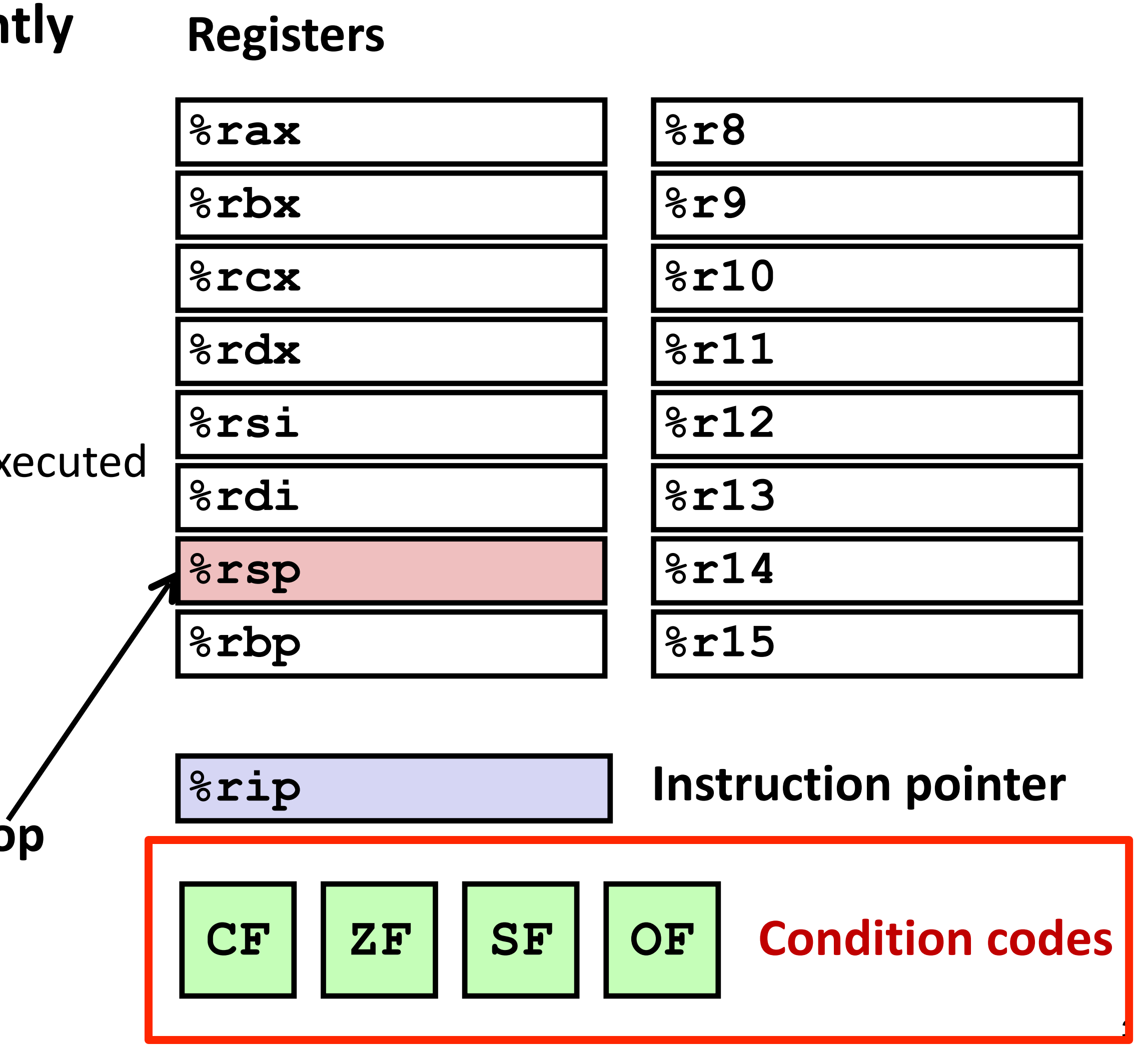
Machine-Level Programming II: Control

Today

- **Control: Condition codes**
- Conditional branches
- Loops
- Switch Statements

Processor State (x86-64, Partial)

- CPU Needs to Store Information About Currently Executing Program
  - Temporary data (`%rax`, …)
  - Information about the stack (`%rsp`)
  - Which instruction is being executed (`%rip`, …)
  - Status of recent tests (CF, ZF, SF, OF)



Registers

`%rax` `%rbx` `%rcx` `%rdx` `%rsi` `%rdi` `%rsp` `%rbp`

`%r8` `%r9` `%r10` `%r11` `%r12` `%r13` `%r14` `%r15`

`%rip` Instruction pointer

CF ZF SF OF Condition codes

Current stack top

Condition Codes (Implicit Setting)

- **Single bit registers**
  - **SF** Sign Flag (for signed)
  - **ZF** Zero Flag

  CF Carry Flag (for unsigned)

  OF Overflow Flag (for signed)

- **Implicitly set (think of it as side effect) by arithmetic operations**
  - Example: **`addq`** *Src, Dest* ↔ **`t = a+b`**
  - **ZF set** if `t == 0`
  - **SF set** if `t < 0` (as signed)
  - **CF set** if carry out from most significant bit (unsigned overflow)
  - **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 (Explicit Setting: Compare)

- **Arithmetic Operations Set Condition Codes Implicitly**
- **Explicit Setting by Compare Instruction**
  - `cmpq` *Src2*, *Src1*
  - `cmpq b,a` like computing `a-b` without setting destination

  - **ZF set** if `(a-b) == 0`
  - **SF set** if `(a-b) < 0` (as signed)
  - **CF set** if carry out from most significant bit (used for unsigned comparisons)
  - **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`**
  - Why the OF and CF flags are not set?
    - Because there is no overflow in bitwise &
  - Why two operands?
    - Useful to have one of the operands be a mask

Summary

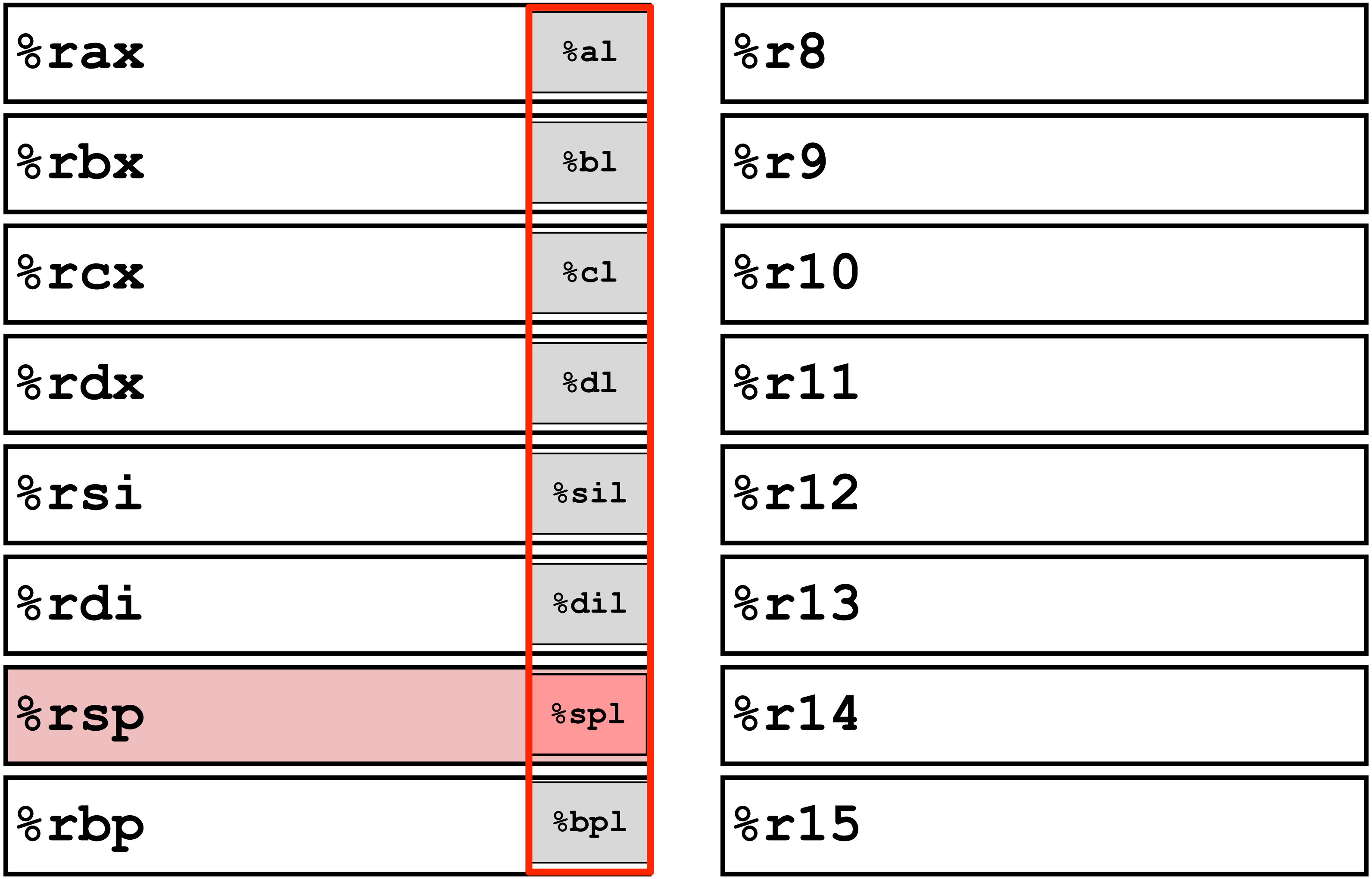
- **How to set condition codes?**
  - **Implicitly**
    - Using arithmetic operations
  - **Explicitly**
    - cmp b,a <=> a - b
    - test b,a <=> b & a

How to Read the Condition Codes?

Reading Condition Codes

SetX Instructions

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

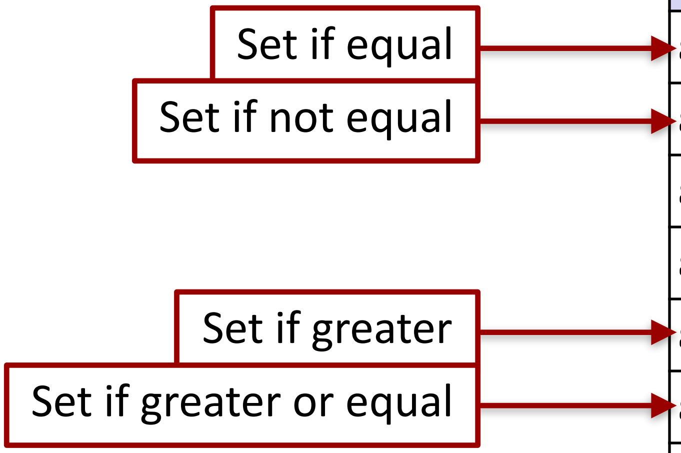


Reading Condition Codes

SetX Instructions

- Does not alter remaining 7 bytes

```
sete    %al      # If ZF == 1, set %al to 1
                # otherwise set it to 0
```



Set if equal

Set if not equal

Set if greater

Set if greater or equal

SetX	Description
`sete`	Equal / Zero
`setne`	Not Equal / Not Zero
`sets`	Negative
`setns`	Nonnegative
`setg`	Greater (Signed)
`setge`	Greater or Equal (Signed)
`setl`	Less (Signed)
`setle`	Less or Equal (Signed)
`seta`	Above (unsigned)
`setb`	Below (unsigned)

Reading Condition Codes

Example

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

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

```
    cmpq   %rsi, %rdi   # (%rdi - %rsi) <=> (x - y)
    setg   %al          # Set when > 0
```

SetX Instructions

- Once the lower byte is set, use **`movzbq`** to set the remaining bits to 0

Practice

```
int eq(long x, long y)
{
  return (x == y);
}
```

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

```
    cmpq   %rsi, %rdi    # (%rdi - %rsi) <=> (x - y)
    sete   %al           # Set when = 0
    movzbq %al, %rax     # Zero rest of %rax
    ret
```

Today

- Control: Condition codes
- **Conditional branches**
- Loops
- Switch Statements

Expressing with Goto Code

- C allows `goto` statement
- Jump to position designated by label

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

```
long absdiff_j
  (long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    goto Done;
 Else:
    result = y-x;
 Done:
    return result;
}
```

Jumping

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

jX	Description
`jmp`	Unconditional
`je`	Equal / Zero
`jne`	Not Equal / Not Zero
`js`	Negative
`jns`	Nonnegative
`jg`	Greater (Signed)
`jge`	Greater or Equal (Signed)
`jl`	Less (Signed)
`jle`	Less or Equal (Signed)
`ja`	Above (unsigned)
`jb`	Below (unsigned)

Conditional Branch Example (Old Style)

- Generation

```
gcc –Og -S –fno-if-conversion control.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

```
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
```

Practice

```
long f2
   (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

```
f2:
   cmpq     %rsi, %rdi   # x - y
   jge      .L4
   movq     %rdi, %rax
   subq     %rsi, %rax
   ret
.L4:        # x >= y
   movq     %rsi, %rax
   subq     %rdi, %rax
   ret
```

General Conditional Expression Translation (Using Branches)

C Code

```
val = Test ? Then_Expr : Else_Expr;
```

```
val = x>y ? x-y : y-x;
```

Goto Version

```
   ntest = !Test;
   if (ntest) goto Else;
   val = Then_Expr;
   goto Done;
Else:
   val = Else_Expr;
Done:
   . . .
```

- Create separate code regions for then & else expressions
- Execute appropriate one

Using Conditional Moves

Conditional Move Instructions

- Instruction supports:

  if (Test)   Dest ←Src

- GCC tries to use them
  - But, only when known to be safe

Why use conditional moves?

- Branches are very disruptive to instruction flow through pipelines
- Conditional moves do not require control transfer

C Code

```
val  =  Test
     ?  Then_Expr
     :  Else_Expr ;
```

Goto Version

```
val  =  Then_Expr ;
eval  =  Else_Expr ;
nt  =  ! Test ;
if  (nt)  val  =  eval;
```

Practice

C Code

```
val = (x>0) ? 1 : -1;
```

Conditional Move Version

```
val = 1;
eval = -1;
if (x<=0) val = eval;
```

Conditional Move Example

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

Register	Use(s)
`%rdi`	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
```

Bad Cases for Conditional Move

Expensive Computations

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

- Both values get computed
- Only makes sense when computations are very simple

Risky Computations

```
val = p ? *p : 0;
```

- Both values get computed
- May have undesirable effects

Computations with side effects

```
val = x > 0 ? x*=7 : x+=3;
```

- Both values get computed
- Must be side-effect free

Practice

```
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

```
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
   cmovne   %rdx, %rax  # if !=, result = eval
   ret
```

Condition Codes (Implicit Setting)

- **Single bit registers**
  - **SF** Sign Flag (for signed)
  - **ZF** Zero Flag

CF Carry Flag (for unsigned)

OF Overflow Flag (for signed)

- **Set**
  - **Implicitly set**
    - Using arithmetic operations
  - **Explicitly set**
    - cmp
    - test

- **Read**
  - **Copy**
    - set
  - **Take action**
    - jmp
    - cmov