

4.3.2.3. Exemple comentate

Modul de acțiune a instrucțiunilor de salt condiționat și instrucțiunea **cmp**

`mov al,80h` ;`al` := 128 = 10000000b = -128 ! (Interesant! – remarcăm faptul că datorită ;regulilor de reprezentare în cod complementar 128 și -128 au aceeași ;reprezentare binară și anume 10000000b!)

(*) `cmp al,0` ;instrucțiunea **cmp** nu interpretează în nici un fel valoarea din AL (ca fiind ;cu semn sau fără semn) ci doar realizează scăderea fictivă `al-0` și afectează ;corespunzător flagurile: SF=1 , CF=ZF=OF=PF=AF=0.

`jl et` ;utilizarea instrucțiunii JL (Jump if Less than) provoacă interpretarea ;comparației `al<0` cu semn (vezi tabelul 4.2), adică cf. tabelului 4.1 se ;testează dacă SF≠OF și cum SF=1 iar OF=0 se decide îndeplinirea condiției ;și saltul la eticheta **et**. Deducem deci că interpretarea valorilor comparate a ;stat la latitudinea programatorului care prin utilizarea instrucțiunii JL a ;decis că dorește să compare -128 cu 0 și cum -128 este “less than” 0 ;condiția a fost îndeplinită (echiv. cu **jnge et**). În contrast, **jnl et** sau **jge et** ;(care vor testa dacă SF=OF) NU vor fi îndeplinite și NU vor provoca saltul ;la eticheta specificată.

`jb et` ;utilizarea instrucțiunii JB (Jump if Below) provoacă interpretarea ;comparației `al<0` fără semn (vezi tabelul 4.2), adică cf. tabelului 4.1 se ;testează dacă CF=1 și cum CF=0 se decide neîndeplinirea condiției deci nu ;se va face saltul la eticheta **et**. Deducem deci că interpretarea valorilor comparate a stat la latitudinea programatorului care prin utilizarea instrucțiunii JB a ;decis că dorește să compare 128 cu 0 și cum 128 NU este “below” 0 condiția NU a fost îndeplinită (echivalent cu **jnae et** sau **jc et**).

`jae et1` ;se testează fără semn dacă `al ≥ 0` (128 ≥ 0?) - CF=0 deci condiție ;îndeplinită (echivalent cu **jnc et1** sau **jnb et1**) – se efectuează saltul la ;eticheta **et1**

`jbe et2` ;se testează fără semn dacă `al ≤ 0` (128 ≤ 0?) – CF = ZF = 0 deci condiția ;(CF=1 sau ZF=1) NU este îndeplinită și ca urmare nu se va face saltul la ;eticheta **et2** – rezultat consistent cu **jb et**, deoarece **jbe** implică **jb** ;(echivalent cu **jna et2**)

`ja et3` ;se testează fără semn dacă `al > 0` (128 > 0?) – CF = ZF = 0 deci condiția ;(CF=0 și ZF=0) este îndeplinită și ca urmare se

va face saltul la eticheta **et3** ;(echivalent cu **jnb et3**) și rezultat consistent cu **jbe et2**, deoarece dacă ;jbe nu este îndeplinită atunci **ja** trebuie să fie.

je et4 ;se testează dacă $al = 0$ ($128 = 0$?) – nu se pune problema semnului dacă se ;testează egalitatea! – cum $ZF=0$, condiția $ZF=1$ nu este îndeplinită deci nu ;se va efectua saltul la eticheta **et4** (echivalent cu **jz et4**). În contrast, **jne ;et4** sau **jnz et4** (care vor testa dacă $ZF=1$) vor fi îndeplinite și vor provoca ;saltul la eticheta specificată.

jle et5 ;se testează cu semn dacă $al \leq 0$ ($-128 \leq 0$?) – $OF = ZF = 0$ și $SF=1$ deci ;condiția ($ZF=1$ sau $SF \neq OF$) este îndeplinită și ca urmare se va face saltul la ;eticheta **et5** (echivalent cu **jng et5**) și rezultat consistent cu **jl et**, deoarece ;jle implică **jl**.

jg et6 ;se testează cu semn dacă $al > 0$ ($-128 > 0$?) – $OF = ZF = 0$ și $SF=1$ deci ;condiția ($ZF=0$ și $SF=OF$) NU este îndeplinită și ca urmare NU se va face ;saltul la eticheta **et6** (echivalent cu **jnl et6**) și rezultat consistent cu **jle ;et5**, deoarece dacă **jg** nu este îndeplinită atunci **jle** trebuie să fie.

jp et7 ;se testează dacă $PF=1$ - $PF=0$ deci condiție neîndeplinită – nu se efectuează ;saltul (echivalent cu **jpe et7** – *Jump if Parity Even*). În contrast, **jnp et7** ;(care testează dacă $PF=0$ – echivalentă cu **jpo et7** – *Jump if Parity Odd*) va ;fi îndeplinită și saltul se va efectua.

jo et8
efectuează

;se testează dacă $OF=1$ - $OF=0$ deci condiție neîndeplinită – nu se ;saltul (nu există depășire). În contrast, **jno et8** (care testează dacă $OF=0$) va ;fi îndeplinită și saltul se va efectua.

js et9 ;se testează dacă în interpretarea cu semn **rezultatul** comparației are semn ;negativ (deoarece așa cum specificam în cadrul prezentării instrucțiunii ;CMP, nu este vorba de a interpreta cu semn sau fără semn **operandii** ;scăderii fictive *d-s*, ci **rezultatul** final al acesteia !) adică testăm dacă $SF=1$ -;condiție îndeplinită în cazul nostru și ca urmare saltul se va efectua !. În ;contrast, **jns et9** (care testează dacă $SF=0$) NU va fi îndeplinită și saltul ;NU se va efectua.

cmp 0,al ;eroare de sintaxă : “*Illegal immediate*” deoarece sintaxa instrucțiunii **cmp** ;interzice specificarea ca prim operand a unei valori imediate (constante). ;dacă totuși dorim forțarea unei

comparații de acest tip (0-al) putem utiliza ;pe post de prim operand un registru inițializat cu valoarea 0.

```
mov bl,0
```

```
cmp bl, al ;realizează scăderea fictivă bl-al (0-al = 0-80h = 0-10000000b = ;10000000b) și afectează corespunzător flagurile: CF=SF=OF=1, ;ZF=PF=AF=0.
```

Exercițiu propus: Reluați discuția efectului tuturor instrucțiunilor de salt condiționat de mai sus (analizate pentru cazul comparației (*) `cmp al,0`) în condițiile în care această comparație e înlocuită de ultimele două instrucțiuni prezentate, adică în cazul în care se efectuează `cmp bl,al` cu `bl=0`.

Care ar fi însă justificarea faptului că în cazul `cmp bl,al` avem $CF = OF = SF = 1$ iar în cazul `cmp al,0` doar $SF=1$ iar $CF = OF = 0$?

Pentru a justifica modurile de setare diferite ale flag-urilor trebuie să luăm în discuție regulile practice de setare a acestor flag-uri. Aceste reguli generale sunt:

- SF ia valoarea bitului de semn al rezultatului obținut;
- CF ia valoarea cifrei de transport : dacă e vorba despre o adunare se analizează dacă rezultatul obținut a provocat ($CF=1$) sau nu ($CF=0$) un transport în afara spațiului de reprezentare; dacă e vorba despre o scădere *d-s*, avem: dacă $|d| \geq |s|$ atunci $CF=0$ (nu e nevoie de cifră de împrumut pentru efectuarea scăderii) iar dacă $|d| < |s|$ atunci $CF=1$ (este nevoie de cifră de împrumut pentru efectuarea scăderii și acest lucru se reflectă în CF)
- OF este setat la valoarea 1 dacă există depășire în interpretarea cu semn a rezultatului (“*OF is set if there exists a signed overflow*”), adică dacă rezultatul obținut nu se încadrează în intervalul de interpretare admis (acesta fiind $[-128..+127]$ dacă este vorba despre octeți și respectiv $[-32768..+32767]$ pentru cuvinte interpretate cu semn).

Ultimele două reguli derivă de fapt din modul de implementare a conceptului de **depășire** (*overflow*) la nivelul procesorului 80x86.

În cazul operațiilor/operanzilor fără semn depășirea va fi semnalată prin setarea indicatorului CF (*carry flag*). În cazul operațiilor/operanzilor cu semn depășirea va fi semnalată prin setarea indicatorului OF (*overflow flag*).

Cum să detectăm însă situațiile de depășire în cazul operațiilor de adunare și scădere ? Care sunt regulile practice de aplicat pentru a înțelege și a putea justifica corect setările de flag-uri pe care le remarcăm în cadrul programelor rulate ? În discuțiile ce urmează ne vom concentra în principal pe justificarea modului de setare a flag-ului OF (*overflow flag*) deoarece și datorită numelui său acesta este principalul factor răspunzător de caracterizarea unei situații din partea programatorilor ca fiind depășire sau nu.

Atragem însă atenția asupra a ceea ce se ignoră de multe ori în acest context și anume faptul că o situație de tipul CF=1 (cu OF=0) semnaleză la rândul ei o depășire, însă pentru cazul numerelor interpretate fără semn.

Pentru ADUNARE: dacă se adună două numere de același semn și rezultatul este de semn diferit atunci se semnalează depășire (OF=1), **în caz contrar nu** (OF=0). Aceasta este deci ceea ce am putea numi *regula depășirii la adunare* (RDA) în cazul interpretării cu semn.

De exemplu, la nivel de octet, dacă vom considera adunarea $100 + 50 = 150$ vom obține depășire (!) cu semn (pare surprinzător, nu-i așa?). Justificare: $100 (= 64h = 01100100b) + 50 (= 32h = 00110010b) = 150 (= 96h = 10010110b)$. Operanzii au același semn dar rezultatul este de semn diferit, deci conform RDA vom avea OF=1. Intuitiv, depășirea se poate justifica prin faptul că $150 \notin [-128..127]$ deci se obține o eroare de tip “out of range”. Deși s-ar putea replica faptul că $150 = 10010110b = -106$ (în interpretarea cu semn), iar $-106 \in [-128..127]$, această ultimă interpretare nu poate fi acceptată deoarece operanzii (100 și 50) au valori pozitive în ambele interpretări (bitul de semn fiind 0). Ca urmare, suma a două numere pozitive nu poate da un număr negativ și astfel singura interpretare ce poate fi acceptată în acest context pentru 10010110b este $150 \notin [-128..127]$ deci se setează OF=1.

Pe de altă parte, CF = 0 (nu există cifră de transport în afara spațiului de reprezentare) deci nu avem depășire în interpretarea fără semn: rezultatul adunării $100 + 50 = 150 \in [0, 255]$ (intervalul de interpretare admis pentru numere fără semn).

Analog, în interpretarea cu semn, suma a două numere negative nu poate furniza un număr pozitiv. Luăm exemplul:

$$\begin{array}{r} 10010110 + \\ 10000010 \\ \hline 1\ 00011000 \end{array}$$

Se observă din reprezentarea binară că există un transport de cifră 1 în afara spațiului de reprezentare admis al celor 8 biți, deci intuitiv este suficient de justificat depășirea. Din punct de vedere al aplicării RDA obținem pe 8 biți în interpretarea cu semn că suma a două numere negative (ele sunt negative deoarece bitul de semn este 1 pentru ambele numere) ar trebui să furnizeze un număr pozitiv: $00011000b = 18h = 24$. Această valoare este de fapt o trunchiere a valorii binare corecte (pe 9 biți!) ce ar fi trebuit obținută ($100011000b = 118h$), iar trunchierea are loc tocmai datorită depășirii. Ca urmare, nu se poate obține un număr pozitiv prin adunarea a două numere negative (decât printr-o trunchiere iar necesitatea trunchierii înseamnă de fapt depășire!). Se observă că o astfel de trunchiere înseamnă întotdeauna și apariția unei cifre de transport 1 în afara dimensiunii de reprezentare a rezultatului, deci vom avea automat și CF=1.

$$\begin{array}{l} 10010110b = 96h = -106 \text{ (în interpretarea cu semn)} = +150 \text{ (în interpretarea fără semn)} \\ 10000010b = 82h = -126 \text{ (în interpretarea cu semn)} = +130 \text{ (în interpretarea fără semn)} \end{array}$$

În interpretarea fără semn avem $150 + 130 = 280 \notin [0..255]$ (justificarea intuitivă a depășirii). Tehnic, am văzut deja că $CF = 1$ și rezultă astfel clar că avem depășire în interpretarea fără semn.

Nu putem avea deci $-106 + (-126) = 24!$ (pentru că $00011000b = 18h = 24$ în ambele interpretări) Acesta este sensul în care se aplică RDA aici. Un alt mod de justificare intuitivă a depășirii în acest tip de situație este:

În interpretarea cu semn avem $-106 + (-126) = -232 \notin [-128..127]$ deci $OF=1$.

Această ultimă motivație este mai intuitivă pentru justificarea depășirii însă astfel de justificări sunt mai greu de exprimat la nivelul unui algoritm. Tehnic vorbind, RDA rămâne “cea mai rapid aplicabilă regulă practică din punct de vedere algoritmic” dacă ne putem exprima așa... (și iată că am putut!)

Rezultă că în cazul în care adunăm două numere de semne diferite nu se va semnala niciodată depășire. De asemenea, dacă adunăm două numere de același semn dar rezultatul are același semn cu operanzii nu se va semnala nici în acest caz depășire (înseamnă că nu a fost nevoie de trunchiere pentru reprezentarea rezultatului pe aceeași dimensiune ca și cea a operanzilor). Se poate verifica ușor din punct de vedere matematic că în nici unul din aceste cazuri nu ieșim din intervalul de interpretare admis.

Pentru SCĂDERE: se interpretează **operanzii** respectivi cu semn, se efectuează scăderea solicitată asupra configurațiilor corespunzătoare de biți și dacă rezultatul obținut interpretat cu semn nu se încadrează în intervalul de interpretare admis (intervalul $[-128..127]$ pentru octeții cu semn și respectiv $[-32768..32767]$ pentru cuvinte interpretate cu semn) atunci se semnalează depășire (*overflow*) și astfel $OF=1$. Această formulare o putem numi *regula depășirii la scădere* (RDS) pentru cazul interpretării cu semn.

În cazul depășirii la scădere fără semn: necesitatea efectuării unei scăderi cu împrumut de cifră este semnalată de către procesor prin setarea $CF=1$, pe care o putem interpreta semnificativ drept “depășire la scădere în interpretarea fără semn”.

Să analizăm în continuare mai multe exemple menite să clarifice aplicarea regulilor de mai sus precum și impactul lor asupra modului de setare al flag-urilor.

Exemple:

i). `mov ah,82h` ;82h = 130 (interpretarea fără semn) = -126 (interpretarea cu semn)
 ; = 10000010b (bitul de semn fiind 1 cele două interpretări diferă)
 `mov bh,2ah` ;2ah = 42 (atât în interpretarea cu semn cât și în cea fără semn)
 ; = 00101010b (bitul de semn fiind 0 cele două interpretări coincid)
 `cmp ah,bh` ;se realizează scăderea fictivă $ah-bh=10000010b - 00101010b =$
01011000b
 ; = 58h = 88 (atât în interpretarea cu semn cât și în cea fără semn
 deoarece ;bitul de semn este 0)

Această scădere setează flag-urile astfel:

SF = 0 (deoarece bitul de semn pentru rezultatul 58h = 01011000b este 0)

CF = 0 (deoarece $|82h| > |2ah|$ nu se pune problema unei scăderi cu împrumut de cifră; deci

nu vom avea depășire în interpretarea fără semn care se efectuează: $130 - 42 = 88$)

OF = 1 (se efectuează scăderea în interpretarea cu semn, adică $ah - bh = -126 - 42 = -168$ și

cum $-168 \notin [-128..127]$ se semnalează *signed overflow* și ca urmare OF=1)

`cmp bh,ah` ;se realizează scăderea fictivă $bh - ah = 00101010b - 10000010b = 10101000b$

; = A8h = 168 (în interpretarea fără semn) = -88 (în interpretarea cu semn)

Această scădere setează flag-urile astfel:

SF = 1 (deoarece bitul de semn pentru rezultatul A8h = 10101000b este 1)

CF = 1 (deoarece $|2ah| < |82h|$ se pune problema unei scăderi cu împrumut de cifră; în interpretarea fără semn scăderea devine $42 - 130 = 168$ (!) provenită de fapt din necesitatea unei scăderi de tipul $(256 + 42) - 130 = 168$ și ca urmare a necesității împrumutului se va semnala depășire în interpretarea fără semn, înțeleasă aici ca “nu se poate efectua corect această scădere fără utilizarea unei cifre de împrumut”)

OF = 1 (se efectuează scăderea în interpretarea cu semn, adică $bh - ah = 42 - (-126) = +168$ și

cum $+168 \notin [-128..127]$ se semnalează *signed overflow* și ca urmare OF=1)

ii). `mov ah,126` ;echivalent cu `mov ah,7eh` deoarece $126 = 7Eh = 01111110b$ (bitul de semn fiind 0 cele două interpretări coincid, ca urmare conținutul lui AH este ;126 atât în interpretarea cu semn cât și în cea fără semn)

`mov bh,2ah` ;2ah = 42 (atât în interpretarea cu semn cât și în cea fără semn)

; = 00101010b (bitul de semn fiind 0 cele două interpretări coincid)

`cmp ah,bh` ;se realizează scăderea fictivă $ah - bh = 01111110b - 00101010b = 01010100b$

; = 54h = 84 = $126 - 42$ (atât în interpretarea cu semn cât și în cea fără semn deoarece bitul de semn al rezultatului este 0)

Această scădere setează flag-urile astfel:

SF = 0 (deoarece bitul de semn pentru rezultatul 54h = 01010100b este 0)

CF = 0 (deoarece $|126| > |42|$ nu se pune problema unei scăderi cu împrumut de cifră, deci nu se va semnala depășire în interpretarea fără semn)

OF = 0 (se efectuează scăderea în interpretarea cu semn, adică $ah - bh = 126 - 42 = 84$ și

cum $84 \in [-128..127]$ NU se semnalează *signed overflow* și ca urmare OF=0)

cmp bh,ah ;se realizează scăderea fictivă $bh - ah = 00101010b - 01111110b = 10101100b$

; $= 42 - 126 = ACh = 172$ (în interpretarea fără semn) = -84 (în interpretarea ;cu semn)

Această scădere setează flag-urile astfel:

SF = 1 (deoarece bitul de semn pentru rezultatul ACh = 10101100b este 1)

CF = 1 (deoarece $|42| < |126|$ se pune problema unei scăderi cu împrumut de cifră ; în interpretarea fără semn scăderea devine $42 - 126 = 172$ (!) provenită de fapt din necesitatea unei scăderi de tipul $(256 + 42) - 126 = 172$ și ca urmare a necesității împrumutului se va semnală depășire în interpretarea fără semn prin setarea flagului carry)

OF = 0 (se efectuează scăderea în interpretarea cu semn, adică $bh - ah = 42 - 126 = -84$ și

cum $-84 \in [-128..127]$ NU se semnalează *signed overflow* și ca urmare OF=0)

Ca regulă generală să observăm că din punctul de vedere al reprezentării binare, dacă rezultatul scăderii $a - b \in [-127..127]$ atunci și $b - a \in [-127..127]$ (situația particulară în care $a - b = -128$ o tratăm mai jos). Analog pentru reprezentări de tip cuvânt la nivelul intervalului $[-32767..32767]$ cu discuție asupra cazului particular -32768. Ca urmare se poate concluziona faptul că instrucțiunile **cmp a,b** și **cmp b,a** vor furniza întotdeauna aceeași valoare pentru OF.

iii). - discuție asupra cazurilor **cmp 80h,0** și **cmp 0,80h**

mov ah,80h ;80h = 128 (interpretarea fără semn) = -128 (interpretarea cu semn)

; = 10000000b (bitul de semn fiind 1 cele două interpretări diferă)

mov bh,0 ;bh:=0

cmp ah,bh ;se realizează scăderea fictivă $ah - bh = 10000000b - 00000000b = 10000000b$

; = 80h = 128 (interpretarea fără semn) = -128 (interpretarea cu semn)

Această scădere setează flag-urile astfel:

SF = 1 (deoarece bitul de semn pentru rezultatul 80h = 10000000b este 1)

CF = 0 (deoarece $|80h| > |0|$ nu se pune problema unei scăderi cu împrumut de cifră, deci nu

poate fi vorba despre depășire în interpretarea fără semn)

OF = 0 (se efectuează scăderea în interpretarea cu semn, adică $ah - bh = -128 - 0 = -128$ și

cum $-128 \in [-128..127]$ NU se semnalează *signed overflow* și ca urmare OF=0)

`cmp bh,ah` ;se realizează scăderea fictivă $bh - ah = 00000000b - 10000000b = 10000000b$

; $= 80h = 128$ (interpretarea fără semn) = -128 (interpretarea cu semn)

Această scădere setează flag-urile astfel:

SF = 1 (deoarece bitul de semn pentru rezultatul $80h = 10000000b$ este 1)

CF = 1 (deoarece $|0h| < |80h|$ se pune problema unei scăderi cu împrumut de cifră; în interpretarea fără semn scăderea devine $0 - 128 = 128$ (!) provenită de fapt din necesitatea unei scăderi de tipul $(256 + 0) - 128 = 128$ și ca urmare a necesității împrumutului se va semnaliza depășire în interpretarea fără semn prin setarea flagului carry)

OF = 1 (se efectuează scăderea în interpretarea cu semn, adică $bh - ah = 0 - (-128) = +128$ și

cum $+128 \notin [-128..127]$ se semnalează *signed overflow* și ca urmare OF=1)

CF = 1 în cazul `cmp 0,80h` deoarece se efectuează o scădere cu împrumut de tipul :

$$\begin{array}{r} 0 - 10000000b = 1\ 00000000 - \\ \underline{10000000} \\ 010000000 \end{array}$$

și cifra de împrumut se transferă în CF.

Să analizăm în acest context ce înseamnă și cum s-a ajuns la domeniul “numerelor cu semn posibil a fi reprezentate pe 1 octet” respectiv domeniul “numerelor cu semn posibil a fi reprezentate pe 1 cuvânt”.

Pe 1 octet se pot reprezenta 256 de valori, indiferent că vorbim despre interpretarea cu semn sau interpretarea fără semn. În interpretarea fără semn aceste valori sunt cele din intervalul $[0..255]$. Care sunt însă cele 256 de valori reprezentabile în interpretarea cu semn ? Este vorba despre intervalul $[-128..127]$ sau despre intervalul $[-127..128]$? Pentru că nu poate fi vorba despre intervalul $[-128..128]$ deoarece în acest interval sunt 257 de valori ! Cu alte cuvinte cineva a trebuit să aleagă una dintre cele două variante și totodată să facă precizarea că numerele -128 și +128 nu pot coexista între limitele aceluiași interval de reprezentare al aceluiași tip de dată! (reamintim că în limbaj de asamblare *tip de dată = dimensiune de reprezentare*)

În acest sens este de observat și impactul acestui mod de reprezentare asupra limbajelor de nivel înalt: de exemplu atât **shortint** cât și **byte** în Turbo Pascal acceptă valoarea 80h (-128 ca *shortint* și $+128$ ca *byte*) însă 80h nu poate avea două interpretări distincte în

cadrul aceluiași tip de dată ! Nu vom întâlni la nivelul nici unui limbaj de programare de nivel înalt valorile -128 și +128 ca fiind prezente în cadrul aceluiași tip de dată !

Ca urmare, s-a luat decizia ca intervalul acceptat al valorilor cu semn reprezentabile pe 1 octet să fie intervalul [-128..+127] (care este exact domeniul de valori și a tipului de dată **shortint** din Turbo Pascal): deci **+128 nu este acceptat ca valoare cu semn reprezentabilă pe 1 octet !**

Totuși, după cum putem verifica foarte ușor, instrucțiunile **mov ah, 128** și **mov ah, -128** sunt amândouă acceptate de către asamblor, efectul fiind în ambele cazuri încărcarea în *ah* a configurației binare 10000000b ! Aceasta deoarece în primul caz va fi vorba de fapt despre interpretarea fără semn pentru 80h iar în al doilea caz va fi vorba despre interpretarea cu semn. Simpla încărcare a unui registru cu o anumită configurație binară nu presupune și necesitatea interpretării respectivei configurații într-un anumit fel. Sarcina interpretării acelei configurații drept cu semn sau fără semn va cădea în sarcina instrucțiunilor ce urmează și care vor folosi ca operanzi aceste valori. De exemplu, utilizarea lui IMUL în loc de MUL va provoca interpretarea configurației binare respective drept un operand cu semn în loc de unul fără semn. Analog, utilizarea lui DIV în loc de IDIV va provoca interpretarea aceluiași operand ca fără semn ș.a.m.d.

În cazul **cmp 80h,0** se efectuează $80h - 0 = 80h = 10000000b$ ($128 - 0 = 128$ în interpretarea fără semn) fără a fi nevoie de o cifră de transport împrumutată pentru a putea efectua scăderea, deci nu avem depășire în interpretarea fără semn și astfel CF = 0. În interpretarea cu semn a operanzilor și a rezultatului final avem $-128 - 0 = -128 \in [-128..127]$ deci nu avem depășire nici în interpretarea cu semn și astfel OF = 0.

Pe de altă parte, avem evident în ambele cazuri SF=1. Justificarea *intuitivă*: în interpretarea cu semn valoarea 10000000b reprezintă un număr strict negativ adică -128. Justificarea *tehnică*: bitul de semn al reprezentării binare 10000000b este 1 deci SF=1.

iv). Să analizăm în continuare modurile în care putem compara valorile 0 și 1 (și apoi 0 și -1) și ce efecte are asupra flagurilor instrucțiunea **cmp** în fiecare dintre situații.

Situația **cmp 1,0** (evidențiată la nivelul unui text sursă de exemplu prin **cmp ah,0** cu **ah=1**) va efectua scăderea fictivă $1 - 0 = 1 = 00000001b$. Efectul asupra flag-urilor va fi CF = SF = OF = ZF = PF = AF = 0. Justificările sunt evidente pe baza discuțiilor din exemplele anterioare.

Situația **cmp 0,1** (evidențiată la nivelul unui text sursă de exemplu prin **cmp ah,1** cu **ah=0**) va efectua scăderea fictivă $0 - 1 = -1 = 11111111b$:

$$\begin{array}{rcl} 0 - 00000001b & = & \begin{array}{r} 1\ 00000000 \\ -\ 00000001 \\ \hline 0\ 11111111 \end{array} \end{array}$$

Efectul asupra flag-urilor va fi $CF = SF = PF = AF = 1$ și $ZF = OF = 0$. Justificarea valorilor din CF și SF este și aici evidentă pe baza discuțiilor din exemplele anterioare iar $OF=0$ deoarece rezultatul în interpretarea cu semn este -1, iar $-1 \in [-128..127]$.

Situația **cmp -1,0** (evidențiată la nivelul unui text sursă de exemplu prin **cmp ah,0** cu **ah = -1**) va efectua scăderea fictivă $-1 - 0 = -1 = 11111111b$. Efectul asupra flag-urilor va fi $SF = PF = 1$ și $CF = OF = ZF = AF = 0$. $SF=1$ deoarece bitul de semn este 1. $OF=0$ deoarece rezultatul în interpretarea cu semn este -1, iar $-1 \in [-128..127]$. $CF=0$ deoarece nu se impune efectuarea unei scăderi cu împrumut.

Situația **cmp 0,-1** (evidențiată la nivelul unui text sursă de exemplu prin **cmp ah,-1** cu **ah = 0**) va efectua scăderea fictivă $0 - (-1) = +1 = 00000001b$:

$$\begin{array}{r} 0 - 11111111b = 1\ 00000000 - \\ \underline{11111111} \\ 0\ 00000001 \end{array}$$

Efectul asupra flag-urilor va fi $CF = AF = 1$ și $OF = SF = ZF = PF = 0$. $SF = 0$ deoarece bitul de semn este 0. $OF=0$ deoarece $0 - (-1) = +1 \in [-128..127]$. $CF = 1$ deoarece se impune efectuarea unei scăderi cu împrumut. Putem justifica și așa: în interpretarea fără semn această scădere înseamnă de fapt $0 - 255 = 1$ (!), care trebuie justificată prin $(256+0) - 255 = 1$, deci e nevoie de cifră de împrumut și astfel se semnalează depășire în cazul interpretării fără semn, deci $CF = 1$.

v). Cazurile studiate anterior (i-iv) s-au referit la operații de scădere datorită analizei pe care am avut-o în vedere asupra efectelor instrucțiunii **cmp**. Să analizăm în continuare și cazul unei depășiri furnizate de operația de adunare revenind astfel la discuția asupra aplicării regulii RDA:

mov ah,126 ; 126 = 01111110b = 7eh (aceeași valoare 126 în ambele interpretări)
add ah, 2 ; 2 = 2h = 00000010b ; AH := 01111110b + 00000010b = 7eh + 02h
 =
 ; 10000000b = 80h (= 128 fără semn = -128 cu semn)
 CF = 0 deoarece: $\begin{array}{r} 01111110 + \\ \underline{00000010} \\ 10000000 \end{array}$ - nu există transport în afara spațiului de reprezentare
 al rez.

$SF = 1$ deoarece bitul de semn al rezultatului este 1 (în interpretarea cu semn rezultatul operației efectuate este strict negativ = -128).

$OF = 1$ deoarece:

- justificare *tehnică* - conform RDA se adună două numere de același semn (bitul de semn este 0 pentru amândouă) iar rezultatul este de semn diferit (bitul de semn este 1).

- justificare *intuitivă* - adunăm două numere fără semn a căror sumă este $126 + 2 = 128$.
Însă numărul $+128 \notin [-128..127]$ deci se semnalează *signed overflow* și ca urmare $OF=1$.

vi). Unul dintre efectele surprinzătoare ale interpretărilor cu semn sau fără semn se referă la situația în care programatorul își inițializează operanzii cu anumite valori inițiale dorite (cu semn sau fără semn, conform necesităților problemei în cauză) și se așteaptă la obținerea unor rezultate sau reacții în conformitate cu valorile furnizate. Atenție însă! De obicei aceste valori au o dublă interpretare posibilă și nu vor fi interpretate în orice situație sub forma furnizată la inițializare!

Utilizarea ulterioară a unor instrucțiuni care forțează prin modul de lor de acțiune interpretarea complementară (cu semn/fără semn) celei de la inițializare poate provoca apariția unor situații în care un utilizator la prima vedere fie să suspecteze erori din partea asamblorului (!) fie din punct de vedere al exprimării în baza 10 să se ajungă la interpretări hilare... Aceasta se întâmplă dacă nu se ține cont în permanență de dubla interpretare posibilă a configurațiilor binare manipulate. Să luăm un exemplu:

```
mov al, 200 ; al = 11001000b = 0C8h = 200 (fără semn) = -56 (cu semn)
mov bl, -1  ; bl = 11111111b = 0FFh = 255 (fără semn) = -1 (cu semn)
cmp al, bl  ; al-bl = 11001001b = C9h = -55 (cu semn) = 201 (fără semn)
              (și se setează corespunzător OF=ZF=0 și CF=SF=1)
```

Deci pe cine comparăm de fapt aici? Pe 200 cu -1 așa cum precizează valorile de la inițializare?

Sau poate pe 200 cu 255? Sau pe -56 cu -1 ? Sau pe -56 cu 255?

Răspuns: comparăm întotdeauna pe 0C8h cu 0FFh sau în exprimare binară pe 11001000 cu 11111111. Efectul va fi unul singur: afectarea corespunzătoare a flag-urilor în urma efectuării scăderii fictive AL-BL. Modul de exprimare corect al comparației efectuate în baza 10 nu este dedus din acțiunea instrucțiunii CMP (care nu distinge absolut de loc între cele 4 variante posibile de comparare de mai sus) ci pe baza unor eventuale instrucțiuni ulterioare care vor avea ele rolul de a interpreta în unul din cele 4 moduri de mai sus comparația efectuată. Să urmărim în acest sens variantele de comparare de mai jos identificate prin utilizarea instrucțiunilor corespunzătoare de salt condiționat:

jl et1 ; evident că $200 \nless -1$ deci la prima vedere pare că nu este îndeplinită condiția necesară pentru efectuarea saltului... să nu uităm însă faptul că JL (Jump If Less) interpretează rezultatul comparației ca fiind cu semn (deci -55) aceasta însemnând implicit și faptul că scăderea este interpretată ca $(-56 - (-1))$ deci și operanzii vor fi amândoi interpretați cu semn... cum $-56 < -1$ iată că și intuitiv condiția se verifică (pe lângă justificarea tehnică a îndeplinirii condiției de salt $SF \neq OF$) și deci saltul se va efectua ! Deci chiar dacă programatorul a furnizat la inițializare valorile 200 și -1, utilizarea instrucțiunii JL a provocat interpretarea comparației ca fiind între -55 și -1 și nu între 200 și -1! (explicația de aici și faptul că saltul se va efectua vă poate ajuta să “demonstrați” unor colegi cum 200 poate fi mai mic decât -1 !!!)

ja et2 ; deoarece $200 > -1$ în acest caz ne-am aștepta ca saltul să se efectueze... însă utilizarea instrucțiunii JA (Jump if Above) impune interpretarea fără semn, deci varianta de comparație corectă aici este comparația lui 200 cu 255 și cum $200 \not> 255$ condiția nu este îndeplinită și deci saltul nu se va efectua (iată deci cum se poate “demonstra” că 200 nu este superior valorii -1 !!!). Ca o confirmare, se poate vedea că nici condiția tehnică impusă de JA nu este îndeplinită: ar trebui să avem $CF=ZF=0$, însă în cazul nostru $CF=1$ deci saltul nu se va efectua.

jb et3 ; intuitiv $200 < 255$, iar tehnic $CF=1$ deci saltul se efectuează

jg et4 ; intuitiv $-56 \not> -1$, iar tehnic deși $ZF=0$ nu este îndeplinită și condiția $SF = OF$ deci
; saltul nu se va efectua

Ca urmare din cele 4 situații teoretic posibile de mai sus, vom întâlni concret numai două:

- comparație fără semn (200 cu 255) - impusă de “above” sau “below”
- comparație cu semn (-56 cu -1) – impusă de “less than” sau “greater than”

Nu putem așadar compara de fapt pe 200 cu -1 așa cum au fost specificate valorile la inițializare și nici pe -56 cu 255 deoarece **interpretarea este ori cu semn ori fără semn pentru ambii operanzi!**

vii). Am studiat în exemplele anterioare modalitatea de reacție (de interpretare) a procesorului 80x86 legată de noțiunea de depășire în cazul operațiilor de adunare și de scădere. Când și cum semnalează însă procesoarele din familia 80x86 depășirea la înmulțire și respectiv la împărțire ?

“Depășirea” la înmulțire. Instrucțiunile MUL și IMUL setează $CF=1$ și $OF=1$ dacă “jumătatea” superioară a produsului (octetul superior dacă este vorba despre produs-cuvânt sau cuvântul superior dacă este vorba despre produs-dublucuvânt) este o valoare diferită de zero. Aceasta este definiția noțiunii de “depășire la înmulțire” în cazul arhitecturii 80x86. Să remarcăm faptul că nu se face distincție între MUL și IMUL și de aceea nici între CF și OF. Ori vor fi amândouă flag-urile setate la valoarea 1 cu semnificația de “depășire la înmulțire” în sensul precizat mai sus, ori vor primi amândouă valoarea 0. Iată un exemplu pe 8 biți:

```
mov al, 5
mov bl, 170
mul bl      ;AX := AL * BL = 5 * 170 = 850 = 0352h și vom avea CF=1 și
OF=1        ;deoarece octetul superior AH = 03 ≠ 0.
```

Varianta cu IMUL va furniza:

```
mov al, 5
mov bl, 170 ;170 = 0aah = -86 în interpretarea cu semn
imul bl     ;AX := AL * BL = 5 * (-86) = - 430 = 0fe52h și vom avea CF=1 și
```

;OF=1 deoarece octetul superior AH = 0feh ≠ 0.

În cazul unor operanzi pe 16 biți putem avea de exemplu:

val1 DW 2000h

val2 DW 0100h

...

mov ax, val1

mul val2 ;DX:AX = 00200000h și vom avea CF=1 și OF=1 deoarece jumătatea ;superioară a produsului DX:AX, adică registrul DX conține valoarea 0020h ≠ 0.

Aceste setări nu trebuie să le interpretăm drept erori. Nu este în nici un caz vorba despre o potențială pierdere de informație ca și în cazul celorlalte depășiri - adunare, scădere sau împărțire. Aceasta deoarece chiar dacă înmulțim valorile maxime posibil a fi reprezentate pe dimensiunea operanzilor ($255 * 255$ pentru octeți și respectiv $65535 * 65535$ pentru cuvinte) tot nu se depășește dublul dimensiunii de reprezentare a operanzilor, adică spațiul pe care îl avem oricum la dispoziție prin definiție, deoarece $255 * 255 = 65025 < 65535$ (numărul maximal fără semn reprezentabil pe un cuvânt) iar $65535 * 65535 = 4\,294\,836\,225 < 4\,294\,967\,295$ (numărul maximal fără semn reprezentabil pe un dublucuvânt).

În cazul înmulțirii cu semn (instrucțiunea IMUL) justificarea este similară: $127 * 127 = 16129 < 32767$ (numărul maximal cu semn ce poate fi reprezentat pe 1 cuvânt), iar $32767 * 32767 = 1\,073\,676\,289 < 2\,147\,483\,647$ (numărul maximal cu semn reprezentabil pe un dublucuvânt).

Depășirea în cazul înmulțirii la nivelul limbajului de asamblare 80x86 este doar o semnalare a faptului că plecându-se de la operanzi octeți (respectiv cuvinte) produsul nu încapă tot într-un octet (respectiv într-un cuvânt) ci este realmente nevoie de o dimensiune dublă pentru memorarea rezultatului. În acest sens, a se vedea și capitolul 1, în care din punct de vedere matematic s-a specificat clar că înmulțirea nu provoacă de fapt depășire, tocmai din cauza alocării unui spațiu suficient pentru reprezentarea produsului. În concluzie, se poate spune că din punct de vedere matematic singura operație care nu provoacă depășire este înmulțirea, însă procesoarele 80x86 promovează totuși noțiunea de “depășire la înmulțire” pentru a diferenția între situațiile în care produsul încapă într-un spațiu de dimensiunea operanzilor și în care nu.

Situațiile în care produsul încapă pe dimensiunea operanzilor vor fi caracterizate de setările CF = OF = 0 (nu avem deci depășire la înmulțire). Iată un exemplu:

mov al, 5

mov bl, 51

mul bl ; AX := AL * BL = 5 * 51 = 255 = 00ffh și vom avea CF=0 și OF=0 ; deoarece octetul superior AH = 0.

Depășirea la împărțire. În cazul împărțirii, specificarea acestei operații sub forma

(I)DIV *operand*

presupune că operandul specificat este împărțitorul (posibil a fi reprezentat fie pe 8 fie pe 16 biți) iar deîmpărțitul este considerat implicit în AX (dacă *operand* este octet) sau în DX:AX (dacă împărțitorul este cuvânt). Efectuarea operației are ca efect:

AX : operand pe 8 biți = câtul în AL și restul în AH;

DX:AX / operand pe 16 biți = câtul în AX și restul în DX;

În cazul împărțirii depășirea apare atunci când rezultatul împărțirii nu încapă în spațiul rezervat conform definiției pentru reprezentare, mai exact, când câtul nu încapă în AL sau respectiv AX. Într-o astfel de situație, procesorul 80x86 emite o întrerupere 0, execuția terminându-se cu un mesaj furnizat de către rutina de tratare a întreruperii 0, de genul “Divide by zero”, “Zero divide” sau “Divide overflow” (în funcție de tipul de procesor și/sau de SO instalat). Pare ciudat la prima vedere că o împărțire prin 0 (de genul *div bh* cu *bh* = 0) ce practic nu se poate efectua din punct de vedere matematic este tratată similar ca efect din punct de vedere al limbajului de asamblare cu o împărțire care matematic se poate efectua. Secvența

```
mov ax,60000
mov bl,2
div bl
```

ar trebui să furnizeze din punct de vedere matematic câtul 30000. Însă conform definiției împărțirii DIV acest cât trebuie memorat în registrul AL, de dimensiune octet. Cum cea mai mare valoare reprezentabilă pe 1 octet este 255, este evident astfel că din punct de vedere al limbajului de asamblare împărțirea de mai sus nu se poate nici ea efectua (similar cu o situație de tip *div 0*) și ca urmare înțelegem acum decizia proiectanților de a trata tot prin emiterea unei întreruperi 0 și o situație de genul celei de mai sus. Să remarcăm în acest sens și faptul că mesajul “*Divide overflow*” (depășire la împărțire) este acceptat în acest context ca similar unui “*Divide by zero*”.

viii). Una dintre erorile logice frecvente pe care o fac programatorii neexperimentați este de a confunda exprimările “*numere cu semn*” și “*numere fără semn*” cu exprimările “*numere negative*” și respectiv “*numere pozitive*”. Numere cu semn nu înseamnă automat numere negative ! Numerele cu semn sunt fie pozitive, fie negative. Numerele fără semn sunt întotdeauna pozitive.

Ce concluzii vom trage relativ la modul de interpretare (cu semn sau fără semn) din enunțul unei probleme care cere efectuarea unei anumite acțiuni “*dacă numărul v este (strict) negativ*”? În primul rând vom concluziona că este vorba despre interpretarea cu semn. Se pune însă întrebarea: cum vom testa practic dacă un număr cu semn este negativ sau nu? (să presupunem că v este octet). Fiind vorba despre interpretarea cu semn, dacă primul bit al configurației binare este 1 atunci numărul este negativ. Deci totul se reduce la un test asupra

primului bit din reprezentarea numărului. Iată două alternative pentru realizarea unui astfel de test:

a). Realizăm o deplasare a primului bit în CF și testăm valoarea sa printr-o instrucțiune adecvată de salt condiționat. Secvența

```
mov al,v      ;pentru a nu afecta destructiv conținutul variabilei v
shl al,1      ;shift stânga cu 1 poziție pentru ca primul bit să treacă în CF.
jc este_negativ ;dacă CF=1 atunci salt la eticheta este_negativ
```

asigură testarea faptului dacă variabila *v* este sau nu un număr negativ.

b). Utilizăm instrucțiunea **cmp** pentru o comparație în raport cu 0:

```
cmp v,0      ;scădere fictivă v-0
jl este_negativ ;dacă v<0 atunci salt la eticheta este_negativ
```

sau alternativ

```
cmp 0,v      ; scădere fictivă 0-v
jg este_negativ ;dacă 0>v atunci salt la eticheta este_negativ
```

ix). Am văzut ca la nivelul efectuării operațiilor de adunare sau scădere procesorul 80x86 nu diferențiază între adunări/scăderi cu semn sau fără semn (tehnice vorbind ele se efectuează drept operații binare cu rezultat interpretabil **ulterior** drept cu semn sau fără). Totuși, în momentul în care se pune problema exprimării în baza 10 a unei operații de adunare sau scădere ne punem întrebarea: cum să exprimăm semantic corect operanzii operației respective pentru ca aceste exprimări să fie consistente cu interpretarea rezultatului final obținut ? Mai concret:

	00000101 +	(= 5 în ambele interpretări)
	<u>11111110</u>	(= 254 fără semn și -2 în interpretarea cu semn)
(1)	00000011	(= 3 în ambele interpretări ale configurației pe 8 biți)

reprezintă $5 + 254 = 259$ (= 1 00000011 – configurație pe 9 biți !) sau reprezintă $5 + (-2) = 3$? După cum vom vedea și aici răspunsul este că putem interpreta în ambele moduri și să justificăm astfel ca două reacții separate modul de setare al flag-urilor CF și respectiv OF.

Datorită cifrei de transport vom avea CF=1 (independent de interpretarea operanzilor sau a rezultatului final drept cu semn sau fără semn, deoarece este vorba despre o consecință tehnică a modului de efectuare a operației binare de adunare). Ca urmare în interpretarea fără semn avem depășire (evident, deoarece $259 > 255$, adică decât numărul maxim reprezentabil pe 1 octet).

Ce se întâmplă cu OF ? Rularea secvenței

```
mov al, 5      ; = 5 în ambele interpretări  
mov bl, 254    ; = -2 în interpretarea cu semn  
add al, bl     ; AL := AL+BL = 5+(-2) = 3
```

nu setează flagul OF la valoarea 1, deci situația de mai sus nu este considerată “depășire” în interpretarea cu semn! Din punct de vedere al justificării modului de setare a flag-ului OF secvența de mai sus ar fi mai corectă dacă ar fi scrisă:

```
mov al, 5  
mov bl, -2  
add al, bl     ; deci 5 + (-2) = 3
```

și este evident că în această interpretare nu este vorba despre nici o depășire (și de aceea și OF = 0).

Să ne reamintim în acest context și exemplele date la prezentarea RDA și RDS de la paginile ??-??: adunarea $100 + 50 = 150$ va semnala depășire (*signed overflow* - conform RDA), iar scăderile $130 - 42$ (interpretată ca $-126 - 42 = -168 \notin [-128..127]$) și $42 - 130$ (interpretată ca scăderea $42 - (-126) = +168 \notin [-128..127]$) produc la rândul lor *signed overflow* și ca urmare OF=1.