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: Cơ bản

Nội dung

- **Sơ lược lịch sử các bộ xử lý và kiến trúc Intel**
- C, assembly, mã máy
- Cơ bản về Assembly: Registers, move
- Các phép tính toán học và logic

Câu hỏi có điểm cộng

Intel x86 Processors

- **Thống trị thị trường laptop/desktop/server**
- **Sự phát triển trong thiết kế**
  - Cho phép tương thích ngược đến 8086 (1978)
  - Hỗ trợ ngày càng nhiều tính năng
- **Complex instruction set computer (CISC)**
  - Nhiều instructions khác nhau với nhiều format khác nhau
  - Khó đạt hiệu suất như Reduced Instruction Set Computers (RISC)
  - Nhưng Intel đã làm được điều đó!

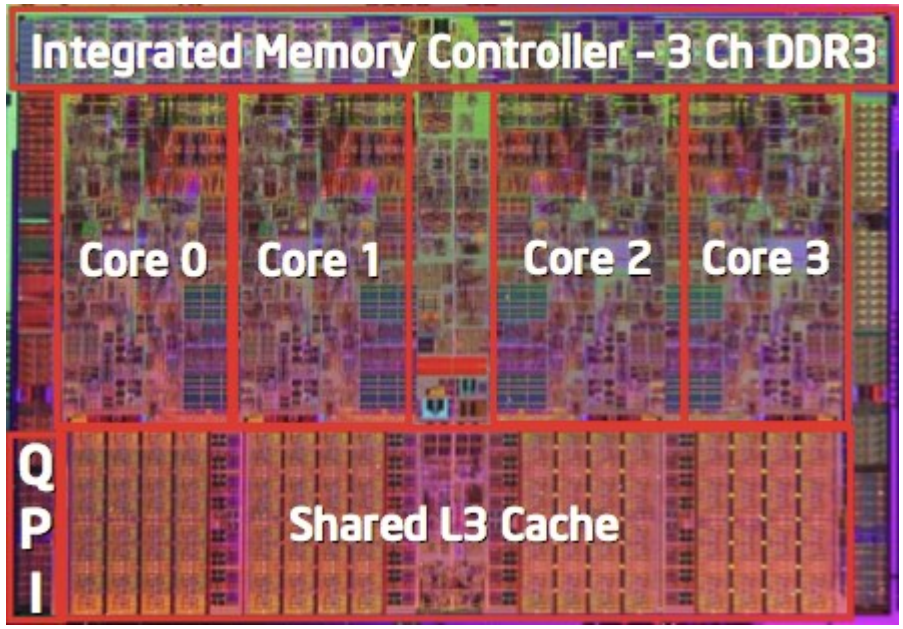
Intel x86: Các mốc phát triển

Tên	*Thời gian*	*Transistors*	*MHz*
8086	**1978**	**29K**	**5-10**
▪ Bộ xử lý Intel 16-bit đầu tiên. Cho IBM PC & DOS			
▪ Không gian địa chỉ 1MB			
386	**1985**	**275K**	**16-33**
▪ Bộ xử lý Intel 32-bit đầu tiên, gọi tắt là IA32			
▪ Được thêm “flat addressing”, có thể chạy Unix			
Pentium 4E	**2004**	**125M**	**2800-3800**
▪ Bộ xử lý Intel 64-bit đầu tiên, gọi tắt là x86-64			
Core 2	**2006**	**291M**	**1060-3500**
▪ Bộ xử lý Intel nhiều core đầu tiên			
Core i7	**2008**	**731M**	**1700-3900**
▪ 4 cores			

Intel x86 Processors (tt)

- **Machine Evolution**

386	1985	0.3M
Pentium	1993	3.1M
Pentium/MMX	1997	4.5M
PentiumPro	1995	6.5M
Pentium III	1999	8.2M
Pentium 4	2001	42M
Core 2 Duo	2006	291M
Core i7	2008	731M



- **Tính năng được thêm**
  - Instructions để hỗ trợ multimedia operations
  - Instructions cho phép các hoạt động có điều kiện hiệu quả hơn
  - Chuyển từ 32 bits sang 64 bits
  - Nhiều core hơn

Phạm vi môn học

- **IA32** (32 bit)
- **x86-64** (64 bit)

Nội dung

- Sơ lược lịch sử các bộ xử lý và kiến trúc Intel
- **C, assembly, mã máy**
- Cơ bản về Assembly: Registers, move
- Các phép tính toán học và logic

Assembly: Vì sao?

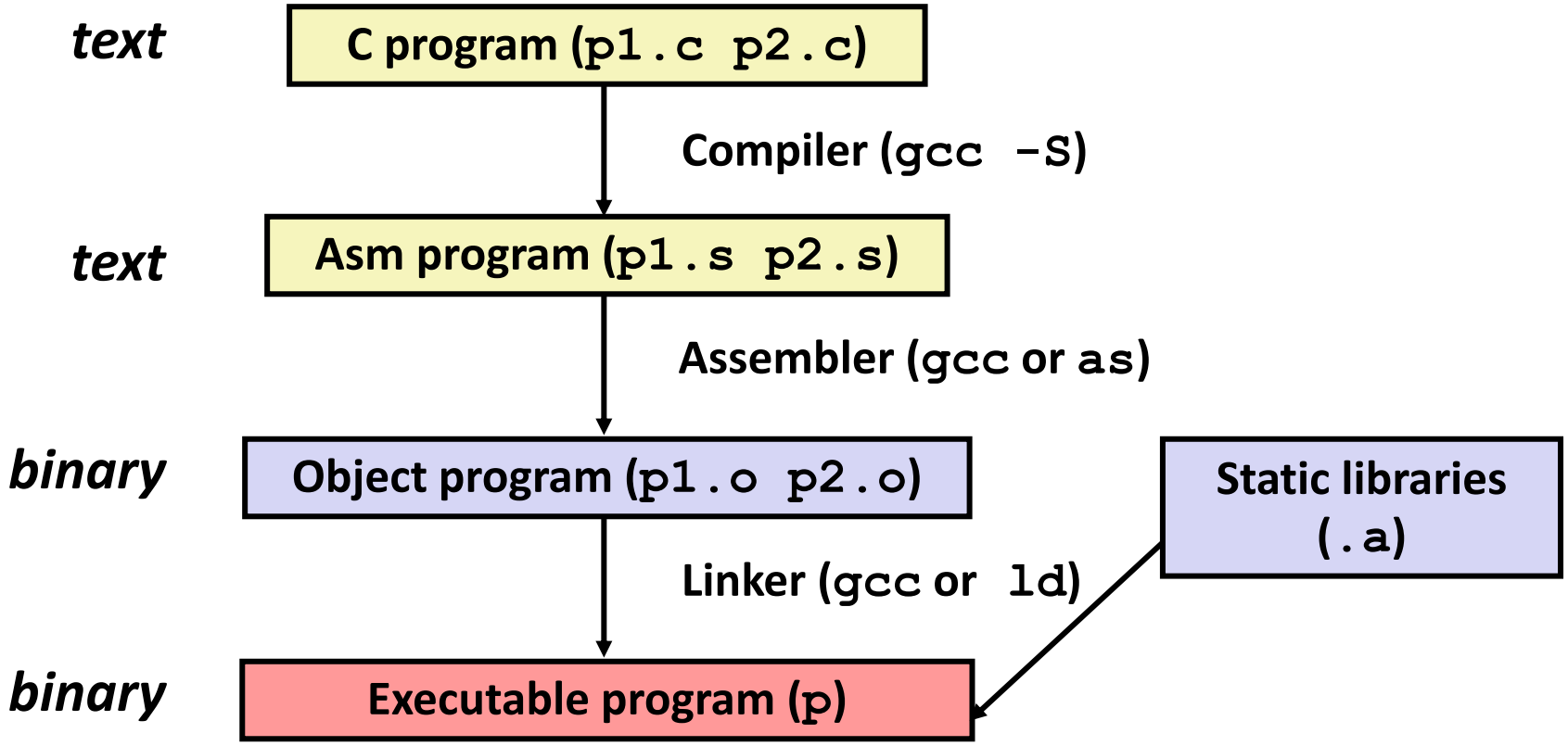
- **Ngôn ngữ cấp cao**
  - Dễ sử dụng
  - Tính năng hỗ trợ: kiểm tra kiểu dữ liệu, phát hiện lỗi…
  - Có thể biên dịch và thực thi trên nhiều máy tính
- **Assembly – Hợp ngữ**
  - Phụ thuộc nhiều vào máy tính thực thi
  - Hiểu được hoạt động của hệ thống lúc thực thi chương trình
    - Stack, bộ nhớ, register…
    - Các lỗ hổng mức hệ thống có thể có khi lập trình
    - Đọc/hiểu assembly: Skill cần thiết cho ATTT!
  - Khả năng tối ưu của chương trình

Các định nghĩa

- **Architecture: (ISA: instruction set architecture) Các thành phần trong thiết kế bộ xử lý cần hiểu để viết được các mã assembly/mã máy**
  - Ví dụ: định nghĩa tập lệnh, registers (thanh ghi).
- **Microarchitecture: hiện thực của architecture.**
  - Ví dụ: kích thước cache và tần số core.
- **Các dạng mã:**
  - Mã máy (Machine Code): Chương trình ở dạng các byte sẽ được các bộ xử lý thực thi
  - Mã hợp ngữ (Assembly Code): Biểu diễn dạng text của mã máy
- **Ví dụ các ISA:**
  - Intel: x86, IA32, Itanium, x86-64
  - ARM: Sử dụng trong hầu hết các thiết bị di động

Từ mã C đến mã thực thi

- Giả sử có các mã C trong các file **p1.c p2.c**
- Quá trình biên dịch với câu lệnh: `gcc p1.c p2.c -o p`
  - File nhị phân sau khi biên dịch được lưu trong file **p**



Từ mã C đến mã thực thi: Ví dụ

```
*dest = t;
```

```
movq %rax, (%rbx)
```

```
0x40059e:  48 89 03
```

- **Mã C**
  - Lưu giá trị của **t** vào vị trí được trỏ bởi **dest**
- **Mã Assembly**
  - Đưa 8-byte giá trị vào bộ nhớ
  - Toán hạng:

    t: Register **%rax**

    dest: Register **%rbx**

    ***dest:** Memory **M[%rbx]**
- **Object Code**
  - Instruction có kích thước 3 bytes
  - Lưu tại địa chỉ **0x40059e**

Mã assembly: Biên dịch từ mã C

C Code

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

Generated IA32 Assembly

```
sum:
   pushl %ebp
   movl %esp,%ebp
   movl 12(%ebp),%eax
   addl 8(%ebp),%eax
   popl %ebp
   ret
```

Thu được với lệnh:

```
gcc –S sum.c
```

Tạo ra file sum.s

***Lưu ý*: Có thể ra file kết quả với nội dung không giống nhau do khác biệt ở phiên bản gcc và các thiết lập của compiler.**

Thêm: Tool cung cấp mã assembly của code C (online): **https://godbolt.org/**

Object code

Code for `sum`

```
0x401040 <sum>:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x5d
    0xc3
```

- **Total of 11 bytes**
- **Each instruction 1, 2, or 3 bytes**
- **Starts at address `0x401040`**

- **Assembler**
  - Chuyển từ file `.s` sang `.o`
  - Biểu diễn nhị phân của mỗi instruction
  - Phiên bản gần hoàn thiện của mã thực thi
  - Thiếu phần liên kết giữa mã code trong nhiều files
- **Linker**
  - Giải quyết các tham chiếu giữa các file
  - Liên kết với các thư viện tĩnh
    - E.g., code của các hàm **malloc, printf**
  - Một số thư viện được *liên kết động*
    - Liên kết được thực hiện khi chương trình bắt đầu chạy

Mã assembly: Disassembling Object Code

Disassembled

```
080483c4 <sum>:
 80483c4:   55          push   %ebp
 80483c5:   89 e5       mov    %esp,%ebp
 80483c7:   8b 45 0c    mov    0xc(%ebp),%eax
 80483ca:   03 45 08    add    0x8(%ebp),%eax
 80483cd:   5d          pop    %ebp
 80483ce:   c3          ret
```

- **Disassembler - objdump**

  `objdump -d <tên file>`

  - Công cụ hữu ích để kiểm tra object code
  - Phân tích các chuỗi bit của chuỗi các instructions
  - Tạo ra mã assembly gần đúng
  - Có thể chạy trên cả file `a.out` (file thực thi đầy đủ) hoặc `.o`

Disassembling: Công cụ khác

Object

```
0x401040:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x5d
    0xc3
```

Disassembled

```
Dump of assembler code for function sum:
0x080483c4 <sum+0>:     push   %ebp
0x080483c5 <sum+1>:     mov    %esp,%ebp
0x080483c7 <sum+3>:     mov    0xc(%ebp),%eax
0x080483ca <sum+6>:     add    0x8(%ebp),%eax
0x080483cd <sum+9>:     pop    %ebp
0x080483ce <sum+10>:    ret
```

- **Bên trong gdb Debugger:** Ví dụ

  gdb `<tên file>`

  disassemble sum

  - Disassemble các hàm (procedure)

  x/11xb sum

  - Kiểm tra giá trị của 11 bytes bắt đầu từ `sum`

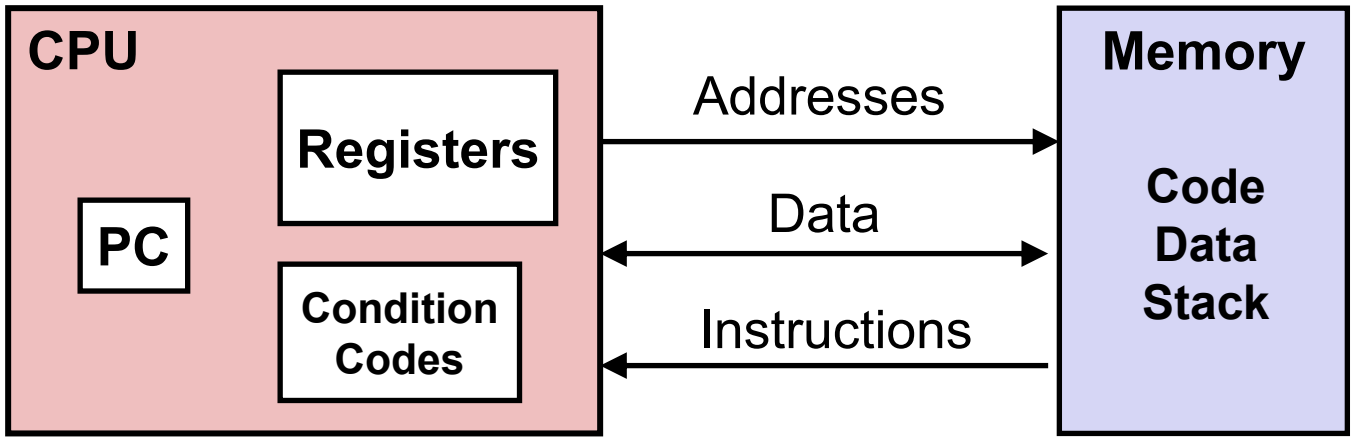
Chúng ta có thể disassembling những gì?

```
% objdump -d WINWORD.EXE

WINWORD.EXE:    file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:
30001001:
30001003:
30001005:
3000100a:
```

Reverse engineering forbidden by Microsoft End User License Agreement

- Bất kỳ thứ gì được xem là mã thực thi (executable code)
- Disassembler kiểm tra các bytes và dựng lại các mã assembly

Góc nhìn của mã assembly/mã máy



Programmer-Visible State

- **PC: Program counter**
  - Địa chỉ của instruction tiếp theo cần thực thi
  - Gọi là "EIP" (IA32) hoặc "RIP" (x86-64)
- **Registers (thanh ghi)**
  - Thường được sử dụng để lưu dữ liệu chương trình
- **Condition codes**
  - Lưu thông tin trạng thái về các phép tính toán học hoặc logic được thực hiện gần nhất.
  - Được dùng để rẽ nhánh có điều kiện

- **Bộ nhớ**
  - Mảng các byte được đánh địa chỉ
  - Chứa code và dữ liệu người dùng
  - Stack hỗ trợ các thủ tục (procedures)

Đặc điểm của mã assembly: Kiểu dữ liệu

- **Các kiểu dữ liệu "số nguyên" có kích thước 1, 2, 4, hoặc 8 bytes**
  - Các giá trị
  - Địa chỉ (pointer chưa được định kiểu)
- **Dữ liệu dấu chấm động (floating point) có kích thước 4, 8, hoặc 10 bytes**
- **Mã code: Chuỗi bytes mã hoá chuỗi các instructions**
- **KHÔNG có kiểu dữ liệu "tích hợp" như mảng hay cấu trúc dữ liệu**
  - Bản chất là những byte được cấp phát liên tiếp trong bộ nhớ

Đặc điểm của mã assembly: Hoạt động

- **Nhóm 1: Chuyển dữ liệu giữa bộ nhớ và thanh ghi**
  - Lấy dữ liệu từ bộ nhớ sang thanh ghi
  - Lưu dữ liệu của thanh ghi vào bộ nhớ
- **Nhóm 2: Thực hiện các phép tính toán trên thanh ghi hoặc dữ liệu trong bộ nhớ**
- **Nhóm 3: Chuyển luồng thực thi**
  - Nhảy không điều kiện
  - Rẽ nhánh có điều kiện
  - Các thủ tục (procedures)

Lưu ý 1: Định dạng assembly của kiến trúc Intel

- **Phạm vi môn học**: Mã assembly dưới định dạng AT&T
  - Định dạng mặc định của các công cụ GCC, Objdump…
- **Định dạng khác**: Intel
  - Microsoft

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

AT&T format

```
    Assembly code for simple in Intel format
1   simple:
2     push    ebp
3     mov     ebp, esp
4     mov     edx, DWORD PTR [ebp+8]
5     mov     eax, DWORD PTR [ebp+12]
6     add     eax, DWORD PTR [edx]
7     mov     DWORD PTR [edx], eax
8     pop     ebp
9     ret
```

Intel format

Lưu ý 2: Do not panic! 32-bit & 64-bit

- Mã assembly của 1 hàm **simple_l** ở 2 phiên bản 32-bit và 64-bit

```
    xp at %ebp+8, y at %ebp+12
1   simple_l:
2     pushl   %ebp
3     movl    %esp, %ebp
4     movl    8(%ebp), %edx
5     movl    12(%ebp), %eax
6     addl    (%edx), %eax
7     movl    %eax, (%edx)
8     popl    %ebp
9     ret
```

IA32

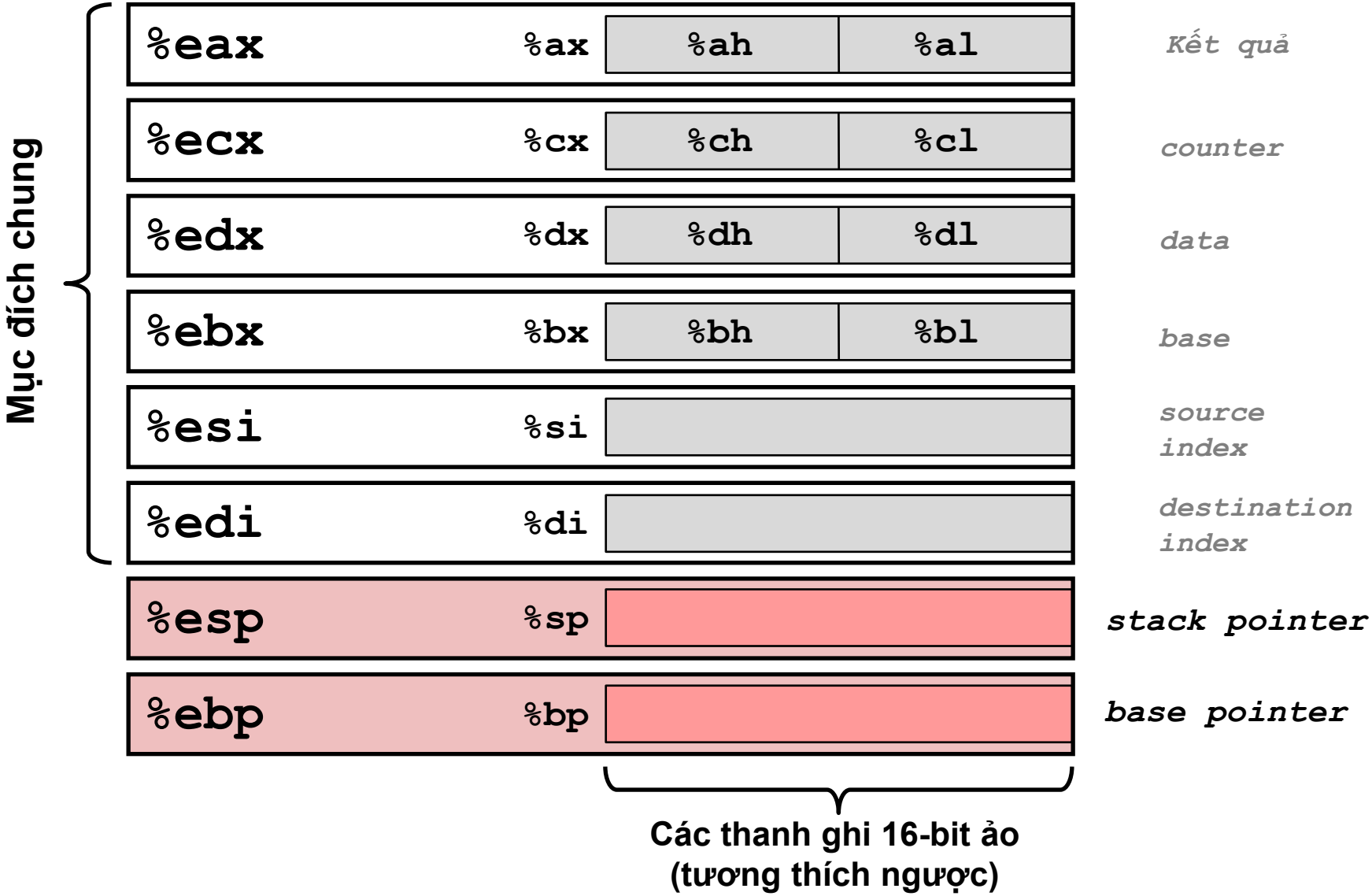
```
    xp in %rdi, y in %rsi
1   simple_l:
2     movq    %rsi, %rax
3     addq    (%rdi), %rax
4     movq    %rax, (%rdi)
5     ret
```

x86_64

Nội dung

- Sơ lược lịch sử các bộ xử lý và kiến trúc Intel
- C, assembly, mã máy
- **Cơ bản về Assembly: Registers, move**
- Các phép tính toán học và logic

Các thanh ghi IA32 – 8 thanh ghi 32 bit



Các thanh ghi x86-64 – 16 thanh ghi

%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

- Mở rộng các thanh ghi 32-bit đã có thành 64-bit, thêm 8 thanh ghi mới.
- **%ebp/%rbp** thành thanh ghi có mục đích chung.
- Có thể tham chiếu đến các 4 bytes thấp (cũng như các 1 & 2 bytes thấp)

Chuyển dữ liệu - Moving Data (IA32)

- **Chuyển dữ liệu**

  `movl` *Source, Dest*

- **Các kiểu toán hạng**
  - ***Immediate – Hằng số:*** Các hằng số nguyên
    - Ví dụ: `$0x400, $-533`
    - Giống hằng số trong C, nhưng có tiền tố `'$'`
    - Mã hoá với 1, 2, hoặc 4 bytes
    - ***Chỉ có thể dùng ở Source!***
  - ***Register – Thanh ghi:*** Các thanh ghi được hỗ trợ
    - Ví dụ: `%eax, %esi`
    - Nhưng `%esp` và `%ebp` được dành riêng với mục đích đặc biệt
    - Một số khác có tác dụng đặc biệt với một số instruction
  - ***Memory – Bộ nhớ:*** các bytes liên tục của bộ nhớ tại địa chỉ nhất định, địa chỉ đó có thể được lưu trong thanh ghi
    - Ví dụ: `0x203,(0x100),(%eax)`
    - Có nhiều "address mode" khác

Các tổ hợp toán hạng cho `movl`

	Source	Dest	Src,Dest	C Analog
`movl`	*Imm*	*Reg*	`movl $0x4,%eax`	`temp = 0x4;`
		Mem	`movl $-147,(%eax)`	`*p = -147;`
	Reg	*Reg*	`movl %eax,%edx`	
		Mem	`movl %eax,(%edx)`	`*p = temp;`
	Mem	*Reg*	`movl (%eax),%edx`	`temp = *p;`

Không thể** thực hiện chuyển dữ liệu **bộ nhớ - bộ nhớ** với **duy nhất 1 instruction!

Các chế độ đánh địa chỉ bộ nhớ đơn giản

- **Thông thường (R) Mem[Reg[R]]**
  - Thanh ghi R xác định địa chỉ bộ nhớ
  - Tương ứng với tham chiếu bằng Pointer trong C

```
movl (%ecx),%eax
```

- **Dịch chuyển D(R) Mem[Reg[R]+D]**
  - Thanh ghi R xác định nơi bắt đầu của vùng nhớ
  - Hằng số D xác định offset từ vị trí bắt đầu đó

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

Các chế độ đánh địa chỉ bộ nhớ đơn giản: Ví dụ 1

- Giả sử ta có **%eax = 0x100** và các giá trị bộ nhớ như hình bên

Memory	Addr
25	**0x100**
146	**0x104**

- Kết quả lưu trong **%ebx** ở 2 câu lệnh dưới giống hay khác nhau?

movl %eax, %ebx

vs

movl (%eax), %ebx

Các chế độ đánh địa chỉ bộ nhớ đầy đủ

- **Dạng tổng quát nhất**

  D(Rb,Ri,S) **Mem[Reg[Rb]+S*Reg[Ri]+ D]**

  - D: Hằng số "dịch chuyển" 1, 2, hoặc 4 bytes
  - Rb: Base register: Bất kỳ thanh ghi nào được hỗ trợ
  - Ri: Index register: Bất kỳ thanh ghi nào, ngoại trừ **%rsp** hoặc **%esp**
  - S: Scale: 1, 2, 4, hoặc 8 (*vì sao là những số này?*)

- **Các trường hợp đặc biệt**

  (Rb,Ri) **Mem[Reg[Rb]+Reg[Ri]]**

  D(Rb,Ri) **Mem[Reg[Rb]+Reg[Ri]+D]**

  (Rb,Ri,S) **Mem[Reg[Rb]+S*Reg[Ri]]**

Tính toán địa chỉ: Ví dụ

`%edx`	`0xf000`
`%ecx`	`0x0100`

D(Rb,Ri,S) = Mem[Reg[Rb]+S*Reg[Ri]+ D]

Biểu thức	Cách tính địa chỉ	Địa chỉ
`0x8(%edx)`		
`(%edx,%ecx)`		
`(%edx,%ecx,4)`		
`0x80(,%edx,2)`		

Lưu ý: **Suffix cho lệnh `mov` trong AT&T**

- **Quyết định số byte dữ liệu sẽ được "move"**
  - mov**b** 1 byte
  - mov**w** 2 bytes
  - mov**l** 4 bytes
  - mov**q** 8 bytes (dùng với các thanh ghi x86_64)
  - mov Số bytes tuỳ ý (phù hợp với tất cả số byte ở trên)
- Lưu ý: **Các thanh ghi** dùng trong lệnh mov cần đảm bảo phù hợp với **suffix**
  - Số byte dữ liệu sẽ được move

Lệnh mov không hợp lệ?

1. `movl %eax, %ebx`
2. `movb $123, %bl`
3. `movl %eax, %bl`
4. `movb $3, (%ecx)`
5. `movl 0x100, (%eax)`
6. `mov %ecx, $100`
7. `mov (%eax), %bl`
8. `movb $3, 0x200`

Giải thích? 

Các chế độ đánh địa chỉ bộ nhớ:
Ví dụ 2 (IA32)

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

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

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

   popl  %ebx
   popl  %ebp
   ret
```

Set Up

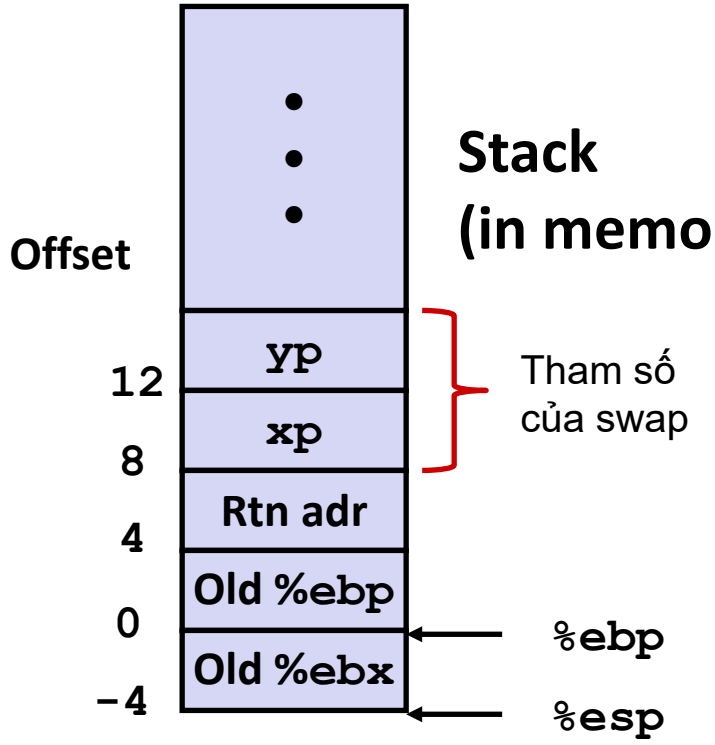
Body

Finish

Hiểu hàm Swap() (IA32)

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

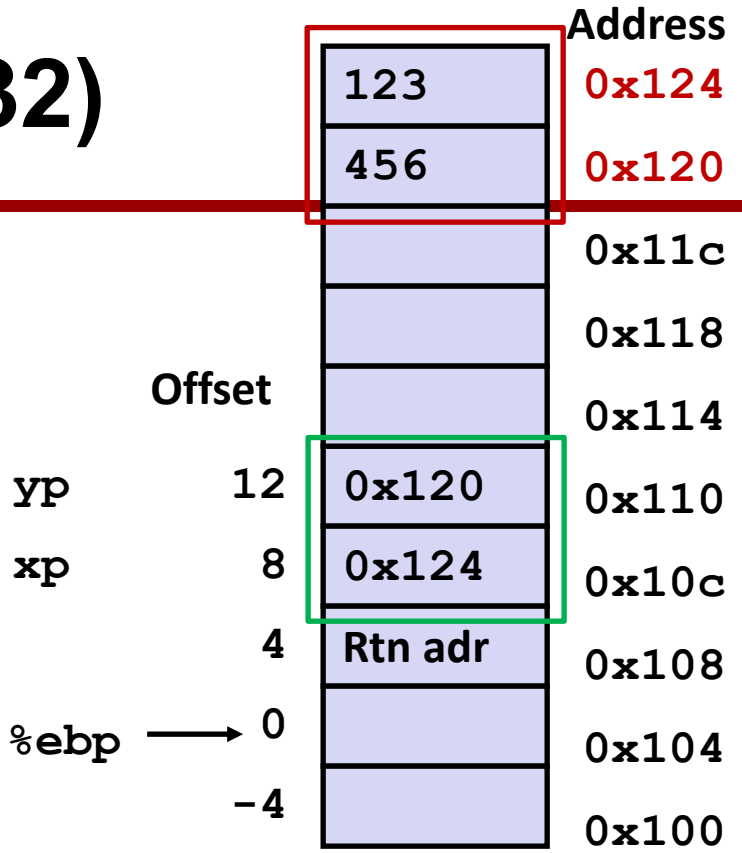
Register	Value
%edx	xp
%ecx	yp
%ebx	t0
%eax	t1



xp lưu ở ebp + 8, yp lưu ở ebp + 12

```
movl   8(%ebp), %edx    # edx = xp
movl   12(%ebp), %ecx   # ecx = yp
movl   (%edx), %ebx     # ebx = *xp (t0)
movl   (%ecx), %eax     # eax = *yp (t1)
movl   %eax, (%edx)     # *xp = t1
movl   %ebx, (%ecx)     # *yp = t0
```

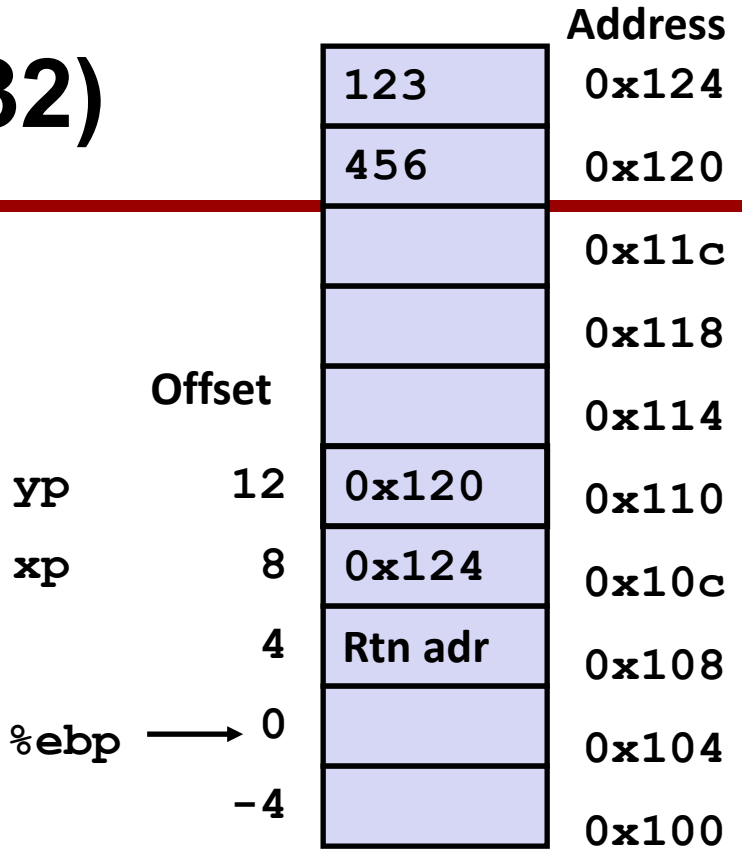
Hiểu hàm Swap() (IA32)



xp lưu ở ebp + 8, yp lưu ở ebp + 12

```
movl   8(%ebp), %edx    # edx = xp
movl   12(%ebp), %ecx   # ecx = yp
movl   (%edx), %ebx     # ebx = *xp (t0)
movl   (%ecx), %eax     # eax = *yp (t1)
movl   %eax, (%edx)     # *xp = t1
movl   %ebx, (%ecx)     # *yp = t0
```

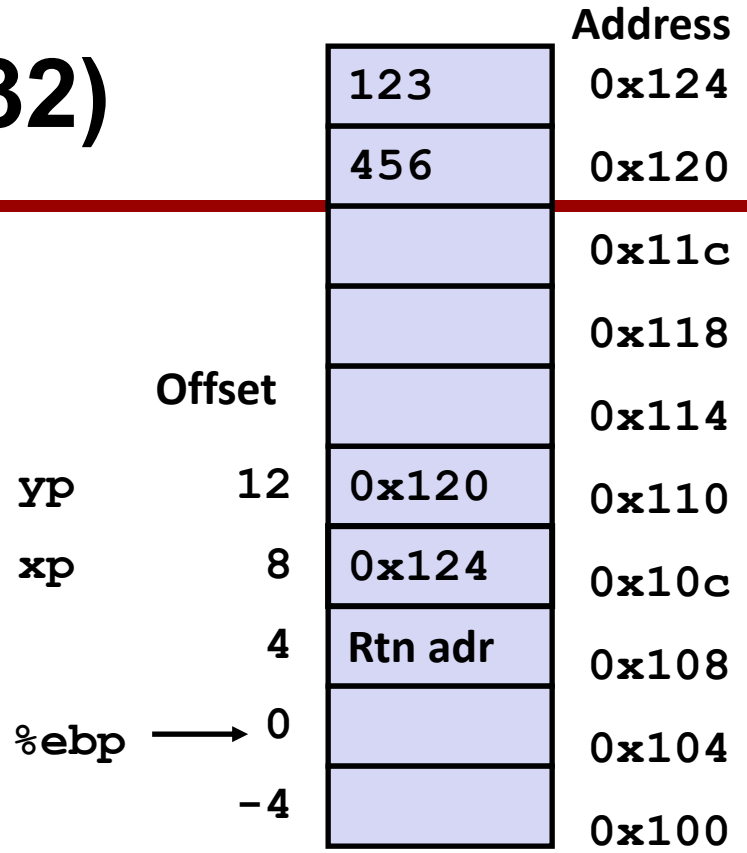
Hiểu hàm Swap() (IA32)




```
xp lưu ở ebp + 8, yp lưu ở ebp + 12
movl   8(%ebp), %edx    # edx = xp
movl   12(%ebp), %ecx   # ecx = yp
movl   (%edx), %ebx     # ebx = *xp (t0)
movl   (%ecx), %eax     # eax = *yp (t1)
movl   %eax, (%edx)     # *xp = t1
movl   %ebx, (%ecx)     # *yp = t0
```

Hiểu hàm Swap() (IA32)

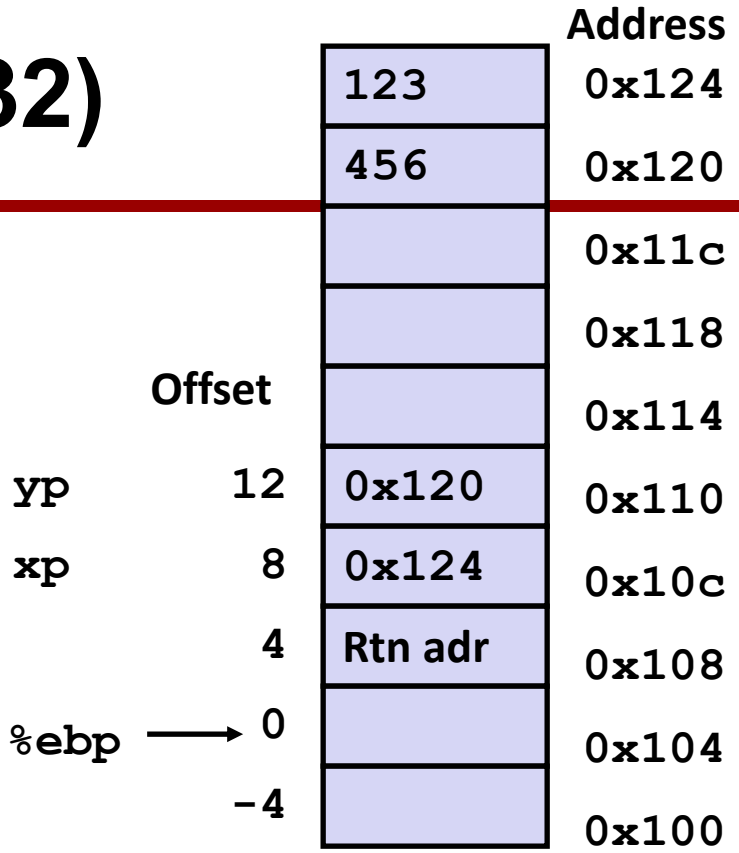
%eax	
%edx	0x124
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104




```
xp lưu ở ebp + 8, yp lưu ở ebp + 12
movl   8(%ebp), %edx     # edx = xp
movl   12(%ebp), %ecx    # ecx = yp
movl   (%edx), %ebx      # ebx = *xp (t0)
movl   (%ecx), %eax      # eax = *yp (t1)
movl   %eax, (%edx)      # *xp = t1
movl   %ebx, (%ecx)      # *yp = t0
```

Hiểu hàm Swap() (IA32)

%eax	
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

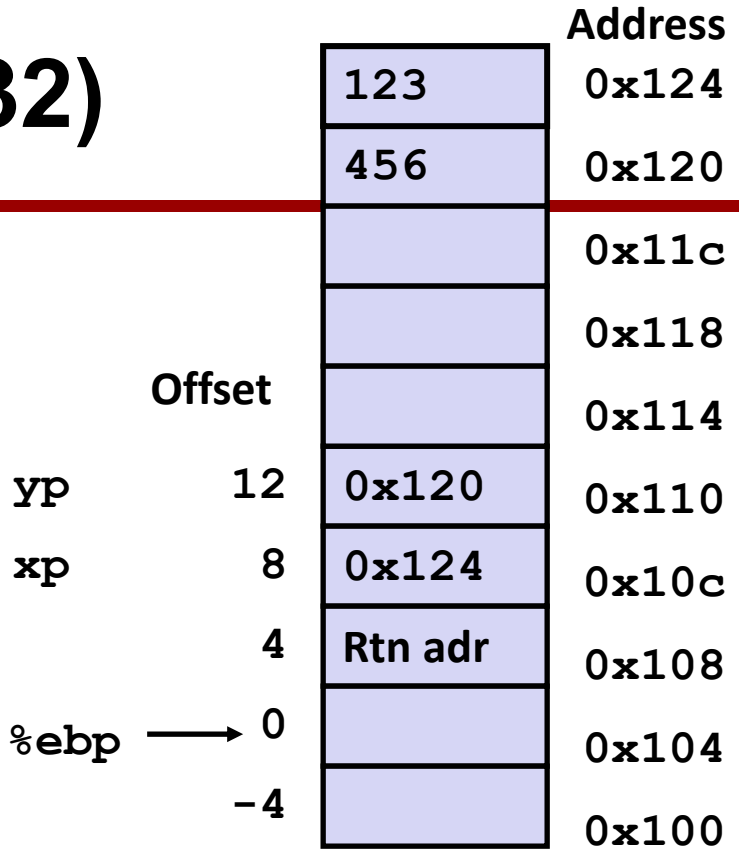


xp lưu ở ebp + 8, yp lưu ở ebp + 12

```
movl   8(%ebp), %edx     # edx = xp
movl   12(%ebp), %ecx    # ecx = yp
movl   (%edx), %ebx      # ebx = *xp (t0)
movl   (%ecx), %eax      # eax = *yp (t1)
movl   %eax, (%edx)      # *xp = t1
movl   %ebx, (%ecx)      # *yp = t0
```

Hiểu hàm Swap() (IA32)

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

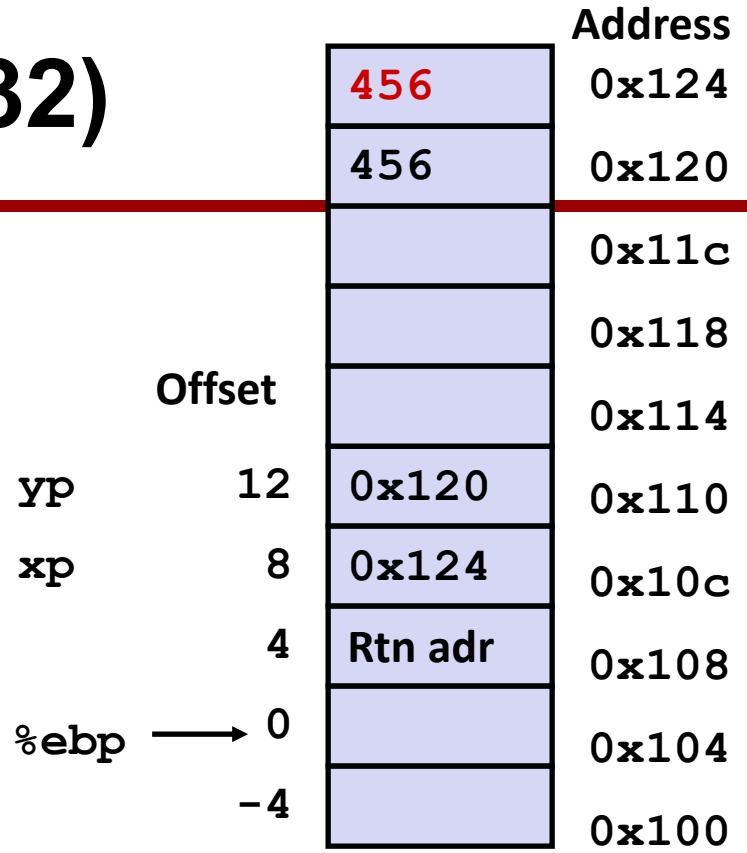


xp lưu ở ebp + 8, yp lưu ở ebp + 12

```
movl   8(%ebp), %edx     # edx = xp
movl   12(%ebp), %ecx    # ecx = yp
movl   (%edx), %ebx      # ebx = *xp (t0)
movl   (%ecx), %eax      # eax = *yp (t1)
movl   %eax, (%edx)      # *xp = t1
movl   %ebx, (%ecx)      # *yp = t0
```

Hiểu hàm Swap() (IA32)

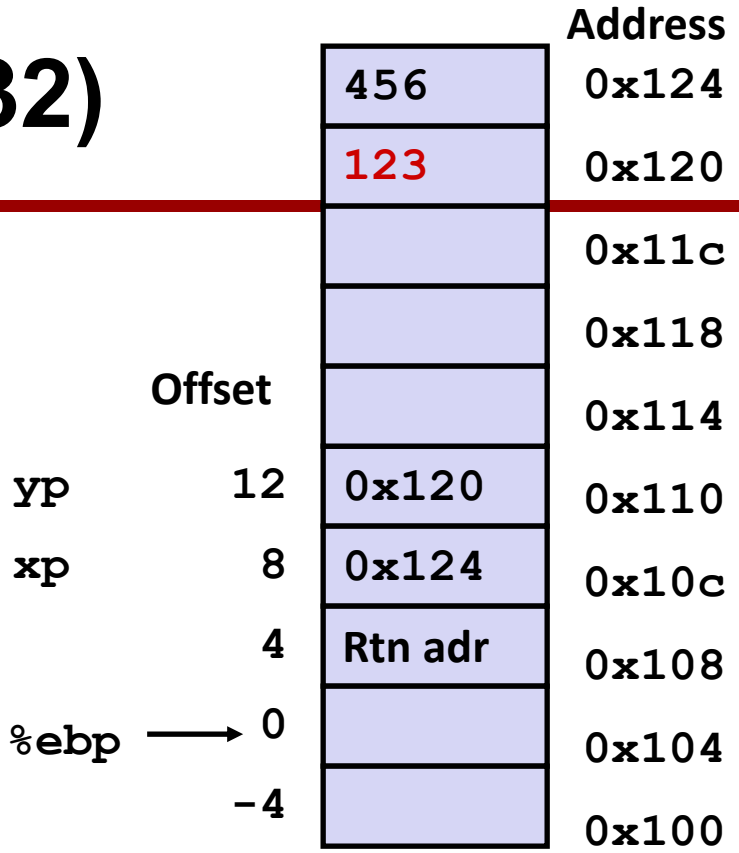
%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104




```
xp lưu ở ebp + 8, yp lưu ở ebp + 12
movl   8(%ebp), %edx     # edx = xp
movl   12(%ebp), %ecx    # ecx = yp
movl   (%edx), %ebx      # ebx = *xp (t0)
movl   (%ecx), %eax      # eax = *yp (t1)
movl   %eax, (%edx)      # *xp = t1
movl   %ebx, (%ecx)      # *yp = t0
```

Hiểu hàm Swap() (IA32)

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104




```
xp lưu ở ebp + 8, yp lưu ở ebp + 12
movl   8(%ebp), %edx    # edx = xp
movl   12(%ebp), %ecx   # ecx = yp
movl   (%edx), %ebx     # ebx = *xp (t0)
movl   (%ecx), %eax     # eax = *yp (t1)
movl   %eax, (%edx)     # *xp = t1
movl   %ebx, (%ecx)     # *yp = t0
```

Các chế độ đánh địa chỉ bộ nhớ:
Ví dụ 2 (x86_64)

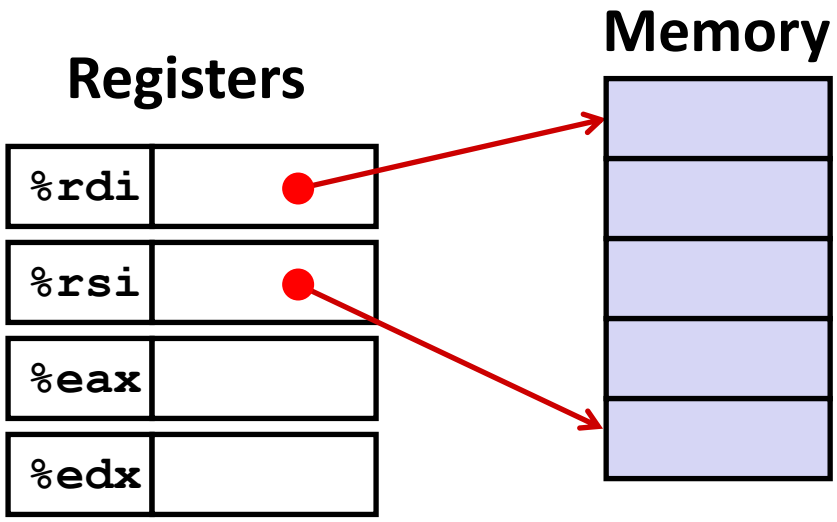
```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
                                    } Set Up

  movl   (%rdi), %edx
  movl   (%rsi), %eax               } Body
  movl   %eax, (%rdi)
  movl   %edx, (%rsi)

  ret                               } Finish
```

Why so easy??

Hiểu hàm `Swap()` (x86_64)

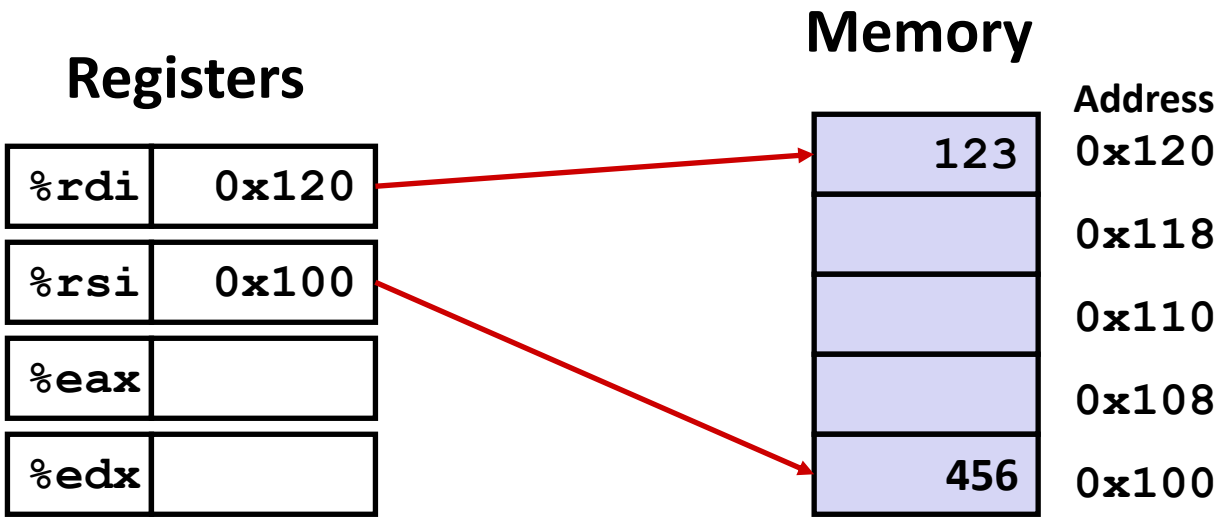
```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



Register	Value
%rdi	xp
%rsi	yp
%eax	t0
%edx	t1

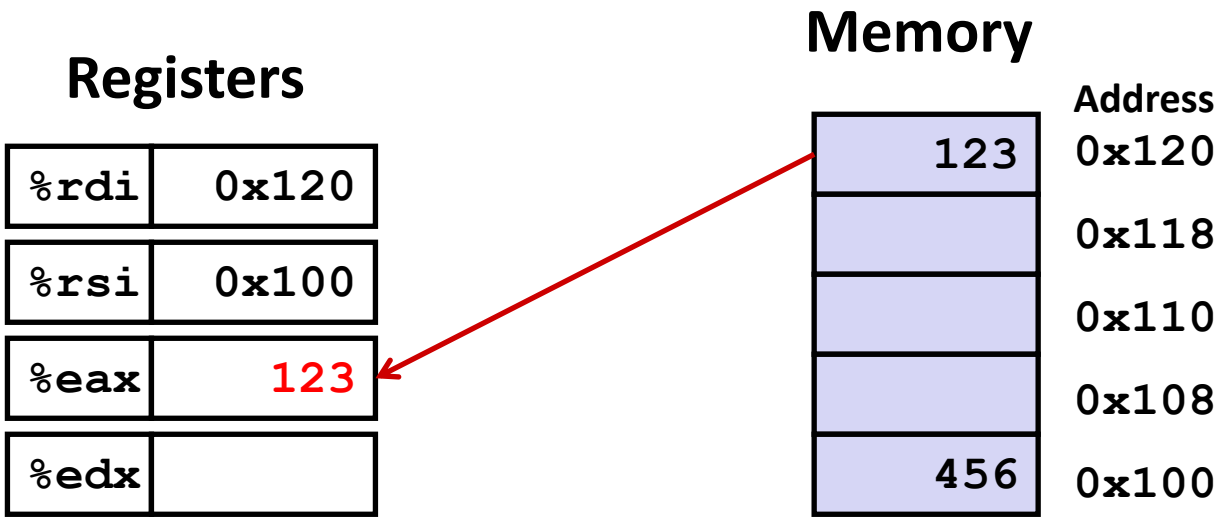
```
swap:
    movl    (%rdi), %eax  # t0 = *xp
    movl    (%rsi), %edx  # t1 = *yp
    movl    %edx, (%rdi)  # *xp = t1
    movl    %eax, (%rsi)  # *yp = t0
    ret
```

Hiểu hàm `Swap()` (x86_64)



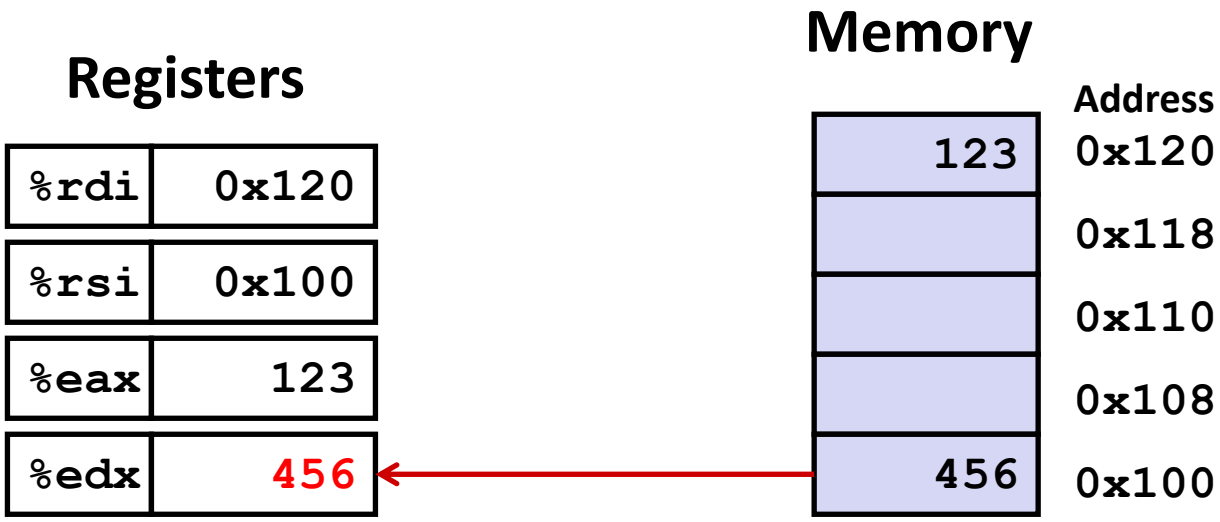

```
swap:
    movl    (%rdi), %eax   # t0 = *xp
    movl    (%rsi), %edx   # t1 = *yp
    movl    %edx, (%rdi)   # *xp = t1
    movl    %eax, (%rsi)   # *yp = t0
    ret
```

Hiểu hàm `Swap()` (x86_64)



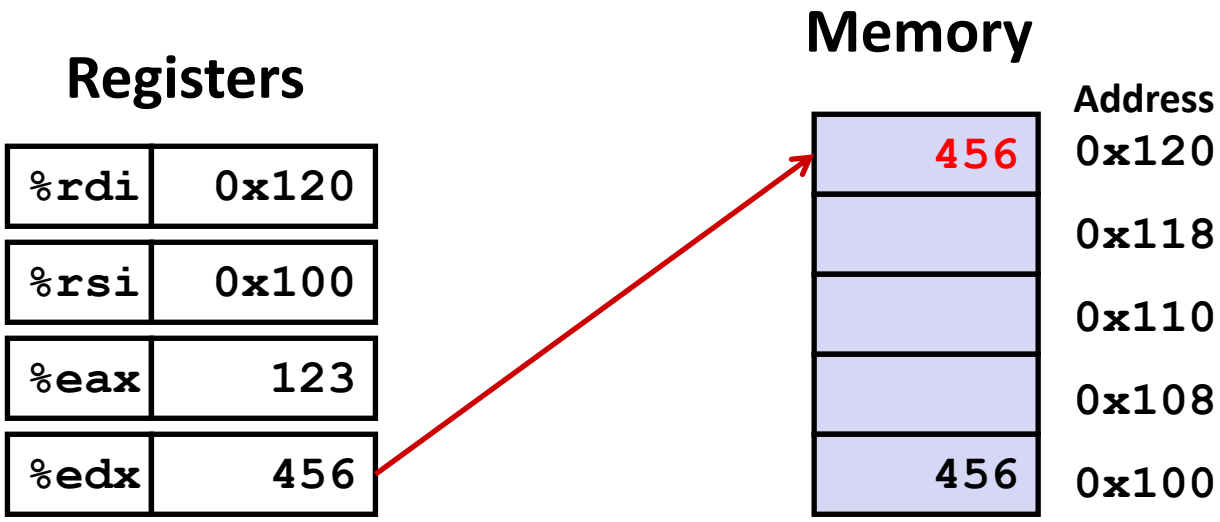

```
swap:
    movl    (%rdi), %eax   # t0 = *xp
    movl    (%rsi), %edx   # t1 = *yp
    movl    %edx, (%rdi)   # *xp = t1
    movl    %eax, (%rsi)   # *yp = t0
    ret
```

Hiểu hàm `Swap()` (x86_64)



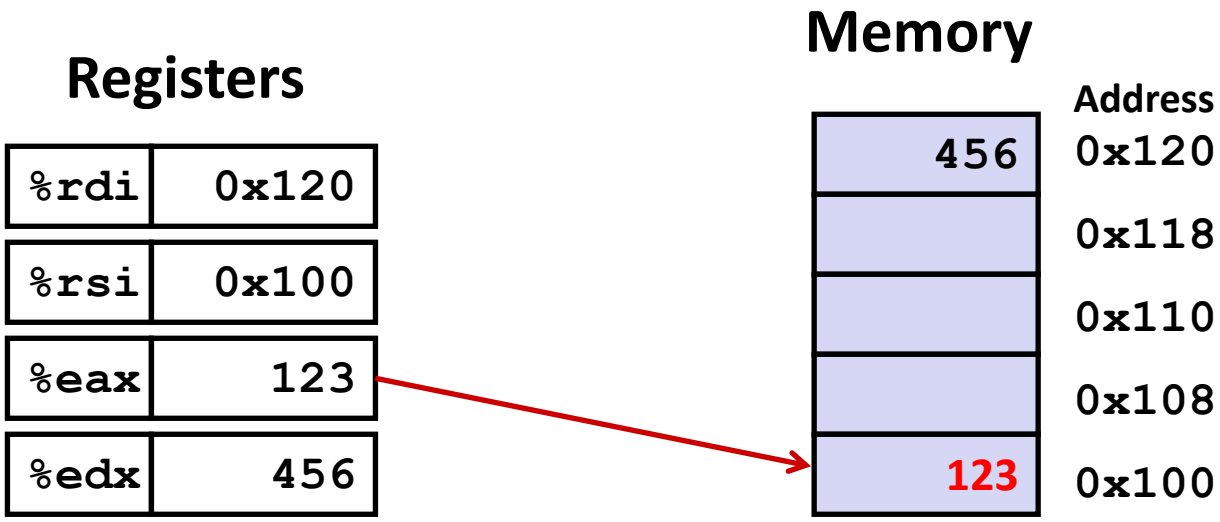

```
swap:
    movl    (%rdi), %eax   # t0 = *xp
    movl    (%rsi), %edx   # t1 = *yp
    movl    %edx, (%rdi)   # *xp = t1
    movl    %eax, (%rsi)   # *yp = t0
    ret
```

Hiểu hàm `Swap()` (x86_64)




```
swap:
    movl    (%rdi), %eax   # t0 = *xp
    movl    (%rsi), %edx   # t1 = *yp
    movl    %edx, (%rdi)   # *xp = t1
    movl    %eax, (%rsi)   # *yp = t0
    ret
```

Hiểu hàm `Swap()` (x86_64)




```
swap:
    movl    (%rdi), %eax   # t0 = *xp
    movl    (%rsi), %edx   # t1 = *yp
    movl    %edx, (%rdi)   # *xp = t1
    movl    %eax, (%rsi)   # *yp = t0
    ret
```

Lệnh tính toán địa chỉ: `leal`

- **`leal`** *`Source, Dest`*
  - `Source` là biểu thức tính toán địa chỉ
  - Gán `Dest` thành địa chỉ được tính toán bằng biểu thức trên

- **Tác dụng**
  - Tính toán địa chỉ ô nhớ mà ***không truy xuất đến ô nhớ***
    - Ví dụ, trường hợp **`p = &x[i];`**
  - Tính toán biểu thức toán học có dạng ***x + k*i + d***
    - i = 1, 2, 4, hoặc 8

- **Ví dụ**

```
int mul12(int x)
{
  return x*12;
}
```

Chuyển sang assembly bằng compiler:

```
leal (%eax,%eax,2), %eax # t <- x+x*2
sall $2, %eax            # return t<<2
```

lea vs mov: Ví dụ

Registers	
%rax	?
%rbx	?
%rcx	0x4
%rdx	0x100
%rdi	?
%rsi	?

Memory	Word Address
0x400	0x120
0xF	0x118
0x8	0x110
0x10	0x108
0x1	0x100

```
leaq (%rdx,%rcx,4), %rax
movq (%rdx,%rcx,4), %rbx
leaq (%rdx), %rdi
movq (%rdx), %rsi
```

Dùng `lea` để tính toán biểu thức

- Giả sử ta có **`%eax = x, %ecx = y`**. Các lệnh sau tính toán các biểu thức gì?

Lệnh	**Biểu thức kết quả**
`leal 6(%eax), %edx`	
`leal (%eax,%ecx), %edx`	
`leal 0xA(,%ecx,4), %edx`	
`leal (%ecx, %eax, 2), %edx`	

Nội dung

- Sơ lược lịch sử các bộ xử lý và kiến trúc Intel
- C, assembly, mã máy
- Cơ bản về Assembly: Registers, operands, move
- **Các lệnh toán học và logic**

Tổng quát về lệnh assembly AT&T

- **Định dạng**

  `opcode` *`source, dest`*

- **Với mọi câu lệnh, `dest`** không bao giờ là hằng số!
- **Không câu lệnh nào** hỗ trợ 2 toán hạng đều là ô nhớ
- Sau mỗi câu lệnh mov hay toán học, thanh ghi/ô nhớ ở vị trí **`dest`** sẽ bị thay đổi giá trị
- Khi có toán hạng ô nhớ, ngoại trừ lệnh **`lea`**, tất cả các lệnh khác đều thực hiện truy xuất giá trị của ô nhớ đó (để đọc hoặc ghi dữ liệu).
- **Suffix** ảnh hưởng đến mọi câu lệnh:
  - addl, addw,…
  - Trường hợp tổng quát nhất là không sử dụng suffix

Một số phép tính toán học (1)

- **Các Instructions với 2 toán hạng:**

Định dạng		*Phép tính*	
`addl`	*Src,Dest*	Dest = Dest + Src	
`subl`	*Src,Dest*	Dest = Dest – Src	
`imull`	*Src,Dest*	Dest = Dest * Src	
`sall`	*Src,Dest*	Dest = Dest << Src	*Cũng được gọi là shll*
`sarl`	*Src,Dest*	Dest = Dest >> Src	*Arithmetic (shift phải toán học)*
`shrl`	*Src,Dest*	Dest = Dest >> Src	*Logical (shift phải luận lý)*
`xorl`	*Src,Dest*	Dest = Dest ^ Src	
`andl`	*Src,Dest*	Dest = Dest & Src	
`orl`	*Src,Dest*	Dest = Dest | Src	

- **Cẩn thận với thứ tự của các toán hạng!**
- **Không có khác biệt giữa signed và unsigned int**

Một số phép tính toán học (2)

- **Các Instructions với 1 toán hạng**

`incl`	*Dest*	*Dest = Dest + 1*
`decl`	*Dest*	*Dest = Dest – 1*
`negl`	*Dest*	*Dest = – Dest*
`notl`	*Dest*	*Dest = ~Dest*

- **Tham khảo thêm các instruction trong giáo trình**

Biểu thức toán học: Ví dụ 1 (IA32)

```
int arith(int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```

```
arith:
  pushl  %ebp
  movl   %esp, %ebp

  movl   8(%ebp), %ecx
  movl   12(%ebp), %edx
  leal   (%edx,%edx,2), %eax
  sall   $4, %eax
  leal   4(%ecx,%eax), %eax
  addl   %ecx, %edx
  addl   16(%ebp), %edx
  imull  %edx, %eax

  popl   %ebp
  ret
```

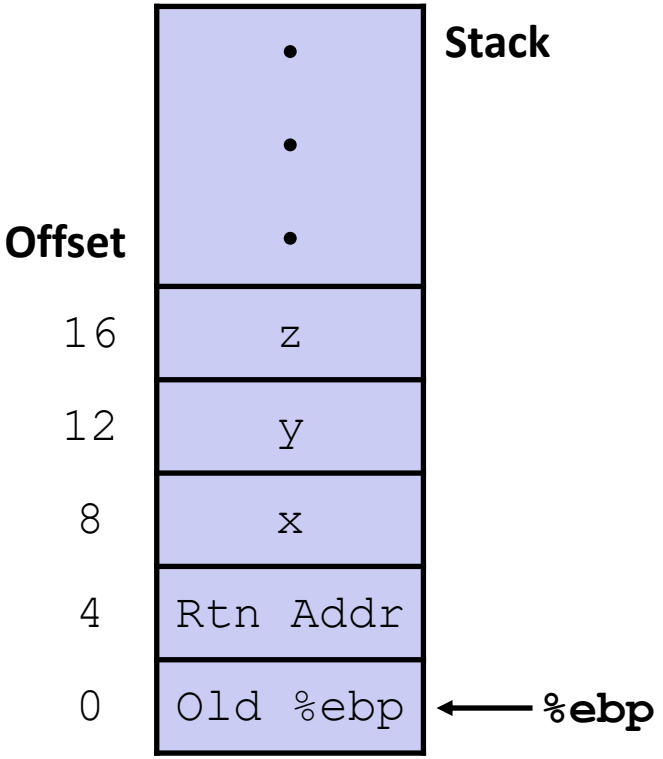
Set Up

Body

Finish

Biểu thức toán học: Ví dụ 1 (IA32)

```
int arith(int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```




```
movl    8(%ebp), %ecx          # ecx = x
movl    12(%ebp), %edx         # edx = y
leal    (%edx,%edx,2), %eax    # eax = y*3
sall    $4, %eax               # eax *= 16 (t4)
leal    4(%ecx,%eax), %eax     # eax = t4 +x+4 (t5)
addl    %ecx, %edx             # edx = x+y (t1)
addl    16(%ebp), %edx         # edx += z (t2)
imull   %edx, %eax             # eax = t2 * t5 (rval)
```

Biểu thức toán học: Ví dụ 1 (x86_64)

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;          (1)
  long t2 = z+t1;         (2)
  long t3 = x+4;          (3)
  long t4 = y * 48;       (4)
  long t5 = t3 + t4;      (5)
  long rval = t2 * t5;    (6)
  return rval;            (7)
}
```

```
%rdi lưu x, %rsi lưu y, %rdx lưu z
arith:
  leaq    (%rdi,%rsi), %rax
  addq    %rdx, %rax
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx
  leaq    4(%rdi,%rdx), %rcx
  imulq   %rcx, %rax
  ret
```

Các instruction cần lưu ý

- **`leaq`**: tính toán địa chỉ
- **`salq`**: shift trái
- **`imulq`**: phép nhân

Thanh ghi	Tác dụng
`%rdi`	Argument **`x`**
`%rsi`	Argument **`y`**
`%rdx`	Argument **`z`**
`%rax`	**`t1`**, **`t2`**, **`rval`**
`%rdx`	**`t4`**
`%rcx`	**`t5`**

Biểu thức toán học: Ví dụ 1 (x86_64)

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;          (1)
  long t2 = z+t1;         (2)
  long t3 = x+4;          (3)
  long t4 = y * 48;       (4)
  long t5 = t3 + t4;      (5)
  long rval = t2 * t5;    (6)
  return rval;            (7)
}
```

```
%rdi lưu x, %rsi lưu y, %rdx lưu z
arith:
  leaq    (%rdi,%rsi), %rax        (1)
  addq    %rdx, %rax               (2)
  leaq    (%rsi,%rsi,2), %rdx      (4)
  salq    $4, %rdx                 (4)
  leaq    4(%rdi,%rdx), %rcx       (3,5)
  imulq   %rcx, %rax               (6)
  ret                              (7)
```

Các instruction cần lưu ý

- `leaq`: tính toán địa chỉ
- `salq`: shift trái
- `imulq`: phép nhân

Thanh ghi	Tác dụng
`%rdi`	Argument `x`
`%rsi`	Argument `y`
`%rdx`	Argument `z`
`%rax`	`t1`, `t2`, `rval`
`%rdx`	`t4`
`%rcx`	`t5`

Extra 2: Khác biệt giữa các định dạng AT&T vs Intel

- **Khác biệt giữa 2 định dạng assembly: AT&T vs Intel**

	AT&T	Intel
Thứ tự toán hạng	`movl source, dest`	`mov dest, source`
Thanh ghi	Có % trước tên thanh ghi `%eax`	Không có prefix trước tên thanh ghi `eax`
Lệnh mov	Có suffix `movl, movlq, movb`...	Không có suffix `mov`
Địa chỉ ô nhớ	`8(%ebp)`	`[ebp + 8]`
Có thể thấy ở đâu?	gcc: option `-masm=att` (mặc định) objdump: option `-M att` (mặc định)	• IDA Pro • gcc: option `-masm=intel` • objdump: option `-M intel`

Nội dung

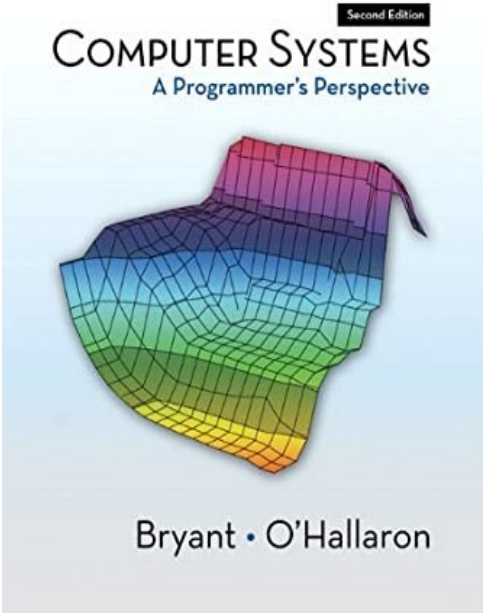
- **Các chủ đề chính:**
  1) Biểu diễn các kiểu dữ liệu và các phép tính toán bit
  2) Ngôn ngữ assembly cơ bản
  3) Điều khiển luồng trong C với assembly
  4) Các thủ tục/hàm (procedure) trong C ở mức assembly
  5) Biểu diễn mảng, cấu trúc dữ liệu trong C
  6) Một số topic ATTT: reverse engineering, 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 :)