+

Assembly Basics *Con't*

Simple Memory Addressing Modes

- **Normal** **(R)** **Mem[Reg[R]]**
  - Register R specifies memory address
  - Example:

```
movq (%rcx),%rax
```

- **Displacement** **D(R)** **Mem[D + Reg[R]]**
  - Register R specifies start of memory region
  - Constant displacement D specifies offset in bytes.
  - Example:

```
movq 8(%rbp),%rdx
```

- **Note**: the normal mode is a special case of displacement mode in which D = 0

Complete Memory Addressing Modes

- General form

D(Rb, Ri, S) **Mem[D + Reg[Rb] + Reg[Ri] * S]**

  - D: Constant "displacement"
  - Rb: Base register: Any of 16 integer registers
  - Ri: Index register: Any, except for **%rsp**
  - S: Scale: 1, 2, 4, or 8

Complete Memory Addressing Modes

- General form

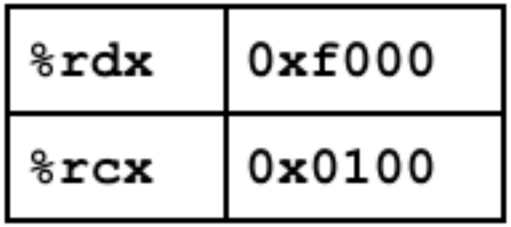
D(Rb, Ri, S) **Mem[D + Reg[Rb] + Reg[Ri] * S]**

  - D: Constant "displacement"
  - Rb: Base register: Any of 16 integer registers
  - Ri: Index register: Any, except for **%rsp**
  - S: Scale: 1, 2, 4, or 8

- Special cases: you can omit certain arguments if not needed.

(Rb,Ri)	**Mem[Reg[Rb] + Reg[Ri]]**
D(Rb,Ri)	**Mem[D + Reg[Rb] + Reg[Ri]]**
(Rb,Ri,S)	**Mem[Reg[Rb] + Reg[Ri] * S]**

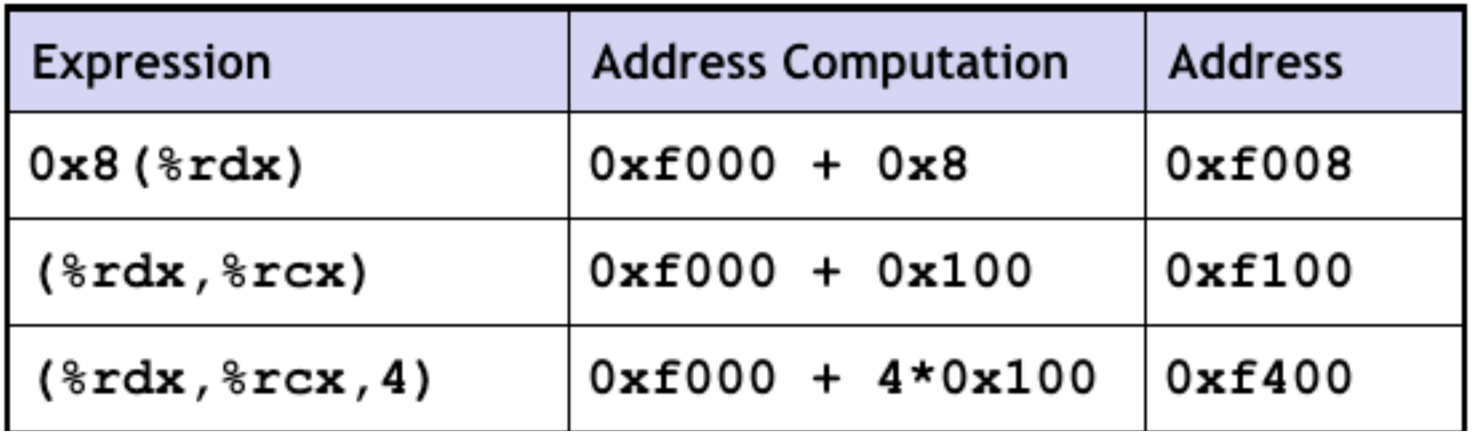
Address Computation Examples



`%rdx`	`0xf000`
`%rcx`	`0x0100`

“Base” register

“Index” register



Expression	Address Computation	Address
`0x8(%rdx)`	`0xf000 + 0x8`	`0xf008`
`(%rdx,%rcx)`	`0xf000 + 0x100`	`0xf100`
`(%rdx,%rcx,4)`	`0xf000 + 4*0x100`	`0xf400`

Address Computation Instruction

- **leaq *src*, *dest***
  - *src* is an address computation expression
  - set *dest* to address denoted by expression

- use case 1
  - Computing addresses without a memory reference
    - E.g., translation of p = &x[i];

- Example

```
char* a2(char* x){
  return &x[2];
}
```

```
leaq 2(%rdi), %rax   # return &x[2]
ret
```

Address Computation Instruction *con't*

- **leaq** ***src*****,** ***dest***
  - *src* is an address computation expression
  - set *dest* to address denoted by expression

- (ab)use case 2
  - Computing arithmetic expressions of the form x + k * y
    - k = 1, 2, 4, or 8

- Example

```
long m12(long x){
  return x * 12;
}
```

```
leaq (%rdi,%rdi, 2), %rax # t = x + x * 2 (3x)
salq $2, %rax             # return t << 2 (4x)
ret
```

Some Arithmetic Operations - Binary

- Two Operand Instructions:

Format	Computation	
addq src, dest	dest = dest + src	
subq src, dest	dest = dest – src	
imulq src, dest	dest = dest * src	
salq src, dest	dest = dest << src	(also called **shlq**)
sarq src, dest	dest = dest >> src	(arithmetic)
shrq src, dest	dest = dest >> src	(logical)
xorg src, dest	dest = dest ^ src	
andq src, dest	dest = dest & src	
orq src, dest	dest = dest | src	

- Watch out for argument order!

- No distinction between signed and unsigned int (except right shift)

+

Some Arithmetic Operations - Unary

- One Operand Instructions:

Format	Computation
incq dest	dest = dest + 1
decq dest	dest = dest – 1
negq dest	dest = –dest
notq dest	dest = ~dest

- See book for more instructions

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    #t1
  addq  %rdx, %rax           #t2
  leaq  (%rsi,%rsi,2), %rdx
  salq  $4, %rdx             #t4
  leaq  4(%rdi,%rdx), %rcx   #t5
  imulq %rcx, %rax           #rval
  ret
```

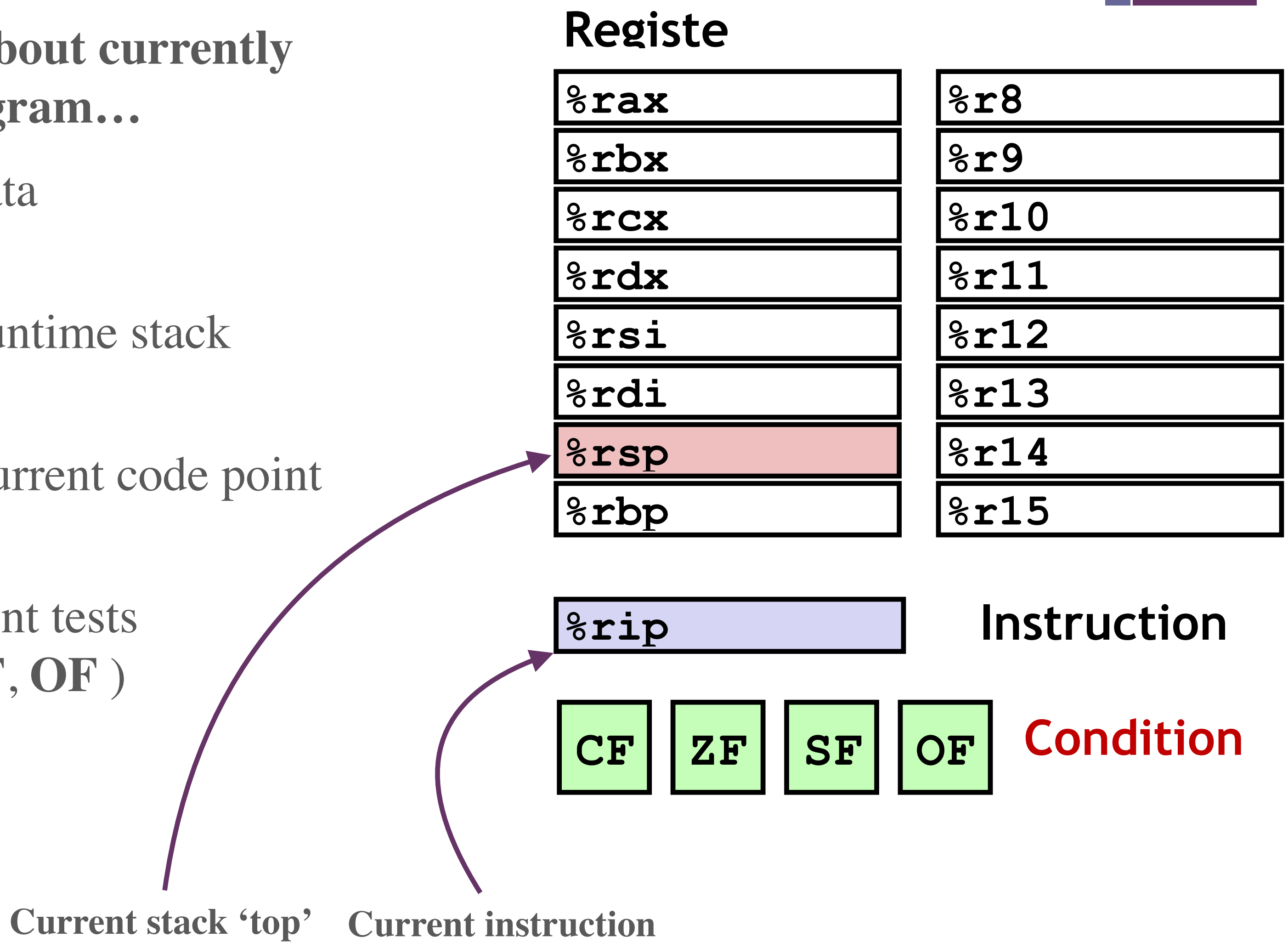
- Noteworthy instructions:
  - **leaq**: “address” computation
  - **salq**: shift
  - **imulq**: integer multiplication

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

Control & Condition Codes

Processor State (x86-64, Partial)

- **Information about currently executing program…**
  - temporary data (**%rax**, …)
  - location of runtime stack (**%rsp**)
  - location of current code point (**%rip**)
  - status of recent tests (**CF**, **ZF**, **SF**, **OF**)



Registe

%rax	%r8
%rbx	%r9
%rcx	%r10
%rdx	%r11
%rsi	%r12
%rdi	%r13
%rsp	%r14
%rbp	%r15

%rip **Instruction**

CF ZF SF OF **Condition**

Current stack ‘top’

Current 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 a side effect) by arithmetic operations**
  - Example: **addq *src*, *dest*** ↔ b = a + b
    - **CF** set if carry out from most significant bit (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 (Explicit Setting)

- **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
      $(a > 0 \;\&\&\; b < 0 \;\&\&\; (a-b) < 0) \;||\; (a < 0 \;\&\&\; b > 0 \;\&\&\; (a-b) > 0)$

- Only purpose of this instruction is to set condition codes!
- There are other instructions like this.

Reading Condition Codes

- **SetX Instructions**
  - Set low-order byte of destination to 0 or 1 based on combinations of condition codes
  - Does not alter remaining 7 bytes

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)

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

- Can reference low-order byte.

Reading Condition Codes *Con't*

- **SetX instructions:**
  - Set single byte based on combination of condition codes
- **One of addressable byte registers**
  - Does not alter remaining bytes
  - Typically use movzbl to finish job
    - 32-bit instructions also set upper 32 bits to 0

```
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 and y
    setg   %al           # Set %al 'on' when x > y
    movzbl %al, %rax     # Copy and zero rest of %rax
    ret
```