

# **CSC 252: Computer Organization**

## **Spring 2018: Lecture 7**

Instructor: Yuhao Zhu

Department of Computer Science  
University of Rochester

### **Action Items:**

- **Assignment 2 is out**

# Announcement

- Programming Assignment 2 is out
  - Due on **Feb 16, 11:59 PM**
  - You may still have 3 slip days...

4	5	6	7	8	9	10
11	12	13	14	15	16	17

due

# Announcement

- There is a faculty candidate talk on Monday

Monday, February 12, 2018  
12:00 PM  
1400 Wegmans Hall

Daniel Epstein  
University of Washington

Everyday Personal Informatics

Personal tracking technology has made it easier for people to better understand themselves and their routines around exercise, eating, finances, and more. This self-knowledge can serve as a first step toward changing behaviors, increasing awareness, or simply satisfying a curiosity. Though some people succeed in achieving their goals, most encounter a fundamental barrier: the design principles used in tracking technology assume people are highly motivated, unwavering in their diligence, and have the expertise necessary to analyze their data.

In this talk, I will demonstrate how the design of tracking technology can be improved to help people overcome two challenges: (1) helping people find value in their tracking, and (2) helping people find support through their tracking. I will present generalizable opportunities for designs to overcome

# 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

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

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

# 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

**Labels** (.L4) are symbolic names referring to memory 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**

# How Should `cmpq` Set Condition Codes?

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

# How Should `cmpq` Set Condition Codes?

`cmpq        %rsi, %rdi`

- Essentially, how do we know `%rdi <= %rsi`?



# How Should `cmpq` Set Condition Codes?

`cmpq        %rsi, %rdi`

- Essentially, how do we know `%rdi <= %rsi`?
- Calculate `%rdi - %rsi`

# How Should `cmpq` Set Condition Codes?

**`cmpq        %rsi, %rdi`**

- Essentially, how do we know `%rdi <= %rsi`?
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`

# How Should `cmpq` Set Condition Codes?

**`cmpq      %rsi, %rdi`**

- Essentially, how do we know `%rdi <= %rsi`?
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`

**ZF** Zero Flag (result is zero)



**ZF**

# How Should `cmpq` Set Condition Codes?

**`cmpq        %rsi, %rdi`**

- Essentially, how do we know `%rdi <= %rsi`?
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`
- `%rdi < %rsi` if and only if: `%rdi - %rsi < 0` (is it correct??)

**ZF** Zero Flag (result is zero)



**ZF**

# How Should `cmpq` Set Condition Codes?

**`cmpq       %rsi, %rdi`**

- Essentially, how do we know `%rdi <= %rsi`?
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`
- `%rdi < %rsi` if and only if: ~~`%rdi - %rsi < 0`~~ (is it correct??)
  - `%rdi - %rsi < 0` and the result doesn't overflow, or

**ZF** Zero Flag (result is zero)



**ZF**

# How Should `cmpq` Set Condition Codes?

**`cmpq        %rsi, %rdi`**

- Essentially, how do we know `%rdi <= %rsi`?
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`
- `%rdi < %rsi` if and only if: ~~`%rdi - %rsi < 0`~~ (is it correct??)
  - `%rdi - %rsi < 0` and the result doesn't overflow, or

No  
Overflow

$$\begin{array}{r} 001 \\ -) 010 \\ \hline 111 \end{array}$$

$$\begin{array}{r} 1 \\ -) 2 \\ \hline -1 \end{array}$$

**ZF** Zero Flag (result is zero)



**ZF**

# How Should `cmpq` Set Condition Codes?

**`cmpq      %rsi, %rdi`**

- Essentially, how do we know `%rdi <= %rsi`?
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`
- `%rdi < %rsi` if and only if: ~~`%rdi - %rsi < 0` (is it correct??)~~
  - `%rdi - %rsi < 0` and the result doesn't overflow, or

No Overflow	$\begin{array}{r} 001 \\ -) 010 \\ \hline 111 \end{array}$	$\begin{array}{r} 1 \\ -) 2 \\ \hline -1 \end{array}$
Overflow	$\begin{array}{r} 101 \\ -) 011 \\ \hline 010 \end{array}$	$\begin{array}{r} -3 \\ -) 3 \\ \hline -6 \end{array}$

**ZF** Zero Flag (result is zero)



# How Should `cmpq` Set Condition Codes?

**`cmpq        %rsi, %rdi`**

- Essentially, how do we know `%rdi <= %rsi`?
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`
- `%rdi < %rsi` if and only if: ~~`%rdi - %rsi < 0`~~ (is it correct??)
  - `%rdi - %rsi < 0` and the result doesn't overflow, or
  - `%rdi - %rsi > 0` and the result does overflow

No Overflow	$\begin{array}{r} 001 \\ -) 010 \\ \hline 111 \end{array}$	$\begin{array}{r} 1 \\ -) 2 \\ \hline -1 \end{array}$
Overflow	$\begin{array}{r} 101 \\ -) 011 \\ \hline 010 \end{array}$	$\begin{array}{r} -3 \\ -) 3 \\ \hline -6 \end{array}$

**ZF** Zero Flag (result is zero)





# How Should `cmpq` Set Condition Codes?

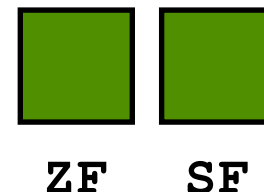
**`cmpq        %rsi, %rdi`**

- Essentially, how do we know `%rdi <= %rsi`?
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`
- `%rdi < %rsi` if and only if: ~~`%rdi - %rsi < 0`~~ (is it correct??)
  - `%rdi - %rsi < 0` and the result doesn't overflow, or
  - `%rdi - %rsi > 0` and the result does overflow

No Overflow	$\begin{array}{r} 001 \\ -) 010 \\ \hline 111 \end{array}$	$\begin{array}{r} 1 \\ -) 2 \\ \hline -1 \end{array}$
	111	-1
Overflow	$\begin{array}{r} 101 \\ -) 011 \\ \hline 010 \end{array}$	$\begin{array}{r} -3 \\ -) 3 \\ \hline -6 \end{array}$
	010	-6

**ZF** Zero Flag (result is zero)

**SF** Sign Flag (result is negative)



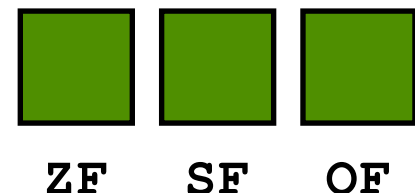
# How Should `cmpq` Set Condition Codes?

**`cmpq        %rsi, %rdi`**

- Essentially, how do we know `%rdi <= %rsi`?
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`
- `%rdi < %rsi` if and only if: ~~`%rdi - %rsi < 0`~~ (is it correct??)
  - `%rdi - %rsi < 0` and the result doesn't overflow, or
  - `%rdi - %rsi > 0` and the result does overflow

No Overflow	$\begin{array}{r} 001 \\ -) 010 \\ \hline 111 \end{array}$	$\begin{array}{r} 1 \\ -) 2 \\ \hline -1 \end{array}$
	111	-1
Overflow	$\begin{array}{r} 101 \\ -) 011 \\ \hline 010 \end{array}$	$\begin{array}{r} -3 \\ -) 3 \\ \hline -6 \end{array}$
	010	-6

**ZF** Zero Flag (result is zero)  
**SF** Sign Flag (result is negative)  
**OF** Overflow Flag (for signed)



# How Should `cmpq` Set Condition Codes?

**`cmpq        %rsi, %rdi`**

- Essentially, how do we know `%rdi <= %rsi`?
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`
- `%rdi < %rsi` if and only if: ~~`%rdi - %rsi < 0`~~ (is it correct??)
  - `%rdi - %rsi < 0` and the result doesn't overflow, or
  - `%rdi - %rsi > 0` and the result does overflow

11111111 10000000  
**`cmpq 0xFF, 0x80`**

**ZF** Zero Flag (result is zero)  
**SF** Sign Flag (result is negative)  
**OF** Overflow Flag (for signed)

0	0	0
<b>ZF</b>	<b>SF</b>	<b>OF</b>

# How Should `cmpq` Set Condition Codes?

**`cmpq      %rsi, %rdi`**

- Essentially, how do we know `%rdi <= %rsi`?
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`
- `%rdi < %rsi` if and only if: ~~`%rdi - %rsi < 0`~~ (is it correct??)
  - `%rdi - %rsi < 0` and the result doesn't overflow, or
  - `%rdi - %rsi > 0` and the result does overflow

11111111 10000000  
**`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 (for signed)

0	0	0
<b>ZF</b>	<b>SF</b>	<b>OF</b>

# How Should `cmpq` Set Condition Codes?

**`cmpq        %rsi, %rdi`**

- Essentially, how do we know `%rdi <= %rsi`?
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`
- `%rdi < %rsi` if and only if: ~~`%rdi - %rsi < 0`~~ (is it correct??)
  - `%rdi - %rsi < 0` and the result doesn't overflow, or
  - `%rdi - %rsi > 0` and the result does overflow

11111111 10000000  
**`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 (for signed)

0	1	0
<b>ZF</b>	<b>SF</b>	<b>OF</b>

# How Should `cmpq` Set Condition Codes?

**`cmpq        %rsi, %rdi`**

- Essentially, how do we know `%rdi <= %rsi`?
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`
- `%rdi < %rsi` if and only if: ~~`%rdi - %rsi < 0`~~ (is it correct??)
  - `%rdi - %rsi < 0` and the result doesn't overflow, or
  - `%rdi - %rsi > 0` and the result does overflow

- `%rdi <= %rsi` if and only if
  - ZF is set, or
  - SF is set but OF is not set, or
  - SF is not set, but OF is set
- or simply: **ZF | (SF ^ OF)**

**ZF** Zero Flag (result is zero)

**SF** Sign Flag (result is negative)

**OF** Overflow Flag (for signed)

0	1	0
<b>ZF</b>	<b>SF</b>	<b>OF</b>

# 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  $ZF \mid (SF \wedge OF)$

0	0	0
ZF	SF	OF



# Conditional Branch Example

```
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

0	0	0
ZF	SF	OF

# Conditional Branch Example

```
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

0	0	0
ZF	SF	OF

# Conditional Jump Instruction

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

# Conditional Jump Instruction

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



Jump to label if  
below or equal to

# Conditional Jump Instruction

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

Jump to label if  
below or equal to

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

# Conditional Jump Instruction

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


← Jump to label if  
below or equal to

- Semantics:

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


- Under the hood:

# Conditional Jump Instruction

**cmpq**            **%rsi, %rdi**  
**jbe**            **.L4**            Jump to label if  
below or equal to

- Semantics:
  - If **%rdi** is less than or equal to **%rsi** (both interpreted as **unsigned** 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**  
**jbe**            **.L4**             Jump to label if  
below or equal to

- Semantics:


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

- Under the hood:

- **cmpq** instruction sets the condition codes
- **jbe** reads and checks the condition codes



# Conditional Jump Instruction

**cmpq**            **%rsi, %rdi**  
**jbe**            **.L4**             Jump to label if  
below or equal to

- Semantics:

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

- Under the hood:

- **cmpq** instruction sets the condition codes
- **jbe** 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**

# How Should `cmpq` Set Condition Codes?

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

# How Should `cmpq` Set Condition Codes?

`cmpq        %rsi, %rdi`

- How do we know `%rdi <= %rsi`? This time for unsigned values

# How Should `cmpq` Set Condition Codes?

`cmpq        %rsi, %rdi`

- How do we know `%rdi <= %rsi`? This time for unsigned values
- Calculate `%rdi - %rsi`

# How Should `cmpq` Set Condition Codes?

**`cmpq        %rsi, %rdi`**

- How do we know `%rdi <= %rsi`? This time for unsigned values
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`

# How Should `cmpq` Set Condition Codes?

`cmpq        %rsi, %rdi`

- How do we know `%rdi <= %rsi`? This time for unsigned values
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`

**ZF** Zero Flag (result is zero)



**ZF**

# How Should `cmpq` Set Condition Codes?

**`cmpq        %rsi, %rdi`**

- How do we know `%rdi <= %rsi`? This time for unsigned values
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`
- `%rdi < %rsi` if a carry is generated during subtraction

**ZF** Zero Flag (result is zero)



**ZF**

# How Should `cmpq` Set Condition Codes?

`cmpq        %rsi, %rdi`

- How do we know `%rdi <= %rsi`? This time for unsigned values
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`
- `%rdi < %rsi` if a carry is generated during subtraction

001	←	1
-) 111	←	7
<hr/>		
C010		

**ZF** Zero Flag (result is zero)



**ZF**



# How Should `cmpq` Set Condition Codes?



**`cmpq      %rsi, %rdi`**

- How do we know `%rdi <= %rsi`? This time for unsigned values
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`
- `%rdi < %rsi` if a carry is generated during subtraction

	001	←	1
-)	111	←	7
<hr/>			
	C010		

**ZF** Zero Flag (result is zero)

**CF** Carry Flag (for unsigned)

	
<b>CF</b>	<b>ZF</b>

# How Should `cmpq` Set Condition Codes?

**`cmpq        %rsi, %rdi`**

- How do we know `%rdi <= %rsi`? This time for unsigned values
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`
- `%rdi < %rsi` if a carry is generated during subtraction

11111111 10000000

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

**ZF** Zero Flag (result is zero)

**CF** Carry Flag (for unsigned)

0	0
<b>CF</b>	<b>ZF</b>

# How Should `cmpq` Set Condition Codes?

**`cmpq`      `%rsi, %rdi`**

- How do we know `%rdi <= %rsi`? This time for unsigned values
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`
- `%rdi < %rsi` if a carry is generated during subtraction

11111111 10000000

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

**ZF** Zero Flag (result is zero)

**CF** Carry Flag (for unsigned)

	<b>10000000</b>	←	<b>128</b>
-)	<b>11111111</b>	←	<b>255</b>
<hr/>			
	<b>c10000001</b>		

<b>0</b>	<b>0</b>
<b>CF</b>	<b>ZF</b>

# How Should `cmpq` Set Condition Codes?

**`cmpq`      `%rsi, %rdi`**

- How do we know `%rdi <= %rsi`? This time for unsigned values
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`
- `%rdi < %rsi` if a carry is generated during subtraction

11111111 10000000

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

	10000000	←	128
-)	11111111	←	255
<hr/>			
	c10000001		

**ZF** Zero Flag (result is zero)

**CF** Carry Flag (for unsigned)

1	0
<b>CF</b>	<b>ZF</b>

# How Should `cmpq` Set Condition Codes?

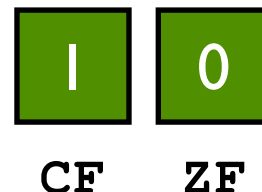
**`cmpq        %rsi, %rdi`**

- How do we know `%rdi <= %rsi`? This time for unsigned values
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`
- `%rdi < %rsi` if a carry is generated during subtraction

- `%rdi <= %rsi` (as unsigned) if and only if:
  - ZF is set, or
  - CF is set
- or simply: **ZF | CF**
- This is what `jbe` checks

**ZF** Zero Flag (result is zero)

**CF** Carry Flag (for unsigned)



# Putting It All Together

# Putting It All Together

- `cmpq` sets all 4 condition codes simultaneously

# Putting It All Together

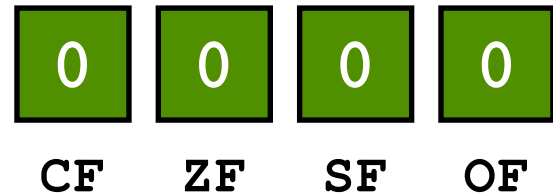
- `cmpq` sets all 4 condition codes simultaneously

**ZF** Zero Flag

**CF** Carry Flag

**SF** Sign Flag

**OF** Overflow Flag (for signed)





# Putting It All Together

- `cmpq` sets all 4 condition codes simultaneously

11111111 10000000  
**cmpq** *0xFF, 0x80*

10000000  
-) 11111111  
-----  
**c**10000001

**ZF** Zero Flag

**CF** Carry Flag

**SF** Sign Flag

**OF** Overflow Flag (for signed)

0	0	0	0
<b>CF</b>	<b>ZF</b>	<b>SF</b>	<b>OF</b>

# Putting It All Together

- `cmpq` sets all 4 condition codes simultaneously

11111111 10000000  
**cmpq** 0xFF, 0x80

10000000  
- ) 11111111  
-----  
c10000001

**ZF** Zero Flag

**CF** Carry Flag

**SF** Sign Flag

**OF** Overflow Flag (for signed)

1	0	1	0
<b>CF</b>	<b>ZF</b>	<b>SF</b>	<b>OF</b>

# Putting It All Together

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

- `cmpq` sets all 4 condition codes simultaneously
- ZF, SF, and OF are used when comparing signed value (e.g., `jle`)

```
11111111 10000000
cmpq 0xFF, 0x80
```

```
      10000000
- )  11111111
-----
c10000001
```

**ZF** Zero Flag

**CF** Carry Flag

**SF** Sign Flag

**OF** Overflow Flag (for signed)

1	0	1	0
CF	ZF	SF	OF

# Putting It All Together

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

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

- `cmpq` sets all 4 condition codes simultaneously
- ZF, SF, and OF are used when comparing signed value (e.g., `jle`)
- ZF, CF are used when comparing unsigned value (e.g., `jbe`)

```
11111111 10000000
cmpq 0xFF, 0x80
```

```
      10000000
- )  11111111
-----
c10000001
```

**ZF** Zero Flag

**CF** Carry Flag

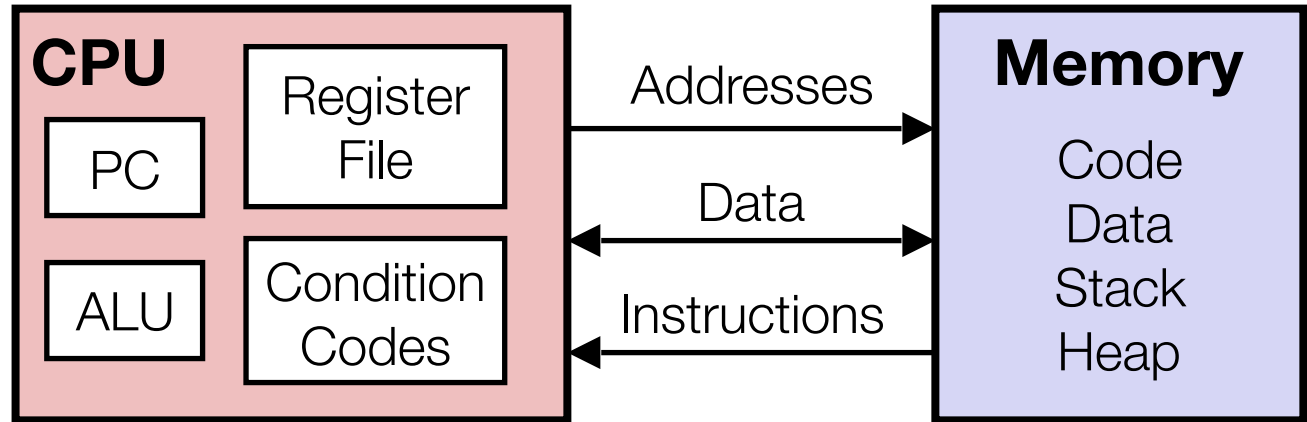
**SF** Sign Flag

**OF** Overflow Flag (for signed)

1	0	1	0
CF	ZF	SF	OF

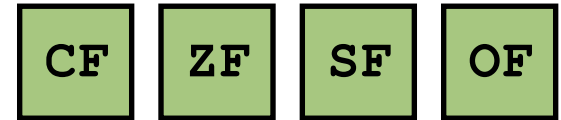
# Condition Codes Hold Test Results

Assembly  
Programmer's  
Perspective  
of a Computer



- Condition Codes

- Hold the status of most recent test
- 4 common condition codes in x86-64
- A set of special registers (more often: bits in one single register)
- Sometimes also called: Status Register, Flag Register



**CF** Carry Flag

**ZF** Zero Flag

**SF** Sign Flag

**OF** Overflow Flag (for signed)

# Jump Instructions

- Jump to different part of code (designated by a label) depending on condition codes

<b>jle</b>	$(SF \wedge OF) \mid ZF$	Less or Equal (Signed)
------------	--------------------------	------------------------

<b>jbe</b>	$CF \mid ZF$	Below or Equal (unsigned)
------------	--------------	---------------------------

# Jump Instructions

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

# Explicit Set Condition Codes: Test

```
testq %rsi, %rdi
```

- Explicit Setting by Test Instruction
  - **test b, a** like computing **a & b**, but instead of setting the result, it sets condition codes
  - Similar to **cmpq b, a**, except **cmpq b, a** does **a - b**
  - **ZF (Zero Flag)**: set if  $a \& b == 0$
  - **SF (Sign Flag)**: set if  $a \& b < 0$
  - OF and CF are always set to 0



# Implicit Set Condition Codes

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

# Implicit Set Condition Codes

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

- Arithmetic instructions implicitly set condition codes (think of it as side effect)

# Implicit 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 (unsigned overflow)

# Implicit 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 (unsigned overflow)
  - **ZF** set if `%rax + %rbx == 0`

# Implicit 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 (unsigned overflow)
  - **ZF** set if `%rax + %rbx == 0`
  - **SF** set if `%rax + %rbx < 0`

# Implicit 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 (unsigned overflow)
  - **ZF** set if `%rax + %rbx == 0`
  - **SF** set if `%rax + %rbx < 0`
  - **OF** set if `%rax + %rbx` as signed numbers overflows

# Implicit 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 (unsigned overflow)
  - **ZF** set if `%rax + %rbx == 0`
  - **SF** set if `%rax + %rbx < 0`
  - **OF** set if `%rax + %rbx` as signed numbers overflows
    - `(%rax > 0 && %rbx > 0 && (%rax + %rbx) < 0) || (%rax < 0 && %rbx < 0 && (%rax + %rbx) >= 0)`

# Implicit 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 (unsigned overflow)
  - **ZF** set if `%rax + %rbx == 0`
  - **SF** set if `%rax + %rbx < 0`
  - **OF** set if `%rax + %rbx` as signed numbers overflows
    - $(\%rax > 0 \ \&\& \ \%rbx > 0 \ \&\& \ (\%rax + \%rbx) < 0) \ ||$   
 $(\%rax < 0 \ \&\& \ \%rbx < 0 \ \&\& \ (\%rax + \%rbx) \geq 0)$

**addq 0xFF, 0x80**

```
      10000000
+) 11111111
-----
  c01111111
```

0	0	0	0
CF	ZF	SF	OF



# Implicit 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 (unsigned overflow)
  - **ZF** set if `%rax + %rbx == 0`
  - **SF** set if `%rax + %rbx < 0`
  - **OF** set if `%rax + %rbx` as signed numbers overflows
    - $(\%rax > 0 \ \&\& \ \%rbx > 0 \ \&\& \ (\%rax + \%rbx) < 0) \ ||$   
 $(\%rax < 0 \ \&\& \ \%rbx < 0 \ \&\& \ (\%rax + \%rbx) \geq 0)$

**addq 0xFF, 0x80**

```
      10000000
+) 11111111
-----
  c01111111
```

<b>1</b>	<b>0</b>	<b>0</b>	<b>0</b>
<b>CF</b>	<b>ZF</b>	<b>SF</b>	<b>OF</b>

# Implicit 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 (unsigned overflow)
  - **ZF** set if `%rax + %rbx == 0`
  - **SF** set if `%rax + %rbx < 0`
  - **OF** set if `%rax + %rbx` as signed numbers overflows
    - $(\%rax > 0 \ \&\& \ \%rbx > 0 \ \&\& \ (\%rax + \%rbx) < 0) \ ||$   
 $(\%rax < 0 \ \&\& \ \%rbx < 0 \ \&\& \ (\%rax + \%rbx) \geq 0)$

**addq 0xFF, 0x80**

```
      10000000
+) 11111111
-----
  c01111111
```

<b>I</b>	<b>0</b>	<b>0</b>	<b>I</b>
<b>CF</b>	<b>ZF</b>	<b>SF</b>	<b>OF</b>

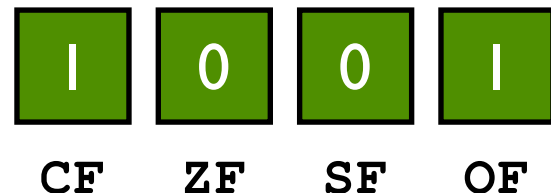
# Implicit Set Condition Codes

**addq %rax, %rbx**

- Arithmetic instructions implicitly set condition codes (think of it as side effect)
  - **CF** set if `%rax + %rbx` generates a carry (unsigned overflow)
  - **ZF** set if `%rax + %rbx == 0`
  - **SF** set if `%rax + %rbx < 0`
  - **OF** set if `%rax + %rbx` as signed numbers overflows
    - $(\%rax > 0 \ \&\& \ \%rbx > 0 \ \&\& \ (\%rax + \%rbx) < 0) \ ||$   
 $(\%rax < 0 \ \&\& \ \%rbx < 0 \ \&\& \ (\%rax + \%rbx) \geq 0)$

**addq 0xFF, 0x80**

**jle .L4**



# Implicit 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 (unsigned overflow)
  - **ZF** set if `%rax + %rbx == 0`
  - **SF** set if `%rax + %rbx < 0`
  - **OF** set if `%rax + %rbx` as signed numbers overflows
    - `(%rax > 0 && %rbx > 0 && (%rax + %rbx) < 0) || (%rax < 0 && %rbx < 0 && (%rax + %rbx) >= 0)`

```
if ( (x+y) < 0 ) {  
    ...  
}
```

```
addq 0xFF, 0x80  
jle .L4
```

I	0	0	I
CF	ZF	SF	OF

# Implicit 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 (unsigned overflow)

- ZF** set if `%rax + %rbx == 0`

- SF** set if `%rax + %rbx < 0`

- OF** set if `%rax + %rbx` as signed numbers overflows

- `(%rax > 0 && %rbx > 0 && (%rax + %rbx) < 0) ||  
(%rax < 0 && %rbx < 0 && (%rax + %rbx) >= 0)`

## Questions?

```
if ( (x+y) < 0 ) {  
    ...  
}
```

```
addq 0xFF, 0x80  
jle .L4
```

I	0	0	I
CF	ZF	SF	OF

# Today: Compute and Control Instructions

- Arithmetic & logical operations
- Control: Condition codes
- Conditional branches (**if... else...**)
- Loops (**for, do... while**)
- 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;  
  }
```

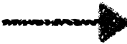
# “Do-While” Loop Assembly

```
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 <b>x</b>
%rax	<b>result</b>


```
    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 <b>x</b>
%rax	<b>result</b>




```
    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 <b>x</b>
%rax	<b>result</b>




```
    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 <b>x</b>
%rax	<b>result</b>



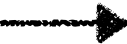
```
    movl    $0, %rax    # result = 0
.L2:                                # loop:
    movq    %rdi, %rdx
    andl    $1, %rdx    # t = x & 0x1
    addq    %rdx, %rax   # result += t
    shrq    $1, %rdi    # x >>= 1
    jne     .L2          # if (x) goto loop
    ret
```

# “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 <b>x</b>
%rax	<b>result</b>



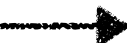
```
    movl    $0, %rax    # result = 0
.L2:                                # loop:
    movq    %rdi, %rdx
    andl    $1, %rdx    # t = x & 0x1
    addq    %rdx, %rax   # result += t
    shrq    $1, %rdi    # x >>= 1
    jne     .L2          # if (x) goto loop
    ret
```

# “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 <b>x</b>
%rax	<b>result</b>



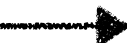
```
    movl    $0, %rax    # result = 0
.L2:                                # loop:
    movq    %rdi, %rdx
    andl    $1, %rdx    # t = x & 0x1
    addq    %rdx, %rax   # result += t
    shrq    $1, %rdi    # x >>= 1
    jne     .L2          # if (x) goto loop
    ret
```

# “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;
}
```

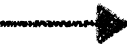
Register	Use(s)
%rdi	Argument <b>x</b>
%rax	<b>result</b>



```
    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
```

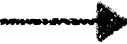


# “Do-While” Loop Assembly



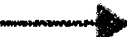
```
long pcount_goto
(unsigned long x) {
    long result = 0;
loop:
    result += x & 0x1;
    x >>= 1;
    if(x) goto loop;
    return result;
}
```

Register	Use(s)
%rdi	Argument <b>x</b>
%rax	<b>result</b>




```
    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



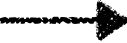
```
long pcount_goto
(unsigned long x) {
    long result = 0;
loop:
    result += x & 0x1;
    x >>= 1;
    if(x) goto loop;
    return result;
}
```

Register	Use(s)
%rdi	Argument <b>x</b>
%rax	<b>result</b>




```
    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 <b>x</b>
%rax	<b>result</b>



```
    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  
}
```

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

# “For” Loop Example

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

init

i = 0

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



# “For” Loop Example

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

init

i = 0

test

i < WSIZE

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

# “For” Loop Example

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

init

i = 0

test

i < WSIZE

update

i++

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

# “For” Loop Example

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

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

init

i = 0

test

i < WSIZE

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
```

# 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
```

Questions?

# Today: Compute and Control Instructions

- Arithmetic & logical operations
- Control: Condition codes
- Conditional branches (**if... else...**)
- Loops (**for, do... while**)
- 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

•  
•  
•

Targ $n-1$ : 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

JTab:

Targ0
Targ1
Targ2
•
•
•
Targn-1

Jump Targets

Targ0: Code Block 0

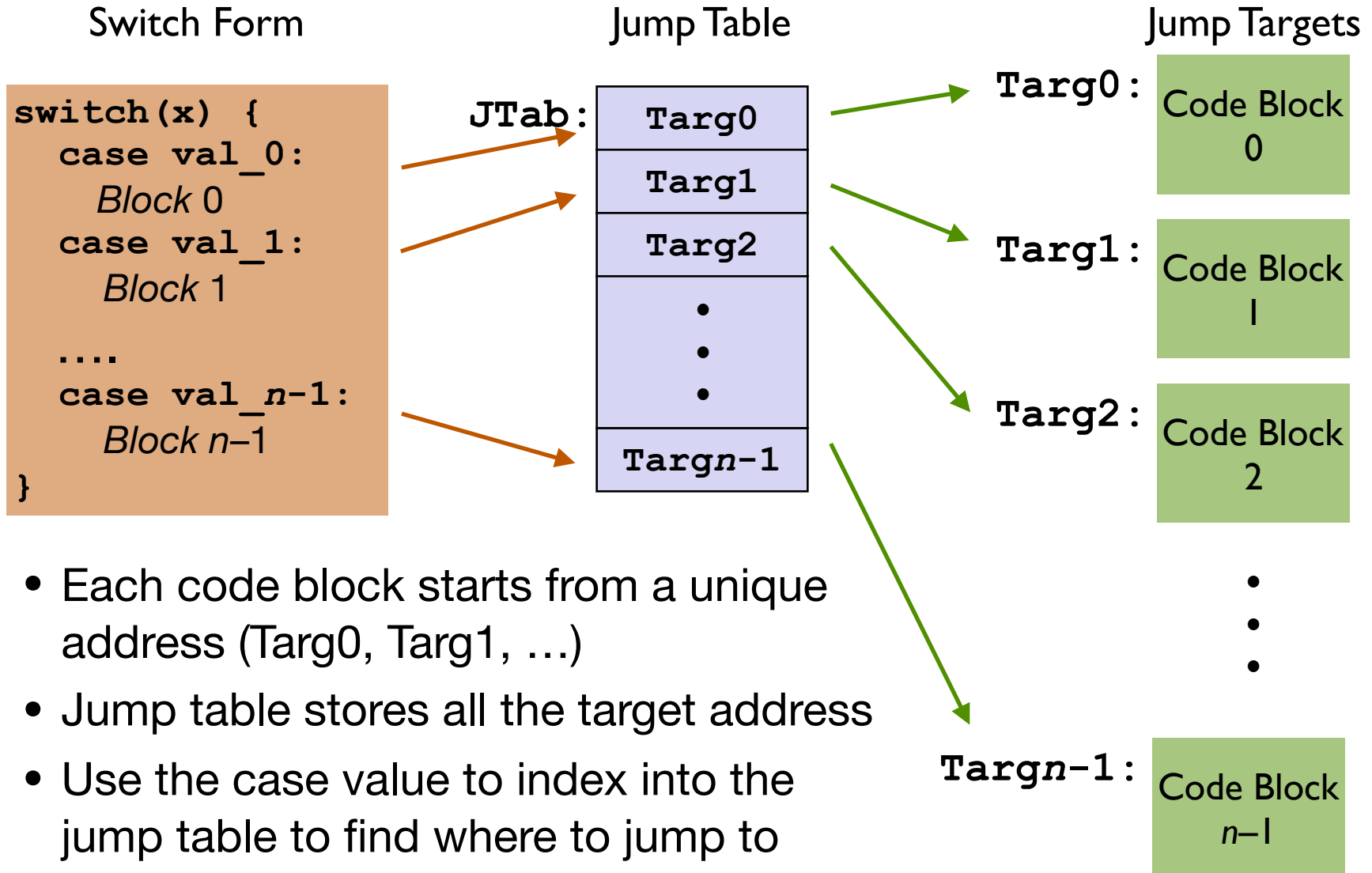
Targ1: Code Block 1

Targ2: Code Block 2

•  
•  
•

Targn-1: Code Block n-1

# Implementing Switch Using Jump Table



# Jump Table in Assembly

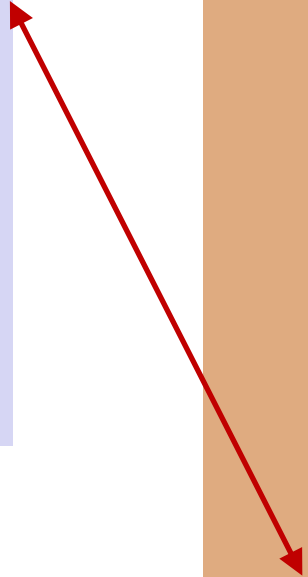
```
.section .rodata
.align 8
.L4:
    .quad .L8 # x = 0
    .quad .L3 # x = 1
    .quad .L5 # x = 2
    .quad .L9 # x = 3
    .quad .L8 # x = 4
    .quad .L7 # x = 5
    .quad .L7 # x = 6
```

```
switch(x) {
case 1:      // .L3
    w = y*z;
    break;
case 2:      // .L5
    w = y/z;
    /* Fall Through */
case 3:      // .L9
    w += z;
    break;
case 5:
case 6:      // .L7
    w -= z;
    break;
default:    // .L8
    w = 2;
}
```

# Jump Table in Assembly

```
.section .rodata
.align 8
.L4:
.quad .L8 # x = 0
.quad .L3 # x = 1
.quad .L5 # x = 2
.quad .L9 # x = 3
.quad .L8 # x = 4
.quad .L7 # x = 5
.quad .L7 # x = 6
```

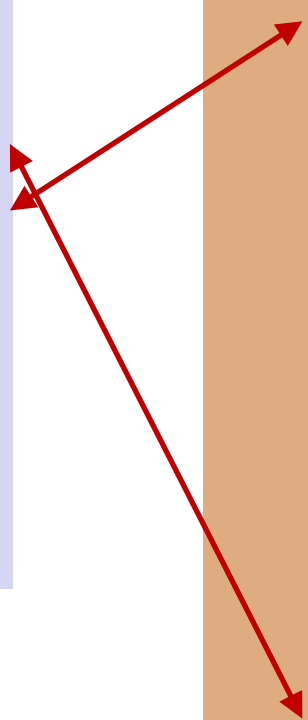
```
switch(x) {
case 1:      // .L3
    w = y*z;
    break;
case 2:      // .L5
    w = y/z;
    /* Fall Through */
case 3:      // .L9
    w += z;
    break;
case 5:
case 6:      // .L7
    w -= z;
    break;
default:    // .L8
    w = 2;
}
```



# Jump Table in Assembly

```
.section .rodata
.align 8
.L4:
.quad .L8 # x = 0
.quad .L3 # x = 1
.quad .L5 # x = 2
.quad .L9 # x = 3
.quad .L8 # x = 4
.quad .L7 # x = 5
.quad .L7 # x = 6
```

```
switch(x) {
case 1:      // .L3
    w = y*z;
    break;
case 2:      // .L5
    w = y/z;
    /* Fall Through */
case 3:      // .L9
    w += z;
    break;
case 5:
case 6:      // .L7
    w -= z;
    break;
default:     // .L8
    w = 2;
}
```



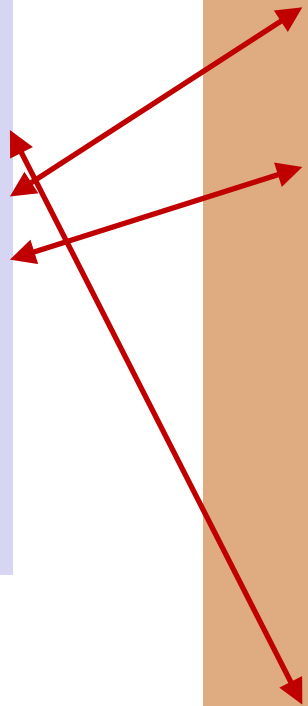
The diagram illustrates the mapping between the assembly jump table and the C switch statement. Red arrows show the following connections:

- From `.L3` in the jump table to `case 1:` in the switch statement.
- From `.L5` in the jump table to `case 2:` in the switch statement.
- From `.L9` in the jump table to `case 3:` in the switch statement.
- From `.L8` in the jump table to `case 5:` and `case 6:` in the switch statement.
- From `.L7` in the jump table to `case 5:` and `case 6:` in the switch statement.

# Jump Table in Assembly

```
.section .rodata
.align 8
.L4:
.quad .L8 # x = 0
.quad .L3 # x = 1
.quad .L5 # x = 2
.quad .L9 # x = 3
.quad .L8 # x = 4
.quad .L7 # x = 5
.quad .L7 # x = 6
```

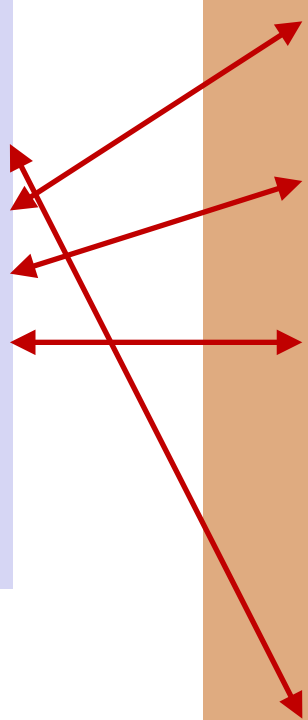
```
switch(x) {
case 1:      // .L3
    w = y*z;
    break;
case 2:      // .L5
    w = y/z;
    /* Fall Through */
case 3:      // .L9
    w += z;
    break;
case 5:
case 6:      // .L7
    w -= z;
    break;
default:     // .L8
    w = 2;
}
```



# Jump Table in Assembly

```
.section .rodata
.align 8
.L4:
.quad .L8 # x = 0
.quad .L3 # x = 1
.quad .L5 # x = 2
.quad .L9 # x = 3
.quad .L8 # x = 4
.quad .L7 # x = 5
.quad .L7 # x = 6
```

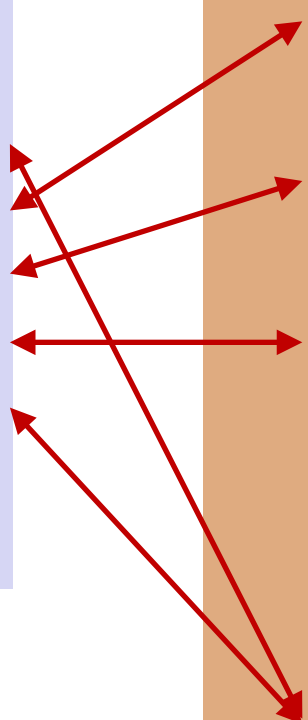
```
switch(x) {
case 1:      // .L3
    w = y*z;
    break;
case 2:      // .L5
    w = y/z;
    /* Fall Through */
case 3:      // .L9
    w += z;
    break;
case 5:
case 6:      // .L7
    w -= z;
    break;
default:    // .L8
    w = 2;
}
```



# Jump Table in Assembly

```
.section .rodata
.align 8
.L4:
.quad .L8 # x = 0
.quad .L3 # x = 1
.quad .L5 # x = 2
.quad .L9 # x = 3
.quad .L8 # x = 4
.quad .L7 # x = 5
.quad .L7 # x = 6
```

```
switch(x) {
case 1:      // .L3
    w = y*z;
    break;
case 2:      // .L5
    w = y/z;
    /* Fall Through */
case 3:      // .L9
    w += z;
    break;
case 5:
case 6:      // .L7
    w -= z;
    break;
default:    // .L8
    w = 2;
}
```





# Jump Table in Assembly

```
.section .rodata
.align 8
```

```
.L4:
```

```
.quad .L8 # x = 0
```

```
.quad .L3 # x = 1
```

```
.quad .L5 # x = 2
```

```
.quad .L9 # x = 3
```

```
.quad .L8 # x = 4
```

```
.quad .L7 # x = 5
```

```
.quad .L7 # x = 6
```

```
switch(x) {
case 1:      // .L3
    w = y*z;
    break;
case 2:      // .L5
    w = y/z;
    /* Fall Through */
case 3:      // .L9
    w += z;
    break;
case 5:
case 6:      // .L7
    w -= z;
    break;
default:     // .L8
    w = 2;
}
```

# Jump Table in Assembly

```
.section .rodata
.align 8
```

```
.L4:
```

```
.quad .L8 # x = 0
.quad .L3 # x = 1
.quad .L5 # x = 2
.quad .L9 # x = 3
.quad .L8 # x = 4
.quad .L7 # x = 5
.quad .L7 # x = 6
```

```
switch(x) {
case 1:      // .L3
    w = y*z;
    break;
case 2:      // .L5
    w = y/z;
    /* Fall Through */
case 3:      // .L9
    w += z;
    break;
case 5:
case 6:      // .L7
    w -= z;
    break;
default:    // .L8
    w = 2;
}
```

# Assembly Directives (Pseudo-Ops)

```
.section .rodata
.align 8
.L4:
.quad .L8 # x = 0
.quad .L3 # x = 1
.quad .L5 # x = 2
.quad .L9 # x = 3
.quad .L8 # x = 4
.quad .L7 # x = 5
.quad .L7 # 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)

```
.section .rodata
.align 8
.L4:
.quad .L8 # x = 0
.quad .L3 # x = 1
.quad .L5 # x = 2
.quad .L9 # x = 3
.quad .L8 # x = 4
.quad .L7 # x = 5
.quad .L7 # 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.

# Assembly Directives (Pseudo-Ops)

```
.section .rodata
.align 8
.L4:
.quad .L8 # x = 0
.quad .L3 # x = 1
.quad .L5 # x = 2
.quad .L9 # x = 3
.quad .L8 # x = 4
.quad .L7 # x = 5
.quad .L7 # 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)
- **.align**: tells the assembler that addresses of the the following data will be aligned to 8 bytes

- Directives:

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

# Assembly Directives (Pseudo-Ops)

```
.section    .rodata
.align 8
.L4:
.quad .L8 # x = 0
.quad .L3 # x = 1
.quad .L5 # x = 2
.quad .L9 # x = 3
.quad .L8 # x = 4
.quad .L7 # x = 5
.quad .L7 # x = 6
```

- Directives:

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

- **.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)
- **.align**: tells the assembler that addresses of the the following data will be aligned to 8 bytes
- **.section**: denotes different parts of the object file

# Assembly Directives (Pseudo-Ops)

```
.section    .rodata
    .align  8
.L4:
    .quad   .L8 # x = 0
    .quad   .L3 # x = 1
    .quad   .L5 # x = 2
    .quad   .L9 # x = 3
    .quad   .L8 # x = 4
    .quad   .L7 # x = 5
    .quad   .L7 # x = 6
```

- Directives:

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

- **.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)
- **.align**: tells the assembler that addresses of the the following data will be aligned to 8 bytes
- **.section**: denotes different parts of the object file
- **.rodata**: read-only data section

# Code Blocks (x == 1)

```
.section .rodata
.align 8
.L4:
    .quad .L8 # x = 0
    .quad .L3 # x = 1
    .quad .L5 # x = 2
    .quad .L9 # x = 3
    .quad .L8 # x = 4
    .quad .L7 # x = 5
    .quad .L7 # x = 6
```



# Code Blocks (x == 1)

```
.section .rodata
.align 8
.L4:
    .quad .L8 # x = 0
    .quad .L3 # x = 1
    .quad .L5 # x = 2
    .quad .L9 # x = 3
    .quad .L8 # x = 4
    .quad .L7 # x = 5
    .quad .L7 # x = 6
```

```
switch(x) {
case 1:      // .L3
    w = y*z;
    break;
    ...
}
```

# Code Blocks (x == 1)

```
.section .rodata
.align 8
.L4:
    .quad .L8 # x = 0
    .quad .L3 # x = 1
    .quad .L5 # x = 2
    .quad .L9 # x = 3
    .quad .L8 # x = 4
    .quad .L7 # x = 5
    .quad .L7 # x = 6
```

```
switch(x) {
case 1:      // .L3
    w = y*z;
    break;
    ...
}
```

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

# Code Blocks (x == 1)

```
.section .rodata
.align 8
.L4:
    .quad .L8 # x = 0
    .quad .L3 # x = 1
    .quad .L5 # x = 2
    .quad .L9 # x = 3
    .quad .L8 # x = 4
    .quad .L7 # x = 5
    .quad .L7 # x = 6
```

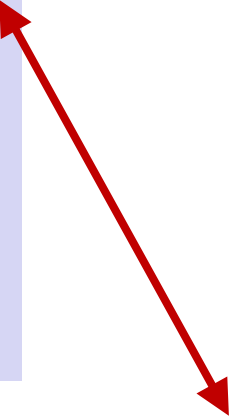
```
switch(x) {
case 1:      // .L3
    w = y*z;
    break;
    ...
}
```

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

```
.L3:
    movq    %rsi, %rax    # y
    imulq   %rdx, %rax    # y*z
    jmp     .done
```

# Code Blocks (x == 1)

```
.section .rodata
.align 8
.L4:
    .quad .L8 # x = 0
    .quad .L3 # x = 1
    .quad .L5 # x = 2
    .quad .L9 # x = 3
    .quad .L8 # x = 4
    .quad .L7 # x = 5
    .quad .L7 # x = 6
```



```
switch(x) {
case 1:      // .L3
    w = y*z;
    break;
    ...
}
```

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

```
.L3:
    movq    %rsi, %rax    # y
    imulq   %rdx, %rax    # y*z
    jmp     .done
```

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

```
.section .rodata
    .align 8
.L4:
    .quad .L8 # x = 0
    .quad .L3 # x = 1
    .quad .L5 # x = 2
    .quad .L9 # x = 3
    .quad .L8 # x = 4
    .quad .L7 # x = 5
    .quad .L7 # x = 6
```

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

```
.section .rodata
.align 8
.L4:
    .quad .L8 # x = 0
    .quad .L3 # x = 1
    .quad .L5 # x = 2
    .quad .L9 # x = 3
    .quad .L8 # x = 4
    .quad .L7 # x = 5
    .quad .L7 # x = 6
```

```
switch(x) {
...
case 2:      // .L5
    w = y/z;
    /* Fall Through */
case 3:      // .L9
    w += z;
    break;
...
}
```

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

```
.section .rodata
.align 8
.L4:
    .quad .L8 # x = 0
    .quad .L3 # x = 1
    .quad .L5 # x = 2
    .quad .L9 # x = 3
    .quad .L8 # x = 4
    .quad .L7 # x = 5
    .quad .L7 # x = 6
```

```
switch(x) {
...
case 2:          // .L5
    w = y/z;
    /* Fall Through */
case 3:          // .L9
    w += z;
    break;
...
}
```

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

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

```
.section .rodata
.align 8
.L4:
    .quad .L8 # x = 0
    .quad .L3 # x = 1
    .quad .L5 # x = 2
    .quad .L9 # x = 3
    .quad .L8 # x = 4
    .quad .L7 # x = 5
    .quad .L7 # x = 6
```

```
switch(x) {
...
case 2:          // .L5
    w = y/z;
    /* Fall Through */
case 3:          // .L9
    w += z;
    break;
...
}
```

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

```
.L5:                                # Case 2
    movq    %rsi, %rax
    cqto
    idivq   %rcx                    # y/z

.L9:                                # Case 3
    addq    %rcx, %rax              # w += z
    jmp     .done
```



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

```
.section .rodata
.align 8
.L4:
    .quad .L8 # x = 0
    .quad .L3 # x = 1
    .quad .L5 # x = 2
    .quad .L9 # x = 3
    .quad .L8 # x = 4
    .quad .L7 # x = 5
    .quad .L7 # x = 6
```

```
switch(x) {
...
case 2:          // .L5
    w = y/z;
    /* Fall Through */
case 3:          // .L9
    w += z;
    break;
...
}
```

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

```
.L5:                                # Case 2
    movq    %rsi, %rax
    cqto
    idivq   %rcx                    # y/z
.L9:                                # Case 3
    addq    %rcx, %rax             # w += z
    jmp     .done
```

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

```
.section .rodata
    .align 8
.L4:
    .quad .L8 # x = 0
    .quad .L3 # x = 1
    .quad .L5 # x = 2
    .quad .L9 # x = 3
    .quad .L8 # x = 4
    .quad .L7 # x = 5
    .quad .L7 # x = 6
```

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

```
.section .rodata
.align 8
.L4:
    .quad .L8 # x = 0
    .quad .L3 # x = 1
    .quad .L5 # x = 2
    .quad .L9 # x = 3
    .quad .L8 # x = 4
    .quad .L7 # x = 5
    .quad .L7 # x = 6
```

```
switch(x) {
...
case 5:    // .L7
case 6:    // .L7
    w -= z;
    break;
default:  // .L8
    w = 2;
}
```

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

```
.section .rodata
.align 8
.L4:
    .quad .L8 # x = 0
    .quad .L3 # x = 1
    .quad .L5 # x = 2
    .quad .L9 # x = 3
    .quad .L8 # x = 4
    .quad .L7 # x = 5
    .quad .L7 # x = 6
```

```
switch(x) {
...
case 5:    // .L7
case 6:    // .L7
    w -= z;
    break;
default:  // .L8
    w = 2;
}
```

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

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

```
.section .rodata
.align 8
.L4:
.quad .L8 # x = 0
.quad .L3 # x = 1
.quad .L5 # x = 2
.quad .L9 # x = 3
.quad .L8 # x = 4
.quad .L7 # x = 5
.quad .L7 # x = 6
```

```
switch(x) {
...
case 5: // .L7
case 6: // .L7
    w -= z;
    break;
default: // .L8
    w = 2;
}
```

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

```
.L7:                                # Case 5,6
    subq    %rdx, %rax # w -= z
    jmp     .done
.L8:                                # Default:
    movl    $2, %eax  # 2
    jmp     .done
```

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

```
.section .rodata
.align 8
.L4:
.quad .L8 # x = 0
.quad .L3 # x = 1
.quad .L5 # x = 2
.quad .L9 # x = 3
.quad .L8 # x = 4
.quad .L7 # x = 5
.quad .L7 # x = 6
```

```
switch(x) {
...
case 5: // .L7
case 6: // .L7
    w -= z;
    break;
default: // .L8
    w = 2;
}
```

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

```
.L7:                                # Case 5,6
    subq  %rdx, %rax # w -= z
    jmp   .done
.L8:                                # Default:
    movl  $2, %eax  # 2
    jmp   .done
```

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

```
.section .rodata
.align 8
.L4:
.quad .L8 # x = 0
.quad .L3 # x = 1
.quad .L5 # x = 2
.quad .L9 # x = 3
.quad .L8 # x = 4
.quad .L7 # x = 5
.quad .L7 # x = 6
```

```
switch(x) {
...
case 5: // .L7
case 6: // .L7
    w -= z;
    break;
default: // .L8
    w = 2;
}
```

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

```
.L7:                                # Case 5,6
    subq %rdx, %rax # w -= z
    jmp  .done
.L8:                                # Default:
    movl $2, %eax  # 2
    jmp  .done
```

# Jump Table and Jump Targets

## Jump Table

```
.section .rodata
.align 8
.L4:
    .quad .L8 # x = 0
    .quad .L3 # x = 1
    .quad .L5 # x = 2
    .quad .L9 # x = 3
    .quad .L8 # x = 4
    .quad .L7 # x = 5
    .quad .L7 # x = 6
```

## Jump Targets

```
.L3:                                # Case 1
    movq    %rsi, %rax
    imulq    %rdx, %rax
    jmp     .done

.L5:                                # Case 2
    movq    %rsi, %rax
    cqto
    idivq    %rcx

.L9:                                # Case 3
    addq     %rcx, %rax
    jmp     .done

.L7:                                # Case 5,6
    subq     %rdx, %rax
    jmp     .done

.L8:                                # 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

JTab:

Targ0
Targ1
Targ2
•
•
•
Targn-1

Jump Targets

Targ0: Code Block 0

Targ1: Code Block 1

Targ2: Code Block 2

•  
•  
•

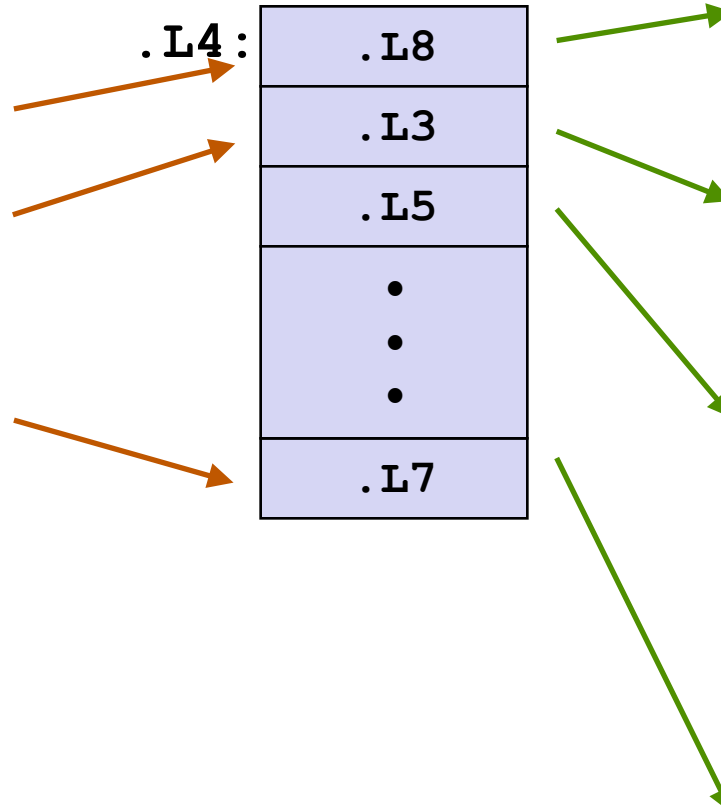
Targn-1: Code Block n-1

# 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

.L8: Code Block 0

.L3: Code Block 1

.L5: Code Block 2

⋮

.L7: 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

.L4:

.L8
.L3
.L5
•
•
•
.L7

Jump Targets

.L8: Code Block 0

.L3: Code Block 1

.L5: Code Block 2

•  
•  
•

.L7: Code Block n-1

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

# Indirect Jump Instruction

```
.section .rodata
    .align 8
.L4:
    .quad .L8 # x = 0
    .quad .L3 # x = 1
    .quad .L5 # x = 2
    .quad .L9 # x = 3
    .quad .L8 # x = 4
    .quad .L7 # x = 5
    .quad .L7 # x = 6
```

# Indirect Jump Instruction

Address we want =  $.L4 + 8 * x$

```
.section .rodata
.align 8
.L4:
    .quad .L8 # x = 0
    .quad .L3 # x = 1
    .quad .L5 # x = 2
    .quad .L9 # x = 3
    .quad .L8 # x = 4
    .quad .L7 # x = 5
    .quad .L7 # x = 6
```

# Indirect Jump Instruction

Address we want =  $.L4 + 8 * x$

```
.section .rodata
.align 8
.L4:
    .quad .L8 # x = 0
    .quad .L3 # x = 1
    .quad .L5 # x = 2
    .quad .L9 # x = 3
    .quad .L8 # x = 4
    .quad .L7 # x = 5
    .quad .L7 # x = 6
```

```
# assume x in %rdi
movq    .L4(,%rdi,8), %rax
jmp     *%rax
```

# Indirect Jump Instruction

Address we want =  $.L4 + 8 * x$

```
.section .rodata
.align 8
.L4:
    .quad .L8 # x = 0
    .quad .L3 # x = 1
    .quad .L5 # x = 2
    .quad .L9 # x = 3
    .quad .L8 # x = 4
    .quad .L7 # x = 5
    .quad .L7 # x = 6
```

```
# assume x in %rdi
movq .L4(,%rdi,8), %rax
jmp  *%rax
```

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

# Indirect Jump Instruction

Address we want =  $.L4 + 8 * x$

```
.section .rodata
.align 8
.L4:
    .quad .L8 # x = 0
    .quad .L3 # x = 1
    .quad .L5 # x = 2
    .quad .L9 # x = 3
    .quad .L8 # x = 4
    .quad .L7 # x = 5
    .quad .L7 # x = 6
```

```
# assume x in %rdi
movq .L4(,%rdi,8), %rax
jmp  *%rax
```

- Indirect Jump: **jmp \*%rax**
  - `%rax` specifies the address to jump to ( $PC = \%rax$ )
- Direct Jump (**jmp .L4**), directly specifies the jump address



# Indirect Jump Instruction

Address we want =  $.L4 + 8 * x$

```
.section .rodata
.align 8
.L4:
```

```
.quad .L8 # x = 0
.quad .L3 # x = 1
.quad .L5 # x = 2
.quad .L9 # x = 3
.quad .L8 # x = 4
.quad .L7 # x = 5
.quad .L7 # x = 6
```

```
# assume x in %rdi
movq .L4(,%rdi,8), %rax
jmp  *%rax
```

- Indirect Jump: **jmp \*%rax**
  - `%rax` specifies the address to jump to ( $PC = \%rax$ )
- Direct Jump (**jmp .L4**), directly specifies the jump address
- Indirect Jump specifies where the jump address is located

# Indirect Jump Instruction

Address we want =  $.L4 + 8 * x$

```
.section .rodata
.align 8
.L4:
```

```
.quad .L8 # x = 0
.quad .L3 # x = 1
.quad .L5 # x = 2
.quad .L9 # x = 3
.quad .L8 # x = 4
.quad .L7 # x = 5
.quad .L7 # x = 6
```

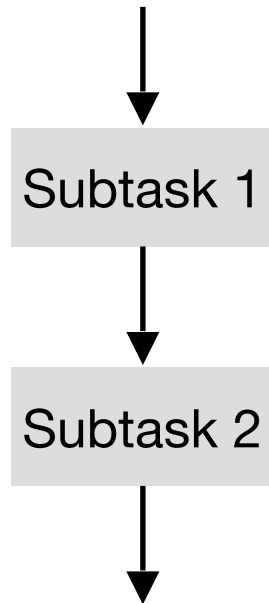
```
# assume x in %rdi
movq .L4(,%rdi,8), %rax
jmp  *%rax
```

- Indirect Jump: **jmp \*%rax**
  - `%rax` specifies the address to jump to ( $PC = \%rax$ )
- Direct Jump (**jmp .L4**), directly specifies the jump address
- Indirect Jump specifies where the jump address is located

```
jmp  *.L4(,%rdi,8)
```

# Summary

## Sequential



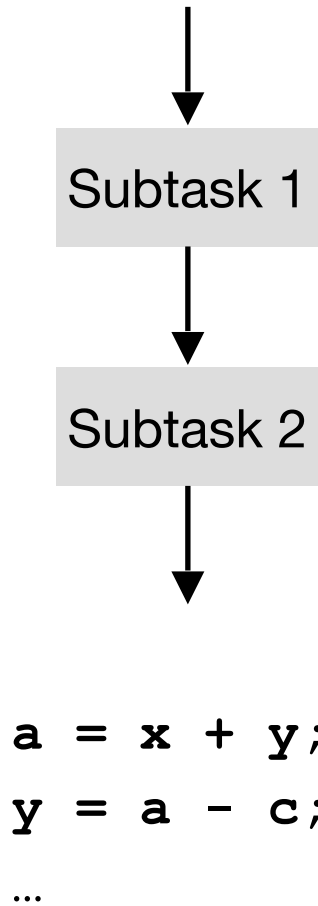
**a = x + y;**

**y = a - c;**

...

# Summary

## Sequential

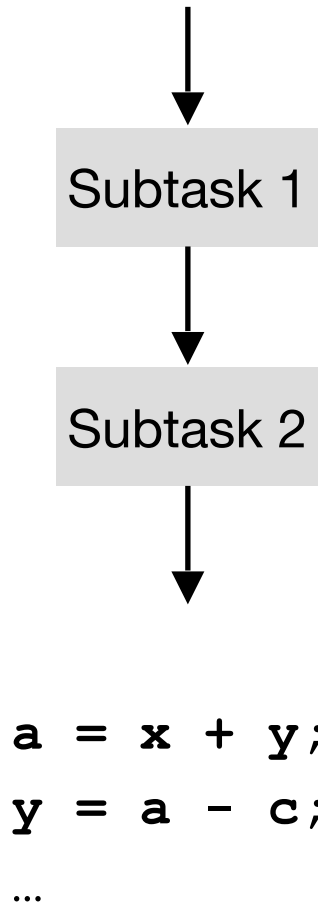


## Memory

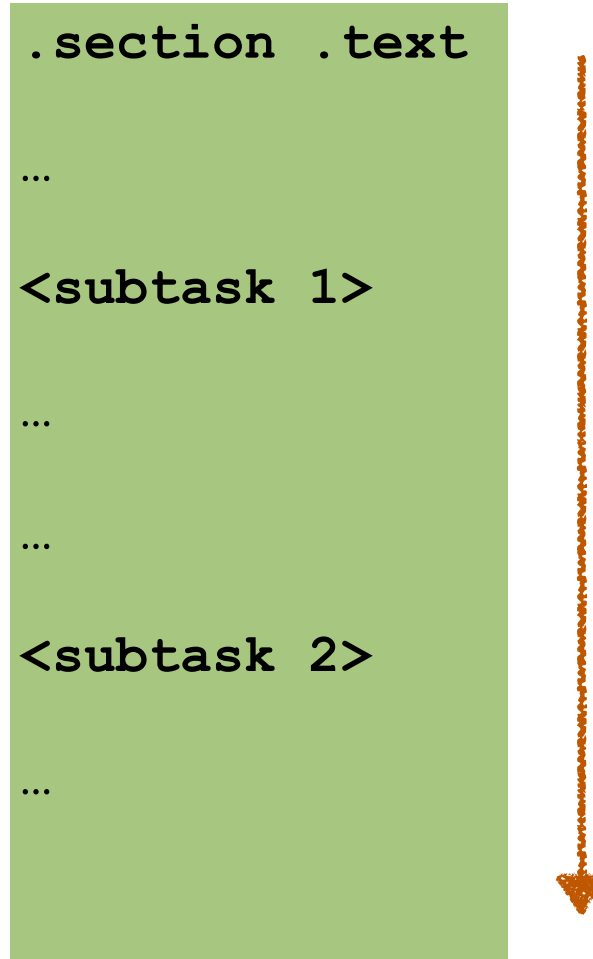
```
.section .text  
...  
<subtask 1>  
...  
...  
<subtask 2>  
...
```

# Summary

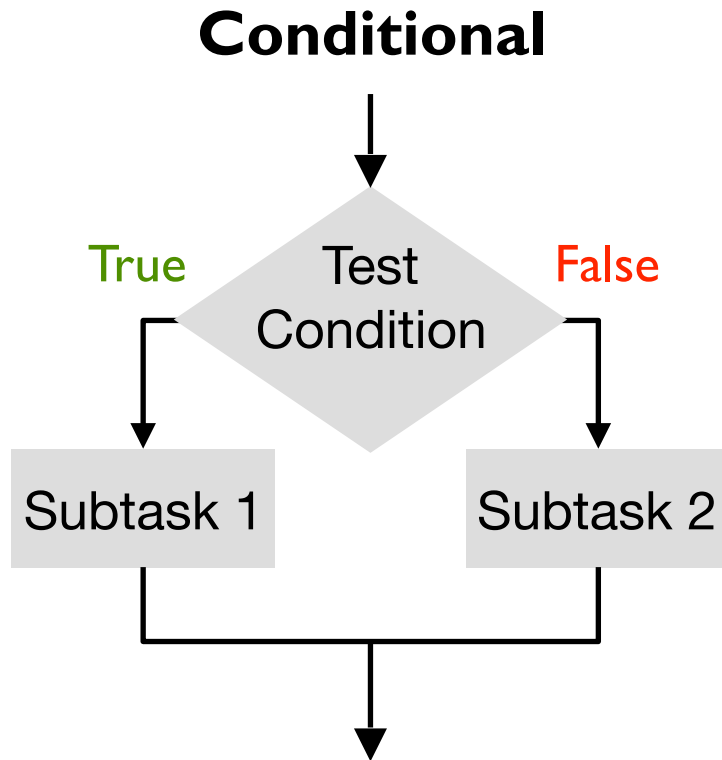
## Sequential



## Memory



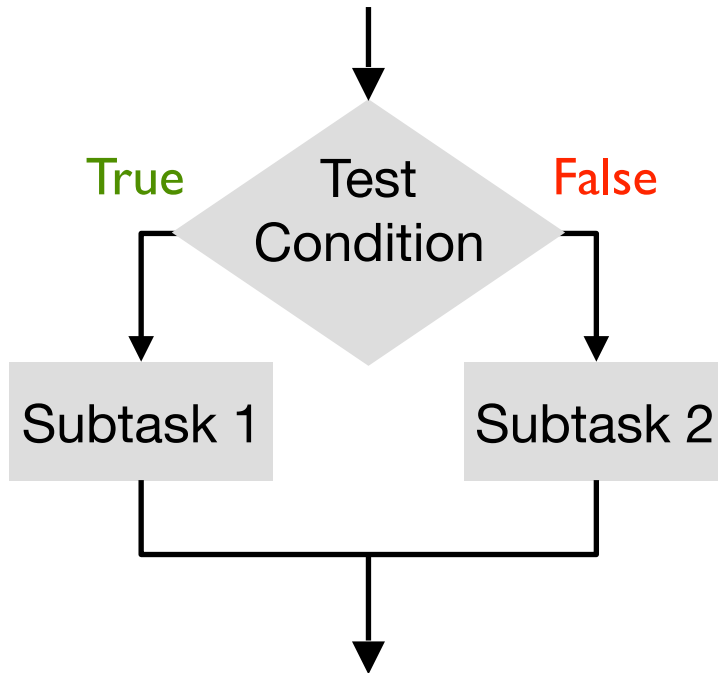
# Summary



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

# Summary

## Conditional



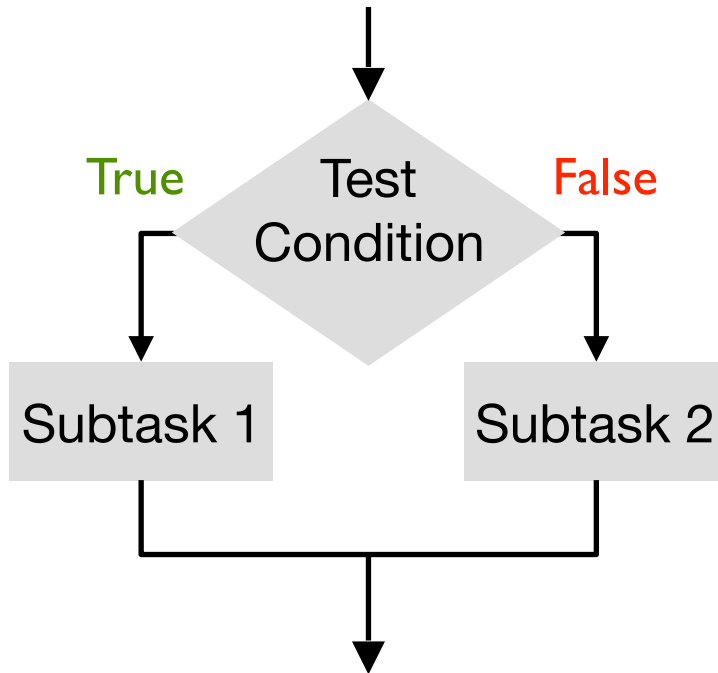
```
if (x > y) r = x - y;  
else r = y - x;
```

## Memory

```
.section .text  
  
...  
cmpq  
jle .L2  
.L1 <subtask 1>  
  
...  
  
jmp .done  
.L2 <subtask 2>  
  
...  
.done
```

# Summary

## Conditional



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

## Memory

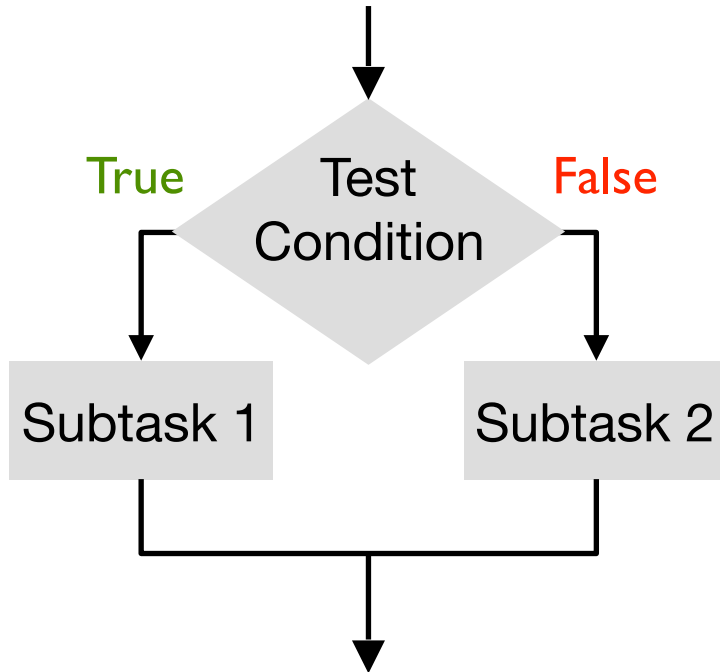
```
.section .text  
  
...  
cmpq  
jle .L2  
.L1 <subtask 1>  
  
...  
  
jmp .done  
.L2 <subtask 2>  
  
...  
.done
```





# Summary

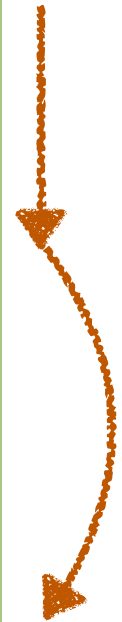
## Conditional



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

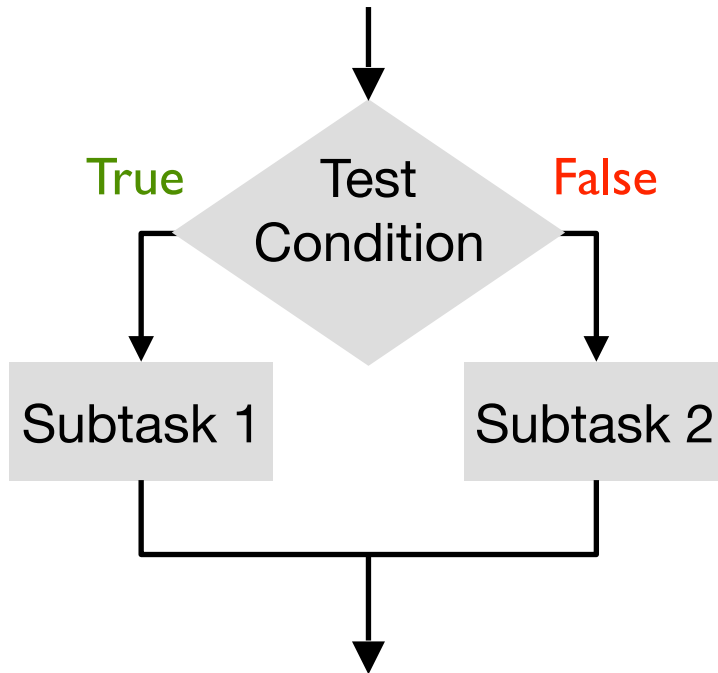
## Memory

```
.section .text  
  
...  
cmpq  
jle .L2  
.L1 <subtask 1>  
  
...  
  
jmp .done  
.L2 <subtask 2>  
  
...  
.done
```



# Summary

## Conditional



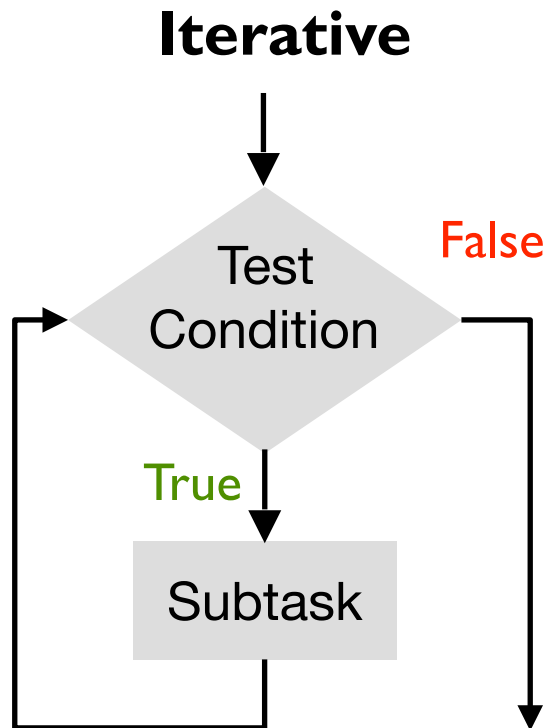
```
if (x > y) r = x - y;  
else r = y - x;
```

## Memory

```
.section .text  
  
...  
cmpq  
jle .L2  
.L1 <subtask 1>  
  
...  
  
jmp .done  
.L2 <subtask 2>  
  
...  
.done
```

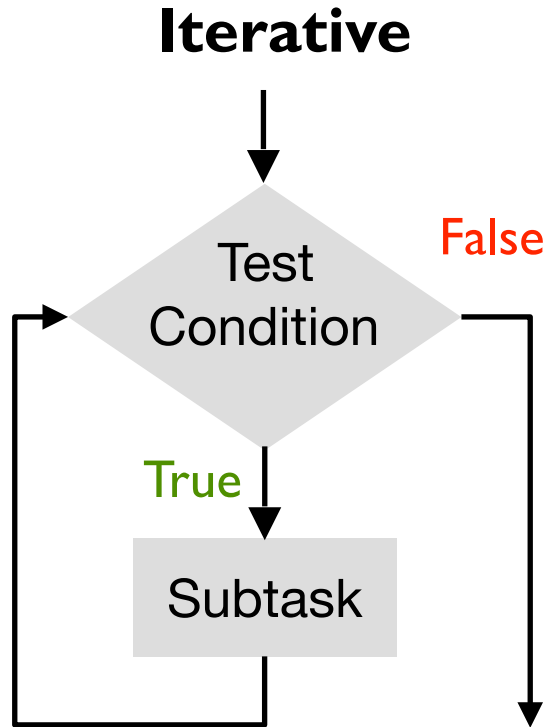


# Summary



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

# Summary



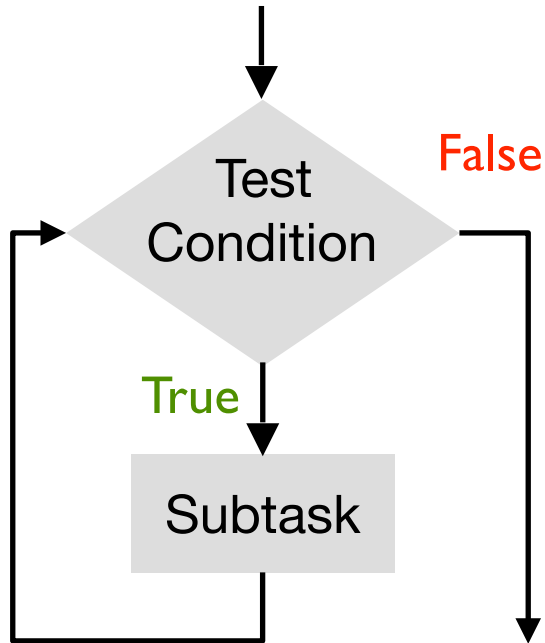
```
while (x > 0) {  
    x-- ;  
}
```

## Memory

```
.section .text  
  
...  
addq  
jmp .L2  
.L1:  
  
...  
    <subtask>  
...  
.L2:  
    cmpq A, B  
    jg .L1  
...  
...
```

# Summary

## Iterative



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

## Memory

```
.section .text
```

```
...
```

```
addq  
jmp .L2
```

```
.L1:
```

```
...
```

```
<subtask>
```

```
...
```

```
.L2:
```

```
cmpq A, B  
jg .L1
```

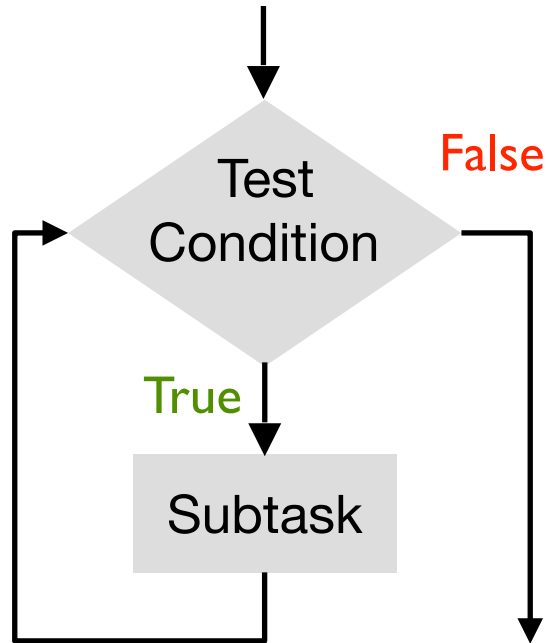
```
...
```

```
...
```



# Summary

## Iterative



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

## Memory

```
.section .text
```

```
...
```

```
addq  
jmp .L2
```

```
.L1:
```

```
...
```

```
<subtask>
```

```
...
```

```
.L2:
```

```
cmpq A, B  
jg .L1
```

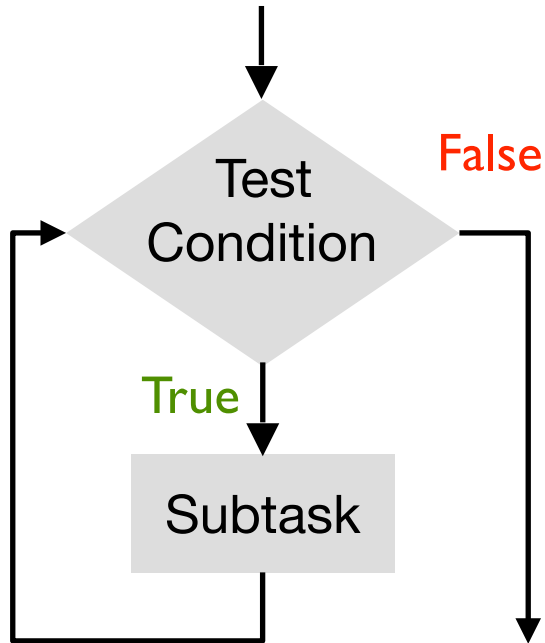
```
...
```

```
...
```



# Summary

## Iterative



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

## Memory

```
.section .text
```

```
...
```

```
addq  
jmp .L2
```

```
.L1:
```

```
...
```

```
<subtask>
```

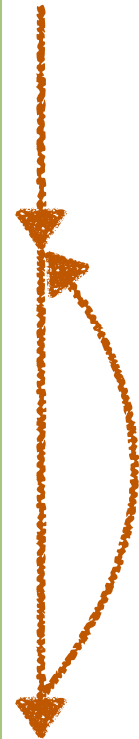
```
...
```

```
.L2:
```

```
cmpq A, B  
jg .L1
```

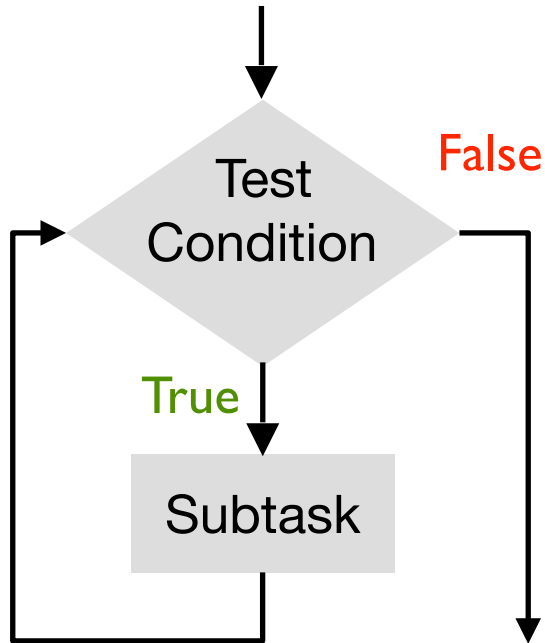
```
...
```

```
...
```



# Summary

## Iterative



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

## Memory

```
.section .text
```

```
...
```

```
addq  
jmp .L2  
.L1:
```

```
...
```

```
<subtask>
```

```
...
```

```
.L2:  
    cmpq A, B  
    jg .L1
```

```
...
```

```
...
```

