Flags Registers

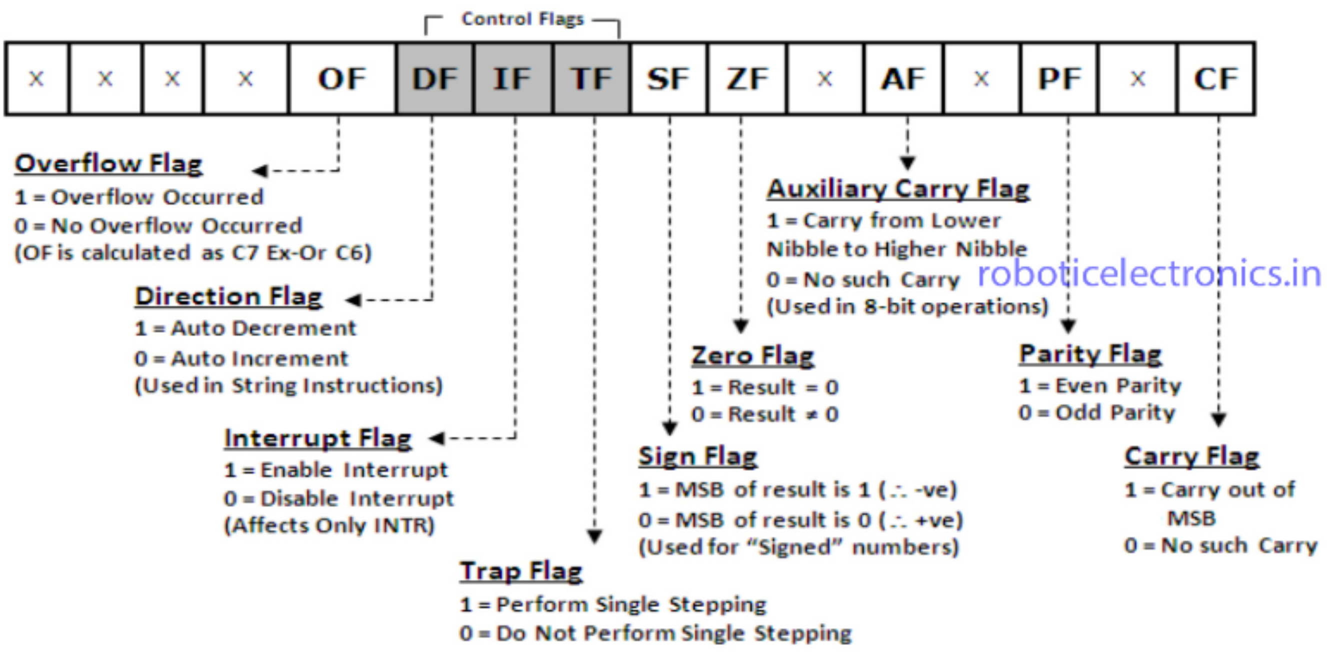
In computer programming, when we perform arithmetic operations, we often need to determine certain properties of the result.

Is it positive, negative, or zero? Is it within the valid range for the destination variable?

To answer these questions, we rely on CPU status flags, which provide valuable information about the outcome of arithmetic operations.

These flags are crucial for error detection and controlling program flow.

Flag register of 8086 is a 16-bit register where status of the latest Arithmetic operation performed.



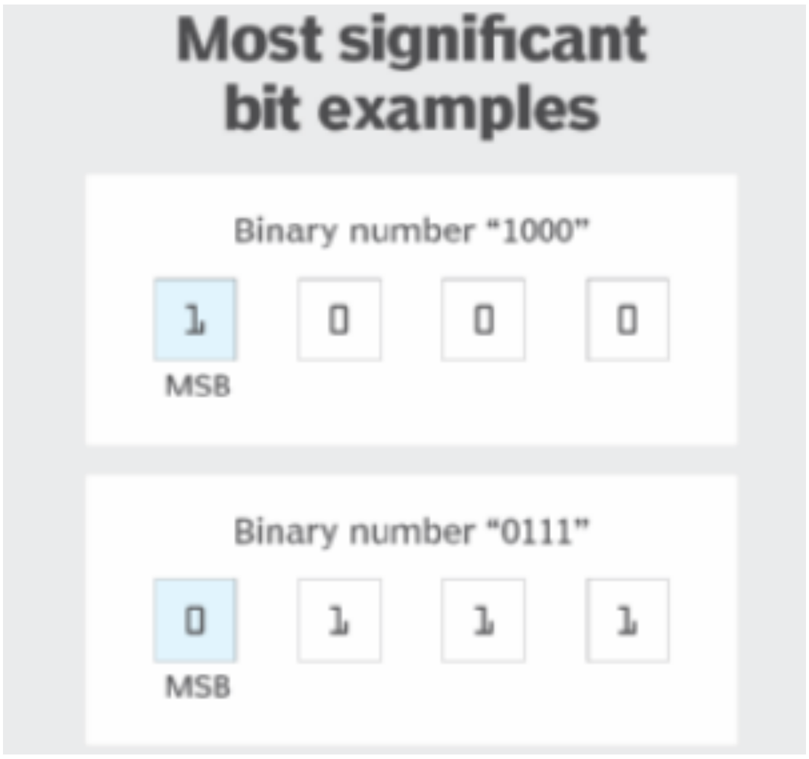
flag register of 8086

As it has 16-bits , it has 16 flags. These 16 flags are classified as

- 7 are don't care flags.
- 3 are control flags (accessible to programmers).
- 6 are status flags (not accessible to programmers).

Here's a straightforward overview of the main status flags.

First, remember this, MSB is the leftmost bit.



==

Carry Flag:

==

Set when the result of an unsigned arithmetic operation is too large to fit into the destination operand.

```
   1111  (Carry-In)
 + 1101
---------
  11000  (Carry-Out)
```

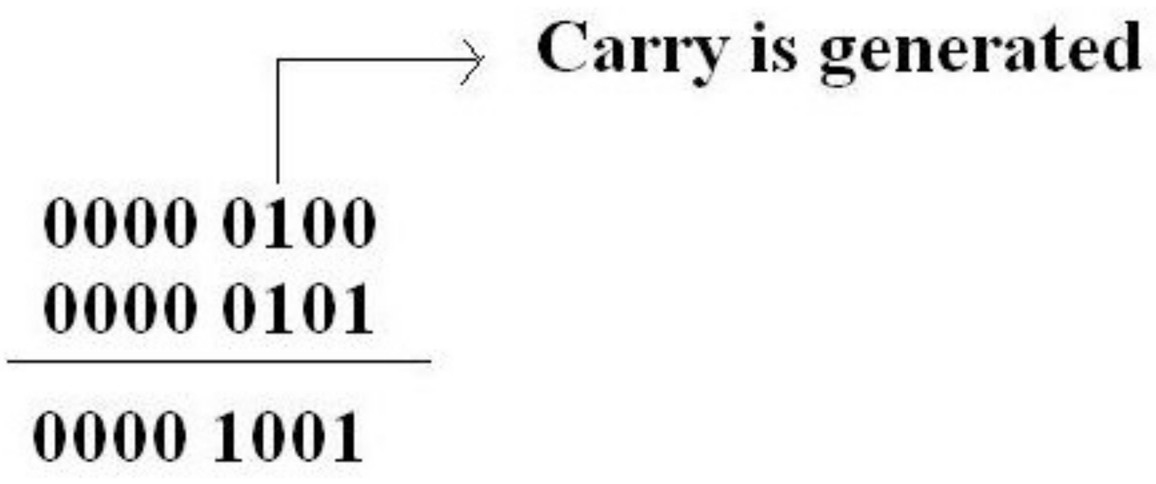
In this example, when you add 1111 and 1101, you have a carry-out from the MSB (leftmost bit), resulting in a final carry-out value of 11000. The Carry flag would be set to indicate this carry-out condition.

For subtraction, there's typically no carry-out from the MSB, as subtraction involves taking away one value from another:

```
   1101
 - 1010
---------
    011
```

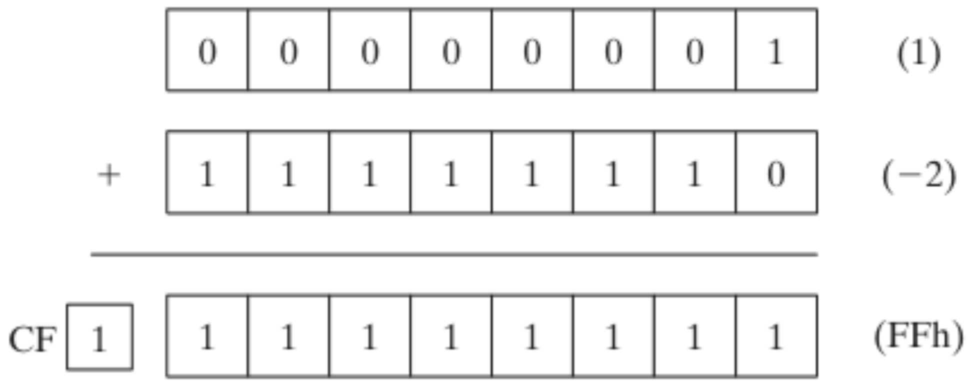
In this subtraction, there's no **carry-out from the LSB**, so the Carry flag would not be set. The carry-out occurs from the least significant bit (LSB), and propagates towards the most significant bit (MSB) during addition, not from the MSB.

This flag indicates unsigned integer overflow. For instance, if an operation has an 8-bit destination operand, but the result exceeds 255 in binary (11111111), the Carry flag is set.



Another example, subtracting 2 from 1 sets the carry flag.

Figure 4–4 Subtracting 2 from 1 sets the Carry flag.



$$\begin{array}{rr} \multicolumn{2}{c}{CF = 1} \\ & \overset{1}{1}0\overset{1}{1}1\ 1100 \\ + & 1001\ 0001 \\ \hline 1 & 0100\ 1101 \end{array} \qquad \begin{array}{rr} \multicolumn{2}{c}{CF = 0} \\ & \overset{1}{1}0\overset{1}{1}1\ 1100 \\ + & 0001\ 0001 \\ \hline & 1100\ 1101 \end{array}$$

If CF = 1; it means carry is generated from the MSB (Most Significant Bit):

• When CF (Carry Flag) is set to 1, it indicates that during an operation like addition, a carry-out occurred from the MSB (the leftmost or highest-order bit) of the result. This typically happens when the sum of two numbers overflows the capacity of the binary representation. In other words, it suggests that the result is too large to fit within the allocated number of bits.

If CF = 0; no carry is generated out of MSB:

• When CF is set to 0, it means that no carry-out occurred from the MSB during the operation. This signifies that the result of the operation did not exceed the capacity of the allocated bits, and there was no overflow.

In summary, the notes are explaining how the Carry Flag (CF) is used to detect whether there was a carry-out from MSB, during arithmetic operations, with CF = 1 indicating overflow and CF = 0 indicating no overflow.

==

Auxiliary Carry Flag

==

Auxiliary Carry Flag: This flag is set when a 1 bit carries out of position 3 in the least significant byte of the destination operand.

position 3 in the least significant byte of the destination operand.

Suppose we want to add 1 to the BCD value 0Fh:

```
mov al, 0Fh  ; Load the value 0Fh into the AL register
add al, 1    ; Add 1 to AL
```

Here's the binary representation and the arithmetic:

1. Binary Representation of 0Fh: 00001111
2. Binary Representation of 1: 00000001

Now, let's add them together, keeping track of the AC flag:

```
  0000 1111  (0Fh)
+ 0000 0001  (1)
------------
  0001 0000  (10h)
```

In this addition, we are adding 1 to 0Fh. As you correctly pointed out, the sum (10h) contains a 1 in bit position 4 that was carried out of bit position 3. This carry from bit 3 to bit 4 is precisely what the Auxiliary Carry (AC) flag represents.

So, in this case:

- **AC = 1:** The AC flag is set because a carry occurred from bit 3 to bit 4 during the addition of 1 to 0Fh. This is a typical scenario in BCD arithmetic, where values are represented in a decimal-like fashion using binary.

The AC flag in assembly language is set when a 1 bit carries out of the 3rd bit (counting from the right, or least significant bit) of the destination operand during an operation.

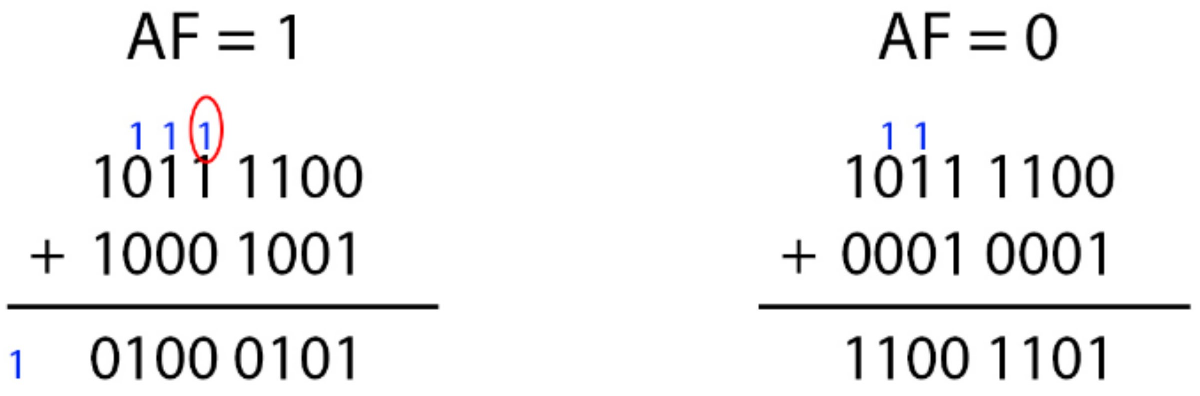
Imagine you have an 8-bit binary number, and you're performing an operation like addition or subtraction. If there's a carry (1) out of the 3rd bit position (bit 2), the AC flag is set to 1. If there's no carry out of that bit position, the AC flag is set to 0.

AF:

AF stands for auxiliary flag. As 8-bits form a byte, similarly 4 bits form a nibble. So in 16 bit operations there are 4 nibbles.

If AF = 1 ; there is a carry out from lower nibble.

If AF = 0 ;no carry out of lower nibble.



==

Parity Flag

==

Parity Flag: The Parity flag informs us whether an even number of 1 bits appears in the least significant byte of the destination operand right after an arithmetic or boolean instruction.

The PF is set when the least significant byte of the destination has an even number of 1 bits. It checks if the result of an operation has an even or odd number of 1 bits in its binary representation.

Example 1: ADD Instruction

```
mov al, 10001100b      ; AL = 10001100
add al, 00000010b      ; Add 00000010 to AL
```

```
   10001100
 + 00000010
-----------
   10001110
```

After the ADD instruction, AL contains the binary value 10001110. In this binary representation, there are four 0 bits and four 1 bits.

Since there is an even number of 1 bits (four), the Parity flag (PF) is set to 1. This means that PF indicates that the result has an even parity.

Example 2: SUB Instruction

```
sub al, 10000000b      ; Subtract 10000000 from AL

   10001110
 - 10000000
-----------
   00001110
```

After the SUB instruction, AL contains the binary value 00001110. In this binary representation, there are three 0 bits and one 1 bit. Since there is an odd number of 1 bits (one), the Parity flag (PF) is set to 0. This indicates that PF detects an odd parity in the result.

In summary:

- **After the ADD instruction, AL contains an even number of 1 bits (four), so PF is set to 1 (even parity).**

- **After the SUB instruction, AL contains an odd number of 1 bits (one), so PF is set to 0 (odd parity).**

The Parity flag (PF) is used for checking whether the result of an operation has even or odd parity, which can be useful in certain applications where parity checks are required for error detection.

PF:

It stands for parity flag.

If PF = 1 ; it means it is even parity in the result (there are even numbers of 1's).

If PF = 0 ; it means it is odd parity.

$$\begin{array}{cc} & \text{PF} = 1 \\ & {}^{1\,1} \\ & 1011\ 1100 \\ + & 1001\ 0001 \\ \hline 1 & 0100\ 1101 \end{array} \qquad\qquad \begin{array}{cc} & \text{PF} = 0 \\ & {}^{1\,1} \\ & 1011\ 1100 \\ + & 0001\ 0001 \\ \hline & 1100\ 1101 \end{array}$$

==

Overflow Flag:

==

Overflow Flag: Set when the result of a signed arithmetic operation is too large or too small to fit into the destination operand.

Unsigned:

This is a 8 bit positive number which ranges from 0 to 255. In hexadecimal it's range is from 00 to FF. In the OF flag, it has nothing to do with unsigned numbers. Only signed numbers are considered in the OF flag.

Signed: This is also a 8-bit number(can be 16-bit too) which is equally distributed among +ve and -ve numbers. By considering MSB , it is decided whether it is a positive or negative number. If MSB=1 , it is a negative number or else its a positive number.

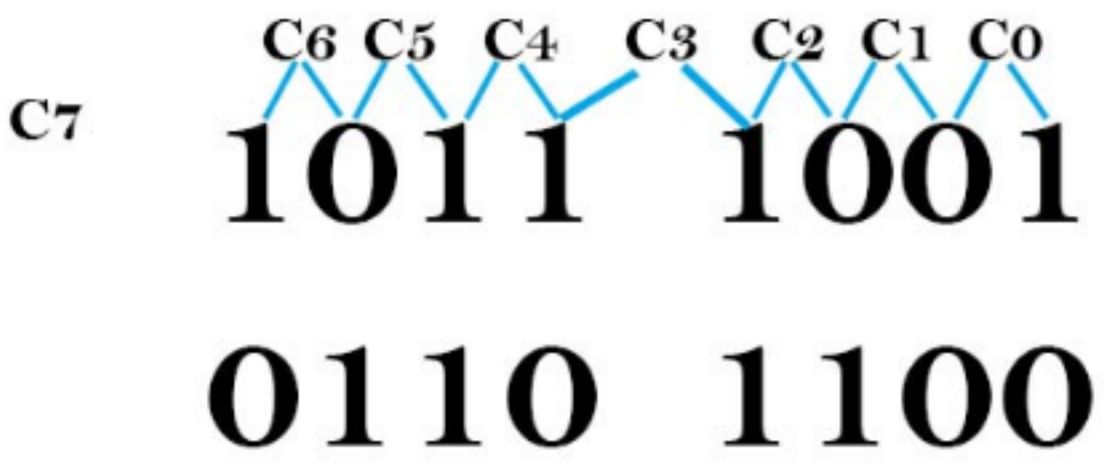
```
1011 0011       ;is -ve

0111 1001       ;is +ve
```

For example, 1011 0011 is negative, and its two's complement is 0100 1101, which is equivalent to 4D in hexadecimal and represents -77 in decimal.

Range of Signed Numbers:
• In an 8-bit signed number, the range is from -128 to 127 (hexadecimal -80 to 7F).
• It consists of 128 positive values (0 to 127) and 128 negative values (-1 to -128).

Overflow Detection:
• Overflow occurs when the result of an operation exceeds the valid range of signed numbers.
• If there's an overflow, the MSB would show incorrect values: MSB = 1 for a positive overflow, MSB = 0 for a negative overflow.

Overflow Detection Mechanism:
• To detect overflow, the processor performs an XOR operation on the carry flags (C6 and C7) during arithmetic operations.
• C0 is the carry from the 0th bit to the 1st bit, and so on. If the XOR operation results in 1, it indicates that the operation has gone beyond the limits of signed numbers and overflowed.



The notes you provided are good to read and accurate. The image you sent shows the following:

> C6 C5 C4 C3 C2 C1 Co
> C7

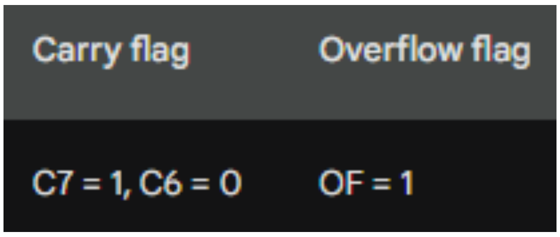
This represents the carry flags from bit 0 to bit 7 of the result of an arithmetic operation.

The **Overflow flag (OF)** is set when the result of a signed arithmetic operation is too large or too small to fit into the destination operand. This can happen when the result is greater than 127 or less than -128.

To detect overflow, the processor performs an XOR operation on the carry flags C6 and C7. If the result of the XOR operation is 1, then overflow has occurred.

For example, let's say we are adding two signed 8-bit numbers: 127 and 1. The result is 128, which is greater than the maximum value for a signed 8-bit number. Therefore, overflow has occurred.

The following table shows how the carry flags and Overflow flag are affected by the addition operation:



Carry flag	Overflow flag
C7 = 1, C6 = 0	OF = 1

This is because the XOR operation of C7 and C6 is 1, which indicates that overflow has occurred.

The Overflow flag can be used to check for errors in arithmetic operations. For example, if we are expecting the result of an addition operation to be within a certain range, we can check the Overflow flag to see if it is set. If it is set, then we know that an error has occurred and we can take corrective action.

Here is an example of how to use the Overflow flag in assembly language:

```
; Add two 8-bit numbers: 127 and 1
add al, 1

; Check the Overflow flag
jnc no_overflow

; Overflow has occurred, handle the error
overflow:
...

; No overflow, continue with the program
no_overflow:
...
```

The assembly code you provided is a simple example of how to use the Overflow flag to detect overflow in an addition operation. The code works as follows:

1. The add al, 1 instruction adds the value 1 to the al register.

2. The jnc no_overflow instruction jumps to the no_overflow label if the Overflow flag is not set.

3. If the Overflow flag is set, then the program jumps to the overflow label, where the error can be handled.

The Overflow flag is set when the result of an addition operation is too large to fit into the destination operand. In this case, the destination operand is the al register, which is an 8-bit register.

The maximum value that can be stored in an 8-bit register is 127. Therefore, if the result of the addition operation is greater than 127, then overflow will occur.

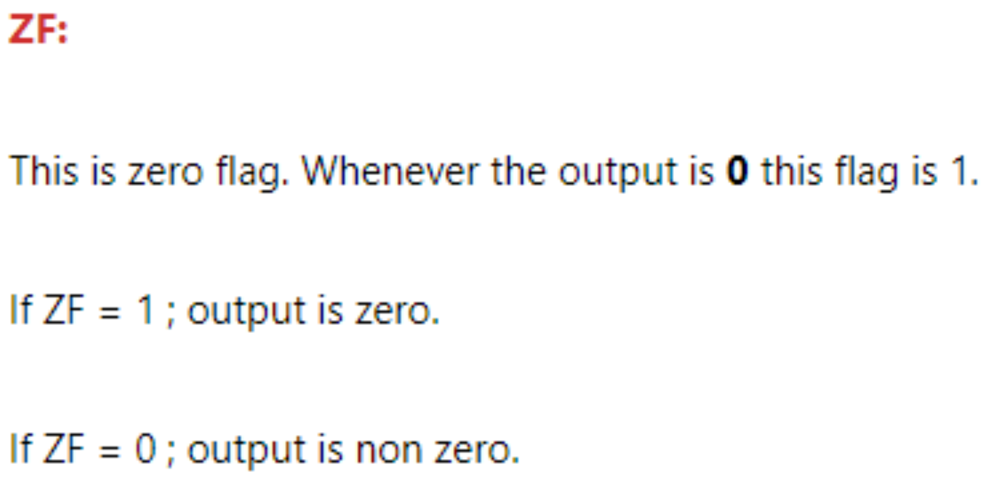
In summary, the Overflow Flag (OF) is used to detect overflow in signed number arithmetic. It's crucial for ensuring the correctness of arithmetic operations, especially when dealing with signed data types. The XOR operation on carry flags is one method used by the processor to detect this overflow condition.

==

Zero Flag:

==

Zero Flag: Set when the result of an arithmetic operation is zero. For example, subtracting two equal values will set the Zero flag.



ZF:

This is zero flag. Whenever the output is **0** this flag is 1.

If ZF = 1 ; output is zero.

If ZF = 0 ; output is non zero.

==

Sign Flag:

==

Sign Flag: Set when the result of an arithmetic operation is negative. If the most significant bit (MSB) of the destination operand is set, indicating a negative number, the Sign flag is set.

The **Sign flag (SF)** is a status flag that is set to the value of the most significant bit (MSB) of the result of an arithmetic operation.

- **If the MSB is 1, then the SF flag is set to 1, which indicates that the result is negative.**

- **If the MSB is 0, then the SF flag is set to 0, which indicates that the result is positive.**

The person is also saying that the status flags, including the SF flag, are controlled by the **ALU (arithmetic logic unit)** and not by the user. This means that the user cannot directly set or clear the status flags.

The person is correct in saying that the status flags can be used to check for errors in arithmetic operations. For example, if the **OF (overflow flag)** is set after an addition operation, then this indicates that the result of the operation is too large to fit into the destination operand. This can be used to detect overflow errors.

Another example is the **ZF (zero flag).** If the ZF is set after an arithmetic operation, then this indicates that the result of the operation is zero. This can be used to check for zero division errors.

The **status flags** are a very important part of the CPU, and they are used by many different instructions. By understanding how the status flags work, you can better understand how the CPU performs arithmetic operations and how to check for errors.

The sign flag can be used to check for errors in arithmetic operations. For example, if the sign flag is set after an addition operation, then this indicates that the result of the operation is negative.

However, if the operands of the addition operation were both positive, then the result should also be positive. **This means that an error has occurred.**

The sign flag can also be used to determine the sign of a number. For example, if the sign flag is set, then the number is negative. If the sign flag is not set, then the number is positive.

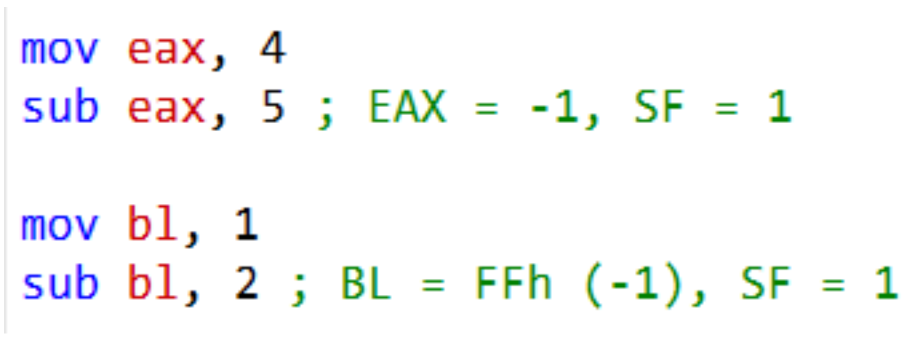
The code you provided shows how the sign flag is affected by a subtraction operation. The first instruction, mov eax, 4, moves the value 4 into the eax register.

The second instruction, sub eax, 5, subtracts 5 from the value in the eax register. Since 5 is greater than 4, the result of the subtraction operation is negative. This is reflected in the value of the sign flag, which is set to 1.

The next part of the code shows the hexadecimal values of the bl register when a negative result is generated. The first instruction, mov bl, 1, moves the value 1 into the bl register.

The second instruction, sub bl, 2, subtracts 2 from the value in the bl register. Since 2 is greater than 1, the result of the subtraction operation is negative.

This is reflected in the hexadecimal value of the bl register, which is FFh, which is the hexadecimal representation of -1.



```
mov eax, 4
sub eax, 5 ; EAX = -1, SF = 1

mov bl, 1
sub bl, 2 ; BL = FFh (-1), SF = 1
```

The sign flag is a useful tool for checking for errors in arithmetic operations and determining the sign of a number.

===

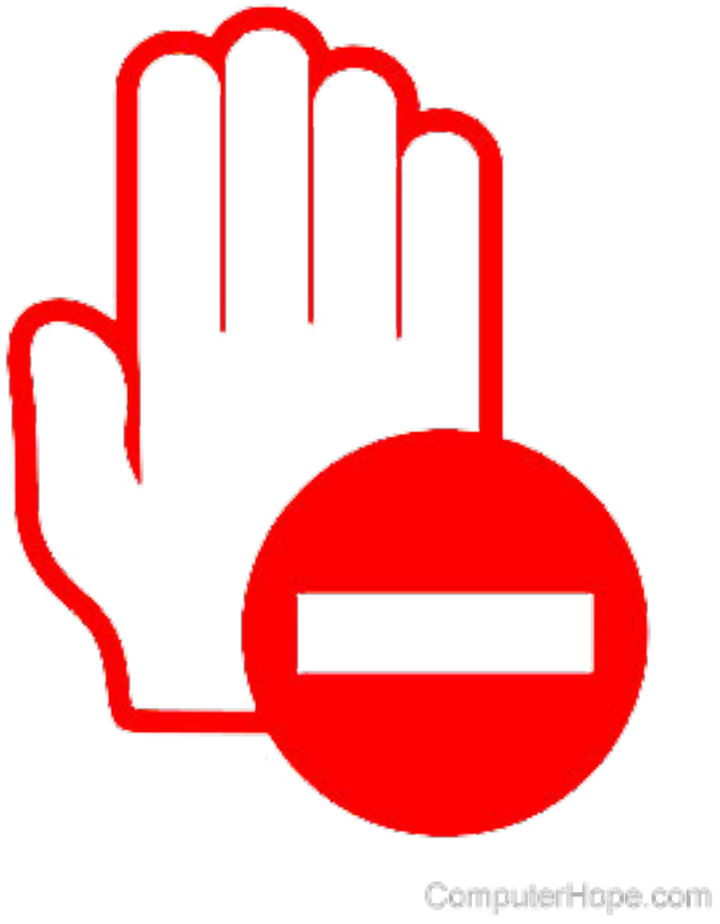
Control Flags:

===

The control flags are three special flags that control the operation of the CPU. They are:

Trap flag (TF): When the TF flag is set to 1, the CPU enters single-step mode. This means that the CPU will execute one instruction and then stop. This is useful for debugging programs, as it allows you to step through a program one instruction at a time.

Interrupt flag (IF): When the IF flag is set to 1, the CPU will allow interrupts. Interrupts are signals from external devices that need the CPU's attention. When an interrupt occurs, the CPU will temporarily stop what it is doing and handle the interrupt. After handling the interrupt, the CPU will return to the program it was executing.



Direction flag (DF): The DF flag controls the direction of string operations. When the DF flag is set to 0, the CPU will increment the string pointer after each operation. When the DF flag is set to 1, the CPU will decrement the string pointer after each operation. The image you sent shows the TF, IF, and DF flags in the flag register of the 8086 microprocessor. The flags are set to 0 by default, which means that the CPU is not in single-step mode, interrupts are disabled, and string operations will increment the string pointer.

Here are some examples of how the control flags can be used:

• **Debugging:** You can use the TF flag to step through a program one instruction at a time. This can be useful for finding bugs in your program. For example, if you are having trouble understanding how the DF flag works, you can set the TF flag and step through the string operations one instruction at a time.

• **Disabling interrupts:** You can use the IF flag to disable interrupts while you are debugging your program. This will prevent interrupts from interfering with your debugging process.

• **Controlling string operations:** You can use the DF flag to control the direction of string operations. For example, if you want to reverse a string, you can set the DF flag to 1.

The **TF flag,** when set to 1, puts the microprocessor in single-step mode, which means that the microprocessor will execute one instruction and then stop. This is useful for debugging programs.

The **IF flag,** when set to 1, enables interrupts. Interrupts are signals from external devices that need the microprocessor's attention. When an interrupt occurs, the microprocessor will temporarily stop what it is doing and handle the interrupt. After handling the interrupt, the microprocessor will return to the program it was executing.

The **DF flag** controls the direction of string operations. When the DF flag is set to 0, the microprocessor will increment the string pointer after each operation. When the DF flag is set to 1, the microprocessor will decrement the string pointer after each operation.

```
;AddSubTest.asm - Arithmetic and Flags Demonstration
.386
.model flat,stdcall
.stack 4096

ExitProcess proto,dwExitCode:dword

.data
    Rval SDWORD ?
    Xval SDWORD 26
    Yval SDWORD 30
    Zval SDWORD 40

.code

main PROC
    ; INC and DEC
    mov ax, 1000h
    inc ax              ; Increment: AX = 1001h
    dec ax              ; Decrement: AX = 1000h

    ; Expression: Rval = -Xval + (Yval - Zval)
    mov eax, Xval       ; Load Xval into EAX
    neg eax             ; Negate EAX (convert to negative)
    mov ebx, Yval       ; Load Yval into EBX
    sub ebx, Zval       ; Subtract Zval from Yval
    add eax, ebx        ; Add the negated Xval and (Yval - Zval) to get Rval
    mov Rval, eax       ; Store the result in Rval
    ; Zero Flag example
    mov cx, 1
    sub cx, 1           ; ZF = 1 because the result is zero

    ; Sign Flag example
    mov cx, 0
    sub cx, 1           ; SF = 1 because the result is negative
    mov ax, 7FFFh
    add ax, 2           ; SF = 1 because the result is negative

    ; Carry Flag example
    mov al, 0FFh
    add al, 1           ; CF = 1, AL = 00 (overflow)

    ; Overflow Flag example
    mov al, 127          ; Maximum positive value
    add al, 1            ; OF = 1 (overflow)
    mov al, -128         ; Minimum negative value
    sub al, 1            ; OF = 1 (overflow)

    INVOKE ExitProcess, 0
main ENDP

END main
```

Repetition of the whole code:

```
; AddSubTest.asm - Arithmetic and Flags Demonstration

.386
.model flat,stdcall
.stack 4096

ExitProcess proto,dwExitCode:dword

.data
    Rval SDWORD ?
    Xval SDWORD 26
    Yval SDWORD 30
    Zval SDWORD 40

.code

main PROC
    ; INC and DEC
    mov ax, 1000h
    inc ax                  ; Increment: AX = 1001h
    dec ax              ; Decrement: AX = 1000h

    ; Expression: Rval = -Xval + (Yval - Zval)
    mov eax, Xval       ; Load Xval into EAX
    neg eax             ; Negate EAX (convert to negative)
    mov ebx, Yval       ; Load Yval into EBX
    sub ebx, Zval       ; Subtract Zval from Yval
    add eax, ebx        ; Add the negated Xval and (Yval - Zval) to get
Rval
    mov Rval, eax       ; Store the result in Rval

    ; Zero Flag example
    mov cx, 1
    sub cx, 1           ; ZF = 1 because the result is zero

    ; Sign Flag example
    mov cx, 0
    sub cx, 1           ; SF = 1 because the result is negative
    mov ax, 7FFFh
    add ax, 2           ; SF = 1 because the result is negative

    ; Carry Flag example
    mov al, 0FFh
    add al, 1           ; CF = 1, AL = 00 (overflow)

    ; Overflow Flag example
    mov al, 127          ; Maximum positive value
    add al, 1            ; OF = 1 (overflow)
    mov al, -128         ; Minimum negative value
    sub al, 1            ; OF = 1 (overflow)

    INVOKE ExitProcess, 0
main ENDP

END main
```