CSCI-UA.0201

Computer Systems Organization

Machine-Level Programming II

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

Some slides adapted (and slightly modified) from:
• Clark Barrett
• Jinyang Li
• Randy Bryant
• Dave O’Hallaron
Arithmetic & Logic Operations
Some Arithmetic Operations

- Two Operand Instructions, examples:

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq</td>
<td>Dest = Dest + Src</td>
</tr>
<tr>
<td>subq</td>
<td>Dest = Dest - Src</td>
</tr>
<tr>
<td>imulq</td>
<td>Dest = Dest * Src</td>
</tr>
<tr>
<td>salq</td>
<td>Dest = Dest &lt;&lt; Src</td>
</tr>
<tr>
<td>sarq</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>shrq</td>
<td>Dest = Dest &gt;&gt;&gt; Src</td>
</tr>
<tr>
<td>xorq</td>
<td>Dest = Dest ^ Src</td>
</tr>
<tr>
<td>andq</td>
<td>Dest = Dest &amp; Src</td>
</tr>
<tr>
<td>orq</td>
<td>Dest = Dest</td>
</tr>
</tbody>
</table>

- Watch out for argument order!
- No distinction between signed and unsigned int (why?)
Some Arithmetic Operations

• **One Operand Instructions**
  - \texttt{incq Dest} \quad \text{Dest} = \text{Dest} + 1
  - \texttt{decq Dest} \quad \text{Dest} = \text{Dest} - 1
  - \texttt{negq Dest} \quad \text{Dest} = - \text{Dest}
  - \texttt{notq Dest} \quad \text{Dest} = \sim \text{Dest}

• See book for more instructions
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;
}
Understanding 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` # t1
- `addq %rdx, %rax` # t2
- `leaq (%rsi,%rsi,2), %rdx` # t3
- `salq $4, %rdx` # t4
- `leaq 4(%rdi,%rdx), %rcx` # t5
- `imulq %rcx, %rax` # rval
- `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>%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>
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  # t1
addq %rdx, %rax         # t2
leaq (%rsi,%rsi,2), %rdx
salq $4, %rdx           # t4
leaq 4(%rdi,%rdx), %rcx # t5
imulq %rcx, %rax        # rval

- Instructions in different order from C code
- Some expressions require multiple instructions
- Some instructions cover multiple expressions
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 Divisio

cqto

- No operands
- Takes the sign bit from rax and replicates it in rdx
Control
Processor State (x86-64, Partial)

- Information about currently executing program
  - Temporary data (%rax, ...)
  - Location of runtime stack (%rsp)
  - Location of current code control point (%rip, ...)
  - Status of recent tests (CF, ZF, SF, OF)

Registers

<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>%r8p</td>
<td>%r14</td>
</tr>
<tr>
<td>%rbp</td>
<td>%r15</td>
</tr>
</tbody>
</table>

Instruction pointer

%rip

Condition codes

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
  - \((a>0 \land \land b>0 \land t<0) \lor \lor (a<0 \land \land b<0 \land t>=0)\)

- 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

```c
compl b,a set condition codes based a-b (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

```null
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 some register based on combinations of condition codes and does not alter remaining 7 bytes. Destination can also be memory location.

<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>ZF</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>

This set of instructions is usually used after a comparison.
Example

```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>

```assembly
cmpq  %rsi, %rdi     # Compare x:y
setg  %al            # Set when >
movzbq %al, %rax     # Zero rest of %rax
ret
```
### x86-64 Integer Registers

<table>
<thead>
<tr>
<th>%rax</th>
<th>%al</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%r8</th>
<th>%r8b</th>
</tr>
</thead>
<tbody>
<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
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.
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>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>
Indirect jump

jmp * Operand

Unconditional jump

Can be:

• register
• Memory address using any of the addressing modes we saw.
Example

```c
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>
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.
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;
nt = !Test;
if (nt) result = eval;
return result;
```
Conditional Move Example

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

Register | Use(s)
---|---
%rdi | Argument x
%rsi | Argument y
%rax | Return value

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

\[
\text{val} = \text{Test}(x) \ ? \ \text{Hard1}(x) : \text{Hard2}(x);
\]

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

Risky Computations

\[
\text{val} = p \ ? \ *p : 0;
\]

- Both values get computed
- May have undesirable effects

Computations with side effects

\[
\text{val} = x > 0 \ ? \ x*=7 : x+=3;
\]

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

• The arithmetic and logic operations can be applied to data of size 1(b), 2(w), 4(l), and 8(q) bytes.

• Condition codes are needed in order to implement conditional branches/jumps.

• The compiler uses the different condition codes and different jump formats to implement the different control structures we have in high-level languages: for-loop, do-while, while, switch, if-then-else