Introduction to Computer Systems
15-213, fall 2009
4th Lecture, Sep 2nd

Instructors:
Majd Sakr and Khaled Harras
Last Time: Floating Point

- Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary
Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Addressing mode, address computation ($lea$)
- Arithmetic operations
Intel x86 Processors

- Totally dominate computer market

- Evolutionary design
  - Backwards compatible up until 8086, introduced in 1978
  - Added more features as time goes on

- 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!
# Intel x86 Evolution: Milestones

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
<th>MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086</td>
<td>1978</td>
<td>29K</td>
<td>5-10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>386</td>
<td>1985</td>
<td>275K</td>
<td>16-33</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium 4F</td>
<td>2005</td>
<td>230M</td>
<td>2800-3800</td>
</tr>
</tbody>
</table>

- First 16-bit processor. Basis for IBM PC & DOS
- 1MB address space
- First 32 bit processor, referred to as IA32
- Added “flat addressing”
- Capable of running Unix
- 32-bit Linux/gcc uses no instructions introduced in later models
- First 64-bit processor
- Meanwhile, Pentium 4s (Netburst arch.) phased out in favor of “Core” line
# Intel x86 Processors: Overview

<table>
<thead>
<tr>
<th>Architectures</th>
<th>Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>X86-16</td>
<td>8086</td>
</tr>
<tr>
<td>X86-32/IA32</td>
<td>286</td>
</tr>
<tr>
<td>MMX</td>
<td>386</td>
</tr>
<tr>
<td></td>
<td>486</td>
</tr>
<tr>
<td></td>
<td>Pentium</td>
</tr>
<tr>
<td></td>
<td>Pentium MMX</td>
</tr>
<tr>
<td>SSE</td>
<td>Pentium III</td>
</tr>
<tr>
<td>SSE2</td>
<td>Pentium 4</td>
</tr>
<tr>
<td>SSE3</td>
<td>Pentium 4E</td>
</tr>
<tr>
<td>X86-64 / EM64t</td>
<td>Pentium 4F</td>
</tr>
<tr>
<td>SSE4</td>
<td>Core 2 Duo</td>
</tr>
<tr>
<td></td>
<td>Core i7</td>
</tr>
</tbody>
</table>

IA: often redefined as latest Intel architecture
Intel x86 Processors, contd.

- **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
  - Core 2 Duo 2006 291M

- **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**
  - Very limited
More Information

- Intel processors (Wikipedia)
- Intel microarchitectures
**New Species: ia64, then IPF, then Itanium,...**

<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>▪ First shot at 64-bit architecture: first called IA64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>▪ Radically new instruction set designed for high performance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>▪ Can run existing IA32 programs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>▪ On-board “x86 engine”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>▪ Joint project with Hewlett-Packard</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Itanium 2</td>
<td>2002</td>
<td>221M</td>
</tr>
<tr>
<td>▪ Big performance boost</td>
<td></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, no good compiler support, Pentium 4 got too good</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
x86 Clones: Advanced Micro Devices (AMD)

- Historically
  - AMD has followed just behind Intel
  - A little bit slower, a lot cheaper

- Then
  - Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies
  - Built Opteron: tough competitor to Pentium 4
  - Developed x86-64, their own extension to 64 bits

- Recently
  - Intel much quicker with dual core design
  - Intel currently far ahead in performance
  - em64t backwards compatible to x86-64
Intel’s 64-Bit

- Intel Attempted Radical Shift from IA32 to IA64
  - Totally different architecture (Itanium)
  - 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!

- Meanwhile: EM64t well introduced, however, still often not used by OS, programs
Our Coverage

- **IA32**
  - The traditional x86
  - Our unix.qatar.cmu.edu machines

- **x86-64/EM64T**
  - The emerging standard

- **Presentation**
  - Book has IA32
  - Lecture will cover IA32
Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Addressing mode, address computation (lea)
- Arithmetic operations
Definitions

- **Architecture**: (also instruction set architecture: ISA) The parts of a processor design that one needs to understand to write assembly code.

- **Microarchitecture**: Implementation of the architecture.

- **Architecture examples**: instruction set specification, registers.

- **Microarchitecture examples**: cache sizes and core frequency.

- **Example ISAs (Intel)**: x86, IA, IPF
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**
  - Byte addressable array
  - Code, user data, (some) OS data
  - Includes stack used to support procedures
Turning C into Object Code

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

Diagram:

- Text: \texttt{C program (p1.c p2.c)} → Compiler \texttt{(gcc -S)}
- Text: \texttt{Asm program (p1.s p2.s)} → Assembler \texttt{(gcc or as)}
- Binary: \texttt{Object program (p1.o p2.o)} → Linker \texttt{(gcc or ld)}
- Binary: \texttt{Executable program (p)} → Static libraries \texttt{(.a)}
Compiling Into Assembly

C Code

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

Generated IA32 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`

Some compilers use single instruction “leave”
Assembly Characteristics: 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
Assembly Characteristics: 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`

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401040</td>
<td><code>&lt;sum&gt;</code>:</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>

- 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

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

**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`

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

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

```assembly
0x401046: 03 45 08
```
**Disassembling Object Code**

**Disassembled**

```
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
```

**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

<table>
<thead>
<tr>
<th>Object</th>
<th>Disassembled</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>0x401040</code>:</td>
<td><code>0x55 0x89 0xe5 0x8b 0x45 0x0c 0x03 0x45 0x08 0x89 0xec 0x5d 0xc3</code></td>
</tr>
<tr>
<td></td>
<td><code>0x401041 &lt;sum+1&gt;: mov    %esp,%ebp</code></td>
</tr>
<tr>
<td></td>
<td><code>0x401043 &lt;sum+3&gt;: mov    0xc(%ebp),%eax</code></td>
</tr>
<tr>
<td></td>
<td><code>0x401046 &lt;sum+6&gt;: add    0x8(%ebp),%eax</code></td>
</tr>
<tr>
<td></td>
<td><code>0x401049 &lt;sum+9&gt;: mov    %ebp,%esp</code></td>
</tr>
<tr>
<td></td>
<td><code>0x40104b &lt;sum+11&gt;: pop    %ebp</code></td>
</tr>
<tr>
<td></td>
<td><code>0x40104c &lt;sum+12&gt;: ret</code></td>
</tr>
<tr>
<td></td>
<td><code>0x40104d &lt;sum+13&gt;: lea    0x0(%esi),%esi</code></td>
</tr>
</tbody>
</table>

### Within gdb Debugger

```
gdb p
disable 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
Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Addressing mode, address computation (lea)
- Arithmetic operations
## Integer Registers (IA32)

<table>
<thead>
<tr>
<th>Register</th>
<th>General Purpose</th>
<th>16-bit Virtual Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>32-bit</td>
<td>%al</td>
</tr>
<tr>
<td>%ecx</td>
<td>16-bit</td>
<td>%ch</td>
</tr>
<tr>
<td>%edx</td>
<td>16-bit</td>
<td>%dh</td>
</tr>
<tr>
<td>%ebx</td>
<td>16-bit</td>
<td>%bh</td>
</tr>
<tr>
<td>%esi</td>
<td>16-bit</td>
<td>%bl</td>
</tr>
<tr>
<td>%edi</td>
<td>16-bit</td>
<td>%sp</td>
</tr>
<tr>
<td>%esp</td>
<td>16-bit</td>
<td>%bp</td>
</tr>
<tr>
<td>%ebp</td>
<td>16-bit</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Origin (mostly obsolete)</th>
</tr>
</thead>
<tbody>
<tr>
<td>accumulate</td>
</tr>
<tr>
<td>counter</td>
</tr>
<tr>
<td>data</td>
</tr>
<tr>
<td>base</td>
</tr>
<tr>
<td>source</td>
</tr>
<tr>
<td>index</td>
</tr>
<tr>
<td>destination</td>
</tr>
<tr>
<td>index</td>
</tr>
<tr>
<td>stack</td>
</tr>
<tr>
<td>pointer</td>
</tr>
<tr>
<td>base</td>
</tr>
<tr>
<td>pointer</td>
</tr>
</tbody>
</table>

*16-bit virtual registers (backwards compatibility)*
Moving Data: IA32

Moving Data

- `movx Source, Dest`
- `x` in `{b, w, l}`

- `movl Source, Dest:`
  Move 4-byte “long word”

- `movw Source, Dest:`
  Move 2-byte “word”

- `movb Source, Dest:`
  Move 1-byte “byte”

Lots of these in typical code
Moving Data: IA32

- Moving Data
  - movl \textit{Source}, \textit{Dest}:

- Operand Types
  - \textit{Immediate}: Constant integer data
    - Example: $0x400, \$-533$
    - Like C constant, but prefixed with \texttt{\$'}
    - Encoded with 1, 2, or 4 bytes
  - \textit{Register}: One of 8 integer registers
    - Example: \%eax, \%edx
    - But \%/esp and \%/ebp reserved for special use
    - Others have special uses for particular instructions
  - \textit{Memory}: 4 consecutive bytes of memory at address given by register
    - Simplest example: (\%eax)
    - Various other “address modes”
### 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>movl</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Imm</td>
<td>Reg</td>
<td>movl $0x4,%eax</td>
<td>temp = 0x4;</td>
</tr>
<tr>
<td></td>
<td>Mem</td>
<td>movl $-147,(%eax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td>Reg</td>
<td>Reg</td>
<td>movl %eax,%edx</td>
<td>temp2 = temp1;</td>
</tr>
<tr>
<td></td>
<td>Mem</td>
<td>movl %eax,(%edx)</td>
<td>*p = temp;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl (%eax),%edx</td>
<td>temp = *p;</td>
</tr>
</tbody>
</table>

*Cannot do memory-memory transfer with a single instruction*
Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Addressing mode, address computation (lea)
- Arithmetic operations
Simple Memory Addressing Modes

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

- **Displacement**  
  \( \text{D}(R) \)  
  \( \text{Mem}[	ext{Reg}[R]+D] \)
  - 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
```
Using Simple Addressing Modes

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

Set Up

Body

Finish
Understanding Swap

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

Stack (in memory)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>yp</td>
</tr>
<tr>
<td>8</td>
<td>xp</td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>0</td>
<td>Old %ebp</td>
</tr>
<tr>
<td>-4</td>
<td>Old %ebx</td>
</tr>
</tbody>
</table>

Register Value

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>yp</td>
</tr>
<tr>
<td>%edx</td>
<td>xp</td>
</tr>
<tr>
<td>%eax</td>
<td>t1</td>
</tr>
<tr>
<td>%ebx</td>
<td>t0</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>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>123</td>
<td>0x120</td>
</tr>
<tr>
<td>%edx</td>
<td>456</td>
<td>0x11c</td>
</tr>
<tr>
<td>%ecx</td>
<td>0</td>
<td>0x118</td>
</tr>
<tr>
<td>%ebx</td>
<td>8</td>
<td>0x114</td>
</tr>
<tr>
<td>%esi</td>
<td>4</td>
<td>0x10c</td>
</tr>
<tr>
<td>%edi</td>
<td>0</td>
<td>0x108</td>
</tr>
<tr>
<td>%esp</td>
<td>-4</td>
<td>0x104</td>
</tr>
<tr>
<td>%ebp</td>
<td>0</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`
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

Carnegie Mellon

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>12</td>
<td>yp</td>
</tr>
<tr>
<td>456</td>
<td>8</td>
<td>xp</td>
</tr>
<tr>
<td>0x120</td>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>0x110</td>
<td>0</td>
<td>%ebp</td>
</tr>
<tr>
<td>0x108</td>
<td>-4</td>
<td></td>
</tr>
<tr>
<td>0x104</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x11c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x118</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x114</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x120</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4</td>
<td>0x100</td>
</tr>
<tr>
<td>0</td>
<td>0x104</td>
</tr>
<tr>
<td>4</td>
<td>0x108</td>
</tr>
<tr>
<td>8</td>
<td>0x124</td>
</tr>
<tr>
<td>12</td>
<td>0x120</td>
</tr>
</tbody>
</table>

| %ebp   | 0x104 |
| %esp   | 0x100 |
| %edi   |       |
| %esi   |       |
| %ebx   | 0x124 |
| %edx   | 0x120 |
| %ecx   | 0x12c |
| %eax   | 0x120 |

%ebp → 0
%esp → 0
%ebp → 0x104
%esp → 0x100

%eax
%edx 0x124
%ecx 0x120
%ebx
%esi
%edi
%esp
%ebp 0x104
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></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

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

\[
\begin{array}{|c|}
\hline
\text{Address} & \\
\hline
0x124 & 123 \\
0x120 & 456 \\
0x11c & \\
0x118 & \\
0x114 & \\
0x110 & \\
0x10c & \\
0x108 & \\
0x104 & \\
0x100 & \\
\hline
\end{array}
\]

\begin{align*}
\text{movl} & \ 12(\%ebp),\%ecx \quad \# \ ecx = yp \\
\text{movl} & \ 8(\%ebp),\%edx \quad \# \ edx = xp \\
\text{movl} & \ (%ecx),\%eax \quad \# \ eax = *yp \ (t1) \\
\text{movl} & \ (%edx),\%ebx \quad \# \ ebx = *xp \ (t0) \\
\text{movl} & \ %eax, (%edx) \quad \# \ *xp = eax \\
\text{movl} & \ %ebx, (%ecx) \quad \# \ *yp = ebx \\
\end{align*}
## Understanding Swap

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
<td>12</td>
<td>yp</td>
<td>0x110</td>
</tr>
<tr>
<td>0x120</td>
<td>8</td>
<td>xp</td>
<td>0x10c</td>
</tr>
<tr>
<td>0x11c</td>
<td>4</td>
<td>Rtn adr</td>
<td>0x108</td>
</tr>
<tr>
<td>0x118</td>
<td></td>
<td></td>
<td>0x104</td>
</tr>
<tr>
<td>0x114</td>
<td></td>
<td></td>
<td>0x100</td>
</tr>
<tr>
<td>0x110</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Registers

<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>

```asm
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></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>0x11c</td>
</tr>
<tr>
<td>%ebx</td>
<td>0x118</td>
</tr>
<tr>
<td>%esi</td>
<td>0x114</td>
</tr>
<tr>
<td>%edi</td>
<td>0x10c</td>
</tr>
<tr>
<td>%esp</td>
<td>0x108</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
<tr>
<td></td>
<td>0x100</td>
</tr>
</tbody>
</table>

## Variables
- yp: 12
- xp: 8
- Rtn adr: 0

## offsets
- bp = 0
- bp = -4

## Code Snippet
```
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
```
Complete Memory Addressing Modes

- **Most General Form**
  \[
  D(Rb, Ri, S) \quad \text{Mem}[\text{Reg}[Rb] + S*\text{Reg}[Ri] + D]
  \]
  - **D:** Constant “displacement” 1, 2, or 4 bytes
  - **Rb:** Base register: Any of 8 integer registers
  - **Ri:** Index register: Any, except for %esp
    - Unlikely you’d use %ebp, either
  - **S:** Scale: 1, 2, 4, or 8 (*why these numbers?*)

- **Special Cases**
  - \((Rb, Ri)\) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri]]
  - \(D(Rb, Ri)\) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] + D]
  - \((Rb, Ri, S)\) \quad \text{Mem}[\text{Reg}[Rb] + S*\text{Reg}[Ri]]
## Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8 (%edx)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%edx, %ecx)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%edx, %ecx, 4)</td>
<td></td>
<td>will disappear blackboard?</td>
</tr>
<tr>
<td>0x80 (,%edx, 2)</td>
<td></td>
<td></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 Examples

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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address 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

- **lea 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 = &x[i];`
  - Computing arithmetic expressions of the form *x* + *k*\*y
    - *k* = 1, 2, 4, or 8

- **Example**
Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Addressing mode, address computation (lea)
- Arithmetic operations
Some Arithmetic Operations

- Two Operand Instructions:

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addl</code></td>
<td><code>Dest = Dest + Src</code></td>
</tr>
<tr>
<td><code>subl</code></td>
<td><code>Dest = Dest - Src</code></td>
</tr>
<tr>
<td><code>imull</code></td>
<td><code>Dest = Dest * Src</code></td>
</tr>
<tr>
<td><code>sall</code></td>
<td><code>Dest = Dest &lt;&lt; Src</code></td>
</tr>
<tr>
<td><code>sarl</code></td>
<td><code>Dest = Dest &gt;&gt; Src</code></td>
</tr>
<tr>
<td><code>shrl</code></td>
<td><code>Dest = Dest &gt;&gt; Src</code></td>
</tr>
<tr>
<td><code>xorl</code></td>
<td><code>Dest = Dest ^ Src</code></td>
</tr>
<tr>
<td><code>andl</code></td>
<td><code>Dest = Dest &amp; Src</code></td>
</tr>
<tr>
<td><code>orl</code></td>
<td>`Dest = Dest</td>
</tr>
</tbody>
</table>

- No distinction between signed and unsigned int (why?)
Some Arithmetic Operations

- One Operand Instructions

  incl \( Dest \) \( \Rightarrow \) \( Dest = Dest + 1 \)

  decl \( Dest \) \( \Rightarrow \) \( Dest = Dest - 1 \)

  negl \( Dest \) \( \Rightarrow \) \( Dest = -Dest \)

  notl \( Dest \) \( \Rightarrow \) \( Dest = \sim Dest \)

- See book for more instructions
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;
}
```

```
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 = 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)
```

Will disappear blackboard?
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)
```
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;
}
```

```
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;
}
```

```
    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;
}
```

```assembly
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**
  - movl 8(%ebp),%eax
    # eax = x
  - xorl 12(%ebp),%eax
    # eax = x^y
  - sarl $17,%eax
    # eax = t1>>17
  - andl $8185,%eax
    # eax = t2 & 8185

- **Body**
  - movl %ebp,%esp
  - popl %esp
  - ret
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:**
- **Set Up**
  - pushl %ebp
  - movl %esp,%ebp

- **Body**
  - movl 8(%ebp),%eax
  - xorl 12(%ebp),%eax
  - sarl $17,%eax
  - andl $8185,%eax

- **Finish**
  - movl %ebp,%esp
  - popl %ebp
  - ret

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:**
- Push the stack pointer (`pushl %ebp`)
- Move the stack pointer to `%ebp` (`movl %esp, %ebp`)
- Move 8 at the base of `%ebp` to `%eax` (`movl 8(%ebp), %eax`)
- XOR 12 with `%ebp` and `%eax` (`xorl 12(%ebp), %eax`)
- SAR 17 with `%eax` (`sarl $17, %eax`)
- AND 8185 with `%eax` (`andl $8185, %eax`)
- Move `%ebp` to `%esp` (`movl %ebp, %esp`)
- Pop `%ebp` from the stack (`popl %ebp`)
- Return (`ret`)

- Move 8 at the base of `%ebp` to `%eax` (`movl 8(%ebp), %eax`)
- Set up the environment (`Set Up`)
- Move 8 at the base of `%ebp` to `%eax` (`movl 8(%ebp), %eax`)
- XOR 12 with `%ebp` and `%eax` (`xorl 12(%ebp), %eax`)
- SAR 17 with `%eax` (`sarl $17, %eax`)
- AND 8185 with `%eax` (`andl $8185, %eax`)
- Move `%ebp` to `%esp` (`movl %ebp, %esp`)
- Pop `%ebp` from the stack (`popl %ebp`)
- Return (`ret`)

- Move 8 at the base of `%ebp` to `%eax` (`movl 8(%ebp), %eax`)
- XOR 12 with `%ebp` and `%eax` (`xorl 12(%ebp), %eax`)
- SAR 17 with `%eax` (`sarl $17, %eax`)
- AND 8185 with `%eax` (`andl $8185, %eax`)
- Move `%ebp` to `%esp` (`movl %ebp, %esp`)
- Pop `%ebp` from the stack (`popl %ebp`)
- Return (`ret`)

**Body**

**Finish**
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:

- **Set Up**
  - `pushl %ebp`
  - `movl %esp,%ebp`

- **Body**
  - `movl 8(%ebp),%eax`  \( eax = \text{x} \)
  - `xorl 12(%ebp),%eax`  \( eax = x^y \)
  - `sarl $17,%eax`  \( eax = t1>>17 \)
  - `andl $8185,%eax`  \( eax = t2\ & \ 8185 \)

- **Finish**
  - `movl %ebp,%esp`
  - `popl %ebp`
  - `ret`

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