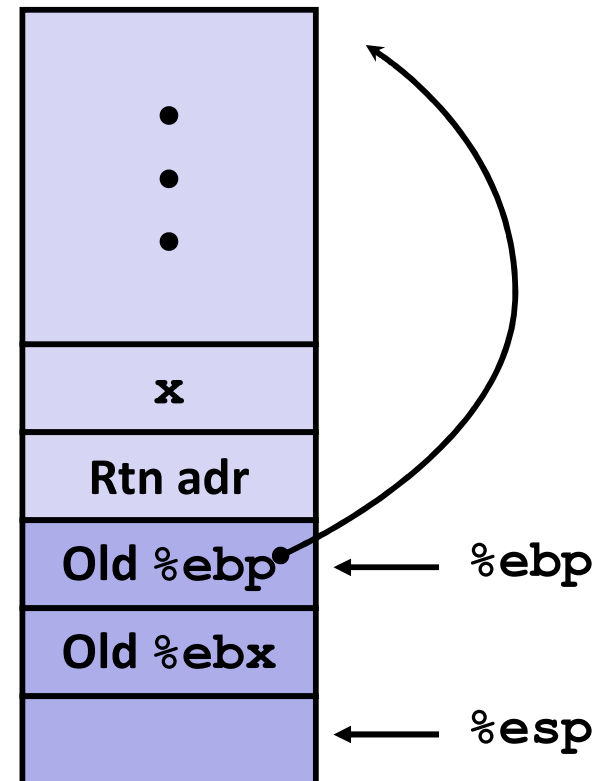
```
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}
```

## ■ 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

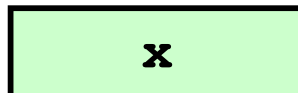
```
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}
```

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

## ■ Actions

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

`%ebx`



# 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

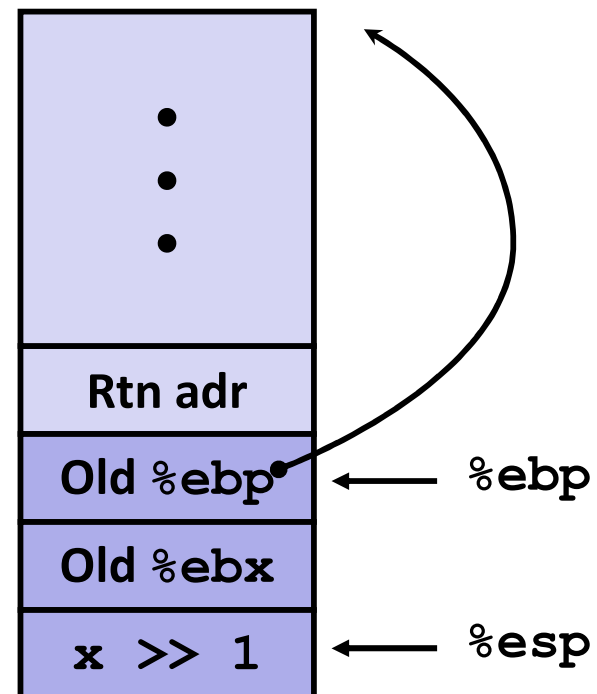
## ■ Tác động

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

`%ebx` x

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

```



# Hàm đệ quy #4

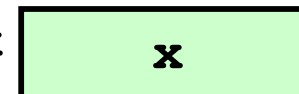
```
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}
```

```
• • •
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`

`%ebx`



## ■ 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);
}
```

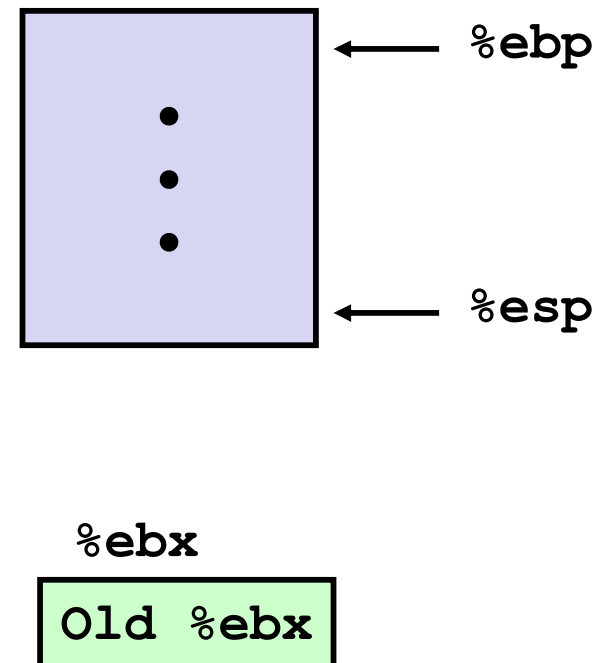
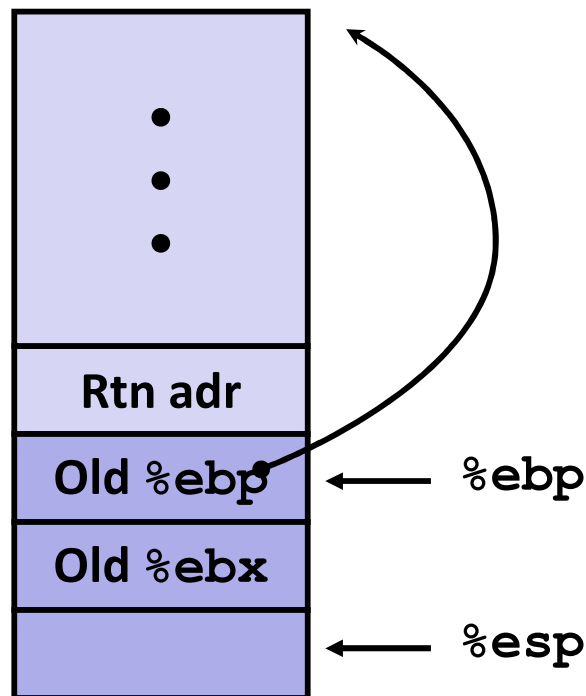
• • •

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);
}
```

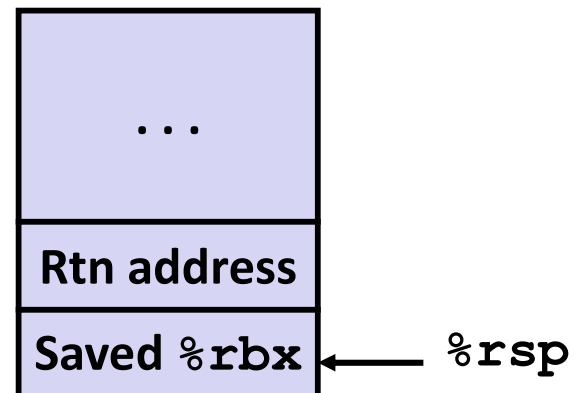
pcount\_r:

```
    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
```

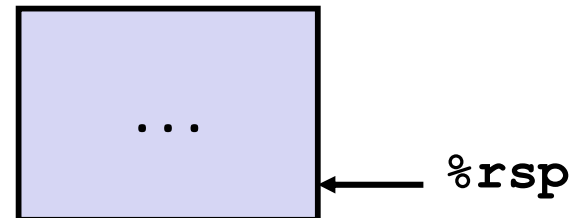
Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Return value	

# 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 họa hàm đệ quy (tự tìm hiểu)
- Bài tập về hàm
- Dịch ngược – Reverse engineering

# Procedure (IA32) – Bài tập 1

Code C

Code assembly

```
1.  push    %ebp
2.  mov     %esp, %ebp
3.  sub     $0x40, %esp
4.  movl    $0x04030201, 0x3c(%esp)
5.  movl    $0x0, 0x38(%esp)
6.  mov     0x0804a02c, %eax
7.  mov     %eax, 0x8(%esp)
8.  movl    $0x32, 0x4(%esp)
9.  lea     0x10(%esp), %eax
10. mov     %eax, (%esp)
11. call    fgets
12. lea     0x10(%esp), %eax
13. mov     %eax, 0x4(%esp)
14. movl    $0x8048610, (%esp)
15. call    printf
16. ...
```

```
int main()
{
    int check = 0x04030201;
    int var = 0x00000000;
    char buf[40];
    fgets(buf, 50, stdin);
    printf("\n[buf]: %s\n", buf);
}
```

*Điền vào bảng sau:*

Hàm	Đoạn code assembly?	Các dòng truyền tham số và giá trị
fgets		
printf		

**Biết: Tại các ô nhớ**

**0x0804a02c** chứa **stdin**

**0x08048610** chứa chuỗi **"\n[buf]: %s\n"**

**(%esp + 16)** là vị trí chuỗi buf

# Procedure (IA32) – Bài tập 2

## Code assembly

```
main:
1.  push    %ebp
2.  mov     %esp, %ebp
3.  sub     $0x40, %esp
4.  movl    $0x04030201, 0x3c(%esp)
5.  movl    $0x0, 0x38(%esp)
6.  mov     0x0804a02c, %eax
7.  mov     %eax, 0x8(%esp)
8.  movl    $0x32, 0x4(%esp)
9.  lea     0x10(%esp), %eax
10. mov     %eax, (%esp)
11. call    fgets
12. lea     0x10(%esp), %eax
13. mov     %eax, 0x4(%esp)
14. movl    $0x8048610, (%esp)
15. call    printf
16. ...
```

Vẽ stack của hàm main sau khi thực thi xong lệnh `call fgets` trong IA32?

Giả sử ban đầu:

`%ebp = 0x800148`                      `%esp = 0x800130`

**Ô nhớ địa chỉ 0x0804a02c chứa stdin**





# Procedure (IA32) – Bài tập 3a

## Code assembly

```
1  proc:
2  pushl    %ebp
3  movl     %esp, %ebp
4  subl     $40, %esp
5  leal     -4(%ebp), %eax
6  movl     %eax, 8(%esp)
7  leal     -8(%ebp), %eax
8  movl     %eax, 4(%esp)
9  movl     $.LC0, (%esp) Pointer
10 call     scanf
    Diagram stack frame at this point
11 movl     -4(%ebp), %eax
12 subl     -8(%ebp), %eax
13 leave
14 ret
```

## Code C

```
1  int proc(void)
2  {
3      int x,y;
4      scanf("%x %x", &y, &x);
5      return x-y;
6  }
```

## Giả sử:

- Khi mới bắt đầu thực thi **proc** (dòng 1):

Register	Value
%esp	0x800040
%ebp	0x800060

- Giá trị của %ebp sau dòng lệnh thứ 3 (có giải thích)?
- Giá trị của %esp sau dòng lệnh thứ 4 (có giải thích)?

# Procedure (IA32) – Bài tập 3b

## Code assembly

```
1  proc:
2  pushl    %ebp
3  movl     %esp, %ebp
4  subl     $40, %esp
5  leal     -4(%ebp), %eax
6  movl     %eax, 8(%esp)
7  leal     -8(%ebp), %eax
8  movl     %eax, 4(%esp)
9  movl     $.LC0, (%esp)
10 call     scanf
    Diagram stack frame at this point
11 movl     -4(%ebp), %eax
12 subl     -8(%ebp), %eax
13 leave
14 ret
```

## Code C

```
1  int proc(void)
2  {
3      int x,y;
4      scanf("%x %x", &y, &x);
5      return x-y;
6  }
```

## Giả sử:

- Khi mới bắt đầu thực thi **proc** (dòng 1):

Register	Value
%esp	0x800040
%ebp	0x800060

c. Đoạn code truyền tham số và gọi scanf? Giải thích?

d. Xác định vị trí lưu của x và y? Giải thích?

# Procedure (IA32) – Bài tập 3c

## Code assembly

```
1  proc:
2  pushl    %ebp
3  movl     %esp, %ebp
4  subl     $40, %esp
5  leal     -4(%ebp), %eax
6  movl     %eax, 8(%esp)
7  leal     -8(%ebp), %eax
8  movl     %eax, 4(%esp)
9  movl     $.LC0, (%esp)
10 call     scanf
    Diagram stack frame at this
11 movl     -4(%ebp), %eax
12 subl     -8(%ebp), %eax
13 leave
14 ret
```

## Code C

```
1  int proc(void)
2  {
3      int x,y;
4      scanf("%x %x", &y, &x);
5      return x-y;
6  }
```

## Giả sử:

- Khi mới bắt đầu thực thi **proc** (dòng 1):

%esp	0x800040
%ebp	0x800060

e. Vẽ stack sau khi thực hiện lệnh **call scanf**.

# Nội dung

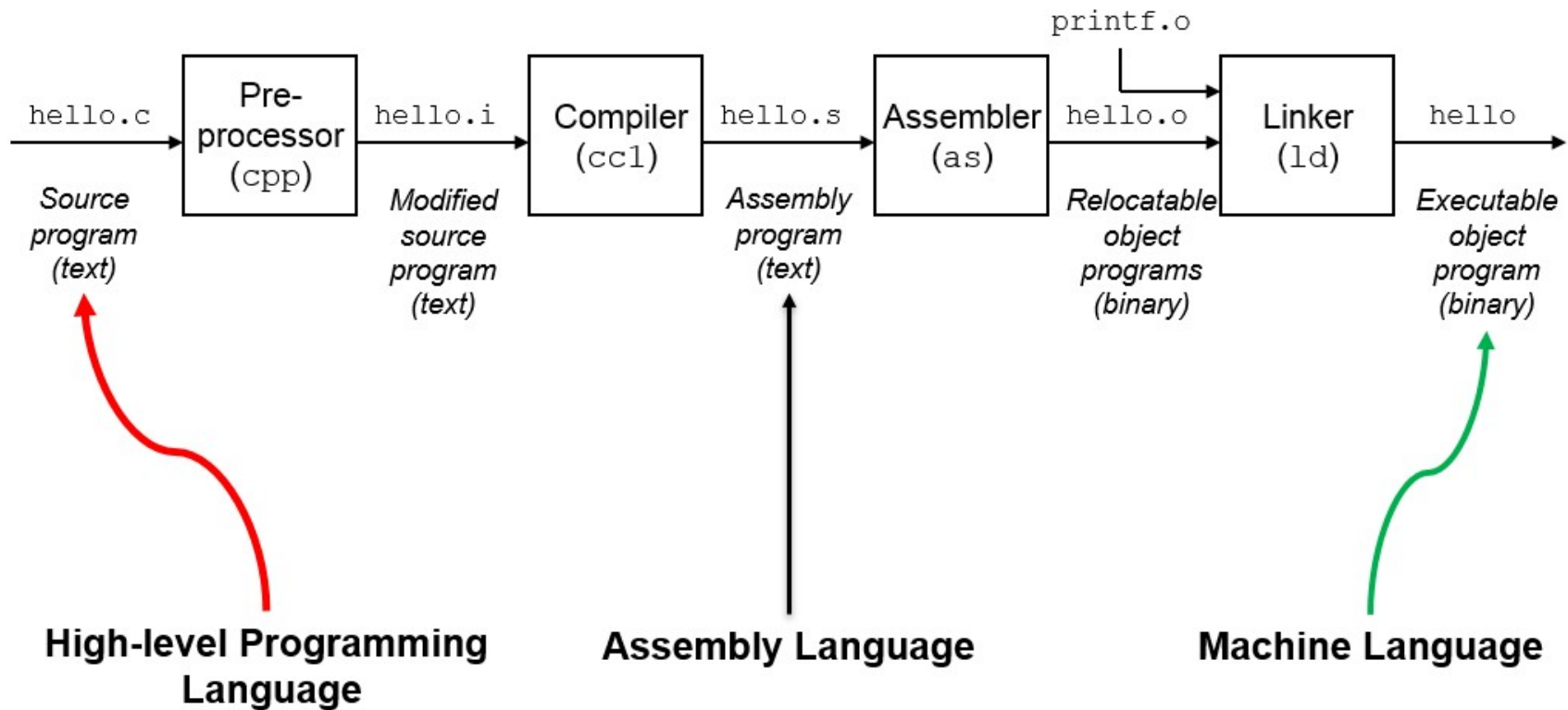
---

## ■ 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

# Dịch ngược - Reverse Engineering?

Biên dịch



File hợp ngữ

File thực thi

Dịch ngược

# 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.

```
8d 4c 24 04
83 e4 f0
ff 71 fc
55
89 e5
51
83 ec 14
65 a1 14 00 00 00
89 45 f4
31 c0
83 ec 0c
68 ec 8b 04 08
```



RE

```
lea    0x4(%esp),%ecx
and     $0xfffffffff0,%esp
pushl   -0x4(%ecx)
push    %ebp
mov     %esp,%ebp
push    %ecx
sub     $0x14,%esp
mov     %gs:0x14,%eax
mov     %eax,-0xc(%ebp)
xor     %eax,%eax
sub     $0xc,%esp
push    $0x8048bec
```



**File thực thi (binary)**

**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)

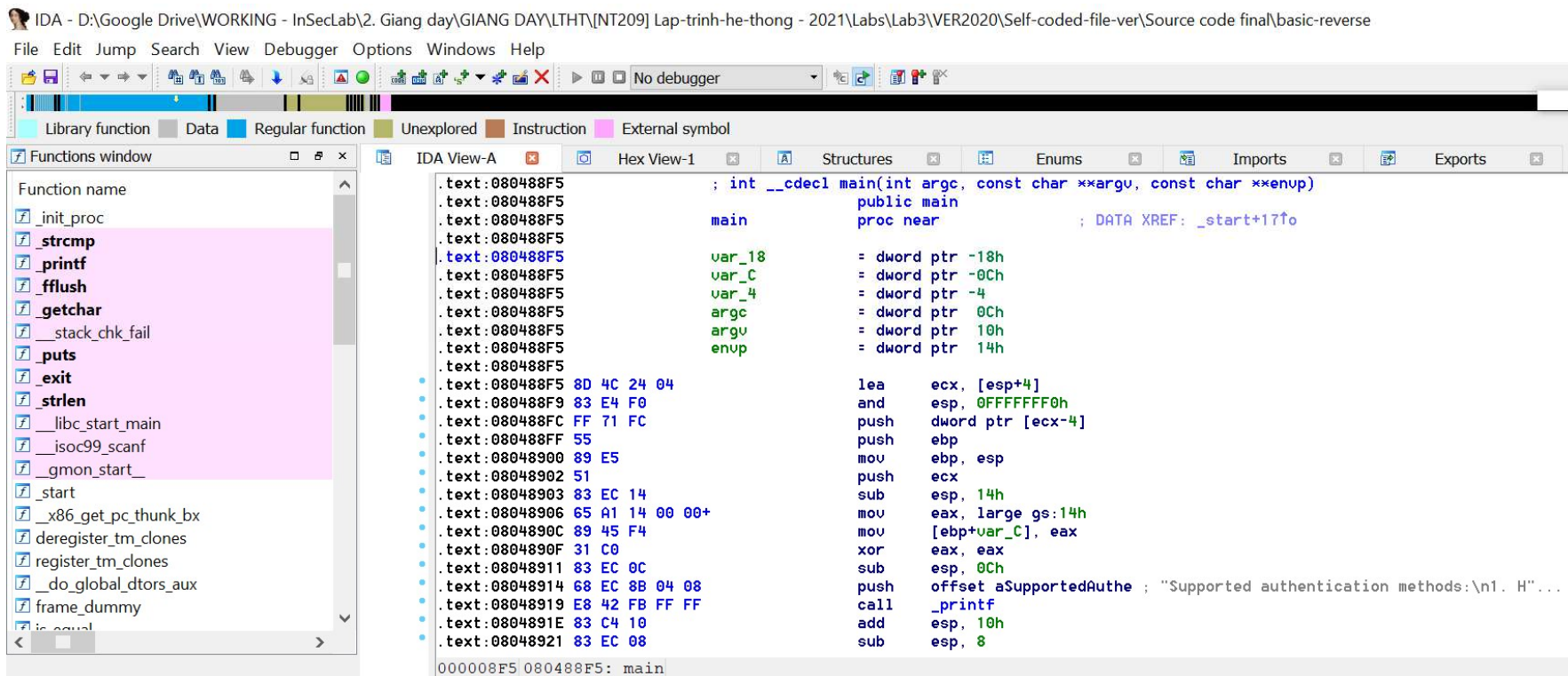
```
ubuntu@ubuntu: ~  
ubuntu@ubuntu:~$ gdb basic-reverse  
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1  
Copyright (C) 2016 Free Software Foundation, Inc.  
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>  
This is free software: you are free to change and redistribute it.  
There is NO WARRANTY, to the extent permitted by law.  
and "show warranty" for details.  
This GDB was configured as "x86_64-linux-gnu".  
Type "show configuration" for configuration.  
For bug reporting instructions, please see  
<http://www.gnu.org/software/gdb/bugs/><br>  
Find the GDB manual and other documentation  
<http://www.gnu.org/software/gdb/documentation/><br>  
For help, type "help".  
Type "apropos word" to search for commands.  
Reading symbols from basic-reverse...  
(gdb) disassemble  
No frame selected.  
(gdb) run  
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
- Cần chạy chương trình để xem mã assembly



# Dịch ngược – Công cụ (3)

## ■ IDA Pro



- Có giao diện, nhiều cửa sổ cung cấp nhiều thông tin
- Có thể chạy trên Windows
- Định dạng assembly: **Intel**
- Có thể phân tích code ở dạng tĩnh (không cần chạy chương trình) và động (thực thi chương trình)

# 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.

# Assignment: Reverse engineering 1

- Cho 2 file thực thi **re1.bin** (Linux) và **re2.exe** (Windows)
- **Tùy chọn công cụ hỗ trợ dịch ngược, hãy phân tích và tìm ra password cần nhập của chương trình.**
  - Khuyến khích phân tích bằng assembly
  - (Nếu dùng IDA Pro) Có thể sử dụng mã giả để hỗ trợ việc phân tích
- **Yêu cầu:**
  - Làm cá nhân.
  - Nộp file báo cáo (**.pdf**) trình bày các bước phân tích, password tìm thấy.
  - Nộp trên moodle (courses.uit.edu.vn).
  - Deadline: **2 tuần**

# 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

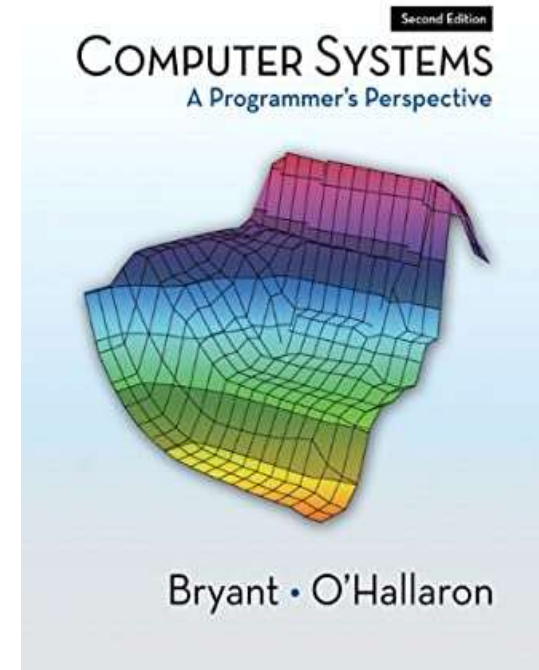
- |   |   |
|---|---|
| ▪ Lab 1: Nội dung <u>1</u>  | ▪ Lab 4: Nội dung 1, <u>2</u> , 3, <u>4</u> , 5, <u>6</u>                         |
| ▪ Lab 2: Nội dung 1, <u>2</u> , <u>3</u>                                  | ▪ Lab 5: Nội dung 1, <u>2</u> , 3, <u>4</u> , 5, <u>6</u>                         |
|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