Class

2

CSC 472 Software Security

IA-32 Registers & Byte Ordering

Dr. Si Chen (schen@wcupa.edu)

Tue / Thu · 2:00 – 3:15 PM · 25 University Ave., Room 158
Fall 2026 · Week 1 · Thursday, August 27, 2026
wcu.ninja/csc472

Recap from Tuesday

- Security = preventing **undesired behavior** against an **adversary**, across **C**onfidentiality/**I**ntegrity/**A**vailability.
- A **vulnerability** is a security-relevant bug; an **exploit** weaponizes it (Heartbleed, the `strcpy` overflow).
- The taste: overflow a 64-byte `buffer` → **overwrite the saved return address** → control execution.
- To *do* that, you must know **where the CPU keeps that address** and **how the stack works** — today's topic.

West Chester University

PART 1

Why we drop to the machine level

Exploitation is applied computer architecture

West Chester University

Two views of the same machine

- Exploits manipulate the CPU's **actual state** — registers, stack, memory — not the source-code abstraction.
- C and assembly are **two views of the same machine**; the **debugger shows the truth**.
- We use **IA-32 (32-bit x86)** first — smaller and clearer than 64-bit, and the classic teaching target.
- Intel uses "IA-32" for the Pentium family, to distinguish it from their 64-bit architectures.
  - Everything today reappears in **stack frames, shellcode, and ROP** over the coming weeks.

West Chester University

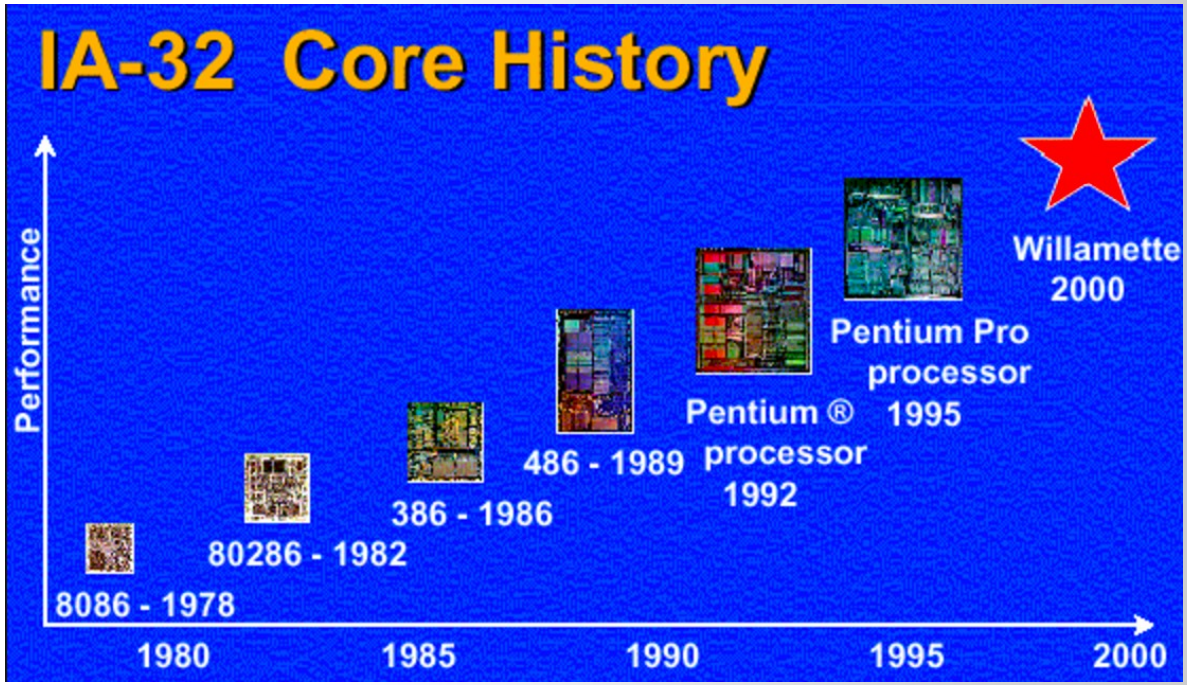
PART 2

The IA-32 register set

Three families of registers — and the three that decide exploitation

Intel IA-32 Processor

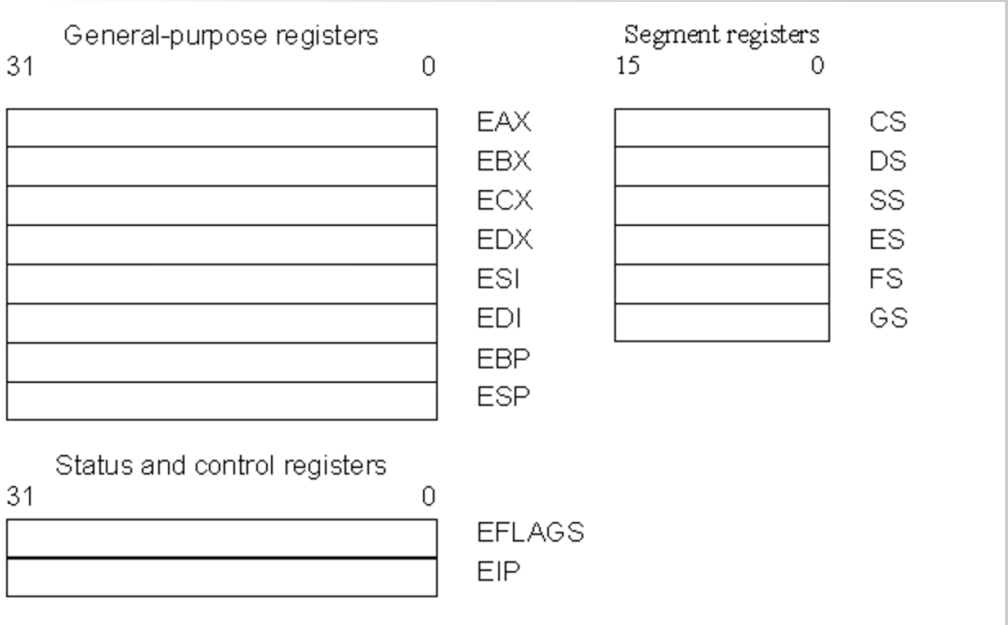
- Intel uses IA-32 to refer to Pentium processor family, in order to distinguish them from their 64-bit architectures.



West Chester University

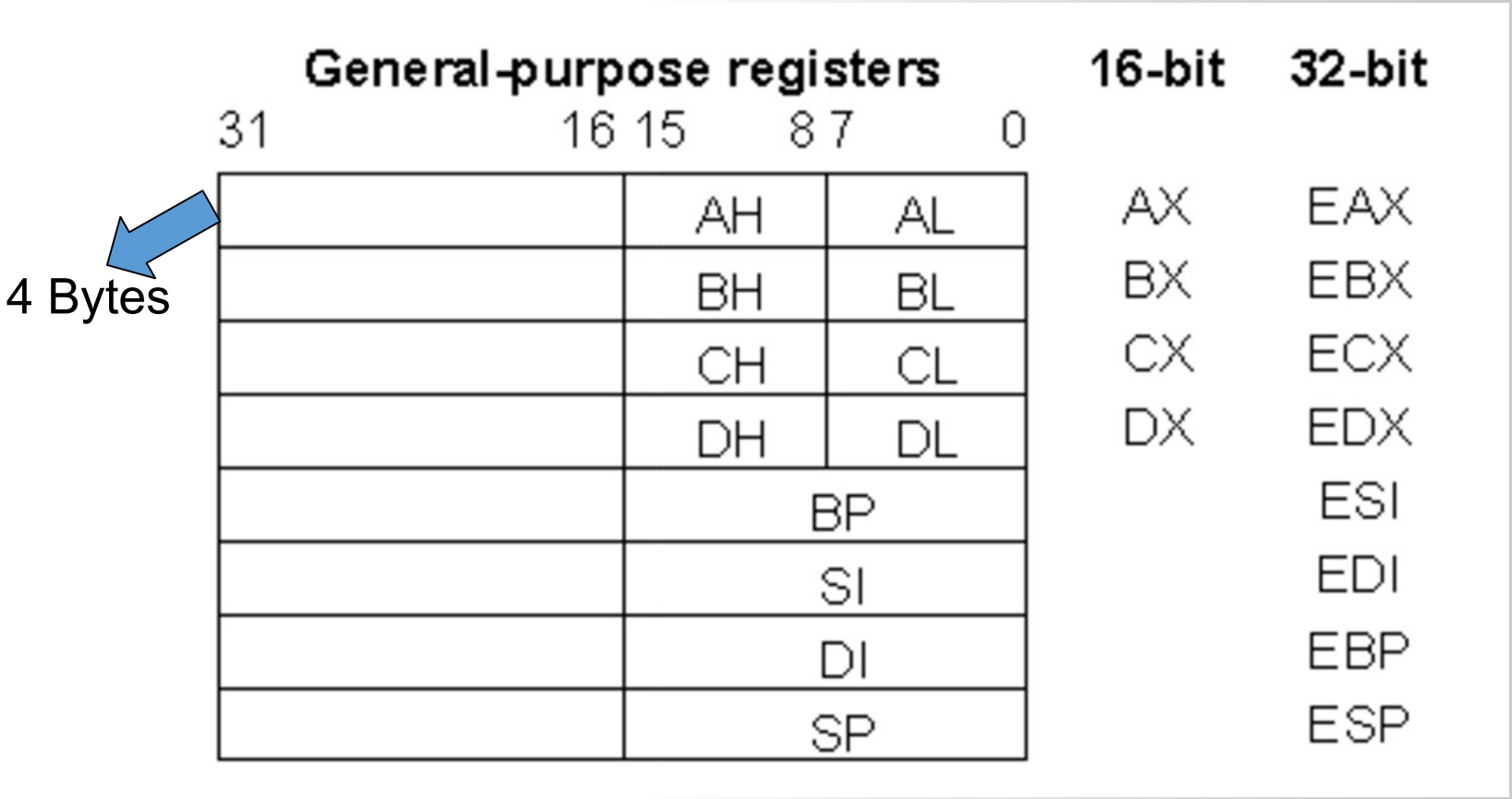
Register Set

- **General-purpose data registers** — eight 32-bit registers holding operands, results, and pointers.
- **Segment registers** — six 16-bit selectors (CS, DS, SS, ES, FS, GS) identifying memory segments.
- **Status & control registers** — `EFLAGS` (status/control flags) and `EIP` (the instruction pointer).
  - We care most about the **general-purpose** ones and **EIP** — the rest matters less for our exploits.



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



Other uses…

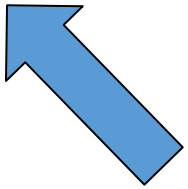
- EAX—Accumulator for operands and results data.
- EBX—Pointer to data in the DS segment.
- ECX—Counter for string and loop operations.
- EDX—I/O pointer.

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)

West Chester University

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.**

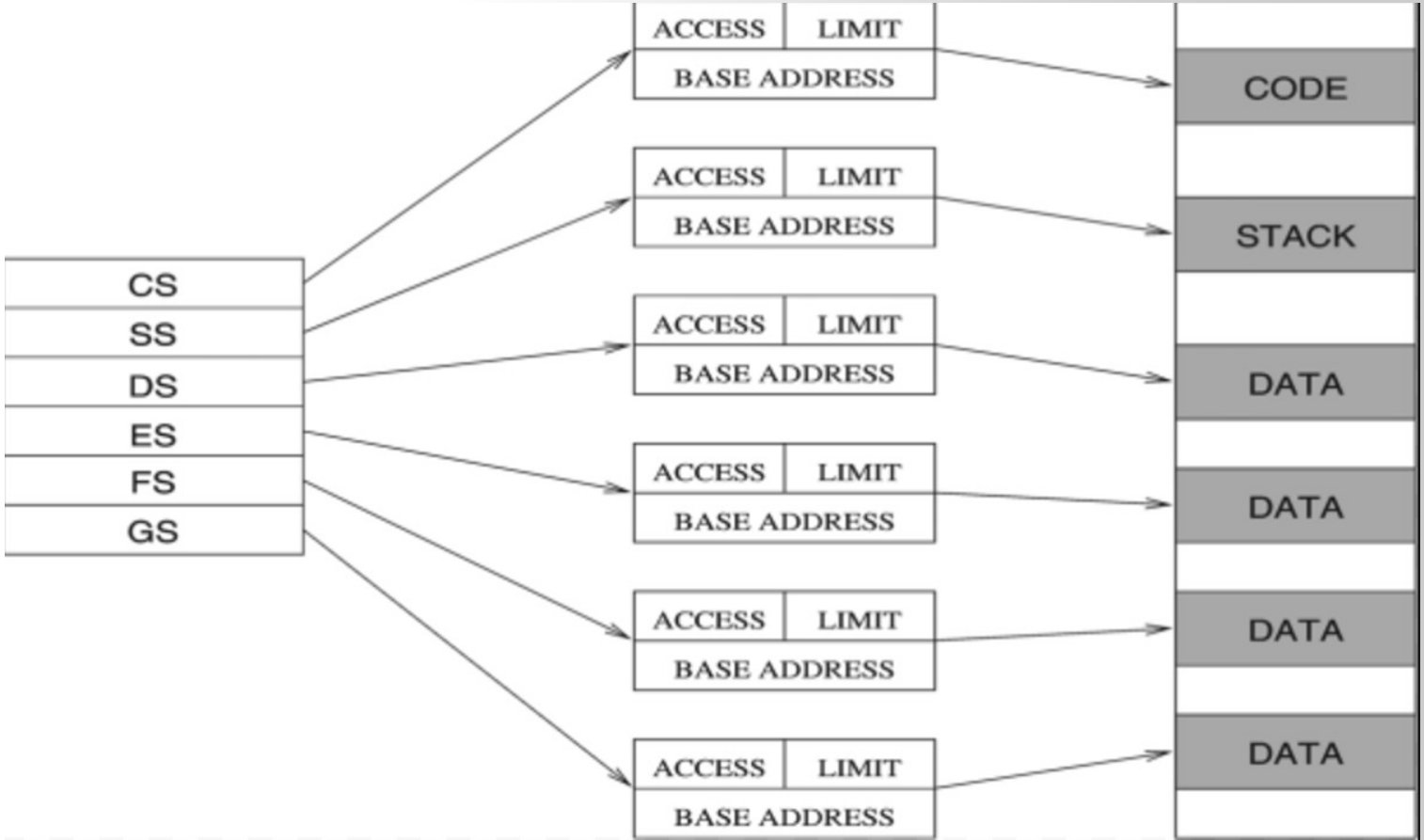


PUSH, POP, CALL, RET

West Chester University

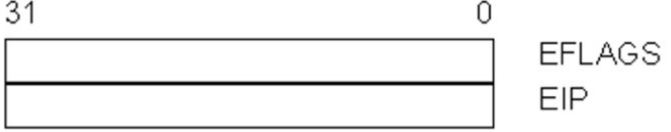
Segment Registers

- 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



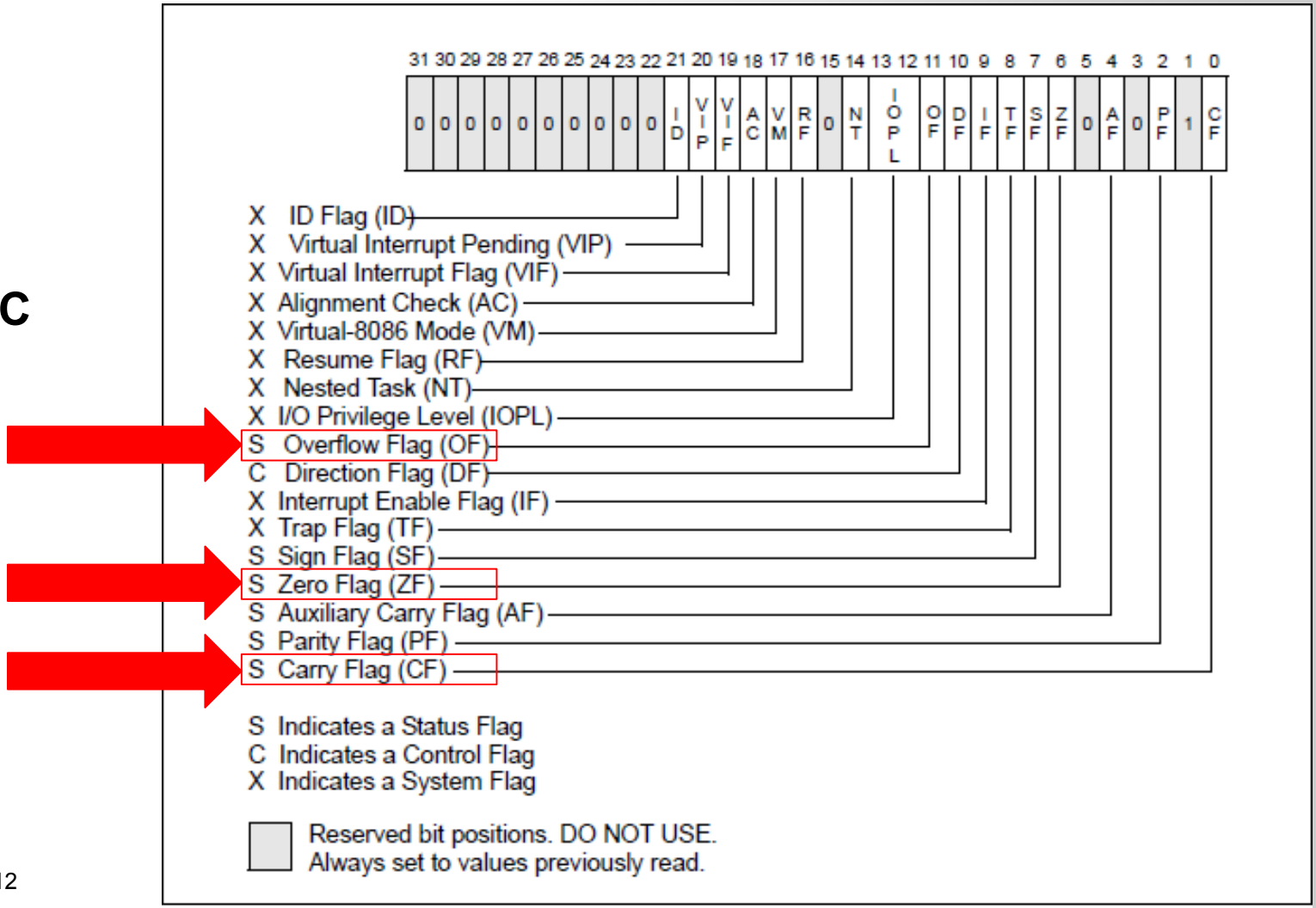
West Chester University

Status and Control Registers



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

JCC



EFLAGS Register

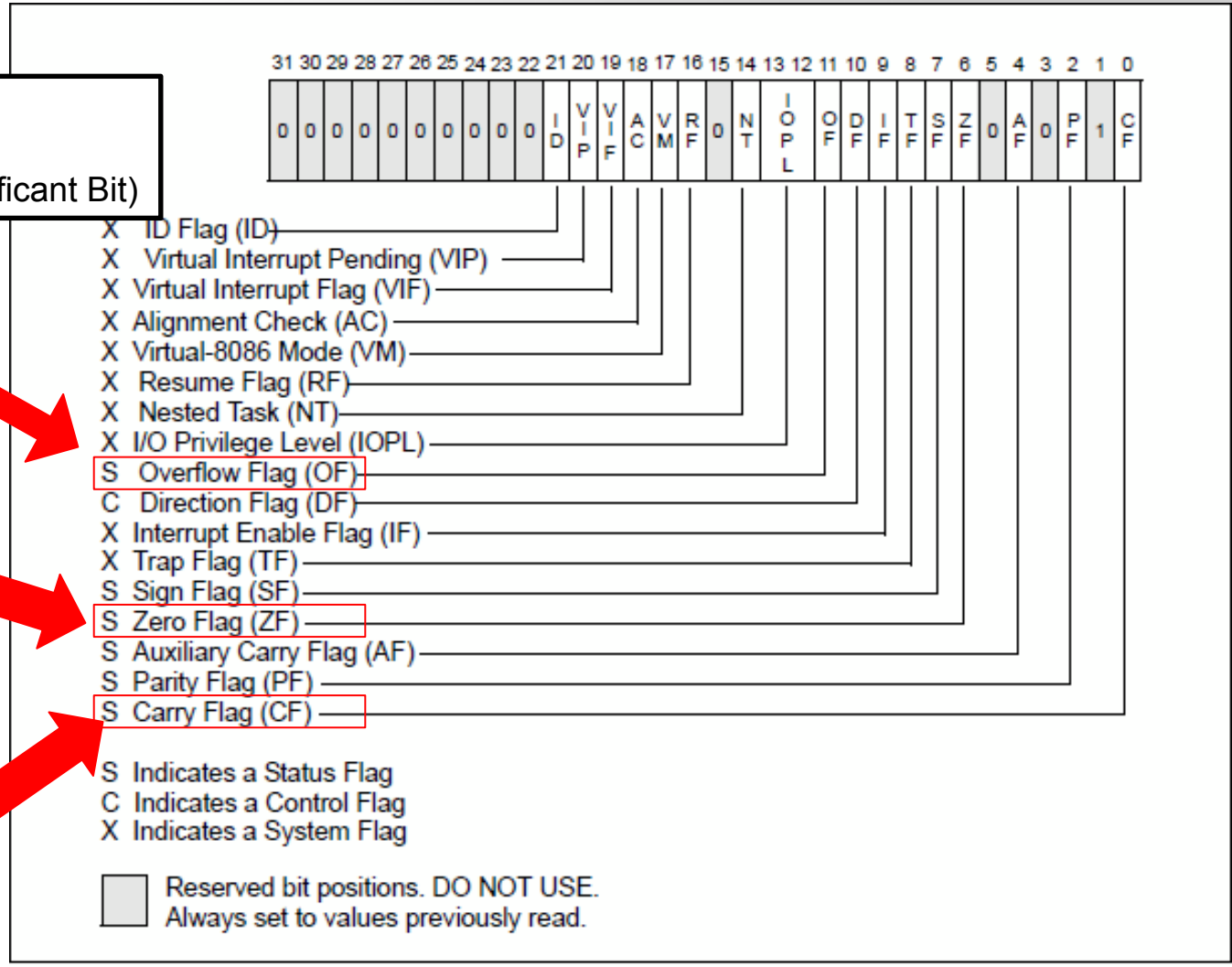
West Chester University

Status and Control Registers

Change to ‘1’ if:
- Signed integer overflow
- Change in MSB (Most Significant Bit)

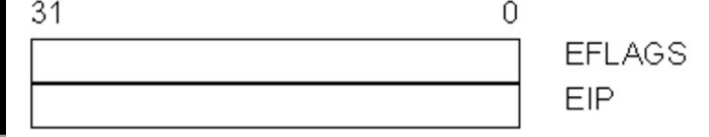
Change to ‘1’ if:
- Calculation result is 0

Change to ‘1’ if:
- unsigned integer overflow



EFLAGS Register

Status and Control Registers



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.

West Chester University

The eight general-purpose registers (32-bit)

```
EAX  Accumulator  – arithmetic; holds a
function's RETURN value
EBX  Base         – general storage / data
pointer
ECX  Counter      – loop & string counts
(decremented each loop)
EDX  Data         – I/O, multiply/divide
ESI  Source Index – source pointer for
string operations
EDI  Dest. Index  – destination pointer
for string operations
EBP  Base Pointer – anchors the current
STACK FRAME   (KEY)
ESP  Stack Pointer– points to the TOP of
the stack    (KEY)
```

- Each has a **conventional role**, but most can be used generally.
- `EAX` holds **return values** — worth watching after every `call`.
- `ECX` is the **loop counter**; `ESI`/`EDI` drive **string/memory copies** (like `strcpy`!).
- `EBP` and `ESP` frame and top the **stack** — the battleground of this course.

Legacy nesting: EAX (32) → AX (16) → AH / AL (8).

West Chester University

The three that decide exploitation

ESP / EBP

- **ESP — Stack Pointer**
- Always points to the **top** of the stack.
- Moved by `push`/`pop`/`call`/`ret`.

- **EBP — Base Pointer**
- Anchors the **current function's frame**.
- Local variables are addressed relative to it.

EIP — the target

- **EIP — Instruction Pointer**
- Holds the **address of the next instruction**.
- You cannot write to it directly...
- ...but `ret` pops the stack **into EIP**.

- **Overwrite the saved return address on the stack**
- → control **EIP** → control execution. That is the whole game.

West Chester University

How the stack instructions move ESP

```
push eax      ; ESP -= 4;  store EAX at
[ESP]
pop  ebx      ; load [ESP] into EBX;  ESP
+= 4

call func     ; push return address (next
EIP); jump to func
ret           ; pop return address from
stack INTO EIP

; the stack grows DOWNWARD (toward lower
addresses)
; EFLAGS records results: Zero (ZF), Carry
(CF), Sign (SF)...
```

- `push`/`pop` move `ESP` by 4 bytes (a 32-bit word).
- `call` **saves the return address** on the stack; `ret` **loads it back into EIP**.
- The stack grows **downward** — toward lower addresses. (Surprising, and important.)
- `EFLAGS` bits (zero, carry, sign) drive conditional jumps — the basis of `if`/loops in asm.

call + ret are why a saved return address exists on the stack for us to overwrite.

West Chester University

PART 3

Byte ordering

x86 stores multi-byte values "backwards" — and it will bite your first payload

West Chester University

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.

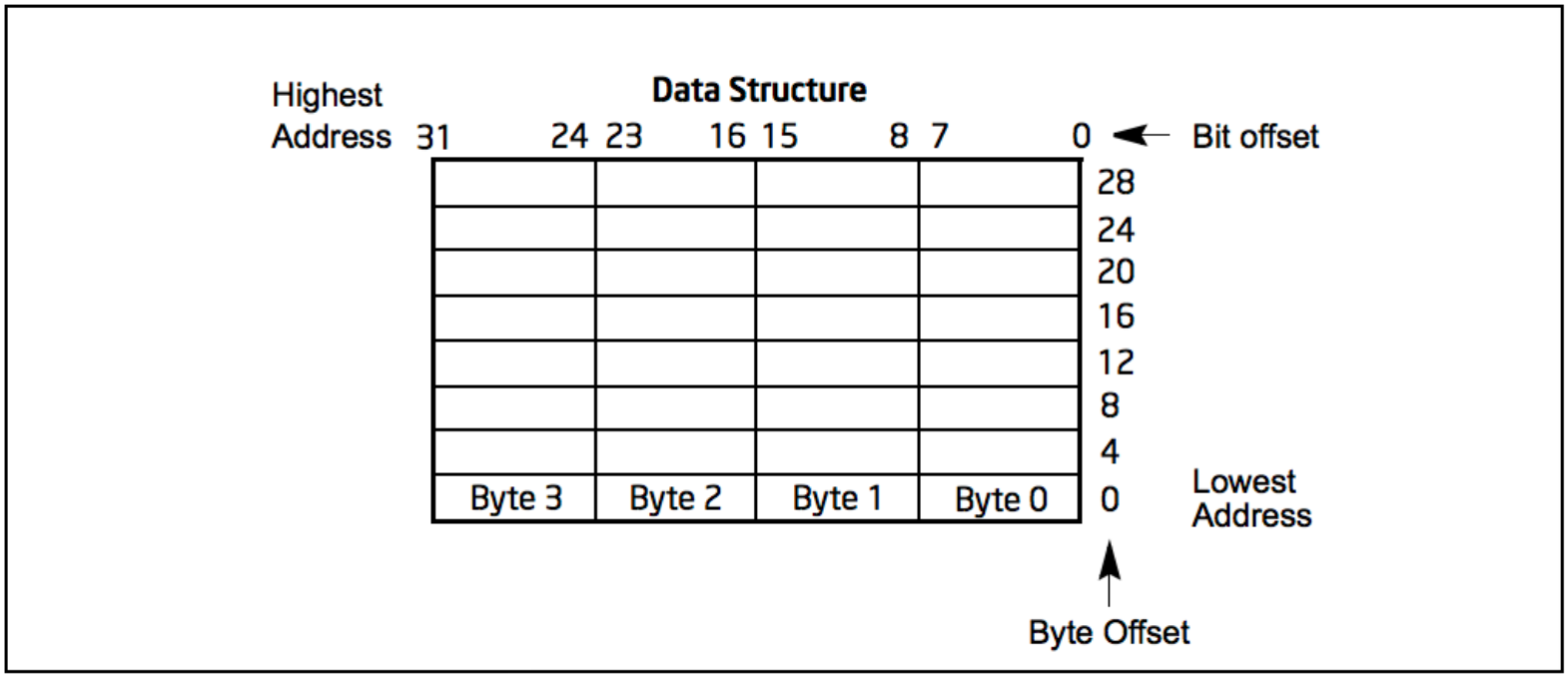
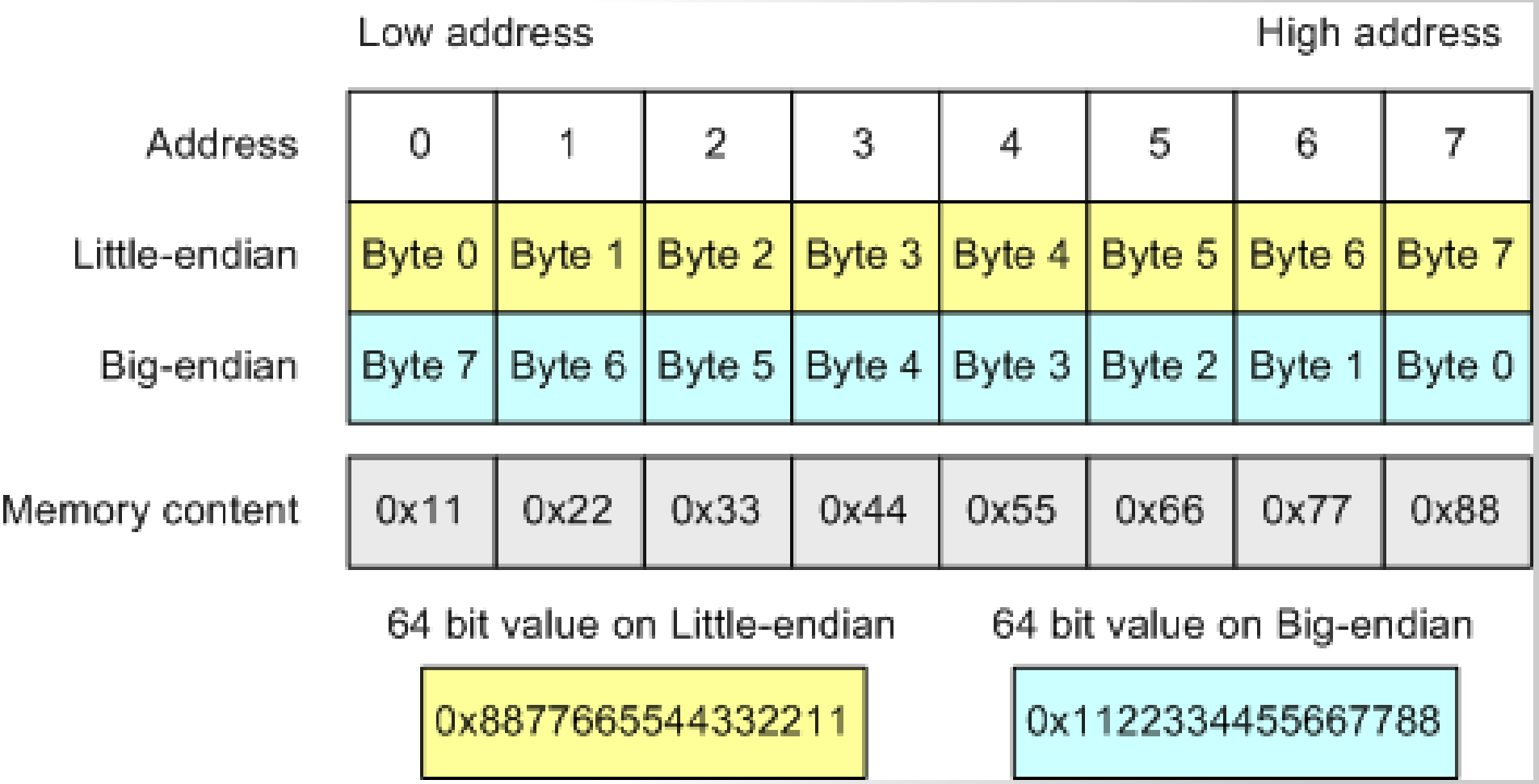


Figure 1-1. Bit and Byte Order

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
0x8877665544332211
64 bit value on Big-endian
0x1122334455667788

little_endian.c

```c
#include "stdio.h"

char c = 0x12;
short sNum = 0x1234;
int num = 0x12345678;
char str[] = "abcde";


int main(int argc, char *argv[])
{
        char lc = c;
        short lsNum = sNum;
        int lnum = num;
        char *lstr = str;

        return 0;
}
```



```
gcc little_endian.c -o little_endian -m32 -no-pie
```

West Chester University

Little-endian: least-significant byte first

```
The 4-byte value  0x12345678  in memory on
x86:

  address:   +0    +1    +2    +3
  byte:      0x78  0x56  0x34  0x12
<-- little-endian

(least-significant byte at the LOWEST
address)

Big-endian (network byte order) would
store:
  byte:      0x12  0x34  0x56  0x78
```

- x86/IA-32 is **little-endian**: the **least-significant byte sits at the lowest address**.
- A value read into a register looks normal; **in memory the bytes are reversed**.
- Big-endian (the opposite) is used by many network protocols — hence "network byte order."
- This is a property of the **architecture**, not your program.

West Chester University

little_endian.c — and why exploits care

```
unsigned int x = 0x12345678;
unsigned char *p = (unsigned char *) &x;
printf("%02x %02x %02x %02x\n",
        p[0], p[1], p[2], p[3]);
// prints:  78 56 34 12   (little-endian!)

# Placing an address in a payload — write
it LSB-first:
#   address 0x08048abc  ->  bytes  bc 8a
04 08
# pwntools does this for you:
p32(0x08048abc)
```

- Read the raw bytes of a value and you **see the reversal** directly.
- When you put a target **address** into an exploit payload, you must write it **LSB-first**.
- `0x08048abc` becomes the bytes `bc 8a 04 08` in your payload.
- Getting this wrong = the CPU jumps to **garbage**. `pwntools p32()` does the flip — know what it does.

Endianness is the #1 reason a "correct-looking" first payload fails. Verify bytes in gdb.

West Chester University

PART 4

See it yourself: the tools

You will live in a debugger — never guess, verify

The debugger is your microscope

gdb (we use this)

- **gdb** (GNU Debugger) — our primary tool. See registers, memory, and the stack **as code runs**, or inspect a crash.
- You will use it to find exact **offsets** and **addresses** — not guess them.
- Static analysis: read code **without running** it.
- Dynamic analysis: watch it **execute** in real time.

The modern toolbox

- Other tools you may meet:
- **Ghidra** — NSA's free reverse-engineering suite (open-sourced 2019; now everywhere).
- **x64dbg** — modern, free, actively-maintained Windows debugger.
- **IDA Pro** — industry-standard disassembler/decompiler.
- **pwntools** — the Python exploit-dev library we script with.
- All answer one question: **what is the machine actually doing?**

Warm-up: registers & endianness

IN-CLASS ⏱ 6–8 min

- No computer needed — reason it out, discuss with a neighbor.
- **1.** Which register would an attacker ultimately want to control to hijack execution, and why?
- **2.** The 4-byte value `0xDEADBEEF` is stored at address `0x1000` on x86. What byte is at `0x1000`? At `0x1003`?
- **3.** You need the address `0x080491e2` in a payload. Write the four bytes **in memory order**.

Answers to check after:

1. EIP — it holds the address of
   the next instruction to execute.
2. LSB-first: 0x1000 = 0xEF,
   0x1003 = 0xDE.
3. e2 91 04 08 (little-endian).

In lab, gdb shows you the real bytes — never guess, verify.

Before next class (Thu, Sep 3)

- Confirm you can **SSH into the course Linux environment** and launch **`gdb`** on a small binary.
- Review your **CSC 231/242** x86 assembly notes and your **CSC 302** systems background.
- Practice: write `0xCAFEBABE` and the address `0x08048000` as **little-endian byte sequences**.
  - Course site: **wcu.ninja/csc472**. Next (**Thu, Sep 3**): x86 assembly + the gdb workflow; **Quiz 1** follows.