Machine-Level Programming I: Introduction
Sept. 10, 2002

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

Machine-Level Programming I: Introduction
Sept. 10, 2002

Topics

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

class05.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

<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></td>
</tr>
<tr>
<td><strong>PentiumPro</strong></td>
<td>1995</td>
<td>6.5M</td>
</tr>
<tr>
<td></td>
<td></td>
<td></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></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></td>
</tr>
<tr>
<td>Added “streaming SIMD” instructions for operating on 128-bit vectors of 1, 2, or 4 byte integer or floating point data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Our fish machines</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium 4</td>
<td>2001</td>
<td>42M</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Added 8-byte formats and 144 new instructions for streaming SIMD mode</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.
  - Exploited fact that Intel distracted by IA64
  - Now are close competitors to Intel

- Developing own extension to 64-bits
X86 Evolution: Clones

Transmeta

- Recent start-up
  - Employer of Linus Torvalds
- Radically different approach to implementation
  - Translates x86 code into “Very Long Instruction Word” (VLIW) code
  - High degree of parallelism
- Shooting for low-power market
### 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></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>Will be able to 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></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Big performance boost</td>
</tr>
</tbody>
</table>
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)  
Compiler (gcc -S)

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

Object program (p1.o p2.o)  
Static libraries (.a)

Executable program (p)  
Linker (gcc or ld)
```

- Text
- Binary

- Compiler (`gcc -S`)
- Assembler (`gcc` or `as`)
- Linker (`gcc` or `ld`)
- Static libraries (.a)
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>:

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

```c
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

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

Similar to expression

x += y
Disassembling Object Code

Disassembled

<table>
<thead>
<tr>
<th>Address</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00401040</td>
<td><code>_sum</code>:</td>
<td>push %ebp</td>
</tr>
<tr>
<td>0:</td>
<td>55</td>
<td></td>
</tr>
<tr>
<td>1:</td>
<td>89 e5</td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td>3:</td>
<td>8b 45 0c</td>
<td>mov 0xc(%ebp),%eax</td>
</tr>
<tr>
<td>6:</td>
<td>03 45 08</td>
<td>add 0x8(%ebp),%eax</td>
</tr>
<tr>
<td>9:</td>
<td>89 ec</td>
<td>mov %ebp,%esp</td>
</tr>
<tr>
<td>b:</td>
<td>5d</td>
<td>pop %ebp</td>
</tr>
<tr>
<td>c:</td>
<td>c3</td>
<td>ret</td>
</tr>
<tr>
<td>d:</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

Object

0x401040: 0x55
0x401041: 0x89 0xe5
0x401043: 0x8b 0x45 0x0c
0x401046: 0x03
0x401049: 0x45 0x08
0x40104b: 0x89 0xec
0x40104c: 0x5d
0x40104d: 0xc3

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 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

moving

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

- Cannot do memory-memory transfers with single instruction
Simple Addressing Modes

Normal \( (R) \) \( \text{Mem}[\text{Reg}[R]] \)
- Register \( R \) specifies memory address
  
  \text{movl} \ (\%\text{ecx}),\%\text{eax}

Displacement \( D(R) \) \( \text{Mem}[\text{Reg}[R]+D] \)
- Register \( R \) specifies start of memory region
- Constant displacement \( D \) specifies offset
  
  \text{movl} \ 8(\%\text{ebp}),\%\text{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 -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
### Understanding Swap

#### Variables
- `%eax`
- `%edx`
- `%ecx`
- `%ebx`
- `%esi`
- `%edi`
- `%esp`
- `%ebp`

#### Address Table

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x100</td>
</tr>
<tr>
<td>-4</td>
<td>0x104</td>
</tr>
<tr>
<td>4</td>
<td>0x108</td>
</tr>
<tr>
<td>8</td>
<td>0x110</td>
</tr>
<tr>
<td>12</td>
<td>0x114</td>
</tr>
<tr>
<td></td>
<td>0x118</td>
</tr>
<tr>
<td></td>
<td>0x120</td>
</tr>
<tr>
<td></td>
<td>0x124</td>
</tr>
</tbody>
</table>

#### Instructions
- `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>%eax</th>
<th>Address</th>
<th>Offset</th>
<th>%ebp</th>
<th>Rtn adr</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
<td>0x124</td>
<td>0x120</td>
<td>0x124</td>
<td>0x124</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x120</td>
<td>0x11c</td>
<td>0x120</td>
<td>0x120</td>
</tr>
<tr>
<td>%ebx</td>
<td>0x110</td>
<td>0x114</td>
<td>0x124</td>
<td>0x124</td>
</tr>
<tr>
<td>%esi</td>
<td>0x114</td>
<td>0x118</td>
<td>0x124</td>
<td>0x124</td>
</tr>
<tr>
<td>%edi</td>
<td>0x118</td>
<td>0x11c</td>
<td>0x124</td>
<td>0x124</td>
</tr>
<tr>
<td>%esp</td>
<td>0x11c</td>
<td>0x120</td>
<td>0x120</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
```
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
```

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp</td>
<td>12</td>
</tr>
<tr>
<td>xp</td>
<td>8</td>
</tr>
<tr>
<td>Return Address</td>
<td>4</td>
</tr>
</tbody>
</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>0x11c</td>
</tr>
<tr>
<td>0</td>
<td>0x118</td>
</tr>
<tr>
<td>-4</td>
<td>0x114</td>
</tr>
</tbody>
</table>

<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>0x10c</td>
</tr>
<tr>
<td>0x108</td>
</tr>
<tr>
<td>0x104</td>
</tr>
<tr>
<td>0x100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reg</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></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>
### Understanding Swap

<table>
<thead>
<tr>
<th>%eax</th>
<th>456</th>
</tr>
</thead>
<tbody>
<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>0x104</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

#### Address Table

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>YP 12</td>
<td>0x120</td>
</tr>
<tr>
<td>xp 8</td>
<td>0x124</td>
</tr>
<tr>
<td>4</td>
<td>0x110</td>
</tr>
<tr>
<td>Rtn adr</td>
<td>0x108</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
<tr>
<td></td>
<td>0x100</td>
</tr>
</tbody>
</table>

#### 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

| %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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp</td>
<td>12</td>
</tr>
<tr>
<td>xp</td>
<td>8</td>
</tr>
<tr>
<td>Rtn adr</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0x120</td>
</tr>
<tr>
<td>-4</td>
<td>0x110</td>
</tr>
<tr>
<td></td>
<td>0x114</td>
</tr>
<tr>
<td></td>
<td>0x118</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>
# 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>

```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
```

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp</td>
<td>0x120</td>
</tr>
<tr>
<td>xp</td>
<td>0x124</td>
</tr>
<tr>
<td>4</td>
<td>0x11c</td>
</tr>
<tr>
<td>8</td>
<td>0x110</td>
</tr>
<tr>
<td>12</td>
<td>0x120</td>
</tr>
<tr>
<td></td>
<td>0x114</td>
</tr>
<tr>
<td></td>
<td>0x118</td>
</tr>
<tr>
<td>Rtn adr</td>
<td>0x108</td>
</tr>
<tr>
<td></td>
<td>0x104</td>
</tr>
<tr>
<td></td>
<td>0x100</td>
</tr>
</tbody>
</table>

## Offset 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>0x11c</td>
</tr>
<tr>
<td></td>
<td>0x110</td>
</tr>
<tr>
<td></td>
<td>0x114</td>
</tr>
<tr>
<td></td>
<td>0x118</td>
</tr>
</tbody>
</table>
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 Instruction**

*leal Src, Dest*

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

**Uses**

- Computing address without doing 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>
</tbody>
</table>
| `addl  
\text{Src,Dest}` | \text{Dest} = \text{Dest} + \text{Src} |
| `subl  
\text{Src,Dest}` | \text{Dest} = \text{Dest} - \text{Src} |
| `imull  
\text{Src,Dest}` | \text{Dest} = \text{Dest} * \text{Src} |
| `sall  
\text{Src,Dest}` | \text{Dest} = \text{Dest} \ll \text{Src} Also called \text{shll} |
| `sarl  
\text{Src,Dest}` | \text{Dest} = \text{Dest} \gg \text{Src} Arithmetic |
| `shrl  
\text{Src,Dest}` | \text{Dest} = \text{Dest} \gg \text{Src} Logical |
| `xorl  
\text{Src,Dest}` | \text{Dest} = \text{Dest} \oplus \text{Src} |
| `andl  
\text{Src,Dest}` | \text{Dest} = \text{Dest} \& \text{Src} |
| `orl   
\text{Src,Dest}` | \text{Dest} = \text{Dest} \mid \text{Src} |
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;
}
```

```assembly
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
```

**Set Up**

**Body**

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

2\(^{13}\) = 8192, 2\(^{13}\) – 7 = 8185

```
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:

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

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

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

Set Up

Body

Finish
CISC Properties

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

Arithmetic operations can read/write memory

Memory reference can involve complex computation
- $R_b + S \times R_i + D$
- 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) goto
- 4) Proc. call
- 5) Proc. return

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

- 1) byte
- 2) 4-byte long word
- 3) branch/jump
- 4) call
- 5) ret
- 4) contiguous byte allocation
- 5) address of initial byte
Pentium Pro (P6)

History
- Announced in Feb. ‘95
- Basis for Pentium II, Pentium III, and Celeron processors
- 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

Microprocessor Report 2/16/95
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**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lea eax, [ecx+ecx*2]</td>
<td>Loads eax with the address of ecx+ecx*2</td>
</tr>
<tr>
<td>sub esp, 8</td>
<td>Subtracts 8 from esp</td>
</tr>
<tr>
<td>cmp dword ptr [ebp-8], 0</td>
<td>Compares eax with the value at [ebp-8]</td>
</tr>
<tr>
<td>mov eax, dword ptr [eax*4+100h]</td>
<td>Moves the value at eax*4+100h into eax</td>
</tr>
</tbody>
</table>

**GAS/Gnu Format**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>leal (%ecx,%ecx,2), %eax</td>
<td>Loads %eax with the address of %ecx+2*%ecx</td>
</tr>
<tr>
<td>subl $8,%esp</td>
<td>Subtracts 8 from esp</td>
</tr>
<tr>
<td>cmpl $0,-8(%ebp)</td>
<td>Compares eax with the value at -8(%ebp)</td>
</tr>
<tr>
<td>movl $0x100(,%eax,4),%eax</td>
<td>Moves the value at 0x100+4*eax into eax</td>
</tr>
</tbody>
</table>

**Intel/Microsoft Differs from GAS**

1. **Operands listed in opposite order**
   - Intel/Microsoft: `mov Dest, Src`
   - GAS/Gnu: `movl Src, Dest`

2. **Constants not preceded by ‘$’, Denote hex with ‘h’ at end**
   - Intel/Microsoft: `100h`
   - GAS/Gnu: `$0x100`

3. **Operand size indicated by operands rather than operator suffix**
   - Intel/Microsoft: `sub`
   - GAS/Gnu: `subl`

4. **Addressing format shows effective address computation**
   - Intel/Microsoft: `[eax*4+100h]`
   - GAS/Gnu: `$0x100(,%eax,4)`