Assembly Programming:
  • More control instructions

Instructor: Alan Beadle

Department of Computer Science
University of Rochester
Data Movement

\texttt{movq (%rdi), %rdx}

• Semantics:
  • Move (really, \textit{copy}) data store in memory location whose address is the value stored in \texttt{%rdi} to register \texttt{%rdx}
Data Movement

\[
\text{movq (~rdi), ~rdx}
\]

• Semantics:
  • Move (really, \textbf{copy}) data store in memory location whose address is the value stored in \texttt{~rdi} to register \texttt{~rdx}

\[
\text{movq ~rdx, (~rdi)}
\]
Data Movement

\[ \text{movq} \quad (%rdi), \quad %rdx \]

• Semantics:

• Move (really, \textbf{copy}) data store in memory location whose address is the value stored in \%rdi to register \%rdx

\[ \text{movq} \quad %rdx, \quad (%rdi) \]
\[ \text{movq} \quad 8(%rdi), \quad %rdx \]
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., MIPS).
Data Movement

\[ \text{movq} \quad (%\text{rdi}), \quad %\text{rdx} \]

• Semantics:
  • Move (really, \textbf{copy}) data store in memory location whose address is the value stored in \%\text{rdi} to register \%\text{rdx}
  
  \[
  \begin{align*}
  \text{movq} & \quad %\text{rdx}, \quad (%\text{rdi}) \\
  \text{movq} & \quad 8(\%\text{rdi}), \quad %\text{rdx} \\
  \text{addq} & \quad 8(\%\text{rdi}), \quad %\text{rdx} \\
  \text{movq} & \quad (%\text{rdi}), \quad (%\text{rdx})
  \end{align*}
  \]
Data Movement

\texttt{movq} (\%rdi), \%rdx

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

\texttt{movq} \%rdx, (\%rdi)
\texttt{movq} 8(\%rdi), \%rdx
\texttt{addq} 8(\%rdi), \%rdx
\texttt{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

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

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

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>x</td>
</tr>
<tr>
<td>%rsi</td>
<td>y</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

ZF 0
SF 0
OF 0
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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>x</td>
</tr>
<tr>
<td>%rsi</td>
<td>y</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

cmpq sets ZF, SF, OF
jle checks 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;
}

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>x</td>
</tr>
<tr>
<td>%rsi</td>
<td>y</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
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;
}
```

```
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
```
Conditional Jump Instruction

```plaintext
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
```

- **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:**
  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`

- **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`  

• 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 \texttt{cmpq} Set Condition Codes?

\texttt{cmpq \%rsi, \%rdi}

- How do we know \texttt{\%rdi} $\leq$ \texttt{\%rsi}? This time for unsigned values
- Calculate \texttt{\%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?

\[
\text{cmpq} \quad \%\text{rsi}, \%\text{rdi}
\]

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

\[\text{ZF} \quad \text{Zero Flag (result is zero)}\]
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)
How Should \texttt{cmpq} Set Condition Codes?

\texttt{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

\textbf{ZF}  Zero Flag (result is zero)

\[
\begin{array}{c}
001 \\
-\) 111 \\
\hline
\text{010}
\end{array}
\]
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
- ) 111
    111
  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`

<table>
<thead>
<tr>
<th>ZF</th>
<th>Zero Flag (result is zero)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CF</td>
<td>Carry Flag (for unsigned)</td>
</tr>
</tbody>
</table>

```
0 0
```

CF | ZF
How Should cmpq Set Condition Codes?

\[
\text{cmpq} \quad \%rsi, \%rdi
\]

- How do we know \(\%rdi \leq \%rsi\)? 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

\[
\begin{array}{c|c}
\text{cmpq} & 0xFF, 0x80 \\
\hline
0x10000000 & 128 \\
- & 255 \\
\hline
0x10000001 & \\
\end{array}
\]

ZF  Zero Flag (result is zero)
CF  Carry Flag (for unsigned)

\[
\begin{array}{c|c}
\text{CF} & 0 \\
\text{ZF} & 0 \\
\end{array}
\]
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
```

```
100000000        128
- ) 11111111     255
```

100000001

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

- `%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

<table>
<thead>
<tr>
<th>CF</th>
<th>ZF</th>
<th>SF</th>
<th>OF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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

\[ \begin{array}{c}
11111111 \ 10000000 \\
\text{cmpq } 0xFF, \ 0x80 \\
\text{\quad 10000000} \\
\text{\quad -) 11111111} \\
\text{\quad \quad 10000001}
\end{array} \]

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

\[ \begin{array}{cccc}
0 & 0 & 0 & 0 \\
\text{CF} & \text{ZF} & \text{SF} & \text{OF}
\end{array} \]
Putting It All Together

- `cmpq` sets all 4 condition codes simultaneously

\[
\begin{align*}
11111111 & \quad 10000000 \\
\text{cmpq} \ 0xFF, \ 0x80 & \\
\quad & \\
10000000 & \\
\text{-)} \quad 11111111 & \\
\underline{c10000001} & \\
\end{align*}
\]

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
- ZF, SF, and OF are used when comparing signed value (e.g., `jle`)

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

- `cmpq 0xFF, 0x80`

```
11111111 10000000
10000000
-
11111111
10000000
```

- 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` 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`)

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

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

10000000
- ) 11111111
   
   c10000001
```

ZF  Zero Flag
CF  Carry Flag
SF  Sign Flag
OF  Overflow Flag (for signed)
Condition Codes Hold Test Results

Assembly Programmer’s Perspective of a Computer

- **CPU**
  - PC
  - ALU
  - Register File
  - Condition Codes

- **Memory**
  - Addresses
  - Data
  - Instructions
  - Code
  - Data
  - Stack
  - Heap

- **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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jle</td>
<td>((SF^OF)</td>
<td>ZF)</td>
</tr>
<tr>
<td>jbe</td>
<td>CF</td>
<td>ZF</td>
</tr>
</tbody>
</table>
## Jump Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Jump Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>je</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>js</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg</td>
<td>~(SF^OF) &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle</td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>ja</td>
<td>~CF &amp; ~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jae</td>
<td>~CF</td>
<td>Above or Equal (unsigned)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
<tr>
<td>jbe</td>
<td>CF</td>
<td>Below or Equal (unsigned)</td>
</tr>
</tbody>
</table>
Implicit Set Condition Codes

```
addq %rax, %rbx
```
Implicit Set Condition Codes

\[ \text{addq } \%rax, \%rbx \]

- 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

\[ \text{addq} \ %rax, \ %rbx \]

- Arithmetic instructions implicitly set condition codes (think of it as a 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

\texttt{addq \%rax, \%rbx}

- Arithmetic instructions implicitly set condition codes (think of it as side effect)
  - \textbf{CF} set if \%rax + \%rbx generates a carry (i.e., unsigned overflow)
  - \textbf{ZF} set if \%rax + \%rbx == 0
  - \textbf{SF} set if \%rax + \%rbx < 0
  - \textbf{OF} set if \%rax + \%rbx overflows when \%rax and \%rbx are treated as signed numbers
Implicit Set Condition Codes

\[ \text{addq} \ \%rax, \ \%rbx \]

- 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
    - \(\%rax > 0, \%rbx > 0, \text{ and } (\%rax + \%rbx) < 0\), or
Implicit Set Condition Codes

\[ \text{addq } \%rax, \%rbx \]

- 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

\( \text{addq} \ %rax, \ %rbx \)

- 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
    - \( %rax > 0, \ %rbx > 0, \text{ and } (%rax + %rbx) < 0 \), or
    - \( %rax < 0, \ %rbx < 0, \text{ and } (%rax + %rbx) >= 0 \)

\( \text{addq} \ 0xFF, \ 0x80 \)

\[
\begin{array}{c}
10000000 \\
+ \ 11111111 \\
\hline
01111111
\end{array}
\]

- CF set
- ZF set
- SF set
- OF set

0 0 0 0
Implicit Set Condition Codes

\[ \text{addq } \%rax, \%rbx \]

- 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 \( (\%rax + \%rbx) >= 0 \)

\[ \text{addq } 0xFF, 0x80 \]

\[
\begin{array}{c}
10000000 \\
+ \quad 11111111 \\
\hline
01111111
\end{array}
\]

\[ 0 \quad 1 \quad 0 \quad 0 \quad 0 \]

\( \text{CF} \quad \text{ZF} \quad \text{SF} \quad \text{OF} \)
Implicit Set Condition Codes

\[ \text{addq} \ %rax, \ %rbx \]

- 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)

\[ \text{addq} \ 0xFF, \ 0x80 \]

```
10000000
+    11111111
    01111111
```

<table>
<thead>
<tr>
<th>CF</th>
<th>ZF</th>
<th>SF</th>
<th>OF</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Implicit Set Condition Codes

\( \text{addq} \ %\text{rax}, \ %\text{rbx} \)

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

\( \text{addq} \ 0xFF, \ 0x80 \)

\( \text{jle} \ .L4 \)
Implicit Set Condition Codes

\[ \text{addq } \%rax, \%rbx \]

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

\[
\text{if}((x+y)<0) \{ \text{...} \} \
\]

\[ \text{addq } 0xFF, 0x80 \]

\[ \text{jle } .L4 \]

\begin{tabular}{|c|c|c|c|c|}
\hline
& CF & ZF & SF & OF \\
\hline
\hline
\end{tabular}
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

```c
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

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

**goto** Version

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

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rax</td>
<td>result</td>
</tr>
</tbody>
</table>
“Do-While” Loop Assembly

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rax</td>
<td>result</td>
</tr>
</tbody>
</table>
“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;
  }

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rax</td>
<td>result</td>
</tr>
</tbody>
</table>
“Do-While” Loop Assembly

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rax</td>
<td>result</td>
</tr>
</tbody>
</table>

```assembly
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

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rax</td>
<td>result</td>
</tr>
</tbody>
</table>

```assembly
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

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rax</td>
<td>result</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rax</td>
<td>result</td>
</tr>
</tbody>
</table>

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

movl     $ 0,  % r a x   #   r e s u l t  =  0
.L2:   # loop:
    movq    %rdi, %rdx
    andl    $1, %rdx  #  t = x & 0x1
    addq    %rdx, %rax #  result += t
    shrq    $1, %rdi  #  x >>= 1
    jne .L2
    #   i f  ( x )  g o t o  l o o p
ret

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rax</td>
<td>result</td>
</tr>
</tbody>
</table>
**“Do-While” Loop Assembly**

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rax</td>
<td>result</td>
</tr>
</tbody>
</table>
“Do-While” Loop Assembly

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rax</td>
<td>result</td>
</tr>
</tbody>
</table>
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

```while (A < B) {
    body;
}
```

```<before>;
goto Version
<after>;
```
General “While” Translation

**while** version

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

**goto** Version

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

Assembly Version

```c
<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

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

```c
long pcound_while (unsigned long x) {
    long result = 0;
    while (x) {
        result += x & 0x1;
        x >>= 1;
    }
    return result;
}
```
“While” Loop Example

while version

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

goto Version

```c
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

```java
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

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

```c
//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

```c
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;
}

<table>
<thead>
<tr>
<th>init</th>
<th>i = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>test</td>
<td>i &lt; 32</td>
</tr>
</tbody>
</table>
"For" Loop Example

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

//assume unsigned int is 4 bytes
long pcount_for (unsigned int x) {

  size_t i;
  long result = 0;
  for (i = 0; i < 32; i++)
  {
    result += (x >> i) & 0x1;
  }
  return result;
}
```

init  
i = 0

test  
i < 32

update  
i++
“For” Loop Example

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

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

- **init**
  - `i = 0`
- **test**
  - `i < 32`
- **update**
  - `i++`
- **body**
  ```c
  { 
      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

```c
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;  // Fall-through case
        case 3:  
            w += z;
            break;
        case 5:  
        case 6:  
            w -= z;
            break;
        default:  
            w = 2;
    }
    return w;
}
Switch Statement Example

```c
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;  // Fall-through case
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
    }
    return w;
}
```
Switch Statement Example

```c
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

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

Converting to a cascade of if-else statements is simple, but cumbersome with too many cases.
Implementing Switch Using Jump Table

Switch Form

```java
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

```c
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

```java
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

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

Jump Table

<table>
<thead>
<tr>
<th>JTab:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Targ0</td>
</tr>
<tr>
<td>Targ1</td>
</tr>
<tr>
<td>Targ2</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Targn-1</td>
</tr>
</tbody>
</table>

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

```java
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

- 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
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)

• `.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)

```
.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)

- **.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 following data will be aligned to 8 bytes

```
.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)

- **.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 following data will be aligned to 8 bytes
- **.section**: denotes different parts of the object file

Directives:

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

```assembly
.sectio n .rodata
.align 8
.L4:
  .quad .L0 # x = 0
  .quad .L1 # x = 1
  .quad .L2 # x = 2
  .quad .L3 # x = 3
  .quad .L4 # x = 4
  .quad .L5 # x = 5
  .quad .L6 # x = 6
```
Assembly Directives (Pseudo-Ops)

- **.section .rodata**: 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 following data will be aligned to 8 bytes
- **.section**: denotes different parts of the object file
- **.rodata**: read-only data section

---

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

```text
.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
```
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
```
Implementing Switch Using Jump Table

Switch Form

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

Jump Table

<table>
<thead>
<tr>
<th>JTab:</th>
<th>Targ0</th>
<th>Targ1</th>
<th>Targ2</th>
<th>Targn-1</th>
</tr>
</thead>
</table>

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

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

Jump Table

<table>
<thead>
<tr>
<th></th>
<th>.LD</th>
<th>.L1</th>
<th>.L2</th>
<th>.L5</th>
</tr>
</thead>
<tbody>
<tr>
<td>.LJ</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Jump Targets

- `.LD: Code Block 0`
- `.L1: Code Block 1`
- `.L2: Code Block 2`
- `...`
- `.L5: Code Block n-1`
- `.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)

```
.switch .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
```

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

.switch .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;
  ...
}

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
Code Blocks (x == 1)

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
Code Blocks (x == 1)

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

```
.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
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
Code Blocks (x == 2, x == 3)

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
Code Blocks (x == 2, x == 3)

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

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

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
Code Blocks (x == 2, x == 3)

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

### Register Use(s)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
Code Blocks (x == 5, x == 6, default)

```asm
.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
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
Code Blocks (x == 5, x == 6, default)

```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 5: // .L5
 case 6: // .L5
   w -= z;
   break;
 default: // .LD
   w = 2;
}
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
Code Blocks (x == 5, x == 6, default)

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
Code Blocks (x == 5, x == 6, default)

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
Code Blocks (x == 5, x == 6, default)

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
Implementing Switch Using Jump Table

Switch Form

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

Jump Table

```
.LD
.L1
.L2
.L5
```

Jump Targets

```
.LD: Code Block 0
.L1: Code Block 1
.L2: Code Block 2
.L5: Code Block n-1
.Done:
```
Implementing Switch Using Jump Table

Switch Form

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

Jump Table

- .LD
- .L1
- .L2
- .L5

Jump Targets

- .LD: Code Block 0
- .L1: Code Block 1
- .L2: Code Block 2
- .L5: Code Block n-1
- .Done:

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

```assembly
.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 $LJ + 8 \times x$

```assembly
.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

```assembly
.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 \( \text{LJ} + 8 \times x \)

```assembly
.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 $\text{LJ} + 8 \times x$

# 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 $LJ + 8 \times x$

```
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
```

```c
# 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 \( .LJ + 8 \times x \)

\[
\text{section .rodata}
\text{.align 8}
.LJ:
\text{.quad .LD \# x = 0}
\text{.quad .L1 \# x = 1}
\text{.quad .L2 \# x = 2}
\text{.quad .L3 \# x = 3}
\text{.quad .LD \# x = 4}
\text{.quad .L5 \# x = 5}
\text{.quad .L5 \# x = 6}
\]

An equivalent syntax in x86:

\[
\begin{align*}
\text{# assume x in %rdi} \\
\text{movq .LJ(,%rdi,8), %rax} \\
\text{jmp *%rax}
\end{align*}
\]

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

\[
\text{jmp *.LJ(,%rdi,8)}
\]