CSC 252: Computer Organization Spring 2020: Lecture 7

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Announcement

- Programming assignment 2 is out
  - Details: https://www.cs.rochester.edu/courses/252/spring2020/labs/assignment2.html
  - Due on **Feb. 14**, 11:59 PM
  - You (may still) have 3 slip days

2	3	4	5	6 **Today**	7	8
9	10	11	12	13	14 **Due**	15

Announcement

- Programming assignment 2 is out
    - Details: https://www.cs.rochester.edu/courses/252/spring2020/labs/assignment2.html
    - Due on **Feb. 14**, 11:59 PM
    - You (may still) have 3 slip days
- Read the instructions before getting started!!!
    - You get 1/4 point off for every wrong answer
    - Maxed out at 10
- Request one bomb per group using one person's email and ID. Email Shuang and Sudhanshu who you are working with.

Announcement

- Grades for lab1 are posted.
- If you think there are some problems
  - Take a deep breath
  - Tell yourself that the teaching staff like you, not the opposite
  - Email/go to Shuang or Sudhanshu's office hours and explain to them why you should get more points, and they will fix it for you

Announcement

- Office hour temporarily moved to 3-4pm this Friday
- Programming assignment 2 is in x86 assembly language. Seek help from TAs.
- TAs are best positioned to answer your questions about programming assignments!!!
- Programming assignments do NOT repeat the lecture materials. They ask you to synthesize what you have learned from the lectures and work out something new.

Data Movement

```
movq   (%rdi), %rdx
```

- Semantics:
  - Move (really, **copy**) data store in memory location whose address is the value stored in **`%rdi`** to register **`%rdx`**

Data Movement

```
movq   (%rdi), %rdx
```

- Semantics:
  - Move (really, **copy**) data store in memory location whose address is the value stored in **`%rdi`** to register **`%rdx`**

```
movq    %rdx, (%rdi)
```

Data Movement

```
movq   (%rdi), %rdx
```

- Semantics:
  - Move (really, **copy**) data store in memory location whose address is the value stored in **`%rdi`** to register **`%rdx`**

```
movq    %rdx, (%rdi)

movq    8(%rdi), %rdx
```

Data Movement

```
movq   (%rdi), %rdx
```

- Semantics:
  - Move (really, **copy**) data store in memory location whose address is the value stored in **`%rdi`** to register **`%rdx`**

```
movq    %rdx, (%rdi)
movq    8(%rdi), %rdx
addq    8(%rdi), %rdx
```

Data Movement

```
movq    (%rdi), %rdx
```

- Semantics:
  - Move (really, **copy**) data store in memory location whose address is the value stored in `%rdi` to register `%rdx`

```
movq    %rdx, (%rdi)
movq    8(%rdi), %rdx
addq    8(%rdi), %rdx
```

Accessing memory and doing computation in one instruction. Allowed in x86, but not all ISAs allow that (e.g., ARM).

Data Movement

```
movq   (%rdi), %rdx
```

- Semantics:
  - Move (really, **copy**) data store in memory location whose address is the value stored in `%rdi` to register `%rdx`

```
movq    %rdx, (%rdi)
movq    8(%rdi), %rdx
addq    8(%rdi), %rdx
movq    (%rdi), (%rdx)
```

Data Movement

```
movq    (%rdi), %rdx
```

- Semantics:
  - Move (really, **copy**) data store in memory location whose address is the value stored in **`%rdi`** to register **`%rdx`**

```
movq     %rdx, (%rdi)
movq     8(%rdi), %rdx
addq     8(%rdi), %rdx
movq     (%rdi), (%rdx)
```

> Illegal in x86 (and almost all other ISAs). Could make microarchitecture implementation inefficient/inelegant.

Today: Control Instructions

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

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

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 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 of **`jle`**:
  - Treat the data in **`%rdi`** and **`%rsi`** as **signed values**.
  - If **`%rdi`** is less than or equal to **`%rsi`**, 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 of **`jle`**:
  - Treat the data in **`%rdi`** and **`%rsi`** as **signed values**.
  - If **`%rdi`** is less than or equal to **`%rsi`**, 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 of `jle`:
    - Treat the data in `%rdi` and `%rsi` as **signed values**.
    - If `%rdi` is less than or equal to `%rsi`, jump to the part of the code with a label `.L4`
- Under the hood:
    - `cmpq` instruction sets the **condition codes (a.k.a., status flags)**

Conditional Jump Instruction

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

Jump to label if less than or equal to

- Semantics of **`jle`**:
  - Treat the data in **`%rdi`** and **`%rsi`** as **signed values**.
  - If **`%rdi`** is less than or equal to **`%rsi`**, jump to the part of the code with a label **`.L4`**

- Under the hood:
  - **`cmpq`** instruction sets the **condition codes (a.k.a., status flags)**
  - **`jle`** reads and checks the status flags

Conditional Jump Instruction

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

Jump to label if less than or equal to

- Semantics of `jle`:
  - Treat the data in `%rdi` and `%rsi` as **signed values**.
  - If `%rdi` is less than or equal to `%rsi`, jump to the part of the code with a label `.L4`
- Under the hood:
  - `cmpq` instruction sets the **condition codes (a.k.a., status flags)**
  - `jle` reads and checks the status flags
  - 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
```

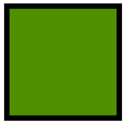
- Essentially, how do we know **`%rdi`** <= **`%rsi`**?
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`

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)



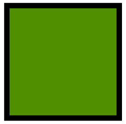
ZF

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?

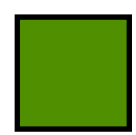
`cmpq %rsi, %rdi`

- 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

```
    001          1
-)  010     -)   2
-------     ------
    111         -1
```

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	`001`
	`-) 010`
	`111`
		`1`
		`-) 2`
		`-1`

Overflow	`101`
	`-) 011`
	`010`
		`-3`
		`-) 3`
		`2`

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

No Overflow

```
    001          1
-)  010     -)   2
-------     ------
    111         -1
```

Overflow

```
    101         -3
-)  011     -)   3
-------     ------
    010          2
```

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

No Overflow

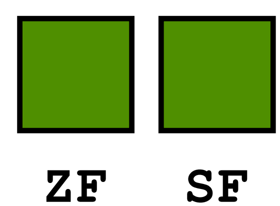
```
    001        1
-)  010   -)   2
-------   ------
    111       -1
```

Overflow

```
    101       -3
-)  011   -)   3
-------   ------
    010        2
```

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

```
    001          1
-)  010     -)   2
   ----        ---
    111         -1
```

Overflow

```
    101         -3
-)  011     -)   3
   ----        ---
    010          2
```

ZF Zero Flag (result is zero)

SF Sign Flag (result is negative)

OF Overflow Flag (results overflow)

ZF SF OF

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 (results overflow)

ZF	SF	OF
0	0	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`
- `%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
------------      --------
    10000001          -127
```

ZF Zero Flag (result is zero)

SF Sign Flag (result is negative)

OF Overflow Flag (results overflow)

ZF	SF	OF
0	0	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`
- `%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
    --------    --------
    10000001        -127
```

ZF Zero Flag (result is zero)

SF Sign Flag (result is negative)

OF Overflow Flag (results overflow)

ZF	SF	OF
0	1	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`
- `%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 (results overflow)

0	1	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

ZF	SF	OF
0	0	0

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

```
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
```

`cmpq` sets ZF, SF, OF

`jle` checks ZF | (SF ^ OF)

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

How Does the Hardware Check Overflow?

- ZF and SF are easily set by just examining the bits

How Does the Hardware Check Overflow?

- ZF and SF are easily set by just examining the bits
- How about OF? How do we know A-B leads to overflow (A and B are treated as signed)

How Does the Hardware Check Overflow?

- ZF and SF are easily set by just examining the bits
- How about OF? How do we know A-B leads to overflow (A and B are treated as signed)
  - If A < 0 & B > 0, but the result > 0, or

How Does the Hardware Check Overflow?

- ZF and SF are easily set by just examining the bits
- How about OF? How do we know A-B leads to overflow (A and B are treated as signed)
  - If $A < 0$ & $B > 0$, but the result > 0, or
  - If $A > 0$ & $B < 0$, but the result < 0

How Does the Hardware Check Overflow?

- ZF and SF are easily set by just examining the bits
- How about OF? How do we know A-B leads to overflow (A and B are treated as signed)
  - If A < 0 & B > 0, but the result > 0, or
  - If A > 0 & B < 0, but the result < 0

No Overflow

```
    001            1
-)  010      -)    2
-------      -------
    111           -1
```

How Does the Hardware Check Overflow?

- ZF and SF are easily set by just examining the bits
- How about OF? How do we know A-B leads to overflow (A and B are treated as signed)
  - If A < 0 & B > 0, but the result > 0, or
  - If A > 0 & B < 0, but the result < 0

No Overflow

```
     001          1
-)   010    -)    2
--------    -------
     111         -1
```

Overflow

```
     101         -3
-)   011    -)    3
--------    -------
     010          2
```

How Does the Hardware Check Overflow?

- ZF and SF are easily set by just examining the bits
- How about OF? How do we know A-B leads to overflow (A and B are treated as signed)
  - If A < 0 & B > 0, but the result > 0, or
  - If A > 0 & B < 0, but the result < 0

No Overflow

```
     001           1
-)   010     -)    2
---------   ---------
     111          -1
```

Overflow

```
     101          -3          011           3
-)   011     -)    3     -)   100     -)   -4
---------   ---------   ---------   ---------
     010           2          111          -1
```

How Does the Hardware Check Overflow?

- ZF and SF are easily set by just examining the bits
- How about OF? How do we know A-B leads to overflow (A and B are treated as signed)
  - If A < 0 & B > 0, but the result > 0, or
  - If A > 0 & B < 0, but the result < 0
  - So again, just have to check the bits

No Overflow

```
     001          1
-)   010     -)   2
---------    ------
     111         -1
```

Overflow

```
     101         -3          011          3
-)   011     -)   3     -)   100     -)  -4
---------    ------     ---------    ------
     010          2          111         -1
```

Conditional Branch Example

```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;
}
```

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

```
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
```

ZF	SF	OF
0	0	0

Conditional Branch Example

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

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

```
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
```

ZF	SF	OF
0	0	0

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 of **`jbe`**:
  - Treat the data in **`%rdi`** and **`%rsi`** as **unsigned values**.
  - If **`%rdi`** is less than or equal to **`%rsi`**, 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 of **`jbe`**:
  - Treat the data in **`%rdi`** and **`%rsi`** as **unsigned values**.
  - If **`%rdi`** is less than or equal to **`%rsi`**, 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 of **`jbe`**:
  - Treat the data in **`%rdi`** and **`%rsi`** as **unsigned values**.
  - If **`%rdi`** is less than or equal to **`%rsi`**, 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 of **`jbe`**:
  - Treat the data in **`%rdi`** and **`%rsi`** as **unsigned values**.
  - If **`%rdi`** is less than or equal to **`%rsi`**, 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 of **`jbe`**:
  - Treat the data in **`%rdi`** and **`%rsi`** as **unsigned values**.
  - If **`%rdi`** is less than or equal to **`%rsi`**, 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
```

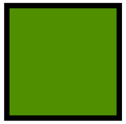
- How do we know **`%rdi`** <= **`%rsi`**? This time for unsigned values
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`

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)



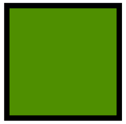
ZF

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
-------
   C010
```

ZF Zero Flag (result is zero)

ZF

How Should `cmpq` Set Condition Codes?

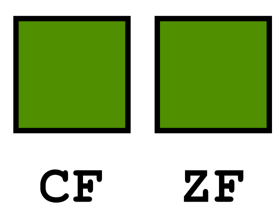
`cmpq %rsi, %rdi`

- 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
-------
   C010
```

ZF Zero Flag (result is zero)

CF Carry Flag (for unsigned)



How Should `cmpq` Set Condition Codes?

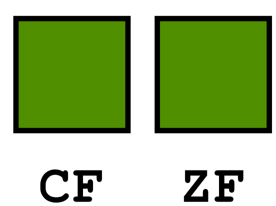
```
cmpq      %rsi, %rdi
```

- 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
- Why don't we look at the SF and OF as in the signed case?

```
    001   ←   1
-)  111   ←   7
-------
   C010
```

ZF Zero Flag (result is zero)

CF Carry Flag (for unsigned)



How Should `cmpq` Set Condition Codes?

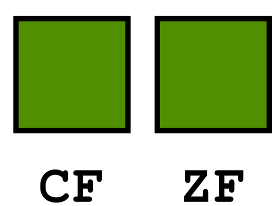
`cmpq     %rsi, %rdi`

- 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
- Why don't we look at the SF and OF as in the signed case?
  - Checking unsigned overflow is much harder

```
    001  ←  1
-)  111  ←  7
-------
   C010
```

ZF Zero Flag (result is zero)

CF Carry Flag (for unsigned)



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
CF	**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

11111111 10000000

cmpq *0xFF, 0x80*

```
    10000000    128
-)  11111111    255
------------
   c10000001
```

ZF Zero Flag (result is zero)

CF Carry Flag (for unsigned)

CF	ZF
0	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`
- `%rdi < %rsi` if a carry is generated during subtraction

11111111 10000000

`cmpq` *0xFF, 0x80*

```
    10000000  ←  128
-)  11111111  ←  255
----------------
   c10000001
```

ZF Zero Flag (result is zero)

CF Carry Flag (for unsigned)

CF	ZF
1	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`
- `%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)

1	0
CF	**ZF**

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)

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

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)

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

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)

CF	ZF	SF	OF
1	0	1	0

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)

CF	ZF	SF	OF
1	0	1	0

Putting It All Together

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

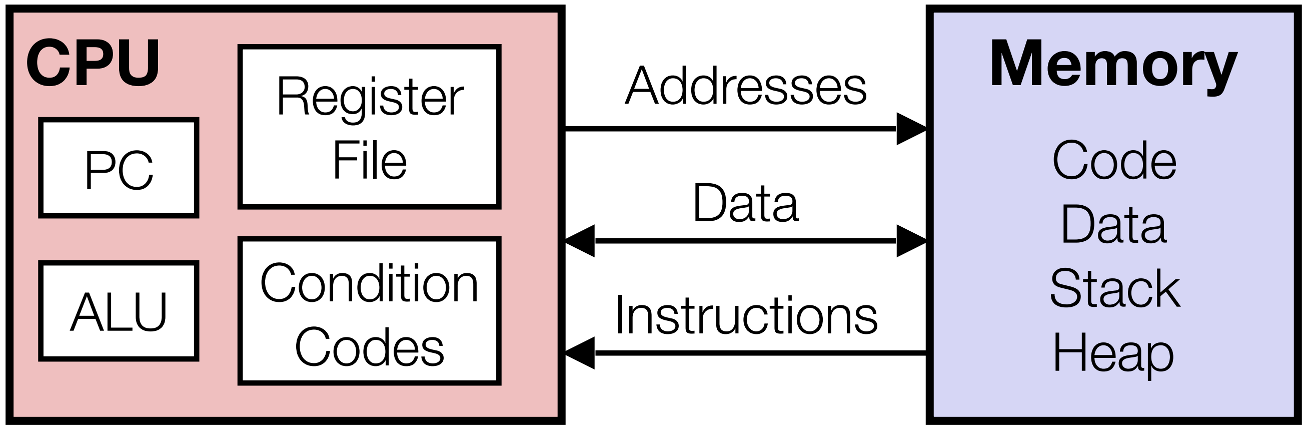
SF Sign Flag

OF Overflow Flag (for signed)

CF	ZF	SF	OF
1	0	1	0

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	ZF	SF	OF

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

jle	(SF^OF)|ZF	Less or Equal (Signed)

jbe	CF|ZF	Below or Equal (unsigned)

Jump Instructions

Instruction	Jump Condition	Description
jmp	`1`	Unconditional
je	`ZF`	Equal / Zero
jne	`~ZF`	Not Equal / Not Zero
js	`SF`	Negative
jns	`~SF`	Nonnegative
jg	`~(SF^OF)&~ZF`	Greater (Signed)
jge	`~(SF^OF)`	Greater or Equal (Signed)
jl	`(SF^OF)`	Less (Signed)
jle	`(SF^OF)|ZF`	Less or Equal (Signed)
ja	`~CF&~ZF`	Above (unsigned)
jae	`~CF`	Above or Equal (unsigned)
jb	`CF`	Below (unsigned)
jbe	`CF|ZF`	Below or Equal (unsigned)

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 (i.e., 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 (i.e., 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 (i.e., 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 (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

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 (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`

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 (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)`

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 (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)`

`addq` *0xFF, 0x80*

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

CF	ZF	SF	OF
0	0	0	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 (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)`

`addq` *0xFF*, *0x80*

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

CF	ZF	SF	OF
1	0	0	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 (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)`

addq *0xFF*, *0x80*

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

CF	ZF	SF	OF
1	0	0	1

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 (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)`

`addq` *0xFF*, *0x80*

`jle .L4`

CF	ZF	SF	OF
1	0	0	1

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 (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)`

```
if((x+y)<0) {
    …
}
```

addq *0xFF*, *0x80*

jle .L4

1	0	0	1
CF	**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
}
```

```
//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 < WSIZE`

“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 < WSIZE`

update

`i++`

“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 < 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
```

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

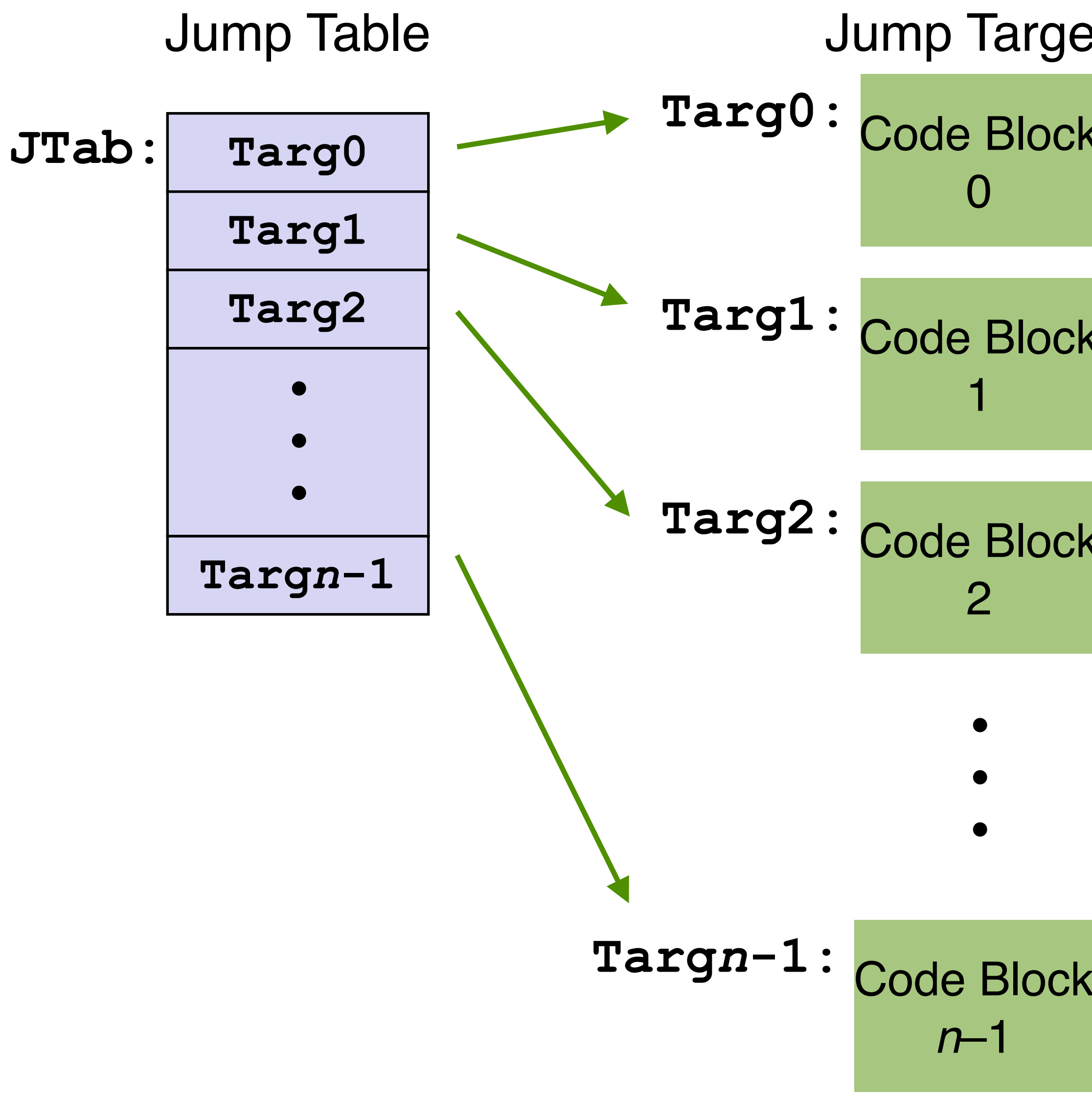
•
•
•

Targ*n*-1: Code Block *n*–1

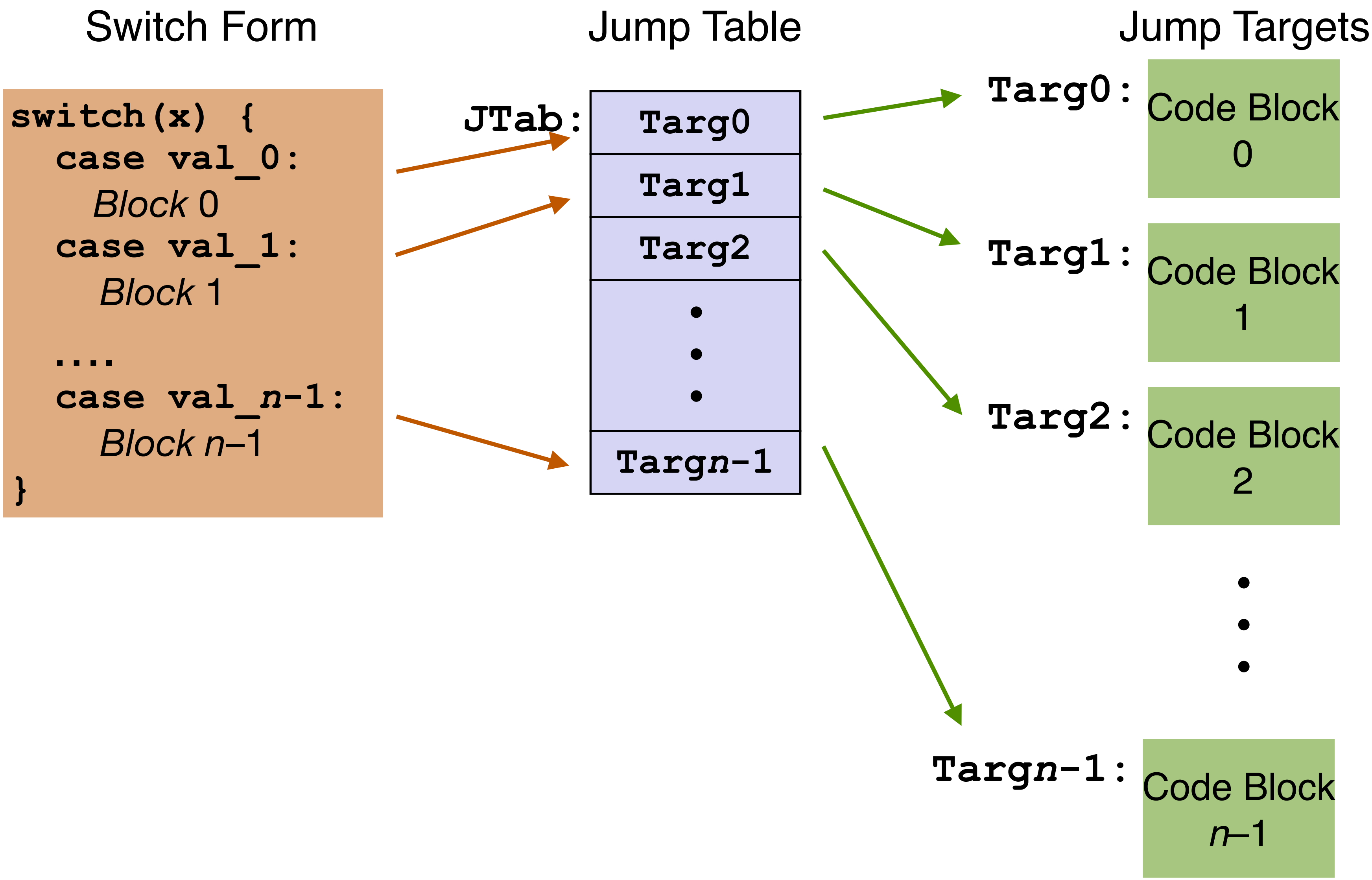
Implementing Switch Using Jump Table

Switch Form

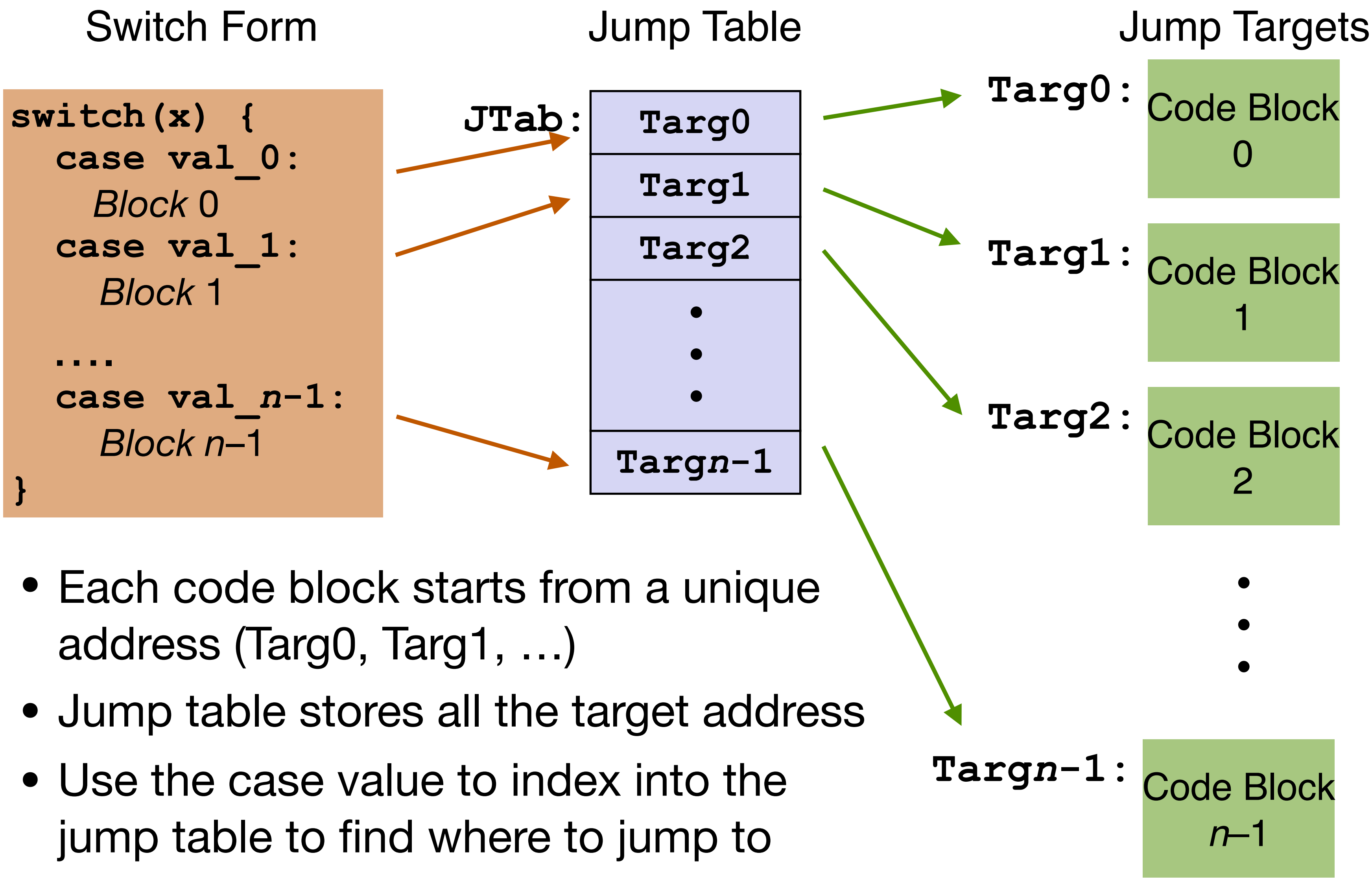
```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1

  ....
  case val_n-1:
    Block n–1
}
```



Implementing Switch Using Jump Table



Implementing Switch Using Jump Table



- Each code block starts from a unique address (Targ0, Targ1, …)
- Jump table stores all the target address
- Use the case value to index into the jump table to find where to jump to

Jump Table in Assembly

```
.section  .rodata
  .align 8
.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;
case 2:         // .L2
    w = y/z;
    /* Fall Through */
case 3:         // .L3
    w += z;
    break;
case 5:
case 6:         // .L5
    w -= z;
    break;
default:        // .LD
    w = 2;
}
```

Jump Table in Assembly

```
.section  .rodata
  .align 8
.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;
case 2:        // .L2
    w = y/z;
    /* Fall Through */
case 3:        // .L3
    w += z;
    break;
case 5:
case 6:        // .L5
    w -= z;
    break;
default:       // .LD
    w = 2;
}
```

Jump Table in Assembly

```
.section   .rodata
  .align 8
.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;
case 2:        // .L2
     w = y/z;
     /* Fall Through */
case 3:        // .L3
     w += z;
     break;
case 5:
case 6:        // .L5
     w -= z;
     break;
default:       // .LD
     w = 2;
}
```

Jump Table in Assembly

```
.section   .rodata
  .align 8
.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;
case 2:        // .L2
    w = y/z;
    /* Fall Through */
case 3:        // .L3
    w += z;
    break;
case 5:
case 6:        // .L5
    w -= z;
    break;
default:       // .LD
    w = 2;
}
```

Jump Table in Assembly

```
.section   .rodata
  .align 8
.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;
case 2:         // .L2
    w = y/z;
    /* Fall Through */
case 3:         // .L3
    w += z;
    break;
case 5:
case 6:         // .L5
    w -= z;
    break;
default:        // .LD
    w = 2;
}
```

Jump Table in Assembly

```
.section  .rodata
  .align 8
.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;
case 2:         // .L2
    w = y/z;
    /* Fall Through */
case 3:         // .L3
    w += z;
    break;
case 5:
case 6:         // .L5
    w -= z;
    break;
default:        // .LD
    w = 2;
}
```

Jump Table in Assembly

```
.section  .rodata
  .align 8
.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;
case 2:         // .L2
    w = y/z;
    /* Fall Through */
case 3:         // .L3
    w += z;
    break;
case 5:
case 6:         // .L5
    w -= z;
    break;
default:        // .LD
    w = 2;
}
```

Jump Table in Assembly

```
.section   .rodata
  .align 8
.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;
case 2:         // .L2
    w = y/z;
    /* Fall Through */
case 3:         // .L3
    w += z;
    break;
case 5:
case 6:         // .L5
    w -= z;
    break;
default:        // .LD
    w = 2;
}
```

Assembly Directives (Pseudo-Ops)

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

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

Assembly Directives (Pseudo-Ops)

```
.section  .rodata
  .align 8
.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)
- .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 .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.

- .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 .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.

- .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

Jump Table and Jump Targets

Jump Table

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

Code Blocks (x == 1)

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

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

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

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

```
.section  .rodata
  .align 8
.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;
     …
}
```

```
.L1:
   movq    %rsi, %rax  # y
   imulq   %rdx, %rax  # y*z
   jmp     .done
```

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

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

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

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

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

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

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

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

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

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

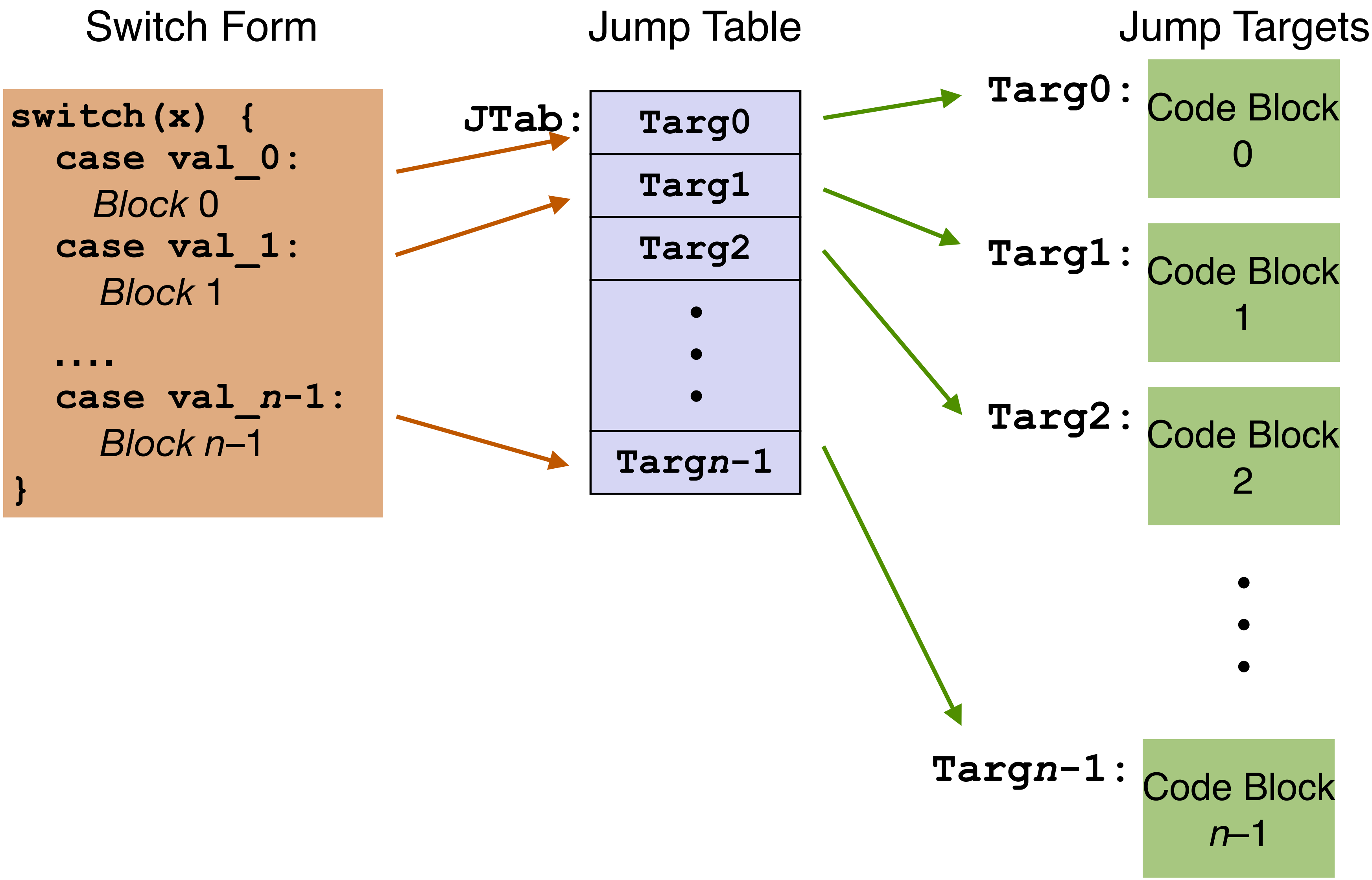
```
.section  .rodata
  .align 8
.L4:
  .quad .LD # x = 0
  .quad .L1 # x = 1
  .quad .L2 # x = 2
  .quad .L3 # x = 3
  .quad .LD # x = 4
  .quad .L5 # x = 5
  .quad .L5 # x = 6
```

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

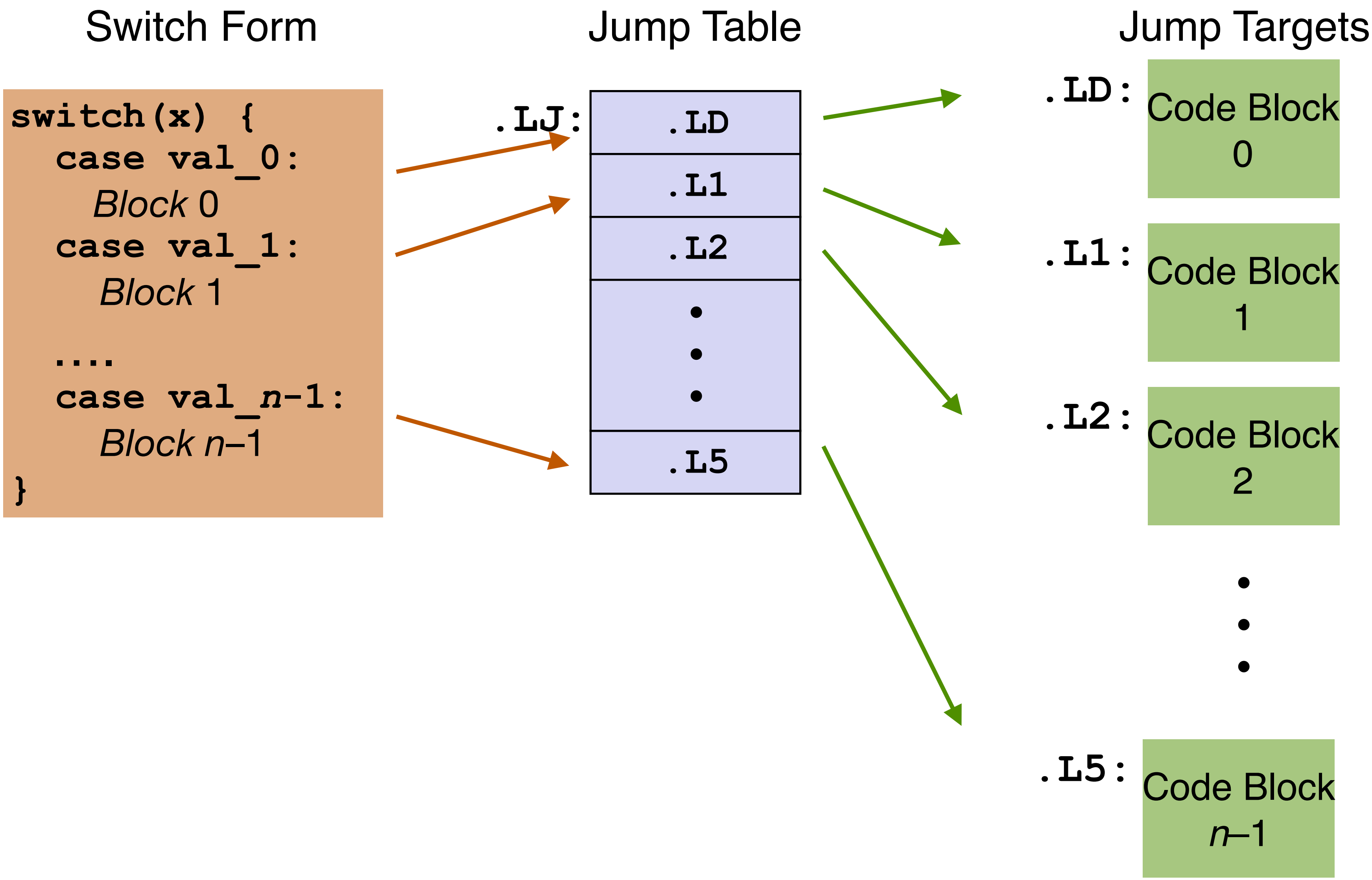
Register	Use(s)
`%rdi`	Argument `x`
`%rsi`	Argument `y`
`%rdx`	Argument `z`
`%rax`	Return value

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

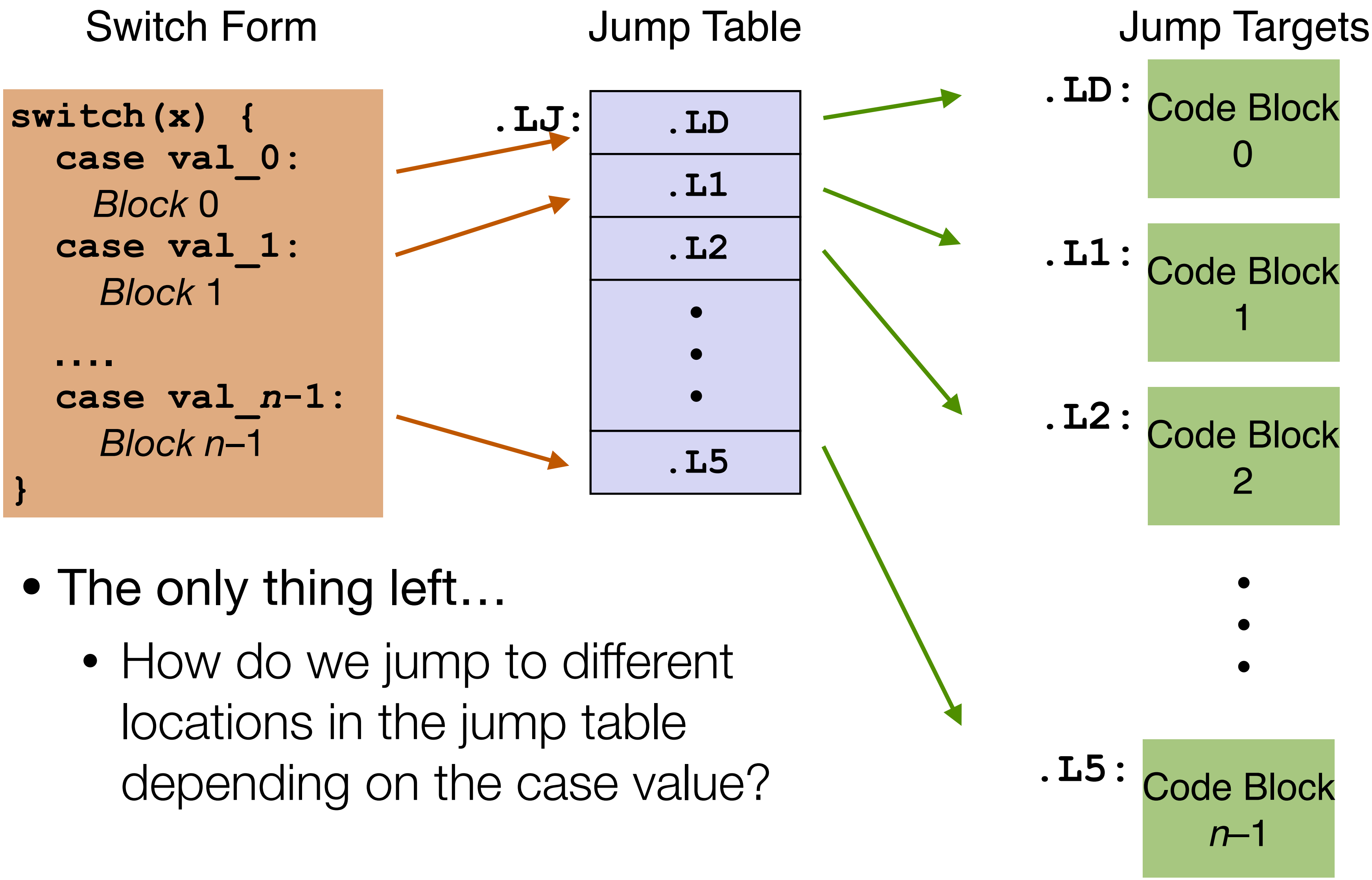
Implementing Switch Using Jump Table



Implementing Switch Using Jump Table



Implementing Switch Using Jump Table



- 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
.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`

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

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

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

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

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

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