

CSC 252/452: Computer Organization

Fall 2025: Lecture 7

Instructor: Yanan Guo

Department of Computer Science
University of Rochester

Announcement

- Programming Assignment 2 is out
 - Details:
<https://www.cs.rochester.edu/courses/252/fall2025/labs/assignment2.html>
 - Due on **Oct. 1st**, 11:59 PM
 - You (may still) have 3 slip days

Complete Memory Addressing Modes

- The General Form: $D(Rb, Ri, S)$
 - Memory address: $Reg[Rb] + S * Reg[Ri] + D$
 - E.g., `8(%eax, %ebx, 4);` // address = $\%eax + 4 * \%ebx + 8$
 - D: Constant “displacement”
 - Rb: Base register: Any of 16 integer registers
 - Ri: Index register: Any, except for `%rsp`
 - S: Scale: 1, 2, 4, or 8
- What is `8(%eax, %ebx, 4)` used for?
- Special Cases

(Rb, Ri)

address = $Reg[Rb] + Reg[Ri]$

$D(Rb, Ri)$

address = $Reg[Rb] + Reg[Ri] + D$

(Rb, Ri, S)

address = $Reg[Rb] + S * Reg[Ri]$

Address Computation Instruction

```
leaq 4(%rsi,%rdi,2), %rax
```



$$\%rax = \%rsi + \%rdi * 2 + 4$$

- **leaq** Src, Dst
 - *Src* is address mode expression
 - Set *Dst* to address denoted by expression
 - No actual memory reference is made
- **Uses**
 - Computing addresses without a memory reference
 - E.g., translation of `p = &x[i];`

Some Arithmetic Operations (2 Operands)

Format	Computation	Notes
addq src, dest	Dest = Dest + Src	
subq src, dest	Dest = Dest - Src	
imulq src, dest	Dest = Dest * Src	
salq src, dest	Dest = Dest << Src	Also called shlq
sarq src, dest	Dest = Dest >> Src	Arithmetic shift
shrq src, dest	Dest = Dest >> Src	Logical shift
xorq src, dest	Dest = Dest ^ Src	
andq src, dest	Dest = Dest & Src	
orq src, dest	Dest = Dest Src	

Some Arithmetic Operations (2 Operands)

No distinction between signed and unsigned (why?)

- Bit level behaviors for signed and unsigned arithmetic are exactly the same — assuming truncation

Some Arithmetic Operations (2 Operands)

No distinction between signed and unsigned (why?)

- Bit level behaviors for signed and unsigned arithmetic are exactly the same — assuming truncation

```
long signed_add  
(long x, long y)  
{  
    long res = x + y;  
    return res;  
}
```

```
#x in %rdx, y in %rax  
addq    %rdx, %rax
```

Some Arithmetic Operations (2 Operands)

No distinction between signed and unsigned (why?)

- Bit level behaviors for signed and unsigned arithmetic are exactly the same — assuming truncation

```
long signed_add  
(long x, long y)  
{  
    long res = x + y;  
    return res;  
}
```

```
#x in %rdx, y in %rax  
addq    %rdx, %rax
```

```
long unsigned_add  
(unsigned long x, unsigned long y)  
{  
    unsigned long res = x + y;  
    return res;  
}
```

```
#x in %rdx, y in %rax  
addq    %rdx, %rax
```

Some Arithmetic Operations (2 Operands)

No distinction between signed and unsigned (why?)

- Bit level behaviors for signed and unsigned arithmetic are exactly the same — assuming truncation

Bit-level

```
  010
+) 101
-----
  111
```

```
long signed_add
(long x, long y)
{
    long res = x + y;
    return res;
}
```

```
#x in %rdx, y in %rax
addq    %rdx, %rax
```

```
long unsigned_add
(unsigned long x, unsigned long y)
{
    unsigned long res = x + y;
    return res;
}
```

```
#x in %rdx, y in %rax
addq    %rdx, %rax
```

Some Arithmetic Operations (2 Operands)

No distinction between signed and unsigned (why?)

- Bit level behaviors for signed and unsigned arithmetic are exactly the same — assuming truncation

Bit-level
010
+) 101

111

Signed
2
+) -3

-1

```
long signed_add  
(long x, long y)  
{  
    long res = x + y;  
    return res;  
}
```

```
#x in %rdx, y in %rax  
addq    %rdx, %rax
```

```
long unsigned_add  
(unsigned long x, unsigned long y)  
{  
    unsigned long res = x + y;  
    return res;  
}
```

```
#x in %rdx, y in %rax  
addq    %rdx, %rax
```

Some Arithmetic Operations (2 Operands)

No distinction between signed and unsigned (why?)

- Bit level behaviors for signed and unsigned arithmetic are exactly the same — assuming truncation

Bit-level
010
+) 101

111

Signed
2
+) -3

-1

Unsigned
2
+) 5

7

```
long signed_add  
(long x, long y)  
{  
    long res = x + y;  
    return res;  
}
```

```
#x in %rdx, y in %rax  
addq    %rdx, %rax
```

```
long unsigned_add  
(unsigned long x, unsigned long y)  
{  
    unsigned long res = x + y;  
    return res;  
}
```

```
#x in %rdx, y in %rax  
addq    %rdx, %rax
```

Side Effect -- Set Condition Codes

```
addq %rax, %rbx
```

- Arithmetic instructions implicitly set condition codes (think of it as side effect)
 - **CF** set if `%rax + %rbx` generates a carry (i.e., unsigned overflow)
 - **ZF** set if `%rax + %rbx == 0`
 - **SF** set if `%rax + %rbx < 0`
 - **OF** set if `%rax + %rbx` overflows when `%rax` and `%rbx` are treated as signed numbers
 - `%rax > 0, %rbx > 0, and (%rax + %rbx) < 0`, or
 - `%rax < 0, %rbx < 0, and (%rax + %rbx) >= 0`

Bit-level
111
+) 010

1001

Signed
-1
+) 2

1

Unsigned
7
+) 2

9

1	0	0	0
CF	ZF	SF	OF

Side Effect -- Set Condition Codes

```
addq %rax, %rbx
```

- Arithmetic instructions implicitly set condition codes (think of it as side effect)
 - **CF** set if `%rax + %rbx` generates a carry (i.e., unsigned overflow)
 - **ZF** set if `%rax + %rbx == 0`
 - **SF** set if `%rax + %rbx < 0`
 - **OF** set if `%rax + %rbx` overflows when `%rax` and `%rbx` are treated as signed numbers
 - `%rax > 0`, `%rbx > 0`, and `(%rax + %rbx) < 0`, or
 - `%rax < 0`, `%rbx < 0`, and `(%rax + %rbx) >= 0`

Bit-level
011
+) 001

100

Signed
3
+) 1

-4

Unsigned
3
+) 1

4

0	0	1	1
CF	ZF	SF	OF

Compare Instruction

■ `cmpq a, b`

- Computes $b - a$ (just like **sub**)
- Sets condition codes based on result, but...
- **Does not change *b***
- All it does is setting condition codes!

How Should `cmpq` Set Condition Codes?

```
cmpq    %rsi, %rdi
```

```
cmpq 0xFF, 0x80
```

10000000	-128
-) 11111111	-) -1
<hr/>	<hr/>
10000001	-127

- ZF** Zero Flag (result is zero)
- SF** Sign Flag (result is negative)
- OF** Overflow Flag (result overflow)

0	1	0	1
ZF	SF	OF	CF

How Should `cmpq` Set Condition Codes?

```
cmpq    %rsi, %rdi
```

```
cmpq 0xFF, 0x80
```

10000000	-128
-) 11111111	-) -1
<hr/>	<hr/>
10000001	-127

- ZF** Zero Flag (result is zero)
- SF** Sign Flag (result is negative)
- OF** Overflow Flag (result overflow)

0	1	0	1
ZF	SF	OF	CF

How Should `cmpq` Set Condition Codes?

```
cmpq    %rsi, %rdi
```

```
cmpq 0xFF, 0x80
```

10000000	-128
-) 11111111	-) -1
<hr/>	<hr/>
10000001	-127

- ZF** Zero Flag (result is zero)
- SF** Sign Flag (result is negative)
- OF** Overflow Flag (result overflow)

0	1	0	1
ZF	SF	OF	CF

How to know if `%rdi = %rsi`?

Check ZF

How Should `cmpq` Set Condition Codes?

```
cmpq    %rsi, %rdi
```

```
cmpq 0xFF, 0x80
```

10000000	-128
-) 11111111	-) -1
<hr/>	<hr/>
10000001	-127

ZF Zero Flag (result is zero)

SF Sign Flag (result is negative)

OF Overflow Flag (result overflow)

0	1	0	1
ZF	SF	OF	CF

How to know if `%rdi = %rsi`?

Check ZF

How to know if `%rdi < %rsi (signed)`?

`%rdi - %rsi < 0` and the result doesn't overflow, or

`%rdi - %rsi > 0` and the result does overflow

or simply: **(SF ^ OF)**

Reading Condition Codes

- SetX Instructions: `setl %al`
 - Set low-order byte of destination to 0 or 1 based on combinations of condition codes

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

Reading Condition Codes (Cont.)

- SetX Instructions:
 - Set single byte based on combination of condition codes
- One of addressable byte registers
 - Does not alter remaining bytes
 - Typically use `movzbl` to finish job
 - 32-bit instructions also set upper bits to 0

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

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

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

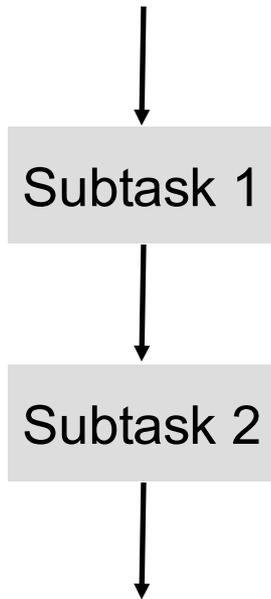
Today: Control Instructions

- Control: Conditional branches (**if... else...**)
- Control: Loops (for, while)
- Control: Switch Statements (**case... switch...**)

Three Basic Programming Constructs

Three Basic Programming Constructs

Sequential



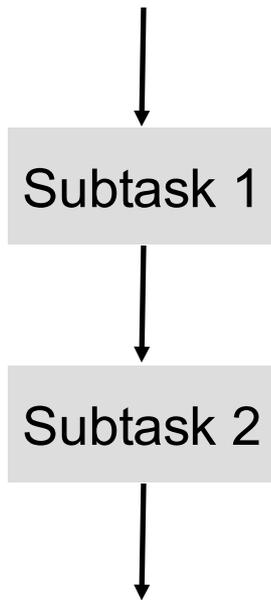
```
a = x + y;
```

```
y = a - c;
```

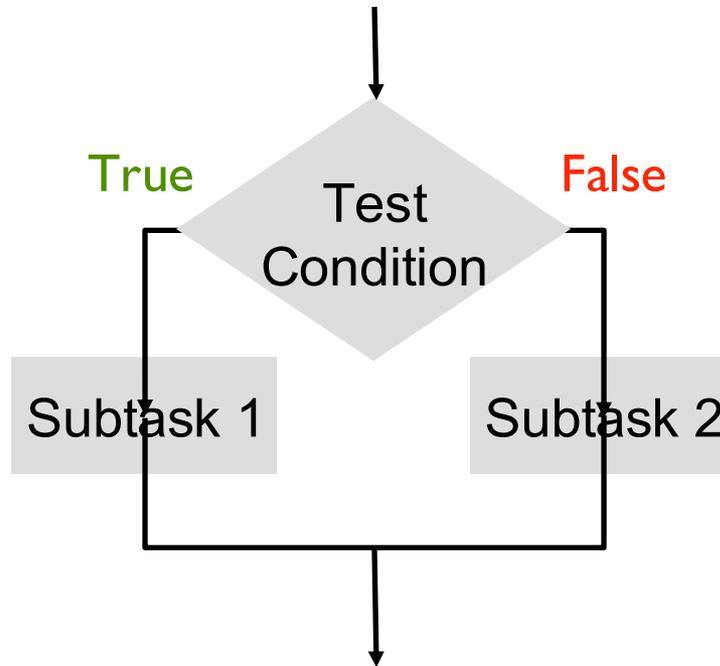
```
...
```

Three Basic Programming Constructs

Sequential



Conditional



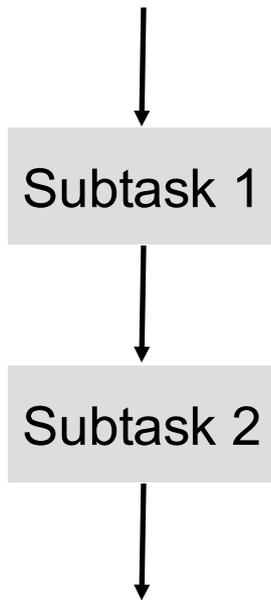
```
a = x + y;  
y = a - c;
```

...

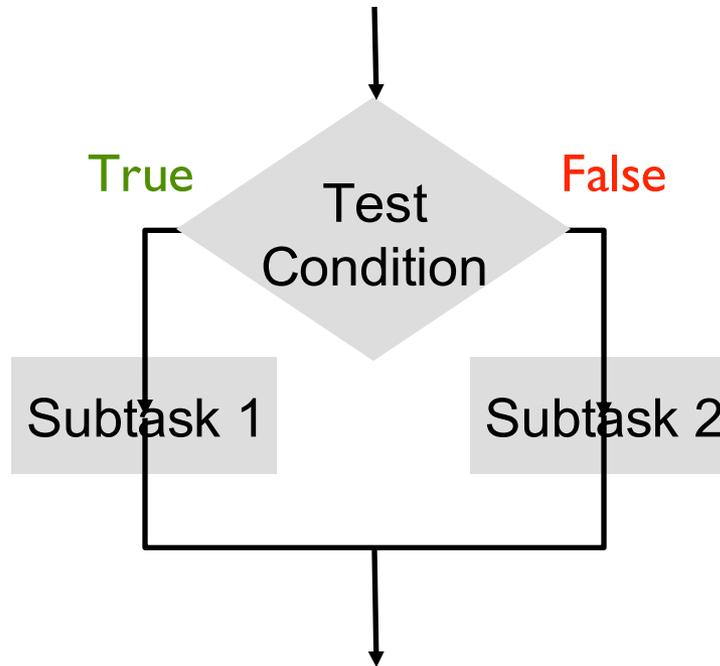
```
if (x > y) r = x - y;  
else r = y - x;
```

Three Basic Programming Constructs

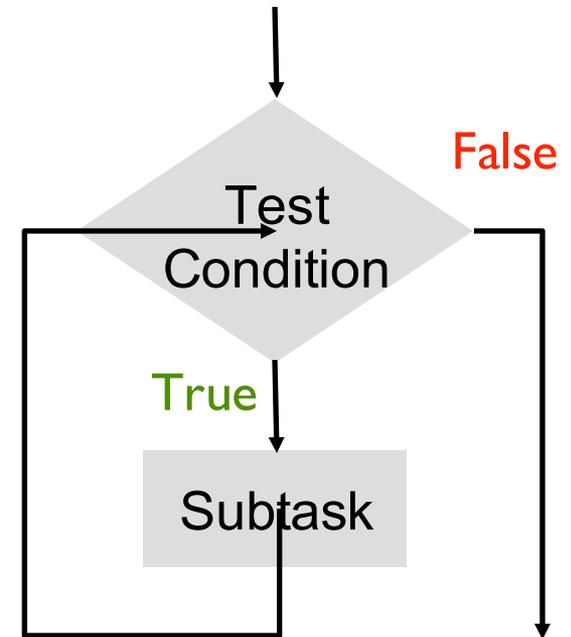
Sequential



Conditional



Iterative



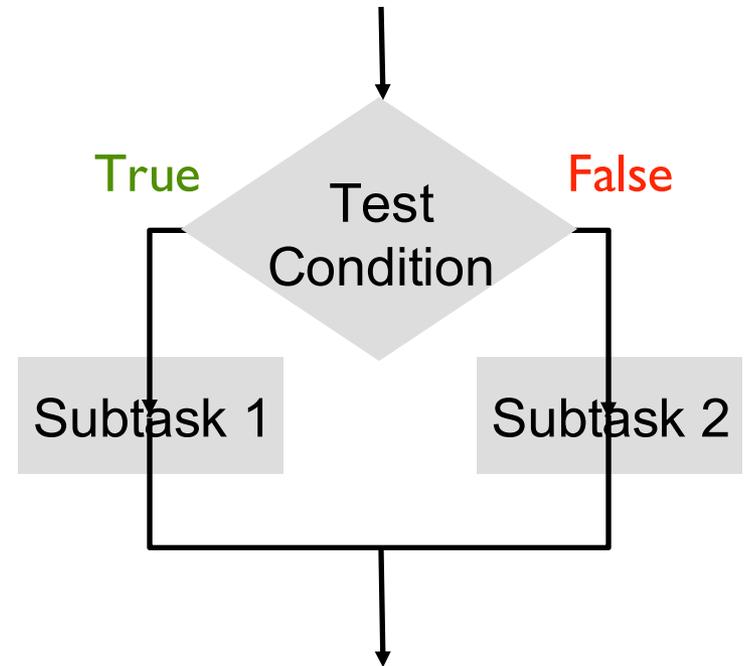
```
a = x + y;  
y = a - c;  
...
```

```
if (x > y) r = x - y;  
else r = y - x;
```

```
while (x > 0) {  
    x--;  
}
```

Three Basic Programming Constructs

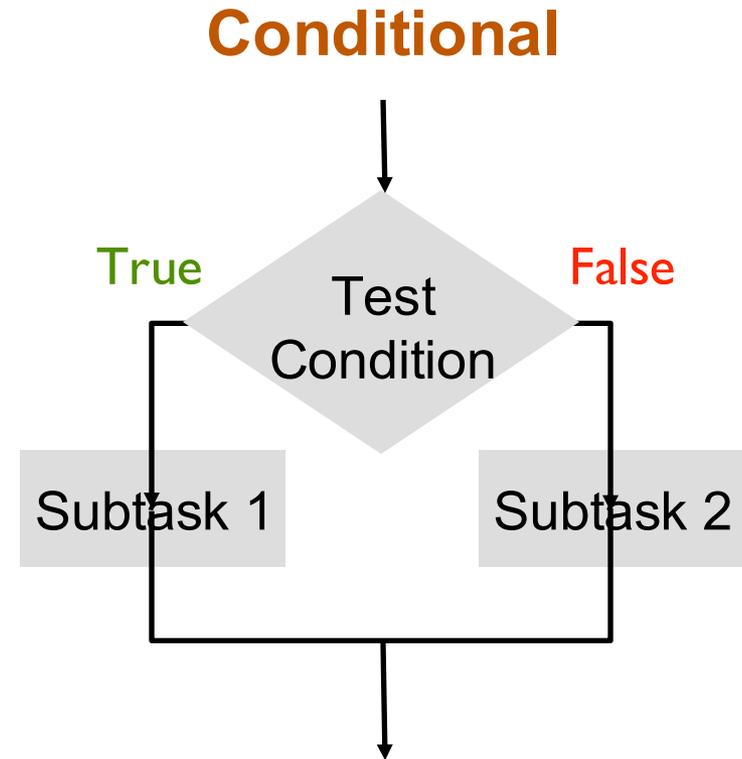
Conditional



```
if (x > y) r = x - y;  
else r = y - x;
```

Three Basic Programming Constructs

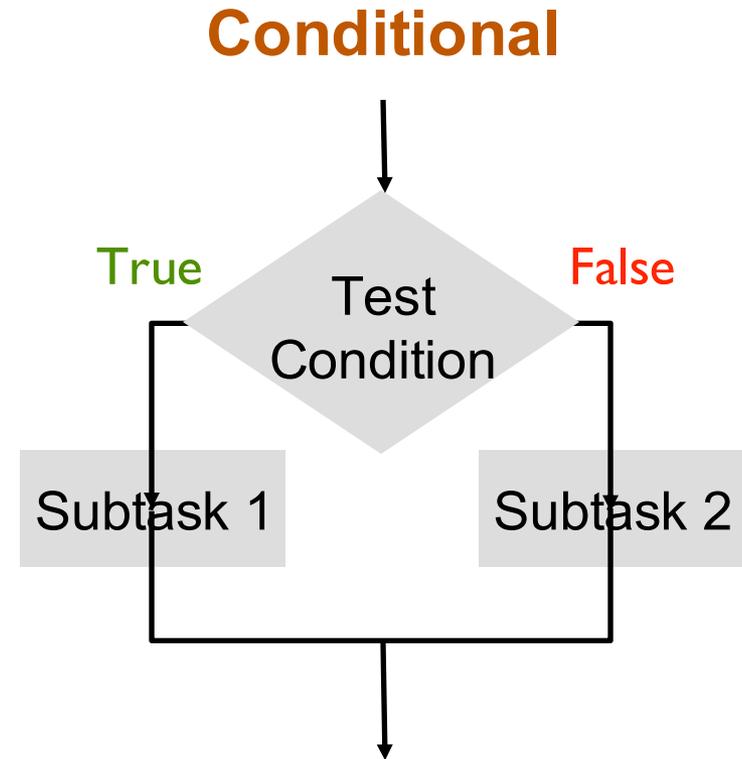
- Both conditional and iterative programming requires altering the sequence of instructions (control flow)



```
if (x > y) r = x - y;  
else r = y - x;
```

Three Basic Programming Constructs

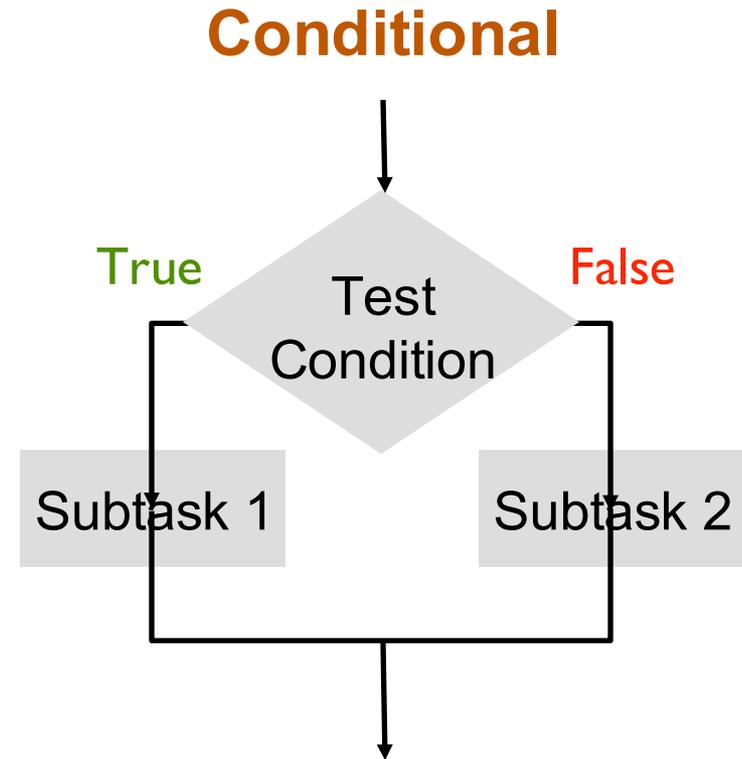
- Both conditional and iterative programming requires altering the sequence of instructions (control flow)
- We need a set of *control instructions* to do so



```
if (x > y) r = x - y;  
else r = y - x;
```

Three Basic Programming Constructs

- Both conditional and iterative programming requires altering the sequence of instructions (control flow)
- We need a set of *control instructions* to do so
- Two fundamental questions:
- How to test condition and how to represent test results?
- How to alter control flow according to the test results?



```
if (x > y) r = x - y;  
else r = y - x;
```

Jumping

■ jX Instructions

- Jump to different part of code depending on condition codes

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	$\sim ZF$	Not Equal / Not Zero
js	SF	Negative
jns	$\sim SF$	Nonnegative
jg	$\sim (SF \wedge OF) \ \& \ \sim ZF$	Greater (Signed)
jge	$\sim (SF \wedge OF)$	Greater or Equal (Signed)
jl	$(SF \wedge OF)$	Less (Signed)
jle	$(SF \wedge OF) \ \ ZF$	Less or Equal (Signed)
ja	$\sim CF \ \& \ \sim ZF$	Above (unsigned)
jb	CF	Below (unsigned)

Conditional Branch Example

Conditional Branch Example

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

Conditional Branch Example

```
gcc -Og -S -fno-if-conversion control.c
```

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

```
absdiff:
    cmpq    %rsi,%rdi # x:y
    jle    .L4
    movq   %rdi,%rax
    subq   %rsi,%rax
    ret

.L4:      # x <= y
    movq   %rsi,%rax
    subq   %rdi,%rax
    ret
```

Conditional Branch Example

```
gcc -Og -S -fno-if-conversion control.c
```

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

```
absdiff:
    cmpq    %rsi,%rdi # x:y
    jle    .L4
    movq   %rdi,%rax
    subq   %rsi,%rax
    ret

.L4:
    # x <= y
    movq   %rsi,%rax
    subq   %rdi,%rax
    ret
```

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

Conditional Branch Example

```
gcc -Og -S -fno-if-conversion control.c
```

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

```
absdiff:
    cmpq    %rsi,%rdi # x:y
    jle    .L4
    movq    %rdi,%rax
    subq    %rsi,%rax
    ret
.L4:      # x <= y
    movq    %rsi,%rax
    subq    %rdi,%rax
    ret
```

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

Labels are symbolic names used to refer to instruction addresses.

Conditional Branch Example

```
gcc -Og -S -fno-if-conversion control.c
```

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

```

```
absdiff:
    cmpq    %rsi,%rdi # x:y
    jle    .L4
    movq   %rdi,%rax
    subq   %rsi,%rax
    ret

.L4:     # x <= y
    movq   %rsi,%rax
    subq   %rdi,%rax
    ret

```

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

Labels are symbolic names used to refer to instruction addresses.

Conditional Branch Example

```
gcc -Og -S -fno-if-conversion control.c
```

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

```

```
absdiff:
    cmpq    %rsi,%rdi # x:y
    jbe    .L4
    movq    %rdi,%rax
    subq    %rsi,%rax
    ret
.L4:      # x <= y
    movq    %rsi,%rax
    subq    %rdi,%rax
    ret

```

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

Labels are symbolic names used to refer to instruction addresses.

Conditional Jump Instruction

```
cmpq    %rsi, %rdi  
jle     .L4
```

Conditional Jump Instruction

```
cmpq    %rsi, %rdi  
jle     .L4
```



Jump to label if less
than or equal to

Conditional Jump Instruction

```
cmpq    %rsi, %rdi  
jle     .L4
```

Jump to label if less
than or equal to

Semantics:

If `%rdi` is less than or
equal to `%rsi` (both
interpreted as **signed
value**), jump to the part
of the code with a label
`.L4`

Conditional Jump Instruction

```
cmpq    %rsi, %rdi  
jle     .L4
```

← Jump to label if less than or equal to

Semantics:

If `%rdi` is less than or equal to `%rsi` (both interpreted as **signed value**), jump to the part of the code with a label `.L4`

- Under the hood:

Conditional Jump Instruction

```
cmpq    %rsi, %rdi  
jle     .L4
```

← Jump to label if less than or equal to

Semantics:

If `%rdi` is less than or equal to `%rsi` (both interpreted as **signed value**), jump to the part of the code with a label `.L4`

- Under the hood:
 - `cmpq` instruction sets the condition codes

Conditional Jump Instruction

```
cmpq    %rsi, %rdi
jle     .L4
```

← Jump to label if less than or equal to

Semantics:

If `%rdi` is less than or equal to `%rsi` (both interpreted as **signed value**), jump to the part of the code with a label `.L4`

- Under the hood:
 - `cmpq` instruction sets the condition codes
 - `jle` reads and checks the **condition codes**

Conditional Jump Instruction

```
cmpq    %rsi, %rdi
jle     .L4
```

← Jump to label if less than or equal to

Semantics:

If `%rdi` is less than or equal to `%rsi` (both interpreted as **signed value**), jump to the part of the code with a label `.L4`

- Under the hood:
 - `cmpq` instruction sets the condition codes
 - `jle` reads and checks the **condition codes**
 - If condition met, modify the Program Counter to point to the address of the instruction with a label `.L4`

Conditional Branch Example

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

```
absdiff:
    cmpq    %rsi,%rdi # x:y
    jle    .L4
    movq    %rdi,%rax
    subq    %rsi,%rax
    ret
.L4:      # x <= y
    movq    %rsi,%rax
    subq    %rdi,%rax
    ret
```

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

0 **0** **0**
ZF SF OF

Conditional Branch Example

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

```
absdiff:
    cmpq    %rsi,%rdi # x:y
    jle     .L4
    movq    %rdi,%rax
    subq    %rsi,%rax
    ret
.L4:      # x <= y
    movq    %rsi,%rax
    subq    %rdi,%rax
    ret
```

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

cmpq sets ZF, SF, OF
jle checks ZF | (SF ^
OF)


ZF SF OF

Today: Control Instructions

- Control: Conditional branches (`if... else...`)
- Control: Loops (**`for`**, **`while`**)
- Control: Switch Statements (`case... switch...`)

“Do-While” Loop Example

- Popcount: Count number of 1's in argument x

do-while version

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

“Do-While” Loop Example

- Popcount: Count number of 1's in argument x

do-while version

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

goto Version

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

“Do-While” Loop Assembly

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

“Do-While” Loop Assembly

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

Register	Use(s)
%rdi	Argument x
%rax	result

```
    movl    $0, %rax    # result = 0
.L2:                                # loop:
    movq   %rdi, %rdx
    andl   $1, %rdx    # t = x & 0x1
    addq   %rdx, %rax  # result += t
    shrq   $1, %rdi    # x >>= 1
    jne    .L2         # if (x) goto loop
    ret
```

“Do-While” Loop Assembly



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

Register	Use(s)
%rdi	Argument x
%rax	result



```
    movl    $0, %rax    # result = 0
.L2:
    # loop:
    movq   %rdi, %rdx
    andl   $1, %rdx    # t = x & 0x1
    addq   %rdx, %rax  # result += t
    shrq   $1, %rdi    # x >>= 1
    jne    .L2         # if (x) goto loop
    ret
```

“Do-While” Loop Assembly



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

Register	Use(s)
%rdi	Argument x
%rax	result



```
    movl    $0, %rax    # result = 0
.L2:                                # loop:
    movq   %rdi, %rdx
    andl   $1, %rdx    # t = x & 0x1
    addq  %rdx, %rax   # result += t
    shrq  $1, %rdi    # x >>= 1
    jne   .L2         # if (x) goto loop
    ret
```

“Do-While” Loop Assembly



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

Register	Use(s)
%rdi	Argument x
%rax	result



```
    movl    $0, %rax    # result = 0
.L2:                                # loop:
    movq    %rdi, %rdx
    andl    $1, %rdx    # t = x & 0x1
    addq    %rdx, %rax  # result += t
    shrq    $1, %rdi    # x >>= 1
    jne     .L2         # if (x) goto loop
    ret
```

“Do-While” Loop Assembly

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

Register	Use(s)
%rdi	Argument x
%rax	result

```
    movl    $0, %rax    # result = 0
.L2:
    # loop:
    movq   %rdi, %rdx
    andl   $1, %rdx    # t = x & 0x1
    addq   %rdx, %rax  # result += t
    shrq   $1, %rdi    # x >>= 1
    jne    .L2         # if (x) goto loop
    ret
```

“Do-While” Loop Assembly

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

Register	Use(s)
%rdi	Argument x
%rax	result

```
    movl    $0, %rax    # result = 0
.L2:
    # loop:
    movq   %rdi, %rdx
    andl  $1, %rdx    # t = x & 0x1
    addq  %rdx, %rax  # result += t
    shrq  $1, %rdi    # x >>= 1
    jne   .L2         # if (x) goto loop
    ret
```

“Do-While” Loop Assembly

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

Register	Use(s)
%rdi	Argument x
%rax	result

```
    movl    $0, %rax    # result = 0
.L2:                                # loop:
    movq    %rdi, %rdx
    andl    $1, %rdx    # t = x & 0x1
    addq    %rdx, %rax  # result += t
    shrq    $1, %rdi    # x >>= 1
    jne     .L2         # if (x) goto loop
    ret
```

“Do-While” Loop Assembly

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

Register	Use(s)
%rdi	Argument x
%rax	result

```
    movl    $0, %rax    # result = 0
.L2:
    # loop:
    movq   %rdi, %rdx
    andl   $1, %rdx    # t = x & 0x1
    addq   %rdx, %rax  # result += t
    shrq   $1, %rdi    # x >>= 1
    jne    .L2         # if (x) goto loop
    ret
```

“Do-While” Loop Assembly



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

Register	Use(s)
%rdi	Argument x
%rax	result



```
    movl    $0, %rax    # result = 0
.L2:                                # loop:
    movq   %rdi, %rdx
    andl   $1, %rdx    # t = x & 0x1
    addq  %rdx, %rax   # result += t
    shrq  $1, %rdi    # x >>= 1
    jne   .L2         # if (x) goto loop
    ret
```

“Do-While” Loop Assembly



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

Register	Use(s)
%rdi	Argument x
%rax	result



```
    movl    $0, %rax    # result = 0
.L2:
    # loop:
    movq   %rdi, %rdx
    andl  $1, %rdx    # t = x & 0x1
    addq  %rdx, %rax  # result += t
    shrq  $1, %rdi    # x >>= 1
    jne   .L2        # if (x) goto loop
    ret
```

General “Do-While” Translation

do-while version

```
<before>;  
do {  
    body;  
} while (A < B) ;  
<after>;
```



goto Version

```
    <before>  
.L1: <body>  
    if (A < B)  
        goto .L1  
    <after>
```

General “Do-While” Translation

do-while version

```
<before>;  
do {  
    body;  
} while (A < B);  
<after>;
```



goto Version

```
    <before>  
.L1: <body>  
    if (A < B)  
        goto .L1  
<after>
```

Replace with a
conditional jump
instruction

General “Do-While” Translation

do-while version

```
<before>;  
do {  
    body;  
} while (A < B);  
<after>;
```



goto Version

```
    <before>  
.L1: <body>  
    if (A < B)  
        goto .L1  
<after>
```



Assembly
Version

```
    <before>  
.L1: <body>  
    cmpq B, A  
    jl .L1  
<after>
```

General “While” Translation

`while` version

```
<before>;  
while (A < B) {  
    body;  
}  
<after>;
```

General “While” Translation

while version

```
<before>;  
while (A < B) {  
    body;  
}  
<after>;
```



goto Version

```
<before>  
goto .L2  
.L1: <body>  
.L2: if (A < B)  
    goto .L1  
<after>
```

General “While” Translation

while version

```
<before>;  
while (A < B) {  
    body;  
}  
<after>;
```



goto Version

```
<before>  
goto .L2  
.L1: <body>  
.L2: if (A < B)  
      goto .L1  
<after>
```



Assembly
Version

```
<before>  
jmp .L2  
.L1: <body>  
.L2: cmpq A, B  
      jg .L1  
<after>
```

General “While” Translation

while version

```
<before>;  
while (A < B) {  
    body;  
}  
<after>;
```



goto Version

```
<before>  
goto .L2  
.L1: <body>  
.L2: if (A < B)  
      goto .L1  
<after>
```



Assembly
Version

```
<before>  
jmp .L2  
.L1: <body>  
.L2: cmpq A, B  
      jg .L1  
<after>
```

General “While” Translation

while version

```
<before>;  
while (A < B) {  
    body;  
}  
<after>;
```



goto Version

```
<before>  
goto .L2  
.L1: <body>  
.L2: if (A < B)  
      goto .L1  
<after>
```



Assembly
Version

```
<before>  
jmp .L2  
.L1: <body>  
.L2: cmpq A, B  
      jg .L1  
<after>
```

“While” Loop Example

while version

```
long pcount_while
(unsigned long x) {

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

“While” Loop Example

while version

```
long pcount_while
(unsigned long x) {

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

goto Version

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

“For” Loop Example

```
for (init; test; update) {  
    body  
}
```

“For” Loop Example

```
for (init; test; update) {  
    body  
}
```

```
//assume unsigned int is 4 bytes  
long pcount_for (unsigned int x)  
{  
  
    size_t i;  
    long result = 0;  
    for (i = 0; i < 32; i++)  
    {  
        result += (x >> i) & 0x1;  
    }  
    return result;  
}
```

“For” Loop Example

```
for (init; test; update) {  
    body  
}
```

init

i = 0

```
//assume unsigned int is 4 bytes  
long pcount_for (unsigned int x)  
{  
  
    size_t i;  
    long result = 0;  
    for (i = 0; i < 32; i++)  
    {  
        result += (x >> i) & 0x1;  
    }  
    return result;  
}
```

“For” Loop Example

```
for (init; test; update) {  
    body  
}
```

init

i = 0

test

i < 32

```
//assume unsigned int is 4 bytes  
long pcount_for (unsigned int x)  
{  
  
    size_t i;  
    long result = 0;  
    for (i = 0; i < 32; i++)  
    {  
        result += (x >> i) & 0x1;  
    }  
    return result;  
}
```

“For” Loop Example

```
for (init; test; update) {  
    body  
}
```

init

i = 0

test

i < 32

update

i++

```
//assume unsigned int is 4 bytes  
long pcount_for (unsigned int x)  
{  
  
    size_t i;  
    long result = 0;  
    for (i = 0; i < 32; i++)  
    {  
        result += (x >> i) & 0x1;  
    }  
    return result;  
}
```

“For” Loop Example

```
for (init; test; update) {  
    body  
}
```

```
//assume unsigned int is 4 bytes  
long pcount_for (unsigned int x)  
{  
  
    size_t i;  
    long result = 0;  
    for (i = 0; i < 32; i++)  
    {  
        result += (x >> i) & 0x1;  
    }  
    return result;  
}
```

init

i = 0

test

i < 32

update

i++

body

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

Convert “For” Loop to “While” Loop

For Version

```
before;  
for (init; test; update) {  
    body;  
}  
after
```

Convert “For” Loop to “While” Loop

For Version

```
before;  
for (init; test; update) {  
    body;  
}  
after
```



While Version

```
before;  
init;  
while (test) {  
    body;  
    update;  
}  
after;
```

Convert “For” Loop to “While” Loop

For Version

```
before;  
for (init; test; update) {  
    body;  
}  
after
```

While Version

```
before;  
init;  
while (test) {  
    body;  
    update;  
}  
after;
```

Assembly
Version

```
before  
init  
jmp .L2  
.L1: body  
    update  
.L2: cmpq A, B  
    jg .L1  
after
```

Today: Control Instructions

- Control: Conditional branches (`if... else...`)
- Control: Loops (`for, while`)
- Control: Switch Statements (`case... switch...`)

Switch Statement Example

```
long switch_eg (long x, long y, long z)
{
    long w = 1;
    switch(x) {
        case 1:
            w = y*z;
            break;
        case 2:
            w = y/z;
        case 3:
            w += z;
            break;
        case 5:
        case 6:
            w -= z;
            break;
        default:
            w = 2;}
    return w;
}
```

Switch Statement Example

```
long switch_eg (long x, long y, long z)
{
    long w = 1;
    switch(x) {
        case 1:
            w = y*z;
            break;
        case 2:
            w = y/z;
        case 3:
            w += z;
            break;
        case 5:
        case 6:
            w -= z;
            break;
        default:
            w = 2;}
    return w;
}
```

Fall-through case

Switch Statement Example

```
long switch_eg (long x, long y, long z)
{
    long w = 1;
    switch(x) {
        case 1:
            w = y*z;
            break;
        case 2:
            w = y/z;
        case 3:
            w += z;
            break;
        case 5:
        case 6:
            w -= z;
            break;
        default:
            w = 2;}
    return w;
}
```

Fall-through case

Multiple case labels

Switch Statement Example

```
long switch_eg (long x, long y, long z)
{
    long w = 1;
    switch(x) {
        case 1:
            w = y*z;
            break;
        case 2:
            w = y/z;
        case 3:
            w += z;
            break;
        case 5:
        case 6:
            w -= z;
            break;
        default:
            w = 2;
    }
    return w;
}
```

Fall-through case

Multiple case labels

For missing cases,
fall back to default

Switch Statement Example

```
long switch_eg (long x, long y, long z)
{
    long w = 1;
    switch(x) {
        case 1:
            w = y*z;
            break;
        case 2:
            w = y/z;
        case 3:
            w += z;
            break;
        case 5:
        case 6:
            w -= z;
            break;
        default:
            w = 2;
    }
    return w;
}
```

Fall-through case

Multiple case labels

For missing cases,
fall back to default

Converting to a cascade of if-else statements is simple, but cumbersome with too many cases.

Implementing Switch Using Jump Table

Switch Form

```
switch(x) {  
  case val_0:  
    Block 0  
  case val_1:  
    Block 1  
  
  ....  
  case val_n-1:  
    Block n-1  
}
```

Implementing Switch Using Jump Table

Switch Form

```
switch(x) {  
  case val_0:  
    Block 0  
  case val_1:  
    Block 1  
  ....  
  case val_n-1:  
    Block n-1  
}
```

Jump Targets

Targ0: Code Block
0

Targ1: Code Block
1

Targ2: Code Block
2

•
•
•

Targn-1: Code Block
n-1

Implementing Switch Using Jump Table

Switch Form

```
switch(x) {  
  case val_0:  
    Block 0  
  case val_1:  
    Block 1  
  ....  
  case val_n-1:  
    Block n-1  
}
```

Jump Table

JTab:

Targ0
Targ1
Targ2
•
•
•
Targn-1

Jump Targets

Targ0: Code Block 0

Targ1: Code Block 1

Targ2: Code Block 2

•
•
•

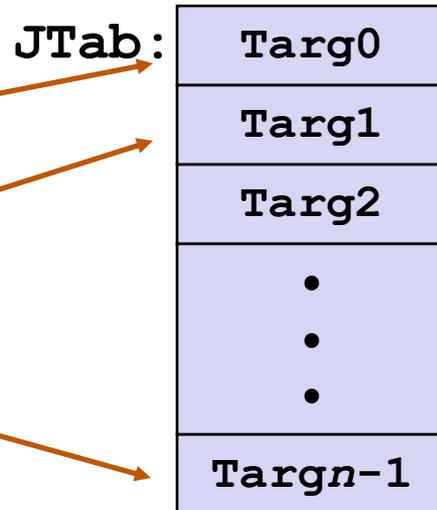
Targn-1: Code Block n-1

Implementing Switch Using Jump Table

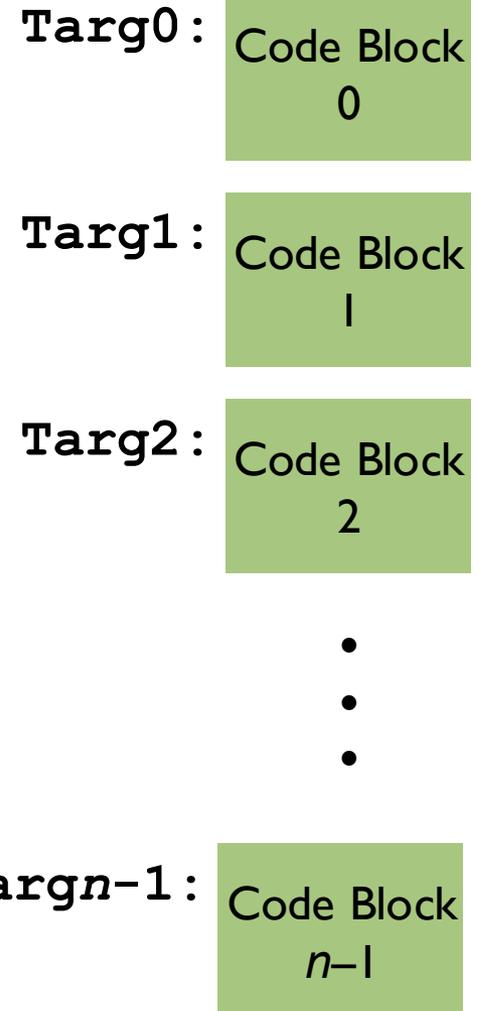
Switch Form

```
switch(x) {  
  case val_0:  
    Block 0  
  case val_1:  
    Block 1  
  ....  
  case val_n-1:  
    Block n-1  
}
```

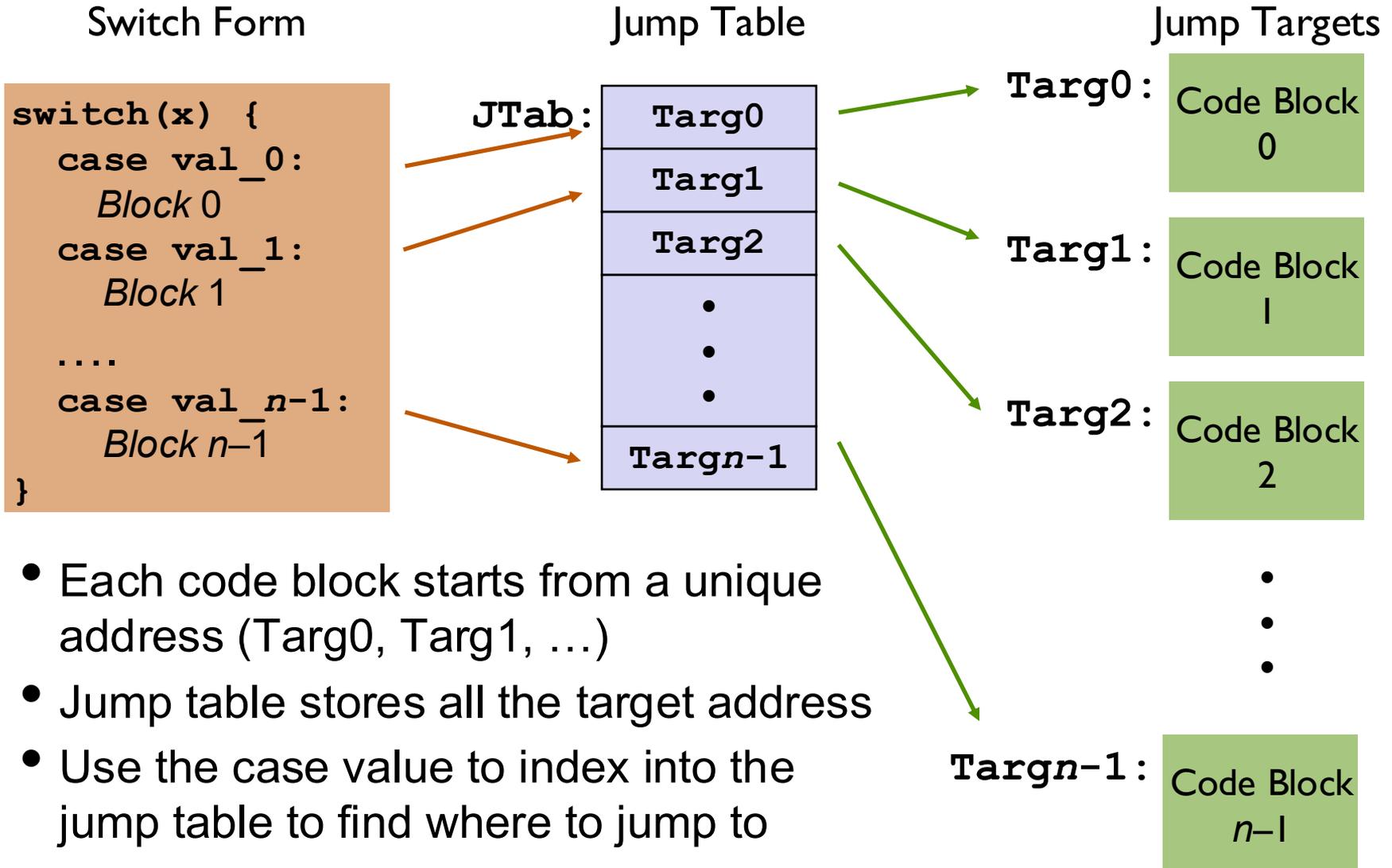
Jump Table



Jump Targets



Implementing Switch Using Jump Table



Assembly Directives (Pseudo-Ops)

```
.L4:
```

```
.quad .LD # x = 0
```

```
.quad .L1 # x = 1
```

```
.quad .L2 # x = 2
```

```
.quad .L3 # x = 3
```

```
.quad .LD # x = 4
```

```
.quad .L5 # x = 5
```

```
.quad .L5 # x = 6
```

- **Directives:**

- Not real instructions, but assist assembler. Think of them as messages to help the assembler in the assembly process.

Assembly Directives (Pseudo-Ops)

.L4:

```
.quad .LD # x = 0
.quad .L1 # x = 1
.quad .L2 # x = 2
.quad .L3 # x = 3
.quad .LD # x = 4
.quad .L5 # x = 5
.quad .L5 # x = 6
```

- **.quad**: tells the assembler to set aside the next 8 bytes in memory and initialize with the value of the operand (a label here, which itself is an address)

- **Directives:**

- Not real instructions, but assist assembler. Think of them as messages to help the assembler in the assembly process.

Jump Table and Jump Targets

Jump Table

```
.L4:  
  .quad .LD # x = 0  
  .quad .L1 # x = 1  
  .quad .L2 # x = 2  
  .quad .L3 # x = 3  
  .quad .LD # x = 4  
  .quad .L5 # x = 5  
  .quad .L5 # x = 6
```

jmp .L3 will go to
.L3 and start
executing from there

Jump Targets

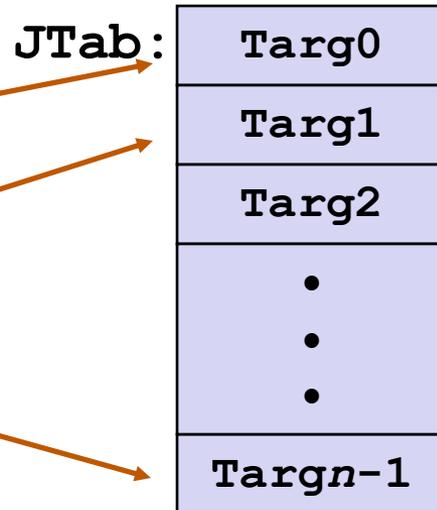
```
.L1:                                # Case 1  
  movq  %rsi, %rax  
  imulq %rdx, %rax  
  jmp   .done  
.L2:                                # Case 2  
  movq  %rsi, %rax  
  cqto  
  idivq %rcx  
.L3:                                # Case 3  
  addq  %rcx, %rax  
  jmp   .done  
.L5:                                # Case 5,6  
  subq  %rdx, %rax  
  jmp   .done  
.LD:                                 # Default  
  movl  $2, %eax  
  jmp   .done
```

Implementing Switch Using Jump Table

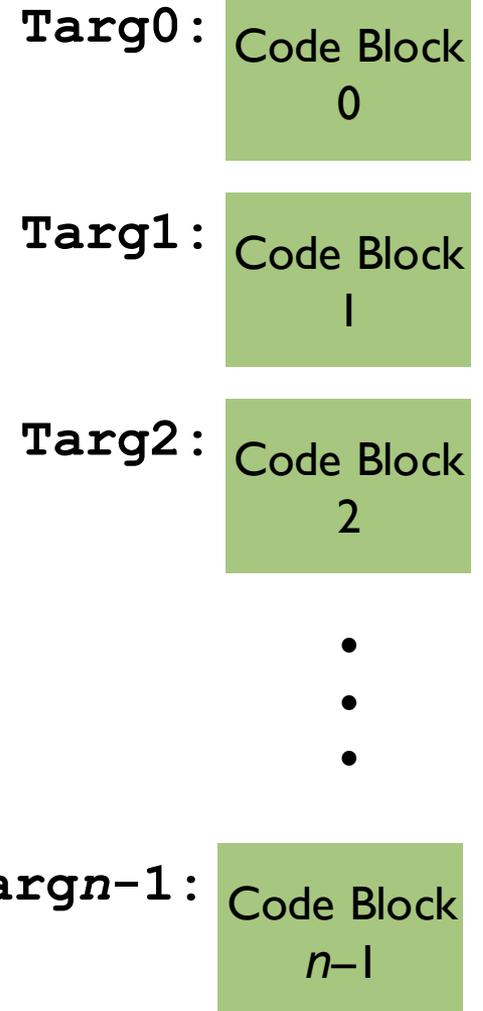
Switch Form

```
switch(x) {  
  case val_0:  
    Block 0  
  case val_1:  
    Block 1  
  ....  
  case val_n-1:  
    Block n-1  
}
```

Jump Table



Jump Targets



Implementing Switch Using Jump Table

Switch Form

```
switch(x) {  
  case val_0:  
    Block 0  
  case val_1:  
    Block 1  
  ....  
  case val_n-1:  
    Block n-1  
}
```

Jump Table

.LJ:	.LD
	.L1
	.L2
	•
	•
	•
	.L5

Jump Targets

.LD: Code Block 0

.L1: Code Block 1

.L2: Code Block 2

•

•

•

.L5: Code Block n-1

.Done:

Code Blocks (x == 1)

.L4:

.quad .LD # x = 0

.quad .L1 # x = 1

.quad .L2 # x = 2

.quad .L3 # x = 3

.quad .LD # x = 4

.quad .L5 # x = 5

.quad .L5 # x = 6

Code Blocks (x == 1)

.L4:

```
.quad .LD # x = 0
.quad .L1 # x = 1
.quad .L2 # x = 2
.quad .L3 # x = 3
.quad .LD # x = 4
.quad .L5 # x = 5
.quad .L5 # x = 6
```

```
switch(x) {
case 1:      // .L1
    w = y*z;
    break;
    ...
}
```

Code Blocks (x == 1)

.L4:

```
.quad .LD # x = 0
.quad .L1 # x = 1
.quad .L2 # x = 2
.quad .L3 # x = 3
.quad .LD # x = 4
.quad .L5 # x = 5
.quad .L5 # x = 6
```

```
switch(x) {
case 1:      // .L1
    w = y*z;
    break;
    ...
}
```

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

Code Blocks (x == 1)

.L4:

```
.quad .LD # x = 0
.quad .L1 # x = 1
.quad .L2 # x = 2
.quad .L3 # x = 3
.quad .LD # x = 4
.quad .L5 # x = 5
.quad .L5 # x = 6
```

```
switch(x) {
case 1:      // .L1
    w = y*z;
    break;
    ...
}
```

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

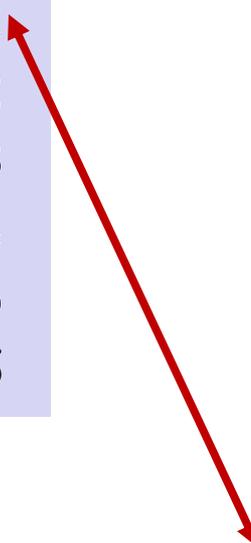
.L1:

```
movq    %rsi, %rax # y
imulq   %rdx, %rax # y*z
jmp     .done
```

Code Blocks (x == 1)

.L4:

```
.quad .LD # x = 0
.quad .L1 # x = 1
.quad .L2 # x = 2
.quad .L3 # x = 3
.quad .LD # x = 4
.quad .L5 # x = 5
.quad .L5 # x = 6
```



```
switch(x) {
case 1:      // .L1
    w = y*z;
    break;
...
}
```

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

.L1:

```
movq    %rsi, %rax # y
imulq   %rdx, %rax # y*z
jmp     .done
```

Code Blocks ($x == 2$, $x == 3$)

.L4:

```
.quad .LD # x = 0
.quad .L1 # x = 1
.quad .L2 # x = 2
.quad .L3 # x = 3
.quad .LD # x = 4
.quad .L5 # x = 5
.quad .L5 # x = 6
```

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

Code Blocks (x == 2, x == 3)

.L4:

```
.quad .LD # x = 0
.quad .L1 # x = 1
.quad .L2 # x = 2
.quad .L3 # x = 3
.quad .LD # x = 4
.quad .L5 # x = 5
.quad .L5 # x = 6
```

```
switch(x) {
...
case 2:          // .L2
    w = y/z;
    /* Fall Through */
case 3:          // .L3
    w += z;
    break;
...}
```

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

Code Blocks (x == 2, x == 3)

.L4:

```
.quad .LD # x = 0
.quad .L1 # x = 1
.quad .L2 # x = 2
.quad .L3 # x = 3
.quad .LD # x = 4
.quad .L5 # x = 5
.quad .L5 # x = 6
```

```
switch(x) {
...
case 2:      // .L2
    w = y/z;
    /* Fall Through */
case 3:      // .L3
    w += z;
    break;
...}
```

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

```
.L2:                                     # Case 2
    movq    %rsi, %rax
    cqto
    idivq   %rcx      # y/z

.L3:                                     # Case 3
    addq    %rcx, %rax # w += z
    jmp     .done
```

Code Blocks (x == 2, x == 3)

.L4:

```
.quad .LD # x = 0
.quad .L1 # x = 1
.quad .L2 # x = 2
.quad .L3 # x = 3
.quad .LD # x = 4
.quad .L5 # x = 5
.quad .L5 # x = 6
```

```
switch(x) {
...
case 2:      // .L2
    w = y/z;
    /* Fall Through */
case 3:      // .L3
    w += z;
    break;
...}
```

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

```
.L2:                                     # Case 2
    movq    %rsi, %rax
    cqto
    idivq   %rcx          # y/z

.L3:                                     # Case 3
    addq    %rcx, %rax   # w += z
    jmp     .done
```

Code Blocks (x == 5, x == 6, default)

.L4:

```
.quad .LD # x = 0
.quad .L1 # x = 1
.quad .L2 # x = 2
.quad .L3 # x = 3
.quad .LD # x = 4
.quad .L5 # x = 5
.quad .L5 # x = 6
```

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

Code Blocks (x == 5, x == 6, default)

.L4:

```
.quad .LD # x = 0
.quad .L1 # x = 1
.quad .L2 # x = 2
.quad .L3 # x = 3
.quad .LD # x = 4
.quad .L5 # x = 5
.quad .L5 # x = 6
```

```
switch(x) {
...
case 5: // .L5
case 6: // .L5
    w -= z;
    break;
default: // .LD
    w = 2; }
```

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

Code Blocks (x == 5, x == 6, default)

.L4:

```
.quad .LD # x = 0
.quad .L1 # x = 1
.quad .L2 # x = 2
.quad .L3 # x = 3
.quad .LD # x = 4
.quad .L5 # x = 5
.quad .L5 # x = 6
```

```
switch(x) {
...
case 5: // .L5
case 6: // .L5
    w -= z;
    break;
default: // .LD
    w = 2; }
```

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

```
.L5: # Case 5,6
    subq %rdx, %rax # w -= z
    jmp .done
.LD: # Default:
    movl $2, %eax # 2
    jmp .done
```

Code Blocks (x == 5, x == 6, default)

.L4:

```
.quad .LD # x = 0
.quad .L1 # x = 1
.quad .L2 # x = 2
.quad .L3 # x = 3
.quad .LD # x = 4
.quad .L5 # x = 5
.quad .L5 # x = 6
```

```
switch(x) {
...
case 5: // .L5
case 6: // .L5
    w -= z;
    break;
default: // .LD
    w = 2; }
```

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

```
.L5: # Case 5,6
    subq %rdx, %rax # w -= z
    jmp .done
.LD: # Default:
    movl $2, %eax # 2
    jmp .done
```

Code Blocks (x == 5, x == 6, default)

.L4:

```
.quad .LD # x = 0
.quad .L1 # x = 1
.quad .L2 # x = 2
.quad .L3 # x = 3
.quad .LD # x = 4
.quad .L5 # x = 5
.quad .L5 # x = 6
```

```
switch(x) {
...
case 5: // .L5
case 6: // .L5
    w -= z;
    break;
default: // .LD
    w = 2; }
```

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

```
.L5: # Case 5,6
    subq %rdx, %rax # w -= z
    jmp .done
.LD: # Default:
    movl $2, %eax # 2
    jmp .done
```

Implementing Switch Using Jump Table

Switch Form

```
switch(x) {  
  case val_0:  
    Block 0  
  case val_1:  
    Block 1  
  ....  
  case val_n-1:  
    Block n-1  
}
```

Jump Table

.LJ:	.LD
	.L1
	.L2
	•
	•
	•
	.L5

Jump Targets

.LD: Code Block 0

.L1: Code Block 1

.L2: Code Block 2

•

•

•

.L5: Code Block n-1

.Done:

Implementing Switch Using Jump Table

Switch Form

```
switch(x) {  
  case val_0:  
    Block 0  
  case val_1:  
    Block 1  
  ....  
  case val_n-1:  
    Block n-1  
}
```

Jump Table

.LJ:	.LD
	.L1
	.L2
	•
	•
	•
	.L5

Jump Targets

.LD: Code Block 0

.L1: Code Block 1

.L2: Code Block 2

•

•

•

.L5: Code Block n-1

.Done:

- The only thing left...
 - How do we jump to different locations in the jump table depending on the case value?

Indirect Jump Instruction

.LJ:

.quad .LD # x = 0

.quad .L1 # x = 1

.quad .L2 # x = 2

.quad .L3 # x = 3

.quad .LD # x = 4

.quad .L5 # x = 5

.quad .L5 # x = 6

Indirect Jump Instruction

The address we want to jump to is stored at $.LJ + 8 * x$

.LJ:

.quad .LD # x = 0

.quad .L1 # x = 1

.quad .L2 # x = 2

.quad .L3 # x = 3

.quad .LD # x = 4

.quad .L5 # x = 5

.quad .L5 # x = 6

Indirect Jump Instruction

The address we want to jump to is stored at $.LJ + 8 * x$

`.LJ:`

```
.quad .LD # x = 0
.quad .L1 # x = 1
.quad .L2 # x = 2
.quad .L3 # x = 3
.quad .LD # x = 4
.quad .L5 # x = 5
.quad .L5 # x = 6
```

```
# assume x in %rdi
movq  .LJ(,%rdi,8), %rax
jmp   *%rax
```

Indirect Jump Instruction

The address we want to jump to is stored at $.LJ + 8 * x$

`.LJ:`

```
.quad .LD # x = 0
.quad .L1 # x = 1
.quad .L2 # x = 2
.quad .L3 # x = 3
.quad .LD # x = 4
.quad .L5 # x = 5
.quad .L5 # x = 6
```

```
# assume x in %rdi
movq  .LJ(,%rdi,8), %rax
jmp   *%rax
```

- Indirect Jump: `jmp *%rax`
 - `%rax` specifies the address to jump to ($PC = \%rax$)

Indirect Jump Instruction

The address we want to jump to is stored at $.LJ + 8 * x$

.LJ:

```
.quad .LD # x = 0
.quad .L1 # x = 1
.quad .L2 # x = 2
.quad .L3 # x = 3
.quad .LD # x = 4
.quad .L5 # x = 5
.quad .L5 # x = 6
```

```
# assume x in %rdi
movq  .LJ(,%rdi,8), %rax
jmp   *%rax
```

- Indirect Jump: **jmp *%rax**
 - `%rax` specifies the address to jump to ($PC = \%rax$)
- Direct Jump (**jmp .LJ**), directly specifies the jump address

Indirect Jump Instruction

The address we want to jump to is stored at $.LJ + 8 * x$

.LJ:

```
.quad .LD # x = 0
.quad .L1 # x = 1
.quad .L2 # x = 2
.quad .L3 # x = 3
.quad .LD # x = 4
.quad .L5 # x = 5
.quad .L5 # x = 6
```

```
# assume x in %rdi
movq  .LJ(,%rdi,8), %rax
jmp   *%rax
```

- Indirect Jump: **jmp *%rax**
 - `%rax` specifies the address to jump to ($PC = \%rax$)
- Direct Jump (**jmp .LJ**), directly specifies the jump address
- Indirect Jump specifies where the jump address is located

Indirect Jump Instruction

The address we want to jump to is stored at $.LJ + 8 * x$

.LJ:

```
.quad .LD # x = 0
.quad .L1 # x = 1
.quad .L2 # x = 2
.quad .L3 # x = 3
.quad .LD # x = 4
.quad .L5 # x = 5
.quad .L5 # x = 6
```

```
# assume x in %rdi
movq  .LJ(,%rdi,8), %rax
jmp   *%rax
```

- Indirect Jump: **jmp *%rax**
 - `%rax` specifies the address to jump to ($PC = \%rax$)
- Direct Jump (**jmp .LJ**), directly specifies the jump address
- Indirect Jump specifies where the jump address is located

An equivalent syntax in x86:

```
jmp   * .LJ(,%rdi,8)
```