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

Machine-Level Programming I:
Introduction
Sept. 09, 2006

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
(Abbreviated)

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086</td>
<td>1978</td>
<td>29K</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16-bit processor. Basis for IBM PC &amp; DOS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Limited to 1MB address space. DOS only gives you 640K</td>
</tr>
<tr>
<td>386</td>
<td>1985</td>
<td>275K</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Extended to 32 bits. Added “flat addressing”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Capable of running Unix</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Referred to as “IA32”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>32-bit Linux/gcc uses no instructions introduced in later models</td>
</tr>
</tbody>
</table>

x86 Evolution: Programmer’s View

Machine Evolution
- 486 1989 1.9M
- Pentium 1993 3.1M
- Pentium/MMX 1997 4.5M
- PentiumPro 1995 6.5M
- Pentium III 1999 8.2M
- Pentium 4 2001 42M

Added Features
- Instructions to support multimedia operations
  - Parallel operations on 1, 2, and 4-byte data, both integer & FP
- Instructions to enable more efficient conditional operations

Linux/GCC Evolution
- None!
New Species: IA64

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

X86 Evolution: Clones

Advanced Micro Devices (AMD)
- Historically
  - AMD has followed just behind Intel
  - A little bit slower, a lot cheaper
- Recently
  - Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies
  - Exploited fact that Intel distracted by IA64
  - Now are close competitors to Intel
- Developed x86-64, its own extension to 64 bits
  - Started eating into Intel’s high-end server market

Intel’s 64-Bit Dilemma

Intel Attempted Radical Shift from IA32 to IA64
- Totally different architecture
- Executes IA32 code only as legacy
- Performance disappointing

AMD Stepped in with Evolutionary Solution
- x86-64 (now called “AMD64”)

Intel Felt Obligated to Focus on IA64
- Hard to admit mistake or that AMD is better

2004: Intel Announces EM64T extension to IA32
- Extended Memory 64-bit Technology
- Almost identical to x86-64!
- Our Saltwater fish machines

Our Coverage

IA32
- The traditional x86

x86-64
- The emerging standard

Presentation
- Book has IA32
- Handout has x86-64
- Lecture will cover both

Labs
- Lab #2 x86-64
- Lab #3 IA32
Assembly Programmer’s View

### Programmer-Visible State
- **PC (Program Counter)**
  - Address of next instruction
  - Called “EIP” (IA32) or “RIP” (x86-64)
- **Register File**
  - Heavily used program data
- **Condition Codes**
  - Store status information about most recent arithmetic operation
  - Used for conditional branching

### Memory
- **Object Code**
- **Program Data**
- **OS Data**

### Stack

### Turning C into Object Code
- **Code in files** `p1.c p2.c`
- **Compile with command:**
  ```bash
gcc -O p1.c p2.c -o p
  ```
  - Use optimizations (`-O`)
  - Put resulting binary in file `p`

#### Text
- **C program** (`p1.c p2.c`)
  ```bash
  Compiler (gcc -S)
  ```
- **Assembler** (`gcc or as`)
  ```bash
  Asm program (p1.s p2.s)
  ```
- **Linker** (`gcc or ld`)
  ```bash
  Object program (p1.o p2.o)
  ```
- **Static libraries** (`.a`)
  ```bash
  Executable program (p)
  ```

### Compiling Into Assembly
**C Code**
```c
int sum(int x, int y)
{
    int t = x+y;
    return t;
}
```

**Generated IA32 Assembly**
```assembly
_sum:  
pushl %ebp
movl %esp,%ebp
movl 12(%ebp),%eax
addl 8(%ebp),%eax
movl %ebp,%esp
popl %ebp
ret
```

#### Obtain with command
```bash
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 \texttt{sum}**

\begin{verbatim}
0x401040 <sum>:
  0x55  push %ebp
  0x89 e5  mov %esp,%ebp
  0x8b
  0x45 0c  mov 0xc(%ebp),%eax
  0x03 0x45 08  add 0x8(%ebp),%eax
  0x89 ec  mov %ebp,%esp
  0x5d  pop %ebp
  0xc3  ret
\end{verbatim}

**Assembler**

- Translates \texttt{.s} into \texttt{.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 \texttt{malloc}, \texttt{printf}
- Some libraries are \textit{dynamically linked}
  - Linking occurs when program begins execution

Machine Instruction Example

**C Code**

\begin{verbatim}
int t = x+y;
\end{verbatim}

**Assembly**

\begin{verbatim}
addl 8(%ebp),%eax
\end{verbatim}

Similar to expression:

\begin{verbatim}
x += y
\end{verbatim}

Or

\begin{verbatim}
int eax;
int *ebp;
eax += ebp[2]
\end{verbatim}

Disassembling Object Code

**Disassembled**

\begin{verbatim}
00401040 <sum>:
  0:  55  push %ebp
  1:  89 e5  mov %esp,%ebp
  3:  8b 45 0c  mov 0xc(%ebp),%eax
  6:  03 45 08  add 0x8(%ebp),%eax
  9:  89 ec  mov %ebp,%esp
  b:  5d  pop %ebp
 c:  c3  ret
d:  8d 76 00  lea 0x0(%esi),%esi
\end{verbatim}

**Disassembler**

\texttt{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 \texttt{a.out} (complete executable) or \texttt{.o} file

Alternate Disassembly

**Object**

\begin{verbatim}
0x401040 <sum>:
  0x55  push %ebp
  0x89 e5  mov %esp,%ebp
  0x8b
  0x45 0c  mov 0xc(%ebp),%eax
  0x03 0x45 08  add 0x8(%ebp),%eax
  0x89 ec  mov %ebp,%esp
  0x5d  pop %ebp
  0xc3  ret
\end{verbatim}

**Disassembled**

\begin{verbatim}
0x401040 <sum>:
  0x55  push %ebp
  0x89 e5  mov %esp,%ebp
  0x8b
  0x45 0c  mov 0xc(%ebp),%eax
  0x03 0x45 08  add 0x8(%ebp),%eax
  0x89 ec  mov %ebp,%esp
  0x5d  pop %ebp
  0xc3  ret
\end{verbatim}

**Within gdb Debugger**

\texttt{gdb p}

- Disassemble procedure

\texttt{x/13b sum}

- Examine the 13 bytes starting at \texttt{sum}
What Can be Disassembled?

Anything that can be interpreted as executable code
Disassembler examines bytes and reconstructs assembly source

Moving Data: IA32

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"

Operand Combinations

Source    Dest    Src, Dest    C Analog

movl

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src, Dest</th>
<th>C Analog</th>
</tr>
</thead>
</table>
| Imm    | Reg  | movl $0x4,%eax | temp = 0x44;
| Mem    | Reg  | movl $-147,(%eax) | *p = -147;
| Mem    | Reg  | movl %eax,%edx | temp2 = temp1;
| Mem    | Reg  | movl (%eax),%edx | temp = *p;

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:

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

Set Up

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

Body

```
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 -4(%ebp),%ebx
movl %ebx,%esp
```

Finish

```
popl %ebp
ret
```

---

Understanding Swap

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

Stack

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

Offset

```
  0x120 0x124 0x128 0x130 0x134 0x138 0x140
```

Register Variable

```
%ecx  yp
%edx  xp
%eax  t1
%ebx  t0
```

Address

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

Offset

```
  yp  12  0x120
  xp  8   0x124
  Rtn adr 4  0x10c
  %ebp 0  0x104
     -4  0x100
```

Body

```
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 -4(%ebp),%ebx
movl %ebx,%esp
```

Finish

```
popl %ebp
ret
```
### Understanding Swap

**Address**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>0x124</td>
</tr>
<tr>
<td>456</td>
<td>0x120</td>
</tr>
<tr>
<td></td>
<td>0x11c</td>
</tr>
<tr>
<td></td>
<td>0x118</td>
</tr>
<tr>
<td></td>
<td>0x114</td>
</tr>
<tr>
<td></td>
<td>0x110</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>

**Offset**

<table>
<thead>
<tr>
<th>yp</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>xp</td>
<td>8</td>
</tr>
<tr>
<td>%ebp</td>
<td>0</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

**Address**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>0x124</td>
</tr>
<tr>
<td>456</td>
<td>0x120</td>
</tr>
<tr>
<td></td>
<td>0x11c</td>
</tr>
<tr>
<td></td>
<td>0x118</td>
</tr>
<tr>
<td></td>
<td>0x114</td>
</tr>
<tr>
<td></td>
<td>0x110</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>

**Offset**

<table>
<thead>
<tr>
<th>yp</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>xp</td>
<td>8</td>
</tr>
<tr>
<td>%ebp</td>
<td>0</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

**Address**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>0x124</td>
</tr>
<tr>
<td>456</td>
<td>0x120</td>
</tr>
<tr>
<td></td>
<td>0x11c</td>
</tr>
<tr>
<td></td>
<td>0x118</td>
</tr>
<tr>
<td></td>
<td>0x114</td>
</tr>
<tr>
<td></td>
<td>0x110</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>

**Offset**

<table>
<thead>
<tr>
<th>yp</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>xp</td>
<td>8</td>
</tr>
<tr>
<td>%ebp</td>
<td>0</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

**Address**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>0x124</td>
</tr>
<tr>
<td>456</td>
<td>0x120</td>
</tr>
<tr>
<td></td>
<td>0x11c</td>
</tr>
<tr>
<td></td>
<td>0x118</td>
</tr>
<tr>
<td></td>
<td>0x114</td>
</tr>
<tr>
<td></td>
<td>0x110</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>

**Offset**

<table>
<thead>
<tr>
<th>yp</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>xp</td>
<td>8</td>
</tr>
<tr>
<td>%ebp</td>
<td>0</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>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>

<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></td>
<td>4</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

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

- **D**: Constant “displacement” 1, 2, or 4 bytes
- **Rb**: Base register: Any of 8 integer registers
- **Ri**: Index register: Any, except for %esp
- **S**: Scale: 1, 2, 4, or 8

#### Special Cases

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

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

**Uses**
- Computing addresses without a memory reference
  - E.g., translation of \( p = 6x[1]; \)
- Computing arithmetic expressions of the form \( x + k \cdot y \)
  - \( k = 1, 2, 4, \) or 8.

Some Arithmetic Operations

### Format Computation

#### Two Operand Instructions
- **addl** *Src, Dest*  \( Dest = Dest + Src \)
- **subl** *Src, Dest*  \( Dest = Dest - Src \)
- **imull** *Src, Dest*  \( Dest = Dest \times Src \)
- **sall** *Src, Dest*  \( Dest = Dest \ll Src \) Also called **shll**
- **sarl** *Src, Dest*  \( Dest = Dest \gg Src \) Arithmetic
- **shrl** *Src, Dest*  \( Dest = Dest \gg Src \) Logical
- **xorl** *Src, Dest*  \( Dest = Dest \oplus Src \)
- **andl** *Src, Dest*  \( Dest = Dest \& Src \)
- **orl** *Src, Dest*  \( Dest = Dest \mid Src \)

#### One Operand Instructions
- **incl** *Dest*  \( Dest = Dest + 1 \)
- **decl** *Dest*  \( Dest = Dest - 1 \)
- **negl** *Dest*  \( Dest = - Dest \)
- **notl** *Dest*  \( Dest = \sim Dest \)

Using **leal** for Arithmetic Expressions

```plaintext
int arith
{
  int t1 = x + y;
  int t2 = t1 + 4;
  int t3 = t2 + 48;
  int t4 = t3 + t4;
  int t5 = t4 + t5;
  int rval = t2 * t5;
  return rval;
}
```

```plaintext
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
```
```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)
```
### 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;
}
```

### 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;
}
```

### 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;
}
```

### 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;
}
```
**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;
}
```

Body

```
movl 8(%ebp),%eax
exorl 12(%ebp),%eax
sarl $17,%eax
andl $8185,%eax
mov %ebp,%esp
popl %ebp
ret
```

Finish

```
2^{13} = 8192, 2^{13} - 7 = 8185
```

**Data Representations: IA32 + x86-64**

Sizes of C Objects (in Bytes)

- **C Data Type**
  - unsigned: Typical 32-bit Intel IA32 x86-64
    - 4
  - int: 4
  - long int: 4
  - char: 1
  - short: 2
  - float: 4
  - double: 8
  - long double: 8
  - char *: 4
- Or any other pointer

**x86-64 General Purpose Registers**

- %rax %eax %edi %edi
- %rbx %r12 %r12
- %rcx %ecx %r13 %r13
- %rdx %edx %r14 %r14
- %rsi %esi %r15 %r15
- %rdi %edi %r8 %r8d
- %rsp %esp %r9 %r9d
- %ebp %ebp %r10 %r10d
- %rip %rip %r11 %r11d

- Extend existing registers. Add 8 new ones.
- Make %ebp/%rbp general purpose
Swap in 32-bit Mode

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

```plaintext
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    mov (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)
    movl (%ecx),%eax
    movl (%edx),%ebx
    popl %ebp
    ret
```

Swap in 64-bit Mode

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

```plaintext
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl (%rdi), %edx
    movl (%rsi), %eax
    movl %eax, (%rdi)
    movl %edx, (%rsi)
    ret
```

- Operands passed in registers
  - First (xp) in %rdi, second (yp) in %rsi
  - 64-bit pointers
- No stack operations required
- 32-bit data
  - Data held in registers %eax and %edx
  - movl operation

Swap Long Ints in 64-bit Mode

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

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

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

Summary

Machine Level Programming
- Assembly code is textual form of binary object code
- Low-level representation of program
  - Explicit manipulation of registers
  - Simple and explicit instructions
  - Minimal concept of data types
  - Many C control constructs must be implemented with multiple instructions

Formats
- IA32: Historical x86 format
- x86-64: Big evolutionary step