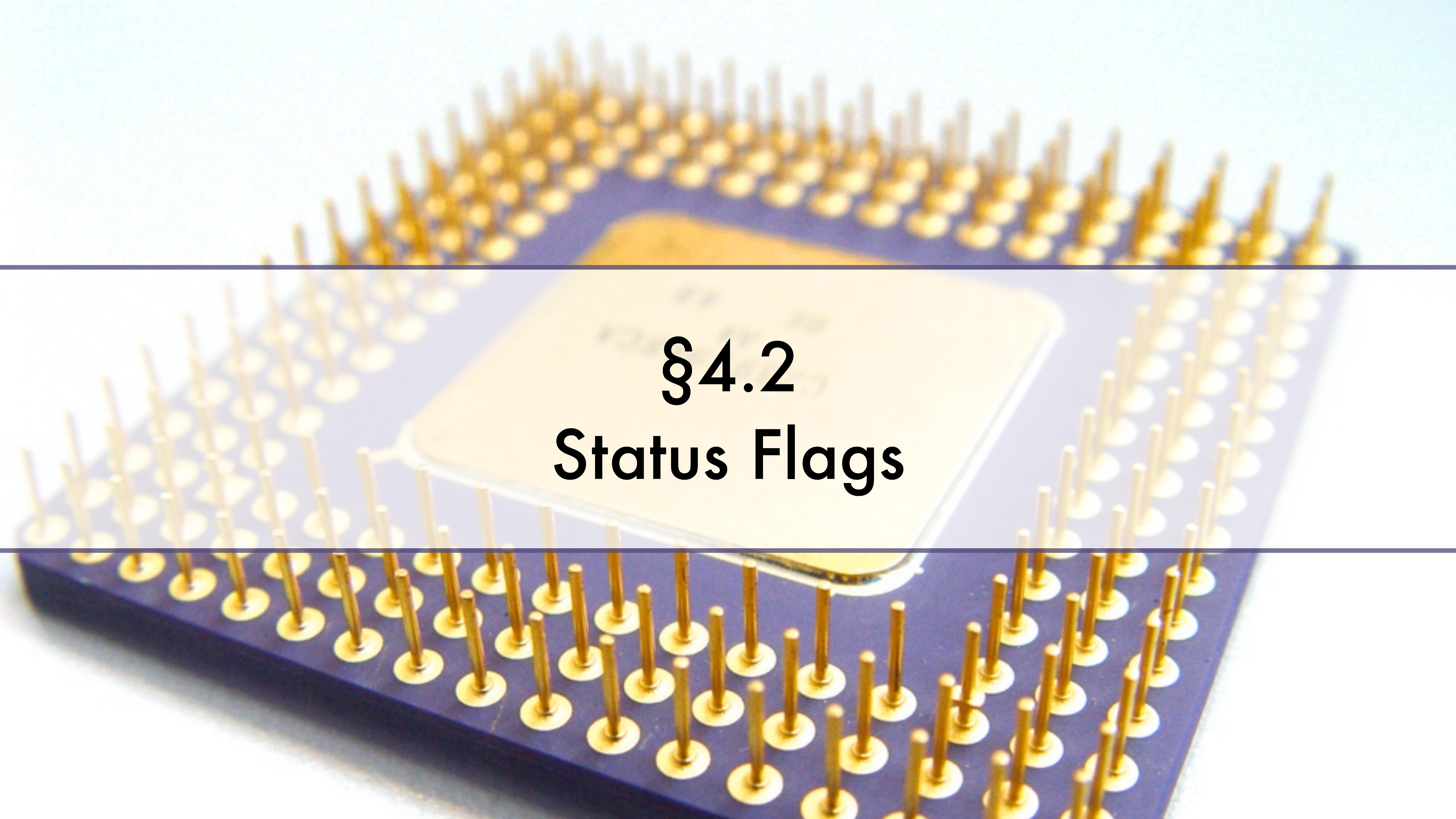
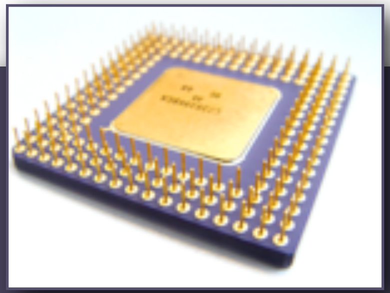


§4.2
Status Flags

Homework



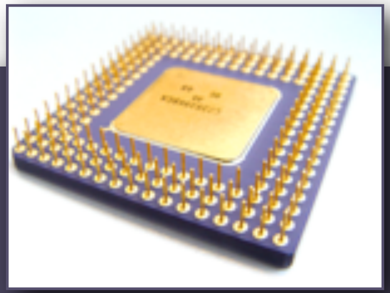
- **Quiz 1** on 9/22 – one week from today
- **Exam 1** on 9/26 – the Friday after the quiz
- Makeup exams must be scheduled in advance and will **not** be given after the exam is given in class.
- **Homework 2** is due this Friday, Sept 19, 11 a.m. (in Canvas)
- For Wednesday: Read **Section 4.2** (covered today) and be prepared to verbally answer:
  - 1. Do the following sequences of instructions leave the same value in AX? in EFLAGS?

    ```
    mov ax, 0FFFFh          mov ax, 0FFFFh
    add ax, 1               inc ax
    ```

  - 2. Does the following sequence of instructions set or clear the parity flag? Why?

    ```
    mov ax, 0804h
    add ax, 1
    ```

Assembly Language Is Untyped

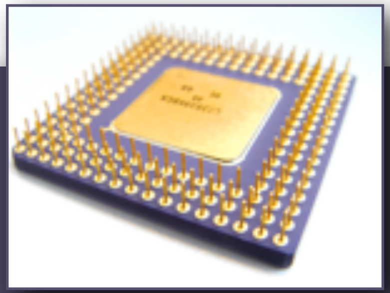


- Binary 10010011 can represent either 147 or –109 (8-bit two's complement)

- **Q.** When the following instructions are translated into machine language, which of their encodings are the same? different?
  - `mov al, 147`
  - `mov al, -109`
  - `mov al, 10010011b`
- **A.** They are all the same! B0h 93h

- **Q.** If AL contains 10010011b, how can you tell if that represents 147 or –109?
- **A.** You can't. It's up to you to remember whether the bits in AL represent an unsigned integer, a signed integer, an ASCII code, etc.

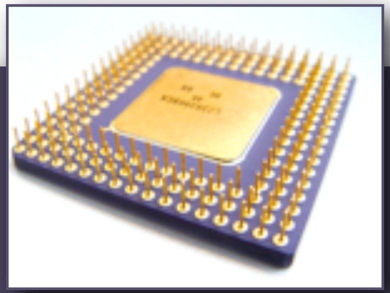
Assembly Language Is Untyped



- So all three of these instructions store the *same* 8 bits in the AL register
  - `mov al, 147`
  - `mov al, -109`
  - `mov al, 10010011b`
- Similarly, all three of these instructions store the *same* 32 bits in EAX
  - `mov eax, 0FFFFFFFFh`
  - `mov eax, 4294967295`
  - `mov eax, -1`
- **Q.** What does this display?

```
mov eax, -1
call WriteDec
```

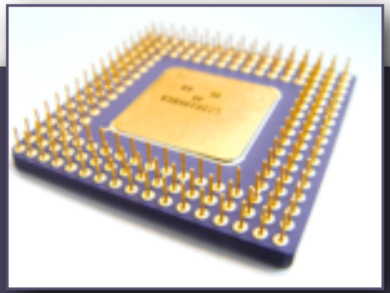
Reading & Writing: Signed vs. Unsigned



- You've used `ReadDec` and `WriteDec` from Kip Irvine's library
  - Read and write 32-bit *unsigned* integers
- There are also routines called `ReadInt` and `WriteInt`
  - Read and write 32-bit *signed* integers
- And also `WriteHex`
- Each `Write`* routine may display the same 32-bit differently

```
mov eax, 0FFFFFFFFh
call WriteDec    ; Prints 4294967295
call WriteInt    ; Prints –1
call WriteHex    ; Prints FFFFFFFF
```

Reading & Writing – Reference



Irvine's library contains three different procedures for displaying integers that should be aware of:

- **call WriteDec** – Interprets the bits in EAX as an unsigned 32-bit integer and prints that value.
- **call WriteInt** – Interprets the bits in EAX as a signed 32-bit integer and prints that value.
- **call WriteHex** – Prints the hexadecimal representation of the bits in EAX.

```
mov eax, 0FFFFFFFFh
call WriteDec  ; Prints 4294967295
call WriteInt  ; Prints -1
call WriteHex  ; Prints FFFFFFFF
```

Likewise, there are three different procedures for reading values:

- **call ReadDec** – Reads an unsigned 32-bit integer and stores its value in EAX.
- **call ReadInt** – Reads a signed 32-bit integer and stores its value in EAX.
- **call ReadHex** – Reads a 32-bit hexadecimal integer and stores its value in EAX.

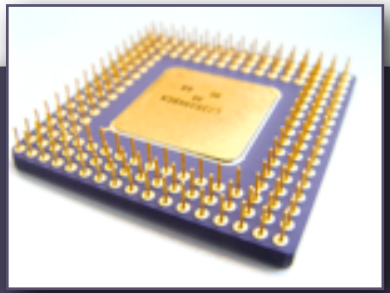
- **call DumpRegs** – Displays register values (in hex) as well as status flags.

Signed/Unsigned Addition/Subtraction

Unsigned Interpretation		Signed Interpretation
105	01101001	105
+ 147	+ 10010011	+ -109
252	11111100	-4

- Binary addition is performed the same way regardless of whether the numbers are unsigned or signed (two's complement)
- The only difference is how you interpret the result
- Subtraction: $A - B$ is computed as $A + (-\mathrm{B})$

Overflow



- **Q.** Recall that an 8-bit register can only hold unsigned integers in the range [0, 255].
  What is the value in AL after this instruction sequence executes?

```
mov al, 255
add al, 1
```

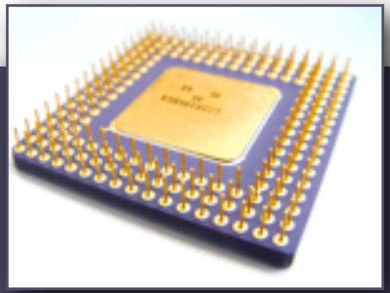
- **A.** AL contains 0, but one of the bits in EFLAGS is set to indicate that *unsigned* overflow occurred.

- **Q.** Recall that an 8-bit register can only hold signed integers in the range [–128, 127].
  What is the value in AL after this instruction sequence executes?

```
mov al, 127
add al, 1
```

- **A.** AL contains 10000000b (+128 or –128, depending on whether you interpret it as signed or unsigned), but a different bit in EFLAGS is set to indicate that *signed* overflow occurred.

x86 Registers (Review)



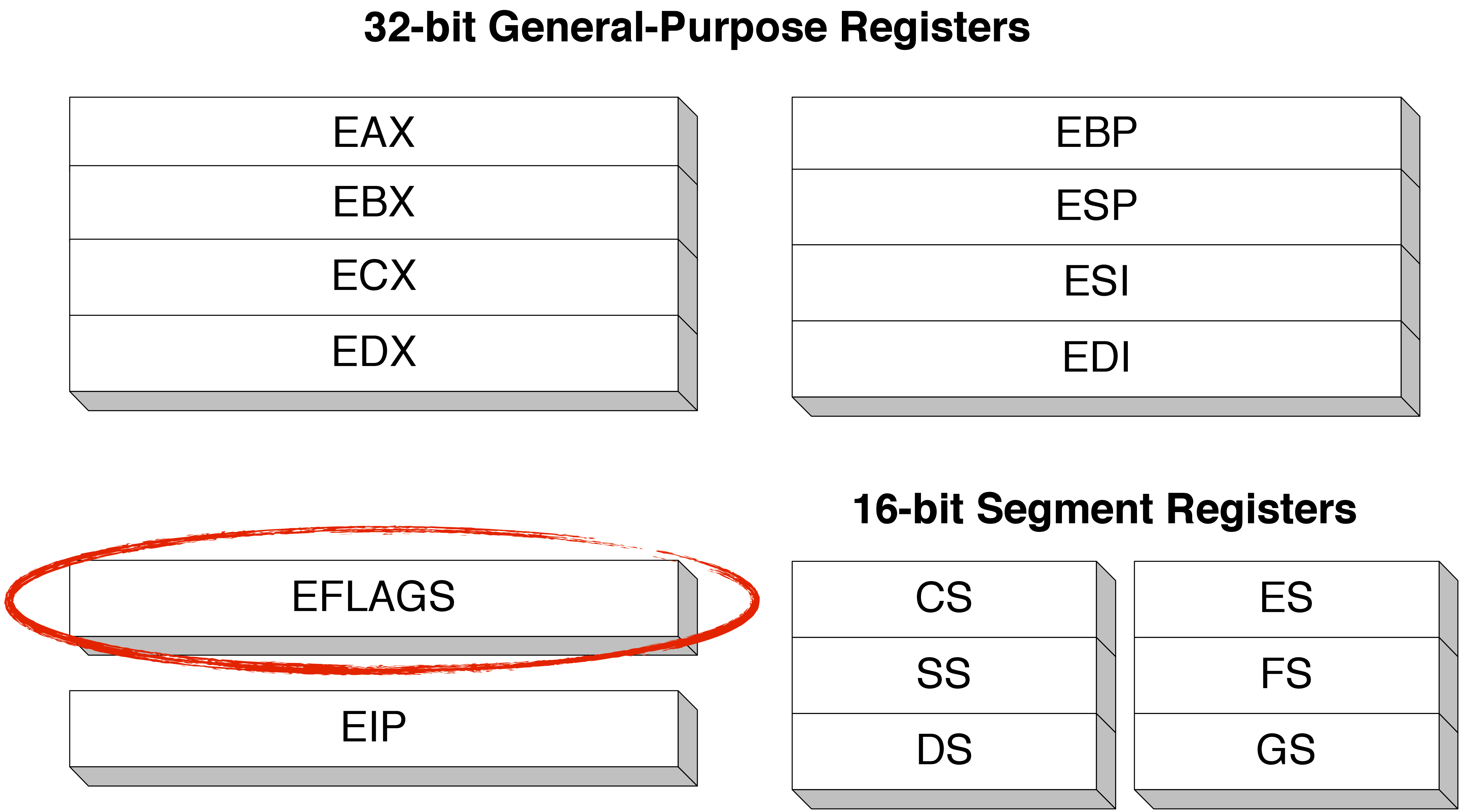
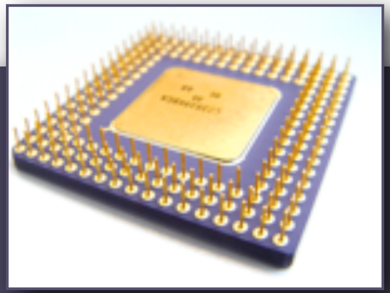


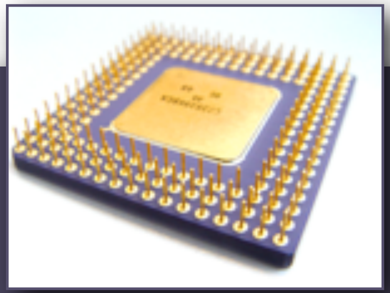
Image Source: *Irvine 6/e*

x86 Registers (Review)



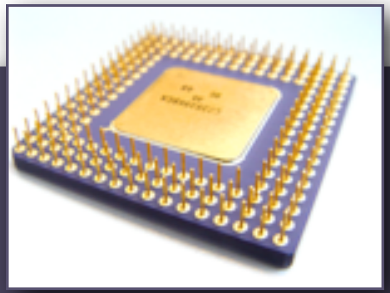
- **EFLAGS – Extended Flags**
- Each bit has a different purpose
- Some bits are *control flags* (e.g., enter protected mode, break after each instruction)
- Other bits are *status flags*
  - Carry flag (CF) – indicates *unsigned* overflow
  - Sign flag (SF)
  - Zero flag (ZF)
  - Overflow flag (OF) – indicates *signed* overflow
  - Parity flag (PF)
  - Auxiliary carry flag (AF)

Whiteboard Notes



- *Whiteboard Notes:* x86 Status Flags and Effects of Addition and Subtraction:
  - Carry
  - Sign
  - Zero
  - Overflow
  - Parity
  - Auxiliary Carry

Arithmetic on 8-bit Values



- Addition/subtraction on *unsigned* byte values:

Difference < 0
CF $= 1$

$0 \leq$ Result ≤ 255
CF $= 0$

Sum > 255
CF $= 1$

0 255

- Addition/subtraction on *signed* byte values:

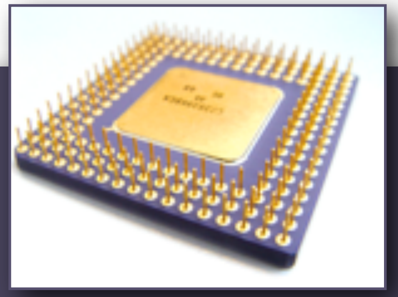
Result < -128
OF $= 1$

$-128 \leq$ Result ≤ 127
OF $= 0$

Result > 255
OF $= 1$

-128 127

Arithmetic on 16-bit Values



- Addition/subtraction on *unsigned* word values:

Difference < 0
CF = 1

0 ≤ Result ≤ 65,535
CF = 0

Sum > 65,535
CF = 1

0 65,535

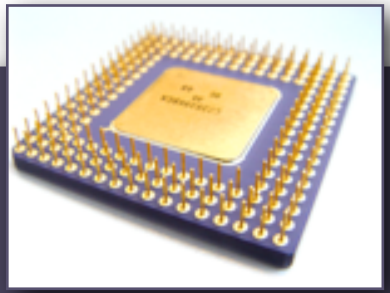
- Addition/subtraction on *signed* word values:

Result < –32,768
OF = 1

–32,768 ≤ Result ≤ 32,767
OF = 0

Result > 32,767
OF = 1

–32,768 32,767



Activity 8