X86 Instructions: arithmetic & control

Lecture 6

Instructor:
Alvin R. Lebeck

Some Slides based on those from
Randy Bryant and Dave O’Hallaron

Admin

- Homework #2 Due today 11:59pm
- Daphne move office hours to 6-8 pm tonight only
- Reading: Chapter 3

Machine Programming (x86)

- Assembly Basics: Registers, operands, move
- Brief intro to x86-64
- Arithmetic
- Control
Basic ISA Classes

Accumulator:
- 1 address: add A, $acc \leftarrow acc + \text{mem}[A]$
- 1+X address: addx A, $acc \leftarrow acc + \text{mem}[A + X]$

Stack:
- 0 address: add, $\text{tos} \leftarrow \text{tos} + \text{next}$ (JAVA VM)

General Purpose Register:
- 2 address: add A B, $B \leftarrow B + A$ (Intel x86)
- 3 address: add A B C, $C \leftarrow B + A$

Load/Store: (MIPS, ARM)
- 3 address: add Ra Rb Rc, $Ra \leftarrow Rb + Rc$
- load Ra Rb, $Ra \leftarrow \text{mem}[Rb]$
- store Ra Rb, $\text{mem}[Rb] \leftarrow Ra$

Adding Registers to an ISA

- A place to hold values that can be named within the instruction
- Like memory, but much smaller
  - 8-64 locations
- Intel operands in either memory or register
  - add A B
Stored Program Computer

- **Instructions**: a fixed set of built-in operations

- Instructions and data are stored in the (same) computer memory

- Fetch-Execute Cycle
  
  ```java
  while (!done)
  {
    fetch instruction
    execute instruction
  }
  ```

- This is usually done by the hardware for speed
- This is what the Java Virtual Machine does

Integer Registers (IA32)

<table>
<thead>
<tr>
<th>%eax</th>
<th>%ax</th>
<th>%ah</th>
<th>%al</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>%cx</td>
<td>%ch</td>
<td>%cl</td>
</tr>
<tr>
<td>%edx</td>
<td>%dx</td>
<td>%dh</td>
<td>%dl</td>
</tr>
<tr>
<td>%ebx</td>
<td>%bx</td>
<td>%bh</td>
<td>%bl</td>
</tr>
<tr>
<td>%esi</td>
<td>%si</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td>%di</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td>%sp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>%bp</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Origin (mostly obsolete)

- accumulate
- counter
- data
- base
- source
- index
- destination
- index
- stack
- pointer
- base
- pointer

16-bit virtual registers (backwards compatibility)
movl Operand Combinations

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src,Dest</th>
<th>C Analogue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imm</td>
<td>Reg</td>
<td>movl $0x4,%eax</td>
<td>temp = 0x4;</td>
</tr>
<tr>
<td>Mem</td>
<td>Movl $-147,(%eax)</td>
<td>*p = -147;</td>
<td></td>
</tr>
<tr>
<td>Reg</td>
<td>Reg</td>
<td>movl %eax,%edx</td>
<td>temp2 = temp1;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl (%eax),%edx</td>
<td>temp = *p;</td>
</tr>
</tbody>
</table>

Cannot do memory-memory transfer with a single instruction

Complete Memory Addressing Modes

- Most General Form
  \[ D(Rb,Ri,S) \] \[ Mem[Reg[Rb]+S*Reg[Ri]+D] \]
  - D: Constant "displacement" 1, 2, or 4 bytes
  - Rb: Base register: Any of 8 integer registers
  - Ri: Index register: Any, except for \%esp
    - Unlikely you’d use \%ebp, either
  - S: Scale: 1, 2, 4, or 8 (why these numbers?)

- Special Cases
  \( (Rb,Ri) \) \[ Mem[Reg[Rb]+Reg[Ri]] \]
  \( D(Rb,Ri) \) \[ Mem[Reg[Rb]+Reg[Ri]+D] \]
  \( (Rb,Ri,S) \) \[ Mem[Reg[Rb]+S*Reg[Ri]] \]
Some Arithmetic and Logical Operations

- **Two Operand Instructions:**
  - **Format**
  - **Computation**
  - `addl` `Src,Dest` `Dest = Dest + Src`
  - `subl` `Src,Dest` `Dest = Dest - Src`
  - `imull` `Src,Dest` `Dest = Dest * Src`
  - `sall` `Src,Dest` `Dest = Dest << Src`  
    Also called `shll`
  - `sarl` `Src,Dest` `Dest = Dest >> Src`  
    **Arithmetic**
  - `shrl` `Src,Dest` `Dest = Dest >> Src`  
    **Logical**
  - `xorl` `Src,Dest` `Dest = Dest ^ Src`
  - `andl` `Src,Dest` `Dest = Dest & Src`
  - `orl` `Src,Dest` `Dest = Dest | Src`

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

Some Arithmetic Operations

- **One Operand Instructions**
  - `incl` `Dest` `Dest = Dest + 1`
  - `decl` `Dest` `Dest = Dest - 1`
  - `negl` `Dest` `Dest = - Dest`
  - `notl` `Dest` `Dest = ~Dest`

- See book for more instructions
- Note: suffix “l” is for 32-bit values, “b” for byte, “w” for 16-bit
Address Computation Instruction

- **leal Src, Dest**
  - Src is address mode expression
  - Set Dest to address denoted by expression

- **Uses**
  - Computing addresses without a memory reference
    - E.g., translation of `p = &x[i];`
  - Computing arithmetic expressions of the form `x + k*y`
    - `k = 1, 2, 4, or 8`

- **Example**

```c
int mul12(int x) {
    return x*12;
}
```

**Converted to ASM by compiler:**

```asm
leal (%eax,%eax,2), %eax  ; t <- x+x*2
sall $2, %eax             ; return t<<2
```

Simple Addition

- **C statement**
  - x86 instruction(s)

```c
int x = 10;
int y = 2;
x += y;
```

```asm
movl $10, %eax
movl $2, %edx
addl %edx, %eax
```

- **If x and y somewhere in memory**

```c
x += y;
```

```asm
movl x, %eax  ; eax = M[x]
leal y, %edx  ; edx = &M[y]
addl (%edx), %eax
```

- **Cannot have both operands in memory**
Successor Instruction

```
main()
{
  int x, y, same; //
  x = 43;        // movl $43, %ecx
  y = 2;         // movl $2, %edx
  same = 0;      // movl $0, %eax
  if (x == y)
    same = 1;    // execute only if x == y
                   // movl $1, %eax
}
```

The Program Counter (PC)

- Special register (PC) that points to instructions
- Intel calls it the IP
- Contains memory address (like a pointer)
- Instruction fetch is
  - inst = Mem[pc]
- To fetch next sequential instruction PC = PC + ?
  - Size of instruction?
The Program Counter

x = 43; // movl $43, %ecx
y = 2; // movl $2, %edx
same = 0; // movl $0, %eax
if (x == y)
same = 1; // movl $1, %eax  execute if x == y

PC is always automatically incremented to next instruction

Note: not actual addresses for x86, illustrative purposes only

Memory

<table>
<thead>
<tr>
<th>PC</th>
<th>0x10000</th>
<th>0x10004</th>
<th>0x10008</th>
<th>0x1000c</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>movl $43, %ecx</td>
<td>movl $2, %edx</td>
<td>movl $0, %eax</td>
<td>movl $1, %eax</td>
</tr>
</tbody>
</table>

Clearly, this is not correct
We cannot always execute both 0x10008 and 0x1000c

Processor State (IA32, Partial)

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

General purpose registers

%eax
%ecx
%edx
%ebx
%esi
%edi
%esp
%ebp

Current stack top
Current stack frame
Instruction pointer
Condition codes
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 (think of it as side effect) by arithmetic operations
  - Example: addl Src, Dest  \( t = a+b \)
  - 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
    \[ (a>0 \&\& b>0 \&\& t<0) \text{ || } (a<0 \&\& b<0 \&\& t>=0) \]

- Not set by lea instruction
- Full documentation (IA32), link on course website

Condition Codes (Explicit Setting: Compare)

- Explicit Setting by Compare Instruction
  - cmpl Src2, Src1
  - cmpl 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
  \[ (a>0 \&\& b<0 \&\& (a-b)<0) \text{ || } (a<0 \&\& b>0 \&\& (a-b)>0) \]
Condition Codes (Explicit Setting: Test)

- Explicit Setting by Test instruction
  - `testl Src2, Src1`
    - `testl b,a` like computing `a&b` without setting destination
  - Sets condition codes based on value of `Src1 & Src2`
  - Useful to have one of the operands be a mask
  - Use same register for both operands to check if zero

- `ZF` set when `a&b == 0`
- `SF` set when `a&b < 0`

- These are the ways to set the condition codes, now let’s use them

Jumping

- `jX` Instructions
  - Jump to different part of code depending on condition codes
  - Often called branch instructions

<table>
<thead>
<tr>
<th><code>jX</code></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>je</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>
The Program Counter

x = 43;  // movl $43, %ecx
y = 2;   // movl $2, %edx
same = 0; // movl $0, %eax
if (x == y)
same = 1; // movl $1, %eax execute if x == y
x = x + y; // addl %edx, %ecx

Understand branches

Successor Instruction

```c
int equal(int a1, int a2) {
    int tsame;
    tsame = 0;
    if (a1 == a2)
        tsame = 1; // only if a1 == a2
    return(tsame);
}
main() {
    int x,y,same; //
    x = 43;       // movl $43, %ecx
    y = 2;        // movl $2, %edx
    same = equal(x,y); // need to call function
    // other computation
}
The Program Counter

- Function and/or procedure invocation?

```asm
x = 43; // movl $43, %ecx
y = 2; // movl $2, %edx
same = equal(x, y);

movl $43, %ecx
movl $2, %edx
"go execute equal"
```

```asm
movl $0, %eax
cmpl %ecx, %edx
jne 0x30418
movl $1, %eax
addl %edx, %ecx
"return %eax"
```

IA32 Stack

- Region of memory managed with stack discipline
- Grows toward lower addresses

- Register %esp contains lowest stack address
  - address of “top” element

Stack Pointer: %esp
IA32 Stack: Push

- pushl $\text{Src}$
  - Fetch operand at $\text{Src}$
  - Decrement $\%\text{esp}$ by 4
  - Write operand at address given by $\%\text{esp}$

Stack Pointer: $\%\text{esp}$

Stack “Bottom”

Increasing Addresses

Stack Grows Down

IA32 Stack: Pop

Stack Pointer: $\%\text{esp}$

Stack “Bottom”

Increasing Addresses

Stack Grows Down
Procedure Control Flow

- Use stack to support procedure call and return
- **Procedure call:** `call label`
  - Push return address on stack
  - Jump to label
- **Return address:**
  - Address of the next instruction right after call
  - Example from disassembly

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x804854e</td>
<td><code>e8 3d 06 00 00</code></td>
</tr>
<tr>
<td></td>
<td><code>call 8048b90 &lt;main&gt;</code></td>
</tr>
<tr>
<td>0x8048553</td>
<td><code>50</code></td>
</tr>
<tr>
<td></td>
<td><code>pushl %eax</code></td>
</tr>
</tbody>
</table>

- **Return address = 0x8048553**

- **Procedure return:** `ret`
  - Pop address from stack
  - Jump to address

Procedure Call Example

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x804854e</td>
<td><code>e8 3d 06 00 00</code></td>
</tr>
<tr>
<td></td>
<td><code>call 8048b90 &lt;main&gt;</code></td>
</tr>
<tr>
<td>0x8048553</td>
<td><code>50</code></td>
</tr>
<tr>
<td></td>
<td><code>pushl %eax</code></td>
</tr>
</tbody>
</table>

%esp 0x108
%esp 0x104
%esp 0x804854e
%esp: program counter
```
Procedure Return Example

8048591:  c3  ret

ret

0x104
0x8048553

%esp 0x104  %esp 0x108  
%eip 0x8048591  %eip 0x8048553

%eip: program counter

Instructions for Procedure Call and Return

```c
int equal(int a1, int a2) {
    int tsame;
    tsame = 0;
    if (a1 == a2) {
        tsame = 1;
        return (tsame);
    }
    return (tsame);
}
main()
{
    int x,y,same;
    x = 43;
    y = 2;
    same = equal(x,y);
    // other computation
}
```

```
PC  M[%esp]
0x1000  ??
0x10004  ??
0x10008  ??
0x30408  0x1000c
0x3040c  0x1000c
0x30410  0x1000c
0x30414  0x1000c
0x30418  0x1000c
0x3041c  0x1000c
0x1000c  0x1000c
```

movl $43, %ecx
movl $2, %edx
call 0x30408
movl $0, %eax
cmpl %ecx, %edx
jne 0x30418
movl $1, %eax
addl %edx, %ecx
ret
**Data Representations: IA32 + x86-64**

### Sizes of C Objects (in Bytes)

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Generic 32-bit</th>
<th>Intel IA32</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long int</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>8</td>
<td>10/12</td>
<td>16</td>
</tr>
<tr>
<td>char *</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

- Or any other pointer

---

**x86-64 Integer Registers**

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
<th>%r8</th>
<th>%r8d</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
<td>%r15</td>
<td>%r15d</td>
</tr>
</tbody>
</table>

- Extend existing registers. Add 8 new ones.
- Make %ebp/%rbp general purpose
X86-64 Instructions

- Long word l (4 Bytes) ↔ Quad word q (8 Bytes)

- New instructions:
  - movl → movq
  - addl → addq
  - sal → salq
  - etc.

- 32-bit instructions that generate 32-bit results
  - Set higher order bits of destination register to 0
  - Example: addl

Reading Condition Codes

- SetX Instructions
  - Set single byte based on combinations of condition codes

<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)&amp;ZF</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>
Reading Condition Codes (Cont.)

- SetX Instructions:
  - Set single byte based on combination of condition codes
  - One of 8 addressable byte registers
    - Does not alter remaining 3 bytes
    - Typically use movzbl to finish job

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

Body

```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
```

Reading Condition Codes: x86-64

- SetX Instructions:
  - Set single byte based on combination of condition codes
  - Does not alter remaining 3 bytes

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

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

Bodies

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

```assembly
cmp %esi, %edi
setg %al
movzbl %al, %eax
```

Is %rax zero?
Yes: 32-bit instructions set high order 32 bits to 0!
IA 32 Instructions & Assembly

- Assembly Basics: Registers, operands, move
  - The x86 move instructions cover wide range of data movement forms
- Types of instructions
  - Arithmetic, logical
  - Control instructions, condition codes
  - Call & return for procedure/function invocation
- Brief differences with x86-64
- Next Time: Assembly Programming