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

Machine-Level Programming I: Introduction
August 27, 2008

Topics
- Assembly Programmer’s Execution Model
- Accessing Information
  - Registers
  - Memory
- Arithmetic operations
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>
<tr>
<td></td>
<td></td>
<td>- 16-bit processor. Basis for IBM PC &amp; DOS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Limited to 1MB address space. DOS only gives you 640K</td>
</tr>
<tr>
<td>386</td>
<td>1985</td>
<td>275K</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Extended to 32 bits. Added “flat addressing”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Capable of running Unix</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Referred to as “IA32”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- 32-bit Linux/gcc uses no instructions introduced in later models</td>
</tr>
</tbody>
</table>
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
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
- Lecture will cover IA32

Labs
- Lab #2 IA32
- 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

---

- **CPU**
  - Registers
  - Condition Codes

- **Memory**
  - Object Code
  - Program Data
  - OS Data

- **Stack**
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`

```
C program (`p1.c` `p2.c`)
```

```
Assembler (gcc or as)
```

```
Object program (`p1.o` `p2.o`)
```

```
Executable program (`p`)
```

```
Static libraries (.a)
```

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`
Compiling Into Assembly

C Code

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

Generated IA32 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
- 0x5d
- 0xc3

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

```c
int t = x+y;
```

- Add two signed integers

Assembly

- 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

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

Similar to expression:

```c
x += y
```

Or

```c
int eax;
int *ebp;
eax += ebp[2]
```

```
0x401046: 03 45 08
```
Disassembling Object Code

Disassembled

```
00401040 <_sum>:
  0:   55  push   %ebp
  1:  89 e5  mov %esp,%ebp
  3:  8b 45 0c  mov 0xc(%ebp),%eax
  6:  03 45 08  add 0x8(%ebp),%eax
  9:  89 ec  mov %ebp,%esp
  b:  5d  pop    %ebp
 c:  c3  ret
 d:  8d 76 00 lea 0x0(%esi),%esi
```

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**

<table>
<thead>
<tr>
<th>Object</th>
<th>Disassembled</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401040:</td>
<td>0x401040 &lt;sum&gt;: push  %ebp</td>
</tr>
<tr>
<td></td>
<td>0x401041 &lt;sum+1&gt;: mov %esp,%ebp</td>
</tr>
<tr>
<td></td>
<td>0x401043 &lt;sum+3&gt;: mov 0xc(%ebp),%eax</td>
</tr>
<tr>
<td></td>
<td>0x401046 &lt;sum+6&gt;: add 0x8(%ebp),%eax</td>
</tr>
<tr>
<td></td>
<td>0x401049 &lt;sum+9&gt;: mov %ebp,%esp</td>
</tr>
<tr>
<td></td>
<td>0x40104b &lt;sum+11&gt;: pop %ebp</td>
</tr>
<tr>
<td></td>
<td>0x40104c &lt;sum+12&gt;: ret</td>
</tr>
<tr>
<td></td>
<td>0x40104d &lt;sum+13&gt;: lea 0x0(%esi),%esi</td>
</tr>
</tbody>
</table>

**Within gdb Debugger**

- `gdb p` disassemble sum
- **Disassemble procedure**
- **Examine the 13 bytes starting at sum**
What Can be Disassembled?

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><strong>Reg</strong></td>
<td>movl $0x4,%eax</td>
<td>temp = 0x4;</td>
<td></td>
</tr>
<tr>
<td><strong>Imm</strong></td>
<td>movl $-147,(%eax)</td>
<td>*p = -147;</td>
<td></td>
</tr>
<tr>
<td><strong>Mem</strong></td>
<td>movl %eax,%edx</td>
<td>temp2 = temp1;</td>
<td></td>
</tr>
<tr>
<td><strong>Reg</strong></td>
<td>movl %eax,(%edx)</td>
<td>*p = temp;</td>
<td></td>
</tr>
<tr>
<td><strong>Mem</strong></td>
<td>movl (%eax),%edx</td>
<td>temp = *p;</td>
<td></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

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

```assembly
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
```

Set Up
Body
Finish
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
void swap(int *xp, int *yp) {
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}

`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

**Register Variable**

<table>
<thead>
<tr>
<th>%ecx</th>
<th>yp</th>
</tr>
</thead>
<tbody>
<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>
Understanding Swap

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>Address</th>
<th>Offset</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0</td>
<td>%ebp</td>
</tr>
<tr>
<td>0x104</td>
<td>-4</td>
<td>%esp</td>
</tr>
<tr>
<td>0x110</td>
<td>4</td>
<td>%esi</td>
</tr>
<tr>
<td>0x114</td>
<td>8</td>
<td>%ebx</td>
</tr>
<tr>
<td>0x118</td>
<td>12</td>
<td>%ecx</td>
</tr>
<tr>
<td>0x120</td>
<td></td>
<td>%edx</td>
</tr>
<tr>
<td>0x124</td>
<td></td>
<td>%eax</td>
</tr>
</tbody>
</table>

```c
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>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>0x110</td>
</tr>
<tr>
<td>0</td>
<td>0x108</td>
</tr>
<tr>
<td>-4</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>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
</tr>
<tr>
<td>0x120</td>
</tr>
<tr>
<td>0x11c</td>
</tr>
<tr>
<td>0x118</td>
</tr>
<tr>
<td>0x114</td>
</tr>
<tr>
<td>0x100</td>
</tr>
<tr>
<td>0x108</td>
</tr>
<tr>
<td>0x104</td>
</tr>
<tr>
<td>0x100</td>
</tr>
<tr>
<td>0x100</td>
</tr>
</tbody>
</table>

### Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x124</td>
</tr>
<tr>
<td>%edx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x11c</td>
</tr>
<tr>
<td>%ebx</td>
<td>0x118</td>
</tr>
<tr>
<td>%esi</td>
<td>0x114</td>
</tr>
<tr>
<td>%edi</td>
<td>0x110</td>
</tr>
<tr>
<td>%esp</td>
<td>0x10c</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x108</td>
</tr>
</tbody>
</table>

### Code Snippet

```assembly
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
```

### Offset Table

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0x124</td>
</tr>
<tr>
<td>8</td>
<td>0x120</td>
</tr>
<tr>
<td>4</td>
<td>0x10c</td>
</tr>
<tr>
<td>0</td>
<td>0x104</td>
</tr>
</tbody>
</table>
# Understanding Swap

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
<th>YP</th>
<th>xp</th>
<th>Rtn adr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
<td>12</td>
<td>0x120</td>
<td>0x124</td>
<td>0x104</td>
</tr>
<tr>
<td>0x120</td>
<td>8</td>
<td>0x110</td>
<td>0x124</td>
<td>0x100</td>
</tr>
<tr>
<td>0x11c</td>
<td>4</td>
<td>0x114</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x118</td>
<td></td>
<td>0x110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x114</td>
<td></td>
<td>0x110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
<td>0x110</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| %eax   | 456    |
| %edx   | 0x124  |
| %ecx   | 0x120  |
| %ebx   | 123    |
| %esi   |        |
| %edi   |        |
| %esp   |        |
| %ebp   | 0x104  |

```asm
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

### Address

<table>
<thead>
<tr>
<th>Offset</th>
<th>%ebp</th>
<th>0x00000010</th>
<th>0x0108</th>
<th>0x0114</th>
<th>0x0118</th>
<th>0x0120</th>
<th>0x0124</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp</td>
<td>12</td>
<td>0x108</td>
<td>0x114</td>
<td>0x118</td>
<td>0x120</td>
<td>0x124</td>
<td>4</td>
</tr>
<tr>
<td>xp</td>
<td>8</td>
<td>0x108</td>
<td>0x114</td>
<td>0x118</td>
<td>0x120</td>
<td>0x124</td>
<td>4</td>
</tr>
<tr>
<td>Rtn adr</td>
<td>0</td>
<td>0x100</td>
<td>0x108</td>
<td>0x114</td>
<td>0x118</td>
<td>0x120</td>
<td>0x124</td>
</tr>
</tbody>
</table>

| %eax  | 456  |
| %edx  | 0x124 |
| %ecx  | 0x120 |
| %ebx  | 123  |
| %esi  |      |
| %edi  |      |
| %esp  | 0x104 |
| %ebp  | 0x104 |

### Assembly Code

```
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>Address</th>
<th>Offset</th>
<th>Rtn adr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
<td>456</td>
<td>0x104</td>
</tr>
<tr>
<td>0x120</td>
<td>123</td>
<td>0x108</td>
</tr>
<tr>
<td>0x11c</td>
<td>0x120</td>
<td>0x10c</td>
</tr>
<tr>
<td>0x118</td>
<td>0x110</td>
<td>0x114</td>
</tr>
<tr>
<td>0x114</td>
<td>0x124</td>
<td>0x108</td>
</tr>
<tr>
<td>0x10c</td>
<td>0x124</td>
<td>0x108</td>
</tr>
<tr>
<td>0x108</td>
<td>0x120</td>
<td>0x104</td>
</tr>
<tr>
<td>0x104</td>
<td>0x120</td>
<td>0x100</td>
</tr>
<tr>
<td>0x100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

%eax  | 456
%edx  | 0x124
%ecx  | 0x120
%ebx  | 123
%esi  | 0x11c
%edi  | 0x118
%esp  | 0x114
%ebp  | 0x10c

```
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
```

\[ \Rightarrow \text{yp} = \text{ebx} \]
Indexed 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 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

- \( (Rb, Ri) \quad \text{Mem[Reg[Rb]+Reg[Ri]]} \)
- \( D(Rb, Ri) \quad \text{Mem[Reg[Rb]+Reg[Ri]+D]} \)
- \( (Rb, Ri, S) \quad \text{Mem[Reg[Rb]+S*Reg[Ri]]} \)
# 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>addl <strong>Src,Dest</strong></td>
<td>( \text{Dest} = \text{Dest} + \text{Src} )</td>
</tr>
<tr>
<td>subl <strong>Src,Dest</strong></td>
<td>( \text{Dest} = \text{Dest} - \text{Src} )</td>
</tr>
<tr>
<td>imull <strong>Src,Dest</strong></td>
<td>( \text{Dest} = \text{Dest} \times \text{Src} )</td>
</tr>
<tr>
<td>sall <strong>Src,Dest</strong></td>
<td>( \text{Dest} = \text{Dest} \ll \text{Src} ) Also called shll</td>
</tr>
<tr>
<td>sarl <strong>Src,Dest</strong></td>
<td>( \text{Dest} = \text{Dest} \gg \text{Src} ) Arithmetic</td>
</tr>
<tr>
<td>shrl <strong>Src,Dest</strong></td>
<td>( \text{Dest} = \text{Dest} \gg \text{Src} ) Logical</td>
</tr>
<tr>
<td>xorl <strong>Src,Dest</strong></td>
<td>( \text{Dest} = \text{Dest} \oplus \text{Src} )</td>
</tr>
<tr>
<td>andl <strong>Src,Dest</strong></td>
<td>( \text{Dest} = \text{Dest} &amp; \text{Src} )</td>
</tr>
<tr>
<td>orl <strong>Src,Dest</strong></td>
<td>( \text{Dest} = \text{Dest} \mid \text{Src} )</td>
</tr>
</tbody>
</table>
# Some Arithmetic Operations

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>One Operand Instructions</strong></td>
<td></td>
</tr>
<tr>
<td>incl (Dest)</td>
<td>(Dest = Dest + 1)</td>
</tr>
<tr>
<td>decl (Dest)</td>
<td>(Dest = Dest - 1)</td>
</tr>
<tr>
<td>negl (Dest)</td>
<td>(Dest = -Dest)</td>
</tr>
<tr>
<td>notl (Dest)</td>
<td>(Dest = \neg Dest)</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;
}
```

```
set up:
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
```
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;
}
```

### Assembly Code:

- `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 \texttt{arith}

\begin{verbatim}
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;
}
\end{verbatim}

\begin{verbatim}
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 (t4)
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)
\end{verbatim}
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

Stack

<table>
<thead>
<tr>
<th>Offset</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>z</td>
</tr>
<tr>
<td>12</td>
<td>y</td>
</tr>
<tr>
<td>8</td>
<td>x</td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>0</td>
<td>Old %ebp</td>
</tr>
</tbody>
</table>

%ebp
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;
}
```

```assembly
movl 8(%ebp),%eax  ;; eax = x
xorl 12(%ebp),%eax  ;; eax = x^y
sarl $17,%eax  ;; eax = t1>>17
andl $8185,%eax  ;; eax = t2 & 8185
...
```

### Breakdown

- **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`
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:**
```
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
```

```plaintext
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
```
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;
}
```

```assembly
logical:
    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

set up

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;
}
```

\[2^{13} = 8192, \quad 2^{13} - 7 = 8185\]

```
logical:
    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
```

\[\text{Set Up}\]

\[\text{Body}\]

\[\text{Finish}\]
## 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
## x86-64 General Purpose Registers

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

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdx</td>
<td>%edx</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
</tr>
<tr>
<td>%rbx</td>
<td>%ebx</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%r8</th>
<th>%r8d</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<td>%r15</td>
<td>%r15d</td>
</tr>
</tbody>
</table>
Swap in 32-bit Mode

```c
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
```
Swap in 64-bit Mode

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

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
  - movl operation
Swap Long Ints in 64-bit Mode

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