

LẬP TRÌNH HỆ THỐNG



nc.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: Điều khiển luồng



Làm thế nào biểu diễn trong assembly?

Code C

```
if (x>y)
```

```
    result = x + y;
```

```
else
```

```
    result = x - y;
```

Assembly code

// lệnh **if** để kiểm tra điều kiện trong assembly??

→ addl %ebx, %eax

// **else**??

→ subl %ebx, %eax



```
for(i=0; i<8; i++)
```

```
    result += i;
```

// **for** lệnh??

and what else??



Ví dụ if/else trong assembly

Code C

```
int result;  
if (x < y)  
    result = y-x;  
else  
    result = x-y;  
return result;
```

Assembly code

x at %ebp+8, y at %ebp+12

- 1 movl 8(%ebp), %edx
- 2 movl 12(%ebp), %eax
- 3 cmpl %eax, %edx
- 4 jge .L2
- 5 subl %edx, %eax
- 6 jmp .L3
- 7 .L2:
- 8 subl %eax, %edx
- 9 movl %edx, %eax
- 10 .L3:

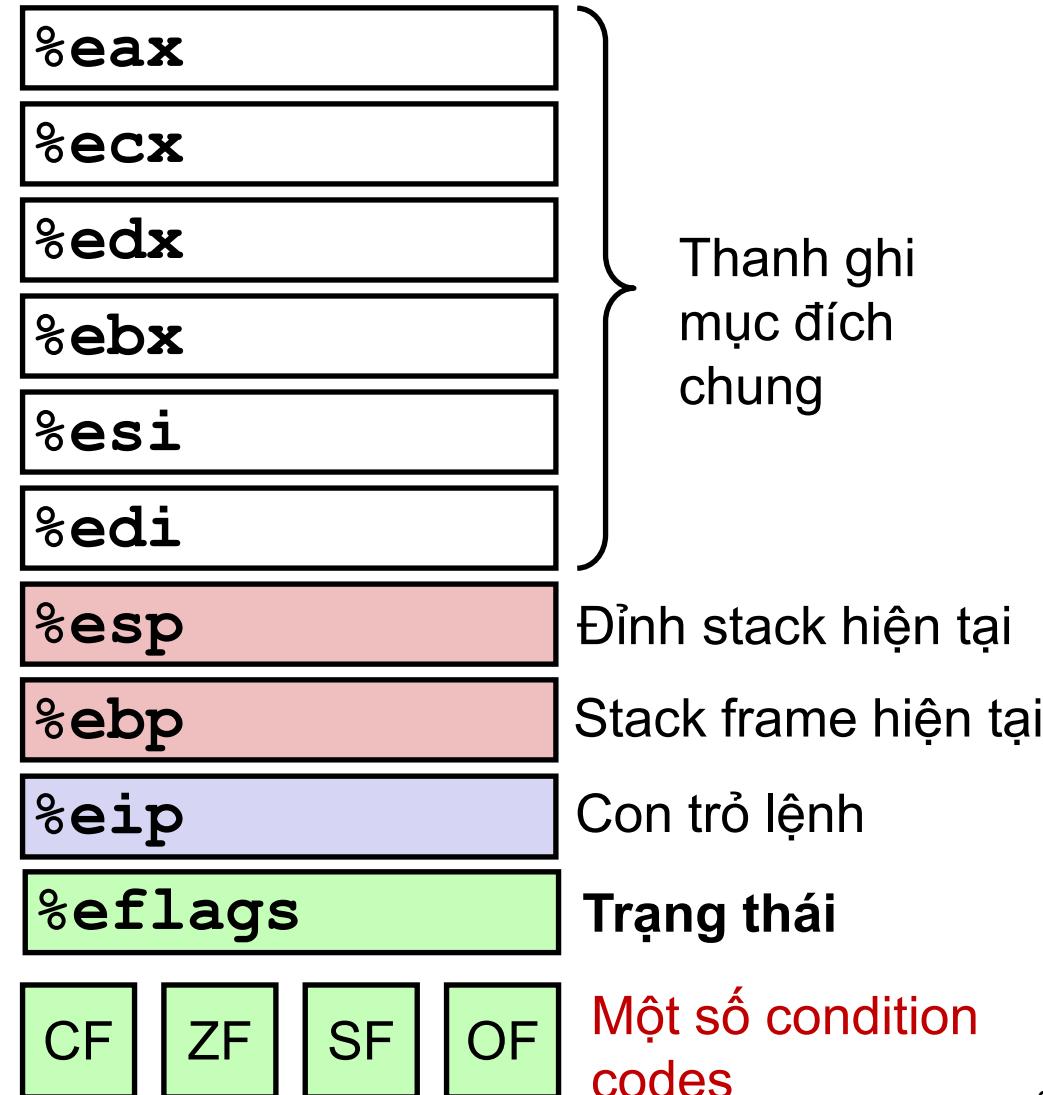
Nội dung

- Điều khiển luồng: Condition codes
- Rẽ nhánh có điều kiện
- Vòng lặp

Trạng thái bộ xử lý (IA32)

■ Các thông tin về chương trình hiện đang thực thi

- Dữ liệu tạm thời (`%eax`, ...)
- Vị trí của stack trong lúc chạy (`%esp`, `%ebp`)
- Vị trí kiểm soát câu lệnh được thực thi (`%eip`, ...)
- Trạng thái của một số test gần nhất (CF, ZF, SF, OF)



Trạng thái bộ xử lý (x86-64)?

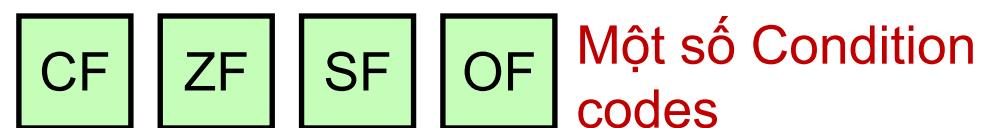
■ Các thông tin về chương trình hiện đang thực thi

- Dữ liệu tạm thời (`%rax`, ...)
- Vị trí của stack trong lúc chạy (`%rsp`)
- Vị trí kiểm soát câu lệnh được thực thi (`%rip`, ...)
- Trạng thái của một số test gần nhất (CF, ZF, SF, OF)

Đỉnh stack hiện tại

Registers

<code>%rax</code>	<code>%r8</code>
<code>%rbx</code>	<code>%r9</code>
<code>%rcx</code>	<code>%r10</code>
<code>%rdx</code>	<code>%r11</code>
<code>%rsi</code>	<code>%r12</code>
<code>%rdi</code>	<code>%r13</code>
<code>%rsp</code>	<code>%r14</code>
<code>%rbp</code>	<code>%r15</code>
<code>%rip</code>	Con trả lệnh
<code>%rflags</code>	Trạng thái



Condition Codes

- Các “thanh ghi” 1-bit (0 hoặc 1)
 - **CF** - Carry Flag (for unsigned) Được bật khi xảy ra tràn số không dấu
 - **SF** - Sign Flag (for signed) Được bật khi kết quả là số âm
 - **ZF** - Zero Flag Được bật khi kết quả là số 0
 - **OF** Overflow Flag (for signed) Được bật khi xảy ra tràn số có dấu
- Chứa trong thanh ghi **%eflag** / **%rfflag**
- **Gán giá trị cho các condition codes**
 - Gán ngầm: qua các phép tính toán học
 - Gán tường minh: các lệnh so sánh, test
- **Condition codes** có thể được dùng để:
 - Thực thi các đoạn lệnh dựa trên các điều kiện
 - Gán giá trị dựa trên điều kiện
 - Chuyển dữ liệu dựa trên các điều kiện

Gán giá trị Condition Codes (1)

Gán ngầm qua phép tính toán học

■ Các “thanh ghi” 1-bit

- CF Carry Flag (for unsigned) SF Sign Flag (for signed)
- ZF Zero Flag OF Overflow Flag (for signed)

■ Được gán ngầm bằng các phép tính toán học

- Có thể được xem là tác dụng phụ (side affect) của các phép toán này

Ví dụ: **addq** Src, Dest $\leftrightarrow t = a+b$

CF được gán nếu có nhón bit ở most significant bit (tràn số không dấu)

ZF được gán nếu $t == 0$

SF được gán nếu $t < 0$ (có dấu)

OF được gán nếu tràn số bù 2 (có dấu)

$(a>0 \&\& b>0 \&\& t<0) \parallel (a<0 \&\& b<0 \&\& t>=0)$

■ Không được gán giá trị bằng instruction leal !

Gán giá trị Condition Codes (2)

Gán tương minh qua phép so sánh

■ Giá trị được gán tương minh bằng các lệnh So sánh

- **cmp1** Src2, Src1
- **cmp1** b, a tương tự như tính $a - b$ mà không cần lưu lại kết quả tính

- CF được gán nếu có nhón bit ở most significant bit (dùng cho so sánh số không dấu)
- ZF được gán nếu $a == b$
- SF được gán nếu $(a-b) < 0$ (phép trừ có dấu - âm)
- OF được gán nếu tràn số bù 2 (có dấu)
 $(a>0 \&\& b<0 \&\& (a-b)<0) \parallel (a<0 \&\& b>0 \&\& (a-b)>0)$

Gán giá trị Condition Codes (3)

Gán tương minh qua lệnh test

■ Gán tương minh bằng lệnh test

- **testl Src2, Src1**
 - **testl b, a** tương tự tính $a \& b$ mà không lưu lại kết quả tính
- Gán giá trị các condition codes dựa trên giá trị của **Src1 & Src2**
- Hữu ích khi có 1 toán hạng đóng vai trò là mask
- ZF được gán khi $a \& b == 0$
- SF được gán khi $a \& b < 0$



Sử dụng Condition Codes

Điều khiển luồng dựa trên điều kiện

■ Gán giá trị dựa trên điều kiện

- setX

■ Chuyển dữ liệu dựa trên điều kiện

- Conditional move

■ Rẽ nhánh có điều kiện

- Instruction rẽ nhánh: jX
- If/else
- Vòng lặp (loop)

Nội dung

- Điều khiển luồng: Condition codes
- Rẽ nhánh có điều kiện
- Vòng lặp

Các câu lệnh jump

■ Các lệnh rẽ nhánh: jX

- **jX** label
- Nhảy đến đoạn mã khác (được gán nhãn label) để thực thi dựa trên các condition codes.

jX	Điều kiện	Mô tả
jmp	1	Nhảy không điều kiện
je	ZF	Equal / Zero
jne	$\sim ZF$	Not Equal / Not Zero
js	SF	Negative
jns	$\sim SF$	Nonnegative
jg	$\sim (SF^OF) \ \& \sim ZF$	Greater (Signed)
jge	$\sim (SF^OF)$	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	$(SF^OF) \ ZF$	Less or Equal (Signed)
ja	$\sim CF \ \& \ \sim ZF$	Above (unsigned)
jb	CF	Below (unsigned)

Các câu lệnh jump kết hợp với so sánh

- Các lệnh jump thường kết hợp với các lệnh so sánh/test
 - Kết quả của lệnh so sánh/test quyết định có thực hiện jump hay không.

`cmp1 src2, src1`

`jX label`

jX	Điều kiện nhảy
je	$\text{src1} == \text{src2}$
jne	$\text{src1} != \text{src2}$
jg	$\text{src1} > \text{src2}$
jge	$\text{src1} \geq \text{src2}$
jl	$\text{src1} < \text{src2}$
jle	$\text{src1} \leq \text{src2}$

Sử dụng lệnh jX nào?

- Cho các giá trị: $%eax = x$ $%ebx = y$ $%ecx = z$
- Một đoạn mã gán nhãn .L1

Điều kiện nhảy đến .L1	Tổ hợp lệnh cmpl/test và jx
$x == y$	
$y != z$	
$z > x$	
$x < 0$	
$y == 0$	
z	
true	

Rẽ nhánh có điều kiện – Ví dụ

```
int absdiff(int x, int y)
{
    int result;
    if (x < y)
        result = y-x;
    else
        result = x-y;
    return result;
}
```

x at %ebp+8, y at %ebp+12

1	movl	8(%ebp), %edx	//x
2	movl	12(%ebp), %eax	//y
3	cmpl	%eax, %edx	
4	jge	.L2	
5	subl	%edx, %eax	
6	jmp	.L3	
7	.L2: # x >= y		
8	subl	%eax, %edx	
9	movl	%edx, %eax	
10	.L3:		

Sử dụng điều kiện nhảy là điều kiện **false** của if

Rẽ nhánh có điều kiện – Ví dụ (tt)

```
int absdiff(int x, int y)
{
    int result;
    if (x < y)
        result = y-x;
    else
        result = x-y;
    return result;
}
```

x at %ebp+8, y at %ebp+12

1. movl 8(%ebp), %edx //x
2. movl 12(%ebp), %eax //y
3. cmpl %eax, %edx
4. jl .L2
5. subl %eax, %edx
6. movl %edx, %eax
7. jmp .L3
8. .L2:
9. subl %edx, %eax
- 10..L3

Sử dụng điều kiện nhảy là điều kiện **true** của if

Chuyển mã rẽ nhánh có điều kiện

Ví dụ 1: if/else - Từ assembly sang C

Assembly code

```
x at ebp+8, y at ebp+12, sum at eax
1.      movl 8(%ebp),%ecx    //x
2.      movl 12(%ebp),%ebx   //y
3.      cmpl $0,%ecx
4.      jle .L2
5.      leal (%ecx,%ebx),%eax
6.      jmp .L3
7. .L2:
8.      movl %ebx,%eax
9.      subl %ecx,%eax
10..L3:
```

Dự đoán Code C?

- Điều kiện **true** của **if**:
- Đoạn code tương ứng với điều kiện **true** của **if**?
- Đoạn code tương ứng với **false** của **if**?

Chuyển mã rẽ nhánh có điều kiện

Từ C sang assembly: Dạng Goto

- C hỗ trợ goto statement → bản chất giống lệnh jmp
- Nhảy đến vị trí xác định bởi label

```
int absdiff(int x, int y)
{
    int result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
int absdiff_j(int x, int y)
{
    int result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    goto Done;
Else:
    result = y-x;
Done:
    return result;
}
```

Chuyển mã rẽ nhánh có điều kiện

Từ C sang assembly: Phương pháp chung

C code

```
if (test-expr)
    then-statement;
else
    else-statement;
```

Dạng Goto (*thực hiện tính toán và luồng
tương tự mã assembly*)

```
nt = !test-expr;
if (nt)
    goto False;
then-statement;
goto Done;
False:
else-statement;
Done:
```

Assembly code

```
...
<instructions to check nt>
jX False
<instructions of then-statement>
jmp Done
False:
<instruction of else-statement>
Done:
...  
...
```

Chuyển mã rẽ nhánh có điều kiện

Từ C sang assembly: Phương pháp chung

C code

```
if (a>b)
    result = a^b;
else
    result = a&b;
```

Dạng Goto (*thực hiện tính toán và luồng tương tự mã assembly*)

```
nt = a <= b;
if (nt)
    goto False;
result = a^b;
goto Done;

False:
result = a&b;

Done:
```

Assembly code

```
%eax = a
%ebx = b
```

```
...
cmpl %ebx, %eax
jle False
xorl %ebx, %eax
jmp Done

False:
andl %ebx, %eax
```

Done:

```
// return value in %eax
```

Chuyển mã rẽ nhánh có điều kiện

Ví dụ 2: if/else - Từ C sang assembly

```
1 int func(int x, int y)
2 {
3     int result = 0;
4     if (x > 2)
5         result = x + y;
6     else
7         result = x - y;
8     return result;
9 }
```

```
// x at %ebp+8, y at %ebp+12
1.    movl    $0, -4(%ebp) //result
2.    cmpl    $2, 8(%ebp)
3.    jle     .L2
4.    movl    8(%ebp), %eax //x
5.    addl    12(%ebp), %eax //x+y
6.    movl    %eax, -4(%ebp)
7.    jmp     .L3
8. .L2:
9.    movl    8(%ebp), %eax //x
10.   subl   12(%ebp), %eax //x-y
11.   movl    %eax, -4(%ebp)
12. .L3:
```

Dạng Goto (*thực hiện tính toán và luồng tương tự mã assembly*)

```
1. int func(int x, int y)
2. {
3.     int result = 0;
4.     not_true = x <= 2 ;
5.     if (not_true)
6.         goto False;
7.     result = x + y;
8.     goto Done;
9. False:
10.    result = x - y;
11. Done:
12. }
```

Chuyển mã rẽ nhánh có điều kiện

Ví dụ 3: Nested if - Từ C sang assembly

C code

```
1 int func(int x, int y)
2 {
3     int result = 0;
4     if (x)    if #1
5     {
6         if (y > 1) if #2
7             result = x + y;
8         else
9             result = x * y;
10    }
11 }
```

Goto code

```
1. int func(int x, int y)
2. {
3.     int result = 0;
4.     notif1 = x == 0 ;
5.     if (notif1)
6.         goto F1;
7.     notif2 = y <= 1;
8.     if (notif2)
9.         goto F2;
10.    result = x + y;
11.    goto Done;
12.    F2:
13.        result = x * y;
14.    F1:
15.    Done:
16. }
```

Chuyển mã rẽ nhánh có điều kiện

Ví dụ 3: Nested if - Từ C sang assembly

C code

```
1 int func(int x, int y)
2 {
3     int result = 0;
4     if (x)    if #1
5     {
6         if (y > 1) if #2
7         result = x + y;
8     else
9         result = x * y;
10 }
11 }
```

Assembly code

```
// x at %ebp+8, y at %ebp+12
1.      movl    $0, -4(%ebp)  #result
2.      cmpl    $0, 8(%ebp)
3.      je      .L2          # if 1
4.      cmpl    $1, 12(%ebp)
5.      jle     .L3          # if 2
6.      movl    8(%ebp), %eax
7.      addl    12(%ebp), %eax
8.      movl    %eax, -4(%ebp)
9.      jmp     .L4
10. .L3: //false của if #2
11.      movl    8(%ebp), %eax
12.      imull   12(%ebp), %eax
13.      movl    %eax, -4(%ebp)
14. .L2: //false của if #1
15. .L4:
...
```

Chuyển mã rẽ nhánh có điều kiện

Ví dụ 4: if/else - Từ C sang assembly

C code

```
1 int func(int x, int y)
2 {
3     int result = 0;
4     if (y && x != y)
5         result = x + y;
6     return result;
7 }
```

Viết Assembly code tương ứng?

Biết giá trị trả về sẽ lưu trong thanh ghi %eax

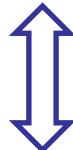
```
// x at %ebp+8, y at %ebp+12
1.    movl    $0, -4(%ebp) //result
2.    cmpl    $0, 12(%ebp)
3.    je      .L2
4.    movl    12(%ebp), %edx
5.    cmpl    8(%ebp), %edx
6.    je      .L2
7.    addl    8(%ebp), %edx
8.    movl    %edx, -4(%ebp)
9. .L2:
10.   movl   -4(%ebp), %eax
```

Chuyển mã rẽ nhánh có điều kiện

Ví dụ 5: if/else - Từ C sang assembly (C1)

C code

```
1. int arith(int a, int b, int c)
2. {
3.     int sum = 0;
4.     if(c < 0 || a == b)
5.         sum = (a & b)^c;
6.     return sum;
7. }
```



```
1. int arith(int a, int b, int c)
2. {
3.     int sum = 0;
4.     if (c < 0)
5.         sum = (a & b)^c;
6.     else if (a == b)
7.         sum = (a & b)^c;
8.     return sum;
9. }
```

Viết **Code Goto** và **Assembly** tương ứng?

Biết giá trị trả về sẽ lưu trong thanh ghi %eax

```
// a at %ebp+8, b at %ebp+12, c at %ebp+16
1.     movl    $0, -4(%ebp) //sum
2.     cmpl    $0, 16(%ebp)
3.     jge     .L2
4. .L1:
5.     movl    8(%ebp), %edx
6.     andl    12(%ebp), %edx
7.     xorl    16(%ebp), %edx
8.     movl    %edx, -4(%ebp)
9.     jmp     .L3
10..L2: //so sánh tiếp a và b
11.     movl    8(%ebp), %edx
12.     cmpl    12(%ebp), %edx
13.     jne     .L3
14.     jmp     .L1
15..L3:
16.     movl    -4(%ebp), %eax
```

Chuyển mã rẽ nhánh có điều kiện

Ví dụ 5: if/else - Từ C sang assembly (C2)

C code

```
1. int arith(int a, int b, int c)
2. {
3.     int sum = 0;
4.     if(c < 0 || a == b)
5.         sum = (a & b)^c;
6.     return sum;
7. }
```



```
1. int arith(int a, int b, int c)
2. {
3.     int sum = 0;
4.     if (c < 0)
5.         sum = (a & b)^c;
6.     else if (a == b)
7.         sum = (a & b)^c;
8.     return sum;
9. }
```

Viết Code Goto và Assembly tương ứng?

Biết giá trị trả về sẽ lưu trong thanh ghi %eax

```
// a at %ebp+8, b at %ebp+12, c at %ebp+16
1.     movl    $0, -4(%ebp) //sum
2.     cmpl    $0, 16(%ebp)
3.     jl      .L2
4.     movl    8(%ebp), %edx
5.     cmpl    12(%ebp), %edx
6.     jne     .L3
7. .L2: //tính sum
8.     movl    8(%ebp), %edx
9.     andl    12(%ebp), %edx
10.    xorl   16(%ebp), %edx
11.    movl    %edx, -4(%ebp)
12. .L3:
13.    movl    -4(%ebp), %eax
```

Nội dung

- Điều khiển luồng: Condition codes
- Rẽ nhánh có điều kiện
- **Vòng lặp**

Vòng lặp – Ví dụ

Code C

```
int i, sum = 0;  
for (i = 0; i < 10; i++)  
    sum += i;  
  
int i = 0, sum = 0;  
while (i < 10)  
{  
    sum += i;  
    i++;  
}
```

Code assembly

```
1.      movl $0, -4(%ebp)    # i  
2.      movl $0, -8(%ebp)    # result  
3.      jmp .test  
4. .Loop:  
5.      movl -4(%ebp), %eax  
6.      addl %eax, -8(%ebp)  
7.      incl -4(%ebp)  
8. .test:  
9.      cmpl $10, -4(%ebp)  
10.     jl .Loop  
11. // outside of loop
```

Vòng lặp (loops)

■ Vòng lặp trong C

- do-while
- while
- for

■ Vòng lặp ở mức máy tính

- Không có instruction hỗ trợ trực tiếp
- Là tổ hợp các phép **kiểm tra** và **jump có điều kiện**
- Dựa trên dạng vòng lặp **do-while**
 - Các dạng vòng lặp khác trong C sẽ được chuyển sang dạng này sau đó biên dịch thành mã máy

Vòng lặp Do-While

C Code

```
int pcount_do(unsigned int x)
{
    int result = 0;
    do {
        result += x & 0x1;
        x >>= 1;
    } while (x);
    return result;
}
```

Goto Version

```
int pcount_goto(unsigned int x)
{
    int result = 0;
loop:
    result += x & 0x1;
    x >>= 1;
    if(x) goto loop;
    return result;
}
```

- Đếm số bit 1 có trong tham số x (“popcount”)
- Sử dụng rẽ nhánh có điều kiện để tiếp tục hoặc thoát khỏi vòng lặp

Biên dịch vòng lặp Do-While

Goto Version

```
int pcount_goto(unsigned int x)
{
    int result = 0;
loop:
    result += x & 0x1;
    x >>= 1;
    if(x) goto loop;
    return result;
}
```

■ Registers:

%edx	x
%ecx	result

movl	\$0, %ecx	#	result = 0
.L2:	# loop:		
movl	%edx, %eax		
andl	\$1, %eax	#	t = x & 1
addl	%eax, %ecx	#	result += t
shrl	%edx	#	x >>= 1
cmpl	\$0, %edx	#	compare x and 0
jne	.L2	#	If !0, goto loop

Chuyển mã vòng lặp Do-while: Tổng quát

C Code

```
do  
    Body  
    while ( Test );
```

Goto Version

```
loop:  
    Body  
    if ( Test )  
        goto loop
```

■ Body:

```
{  
    Statement1;  
    Statement2;  
    ...  
    Statementn;  
}
```

Chuyển mã vòng lặp While

- Khác biệt giữa **do-while** và **while**?
 - **Do-while**: thực hiện body ít nhất 1 lần
 - **While**: có thể không thực hiện
- **Chuyển While sang Do-while**
 - Cần đảm bảo thực hiện kiểm tra điều kiện trước tiên!

Chuyển mã vòng lặp While – Dạng 1

- Chuyển mã dạng “nhảy vào giữa” → kiểm tra điều kiện trước
- Sử dụng với option -Og

While version

```
while (Test)
    Body
```



Goto Version

```
goto test;
loop:
Body
test:
if (Test)
    goto loop;
done:
```

Chuyển mã vòng lặp While – Dạng 1 – Ví dụ

C Code

```
int pcount_while  
(unsigned int x)  
{  
    int result = 0;  
    while (x) {  
        result += x & 0x1;  
        x >>= 1;  
    }  
    return result;  
}
```

Dạng “Nhảy vào giữa”

```
int pcount_goto_jtm  
(unsigned int x)  
{  
    int result = 0;  
    goto test;  
loop:  
    result += x & 0x1;  
    x >>= 1;  
test:  
    if(x) goto loop;  
    return result;  
}
```

- Goto đầu tiên bắt đầu vòng lặp tại test để kiểm tra điều kiện trước

Chuyển mã vòng lặp While – Dạng 2

While version

```
while ( Test)  
    Body
```



Do-While Version

```
if ( ! Test)  
    goto done;  
do  
    Body  
    while( Test);  
done:
```

- Chuyển sang dạng “Do-while”
- Sử dụng với option -O1

Goto Version

```
if ( ! Test)  
    goto done;  
loop:  
    Body  
    if ( Test)  
        goto loop;  
done:
```



Chuyển mã vòng lặp While – Dạng 2 – Ví dụ

C Code

```
int pcount_while  
(unsigned int x)  
{  
    int result = 0;  
    while (x) {  
        result += x & 0x1;  
        x >>= 1;  
    }  
    return result;  
}
```

Dạng Do-While

```
int pcount_goto_dw  
(unsigned int x)  
{  
    int result = 0;  
    if (!x) goto done;  
loop:  
    result += x & 0x1;  
    x >>= 1;  
    if(x) goto loop;  
done:  
    return result;  
}
```

- Điều kiện ban đầu được kiểm tra trước khi vào vòng lặp

Dạng vòng lặp For

for (*Init*; *Test* ; *Update*)

Body

```
#define WSIZE 8*sizeof(int)
int pcount_for(unsigned int x)
{
    size_t i;
    int result = 0;
    for (i = 0; i < WSIZE; i++)
    {
        unsigned bit =
            (x >> i) & 0x1;
        result += bit;
    }
    return result;
}
```

Khởi tạo

i = 0

Kiểm tra

i < WSIZE

Cập nhật

i++

Body

```
{
    unsigned bit =
        (x >> i) & 0x1;
    result += bit;
}
```

Vòng lặp For → Vòng lặp While

For Version

```
for (Init; Test; Update)
```

Body



While Version

```
Init;
```

```
while (Test) {
```

Body

Update;

```
}
```

Chuyển vòng lặp For sang While

Khởi tạo

```
i = 0
```

Kiểm tra

```
i < WSIZE
```

Cập nhật

```
i++
```

Body

```
{  
    unsigned bit =  
        (x >> i) & 0x1;  
    result += bit;  
}
```

```
int pcount_for_while(unsigned int x)  
{  
    size_t i;  
    long result = 0;  
    i = 0;  
    while (i < WSIZE)  
    {  
        unsigned bit =  
            (x >> i) & 0x1;  
        result += bit;  
        i++;  
    }  
    return result;  
}
```

Chuyển vòng lặp For sang Do-While

C Code

```
int pcount_for(unsigned int x)
{
    size_t i;
    int result = 0;
    for (i = 0; i < WSIZE; i++)
    {
        unsigned bit =
            (x >> i) & 0x1;
        result += bit;
    }
    return result;
}
```

Goto Version

```
int pcount_for_goto_dw
(unsigned int x) {
    size_t i;
    int result = 0;
    i = 0;
    if (!(i < WSIZE))
        goto done;
loop:
{
    unsigned bit =
        (x >> i) & 0x1;
    result += bit;
}
i++; Update
if (i < WSIZE)
    goto loop; Test
done:
    return result;
}
```

Init

! Test

Body

Update

Test

Chuyển mã vòng lặp – Từ C sang assembly

Ví dụ

C Code

```
int func1(int a)
{
    int sum = 0;
    for (int i = 0; i < a; i+=2)
        sum += (a - i);
    return sum;
}
```

Code assembly

```
// a ở ô nhớ %ebp+8
1. ...
2. movl $0, -4(%ebp) # sum
3. movl $0, -8(%ebp) # i
4. jmp .test
5. .Loop:
6. movl 8(%ebp), %eax
7. subl -8(%ebp), %eax
8. addl %eax, -4(%ebp)
9. addl $2, -8(%ebp)
10..test:
11. movl 8(%ebp), %eax
12. cmpl -8(%ebp), %eax
13. jg .Loop
14.// return sum
```

Chuyển mã vòng lặp – Từ assembly sang C

Ví dụ 1

```
// x at %ebp+8
1. func:
2.     ...
3.     movl $0,-4(%ebp)    # count
4. .L1:
5.     addl $2,8(%ebp)
6.     incl -4(%ebp)
7.     cmpl $9,8(%ebp)
8.     jle .L1
9.     movl -4(%ebp),%eax
10.    leave
11.    ret
```

- Khởi tạo?

count = 0;

- Điều kiện kiểm tra?

x <= 9

- Body?

x += 2;

count ++;

```
int count = 0;
do{
    x += 2;
    count++;
} while (x <= 9)
```

Chuyển mã vòng lặp – Từ assembly sang C

Ví dụ 2

```
1. func:  
2.     ...  
3.     movl $0,-8(%ebp)      # count  
4.     movl $0,-4(%ebp)      # i  
5. .L2:  
6.     cmpl $19,-4(%ebp)  
7.     jg .L3  
8.     movl -4(%ebp),%eax  
9.     addl %eax,-8(%ebp)  
10.    incl -4(%ebp)  
11.    jmp .L2  
  
12..L3:  
13.    leave  
14.    ret
```

- Khởi tạo?
 $i = 0; count = 0;$
- Điều kiện kiểm tra?
 $i \leq 19$
- Cập nhật?
 $i++$
- Body?
 $count += i$

```
count = 0;  
for (i = 0; i \leq 19; i++)  
    count += i;
```

Chuyển mã vòng lặp – Từ assembly sang C

Ví dụ 3.1

```
1.         movl $0,-8(%ebp)      # count
2.         movl $0,-4(%ebp)      # i
3. .L1:
4.         cmpl $25,-4(%ebp)    } // Kiểm tra điều
5.         jge .L3             } kiện trước tiên
6.         movl -4(%ebp),%eax
7.         cmpl -8(%ebp), %eax
8.         jg .L2
9.         addl %eax,-8(%ebp)
10.        jmp .Done
11..L2:
12.        subl %eax, -8(%ebp)
13..Done:
14.        incl -4(%ebp)
15.        jmp .L1
16..L3:
17.        leave
18.        ret
```

■ Khởi tạo?

i = 0; count = 0;

■ Điều kiện kiểm tra vòng lặp?

i < 25

■ Body?

```
if (i <= count)  
    count += i;
```

else

```
    count -= i;
```

```
    i++;
```

```
int i = 0, count = 0;  
while (i < 25) {  
    if (i <= count)  
        count += i;  
    else count -= i;  
    i++;  
}
```

Chuyển mã vòng lặp – Từ assembly sang C

Ví dụ 3.2

```
1.      movl $0,-8(%ebp)    # count
2.      movl $0,-4(%ebp)    # i
3. .L1:
4.      cmpl $25,-4(%ebp)  ] // Kiểm tra điều
5.      jge  .L3           ] kiện trước tiên
6.      movl -4(%ebp),%eax
7.      cmpl -8(%ebp), %eax
8.      jg   .L2
9.      addl %eax,-8(%ebp)
10. .L2:
11.      subl %eax, -8(%ebp)
12.      incl -4(%ebp)
13.      jmp  .L1
14. .L3:
15.      leave
16.      ret
```

- Khởi tạo?
 $i = 0; count = 0;$
- Điều kiện kiểm tra vòng lặp?
 $i < 25$
- Body?

```
if (i <= count)
    count += i;
count--;
i++;
```
- Code C:

```
int i = 0, count = 0;
while (i < 25) {
    if (i <= count)
        count += i;
    count--;
    i++;
}
```

Chuyển mã vòng lặp – Từ assembly sang C

Ví dụ 4 – For (Bài tập bonus)

Cho mảng ký tự **char*** a có độ dài len

```
// địa chỉ của mảng a ở %ebp+8, len ở %ebp+12
1. ...
2.     movl    $0, -8(%ebp) # result
3.     movl    $0, -4(%ebp) # i
4.     jmp     .L2
5. .L3:
6.     movl    -4(%ebp), %edx
7.     movl    8(%ebp), %eax
8.     addl    %edx, %eax
9.     mov     (%eax), %al
10.    subl   $48, %eax
11.    addl   %eax, -8(%ebp)
12.    addl   $1, -4(%ebp)
13..L2:
14.    movl    -4(%ebp), %eax
15.    cmpl    12(%ebp), %eax
16.    jl      .L3
17.    movl    -8(%ebp), %eax #return
```

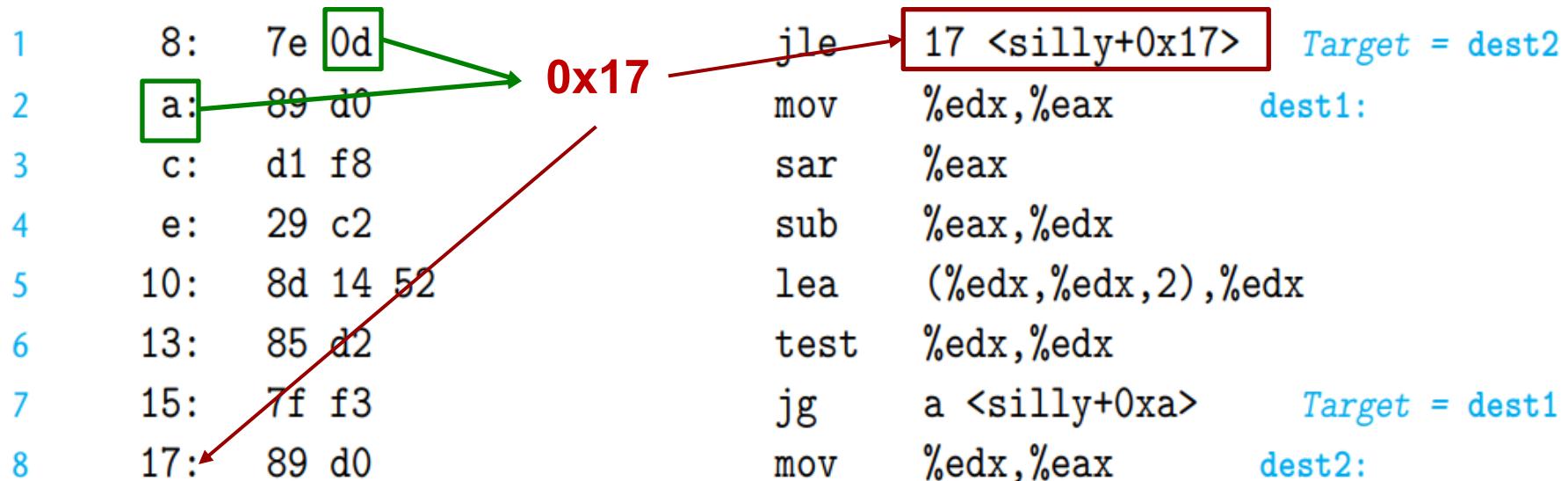
- Khởi tạo?
 - i = 0
 - result = 0
 - Điều kiện kiểm tra?
 - i >= len
 - Cập nhật?
 - i = i + 1
 - Body?
 - result += a[i] – 48;
- Viết code C tương ứng?**

Extra 1: Các câu lệnh jump - Label

- Vị trí sẽ nhảy đến của các lệnh jump trong mã assembly được biểu diễn dưới dạng các *label*.
- **Assembler** và **Linker** có thể lựa chọn 1 trong 2 cách để xác định vị trí nhảy đến:
 - *Địa chỉ tuyệt đối*: 4 (hoặc 8) bytes địa chỉ chính xác của instruction đích muốn nhảy đến.
 - *PC relative – địa chỉ tương đối*: khoảng cách tương đối giữa instruction đích và vị trí instruction liền sau lệnh jump (giá trị thanh ghi PC).

Extra 1: Các câu lệnh jump - Label

■ PC relative – địa chỉ tương đối



Extra 2: Sử dụng Condition Codes

Gán giá trị dựa trên điều kiện

■ Các instruction SetX

- **setx dest**
- Gán **byte thấp nhất (low-order byte)** của destination thành 1 hoặc 0 dựa trên 1 nhóm các condition codes.
- Không thay đổi 7 bytes còn lại

SetX	Condition	Description
sete	ZF	Equal / Zero
setne	~ZF	Not Equal / Not Zero
sets	SF	Negative
setns	~SF	Nonnegative
setg	~(SF^OF) & ~ZF	Greater (Signed)
setge	~(SF^OF)	Greater or Equal (Signed)
setl	(SF^OF)	Less (Signed)
setle	(SF^OF) ZF	Less or Equal (Signed)
seta	~CF & ~ZF	Above (unsigned)
setb	CF	Below (unsigned)

Các thanh ghi x86-64: low-order byte?

%rax	%al	%r8	%r8b
%rbx	%bl	%r9	%r9b
%rcx	%cl	%r10	%r10b
%rdx	%dl	%r11	%r11b
%rsi	%sil	%r12	%r12b
%rdi	%dil	%r13	%r13b
%rsp	%spl	%r14	%r14b
%rbp	%bp1	%r15	%r15b

- Có thể tham chiếu đến các byte thấp này

Extra 2: Sử dụng Condition Codes

Gán giá trị dựa trên điều kiện (tt)

- Các instruction SetX:
 - Gán giá trị cho 1 byte dựa trên 1 nhóm các condition codes
- Thay đổi 1 byte trong các thanh ghi
 - Không thay đổi các bytes còn lại
 - Thường dùng **movzbl**
 - Instruction 32-bit cũng gán 32 bits cao thành 0

```
int gt (long x, long y)
{
    return x > y;
}
```

Thanh ghi	Tác dụng
%rdi	Tham số x
%rsi	Tham số y
%rax	Giá trị trả về

```
cmpq    %rsi, %rdi      # Compare x:y
setg    %al               # Set when >
movzbl  %al, %eax       # Zero rest of %rax
ret
```

Extra 3: Sử dụng Condition Codes

Chuyển giá trị có điều kiện (conditional move)

- Các instruction move có điều kiện
 - Hỗ trợ thực hiện:
if (Test) Dest \leftarrow Src
 - Hỗ trợ trong các bộ xử lý x86 từ 1995 trở về sau
 - GCC tries to use them
 - But, only when known to be safe
- Why?
 - Branches are very disruptive to instruction flow through pipelines
 - Conditional moves không cần chuyển luồng

C Code

```
val = Test  
? Then_Expr  
: Else_Expr;
```

Goto Version

```
result = Then_Expr;  
eval = Else_Expr;  
nt = !Test;  
if (nt) result = eval;  
return result;
```

Chuyển giá trị có điều kiện (conditional move)

Ví dụ

```
long absdiff
    (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

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

```
absdiff:
    movq    %rdi, %rax    # x
    subq    %rsi, %rax    # result = x-y
    movq    %rsi, %rdx
    subq    %rdi, %rdx    # eval = y-x
    cmpq    %rsi, %rdi    # x:y
    cmovle %rdx, %rax    # if <=, result = eval
    ret
```

Chuyển giá trị có điều kiện (conditional move)

Bad cases

Tính toán phức tạp

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

- Cả 2 giá trị đều được tính toán
- Chỉ hữu ích khi các phép tính toán đều đơn giản

Tính toán có rủi ro

```
val = p ? *p : 0;
```

- Cả 2 giá trị đều được tính toán
- Có thể có những ảnh hưởng không mong muốn (p null?)

Tính toán có tác động phụ

```
val = x > 0 ? x*=7 : x+=3;
```

- Cả 2 giá trị đều được tính toán
- Cần loại bỏ tác động phụ

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

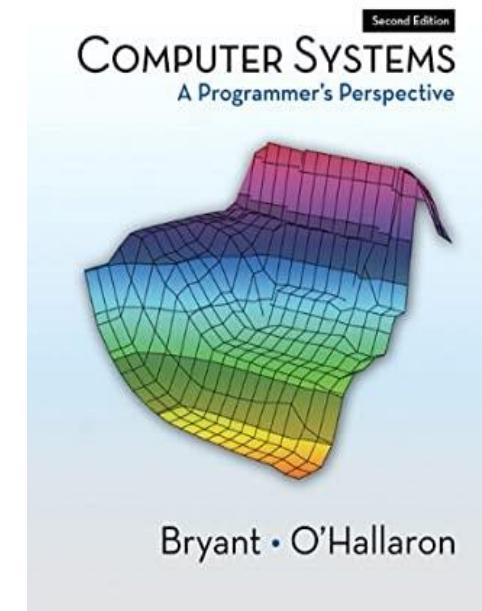
- Lab 1: Nội dung 1
- Lab 2: Nội dung 1, 2, 3
- Lab 3: Nội dung 1, 2, 3, 4, 5, 6
- Lab 4: Nội dung 1, 2, 3, 4, 5, 6
- Lab 5: Nội dung 1, 2, 3, 4, 5, 6
- Lab 6: Nội dung 1, 2, 3, 4, 5, 6

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 :)