Machine-Level Programming II: Arithmetic & Control

Andrew Case

Adapted from Jinyang Li, Randy Bryant and Dave O’Hallaron
Today

- address computation and lea
- Arithmetic operations
- Control: Condition codes
- Conditional branches
- While loops
Memory Addressing Modes

- **Most General Form**
  \[ D(\text{Rb}, \text{Ri}, S) \]
  Memory Address = \[ \text{Reg}[\text{Rb}] + S \times \text{Reg}[\text{Ri}] + D \]

- **Special Cases**
  - \((\text{Rb}, \text{Ri})\) \hspace{1cm} Mem addr = Reg[\text{Rb}] + Reg[\text{Ri}]
  - \(D(\text{Rb}, \text{Ri})\) \hspace{1cm} Mem addr = Reg[\text{Rb}] + Reg[\text{Ri}] + D
  - \((\text{Rb}, \text{Ri}, S)\) \hspace{1cm} Mem addr = Reg[\text{Rb}] + S \times \text{Reg}[\text{Ri}]
# Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8(%edx)</td>
<td>0xf000 + 0x8</td>
<td>0xf008</td>
</tr>
<tr>
<td>(%edx,%ecx)</td>
<td>0xf000 + 0x100</td>
<td>0xf100</td>
</tr>
<tr>
<td>(%edx,%ecx,4)</td>
<td>0xf000 + 4*0x100</td>
<td>0xf400</td>
</tr>
<tr>
<td>0x80(%edx,2)</td>
<td>2*0xf000 + 0x80</td>
<td>0x1e080</td>
</tr>
</tbody>
</table>
lea instruction -
(Load Effective Address)

**lea Src, Dest**
- Src is address mode expression
- Set Dest to address denoted by expression (allows computation)

**common uses**
- Does register arithmetic
- Computing addresses (without a direct memory reference)
  - Array items addresses: E.g., translation of $p = \&x[i]$;
- Computing expressions like: $x + k*y$; for $k = 1, 2, 4,$ or 8
- Example

```c
int foo(int x)
{
  return x*12;
}
```
```
leal (%eax,%eax,2), %eax ;t <- x+x*2
sall $2, %eax ;return t<<2
```
Today

- address computation and leal
- Arithmetic operations
- Control: Condition codes
- Conditional branches
- While loops
Arithmetic Operations

addl    Src, Dest ;   Dest = Dest + Src
subl    Src, Dest ;   Dest = Dest − Src
imull   Src, Dest ;   Dest = Dest * Src
incl    Dest ;       Dest = Dest + 1
decl    Dest ;       Dest = Dest − 1
negl    Dest ;       Dest = − Dest
notl    Dest ;       Dest = ~Dest

sall    Src, Dest ;   Dest = Dest << Src   (shift arithmetic left)
sarl    Src, Dest ;   Dest = Dest >> Src   (shift arithmetic right)
xorl    Src, Dest ;   Dest = Dest ^ Src
andl    Src, Dest ;   Dest = Dest & Src
orl     Src, Dest ;   Dest = Dest | Src
notl    Dest ;       Dest = ~Dest
Arithmetic Expression Example

```c
int foo(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

```assembly
foo:
pushl %ebp
movl %esp, %ebp

movl 8(%ebp), %ecx
movl 12(%ebp), %edx
leal (%edx,%edx,2), %eax
sall $4, %eax
leal 4(%ecx,%eax), %eax
addl %ecx, %edx
addl 16(%ebp), %edx
imull %edx, %eax

popl %ebp
ret
```

Set Up

Body

Finish
Understanding arithmetic

```c
int foo(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

```assembly
movl 8(%ebp), %ecx      # ecx = x
movl 12(%ebp), %edx     # edx = y
leal (%edx,%edx,2), %eax # eax = y*3
sall $4, %eax            # eax *= 16 (t4)
leal 4(%ecx,%eax), %eax # eax = t4 +x+4 (t5)
addl %ecx, %edx          # edx = x+y (t1)
addl 16(%ebp), %edx      # edx += z (t2)
imull %edx, %eax         # eax = t2 * t5 (rval)
```
Observations

Instructions in different order from C code

Some expressions require multiple instructions

Some instructions cover multiple expressions

Get exact same code when compile:

\[(x+y+z) \times (x+4+48 \times y)\]

```c
int foo(int x, int y, int z) {
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 \times t5;
    return rval;
}
```

```assembly
movl 8(%ebp), %ecx
    # ecx = x
movl 12(%ebp), %edx
    # edx = y
leal (%edx,%edx,2), %eax
    # eax = y*3
sall $4, %eax
    # eax *= 16 (t4)
leal 4(%ecx,%eax), %eax
    # eax = t4 +x+4 (t5)
addl %ecx, %edx
    # edx = x+y (t1)
addl 16(%ebp), %edx
    # edx += z (t2)
imull %edx, %eax
    # eax = t2 \times t5 (rval)
```
int bar(int x, int y) {
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}

movl 12(%ebp),%eax  # eax = y
xorl 8(%ebp),%eax   # eax = x^y (t1)
sarl $17,%eax       # eax = t1>>17 (t2)
andl $8185,%eax     # eax = t2 & mask (rval)
Today

- Address computation and lea
- Arithmetic operations
- **Control: Condition codes**
- Conditional branches
- Loops
Today

- Address computation and lea
- Arithmetic operations
- Control: Condition codes
- Conditional branches
- Loops
Processor State (IA32)

- Processor state = information kept in CPU on currently executing program

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax, %ecx, %edx, %ebx, %esi, %edi</td>
<td>General purpose registers</td>
</tr>
<tr>
<td>%esp, %ebp</td>
<td>Stack top, Stack frame</td>
</tr>
<tr>
<td>%eip</td>
<td>Instruction pointer</td>
</tr>
</tbody>
</table>

Condition codes: CF, ZF, SF, OF
Setting Condition Codes

- Can be implicitly set by arithmetic operations

Example: addl 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, i.e. carry out from 30-th bit \( (a > 0 \&\& b > 0 \&\& t < 0) \) || \( (a < 0 \&\& b < 0 \&\& t \geq 0) \)

- **Not set by lea instruction**
Setting Condition Codes

- **cmp/test instructions (Compare/Test)**
  - Used to set conditional codes explicitly
    - `cmpl b,a` set condition codes based on `(a-b)`
      - **CF** is set if `(a-b)` results in unsigned overflow
      - **ZF** is set if `a == b`
      - **SF** is set if `(a-b) < 0` (as signed)
      - **OF** is set if `(a-b)` results in signed overflow
    - `testl b,a` set condition codes based on value of `a & b`
### SetX instruction: Set the lower byte of some register based on combinations of condition codes

<table>
<thead>
<tr>
<th>Instruction (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>settle</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>
Reading Condition Codes

Example

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

```assembly
movl 12(%ebp),%eax  # eax = y
cmpl %eax,8(%ebp)  # Compare x : y
setg %al          # al = x > y
movzbl %al,%eax   # Zero rest of %eax
```

Move Zero-Extended Byte to Long
Reading Condition Codes: x86-64

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

cmpq %rsi, %rdi
setg %al
movzbl %al, %eax

Is %rax zero?
Yes: 32-bit instructions set high order 32 bits to 0!
Today

- Address computation and lea
- Arithmetic operations
- Control: Condition codes
- Conditional branches & Moves
- Loops
Jumping

- **jX Instruction: Jump to different part of code depending on condition codes**

<table>
<thead>
<tr>
<th>Instruction (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 Example

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

```assembly
absdiff:
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %edx
    movl 12(%ebp), %eax
    cmpl %eax, %edx
    jle .L6
    subl %eax, %edx
    movl %edx, %eax
    jmp .L7
.L6:
    subl %edx, %eax
.L7:
    popl %ebp
    ret
```

**Setup**
- Push the base pointer (%ebp) to the stack.
- Move the current stack pointer (%esp) to the base pointer (%ebp).
- Move the value at 8(%ebp) into %edx.
- Move the value at 12(%ebp) into %eax.
- Compare %eax with %edx.
  - If %eax is less than or equal to %edx, jump to .L6.
- Subtract %eax from %edx and move the result to %eax.

**Body1**
- Move %edx into %eax.

**Body2a**
- Subtract %edx from %eax.

**Body2b**
- Load the base pointer (%ebp) from the stack.
- Return from the function.
**Conditional Branch Example**

```c
int goto_ad(int x, int y)
{
    int result;
    if (x <= y) goto Else;
    result = x-y;
    goto Exit;
Else:
    result = y-x;
Exit:
    return result;
}
```

```assembly
absdiff:
    pushl  %ebp
    movl   %esp, %ebp
    movl   8(%ebp), %edx
    movl   12(%ebp), %eax
    cmpl   %eax, %edx
    jle    .L6
    subl   %eax, %edx
    movl   %edx, %eax
    jmp .L7
.L6:
    subl  %edx, %eax
.L7:
    popl  %ebp
    ret
```
int goto_ad(int x, int y) {
    int result;
    if (x <= y) goto Else;
    result = x-y;
    goto Exit;
Else:
    result = y-x;
Exit:
    return result;
}

absdiff:
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %edx
    movl 12(%ebp), %eax
    cmpl %eax, %edx
    jle .L6
    subl %eax, %edx
    jmp .L7
.L6:
    subl %edx, %eax
.L7:
    popl %ebp
    ret
Conditional Branch Example

(Cont.)

```c
int goto_ad(int x, int y)
{
    int result;
    if (x <= y) goto Else;
    result = x - y;
    goto Exit;
Else:
    result = y - x;
Exit:
    return result;
}
```

```assembly
absdiff:
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %edx
    movl 12(%ebp), %eax
    cmpl %eax, %edx
    jle .L6
    subl %eax, %edx
    movl %edx, %eax
    jmp .L7
.L6:
    subl %edx, %eax
.L7:
    popl %ebp
    ret
```
int goto_ad(int x, int y) {
    int result;
    if (x <= y) goto Else;
    result = x - y;
    goto Exit;
  Else:
    result = y - x;
  Exit:
    return result;
}
Today

- Complete addressing mode, address computation (lea)
- Arithmetic operations
- Control: Condition codes
- Conditional branches and moves
- Loops
“Do-While” Loop Example

C Code

```c
int pcount_do(unsigned x)
{
    int result = 0;
    do {
        result += x & 0x1;
        x >>= 1;
    } while (x);
    return result;
}
```

Goto Version

```c
int pcount_do(unsigned x)
{
    int result = 0;
    loop:
        result += x & 0x1;
        x >>= 1;
        if (x)
            goto loop;
    return result;
}
```

- Count number of 1’s in argument x
“Do-While” Loop Compilation

Goto Version

```c
int pcount_do(unsigned x) {
    int result = 0;
    loop:
        result += x & 0x1;
        x >>= 1;
        if (x)
            goto loop;
    return result;
}
```

<table>
<thead>
<tr>
<th>Registers:</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
</tr>
<tr>
<td>%ecx</td>
</tr>
</tbody>
</table>

- `movl $0, %ecx` # result = 0
- `movl %edx, %eax` # loop:
- `andl $1, %eax` # t = x & 1
- `addl %eax, %ecx` # result += t
- `shrl %edx` # x >>= 1
- `jne .L2` # If !0, goto loop
General “Do-While” and “While” Translation

```
do
    Body
while (Test);
```

```
loop:
    Body
    if (Test)
        goto loop
```

```
while (Test)
    Body
```

```
if (!Test)
    goto done;
loop:
    Body
    if (Test)
        goto loop;
done:
```
“For” Loop Translation

for (Init; Test; Update)  
  Body

Init;
if (!Test)  
goto done;
loop:  
  Body
  Update
  if (Test)  
goto loop;
done:
C Code

```c
#define WSIZE 8*sizeof(int)
int pcount_for(unsigned x) {
    int i;
    int result = 0;
    for (i = 0; i < WSIZE; i++) {
        unsigned mask = 1 << i;
        result += (x & mask) != 0;
    }
    return result;
}
```

Goto Version

```c
int pcount_for_gt(unsigned x) {
    int i;
    int result = 0;
    i = 0;
    if (!(i < WSIZE)) goto done;
    loop:
    {
        unsigned mask = 1 << i;
        result += (x & mask) != 0;
    }
    i++;
    if (i < WSIZE) goto loop;
    done:
    return result;
}
```
Summary

Today
- Complete addressing mode, address computation (lea)
- Arithmetic operations
- Control: Condition codes
- Conditional branches & conditional moves
- Loops

Next Time
- Switch statements
- Stack
- Call / return
- Procedure call discipline
Using Conditional Moves

- **Conditional Move Instructions**
  - Instruction supports:
    - if (Test) Dest  Src
  - Supported in post-1995 x86 processors
  - GCC does not always use them
  - Wants to preserve compatibility with ancient processors
  - Enabled for x86-64
  - Use switch –march=686 for IA32

- **Why?**
  - Branches are very disruptive to instruction flow through pipelines
  - Conditional move do not require control transfer

---

**C Code**

```c
val = Test
? Then_Expr
: Else_Expr;
```

**Goto Version**

```c
tval = Then_Expr;
result = Else_Expr;
t = Test;
if (t) result = tval;
return result;
```
Conditional Move Example: x86-64

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

x in %edi, y in %esi

absdiff:
```
movl %edi, %edx
subl %esi, %edx  # tval = x-y
movl %esi, %eax
subl %edi, %eax  # result = y-x
cmpl %esi, %edi  # Compare x:y
cmovg %edx, %eax  # If >, result = tval
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