Discussion - Week 2

TA : Mathanky
Email : mathanky04@ucla.edu
Office Hours : Thursday, 11.30PM - 1.30PM, Boelter Hall 3256F
REVIEW OF WEEK 2

- **Machine Level Programming - Basics**
  - Registers and Memory
  - Arithmetic and Logical Operations
  - Assembly Code

- **Machine Level Programming - Control**
  - Conditional Codes
  - Branches
  - Loops
  - Switch Statements
Machine Level Programming - Basics

Assembly/Machine Code View

CPU
- Registers
- Condition Codes
- PC

Memory
- Code
- Data
- Stack

Programmer-Visible State
- PC: Program counter
  - Address of next instruction
  - Called “RIP” (x86-64)
- Register file
  - Heavily used program data
- Condition codes
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching
- Memory
  - Byte addressable array
  - Code and user data
  - Stack to support procedures
### movq Operand Combinations

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src,Dest</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Imm</strong></td>
<td>Reg</td>
<td>movq $0x4,%rax</td>
<td>temp = 0x4;</td>
</tr>
<tr>
<td></td>
<td>Mem</td>
<td>movq $-147,(%rax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td><strong>Reg</strong></td>
<td>Reg</td>
<td>movq %rax,%rdx</td>
<td>temp2 = temp1;</td>
</tr>
<tr>
<td></td>
<td>Mem</td>
<td>movq %rax,(%rdx)</td>
<td>*p = temp;</td>
</tr>
<tr>
<td><strong>Mem</strong></td>
<td>Reg</td>
<td>movq (%rax),%rdx</td>
<td>temp = *p;</td>
</tr>
</tbody>
</table>
Simple Memory Addressing Modes

- **Normal** \( R \) \( \text{Mem[Reg[R]]} \)
  - Register \( R \) specifies memory address
  - Aha! Pointer dereferencing in C
  
  \texttt{movq (\%rcx),\%rax}

- **Displacement** \( D(R) \) \( \text{Mem[Reg[R]+D]} \)
  - Register \( R \) specifies start of memory region
  - Constant displacement \( D \) specifies offset

  \texttt{movq 8(\%rbp),\%rdx}
Complete Memory Addressing Modes

■ Most General Form

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

- **D**: Constant “displacement” 1, 2, or 4 bytes
- **Rb**: Base register: Any of 16 integer registers
- **Ri**: Index register: Any, except for %rsp
- **S**: Scale: 1, 2, 4, or 8 (why these numbers?)

■ Special Cases

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

- **lea $\text{Src}$, $\text{Dst}$**
  - $\text{Src}$ is address mode expression
  - Set $\text{Dst}$ to address denoted by expression

- **Example**

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

Converted to ASM by compiler:

```
leaq (%rdi,%rdi,2), %rax  # t = x+2*x
salq $2, %rax             # return t<<2
```
### mov vs lea

- `movl (%rdx), %rax`
- `leal (%rdx), %rax`

`movl` takes the **contents** of what’s stored in register `%rdx` and moves it to `%rax`. 

`leal` computes the load effective **address** and stores it in `%rax`. `leal` analogous to returning a pointer, whereas `movl` is analogous to returning a dereferenced pointer.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source, Destination</th>
<th>Destination Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq</td>
<td>Src, Dest</td>
<td>Dest = Dest + Src</td>
</tr>
<tr>
<td>subq</td>
<td>Src, Dest</td>
<td>Dest = Dest − Src</td>
</tr>
<tr>
<td>imulq</td>
<td>Src, Dest</td>
<td>Dest = Dest * Src</td>
</tr>
<tr>
<td>salq</td>
<td>Src, Dest</td>
<td>Dest = Dest &lt;&lt; Src</td>
</tr>
<tr>
<td>sarq</td>
<td>Src, Dest</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>shrq</td>
<td>Src, Dest</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>xorq</td>
<td>Src, Dest</td>
<td>Dest = Dest ^ Src</td>
</tr>
<tr>
<td>andq</td>
<td>Src, Dest</td>
<td>Dest = Dest &amp; Src</td>
</tr>
<tr>
<td>orq</td>
<td>Src, Dest</td>
<td>Dest = Dest</td>
</tr>
</tbody>
</table>
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;
}
```

```assembly
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
```

Interesting Instructions
- `leaq`: address computation
- `salq`: shift
- `imulq`: multiplication
  - But, only used once
Machine Level Programming - Control

Condition Codes (Implicit Setting)

- Single bit registers
  - CF  : Carry Flag (for unsigned)
  - SF  : Sign Flag (for signed)
  - ZF  : Zero Flag
  - OF  : Overflow Flag (for signed)

- Implicitly set (as side effect) of arithmetic operations
  Example: \texttt{addq \textit{Src},Dest} \leftrightarrow \texttt{t = a+b}

  - CF set if carry/borrow out from most significant bit (unsigned overflow)
  - ZF set if \texttt{t} == 0
  - SF set if \texttt{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 \texttt{leaq} instruction
ZF set when

\[ \begin{array}{c}
000000000000 \ldots 00000000000
\end{array} \]

SF set when

\[ \begin{array}{c}
y\ldots
\hline
y\ldots
\hline
1\ldots
\end{array} \]
CF set when

For unsigned arithmetic, this reports overflow
OF set when

\[ z = \sim y \]

\[(a > 0 \land b > 0 \land t < 0) \quad \text{||} \quad (a < 0 \land b < 0 \land t \geq 0)\]

For signed arithmetic, this reports overflow
Condition Codes (Explicit Setting: Compare)

- Explicit Setting by Compare Instruction
  - `cmpq Src2, Src1`
  - `cmpq b, a` like computing `a - b` without setting destination

- **CF set** if carry/borrow 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)`
Condition Codes (Explicit Reading: Set)

- **Explicit Reading by Set Instructions**
  - `setX Dest`: Set low-order byte of destination `Dest` to 0 or 1 based on combinations of condition codes
  - Does not alter remaining 7 bytes of `Dest`

<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>
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;
}
```

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

<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>
# Jumping

- **jX Instructions**
  - Jump to different part of code depending on condition codes
  - Implicit reading of 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>Less or Equal (signed)</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>
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

<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>
Conditional Move Example

```c
long absdiff
 (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    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>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

```assembly
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
    cmovule %rdx, %rax  # if <=, result = eval
    ret
```
Bad Cases for Conditional Move

Expensive Computations

```java
val = Test(x) ? Hard1(x) : Hard2(x);
```

- Both values get computed
- Only makes sense when computations are very simple

Bad Performance

Risky Computations

```java
val = p ? *p : 0;
```

- Both values get computed
- May have undesirable effects

Unsafe

Computations with side effects

```java
val = x > 0 ? x*=7 : x+=3;
```

- Both values get computed
- Must be side-effect free

Illegal
“Do-While” Loop Compilation

```c
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, %eax    # result = 0
.L2:      # loop:
  movq    %rdi, %rdx
  andl    $1, %edx    # t = x & 0x1
  addq    %rdx, %rax  # result += t
  shrq    %rdi        # x >>= 1
  jne     .L2         # if(x) goto loop
  rep; ret
```
General “While” Translation #2

While version

while (Test)
  Body

Do-While Version

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

Goto Version

if (!Test)
  goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
For-While Conversion

Init
\[ i = 0 \]

Test
\[ i < \text{WSIZE} \]

Update
\[ i++ \]

Body
\[
\{ \\
\text{unsigned bit} = \ (x >> i) \ & \ 0x1; \\
\text{result} += \text{bit}; \\
\}
\]

long pcount_for_while
\( (\text{unsigned long } x) \)
\[
\{ \\
\text{size_t } i; \\
\text{long result} = 0; \\
i = 0; \\
\text{while} \ (i < \text{WSIZE}) \\
\{ \\
\text{unsigned bit} = \ (x >> i) \ & \ 0x1; \\
\text{result} += \text{bit}; \\
i++; \\
\}
\}
\text{return result;} \]
Which of these C-functions converts to the above assembly code?