Machine-Level Programming I: Introduction

Dr. Steve Goddard
goddard@cse.unl.edu

http://cse.unl.edu/~goddard/Courses/CSCE230J

Giving credit where credit is due

- Most of slides for this lecture are based on slides created by Drs. Bryant and O’Hallaron, Carnegie Mellon University.
- I have modified them and added new slides.
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></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>80286</td>
<td>1982</td>
<td>134K</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added elaborate, but not very useful, addressing scheme</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Basis for IBM PC-AT and Windows</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>Linux/gcc uses no instructions introduced in later models</td>
</tr>
</tbody>
</table>

<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>
</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>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>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

```
text
  C program (p1.c p2.c)
  Asm program (p1.s p2.s)

binary
  Object program (p1.o p2.o)
  Executable program (p)
```

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`

```plaintext
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;
```

Assembly

- Add two signed integers

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

Similar to expression

```
x += y
```

Object Code

- 3-byte instruction
- Stored at address `0x401046`
Disassembling Object Code

Disassembled

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00401040</td>
<td><code>&lt;_sum&gt;</code>:</td>
<td>push %ebp</td>
</tr>
<tr>
<td>0:</td>
<td>55</td>
<td>push %ebp</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

<table>
<thead>
<tr>
<th>Address</th>
<th>Disassembled</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401040:</td>
<td>push %ebp</td>
</tr>
<tr>
<td>0x55</td>
<td></td>
</tr>
<tr>
<td>0x89</td>
<td></td>
</tr>
<tr>
<td>0xe5</td>
<td></td>
</tr>
<tr>
<td>0x8b</td>
<td></td>
</tr>
<tr>
<td>0x45</td>
<td></td>
</tr>
<tr>
<td>0x0c</td>
<td></td>
</tr>
<tr>
<td>0x03</td>
<td></td>
</tr>
<tr>
<td>0x45</td>
<td></td>
</tr>
<tr>
<td>0x08</td>
<td></td>
</tr>
<tr>
<td>0x89</td>
<td></td>
</tr>
<tr>
<td>0xec</td>
<td></td>
</tr>
<tr>
<td>0x5d</td>
<td></td>
</tr>
<tr>
<td>0xc3</td>
<td></td>
</tr>
</tbody>
</table>

Within gdb Debugger

`gdb p disassemble <sum>`
- Disassemble procedure
`x/13b <sum>`
- 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 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>Imm</strong></td>
<td>Reg</td>
<td>movl $0x4, %eax</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl $-147, (%eax)</td>
</tr>
<tr>
<td>Reg</td>
<td>Reg</td>
<td>movl %eax, %edx</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl %eax, (%edx)</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl (%eax), %edx</td>
</tr>
</tbody>
</table>

- Cannot do memory-memory transfers with 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
**Example**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0xFF</td>
</tr>
<tr>
<td>0x104</td>
<td>0x00</td>
</tr>
<tr>
<td>0x108</td>
<td>0x13</td>
</tr>
<tr>
<td>0x10C</td>
<td>0x11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x100</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operand</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>0x1</td>
</tr>
<tr>
<td>(%eax)</td>
<td>0xFF</td>
</tr>
<tr>
<td>8(%eax)</td>
<td>0x13</td>
</tr>
<tr>
<td>263(%ecx)</td>
<td>0x13</td>
</tr>
</tbody>
</table>

**Exercise**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0x0</td>
</tr>
<tr>
<td>0x104</td>
<td>0x1</td>
</tr>
<tr>
<td>0x108</td>
<td>0x2</td>
</tr>
<tr>
<td>0x10C</td>
<td>0x3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x104</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operand</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x104</td>
</tr>
<tr>
<td>(%ecx)</td>
<td>0x0</td>
</tr>
<tr>
<td>4(%eax)</td>
<td>0x2</td>
</tr>
<tr>
<td>0xC(%ecx)</td>
<td>0x3</td>
</tr>
</tbody>
</table>
Using Simple Addressing Modes

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

Understanding Swap

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

### Swap:

- Stack
  - Offset
  - 12: yp
  - 8: xp
  - 4: Rtn adr
  - 0: Old %ebp
  - -4: Old %ebx

- Register Variable
  - %ecx: yp
  - %edx: xp
  - %eax: t1
  - %ebx: t0

Set Up

- pushl %ebp
- movl %esp, %ebp
- pushl %ebx

Body

- movl 12(%ebp), %ecx
- movl 8(%ebp), %edx
- movl (%ecx), %eax
- movl (%edx), %ebx
- movl %eax, (%edx)
- movl %ebx, (%ecx)

Finish

- movl -4(%ebp), %ebx
- movl %ebp, %esp
- popl %ebp
- ret
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
123 0x124
456 0x120
0x11c 0x118
0x114 0x110
0x10c 0x108
0x104 0x100
0x100

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

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

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

Address

Offset

0x124
0x120
0x11c
0x118
0x114
0x110
0x10c
0x108
0x104
0x100

%%
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

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>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></td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
</tbody>
</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>Rtn adr</td>
</tr>
<tr>
<td>0</td>
<td>0x104</td>
</tr>
<tr>
<td>-4</td>
<td>0x100</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**

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

- **D:** Constant “displacement” 1, 2, or 4 bytes
- **Rb:** Base register: Any of 8 integer registers
- **Ri:** Index register: Any, except for %esp
  - Unless 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\cdot\text{Reg[Ri]}]}
Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8(%edx)</td>
<td>0xf000 + 0x8</td>
<td>0xf008</td>
</tr>
<tr>
<td>(%edx,%ecx)</td>
<td>0xf000 + 0x100</td>
<td>0xf100</td>
</tr>
<tr>
<td>(%edx,%ecx,4)</td>
<td>0xf000 + 4*0x100</td>
<td>0xf400</td>
</tr>
<tr>
<td>0x80,(%edx,2)</td>
<td>2*0xf000 + 0x80</td>
<td>0x1e080</td>
</tr>
</tbody>
</table>

Another Example

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0xFF</td>
</tr>
<tr>
<td>0x104</td>
<td>0x00</td>
</tr>
<tr>
<td>0x108</td>
<td>0x13</td>
</tr>
<tr>
<td>0x10C</td>
<td>0x11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x100</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x1</td>
</tr>
<tr>
<td>%edx</td>
<td>0x3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operand</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>9(%eax,%edx)</td>
<td>0x11</td>
</tr>
<tr>
<td>0x100(%ecx,4)</td>
<td>0x00</td>
</tr>
<tr>
<td>(%eax,%edx,4)</td>
<td>0x11</td>
</tr>
<tr>
<td>252(%ecx,%edx)</td>
<td>0xFF</td>
</tr>
</tbody>
</table>
Exercise

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0xFF</td>
</tr>
<tr>
<td>0x104</td>
<td>0x00</td>
</tr>
<tr>
<td>0x108</td>
<td>0x13</td>
</tr>
<tr>
<td>0x10C</td>
<td>0x11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x100</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x104</td>
</tr>
<tr>
<td>%edx</td>
<td>0x1</td>
</tr>
<tr>
<td>%ebx</td>
<td>0x8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operand</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>3(%eax,%edx)</td>
<td>0x00</td>
</tr>
<tr>
<td>254(%edx,2)</td>
<td>0xFF</td>
</tr>
<tr>
<td>(%eax, %edx, 4)</td>
<td>0x00</td>
</tr>
<tr>
<td>(%ecx,%ebx)</td>
<td>0x11</td>
</tr>
</tbody>
</table>

More on Data Movement

MOVW and MOVB

MOVW moves two bytes, when one of its operands is a register, it must be one of the 8 two-byte registers
e.g. MOVW %ax, %dx

MOVB moves a single byte, when one of its operands is a register, it must be one of the 8 single-byte registers
e.g. MOVB %al, %ah
More on Data Movement

**MOVBL and MOVZBL**

- MOVBL sign-extends a single byte, and copies it into a double-word destination
- MOVZBL expands a single byte to 32 bits with 24 leading zeros, and copies it into a double-word destination

**Example:**

```bash
%eax = 0x12345678
%edx = 0xAAAABBBB

MOVBL %dh, %eax %eax = 0x123456BB
MOVZBL %dh, %eax %eax = 0xFFFFFFFF
```

**Exercise**

```bash
%eax = 0x12345678
%edx = 0xAAAA22CC

MOVBL %dh, %eax %eax = #2
MOVZBL %dh, %eax %eax = #3
MOVZBL %dh, %eax %eax = #3
MOVZBL %dl, %eax %eax = #5
```

1. 0x12345622
2. 0x12342278
3. 0x00000022
4. 0xFFFFFFFF
5. 0xFFFFFFFFC
Address Computation Instruction

**lea** <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.

---

**Example**

Assume register `%eax` holds value *X*  
%`ecx` holds value *Y*

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result in <code>%edx</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lea 8(%eax, %edx)</code></td>
<td>X+8</td>
</tr>
<tr>
<td><code>lea (%eax,%ecx), %edx</code></td>
<td>X+Y</td>
</tr>
<tr>
<td><code>lea 8(%eax,%ecx), %edx</code></td>
<td>X+Y+8</td>
</tr>
<tr>
<td><code>lea 8(%eax,%eax,4), %edx</code></td>
<td>5X+8</td>
</tr>
<tr>
<td><code>lea 8(%eax,%ecx,2), %edx</code></td>
<td>X+2Y+8</td>
</tr>
</tbody>
</table>
### Some Arithmetic Operations

#### Format

#### Computation

**Two Operand Instructions**

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

**One Operand Instructions**

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

```c
int arith
  (int x, int y, int z)
  {
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
  }
```

arith:
  pushl %ebp
  movl %esp,%ebp
  movl 8(%ebp),%eax
  movl 12(%ebp),%edx
  leal (%edx,%eax),%ecx
  leal (%edx,%edx,2),%edx
  sal $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
  sal $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;
}
```

```c
# 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;
}
```

```c
logical:
pushl %ebp
movl %esp, %ebp
set up
movl 8(%ebp), %eax
xorl 12(%ebp), %eax
sarl $17, %eax
andl $8185, %eax
body
movl %ebp, %esp
popl %ebp
finish
ret
```

\(2^{13} = 8192, 2^{13} - 7 = 8185\)
**Push and Pop**

PUSHL takes a single operand: the data source, and store it to the top of stack.

For example,

PUSHL %eax has equivalent behavior as

```
subl $4, %esp ; stack grows downward
movl %eax, (%esp)
```

POPL takes the data destination, and pop the top element of stack onto the destination.

POPL %eax has equivalent behavior as

```
movl (%esp), %eax
addl $4, %esp
```

---

**CISC Properties**

Instruction can reference different operand types

- Immediate, register, memory

Arithmetic operations can read/write memory

Memory reference can involve complex computation

- \( R_b + S \cdot R_i + D \)
- 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

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

1) byte
2) 2-byte word
3) 4-byte long word
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
PentiumPro 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?

<table>
<thead>
<tr>
<th>Intel/Microsoft Format</th>
<th>GAS/Gnu Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>lea eax, [ecx+ecx*2]</td>
<td>leal (%ecx, %ecx, 2), %eax</td>
</tr>
<tr>
<td>sub esp, 8</td>
<td>subl $8, %esp</td>
</tr>
<tr>
<td>cmp dword ptr [ebp-8], 0</td>
<td>cmpl $0, -8(%ebp)</td>
</tr>
<tr>
<td>mov eax, dword ptr [eax*4+100h]</td>
<td>movl $0x100(, %eax, 4), %eax</td>
</tr>
</tbody>
</table>

**Intel/Microsoft Differ 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)