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

Machine-Level Programming I: Introduction
Jan. 22, 2008

Topics

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

Lab 1

- Time roughly 50% done
- Many have started early and made good progress
  - Good
- Warning to others...
  - This isn't the same kind of thing you've done before
  - Please don't leave it to the last minute

Fish-machine log-ins

- Please let us know (staff mailing list) if you can't log in to any machine
Outline

Some computer languages
- Whitespace
- Intercal
- M

Some discussion of x86, x86-64
- Warning: Chapter 3 doesn't compress well
  - 100 pages of discussion about machine language
  - ... after 75 pages of data representation in Chapter 2
- Please plan to spend time reading the text!
A Whitespace Program

“Count from 1 to 10” (partial listing)

Features of Whitespace

- Only space, tab, and line-feed encode program statements
- All other characters (A-Z, a-z, 0-9, etc.) encode comments
- Simple stack-based language
# Whitespace “Explained”

<table>
<thead>
<tr>
<th>Statement</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Space][Space][Space][Space] [Tab][LF]</td>
<td>Push 1 onto stack</td>
</tr>
<tr>
<td>[LF][Space][Space][Space][Space] [Tab][Space][Space][Space][Space] [Space][Tab][Tab][LF]</td>
<td>Set a label at this point</td>
</tr>
<tr>
<td>[Space][LF][Space]</td>
<td>Duplicate the top stack item</td>
</tr>
<tr>
<td>[Tab][LF][Space][Tab]</td>
<td>Output the current value</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
INTERCAL

Features of INTERCAL

- Designed late one night in 1972 by two Princeton students
- Deliberately obfuscated language

Variables

- 16-bit integers, .1 through .65535
- 32-bit integers, :1 through :65535

Operators

- Binary: “mingle”, “select”
- Unary: AND, OR, XOR
  - How are those unary???
  - Simple: AND and's together adjacent bits in a word
- Simplest way to put 65536 in a 32-bit variable?
  - \textbf{DO :1 <- #0¢#256}
The language “M”

Features of M

- Also designed in the 1970's
- More widely used than Whitespace, INTERCAL

Variables

- 32-bit integer variables: A, B, C, D, DI, F, S, SI
- One array, M[
  - Valid subscripts range from near zero to a large number
  - But most subscripts in that range will crash your program!

Statements

- Lots of arithmetic and logical operations
- Input and output use a special statement called OUCH!
A Program in M

C Code

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

M

sum:
    A = M[S+4]
    A += M[S+8]
    DONE
A Program in M

C Code

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

M

```plaintext
sum:
    A = M[S+4]
    A += M[S+8]
    DONE
```

Had enough of M?

- Too bad! We'll study it for much of the semester!
- Why???
M is “The Language of the Machines”
Everything Else is Illusion

```java
* Forks and waits on a new (@link Process).
* @param parentIn the (@link InputStream) that contains the child's input.
* @param parentOut the (@link OutputStream) that receive the child's output.
* @param execArg the command and arguments to execute.
* @return the return code of the child process.
* @throws IOException when something goes wrong.
* /\n* final public static int execAndWaitFor; //
* final InputStream parentIn, final OutputStream parentOut, String... execArg)
* throws IOException {
*     ProcessBuilder pb = new ProcessBuilder(execArg);
*     pb.redirectErrorStream(true);
*     Process p = null;
*     final AtomicReference<IOException> childThreadException = new AtomicReference<IOException>(null);
*     try {
*         p = pb.start();
*         final InputStream in = p.getInputStream();
*         final OutputStream out = p.getOutputStream();
*         Thread t1 = new Thread("Process Writer") {
*             public void run() {
*                 try {
*                     for (int i = 1; i <= parentIn.read(); ++i) {
*                           // Write
*                     }
*                 }
*             }
*         }
*     }
*     finally {
*         childThreadException.get() = null;
*     }
*     return p.waitFor();
*   }
* }
```
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

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Itanium</td>
<td>2001</td>
<td>10M</td>
</tr>
<tr>
<td>Extends to IA64, a 64-bit architecture</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radically new instruction set designed for high performance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Can run existing IA32 programs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>On-board “x86 engine”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Joint project with Hewlett-Packard</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Itanium 2</td>
<td>2002</td>
<td>221M</td>
</tr>
<tr>
<td>Big performance boost</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Itanium 2 Dual-Core</td>
<td>2006</td>
<td>1.7B</td>
</tr>
<tr>
<td>Itanium has not taken off in marketplace</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lack of backward compatibility</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 Saltwater fish machines
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`

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

```
Asm program (p1.s p2.s)
```

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

```
Executable program (p)
```

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

- Linker (`gcc` or `ld`)

```
Compiler (gcc -s)
```

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

- 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**
- 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

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

```
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**

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>00401040</td>
<td>&lt;__sum&gt;:</td>
</tr>
<tr>
<td>0:</td>
<td>55 push %ebp</td>
</tr>
<tr>
<td>1:</td>
<td>89 e5 mov %esp,%ebp</td>
</tr>
<tr>
<td>3:</td>
<td>8b 45 0c mov 0xc(%ebp),%eax</td>
</tr>
<tr>
<td>6:</td>
<td>03 45 08 add 0x8(%ebp),%eax</td>
</tr>
<tr>
<td>9:</td>
<td>89 ec mov %ebp,%esp</td>
</tr>
<tr>
<td>b:</td>
<td>5d pop %ebp</td>
</tr>
<tr>
<td>c:</td>
<td>c3 ret</td>
</tr>
<tr>
<td>d:</td>
<td>8d 76 00 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

Object

| 0x401040: | 0x55 0x89 0xe5 0x03 0x45 0x0c 0x8b 0x45 0x08 0x89 0x0c |

Disassembled

| 0x401040 <sum>: | push %ebp |
| 0x401041 <sum+1>: | mov %esp,%ebp |
| 0x401043 <sum+3>: | mov 0xc(%ebp),%eax |
| 0x401046 <sum+6>: | add 0x8(%ebp),%eax |
| 0x401049 <sum+9>: | mov %ebp,%esp |
| 0x40104b <sum+11>: | pop %ebp |
| 0x40104c <sum+12>: | ret |
| 0x40104d <sum+13>: | lea 0x0(%esi),%esi |

Within gdb Debugger

gdb p
disassemble sum

- Disassemble procedure
x/13b sum
- 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

% 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
Moving Data: IA32

Moving Data

\texttt{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>Reg</td>
<td>Reg</td>
<td>movl %eax,%edx</td>
<td>temp2 = temp1;</td>
</tr>
<tr>
<td>Reg</td>
<td>Mem</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**
So Many Addressing Modes?

Consider C code

```c
struct s {
    double d; // Occupies bytes 0...7
    int i;   // Occupies bytes 8...B
    int j;   // Occupies bytes C...F
};

int get_i(struct s *sarray, int which)
{
    return sarray[which].i;
}
```

Where is “sarray[which].i”?

- i starts at byte 8 of a struct...
- ...which is located at (which*16) from the base of sarray
- ...which somebody knows the “base address” of...
- So our value is at M[8 + (base + (which*16))]
  - Parts: constant 8, variable which, variable/constant base
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
Simple Addressing Modes

Normal (R) Mem[Reg[R]]

- Register R specifies memory address

  \texttt{movl \%ecx,\%eax}

Displacement D(R) Mem[Reg[R]+D]

- Register R specifies start of memory region
- Constant displacement D specifies offset

  \texttt{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;
}
```

```
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 %eax,(%edx)
    movl %ebx,(%ecx)
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```
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;
}

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

```
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

\[
\begin{align*}
\text{movl } 12(\%ebp),&\%ecx & \# ecx = yp \\
\text{movl } 8(\%ebp),&\%edx & \# edx = xp \\
\text{movl } (\%ecx),&\%eax & \# eax = *yp \ (t1) \\
\text{movl } (\%edx),&\%ebx & \# ebx = *xp \ (t0) \\
\text{movl } \%eax,(\%edx) & \# *xp = eax \\
\text{movl } \%ebx,(\%ecx) & \# *yp = ebx \\
\end{align*}
\]
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>123</td>
<td>0x120</td>
<td>0x114</td>
</tr>
<tr>
<td>0x120</td>
<td>8</td>
<td>456</td>
<td>0x124</td>
<td>0x110</td>
</tr>
<tr>
<td>0x11c</td>
<td>4</td>
<td></td>
<td>0x124</td>
<td>0x10c</td>
</tr>
<tr>
<td>0x118</td>
<td>0</td>
<td></td>
<td></td>
<td>0x108</td>
</tr>
<tr>
<td>0x114</td>
<td>-4</td>
<td></td>
<td></td>
<td>0x104</td>
</tr>
<tr>
<td>0x10c</td>
<td></td>
<td></td>
<td></td>
<td>0x100</td>
</tr>
</tbody>
</table>

\[
\begin{array}{cccc}
%eax & %edx & %ecx & %ebx \\
\hline
0x124 & 0x120 & 0x11c & 0x118 \\
\end{array}
\]

\[
\begin{array}{cccc}
%esi & %edi & %esp & %ebp \\
\hline
0x120 & 0x104 & 0x124 & 0x100 \\
\end{array}
\]

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

\[
\begin{array}{|c|c|}
\hline
\%eax & 456 \\
\%edx & 0x124 \\
\%ecx & 0x120 \\
\%ebx & \\
\%esi & \\
\%edi & \\
\%esp & \text{green} \ 0x104 \\
\%ebp & \text{green} \ 0x104 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{Offset} & \text{Address} \\
\hline
yp & 12 \ 0x120 \\
xp & 8 \ 0x124 \\
\text{Rtn adr} & 4 \ 0x10c \\
0 & 0 \ 0x104 \\
-4 & 0 \ 0x100 \\
\hline
\end{array}
\]

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

```plaintext
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>YP</th>
<th>XP</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0x120</td>
<td>0x124</td>
</tr>
<tr>
<td>8</td>
<td>0x110</td>
<td>0x114</td>
</tr>
<tr>
<td>4</td>
<td>0x104</td>
<td>0x108</td>
</tr>
</tbody>
</table>

Rtn adr

| 0 | 0x104 |
| 0x100 |

%ebp

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

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

| %eax | 456    |
| %edx | 0x124  |
| %ecx | 0x120  |
| %ebx | 123    |
| %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
```
Indexed Addressing Modes

Most General Form

\[ D(R_b, R_i, S) \quad \text{Mem}[\text{Reg}[R_b]+S\times\text{Reg}[R_i]+D] \]

- **D**: Constant “displacement” 1, 2, or 4 bytes
- **R_b**: Base register: Any of 8 integer registers
- **R_i**: Index register: Any, except for \( %\text{esp} \)
  - Unlikely you’d use \( %\text{ebp} \), either
- **S**: Scale: 1, 2, 4, or 8

Special Cases

\( (R_b, R_i) \quad \text{Mem}[\text{Reg}[R_b]+\text{Reg}[R_i]] \)

\( D(R_b, R_i) \quad \text{Mem}[\text{Reg}[R_b]+\text{Reg}[R_i]+D] \)

\( (R_b, R_i, S) \quad \text{Mem}[\text{Reg}[R_b]+S\times\text{Reg}[R_i]] \)
# Address Computation Examples

<table>
<thead>
<tr>
<th>%edx</th>
<th>0xf000</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>0x100</td>
</tr>
</tbody>
</table>

<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><strong>addl</strong></td>
<td><strong>Dest = Dest + Src</strong></td>
</tr>
<tr>
<td><strong>subl</strong></td>
<td><strong>Dest = Dest - Src</strong></td>
</tr>
<tr>
<td><strong>imull</strong></td>
<td><strong>Dest = Dest * Src</strong></td>
</tr>
<tr>
<td><strong>sall</strong></td>
<td><strong>Dest = Dest &lt;&lt; Src</strong> Also called <strong>shll</strong></td>
</tr>
<tr>
<td><strong>sarl</strong></td>
<td><strong>Dest = Dest &gt;&gt; Src</strong> Arithmetic</td>
</tr>
<tr>
<td><strong>shr1</strong></td>
<td><strong>Dest = Dest &gt;&gt; Src</strong> Logical</td>
</tr>
<tr>
<td><strong>xorl</strong></td>
<td><strong>Dest = Dest ^ Src</strong></td>
</tr>
<tr>
<td><strong>andl</strong></td>
<td><strong>Dest = Dest &amp; Src</strong></td>
</tr>
<tr>
<td><strong>orl</strong></td>
<td>**Dest = Dest</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>$Dest = \sim Dest$</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;
}

arith:
    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)
```
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)
```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)
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;
}
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)
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;
}
```

```
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
```

**Set Up**

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

**Body**

```
movl 8(%ebp),%eax
xorl 12(%ebp),%eax
sarl $17,%eax
andl $8185,%eax
```
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;
}
```

```
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
```

logical:

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

movl 8(%ebp),%eax  ; Set Up
xorl 12(%ebp),%eax
sarl $17,%eax
andl $8185,%eax

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

<table>
<thead>
<tr>
<th>Set Up</th>
<th>Body</th>
<th>Finish</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl 8(%ebp),%eax  ; eax = x</td>
<td>xorl 12(%ebp),%eax  ; eax = x^y (t1)</td>
<td>sarl $17,%eax  ; eax = t1&gt;&gt;17 (t2)</td>
</tr>
</tbody>
</table>
| 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;
}
```

### 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
```

#### 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

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

code:

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 (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

Set Up
Body
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
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

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

- 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

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

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

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

Languages
- http://compsoc.dur.ac.uk/whitespace/
- http://catb.org/~esr/intercal/

Screen shots
- Eclipse shot courtesy of Roy Liu, http://hubris.ucsd.edu