

# IF2130 – Organisasi dan Arsitektur Komputer

sumber: Greg Kesden, CMU 15-213, 2012

Machine-Level Programming: Basics

Achmad Imam Kistijantoro ([imam@staff.stei.itb.ac.id](mailto:imam@staff.stei.itb.ac.id))

Robithoh Annur ([robithoh@staff.stei.itb.ac.id](mailto:robithoh@staff.stei.itb.ac.id))

Bara Timur ([bara@staff.stei.itb.ac.id](mailto:bara@staff.stei.itb.ac.id))

Monterico Adrian ([monterico@staff.stei.itb.ac.id](mailto:monterico@staff.stei.itb.ac.id))

# Dasar Bahasa Pemrograman Level Mesin

---

- ▶ Sejarah arsitektur dan prosesor Intel
- ▶ C, assembly, machine code
- ▶ Assembly Basics: Registers, operands, move
- ▶ Intro to x86-64



# 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

---

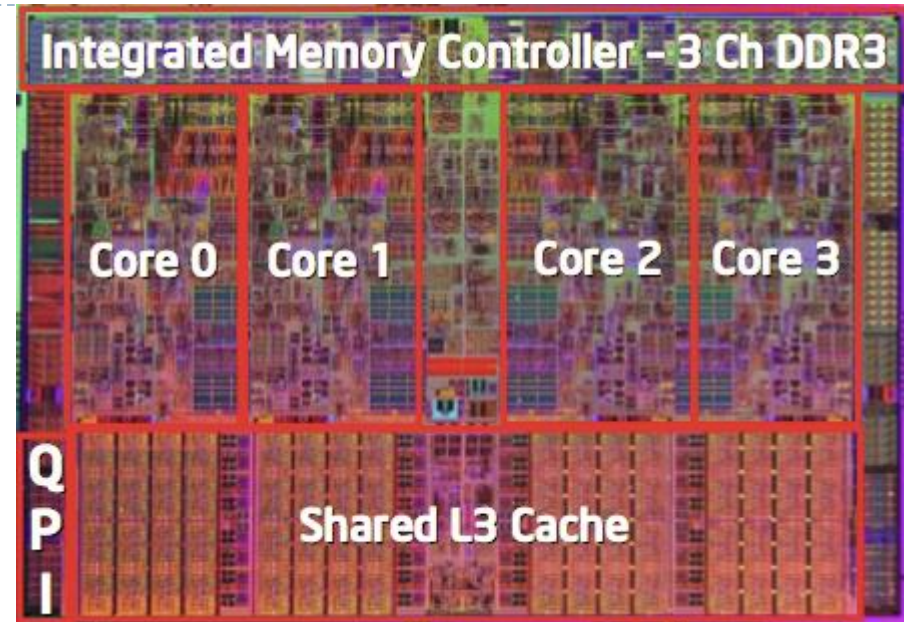
<i>Name</i>	<i>Date</i>	<i>Transistors</i>	<i>MHz</i>
▶ 8086	1978	29K	5-10
▶ First 16-bit Intel processor. Basis for IBM PC & DOS			
▶ 1MB address space			
▶ 386	1985	275K	16-33
▶ First 32 bit Intel processor , referred to as IA32			
▶ Added “flat addressing”, capable of running Unix			
▶ Pentium 4	2004	125M	2800-3800
▶ First 64-bit Intel processor, referred to as x86-64			
▶ Core 2	2006	291M	1060-3500
▶ First multi-core Intel processor			
▶ Core i7	2008	731M	1700-3900
▶ Four cores			



# Intel x86 Processors, cont.

## ▶ Machine Evolution

▶ 386	1985	0.3M
▶ 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
▶ Core i7	2008	731M



## ▶ Added Features

- ▶ Instructions to support multimedia operations
- ▶ Instructions to enable more efficient conditional operations
- ▶ Transition from 32 bits to 64 bits
- ▶ More cores

# 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
- ▶ `shark> gcc -m32 hello.c`

## ▶ x86-64

- ▶ The emerging standard
- ▶ `shark> gcc hello.c`
- ▶ `shark> gcc -m64 hello.c`

## ▶ 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



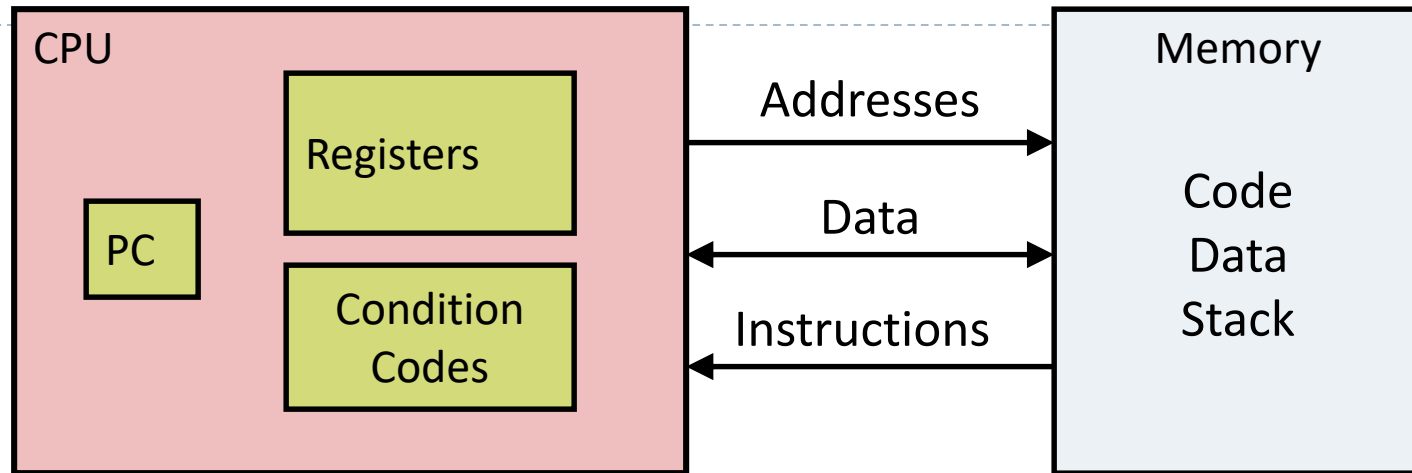
# Definitions

---

- ▶ **Architecture:** (also ISA: instruction set architecture) 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



# 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**

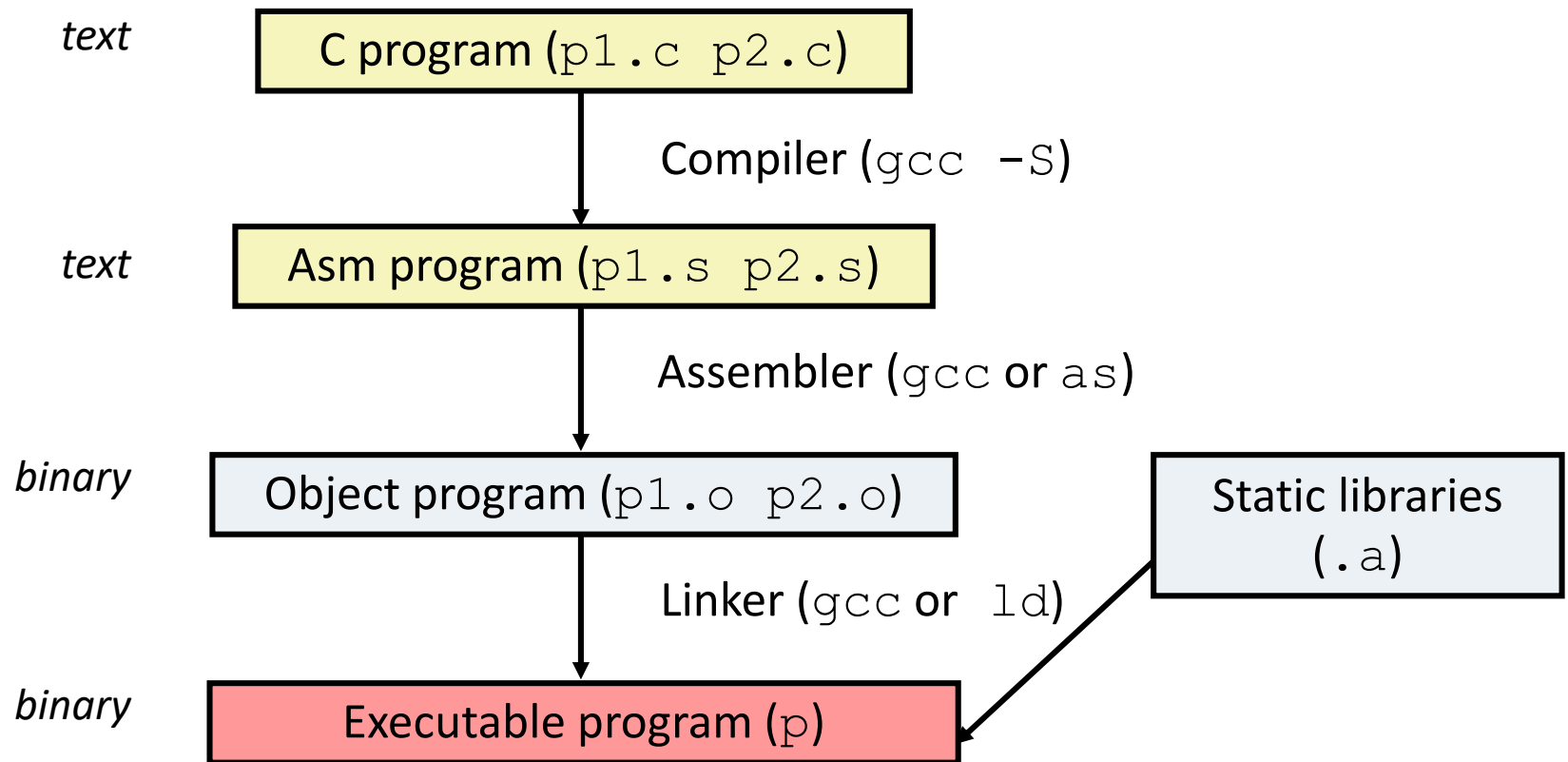
- ▶ Store status information about most recent arithmetic operation
- ▶ Used for conditional branching

### ▶ **Memory**

- ▶ Byte addressable array
- ▶ Code and user data
- ▶ Stack 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`



# Compiling Into Assembly

## C Code

```
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
```

Obtain with command

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

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



# 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

0x5d

0xc3

- Total of 11 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

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

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

Similar to expression:

```
x += y
```

More precisely:

```
int eax;
```

```
int *ebp;
```

```
eax += ebp[2]
```

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

## ▶ 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**

# 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?

---

```
% 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

# Today: Machine Programming I: Basics

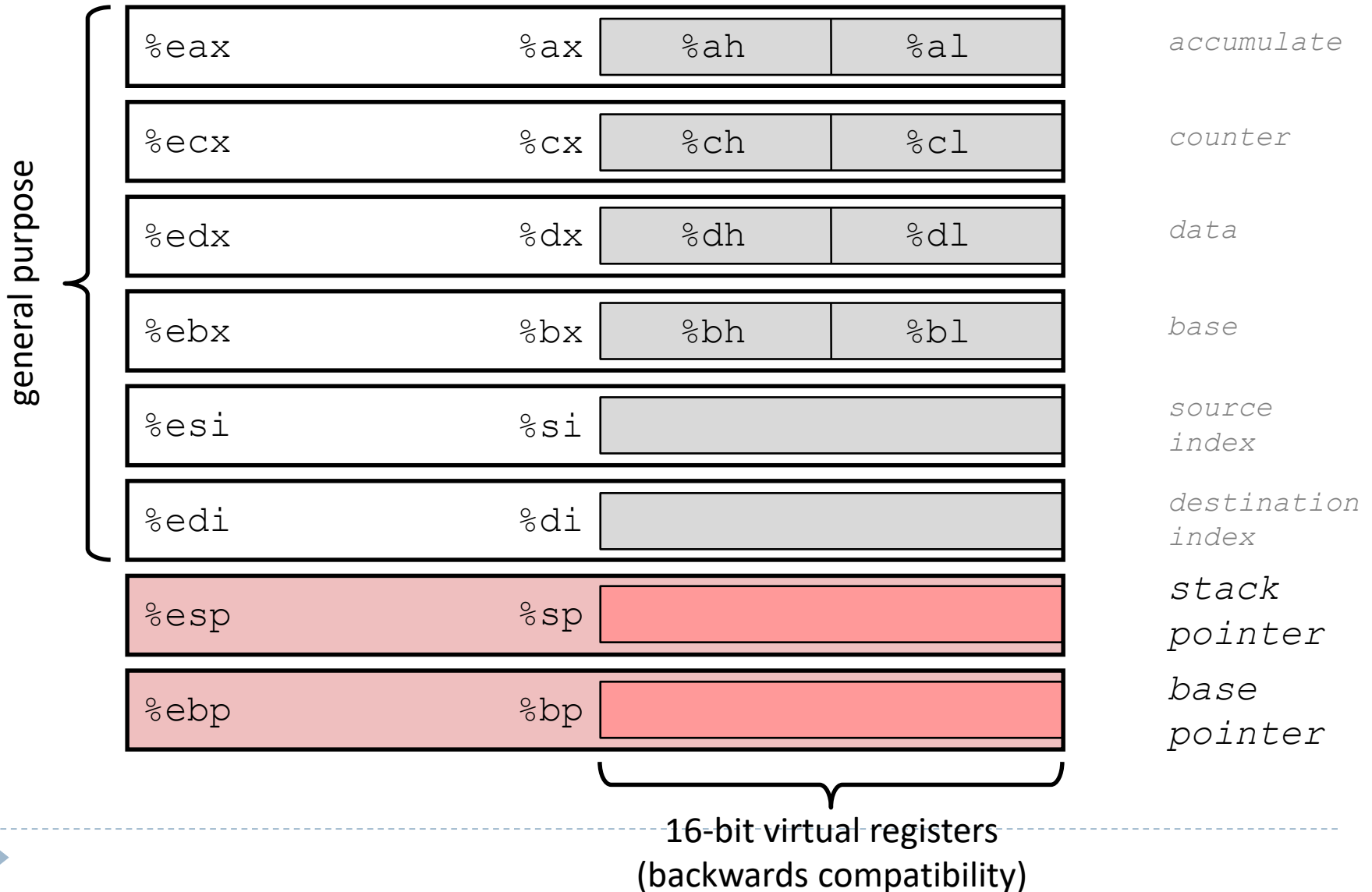
---

- ▶ History of Intel processors and architectures
- ▶ C, assembly, machine code
- ▶ **Assembly Basics: Registers, operands, move**
- ▶ Intro to x86-64



# Integer Registers (IA32)

Origin  
(mostly obsolete)



# Moving Data: IA32

---

## ▶ 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`
  - ▶ 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”

`%eax`

`%ecx`

`%edx`

`%ebx`

`%esi`

`%edi`

`%esp`

`%ebp`



# movl Operand Combinations

	Source	Dest	Src, Dest	C Analog
movl	Imm	Reg	movl \$0x4, %eax	temp = 0x4;
		Mem	movl \$-147, (%eax)	*p = -147;
	Reg	Reg	movl %eax, %edx	temp2 = temp1;
		Mem	movl %eax, (%edx)	*p = temp;
	Mem	Reg	movl (%eax), %edx	temp = *p;

*Cannot do memory-memory transfer with a single instruction*





# Simple Memory Addressing Modes

---

## ▶ Normal (R) Mem[Reg[R]]

- ▶ Register R specifies memory address
- ▶ Aha! Pointer dereferencing in C

```
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
```



# Using Simple Addressing Modes

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

swap:

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

} Set Up

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

} Body

```
popl  %ebx
popl  %ebp
ret
```

} Finish



# Using Simple Addressing Modes

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

swap:

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

} Set Up

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

} Body

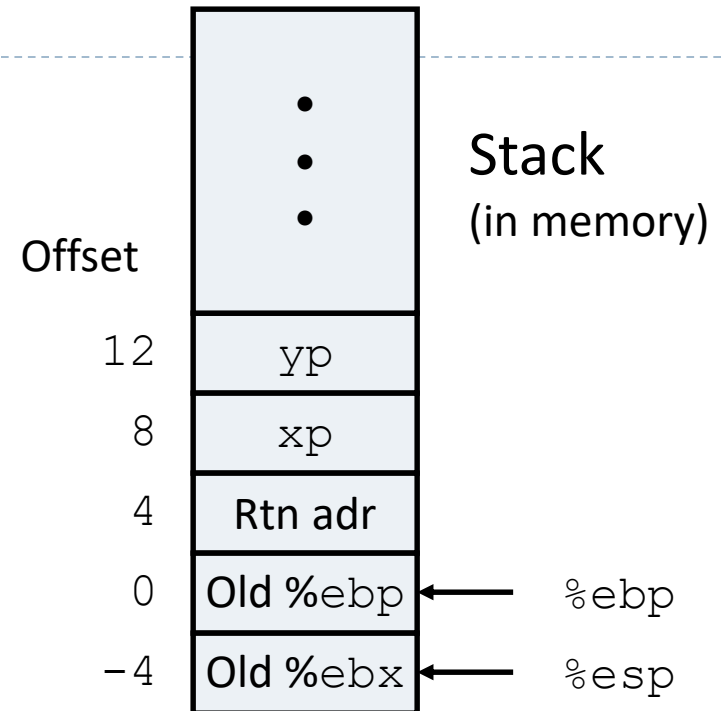
```
popl  %ebx
popl  %ebp
ret
```

} Finish



# Understanding Swap

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



Register	Value
%edx	xp
%ecx	yp
%ebx	t0
%eax	t1

```
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	
%ecx	
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

		Address
		123 0x124
		456 0x120
		0x11c
		0x118
		0x114
yp	12	0x120 0x110
xp	8	0x124 0x10c
	4	Rtn adr 0x108
%ebp	0	0x104
	-4	0x100

```

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

		Address
		123
		0x124
		456
		0x120
		0x11c
		0x118
		0x114
yp	12	0x120
		0x110
xp	8	0x124
		0x10c
	4	Rtn adr
		0x108
%ebp	0	
		0x104
	-4	
		0x100

```

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

		Address
		123
		456
yp	12	0x120
xp	8	0x124
	4	Rtn adr
%ebp	0	
	-4	

```

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	123
%esi	
%edi	
%esp	
%ebp	0x104

		Address
		0x124
		0x120
		0x11c
		0x118
		0x114
yp	12	0x120
xp	8	0x124
	4	Rtn adr
%ebp	0	
	-4	
		0x104
		0x100

```

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
		123
		0x124
		456
		0x120
		0x11c
		0x118
		0x114
yp	12	0x120
		0x110
xp	8	0x124
		0x10c
	4	Rtn adr
		0x108
%ebp	0	
		0x104
	-4	
		0x100

```

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
		0x124
		0x120
		0x11c
		0x118
		0x114
yp	12	0x120
xp	8	0x124
	4	Rtn adr
%ebp	0	
	-4	
		0x104
		0x100

```

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
		456 0x124
		123 0x120
		0x11c
		0x118
		0x114
yp	12	0x120 0x110
xp	8	0x124 0x10c
	4	Rtn adr 0x108
%ebp	0	0x104
	-4	0x100

```

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

$D(Rb, Ri, S)$                        $Mem[Reg[Rb] + S * Reg[Ri] + D]$

- ▶ 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

$(Rb, Ri)$                        $Mem[Reg[Rb] + Reg[Ri]]$

$D(Rb, Ri)$                        $Mem[Reg[Rb] + Reg[Ri] + D]$

$(Rb, Ri, S)$                        $Mem[Reg[Rb] + S * Reg[Ri]]$

---

# Today: Machine Programming I: Basics

---

- ▶ History of Intel processors and architectures
- ▶ C, assembly, machine code
- ▶ Assembly Basics: Registers, operands, move
- ▶ Intro to x86-64



# Data Representations: IA32 + x86-64

## ► Sizes of C Objects (in Bytes)

<i>C Data Type</i>	<i>Generic 32-bit</i>	<i>Intel IA32</i>	<i>x86-64</i>
► unsigned	4	4	4
► int	4	4	4
► long int	4	4	8
► char	1	1	1
► short	2	2	2
► float	4	4	4
► double	8	8	8
► long double	8	10/12	10/16
► char *	4	4	8

□ *Or any other pointer*

# x86-64 Integer Registers

<b>%rax</b>	<b>%eax</b>
<b>%rbx</b>	<b>%ebx</b>
<b>%rcx</b>	<b>%ecx</b>
<b>%rdx</b>	<b>%edx</b>
<b>%rsi</b>	<b>%esi</b>
<b>%rdi</b>	<b>%edi</b>
<b>%rsp</b>	<b>%esp</b>
<b>%rbp</b>	<b>%ebp</b>

<b>%r8</b>	<b>%r8d</b>
<b>%r9</b>	<b>%r9d</b>
<b>%r10</b>	<b>%r10d</b>
<b>%r11</b>	<b>%r11d</b>
<b>%r12</b>	<b>%r12d</b>
<b>%r13</b>	<b>%r13d</b>
<b>%r14</b>	<b>%r14d</b>
<b>%r15</b>	<b>%r15d</b>

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

# Instructions

---

- ▶ Long word **l** (4 Bytes)  $\leftrightarrow$  Quad word **q** (8 Bytes)
- ▶ New instructions:
  - ▶ `movl`  $\rightarrow$  `movq`
  - ▶ `addl`  $\rightarrow$  `addq`
  - ▶ `sall`  $\rightarrow$  `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

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

swap:

pushl %ebp	}	Set Up
movl %esp, %ebp		
pushl %ebx		
movl 8(%ebp), %edx	}	Body
movl 12(%ebp), %ecx		
movl (%edx), %ebx		
movl (%ecx), %eax		
movl %eax, (%edx)		
movl %ebx, (%ecx)		
popl %ebx	}	Finish
popl %ebp		
ret		



# 64-bit code for swap

swap:

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

```
movl    (%rdi), %edx
movl    (%rsi), %eax
movl    %eax, (%rdi)
movl    %edx, (%rsi)
```

ret

} Set Up

} Body

} Finish

- ▶ Operands passed in registers (why useful?)
  - ▶ First (**x**p) in %rdi, second (**y**p) 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

swap\_1:

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

```
movq    (%rdi), %rdx
movq    (%rsi), %rax
movq    %rax, (%rdi)
movq    %rdx, (%rsi)
```

ret

} Set Up

} Body

} Finish

## ► 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

