CSCI-UA.0201

Computer Systems Organization

Machine Level – Assembly (x86-64) basics

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

Arithmetic & Logic Operations

Arithmetic Expression Example

```c
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
  leaq    (%rdi,%rsi), %rax
  addq    %rdx, %rax
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx
  leaq    4(%rdi,%rdx), %rcx
  imulq   %rcx, %rax
  ret
```

Understanding Arithmetic Expression Example

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
  leaq    (%rdi,%rsi), %rax
  addq    %rdx, %rax
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx
  leaq    4(%rdi,%rdx), %rcx
  imulq   %rcx, %rax
  ret
```

Register	Use(s)
`%rdi`	Argument **x**
`%rsi`	Argument **y**
`%rdx`	Argument **z**
`%rax`	`t1, t2, rval`
`%rdx`	`t4`
`%rcx`	`t5`

Understanding Arithmetic Expression Example

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
  leaq    (x,y), rval
  addq    z, rval
  leaq    (y,y,2), t4
  salq    $4, t4
  leaq    4(x,z), t5
  imulq   t5, rval
  ret
```

Multiplication

- **Unsigned**
  - form 1: `imulq s, d`
    - `d = s * d`
    - multiply two 64-bit operands and put the result in 64-bit operand
  - form 2: `mulq s`
    - one operand is `%rax`
    - The other operand given in the instruction
    - product is stored in `%rdx` (high-order part) and `%rax` (low order part) → full 128-bit result
- **Signed**
  - form 1: `imulq s, d`
    - `d = s * d`
    - multiply two 64-bit operands and put the result in 64-bit operand
  - form 2: `imulq s`
    - one operand is `%rax`
    - The other operand given in the instruction
    - product is stored in `%rdx` (high-order part) and `%rax` (low order part) → full 128-bit result

Division

- **Unsigned**
  - `divq s`
    - Dividend given in `%rdx` (high order) and `%rax` (low order)
    - Divisor is `s`
    - Quotient stored in `%rax`
    - Remainder stored in `%rdx`
- **Signed**
  - `idivq s`
    - Dividend given in `%rdx` (high order) and `%rax` (low order)
    - Divisor is `s`
    - Quotient stored in `%rax`
    - Remainder stored in `%rdx`

Useful Instruction for Division

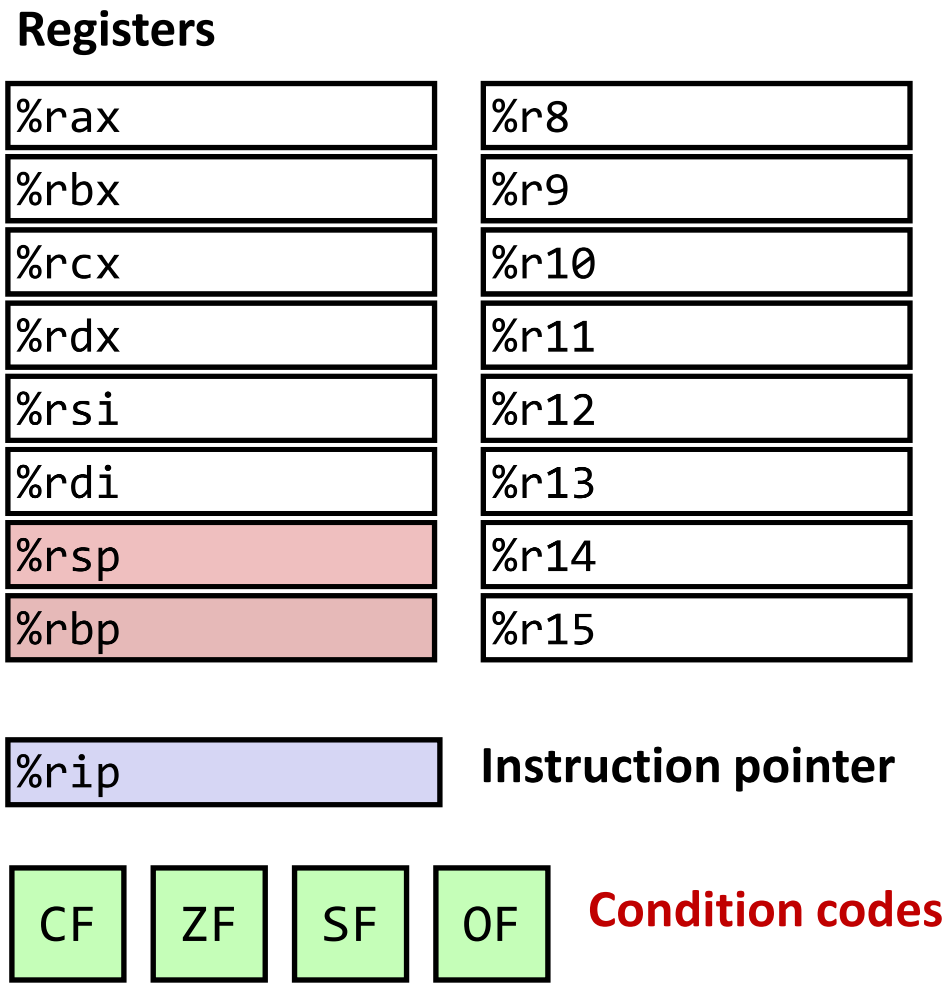
`cqto`

- convert quad word to octal word
- no operands
- takes the sign bit from `%rax` and replicates it in all bits of `%rdx`
- effect: sign extend 64-bit signed `%rax` to 128-bit signed `%rdx:%rax`.

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!

Effect of Logical Operations

- The carry and overflow flags are set to zero.
- For shift instructions:
  - The carry flag is set to the value of the last bit shifted out.
  - Overflow flag is set to zero.

INC and DEC instructions

- Affect the overflow and zero flags
- Leave carry flag unchanged

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

Setting Condition Codes Explicitly

- Can also be explicitly set

  `testq b,a` set condition codes based on value of **`(a & b)`** *without storing the result in any destination*

ZF set if `(a & b) == 0`

SF set if `(a & b) < 0`

Setting Condition Codes

Important

The processor does not know if you are using signed or unsigned integers.

OF and CF are set for every arithmetic operation.

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.

Reading Condition Codes

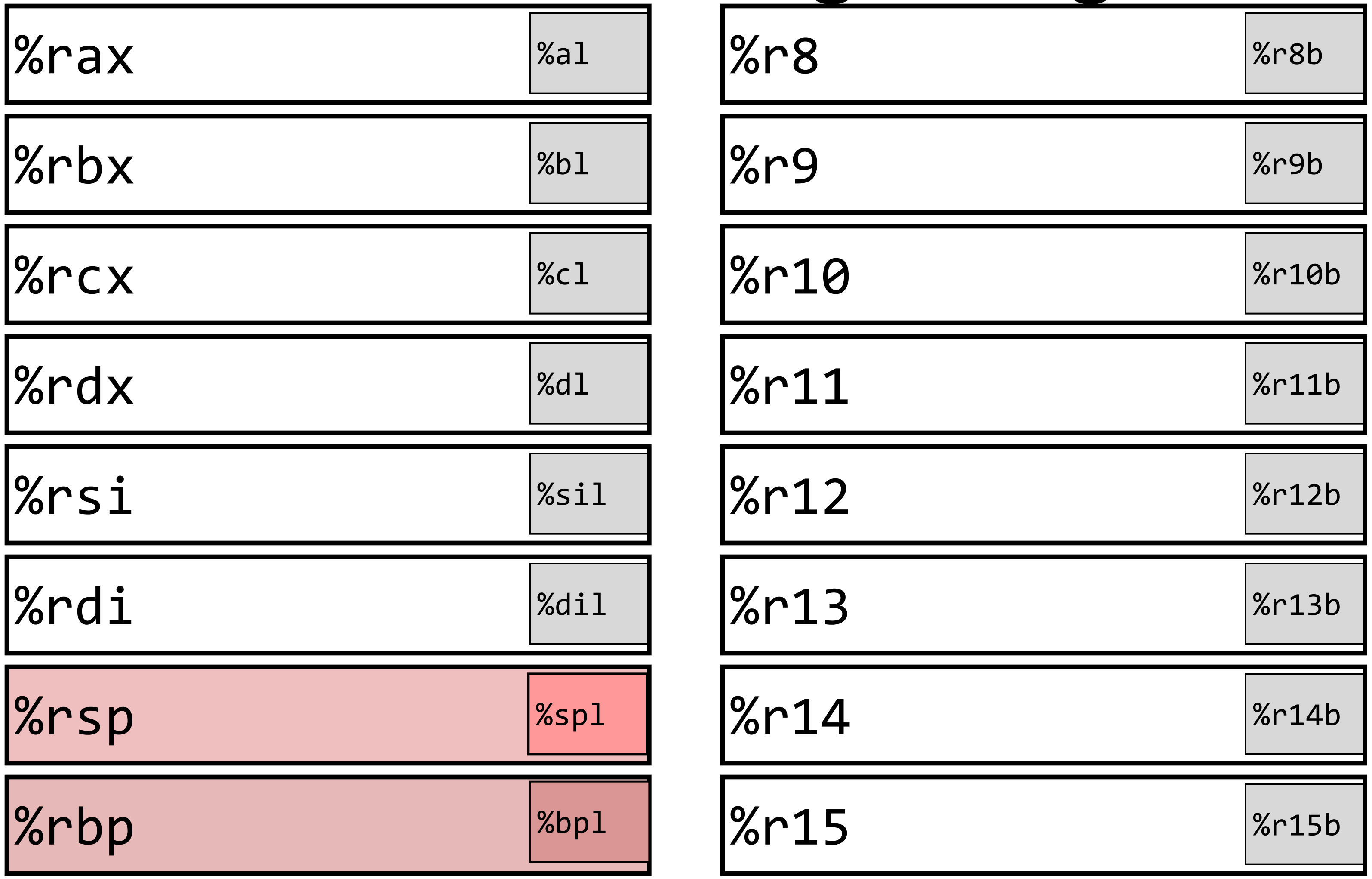
- **`setX`** ***dest***

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**
`%rax`	Return value

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