ASSMEBLY BASICS

Spring, 2013
Euiseong Seo
(euiseong@skku.edu)

This Powerpoint slides are modified from its original version available at http://www.cs.cmu.edu/afs/cs/academic/class/15213-s09/www/lectures/ppt-sources/
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
- Intro to x86-64
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>First 16-bit processor. Basis for IBM PC &amp; DOS</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1MB address space</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>First 32 bit processor, referred to as IA32</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added “flat addressing”</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Capable of running Unix</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>32-bit Linux/gcc uses no instructions introduced in later models</td>
<td></td>
</tr>
<tr>
<td>Pentium 4F</td>
<td>2005</td>
<td>230M</td>
<td>2800-3800</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>First 64-bit processor</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Meanwhile, Pentium 4s (Netburst arch.) phased out in favor of “Core” line</td>
<td></td>
</tr>
</tbody>
</table>
## 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></td>
<td>Core 2 Duo</td>
</tr>
<tr>
<td></td>
<td>Core i7</td>
</tr>
</tbody>
</table>

IA: often redefined as latest Intel architecture
Machine Evolution

- 486 1989 1.9M
- 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

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></td>
<td></td>
<td></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>
**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 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
- Intro to x86-64
**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
Assembler 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

- **Stack**

---

- **CPU**
  - **PC**
  - **Registers**
  - **Condition Codes**

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

- **Addresses**
  - Data
  - Instructions
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) --> 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) --> Static libraries (.a)
```
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

```
/usr/local/bin/gcc -O1 -S code.c
```

Produces file `code.s`
Data Types in Assembly

- "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
Operations in Assembly

- 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
int t = x + y;
```

**C Code**
- Add two signed integers

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

**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**

- **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

```
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
```
Within **gdb** Debugger

- **gdb p**
- disassemble **sum**
- Disassemble procedure
- **x/13b sum**
- Examine the 13 bytes starting at **sum**

<table>
<thead>
<tr>
<th>Disassembled</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401040 &lt;sum&gt;: push %ebp</td>
</tr>
<tr>
<td>0x401041 &lt;sum+1&gt;: mov %esp, %ebp</td>
</tr>
<tr>
<td>0x401043 &lt;sum+3&gt;: mov 0xc(%ebp), %eax</td>
</tr>
<tr>
<td>0x401046 &lt;sum+6&gt;: add 0x8(%ebp), %eax</td>
</tr>
<tr>
<td>0x401049 &lt;sum+9&gt;: mov %ebp, %esp</td>
</tr>
<tr>
<td>0x40104b &lt;sum+11&gt;: pop %ebp</td>
</tr>
<tr>
<td>0x40104c &lt;sum+12&gt;: ret</td>
</tr>
<tr>
<td>0x40104d &lt;sum+13&gt;: lea 0x0(%esi), %esi</td>
</tr>
</tbody>
</table>

**Object**

<table>
<thead>
<tr>
<th>0x401040:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x55</td>
</tr>
<tr>
<td>0x89</td>
</tr>
<tr>
<td>0xe5</td>
</tr>
<tr>
<td>0x8b</td>
</tr>
<tr>
<td>0x45</td>
</tr>
<tr>
<td>0x0c</td>
</tr>
<tr>
<td>0x03</td>
</tr>
<tr>
<td>0x45</td>
</tr>
<tr>
<td>0x08</td>
</tr>
<tr>
<td>0x89</td>
</tr>
<tr>
<td>0xec</td>
</tr>
<tr>
<td>0x5d</td>
</tr>
<tr>
<td>0xc3</td>
</tr>
</tbody>
</table>
## What Can be Disassembled?

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

```plaintext
% 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 $0x304cdcc91
```
Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Intro to x86-64
Intel uses “word” to refer to a 16-bit data type

- 32-bit quantities as double words and 64-bit quantities as quad words

<table>
<thead>
<tr>
<th>C declaration</th>
<th>Intel data type</th>
<th>GAS suffix</th>
<th>Size (B)</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>unsigned</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>unsigned long</td>
<td>Double word</td>
<td>l</td>
<td>4</td>
</tr>
<tr>
<td>char *</td>
<td>Double word</td>
<td>l</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>

- mov: movb, movw, movl
First six registers are general purpose
  - No restriction placed on their use
Final two registers contain pointers to important places in the stack
  - The should only be altered by a set of conventional stack management
Backward compatibility
  - Low-order of 2 bytes of first 4 registers
    - Independently read or written by byte operation instructions
    - To support 8086 operations
INTEGER REGISTER ANATOMY
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
### Operands Types

- **Moving Data**
  - `movl Source, 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`
  - **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>Destination</th>
<th>Example</th>
<th>C analogy</th>
</tr>
</thead>
<tbody>
<tr>
<td>immediate (Imm)</td>
<td>register</td>
<td><code>movl $0x4,%eax</code></td>
<td><code>temp = 0x4;</code></td>
</tr>
<tr>
<td></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></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>

- Cannot do memory-memory transfer with a single instruction
**Simple Memory Addressing Modes**

- **Normal**
  - (R)
    - Mem[ Reg[ R ]]
  - Register R specifies memory address
  - `movl (%ecx),%eax`

- **Displacement**
  - D(R)
    - Mem[ Reg[ R ] + D ]
  - Register R specifies start of memory region
  - Constant displacement D specifies offset
  - `movl 8(%ebp),%edx`
Indexed Addressing Modes

► Most General Form
  ° \( D(R_b, R_i, S) \)
    • \( \text{Mem}[\text{Reg}[R_b] + S \times \text{Reg}[R_i] + D] \)
  ° \( D \) (constant) : displacement 1, 2, or 4 bytes
  ° \( R_b \) (base register) : Any of 8 integer registers
  ° \( R_i \) (index register) : Any, except for \%esp
    • Unlikely you’d use \%ebp, either
  ° \( S \) (scale) : 1, 2, 4, or 8

► Special Cases
  ° \( (R_b, R_i) \Rightarrow \text{Mem}[\text{Reg}[R_b] + \text{Reg}[R_i]] \)
  ° \( D(R_b, R_i) \Rightarrow \text{Mem}[\text{Reg}[R_b] + \text{Reg}[R_i]+D] \)
  ° \( (R_b, R_i, S) \Rightarrow \text{Mem}[\text{Reg}[R_b] + S \times \text{Reg}[R_i]] \)
### Operand Forms

<table>
<thead>
<tr>
<th>Type</th>
<th>Form</th>
<th>Operand Value</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>$Imm</td>
<td>Imm</td>
<td>Immediate</td>
</tr>
<tr>
<td>Register</td>
<td>$E_a</td>
<td>R[ E_a ]</td>
<td>Register</td>
</tr>
<tr>
<td>Memory</td>
<td>Imm (E_b)</td>
<td>M[ Imm ]</td>
<td>Absolute</td>
</tr>
<tr>
<td>Memory</td>
<td>(E_b, E_i)</td>
<td>M[ Imm + R[ E_b ] ]</td>
<td>Indirect</td>
</tr>
<tr>
<td>Memory</td>
<td>Imm (E_b, E_i)</td>
<td>M[ Imm + R[ E_b ] + R[ E_i ] ]</td>
<td>Base + Displacement</td>
</tr>
<tr>
<td>Memory</td>
<td>(E_i, s)</td>
<td>M[ R[ E_i ] × s ]</td>
<td>Indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>Imm (, E_i, s)</td>
<td>M[ Imm + R[ E_i ] × s ]</td>
<td>Indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>(E_b, E_i, s)</td>
<td>M[ R[ E_b ] + R[ E_i ] × s ]</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>Imm (E_b, E_i, s)</td>
<td>M[ Imm + R[ E_b ] + R[ E_i ] × s ]</td>
<td>Scaled indexed</td>
</tr>
</tbody>
</table>

Where:
- $E_a$: Immediate value
- $E_b$: Base register
- $E_i$: Index register
- $s$: Scale factor
## Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8 (%edx)</td>
<td>0xf000 + 0x8</td>
<td>0xf008</td>
</tr>
<tr>
<td>(%edx,%ecx)</td>
<td>0xf000 + 0x100</td>
<td>0xf100</td>
</tr>
<tr>
<td>(%edx,%ecx,4)</td>
<td>0xf000 + 4 × 0x100</td>
<td>0xf400</td>
</tr>
<tr>
<td>0x80 (%edx,2)</td>
<td>2 × 0xf000 + 0x80</td>
<td>0x1e080</td>
</tr>
</tbody>
</table>

Practice Problem 3.1
## Data Movements Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>movl S,D</code></td>
<td>D ← S</td>
<td>Move double word</td>
</tr>
<tr>
<td><code>movw S,D</code></td>
<td>D ← S</td>
<td>Move word</td>
</tr>
<tr>
<td><code>movb S,D</code></td>
<td>D ← S</td>
<td>Move byte</td>
</tr>
<tr>
<td><code>movsbl S,D</code></td>
<td>D ← SignExtend(S)</td>
<td>Move sign-extended byte</td>
</tr>
<tr>
<td><code>movzbl S,D</code></td>
<td>D ← ZeroExtend(S)</td>
<td>Move zero-extended byte</td>
</tr>
<tr>
<td><code>pushl S</code></td>
<td>R[ %esp ] ← R[ %esp ] - 4; M[R[ %esp ]] ← S</td>
<td>Push</td>
</tr>
<tr>
<td><code>popl D</code></td>
<td>D ← M[R[ %esp ]]; R[ %esp ] ← R[ %esp ] + 4</td>
<td>Pop</td>
</tr>
</tbody>
</table>

### Exercise

%dh = 0x8d, %eax = 98765432

- `movb %dh, %eax` %eax = 0x9876548D
- `mobsbl %dh, %eax` %eax = 0xFFFFFFF8D
- `mobzbl %dh, %eax` %eax = 0x0000008D
### Stack Operation

- **Initially**
  - `%eax` 0x123
  - `%edx` 0
  - `%esp` 0x108

- **pushl %eax**
  - `%eax` 0x123
  - `%edx` 0
  - `%esp` 0x104

- **potrl %edx**
  - `%eax` 0x123
  - `%edx` 0x123
  - `%esp` 0x108
# 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
```
void swap(int *xp, int *yp) {
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}

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>
<th>%esi</th>
<th>%edi</th>
<th>%esp</th>
<th>%ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x104</td>
<td></td>
</tr>
</tbody>
</table>

Offset

- yp: 12
- xp: 8
- %ebp: 0
- %eax: -4

Address

| 123 | 0x124 |
| 456 | 0x120 |
| 456 | 0x11c |
| 0x118 |
| 0x114 |
| 0x10c |
| Rtn adr |
| 0x108 |
| 0x104 |
| 0x100 |

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**

```
%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
```

---

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>xp</td>
<td>0x124</td>
</tr>
<tr>
<td>yp</td>
<td>0x120</td>
</tr>
<tr>
<td>Rtn adr</td>
<td>0x108</td>
</tr>
<tr>
<td></td>
<td>0x104</td>
</tr>
<tr>
<td></td>
<td>0x100</td>
</tr>
</tbody>
</table>

---

**Sungkyunkwan University**
Understanding Swap

%eax
%edx 0x124
%ecx 0x120
%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**

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x1120</td>
</tr>
<tr>
<td>%edx</td>
<td>0x118</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x11c</td>
</tr>
<tr>
<td>%ebx</td>
<td>123</td>
</tr>
<tr>
<td>%esi</td>
<td>0x10c</td>
</tr>
<tr>
<td>%edi</td>
<td>0x108</td>
</tr>
<tr>
<td>%esp</td>
<td>0x104</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x104</td>
</tr>
<tr>
<td>4</td>
<td>0x108</td>
</tr>
<tr>
<td>8</td>
<td>0x110</td>
</tr>
<tr>
<td>12</td>
<td>0x120</td>
</tr>
</tbody>
</table>

```asm
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 |

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>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>

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp</td>
<td>12</td>
</tr>
<tr>
<td>xp</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td>Rtn adr</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>-4</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 |

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
<td>456</td>
</tr>
<tr>
<td>0x120</td>
<td>123</td>
</tr>
<tr>
<td>0x11c</td>
<td>0</td>
</tr>
<tr>
<td>0x118</td>
<td>8</td>
</tr>
<tr>
<td>0x114</td>
<td>4</td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x124</td>
<td>12</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>

```
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
```
Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Intro to x86-64
## 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
Long word \( l \) (4 Bytes) \( \leftrightarrow \) Quad word \( q \) (8 Bytes)

New instructions:
- \( \text{movl} \rightarrow \text{movq} \)
- \( \text{addl} \rightarrow \text{addq} \)
- \( \text{sall} \rightarrow \text{salq} \)
- etc.

32-bit instructions that generate 32-bit results
- Set higher order bits of destination register to 0
- Example: \( \text{addl} \)
32-BIT CODE FOR SWAP

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

**Set Up**

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

**Body**

```
movl 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
```
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;
}
```

- **Set Up**
  - `movq (%rdi), %rdx`
  - `movq (%rsi), %rax`

- **Body**
  - `movq %rax, (%rdi)`
  - `movq %rdx, (%rsi)`

- **Finish**
  - `ret`

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