PMD – Lucrarea 2

Această lucrare de laborator își propune să introducă instrucțiunile de înmulțire și împărțire, instrucțiunile de deplasare, precum și instrucțiunile de salt condiționat și necondiționat. Acestea din urmă sunt folosite pentru a ilustra implementarea în asamblare a unor bucle de tip while, do while și for.

2.1 Instrucțiunile de înmulțire și împărțire

Operațiile de înmulțire și împărțire se pot face cu operanzi pe 1, 2 sau 4 octeți. Instrucțiunile folosite au un singur operand. Deînmulțitul / deîmpărțitul se află implicit în registrul acumulator (pe 8 biți în AL, pe 16 biți în AX sau pe 32 de biți în EAX), în timp ce celălalt operand poate fi într-un registru sau memorie [3]..

MUL	Operand pe 1, 2 sau 4 octeți	AX = AL * operand	- Înmulțire fără semn
IMUL		DX:AX = AX * operand EDX:EAX=EAX * operand	- Înmulțire de întregi cu semn

În cazul înmulțirii fără semn a unor operanzi pe câte 8 biți, comportamentul implicit, hardcodat, al instrucțiunii **MUL operand** este AX = AL * operand, motiv pentru care trebuie să ne asigurăm că unul dintre operanzi se află deja în AL înainte de a apela instrucțiunea **MUL operand**. Rezultatul înmulțirii poate fi de lungime dublă față de operanzi, motiv pentru care este stocat în AX, pe 16 biți.

În cazul înmulțirii cu semn a unor operanzi pe 16 biți, comportamentul implicit, hardcodat, al instrucțiunii IMUL operand este DX:AX = AX * operand, ceea ce înseamnă că unul dintre operanzi trebuie să se afle în AX înaintea apelării instrucțiunii, iar rezultatul poate avea până la 32 de biți, motiv pentru care este stocat în registrul de 32 de biți format prin concatenarea lui DX cu AX, adică cei 16 biți mai semnificativi ai rezultatului se vor afla în DX, iar ceilalți 16 biți mai puțin semnificativi vor fi în AX.

Exemplu 1:

```
MOV AL, 1

MOV BL, 7

MUL BL ; procesorul execută operația AX = AL * BL. Rezultatul (valoarea 7 în zecimal) va fi stocat în AX, în acest caz (fiind un număr < 255) ocupând doar jumătatea inferioară a lui AX, adică chiar AL, pe 8 biți. Practic, în urma înmulțirii, rezultatul s-a suprascris peste unul dintre operanzi, în AL.
```

MUL 5 *; AX = AL * 5. În AL aveam valoarea zecimală 7 de la înmulțirea anterioară, iar în urma acestei înmulțiri în AL avem 35 în zecimal.*

Exemplu 2:

MOV AL, X

MUL AL *; în acest caz, în urma apelului instrucțiunii, în AX vom avea valoarea variabilei X ridicată la pătrat; AX = AL * AL.*

MUL AL *; AX = AL * AL, conținutul lui AL se înmulțește din nou cu AL și avem în AX valoarea lui X ridicată la puterea a patra.*

Împărțirea presupune că deîmpărțitul este în registrul acumulator (AX, DX:AX sau EDX:EAX), iar împărțitorul este într-un registru sau în memorie [3]..

DIV	Operand pe 1,2 sau 4 octeți (împărțitor)	AL = AX / operand; AH = AL % operand	- Împărțire fără semn
IDIV		AX = DX:AX / operand; DX = DX:AX % operand	- Împărțire de întregi cu semn

Exemplu:

MOV AX, 17

MOV BL, 4

DIV BL *; AL = AX / BL și AH = AL % BL; în AL vom avea câtul împărțirii lui 17 la 4, adică 4, iar în AH vom avea restul împărțirii lui 17 la 4, adică 1.*

2.2 Registrul FLAG

FLAG	.....NT	OF DF IF TF SF ZF AF PF CF	Flag register

Registrul FLAG conține o serie de indicatori de condiție, numiți și fanioane sau flag-uri, care au următoarea semnificație:

CF (Carry Flag) – se poziționează pe "1" dacă a apărut un transport din sau s-a făcut un împrumut în rangul cel mai semnificativ. De asemenea instrucţiunile de rotire şi deplasare pot poziţiona acest indicator.

PF (Parity Flag) - se poziționează pe "1" dacă s-a obţinut un rezultat pentru care octetul mai puţin semnificativ are un număr par de biţi cu valoarea "1".

AF (Auxiliar Carry) - se poziționează pe "1" dacă în execuţia unei instrucţiuni care poziţionează acest indicator a apărut un transport din rangul 3 spre rangul 4 sau a fost efectuat un împrumut din rangul 4 spre rangul 3. Acest indicator este utilizat pentru operaţiile cu numere BCD.

ZF (Zero Flag) - se poziționează pe "1" dacă s-a obţinut rezultatul zero.

SF (Sign Flag) - se poziționează pe "1" dacă s-a obţinut un rezultat pentru care bitul cel mai semnificativ este "1".

TF (Trace) - este utilizat pentru controlul execuţiei instrucţiunilor în regim pas cu pas în scopul depanării programelor.

IF (Interrupt) - controlează acceptarea semnalelor de întrerupere externă.

DF (Direction) - indică direcţia de parcurgere a şirurilor de octeţi în cazul instrucţiunilor pe şiruri de octeți. Valoarea "0" indică parcurgerea de la adrese mici spre adrese mari.

OF (Overflow) - se poziționează pe "1" dacă în execuţia unei instrucţiuni aritmetice cu semn a apărut o depăşire.

2.3 Instrucțiunile de deplasare

Instructiunile de deplasare se aplică pe registre de 1, 2 sau 4 octeți [3]..

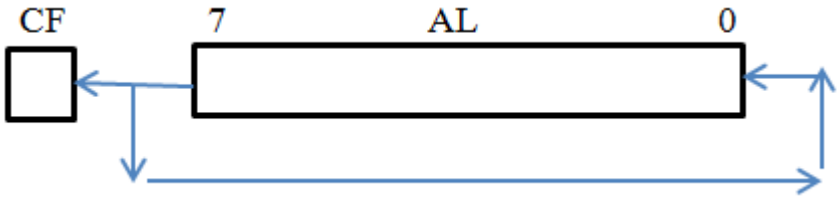
Deplasările pot fi:

- spre stânga sau spre dreapta;
- logice sau aritmetice;
- deschise sau circulare (rotate).

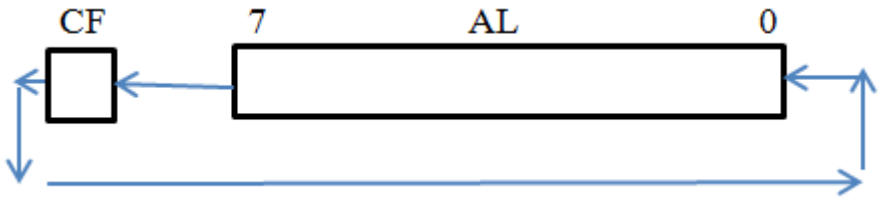
SAL	Reg, 1 Reg, n Reg, CL	Shift Aritmetic Left – deplasare deschisă aritmetică spre stânga
SHL		Shift (logic) Left – deplasare logică spre stânga
SAR		Shift Aritmetic Right – deplasare aritmetică spre dreapta
SHR		Shift (logic) Right
ROL		Rotate (logic) Left – deplasare circulară (rotire) spre stânga
ROR		Rotate (logic) Right – deplasare circulară (rotire) spre dreapta
RCL		Rotate (logic) Left with Carry
RCR		Roate (logic) Right with Carry

Diferența dintre instrucțiunea de rotire simplă spre stânga (ROL) și instrucțiunea de rotire prin carry la stânga (RCL) este ilustrată mai jos:

ROL AL, 1



RCL AL, 1



În cazul instrucțiunii RCL AL, 1, conținutul registrului AL este rotit spre stânga, folosind ca și buffer bitul de carry flag (CF). La rotirea cu un bit spre stânga, conținutul lui CF este copiat peste bitul LSB din AL, bitul MSB din registrul AL este copiat peste CF, iar restul biților se deplasează cu o poziție spre stânga.

Pe de altă parte, în cazul instrucțiunii ROL AL, 1, bitul MSB din registrul AL este copiat direct în poziția bitului LSB din AL, fără a mai trece prin CF.

Exemplu: Dorim sa testăm paritatea unui număr. Pentru aceasta, trebuie verificat bitul LSB al numărului. Daca bitul LSB este 0, numărul este par, iar daca bitul LSB este 1, numărul este impar. O modalitate prin care putem face aceasta verificare este rotirea prin carry spre dreapta cu o poziție. Instrucțiunile de salt condiționat pot verifica carry flag-ul, asa cum veți vedea în paragraful următor.

MOV AL, N *; numărul pentru care doresc să verific paritatea este N*

RCR AL, 1 *; rotesc conținutul registrului AL (în care l-am copiat anterior pe N) cu o poziție spre dreapta, ceea ce determină copierea bitului LSB peste carry flag (CF).*

În continuare, folosind instrucțiunea JC (jump if carry) sau JNC (jump if not carry), pot lua decizii in funcție de paritatea numărului.

2.4 Instrucțiunile de salt

Instrucțiunile de salt pot fi încadrate în două mari categorii: salt necondiționat și salt condiționat. După fiecare instrucțiune aritmetică sau logică, în funcție de valoarea rezultatului, se poziționează indicatorii de condiție din registrul FLAGS, care pot fi testate de instrucțiunile de salt condiționat. Acest mecanism stă la baza construcției instrucțiunilor decizionale în asamblare (similare ca și concept cu *if / else*), precum și a buclelor de program, de tipul *while / do while / for*.

Saltul de tip necondiționat se realizează, în varianta cea mai simplă, prin instrucțiunea JMP urmată de numele etichetei la care se dorește sa se realizeze saltul. Acesta se execută întotdeauna, în momentul în care fluxul secvențial de instrucțiuni executate de procesor ajunge la instrucțiunea JMP, indiferent de rezultatul instrucțiunilor realizate anterior.

Instrucțiunile de salt condiționat au forma generală **Jcond eticheta** și modul lor de execuție este următorul: daca condiția testată prin acel flag este îndeplinită, se face un salt la instrucțiunea indicată de eticheta specificată.

Instrucțiunea de salt condiționat	Condiție testată	Explicații	
JS sau JL	S = 1	Salt la rezultat negativ	Jump on sign / jump if lower
JNS sau JGE	S = 0	Salt la rezultat mai mare sau egal cu 0	Jump if no sign / jump if greater or equal
JZ sau JE	Z = 1	Salt la rezultat egal cu 0	Jump if zero / jump if equal
JNZ sau JNE	Z = 0	Salt la rezultat diferit de zero (pozitiv sau negativ)	Jump if not zero / jump if not equal
JC	C = 1	Salt la transport activ	Jump if carry (poate fi folosit dupa o rotire prin carry)
JNC	C = 0	Salt la transport inactiv	Jump if not carry
LOOP	CX <> 0	Decrementare automata CX (CX = CX – 1) și salt dacă și numai dacă CX <> 0. Poate fi folosită pentru implementarea unei bucle de tip *for*.	

Exemplu: Instrucțiunea JS L1 testează daca flagul de semn (sign flag – SF) este 1, ceea ce înseamnă de fapt că rezultatul ultimei instrucțiuni aritmetice sau logice executate a fost negativ, iar daca SF este 1, procesorul va continua prin execuția instrucțiunii indicate de eticheta L1. În caz contrar, procesorul execută instrucțiunea imediat următoare de dupa JS L1.

```
    MOV AL, 10
    SUB AL, 11 ; în urma scăderii, rezultatul din AL devine negativ și se poziționează SF = 1
    JS L1
    MOV AL, 10
L1: MOV AL, 0
```

În urma execuției acestei porțiuni de cod, scăderea duce la modificarea registrului AL la valoarea -1 și activarea sign flag-ului: SF=1. Instrucțiunea de salt condiționat JS L1 va provoca modificarea fluxului normal de execuție secvențială a instrucțiunilor, se va sări peste instrucțiunea MOV AL, 10 și se va continua direct cu instrucțiunea MOV AL, 0. La finalul secvenței de cod, în registrul AL vom avea valoarea 0.

2.4 Implementarea instrucțiunilor decizionale de tip *if-else* și a buclelor de tip *while, do while, for* în asamblare

2.4.1 If-Else

Presupunem că avem urmatoarea secvență de cod în C:

```
if (N >= 0)
    N--;
else
    N++;
printf("%d", N);
```

Pentru a scrie echivalentul acesteia în limbajul de asamblare x86, putem folosi instrucțiunea CMP op1, op2, care compară cei doi operanzi prin realizarea unei scăderi, fără a suprascrie operandul destinație și poziționează indicatorii de condiție în funcție de valoarea rezultatului.

CMP op1, op2 ⇔ if (op1 – op2 = 0),

{ ZF = 1;

SF = 0; }

Else if (op1 – op2 < 0),

{ ZF = 0;

SF = 1; }

Else

{ ZF = 0;

SF = 0; }

Secvența de mai sus poate fi implementată astfel:

```
MOV AL, N
CMP AL, 0
```

```
            JS LBL_ELSE ; se sare la instrucțiunea indicată de eticheta LBL_ELSE dacă și numai dacă numărul din AL este strict mai mic decât 0

LBL_IF:     DEC N ; instrucțiunile din dreptul etichetei LBL_IF se execută dacă și numai dacă numărul din AL a fost mai mare sau egal cu 0

            MOV AL, N

            JMP FINAL

LBL_ELSE:   INC N ; instrucțiune aritmetică de incrementare

            MOV AL, N

FINAL:      MOV AH, 2

            MOV DL, N

            INT 21H
```

Observăm că secvența de instrucțiuni de pe ramura if-ului (LBL_IF) se încheie cu un salt necondiționat (JMP) la finalul programului. În lipsa acestui salt JMP, în situația în care N>=0, s-ar executa în asamblare atât instrucțiunile de pe *if*, cât și cele de pe *else*.

2.4.2 While

Presupunem că dorim să implementăm în asamblare o bucla simplă de tip while, corespunzătoare următorului cod C:

```
while (N != 0) {
    N--;
}
```

O metodă de implementare presupune utilizarea instrucțiunii de comparare și a unui salt condiționat:

```
            MOV AL, N
LBL_WHILE:  CMP  AL, 0
            JZ FINAL ;condiția de ieșire din buclă este N = 0 (AL = 0 în acest caz)
            DEC N
            MOV AL, N
            JMP LBL_WHILE
```

```
FINAL:      NOP ; no operation - instrucțiune dummy, fără efect
```

2.4.3 For

Presupunem că dorim să implementăm în asamblare o buclă de tip for, pentru care se cunoaște cu exactitate numărul de pași ce urmează a fi executați:

```
For(i=0; i<N; i++) {
        X = X / 2;
}
```

Aceasta poate fi implementată cu instrucțiunea LOOP *eticheta*, care folosește pe post de contor registrul CX și îl decrementează automat la fiecare iterație. Saltul la instrucțiunea indicată de *eticheta* se realizează dacă și numai dacă CX este diferit de 0.

```
        MOV CX, N
LBL_FOR:  MOV AL, X
        SHR AL, 1 ;împărțire la 2 realizată prin deplasare la dreapta cu o poziție
        MOV X, AL
        LOOP LBL_FOR
```

2.5 Întrebări

1. Presupunând că doresc să ridic la pătrat numărul salvat în registrul AL, este corect să scriu instrucțiunea MUL AL, AL? Explicați.

2. Care este secvența de instrucțiuni necesară pentru calcularea cubului numărului salvat în registrul AL?

3. Cum pot sa înmulțesc cu 8 conținutul registrului AL folosind doar o instrucțiune de deplasare?

4. Dacă rezultatul ultimei instrucțiuni artimetice executate de procesor a fost -5, care va fi valoarea flag-urilor ZF și SF? Dar dacă rezultatul a fost 0?

5. Care este instrucțiunea care decrementează registrul CX automat la fiecare pas și poate fi folosită pentru a implementa o structură de tip *for*?

6. În urma execuției instrucțiunii CMP AL, BL, ce flag-uri se vor poziționa pe valoarea 1, dacă AL și BL sunt egale? Dar în situația în care AL < BL?

7. Ce instrucțiuni de salt se pot folosi pentru a sări la eticheta E0 în situația în care rezultatul ultimei instrucțiuni aritmetice executate de procesor a fost zero?

2.6 Probleme

1. Scrieți un program care calculează valoarea următoarei funcții, pentru un număr x pe 8 biți, hardcodat în program (declarat ca variabilă sau constantă în secțiune *.data*). Paritatea numărului se va verifica prin împărțire la 2 și verificarea restului, folosind instrucțiunea DIV.

$$f(x)=\begin{cases} x^2, dacă\ x\ este\ par \\ x^3-1, dacă\ x\ este\ impar \end{cases}$$

2. Scrieți un program care calculează valoarea următoarei funcții, pentru un număr x pe 8 biți, hardcodat în program (declarat ca variabilă sau constantă în secțiune *.data*). Paritatea numărului se va verifica prin utilizarea instrucțiunilor de rotire prin carry și a instrucțiunilor de salt condiționat.

$$f(x)=\begin{cases} x^2+x+5, dacă\ x\ este\ par \\ x^3-x, dacă\ x\ este\ impar \end{cases}$$

3. Scrieți un program care calculează cel mai mare divizor comun (cmmdc) prin scăderi succesive pentru două numere pe câte 8 biți, a și b, hardcodate în program. La final, calculați și cel mai mic multiplu comun (cmmmc) al celor două numere. Algoritmul în C de la care puteți pleca este:

```
while (a != b) {
    If (a > b)
        a = a – b;
    else
        b = b – a;
     }
cmmdc = a;
cmmmc = (a * b) / cmmdc;
```

4. Scrieți un program care calculează primii N termeni ai seriei Fibonacci (0 1 1 2 3 5 …), pentru un număr N hardcodat în program, însă salvează într-un vector doar termenii care sunt numere impare: 1 1 3 5 13 21 55 89 233… .

5. Scrieți un program care consideră 3 numere a, b și c hardcodate, reprezentând coeficienții ecuației de gradul II, $ax^2+bx+c=0$, și calculează valoarea determinantului Δ (delta).

6. Scrieți un program care calculează n! pentru n declarat în program, 2<=n<5.