**Flags**

A *flag* is an indicator represented on 1 bit. A configuration of the FLAGS register shows a synthetic overview of the execution of the each instruction. For x86 the EFLAGS register (the status register) has 32 bits but only 9 are actually used.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **31** | **30** | **…** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| x | x | … | x | OF | DF | IF | TF | SF | ZF | x | AF | x | PF | x | CF |

**CF** (*Carry Flag*) is the transport flag. It will be set to 1 if in the LPO there was a transport digit outside the representation domain of the obtained result and set to 0 otherwise. For example, in the addition

1001 0011 + 147 + 93h + -109 +

0111 0011 115 there is transport and CF 73h 115

**1** 0000 0110 262 is set therefore to 1 106h 06

**CF flags the UNSIGNED overflow!** => CF = 1

**PF** (*Parity Flag*) – Its value is set so that together with the bits 1 from the least significant byte of the representation of the LPO’s result an odd number of 1 digits to be obtained.

1001 0011 +

0111 0011

**1** 0000 0110 => PF = 1 (without that CF, because he is not into the memory, he appears into EFLAGS registers)

**AF** (*Auxiliary Flag*) shows the transport value from bit 3 to bit 4 of the LPO’s result. For the above example the transport is 0.

0110 1100 + 6ch + c4h = 130h

1100 0100

1 0011 0000

1 3 0 h AF = 1

4 bits = 1 semioctet = 1 nibble = 1 hexadecimal digit

**ZF** (*Zero Flag*) is set to 1 if the result of the LPO was zero and set to 0 otherwise.

**SF** (*Sign Flag*) is set to 1 if the result of the LPO is a strictly negative number and is set to 0 otherwise.

**TF** (*Trap Flag*) is a debugging flag. If it is set to 1, then the machine stops after every instruction step by step. For example, if we reach the breakpoint = > TF = 1 otherwise TF = 0.

**IF** (*Interrupt Flag*) is an interrupt flag. If set to 1, interrupts are allowed, if set to 0 interrupts will not be handled.

**DF** (*Direction Flag*) – for operating string instructions. If set to 0, then string parsing will be performed in an ascending order (from the beginning to its end) and in a descending order if set to 1.

**OF** (*Overflow Flag*) flags the **signed overflow**. If the result of the LPO (considered in the signed interpretation) didn’t fit the reserved space, then OF will be set to 1 and will be set to 0 otherwise.

0101 0011 + 83 + 53h + 83 +

0111 0011 115 a carry DOES NOT 73h 115

1100 0110 198 occur so CF=0 C6h - 58

**(unsigned) (hexa) (signed)**

= > CF = 0 = > OF = 1

The flags can be split into 2 categories:

a). with a previous effect generated by the Last Performed Operation (LPO): CF, PF, AF, ZF, SF, OF

b). having a future effect after their setting by the programmer, to influence the way the next instructions are run: CF, TF, DF, IF.

Considering the b) category, it is normal that the assembly language to provide specific instructions to set the values of the flags that will have a future effect. So, we have 7 such instructions:

; we have 3 instructions for CF

CLC – the effect is CF = 0

STC – sets CF = 1

CMC – complements the value of the CF

; 2 instructions for DF

CLD – sets DF = 0

STD – sets DF = 1

; 2 instructions for IF – they can be used by the programmer only on 16 bits programming

; on 32 bits, the OS restricts the access to these instructions

CLI – sets IF = 0

STI – sets IF = 1

Given the major risk of accidentally setting the value from TF and also its absolutely special role to develop debuggers (without TF we do will not have the debugger), there are NO instructions to directly access the value of TF.

|  |  |
| --- | --- |
| **PUSHF** | Pushes EFLAGS in the stack |
| **POPF** | Pops the top of the stack and transfers it to EFLAGS |

The following four instructions are *flags transfer instructions*:

**LAHF** (***L****oad register* ***AH*** *from* ***F****lags*) copies SF, ZF, AF, PF and CF from FLAGS register in the bits 7, 6, 4, 2 and 0, respectively, of register AH. The contents of bits 5, 3 and 1 are undefined. Other flags are not affected (meaning that LAHF does not generate itself other effects on some other flags – it just transfers the flags values and that’s all).

**SAHF** (***S****tore register* ***AH*** *into* ***F****lags*) transfers the bits 7, 6, 4, 2 and 0 of register AH in SF, ZF, AF, PF and CF respectively, replacing the previous values of these flags.

If we use jumps with condition, we can check it with flags:

JB, JNAE, JC = > CF = 1

JAE, JNB, JNC = > CF = 0

JBE, JNA = > CF = 1 or ZF = 1

JA, JNBE = > CF = 0 and ZF = 0

JE, JZ = > ZF = 1

JNE, JNZ = > ZF = 0

JL, JNGE = > SF != OF

JGE, JNL = > SF = OF

JLE, JNG => ZF = 1 or SF != OF

JG, JNLE = > ZF = 0 and SF = OF

JP, JPE = > PF = 1

JNP, JPO = > PF =0

JS = > SF = 1

JNS = > SF = 0

JO = > OF = 1

JNO = > OF = 0

OF will be set to 1 (*signed overflow*) if for the addition operation we are in one of the following situations (*overflow rules for addition in signed interpretation*). These are the only 2 situations that can issue overflow status for the addition operation:

0……+ or 1…..+ (Semantically, the two situations denote the impossibility of mathematical acceptance

0…… 1….. of the 2 operations: we cannot add two positive numbers and obtain a negative result

-------- ------- and we cannot add two negative numbers and obtain a positive result).

1…… 0…..

In the case of subtraction, we also have two overflow rules in the signed interpretation, a consequence of the two overflow rules for addition:

1……- or 0…..- (Semantically, the two situations denote the impossibility of mathematical acceptance

0…… 1….. of the two operations: we cannot subtract a positive number from a negative one and

------ ------- obtaining a positive one, neither can we subtract a negative number from a positive one

0…… 1….. and obtaining a negative one).

The multiplication operation does not produce overflow at the level of 80x86 architecture, the reserved space for the result being enough for both interpretations. Anyway, even in the case of multiplication, the decision was taken to set both CF = OF = 0, in the case that the size of the result is the same as the size of the operators, b\*b = b, w\*w = w or d\*d = d, meaning no multiplication overflow. In the case that b\*b = w, w\*w = d, d\*d = qword, then CF = OF = 1, meaning multiplication overflow.

The worst effect in case of overflow is in the case for the division operation: in this situation, if the quotient does not fit in the reserved space (the space reserved by the assembler being byte for the division word/byte, word for the division doubleword/word and respectively doubleword for division quadword/doubleword) then the division overflow will signal a ‘Run-time error’ and the operating system will stop the running of the program and will issue one of the 3 semantic equivalent messages: ‘Divide overflow’, ‘Division by zero’ or ‘Zero divide’.

In the case of a correct division CF and OF are undefined. If we have a division overflow, the program crashes, the execution stops and of course it doesn’t matter which are the values from CF and OF.