15-213
“The course that gives CMU its Zip!”

Machine-Level Programming I: Introduction
Sept. 10, 2007

Topics

- Assembly Programmer’s Execution Model
- Accessing Information
  - Registers
  - Memory
- Arithmetic operations

class04.ppt
IA32 Processors

Totally Dominate Computer Market

Evolutionary Design

- Starting in 1978 with 8086
- Added more features as time goes on
- Still support old features, although obsolete

Complex Instruction Set Computer (CISC)

- Many different instructions with many different formats
  - But, only small subset encountered with Linux programs
- Hard to match performance of Reduced Instruction Set Computers (RISC)
- But, Intel has done just that!
x86 Evolution: Programmer’s View (Abbreviated)

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086</td>
<td>1978</td>
<td>29K</td>
</tr>
</tbody>
</table>

- 16-bit processor. Basis for IBM PC & DOS
- Limited to 1MB address space. DOS only gives you 640K

| 386  | 1985 | 275K        |

- Extended to 32 bits. Added “flat addressing”
- Capable of running Unix
- Referred to as “IA32”
- 32-bit Linux/gcc uses no instructions introduced in later models
x86 Evolution: Programmer’s View

Machine Evolution

- 486 1989 1.9M
- Pentium 1993 3.1M
- Pentium/MMX 1997 4.5M
- PentiumPro 1995 6.5M
- Pentium III 1999 8.2M
- Pentium 4 2001 42M

Added Features

- Instructions to support multimedia operations
  - Parallel operations on 1, 2, and 4-byte data, both integer & FP
- Instructions to enable more efficient conditional operations

Linux/GCC Evolution

- None!
New Species: IA64

Name       Date       Transistors

Itanium    2001       10M
- Extends to IA64, a 64-bit architecture
- Radically new instruction set designed for high performance
- Can run existing IA32 programs
  ● On-board “x86 engine”
- Joint project with Hewlett-Packard

Itanium 2   2002       221M
- Big performance boost

Itanium 2 Dual-Core  2006   1.7B

Itanium has not taken off in marketplace
- Lack of backward compatibility
X86 Evolution: Clones

Advanced Micro Devices (AMD)

- Historically
  - AMD has followed just behind Intel
  - A little bit slower, a lot cheaper

- Recently
  - Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies
  - Exploited fact that Intel distracted by IA64
  - Now are close competitors to Intel

- Developed x86-64, its own extension to 64 bits
  - Started eating into Intel’s high-end server market
Intel’s 64-Bit Dilemma

Intel Attempted Radical Shift from IA32 to IA64

- Totally different architecture
- Executes IA32 code only as legacy
- Performance disappointing

AMD Stepped in with Evolutionary Solution

- x86-64 (now called “AMD64”)

Intel Felt Obligated to Focus on IA64

- Hard to admit mistake or that AMD is better

2004: Intel Announces EM64T extension to IA32

- Extended Memory 64-bit Technology
- Almost identical to x86-64!
Our Coverage

IA32
- The traditional x86

x86-64
- The emerging standard

Presentation
- Book has IA32
- Handout has x86-64
- Lecture will cover both

Labs
- Lab #2 x86-64
- Lab #3 IA32
Assembly Programmer’s View

Programmer-Visible State

- **PC**  Program Counter
  - Address of next instruction
  - Called “EIP” (IA32) or “RIP” (x86-64)

- **Register File**
  - Heavily used program data

- **Condition Codes**
  - Store status information about most recent arithmetic operation
  - Used for conditional branching

- **Memory**
  - Byte addressable array
  - Code, user data, (some) OS data
  - Includes stack used to support procedures
Turning C into Object Code

- **Code in files**  
  p1.c  p2.c

- **Compile with command:**  
  gcc -O p1.c p2.c -o p
  - Use optimizations (-O)
  - Put resulting binary in file p

![Diagram showing the process from C program to Executable program](image)

- **C program** (p1.c  p2.c)
  - Compiler (gcc -S)

- **Asm program** (p1.s  p2.s)
  - Assembler (gcc or as)

- **Object program** (p1.o  p2.o)
  - Linker (gcc or ld)

- **Executable program** (p)
  - Static libraries (.a)
Compiling Into Assembly

C Code

```c
int sum(int x, int y)
{
    int t = x+y;
    return t;
}
```

Generated IA32 Assembly

```assembly
_sum:  
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

Obtain with command

```
gcc -O -S code.c
```

Produces file `code.s`
Assembly Characteristics

Minimal Data Types

- “Integer” data of 1, 2, or 4 bytes
  - Data values
  - Addresses (untyped pointers)
- Floating point data of 4, 8, or 10 bytes
- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory

Primitive Operations

- Perform arithmetic function on register or memory data
- Transfer data between memory and register
  - Load data from memory into register
  - Store register data into memory
- Transfer control
  - Unconditional jumps to/from procedures
  - Conditional branches
Object Code

Code for `sum`

0x401040 <sum>:

0x55
0x89
0xe5
0x8b
0x45
0x0c
0x03
0x45
0x08
0x89
0xec
0xc3

- Total of 13 bytes
- Each instruction 1, 2, or 3 bytes
- Starts at address 0x401040

Assembler

- Translates `.s` into `.o`
- Binary encoding of each instruction
- Nearly-complete image of executable code
- Missing linkages between code in different files

Linker

- Resolves references between files
- Combines with static run-time libraries
  - E.g., code for `malloc`, `printf`
- Some libraries are `dynamically linked`
  - Linking occurs when program begins execution
Machine Instruction Example

C Code

int t = x+y;

Assembly

- Add two signed integers
- Add 2 4-byte integers
  - "Long" words in GCC parlance
  - Same instruction whether signed or unsigned
- Operands:
  - x: Register %eax
  - y: Memory M[%ebp+8]
  - t: Register %eax
  - Return function value in %eax

Object Code

- 3-byte instruction
- Stored at address 0x401046

```c
int t = x+y;

addl 8(%ebp),%eax
```

Similar to expression:
```
x += y
```

Or
```
int eax;
int *ebp;
eax += ebp[2]
```
Disassembling Object Code

Disassembled

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00401040</td>
<td>55</td>
<td>push %ebp</td>
</tr>
<tr>
<td>00401041</td>
<td>89 e5</td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td>00401043</td>
<td>8b 45 0c</td>
<td>mov 0xc(%ebp),%eax</td>
</tr>
<tr>
<td>00401046</td>
<td>03 45 08</td>
<td>add 0x8(%ebp),%eax</td>
</tr>
<tr>
<td>00401049</td>
<td>89 ec</td>
<td>mov %ebp,%esp</td>
</tr>
<tr>
<td>0040104b</td>
<td>5d</td>
<td>pop %ebp</td>
</tr>
<tr>
<td>0040104c</td>
<td>c3</td>
<td>ret</td>
</tr>
<tr>
<td>0040104d</td>
<td>8d 76 00</td>
<td>lea 0x0(%esi),%esi</td>
</tr>
</tbody>
</table>

Disassembler

objdump -d p

- Useful tool for examining object code
- Analyzes bit pattern of series of instructions
- Produces approximate rendition of assembly code
- Can be run on either a .out (complete executable) or .o file
Alternate Disassembly

Disassembled

Object

Disassembled

Within gdb Debugger

gdb p

disassemble sum

■ Disassemble procedure

x/13b sum

■ Examine the 13 bytes starting at sum

Object

Disassembled

0x401040: 0x55 0x89 0xe5 0x8b 0x45 0x0c 0x03 0x45 0x08 0x89 0xec 0xc3
What Can be Disassembled?

% objdump -d WINWORD.EXE

WINWORD.EXE:     file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:  55             push   %ebp
30001001:  8b ec mov %esp,%ebp
30001003:  6a ff          push   $0xffffffff
30001005:  68 90 10 00 30 push   $0x30001090
3000100a:  68 91 dc 4c 30 push   $0x304cdc91

- Anything that can be interpreted as executable code
- Disassembler examines bytes and reconstructs assembly source
Moving Data: IA32

Moving Data

movl Source, Dest:
- Move 4-byte ("long") word
- Lots of these in typical code

Operand Types

- Immediate: Constant integer data
  - Like C constant, but prefixed with ‘$’
  - E.g., $0x400, $-533
  - Encoded with 1, 2, or 4 bytes
- Register: One of 8 integer registers
  - But %esp and %ebp reserved for special use
  - Others have special uses for particular instructions
- Memory: 4 consecutive bytes of memory
  - Various “address modes”
### movl Operand Combinations

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src, Dest</th>
<th>C Analog</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>Reg</td>
<td>movl $-147,(%eax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td>Mem</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>*p = temp;</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
Simple Addressing Modes

Normal (R) Mem[Reg[R]]
- Register R specifies memory address

```
    movl (%ecx),%eax
```

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

```
    movl 8(%ebp),%edx
```
Using Simple Addressing Modes

void swap(int *xp, int *yp) {
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}

swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
Using Simple Addressing Modes

void swap(int *xp, int *yp) {
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}

swap:

{ Set Up
  pushl %ebp
  movl %esp,%ebp
  pushl %ebx
  movl 12(%ebp),%ecx
  movl 8(%ebp),%edx
  movl (%ecx),%eax
  movl (%edx),%ebx
  movl %eax,(%edx)
  movl %ebx,(%ecx)

  Body
  movl -4(%ebp),%ebx
  movl %ebp,%esp
  popl %ebp
  ret

  Finish
}
Understanding Swap

```c
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>yp</td>
</tr>
<tr>
<td>%edx</td>
<td>xp</td>
</tr>
<tr>
<td>%eax</td>
<td>t1</td>
</tr>
<tr>
<td>%ebx</td>
<td>t0</td>
</tr>
</tbody>
</table>

- `movl 12(%ebp),%ecx` # ecx = yp
- `movl 8(%ebp),%edx` # edx = xp
- `movl (%ecx),%eax` # eax = *yp (t1)
- `movl (%edx),%ebx` # ebx = *xp (t0)
- `movl %eax,(%edx)` # *xp = eax
- `movl %ebx,(%ecx)` # *yp = ebx

Stack

<table>
<thead>
<tr>
<th>Offset</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>yp</td>
</tr>
<tr>
<td>8</td>
<td>xp</td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>0</td>
<td>Old %ebp</td>
</tr>
<tr>
<td>-4</td>
<td>Old %ebx</td>
</tr>
</tbody>
</table>
Understanding Swap

\[
\begin{array}{c|c}
\%eax & 0x120 \\
\%edx & 0x124 \\
\%ecx & 0x128 \\
\%ebx & 0x12c \\
\%esi & 0x130 \\
\%edi & 0x134 \\
\%esp & 0x138 \\
\%ebp & 0x104 \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{Offset} & \text{Address} \\
\hline
12 & 0x120 \\
8 & 0x124 \\
4 & 0x108 \\
0 & 0x10c \\
-4 & 0x110 \\
\end{array}
\]

\[
\begin{array}{l}
\text{movl} \ 12(\%ebp),\%ecx \quad \# \ \text{ecx} = \text{yp} \\
\text{movl} \ 8(\%ebp),\%edx \quad \# \ \text{edx} = \text{xp} \\
\text{movl} \ (\%ecx),\%eax \quad \# \ \text{eax} = \text{*yp} \ (t1) \\
\text{movl} \ (\%edx),\%ebx \quad \# \ \text{ebx} = \text{*xp} \ (t0) \\
\text{movl} \ \%eax, (\%edx) \quad \# \ \text{*xp} = \text{eax} \\
\text{movl} \ \%ebx, (\%ecx) \quad \# \ \text{*yp} = \text{ebx}
\end{array}
\]
Understanding Swap

\[
\begin{array}{c|c}
\%eax & \\
\%edx & \\
\%ecx & 0x120 \\
\%ebx & \\
\%esi & \\
\%edi & \\
\%esp & \\
\%ebp & 0x104 \\
\end{array}
\]

\begin{align*}
\text{movl} & \ 12(\%ebp),\%ecx \quad \# \ \text{ecx} = \text{yp} \\
\text{movl} & \ 8(\%ebp),\%edx \quad \# \ \text{edx} = \text{xp} \\
\text{movl} & \ (%ecx),\%eax \quad \# \ \text{eax} = *\text{yp} \ (t1) \\
\text{movl} & \ (%edx),\%ebx \quad \# \ \text{ebx} = *\text{xp} \ (t0) \\
\text{movl} & \ \%eax,(%edx) \quad \# \ *\text{xp} = \text{eax} \\
\text{movl} & \ \%ebx,(%ecx) \quad \# \ *\text{yp} = \text{ebx} \\
\end{align*}
Understanding Swap

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
<th>Rtn adr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
<td>12</td>
<td>0x120</td>
</tr>
<tr>
<td>0x120</td>
<td>8</td>
<td>0x124</td>
</tr>
<tr>
<td>0x11c</td>
<td>4</td>
<td>0x108</td>
</tr>
<tr>
<td>0x118</td>
<td></td>
<td>0x104</td>
</tr>
<tr>
<td>0x114</td>
<td></td>
<td>0x100</td>
</tr>
</tbody>
</table>

%eax
%edx 0x124
%ecx 0x120
%ebx
%esi
%edi
%esp
%ebp 0x104

movl 12(%ebp),%ecx  # ecx = yp
movl 8(%ebp),%edx  # edx = xp
movl (%ecx),%eax  # eax = *yp (t1)
movl (%edx),%ebx  # ebx = *xp (t0)
movl %eax, (%edx)  # *xp = eax
movl %ebx, (%ecx)  # *yp = ebx
Understanding Swap

```
| %eax  | 456 |
| %edx  | 0x124 |
| %ecx  | 0x120 |
| %ebx  |     |
| %esi  |     |
| %edi  |     |
| %esp  |     |
| %ebp  | 0x104 |

movl 12(%ebp), %ecx  # ecx = yp
movl 8(%ebp), %edx   # edx = xp
movl (%ecx), %eax    # eax = *yp (t1)
movl (%edx), %ebx    # ebx = *xp (t0)
movl %eax, (%edx)    # *xp = eax
movl %ebx, (%ecx)    # *yp = ebx
```

Address

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x124</td>
</tr>
<tr>
<td></td>
<td>0x120</td>
</tr>
<tr>
<td></td>
<td>0x11c</td>
</tr>
<tr>
<td></td>
<td>0x118</td>
</tr>
<tr>
<td></td>
<td>0x114</td>
</tr>
<tr>
<td></td>
<td>0x120</td>
</tr>
<tr>
<td></td>
<td>0x110</td>
</tr>
<tr>
<td></td>
<td>0x124</td>
</tr>
<tr>
<td></td>
<td>0x10c</td>
</tr>
<tr>
<td></td>
<td>0x108</td>
</tr>
<tr>
<td></td>
<td>0x104</td>
</tr>
<tr>
<td></td>
<td>0x100</td>
</tr>
</tbody>
</table>

yp    xp    Rtn adr
Understanding Swap

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>456</td>
</tr>
<tr>
<td>%edx</td>
<td>0x124</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ebx</td>
<td>123</td>
</tr>
<tr>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

```
movl 12(%ebp),%ecx  # ecx = yp
movl 8(%ebp),%edx   # edx = xp
movl (%ecx),%eax    # eax = *yp (t1)
movl (%edx),%ebx    # ebx = *xp (t0)
movl %eax,(%edx)    # *xp = eax
movl %ebx,(%ecx)    # *yp = ebx
```
Understanding Swap

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>456</td>
</tr>
<tr>
<td>%edx</td>
<td>0x124</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ebx</td>
<td>123</td>
</tr>
<tr>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%esp</td>
<td>0x104</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

```
movl 12(%ebp),%ecx  # ecx = yp
movl 8(%ebp),%edx   # edx = xp
movl (%ecx),%eax    # eax = *yp (t1)
movl (%edx),%ebx    # ebx = *xp (t0)
movl %eax,(%edx)    # *xp = eax
movl %ebx,(%ecx)    # *yp = ebx
```

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0x120</td>
</tr>
<tr>
<td>8</td>
<td>0x124</td>
</tr>
<tr>
<td>4</td>
<td>0x10c</td>
</tr>
<tr>
<td>0</td>
<td>0x108</td>
</tr>
<tr>
<td>-4</td>
<td>0x100</td>
</tr>
<tr>
<td></td>
<td>0x104</td>
</tr>
<tr>
<td></td>
<td>0x114</td>
</tr>
<tr>
<td></td>
<td>0x118</td>
</tr>
<tr>
<td>456</td>
<td>0x120</td>
</tr>
<tr>
<td>456</td>
<td>0x124</td>
</tr>
<tr>
<td>456</td>
<td>0x124</td>
</tr>
<tr>
<td></td>
<td>0x10c</td>
</tr>
</tbody>
</table>

Address Table:

- **456**: 0x124
- **456**: 0x120
- **456**: 0x124
- **456**: 0x124
- **456**: 0x10c
- **456**: 0x10c
- **456**: 0x10c
- **456**: 0x10c
- **456**: 0x10c
- **456**: 0x10c
- **456**: 0x10c
- **456**: 0x10c

Rtn adr: 0x104

Offset:
- **YP**: 12
- **xp**: 8
- **ebp**: 0
- **%ebp**: -4
# Understanding Swap

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>456</td>
</tr>
<tr>
<td>%edx</td>
<td>0x124</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ebx</td>
<td>123</td>
</tr>
<tr>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset</th>
<th>YP</th>
<th>xp</th>
<th>Rtn adr</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4</td>
<td>12</td>
<td>8</td>
<td>0x104</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>0x108</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0x100</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>0x10c</td>
</tr>
<tr>
<td>0x120</td>
<td></td>
<td></td>
<td>0x110</td>
</tr>
<tr>
<td>0x124</td>
<td></td>
<td></td>
<td>0x114</td>
</tr>
<tr>
<td>0x118</td>
<td></td>
<td></td>
<td>0x118</td>
</tr>
<tr>
<td>0x120</td>
<td></td>
<td></td>
<td>0x120</td>
</tr>
</tbody>
</table>

```
movl 12(%ebp),%ecx # ecx = yp
movl 8(%ebp),%edx # edx = xp
movl (%ecx),%eax # eax = *yp (t1)
movl (%edx),%ebx # ebx = *xp (t0)
movl %eax,(%edx) # *xp = eax
movl %ebx,(%ecx) # *yp = ebx
```
Indexed Addressing Modes

Most General Form

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

- **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

Special Cases

- \((Rb, Ri)\) \rightarrow \text{Mem} [\text{Reg}[Rb] + \text{Reg}[Ri]]
- \(D(Rb, Ri)\) \rightarrow \text{Mem} [\text{Reg}[Rb] + \text{Reg}[Ri] + D]
- \((Rb, Ri, S)\) \rightarrow \text{Mem} [\text{Reg}[Rb] + S \cdot \text{Reg}[Ri]]
Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>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>
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.`
# Some Arithmetic Operations

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Two Operand Instructions</strong></td>
<td></td>
</tr>
<tr>
<td><code>addl Src,Dest</code></td>
<td><code>Dest = Dest + Src</code></td>
</tr>
<tr>
<td><code>subl Src,Dest</code></td>
<td><code>Dest = Dest - Src</code></td>
</tr>
<tr>
<td><code>imull Src,Dest</code></td>
<td><code>Dest = Dest * Src</code></td>
</tr>
<tr>
<td><code>sall Src,Dest</code></td>
<td><code>Dest = Dest &lt;&lt; Src</code> Also called <code>shll</code></td>
</tr>
<tr>
<td><code>sarl Src,Dest</code></td>
<td><code>Dest = Dest &gt;&gt; Src</code> Arithmetic</td>
</tr>
<tr>
<td><code>shrl Src,Dest</code></td>
<td><code>Dest = Dest &gt;&gt; Src</code> Logical</td>
</tr>
<tr>
<td><code>xorl Src,Dest</code></td>
<td><code>Dest = Dest ^ Src</code></td>
</tr>
<tr>
<td><code>andl Src,Dest</code></td>
<td><code>Dest = Dest &amp; Src</code></td>
</tr>
<tr>
<td><code>orl Src,Dest</code></td>
<td>`Dest = Dest</td>
</tr>
</tbody>
</table>
### Some Arithmetic Operations

#### Format

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>incl Dest</code></td>
<td><code>Dest = Dest + 1</code></td>
</tr>
<tr>
<td><code>decl Dest</code></td>
<td><code>Dest = Dest - 1</code></td>
</tr>
<tr>
<td><code>negl Dest</code></td>
<td><code>Dest = - Dest</code></td>
</tr>
<tr>
<td><code>notl Dest</code></td>
<td><code>Dest = ~ Dest</code></td>
</tr>
</tbody>
</table>
Using `leal` for Arithmetic Expressions

```c
int arith
    (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;
}
```

**arith:**

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

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

movl %ebp,%esp
popl %ebp
ret
```

Set Up

Body

Finish
Understanding arith

```c
int arith
  (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;
}
```

```
movl 8(%ebp),%eax  # eax = x
movl 12(%ebp),%edx  # edx = y
leal (%edx,%eax),%ecx  # ecx = x+y  (t1)
leal (%edx,%edx,2),%edx  # edx = 3*y
sall $4,%edx  # edx = 48*y  (t4)
addl 16(%ebp),%ecx  # ecx = z+t1  (t2)
leal 4(%edx,%eax),%eax  # eax = 4+t4+x  (t5)
imull %ecx,%eax  # eax = t5*t2  (rval)
```
int arith
  (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;
}

movl 8(%ebp),%eax         # eax = x
movl 12(%ebp),%edx        # edx = y
leal (%edx,%eax),%ecx     # ecx = x+y (t1)
leal (%edx,%edx,2),%edx    # edx = 3*y
sall $4,%edx              # edx = 48*y (t4)
addl 16(%ebp),%ecx        # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax    # eax = 4+t4+x (t5)
imull %ecx,%eax
                      # eax = t5*t2 (rval)
Understanding arith

```c
int arith
    (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;
}
```

movl 8(%ebp),%eax  # eax = x
movl 12(%ebp),%edx  # edx = y
leal (%edx,%eax),%ecx  # ecx = x+y (t1)
leal (%edx,%edx,2),%edx  # edx = 3*y
sall $4,%edx  # edx = 48*y (t4)
addl 16(%ebp),%ecx  # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax  # eax = 4+t4+x (t5)
imull %ecx,%eax  # eax = t5*t2 (rval)
Understanding arith

```c
int arith
    (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;
}
```

```
movl 8(%ebp),%eax  # eax = x
movl 12(%ebp),%edx  # edx = y
lea (%edx,%eax),%ecx  # ecx = x+y (t1)
lea (%edx,%edx,2),%edx  # edx = 3*y
sal $4,%edx  # edx = 48*y (t4)
addl 16(%ebp),%ecx  # ecx = z+t1 (t2)
lea 4(%edx,%eax),%eax  # eax = 4+t4+x (t5)
imull %ecx,%eax  # eax = t5*t2 (rval)
```
int arith
  (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;
}

Understanding arith

movl 8(%ebp),%eax  # eax = x
movl 12(%ebp),%edx  # edx = y
leal (%edx,%eax),%ecx  # ecx = x+y (t1)
leal (%edx,%edx,2),%edx  # edx = 3*y
sall $4,%edx  # edx = 48*y (t4)
addl 16(%ebp),%ecx  # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax  # eax = 4+t4+x (t5)
imull %ecx,%eax  # eax = t5*t2 (rval)
int arith
  (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;
}

movl 8(%ebp),%eax  # eax = x
movl 12(%ebp),%edx  # edx = y
leal (%edx,%eax),%ecx  # ecx = x+y (t1)
leal (%edx,%edx,2),%edx  # edx = 3*y
sall $4,%edx  # edx = 48*y (t4)
addl 16(%ebp),%ecx  # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax  # eax = 4+t4+x (t5)
imull %ecx,%eax  # eax = t5*t2 (rval)
Another Example

```c
int logical(int x, int y)
{
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

Logical:
- **Set Up**
  - `pushl %ebp`
  - `movl %esp,%ebp`

- **Body**
  - `movl 8(%ebp),%eax`
  - `xorl 12(%ebp),%eax`
  - `sarl $17,%eax`
  - `andl $8185,%eax`
- **Finish**
  - `movl %ebp,%esp`
  - `popl %ebp`
  - `ret`

**Assembly Code**
- `movl 8(%ebp),%eax`  
  \( \text{eax} = x \)
- `xorl 12(%ebp),%eax`  
  \( \text{eax} = x^y \)
- `sarl $17,%eax`  
  \( \text{eax} = t1 >> 17 \)
- `andl $8185,%eax`  
  \( \text{eax} = t2 \& 8185 \)
Another Example

```c
int logical(int x, int y)
{
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

logical:

```assembly
        pushl %ebp
        movl %esp,%ebp
        movl 8(%ebp),%eax        eax = x
        xorl 12(%ebp),%eax       eax = x^y (t1)
        sarl $17,%eax            eax = t1>>17 (t2)
        andl $8185,%eax          eax = t2 & 8185
        movl %ebp,%esp
        popl %ebp
        ret
```
Another Example

```c
int logical(int x, int y) {
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

**logical:**

```asm
pushl %ebp
movl %esp,%ebp
movl 8(%ebp),%eax
xorl 12(%ebp),%eax
sarl $17,%eax
andl $8185,%eax
movl %ebp,%esp
popl %ebp
ret
```

- **Set Up**
  - movl 8(%ebp),%eax  
    - eax = x
  - xorl 12(%ebp),%eax  
    - eax = x^y (t1)
  - sarl $17,%eax  
    - eax = t1>>17 (t2)
  - andl $8185,%eax  
    - eax = t2 & 8185

- **Body**

- **Finish**
Another Example

```c
int logical(int x, int y)
{
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

2^{13} = 8192, 2^{13} – 7 = 8185

Logical:
- **Set Up**
  - pushl %ebp
  - movl %esp,%ebp
- **Body**
  - movl 8(%ebp),%eax
  - xorl 12(%ebp),%eax
  - sarl $17,%eax
  - andl $8185,%eax
- **Finish**
  - movl %ebp,%esp
  - popl %ebp
  - ret

- eax = x
- eax = x^y (t1)
- eax = t1>>17 (t2)
- eax = t2 & 8185 (rval)
Data Representations: IA32 + x86-64

Sizes of C Objects (in Bytes)

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Typical 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
- Extend existing registers. Add 8 new ones.
- Make %ebp/%rbp general purpose
Swap in 32-bit Mode

```c
void swap(int *xp, int *yp) {
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

**swap:**

```assembly
pushl %ebp
movl %esp,%ebp
pushl %ebx
movl 12(%ebp),%ecx
movl 8(%ebp),%edx
movl (%ecx),%eax
movl (%edx),%ebx
movl %eax,(%edx)
movl %ebx,(%ecx)
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```
Swap in 64-bit Mode

```c
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```assembly
swap:
    movl (%rdi), %edx
    movl (%rsi), %eax
    movl %eax, (%rdi)
    movl %edx, (%rsi)
    ret
```

- **Operands passed in registers**
  - First \((xp)\) in \(%rdi\), second \((yp)\) in \(%rsi\)
  - 64-bit pointers
- **No stack operations required**
- **32-bit data**
  - Data held in registers \(%eax\) and \(%edx\)
  - \texttt{movl} operation
Swap Long Ints in 64-bit Mode

```c
void swap_l
    (long int *xp, long int *yp)
{
    long int t0 = *xp;
    long int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

swap_l:
    movq (%rdi), %rdx
    movq (%rsi), %rax
    movq %rax, (%rdi)
    movq %rdx, (%rsi)
    ret

- 64-bit data
  - Data held in registers %rax and %rdx
  - movq operation
    » “q” stands for quad-word
Summary

Machine Level Programming

- Assembly code is textual form of binary object code
- Low-level representation of program
  - Explicit manipulation of registers
  - Simple and explicit instructions
  - Minimal concept of data types
  - Many C control constructs must be implemented with multiple instructions

Formats

- IA32: Historical x86 format
- x86-64: Big evolutionary step