Assembly Basics Con’t
Simple Memory Addressing Modes

- **Normal** (R) Mem[ Reg[R] ]
  - Register R specifies memory address
  - Example:
    \[
    \text{movq } (%\text{rcx}), %\text{rax}
    \]

- **Displacement** D(R) Mem[ D + Reg[R] ]
  - Register R specifies start of memory region
  - Constant displacement D specifies offset in bytes.
  - Example:
    \[
    \text{movq 8(\%rbp), \%rdx}
    \]

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

- General form

\[ D(Rb, Ri, S) \quad \text{Mem}[D + \text{Reg}[Rb] + \text{Reg}[Ri] \times S] \]

- **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

\[
D(Rb, Ri, S) \quad \text{Mem}[ \, D + \text{Reg[Rb]} + \text{Reg[Ri]} \times S \, ]
\]

- 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.

\[
(Rb, Ri) \quad \text{Mem}[ \, \text{Reg[Rb]} + \text{Reg[Ri]} \, ]
\]
\[
D(Rb, Ri) \quad \text{Mem}[ \, D + \text{Reg[Rb]} + \text{Reg[Ri]} \, ]
\]
\[
(Rb, Ri, S) \quad \text{Mem}[ \, \text{Reg[Rb]} + \text{Reg[Ri]} \times S \, ]
\]
Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8(%rdx)</td>
<td>0xf000 + 0x8</td>
<td>0xf008</td>
</tr>
<tr>
<td>(%rdx,%rcx)</td>
<td>0xf000 + 0x100</td>
<td>0xf100</td>
</tr>
<tr>
<td>(%rdx,%rcx,4)</td>
<td>0xf000 + 4*0x100</td>
<td>0xf400</td>
</tr>
</tbody>
</table>

“Base” register

“Index” register
Address Computation Instruction

- **leaq** <i>src</i>, <i>dest</i>
  - <i>src</i> is an address computation expression
  - set <i>dest</i> to address denoted by expression

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

- Example

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

```assembly
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 \( x + k \times y \)
    - \( k = 1, 2, 4, \text{ or } 8 \)

- Example

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

```assembly
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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>%rax</td>
<td>t1, t2, rval</td>
</tr>
<tr>
<td>%rdx</td>
<td>t4</td>
</tr>
<tr>
<td>%rcx</td>
<td>t5</td>
</tr>
</tbody>
</table>
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 )

<table>
<thead>
<tr>
<th>Register</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>CF</td>
</tr>
<tr>
<td>%rbx</td>
<td>ZF</td>
</tr>
<tr>
<td>%rcx</td>
<td>SF</td>
</tr>
<tr>
<td>%rdx</td>
<td>OF</td>
</tr>
<tr>
<td>%rsi</td>
<td></td>
</tr>
<tr>
<td>%rdi</td>
<td></td>
</tr>
<tr>
<td>%rsp</td>
<td></td>
</tr>
<tr>
<td>%rbp</td>
<td></td>
</tr>
</tbody>
</table>

Current stack ‘top’  Current instruction

Instruction

Condition
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
      \[(a > 0 \&\& b > 0 \&\& t < 0) \lor (a < 0 \&\& b < 0 \&\& t > 0)\]
  - 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
      $$(a > 0 \&\& b < 0 \&\& (a-b) < 0) \lor (a < 0 \&\& b > 0 \&\& (a-b) > 0)$$

- 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

<table>
<thead>
<tr>
<th>SetX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sete</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>setne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>sets</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>setns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg</td>
<td>~(SF^OF) &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>setge</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>setl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>setle</td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>seta</td>
<td>~CF &amp; ~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>setb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
- 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

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

```c
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
```