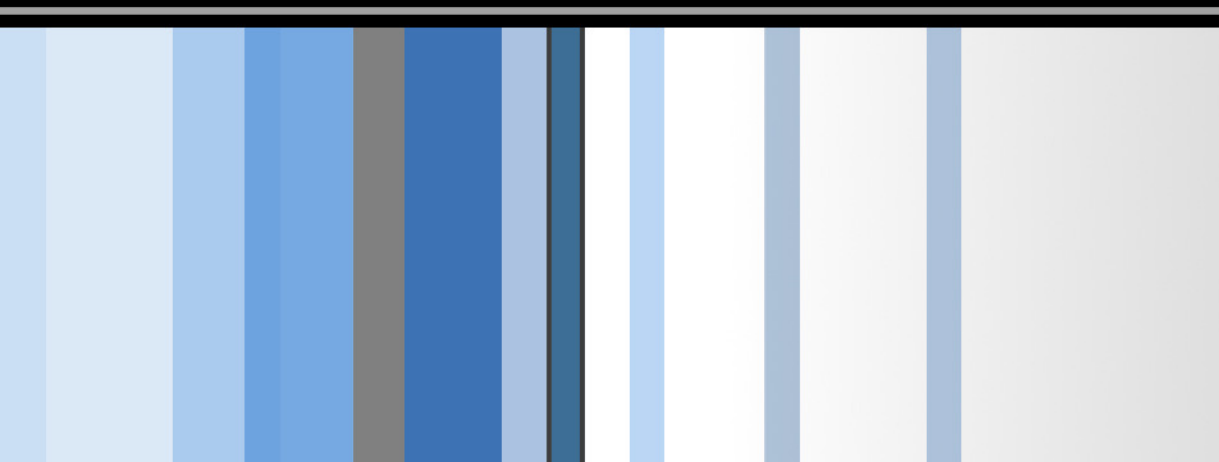


# CSC 471 Modern Malware Analysis

## IA-32 Registers & Byte Ordering

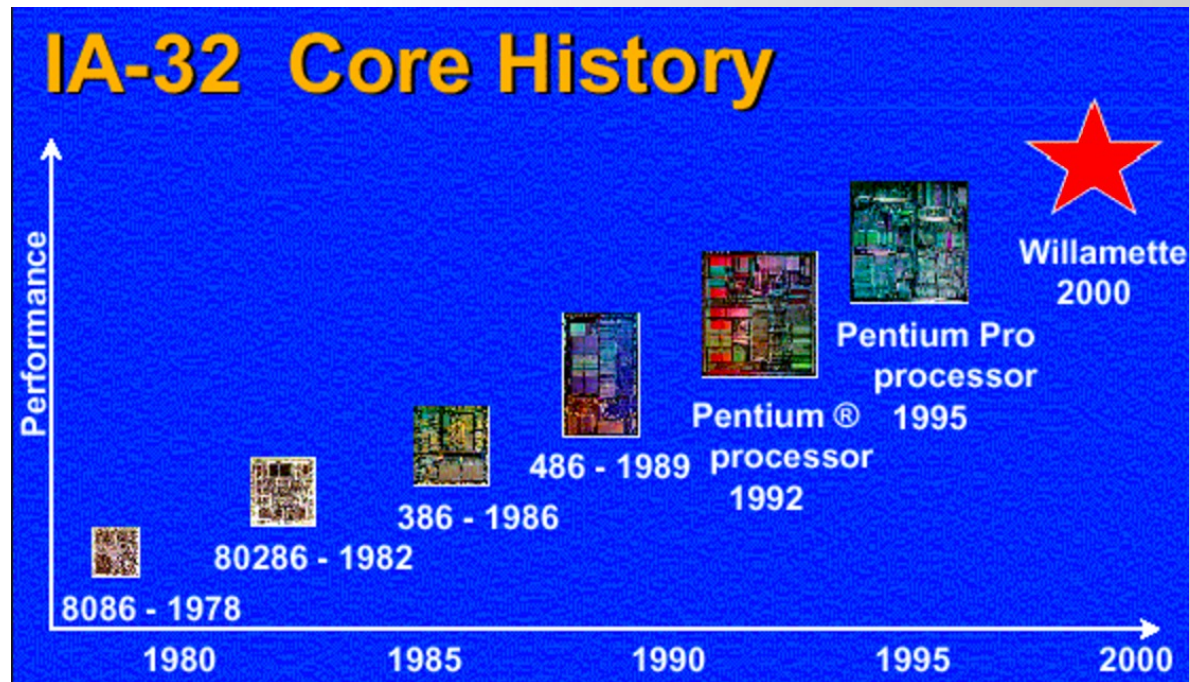
Si Chen (schen@wcupa.edu)



# IA-32 Register

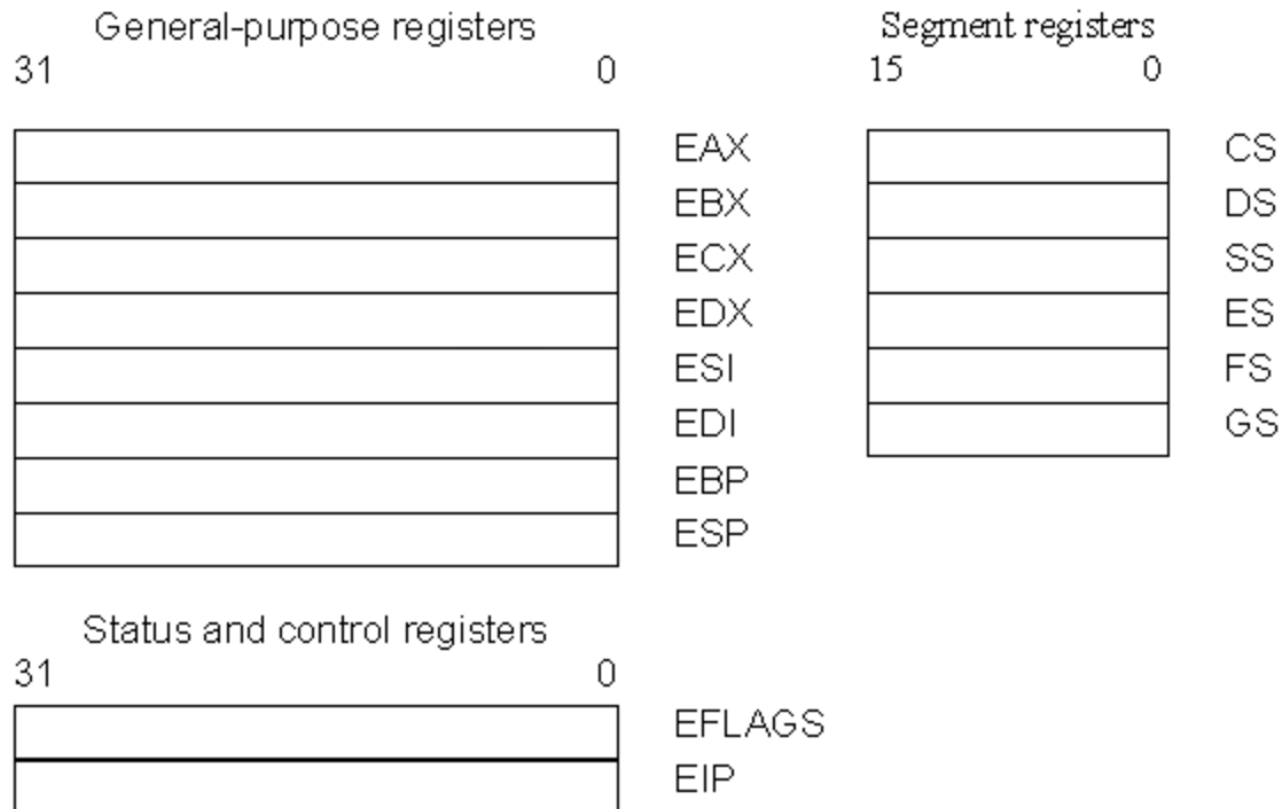
# Intel IA-32 Processor

- Intel uses IA-32 to refer to Pentium processor family, in order to distinguish them from their 64-bit architectures.




# Register Set

- There are three types of registers:
  - general-purpose data registers,
  - segment registers,
  - status and control registers.



# General-purpose Registers

- The **eight** 32-bit general-purpose data registers are used to hold operands for logical and arithmetic operations, operands for address calculations and memory pointers

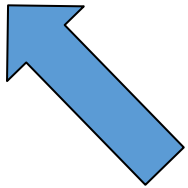
General-purpose registers				16-bit	32-bit
31	16	15	8 7	0	
 4 Bytes		AH	AL	AX	EAX
		BH	BL	BX	EBX
		CH	CL	CX	ECX
		DH	DL	DX	EDX
		BP			ESI
		SI			EDI
		DI			EBP
		SP			ESP

# Other uses...

- EAX—Accumulator for operands and results data.
  - EBX—Pointer to data in the DS segment.
  - ECX—Counter for string and loop operations.
  - EDX—I/O pointer.
- 
1. We use these four registers when we perform arithmetic operations (ADD, SUB, XOR, OR) -- store constant or variable's value.
  2. Some assembly operations (MUL, DIV, LODS) directly operate these register and altered the value when finished.
  3. ECX is used for loop count → decrease 1 after each loop
  4. EAX is used for storing the return value of a function (Win32 API)

# Other uses...

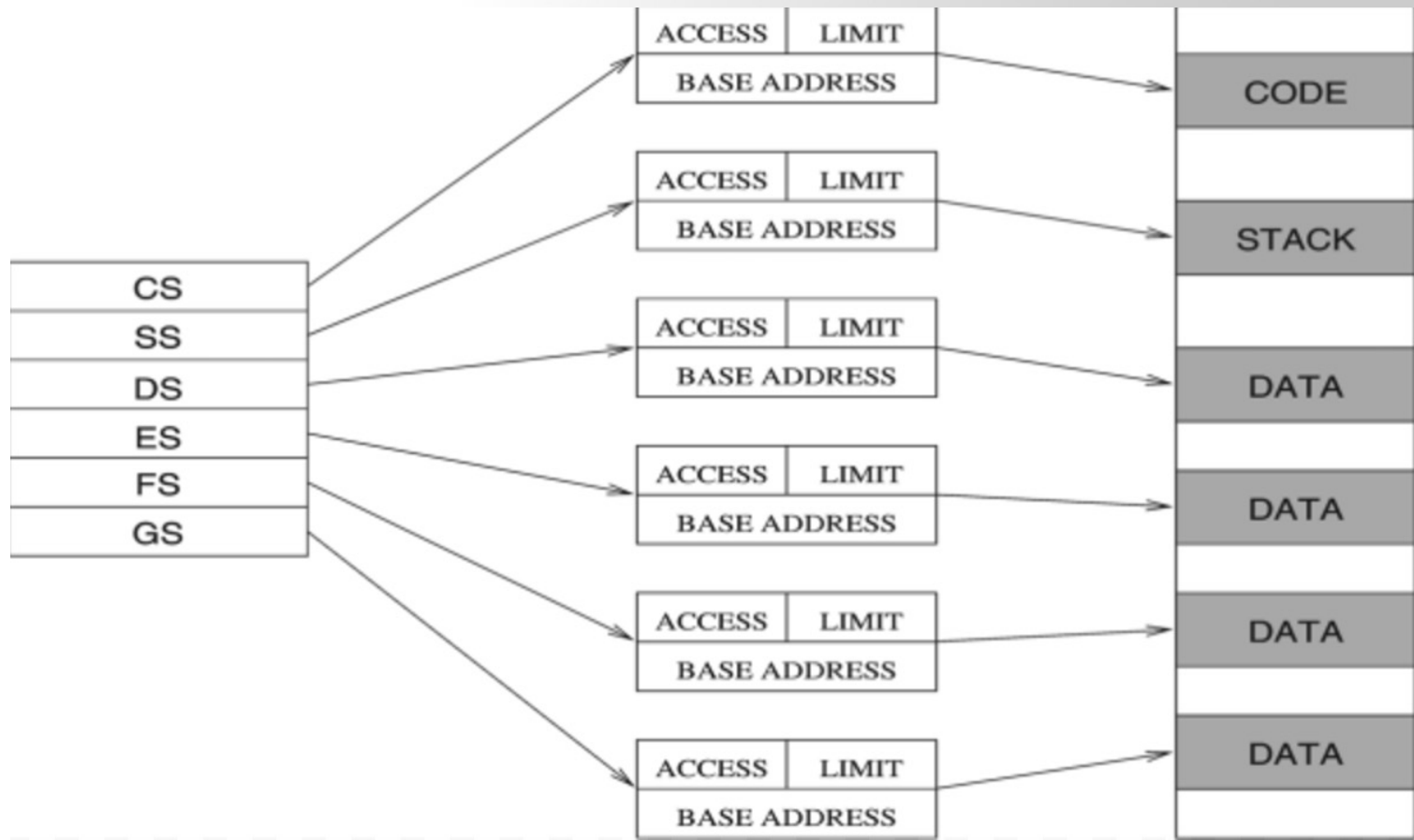
- ESI—Pointer to data in the segment pointed to by the DS register; source pointer for string operations.
- EDI—Pointer to data (or destination) in the segment pointed to by the ES register; destination pointer for string operations.
- **EBP—Pointer to data on the stack.**
- **ESP—Stack pointer.**



PUSH, POP, CALL, RET

# Segment Registers

- There are six segment registers that hold 16-bit segment selectors. A segment selector is a special pointer that identifies a segment in memory.
  - CS: code segment register
  - SS: stack segment register
  - DS, ES, FS, GS: data segment registers



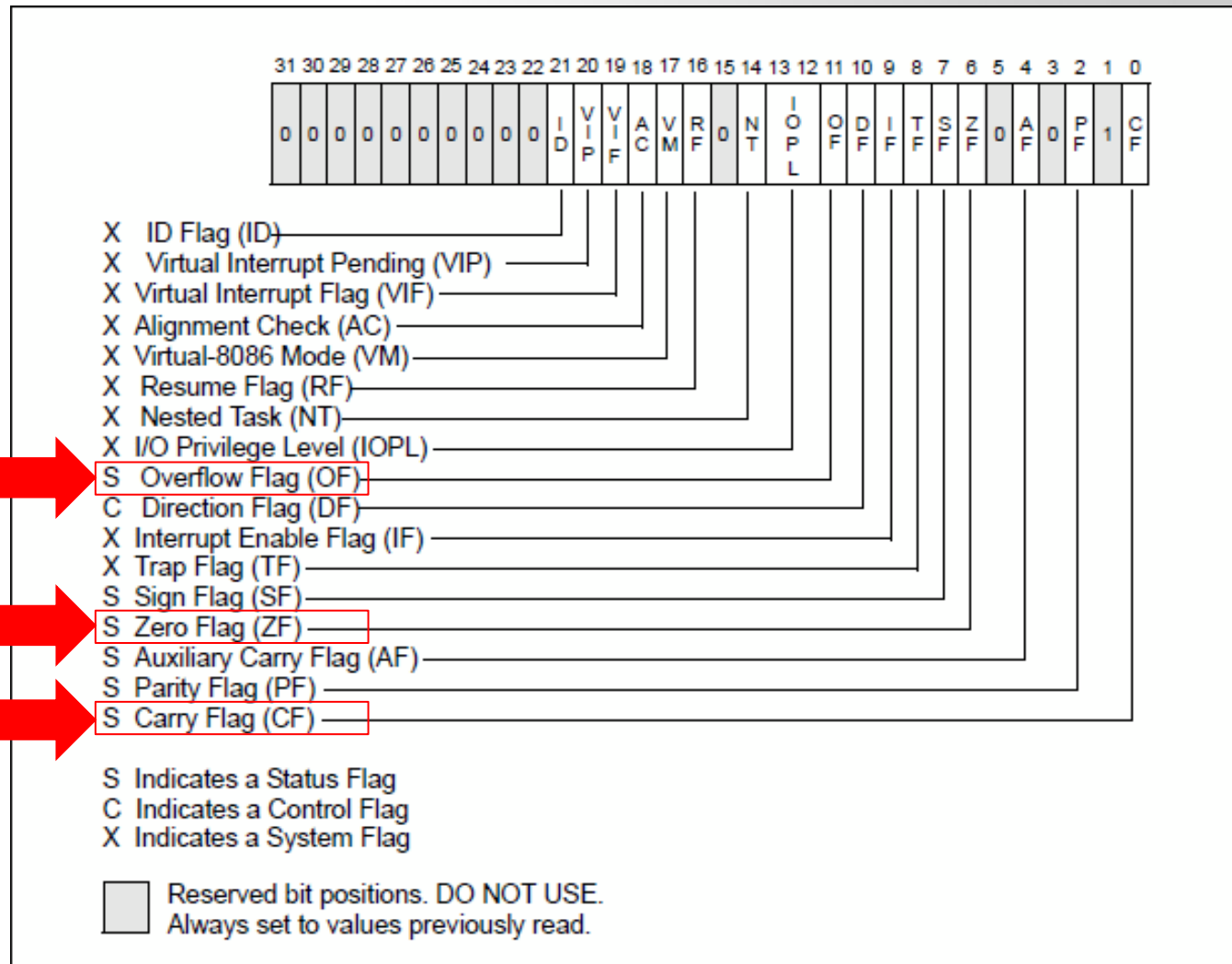


# Status and Control Registers

31	0
	EFLAGS
	EIP

The 32-bit EFLAGS register contains **a group of status flags**, **a control flag**, and **a group of system flags**.

JCC



EFLAGS Register

# Status and Control Registers

Change to '1' if:

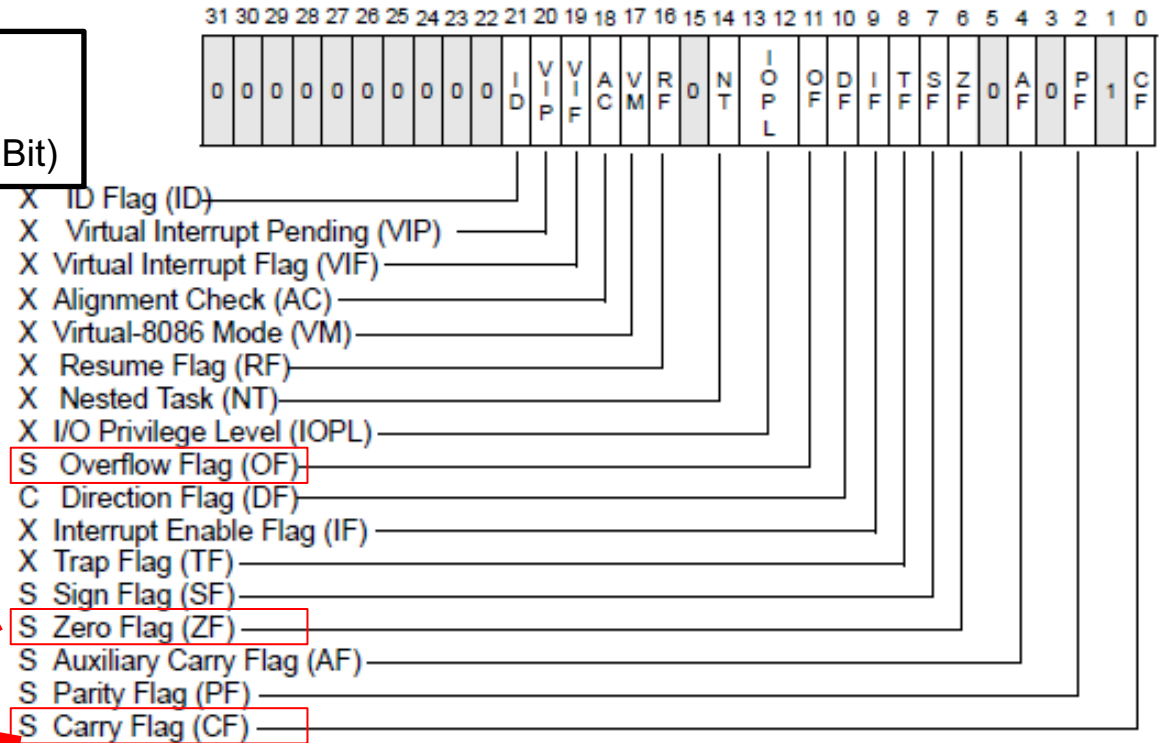
- Signed integer overflow
- Change in MSB (Most Significant Bit)

Change to '1' if:


- Calculation result is 0

Change to '1' if:

- unsigned integer overflow



S Indicates a Status Flag  
C Indicates a Control Flag  
X Indicates a System Flag

 Reserved bit positions. DO NOT USE.  
Always set to values previously read.

EFLAGS Register

## EIP Register (Instruction Pointer)

The EIP register (or instruction pointer) can also be called "**program counter**."

It contains the **offset** in the current code segment for the **next instruction to be executed**.

It is advanced from one instruction boundary to the next in straight-line code or it is moved ahead or backwards by a number of instructions when executing JMP, Jcc, CALL, RET, and IRET instructions.

# Byte Order

# Little endian

- IA-32 processors use "little endian" as their byte order. This means that the bytes of a word are numbered starting from the least significant byte and that the least significant bit starts of a word starts in the least significant byte.

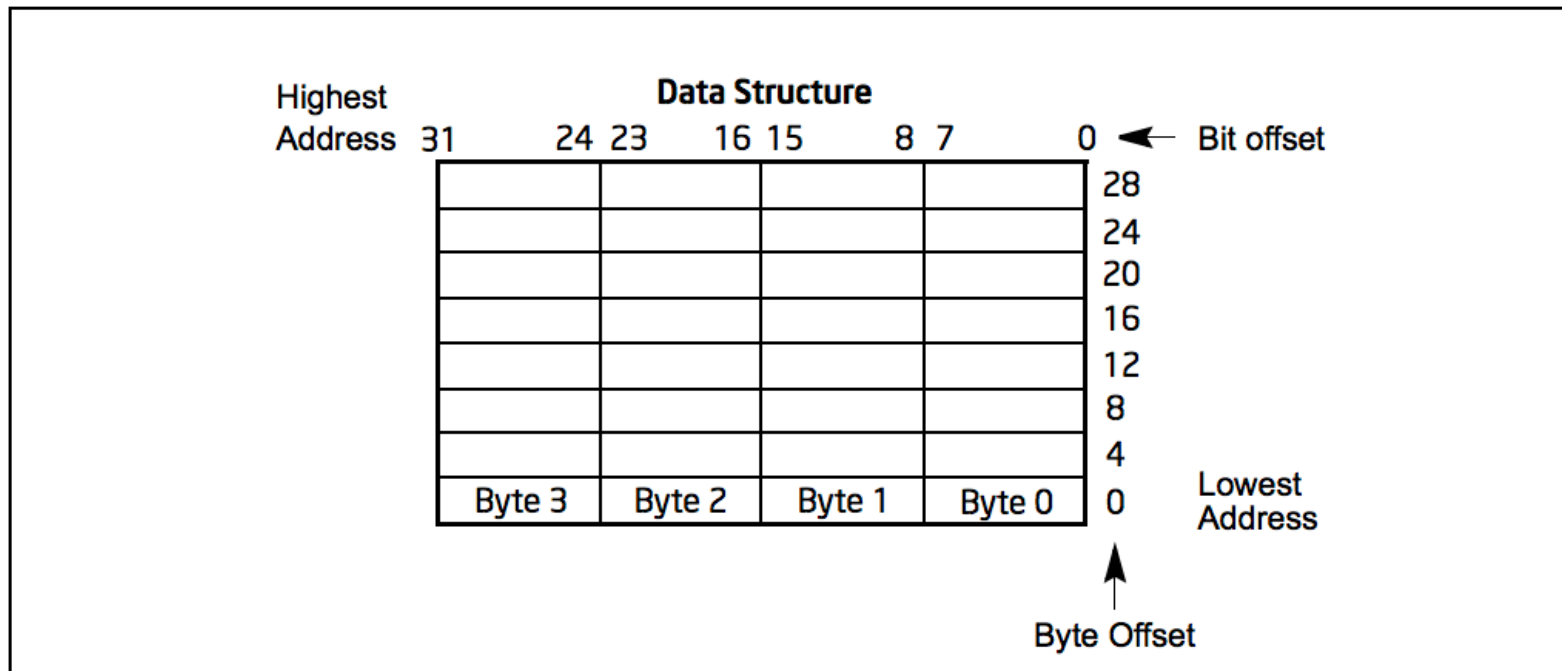


Figure 1-1. Bit and Byte Order

# Byte Order

	Low address				High address			
Address	0	1	2	3	4	5	6	7
Little-endian	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Big-endian	Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0
Memory content	0x11	0x22	0x33	0x44	0x55	0x66	0x77	0x88
64 bit value on Little-endian				64 bit value on Big-endian				
0x8877665544332211				0x1122334455667788				

# LittleEndian.cpp

```
1  #include "windows.h"
2
3  BYTE b = 0x12;
4  WORD w = 0x1234;
5  DWORD dw = 0x12345678;
6  char str[] = "abcde";
7
8
9  int main(int argc, char *argv[])
10 {
11     BYTE lb = b;
12     WORD lw = w;
13     DWORD ldw = dw;
14     char *lstr = str;
15
16
17     return 0;
18 }
19
```

# X86 ASM



# MOV

- Move **reg/mem** value to **reg/mem**

- mov A, B is "Move B to A" (A=B)
- Same data size

**mov eax, 0x1337**

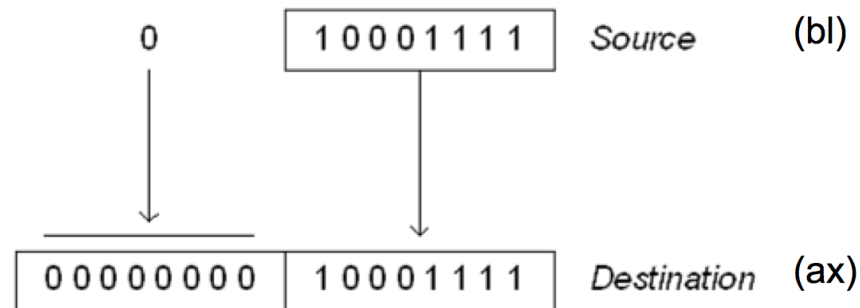
**mov bx, ax**

**mov [esp+4], bl**

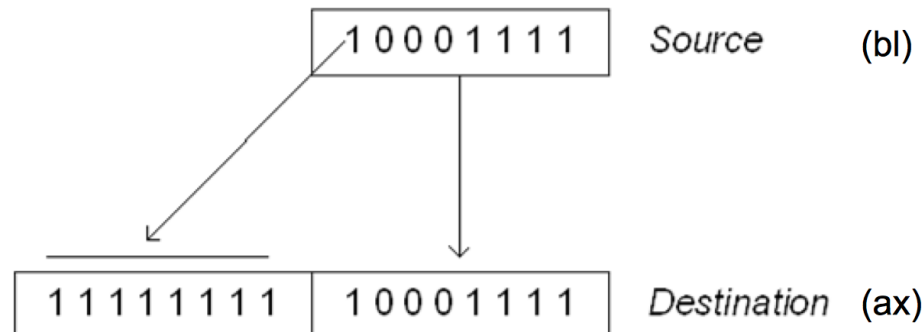
# MOVZX / MOVSX

- From small register to large register
- Zero-extend (MOVZX) / sign-extend (MOVSX)
- Example: `movzx ebx, al`

When copy a smaller value into a larger destination, MOVZX instruction fills (extends) the upper half of the destination with zeros



MOVSX fills the upper half of the destination with a copy of the source operand's sign bit



# More About Memory Access

- `mov ebx, [esp + eax * 4]` **Intel**
- `mov (%esp, %eax, 4), %ebx` **AT&T**
- `mov BYTE [eax], 0x0f`

You must indicate the data size: BYTE/WORD/DWORD

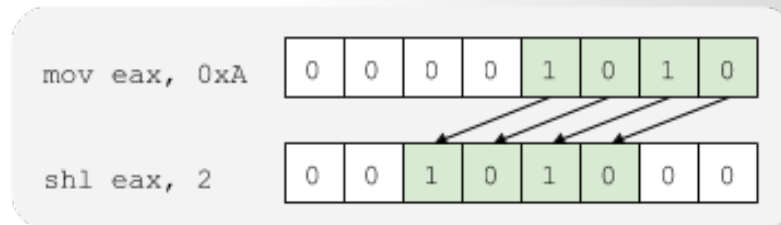
# ADD / SUB

- ADD / SUB
- Normally "reg += reg" or "reg += imm"
- Data size should be equal
  - ADD eax, ebx
  - sub eax, 123
  - sub eax, BL ; Illegal

- **inc, dec** — Increment, Decrement
- The **inc** instruction increments the contents of its operand by one.  
The **dec** instruction decrements the contents of its operand by one.
- *Syntax*  
inc <reg>  
inc <mem>  
dec <reg>  
dec <mem>
- *Examples*  
DEC EAX — subtract one from the contents of EAX.  
INC DWORD PTR [var] — add one to the 32-bit integer stored at location *var*

# SHL / SHR / SAR

- Shift logical left / right
- Shift arithmetic right
- Common usage: **SHL eax, 2** (when calculate memory address)



# Jump

- Unconditional jump: jmp
- Conditional jump: je/jne and ja/jae/jb/jbe/jg/jge/jl/jle ...
- Sometime with "cmp A, B" -- compare these two values and set eflags
- Conditional jump is decided by some of the eflags bits.

## The JMP Instruction

- JMP (jump) instruction causes an unconditional jump
- Syntax is: **JMP destination/target\_label**
- JMP can be used to get around the range restriction [126/127 byte]
- Flags – no change

```
TOP:
; body of the loop, say 2 instructions
DEC  CX      ; decrement counter
JNZ  TOP     ; keep looping if CX > 0
MOV  AX, BX
```

```
TOP:
; the loop body contains so many instructions
; that label TOP is out of range for JNZ. Solution is-
      DEC  CX
      JNZ  BOTTOM
      JMP  EXIT
```

```
BOTTOM:
      JMP  TOP
EXIT:
      MOV  AX, BX
```

Section 6-3: Assembly Language Programming

## Unsigned and Signed Jumps.

Condition	Unsigned	Signed
source < dest	JB	JL
source <= dest	JBE	JLE
source != dest	JNE(JNZ)	JNE(JNZ)
source = dest	JE(JZ)	JE(JZ)
source >= dest	JAE	JGE
source > dest	JA	JG

# Jump

- ja/jae/jb/jbe are unsigned comparison
- jg/jge/jl/jle are signed comparison

Unsigned and Signed Jumps.

Condition	Unsigned	Signed
$\text{source} < \text{dest}$	JB	JL
$\text{source} \leq \text{dest}$	JBE	JLE
$\text{source} \neq \text{dest}$	JNE(JNZ)	JNE(JNZ)
$\text{source} = \text{dest}$	JE(JZ)	JE(JZ)
$\text{source} \geq \text{dest}$	JAЕ	JGE
$\text{source} > \text{dest}$	JA	JG



- **cmp** — Compare
- Compare the values of the two specified operands, setting the condition codes in the machine status word appropriately. This instruction is equivalent to the sub instruction, except the result of the subtraction is discarded instead of replacing the first operand. *Syntax*  
cmp <reg>,<reg>  
cmp <reg>,<mem>  
cmp <mem>,<reg>  
cmp <reg>,<con>
- *Example*  
cmp DWORD PTR [var], 10  
jeq loop
- If the 4 bytes stored at location *var* are equal to the 4-byte integer constant 10, jump to the location labeled *loop*.

Q & A

