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

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

- 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

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

- 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.

\[
\begin{align*}
(Rb,Ri) & \quad \text{Mem}[ \; \text{Reg}[Rb] + \text{Reg}[Ri] \;] \\
D(Rb,Ri) & \quad \text{Mem}[ \; D + \text{Reg}[Rb] + \text{Reg}[Ri] \;] \\
(Rb,Ri,S) & \quad \text{Mem}[ \; \text{Reg}[Rb] + \text{Reg}[Ri] * S \;]
\end{align*}
\]
Address Computation Examples

"Base" register

"Index" register

<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>
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

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

```assembly
leaq 2(%rdi), %rax  # return &x[2]
ret
```
Address Computation Instruction \textit{con’t}

- **leaq** \textit{src, dest}
  - \textit{src} is an address computation expression
  - set \textit{dest} to address denoted by expression

- (ab)use case 2
  - Computing arithmetic expressions of the form $x + k \times y$
    - $k = 1, 2, 4,$ 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

```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;
}
```

- 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 )

Registe

<table>
<thead>
<tr>
<th>%rax</th>
<th>%r8</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%r9</td>
</tr>
<tr>
<td>%rcx</td>
<td>%r10</td>
</tr>
<tr>
<td>%rdx</td>
<td>%r11</td>
</tr>
<tr>
<td>%rsi</td>
<td>%r12</td>
</tr>
<tr>
<td>%rdi</td>
<td>%r13</td>
</tr>
<tr>
<td>%rsp</td>
<td>%r14</td>
</tr>
<tr>
<td>%rbp</td>
<td>%r15</td>
</tr>
</tbody>
</table>

Instruction

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
      \[(a > 0 && b > 0 && t < 0) \lor (a < 0 && b < 0 && t >= 0)\]

- 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
      \[(a > 0 && b < 0 && (a-b) < 0) || (a < 0 && b > 0 && (a-b) > 0)\]
  - 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>
### x86-64 Integer Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%al</td>
</tr>
<tr>
<td>%rbx</td>
<td>%bl</td>
</tr>
<tr>
<td>%rcx</td>
<td>%cl</td>
</tr>
<tr>
<td>%rdx</td>
<td>%dl</td>
</tr>
<tr>
<td>%rsi</td>
<td>%sil</td>
</tr>
<tr>
<td>%rdi</td>
<td>%dil</td>
</tr>
<tr>
<td>%rsp</td>
<td>%spl</td>
</tr>
<tr>
<td>%rbp</td>
<td>%bpl</td>
</tr>
<tr>
<td>%r8</td>
<td>%r8b</td>
</tr>
<tr>
<td>%r9</td>
<td>%r9b</td>
</tr>
<tr>
<td>%r10</td>
<td>%r10b</td>
</tr>
<tr>
<td>%r11</td>
<td>%r11b</td>
</tr>
<tr>
<td>%r12</td>
<td>%r12b</td>
</tr>
<tr>
<td>%r13</td>
<td>%r13b</td>
</tr>
<tr>
<td>%r14</td>
<td>%r14b</td>
</tr>
<tr>
<td>%r15</td>
<td>%r15b</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>

```
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
```
Conditional Branches
Jumping

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

<table>
<thead>
<tr>
<th>jX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>je</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>js</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg</td>
<td>~ (SF^OF) &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge</td>
<td>~ (SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle</td>
<td>(SF^OF)</td>
<td>ZF</td>
</tr>
<tr>
<td>ja</td>
<td>~CF &amp; ~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
### Conditional Branch / Jumping

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

```assembly
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
```

- **Note:** must use `-fno-if-conversion` argument to gcc, otherwise assembly will not use jumps in this program, we’ll see why in a moment.

<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>
Rewriting C with `goto` Statements

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

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

```c
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;
}
```

- Why? "goto" form resembles assembly instructions using jumps
General Conditional Expression Translation

- **C code**

```c
val = test ? then_expr : else_expr;
```

- **Example**

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

- **Goto version**

```c
if (!test) goto Else;
val = then_expr;
goto Done;
Else:
    val = else_Expr;
Done:
    . . .
```

- Create separate code regions for then & else expressions
- Execute appropriate one
- This is how we can think about ‘jumping’
Alternate Approach: 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

```c
val = test
    ? then_expr
    : else_expr;
```

### Goto Version

```c
result = then_expr;
eval = else_expr;
if (!test) result = eval;
return result;
```
Conditional Move Example

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

**Register Use(s)**

<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>

**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 == 0 ? 0 : 5 / p;
  ```

  - 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
Loops
General “Do-While” Translation

C Code

do
    Body
while (Test);

Goto Version

loop:
    Body
    if (Test)
    goto loop

Body

{ 
    statement_1;
    statement_2;
    ...
    statement_n;
}
“Do-While” Loop Example

- Count number of 1’s in argument x (‘popcount’)
- Use conditional branch to either continue looping or to exit loop
```
long pcount_goto(unsigned long x)
{
    long result = 0;
    loop:
        result += x & 0x1;
        x >>= 1;
        if(x) goto loop;
    return result;
}
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rax</td>
<td>result</td>
</tr>
</tbody>
</table>

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

- Note: some processors' branch predictors behave badly when a branch's target or fall-through is a `ret` instruction, and adding the `rep;` prefix avoids this.
General “While” Translation #1

- “Jump-to-middle” translation
- Used with **gcc -Og** (our setting)

While Version:

```
while (Test) Body
```

Goto Version:

```
goto test;
loop:
    Body
test:
    if (Test)
        goto loop;
done:
```
While Loop Example #1

- Compare to do-while version of function
- Initial goto starts loop at test
General “While” Translation #2

- “Do-while” conversion
- Used with gcc –O1

**While Version**

```
while (Test)
  Body
```

**Do-While Version**

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

**Goto Version**

```
if (!Test)
go to done;
loop:
  Body
  if (Test)
    goto loop;
done:
```
While Loop Example #2

C Code

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

Do While

```c
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
```
#define WSIZE 8*sizeof(int)
long pcount_for(unsigned long x)
{
    size_t i;
    long result = 0;
    for (i = 0; i < WSIZE; i++)
    {
        unsigned bit = (x >> i) & 0x1;
        result += bit;
    }
    return result;
}
```
“For” Loop -> “While” Loop

For Version

```
for (Init; Test; Update )
  Body
```

While Version

```
Init;
while (Test) {
  Body
  Update;
}
```
For-While Conversion

```c
long pcount_for_while(unsigned long x)
{
    size_t i;
    long result = 0;
    i = 0;
    while (i < WSIZE)
    {
        unsigned bit = (x >> i) & 0x1;
        result += bit;
        i++;
    }
    return result;
}
```
“For” Loop Do-While Conversion

Initial test may be optimized away if compiler knows its safe