Computer Systems Principles

x86-64 Assembly (Part 2)
Objectives

• x86-64 Assembly Language
  – To learn about condition codes
  – To learn about conditional branches
  – To learn about loop
Condition Codes (Implicit Setting)

- **Single bit registers**
  - **CF**  Carry Flag (for unsigned)
  - **ZF**  Zero Flag
  - **SF**  Sign Flag (for signed)
  - **OF**  Overflow Flag (for signed)
Condition Codes (Implicit Setting)

■ Single bit registers
  – CF  Carry Flag (for unsigned)  SF  Sign Flag (for signed)
  – ZF  Zero Flag                  OF  Overflow Flag (for signed)
Condition Codes (Implicit Setting)

- **Single bit registers**
  - CF  Carry Flag (for unsigned)
  - ZF  Zero Flag
  - SF  Sign Flag (for signed)
  - OF  Overflow Flag (for signed)
Condition Codes (Implicit Setting)

- **Single bit registers**
  - CF  Carry Flag (for unsigned)
  - SF  Sign Flag (for signed)
  - ZF  Zero Flag
  - OF  Overflow Flag (for signed)
Condition Codes (Implicit Setting)

- **Single bit registers**
  - CF  Carry Flag (for unsigned)
  - ZF  Zero Flag
  - SF  Sign Flag (for signed)
  - OF  Overflow Flag (for signed)

- **Implicitly set (think of it as side effect) by arithmetic operations**
  - Example: `addq Src, Dest ↔ t = a+b`
  - **CF set** if (unsigned)t < (unsigned)a (unsigned overflow)
  - **ZF set** if t == 0
  - **SF set** if t < 0 (as signed)
  - **OF set** if two’s-complement (signed) overflow
    \[(a>0 \land \land b>0 \land \land t<0) \lor (a<0 \land \land b<0 \land \land t>=0)\]
Condition Codes (Implicit Setting)

- **Single bit registers**
  - **CF**  Carry Flag (for unsigned)
  - **SF**  Sign Flag (for signed)
  - **ZF**  Zero Flag
  - **OF**  Overflow Flag (for signed)

- **Implicitly set (think of it as side effect) by arithmetic operations**
  
  Example: `addq Src, Dest ↔ t = a + b`

  - **CF set** if (unsigned)\(t < (\text{unsigned})a\) (unsigned overflow)
  - **ZF set** if \(t = 0\)
  - **SF set** if \(t < 0\) (as signed)
  - **OF set** if two’s-complement (signed) overflow
    \((a > 0 \land b > 0 \land t < 0) \lor (a < 0 \land b < 0 \land t >= 0)\)

- **Not set by `leaq` instruction**
Condition Codes (Implicit Setting)

- **Single bit registers**
  - **CF** Carry Flag (for unsigned)
  - **ZF** Zero Flag
  - **SF** Sign Flag (for signed)
  - **OF** Overflow Flag (for signed)

- **Implicitly set (think of it as side effect) by arithmetic operations**
  - Example: `addq Src, Dest ↔ t = a+b`

  - **CF set** if (unsigned)\( t < (\text{unsigned})a \) (unsigned overflow)
  - **ZF set** if \( t == 0 \)
  - **SF set** if \( t < 0 \) (as signed)
  - **OF set** if two’s-complement (signed) overflow
    \[
    (a>0 \&\& b>0 \&\& t<0) \mid\mid (\text{a}<0 \&\& \text{b}<0 \&\& t>=0)
    \]

  - **Not set by leaq instruction**
  - **Mostly ignored**
Condition Codes (Explicit Setting: Compare)

- **Explicit Setting by Compare Instruction**
  - `cmpq Src2, Src1`
  - `cmpq b, a` like computing $a - b$ without setting destination

- **CF set** if carry out from most significant bit (used for unsigned comparisons)
- **ZF set** if $a == b$
- **SF set** if $(a - b) < 0$ (as signed)
- **OF set** if two’s-complement (signed) overflow
  $(a > 0 \&\& b < 0 \&\& (a - b) < 0) \lor (a < 0 \&\& b > 0 \&\& (a - b) > 0)$
Condition Codes (Explicit Setting: Test)

- Explicit Setting by Test instruction
  - `testq Src2, Src1`
    - `testq b, a` like computing `a&b` without setting destination
      - Sets condition codes based on value of `Src1` & `Src2`
        - **ZF set** when `a&b == 0`
        - **SF set** when `a&b < 0`
Condition Codes (Explicit Setting: Test)

- Explicit Setting by Test instruction
  - `testq Src2, Src1`
    - `testq b, a` like computing `a&b` without setting destination

  - Sets condition codes based on value of `Src1 & Src2`
    - **ZF set** when `a&b == 0`
    - **SF set** when `a&b < 0`

  - Useful to have one of the operands be a mask
Condition Codes (Explicit Setting: Test)

• **Explicit Setting by Test instruction**
  
  – **testq** *Src2, Src1*
    
    • **testq** *b, a* like computing *a&b* without setting destination

  – Sets condition codes based on value of *Src1 & Src2*
    
    • **ZF set** when *a&b == 0*
    • **SF set** when *a&b < 0*

  – Useful to have one of the operands be a mask
  – Typical use: the same operand is repeated

  • Example: **testq %rax, %rax**
Reading Condition Codes

• **SetX Instructions**
  – Set low-order byte of destination to 0 or 1 based on combinations of condition codes
x86-64 Integer Registers

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

– setx does not alter remaining 7 bytes
Reading Condition Codes

• **SetX Instructions**
  – Set low-order byte of destination to 0 or 1 based on combinations of condition codes

<table>
<thead>
<tr>
<th>SetX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sete</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>setne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>sets</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>setns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg</td>
<td>~(SF^OF) &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>setge</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>setl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>setle</td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>seta</td>
<td>~CF&amp;~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>setb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
### Reading Condition Codes

- **SetXInstructions**
  - Set low-order byte of destination to 0 or 1 based on combinations of condition codes

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

### Instructions

- **CMP S1, S2**
  - cmpb: compare byte
  - cmpw: compare word
  - cmpl: compare long word
  - cmpq: compare quad word

- **mov S, D**
  - movb: mov byte
  - movw: mov word
  - movl: mov long word
  - movq: mov quad word

- **add S, D**
  - addb: add byte
  - addw: add word
  - addl: add long word
  - addq: add quad word
### Carnegie Mellon

#### Reading Condition Codes

- **SetX Instructions** – Set low-order byte of destination to 0 or 1 based on combinations of condition codes

<table>
<thead>
<tr>
<th>SetX</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sete</code></td>
<td>ZF</td>
</tr>
<tr>
<td><code>setne</code></td>
<td>~ZF</td>
</tr>
<tr>
<td><code>setns</code></td>
<td>~SF</td>
</tr>
<tr>
<td><code>setng</code></td>
<td>~(SF^OF) &amp; ~ZF</td>
</tr>
<tr>
<td><code>setge</code></td>
<td>~(SF^OF)</td>
</tr>
<tr>
<td><code>setl</code></td>
<td>(SF^OF)</td>
</tr>
<tr>
<td><code>setle</code></td>
<td>(SF^OF)</td>
</tr>
<tr>
<td><code>seta</code></td>
<td>~CF &amp; ~ZF</td>
</tr>
<tr>
<td><code>setb</code></td>
<td>CF</td>
</tr>
</tbody>
</table>

- **CMP S1, S2**
  - `cmpb`: compare byte
  - `cmpw`: compare word
  - `cmpl`: compare long word
  - `cmpq`: compare quad word

- **mov S, D**
  - `movb`: mov byte
  - `movw`: mov word
  - `movl`: mov long word
  - `movq`: mov quad word

- **add S, D**
  - `addb`: add byte
  - `addw`: add word
  - `addl`: add long word
  - `addq`: add quad word
## Reading Condition Codes

- **SetX Instructions**
  - Set low-order byte of destination to 0 or 1 based on combinations of condition codes

<table>
<thead>
<tr>
<th>SetX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sete</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>setne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>sets</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>setns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg</td>
<td>~ (SF^OF) &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>setge</td>
<td>~ (SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>setl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>setle</td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>seta</td>
<td>~CF&amp;~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>setb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
## Reading Condition Codes

### SetX Instructions
- Set low-order byte of destination to 0 or 1 based on combinations of condition codes

<table>
<thead>
<tr>
<th>SetX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sete</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>setne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>sets</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>setns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg</td>
<td>~(SF^OF)&amp;~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>setge</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>setl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>setle</td>
<td>(SF^OF)</td>
<td>ZF</td>
</tr>
<tr>
<td>seta</td>
<td>~CF&amp;~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>setb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
Reading Condition Codes

- **SetX Instructions**
  - Set low-order byte of destination to 0 or 1 based on combinations of condition codes

<table>
<thead>
<tr>
<th>SetX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sete</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>setne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>sets</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>setns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg</td>
<td>~(SF^OF) &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>setge</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>setl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>setle</td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>seta</td>
<td>~CF &amp; ~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>setb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
Reading Condition Codes

- **SetX Instructions**
  - Set low-order byte of destination to 0 or 1 based on combinations of condition codes

<table>
<thead>
<tr>
<th>SetX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sete</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>setne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>sets</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>setns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg</td>
<td>~(SF^OF) &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>setge</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>setl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>setle</td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>seta</td>
<td>~CF&amp;~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>setb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
Reading Condition Codes (Cont.)

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

<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>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

- `cmpq %rsi, %rdi` # Compare x:y
- `setg %al` # Set when >
- `movzbl %al, %eax` # Zero rest of %rax
- `ret`
Reading Condition Codes (Cont.)

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

<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>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

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

```assembly
%rax  %eax  %ax  %al
```
Reading Condition Codes (Cont.)

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

cmpq %rsi, %rdi # Compare x:y
setg %al       # Set when >
movzbl %al, %eax # Zero rest of %rax
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>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
Reading Condition Codes (Cont.)

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

- Does not alter remaining bytes

<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>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

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

%rax
%eax
%ax
%al
Reading Condition Codes (Cont.)

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

<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>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

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

- Does not alter remaining bytes
- Typically use `movzbl` to finish job
Reading Condition Codes (Cont.)

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

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

- Does not alter remaining bytes
- Typically use `movzbl` to finish job
Reading Condition Codes (Cont.)

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

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

- Does not alter remaining bytes
- Typically use `movzbl` to finish job
- 32-bit instruction result also set upper 32 bits to 0

<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>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
Reading Condition Codes (Cont.)

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

<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>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

- Does not alter remaining bytes
- Typically use `movzbl` to finish job
- 32-bit instruction result also set upper 32 bits to 0
- Pattern: `cmp + set + movz`
iClicker question

For the C Code
int comp(data_t a, data_t b) {
    return a COMP b;
}
the compiler generate this instruction sequence
cmpl %esi, %edi
setl %al
Suppose a is in some portion of %rdi while b is in some portion of %rsi. What is the size of data type data_t and which is comparison COMP?
A. 32-bit, >  B. 16-bit, <  C. 32-bit, <  D. 16-bit, >
For the C Code
int comp(data_t a, data_t b) {
    return a COMP b;
}
the compiler generate this instruction sequence
cmpl %esi, %edi
setl %al
Suppose a is in some portion of %rdi while b is in some portion of %rsi. What is the size of data type data_t and which is comparison COMP? Sol: C
A. 32-bit, > B. 16-bit, < C. 32-bit, < D. 16-bit, >
CONDITIONAL BRANCHES
Jumping

• jX Instructions
  – Jump to different part of code depending on condition codes
Jumping

- **jX Instructions**
  - Jump to different part of code depending on condition codes

<table>
<thead>
<tr>
<th>jX</th>
<th>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>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
Conditional Jump

cmpq %rsi, %rdi
jle .L2
Conditional Jump

```
cmpq   %rsi, %rdi
jle    .L2
```
Conditional Jump

cmpq %rsi, %rdi
jle .L2

%rdi > %rsi: PC++
Conditional Jump

cmpq %rsi, %rdi
jle .L2

%rdi > %rsi: PC++
%rdi <= %rsi: PC = address- of(.L2)
Conditional Branch Example

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

- Generation
  
  `gcc -Og -S control.c`

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

• Generation
  
gcc -Og -S control.c

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

absdiff:
  cmpq  %rsi, %rdi  # x:y
  jle   .L2
  movq  %rdi, %rax
  subq  %rsi, %rax
  ret
  .L2:    # 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>Argument \textit{x}</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument \textit{y}</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
### Conditional Branch in a pipeline

<table>
<thead>
<tr>
<th></th>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Write back</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>cmpq %rsi, %rdi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
absdiff:
    cmpq %rsi, %rdi    # x:y
    jle .L2
    movq %rdi, %rax
    subq %rsi, %rax
    ret
.L2:                # x <= y
    movq %rsi, %rax
    subq %rdi, %rax
    ret
```
## Conditional Branch in a pipeline

<table>
<thead>
<tr>
<th></th>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Write back</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>cmpq %rsi, %rdi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t2</td>
<td>jle .L2</td>
<td>cmpq %rsi, %rdi</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### absdiff:

```
cmpq %rsi, %rdi  # x:y
jle .L2
movq %rdi, %rax
subq %rsi, %rax
ret

.L2:  # x <= y
    movq %rsi, %rax
    subq %rdi, %rax
    ret```

- **t1**: Fetch cmpq %rsi, %rdi
- **t2**: Decode jle .L2
- **t3**: Execute cmpq %rsi, %rdi
- **t4**: Memory
- **t5**: Write back
### Conditional Branch in a pipeline

<table>
<thead>
<tr>
<th></th>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Write back</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>cmpq %rsi, %rdi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t2</td>
<td>jle .L2</td>
<td></td>
<td>cmpq %rsi, %rdi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t3</td>
<td>?</td>
<td>jle .L2</td>
<td>cmpq %rsi, %rdi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**absdiff:**
- cmpq %rsi, %rdi  # x:y
- jle .L2
- movq %rdi, %rax
- subq %rsi, %rax
- ret

```
.absdiff:
    cmpq %rsi, %rdi  # x:y
    jle .L2
    movq %rdi, %rax
    subq %rsi, %rax
    ret

.L2:       # x <= y
    movq %rsi, %rax
    subq %rdi, %rax
    ret
```
Conditional Branch in a pipeline

<table>
<thead>
<tr>
<th></th>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Write back</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>cmpq %rsi, %rdi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t2</td>
<td>jle .L2</td>
<td>cmpq %rsi, %rdi</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t3</td>
<td>movq %rdi, %rax</td>
<td>jle .L2</td>
<td></td>
<td>cmpq %rsi, %rdi</td>
<td></td>
</tr>
<tr>
<td>t4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

absdiff:

```assembly
  cmpq %rsi, %rdi # x:y
  jle .L2
  movq %rdi, %rax
  subq %rsi, %rax
  ret
.L2:       # x <= y
  movq %rsi, %rax
  subq %rdi, %rax
  ret
```
### Conditional Branch in a pipeline

<table>
<thead>
<tr>
<th></th>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Write back</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>cmpq %rsi, %rdi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t2</td>
<td>jle .L2</td>
<td>cmpq %rsi, %rdi</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t3</td>
<td>movq %rdi, %rax</td>
<td>jle .L2</td>
<td>cmpq %rsi, %rdi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t4</td>
<td>subq %rsi, %rax</td>
<td>movq %rdi, %rax</td>
<td>jle .L2</td>
<td>cmpq %rsi, %rdi</td>
<td></td>
</tr>
<tr>
<td>t5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
absdiff:
    cmpq %rsi, %rdi  # x:y
    jle .L2
    movq %rdi, %rax
    subq %rsi, %rax
    ret

.L2:       # x <= y
    movq %rsi, %rax
    subq %rdi, %rax
    ret
```
Conditional Branch in a pipeline

<table>
<thead>
<tr>
<th></th>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Write back</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>cmpq %rsi, %rdi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t2</td>
<td>jle .L2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t3</td>
<td>movq %rdi, %rax</td>
<td>jle .L2</td>
<td>cmpq %rsi, %rdi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t4</td>
<td>subq %rsi, %rax</td>
<td>movq %rdi, %rax</td>
<td>jle .L2</td>
<td>cmpq %rsi, %rdi</td>
<td></td>
</tr>
<tr>
<td>t5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Possibility I:** the conditional branch is not taken.

```assembly
 absdiff:
    cmpq %rsi, %rdi  # x:y
    jle .L2
    movq %rdi, %rax
    subq %rsi, %rax
    ret

.L2:      # x <= y
    movq %rsi, %rax
    subq %rdi, %rax
    ret
```

Possibility I: the conditional branch is not taken.
Conditional Branch in a pipeline

<table>
<thead>
<tr>
<th></th>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Write back</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>cmpq %rsi, %rdi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t2</td>
<td>jle .L2</td>
<td>cmpq %rsi, %rdi</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t3</td>
<td>movq %rdi, %rax</td>
<td>jle .L2</td>
<td>cmpq %rsi, %rdi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t4</td>
<td>subq %rsi, %rax</td>
<td>movq %rdi, %rax</td>
<td>jle .L2</td>
<td>cmpq %rsi, %rdi</td>
<td></td>
</tr>
<tr>
<td>t5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

absdiff:
  cmpq %rsi, %rdi  # x:y
  jle .L2
  movq %rdi, %rax
  subq %rsi, %rax
  ret

.L2:
  # x <= y
  movq %rsi, %rax
  subq %rdi, %rax
  ret

Possibility II: the conditional branch is taken
### Conditional Branch in a pipeline

<table>
<thead>
<tr>
<th></th>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Write back</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>cmpq %rsi, %rdi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t2</td>
<td>jle .L2</td>
<td></td>
<td>cmpq %rsi, %rdi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t3</td>
<td>movq %rdi, %rax</td>
<td>jle .L2</td>
<td></td>
<td>cmpq %rsi, %rdi</td>
<td></td>
</tr>
<tr>
<td>t4</td>
<td>subq %rax, %rax</td>
<td></td>
<td>movq %rdi, %rax</td>
<td>jle .L2</td>
<td>cmpq %rsi, %rdi</td>
</tr>
<tr>
<td>t5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
absdiff:
    cmpq %rsi, %rdi      # x:y
    jle .L2
    movq %rdi, %rax
    subq %rsi, %rax
    ret

.L2:     # x <= y
    movq %rsi, %rax
    subq %rdi, %rax
    ret
```

Possibility II: the conditional branch is taken
### Conditional Branch in a pipeline

<table>
<thead>
<tr>
<th></th>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Write back</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>t1</strong></td>
<td>cmpq %rsi, %rdi</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td><strong>t2</strong></td>
<td>jle .L2</td>
<td>cmpq %rsi, %rdi</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td><strong>t3</strong></td>
<td>movq %rdi, %rax</td>
<td>jle .L2</td>
<td>cmpq %rsi, %rdi</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td><strong>t4</strong></td>
<td>*</td>
<td>movq %rdi, %rax</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td><strong>t5</strong></td>
<td>*</td>
<td>*</td>
<td>jle .L2</td>
<td>cmpq %rsi, %rdi</td>
<td>*</td>
</tr>
</tbody>
</table>

**absdiff:**
```
cmpq %rsi, %rdi  # x:y
jle .L2
movq %rdi, %rax
subq %rsi, %rax
ret
```
```
.L2:  # x <= y
movq %rsi, %rax
subq %rdi, %rax
ret
```

Possibility II: the conditional branch is taken
Conditional Branch in a pipeline

<table>
<thead>
<tr>
<th></th>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Write back</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>cmpq %rsi, %rdi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t2</td>
<td>jle .L2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t3</td>
<td>movq %rdi, %rax</td>
<td>jle .L2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t4</td>
<td>subq %rsi, %rax</td>
<td>movq %rdi, %rax</td>
<td>jle .L2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t5</td>
<td>movq %rsi, %rax</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

absdiff:
- cmpq %rsi, %rdi  # x:y
- jle .L2
- movq %rdi, %rax
- subq %rsi, %rax
- ret

.L2:
- # x <= y
  - movq %rsi, %rax
  - subq %rdi, %rax
  - ret

Possibility II: the conditional branch is taken
### Conditional Branch in a pipeline

<table>
<thead>
<tr>
<th></th>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Write back</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>cmpq %rsi, %rdi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t2</td>
<td>jle .L2</td>
<td>cmpq %rsi, %rdi</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t3</td>
<td>movq %rdi, %rax</td>
<td>jle .L2</td>
<td>cmpq %rsi, %rdi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t4</td>
<td>subq %rsi, %rax</td>
<td>movq %rdi, %rax</td>
<td>jle .L2</td>
<td>cmpq %rsi, %rdi</td>
<td></td>
</tr>
<tr>
<td>t5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**absdiff:**
```
cmpq %rsi, %rdi  # x:y
jle    .L2
movq  %rdi, %rax
subq  %rsi, %rax
ret
.L2:     # x <= y
movq  %rsi, %rax
subq  %rdi, %rax
ret
```

Possibility I: the conditional branch is not taken: no penalty
Conditional Branch in a pipeline

<table>
<thead>
<tr>
<th></th>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Write back</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>cmpq %rsi, %rdi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t2</td>
<td>jle .L2</td>
<td>cmpq %rsi, %rdi</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t3</td>
<td>movq %rdi, %rax</td>
<td>jle .L2</td>
<td></td>
<td>cmpq %rsi, %rdi</td>
<td></td>
</tr>
<tr>
<td>t4</td>
<td>subq %rax, %rax</td>
<td>movq %rdi, %rax</td>
<td>jle .L2</td>
<td></td>
<td>cmpq %rsi, %rdi</td>
</tr>
<tr>
<td>t5</td>
<td>movq %rsi, %rax</td>
<td></td>
<td></td>
<td>jle .L2</td>
<td>cmpq %rsi, %rdi</td>
</tr>
</tbody>
</table>

Possibility II: the conditional branch is taken: two empty time units due to misprediction
## Conditional Branch in a pipeline

<table>
<thead>
<tr>
<th></th>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Write back</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>t1</strong></td>
<td>cmpq %rsi, %rdi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>t2</strong></td>
<td>jle .L2</td>
<td>cmpq %rsi, %rdi</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>t3</strong></td>
<td>movq %rdi, %rax</td>
<td>jle .L2</td>
<td>cmpq %rsi, %rdi</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>t4</strong></td>
<td>subq %rsi, %rax</td>
<td>movq %rdi, %rax</td>
<td>jle .L2</td>
<td>cmpq %rsi, %rdi</td>
<td></td>
</tr>
<tr>
<td><strong>t5</strong></td>
<td>movq %rsi, %rax</td>
<td></td>
<td></td>
<td>jle .L2</td>
<td>cmpq %rsi, %rdi</td>
</tr>
</tbody>
</table>

### absdiff:

```assembly
absdiff:
    cmpq %rsi, %rdi  # x:y
    jle .L2
    movq %rdi, %rax
    subq %rsi, %rax
    ret
.L2:       # x <= y
    movq %rsi, %rax
    subq %rdi, %rax
    ret
```

*We would rather predict a branch outcome!*
Branch prediction

- Predict whether a branch is taken
Branch prediction

- Predict whether a branch is taken
- No penalty if correct
Branch prediction

- Predict whether a branch is taken
- No penalty if correct
- Huge penalty if incorrect
Branch prediction

- Predict whether a branch is taken
- No penalty if correct
- Huge penalty if incorrect
  - Modern processors have a deep pipeline

<table>
<thead>
<tr>
<th>Microarchitecture</th>
<th>Pipeline stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>P5 (Pentium)</td>
<td>5</td>
</tr>
<tr>
<td>P6 (Pentium 3)</td>
<td>10</td>
</tr>
<tr>
<td>P6 (Pentium Pro)</td>
<td>14</td>
</tr>
</tbody>
</table>
Branch prediction

• **Predict whether a branch is taken**
• **No penalty if correct**
• **Huge penalty if incorrect**
  • Modern processors have a deep pipeline

<table>
<thead>
<tr>
<th>Microarchitecture</th>
<th>Pipeline stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>P5 (Pentium)</td>
<td>5</td>
</tr>
<tr>
<td>P6 (Pentium 3)</td>
<td>10</td>
</tr>
<tr>
<td>P6 (Pentium Pro)</td>
<td>14</td>
</tr>
</tbody>
</table>

• **Multiple instructions can be executed in a unit of time**
Branch prediction

- Predict whether a branch is taken
- No penalty if correct
- Huge penalty if incorrect
  - Modern processors have a deep pipeline

<table>
<thead>
<tr>
<th>Microarchitecture</th>
<th>Pipeline stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>P5 (Pentium)</td>
<td>5</td>
</tr>
<tr>
<td>P6 (Pentium 3)</td>
<td>10</td>
</tr>
<tr>
<td>P6 (Pentium Pro)</td>
<td>14</td>
</tr>
</tbody>
</table>

- Multiple instructions can be executed in a unit of time
- 14 stage pipeline, 4 instructions/time unit => 56 possible instructions worth of work wasted
Branch prediction

- Predict whether a branch is taken
- No penalty if correct
- Huge penalty if incorrect
  - Modern processors have a deep pipeline

<table>
<thead>
<tr>
<th>Microarchitecture</th>
<th>Pipeline stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>P5 (Pentium)</td>
<td>5</td>
</tr>
<tr>
<td>P6 (Pentium 3)</td>
<td>10</td>
</tr>
<tr>
<td>P6 (Pentium Pro)</td>
<td>14</td>
</tr>
</tbody>
</table>

- Multiple instructions can be executed in a unit of time
- 14 stage pipeline, 4 instructions/time unit => 56 possible instructions worth of work wasted
How about no branch prediction but do both branches?

- Do the work of both paths
How about no branch prediction but do both branches?

- Do the work of both paths
- Throw away the work from the wrong path
How about no branch prediction but do both branches?

- Do the work of both paths
- Throw away the work from the wrong path
- Waste is up to 50%
How about no branch prediction but do both branches?

- Do the work of both paths
- Throw away the work from the wrong path
- Waste is up to 50% (VS branch prediction)
  - branch prediction: no penalty if correct; huge penalty if wrong (14 stage pipeline, 4 instructions/time unit => roughly 50 possible instructions worth of work wasted)
How about no branch prediction but do both branches?

- Do the work of both paths
- Throw away the work from the wrong path
- Waste is up to 50% (VS branch prediction)
  - branch prediction: no penalty if correct; huge penalty if wrong (14 stage pipeline, 4 instructions/time unit => roughly 50 possible instructions worth of work wasted)
- Different types of condition branches
How about no branch prediction but do both branches?

- Do the work of both paths
- Throw away the work from the wrong path
- Waste is up to 50% (VS branch prediction)
  - branch prediction: no penalty if correct; huge penalty if wrong (14 stage pipeline, 4 instructions/time unit => roughly 50 possible instructions worth of work wasted)
- Different types of condition branches
  - Large conditional statements
How about no branch prediction but do both branches?

- Do the work of both paths
- Throw away the work from the wrong path
- Waste is up to 50% (VS branch prediction)
  - branch prediction: no penalty if correct; huge penalty if wrong (14 stage pipeline, 4 instructions/time unit => roughly 50 possible instructions worth of work wasted)
- Different types of condition branches
  - Large conditional statements: the amount of waste with doing both branches is larger than the penalty of branch misprediction
How about no branch prediction but do both branches?

- Do the work of both paths
- Throw away the work from the wrong path
- Waste is up to 50% (VS branch prediction)
  - branch prediction: no penalty if correct; huge penalty if wrong (14 stage pipeline, 4 instructions/time unit => roughly 50 possible instructions worth of work wasted)
- Different types of condition branches
  - Large conditional statements: the amount of waste with doing both branches is larger than the penalty of branch misprediction (winner: branch prediction wins)
How about no branch prediction but do both branches?

- Do the work of both paths
- Throw away the work from the wrong path
- Waste is up to 50% (VS branch prediction)
  - branch prediction: no penalty if correct; huge penalty if wrong (14 stage pipeline, 4 instructions/time unit => roughly 50 possible instructions worth of work wasted)
- Different types of condition branches
  - Large conditional statements: the amount of waste with doing both branches is larger than the penalty of branch misprediction (winner: branch prediction wins)
  - Small conditional statements:
How about no branch prediction but do both branches?

- Do the work of both paths
- Throw away the work from the wrong path
- Waste is up to 50% (VS branch prediction)
  - branch prediction: no penalty if correct; huge penalty if wrong (14 stage pipeline, 4 instructions/time unit => roughly 50 possible instructions worth of work wasted)
- Different types of condition branches
  - Large conditional statements: the amount of waste with doing both branches is larger than the penalty of branch misprediction (winner: branch prediction wins)
  - Small conditional statements: depending on the accuracy of branch prediction.
How about no branch prediction but do both branches?

- Do the work of both paths
- Throw away the work from the wrong path
- Waste is up to 50% (VS branch prediction)
  - branch prediction: no penalty if correct; huge penalty if wrong (14 stage pipeline, 4 instructions/time unit => roughly 50 possible instructions worth of work wasted)
- Different types of condition branches
  - Large conditional statements: the amount of waste with doing both branches is larger than the penalty of branch misprediction (winner: branch prediction wins)
  - Small conditional statements: depending on the accuracy of branch prediction. (The less branch prediction accuracy is, the more in favor of doing both branches).
Conditional Move Example

- Generation
  
  gcc -O -S control.c
Conditional Move Example

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

### Register Use(s)
- **%rdi**: Argument \( x \)
- **%rsi**: Argument \( y \)
- **%rax**: Return value

### Generation
```
gcc -O -S control.c
```

### Assembly Code
```
absdiff:
  movq  %rdi, %rax  # x
  subq  %rsi, %rax  # result = x-y
  movq  %rsi, %rdx
  subq  %rdi, %rdx  # eval = y-x
  cmpq  %rsi, %rdi  # x:y
  cmovle %rdx, %rax  # if <=, result = eval
  ret
```
LOOP
jump conditional
loop:
  ...
conditional:
cmp 8, %rdi
jle loop

while(%rdi<=8) {
  ...  
}
i-clicker question

• How do we implement looping in assembly?
  A. Use for loop
  B. Use while loop
  C. Use conditional jump to jump back
  D. Use unconditional jump to jump back
i-clicker question

• How do we implement looping in assembly? Sol: C
  A. Use for loop
  B. Use while loop
  C. Use conditional jump to jump back
  D. Use unconditional jump to jump back
i-clicker question

• Which one of the following assembly code does not contain a loop?

A. movl $0, %eax
   .L11:
   movq %rdi, %rdx
   andl $1, %edx
   addq %rdx, %rax
   shrq 1, %rdi
   jne .L11

B. movl $0, %eax
   jmp .L13
   .L14:
   movq %rdi, %rdx
   andl $1, %edx
   addq %rdx, %rax
   shrq %rdi
   .L13:
   testq %rdi, %rdi
   jne .L11

C. movl $0, %eax
   movl $0, %ecx
   jmp .L16
   .L17:
   movq %rdi, %rdx
   shrq %cl, %rdx
   addq %rdx, %rax
   addl $1, %ecx
   .L16:
   cmpl $63, %ecx
   jbe .L17

D. testq %rdi, %rdi
   jns .L20
   movl $1, %eax
   ret
   .L20:
   movl $0, %eax
   ret
i-clicker question

• Which one of the following assembly code does not contain a loop? **Sol: D**

A.

```
    movl $0, %eax
.L11:
    movq %rdi, %rdx
    andl $1, %edx
    addq %rdx, %rax
    shrq 1, %rdi
    jne .L11
```

B.

```
    movl $0, %eax
    jmp .L13
.L14:
    movq %rdi, %rdx
    andl $1, %edx
    addq %rdx, %rax
    shrq %rdi
.L13:
    testq %rdi, %rdi
    jne .L14
```

C.

```
    movl $0, %eax
    movl $0, %ecx
    jmp .L16
.L17:
    movq %rdi, %rdx
    shrq %cl, %rdx
    addq %rdx, %rax
    shrq %rdi
.L16:
    cmpl $63, %ecx
    jbe .L17
```

D. `testq %rdi, %rdi
    jns .L20
    movl $1, %eax
    ret`