CSCI-UA.0201

Computer Systems Organization

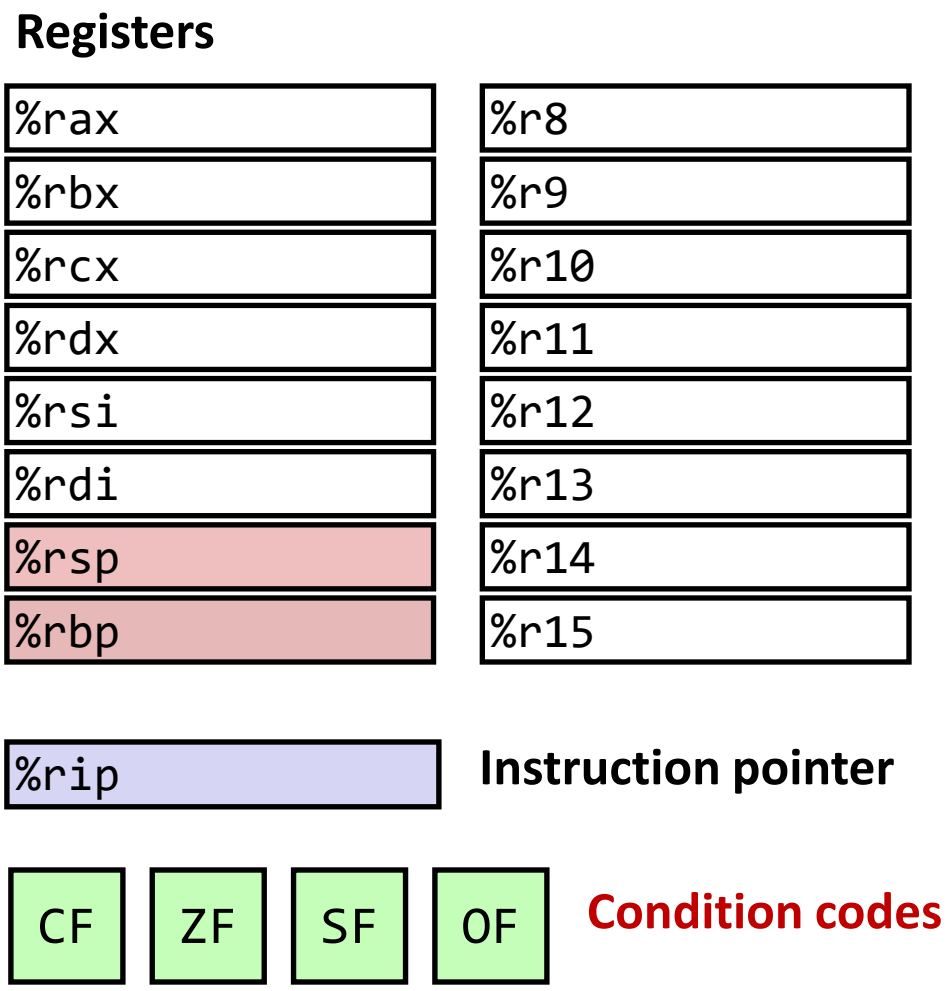
Machine Level – Control

Thomas Wies
wies@cs.nyu.edu
https://cs.nyu.edu/wies

Control

Processor State (x86-64, Partial)

- Information about currently executing program
  - Temporary data (`%rax`, …)
  - Location of runtime stack (`%rsp`, `%rbp`)
  - Location of current code control point (`%rip`)
  - Status of recent tests (`CF`, `ZF`, `SF`, `OF`)



Setting Condition Codes Implicitly

- Can be implicitly set by arithmetic operations

  Example: `addq` *Src*,*Dest* `(t = a+b)`

  CF (Carry flag) set if carry out from most significant (31-st) bit (unsigned overflow)

  ZF (Zero flag) set if `t == 0`

  SF (Sign flag) set if `t < 0` (as signed)

  OF (Overflow flag) set if signed overflow

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

- Condition codes not set by `lea` instruction!

Setting Condition Codes Explicitly

- Can also be explicitly set

  `cmpl b,a` set condition codes based on computing `a-b` without storing the result in any 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 `(a-b)` results in signed overflow

Reading Condition Codes

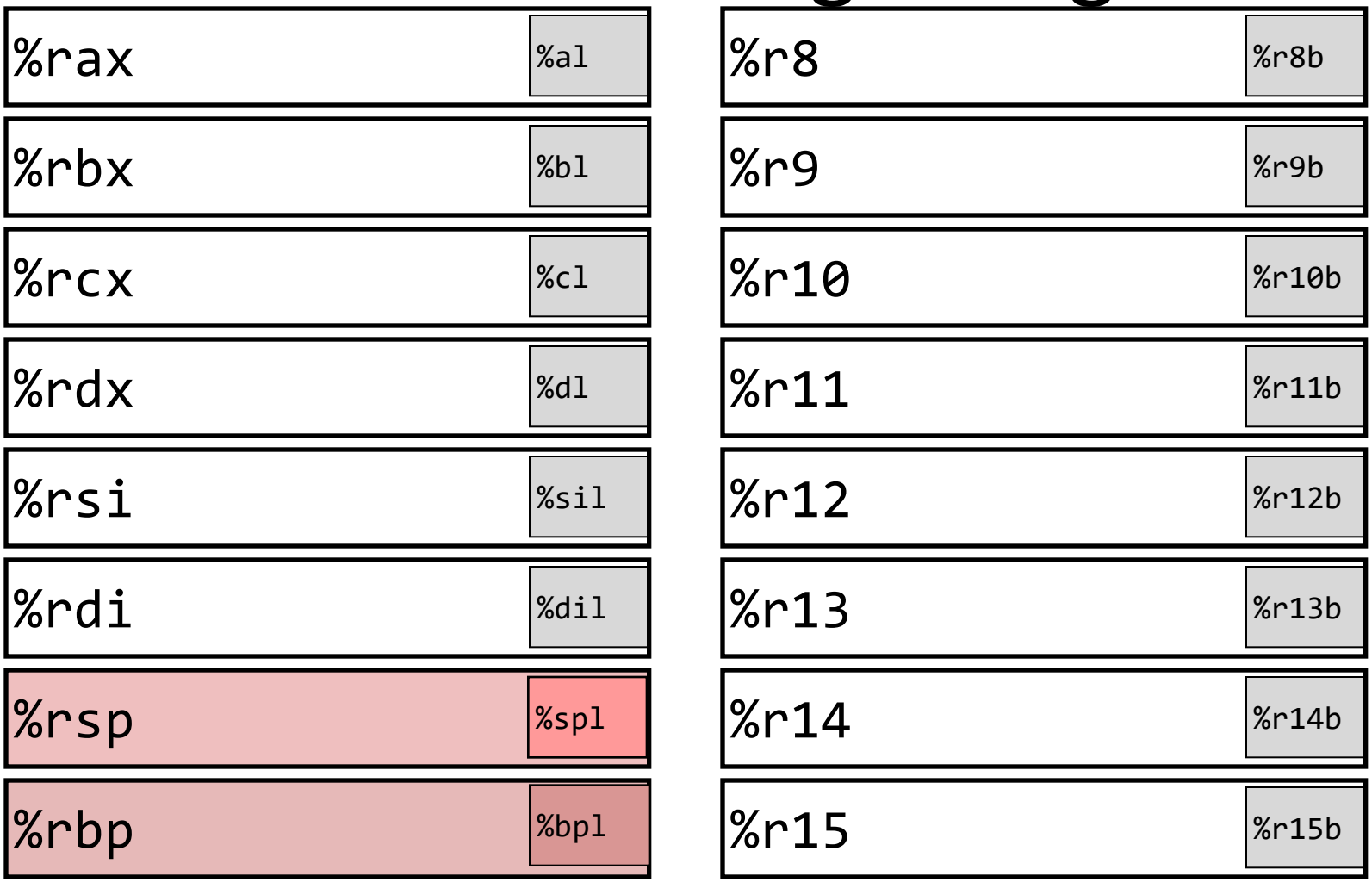
- **`setX` *dest***

Sets the lower byte of *dest* based on combinations of condition codes and does not alter remaining 7 bytes. Destination can also be memory location.

These instructions are usually used after a comparison.

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)**

Recall: x86-64 Integer Registers



– Can reference low-order byte

Example

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

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

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

What do we do with condition codes?

1. Setting a single byte to 0 or 1 based on some combination of the condition codes.
2. Conditionally jump to other parts of the program.
3. Conditionally transfer data.

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)**

Indirect jump

jmp * Operand

Unconditional jump

Can be:

- register
- Memory address using any of the addressing modes we saw.

Example

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

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

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

What do we do with condition codes?

1. Setting a single byte to 0 or 1 based on some combination of the condition codes.
2. Conditionally jump to other parts of the program.
3. Conditionally transfer data.

Conditional Moves

- Conditional Move Instructions
  - Instruction supports:
    if (Test) Dest ← Src
  - Supported in post-1995 x86 processors
  - GCC tries to use them
    - But, only when known to be safe
- Why?
  - 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

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

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

What we have seen so far …

- The arithmetic and logic operations can be applied to data of size 1(b), 2(w), 4(l), and 8(q) bytes.
- Condition codes are needed in order to implement conditional branches/jumps.
- The compiler uses the different condition codes and different jump formats to implement the different control structures we have in high-level languages: for-loop, do-while, while, switch, if-then-else, etc.

How does the compiler translate loops?

- do-while
- while
- for

“Do-While” Loop Compilation

C Code

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

Goto Version

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

- Count number of 1s in argument x
- Use conditional branch to either continue looping or to exit loop

“Do-While” Loop Compilation

Register	Use(s)
%rdi	Argument **x**
%rax	**Result**

Goto Version

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

Assembly Version

```
   movl    $0, %rax    #  result = 0
.L2:                   # loop:
   movq    %rdi, %rdx  #  t = x
   andl    $1, %rdx    #  t &= 0x1
   addq    %rdx, %rax  #  result += t
   shrq    %rdi        #  x >>= 1
   jne     .L2         #  if (x) goto loop
   ret
```

General “Do-While” Translation

C Code

```
do
   Body
   while (Test);
```

Goto Version

```
loop:
   Body
   if (Test)
      goto loop
```

General “While” Translation #1

- “Jump-to-middle” translation

While version

```
while (Test)
  Body
```

Goto Version

```
  goto test;
loop:
  Body
test:
  if (Test)
    goto loop;
done:
```

While Loop Example #1

C Code

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

Jump to Middle

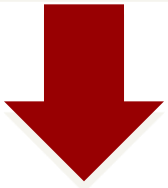
```
long pcount_goto_jtm
  (unsigned long x) {
  long result = 0;
  goto test;
 loop:
  result += x & 0x1;
  x >>= 1;
 test:
  if(x) goto loop;
  return result;
}
```

- Compare to do-while version of function
- Initial goto starts loop at test

General “While” Translation #2

While version

```
while (Test)
  Body
```



Do-While Version

```
  if (!Test)
    goto done;
  do
    Body
  while(Test);
done:
```

- “Do-while” conversion

Goto Version

```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

While Loop Example #2

C Code

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

Do-While Version

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

- Compare to do-while version of function
- Initial conditional guards entrance to loop