Machine-Level Programming: Basics

Fall 2012

Instructors:
Aykut & Erkut Erdem

Acknowledgement: The course slides are adapted from the slides prepared by R.E. Bryant, D.R. O’Hallaron, G. Kesden and Markus Püschel of Carnegie-Mellon Univ.
Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Intro to x86-64
- Complete addressing mode, address computation (leal)
- Arithmetic operations
Intel x86 Processors

- Totally dominate laptop/desktop/server 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!
    - In terms of speed. Less so for low power.
## 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>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>Pentium 4F</td>
<td>2004</td>
<td>125M</td>
<td>2800-3800</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Core 2</td>
<td>2006</td>
<td>291M</td>
<td>1060-3500</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Core i7</td>
<td>2008</td>
<td>731M</td>
<td>1700-3900</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></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
- First 64-bit processor, referred to as x86-64
- First multi-core Intel processor
- Four cores
# 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></td>
<td>286</td>
</tr>
<tr>
<td>X86-32/IA32</td>
<td>386</td>
</tr>
<tr>
<td></td>
<td>486</td>
</tr>
<tr>
<td>MMX</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
Moore’s Law

- The number of transistors on an integrated circuit will approximately double every two years.

*Gordon Moore (co-founder of the Intel)*

![Graph showing the growth of Intel microprocessor complexity over time. The graph includes points for different processors such as 8086, i386, Pentium, Pentium II, Pentium III, Pentium 4e, Core 2 Duo, and Core i7. The x-axis represents the year from 1975 to 2010, and the y-axis represents the number of transistors in billions (10^9 to 10^4). The line shows a steady increase in complexity over time.]
Intel x86 Processors, contd.

■ Machine Evolution
- 386 1985 0.3M
- Pentium 1993 3.1M
- Pentium/MMX 1997 4.5M
- Pentium Pro 1995 6.5M
- Pentium III 1999 8.2M
- Pentium 4 2001 42M
- Core 2 Duo 2006 291M
- Core i7 2008 731M

■ 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
- Two major steps: 1) support 32-bit 386. 2) support 64-bit x86-64
More Information

- Intel processors ([Wikipedia](https://en.wikipedia.org/wiki/Intel_processor))
- 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
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!

- All but low-end x86 processors support x86-64
  - But, lots of code still runs in 32-bit mode
Our Coverage

- **IA32**
  - The traditional x86

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

**Presentation**
- Book presents IA32 in Sections 3.1—3.12
- Covers x86-64 in 3.13
- We will cover both simultaneously
Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Intro to x86-64
- Complete addressing mode, address computation (leal)
- Arithmetic operations
Definitions

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

- **Microarchitecture**: Implementation of the architecture.
  - 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 -O1 p1.c p2.c -o p`
  - Use basic optimizations (`-O1`)
  - Put resulting binary in file `p`

```
C program (p1.c p2.c)
```

```
Asm program (p1.s p2.s)
```

```
Object program (p1.o p2.o)
```

```
Executable program (p)
```

Static libraries (.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
    popl %ebp
    ret
```

Some compilers use instruction “leave”

Obtain with command

```
gcc -O1 -S -m32 code.c
```

“−m32” flag generates code compatible with any IA32 machine on a X86-64 machine

Produces file code.s
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

<table>
<thead>
<tr>
<th>C declaration</th>
<th>Intel data type</th>
<th>Assembly code suffix</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>Byte</td>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>Word</td>
<td>w</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>Double word</td>
<td>l</td>
<td>4</td>
</tr>
<tr>
<td>long int</td>
<td>Double word</td>
<td>l</td>
<td>4</td>
</tr>
<tr>
<td>long long int</td>
<td>—</td>
<td>—</td>
<td>4</td>
</tr>
<tr>
<td>char *</td>
<td>Double word</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>float</td>
<td>Single precision</td>
<td>s</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>Double precision</td>
<td>l</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>Extended precision</td>
<td>t</td>
<td>10/12</td>
</tr>
</tbody>
</table>
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

\[ \text{0x401040} \ <\text{sum}> : \]

\[ \text{0x55} \]
\[ \text{0x89} \]
\[ \text{0xe5} \]
\[ \text{0x8b} \]
\[ \text{0x0c} \]
\[ \text{0x03} \]
\[ \text{0x45} \]
\[ \text{0x0c} \]
\[ \text{0x08} \]
\[ \text{0x5d} \]
\[ \text{0xc3} \]

- Total of 11 bytes
- Each instruction 1, 2, or 3 bytes
- Starts at address \text{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 \texttt{malloc, printf}
- Some libraries are dynamically linked
- Linking occurs when program begins execution

To obtain code .o, use command

```
gcc -O1 -m32 -c code.c
```
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 \( 0x80483ca \)

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

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

Similar to expression:
```c
x += y
```

More precisely:
```c
int eax;
int *ebp;
eax += ebp[2]
```

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

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

```
0x80483ca:  03 45 08
```

Disassembling Object Code

Disassembled

080483c4 <sum>:

- 80483c4: 55 push %ebp
- 80483c5: 89 e5 mov %esp,%ebp
- 80483c7: 8b 45 0c mov 0xc(%ebp),%eax
- 80483ca: 03 45 08 add 0x8(%ebp),%eax
- 80483cd: 5d pop %ebp
- 80483ce: c3 ret

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

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

Disassembled

Dump of assembler code for function sum:
0x080483c4 <sum+0>: push %ebp
0x080483c5 <sum+1>: mov %esp,%ebp
0x080483c7 <sum+3>: mov 0xc(%,ebp),%eax
0x080483ca <sum+6>: add 0x8(%,ebp),%eax
0x080483cd <sum+9>: pop %ebp
0x080483ce <sum+10>: ret

Within gdb Debugger

gdb p
disassemble sum
  Disassemble procedure
x/11xb sum
  Examine the 11 bytes starting at sum
What Can be Disassembled?

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

```
% objdump -d WINWORD.EXE

WINWORD.EXE: file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:  55 push %ebp
30001001:  8b  ec mov %esp,%ebp
30001003:  6a  ff push $0xffffffff
30001005:  68  90 10 00 30 push $0x30001090
3000100a:  68  91  dc 4c 30 push $0x304cdc91
```
Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Intro to x86-64
- Complete addressing mode, address computation (leal)
- Arithmetic operations
Integer Registers (IA32)

- `%eax` (general purpose)
  - `%ax` (accumulation)
  - `%ah`
  - `%al`

- `%ecx` (general purpose)
  - `%cx` (counter)
  - `%ch`
  - `%cl`

- `%edx` (general purpose)
  - `%dx` (data)
  - `%dh`
  - `%dl`

- `%ebx` (general purpose)
  - `%bx` (base)
  - `%bh`
  - `%bl`

- `%esi` (general purpose)
  - `%si` (source index)

- `%edi` (general purpose)
  - `%di` (destination index)

- `%esp` (16-bit virtual registers)
  - `%sp` (stack pointer)

- `%ebp` (16-bit virtual registers)
  - `%bp` (base pointer)

Origin (mostly obsolete):
- `accumulate`
- `counter`
- `data`
- `base`
- `source`
- `index`
Moving Data: IA32

Moving Data

\[ \text{mov\{b,w,l\} Source, Dest} \quad (b: \text{byte, } w: \text{word, } l: \text{double word}) \]

- Move byte (1 byte)
  \[ \text{movb Source, Dest} \]

- Move word (2 bytes)
  \[ \text{movw Source, Dest} \]

- Move double word (4 bytes)
  \[ \text{movl Source, Dest} \]
Moving Data: IA32

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

- **Operand Types**
  
  - **Immediate**: Constant integer data
    
    - Example: \$0\times400, \$-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”
## 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>Mem</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>Mem</td>
<td>Reg</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>
</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 8(%ebp), %edx
    movl 12(%ebp), %ecx
    movl (%edx), %ebx
    movl (%ecx), %eax
    movl %eax, (%edx)
    movl %ebx, (%ecx)
    popl %ebx
    popl %ebp
    ret
```

Set Up

Body

Finish
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 8(%ebp), %edx
    movl 12(%ebp), % ecx
    movl (%edx), %ebx
    movl (%ecx), %eax
    movl %eax, (%edx)
    movl %ebx, (%ecx)
    popl %ebx
    popl %ebp
    ret
```

Set Up

Body

Finish
### Understanding Swap

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

#### Register Values
- **%edx**: `xp`
- **%ecx**: `yp`
- **%ebx**: `t0`
- **%eax**: `t1`

#### 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>

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

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

### Variables
- `%eax`  
- `%edx`  
- `%ecx`  
- `%ebx`  
- `%esi`  
- `%edi`  
- `%esp`  
- `%ebp`  

| `%ebp` | 0x104 |

### Assembly Code

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

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

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

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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</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>
<tr>
<td>0</td>
<td>0x108</td>
</tr>
<tr>
<td>-4</td>
<td>0x104</td>
</tr>
<tr>
<td></td>
<td>0x100</td>
</tr>
</tbody>
</table>

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

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

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

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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x124</td>
</tr>
<tr>
<td></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>yp</td>
<td>12</td>
</tr>
<tr>
<td>xp</td>
<td>8</td>
</tr>
<tr>
<td>Rtn adr</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0x104</td>
</tr>
<tr>
<td></td>
<td>0x100</td>
</tr>
</tbody>
</table>

- \texttt{movl 8(\%ebp), \%edx}  \# edx = xp
- \texttt{movl 12(\%ebp), \%ecx}  \# ecx = yp
- \texttt{movl (%edx), \%ebx}  \# ebx = *xp (t0)
- \texttt{movl (%ecx), \%eax}  \# eax = *yp (t1)
- \texttt{movl \%eax, (%edx)}  \# *xp = t1
- \texttt{movl \%ebx, (%ecx)}  \# *yp = t0
Understanding Swap

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</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>
<tr>
<td>0</td>
<td>0x11c</td>
</tr>
<tr>
<td>-4</td>
<td>0x10c</td>
</tr>
</tbody>
</table>

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

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

Address Table:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</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>0x114</td>
</tr>
<tr>
<td></td>
<td>0x110</td>
</tr>
<tr>
<td></td>
<td>0x118</td>
</tr>
<tr>
<td></td>
<td>0x11c</td>
</tr>
<tr>
<td>0</td>
<td>0x108</td>
</tr>
<tr>
<td>-4</td>
<td>0x100</td>
</tr>
<tr>
<td></td>
<td>0x104</td>
</tr>
<tr>
<td></td>
<td>0x108</td>
</tr>
<tr>
<td></td>
<td>0x10c</td>
</tr>
</tbody>
</table>

Code:

```
movl 8(%ebp), %edx      # edx = xp
movl 12(%ebp), %ecx     # ecx = yp
movl (%edx), %ebx       # ebx = *xp (t0)
movl (%ecx), %eax       # eax = *yp (t1)
movl %eax, (%edx)       # *xp = t1
movl %ebx, (%ecx)       # *yp = t0
```
Complete Memory 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`
  - Unlikely you’d use `%ebp`, either
- **S:** Scale: 1, 2, 4, or 8 (*why these numbers?*)

**Special Cases**

- \((Rb, Ri)\) \rightarrow Mem[Reg[Rb]+Reg[Ri]]
- \(D(Rb, Ri)\) \rightarrow Mem[Reg[Rb]+Reg[Ri]+D]
- \((Rb, Ri, S)\) \rightarrow Mem[Reg[Rb]+S*Reg[Ri]]
Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Intro to x86-64
- Complete addressing mode, address computation (leal)
- Arithmetic operations
# Data Representations: IA32 + x86-64

## Sizes of C Objects (in Bytes)

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Generic 32-bit</th>
<th>Intel IA32</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long int</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>8</td>
<td>10/12</td>
<td>16</td>
</tr>
<tr>
<td>char *</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

– Or any other pointer
## x86-64 Integer Registers

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%ebx</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
</tr>
<tr>
<td>%r8</td>
<td>%r8d</td>
</tr>
<tr>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<td>%r15</td>
<td>%r15d</td>
</tr>
</tbody>
</table>

- Extend existing registers. Add 8 new ones.
- Make %ebp/%rbp general purpose
Instructions

- Long word 1 (4 Bytes) ↔ Quad word q (8 Bytes)

- New instructions:
  - movl ➔ movq
  - addl ➔ addq
  - sall ➔ salq
  - etc.

- 32-bit instructions that generate 32-bit results
  - Set higher order bits of destination register to 0
  - Example: addl
32-bit code for swap

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

`swap:`

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

- **Body**
  - `movl 8(%ebp), %edx`
  - `movl 12(%ebp), %ecx`
  - `movl (%edx), %ebx`
  - `movl (%ecx), %eax`
  - `movl %eax, (%edx)`
  - `movl %ebx, (%ecx)`

- **Finish**
  - `popl %ebx`
  - `popl %ebp`
  - `ret`
64-bit code for swap

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

swap:
    movl (%rdi), %edx
    movl (%rsi), %eax
    movl %eax, (%rdi)
    movl %edx, (%rsi)
    ret

- **Operands passed in registers (why useful?)**
  - 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
64-bit code for long int swap

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

64-bit data

- Data held in registers `%rax` and `%rdx`
- `movq` operation
  - “q” stands for quad-word
Machine Programming I: Summary

- History of Intel processors and architectures
  - Evolutionary design leads to many quirks and artifacts

- C, assembly, machine code
  - Compiler must transform statements, expressions, procedures into low-level instruction sequences

- Assembly Basics: Registers, operands, move
  - The x86 move instructions cover wide range of data movement forms

- Intro to x86-64
  - A major departure from the style of code seen in IA32
Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Intro to x86-64
- Complete addressing mode, address computation (leal)
- Arithmetic operations
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>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>

<table>
<thead>
<tr>
<th>%edx</th>
<th>0xf000</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>0x0100</td>
</tr>
</tbody>
</table>
# Address Computation Examples

<table>
<thead>
<tr>
<th>%edx</th>
<th>0xf000</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>0x0100</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 + 0x0100</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 = &x[i];`
  - Computing arithmetic expressions of the form `x + k*y`
    - `k = 1, 2, 4,` or `8`

- **Example**

  ```c
  int mul12(int x) {
    return x*12;
  }
  ```

  Converted to ASM by compiler:

  ```asm
  leal (%eax,%eax,2), %eax ; t <- x+x*2
  sall $2, %eax           ; return t<<2
  ```
Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Intro to x86-64
- Complete addressing mode, address computation (leal)
- 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>sarll</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>

- **Watch out for argument order!**

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

- **One Operand Instructions**
  
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>incl</td>
<td>Dest</td>
<td>Dest = Dest + 1</td>
</tr>
<tr>
<td>decl</td>
<td>Dest</td>
<td>Dest = Dest - 1</td>
</tr>
<tr>
<td>negl</td>
<td>Dest</td>
<td>Dest = - Dest</td>
</tr>
<tr>
<td>notl</td>
<td>Dest</td>
<td>Dest = ~Dest</td>
</tr>
</tbody>
</table>

- See book for more instructions
Arithmetic Expression Example

```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;
}
```

```
arith:
pushl %ebp
movl %esp, %ebp

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

popl %ebp
ret
```

Set Up

Body

Finish
Understanding arith

```c
int arith(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

```assembly
movl 8(%ebp), %ecx
movl 12(%ebp), %edx
leal (%edx,%edx,2), %eax
sall $4, %eax
leal 4(%ecx,%eax), %eax
addl %ecx, %edx
addl 16(%ebp), %edx
imull %edx, %eax
```
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;
}
```

Stack:

- Offset 16: z
- Offset 12: y
- Offset 8: x
- Offset 4: Rtn Addr
- Offset 0: Old %ebp

Machine Code:

- `movl 8(%ebp), %ecx` # ecx = x
- `movl 12(%ebp), %edx` # edx = y
- `leal (%edx,%edx,2), %eax` # eax = y*3
- `sall $4, %eax` # eax *= 16 (t4)
- `leal 4(%ecx,%eax), %eax` # eax = t4 + x+4 (t5)
- `addl %ecx, %edx` # edx = x+y (t1)
- `addl 16(%ebp), %edx` # edx += z (t2)
- `imull %edx, %eax` # eax = t2 * t5 (rval)
Observations about 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;
}
```

- Instructions in different order from C code
- Some expressions require multiple instructions
- Some instructions cover multiple expressions
- Get exact same code when compile:
- 
  \[(x+y+z) \times (x+4+48*y)\]
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;
}
```

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

**logical:**

```assembly
pushl %ebp
movl %esp,%ebp

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

popl %ebp
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 12(%ebp),%eax`
  - `xorl 8(%ebp),%eax`
  - `sarl $17,%eax`
  - `andl $8185,%eax`
- **Finish**
  - `popl %ebp`
  - `ret`

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

logica:  
pushl %ebp
movl %esp,%ebp

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

popl %ebp  
ret

\[ 2^{13} = 8192, 2^{13} - 7 = 8185 \]
Reading assignment

“An Introduction to 64-bit Computing and x86-64”, Jon Stokes, arstechnica.com, 2002