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

Machine-Level Programming I: Introduction
Sept. 14, 2004

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

<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>■ 16-bit processor. Basis for IBM PC &amp; DOS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>■ Limited to 1MB address space. DOS only gives you 640K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80286</td>
<td>1982</td>
<td>134K</td>
</tr>
<tr>
<td>■ Added elaborate, but not very useful, addressing scheme</td>
<td></td>
<td></td>
</tr>
<tr>
<td>■ Basis for IBM PC-AT and Windows</td>
<td></td>
<td></td>
</tr>
<tr>
<td>386</td>
<td>1985</td>
<td>275K</td>
</tr>
<tr>
<td>■ Extended to 32 bits. Added “flat addressing”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>■ Capable of running Unix</td>
<td></td>
<td></td>
</tr>
<tr>
<td>■ Linux/gcc uses no instructions introduced in later models</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
x86 Evolution: Programmer’s View

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
</tr>
</thead>
<tbody>
<tr>
<td>486</td>
<td>1989</td>
<td>1.9M</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>3.1M</td>
</tr>
<tr>
<td>Pentium/MMX</td>
<td>1997</td>
<td>4.5M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added special collection of instructions for operating on 64-bit vectors of 1, 2, or 4 byte integer data</td>
</tr>
<tr>
<td>PentiumPro</td>
<td>1995</td>
<td>6.5M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added conditional move instructions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Big change in underlying microarchitecture</td>
</tr>
</tbody>
</table>
# x86 Evolution: Programmer’s View

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium III</td>
<td>1999</td>
<td>8.2M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Added “streaming SIMD” instructions for operating on 128-bit vectors of 1, 2, or 4 byte integer or floating point data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Our Fish machines</td>
</tr>
<tr>
<td>Pentium 4</td>
<td>2001</td>
<td>42M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Added 8-byte formats and 144 new instructions for streaming SIMD mode</td>
</tr>
<tr>
<td>Pentium 4 HT</td>
<td>2002</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Adds “hyperthreading technology,” the ability to simultaneously run to computational threads on a single processor.</td>
</tr>
</tbody>
</table>
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></td>
<td></td>
<td>- Extends to IA64, a 64-bit architecture</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Radically new instruction set designed for high performance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Can run existing IA32 programs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- On-board “x86 engine”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Joint project with Hewlett-Packard</td>
</tr>
<tr>
<td>Itanium 2</td>
<td>2002</td>
<td>221M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Big performance boost</td>
</tr>
</tbody>
</table>

Itanium has not taken off in marketplace
- Lack of backward compatibility
- Disappointing performance
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.
  - Exploited fact that Intel distracted by IA64
  - Now are close competitors to Intel

- Developing x86-64, its own extension to 64 bits
  - This starts to eat 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

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
- Will be available on high-end Pentium 4s
- Almost identical to x86-64!
Assembly Programmer’s View

Programmer-Visible State
- **EIP**  Program Counter
  - Address of next instruction
- **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)
```

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

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

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

```
Executable program (p)
```

- Compiler (`gcc -S`)
- Assembler (`gcc` or `as`)
- Linker (`gcc` or `ld`)
Compiling Into Assembly

C Code

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

Generated 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>`:
- 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

```
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

00401040 <_sum>:

- 55  push %ebp
- 89 e5  mov %esp,%ebp
- 8b 45 0c  mov 0xc(%ebp),%eax
- 03 45 08  add 0x8(%ebp),%eax
- 89 ec  mov %ebp,%esp
- 5d  pop %ebp
- c3  ret
- 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

Object

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
x/13b sum
```
- Disassemble procedure
- Examine the 13 bytes starting at sum
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

Moving Data

\texttt{movl \textit{Source,Dest}}:
\begin{itemize}
  \item Move 4-byte ("long") word
  \item Lots of these in typical code
\end{itemize}

Operand Types

\begin{itemize}
  \item Immediate: Constant integer data
    \begin{itemize}
      \item Like C constant, but prefixed with ‘$’
      \item E.g., $0x400, $-533$
      \item Encoded with 1, 2, or 4 bytes
    \end{itemize}
  \item Register: One of 8 integer registers
    \begin{itemize}
      \item But \%esp and \%ebp reserved for special use
      \item Others have special uses for particular instructions
    \end{itemize}
  \item Memory: 4 consecutive bytes of memory
    \begin{itemize}
      \item Various "address modes"
    \end{itemize}
\end{itemize}
### 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>Imm</strong></td>
<td><strong>Reg</strong></td>
<td>movl $0x4,%eax</td>
<td>temp = 0x4;</td>
</tr>
<tr>
<td></td>
<td><strong>Mem</strong></td>
<td>movl $-147,(%eax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td><strong>Reg</strong></td>
<td><strong>Reg</strong></td>
<td>movl %eax,%edx</td>
<td>temp2 = temp1;</td>
</tr>
<tr>
<td></td>
<td><strong>Mem</strong></td>
<td>movl %eax,(%edx)</td>
<td>*p = temp;</td>
</tr>
<tr>
<td><strong>Mem</strong></td>
<td><strong>Reg</strong></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

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
```

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

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
```

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

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

Stack

Register | Variable
---|---
%ecx | yp
%edx | xp
%eax | t1
%ebx | 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
```

Address Table:

<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>0x108</td>
</tr>
<tr>
<td>0</td>
<td>0x104</td>
</tr>
<tr>
<td>-4</td>
<td>0x100</td>
</tr>
</tbody>
</table>

Variables:

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

Rtn adr:

0x100
Understanding Swap

| %eax | 0x120 |
| %edx |       |
| %ecx | 0x120 |
| %ebx |       |
| %esi |       |
| %edi |       |
| %esp | 0x104 |
| %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

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

Address
| 123  | 0x124 |
| 456  | 0x120 |
| 0x11c |      |
| 0x118 |      |
| 0x114 |      |
| 0x110 |      |
| 0x120 |      |
| 0x124 |      |
| 0x10c |      |
| 0x108 |      |
| 0x104 |      |
| 0x100 |      |
Understanding Swap

<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>0x120</td>
</tr>
<tr>
<td>%ebx</td>
<td></td>
</tr>
<tr>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td>0x120</td>
</tr>
<tr>
<td>%ebp</td>
<td></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>
<tr>
<td>%esp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

The code snippet shows the swap operation:

```
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

<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>

```
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

\[ D(Rb, Ri, S) \quad Mem[Reg[Rb]+S*Reg[Ri]+ D] \]

- **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 Mem[Reg[Rb]+Reg[Ri]]
- \(D(Rb, Ri)\) \quad Mem[Reg[Rb]+Reg[Ri]+D]
- \((Rb, Ri, S)\) \quad 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><code>0x8(%edx)</code></td>
<td><code>0xf000 + 0x8</code></td>
<td><code>0xf008</code></td>
</tr>
<tr>
<td><code>(%edx,%ecx)</code></td>
<td><code>0xf000 + 0x100</code></td>
<td><code>0xf100</code></td>
</tr>
<tr>
<td><code>(%edx,%ecx,4)</code></td>
<td><code>0xf000 + 4*0x100</code></td>
<td><code>0xf400</code></td>
</tr>
<tr>
<td><code>0x80(%edx,2)</code></td>
<td><code>2*0xf000 + 0x80</code></td>
<td><code>0x1e080</code></td>
</tr>
</tbody>
</table>

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

\textbf{leal} \textit{Src, Dest}

- \textit{Src} is address mode expression
- Set \textit{Dest} to address denoted by expression

\textbf{Uses}

- Computing addresses without a memory reference
  - E.g., translation of \texttt{p = \&x[i];}
- Computing arithmetic expressions of the form \texttt{x + k*y}
  - \texttt{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</td>
<td>$Dest = Dest + Src$</td>
</tr>
<tr>
<td>subl</td>
<td>$Dest = Dest - Src$</td>
</tr>
<tr>
<td>imull</td>
<td>$Dest = Dest \times Src$</td>
</tr>
<tr>
<td>sall</td>
<td>$Dest = Dest \ll Src$</td>
</tr>
<tr>
<td>sarl</td>
<td>$Dest = Dest \gg Src$</td>
</tr>
<tr>
<td>shrl</td>
<td>$Dest = Dest \ggg Src$</td>
</tr>
<tr>
<td>xorl</td>
<td>$Dest = Dest \oplus Src$</td>
</tr>
<tr>
<td>andl</td>
<td>$Dest = Dest &amp; Src$</td>
</tr>
<tr>
<td>orl</td>
<td>$Dest = Dest \mid Src$</td>
</tr>
</tbody>
</table>

Also called **shll**
# 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><code>incl Dest</code></td>
<td>$Dest = Dest + 1$</td>
</tr>
<tr>
<td><code>decl Dest</code></td>
<td>$Dest = Dest - 1$</td>
</tr>
<tr>
<td><code>negl Dest</code></td>
<td>$Dest = - Dest$</td>
</tr>
<tr>
<td><code>notl Dest</code></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:

```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
```
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
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;
}
```

```
Understanding arith

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;
}

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  # edx = 48*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)
### 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;
}
```

### Stack

<table>
<thead>
<tr>
<th>Offset</th>
<th>Variable</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>

Usage:
- `%ebp` is the offset from the stack pointer to the return address.
- `%ebp` is the offset from the stack pointer to the old `%ebp`.

### Assembly Code

```assembly
    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)
```

---

15-213, F'04
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)
```
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
sar1 $17,%eax         ; eax = t1>>17
and1 $8185,%eax       ; eax = t2 & 8185
```

logical:
```
pushl %ebp
movl %esp,%ebp
movl 8(%ebp),%eax
xorl 12(%ebp),%eax
sar1 $17,%eax
and1 $8185,%eax
movl %ebp,%esp
popl %ebp
ret
```

Set Up

Body

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

- Set Up
- Body
- Finish

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

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:

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

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)
CISC Properties

Instruction can reference different operand types
  ■ Immediate, register, memory

Arithmetic operations can read/write memory

Memory reference can involve complex computation
  ■ Rb + S*Ri + D
  ■ Useful for arithmetic expressions, too

Instructions can have varying lengths
  ■ IA32 instructions can range from 1 to 15 bytes
Summary: Abstract Machines

Machine Models

- **C**
  - mem
  - proc

Data

1) char
2) int, float
3) double
4) struct, array
5) pointer

Control

1) loops
2) conditionals
3) switch
4) Proc. call
5) Proc. return

Assembly

- mem
- Stack
- regs
- Cond. Codes
- processor
- alu

1) byte
2) 2-byte word
3) 4-byte long word
4) contiguous byte allocation
5) address of initial byte

3) branch/jump
4) call
5) ret
Pentium Pro (P6)

History
- Announced in Feb. ‘95
- Basis for Pentium II, Pentium III
- Pentium 4 similar idea, but different details

Features
- Dynamically translates instructions to more regular format
  - Very wide, but simple instructions
- Executes operations in parallel
  - Up to 5 at once
- Very deep pipeline
  - 12–18 cycle latency
Pentium Pro Block Diagram
PentiumPro Operation

Translates instructions dynamically into “Uops”
- 118 bits wide
- Holds operation, two sources, and destination

Executes Uops with “Out of Order” engine
- Uop executed when
  - Operands available
  - Functional unit available
- Execution controlled by “Reservation Stations”
  - Keeps track of data dependencies between uops
  - Allocates resources

Consequences
- Indirect relationship between IA32 code & what actually gets executed
- Tricky to predict / optimize performance at assembly level
**Whose Assembler?**

### Intel/Microsoft Format

- `lea  eax, [ecx+ecx*2]`
- `sub  esp, 8`
- `cmp  dword ptr [ebp-8], 0`
- `mov  eax, dword ptr [eax*4+100h]`

### GAS/Gnu Format

- `leal (%ecx,%ecx,2),%eax`
- `subl $8,%esp`
- `cmpl $0,-8(%ebp)`
- `movl $0x100(,%eax,4),%eax`

### Intel/Microsoft Differs from GAS

- **Operands listed in opposite order**
  - `mov Dest, Src`
  - `movl Src, Dest`

- **Constants not preceded by ‘$’, Denote hex with ‘h’ at end**
  - `100h`
  - `$0x100`

- **Operand size indicated by operands rather than operator suffix**
  - `sub`
  - `subl`

- **Addressing format shows effective address computation**
  - `[eax*4+100h]`
  - `$0x100(,%eax,4)`