

# Lecture 2

Introduction to Intel Architecture and Programming

# Getting to the nuts and bolts

- Any (modern) operating systems require hardware support
- The kind of support needed is generally the same but the details differ depending on the processor architecture
- We will focus on Intel x86
  - Requires basic knowledge of computer architecture
  - Need to learn basic assembly programming

# A small digression – history of semiconductors

- In the beginning... there was the transistor



John Bardeen  
(1908-1991)

Won a second  
Nobel prize for  
Physics in 1972!



Walter Houser Brattain  
(1902-1987)



William Shockley  
(1910-1989)

The 1956 Nobel Prize in Physics

"for their researches on semiconductors and their discovery of the transistor effect"

# On December 23, 1947...

- A single invention was to change the world



# And Silicon Valley came to be

- In 1956, William Shockley opened Shockley Semiconductor Laboratory
- In 1957, Shockley and eight others started Fairchild Semiconductors
- The **traitorous eight** left Fairchild and started various ventures in semiconductors
  - Robert Noyce and Gordon Moore founded Intel Corp. in 1968
- In 1969, another group of Fairchild engineers founded AMD
- Most in today's Silicon Valley area



From left to right: [Gordon Moore](#), [C. Sheldon Roberts](#), [Eugene Kleiner](#), [Robert Noyce](#), [Victor Grinich](#), [Julius Blank](#), [Jean Hoerni](#) and [Jay Last](#) (1960)

Source: Wikipedia

## Microprocessor

- As opposed to mainframes
- All CPU functionality on a single chip
- Started with popular home computers
  - 8-bit 6502/6510, Z80
  - 32-bit Motorola 68000



The first single chip microprocessor, the 4004, invented by a group of Intel engineers and released in 1971.

# History

- 4004
  - 1971: First single-chip microprocessor; 740 kHz
- 8008
  - 1972: First 8-bit microprocessor; 800 kHz
- 8080
  - 1974: larger instruction set; 2 MHz
- 8086/88
  - 1979: ~5MHz 29,000 transistors (today >1B)
  - Used by NASA at least until 2002 for space shuttle operations
  - IBM PC revolution
  - Manufactured by:
    - Intel
    - AMD
    - NEC
    - Fujitsu,
    - OKI,
    - Siemens
    - ...



# Intel Microprocessors

- Intel introduced the 8086 microprocessor in 1979
- 8086, 8087, 8088, and 80186 processors
  - 16-bit processors with 16-bit registers
  - 16-bit data bus and 20-bit address bus
    - Physical address space =  $2^{20}$  bytes = 1 MB
  - 8087 Floating-Point co-processor
  - Uses segmentation and real-address mode to address memory
    - Each segment can address  $2^{16}$  bytes = 64 KB
  - 8088 is a less expensive version of 8086
    - Uses an 8-bit data bus
  - 80186 is a faster version of 8086

# Intel 80286 and 80386 Processors

- 80286 was introduced in 1982
  - 24-bit address bus  $\Rightarrow 2^{24}$  bytes = 16 MB address space
  - Introduced **protected mode**
    - Segmentation in protected mode is different from the real mode
- 80386 was introduced in 1985
  - First **32-bit processor** with 32-bit general-purpose registers
  - First processor to define the IA-32 architecture
  - 32-bit data bus and 32-bit address bus
  - $2^{32}$  bytes  $\Rightarrow$  4 GB address space
  - Introduced **paging, virtual memory**, and the **flat memory model**
    - Segmentation can be turned off

# Intel 80486 and Pentium Processors

- 80486 was introduced 1989
  - Improved version of Intel 80386
  - Crossed the million transistor mark
  - On-chip **Floating-Point unit** (DX versions)
  - On-chip unified **Instruction/Data Cache** (8 KB)
  - Uses **Pipelining**: can execute up to 1 instruction per clock cycle
- Pentium (80586) was introduced in 1993
  - Wider 64-bit data bus, but address bus is still 32 bits
  - Two execution pipelines: U-pipe and V-pipe
    - **Superscalar** performance: can execute 2 instructions per clock cycle
  - Separate 8 KB instruction and 8 KB data caches
  - **MMX instructions** (later models) for multimedia applications

# Intel P6 Processor Family

- P6 Processor Family: Pentium Pro, Pentium II and III
- Pentium Pro was introduced in 1995
  - **Three-way superscalar**: can execute 3 instructions per clock cycle
  - 36-bit address bus ⇒ up to 64 GB of physical address space
  - Introduced dynamic execution
    - **Out-of-order** and **speculative** execution
  - Integrates a 256 KB second level **L2 cache** on-chip
- Pentium II was introduced in 1997
  - Added **MMX instructions** (already introduced on Pentium MMX)
- Pentium III was introduced in 1999
  - Added **SSE instructions** and eight new 128-bit XMM registers

# Pentium 4 and Xeon Family

- Pentium 4 is a seventh-generation x86 architecture
  - Introduced in 2000
  - New micro-architecture design called Intel **Netburst**
  - Very deep instruction pipeline, scaling to very high frequencies
  - Introduced the **SSE2 instruction set** (extension to SSE)
    - Tuned for multimedia and operating on the 128-bit XMM registers
- In 2002, Intel introduced Hyper-Threading technology
  - Allowed 2 programs to run simultaneously, sharing resources
- Xeon is Intel's name for its server-class microprocessors
  - Xeon chips generally have more cache
  - Support larger multiprocessor configurations

# Pentium-M and EM64T

- Pentium M (**Mobile**) was introduced in 2003
  - Designed for **low-power** laptop computers
  - Modified version of Pentium III, optimized for power efficiency
  - Large second-level cache (2 MB on later models)
  - Runs at lower clock than Pentium 4, but with better performance
- Extended Memory 64-bit Technology (EM64T)
  - Introduced in 2004
  - 64-bit superset of the IA-32 processor architecture
  - 64-bit general-purpose registers and integer support
  - Number of general-purpose registers increased from 8 to 16
  - 64-bit pointers and flat virtual address space
  - Large physical address space: up to  $2^{40} = 1$  Terabytes

# The Big Break

## IBM PC



### IBM PC specs (1981):

- 8088 4.77 MHz
  - External only 8 bit
  - No FPU
- 64KB memory (256KB max)
- 1 x 5.25" floppy 160KB
- Monochrome monitor
- PC Dos 1.0

# The Second Generation

## IBM AT

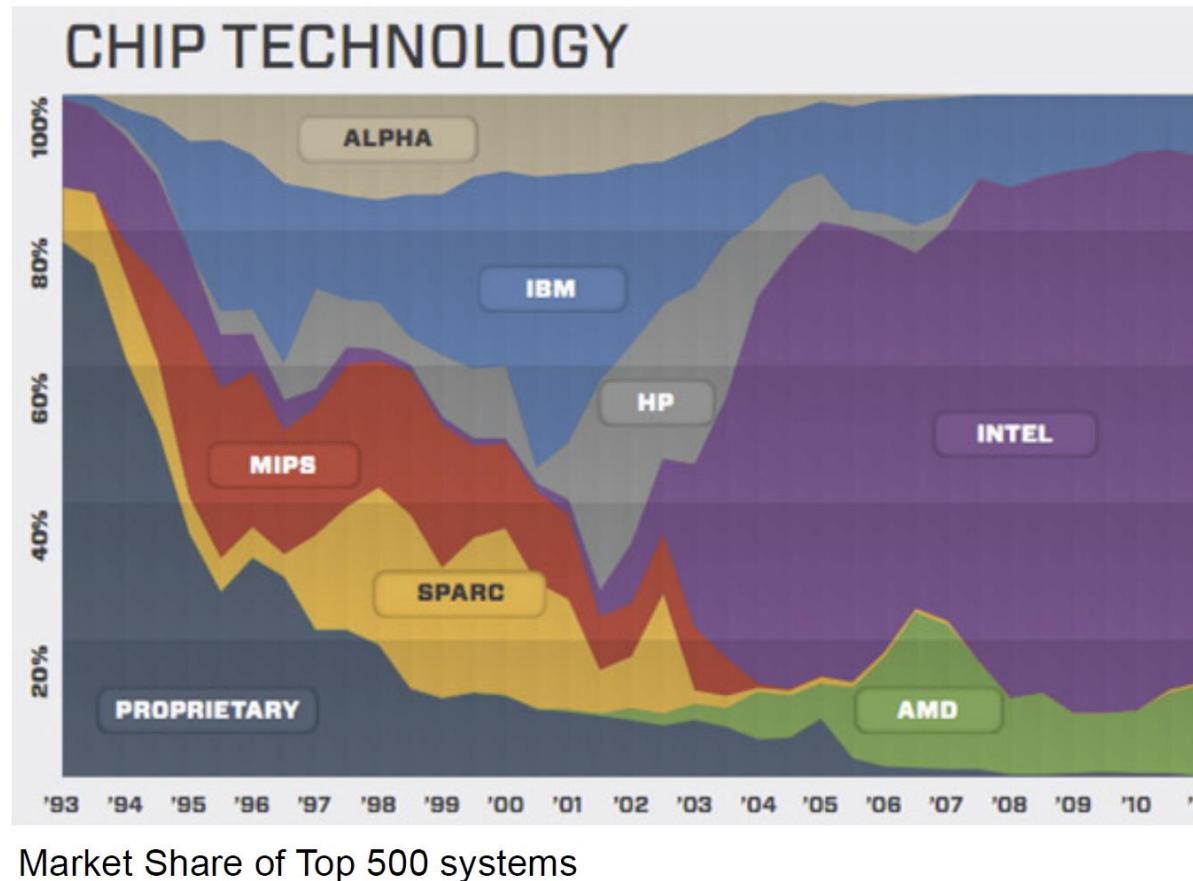


### IBM AT specs:

- 80286 6 MHz
  - No FPU
  - true 16-bit
- 256KB memory (16MB max)
- 1 x 5.25" floppy 1.2MB
- 20MB HDD
- Monochrome monitor

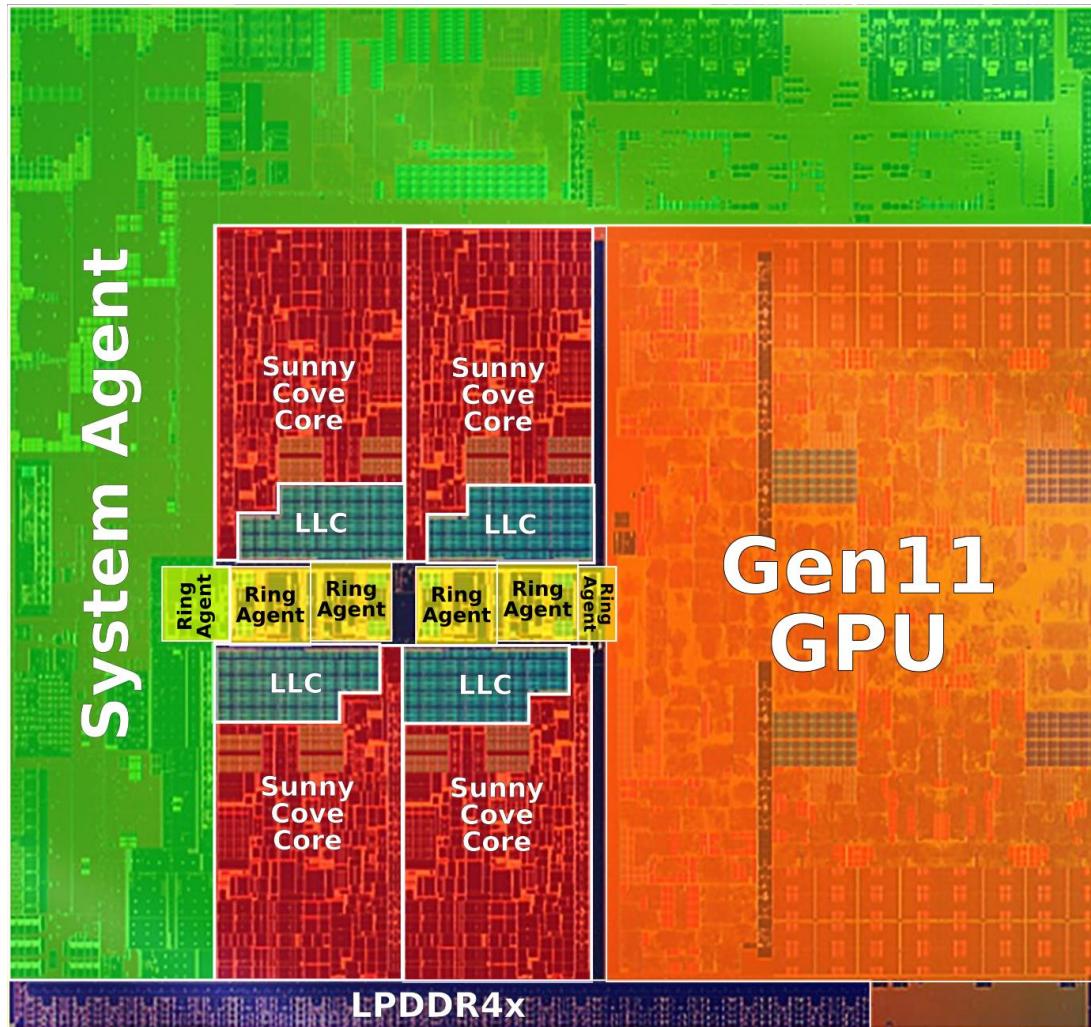


# The Dominant Technology



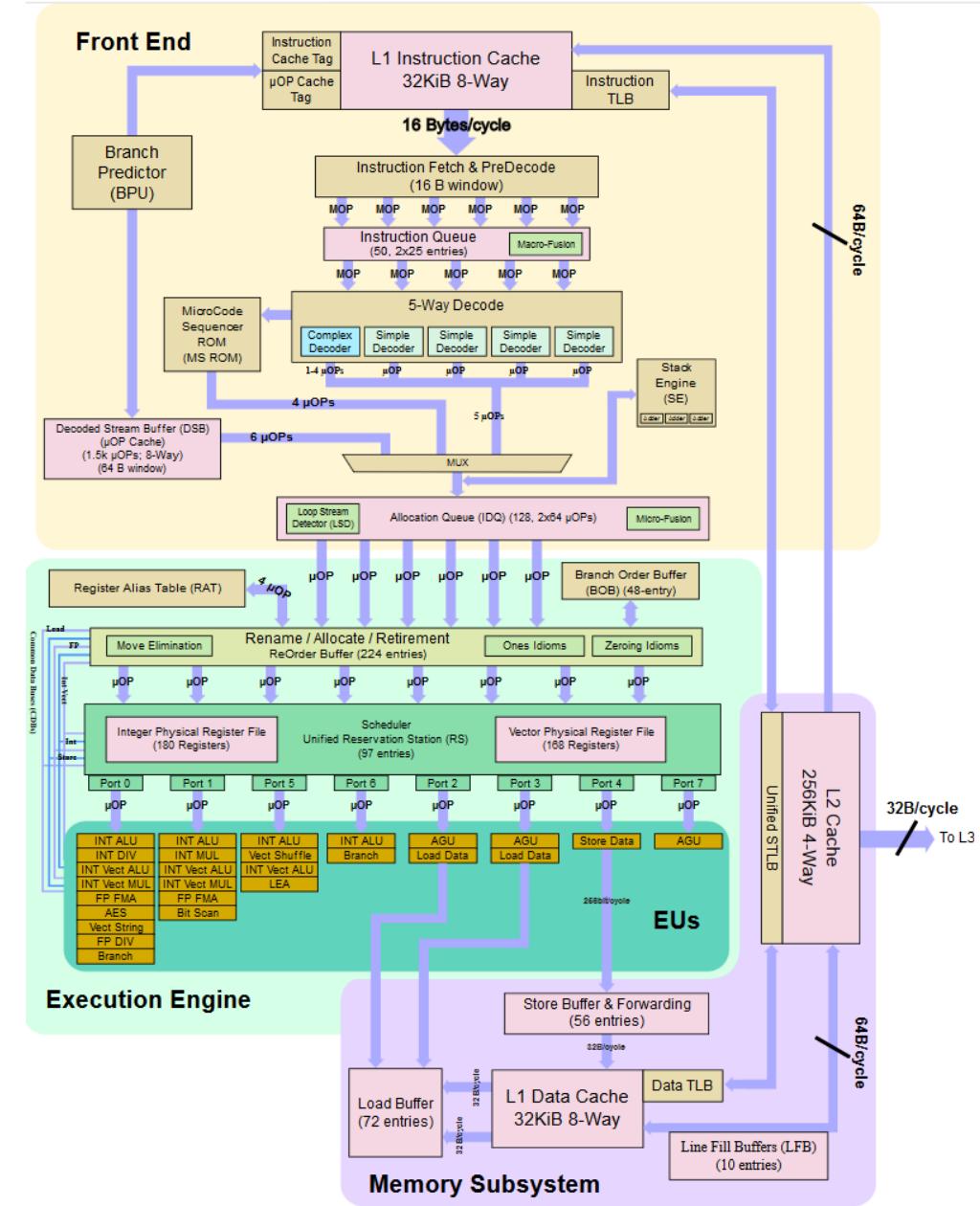
Until ARM came along

# Intel Ice Lake



10nm,  
> 1 billion  
transistors

# Intel Core Architecture



# Apple M1



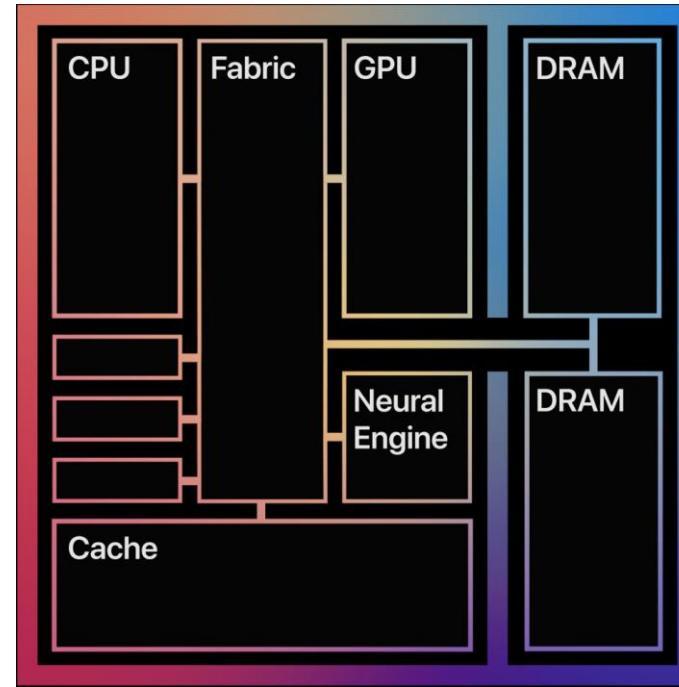
ThinkDifferent

APPLE MICROSOFT GOOGLE INTERNET

f Share ...

## Apple's M1 chip outperforms x86 chips even when emulating with Rosetta

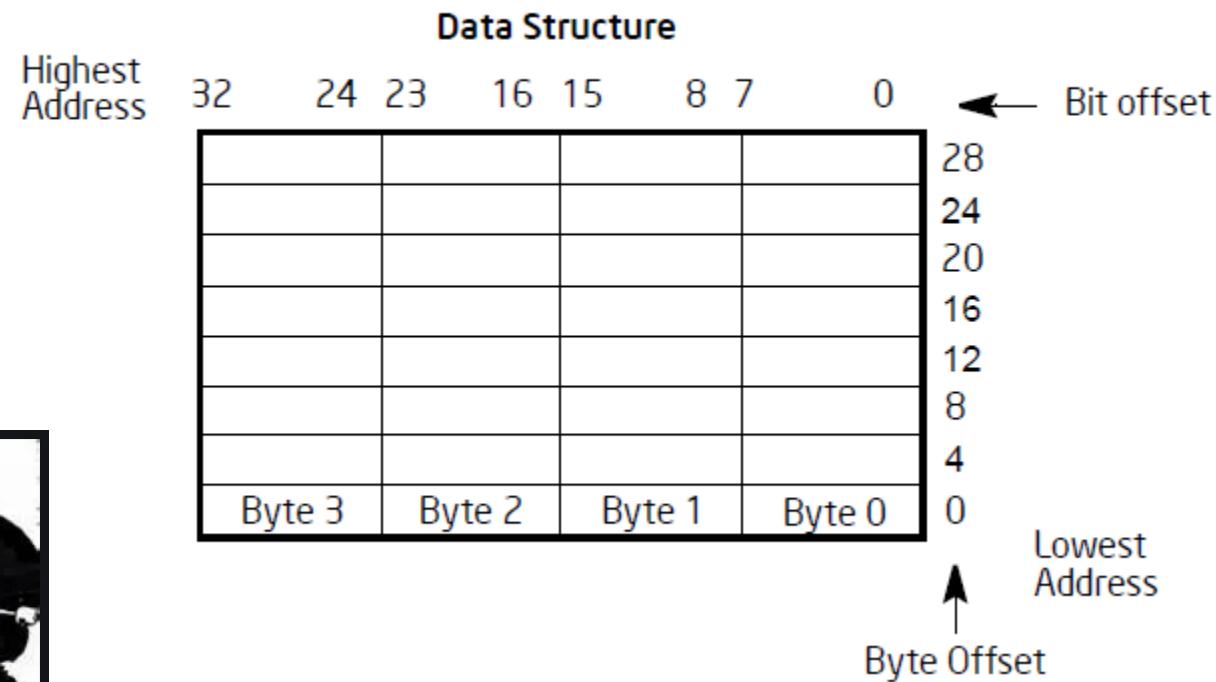
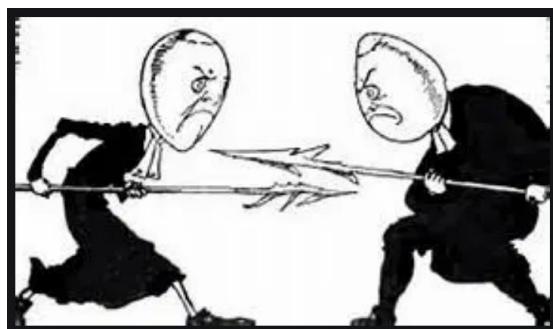
BY RIDA IMRAN NOV 16, 2020



# Nomenclature

- “IA-32” – 32 bit x86
- “Intel 64” – Intel’s version of AMD64 which was AMD’s 64 bit extension of IA-32
- “IA-64” – Itanium (deadend)

# Little Endian



# The Registers

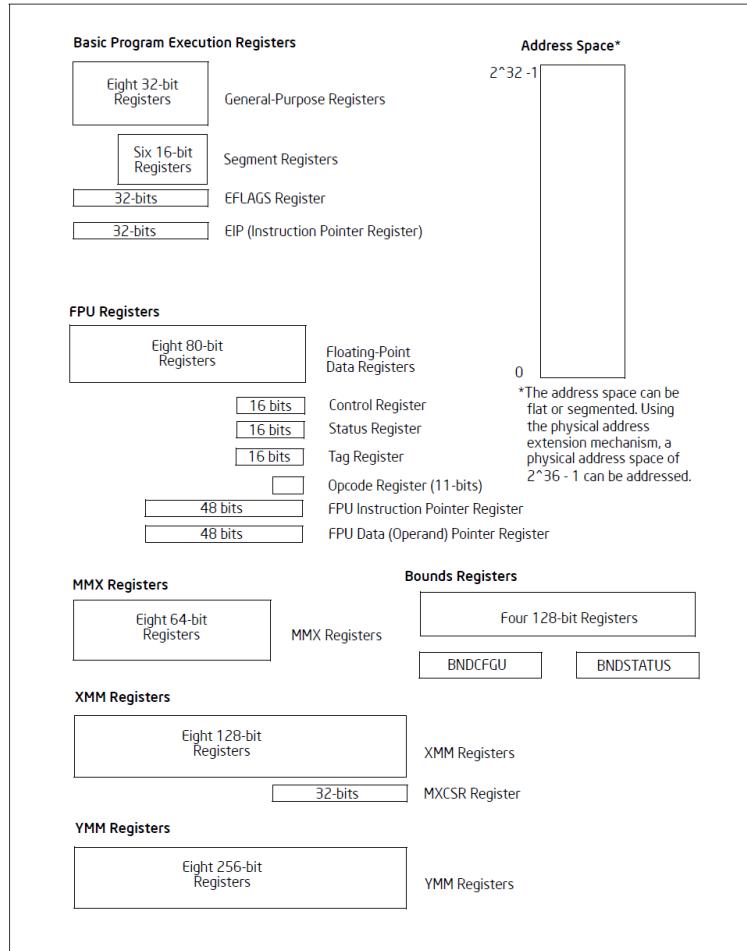


Figure 3-1. IA-32 Basic Execution Environment for Non-64-bit Modes

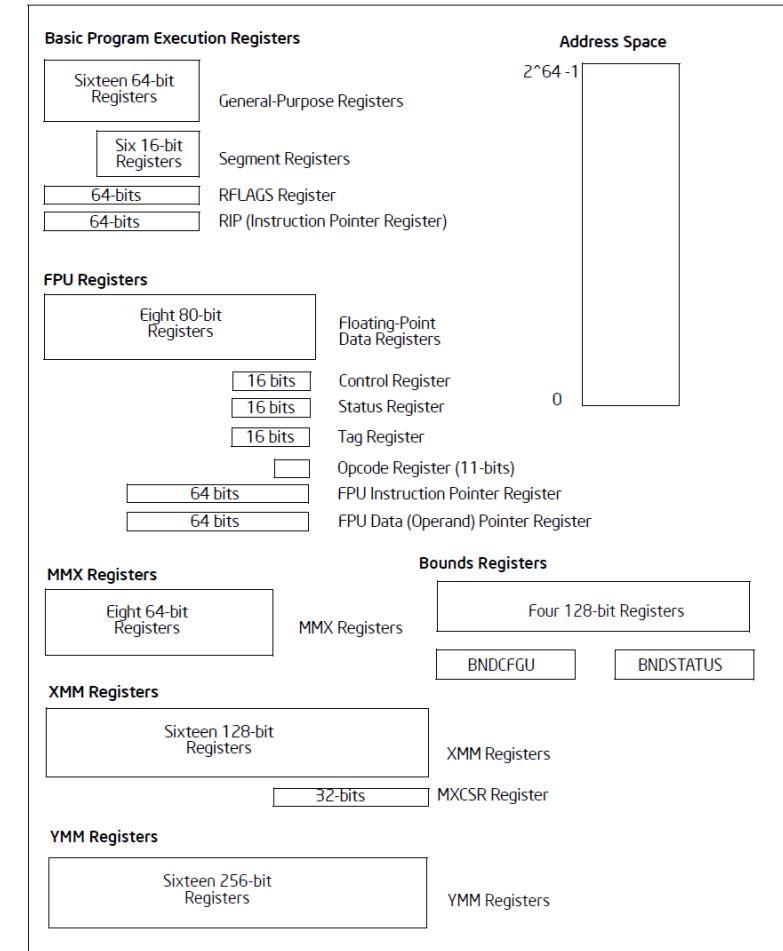
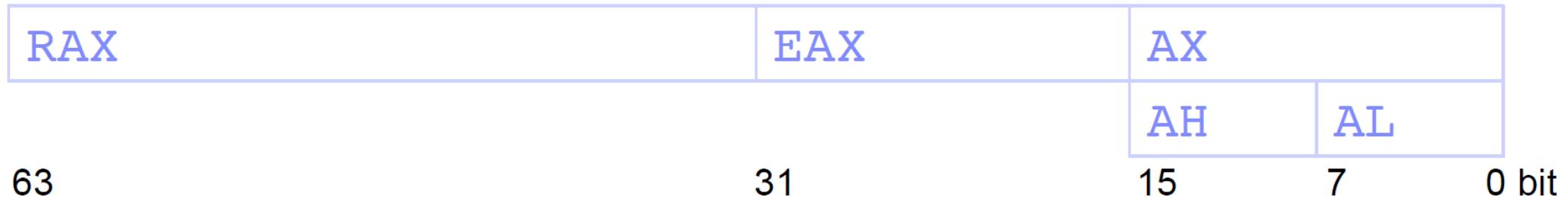


Figure 3-2. 64-Bit Mode Execution Environment

# Intel 64 registers



32-bit: **EAX** (Accumulator), **EBX** (Base Index), **ECX** (Counter), **EDX** (Data), **ESI** (Source Index), **EDI** (Destination Index), **ESP** (Stack Pointer), **EBP** (Base Pointer), **EIP** (Instruction Pointer)

16 bit: **CS** (Code Segment), **DS** (Data Segment), **SS** (Stack Segment), **ES**, **FS**, **GS**

64-bit: RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP, RIP, **R8**, ..., **R15**

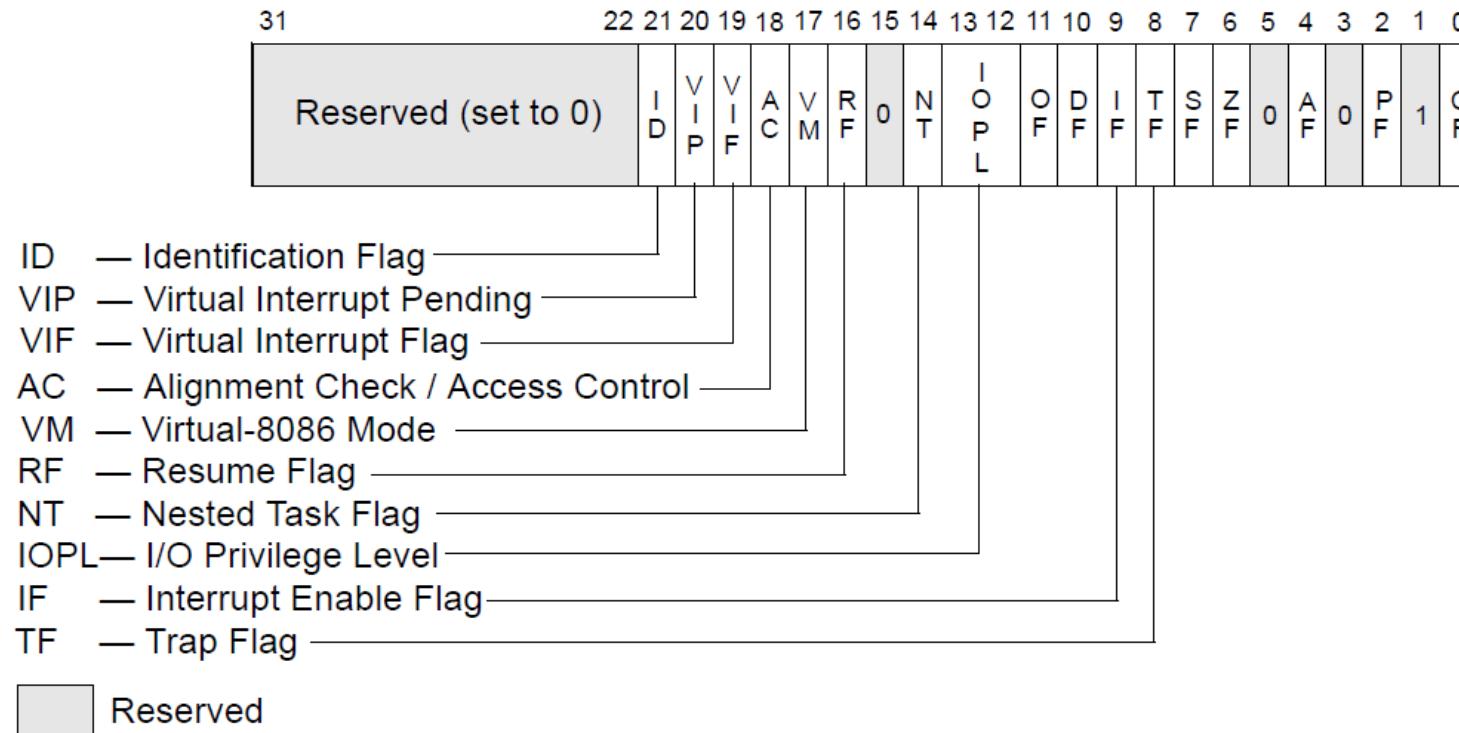
# (Sub)register names

| Identifiers to access registers and parts thereof |             |  |         |  |       |  |       |  |               |  |                    |  |        |  |             |  |
|---|-------------|--|---------|--|-------|--|-------|--|---------------|--|--------------------|--|--------|--|-------------|--|
| Register  | Accumulator |  | Counter |  | Data  |  | Base  |  | Stack Pointer |  | Stack Base Pointer |  | Source |  | Destination |  |
| 64-bit  | RAX         |  | RCX     |  | RDX   |  | RBX   |  | RSP           |  | RBP                |  | RSI    |  | RDI         |  |
| 32-bit  | EAX         |  | ECX     |  | EDX   |  | EBX   |  | ESP           |  | EBP                |  | ESI    |  | EDI         |  |
| 16-bit  | AX          |  | CX      |  | DX    |  | BX    |  | SP            |  | BP                 |  | SI     |  | DI          |  |
| 8-bit   | AH AL       |  | CH CL   |  | DH DL |  | BH BL |  | SPL           |  | BPL                |  | SIL    |  | DIL         |  |

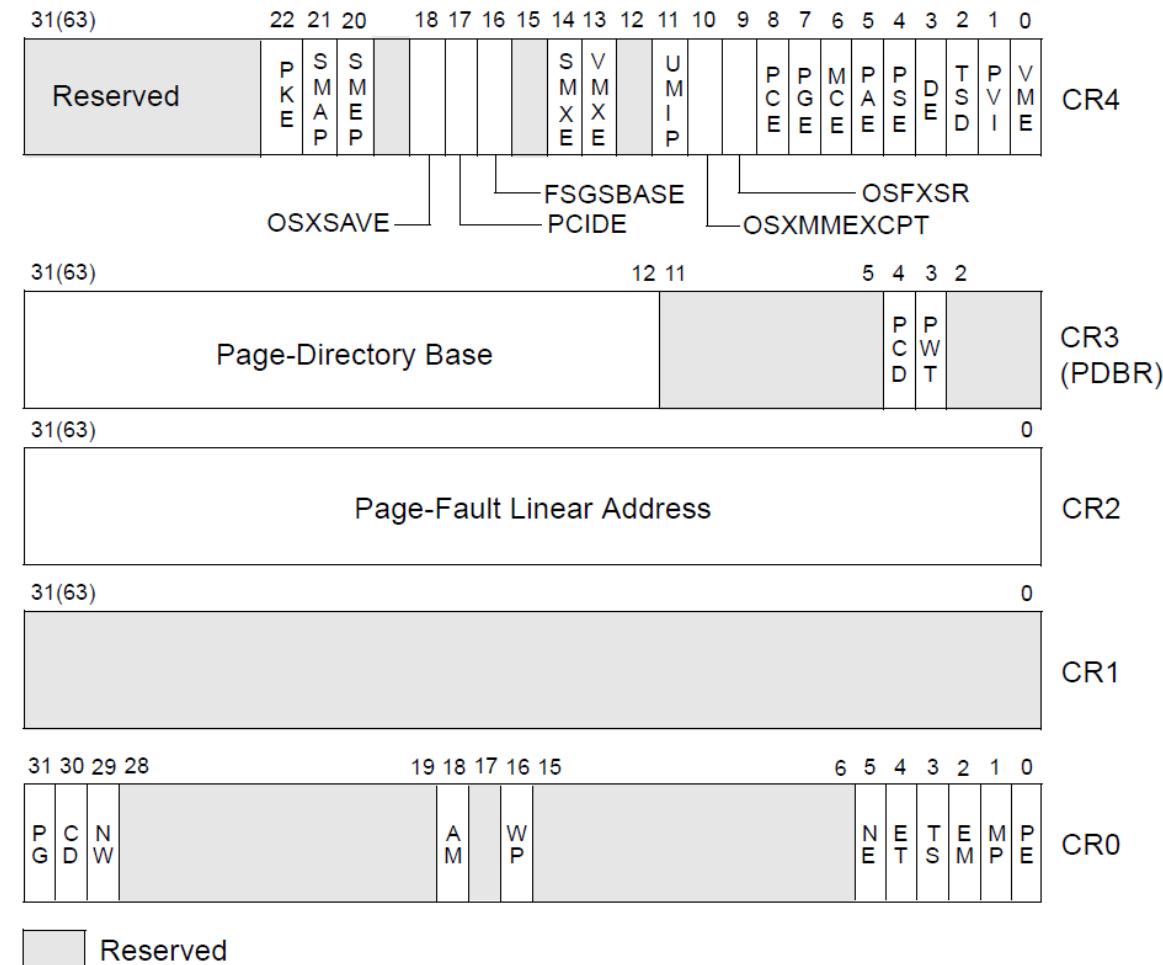
Table 3-2. Addressable General Purpose Registers

| Register Type        | Without REX                            | With REX   |
|----------------------|--|--|
| Byte Registers       | AL, BL, CL, DL, AH, BH, CH, DH         | AL, BL, CL, DL, DIL, SIL, BPL, SPL, R8B - R15B     |
| Word Registers       | AX, BX, CX, DX, DI, SI, BP, SP         | AX, BX, CX, DX, DI, SI, BP, SP, R8W - R15W         |
| Doubleword Registers | EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP | EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D - R15D |
| Quadword Registers   | N.A.                                   | RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8 - R15   |

# EFLAGS



# Control Registers



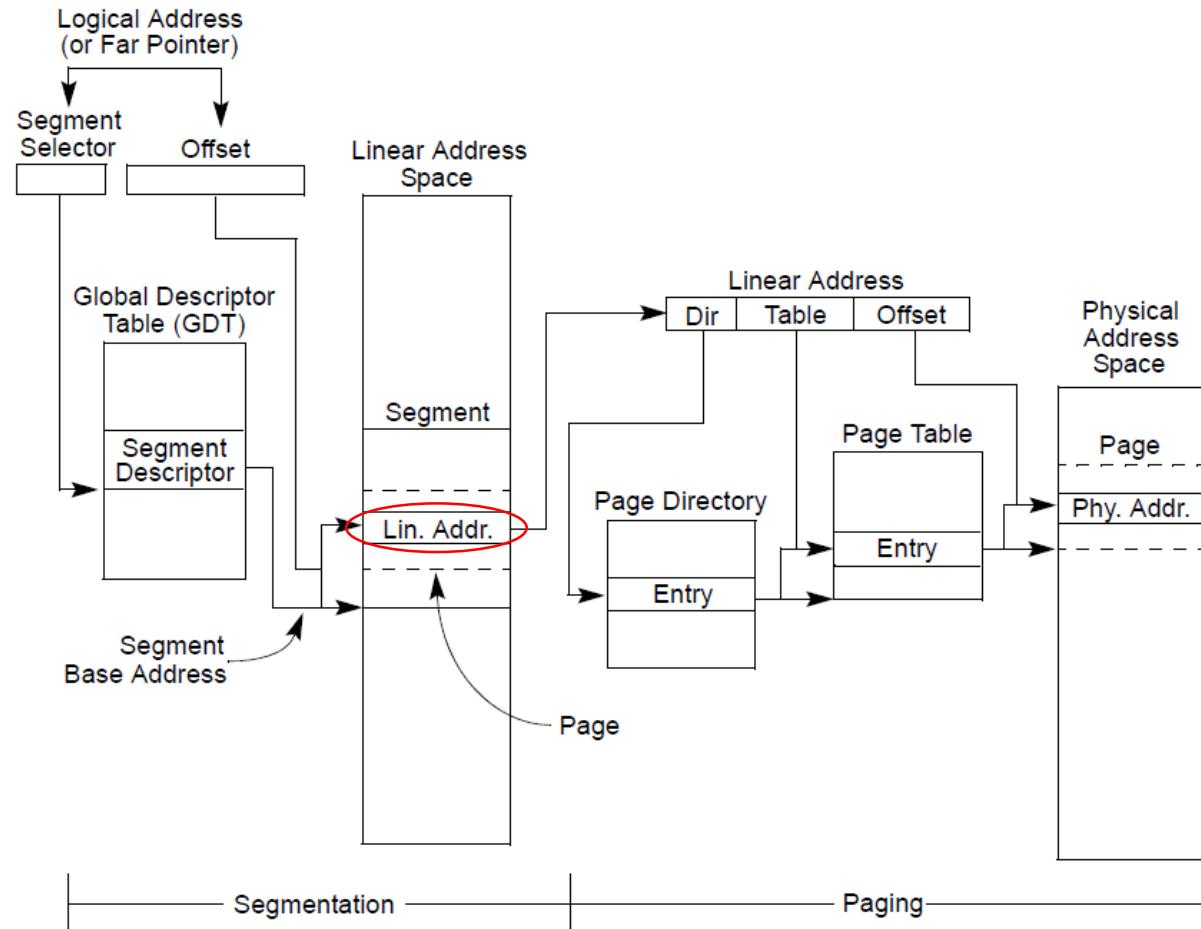
# The Operating Modes (1)

- **Real-address Mode**: This mode implements the programming environment of the Intel 8086 processor with extensions (such as the ability to switch to protected or system management mode). The processor is placed in real-address mode following power-up or a reset.
- **Protected Mode**: Normal operation of the processor with virtual memory and **privilege rings** enabled.
- **System Management Mode**: This mode provides an operating system or executive with a transparent mechanism for implementing platform-specific functions such as power management and system security.

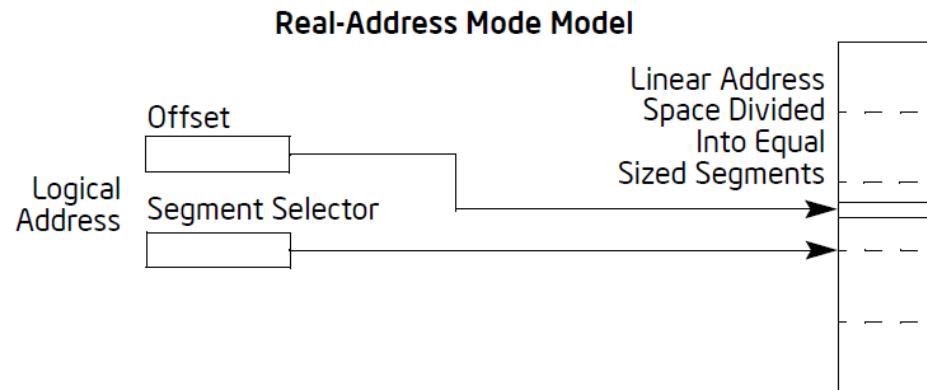
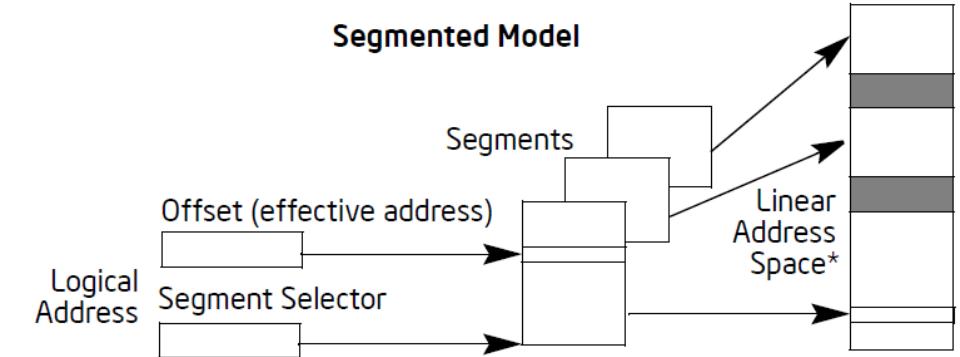
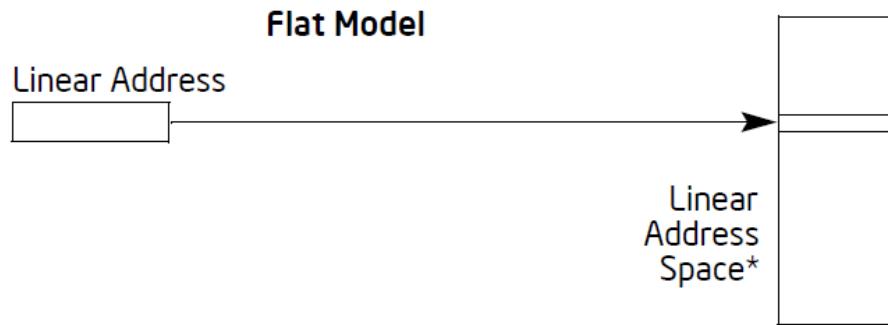
# The Operating Modes (2)

- **Virtual-8086 Mode:** This mode allows the processor execute 8086 software in a protected, multitasking environment.
- **Intel 64 IA-32e Mode:** the processor supports two sub-modes: compatibility mode and 64-bit mode. 64-bit mode provides 64-bit linear addressing and support for physical address space larger than 64 GBytes. Compatibility mode allows most legacy protected-mode applications to run unchanged.

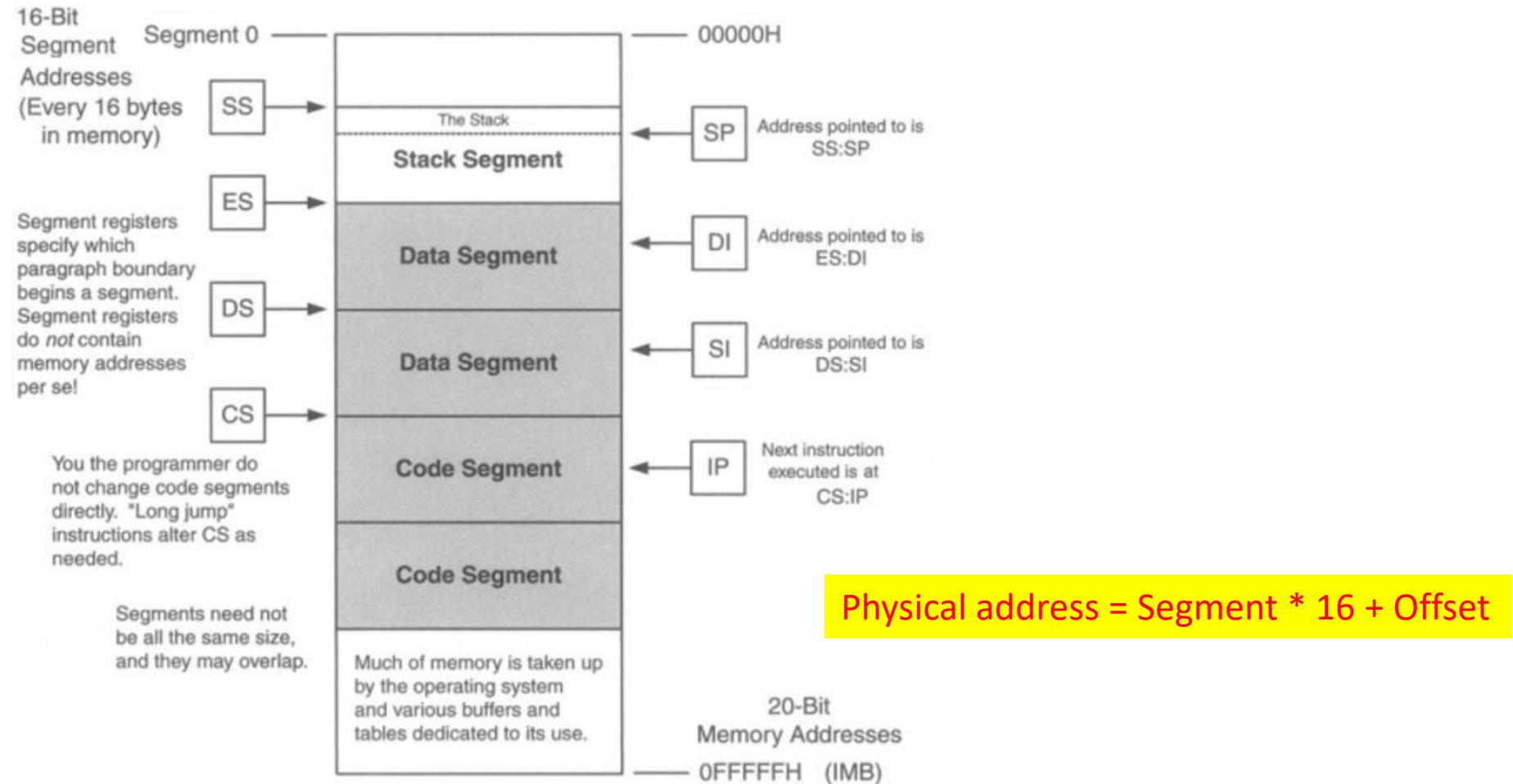
# Memory addressing – Segmentation and Paging



# The Three Addressing Models



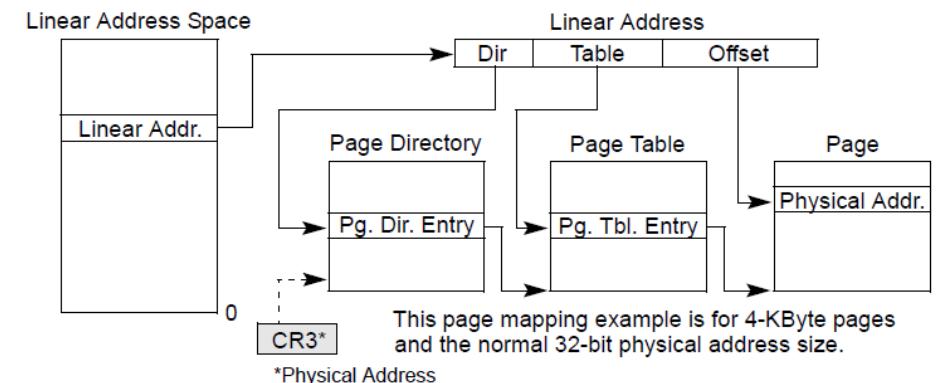
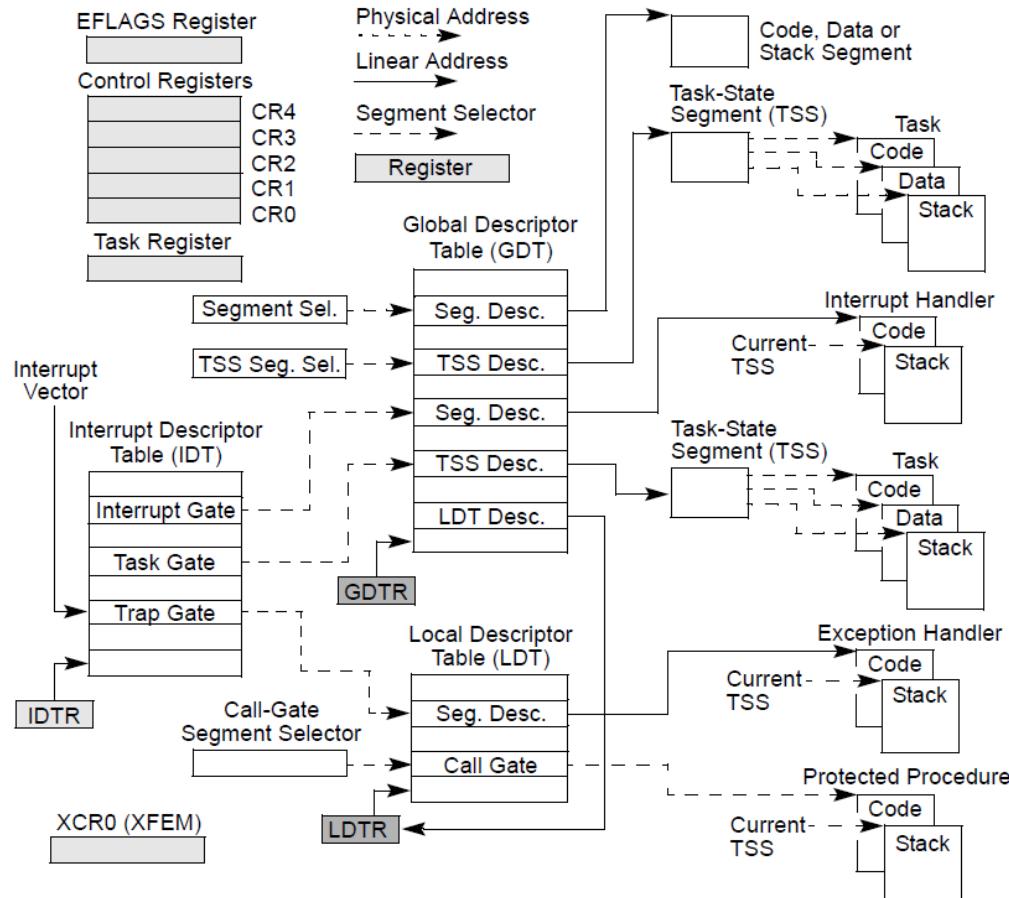
# Real-address Mode Model



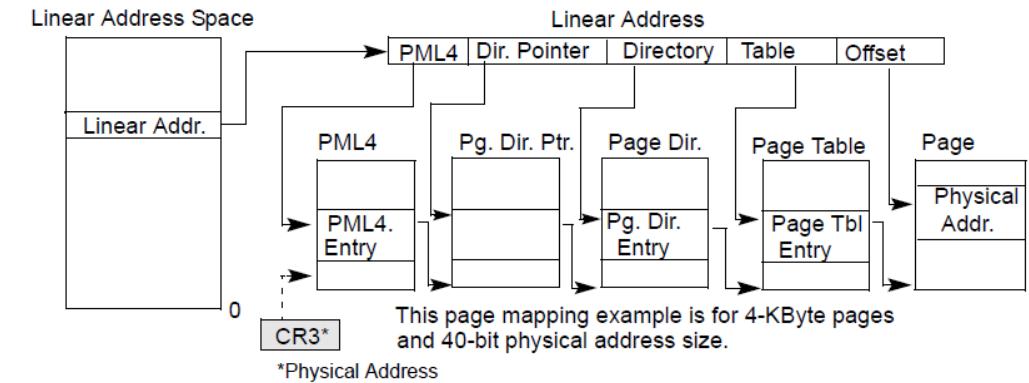
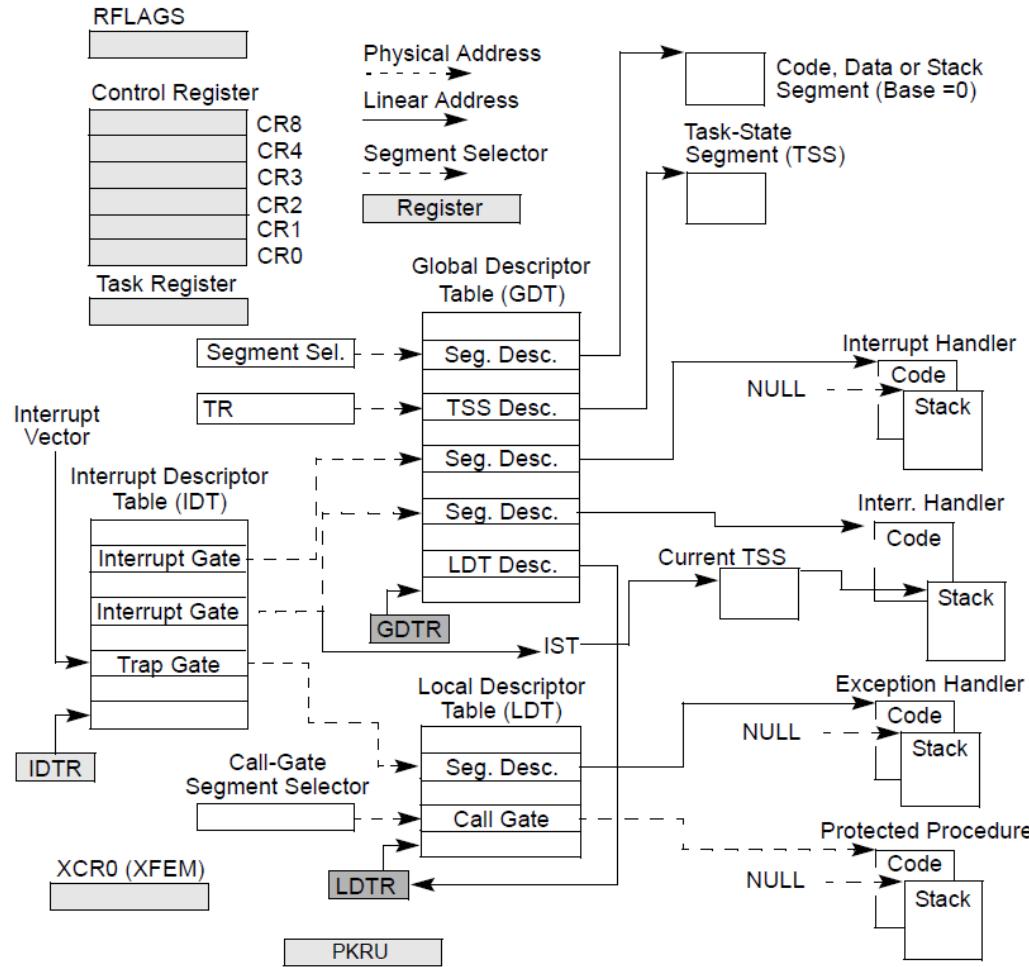
# The Modes vs Three Models

- Real-address Mode – can only use real-address mode model (i.e., segmented real address)
- Protected Mode – can use any of the three models
- System Management Mode - processor switches to a separate address space, called the system management RAM (SMRAM) that is similar to the real-address mode model.

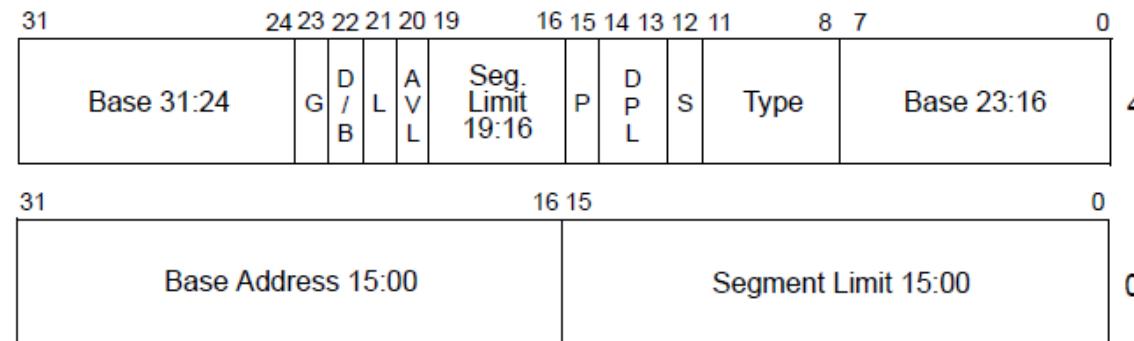
# IA-32 System-Level Registers and Data Structures



# System-Level Registers and Data Structures in IA-32e (64 bit) Mode



# IA-32 Segment Descriptor



L — 64-bit code segment (IA-32e mode only)

AVL — Available for use by system software

BASE — Segment base address

D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)

DPL — Descriptor privilege level

G — Granularity

LIMIT — Segment Limit

P — Segment present

S — Descriptor type (0 = system; 1 = code or data)

TYPE — Segment type

# The Descriptor Tables

| System Table Registers |                                |                    |   |
|------------------------|--------------------------------|--------------------|---|
|                        | 47(79)                         | 16 15              | 0 |
| GDTR                   | 32(64)-bit Linear Base Address | 16-Bit Table Limit |   |
| IDTR                   | 32(64)-bit Linear Base Address | 16-Bit Table Limit |   |

| System Segment Registers |           | Segment Descriptor Registers (Automatically Loaded) |               |  |            |
|--------------------------|-----------|---|---------------|--|------------|
|                          | 15 0      |   |               |  | Attributes |
| Task Register            | Seg. Sel. | 32(64)-bit Linear Base Address                      | Segment Limit |  |            |
| LDTR                     | Seg. Sel. | 32(64)-bit Linear Base Address                      | Segment Limit |  |            |

# Segment Addressing

## Real mode

Segment register \* 16 + offset

Example: CS:IP

CS = 0xA01

IP = 0x20

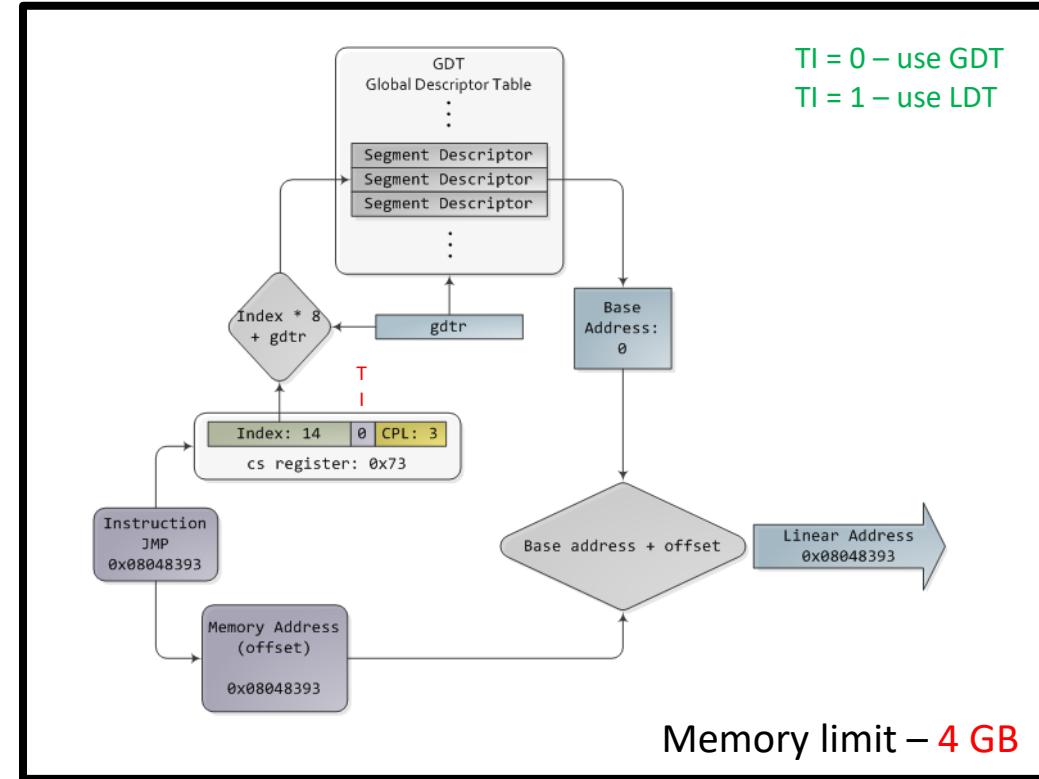
$$\begin{aligned}\text{Real address} &= \text{CS} * 16 + \text{IP} \\ &= 0xA030\end{aligned}$$

Note: Offset limited to 16 bits

Each segment at most **64KB**

Total memory limited to **1MB**

## Protected mode

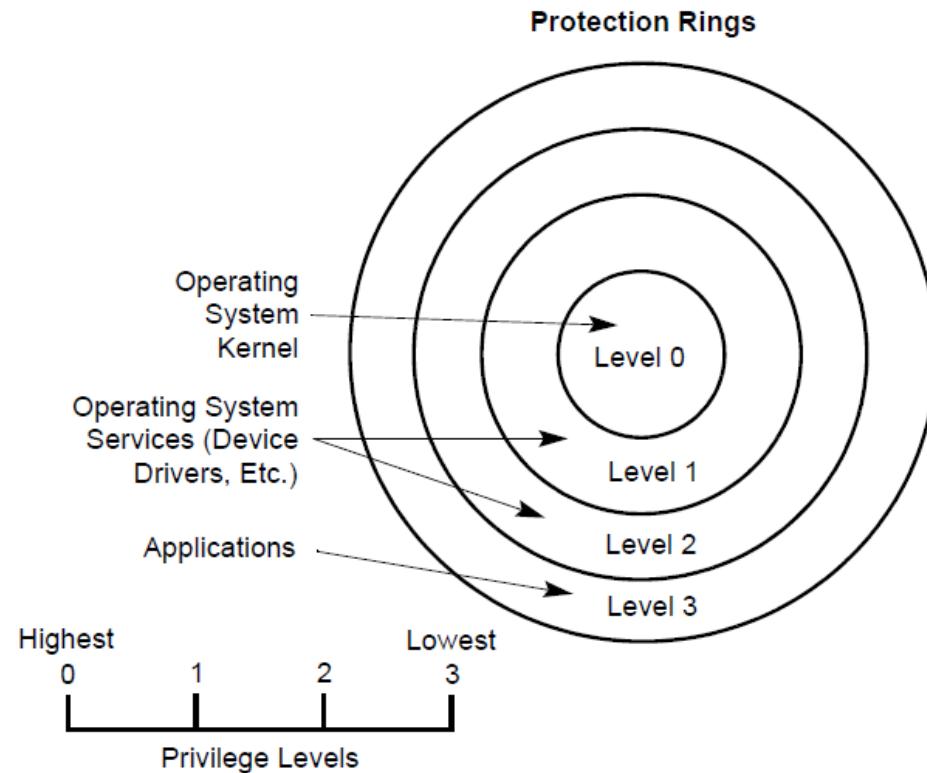


Source: <http://duartes.org/gustavo/blog/post/memory-translation-and-segmentation/>

# Linux's GDT

| <i>Linux's GDT</i> | <i>Segment Selectors</i> | <i>Linux's GDT</i>  | <i>Segment Selectors</i> |
|--------------------|--------------------------|---------------------|--------------------------|
| null               | 0x0                      | TSS                 | 0x80                     |
| reserved           |                          | LDT                 | 0x88                     |
| reserved           |                          | PNPBIOS 32-bit code | 0x90                     |
| reserved           |                          | PNPBIOS 16-bit code | 0x98                     |
| not used           |                          | PNPBIOS 16-bit data | 0xa0                     |
| not used           |                          | PNPBIOS 16-bit data | 0xa8                     |
| TLS #1             | 0x33                     | PNPBIOS 16-bit data | 0xb0                     |
| TLS #2             | 0x3b                     | APMBIOS 32-bit code | 0xb8                     |
| TLS #3             | 0x43                     | APMBIOS 16-bit code | 0xc0                     |
| reserved           |                          | APMBIOS data        | 0xc8                     |
| reserved           |                          | not used            |                          |
| reserved           |                          | not used            |                          |
| kernel code        | 0x60 (_KERNEL_CS)        | not used            |                          |
| kernel data        | 0x68 (_KERNEL_DS)        | not used            |                          |
| user code          | 0x73 (_USER_CS)          | not used            |                          |
| user data          | 0x7b (_USER_DS)          | double fault TSS    | 0xf8                     |

# Protection Rings



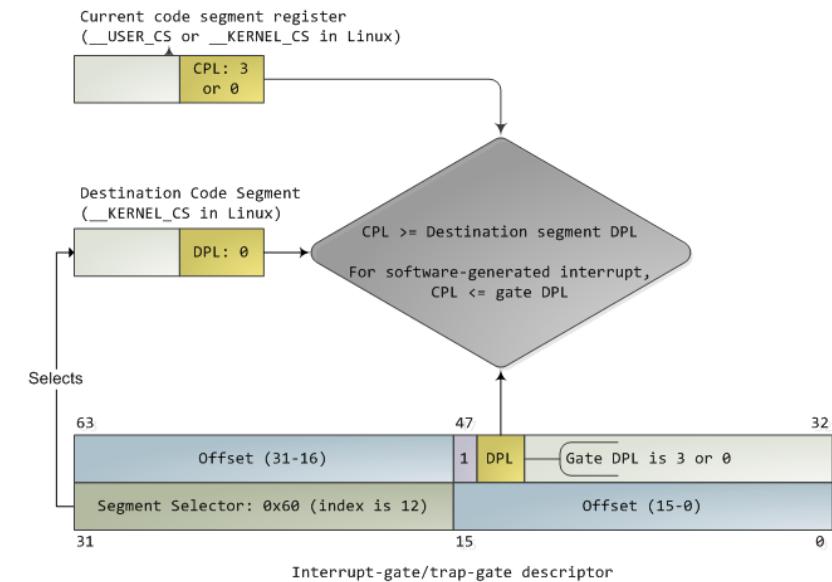
# Segment Protection

- Two crucial points of protection
    - when a segment selector is loaded
    - when a page of memory is accessed with a linear address
  - All user code runs in Ring 3
  - All kernel code runs in Ring 0
  - Current Privilege Level (CPL) – 2 bits in CS register



# Switching from Ring 3 to Ring 0

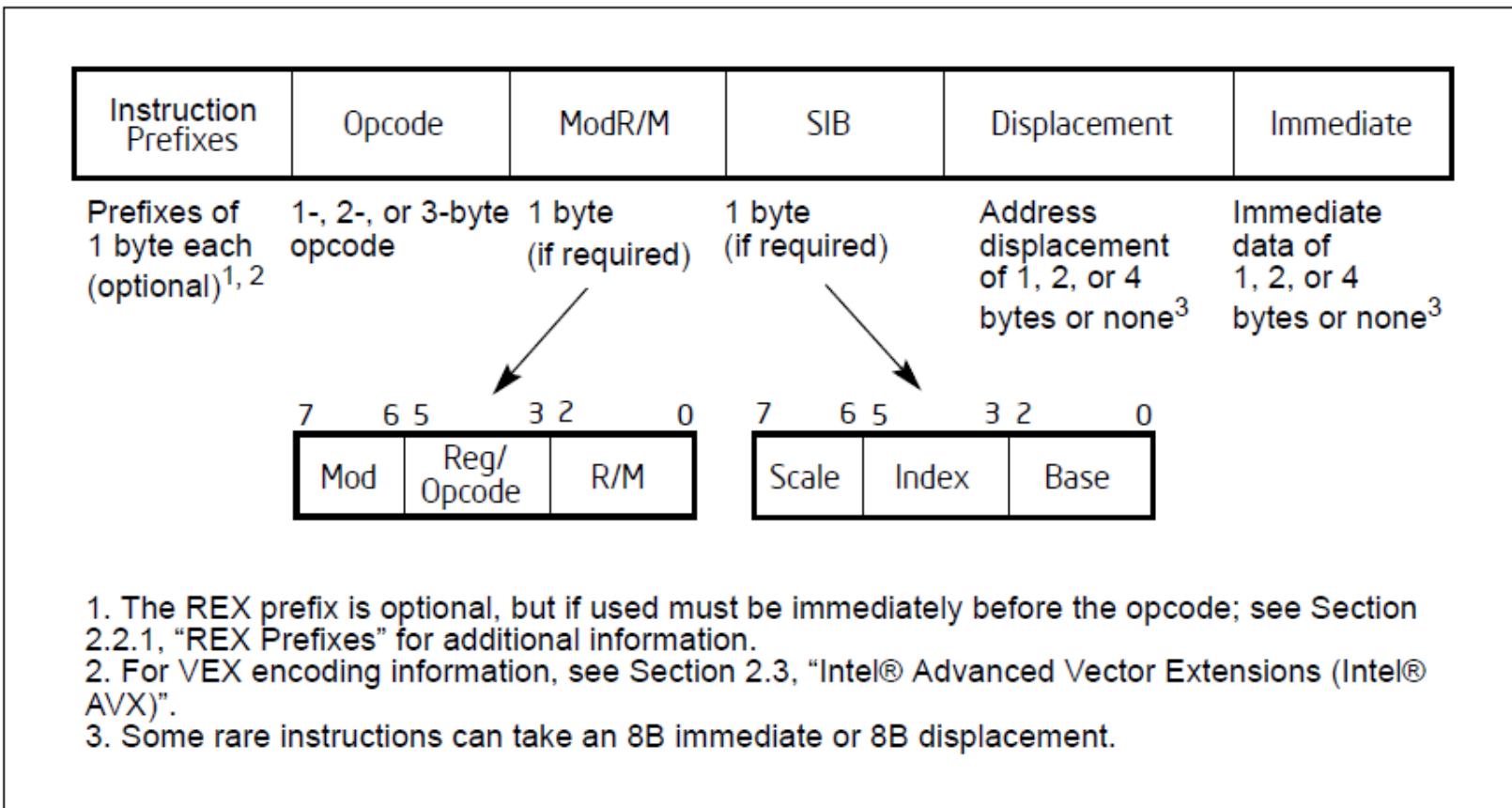
- Transit due to interrupt using an interrupt-gate (the handler)
  - Interrupt can **never** go from higher privilege to lower privilege
- Using SYSENTER instruction



# Switching from Ring 0 to Ring 3

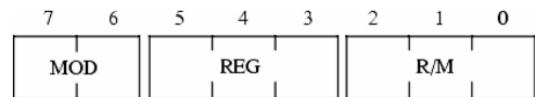
- Through an interrupt return instruction (IRET)
- Through SYSEXIT instruction
- For more details: <http://duartes.org/gustavo/blog/post/cpu-rings-privilege-and-protection/>

# Instruction Encoding



## Intel 64 and IA-32 Architectures Instruction Format

# ModRM byte



The **MOD** field specifies x86 addressing mode:

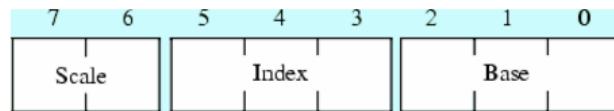
| MOD | Meaning   |
|-----|---|
| 00  | Register indirect addressing mode or SIB with no displacement (when R/M = 100) or Displacement only addressing mode (when R/M = 101). |
| 01  | One-byte signed displacement follows addressing mode byte(s).   |
| 10  | Four-byte signed displacement follows addressing mode byte(s).  |
| 11  | Register addressing mode.   |

The **REG** field specifies source or destination **register**:

| REG Value | Register if data size is eight bits | Register if data size is 16-bits | Register if data size is 32 bits |
|-----------|-------------------------------------|----------------------------------|----------------------------------|
| 000       | al                                  | ax                               | eax                              |
| 001       | cl                                  | cx                               | ecx                              |
| 010       | dl                                  | dx                               | edx                              |
| 011       | bl                                  | bx                               | ebx                              |
| 100       | ah                                  | sp                               | esp                              |
| 101       | ch                                  | bp                               | ebp                              |
| 110       | dh                                  | si                               | esi                              |
| 111       | bh                                  | di                               | edi                              |

Source: <http://www.c-jump.com/CIS77/CPU/x86/lecture.html>

# SIB byte



| Index | Register |
|-------|----------|
| 000   | EAX      |
| 001   | ECX      |
| 010   | EDX      |
| 011   | EBX      |
| 100   | Illegal  |
| 101   | EBP      |
| 110   | ESI      |
| 111   | EDI      |

| Scale Value | Index*Scale Value |
|-------------|-------------------|
| 00          | Index*1           |
| 01          | Index*2           |
| 10          | Index*4           |
| 11          | Index*8           |

| Base | Register   |
|------|--|
| 000  | EAX  |
| 001  | ECX  |
| 010  | EDX  |
| 011  | EBX  |
| 100  | ESP  |
| 101  | Displacement-only if MOD = 00, EBP if MOD = 01 or 10 |
| 110  | ESI  |
| 111  | EDI  |

# In 64 bits

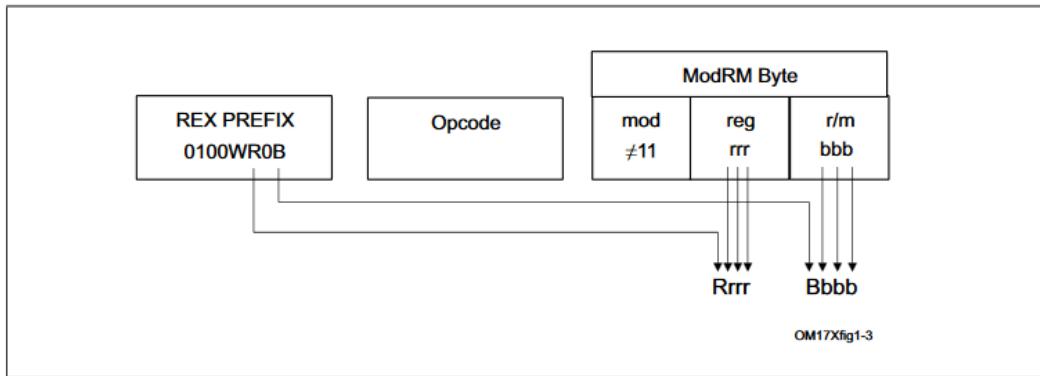


Figure 2-4. Memory Addressing Without an SIB Byte; REX.X Not Used

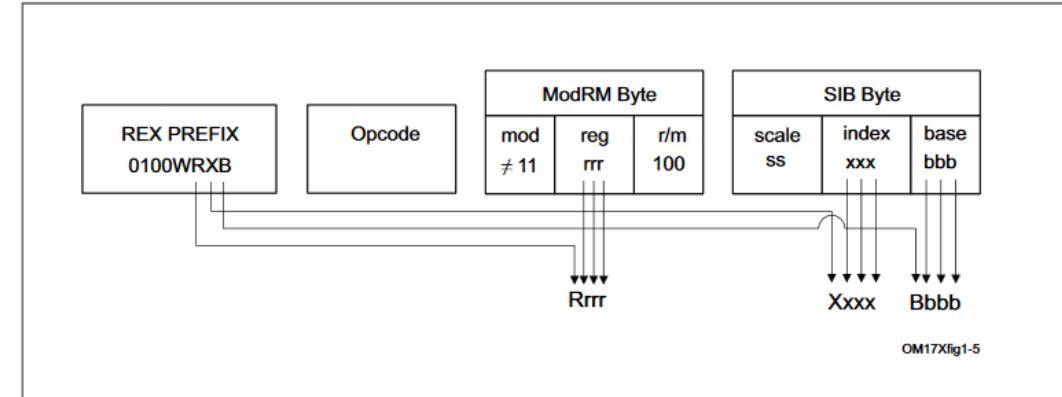


Figure 2-6. Memory Addressing With a SIB Byte

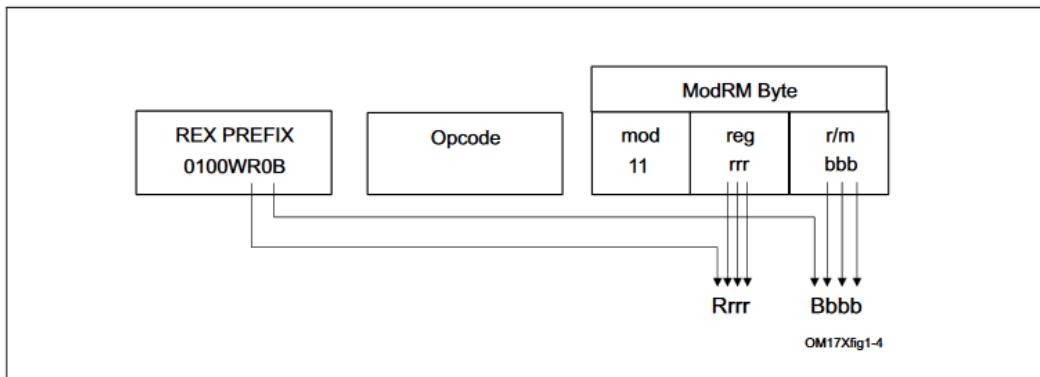


Figure 2-5. Register-Register Addressing (No Memory Operand); REX.X Not Used

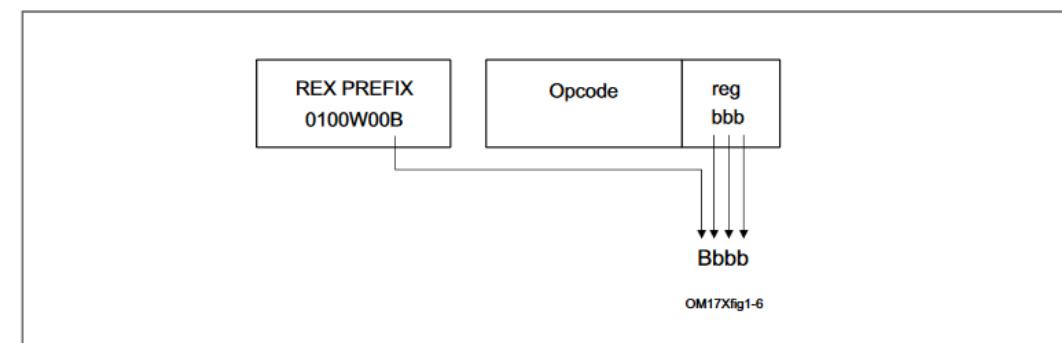


Figure 2-7. Register Operand Coded in Opcode Byte; REX.X & REX.R Not Used

# ModRM in 64 bit

| 32/64-bit | B.R/M          |       |       |       |                |                                   |       |       |       |       |       |                |                                   |       |       |       |  |  |  |  |  |  |
|-----------|----------------|-------|-------|-------|----------------|-----------------------------------|-------|-------|-------|-------|-------|----------------|-----------------------------------|-------|-------|-------|--|--|--|--|--|--|
| Mod       | 0.000          | 0.001 | 0.010 | 0.011 | 0.100          | 0.101                             | 0.110 | 0.111 | 1.000 | 1.001 | 1.010 | 1.011          | 1.100                             | 1.101 | 1.110 | 1.111 |  |  |  |  |  |  |
|           | AX             | CX    | DX    | BX    | SP             | BP                                | SI    | DI    | R8    | R9    | R10   | R11            | R12                               | R13   | R14   | R15   |  |  |  |  |  |  |
| 00        | [r/m]          |       |       |       | [SIB]          | [RIP/EIP <sup>1,2</sup> + disp32] |       | [r/m] |       |       |       | [SIB]          | [RIP/EIP <sup>1,2</sup> + disp32] |       | [r/m] |       |  |  |  |  |  |  |
| 01        | [r/m + disp8]  |       |       |       | [SIB + disp8]  | [r/m + disp8]                     |       |       |       |       |       | [SIB + disp8]  | [r/m + disp8]                     |       |       |       |  |  |  |  |  |  |
| 10        | [r/m + disp32] |       |       |       | [SIB + disp32] | [r/m + disp32]                    |       |       |       |       |       | [SIB + disp32] | [r/m + disp32]                    |       |       |       |  |  |  |  |  |  |
| 11        | r/m            |       |       |       |                |                                   |       |       |       |       |       |                |                                   |       |       |       |  |  |  |  |  |  |

# SIB in 64 bits

|     |                       | B.Base               |          |          |          |          |                        |                      |          |          |          |           |                        |                      |                        |           |           |  |  |  |  |  |  |  |
|-----|-----------------------|----------------------|----------|----------|----------|----------|------------------------|----------------------|----------|----------|----------|-----------|------------------------|----------------------|------------------------|-----------|-----------|--|--|--|--|--|--|--|
| Mod | X.Index               | 0.000 AX             | 0.001 CX | 0.010 DX | 0.011 BX | 0.100 SP | 0.101 <sup>1</sup> BP  | 0.110 SI             | 0.111 DI | 1.000 R8 | 1.001 R9 | 1.010 R10 | 1.011 R11              | 1.100 R12            | 1.101 <sup>1</sup> R13 | 1.110 R14 | 1.111 R15 |  |  |  |  |  |  |  |
| 00  | 0.000 AX              | [base + (index * s)] |          |          |          |          | [(index * s) + disp32] | [base + (index * s)] |          |          |          |           | [(index * s) + disp32] | [base + (index * s)] |                        |           |           |  |  |  |  |  |  |  |
|     | 0.001 CX              |                      |          |          |          |          |                        |                      |          |          |          |           |                        |                      |                        |           |           |  |  |  |  |  |  |  |
|     | 0.010 DX              |                      |          |          |          |          |                        |                      |          |          |          |           |                        |                      |                        |           |           |  |  |  |  |  |  |  |
|     | 0.011 BX              |                      |          |          |          |          |                        |                      |          |          |          |           |                        |                      |                        |           |           |  |  |  |  |  |  |  |
|     | 0.100 <sup>2</sup> SP | [base]               |          |          | [disp32] |          | [base]                 |                      |          |          |          | [disp32]  |                        | [base]               |                        |           |           |  |  |  |  |  |  |  |
|     | 0.101 BP              | [base + (index * s)] |          |          |          |          |                        | [base + (index * s)] |          |          |          |           |                        |                      |                        |           |           |  |  |  |  |  |  |  |
|     | 0.110 SI              |                      |          |          |          |          |                        |                      |          |          |          |           |                        |                      |                        |           |           |  |  |  |  |  |  |  |
|     | 0.111 DI              |                      |          |          |          |          |                        |                      |          |          |          |           |                        |                      |                        |           |           |  |  |  |  |  |  |  |
|     | 1.000 R8              |                      |          |          |          |          |                        |                      |          |          |          |           |                        |                      |                        |           |           |  |  |  |  |  |  |  |
|     | 1.001 R9              |                      |          |          |          |          |                        |                      |          |          |          |           |                        |                      |                        |           |           |  |  |  |  |  |  |  |
|     | 1.010 R10             |                      |          |          |          |          |                        |                      |          |          |          |           |                        |                      |                        |           |           |  |  |  |  |  |  |  |
|     | 1.011 R11             |                      |          |          |          |          |                        |                      |          |          |          |           |                        |                      |                        |           |           |  |  |  |  |  |  |  |
|     | 1.100 R12             |                      |          |          |          |          |                        |                      |          |          |          |           |                        |                      |                        |           |           |  |  |  |  |  |  |  |
|     | 1.101 R13             |                      |          |          |          |          |                        |                      |          |          |          |           |                        |                      |                        |           |           |  |  |  |  |  |  |  |
|     | 1.110 R14             |                      |          |          |          |          |                        |                      |          |          |          |           |                        |                      |                        |           |           |  |  |  |  |  |  |  |
|     | 1.111 R15             |                      |          |          |          |          |                        |                      |          |          |          |           |                        |                      |                        |           |           |  |  |  |  |  |  |  |

# The Ultimate Authority

<https://software.intel.com/en-us/articles/intel-sdm>

All 5066 pages of the full glory...

“The x86 isn't all that complex...

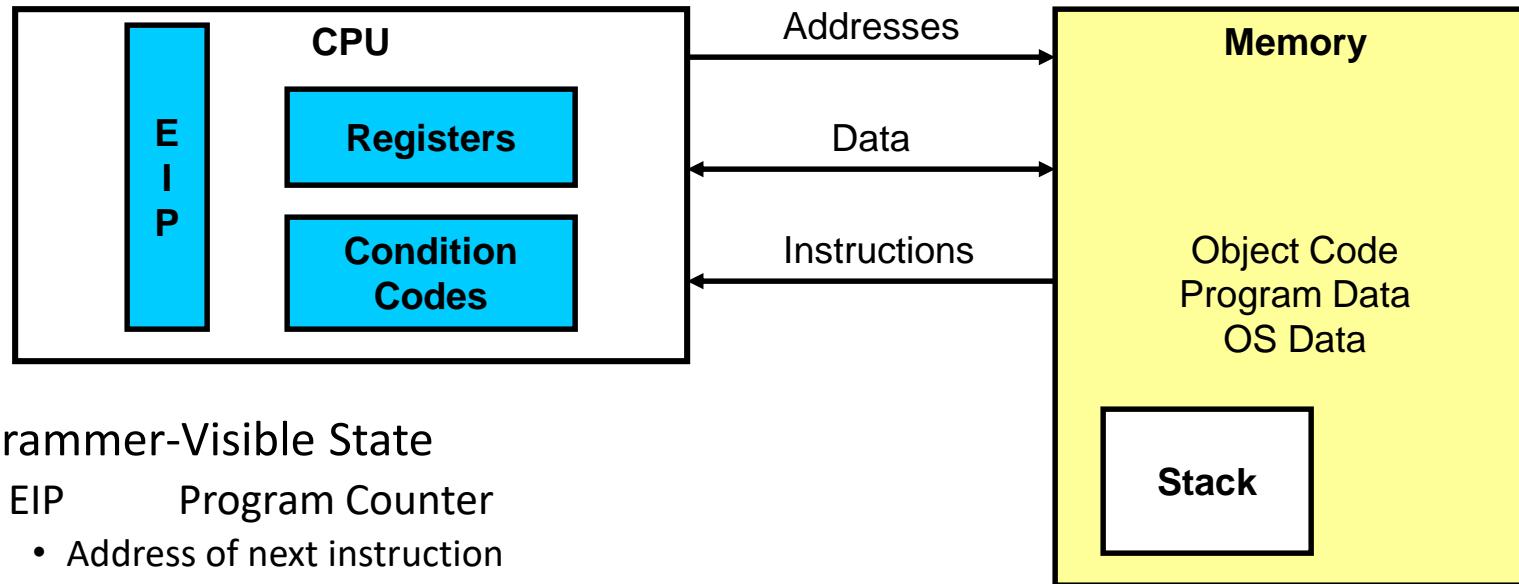
it just doesn't make a lot of sense”

Mike Johnson, AMD, 1994

# IA programming

Slides adapted from CMU's 2002 mounting of 15-213 module

# Assembly Programmer's View



- Programmer-Visible State

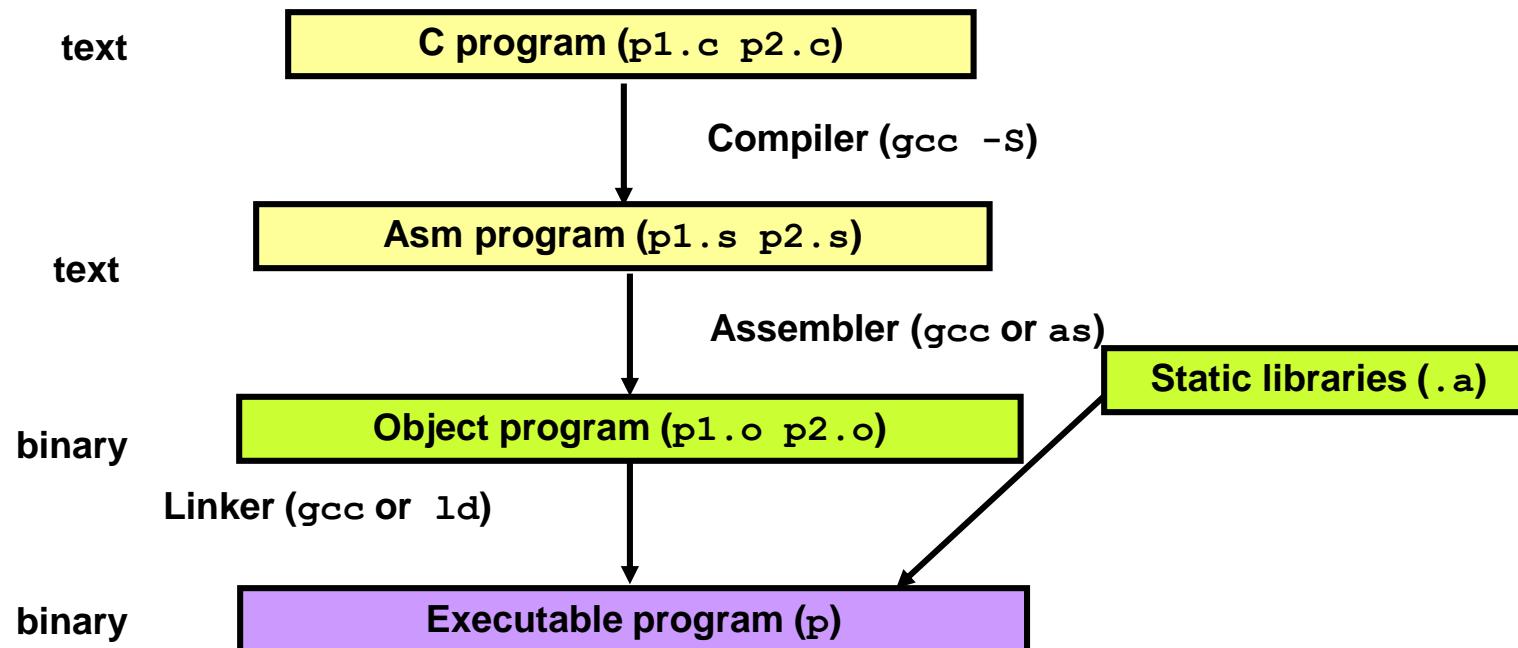
- EIP      Program Counter
  - Address of next instruction
- Register File
  - Heavily used program data
- Condition Codes
  - Store status information about most recent arithmetic operation
  - Used for conditional branching

- Memory

- Byte addressable array
- Code, user data, (some) OS data
- Includes stack used to support procedures

# Turning C into Object Code

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



# Compiling Into Assembly

- C Code

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

## Generated Assembly

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

Obtain with command

```
gcc -O -S code.c
```

Produces file code.s

# Assembly Characteristics

- Minimal 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
- Primitive 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

# Beware: the two x86 assembly syntax

- Intel syntax: destination operand before source operands
- AT&T syntax: source operands come first before destination operand
- Different assemblers use one or both

# Machine Instruction Example

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

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

Similar to  
expression  $x += y$

```
0x401046:
```

```
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 0x401046

# Disassembling Object Code

## Disassembled

```
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

# Moving Data

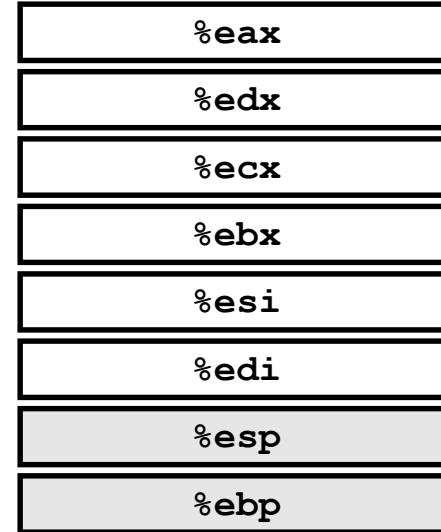
- Moving Data

`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 reserved for special use
  - Others have special uses for particular instructions
- Memory: 4 consecutive bytes of memory
  - Various “address modes”



# movl Operand Combinations

|      | <b>Source</b> | <b>Destination</b> | <b>C Analog</b>                 |
|------|---------------|--------------------|---------------------------------|
| movl | <i>Imm</i>    | <i>Reg</i>         | <code>movl \$0x4,%eax</code>    |
|      |               | <i>Mem</i>         | <code>movl \$-147,(%eax)</code> |
|      | <i>Reg</i>    | <i>Reg</i>         | <code>movl %eax,%edx</code>     |
|      | <i>Mem</i>    | <i>Mem</i>         | <code>movl %eax,(%edx)</code>   |
|      |               | <i>Reg</i>         | <code>movl (%eax),%edx</code>   |

- Cannot do memory-memory transfers with single instruction

# Simple 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`

# 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
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
```

The assembly code is grouped into three sections by curly braces on the right:

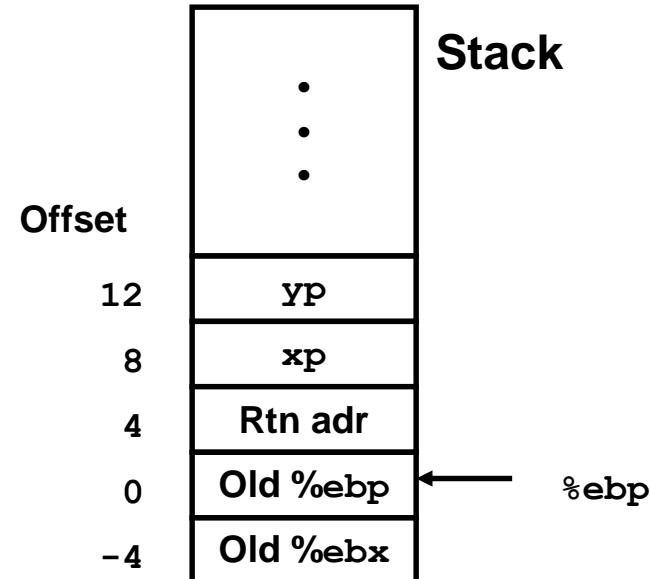
- Set Up:** pushl %ebp  
movl %esp,%ebp  
pushl %ebx
- Body:** movl 12(%ebp),%ecx  
movl 8(%ebp),%edx  
movl (%ecx),%eax  
movl (%edx),%ebx  
movl %eax,(%edx)  
movl %ebx,(%ecx)
- Finish:** movl -4(%ebp),%ebx  
movl %ebp,%esp  
popl %ebp  
ret

# Understanding Swap

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

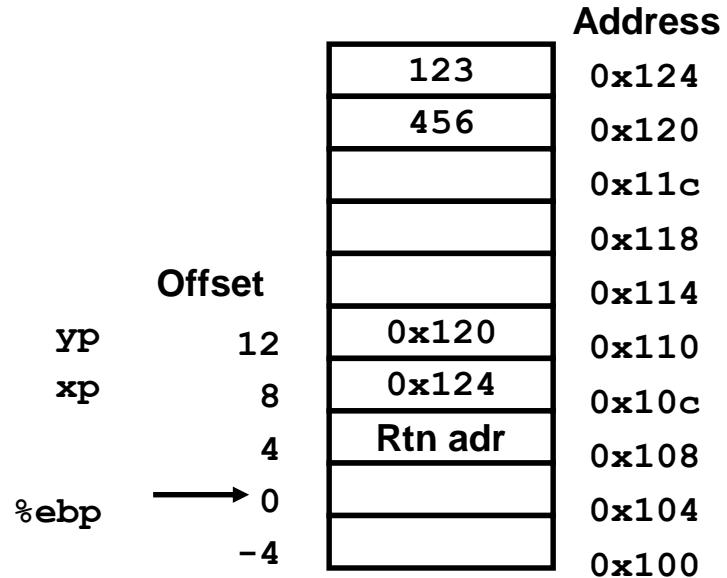
| Register | Variable |
|----------|----------|
| %ecx     | yp       |
| %edx     | xp       |
| %eax     | t1       |
| %ebx     | t0       |

```
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

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



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

# Understanding Swap

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

|         | Address |
|---------|---------|
| 123     | 0x124   |
| 456     | 0x120   |
|         | 0x11c   |
|         | 0x118   |
|         | 0x114   |
| 0x120   | 0x110   |
| 0x124   | 0x10c   |
| Rtn adr | 0x108   |
|         | 0x104   |
|         | 0x100   |

Offset

yp      12  
xp      8  
      4  
%ebp → 0  
     -4

```
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

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

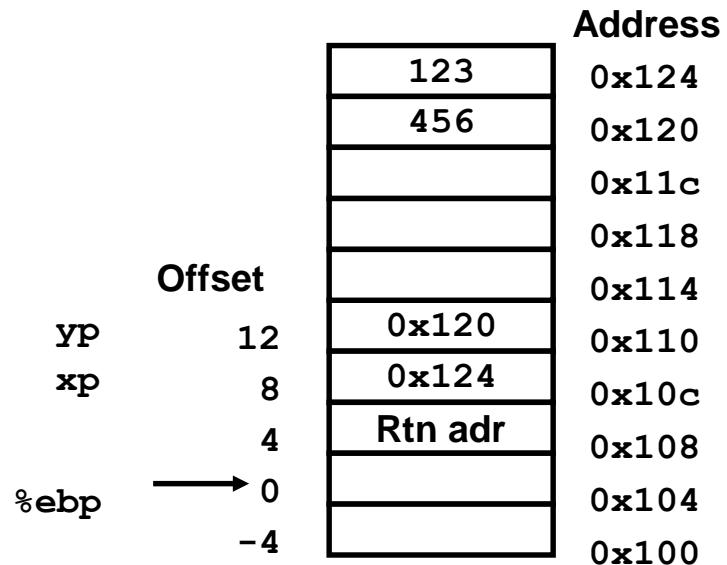
|         | Address |
|---------|---------|
| 123     | 0x124   |
| 456     | 0x120   |
|         | 0x11c   |
|         | 0x118   |
|         | 0x114   |
| 0x120   | 0x110   |
| 0x124   | 0x10c   |
| Rtn adr | 0x108   |
|         | 0x104   |
|         | 0x100   |

Offset  
yp      12  
xp      8  
%ebp    4  
          → 0  
          -4

```
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

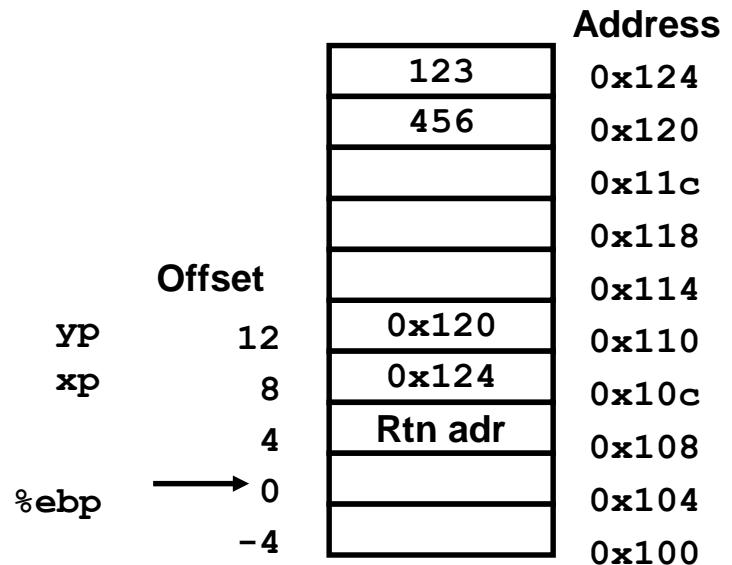
|      |       |
|------|-------|
| %eax | 456   |
| %edx | 0x124 |
| %ecx | 0x120 |
| %ebx |       |
| %esi |       |
| %edi |       |
| %esp |       |
| %ebp | 0x104 |



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

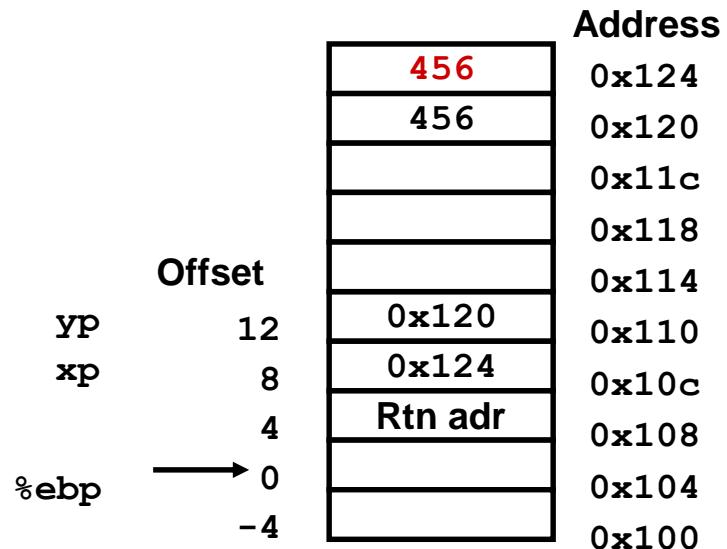
# Understanding Swap

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



# Understanding Swap

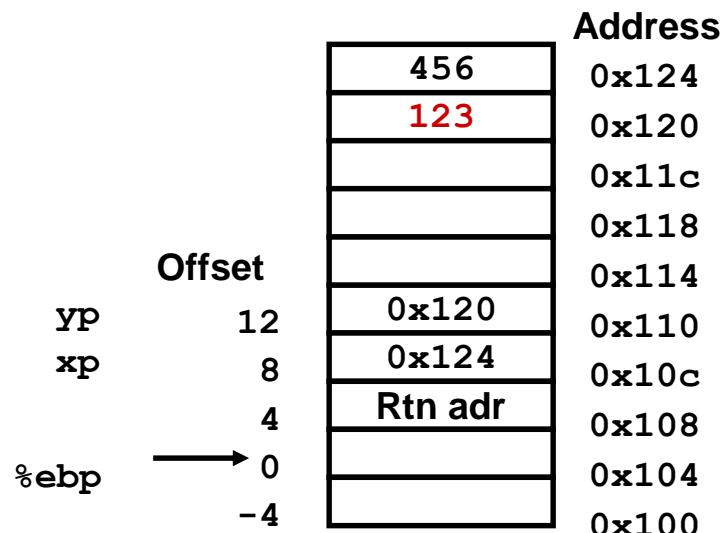
|      |       |
|------|-------|
| %eax | 456   |
| %edx | 0x124 |
| %ecx | 0x120 |
| %ebx | 123   |
| %esi |       |
| %edi |       |
| %esp |       |
| %ebp | 0x104 |



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

# Understanding Swap

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



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

# Indexed Addressing Modes

- Most General Form

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

- 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

- Special Cases

|            |   |
|------------|---|
| $(Rb, Ri)$ | $\text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri]]$ |
|------------|---|

|             |   |
|-------------|---|
| $D(Rb, Ri)$ | $\text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] + D]$ |
|-------------|---|

|               |   |
|---------------|---|
| $(Rb, Ri, S)$ | $\text{Mem}[\text{Reg}[Rb] + S * \text{Reg}[Ri]]$ |
|---------------|---|

# Address Computation Examples

|      |        |
|------|--------|
| %edx | 0xf000 |
| %ecx | 0x100  |

| Expression                 | Computation                   | Address              |
|----------------------------|-------------------------------|----------------------|
| <code>0x8(%edx)</code>     | <code>0xf000 + 0x8</code>     | <code>0xf008</code>  |
| <code>(%edx,%ecx)</code>   | <code>0xf000 + 0x100</code>   | <code>0xf100</code>  |
| <code>(%edx,%ecx,4)</code> | <code>0xf000 + 4*0x100</code> | <code>0xf400</code>  |
| <code>0x80(,%edx,2)</code> | <code>2*0xf000 + 0x80</code>  | <code>0x1e080</code> |

# Address Computation Instruction

- `leal Src,Dest`
  - *Src* is address mode expression
  - Set *Dest* to address denoted by expression
- Uses
  - Computing address without doing memory reference
    - E.g., translation of `p = &x[i];`
  - Computing arithmetic expressions of the form  $x + k*y$ 
    - $k = 1, 2, 4, \text{ or } 8.$

# Some Arithmetic Operations

| Format                     | Computation              |
|----------------------------|--------------------------|
| • Two Operand Instructions |                          |
| addl <i>Src,Dest</i>       | $Dest = Dest + Src$      |
| subl <i>Src,Dest</i>       | $Dest = Dest - Src$      |
| imull <i>Src,Dest</i>      | $Dest = Dest * Src$      |
| sall <i>Src,Dest</i>       | $Dest = Dest \ll Src$    |
| sarl <i>Src,Dest</i>       | $Dest = Dest \gg Src$    |
| shrl <i>Src,Dest</i>       | $Dest = Dest \gg Src$    |
| xorl <i>Src,Dest</i>       | $Dest = Dest \wedge Src$ |
| andl <i>Src,Dest</i>       | $Dest = Dest \& Src$     |
| orl <i>Src,Dest</i>        | $Dest = Dest \mid Src$   |

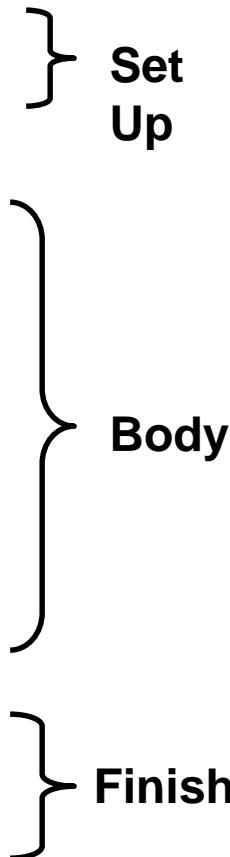
# Some Arithmetic Operations

| Format                     | Computation        |
|----------------------------|--------------------|
| • One Operand Instructions |                    |
| incl <i>Dest</i>           | $Dest = Dest + 1$  |
| decl <i>Dest</i>           | $Dest = Dest - 1$  |
| negl <i>Dest</i>           | $Dest = - Dest$    |
| notl <i>Dest</i>           | $Dest = \sim Dest$ |

# Using leal for Arithmetic Expressions

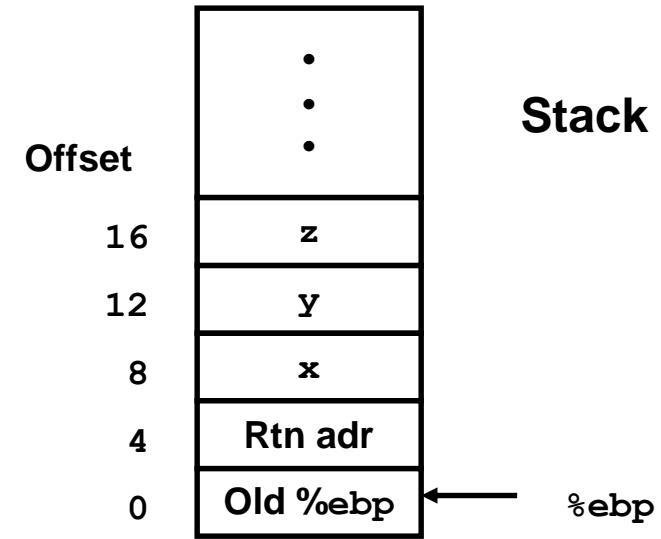
```
int arith  
    (int x, int y, int z)  
{  
    int t1 = x+y;  
    int t2 = z+t1;  
    int t3 = x+4;  
    int t4 = y * 48;  
    int t5 = t3 + t4;  
    int rval = t2 * t5;  
    return rval;  
}
```

```
arith:  
    pushl %ebp  
    movl %esp,%ebp  
  
    movl 8(%ebp),%eax  
    movl 12(%ebp),%edx  
    leal (%edx,%eax),%ecx  
    leal (%edx,%edx,2),%edx  
    sall $4,%edx  
    addl 16(%ebp),%ecx  
    leal 4(%edx,%eax),%eax  
    imull %ecx,%eax  
  
    movl %ebp,%esp  
    popl %ebp  
    ret
```



# Understanding arith

```
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;
}
```



```
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
sall $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)
```

# Understanding arith

```
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
```

# Condition Codes

- Single Bit Registers

CF      Carry Flag

SF      Sign Flag

ZF      Zero Flag

OF      Overflow Flag

- Implicitly Set By Arithmetic Operations

`addl Src,Dest`

C analog:  $t = a + b$

- CF set if carry out from most significant bit
  - Used to detect unsigned overflow
- ZF set if  $t == 0$
- SF set if  $t < 0$
- OF set if two's complement overflow  
$$(a>0 \ \&\& \ b>0 \ \&\& \ t<0) \ \|\ (a<0 \ \&\& \ b<0 \ \&\& \ t>=0)$$
- *Not Set by leal instruction*

# Setting Condition Codes (cont.)

- Explicit Setting by Compare Instruction

`cmpl Src2,Src1`

- `cmpl b,a` like computing  $a-b$  without setting destination
- CF set if carry out from most significant bit
  - Used for unsigned comparisons
- ZF set if  $a == b$
- SF set if  $(a-b) < 0$
- OF set if two's complement overflow

$(a>0 \ \&\& \ b<0 \ \&\& \ (a-b)<0) \ \mid\mid \ (a<0 \ \&\& \ b>0 \ \&\& \ (a-b)>0)$

# Setting Condition Codes (cont.)

- Explicit Setting by Test instruction

`testl Src2,Src1`

- Sets condition codes based on value of *Src1* & *Src2*
  - Useful to have one of the operands be a mask
- `testl b,a` like computing `a&b` without setting destination
- ZF set when `a&b == 0`
- SF set when `a&b < 0`

# Jumping

- jX Instructions
  - Jump to different part of code depending on condition codes

| jX  | Condition      | Description               |
|-----|----------------|---------------------------|
| jmp | 1              | Unconditional             |
| je  | ZF             | Equal / Zero              |
| jne | ~ZF            | Not Equal / Not Zero      |
| js  | SF             | Negative                  |
| jns | ~SF            | Nonnegative               |
| jg  | ~(SF^OF) & ~ZF | Greater (Signed)          |
| jge | ~(SF^OF)       | Greater or Equal (Signed) |
| jl  | (SF^OF)        | Less (Signed)             |
| jle | (SF^OF)   ZF   | Less or Equal (Signed)    |
| ja  | ~CF & ~ZF      | Above (unsigned)          |
| jb  | CF             | Below (unsigned)          |

# Conditional Branch Example

```
int max(int x, int y)
{
    if (x > y)
        return x;
    else
        return y;
}
```

\_max:

L9:

```
pushl %ebp  
movl %esp,%ebp  
  
movl 8(%ebp),%edx  
movl 12(%ebp),%eax  
cmpl %eax,%edx  
jle L9  
movl %edx,%eax  
  
movl %ebp,%esp  
popl %ebp  
ret
```

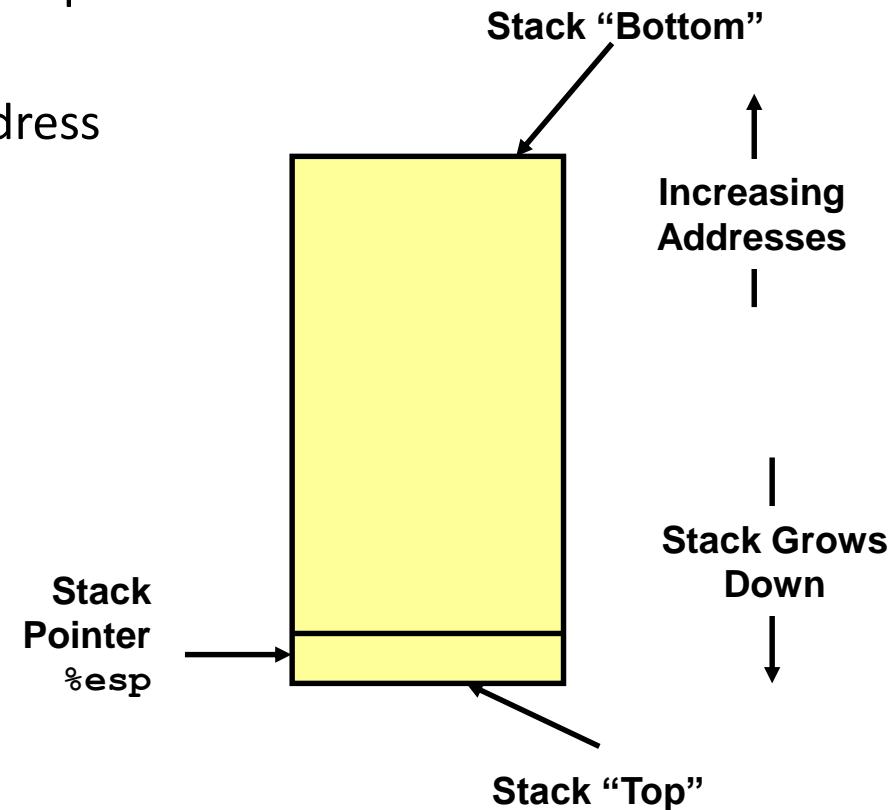
} Set Up

} Body

} Finish

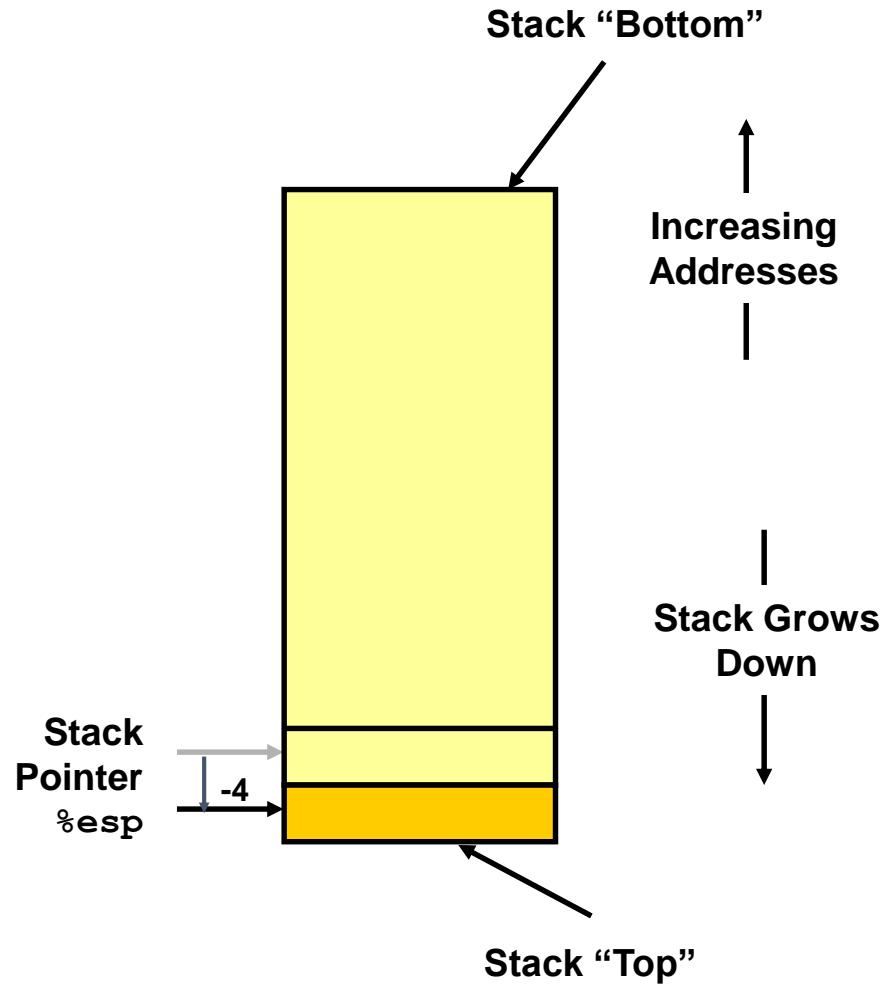
# IA32 Stack

- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%esp` indicates lowest stack address
  - address of top element



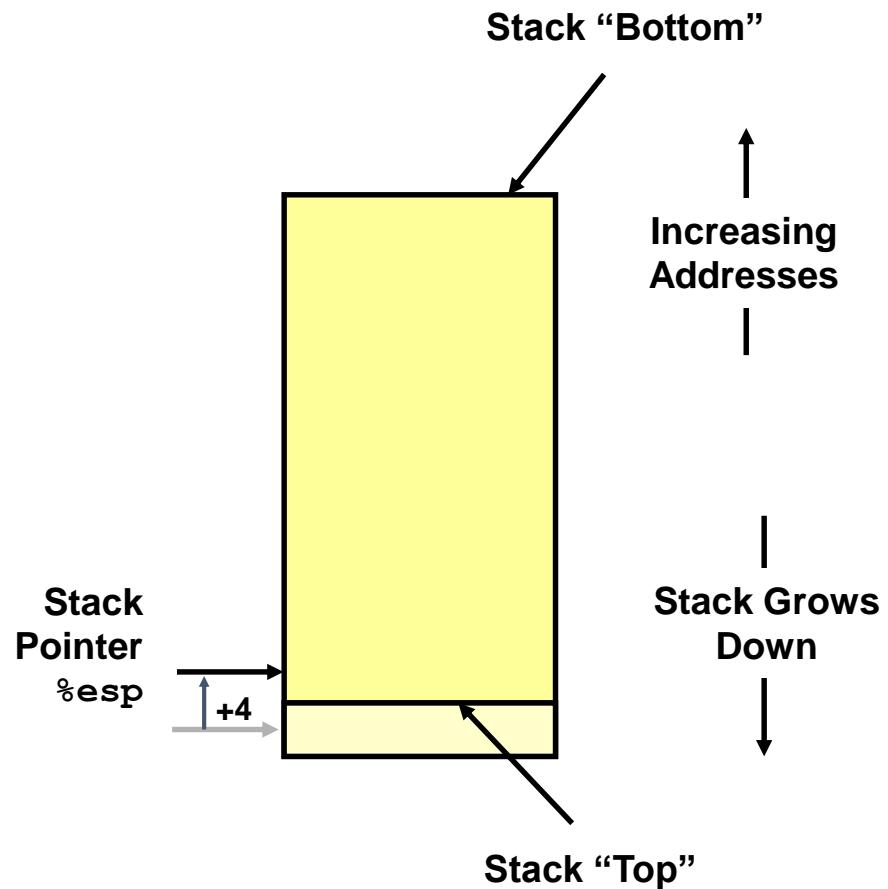
# IA32 Stack Pushing

- Pushing
  - **pushl Src**
  - Fetch operand at *Src*
  - Decrement **%esp** by 4
  - Write operand at address given by **%esp**

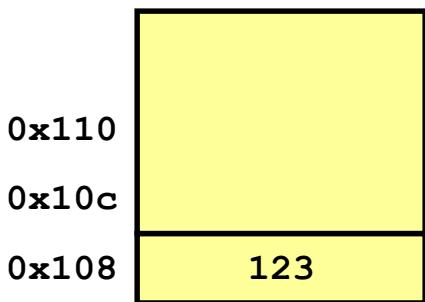


# IA32 Stack Popping

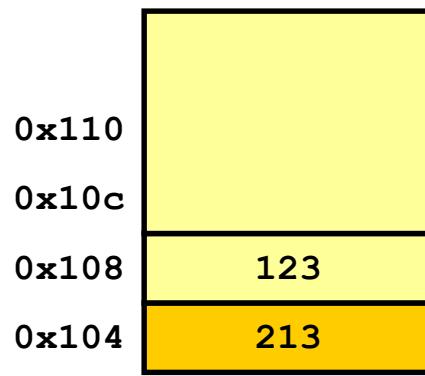
- Popping
  - `popl Dest`
  - Read operand at address given by `%esp`
  - Increment `%esp` by 4
  - Write to `Dest`



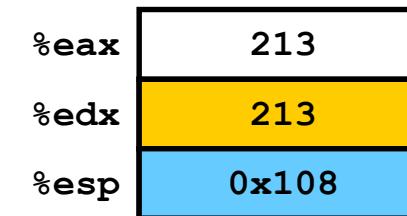
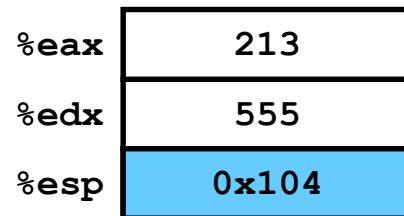
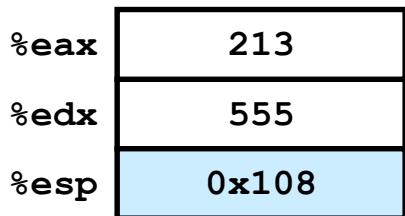
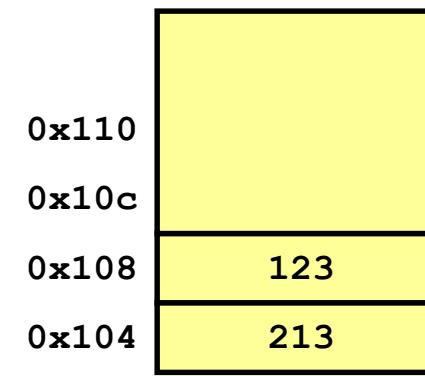
# Stack Operation Examples



**pushl %eax**



**popl %edx**



# Procedure Control Flow

- Use stack to support procedure call and return

- Procedure call:

`call label` Push return address on stack; Jump to `label`

- Return address value

- Address of instruction beyond `call`

- Example from disassembly

`804854e: e8 3d 06 00 00      call    8048b90 <main>`

`8048553: 50                      pushl    %eax`

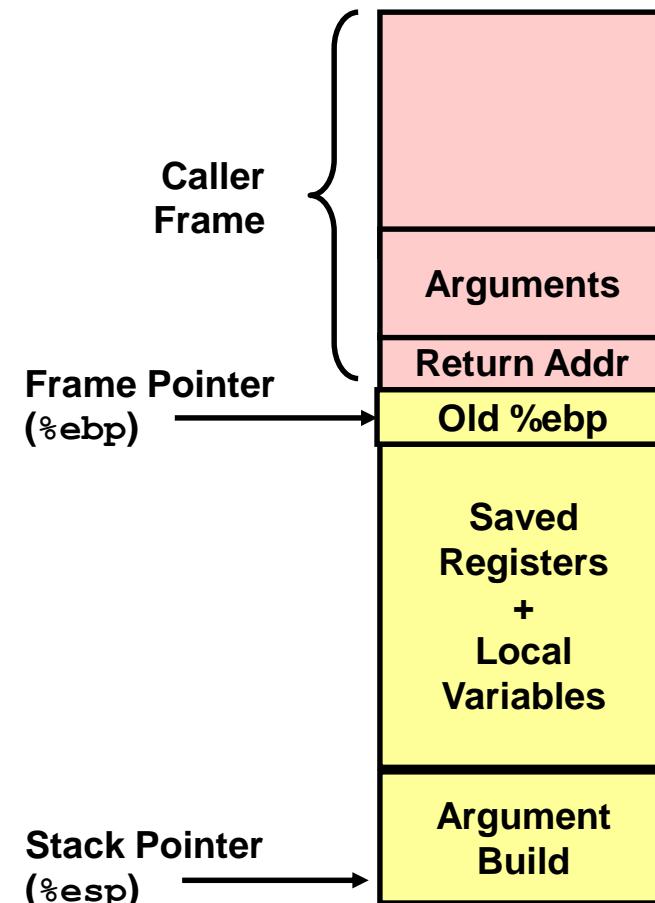
- Return address = `0x8048553`

- Procedure return:

- `ret` Pop address from stack; Jump to address

# IA32/Linux Stack Frame

- Current Stack Frame (“Top” to Bottom)
  - Parameters for function about to call
    - “Argument build”
  - Local variables
    - If can’t keep in registers
  - Saved register context
  - Old frame pointer
- Caller Stack Frame
  - Return address
    - Pushed by `call` instruction
  - Arguments for this call



# Revisiting swap

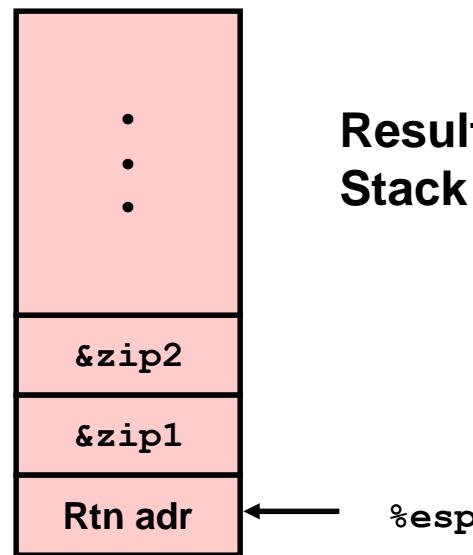
```
int zip1 = 15213;
int zip2 = 91125;

void call_swap()
{
    swap(&zip1, &zip2);
}
```

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

## Calling swap from call\_swap

```
call_swap:
• • •
pushl $zip2          # Global Var
pushl $zip1          # Global Var
call swap
• • •
```



Resulting Stack

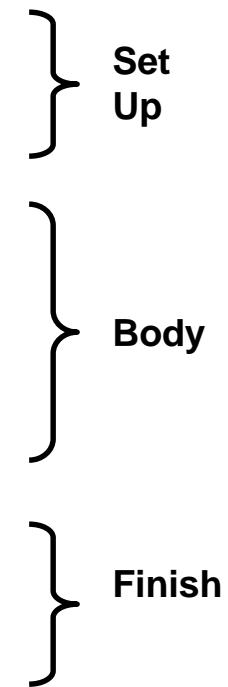
# Revisiting swap

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

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

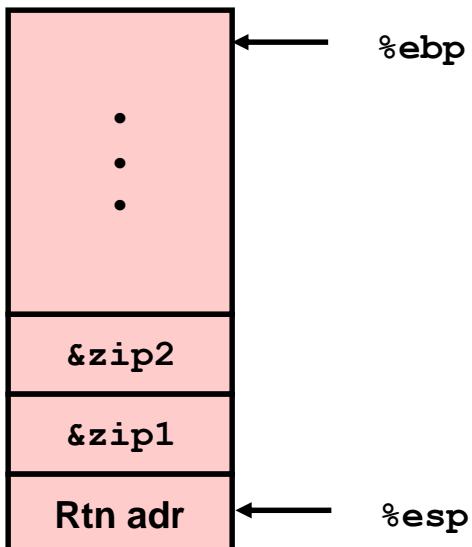
    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)

    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

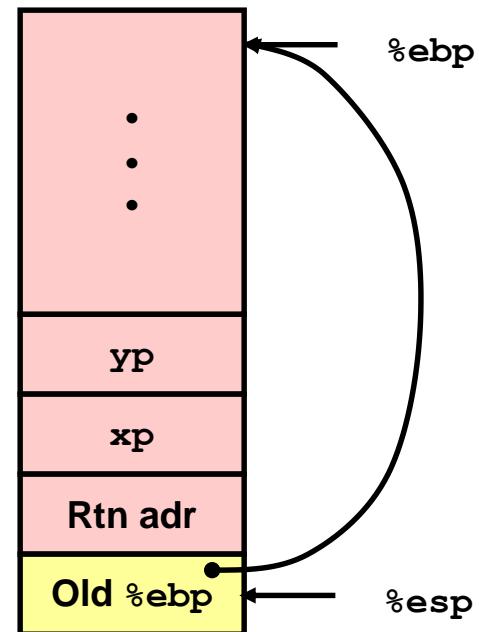


# swap Setup #1

Entering Stack



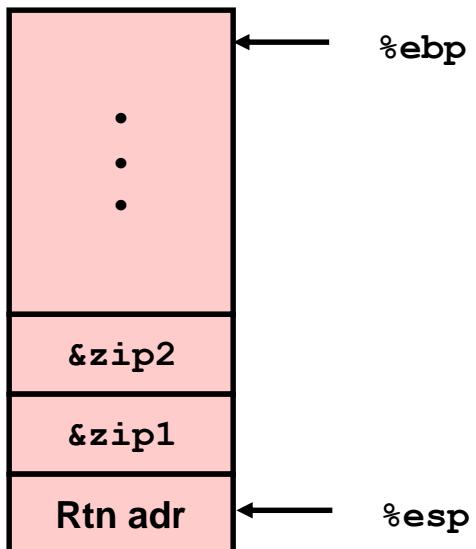
Resulting Stack



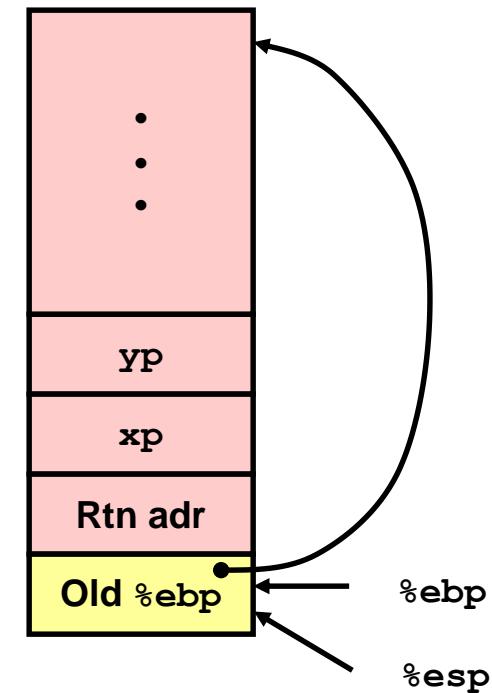
```
swap:  
    pushl %ebp  
    movl %esp,%ebp  
    pushl %ebx
```

# swap Setup #2

Entering  
Stack



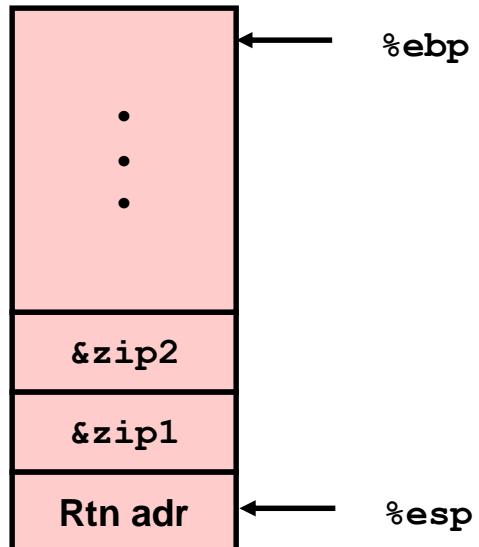
Resulting  
Stack



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

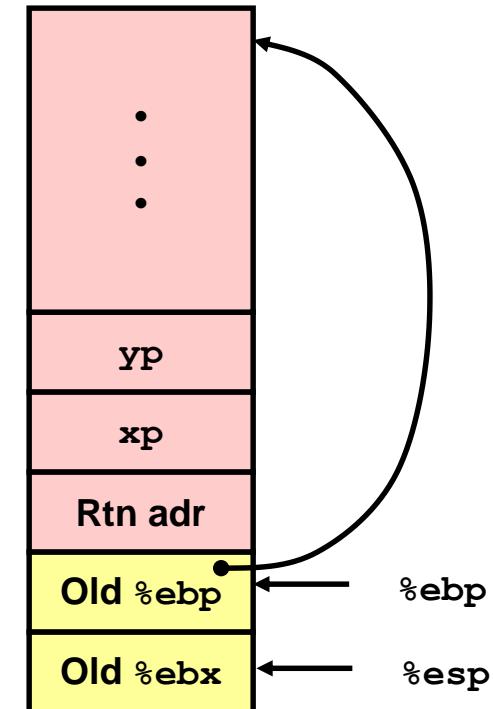
# swap Setup #3

Entering Stack

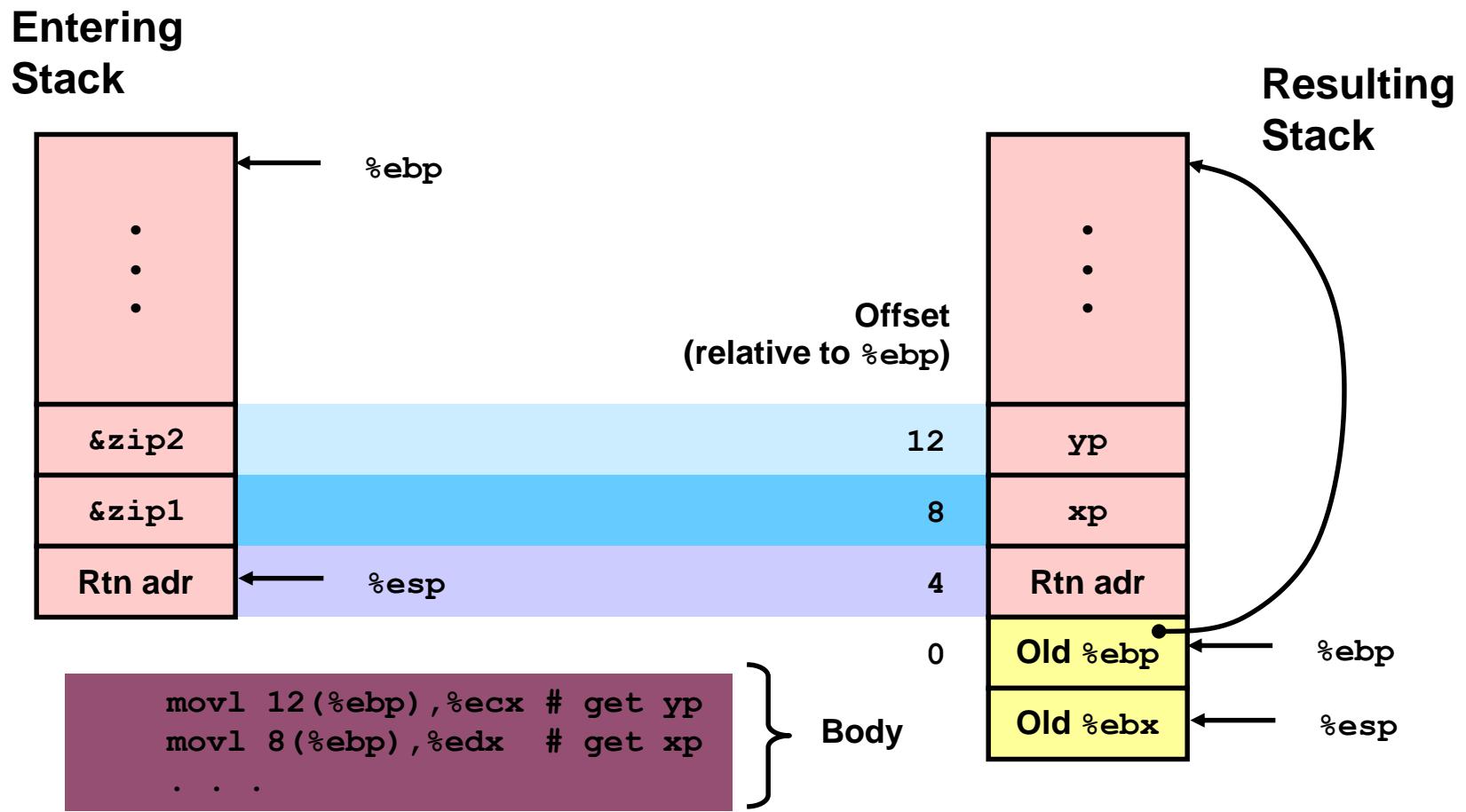


```
swap:  
    pushl %ebp  
    movl %esp,%ebp  
    pushl %ebx
```

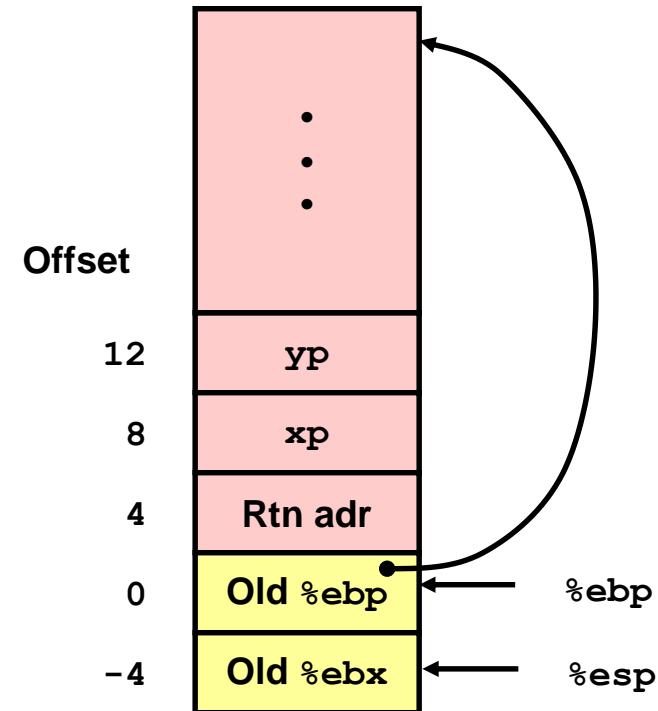
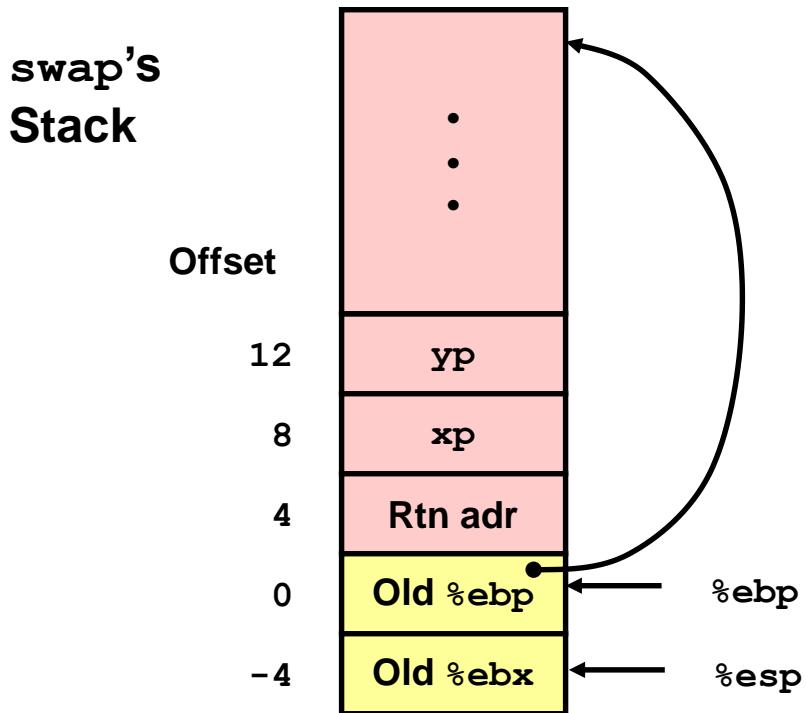
Resulting Stack



# Effect of **swap** Setup

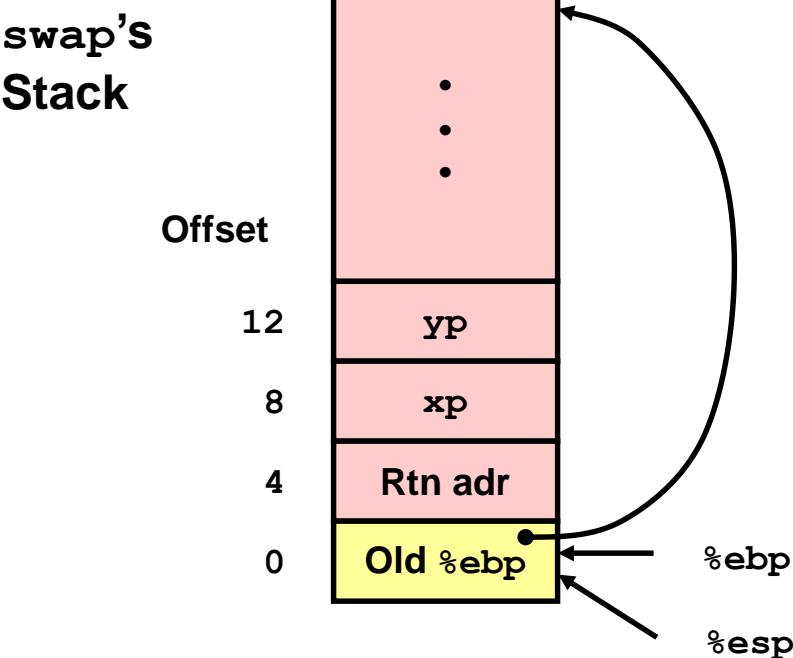
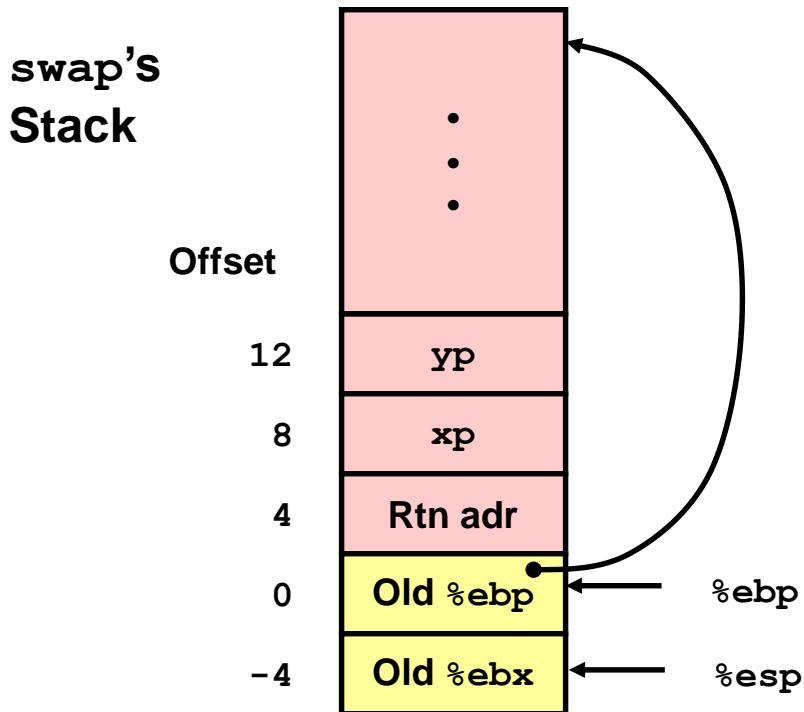


# swap Finish #1



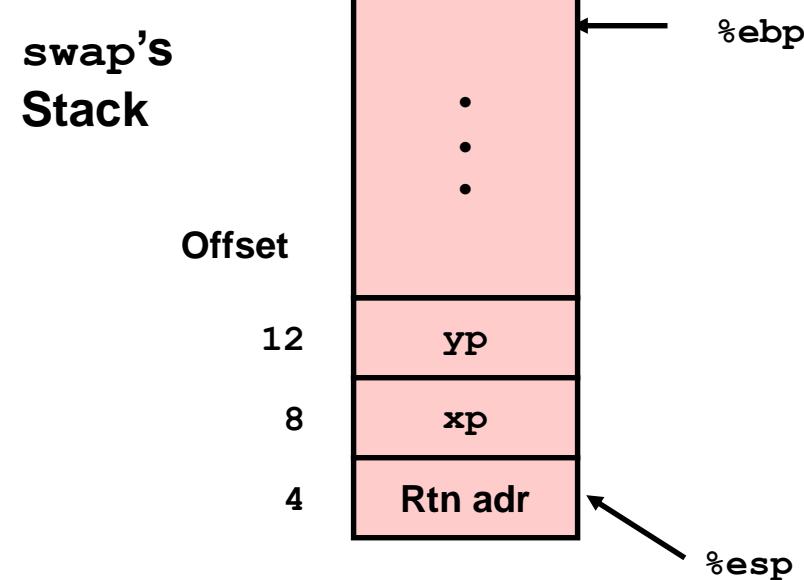
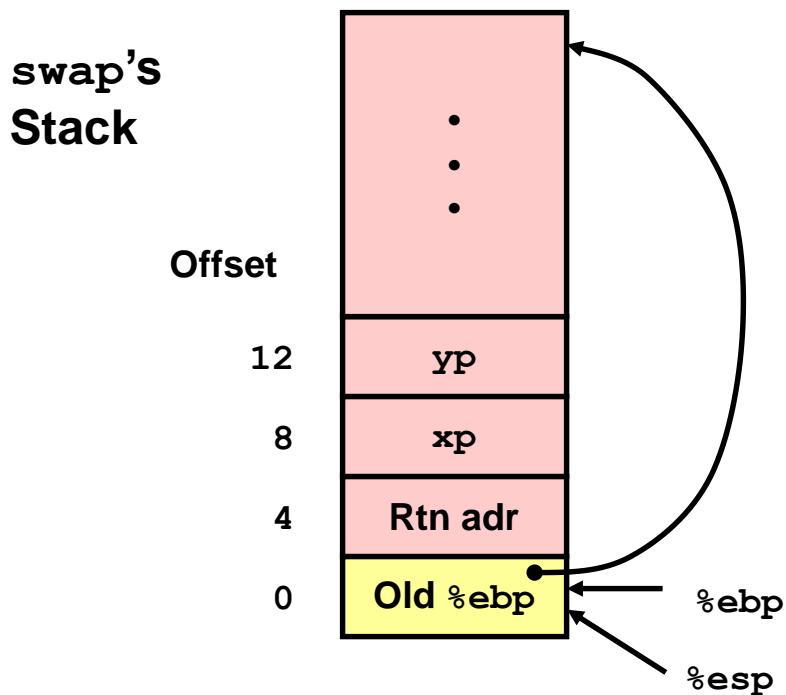
```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

## swap Finish #2



```
movl -4(%ebp), %ebx  
movl %ebp, %esp  
popl %ebp  
ret
```

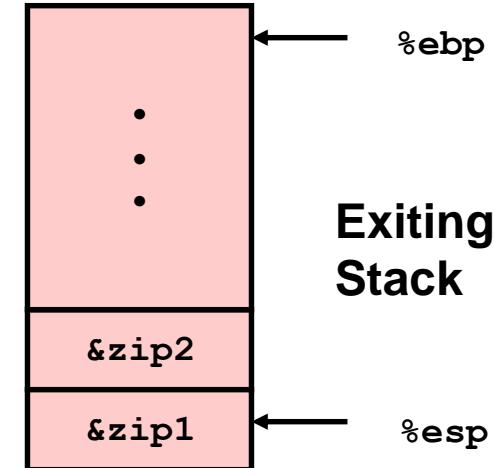
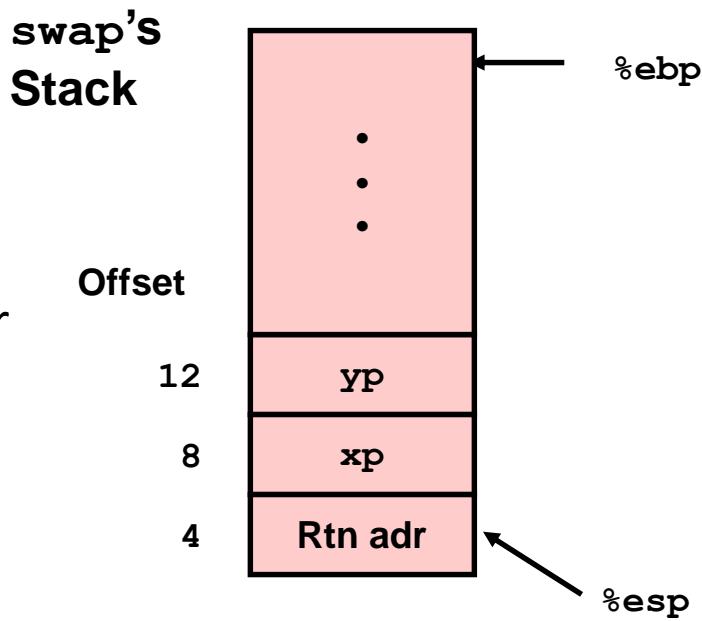
# swap Finish #3



```
movl -4(%ebp),%ebx  
movl %ebp,%esp  
popl %ebp  
ret
```

# swap Finish #4

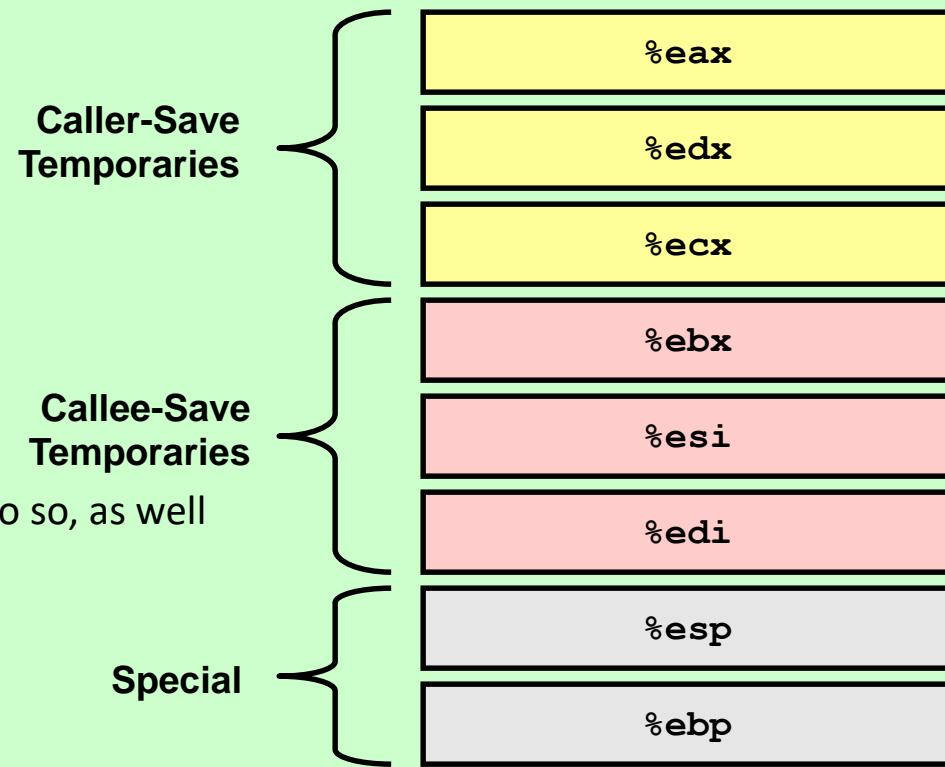
- Observation
  - Saved & restored register **%ebx**
  - Didn't do so for **%eax**, **%ecx**, or **%edx**



```
movl -4(%ebp),%ebx  
movl %ebp,%esp  
popl %ebp  
ret
```

# IA32/Linux Register Usage

- Integer Registers
  - Two have special uses  
**%ebp, %esp**
  - Three managed as callee-save  
**%ebx, %esi, %edi**
    - Old values saved on stack prior to using
  - Three managed as caller-save  
**%eax, %edx, %ecx**
    - Do what you please, but expect any callee to do so, as well
  - Register **%eax** also stores returned value



# Linux x86-64 Calling Convention

- Uses the System V AMD64 ABI
  - See: [https://en.wikipedia.org/wiki/X86\\_calling\\_conventions](https://en.wikipedia.org/wiki/X86_calling_conventions)
- Full details:  
<https://web.archive.org/web/20160801075146/http://www.x86-64.org/documentation/abi.pdf>

# Linux 64 bit Procedure Call Register Usage

| Register     | Usage   | Preserved across function calls |
|--------------|---|---------------------------------|
| %rax         | temporary register; with variable arguments passes information about the number of vector registers used; 1 <sup>st</sup> return register | No                              |
| %rbx         | callee-saved register; optionally used as base pointer  | Yes                             |
| %rcx         | used to pass 4 <sup>th</sup> integer argument to functions  | No                              |
| %rdx         | used to pass 3 <sup>rd</sup> argument to functions; 2 <sup>nd</sup> return register   | No                              |
| %rsp         | stack pointer   | Yes                             |
| %rbp         | callee-saved register; optionally used as frame pointer   | Yes                             |
| %rsi         | used to pass 2 <sup>nd</sup> argument to functions  | No                              |
| %rdi         | used to pass 1 <sup>st</sup> argument to functions  | No                              |
| %r8          | used to pass 5 <sup>th</sup> argument to functions  | No                              |
| %r9          | used to pass 6 <sup>th</sup> argument to functions  | No                              |
| %r10         | temporary register, used for passing a function's static chain pointer  | No                              |
| %r11         | temporary register  | No                              |
| %r12-%r15    | callee-saved registers  | Yes                             |
| %xmm0-%xmm1  | used to pass and return floating point arguments  | No                              |
| %xmm2-%xmm7  | used to pass floating point arguments   | No                              |
| %xmm8-%xmm15 | temporary registers   | No                              |
| %mmx0-%mmx7  | temporary registers   | No                              |
| %st0,%st1    | temporary registers; used to return long double arguments   | No                              |
| %st2-%st7    | temporary registers   | No                              |
| %fs          | Reserved for system (as thread specific data register)  | No                              |
| mxcsr        | SSE2 control and status word  | partial                         |
| x87 SW       | x87 status word   | No                              |
| x87 CW       | x87 control word  | Yes                             |

END