Computer Systems Principles

x86-64 Assembly (Part 2)

UMASSCS

SCHOOL OF COMPUTER SCIENCE

Objectives

- **x86-64 Assembly Language**
  - To learn about condition codes
  - To learn about conditional branches
  - To learn about loop

CONDITION CODES

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) **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) **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) **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) **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 < (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) || (a<0 && b<0 && 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 < (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) || (a<0 && b<0 && t>=0)`

- **Not set by `leaq` instruction**

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 < (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) || (a<0 && 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) || (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

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)

odes

CMP S1, S2
cmpb: compare b
cmpw: compar
cmpl: compare
cmpq: compare q

mov S, D
movb: mov byte
movw: mov w
movl: mov lor
movq: mov

add S, D
addb: add byte
addw: add word
addl: add long word
addq: add quad word

SetX	Condition	
`sete`	`ZF`	
`setne`	`~ZF`	Not
`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)

odes

CMP S1, S2
cmpb: compare b
cmpw: compar
cmpl: compare
cmpq: compare q

mov S, D
movb: mov byte
movw: mov w
movl: mov lor
movq: mov

add S, D
addb: add byte
addw: add word
addl: add long word
addq: add quad word

SetX	Condition	
`sete`	`ZF`	
`setne`	`~ZF`	Not
`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)

Reading Condition Codes

- **SetX Instructions**
  - 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)

Reading Condition Codes

- **SetX Instructions**
  – 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)

Reading Condition Codes

- **SetX Instructions**
  - 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)

Reading Condition Codes

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

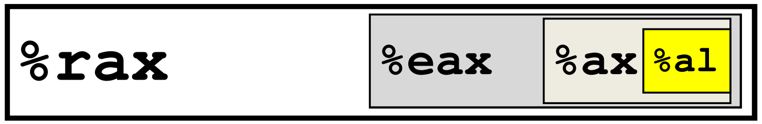
SetX	Condition	Description
`sete`	`ZF`	Equal / Zero
`setne`	`~ZF`	Not Equal / Not Zero
`sets`	`SF`	Negative
`setns`	`~SF`	Nonnegative
`setg`	`~(SF^OF)&~ZF`	Greater (Signed)
`setge`	`~(SF^OF)`	Greater or Equal (Signed)
`setl`	`(SF^OF)`	Less (Signed)
`setle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`seta`	`~CF&~ZF`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

Reading Condition Codes (Cont.)

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

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

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



Reading Condition Codes (Cont.)

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

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

```
    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.)

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

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

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

`%rax` `%eax` `%ax` `%al`

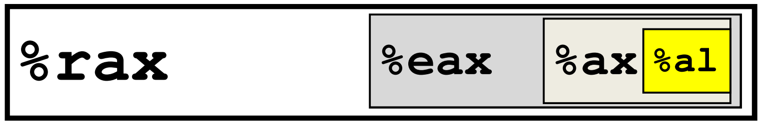
Reading Condition Codes (Cont.)

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

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

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

– Does not alter remaining bytes



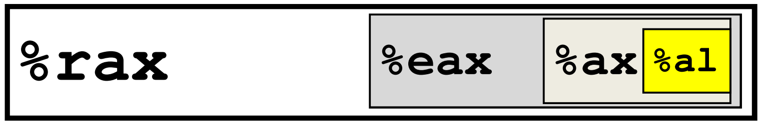
Reading Condition Codes (Cont.)

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

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

```
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.)

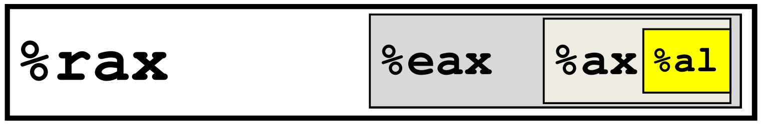
```c
int gt (long x, long y)
{
  return x > y;
}
```

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

```
    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



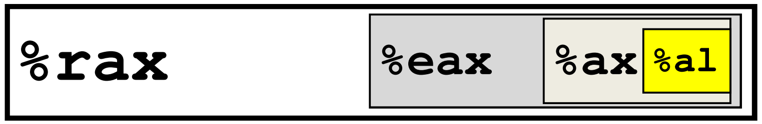
Reading Condition Codes (Cont.)

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

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

```
    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



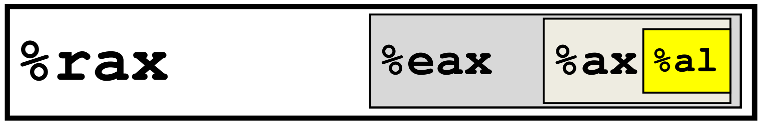
Reading Condition Codes (Cont.)

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

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

```
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
- 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, >

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? 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

jX	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)
`jb`	`CF`	Below (unsigned)

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

```
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;
}
```

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;
}
```

```
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
```

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

Conditional Branch in a pipeline

	Fetch	Decode	Execute	Memory	Write back
t1	cmpq %rsi, %rdi				
t2					
t3					
t4					
t5					

```
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

	Fetch	Decode	Execute	Memory	Write back
t1	cmpq %rsi, %rdi				
t2	jle .L2	cmpq %rsi, %rdi			
t3					
t4					
t5					

```
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

	Fetch	Decode	Execute	Memory	Write back
t1	cmpq %rsi, %rdi				
t2	jle .L2	cmpq %rsi, %rdi			
t3	?	jle .L2	cmpq %rsi, %rdi		
t4					
t5					

```
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

	Fetch	Decode	Execute	Memory	Write back
t1	cmpq %rsi, %rdi				
t2	jle .L2	cmpq %rsi, %rdi			
t3	movq %rdi, %rax	jle .L2	cmpq %rsi, %rdi		
t4					
t5					

```
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

	Fetch	Decode	Execute	Memory	Write back
t1	cmpq %rsi, %rdi				
t2	jle .L2	cmpq %rsi, %rdi			
t3	movq %rdi, %rax	jle .L2	cmpq %rsi, %rdi		
t4	subq %rsi, %rax	movq %rdi, %rax	jle .L2	cmpq %rsi, %rdi	
t5					

```
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

	Fetch	Decode	Execute	Memory	Write back
t1	cmpq %rsi, %rdi				
t2	jle .L2	cmpq %rsi, %rdi			
t3	movq %rdi, %rax	jle .L2	cmpq %rsi, %rdi		
t4	subq %rsi, %rax	movq %rdi, %rax	jle .L2	cmpq %rsi, %rdi	
t5					

```
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

	Fetch	Decode	Execute	Memory	Write back
t1	cmpq %rsi, %rdi				
t2	jle .L2	cmpq %rsi, %rdi			
t3	movq %rdi, %rax	jle .L2	cmpq %rsi, %rdi		
t4	subq %rsi, %rax	movq %rdi, %rax	jle .L2	cmpq %rsi, %rdi	
t5					

```
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

	Fetch	Decode	Execute	Memory	Write back
t1	cmpq %rsi, %rdi				
t2	jle .L2	cmpq %rsi, %rdi			
t3	movq %rdi, %rax	jle .L2	cmpq %rsi, %rdi		
t4	~~subq %rsi, %rax~~	~~movq %rdi, %rax~~	jle .L2	cmpq %rsi, %rdi	
t5					

```
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

	Fetch	Decode	Execute	Memory	Write back
t1	cmpq %rsi, %rdi				
t2	jle .L2	cmpq %rsi, %rdi			
t3	movq %rdi, %rax	jle .L2	cmpq %rsi, %rdi		
t4	~~subq %rsi, %rax~~	~~movq %rdi, %rax~~	jle .L2	cmpq %rsi, %rdi	
t5				jle .L2	cmpq %rsi, %rdi

```
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

	Fetch	Decode	Execute	Memory	Write back
t1	cmpq %rsi, %rdi				
t2	jle .L2	cmpq %rsi, %rdi			
t3	movq %rdi, %rax	jle .L2	cmpq %rsi, %rdi		
t4	~~subq %rsi, %rax~~	~~movq %rdi, %rax~~	jle .L2	cmpq %rsi, %rdi	
t5	movq %rsi, %rax			jle .L2	cmpq %rsi, %rdi

```
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

	Fetch	Decode	Execute	Memory	Write back
t1	cmpq %rsi, %rdi				
t2	jle .L2	cmpq %rsi, %rdi			
t3	movq %rdi, %rax	jle .L2	cmpq %rsi, %rdi		
t4	subq %rsi, %rax	movq %rdi, %rax	jle .L2	cmpq %rsi, %rdi	
t5					

```
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

	Fetch	Decode	Execute	Memory	Write back
t1	cmpq %rsi, %rdi				
t2	jle .L2	cmpq %rsi, %rdi			
t3	movq %rdi, %rax	jle .L2	cmpq %rsi, %rdi		
t4	~~subq %rsi, %rax~~	~~movq %rdi, %rax~~	jle .L2	cmpq %rsi, %rdi	
t5	movq %rsi, %rax			jle .L2	cmpq %rsi, %rdi

```
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: two empty time units due to misprediction

Conditional Branch in a pipeline

	Fetch	Decode	Execute	Memory	Write back
t1	cmpq %rsi, %rdi				
t2	jle .L2	cmpq %rsi, %rdi			
t3	movq %rdi, %rax	jle .L2	cmpq %rsi, %rdi		
t4	~~subq %rsi, %rax~~	~~movq %rdi, %rax~~	jle .L2	cmpq %rsi, %rdi	
t5	movq %rsi, %rax			jle .L2	cmpq %rsi, %rdi

```
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**

Microarchitecture	Pipeline stages
P5 (Pentium)	5
P6 (Pentium 3)	10
P6 (Pentium Pro)	14

Branch prediction

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

Microarchitecture	Pipeline stages
P5 (Pentium)	5
P6 (Pentium 3)	10
P6 (Pentium Pro)	14

  - **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**

Microarchitecture	Pipeline stages
P5 (Pentium)	5
P6 (Pentium 3)	10
P6 (Pentium Pro)	14

  - **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**

Microarchitecture	Pipeline stages
P5 (Pentium)	5
P6 (Pentium 3)	10
P6 (Pentium Pro)	14

  - **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

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

- **Generation**

  `gcc –O –S control.c`

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

```
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    .L14
```

C.

```
        movl   $0, %eax
        movl   $0, %ecx
        jmp    .L16
.L17:
        movq   %rdi, %rdx
        shrq   %cl, %rdx
        andl   $1, %edx
        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
      andl   $1, %edx
      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
```