LẬP TRÌNH HỆ THỐNG

ThS. Đỗ Thị Thu Hiền
(hiendtt@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) (tt)

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**

Nhắc lại: Gọi hàm trong IA32

- **Lệnh assembly dùng để gọi hàm?**

  `call` <*label*>

Nhắc lại: Gọi hàm trong IA32

- **Thực thi lệnh `call` có ảnh hưởng đến stack không?**

A. Có – Tăng kích thước stack

B. Có – Giảm kích thước stack

C. Không ảnh hưởng

D. Em quên rồi ☹

Nhắc lại: Gọi hàm trong IA32

- **1. Dữ liệu gì được đưa vào stack khi thực thi lệnh `call`?**
- **2. Kích thước bao nhiêu bytes?**

1. Địa chỉ trả về (return address)
2. 4 bytes

Nhắc lại: Gọi hàm trong IA32

- **Không gian stack của hàm được định nghĩa bởi thanh ghi nào?** *(nếu có nhiều thanh ghi thì phân cách bằng dấu ,)*

  `%ebp, %esp`

Nhắc lại: Gọi hàm trong IA32

- **Tham số thứ 1 và thứ 2 cho 1 hàm có thể lấy ở các vị trí địa chỉ nào?**

```
8(%ebp), 12(%ebp)
```

Nhắc lại: Gọi hàm trong IA32

- **Giá trị trả về của 1 hàm (nếu có) được lưu ở đâu?**

  Thanh ghi **`%eax`**

Nhắc lại: Gọi hàm trong IA32

- **Tác vụ nào cần phải thực hiện trong mã của hàm con?**

  A. Chuẩn bị tham số cần thiết để hoạt động

  B. Lưu lại `%ebp` của hàm mẹ

  C. Lưu địa chỉ trả về vào thanh ghi %eax

  D. Cả B và C đều đúng

Nhắc lại: Gọi hàm trong IA32

- **Các biến cục bộ (nếu có) của hàm sẽ nằm ở vị trí như thế nào so với %ebp của hàm?**

  A. Các ô nhớ có địa chỉ cao hơn so với vị trí %ebp trỏ đến

  B. Các ô nhớ có địa chỉ thấp hơn so với vị trí %ebp trỏ đến

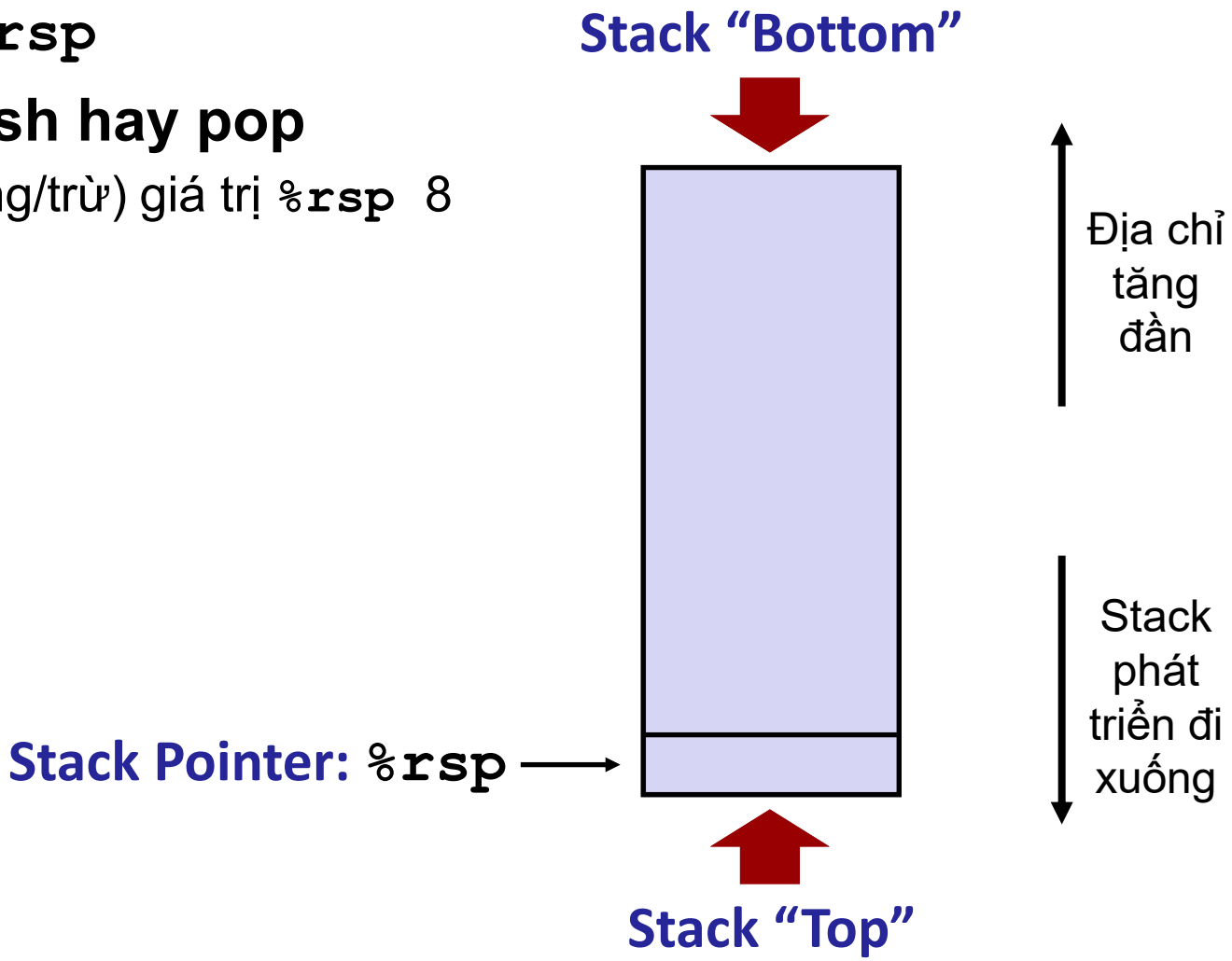
Điểm chung của hàm trong IA32 và x86-64

- Stack hỗ trợ việc gọi hàm

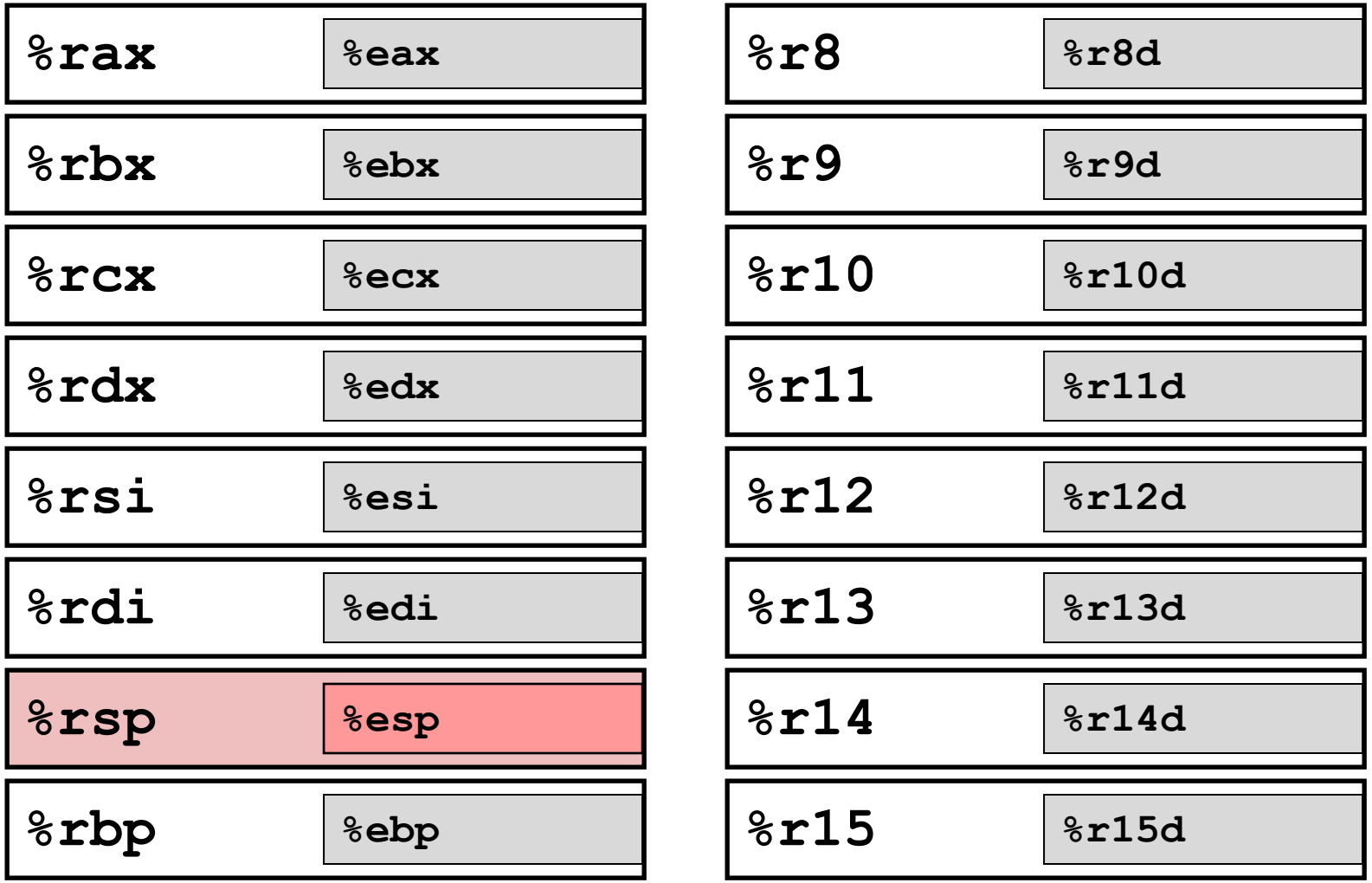
- Sử dụng lệnh **`call/ret`**
  - Địa chỉ trả về (return address) được đưa vào stack
    - Địa chỉ câu lệnh assembly ngay sau lệnh **`call`**

x86-64 Stack?

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



Thanh ghi x86-64



- 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 có thể truyền 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

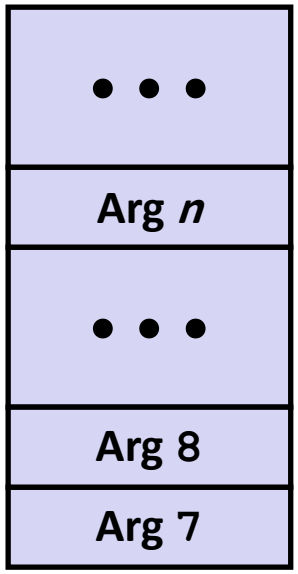
- Sử dụng các thanh ghi
- 6 tham số đầu tiên

`%rdi`	Tham số 1
`%rsi`	Tham số 2
`%rdx`	Tham số 3
`%rcx`	Tham số 4
`%r8`	Tham số 5
`%r9`	Tham số 6

- Giá trị trả về

`%rax`

- Stack



- Chỉ cấp phát không gian trong stack khi cần thiết

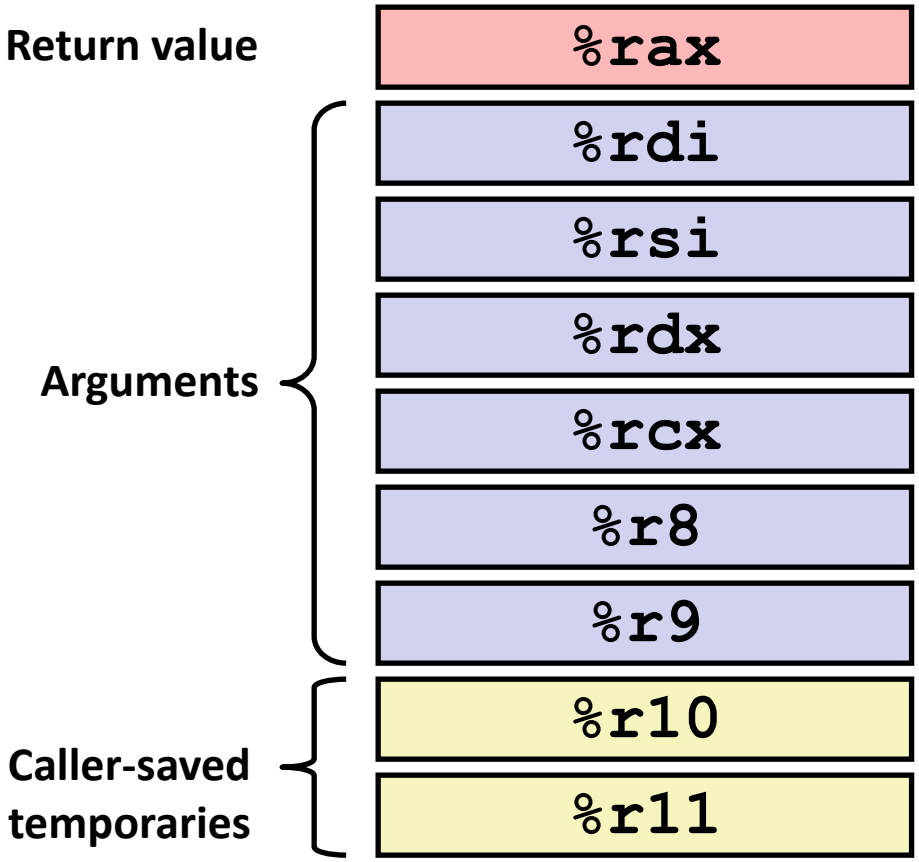
Thanh ghi x86-64: Quy ước sử dụng

%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

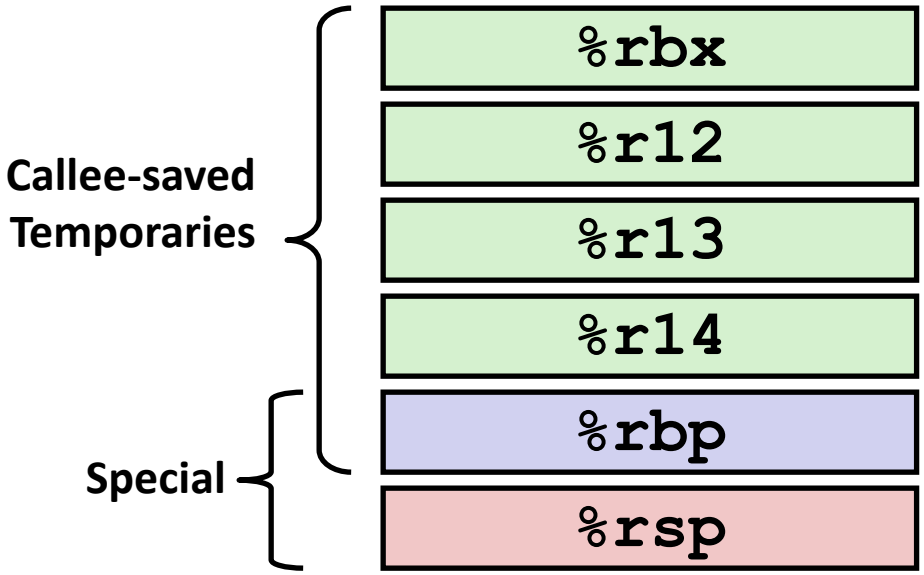
Sử dụng các thanh ghi x86-64 #1

- `%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



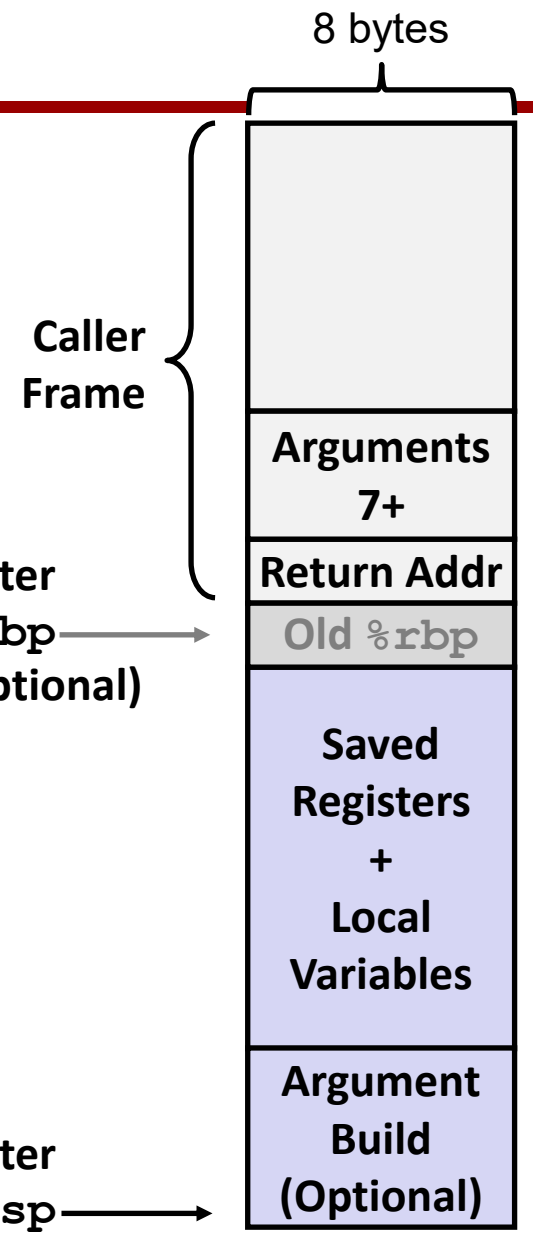
Sử dụng các thanh ghi x86-64 #2

- **`%rbx`, `%r12`, `%r13`, `%r14`**
  - Hàm con lưu lại (callee-saved)
  - Hàm con cần lưu và khôi phục lại
- **`%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



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)*** 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)

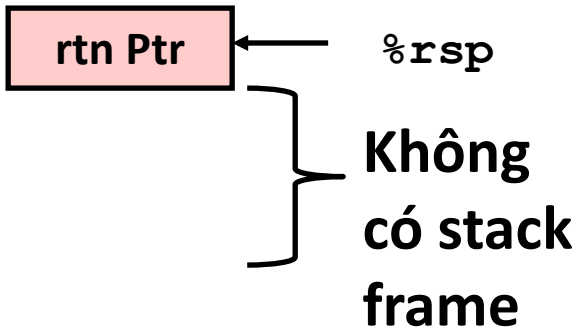


Ví dụ hàm trong x86-64: Long Swap

```
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
```

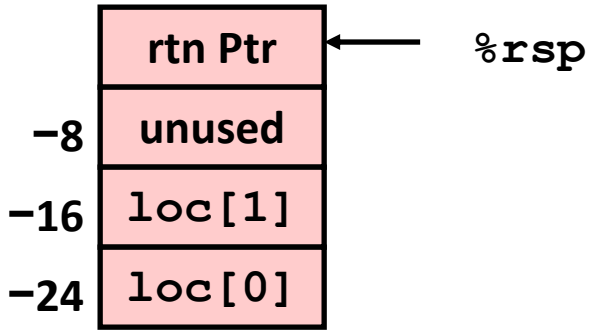
Register	Use(s)
`%rdi`	Argument `p`
`%rsi`	Argument `val`, `y`
`%rax`	`x`, Return value

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
```

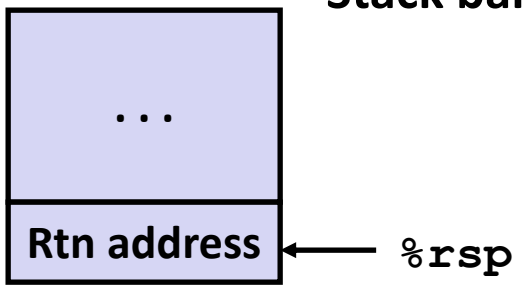
- **Hạn chế thay đổi stack pointer (`%rsp`)**
  - Có thể lưu tất cả thông tin trong vùng nhớ nhỏ gần stack pointer



Biến cục bộ trong hàm x86_64 – VD 2 #1

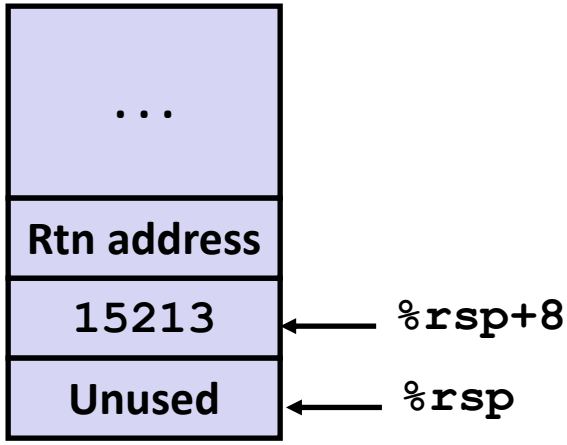
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

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
```

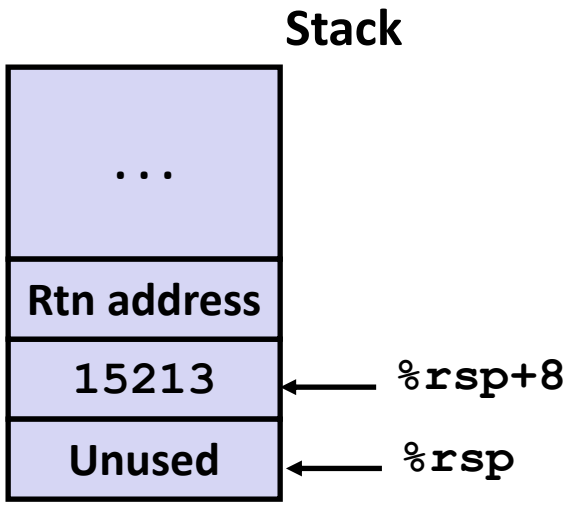
Stack sau khi thay đổi



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
```

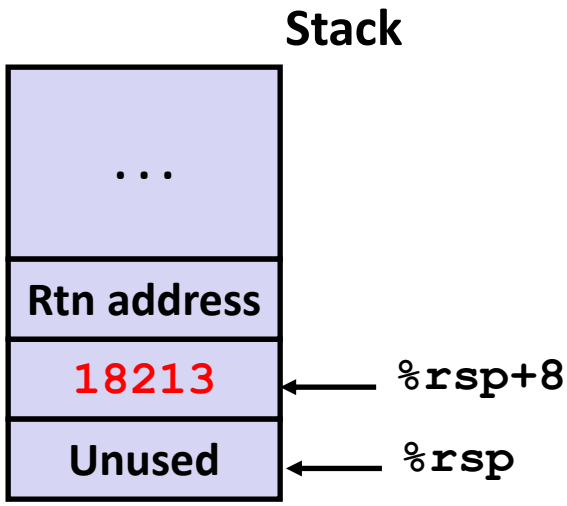


Register	Use(s)
%rdi	&v1
%rsi	3000

Biến cục bộ trong hàm x86_64 – VD2 #2

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

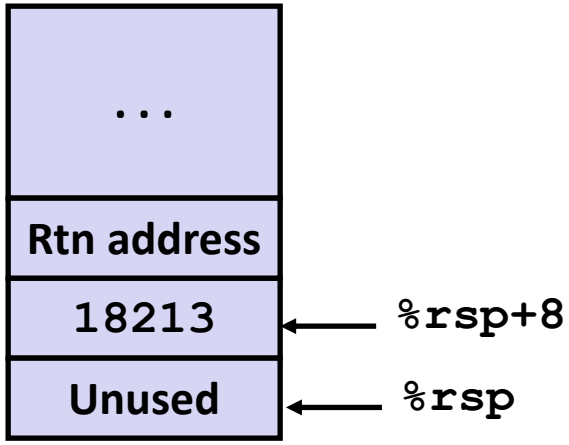
```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```



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

Biến cục bộ trong hàm x86_64 – VD2 #3

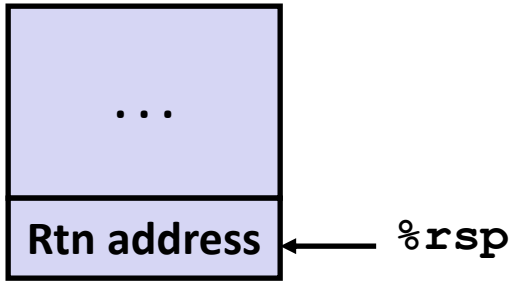
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```




```
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)
%rax	Return value

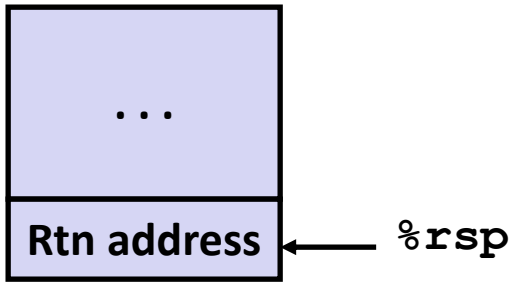
Stack sau khi cập nhật %rsp



Biến cục bộ trong hàm x86_64 – VD2 #4

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

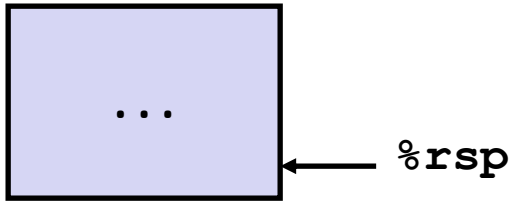
Stack sau khi cập nhật %rsp




```
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 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

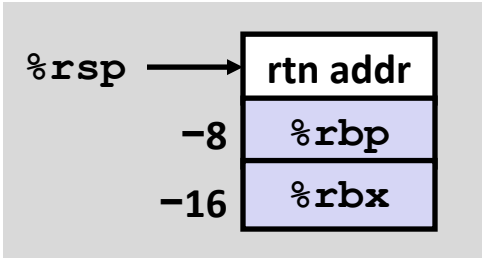
```
swap_ele_su:
    movq    %rbx, -16(%rsp)
    movq    %rbp, -8(%rsp)
    subq    $16, %rsp
    movslq  %esi,%rax
    leaq    8(%rdi,%rax,8), %rbx
    leaq    (%rdi,%rax,8), %rbp
    movq    %rbx, %rsi
    movq    %rbp, %rdi
    call    swap
    movq    (%rbx), %rax
    imulq   (%rbp), %rax
    addq    %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               # 2nd argument
   movq     %rbp, %rdi               # 1st 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
```

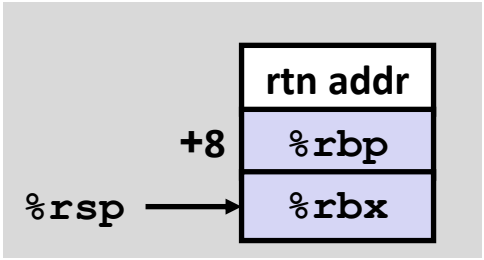
Hiểu x86-64 Stack Frame (2)

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




```
subq    $16, %rsp              # Allocate stack frame
```

● ● ●




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

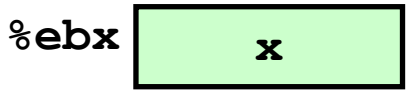
```
pcount_r:
    pushl  %ebp
    movl   %esp, %ebp
    pushl  %ebx
    subl   $4, %esp
    movl   8(%ebp), %ebx
    movl   $0, %eax
    testl  %ebx, %ebx
    je   .L3
    movl   %ebx, %eax
    shrl   %eax
    movl   %eax, (%esp)
    call   pcount_r
    movl   %ebx, %edx
    andl   $1, %edx
    leal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

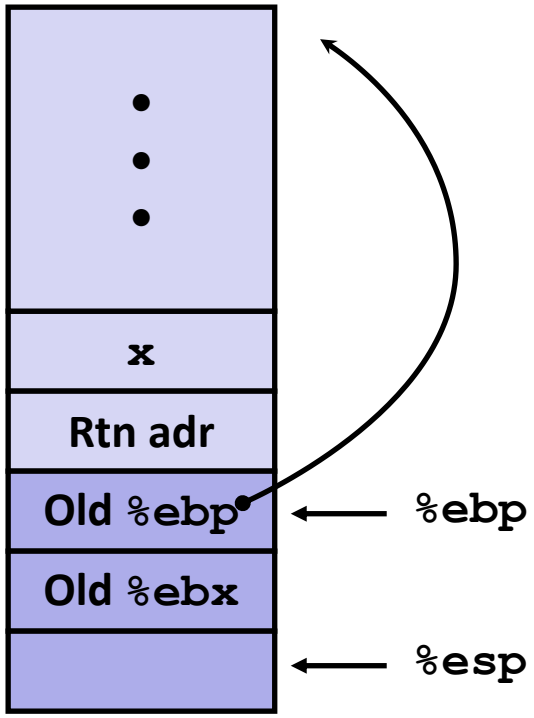
Hàm đệ quy #1

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

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

- **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**



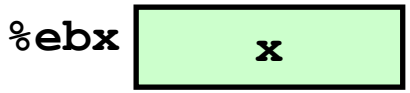


Hàm đệ quy #2

```
/* 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

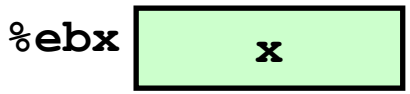


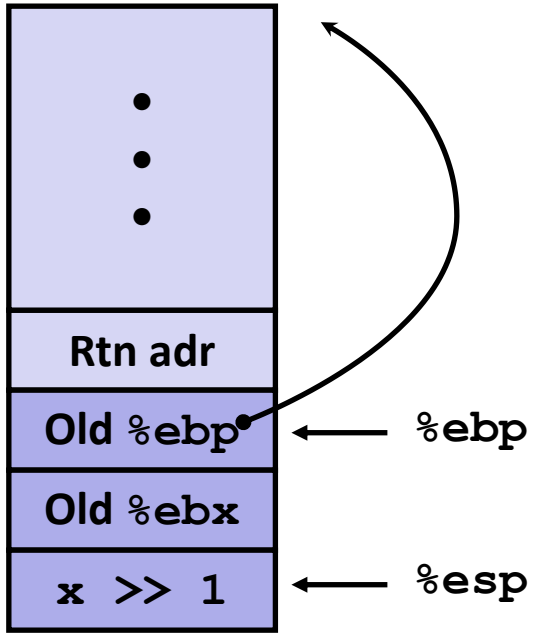
Hàm đệ quy #3

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

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

- **Actions**
  - Lưu x >> 1 vào stack
  - Gọi hàm đệ quy
- **Tác động**
  - **%eax** được gán là giá trị trả về
  - **%ebx** vẫn giữ giá trị của x



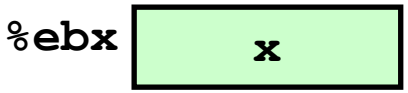


Hàm đệ quy #4

```
/* 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



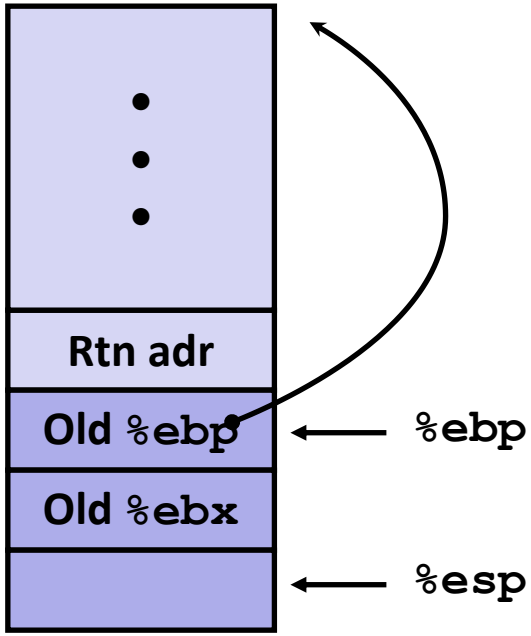
- **Actions**
  - Tính (x & 1) + giá trị đã tính được
- **Ảnh hưởng**
  - **%eax** được gán bằng kết quả của hàm

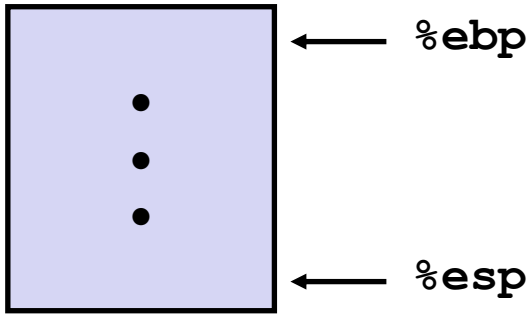
Hàm đệ quy #5

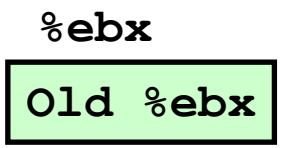
```
/* Recursive popcount */
int pcount_r(unsigned x) {
  if (x == 0)
    return 0;
  else return
    (x & 1) + pcount_r(x >> 1);
}
```

```
    • • •
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
```

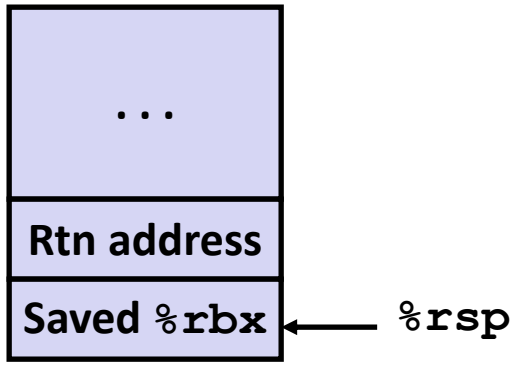
Register	Use(s)	Type
%rdi	x	Argument
%rax	Return value	Return value

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
```

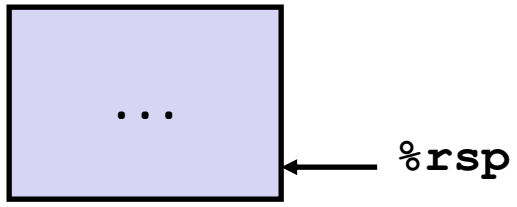
Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Return value	

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**

Procedure (IA32) – Bài tập 1.1

Code assembly

```
1.   .LC0: .string "%d"
2.   .LC1: .string "%d %d"
3.   example:
4.       pushl   %ebp
5.       movl    %esp, %ebp
6.       subl    $8, %esp
7.       movl    $5, -4(%ebp)
8.       movl    $10, -8(%ebp)
9.       leal    -8(%ebp), %eax
10.      pushl   %eax
11.      pushl   $.LC0
12.      call    scanf
13.      addl    $8, %esp
14.      movl    -8(%ebp), %eax
15.      pushl   -4(%ebp)
16.      pushl   %eax
17.      pushl   $.LC1
18.      call    printf
19.      addl    $12, %esp
20.      movl    $0, %eax
21.      leave
22.      ret
```

Code C?

- Có bao nhiêu **biến cục bộ?** Có giá trị bao nhiêu?

+ Tại -4(%ebp), có giá trị 5 → int a = 5
+ Tại -8(%ebp), có giá trị 10 → int b = 10

- Gọi hàm **scanf**

Dòng 9 - 12

+ Lần lượt push 2 giá trị vào stack: Địa chỉ của biến cục bộ thứ 2 và địa chỉ chuỗi .LC0 (“%d”)
+ **scanf(“%d”, &b);**

- Gọi hàm **printf**

Dòng 14 - 18

+ Lần lượt push 3 giá trị vào stack: Giá trị biến a, giá trị biến b, địa chỉ chuỗi “%d %d”
+ **printf(“%d %d”, b, a);**

Procedure (IA32) – Bài tập 1.1

Code assembly

```
1.   .LC0: .string "%d"
2.   .LC1: .string "%d %d"
3.   example:
4.       pushl   %ebp
5.       movl    %esp, %ebp
6.       subl    $8, %esp
7.       movl    $5, -4(%ebp)
8.       movl    $10, -8(%ebp)
9.       leal    -8(%ebp), %eax
10.      pushl   %eax
11.      pushl   $.LC0
12.      call    scanf
13.      addl    $8, %esp
14.      movl    -8(%ebp), %eax
15.      pushl   -4(%ebp)
16.      pushl   %eax
17.      pushl   $.LC1
18.      call    printf
19.      addl    $12, %esp
20.      movl    $0, %eax
21.      leave
22.      ret
```

Code C?

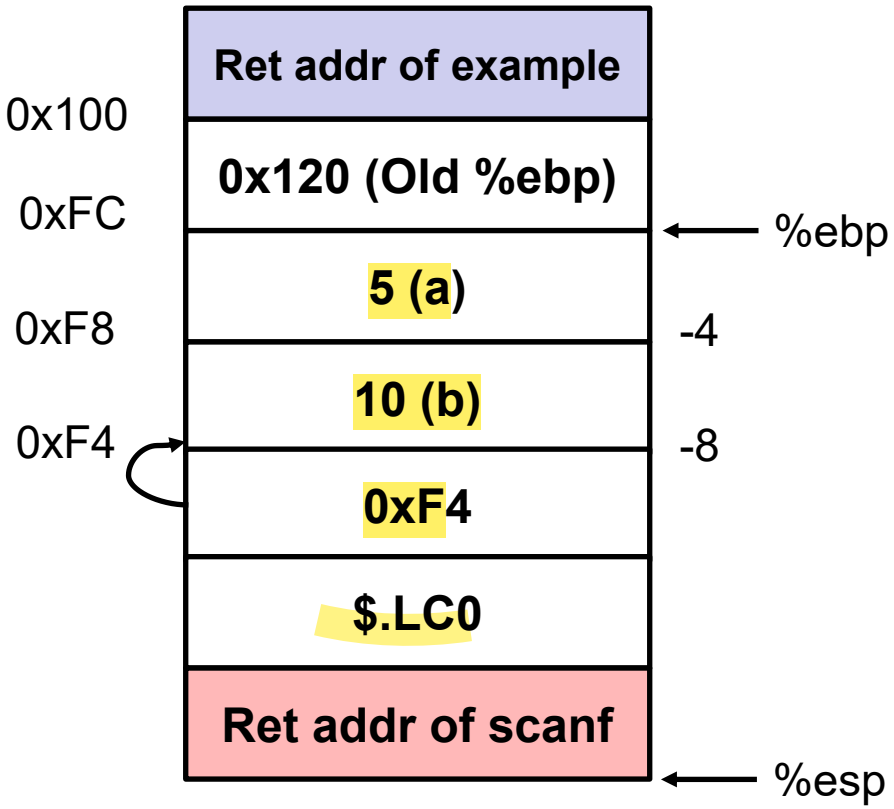
```
int example()
{
    int a = 5;
    int b = 10;
    scanf("%d", &b);
    printf("%d %d", b, a);
    return 0;
}
```

Procedure (IA32) – Bài tập 1.2

Code assembly

```
1.   .LC0: .string "%d"
2.   .LC1: .string "%d %d"
3.   example:
4.       pushl   %ebp
5.       movl    %esp, %ebp
6.       subl    $8, %esp
7.       movl    $5, -4(%ebp)
8.       movl    $10, -8(%ebp)
9.       leal    -8(%ebp), %eax
10.      pushl   %eax
11.      pushl   $.LC0
12.      call    scanf
13.      addl    $8, %esp
14.      movl    -8(%ebp), %eax
15.      pushl   -4(%ebp)
16.      pushl   %eax
17.      pushl   $.LC1
18.      call    printf
19.      addl    $12, %esp
20.      movl    $0, %eax
21.      leave
22.      ret
```

Vẽ stack của **example** đến khi thực hiện lệnh call **scanf**?
Giả sử trước khi thực thi lệnh 4: **%esp = 0x100**, **%ebp = 0x120**



a lưu ở 0xF8, b lưu ở 0xF4

Procedure (IA32) – Bài tập 2.1

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
```

Biết: Tại các ô nhớ
0x0804a02c chứa **stdin (số nguyên)**
0x08048610 chứa chuỗi "\n[buf]: %s\n"
Hàm trên có 3 biến cục bộ
Tại vị trí 16(%esp) là 1 chuỗi buf
fgets cần 3 tham số

Code C

Xác định biến cục bộ?

Từ đoạn mã:
0x3C(%esp) = -4(%ebp) = 0x04030201 → int a
0x38(%esp) = -8(%ebp) = 0 → int b
0x10(%esp) = vị trí 1 chuỗi buf → char * buf

Gọi hàm fgets

Dòng 6 - 11

+ Đưa 3 giá trị vào các vị trí (%esp), 4(%esp), 8(%esp) → ứng với các vị trí tham số thứ 1, thứ 2 và thứ 3.
+ 3 tham số theo thứ tự tăng: địa chỉ vùng nhớ 0x10(%esp), số nguyên 50, giá trị stdin
+ **fgets(buf, 50, stdin);**

Gọi hàm printf

Dòng 12 – 15

+ Đưa 2 giá trị vào các vị trí (%esp), 4(%esp)→ ứng với các vị trí tham số thứ 1, thứ 2.
+ 2 tham số theo thứ tự tăng: địa chỉ chuỗi "\n[buf]: %s\n", địa chỉ chuỗi buf.
+ **printf("\n[buf]: %s\n", buf);**

Procedure (IA32) – Bài tập 2.1

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
```

Biết: Tại các ô nhớ
0x0804a02c chứa **stdin (số nguyên)**
0x08048610 chứa chuỗi "\n[buf]: %s\n"
Hàm trên có 3 biến cục bộ
Tại vị trí 16(%esp) là 1 chuỗi buf
fgets cần 3 tham số

Code C

```
int main(){
    int a = 0x04030201;
    int b = 0;
    char * buf;
    fgets(buf, 50, stdin);
    printf("\n[buf]: %s\n", buf);
    ...
}
```

Procedure (IA32) – Bài tập 2.2

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.  …
```

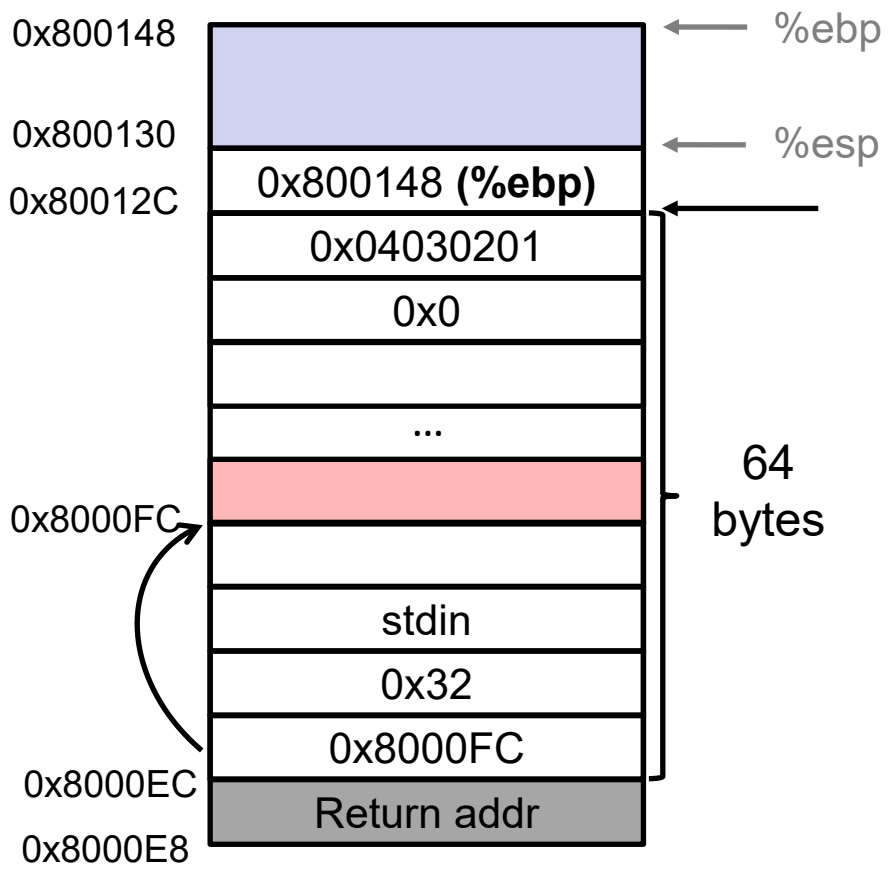
Xác định địa chỉ cụ thể sẽ lưu chuỗi buf?

Vẽ stack đến lệnh call fgets

Giả sử ban đầu:

%ebp = 0x800148 %esp = 0x800130

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



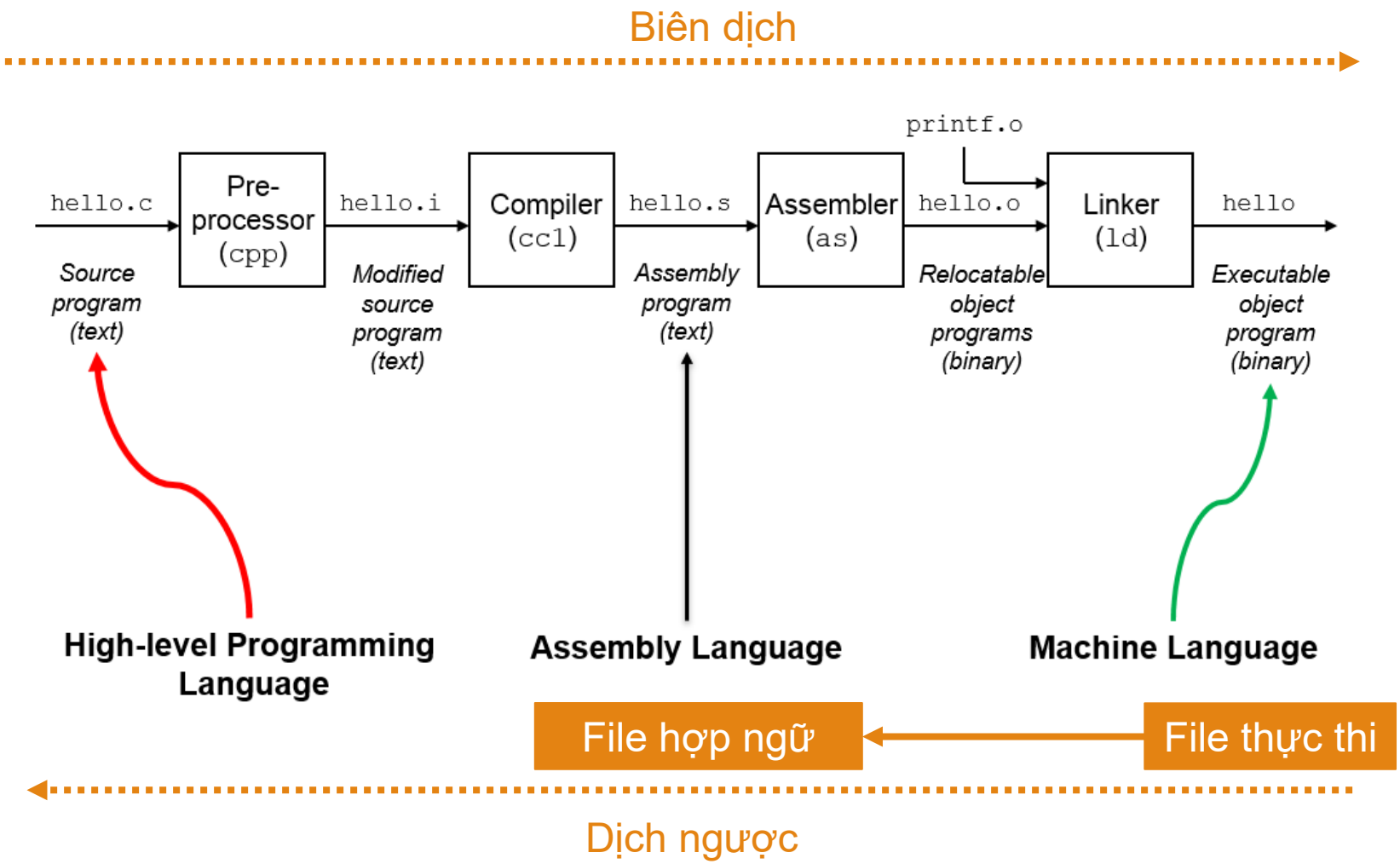
Assignment 3 - Procedure (IA32)

Xem chi tiết Assignment trên Website môn học.

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**

Dịch ngược - Reverse Engineering?



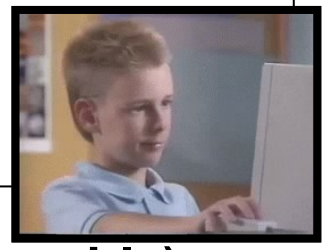
Dịch ngược - Reverse Engineering?

- **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.

```
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     $0xfffffff0,%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)

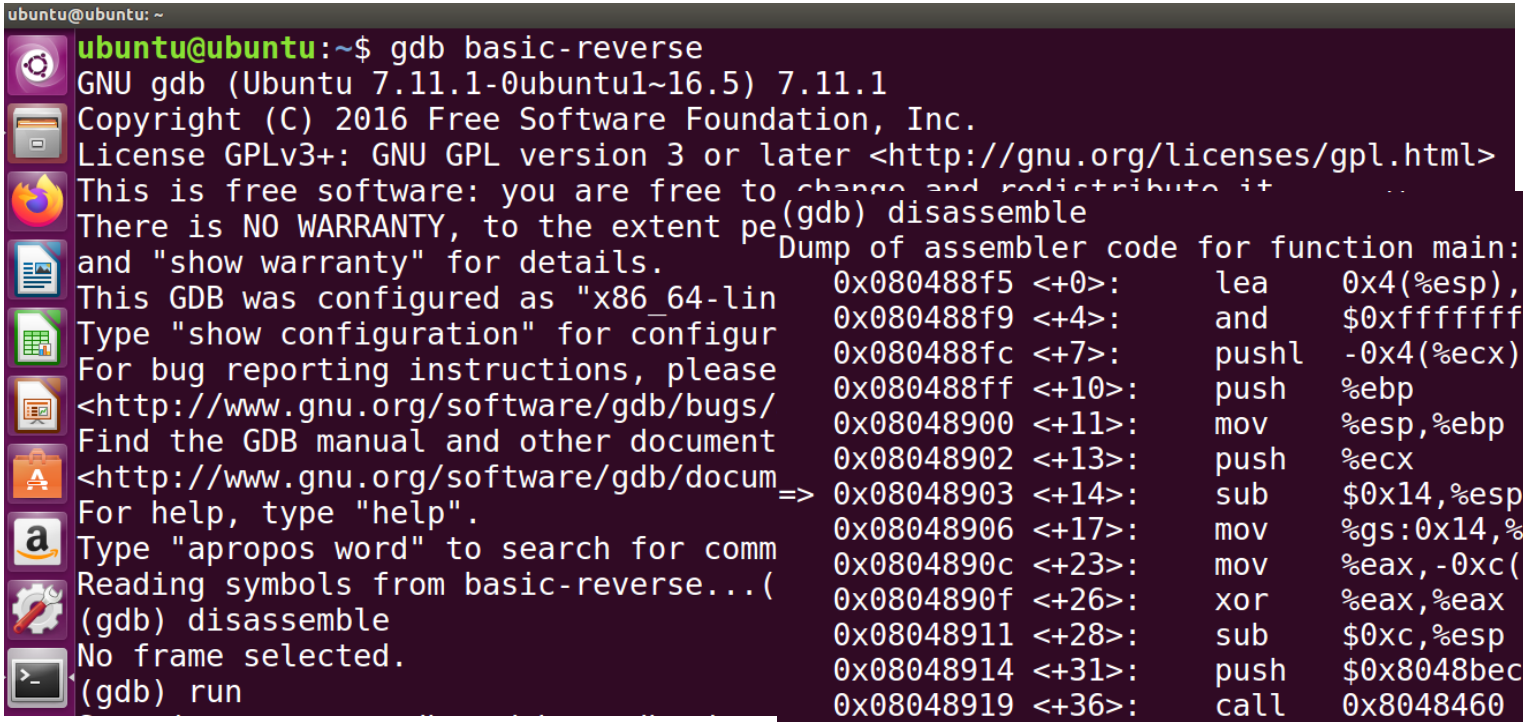
- **objdump – Xuất mã assembly của file thực thi**

```
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)

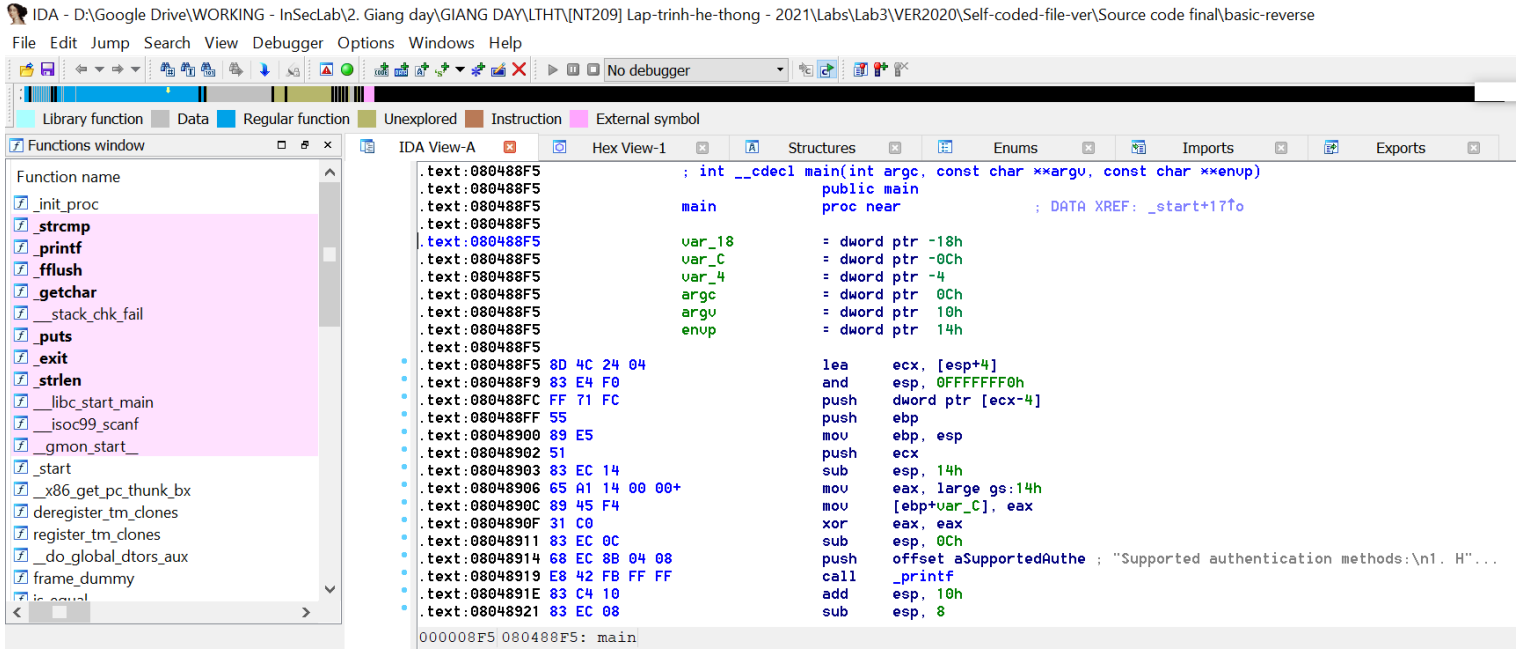
- **GDB Debugger (Phần 3.11 trong giáo trình chính)**



- Command line
- Thường có trên Linux
- Định dạng assembly mặc định: AT&T
- **Có thể phân tích tĩnh hoặc động (debug)**

Dịch ngược – Công cụ (3)

- **IDA Pro**



- Có **giao diện, nhiều cửa sổ cung cấp thông tin, đặc biệt là mã giả**
- Có thể chạy trên Windows
- Định dạng assembly mặc định: **Intel**
- **Có thể phân tích code ở dạng tĩnh (không cần chạy chương trình) và động (debug)**

Nội dung

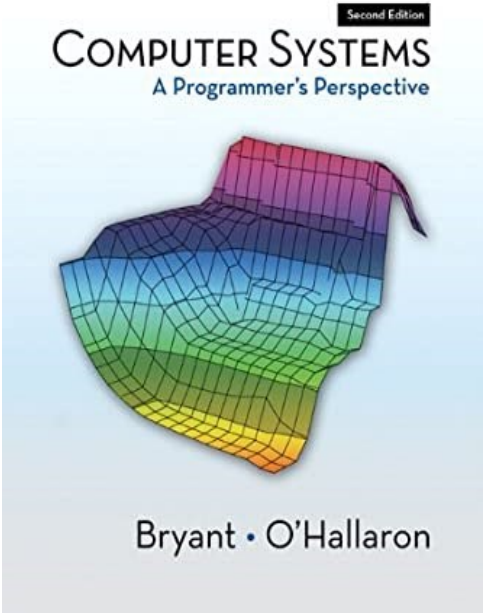
- **Các chủ đề chính:**
  1) Biểu diễn các kiểu dữ liệu và các phép tính toán bit
  2) Ngôn ngữ assembly
  3) Điều khiển luồng trong C với assembly
  4) Các thủ tục/hàm (procedure) trong C ở mức assembly
  5) Biểu diễn mảng, cấu trúc dữ liệu trong C
  6) Một số topic ATTT: reverse engineering, bufferoverflow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

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 :)