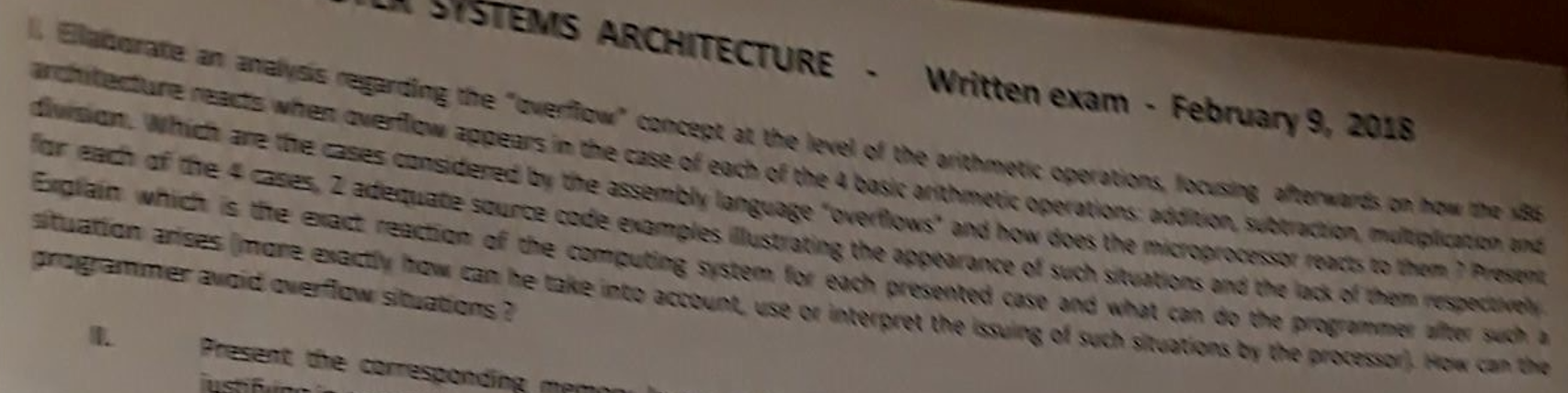
**Conceptul de Overflow**

**

O depășire este o situație matematică ce exprimă faptul că rezultatul unei operații nu a încăput în spațiul rezervat acesteia. Pentru procesor o depășire reprezintă și faptul că o operație efectuată este un non-sens matematic în respectiva interpretare și nu poate fi acceptată ca fiind o operație matematică corectă. Procesorul x86 marchează o depășire prin 2 flag-uri, și anume Carry Flag (pentru depășirea în cazul interpretării fără semn) și Overflow Flag (pentru depășirea în cazul interpretării cu semn).

**Adunarea** **(ADD oper1 (n biți), oper2 (n biți))**

**Interpretare fără semn**: Vom avea depășire în cazul în care rezultatul adunării celor doua configurații de n biți va fi o configurație de n + 1 biți (iar în oper1 se vor salva doar biții de la bitul 0 la bitul n, fără bitul cel mai semnificativ). În acest caz CF va primi valoarea 1, în caz contrar va primi valoarea 0.  
 **mov AL, 1000 0000b // = 128  
 mov BL, 1000 0001b // = 129  
 add AL, BL** // 1000 0000b + 1000 0001b = 1 0000 0001b, dar cum byte + byte = byte, în AL se vor păstra ultimi 8 biți (0000 0001b = 1) și se va marca depășirea, CF devenind 1. Observăm că în baza 10 operația este 128 + 129 = 1 care nu are niciun sens.  
 **mov AL, 1000 0000b // = 128  
 mov BL, 0100 0000b // = 64  
 add AL, BL** // AL 🡨 1100 0000b = 192 care poate fi reprezentat pe 8 biți, deci CF va deveni 0. Observăm că în baza 10 operația este 128 + 64 = 192 care este o operație corectă

**Interpretarea cu semn**: Vom avea depășire doar în care cei doi operanzi au același bit de semn (bitul cel mai semnificativ), iar bitul de semn al rezultatului are valoarea bitului de semn complementară cu valoarea bitului de semn al operanzilor. (1... + 1... = 0... sau 0... + 0... = 1...)  
 **mov AL, 0100 0000b // = 64  
 mov BL, 0100 0001b // = 65  
 add AL, BL** // 0100 0000b + 0100 0001b = 1000 0001b = -127, bitul de semn al celor doi operanzi este 0, iar cel al rezultatului este 1, deci OF va deveni 1. Observăm că în baza 10 operația este 64 + 65 = -127 care nu are niciun sens.  
 **mov AL, 1000 0000b // = -128  
 mov BL, 0100 0000b // = 64  
 add AL, BL** // 1000 0000b + 0100 0000b = 1100 0000b = -64, biții de semn al celor doi operanzi sunt diferiți deci OF va fi setat la 0. Observăm că în baza 10 operația este -128 + 64 = -64 care are sens.

**Scăderea (SUB oper1 (n biți), oper2 (n biți))**

**Interpretare fără semn**: Vom avea depășire în cazul în care va trebui să ne împrumutăm cu un 1 pentru a putea efectua corect scăderea, cu alte cuvinte dacă oper1 este mai mic decât oper2. CF va fi marcat cu 1.  
 **mov AL, 0000 0000b // = 0  
 mov BL, 1111 1111b // = 255  
 sub AL, BL** // **(1)** 0000 0000b – 1111 1111b = 0000 0001b = 1, este nevoie de împrumut deci CF va fi setat la 1. Observăm că în baza 10 operația este 0 – 255 = 1 care nu are niciun sens matematic.  
 mov AL, 1000 0000b // = 128  
 mov BL, 0100 0000b // = 64  
 sub AL, BL // 1000 0000b – 0100 0000b = 0100 0000b = 64, nu este nevoie de împrumut deci CF va fi setat la 0. Observăm că în baza 10 operația este 128 – 64 = 64 care este o operație corectă.

**Interpretarea cu semn**: Vom avea depășire în cazul în care dacă biții de semn ai celor 2 operanzi diferă și bitul de semn al rezultatului coincide cu bitul de semn al celui de-al doilea operand. OF va fi marcat cu 1.  
 **mov AL, 1000 0001b // = -127  
 mov BL, 0100 0001b // = 65  
 sub AL, BL** // 1000 0001b – 0100 0001b = 0100 0000b = 64. Observăm că în baza 10 operația este -127 – 65 = 64 care nu are niciun sens matematic, OF va fi marcat cu 1.  
 **mov AL, 1100 0000b // = -64  
 mov BL, 0010 0000b // = 32  
 sub AL, BL** // 1100 0000b – 0010 0000b = 1010 0000b = -96, deoarece, deși biții de semn ai operanzilor alternează, bitul de semn al rezultatului coincide cu bitul de semn al primului operand, deci OF va fi setat la 0. Observăm că în baza 10 operația este -64 – 32 = -96 care este o operație corectă.

**Înmulțirea (MUL/IMUL oper1 (n biți))**

Pentru înmulțire nu există o depășire propriu zisă, în NASM rezultatul fiind amplasat întotdeauna pe un registru cu dimensiunea dublă față de operandul explicit (AX pentru n = 8, DX:AX pentru n = 16, EAX:EDX pentru n = 32) care va fi înmulțit cu registrul corespunzător (AL pentru n = 8, AX pentru n = 16, EAX pentru n = 32), iar dacă sunt înmulțite două configurații de n biți rezultatul va avea întotdeauna dimensiunea maximă de 2 \* n biți. Dar, pentru ca flag-urile CF și OF să nu rămână nefolosite, s-a găsit o soluție: în cazul în care rezultatul dintre înmulțirea operandului explicit (oper1) cu operandul implicit este o configurație de biți ce încape pe n biți, atunci CF și OF vor fi setate la 0, în caz contrar CF și OF vor fi setate la 1.

**Împărțirea (DIV/IDIV oper1 (n biți))**

Pentru împărțire conceptul de overflow este cel mai grav aplicat. În NASM, dacă operandul explicit (care este împărțitorul) are n biți atunci operandul implicit (care este deîmpărțitul) are 2\*n biți (AX pentru n = 8, DX:AX pentru n = 16, EDX:EAX pentru n = 32) iar rezultatele vor fi salvate în 2 operanzi a câte n biți, și anume pentru cât AL pentru n = 8, AX pentru n = 16 și EAX pentru n = 32 și pentru rest AH pentru n = 8, DX pentru n = 16 și EDX pentru n = 32. În cazul în care rezultatul încape în spațiul alocat, atunci nu vom avea depășire și flag-urile vor rămâne nemodificate, în cazul în care rezultatul nu încape atunci programul va transmite unul dintre următoarele mesaje de eroare: “Division Overflow”, “Divide by Zero” și va crapa, nu ne intereseaza ce se află în flag-uri.

La înmulțire programatorul nu este nevoit să evite situațiile de depășire deoarece limbajul de asamblare reține rezultatul într-un spatiu dublu operandului. O metodă prin care programatorul poate evita situațiile de depășire este să folosească registrii cu spațiu mai mare decât dimensiunea maximă a operanzilor cu care lucrează (de exemplu dacă vrea să facă o operație cu 2 octeți, să lucreze cu cuvinte). Dacă nu este posibil, NASM pune la dispoziție mai multe instrucțiuni, precum **ADC** (Add with Carry), **SBB** (Subtract with Borrow) și **Jcc** (Jump if condition is true), astfel încât dacă vrea să adune de exemplu 2 dublucuvinte (EDX:EAX + ECX:EBX) în interpretarea fără semn și să afișeze rezultatul, programatorul poate scrie următoarea secvență de instrucțiuni:  
**segment data  
 a dq 0  
 formatnotcarry db ‘%llu’, 0  
 formatcarry db ‘1%llu’, 0  
segment code  
 start:  
 . . . . . . . . . .  
 add EAX, EBX  
 adc EDX, ECX  
 jc afisare\_carry  
 push EAX  
 push EDX  
 push dword formatnotcarry  
 call [printf]  
 jmp continua  
 afisare\_carry:  
 push EAX  
 push EDX  
 push dword formatcarry  
 call [printf]  
 continua:  
 . . . . . . . . . . . .**