Introduction to Computer Systems
15-213/18-243, fall 2009
4th Lecture, Sep. 3rd

Instructors:
Roger B. Dannenberg and Greg Ganger
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
Intel x86 Processors

- Totally dominate computer market

- Evolutionary design
  - Backwards compatible back to 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>SSE</td>
<td>486</td>
</tr>
<tr>
<td>SSE2</td>
<td>Pentium</td>
</tr>
<tr>
<td>SSE3</td>
<td>Pentium MMX</td>
</tr>
<tr>
<td>X86-64 / EM64t</td>
<td>Pentium III</td>
</tr>
<tr>
<td>SSE4</td>
<td>Pentium 4</td>
</tr>
<tr>
<td></td>
<td>Pentium 4E</td>
</tr>
<tr>
<td></td>
<td>Pentium 4F</td>
</tr>
<tr>
<td></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](https://en.wikipedia.org/wiki/Intel_processor))
- Intel [microarchitectures](https://en.wikipedia.org/wiki/Intel_microarchitecture)
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></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>▪ First shot at 64-bit architecture: first called IA64</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▪ Radically new instruction set designed for high performance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▪ Can 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>
<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></td>
<td></td>
<td>▪ Lack of backward compatibility, no good compiler support, Pentium 4 got too good</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!
  - Our Saltwater fish machines

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

- **IA32**
  - The traditional x86

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

**Presentation**

- Book has IA32
- Handout has x86-64
- Lecture will cover both
Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
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: `p1.c p2.c`
- Compile with command: `gcc -O p1.c p2.c -o p`
  - Use optimizations (`-O`)
  - Put resulting binary in file `p`

```plaintext
text
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 IA32 Assembly**

```asm
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`

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

0x401040 <sum>:
- 0x55
- 0x89
- 0xe5
- 0x8b
- 0x45
- 0x0c
- 0x03
- 0x45
- 0x08
- 0x89
- 0xec
- 0xc3
- Total of 13 bytes
- Each instruction 1, 2, or 3 bytes
- Starts at address 0x401040

- Assembler
  - Translates .s into .o
  - Binary encoding of each instruction
  - Nearly-complete image of executable code
  - Missing linkages between code in different files

- Linker
  - Resolves references between files
  - Combines with static run-time libraries
    - E.g., code for `malloc`, `printf`
  - Some libraries are dynamically linked
    - Linking occurs when program begins execution
**Machine Instruction Example**

- **C Code**
  - Add two signed integers

- **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 \( %eax \)

- **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>_sum:</code></td>
<td></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>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401040</td>
<td>0x55 0x89 0xe5 0x8b 0x45 0x0c 0x03 0x45 0x08 0x89 0xec 0x5d 0xc3</td>
</tr>
</tbody>
</table>

Disassembled

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401040</td>
<td>push %ebp</td>
</tr>
<tr>
<td>0x401041</td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td>0x401043</td>
<td>mov 0xc(%ebp),%eax</td>
</tr>
<tr>
<td>0x401046</td>
<td>add 0x8(%ebp),%eax</td>
</tr>
<tr>
<td>0x401049</td>
<td>mov %ebp,%esp</td>
</tr>
<tr>
<td>0x40104b</td>
<td>pop %ebp</td>
</tr>
<tr>
<td>0x40104c</td>
<td>ret</td>
</tr>
<tr>
<td>0x40104d</td>
<td>lea 0x0(%esi),%esi</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
Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
Integer Registers (IA32)

- **%eax**
  - %ax
  - %ah
  - %al
- **%ecx**
  - %cx
  - %ch
  - %cl
- **%edx**
  - %dx
  - %dh
  - %dl
- **%ebx**
  - %bx
  - %bh
  - %bl
- **%esi**
  - %si
- **%edi**
  - %di
- **%esp**
  - %sp
- **%ebp**
  - %bp

**16-bit virtual registers (backwards compatibility)**

- Origin (mostly obsolete)
  - accumulate
  - counter
  - data
  - base
  - source
  - index
  - destination
  - index
  - stack
  - pointer
  - base
  - pointer
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**
  
  \texttt{movl \textit{Source}, \textit{Dest}}:

- **Operand Types**
  - **Immediate**: Constant integer data
    - Example: $\$0x400$, $\$-533$
    - Like C constant, but prefixed with `\$'
    - Encoded with 1, 2, or 4 bytes
  - **Register**: One of 8 integer registers
    - Example: \%eax, \%edx
    - But \%esp and \%ebp reserved for special use
    - Others have special uses for particular instructions
  - **Memory**: 4 consecutive bytes of memory at address given by register
    - Simplest example: (\%eax)
    - Various other “address modes”

<table>
<thead>
<tr>
<th>%eax</th>
<th>%ecx</th>
<th>%edx</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ebx</td>
<td>%esi</td>
<td>%edi</td>
</tr>
<tr>
<td>%esp</td>
<td>%ebp</td>
<td></td>
</tr>
</tbody>
</table>
### 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>Imm</td>
<td>Reg</td>
<td>movl $0x4,%eax</td>
<td>temp = 0x4;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl $-147,(%eax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td>Reg</td>
<td>Mem</td>
<td>movl %eax,%edx</td>
<td>temp2 = temp1;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</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*
Simple Memory Addressing Modes

- Normal  (R)  Mem[Reg[R]]
  - Register R specifies memory address

  \texttt{movl} (\%ecx),\%eax

- Displacement  D(R)  Mem[Reg[R]+D]
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

  \texttt{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;
}
```

```assembly
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
```

<table>
<thead>
<tr>
<th>Set Up</th>
<th>Body</th>
<th>Finish</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushl %ebp movl %esp,%ebp pushl %ebx</td>
<td>movl 12(%ebp),%ecx movl 8(%ebp),%edx movl (%ecx),%eax movl (%edx),%ebx movl %eax,(%edx) movl %ebx,(%ecx)</td>
<td>movl -4(%ebp),%ebx movl %ebp,%esp popl %ebp ret</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;
}
```

```assembly
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;
}
```

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>

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

<table>
<thead>
<tr>
<th>%eax</th>
<th>%edx</th>
<th>%ecx</th>
<th>%ebx</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></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

Offset

<table>
<thead>
<tr>
<th>yp</th>
<th>12</th>
<th>0x120</th>
</tr>
</thead>
<tbody>
<tr>
<td>xp</td>
<td>8</td>
<td>0x124</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>%ebp</td>
<td>0</td>
<td>0x108</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
</tr>
<tr>
<td>456</td>
</tr>
<tr>
<td>123</td>
</tr>
<tr>
<td>0x120</td>
</tr>
<tr>
<td>0x120</td>
</tr>
<tr>
<td>0x114</td>
</tr>
<tr>
<td>0x110</td>
</tr>
<tr>
<td>0x124</td>
</tr>
<tr>
<td>0x10c</td>
</tr>
<tr>
<td>0x108</td>
</tr>
<tr>
<td>0x104</td>
</tr>
<tr>
<td>0x100</td>
</tr>
</tbody>
</table>

%ebp 0x104
Understanding Swap

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

<table>
<thead>
<tr>
<th>%eax</th>
<th>%edx 0x124</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx 0x120</td>
<td></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 0x104</td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{array}{cccc}
\text{Offset} & \text{Address} \\
0 & 0x104 \\
-4 & 0x100 \\
4 & 0x108 \\
8 & 0x10c \\
12 & 0x110 \\
 & 0x114 \\
 & 0x118 \\
 & 0x120 \\
 & 0x124 \\
\end{array}
\]

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
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

<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>

### Memory Addresses

<table>
<thead>
<tr>
<th>Offset</th>
<th>Value</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>
</tbody>
</table>

### Code Snippet

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

- **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
------------------
456 0x124
456 0x120
0x11c 0x118
0x114 0x110
0x124 0x10c
Rtn adr 0x108
0x104 0x100

Offset:
- yp 12 0x120 0x110
- xp 8 0x124 0x10c
- ebx 4 0x108 0x104
- eax -4 0x100 0x100

%ebp → 0
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
Complete Memory Addressing Modes

- **Most General Form**

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

  - **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**

  \[
  \begin{align*}
  (Rb, Ri) & \quad Mem[\text{Reg}[Rb] + \text{Reg}[Ri]] \\
  D(Rb, Ri) & \quad Mem[\text{Reg}[Rb] + \text{Reg}[Ri] + D] \\
  (Rb, Ri, S) & \quad Mem[\text{Reg}[Rb] + S*\text{Reg}[Ri]]
  \end{align*}
  \]