Assembly Language for x86 Processors

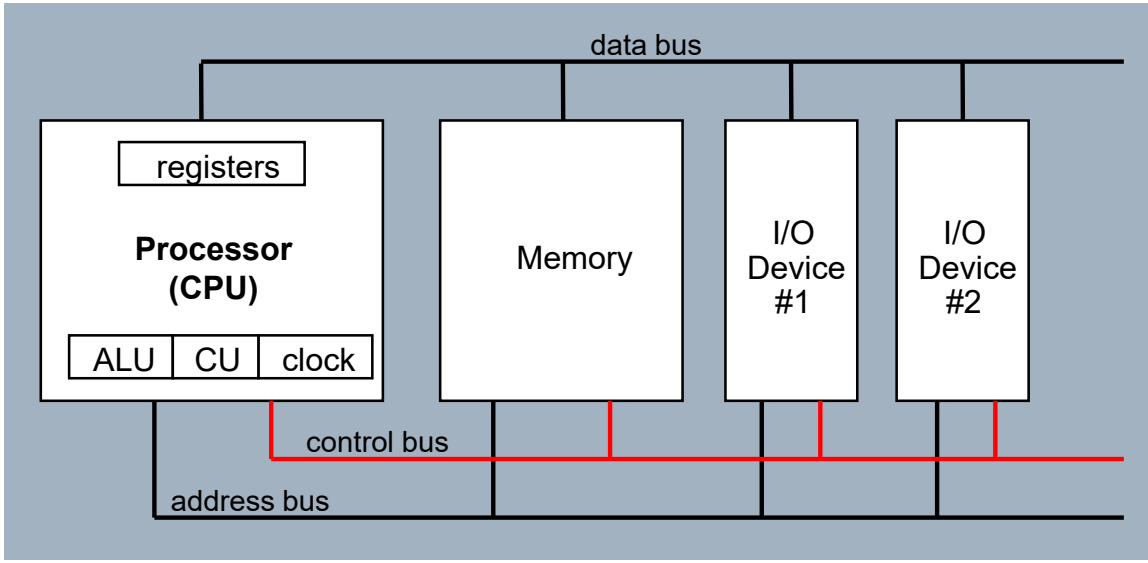
- X86 Processor Architecture

Outline

- Basic IA Computer Organization
- IA-32 Registers
- Instruction Execution Cycle

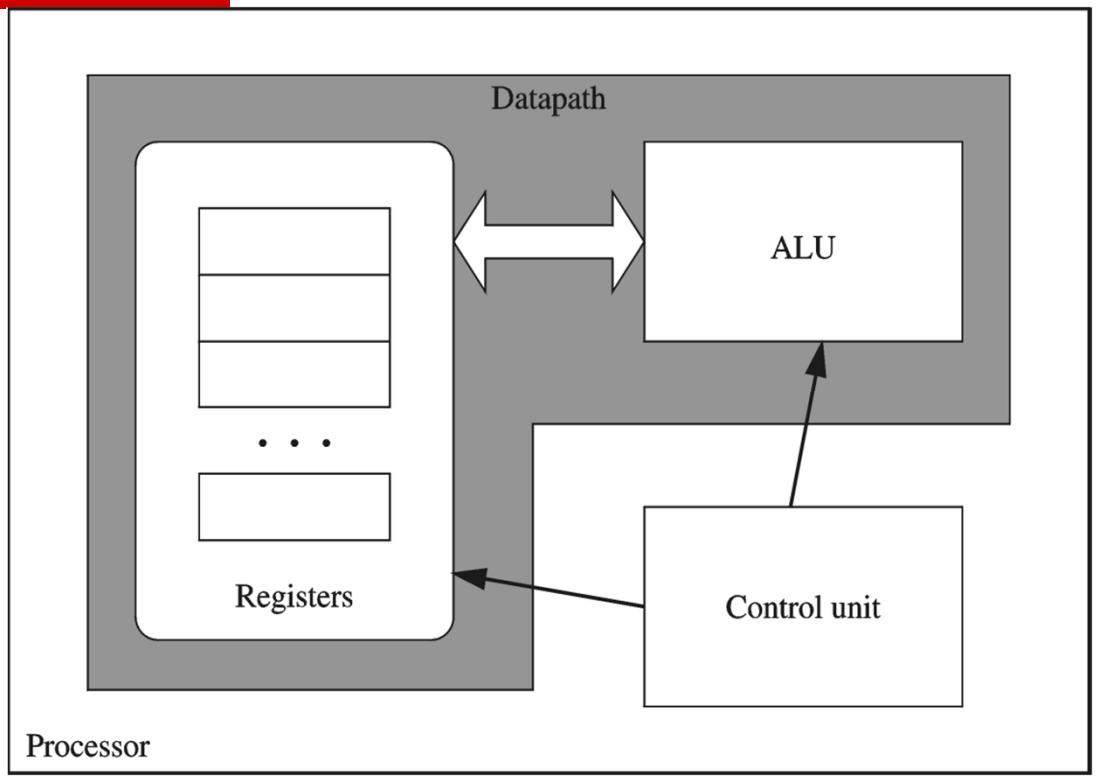
Basic IA Computer Organization

- Since the 1940's, the *Von Neumann* computers contains three key components:
  - Processor, called also the CPU (Central Processing Unit)
  - Memory and Storage Devices
  - I/O Devices
- Interconnected with one or more buses
  - Data Bus
  - Address Bus
  - Control Bus
- IA: Intel Architecture 32-bit (or i386)

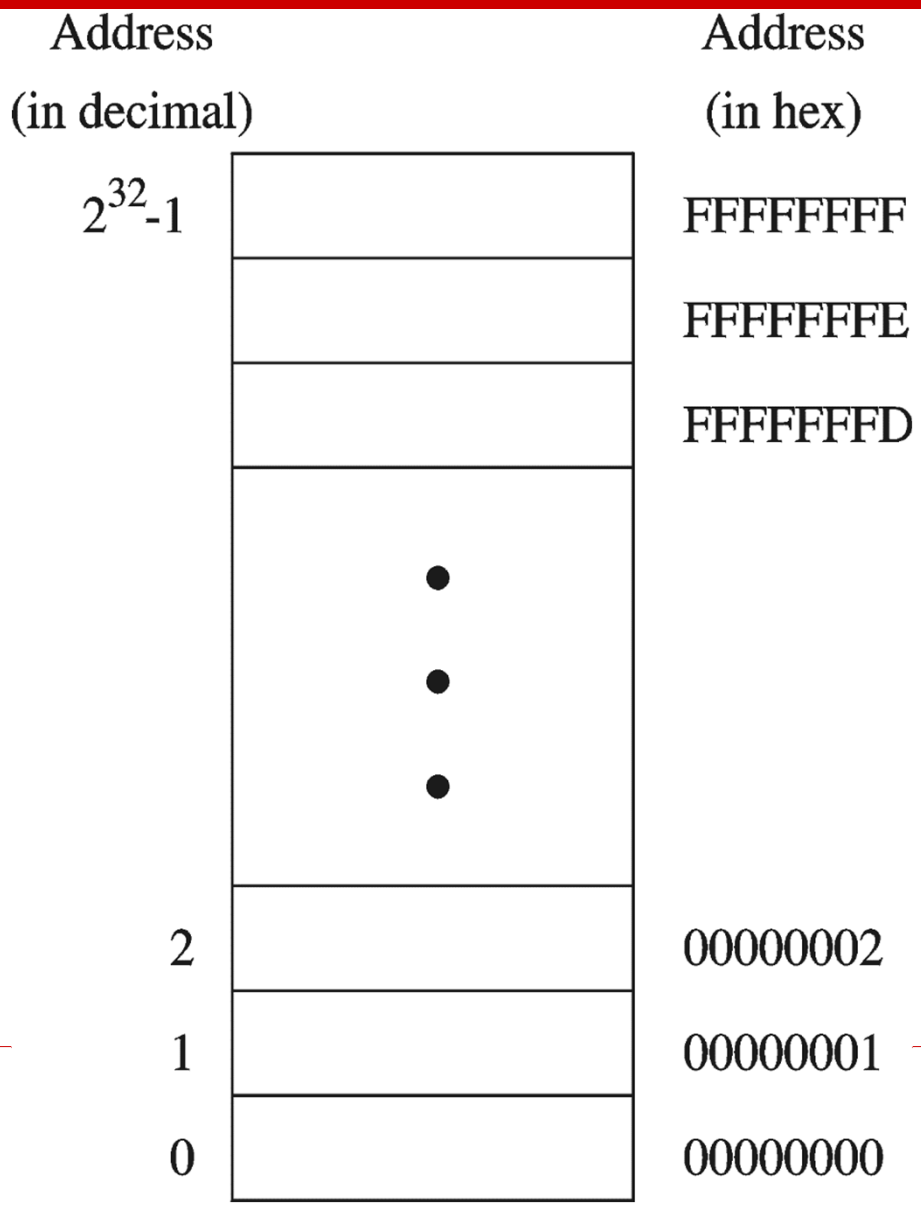


Processor

- The processor consists of
  - Datapath
    - ALU
    - Registers
  - Control unit
- ALU (Arithmetic logic unit)
  - Performs arithmetic and logic operations
- Control unit (CU)
  - Generates the control signals required to execute instructions



Memory Address Space



Address Space is the set of memory locations (bytes) that are addressable

Next ...

- Basic Computer Organization
- IA-32 Registers
- Instruction Execution Cycle

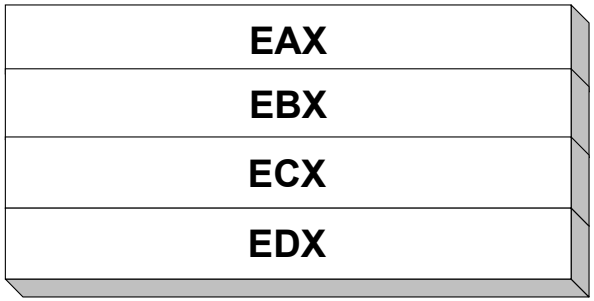
Registers

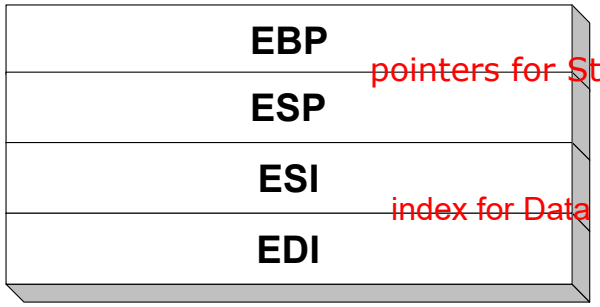
- Registers are high speed memory inside the CPU
  - Eight 32-bit general-purpose registers
  - Six 16-bit segment registers
  - Processor Status Flags (EFLAGS) and Instruction Pointer (EIP)

Extended Instruction Pointer

AX=AH~AL
EAX=AY~AX

32-bit General-Purpose Registers



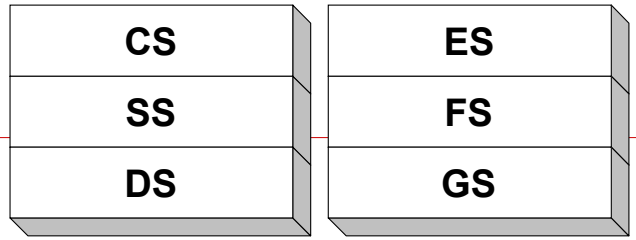


pointers for Stack segment

index for Data Segment

16-bit Segment Registers

like PC

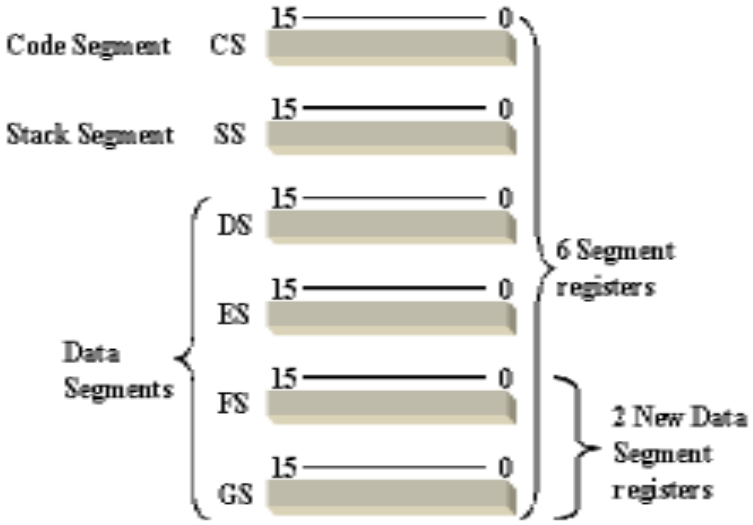


General-Purpose Registers

- Used primarily for arithmetic and data movement
  - `mov eax 10 ;` move constant integer 10 into register eax
- (But have) **specialized uses** (as well) of Registers:
  - **eax** – Accumulator register
    - Automatically used by multiplication and division instructions
  - **ecx** – Counter register
    - Automatically used by LOOP instructions
  - **esp**– Stack Pointer register
    - Used by PUSH and POP instructions, points to top of stack
  - **esi** and **edi** – Source Index and Destination Index register
    - Used by string instructions
  - **ebp** – Base Pointer register
    - Used to reference parameters and local variables on the stack

Special-Purpose & Segment Registers

- EIP = Extended Instruction Pointer ==> PC
  - Contains address of **next** instruction to be **executed**
- EFLAGS = Extended Flags Register
  - Contains status and control flags
  - Each flag is a single binary bit
- Six 16-bit Segment Registers
  - Support segmented memory
  - Segments contain distinct contents
    - Code
    - Data
    - Stack



For 8086

Segment Registers

Buses and operation

All internal registers, as well as internal and external data buses, are 16 bits wide, which firmly established the "16-bit microprocessor" identity of the 8086. A 20-bit external address bus provides a 1 MB physical address space (2^{20} = 1,048,576 x 1 byte). This address space is addressed by means of internal memory "segmentation". The data bus is multiplexed with the address bus in order to fit all of the control lines into a standard 40-pin dual in-line package. It provides a 16-bit I/O address bus, supporting 64 KB of separate I/O space. The maximum linear address space is limited to 64 KB, simply because internal address/index registers are only 16 bits wide. Programming over 64 KB memory boundaries involves adjusting the segment registers (see below); this difficulty existed until the 80386 architecture introduced wider (32-bit) registers (the memory management hardware in the 80286 did not help in this regard, as its registers are still only 16 bits wide).

Segmentation

There are also four 16-bit segment registers (see figure) that allow the 8086 CPU to access one megabyte of memory in an unusual way. Rather than concatenating the segment register with the address register, as in most processors whose address space exceeds their register size, the 8086 shifts the 16-bit segment only four bits left before adding it to the 16-bit offset (16×segment + offset), therefore producing a 20-bit external (or effective or physical) address from the 32-bit segment:offset pair. As a result, each external address can be referred to by 2^{12} = 4096 different segment:offset pairs.

```
        0110 1000 1000 0111 0000      Segment,   16 bits, shifted 4 bits left (or multiplied by 0x10)
+            0011 0100 1010 1001      Offset,    16 bits

        0110 1011 1101 0001 1001      Address,   20 bits
```

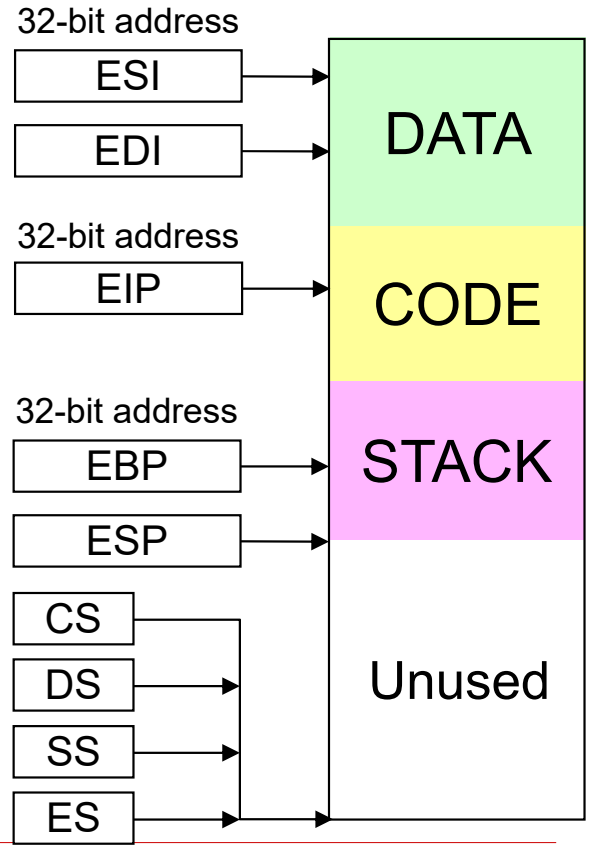
See Also: https://www.geeksforgeeks.org/memory-segmentation-8086-microprocessor/

For 80386

Programmer View of Flat Memory

- Same base address for all segments
  - All segments are mapped to the same linear address space
- EIP Register
  - Points at next instruction
- ESI and EDI Registers
  - Contain data addresses
  - Used also to index arrays
- ESP and EBP Registers
  - ESP points at top of stack
  - EBP is used to address parameters and variables on the stack

Linear address space of a program (up to 4 GB)



base address = 0
for all segments

Background for EFLAGS

Overflow detection

- *X*, *Y* and *Z* are *N*-bit 2's-complement numbers and $Z_{2c}=X_{2c}+Y_{2c}$
- Overflow occurs if $X_{2c}+Y_{2c}$ exceeds the maximum value represented by N-bits.

A **If the signs of *X* and *Y* are different, don't detect overflow for $Z_{2c}=X_{2c}+Y_{2c}$.**

B **In case the signs of *X* and *Y* are the same, if the sign of $X_{2c}+Y_{2c}$ is opposite, overflow detected.**

- Case 1: X, Y positive, Z sign bit ='1'
- Case 2: X, Y negative, Z sign bit ='0'

C If X, Y and Z have same, dont detect overflow.

Background for EFLAGS

Signed integers representation

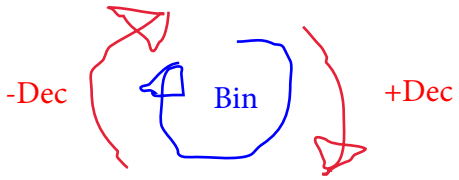
signed-binary

Eight-bit sign–magnitude

Binary value	Sign–magnitude interpretation	Unsigned interpretation
00000000	0	0
00000001	1	1
⋮	⋮	⋮
01111101	125	125
01111110	126	126
01111111	127	127
10000000	−0	128
10000001	−1	129
10000010	−2	130
⋮	⋮	⋮
11111101	−125	253
11111110	−126	254
11111111	−127	255

- Two Zeros
- Range: -127 ~ +127
- FP's Significand uses this!

-ve number: Signed value = 2^(n-1) - Unsigned value
Signed value = 128 - Unsigned value



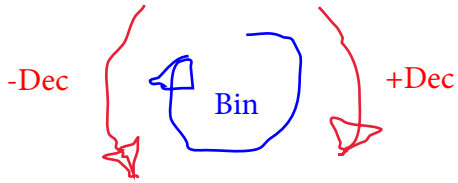
1's complement

Eight-bit ones' complement

Binary value	Ones' complement interpretation	Unsigned interpretation
00000000	0	0
00000001	1	1
⋮	⋮	⋮
01111101	125	125
01111110	126	126
01111111	127	127
10000000	−127	128
10000001	−126	129
10000010	−125	130
⋮	⋮	⋮
11111101	−2	253
11111110	−1	254
11111111	−0	255

- Two Zeros
- Range: -127 ~ +127

Signed value = Unsigned value - 2^n + 1
Signed value = Unsigned value - 255



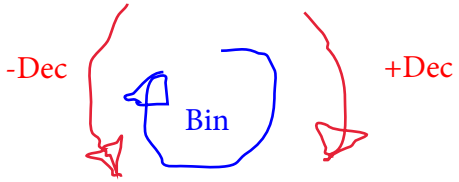
2's complement *

Eight-bit two's complement

Binary value	Two's complement interpretation	Unsigned interpretation
00000000	0	0
00000001	1	1
⋮	⋮	⋮
01111110	126	126
01111111	127	127
10000000	−128	128
10000001	−127	129
10000010	−126	130
⋮	⋮	⋮
11111110	−2	254
11111111	−1	255

- one Zero only
- Range: -128 ~ +127

Signed value = Unsigned value - 2^n
Signed value = Unsigned value - 256



* - **Addition and subtraction require different behavior** depending on the sign bit in Signed Binary, whereas one's complement can ignore the sign bit and just do an end-around carry, and **two's complement can ignore the sign bit and depend on the overflow behavior**.
- **Comparison also require inspecting the sign bit** in Signed Binary and 1's complement, whereas in two's complement, one can simply subtract the two numbers, and check if the outcome is positive or negative.

If the signs of X and Y are different, dont detect overflow for $Z_{2c}=X_{2c}+Y_{2c}$

Example

Case-A

- $X_{2c}=(01111010)_{2c}$, $Y_{2c}=(10001010)_{2c}$, $X_{2c}+Y_{2c}=(00000100)_{2c}$ No overflow!

	0	1	1	1	1	0	1	0	122 +
	1	0	0	0	1	0	1	0	-118 =
1	0	0	0	0	0	1	0	0	+4

Overflow (carry-out) 1 BUT ignore it

Good! No problem!

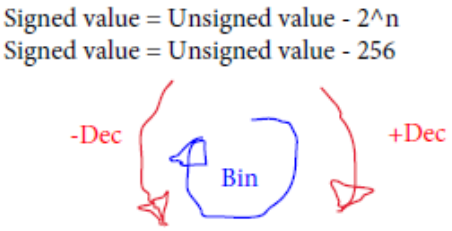
- Case 1: X, Y positive, Z sign bit ='1'
- Case 2: X, Y negative, Z sign bit ='0'

Example

Detect overflow!

Case-B-1

- $X_{2c}=(01111010)_{2c}$, $Y_{2c}=(00001010)_{2c}$, $X_{2c}+Y_{2c}=(10000100)_{2c}$ Overflow detected



	0	1	1	1	1	0	1	0
	0	0	0	0	1	0	1	0
	1	0	0	0	0	1	0	0

122 + 10 = 132

BUT actual answer is -124 ==> problem.

Sign=1 negative

No carry-out, but sign-bit is the O.F. bit and dont ignore it!

Complement it to fix this, once this OF is detected! i.e. 2^8+(-124)=132

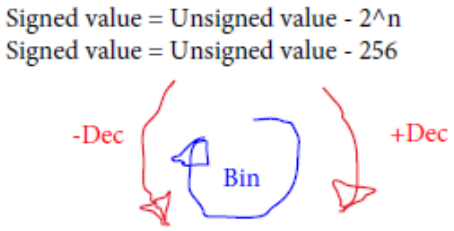
- Case 1: X, Y positive, Z sign bit ='1'
- Case 2: X, Y negative, Z sign bit ='0'

Example

Detect overflow!

Case-B-2

- $X_{2c}=(10011010)_{2c}$, $Y_{2c}=(10001010)_{2c}$, $X_{2c}+Y_{2c}=(00100100)_{2c}$ Overflow detected



	1	0	0	1	1	0	1	0
	1	0	0	0	1	0	1	0
1	0	0	1	0	0	1	0	0

-102 + -118 = -220

Sign =0

O.F. (carry-out) 1 BUT dont ignore it!

BUT actual answer is 36 ==> problem.

Solution:
Once this OF is detected, Do this:
36-2^8 = -220

- Case 1: X, Y positive, Z sign bit ='1'
- Case 2: X, Y negative, Z sign bit ='0'

Then, detect overflow!

Example

Case-C-1

$X_{2c}=(00111010)_{2c}$, $Y_{2c}=(00001010)_{2c}$,

$X_{2c}+ Y_{2c}=(01000100)_{2c}$ No overflow

Good!
No problem!

	0	0	1	1	1	0	1	0
+	0	0	0	0	1	0	1	0
=	0	1	0	0	0	1	0	0

58 + 10 = 68

Sign=0 positive

Correct math and actual answer, so no need to make correction and hence, No importance of OF even if there were to be one!

- Case 1: X, Y positive, Z sign bit ='1'
- Case 2: X, Y negative, Z sign bit ='0'

Then, detect overflow!

Example

Case-C-2

$X_{2c}=(11111010)_{2c}$,
$Y_{2c}=(10001010)_{2c}$,
$X_{2c}+Y_{2c}=(10000100)_{2c}$ No overflow

Good!
No problem!

	1	1	1	1	1	0	1	0
	1	0	0	0	1	0	1	0
1	1	0	0	0	0	1	0	0

-6
+
-118
=
-124

Sign = 1

Overflow (carry-out) 1 BUT ignore it

Summary: CF, OF (US, Signed int)

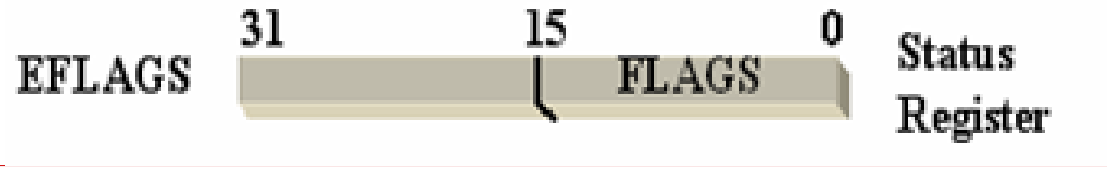
For Z=X+Y

Analyzing the combination of MSBs of the operands and related results' combinations possible, and the analysis of the validity of the result:

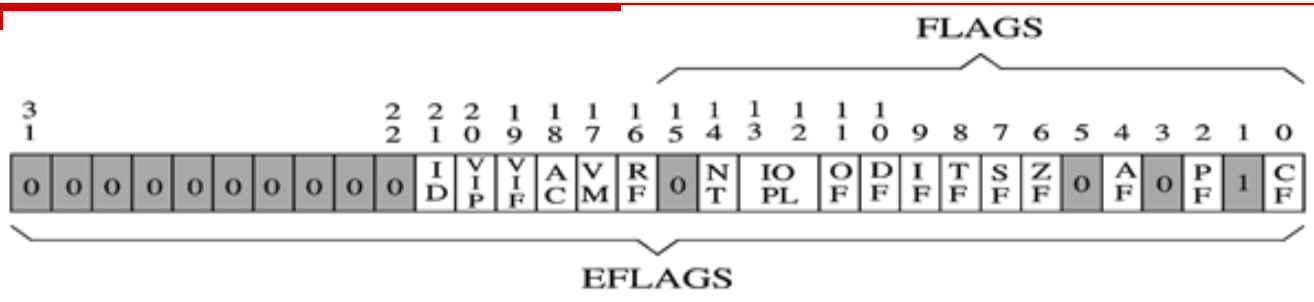
```
        MSB
        0 ----------- X
        1 ----------- Y
------------
0       0 ----------- Z (e.g. P+N=P; this combo is impossible for US; for Signed, its correct results as is)
0       1 ----------- Z (e.g. P+N=N; this results is correct as is for both US and Signed)

1       0 ----------- Z     (A)
CF      SF

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

        0
        0
------------
0       1       B1*   invalid result; correction needed
0       0       C1

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

        1
        1
------------
1       0       B2*   invalid result; correction needed
1       1       C2
```

Special-Purpose & Segment Registers

- For each operation that performed in CPU, there must be *a mechanism* to determine if the operation is success or not
  - The flags are used for this purpose
- IA-32 uses a single register: EFLAGS = **Extended** Flags Register (32 bits)
  - Contains status and control flags
  - Each flag is *a single binary bit*



EFLAGS Register



Status flags

CF = Carry flag
PF = Parity flag
AF = Auxiliary carry flag
ZF = Zero flag
SF = Sign flag
OF = Overflow flag

Control flags

DF = Direction flag

System flags

TF = Trap flag
IF = Interrupt flag
IOPL = I/O privilege level
NT = Nested task
RF = Resume flag
VM = Virtual 8086 mode
AC = Alignment check
VIF = Virtual interrupt flag
VIP = Virtual interrupt pending
ID = ID flag

- Status Flags
  - Status of arithmetic and logical operations
- Control and System flags
  - Control the CPU operation
- Programs can set and clear individual bits in the EFLAGS register

Status Flags

- Carry Flag
  - Set when unsigned arithmetic result is out of range
- Overflow Flag
  - Set when signed arithmetic result is out of range
- Sign Flag
  - Copy of sign bit, set when result is negative
- Zero Flag
  - Set when result is zero
- Auxiliary Carry Flag (was designed for BCD arithmetic)
  - Set when there is a carry from bit 3 to bit 4
- Parity Flag (uses Odd parity!)
  - Set when parity is even
  - Least-significant byte in the result contains even number of 1s

Status Flags

- Carry Flag
  - Set when unsigned arithmetic result is out of range
- Overflow Flag
  - Set when signed arithmetic result is out of range
- Sign Flag
  - Copy of sign bit, set when result is negative

Add 255 and 255 using 8-bit registers. The result should be 510 which is the 9-bit value 111111110 in binary. The 8 least significant bits always stored in the register would be 11111110 binary (254 decimal) but since there is carry out of bit 7 (the eight bit), the carry is set, indicating that the result needs 9 bits. The valid 9-bit result is the concatenation of the carry flag with the result.

For x86 ALU size of 8 bits, an 8-bit two's complement interpretation, the addition operation 1111 1111 + 1111 1111 results in 1 1111 1110, Carry_Flag set, Sign_Flag set, and Overflow_Flag clear.

If 11111111 represents two's complement signed integer −1, then the interpretation of the result is -2 because Overflow_Flag is clear, and Carry_Flag is ignored. The sign of the result is negative, because Sign_Flag is set. 11111110 is the two's complement form of signed integer −2.

If 11111111 represents unsigned integer binary number 255 (ADD al,255), then the interpretation of the result would be 254, which is not correct, because the most significant bit of the result went into the Carry_Flag, which therefore cannot be ignored. The Overflow_Flag and the Sign_Flag are ignored.

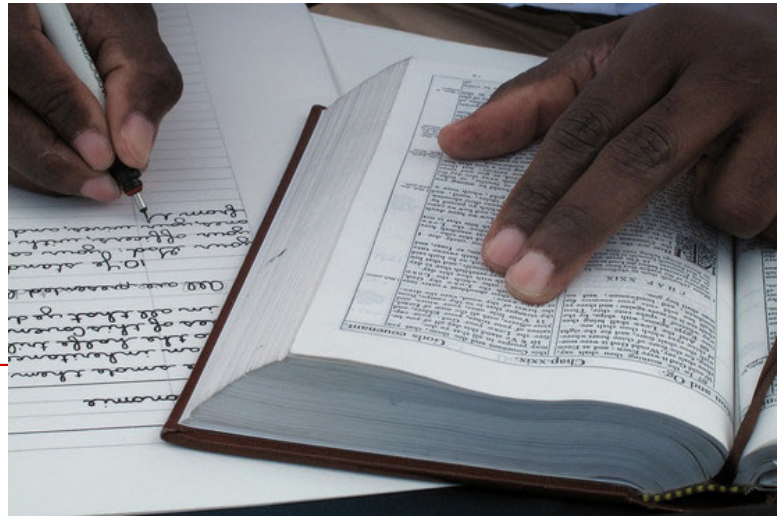
Even parity check

- Even parity check
- Example: input: A(7…0), Output: even_parity bit
  - If there are even numbers of 1 in A, even_parity = ‘0’,
  - If there are odd numbers of 1 in A, even_parity = ‘1’

e.g., A = “10100001”,
even_parity = ‘1’

A = “10100011”,
even_parity = ‘0’

Odd parity check



- Odd parity check
- Example: input: A(7…0), Output: odd_parity bit
  - If there are odd numbers of 1 in A, odd_parity = ‘0’,
  - If there are even numbers of 1 in A, odd_parity = ‘1’

e.g., A = “10100001”,
odd_parity = ‘0’

A = “10100011”,
odd_parity = ‘1’

64-Bit Processors

- 64-Bit Operation Modes
  - Compatibility mode – can run existing 16-bit and 32-bit applications (Windows supports only 32-bit apps in this mode)
  - 64-bit mode – Windows 64 uses this
- Basic Execution Environment
  - addresses can be 64 bits (48 bits, in practice)
  - 16 64-bit general purpose registers
  - 64-bit instruction pointer named RIP

Return Instruction Pointer

64-Bit General Purpose Registers

Compatibility mode

- 32-bit general purpose registers:
  - EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D, R9D, R10D, R11D, R12D, R13D, R14D, R15D
- 64-bit general purpose registers:
  - RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8, R9, R10, R11, R12, R13, R14, R15

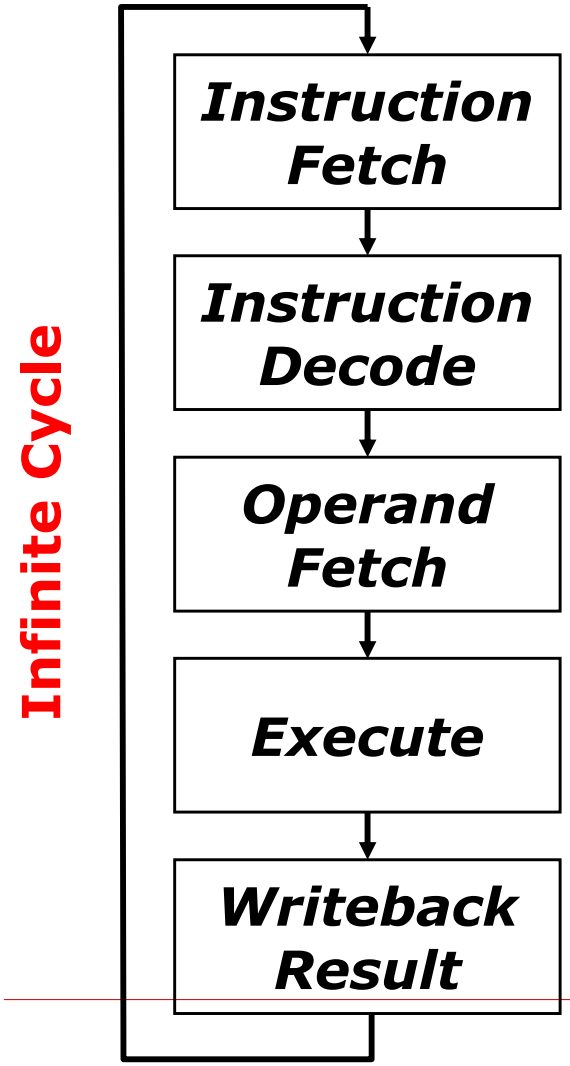
Next ...

- Basic Computer Organization
- IA-32 Registers
- Instruction Execution Cycle

Fetch-Execute Cycle: The Heart-beat of CPU

- Each machine language instruction is first fetched from the memory and stored in an **Instruction Register** or simply **IR**.
- The address of the instruction to be fetched is stored in a register called the **Instruction Pointer** or **EIP**. In some computers this register is called **Program Counter** or simply **PC**.
- After fetching the instruction, the **EIP** (or **PC**) is incremented to point to the address of the next instruction.
- The fetched instruction is decoded (to determine what needs to be done) and executed by the CPU.

Instruction Execute Cycle



Obtain instruction from program storage

Determine required actions and instruction size

Locate and obtain operand data

Compute result value and status

Deposit results in storage for later use