CS429: Computer Organization and Architecture
Instruction Set Architecture II

Warren Hunt, Jr. and Bill Young
Department of Computer Sciences
University of Texas at Austin

Last updated: October 1, 2014 at 08:36
Topics of this Slideset

- Assembly Programmer’s Execution Model
- Accessing Information
- Registers
- Memory
- Arithmetic operations

*BTW: We’re through with Y86 for a while, and starting the x86. We’ll come back to the Y86 later for pipelining.*
x86 processors totally dominate the computer market.

**Evolutionary Design**
- Starting in 1978 with 8086
- Added more features over time.

**Complex Instruction Set Computer (CISC)**
- Still support many old, now obsolete, features.
- There are many different instructions with many different formats, but only a small subset are encountered with Linux programs.
- Hard to match performance of Reduced Instruction Set Computers (RISC), though Intel has done just that!
<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>
</tbody>
</table>
|       |       | 16-bit processor. Basis for IBM PC and DOS.  
|       |       | Limited to 1MB address space. DOS only gives you 640K |
| 80286 | 1982  | 134K        |
|       |       | Added elaborate, but not very useful, addressing scheme.  
|       |       | Basis for IBM PC-AT and Windows |
| 386   | 1985  | 275K        |
|       |       | Extended to 32 bits. Added “flat addressing.”  
|       |       | Capable of running Unix.  
<p>|       |       | Linux/gcc uses no instructions introduced in later models. |</p>
<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
</tr>
</thead>
<tbody>
<tr>
<td>486 Pro</td>
<td>1989</td>
<td>1.9M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Added on chip floating point unit.</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>3.1M</td>
</tr>
<tr>
<td>Pentium/MMX</td>
<td>1997</td>
<td>4.5M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Added special collection of instructions for operating on 64-bit vectors of 1, 2, or 4 byte integer data.</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>1995</td>
<td>6.5M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Added conditional move instructions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Big change in underlying microarchitecture.</td>
</tr>
<tr>
<td>Name</td>
<td>Date</td>
<td>Transistors</td>
</tr>
<tr>
<td>-----------------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>Pentium III</td>
<td>1999</td>
<td>8.2M</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Added “streaming SIMD” instructions for operating on 128-bit vectors of 1, 2, or 4 byte integer or FP data.</em></td>
</tr>
<tr>
<td>Pentium 4</td>
<td>2001</td>
<td>42M</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Added 8-byte formats and 144 new instructions for streaming SIMD.</em></td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>“Superpipelined” with very fast clocks.</em></td>
</tr>
<tr>
<td>Pentium 4 Xeon</td>
<td>2003</td>
<td>125M</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Added hyperthreading, large caches.</em></td>
</tr>
<tr>
<td>Pentium M</td>
<td>2003</td>
<td>77M</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Added hyperthreading, lower power.</em></td>
</tr>
</tbody>
</table>
### x86 Evolution: Programmer’s View

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium 4 EE</td>
<td>2005</td>
<td>164M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Includes hyperthreading.</td>
</tr>
<tr>
<td>Pentium EE 840</td>
<td>2005</td>
<td>230M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Dual core, shared L2 cache.</td>
</tr>
<tr>
<td>Pentium EE 840</td>
<td>2005</td>
<td>675M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- 8 Mbyte L3 Cache.</td>
</tr>
<tr>
<td>Pentium Core Duo</td>
<td>2006</td>
<td>250M+</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Dual core, shared L2 cache.</td>
</tr>
<tr>
<td>“Nehalem” Core</td>
<td>2008</td>
<td>731M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Added 256-bit media instructions, on-chip memory controller.</td>
</tr>
</tbody>
</table>
“Tick-tock” implementation strategy.

- Change processor in middle of stable technology.
- Change technology in middle of stable design.

Pentium i3, i5, i7 2010 1.16B
- “Sandy Bridge” 4 core, 2.27B transitors, 8 core.

Pentium i3, i5, i7 2012 1.4B
- “Ivy Bridge” Tri-gate (3-D) transistor technology.
New Species: IA64

<table>
<thead>
<tr>
<th>Itanium</th>
<th>Year</th>
<th>Max Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>2001</td>
<td>25M</td>
<td></td>
</tr>
<tr>
<td>2002</td>
<td>221M</td>
<td></td>
</tr>
<tr>
<td>2006</td>
<td>410M</td>
<td></td>
</tr>
</tbody>
</table>

- Extends to IA64, a 64-bit architecture
- Radically new instruction set designed for high performance.
- Able to run existing IA32 programs with on-board “x86 engine.”
- Joint project with Hewlett-Packard.

Itanium 2 2002 221M
- Big performance boost.

Itanium 2, Big 2006 410M
- 24 Mbyte cache, good for server operations, expensive ($500 to $4000) per processor.

Not good market acceptance; Intel ended support.
Transmeta
Radically different approach to implementation.
- Translate x86 code into “very long instruction word” (VLIW) code.
- Very high degree of parallelism.

Centaur / Via
- Continued evolution from Cyrix, the 3rd x86 vendor. Low power, design team in Austin.
- 32-bit processor family.
  - At 2 GHz, around 2 watts; at 600 MHz around 0.5 watt.
- 64-bit processor family, used by HP, Lenovo, OLPC, IBM.
  - Very low power, only a few watts at 1.2 GHz.
  - Full virtualization and SSE support.
Advanced Micro Devices (AMD)

Historically,

- AMD has followed just behind Intel.
- A little bit slower, but enough cheaper to be competitive.

Recently,

- Recruited top circuit designers from Digital Equipment Corp.
- Exploited the fact that Intel was distracted by IA64.
- Now is a close competitor to Intel.

Developed first commerically-available x86 extension to 64-bits, which forced Intel to do the same.

Current,

- Delivering 64-bit x86 processors for desktop and server market.
- Have sophisticated point-to-point IO bus.
- Providing competitive products in desktop and server market.
Abstract and Concrete Machine Models

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
Programmer Visible State

- EIP (Program Counter): address of next instruction.
- Register file: heavily used program data.
- Condition codes:
  - Store status info about most recent arithmetic operation.
  - Used for conditional branching.

Memory

- Byte addressable array.
- Code, user data, (some) OS data.
- Includes stack.
ISA Principles

- Contract between programmer and the hardware.
  - Defines visible state of the system.
  - Defines how state changes in response to instructions.

- For Programmer: ISA is model of how a program will execute.

- For Hardware Designer: ISA is formal definition of the correct way to execute a program.
  - With a stable ISA, SW doesn’t care what the HW looks like under the hood.
  - Hardware implementations can change drastically.
  - As long as the HW implements the same ISA, all prior SW should still run.
  - Example: x86 ISA has spanned many chips; instructions have been added but the SW for prior chips still runs.

- ISA specification: the binary encoding of the instruction set.
ISA Basics

Instruction formats
Instruction types
Addressing modes

Op   Mode    Ra    Rb
After StateBefore State
Instruction
Data type
Interrupts / Events
Operations
Machine State
Memory organization
Register organization
Instruction formats
Addressing modes
Instruction types

Machine State
Memory organization
Register organization
Architecture: defines what a computer system does in response to a program and set of data.

- Programmer visible elements of computer system.

Implementation: defines how a computer does it.

- Sequence of steps to complete operations.
- Time to execute each operation.
- Hidden “bookkeeping” function.
Architecture or Implementation?

- Number of general purpose registers
- Width of memory bus
- Binary representation of the instruction: `sub r4, r2, #27`
- Number of cycles to execute a FP instruction
- Which condition code bits are set on a `move` instruction
- Size of the instruction cache
- Type of FP format
Turning C into Object Code

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

![Diagram showing the process of compiling C programs into object code, executables, and static libraries](image)
Compiling into Assembly

C Code:

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

Run command: `gcc -O -S code.c` produces file `code.s`.

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

**Minimal Data Types**
- “Integer” data of 1, 2 or 4 bytes
- 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 functions 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

0x401040: <sum>:
  0x55  # total of
  0x89  # 13 bytes
  0xe5  # each inst
  0x8b  # 1, 2, or
  0x45  # 3 bytes
  0x0c  # starts at
  0x03  # addr
  0x45  # 0x401040
  0x08  # 0x89
  0xec  # 0x401040
  0x5d  # addr
  0xc3

Assembler
- Translates .s into .o
- Binary encoding of each inst.
- 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 just before execution.
Machine Instruction Example

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

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

Similar to expression: `x += y`.

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

**Assembly**
- Add two 4-byte integers
- “Long” words in GCC terms
- Same instruction signed or unsigned
- Operands:
  - `x`: Register `%eax`
  - `y`: Memory `M[%ebp+8]`
  - `t`: Register `%eax`
- Return value in `%eax`

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

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

Object code:

```
0x401040:
  0x55
  0x89
  0xe5
  0x8b
  0x45
  0x0c
  0x45
  0x03
  0x45
  0x08
  0x89
  0xec
  0x5d
  0xc3
```

Disassemble procedure:

```
x/13b sum
```

Examine the 13 bytes starting at sum.

```gdb p
disable sum
```
What Can be Disassembled?

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

```bash
% 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 fff       push $0xffffffff
30001005:  68 90 10 00 30 push $0x30001090
3000100a:  68 91 dc 4c 30 push $0x304cdcc91
```
### Intel/Microsoft Format

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

### Intel/Microsoft Differs from GAS

- **Operands are 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)`
From now on we’ll always use GAS assembler format.

**Moving Data:**
- Form: `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` are reserved for special use
  - Others have special uses for particular instructions
- Memory: source/dest is first address of block (4 bytes for an int)
  - Various “addressing modes”
IA32 uses the same 8 registers as Y86.

\[
\begin{array}{|c|}
\hline
%eax \\
%edx \\
%ecx \\
%ebx \\
%esi \\
%edi \\
%esp \\
%ebp \\
\hline
\end{array}
\]

Some have addressable internal structure; more on that later.
Unlike the Y86, we don’t distinguish the operator depending on the operand addressing modes.

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest.</th>
<th>Assembler</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>Register</td>
<td><code>movl $0x4,%eax</code></td>
<td><code>temp = 0x4;</code></td>
</tr>
<tr>
<td>Immediate</td>
<td>Memory</td>
<td><code>movl $-147,(%eax)</code></td>
<td><code>*p = -147;</code></td>
</tr>
<tr>
<td>Register</td>
<td>Register</td>
<td><code>movl %eax,%edx</code></td>
<td><code>temp2 = temp1;</code></td>
</tr>
<tr>
<td>Register</td>
<td>Memory</td>
<td><code>movl %eax,(%edx)</code></td>
<td><code>*p = temp;</code></td>
</tr>
<tr>
<td>Memory</td>
<td>Register</td>
<td><code>movl (%eax),%edx</code></td>
<td><code>temp = *p</code></td>
</tr>
</tbody>
</table>

Memory–memory transfers are not allowed within a single instruction.
Simple Addressing Modes

- **Register**: $\text{Reg}[R]$
  
  ```
  movl \%ecx, \%ebx
  ```

- **Normal (R)**: $\text{Mem}[\text{Reg}[R]]$
  - Register R specifies memory address.
  - This is often called *indirect* addressing.
  
  ```
  movl (\%ecx), \%eax
  ```

- **Displacement D(R)**: $\text{Mem}[\text{Reg}[R] + D]$
  - Register R specifies start of memory region.
  - Constant displacement D specifies offset
  
  ```
  movl 8(\%ecx), \%edx
  ```
Addresses and Pointers in C

C programming model is close to machine language.
- Machine language manipulates memory addresses.
  - For address computation;
  - To store addresses in registers or memory.
- C employs pointers, which are just addresses of primitive data elements or data structures.

Examples of operators * and &:
- `int a, b; /* declare integers a and b */`
- `int *a_ptr; /* a is a pointer to an integer */`
- `a_ptr = a; /* illegal, types don’t match*/`
- `a_ptr = &a; /* a_ptr holds address of a */`
- `b = *a_ptr; /* dereference a_ptr and assign value to b */`
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)

# the following is tricky:
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;
}
```

```assembly
movl 12(%ebp),%ecx  # ecx = yp
movl 8(%ebp),%edx   # edx = xp
movl (%ecx),%eax    # eax = *yp
movl (%edx),%ebx    # ebx = *xp
movl %eax,(%edx)    # *xp = eax
movl %ebx,(%ecx)    # *yp = ebx
```
**Understanding Swap (2)**

```
movl 12(%ebp),%ecx  # ecx = yp
movl 8(%ebp),%edx  # edx = xp
movl (%ecx),%eax  # eax = ∗yp
movl (%edx),%ebx  # ebx = ∗xp
movl %eax,(%edx)  # ∗xp = eax
movl %ebx,(%ecx)  # ∗yp = ebx
```

| `%eax` | `0x124` |
| `%edx` | `0x120` |
| `%ecx` | `0x11c` |
| `%ebx` | `0x118` |
| `%esi` | `0x114` |
| `%edi` | `0x110` |
| `%esp` | `0x10c` |
| `%ebp` | `0x108` |

| `yp` | `0x120` | `0x110` |
| `xp` | `0x124` | `0x10c` |
| `%ebp` | `0x108` |
| `Old %ebp` | `0x104` |
| `Old %ebx` | `0x100` |
Understanding Swap (3)

```assembly
movl 12(%ebp),%ecx # ecx = yp
movl 8(%ebp),%edx # edx = xp
movl (%ecx),%eax # eax = ∗yp
movl (%edx),%ebx # ebx = ∗xp
movl %eax,(%edx) # ∗xp = eax
movl %ebx,(%ecx) # ∗yp = ebx
```

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp 12</td>
<td>0x120</td>
</tr>
<tr>
<td>xp 8</td>
<td>0x124</td>
</tr>
<tr>
<td>4</td>
<td>Rtn addr 0x108</td>
</tr>
<tr>
<td>%ebp 0</td>
<td>Old %ebp 0x104</td>
</tr>
<tr>
<td>-4</td>
<td>Old %ebx 0x100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>123 0x124</td>
</tr>
<tr>
<td>456 0x120</td>
</tr>
<tr>
<td>0x11c</td>
</tr>
<tr>
<td>0x118</td>
</tr>
<tr>
<td>0x114</td>
</tr>
<tr>
<td>0x10c</td>
</tr>
<tr>
<td>0x108</td>
</tr>
<tr>
<td>0x104</td>
</tr>
<tr>
<td>0x100</td>
</tr>
</tbody>
</table>
Understanding Swap (4)

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

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>123 0x124</td>
<td>12 0x120</td>
</tr>
<tr>
<td>456 0x120</td>
<td>8 0x11c</td>
</tr>
<tr>
<td>0x11c 0x118</td>
<td>4 Rtn addr 0x108</td>
</tr>
<tr>
<td>0x118 0x114</td>
<td>%ebp 0x104</td>
</tr>
<tr>
<td>0x114 0x110</td>
<td>Old %ebp 0x10c</td>
</tr>
<tr>
<td>0x110 0x108</td>
<td>Old %ebx 0x100</td>
</tr>
<tr>
<td>0x108 0x104</td>
<td>Rtn addr 0x104</td>
</tr>
<tr>
<td>0x104 0x100</td>
<td>Old %ebp 0x108</td>
</tr>
</tbody>
</table>

CS429 Slideset 7: 36 Instruction Set Architecture II
### Understanding Swap (5)

<table>
<thead>
<tr>
<th><code>movl</code> 12(%ebp),%ecx</th>
<th># ecx = yp</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>movl</code> 8(%ebp),%edx</td>
<td># edx = xp</td>
</tr>
<tr>
<td><code>movl</code> (%ecx),%eax</td>
<td># eax = yp</td>
</tr>
<tr>
<td><code>movl</code> (%edx),%ebx</td>
<td># ebx = xp</td>
</tr>
<tr>
<td><code>movl</code> %eax,(%edx)</td>
<td># xp = eax</td>
</tr>
<tr>
<td><code>movl</code> %ebx,(%ecx)</td>
<td># yp = ebx</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>%eax</code></th>
<th>456</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%edx</code></td>
<td>0x124</td>
</tr>
<tr>
<td><code>%ecx</code></td>
<td>0x120</td>
</tr>
<tr>
<td><code>%ebx</code></td>
<td></td>
</tr>
<tr>
<td><code>%esi</code></td>
<td></td>
</tr>
<tr>
<td><code>%edi</code></td>
<td></td>
</tr>
<tr>
<td><code>%esp</code></td>
<td></td>
</tr>
<tr>
<td><code>%ebp</code></td>
<td>0x104</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>Offset</td>
</tr>
<tr>
<td>yp</td>
</tr>
<tr>
<td>xp</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>%ebp</td>
</tr>
<tr>
<td>-4</td>
</tr>
</tbody>
</table>
### Understanding Swap (6)

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

movl (%edx),%ebx  # ebx = *xp

movl %eax,(%edx)  # *xp = eax
movl %ebx,(%ecx)  # *yp = ebx
```

<table>
<thead>
<tr>
<th>%eax</th>
<th>456</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
<td>0x124</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ebx</td>
<td>123</td>
</tr>
<tr>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>yp</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>xp</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>0x124</td>
</tr>
<tr>
<td>456</td>
<td>0x120</td>
</tr>
<tr>
<td>0x11c</td>
<td>0x118</td>
</tr>
<tr>
<td>0x114</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x10c</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x104</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td></td>
</tr>
</tbody>
</table>

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

%ebp → 0 
Old %ebp 0x104
Old %ebx 0x100

---

CS429 Slideset 7: 38  Instruction Set Architecture II
Understanding Swap (7)

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

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

| %eax     | 456   |
| %edx     | 0x124 |
| %ecx     | 0x120 |
| %ebx     | 123   |
| %esi     |       |
| %edi     |       |
| %esp     |       |
| %ebp     | 0x104 |

yp 12 0x120 0x110
xp 8 0x124 0x10c
4 Rtn addr 0x108
%ebp +0 0x104
Old %ebp 0x104
Old %ebx 0x100
### Understanding Swap (8)

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

%eax     : 456
%edx     : 0x124
%ecx     : 0x120
%ebx     : 123
%esi     :
%edi     :
%esp     :
%ebp     : 0x104
```

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

Rtn addr: 0x100

yp: 12
xp: 8
Rtn addr: 0x100

<table>
<thead>
<tr>
<th>%ebp</th>
<th>Old %ebp</th>
<th>0x104</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x104</td>
<td></td>
</tr>
</tbody>
</table>

Old %ebp: 0x104
Indexed Addressing Modes

Most General Form:

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

- **D**: Constant “displacement” of 1, 2 or 4 bytes
- **Rb**: Base register, any of the 8 integer registers
- **Ri**: Index register, any except %esp (and probably not %ebp)
- **S**: Scale, one of 1, 2, 4 or 8.

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]]
<table>
<thead>
<tr>
<th>Type</th>
<th>Form</th>
<th>Operand value</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>$D$</td>
<td>$D$</td>
<td>Immediate</td>
</tr>
<tr>
<td>Register</td>
<td>$E_a$</td>
<td>$R[E_a]$</td>
<td>Register</td>
</tr>
<tr>
<td>Memory</td>
<td>$D$</td>
<td>$M[D]$</td>
<td>Absolute</td>
</tr>
<tr>
<td>Memory</td>
<td>$(E_a)$</td>
<td>$M[R[E_a]]$</td>
<td>Indirect</td>
</tr>
<tr>
<td>Memory</td>
<td>$D(E_b)$</td>
<td>$M[D + R[E_b]]$</td>
<td>Base + displacement</td>
</tr>
<tr>
<td>Memory</td>
<td>$(E_b, E_i)$,</td>
<td>$M[R[E_b] + R[E_i]]$</td>
<td>Indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$D(E_b, E_i)$,</td>
<td>$M[D + R[E_b] + R[E_i]]$</td>
<td>Indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$(E_b, E_i, s)$</td>
<td>$M[R[E_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$D(E_b, E_i, s)$</td>
<td>$M[D + R[E_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$(E_b, E_i, s)$</td>
<td>$M[R[E_b] + R[E_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$D(E_b, E_i, s)$</td>
<td>$M[D + R[E_b] + R[E_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
</tbody>
</table>

The scaling factor $s$ must be either 1, 2, 4, or 8.
Address Computation Example

<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>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>
**Form:** leal Src, Dest

- Src is address mode expression.
- Sets Dest to address denoted by the expression.

**Uses:**

- Computing address without doing a memory reference:
  - E.g., translation of `p = &x[i];`
- Computing arithmetic expressions of the form `x + k \times y`, where `i \in \{1, 2, 4, 8\}`
### Two operand instructions:

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<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</td>
</tr>
<tr>
<td>sarl Src, Dest</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>shr1 Src, Dest</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>xorl Src, Dest</td>
<td>Dsst = 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>
### One operand instructions:

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<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 = ~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:
    # set up
    pushl %ebp
    movl %esp, %ebp

    # body
    movl 8(%ebp), %eax
    movl 12(%ebp), %edx
    leal (%edx,%eax),%ecx
    leal (%edx,%edx,2),%edx
    sal $4,%edx
    addl 16(%ebp),%ecx
    leal 4(%edx,%eax),%eax
    imull %ecx,%eax

    # finish
    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;
}
```

**offset table:**

- offset 0: Old %ebp
- offset 4: Rtn addr
- offset 8: x
- offset 12: y
- offset 16: z

**assembly code:**

```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
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)
```

---

CS429 Slideset 7: 48  |  Instruction Set Architecture II
Understanding Arithmetic

```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;
}
```

```
# eax = x
  movl 8(%ebp), %eax

# edx = y
  movl 12(%ebp), %edx

# ecx = x+y (t1)
  leal (%edx,%eax),%ecx

# edx = 3*y
  leal (%edx,%edx,2),%edx

# edx = 48*y (t4)
  sall $4,%edx

# ecx = z+t1 (t2)
  addl 16(%ebp),%ecx

# eax = 4+t4+x (t5)
  leal 4(%edx,%eax),%eax

# eax = t5*t2 (rval)
  imull %ecx,%eax
```
Another Example

```c
int logical(int x, int y)
{
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

Note:

\[ 2^{13} = 8192; 2^{13} - 7 = 8185 \]

```
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
```

Note:

\[ 2^{13} = 8192; 2^{13} - 7 = 8185 \]