Machine-Level Programming I: Introduction
Sept. 16, 2009

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!
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
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
Abstract Machines

Machine Models

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

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

Control
1) branch/jump
2) call
3) ret
Whose Assembler?

Intel/Microsoft Format | GAS/Gnu Format
--- | ---
lea eax, [ecx+ecx*2] | leal (%ecx,%ecx,2),%eax
sub esp, 8 | subl $8,%esp
cmp dword ptr [ebp-8], 0 | cmpl $0,-8(%ebp)
mov eax, dword ptr [eax*4+100h] | 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)
Turning C into Object Code

- Code in files: `p1.c` `p2.c`
- Compile with command:
  - Use optimizations (`-O`)
  - Put resulting binary in file `p`
  
  ```
  gcc -O p1.c p2.c -o p
  ```

![Diagram showing the process of compiling C programs into an executable program.]

- **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`) -> **Linker** (`gcc` or `ld`) -> **Executable program** (`p`)
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`

- 8 -
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
  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

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

- Add two signed integers

Assembly

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

- 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`
Disassembling Object Code

Sum Disassembled

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>00401040</td>
<td><code>_sum</code></td>
<td></td>
</tr>
<tr>
<td>0:</td>
<td>55</td>
<td><code>push %ebp</code></td>
</tr>
<tr>
<td>1:</td>
<td>89 e5</td>
<td><code>mov %esp,%ebp</code></td>
</tr>
<tr>
<td>3:</td>
<td>8b 45 0c</td>
<td><code>mov 0xc(%ebp),%eax</code></td>
</tr>
<tr>
<td>6:</td>
<td>03 45 08</td>
<td><code>add 0x8(%ebp),%eax</code></td>
</tr>
<tr>
<td>9:</td>
<td>89 ec</td>
<td><code>mov %ebp,%esp</code></td>
</tr>
<tr>
<td>b:</td>
<td>5d</td>
<td><code>pop %ebp</code></td>
</tr>
<tr>
<td>c:</td>
<td>c3</td>
<td><code>ret</code></td>
</tr>
<tr>
<td>d:</td>
<td>8d 76 00</td>
<td><code>lea 0x0(%esi),%esi</code></td>
</tr>
</tbody>
</table>

Disassembler

`objdump -d`

- 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

Within gdb Debugger

gdb p
disable 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

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><code>Imm</code></td>
<td><code>Reg</code></td>
<td><code>movl $0x4, %eax</code></td>
</tr>
<tr>
<td></td>
<td><code>Mem</code></td>
<td><code>movl $-147, (%eax)</code></td>
</tr>
<tr>
<td><code>Reg</code></td>
<td><code>Reg</code></td>
<td><code>movl %eax, %edx</code></td>
</tr>
<tr>
<td></td>
<td><code>Mem</code></td>
<td><code>movl %eax, (%edx)</code></td>
</tr>
<tr>
<td><code>Mem</code></td>
<td><code>Reg</code></td>
<td><code>movl (%eax), %edx</code></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
```
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
```
Understanding Swap

```c
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

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

---

<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>0x108</td>
</tr>
<tr>
<td>8</td>
<td>0x110</td>
</tr>
<tr>
<td>12</td>
<td>0x120</td>
</tr>
<tr>
<td>YP</td>
<td>123</td>
</tr>
<tr>
<td>XP</td>
<td>456</td>
</tr>
<tr>
<td>Rtn adr</td>
<td>0x124</td>
</tr>
</tbody>
</table>
Understanding Swap

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x120</td>
</tr>
<tr>
<td>%edx</td>
<td>0x124</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ebx</td>
<td>0x104</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
```
### Understanding Swap

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
<th>YP</th>
<th>XP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
<td>12</td>
<td>123</td>
<td>0x120</td>
</tr>
<tr>
<td>0x120</td>
<td>8</td>
<td>456</td>
<td>0x110</td>
</tr>
<tr>
<td>0x11c</td>
<td>4</td>
<td>0x124</td>
<td>0x10c</td>
</tr>
<tr>
<td>0x118</td>
<td></td>
<td>0x124</td>
<td>0x10c</td>
</tr>
<tr>
<td>0x114</td>
<td></td>
<td>0x120</td>
<td>0x10c</td>
</tr>
<tr>
<td>0x110</td>
<td></td>
<td>0x110</td>
<td>0x10c</td>
</tr>
<tr>
<td>0x108</td>
<td></td>
<td>0x114</td>
<td>0x108</td>
</tr>
<tr>
<td>0x104</td>
<td></td>
<td>0x118</td>
<td>0x104</td>
</tr>
<tr>
<td>0x100</td>
<td></td>
<td>0x11c</td>
<td>0x100</td>
</tr>
</tbody>
</table>

#### Memory Addresses

<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></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 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
```
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

\[
\begin{array}{|c|c|}
\hline
\%eax & 456 \\
\%edx & 0x124 \\
\%ecx & 0x120 \\
\%ebx & 123 \\
\%esi & \\
\%edi & \\
\%esp & \\
\%ebp & 0x104 \\
\hline
\end{array}
\]

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

0x124
0x120
0x11c
0x118
0x114
0x110
0x124
0x10c
0x104
0x108
0x100
Indexed Addressing Modes

Most General Form

\[ D(R_b,R_i,S) \quad Mem[Reg[R_b]+S*Reg[R_i]+D] \]

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

Special Cases

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

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

\[ (R_b,R_i,S) \quad Mem[Reg[R_b]+S*Reg[R_i]] \]
# 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>

%edx 0xf000
%ecx 0x100
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 \( x + k \times y \)
  - \( k = 1, 2, 4, \) or \( 8 \).
## Some Arithmetic Operations

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Two Operand Instructions</strong></td>
<td></td>
</tr>
<tr>
<td>addl $Src, Dest$</td>
<td>$Dest = Dest + Src$</td>
</tr>
<tr>
<td>subl $Src, Dest$</td>
<td>$Dest = Dest - Src$</td>
</tr>
<tr>
<td>imull $Src, Dest$</td>
<td>$Dest = Dest * Src$</td>
</tr>
<tr>
<td>sall $Src, Dest$</td>
<td>$Dest = Dest &lt;&lt; Src$  Also called shll</td>
</tr>
<tr>
<td>sarl $Src, Dest$</td>
<td>$Dest = Dest &gt;&gt; Src$  Arithmetic</td>
</tr>
<tr>
<td>shrl $Src, Dest$</td>
<td>$Dest = Dest &gt;&gt; Src$  Logical</td>
</tr>
<tr>
<td>xorl $Src, Dest$</td>
<td>$Dest = Dest ^ Src$</td>
</tr>
<tr>
<td>andl $Src, Dest$</td>
<td>$Dest = Dest &amp; Src$</td>
</tr>
<tr>
<td>orl $Src, Dest$</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;
}
```

```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
**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)
```

Stack Table:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</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>
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;
}
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;
}
```

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

```c
for (i=0; i<n; i++)
    v += 1;
```

```
gcc -c -O -S foo.c

.text
.globl _foo
_foo:
pushl %ebp
movl %esp, %ebp
subl $24, %esp
movl $0, -20(%ebp)
jmp L2
L3:
    movl -20(%ebp), %eax
    leal -12(%ebp), %edx
    addl %eax, (%edx)
    leal -20(%ebp), %eax
    incl (%eax)
L2:
    movl -20(%ebp), %eax
    cmpl -16(%ebp), %eax
    jl L3
    movl -12(%ebp), %eax
    leave
    ret
```

```
gcc –c -S foo.c

.text
.globl _foo
_foo:
pushl %ebp
movl %esp, %ebp
subl $24, %esp
movl $0, -20(%ebp)
jmp L2
L3:
    movl -20(%ebp), %eax
    leal -12(%ebp), %edx
    addl %eax, (%edx)
    leal -20(%ebp), %eax
    incl (%eax)
L2:
    movl -20(%ebp), %eax
    cmpl -16(%ebp), %eax
    jl L3
    movl -12(%ebp), %eax
    leave
    ret
```
Additional Help

Intel Instruction set online

Useful GDB command if you’ve worked with Intel format
- set assembly-flavor intel
- set assembly-flavor att