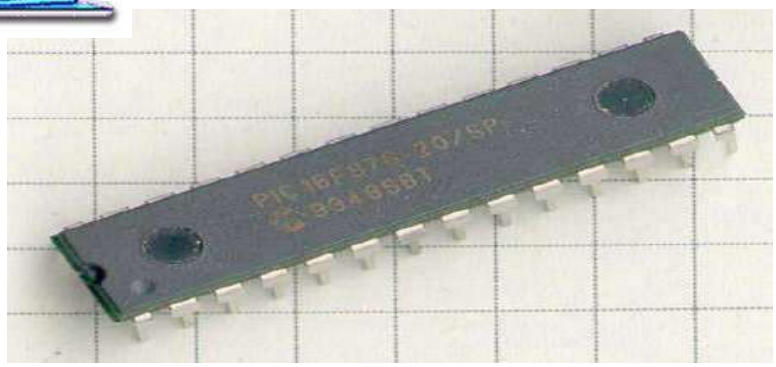
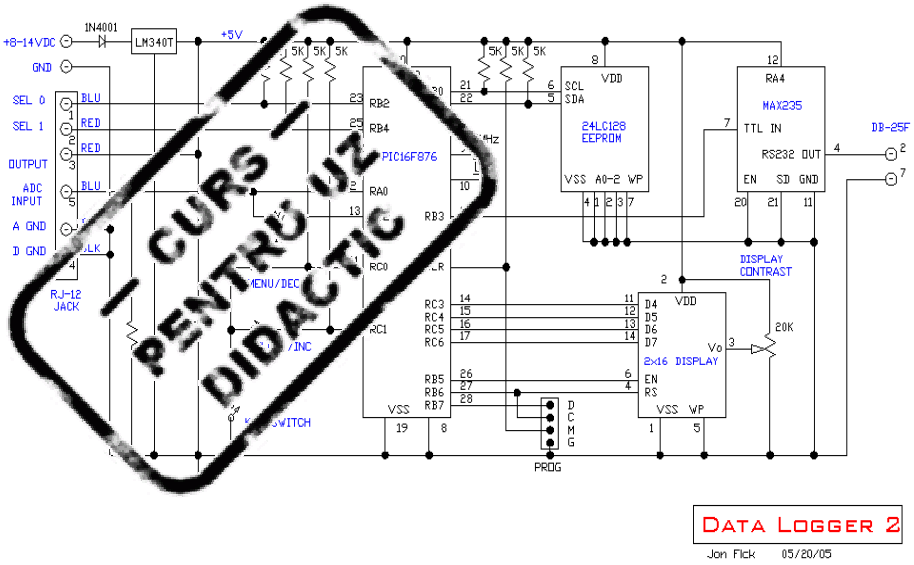
MINISTERUL EDUCAȚIEI ŞI CERCETĀRII
UNIVERSITATEA DIN BACĀU
FACULTATEA DE INGINERIE

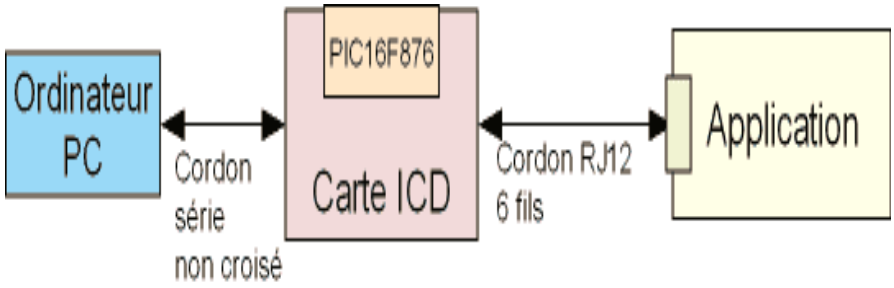


DAN ROTAR

MICROPROCESOARE

PROGRAMAREA ÎN LIMBAJ DE ASAMBLARE A MICROPROCESOARELOR: 80X86; TMS 320F240;PIC 16F87x





EDITURA ALMA MATER

CUPRINS

		pag.
	CAPITOLUL 1	
	PROGRAMAREA ÎN LIMBAJ DE ASAMBLARE	5
1.1.	Introducere	5
1.2.	Caracterizarea limbajului de asamblare	7
1.3.	Etapele elaborării unui program în cod maşină	9
1.4.	Formatul fişierelor hex (.hex, .hxl, .hxh)	12
	CAPITOLUL 2	
	PROGRAMAREA MICROPROCESORULUI INTEL 8086	18
2.1.	Structura microprocesorului 8086	18
2.2.	Instrucțiunile microprocesorului 8086	20
2.3.	Extinderea structurii unității centrale la familia 80x86	35
2.3.1.	Unitatea centrală 80x86 din punct de vedere al programatorului	35
2.3.2.	Registrele de uz general ale unității centrale 8086	35
2.3.3.	Registrele de segment 8086	36
2.3.4.	Registrele de uz special	37
2.3.5.	Registrele 80286	38
2.3.6.	Registrele procesoarelor 80386/80486	38
2.3.7.	Organizarea memoriei fizice la 80x86	39
2.3.8.	Segmentele la 80x86	40
2.3.9.	Adrese normalizate la 80x86	42
2.3.10.	Registrele de segment la procesoarele 80x86	43
2.4.	Modurile de adresare la procesoarele 80x86	44
2.4.1.	Modul de adresare a registrelor la procesorul 8086	45
2.4.2.	Modurile de adresare ale memoriei la procesorul 8086	45
2.4.2.1.	Modul de adresare numai prin deplasament	46
2.4.2.2.	Modul de adresare indirectă prin registre	47
2.4.2.3.	Modurile de adresare indexate	49
2.4.2.4.	Modul de adresare indexat bazat	50
2.4.2.5.	Adresare indexată bazată plus deplasament	50
2.4.2.6.	Un mod simplu de a reține modurile de adresare a memoriei la procesorul 8086	52

2.4.2.7. Câteva comentarii finale asupra modurilor de adresare la procesorul 8086 52
2.4.3. Modurile de adresare a registrelor la 80386 53
2.4.3.1. Modurile de adresare a memoriei la 80386 53
2.4.3.2. Modul de adresare indirectă prin registre 53
2.4.3.3. Modurile de adresare indexat, indexat/bazat şi bazat/indexat/deplasament la procesorul 80386 53
2.4.3.4. Modul de adresare scalat indexat la procesorul 80386 55
2.4.3.5. Câteva considerații finale asupra modurilor de adresare a memoriei la 80386 55
2.5. Instrucțiunea MOV la procesorul 8086 56
2.6. Comentarii finale asupra instrucțiunilor MOV 59
2.7. Câteva instrucțiuni suplimentare 59
2.8. Structura unui program în limbaj de asamblare 60
2.8.1. Directivele de segmentare 61
2.8.2. Directivele pentru definirea datelor 68
2.8.3. Concluzii privind limbajul de asamblare 68
2.9. Scrierea aplicațiilor Windows în limbaj de asamblare 77
2.9.1. Includerea limbajului de asamblare în programele Visual Basic 77

CAPITOLUL 3
PROGRAMAREA MICROPROCESORULUI TMS 320F240 81

3.1. Setul de instrucțiuni a procesoarelor Texas Instruments C5X/C2XX 81
3.2. Turbo-Asamblorul (TASM) 91
3.3. Exemple de programe în limbaj de asamblare, pentru microprocesorul TMS 320F240 107

CAPITOLUL 4
PROGRAMAREA MICROCONTROLERELOR DE TIP PIC12, PIC16 ŞI PIC 18 121

4.1. Organizarea memoriei microcontrolerelor PIC 121
4.1.1. Memoria program 123
4.1.2. Memoria de date 123
4.1.3. Registrele SFR 123
4.1.4. Bancuri de Memorie 123
4.1.5. Contorul de Program 124
4.1.6. Stiva 124
4.1.7. Registrul STATUS (ADRESA: 03h, 83h) 124
4.1.8. Registrul OPTION (ADRESA: 81h) 126
4.1.9. Registrul INTCON (ADRESA: 0Bh, 8Bh) 127
4.1.10. PCL şi PCLATH 128
4.1.11. Memoria de date EEPROM 128
4.1.12. Registrul EECON1 (ADRESA: 88h) 129
4.1.13. Citirea memoriei EEPROM 129

4.1.14. Scrierea în memoria de date EEPROM 130
4.1.15. Verificarea scrierii 131
4.1.16. Harta memoriei RAM 132
4.1.17. Moduri de adresare 132
4.2. Porturile microcontrolerului 134
4.3. Setul de instrucţiuni a unităţilor centrale de tip RISC PIC12, PIC16 şi PIC18 136
4.4. Exemple de programme în limbaj de asamblare 143
4.4.1. Iniţializarea unei zone de memorie RAM 143
4.4.2. Salvarea şi restaurarea regiştrilor (echivalentul instrucţiunilor PUSH şi POP) 144
4.4.3. Testarea conţinutului unui registru 145
4.4.4. Conversie binar-ASCII 145
4.4.5. Afişarea unui şir pe un display LCD 145

BIBLIOGRAFIE 153

CAPITOLUL 1

PROGRAMAREA ÎN LIMBAJ DE ASAMBLARE

1.1. Introducere

Microprocesoarele reprezintă unităţi centrale integrate într-un singur circuit integrat pe scară foarte largă (VLSI – Very Large Scale Integration), care au căpătat o largă dezvoltare o dată cu dezvoltarea tehnologiei de integrare şi cu răspândirea utilizării sistemelor cu microprocesor în cele mai diverse domenii de activitate.

Pe de altă parte, tendinţa de miniaturizare continuă a sistemelor numerice a dus la apariţia şi dezvoltarea calculatoarelor integrate, utilizate în cele mai diverse domenii ale activităţii umane.

Procesarea digitală a semnalelor (DSP – Digital Signal Processing) se distinge de alte domenii ale ştiinţei calculatoarelor prin faptul că există un singur tip de date utilizate şi anume semnalele. În marea majoritate a cazurilor aceste semnale provin de la senzori care preiau mărimi din lumea reală: vibraţii seismice, imagini, sunete etc.

DSP reprezintă matematica, algoritmii şi tehnicile utilizate pentru prelucrarea acestor semnale după ce acestea au fost transformate în prealabil în format digital. Acestă prelucrare se face în diferite scopuri, obiectivele urmărite având un spectru larg de aplicabilitate: analiza imaginilor, recunoaşterea formelor, recunoaşterea şi generarea vorbirii, compresia datelor pentru stocare sau transmitere etc.

Daca vom ataşa un convertor analog-digital unui calculator în scopul preluării unei anumite cantităţi de date din lumea reală, tehnica DSP ne va ajuta să interpretăm aceste date.

Începuturile DSP se localizează la începuturile anilor 1960 şi 1970 cand calculatoarele numerice au început să fie folosite în diferite ramuri ale ştiinţei şi tehnicii. În această perioadă însă calculatoarele erau foarte scumpe şi din acest motiv aplicaţiile DSP erau limitate doar la câteva domenii de mare interes. Încercări de pionerat s-au făcut în domeniile cheie ca: tehnologia radar care presupune creşterea securităţii naţionale, exploatarea petrolului care aduce venituri însemnate, explorarea spaţiului cosmic unde utilizarea acestei tehnologii este indispensabilă şi analiza imaginilor în domeniul medical care permite salvarea de vieţi omeneşti.

Revoluţia calculatoarelor personale din anii 1980 şi 1990 a dus la dezvoltarea spectaculoasă a tehnicilor DSP într-un număr impresionant de domenii. Dacă la început această tehnică era utilizată aproape exclusiv în aplicaţii militare sau guvernamentale, scăderea preţului de cost a tehnicii de calcul o dată cu dezvoltarea spectaculoasă a tehnologiei digitale a dus la utilizarea tehnologiei DSP în multe domenii comerciale cum sunt: telefonia mobilă, CD playere, poştă electronică vocală etc.

⟺ Microprocesoare --------> μP ⟺

Dezvoltarea tehnologiei DSP a dus la apariția procesoarelor de semnal (DSP – Digital Signal Processor) care reprezintă calculatoare integrate specializate pentru acest domeniu. Observăm faptul că acronimul DSP este folosit atât pentru tehnica de prelucrare digitală a semnalelor cât şi pentru dispozitivul utilizat pentru aceasta.

În final trebuie remarcat că nu există o graniță clară între tehnologia DSP şi alte domenii ale ştiinței. Dintre domeniile care se întrepătrund cu tehnologia DSP se pot aminti:

- teoria comunicației;
- analiza numerică;
- statistica şi probabilitățile;
- procesarea analogică a semnalelor;
- teoria deciziei;
- electronica digitală;
- electronica analogică.

Indiferent de structura sistemului de calcul utilizat, modul de programare al unității centrale se face în acelaşi fel existând similitudini evidente dar şi diferențe importante între diferitele tipuri de unități centrale.

O unitate centrală are un limbaj propriu, care diferă de la o unitate centrală la alta, instrucțiunile unității centrale fiind reprezentate de şiruri de numere binare. Producătorul unității centrale stabileşte tipurile de instrucțiuni, codificarea, structura şi modul de utilizare a acestora. Un program scris în binar cu ajutorul acestor instrucțiuni se numeşte *program maşină* iar codul în care este scris se numeşte *cod obiect (sau cod binar) direct executabil*. Primele programe au fost scrise în acest fel dar evident, scrierea unor astfel de programe este dificilă iar riscul de eroare este ridicat.

Pentru simplificarea scrierii programelor în cod obiect direct executabil, producătorii unităților centrale asociază codului binar corespunzător unei instrucțiuni, un nume care să fie semnificativ şi care să sugereze acțiunea realizată de instrucțiune. Acest nume poartă denumirea de *mnemonică*. Programarea cu mnemonici este mai uşor de realizat dar este necesar un program de traducere din mnemonici în cod binar. Un astfel de program prevăzut cu o serie de facilități care să uşureze munca programatorului se numeşte *asamblor* iar programele scrise cu ajutorul mnemonicelor, pentru asamblor, se numesc programe în *limbaj de asamblare*.

Astăzi programarea unităților centrale se face în limbaj de asamblare. Deoarece acest program se adresează direct structurii fizice a unui sistem de calcul, se spune că limbajul de asamblare este un limbaj de programare de nivel scăzut spre deosebire de limbajele de nivel înalt (C, PASCAL, Java etc) care sunt limbaje de programare de nivel înalt.

Programele scrise în limbaj de asamblare nu pot fi rulate decât pe unitatea centrală pentru care au fost scrise şi din acest motiv se spune că programele scrise în limbaj de asamblare nu sunt *portabile*.

Avantajul utilizării programelor în limbaj de asamblare este reprezentat de faptul că ele permit accesul programatorului la structurile intime ale sistemului de calcul (ceea ce nu se întâmplă la limbajele de nivel înalt) şi permit scrierea unor programe de dimensiuni mici ce se execută în timp scurt iar uneori astfel de cerințe sunt impuse.

Din acest motiv şi limbajele de programare de nivel înalt permit mecanisme de inserare a unor secvențe de program scrise în limbaj de asamblare.

1.2. Caracterizarea limbajului de asamblare

Prezentăm în continuare câteva motive pentru studiul limbajului de asamblare:

- pentru a face programe mai scurte şi care să lucreze mai repede;
- pentru a înțelege mai bine cum lucrează calculatoarele;
- pentru a scrie un cod eficient.

Limbajul de asamblare este puțin răspândit printre nespecialişti. Acest lucru se datorează unor prejudecăți răspândite de-a lungul timpului pe care le vom analiza în continuare în scopul caracterizării corecte a acestui limbaj.

Dificultățile limbajului de asamblare:

1) este greu de învățat;
2) este greu de citit şi de înțeles;
3) este greu de depanat;
4) este greu de întreținut;
5) este greu de scris;
6) programarea în acest limbaj este mare consumatoare de timp;
7) tehnologia îmbunătățită a compilatoarelor a eliminat nevoia de limbaj de asamblare;
8) maşinile actuale sunt atât de rapide încât nu mai este necesară programarea în limbaj de asamblare;
9) dacă este nevoie de viteză se pot folosi algoritmi performanți mai degrabă deât programarea în limbaj de asamblare;
10) maşinile actuale dispun de mari cantități de memorie şi economia adusă de limbajul de asamblare devine neimportantă;
11) limbajul de asamblare nu este portabil.

În general aceste afirmații sunt facute de persoane care nu utilizează limbajul de asamblare şi nici nu au o idee precisă asupra acestui limbaj. Din acest motiv, afirmațiile de mai sus vor fi explicate în ideea că ele pot să nu fie adevărate în momentul când cunoaştem şi folosim limbajul de asamblare.

1) **Limbajul de asamblare este greu de învățat.** Dacă stăpâniți un limbaj de programare cum este, de exemplu, Pascal, atunci învățarea altor limbaje ca de exemplu: C, BASIC, FORTRAN, Modula-2 sau Ada este relativ simplă pentru că ele sunt destul de asemănătoare cu Pascal. Pe de altă parte învățarea unui limbaj ce diferă mult de Pascal, cum este Prolog, nu e simplă. Şi limbajul de asamblare este diferit de Pascal şi atunci el va fi puțin mai dificil de învățat. În orice caz învățarea limbajului de asamblare nu este mai grea decât învățarea pentru prima dată a unui limbaj de programare.

2) **Limbajul de asamblare este greu de citit şi înțeles.** Această afirmație este făcută de persoanele ce nu cunosc acest limbaj. Evident că pot fi scrise programe de neînțeles în limbaj de asamblare ca şi în alte limbaje. După câştigarea

experienței în limbaj de asamblare vă veți da seama că este mai uşor de citit decât alte limbaje.

3) **Limbajul de asamblare este greu de depanat.** Acelaşi argument trebuie explicat ca mai sus. Odată câştigată experiență nimic nu va parea mai simplu.

4) **Limbajul de asamblare este greu de întreținut.** Programele în C sunt greu de întreținut. Aptitudinea de a scrie programe uşor de întreținut se câştigă după oarecare experiență.

5) **Limbajul de asambalre este greu de scris.** Această afirmație are un sâmbure de adevăr. O lungă perioadă de timp programatorii în limbaj de asamblare au scris programele în întregime de la început la sfârşit, reinventând de fiecare dată roata. Limbajele de nivel înalt beneficiază de biblioteci ce simplifică mult munca. Acelaşi lucru poate fi făcut şi în limbaj de asamblare dacă ținem cont că sunt disponibile bibloteci (cele mai multe gratuite) cu majoritatea rutinelor necesare scrierii programelor.

6) **Programarea în limbaj de asamblare este consumatoare de timp.** Este adevărat că eleborarea unui program în limbaj de asamblare necesită mai mult timp (uneori dublu) față de scrierea programelor în limbaje de nivel înalt. Oricum câştigul de timp nu poate umbri celelalte beneficii aduse de limbajul de asamblare.

7) **Tehnologia îmbunătățită a compilatoarelor a eliminat nevoia de limbaj de asamblare.** Acest lucru nu este adevărat şi probabil nu va fi niciodată adevărat. Performanța programelor scrise în limbaj de asamblare constă în modul de scriere al acestora şi de talentul şi inventivitatea programatorului ceea ce nu se poate compara cu acțiunea unui compilator.

8) **Maşinile actuale dispun de mari cantități de memorie şi economia adusă de limbajul de asamblare devine neimportantă.** Este uimitor faptul că oamenii preferă să cheltuie bani ca să cumpere maşini mai rapide în loc să consume timp să scrie programe în limbaj de asamblare. Un fapt rămâne: tot timpul se doreşte mai multă viteză. Pentru o maşină dată cele mai rapide programe rămân cele scrise în limbaj de asamblare.

9) **Dacă este nevoie de viteză se pot folosi algoritmi performanți mai degrabă decât programarea în limbaj de asamblare.** Orice algoritm ce poate fi implementat în limbajele de nivel înalt poate fi implementat şi în limbaj de asamblare şi deci va fi mai rapid aici. Pe de altă parte, există algoritmi ce nu pot fi implementați decât în limbaj de asamblare.

10) **Maşinile actuale dispun de mari cantități de memorie şi economia adusă de limbajul de asamblare devine neimportantă.** Dă-i cuiva un centimetru şi-ți va cere un metru. Este evident că oricât de multă memorie este disponibilă ea nu va

ajunge. De asemenea, din motive tehnice este recomandabil ca programatorii să scrie programe cât mai scurte.

11) **Limbajul de asamblare nu este portabil.** Acest fapt este de necontestat. Dacă programul trebuie să funcţioneze pe procesoare diferite atunci limbajul de asamblare nu este o soluţie.

Limbajul de asamblare prezintă şi avantaje incontestabile pe care le vom enumera în continuare. Aceste avantaje reprezintă argumente serioase în scopul învăţării acestui limbaj.

Avantajele programării în limbaj de asamblare:

- viteză – programele scrise în limbaj de asamblare sunt în general cele mai rapide;
- spaţiu – programele scrise în limbaj de asamblare sunt de regulă cele mai mici;
- performaţă – aceste programe vă permit să faceţi ceea ce este imposibil în limbajele de nivel înalt;
- cunoştiinţe – cunoaşterea limbajului de asamblare vă permite să scrieţi programe mai performante ăn limbajele de nivel înalt.

1.3. Etapele elaborării unui program în cod maşină

Etapele elaborării unui program în cod maşină diferă de la caz la caz. Astfel, dacă dorim să eleborăm un program în cod maşină pentru unitatea centrală a sistemului pe care lucrăm, atunci la sfârşit programul realizat se va găsi în memoria calculatorului, gata de execuţie.

Acest caz îl întâlnim atunci când scriem un program în limbaj de asamblare pentru microprocesorul 8086 pe un calculator personal de tip IBM PC. Aceste calculatoare sunt dotate cu procesoare Intel 80x86 sau compatibile care pot executa programe în cod obiect scrise pentru unitatea centrală 8086 deoarece politica Intel a fost de a păstra compatibilitatea procesoarelor de jos (începând cu 8086) în sus (ultima generaţie de procesor Intel).

Etapele elaborării unui program pentru microprocesorul 8086, lucrând pe un calculator compatibil IBM PC vor fi:

- se scrie programul în limbaj de asamblare cu ajutorul unui editor de texte, obţinându-se un fişier text numit *fişier sursă* care are cel mai adesea extensia **.asm** (fişierul sursă reprezintă instrumentul de lucru al programatorului conţinând numeroase comentarii şi explicaţii);
- traducerea (translatarea) fişierului sursă în format format binar se face cu ajutorul programului *asamblor*. În această etapă asamblorul semnalează eventualele erori de sintaxă ale programului şi generează, în cazul în care programul este corect din punct de vedere sintactic (la acest nivel nu se poate face şi verificarea logică), un fişier în cod binar. În funcţie de

necesităţile programatorului, asamblorul poate genera două tipuri de fişiere:

- fişiere în cod obiect absolut, direct executabile de către unitatea centrală, care sunt aşezate în memorie la adresa de unde vor fi executate (fişierul generat conţine adresele absolute ale programului) şi care vor avea extensia **.com** sau **.exe** (modul de stabilire a extensiei se va explica mai târziu), sau
- fişiere în cod obiect relativ, care conţin codul obiect dar adresele sunt relative (simbolice) şi care nu pot fi executate direct dar pot fi puse în biblioteci (pentru o utilizare ulterioară) şi care pot fi utilizate împreună cu alte programe în cod obiect relativ din biblotecile deja create, pentru obţinerea programului final, în cod obiect absolut;

- dacă s-a obţinut fişierul în cod obiect absolut (extensia **.com** sau **.exe**) se poate trece la lansarea în execuţie în scopul verificării şi eventual a depanării (cu ajutorul unui program de *depanare – debugger*) funcţionării logice a acestuia;
- dacă s-a obţinut un fişier în cod obiect relocabil, fişierul poate fi adăugat unei *biblioteci* (*library*) cu ajutorul unui program *bibliotecar* (*librarian*) sau se poate genera programul în cod obiect absolut cu ajutorul unui *editor de legături* (*linkeditor*). Editorul de legături caută în biblioteci legăturile solicitate, adaugă în programul în cod obiect absolut secvenţele extrase din biblioteci, semnalează eventualele referinţe nerezolvate şi, în cazul în care nu au fost erori, generează codul obiect absolut.

Să exemplificăm modul în care se poate realiza un program care conţine o operaţie de înmulţire utilizată dintr-o bibliotecă matematică externă. Pentru aceasta trebuie să cunoaştem numele rutinei de înmulţire din biblioteca utilizată.

Se scrie programul in limbaj de asamblare iar acolo unde se foloseşte operaţia de înmultire se scrie numele rutinei din biblioteca externă şi se respectă convenţiile de utilizare specificate pentru biblioteca respectivă. De asemenea se specifică în program faptul că numele folosit este o referinţă externă în aşa fel încât asamblorul să nu semnaleze o eroare. Modul de lucru este similar cu cel din limbajele de nivel înalt când se folosesc funcţii sau proceduri din bibliotecile externe.

Se generează fişierul în cod obiect relocabil, după care se trece la prelucrarea cu ajutorul editorului de legături. Linkeditorului i se specifică numele bibliotecii în care să caute rutina de înmulţire şi dacă acesta o găseşte, include în programul în cod obiect absolut secvenţa de cod corespunzătoare rutinei.

În afară de uneltele (programele) folosite pentru prelucrare, se mai foloseşte şi un program numit *dezasamblor* (*disassembler*), util în depanare, care are acţiune inversă asamblorului: traduce formatul în cod obiect absolut în textul corespunzător.

În cazul în care se dezvoltă programe pe o altă maşină decât cea pentru care se scriu programele, etapele sunt aceleaşi (asamblorul se numeşte în acest caz *crossasamblor – crossassembler*), numai că la final programul trebuie transferat de pe maşina pe care s-a lucrat pe maşina pentru care s-a scris programul. Transferul se face de regulă cu ajutorul programatoarelor care transferă codul obiect absolut din memoria maşinii gazdă în memoria maşinii pentru care s-a scris programul. Pentru transfer se foloseşte formatul *IntelHEX*.

Pentru a exemplifica acest lucru, vom arăta modul în care se dezvoltă programe pentru microcontrolere de tip PIC (Microchip) pe maşini de calcul compatibile IBM PC (calculatoare personale).

Pe calculatorul personal se foloseşte programul MPLAB, furnizat în mod gratuit de firma Microchip, care este un mediu de dezvoltare (IDE – Integrated Development Environment) care conține un crosasamblor, un dezasamblor, un linkeditor, un simulator, un bibliotecar, help şi alte facilități pentru realizarea programelor.

În acest fel, programele se pot dezvolta si pune la punct pe calculatorul personal şi apoi se pot transfera în memoria microcontrolerului. Verificarea finală se face însă tot prin execuția programului pe microcontroler în aşa fel încât să se poată verifica în condiții reale funcționarea programului.

Generarea codului absolut.

Codul absolut este ieşirea implicită pentru (cross)asamblorul MPASM. Procesul este arătat în figura 1.1.

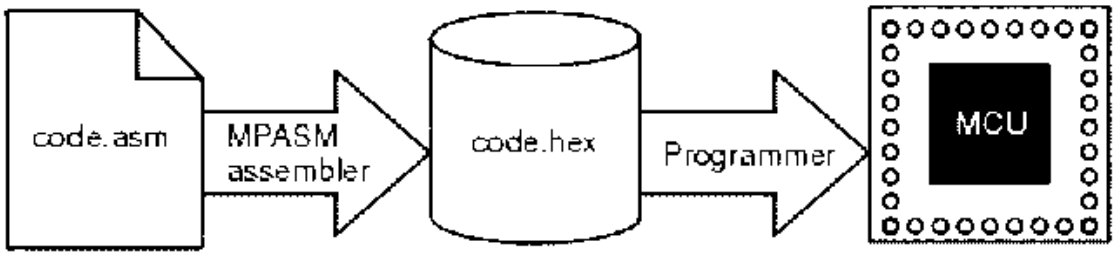


Figura 1.1. Generarea codului absolut pentru microcontroler.

Când un fişier sursă este asamblat în acest fel, toate variabilele şi subprogramele folosite în fişierul sursă trebuie să fie definite în acest fişier sursă sau să fi fost incluse în acest fişier. Dacă asamblarea se realizează fără erori, se va genera un fişier hex ce conține codul execurtabil pentru dispozitivul țintă. Acest fişier poate fi utilizat cu un depanator pentru a teste execuția codului sau cu un programator pentru programarea dispozitivului.

Generarea codului relocabil.

Asamblorul MPASM are de asemenea posibilitatea de a genera un modul obiect relocabil care poate fi legat cu alte module utilzând MPLINK pentru obținerea codului executabil. Această metodă este foarte folositoare pentru crearea modulelor reutilizabile.

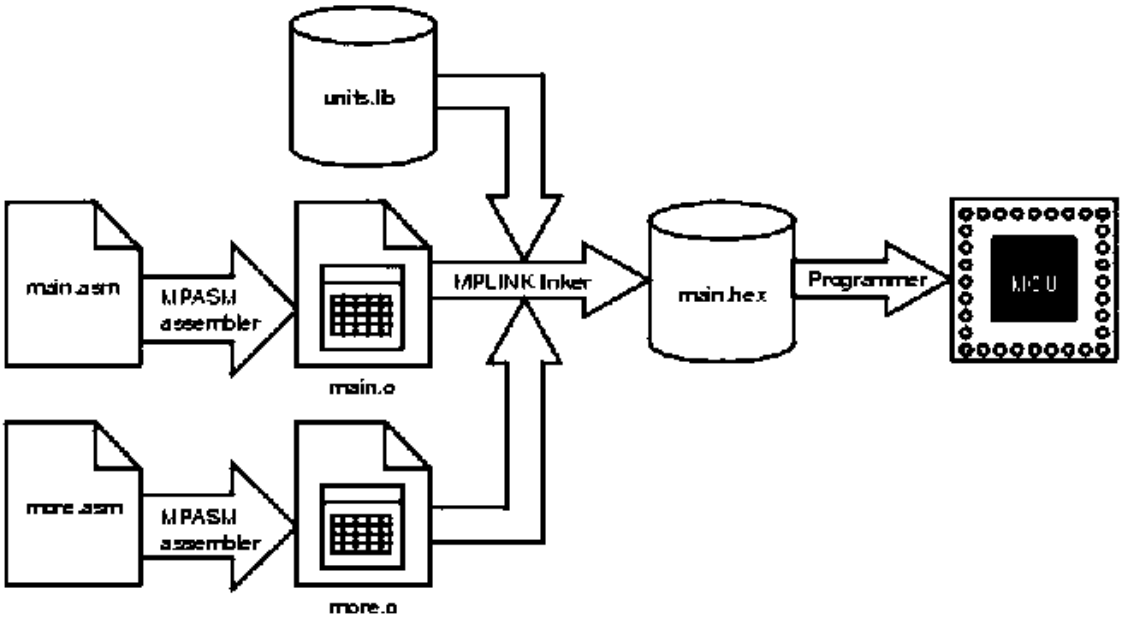


Figura 1.2. Generarea şi utilizarea fişierelor relocabile.

Modulele înrudite pot fi grupate şi stocate împreună folosind bibliotecarul MPLIB. Bibliotecile necesare pot fi specificate la link-editare şi numai rutinele necesare vor fi incluse ţn fisierul executabil.

1.4. Formatul fişierelor hex (.hex, .hxl, .hxh)

Asamblorul MPASM şi linkeditorul MPLINK pot crea fişiere text ASCII de tip hex de diferite formate:

Numele formatului	Tipul formatului	Extensia fişierului	Utilizare
Format Intel Hex	INHX8M	.hex	Programatoare pentru dispozitive de 8 biţi
Format Intel Split Hex	INHX8S	.hxl, .hxh	Programatoare par/impar
Intel Hex 32 Format	INHX32	.hex	Programatoare pentru dispozitive de 16 biţi

Aceste formate de fişiere sunt folositoare pentru transferarea codului pentru PIC MCU către programatoare.

Formatul Intel HEX

Acest format crează un fişier hex pe 8 biţi cu combinaţia octetul cel mai puţin semnificativ, octetul cel mai semnificativ (low byte, high byte). Chiar dacă fiecare adresă conţine 8 biţi, în acest format toate adresele sunt dublate.

Fiecare înregistrare începe cu un prefix de 9 caractere şi se termină cu o suma de contro de 2 caractere. Fiecare înregistrare are următorul format:

:BBAAAATTHHHH....HHHCC

unde:

BB — un număr de un octet cu două cifre hexazecimale ce reprezintă numărul de octeți de date ce apar pe linie;

AAAA — o adresă hexazecimală de 4 cifre care arată adresa de start a înregistrării de date;

TT — tipul înregistrării reprezentată pe 2 cifre hexa care este întotdeauna "00" cu excepția înregistrării de sfârşit de fişier (end of file) care este "01";

HH — un octet de date reprezentat cu 2 cifre hexa în ordinea Low byte/High byte;

CC — o sumă de control cu 2 cifre hexa care reprezintă suma tuturor octeților precedenți ai înregistrării în complement față de 2. (Notă - complementul față de 2 se calculeaza ca suma octeților precedenți care apoi se scade din 256. De exemplu suma = 5 iar in complement fața de 2=256-5=251)

Exemplu: INHX8M

nume_fisier.hex

```
:100000000000000000000000000000000000000F0
:0400100000000000EC
:1000320000002800400068 00A800E800C80028016D
:100042006801A9018901EA01280208026A02BF02C5
:10005200E002E80228036803BF03E803C8030804B8
:1000620008040804030443050306E807E807FF0839
:06007200FF08FF08190A57
:00000001FF
```

Formatul Intel Split Hex

Formatul divizat (split) creaza 2 fisiere: .hxl şi .hxh. Formatul este asemanator cu formatul normal pe 8 biți cu excepția faptului că octeții cei mai puțin semnificativi ai cuvintelor de date sunt stocați în fişierul .hxl iar octeții cei mai semnificativi ai cuvintelor de date sunt stocați în fişierul .hxh iar adresele sunt împşrțite la 2. Acest format este folosit la programarea cuvintelor de 16 biți în două memorii EPROM de 8 biți care vor fi folosite pereche (împreună).

Exemplu: INHX8S

nume_fisier.hxl

:0A0000000000000000000000000000F6
:1000190000284068A8E8C82868A989EA28086ABFAA
:10002900E0E82868BFE8C8080808034303E8E8FFD0
:03003900FFFF19AD
:00000001FF

nume_fisier.hxh

:0A0000000000000000000000000000F6
:100019000000000000000001010101010202020 2CA
:10002900020203030303030404040405060707 0883
:0300390008080AAA
:00000001FF

Format Intel Hex 32

Formatul extins la 32 de biți este similar formatului pe 8 biți cu excepția faptului că adresa stabileşte cei mai semnificativi 16 biți ai adresei de date. Acest format este folosit pentru dispozitive pe 16 biți la care memoria de program adresabilă depăşeşte 64 de kocteți.

Fiecare înregistrare de date începe cu un prefix de 9 cartactere şi se termină cu o suma de control de 2 caractere. Fiecare înregistrare are următorul format:

:BBAAAATTHHHH.....HHHCC

unde:

BB — un număr de un octet cu două cifre hexazecimale ce reprezintă numărul de octeți de date ce apar pe linie;

AAAA — o adresă hexazecimală de 4 cifre care arată adresa de start a înregistrării de date;

TT — tipul înregistrării reprezentată pe 2 cifre:
00 - înregistrare de date
01 - înregistrare end of file
02 - Segment address record
04 - Linear address record

HH — un octet de date reprezentat cu 2 cifre hexa în ordinea Low byte/High byte;

CC — o sumă de control cu 2 cifre hexa care reprezintă suma tuturor octeților precedenți ai înregistrării în complement față de 2. (Notă - complementul față de 2 se calculeaza ca suma octeților precedenți care apoi se scade din 256. De exemplu suma = 5 iar in complement fața de 2=256-5=251)

Analiza generarii unui fişier INTEL HEX

Folosim urmatorul studiu de caz:

Fişierul listing:

```
MPASM  4.02 Released
TEST1.ASM   12-13-2005  8:09:51         PAGE  1
LOC  OBJECT CODE     LINE SOURCE TEXT
  VALUE

                      00001 ;Program pentru initializarea
portului B si setarea pinilor
                      00002 ;la starea unu logic
                      00003
                      00004 ;Declaratia si configuratia
procesorului
                      00005
                      00006         PROCESSOR 16F84A
                      00007         #include "p16f84A.inc"
                      00001         LIST
                      00002 ; P16F84A.INC  Standard Header
File, Version 2.00    Microchip Technology, Inc.
                      00134         LIST
                      00008
Warning[205]: Found directive in column 1. (LIST)
                      00009 LIST
2007   3FF1           00010         __CONFIG _CP_OFF
&_WDT_OFF &_PWRTE_ON &_XT_OSC
                      00011
0000                  00012         org 0x00
;vector reset
0000   2805           00013         goto main
0004                  00014         org 0x04
;rutina de intrerupere nu exista
0004   2805           00015         goto main
                      00016
0005                  00017 main
0005   0000           00018         nop
;programul principal
0006   0000           00019         nop
0007   2805           00020         goto main
```

Din acest listing rezultă că trebuie scris în memorie:

Adresa	Valoare cuvânt
2007	3FF1
0000	2805
0004	2805
0005	0000
0006	0000
0007	2805

Fişierul HEX obţinut:

```
:020000040000FA
:020000000528D1
:08000800052800000000052896
:02400E00F13F80
:00000001FF
```

- prima linie: 020000040000FA este compusă din: 02 - numarul de octeţi de date ce apar pe linie, 0000 - adresa de start a înregistrării de date (în acest caz aici trebuie să fie întotdeauna 0000), 04 - extended linear address record, 0000 - cei mai semnificativi 16 biti ai adresei, FA - suma de control = 01h + NOTA (02h + 00h + 00h + 04h + 00h + 00h). Modul de calcul se face în felul urmator: 02h+04h+06h, 06h mod 100h = 06h, NOT(06h) = FFh - 06h = F9h, F9h+01h = FAh.
- a doua linie: 020000000528D1 este compusa din: 02 - numarul de octeţi de date ce apar pe linie, 0000 - adresa de start a înregistrării de date, 00 - înregistrare de date, 0528 - datele scrise în ordinea: octetul cel mai putin semnificativ, octetul cel mai semnificativ, deci este de fapt 2805 în listing, D1 - suma de control: 01h + NOT (02h+05h+28h) = 01h + NOT(2Fh) ((2Fh MOD 100h=2Fh)), 01h+FFh-2Fh=D1.
- a treia linie: 08000800052800000000052896: 08 - numărul de octeti de date, 0008 - adresa de start, 00 - înregistrare de date, octeţii de date sunt: 0528000000000528, ceea ce înseamna: 2805, 0000, 0000, 2805. Se pare că din cauză că pentru fiecare adresă sunt câte 2 octeţi, aceasta apare dublată. Deci adresa de start 0008 este de fapt adresa 0004. Suma de control: 08h+08h+05h+28h+05h+28h=6Ah, 6Ah MOD 100h = 6Ah, FFh-6Ah=95h+01h=96h.
- a patra linie: 02400E00F13F80: 02 - 2 0cteti de date, 400Eh - adresa de start care e în realitate 400Eh/2h=2007h (ca în listing), 00 - înregistrare de date, datele: F13F care se citesc 3FF1, şi suma de control: 02h+40h+0Eh+F1h+3Fh = 180h MOD 100h = 80h, FFh - 80h = 7Fh + 01h =80h.
- ultima linie: 00000001FF este linia de End of File.

Acest studiu de caz a fost aplicat pentru un exemplu realizat în MPLAB. Se pot trage următoarele concluzii:

- se foloseşte adresarea lineară extinsă;
- adresa de start a datelor de pe o linie din fişierul HEX este dublul adresei reale (din listing);
- fiecărei adrese îi sunt asociați 2 octeți, 4 cifre hexa, scrise în ordinea: cel mai putin semnificativ octet urmat de cel mai semnificativ octet;
- restul regulilor sut respectate de la formatul INTEL HEX.

CAPITOLUL 2

PROGRAMAREA MICROPROCESORULUI INTEL 8086

Vom reaminti, pentru început, principalele caracteristici ale structurii accesibile programatorului, pentru microprocesorul I8086, utile în programarea în limbaj de asamblare a acestuia.

2.1. Structura microprocesorului 8086

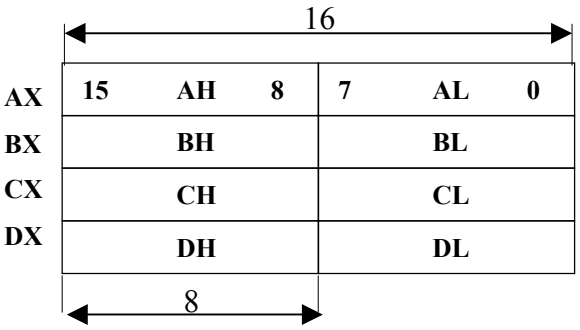
Există trei categorii de registre: registre de uz general, registre de adrese, indicatori, pointeri şi index şi registre de uz special.

Unitatea centrală 8086 are patru registre de uz general, pe 16 biţi:

AX, BX, CX, DX

care pot fi folosite şi ca registre de 8 biţi:

AH, AL, BH, BL, CH, CL, DH, DL



aşa cum se arată în figura alăturată.

Fiecare dintre registrele de 16 biţi poate fi folosit ca destinaţie a datelor (acumulator) dar registrul acumulator implicit este registrul AX.

Registrul bistabililor de condiţii şi de control al microprocesorului, FX, asociat cu registrul acumulator (de regulă registrul AX):

FX	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	X	X	X	X	OF	DF	IF	TF	SF	ZF	X	AF	X	PF	X	X

unde cu X s-a simbolizat bistabilul inaccesibil utilizatorului.

Semnificaţia fanioanelor din registrul **FX** este:

CF = "carry flag": depăşire aritmetică (CF = 0 – nu s-a produs o depăşire aritmetică, CF = 1 – s-a produs o depăşire aritmetică);

PF = “parity flag”: paritate;

AF = "auxiliary flag": transport între bitul 7 şi 8;

ZF = "zero flag": valoare zero (ZF = 0 – în acumulator este o valoare diferită de zero, ZF = 1 – în urma unei operaţii, valoarea rezultată în acumulator este zero);

SF = "sign flag": semnul în reprezentarea numerelor cu semn (bitul 15);

TF = "trip flag"; TF = 1 determină UCP să lucreze în mod pas cu pas ("single step"), în care se generează o întrerupere internă după fiecare execuţie a unei instrucţiuni;

IF = masca pentru întreruperi externe (IF = 1 => validarea întreruperilor; IF = 0 => invalidarea întreruperilor);

DF = "direction flag"- indică direcţia deplasării adresei la operaţiile cu şiruri de date (DF = 1 => autodecrementare, DF = 0 => autoincrementare, după o operaţie elementară);

OF = V (depăşire).

Registrele de adrese, indicatori, pointeri şi index (se utilizează numai pe 16 biţi): **SP** (pointer stivă), **BP** (pointer adresă de bază), **SI** (index sursă), **DI** (index destinaţie).

Utilizările implicite ale registrelor sunt:

AX: utilizat pentru operaţii aritmetice înmulţire, împărţire pe 16 biţi şi pentru operaţii de I/E pe 16 biţi; în mod analog AL este utilizat pe 8 biţi şi în plus pentru aritmetică zecimală şi conversii de cod; AH este utilizat la înmulţiri şi împărţiri pe 8 biţi;

BX: utilizat în conversii de cod şi ca registru de bază de adrese;

CX: utilizat în operaţii cu şiruri, cu rol de contor de cicluri;

CL: utilizat în deplasări (stânga, dreapta – cu un număr de paşi daţi ca parametru de valoare lui CL);

DX: utilizat la înmulţiri, împărţiri pe 16 biţi şi ca registru de adresare indirectă la porţile de intrare – ieşire (I/E);

SP,BP: utilizat implicit în toate operaţiile cu stiva;

SI, DI: utilizate în operaţiile asupra şirurilor de date, **SI** conţine adresa sursei iar **DI** adresa destinaţiei.

Registrele de uz special sunt cele destinate adresării segmentate:

CS, DS, SS, ES = sunt registre segment care conţin adresele de bază ale segmentelor logice de cod, date, stivă şi extrasegment;

IP = Instruction Printer = contor de program, cu 16 biţi. Valoare ce reprezintă adresa relativă (offset-ul) a instrucţiunii curente în segmentul de cod (relativ la CS). În cazul unei instrucţiuni de salt, IP este salvat în vârful stivei (împreună cu CS, deci saltul este inter-segment) şi apoi încărcat cu adresa relativă în segmentul de cod a instrucţiunii ţintă.

Între registre de adrese, indicatori, pointeri şi index şi registre de uz special există anumite relaţii în funcţionarea microprocesorului 8086. Segmentul de date are ca registru segment registrul DS şi ca registru pointer implicit, registrul DX.

Relaţiile între registrele microprocesorului 8086 sunt prezentate în figura 2.1.

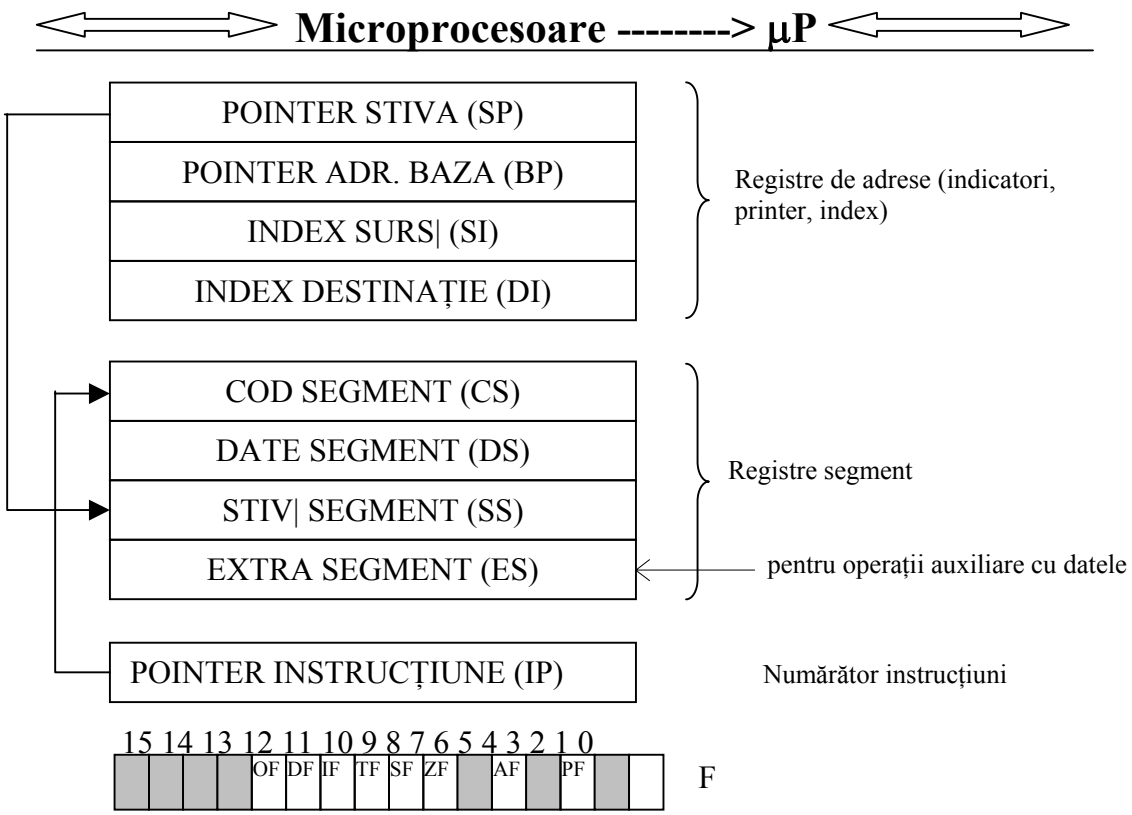


Figura 2.1. Relația între registrele microprocesorului 8086.

2.2. Instrucțiunile microprocesorului 8086

Microprocesorul 8086 are un set complex de instrucțiuni. Mnemonicile utilizate sunt prezentate în tabelul 2.1.

TABELUL 2.1.

	CMPSB				MOV		
AAA	CMPSW	JAE	JNBE	JPO	MOVSB	RCR	SCASB
AAD	CWD	JB	JNC	JS	MOVSW	REP	SCASW
AAM	DAA	JBE	JNE	JZ	MUL	REPE	SHL
AAS	DAS	JC	JNG	LAHF	NEG	REPNE	SHR
ADC	DEC	JCXZ	JNGE	LDS	NOP	REPNZ	STC
ADD	DIV	JE	JNL	LEA	NOT	REPZ	STD
AND	HLT	JG	JNLE	LES	OR	RET	STI
CALL	IDIV	JGE	JNO	LODSB	OUT	RETF	STOSB
CBW	IMUL	JL	JNP	LODSW	POP	ROL	STOSW
CLC	IN	JLE	JNS	LOOP	POPA	ROR	SUB
CLD	INC	JMP	JNZ	LOOPE	POPF	SAHF	TEST
CLI	INT	JNA	JO	LOOPNE	PUSH	SAL	XCHG
CMC	INTO	JNAE	JP	LOOPNZ	PUSHA	SAR	XLATB
CMP	IRET	JNB	JPE	LOOPZ	PUSHF	SBB	XOR
	JA				RCL		

Semnificația acestor mnemonici va fi prezentată pe scurt, în continuare.

AAA - ASCII adjust for addition (ajustare ASCII pentru adunare)

Indicatori afectati: AF, CF

Descriere: Daca cei mai putin semnificativi 4 biti ai lui AL sint mai mari decit 9 sau daca carry auxiliar este 1, atunci aduna 6 la AL si 1 la AH. AF si CF sint actualizati.

AAD - ASCII adjust for division (ajustare ASCII pentru impartire)

Indicatori afectati: PF,SF,ZF

Descriere: Octetul semnificativ a lui AH este inmultit cu 10 si adunat la AL.

AAM - ASCII adjust for multiply (ajustare ASCII pentru inmultire)

Indicatori afectati: PF,SF,ZF

Descriere: Daca jumatatea mai putin semnificativa a lui AL este mai mica de 9 sau daca (AF)=1 atunci se scade 6 din AL si 1 din AH. Indicatorii (AF) si (CF) devin 1. Vechea valoare a lui AL este inlocuita de un octet in care jumatatea superioara este 0 iar jumatatea inferioara este un numar creat de scaderea anterioara.

AAS - ASCII adjust for subtraction (ajustare ASCII pentru scadere)

Indicatori afectati: AF,CF

Descriere: Daca jumatatea mai putin semnificativa a lui AL este mai mica de 9 sau daca (AF)=1 atunci se scade 6 din AL si 1 din AH. Indicatorii (AF) si (CF) devin 1. Vechea valoare a lui AL este inlocuita de un octet in care jumatatea superioara este 0 iar jumatatea inferioara este un numar creat de scaderea anterioara.

ADC - add with carry (aduna cu carry)

Indicatori afectati: AF,CF,OF,PF,SF,ZF

Desciere: Suma celor doi operanzi si a lui carry este memorata in operandul destinatie (stinga).

ADD – addition (adunare)

Indicatori afectati: AF,CF,OF,PF,SF,ZF

Descriere: Suma celor doi operanzi este memorata in operandul destinatie (stinga)

AND - logic and (si logic)

Indicatori afectati: CF,OF,PF,SF,ZF

Descriere: Se realizeaza si logic intre cei doi operanzi, rezultatul va avea 1 in pozitiile in care ambii operanzi au 1, restul fiind 0. Rezultatul este memorat la operandul din stinga. Carry si overflow sunt pusi pe 0.

CALL - call a procedure (apel de procedura)

Indicatori afectati: niciunul

Descriere: Daca este un apel intersegmente, stiva este decrementata cu 2 si continutul lui CS este salvat in ea. CS va fi umplut cu al doilea cuvint al dublului cuvint de adresare. Apoi se salveaza in stiva, in acelasi mod, si continutul lui IP. Ultimul pas este de a inlocui continutul lui IP cu offset-ul adresei de destinatie, adica offset-ul primei instructiuni din procedura. Un apel in cadrul aceluiasi segment sau grup are numai pasii 2,3 si 4.

CBW - convert byte to word (converteste octet la cuvint)

Indicatori afectati: niciunul

Descriere: Daca AL e mai mic decit 80h, atunci AH devine 0. Altfel, AH este setat la 0ffh. Este echivalent cu a replica bitul 7 a lui AL la AH.

CLC - clear carry flag (sterge indicatorul carry)
Indicatori afectati: CF
Descriere: Indicatorul carry este pus la zero.

CLD - clear direction flag (sterge indicatorul directie)
Indicatori afectati: DF
Descriere: Indicatorul directie este pus la zero.

CLI - clear interrupt flag (sterge indicatorul intrerupere)
Indicatori afectati: IF
Descriere: Indicatorul intrerupere este sters.

CMC - complement carry flag (complementeaza indicatorul carry)
Indicatori afectati: CF
Descriere: Daca carry este 0, el devine 1; daca este 1 devine 0.

CMP - compare two operands (compara doi operanzi)
Indicatori afectati: AF,CF,OF,PF,SF,ZF
Descriere: Operandul sursa (stinga) este scazut din operandul destinatie (dreapta). Indicatorii dint afectati dar operanzii nu.

CMPS - compare byte string, compare word string (compara sir de octet,compara sir de cuvint)
Indicatori afectati: AF,CF,OF,PF,SF,ZF
Descriere: Operandul din dreapta, utilizind DI ca registru index este scazut din operandul din dreapta, care utilizeaza registrul SI ca index. Sint afectati numai indicatorii. DI si SI sint incrementate daca indicatorul de directie este 0, si decrementate daca e 1. Incrementul este 1 pentru sir de octeti si 2 pentru cel de cuvinte.

CWD - convert word to doubleword (converteste cuvint la dublucuvint)
Indicatori afectati: niciunul
Descriere: Cel mai semnificativ bit din AX este replicat in DX.

DAA - decimal adjust for addition (ajustare zecimala pentru adunare)
Indicatori afectati: AF,CF,PF,SF,ZF
Descriere: Daca cei mai putin semnificativi (4) biti a lui AL sint mai mari decit 9 sau daca carry auxiliar este 1 , atunci aduna 6 la AL si AF devine 1. Daca AL este mai mare decit 9fh sau carry este 1 atunci aduna 60h la AL si seteaza CF.

DAS - decimal adjust for subtraction (ajustare zecimala pentru scadere)
Indicatori afectati: AF,CF,PF,SF,ZF
Descriere: Daca cei mai putin semnificativi (4) biti a lui AL sunt mai mari decit 9 sau daca carry auxiliar este 1, atunci scade 6 din AL si AF devine 1. Daca AL este mai mare decit 9fh sau carry este 1 atunci scade 60h din AL si seteaza CF.

DEC - decrement destination by one (decrementeaza destinatia cu unu)
Indicatori afectati: AF,OF,PF,SF,ZF
Descriere: Operandul specificat este redus cu 1.

DIV - division, unsigned (impartire, fara semn)
Indicatori afectati: rezultatele indicatorilor nu sint valide
Descriere: Daca rezultatul impartirii e o valoare care nu poate fi pastrata in registrul corespunzator, se genereaza o intrerupere de nivel 0. Indicatorii sint pusi in stiva, IF si TF devin 0, CS este de asemenea pus in stiva, fiind apoi umplut cu valoarea de la adresa 2. Si IP curent este salvat si apoi incarcat cu valoarea de la adresa 0. Aceasta secventa include un apel lung la rutina de intreruperi ale carui segment si offset sint memorate la locatiile 2 si 0. Daca

rezultatul incape atunci citul este memorat in AL sau AX (pentru operatii pe cuvint) si respectiv restul in AH sau DX.

ESC - escape

Indicatori afectati: niciunul

Descriere: Instructiunea ESC furnizeaza un mecanism prin care alte procesoare pot primi instructiuni de la 8086 si utilizeaza modul de adresare a lui 8086. Procesorul 8086 nu are alta operatie pentru ESC decit de a accesa un operand din memorie si de a-l plasa pe magistrala.

HLT - halt

Indicatori afectati: nici unul

Descriere: Instructiunea HLT determina procesorul 8086 sa intre in starea halt. Starea halt este stearsa prin intrerupere externa valida sau reset.

IDIV - integer division, signed (impartire intreaga, cu semn)

Indicatori afectati: AF,CF,OF,PF,SF,ZF dar sint toti nedefiniti

Descriere: Daca rezultatul impartirii e o valoare care nu poate fi pastrata in registrul corespunzator, se genereaza o intrerupere de nivel 0. Indicatorii sint pusi in stiva, IF si TF devin 0, CS este de asemenea pus in stiva, fiind apoi umplut cu valoarea de la adresa 2. Si IP curent este salvat si apoi incarcat cu valoarea de la adresa 0. Aceasta secventa include un apel lung la rutina de intreruperi ale carui segment si offset sint memorate la locatiile 2 si 0. Daca rezultatul incape atunci citul este memorat in AL sau AX (pentru operatii pe cuvint) si respectiv restul in AH sau DX.

IMUL - integer multiply accumulator by register-or-memory, signed (inmultire intreaga intre acumulator si registru sau memorie, cu semn)

Indicatori afectati: CF,OF

Descriere: Acumulatorul (AL pentru octet, AX pentru cuvint) e inmultit prin operandul specificat. Daca jumatatea superioara a rezultatului este extensia de semn a jumatatii inferioare, indicatorii carry si overflow sint stersi, altfel sint 1.

IN - input byte and input word (input de octet si input de cuvint)

Indicatori afectati: nici unul

Descriere: Continutul acumulatorului este inlocuit de continutul portului designat. Destinatia pentru input trebuie sa fie AX sau AL, si trebuie specificata cu scopul comunicarii asamblorului a tipului intrarii. Numele portului trebuie sa fie o valoare imediata intre 0 si 255 sau numele registrului DX care trebuie umplut mai devreme cu locatia portului.

INC - increment destination by one (incrementeaza destinatia cu unu)

Indicatori afectati: AF,OF,PF,SF,ZF

Descriere: Operandul specificat este adunat cu 1. Nu se genereaza carry.

INT – interrupt (intrerupere)

Indicatori afectati: IF,TF

Descriere: Pointer-ul de stiva este decrementat cu 2 si indicatorii sint salvati in stiva. Indicatorii de intrerupere si capcana sint pusi la 0, din nou SP e decrementat 2 iar continutul lui CS este salvat. CS este umplut cu partea semnificativa a vectorului de intreruperi (dublu cuvint), deci cu segmentul de baza al rutinei de intreruperi pentru acest tip de intreruperi. SP e din nou decrementat cu doi, de data asta se salveaza IP in stiva. IP va fi umplut cu cuvintul mai putin semnificativ al vectorului de intreruperi,locatat la adresa

absoluta TYPE*4. Astfel se completeaza un apel intersegment la procedura care prelu creaza acest tip de intrerupere (vezi de asemenea PUSHF, INTO, IRET).

INTO - interrupt if overflow (intrerupere daca exista overflow)

Indicatori afectati: nici unul

Descriere: Daca exista overflow pointer-ul de stiva este decrementat cu 2 si indicatorii sint salvati in stiva. Indicatorii de intrerupere si capcana sint pusi la 0, din nou SP e decrementat 2 iar continutul lui CS este salvat. CS este umplut cu partea semnificativa a vectorului de intreruperi(dublu cuvint), deci cu segmentul de baza al rutinei de intreruperi pentru tipul 4 de intreruperi. SP e din nou decrementat cu doi, de data asta se salveaza IP in stiva. IP va fi umplut cu cuvintul mai putin semnificativ al vectorului de intreruperi, locatat la adresa absoluta 16(10h). Astfel se completeaza un apel intersegment la procedura care prelucreaza acest tip de intrerupere (vezi de asemenea INT, IRET, PUSHF).

IRET - interrupt return (retur din intrerupere)

Indicatori afectati: toti

Descriere: IP este umplut cu cuvintul din capul stivei. Ca urmare pointer-ul de stiva este incrementat cu 2, si cuvintul din capul stivei este introdus in CS. Astfel se intoarce controlul in punctul in care a fost intilnita intreru perea. SP este din nou incrementat cu 2 si se refac indicatorii cu cuvintul din virful stivei. SP se incrementeaza din nou.

JA/JNBE - jump if not below nor equal, or jump if above (salt daca nu e mai mic nici egal, sau salt la mai mare)

Indicatori afectati: nici unul

Descriere: Daca atit indicatorul de carry cit si cel de zero sint 0 atunci distanta de la sfirsitul acestei instructiuni pina la eticheta tinta este adunata la IP, efectuind un transfer. Daca CF sau ZF sint 1 nu rezulta nici un salt.

JAE/JNB - jump if not below, or jump if above or equal(salt daca nu e mai mic,sau salt e mai mare sau egal)

Indicatori afectati: nici unul

Descriere: Daca indicatorul de carry este 0 atunci distanta de la sfirsitul acestei instructiuni pina la eticheta tinta este adunata la IP, efectuind un transfer. Daca CF eate 1 nu rezulta nici un salt.

JNAE/JB - jump if below, or jump if not above nor equal(salt daca e mai mic, sau salt daca nu e mai mare nici egal)

JC - jump if carry(salt daca exista carry)

Indicatori afectati: nici unul

Descriere: Daca indicatorul de carry este 1 atunci distanta de la sfirsitul acestei instructiuni pina la eticheta tinta este adunata la IP, efectuind un transfer. Daca CF eate 0 nu rezulta nici un salt.

Observatie: Eticheta tinta trebuie sa fie in intervalul -128 la +127 octeti fata de aceasta instruc tiune. Comparatiile si deci implicit relatiile (mai mici, mai mari) se refera la doua valori fara semn.

JNA/JBE - jump if below or equal, or jump if not above (salt daca e mai mic sau egal, sau salt daca nu e mai mare)

Indicatori afectati: nici unul

Descriere: Daca indicatorul de carry sau zero sint 1 atunci distanta de la sfirsitul acestei instructiuni pina la eticheta tinta este adunata la IP, efectuind un transfer. Daca CF si ZF sint 0 nu rezulta nici un salt.

Observatie: Eticheta tinta trebuie sa fie in intervalul -128 la +127 octeti fata de aceasta instructiune. Comparatiile si deci implicit relatiile (mai mici, mai mari) se refera la doua valori fara semn.

JCXZ - jump if CX is zero (salt daca CX este zero)

Indicatori afectati: nici unul

Descriere: Daca registrul numarator CX este 0 atunci distanta de la sfirsitul acestei instructiuni pina la eticheta tinta este adunata la
IP, efectuind un transfer. Daca CX este 1 nu rezulta nici un salt.

Observatie: Eticheta tinta trebuie sa fie in intervalul -128 la +127 octeti fata de aceasta instructiune. Comparatiile si deci implicit relatiile (mai mici, mai mari) se refera la doua valori fara semn.

JE/JZ - jump if equal, jump if zero(salt daca e egal, salt la zero)

Indicatori afectati: nici unul

Descriere: Daca ultima operatie care a afectat indicatorul zero a dat un rezultat zero atunci (ZF) va fi 1. Daca (ZF)=1 atunci distanta de sfirsitul acestei instructiuni pina la eticheta tinta este adunata la IP, efectuind un transfer. Daca ZF este 0 nu rezulta nici un salt.

Observatie: Eticheta tinta trebuie sa fie in intervalul -128 la +127 octeti fata de aceasta instructiune. Comparatiile si deci implicit relatiile (mai mici, mai mari) se refera la doua valori fara semn.

JNLE/JG - jump if not less nor equal, or jump if greater (salt daca nu e mai mic sau egal, sau salt la mai mare)

Indicatori afectati: nici unul

Descriere : Daca indicatorul zero este 0 si indicatorii sign si overflow sint egali atunci distanta de la sfirsitul acestei instructiuni pina la eticheta tinta este adunata la IP, efectuind un transfer. Daca ZF este 1 sau (SF)<>(OF) nu rezulta nici un salt.

Observatie: Eticheta tinta trebuie sa fie in intervalul -128 la +127 octeti fata de aceasta instructiune. Comparatiile si deci implicit relatiile (mai mici, mai mari) se refera la doua valori fara semn.

JNL/JGE - jump if not less, or jump if greater or equal (salt daca nu e mai mic, sau salt la mai mare sau egal)

Indicatori afectati: nici unul

Descriere: Daca indicatorii sign si overflow sint egali atunci distanta de la sfirsitul acestei instructiuni pina la eticheta tinta este adunata la IP, efectuind un transfer. Daca (SF)<>(OF) nu rezulta nici un salt.

Observatie: Eticheta tinta trebuie sa fie in intervalul -128 la +127 octeti fata de aceasta instructiune. Comparatiile si deci implicit relatiile (mai mici, mai mari) se refera la doua valori fara semn.

JL/JNGE - jump on less, or jump on not greater nor equal (salt la mai mic, sau salt daca nu e mai mare sau egal)

Indicatori afectati: nici unul

Descriere: Daca indicatorii sign si overflow nu sint egali (asta inseamna ca (SF) sau-exclusiv cu (OF) este 1 atunci distanta de la sfirsitul acestei instructiuni pina la eticheta tinta este adunata la IP, efectuind un transfer. Daca (SF)=(OF) nu rezulta nici un salt.

Observatie: Eticheta tinta trebuie sa fie in intervalul -128 la +127 octeti fata de aceasta instructiune. Comparatiile si deci implicit relatiile (mai mici, mai mari) se refera la doua valori fara semn.

JLE/JNG - jump if less or equal, or jump if not greater (salt daca e mai mic sau egal, sau salt daca nu e mai mare

Indicatori afectati: nici unul

Descriere: Daca indicatorii sign si overflow nu sint egali (asta inseamna ca (SF) sau-exclusiv cu (OF) este 1 sau daca indicatorul zero e setat atunci distanta de la sfirsitul acestei instructiuni pina la eticheta tinta este adunata la IP, efectuind un transfer.

Observatie: Eticheta tinta trebuie sa fie in intervalul -128 la +127 octeti fata de aceasta instructiune. Comparatiile si deci implicit relatiile (mai mici, mai mari) se refera la doua valori fara semn.

JMP - jump (salt)

Indicatori afectati: niciunul

Descriere: IP este inlocuit de offset-ul etichetei tinta in toate salturile inter-segment, acelasi lucru si pentru salturile indirecte in cadrul aceluiasi segment. Daca este un salt direct in acelasi segment atunci distanta de la sfirsitul instructiunii pina la eticheta tinta e adunata la IP. Salturile inter-segment inlocuiesc prima data continutul lui CS, utilizind cuvintul urmator instructiunii (direct) sau utilizind cuvintul urmator al adresei indicate (indirect).

JNA/JBE - jump if below or equal, or jump if not above (salt daca e mai mic sau egal, sau salt daca nu e mai mare

Indicatori afectati: nici unul

Descriere: Daca indicatorul carry sau zero este setat atunci distanta de la sfirsitul acestei instructiuni pina la eticheta tinta este adunata la IP, efectuind un transfer. Daca (CF)=0 si (ZF)=0 nu rezulta nici un salt.

Observatie: Eticheta tinta trebuie sa fie in intervalul -128 la +127 octeti fata de aceasta instructiune. Comparatiile si deci implicit relatiile (mai mici, mai mari) se refera la doua valori fara semn.

JNAE/JB - jump if below, or jump if not above nor equal (salt daca e mai mic, sau salt daca nu e mai mare nici egal)

Indicatori afectati: nici unul

Descriere: Daca indicatorul carry este setat atunci distanta de la sfirsitul acestei instructiuni pina la eticheta tinta este adunata la IP, efectuind un transfer. Daca (CF)=0 nu rezulta nici un salt.

Observatie: Eticheta tinta trebuie sa fie in intervalul -128 la +127 octeti fata de aceasta instructiune. Comparatiile si deci implicit relatiile (mai mici, mai mari) se refera la doua valori fara semn.

JNB/JAE - jump if not below , or jump if above or equal (salt daca nu e mai mic, sau salt daca e mai mare sau egal)

JNC - jump if no carry (salt daca nu e carry)

Indicatori afectati: nici unul

Descriere: Daca indicatorul carry este zero atunci distanta de la sfirsitul acestei instructiuni pina la eticheta tinta este adunata la IP, efectuind un transfer. Daca (CF)=1 nu rezulta nici un salt.

Observatie: Eticheta tinta trebuie sa fie in intervalul -128 la +127 octeti fata de aceasta instructiune. Comparatiile si deci implicit relatiile (mai mici, mai mari) se refera la doua valori fara semn.

JNBE - jump if not below nor equal (salt daca nu e mai mic nici egal)

Indicatori afectati: nici unul

Descriere: Daca nici indicatorul carry nici zero nu sint setate atunci distanta de la sfirsitul acestei instructiuni pina la eticheta tinta este adunata la IP, efectuind un transfer. Daca (CF)=1 sau (ZF)=1 nu rezulta nici un salt.

Observatie: Eticheta tinta trebuie sa fie in intervalul -128 la +127 octeti fata de aceasta instructiune. Comparatiile si deci implicit relatiile (mai mici, mai mari) se refera la doua valori fara semn.

JNE/JNZ - jump if not equal, or jump if not zero (salt daca nu e egal, sau salt daca nu e zero)

Indicatori afectati: nici unul

Descriere: Daca indicatorul zero nu e setat atunci distanta de la sfirsitul acestei instructiuni pina la eticheta tinta este adunata la IP, efectuind un transfer. Daca (ZF)=1 nu rezulta nici un salt.

Observatie: Eticheta tinta trebuie sa fie in intervalul -128 la +127 octeti fata de aceasta instructiune. Comparatiile si deci implicit relatiile (mai mici, mai mari) se refera la doua valori fara semn.

JNG/JLE - jump if not greater, or jump if less or equal (salt daca nu e mai mare, sau salt daca e mai mic sau egal)

Indicatori afectati: nici unul

Descriere: Daca indicatorul zero e setat, sau daca indicatorul sign nu e egal cu indicatorul overflow atunci distanta de la sfirsitul acestei instructiuni pina la eticheta tinta este adunata la IP, efectuind un transfer. Daca (ZF)=0 si (SF)=(OF) nu rezulta nici un salt.

Observatie: Eticheta tinta trebuie sa fie in intervalul -128 la +127 octeti fata de aceasta instructiune. Comparatiile si deci implicit relatiile (mai mici, mai mari) se refera la doua valori fara semn.

JNGE/JL - jump if less, or jump if not greater nor equal (salt daca e mai mic, sau salt daca nu e mai mare sau egal)

Indicatori afectati: nici unul

Descriere: Daca indicatorul sign nu e egal cu indicatorul overflow atunci distanta de la sfirsitul acestei instructiuni pina la eticheta tinta este adunata la IP, efectuind un transfer. Daca (SF)=(OF) nu rezulta nici un salt.

Observatie: Eticheta tinta trebuie sa fie in intervalul -128 la +127 octeti fata de aceasta instructiune. Comparatiile si deci implicit relatiile (mai mici, mai mari) se refera la doua valori fara semn.

JGE/JNL - jump if not less, or jump if greater or equal (salt daca nu e mai mic, sau salt daca e mai mare sau egal)

Indicatori afectati: nici unul

Descriere: Daca indicatorul sign e egal cu indicatorul overflow atunci distanta de la sfirsitul acestei instructiuni pina la eticheta tinta este adunata la IP, efectuind un transfer. Daca (SF)<>(OF) nu rezulta nici un salt.

Observatie: Eticheta tinta trebuie sa fie in intervalul -128 la +127 octeti fata de aceasta instructiune. Comparatiile si deci implicit relatiile (mai mici, mai mari) se refera la doua valori fara semn.

JG/JNLE - jump if not less nor equal, or jump if greater (salt daca nu e mai mic nici egal, sau salt daca e mai mare)

Indicatori afectati: nici unul

Descriere: Daca indicatorul zero e resetat si indicatorul sign e egal cu indicatorul overflow atunci distanta de la sfirsitul acestei instructiuni pina la eticheta tinta este adunata la IP, efectuind un transfer. Daca (ZF)=1 sau (SF)<>(OF) nu rezulta nici un salt.

Observatie: Eticheta tinta trebuie sa fie in intervalul -128 la +127 octeti fata de aceasta instructiune. Comparatiile si deci implicit relatiile (mai mici, mai mari) se refera la doua valori fara semn.

JNO - jump if not overflow (salt daca nu exista overflow)

Indicatori afectati: nici unul

Descriere: Daca indicatorul overflow este 0 atunci distanta de la sfirsitul acestei instructiuni pina la eticheta tinta este adunata la IP, efectuind un transfer. Daca (OF)=1 nu rezulta nici un salt.

Observatie: Eticheta tinta trebuie sa fie in intervalul -128 la +127 octeti fata de aceasta instructiune.

JNS - jump on not sign, jump if positive (salt daca nu exista sign, salt daca e valoare pozitiva)

Indicatori afectati: nici unul

Descriere: Daca indicatorul sign este 0 atunci distanta de la sfirsitul acestei instructiuni pina la eticheta tinta este adunata la IP, efectuind un transfer. Daca (SF)=1 nu rezulta nici un salt.

Observatie: Eticheta tinta trebuie sa fie in intervalul -128 la +127 octeti fata de aceasta instructiune.

JNZ/JNE - jump on not zero, jump if not equal (salt daca nu exista zero, salt daca nu e egalitate)

Indicatori afectati: nici unul

Descriere: Daca indicatorul zero este 0 atunci distanta de la sfirsitul acestei instructiuni pina la eticheta tinta este adunata la IP, efectuind un transfer. Daca (ZF)=1 nu rezulta nici un salt.

Observatie: Eticheta tinta trebuie sa fie in intervalul -128 la +127 octeti fata de aceasta instructiune.

JO - jump on overflow (salt daca exista overflow)

Indicatori afectati: nici unul

Descriere: Daca indicatorul overflow este 1 atunci distanta de la sfirsitul acestei instructiuni pina la eticheta tinta este adunata la IP, efectuind un transfer. Daca (OF)=0 nu rezulta nici un salt.

Observatie: Eticheta tinta trebuie sa fie in intervalul -128 la +127 octeti fata de aceasta instructiune.

JP/JPE - jump on parity, or jump if parity even (salt daca exista parity, sau daca paritatea e para)

Indicatori afectati: nici unul

Descriere: Daca indicatorul parity este 1 atunci distanta de la sfirsitul acestei instructiuni pina la eticheta tinta este adunata la IP, efectuind un transfer. Daca (PF)=0 nu rezulta nici un salt.

Observatie: Eticheta tinta trebuie sa fie in intervalul -128 la +127 octeti fata de aceasta instructiune.

JNP/JPO - jump on no parity, or jump if parity odd (salt daca nu exista parity, sau daca paritatea e impara)

Indicatori afectati: nici unul

Descriere: Daca indicatorul parity este 0 atunci distanta de la sfirsitul acestei instructiuni pina la eticheta tinta este adunata la IP, efectuind un transfer. Daca (PF)=1 nu rezulta nici un salt.

Observatie: Eticheta tinta trebuie sa fie in intervalul -128 la +127 octeti fata de aceasta instructiune.

JS - jump on sign (salt daca exista sign)

Indicatori afectati: nici unul

Descriere: Daca indicatorul sign este 1 atunci distanta de la sfirsitul acestei instructiuni pina la eticheta tinta este adunata la IP, efectuind un transfer. Daca (SF)=0 nu rezulta nici un salt.

Observatie: Eticheta tinta trebuie sa fie in intervalul -128 la +127 octeti fata de aceasta instructiune.

JZ/JE - jump if equal, jump if zero (salt daca exista egalitate, salt daca este zero)

Indicatori afectati: nici unul

Descriere: Daca indicatorul zero este 1 atunci distanta de la sfirsitul acestei instructiuni pina la eticheta tinta este adunata la IP, efectuind un transfer. Daca (ZF)=0 nu rezulta nici un salt.

Observatie: Eticheta tinta trebuie sa fie in intervalul -128 la +127 octeti fata de aceasta instructiune.

LAHF - load AH from flags (incarca AH cu indicatorii de conditie)

Indicatori afectati: nici unul

Descriere: Bitii registrului AH sint umpluti dupa cum urmeaza: indicatorul sign umple bitul 7; indicatorul zero bitul 6; indicatorul carry auxiliar bitul 4; indicatorul parity bitul 2; indicatorul carry bitul 0. Bitii 1, 3 si 5 a lui AH ramin nedeterminati.

LDS - load data segment (incarca segmentul de date)

Indicatori afectati: nici unul

Descriere: 1) Continutul registrului specificat este inlocuit de partea mai putin semnificativa a cuvintului adresat de operandul (de tip dublu cuvint) al instructiunii.

LEA - load effective address (incarca adresa efectiva)

Indicatori afectati: nici unul

Descriere: Continutul registrului specificat este inlocuit de offset-ul variabilei indicate sau a etichetei sau a expresiei de tip adresa.

LES - load extra-segment register (incarca registrul de segment auxiliar)

Indicatori afectati: nici unul

Descriere: 1) Continutul registrului specificat este inlocuit de partea mai putin semnificativa a cuvintului adresat de operandul (de tip dublu cuvint) al instructiunii. (REG)=(EA) 2) Continutul registrului ES este inlocuit de partea semnificativa a cuvintului adresat de operandul (de tip dublu cuvint) al instructiunii. (ES)=(EA+2)

LOCK

Indicatori afectati: nici unul

Descriere: Orice instructiune poate fi precedata de un octet special de tip "lock". El face ca procesorul sa serveasca semnalul de "bus-lock" (magistrala ocupata)

pe timpul de executie al instructiunii. In sistemele cu procesoare multiple care folosesc in comun resursele este necesar sa se asigure un mecanism de control al accesului la aceste resurse. Se presupune ca hardware-ul extern, dupa receptia acestui semnal va asigura accesul la magistrala pentru alti "masteri" in timpul perioadei de asertiune a lui "bus-lock".

LODS - load byte or word string (incarca sir de octeti sau cuvinte)

Indicatori afectati: nici unul

Descriere: Octetul sursa (sau cuvintul) este incarcat in AL (sau AX). Indexul sursa este incrementat cu 1 (sau 2 pentru siruri de cuvinte) daca indicatorul direction este 0; altfel SI e decrementat cu 1 (sau 2).

LOOP - loop, or iterate instruction sequence until count complete (bucla, sau secventa de iterare a instructiunilor pina la epuizarea numaratorului)

Indicatori afectati: nici unul

Descriere: Registrul numarator (CX) este decrementat cu 1. Daca noul CX nu e 0, atunci distanta de la sfirsitul acestei instructiuni pina la eticheta tinta este adunata la IP, efectuind un salt. Daca CX=0 , nu apare nici un salt.

LOOPE/LOOPZ - loop on equal, or loop on zero (bucla la egal, sau bucla la zero)

Indicatori afectati: nici unul

Descriere: Registrul numarator (CX) este decrementat cu 1. Daca noul CX nu e 0 si indicatorul zero este 1, atunci distanta de la sfirsitul acestei instructiuni pina la eticheta tinta este adunata la IP, efectuind un salt. Daca CX=0 sau daca (ZF)=0 nu apare nici un salt.

LOOPNE/LOOPNZ - loop on not equal, or loop on not zero (bucla la neegal, sau bucla la nezero

Indicatori afectati: nici unul

Descriere: Registrul numarator (CX) este decrementat cu 1. Daca noul CX nu e 0 si indicatorul zero este 0, atunci distanta de la sfirsitul acestei instructiuni pina la eticheta tinta este adunata la IP, efectuind un salt. Daca CX=0 sau daca (ZF)=1 nu apare nici un salt.

MOV – move (muta)

Indicatori afectati: nici unul

Descriere: Exista 7 tipuri distincte de instructiuni de transfer. Fiecare tip are utilizari multiple depinzind de tipul datelor de mutat si de locatia acestor date.

TIP 1: in memorie de la acumulator
TIP 2: in acumulator din memorie
TIP 3: in registru de segment din operand de tip memorie/registru
TIP 4: in registru/memorie din registru segment
TIP 5: (a) in registru din registru
TIP 6: in registru din data imediata
TIP 7: in memorie / registru din data imediata

MOVS - move byte string or move word string (muta sir de octeti sau muta sir de cuvinte)

Indicatori afectati: nici unul

Descriere: Sirul sursa al carui offset se gaseste in SI este incarcat in locatia din segmentul auxiliar al carui offset este in DI. SI si DI sint amindoua incrementate, daca indicatorul direction este 0, sau amindoua decrementate daca (DF)=1. Incrementul sau decrementul e 1 pentru siruri de octeti si 2 pentru siruri de cuvinte.

NEG - negate, or form 2's complement (neaga, sau formeaza complementul fata de 2)

Indicatori afectati: AF,CF,OF,PF,SF,ZF

Descriere: Operandul specificat este scazut din 0FFH pentru octeti sau 0FFFFH pentru cuvinte. Se adauga 1 si rezultatul este memorat in operandul dat.

NOP - no operation (nici o operatie)

Indicatori afectati: nici unul

Descriere: NOP nu determina nici o operatie dar tine 3 perioade de ceas. Urmatoarea instructiune din secventa este apoi executata.

NOT - not, or form 1's complement (nu, sau formeaza complementul fata de 1)

Indicatori afectati: nici unul

Descriere: Operandul specificat este scazut din 0FFH pentru octeti sau 0FFFFH pentru cuvinte. Rezultatul este memorat in operandul dat.

OR - or, inclusive (sau, inclusive)

Indicatori afectati: CF,OF,PF,SF,ZF

Descriere: Fiecare pozitie de bit in operandul destinatie (stinga) devine 1, pina cind atit el cit si bitul corespunzator din operandul sursa (dreapta) sint 0. Indicatorii carry si overflow devin 0.

OUT - output byte and output word (output de octet si output de cuvint)

Indicatori afectati: nici unul

Descriere: Continutul portului designat este inlocuit de continutul acumulatorului.

POP - pop word off stack into destination (sterge un cuvint din stiva si pune-l in destinatie)

Indicatori afectati: nici unul

Descriere: POP transfera un cuvint de la locatia din stiva adresata de SP la operandul destinatie si incrementeaza SP cu 2.

POPF - pop flags off stack (reface indicatorii din stiva)

Indicatori afectati: toti

Descriere: Indicatorii = ((SP)+1:(SP)), (SP)=(SP)+2 Registrul de indicatori sunt umpluti cu pozitiile corespunzatore de bit din cuvintul din virful stivei: overflow = bit 11, direction = bit 10, interrupt = bit 9, trap = bit 8, sign = bit 7, zero = bit 6, auxiliary carry = bit 4, parity = bit 2, carry = bit 0. SP este apoi incrementat cu 2.

PUSH - push word onto stack (salveaza cuvint in stiva)

Indicatori afectati: nici unul

Descriere: 1) pointerul de stiva este decrementat cu 2, (SP)=(SP)-2 2) continutul destinatiei este pus in cuvintul din virful stivei

PUSHF - push flags on stack (salveaza indicatorii in stiva)

Indicatori afectati: nici unul

Descriere: SP este decrementat cu 2, apoi indicatorii inlocuiesc bitii corespunzatori ai cuvintului din virful stivei (vezi POPF). (SP)=(SP)-2, ((SP)+1:(SP))=indicatorii

RCL - rotate left through carry (roteste stinga cu carry)

Indicatori afectati: CF,OF

Descriere: Operandul specificat ca destinatie (stinga) e rotit la stinga impreuna cu carry de un numar de ori (COUNT). Acest numar este sau exact 1, specificat de numarul absolut 1, sau este numarul tinut in registrul CL, specificat explicit ca operand. Rotatia continua pina cind COUNT=0. CF este pastrat si e rotit in

bitul 0 al destinatiei. Bitul cel mai semnificativ al destinatiei e rotit in CF. Daca COUNT=1 si cei doi biti mai semnificativi ai destinatiei au valori neegale atunci indicatorul overflow devine 1. Daca COUNT<>1, OF e nedefinit.

RCR - rotate right through carry (roteste dreapta cu carry)

Indicatori afectati: CF,OF

Descriere: Operandul specificat ca destinatie (stinga) e rotit la dreapta impreuna cu carry de un numar de ori (COUNT). Acest numar este sau exact 1, specificat de numarul absolut 1, sau este numarul tinut in registrul CL, specificat explicit ca operand. Rotatia continua pina cind COUNT=0. CF este pastrat si e rotit in bitul cel mai semnificativ al destinatiei. Bitul 0 e rotit in CF. Daca COUNT=1 si cei doi biti mai semnificativi ai destinatiei au valori neegale atunci indicatorul overflow devine 1. Daca COUNT<>1, OF e nedefinit.

REP/REPZ/REPE/REPNZ - repeat string operation (repeta operatiile pe siruri)

Indicatori afectati: depind de operatiile pe sir realizate

Descriere: Operatia pe sir specificata este realizata de un numar de ori, pina cind CX devine 0. CX este decrementat cu 1 dupa fiecare operatie. Operatiile de comparare si scanare a sirurilor determina o iesire din bucla daca indicatorul zero nu e egal cu valoarea bitului 0 al acestui octet de instructiune.

RET - return from procedure (intorcere din procedura)

Indicatori afectati: nici unul

Descriere: Pointerul de instructiune este inlocuit de cuvintul din virful stivei. SP este incrementat cu 2. Pentru intoarcerea din alt segment, registrul CS este inlocuit cu cuvintul acum in virful stivei si SP este din nou incrementat cu 2. Daca s-a specificat o valoare imediata in instructiunea RET aceasta valoare este adunata la SP.

ROL - rotate left (roteste stinga)

Indicatori afectati: CF,OF

Descriere: Operandul specificat ca destinatie (stinga) e rotit la stinga impreuna cu carry de un numar de ori (COUNT). Acest numar este sau exact 1, specificat de numarul absolut 1, sau este numarul tinut in registrul CL, specificat explicit ca operand. Rotatia continua pina cind COUNT=0. CF este pierdut. Bitul cel mai semnificativ al destinatiei e rotit in CF. Daca COUNT=1 si cei doi biti mai semnificativi ai destinatiei au valori neegale atunci indicatorul overflow devine 1. Daca COUNT<>1, OF e nedefinit.

ROR - rotate right (roteste dreapta)

Indicatori afectati: CF,OF

Descriere: Operandul specificat ca destinatie (stinga) e rotit la dreapta impreuna cu carry de un numar de ori (COUNT). Acest numar este sau exact 1, specificat de numarul absolut 1, sau este numarul tinut in registrul CL, specificat explicit ca operand. Rotatia continua pina cind COUNT=0. CF este pierdut. Bitul cel mai putin semnificativ al destinatiei e rotit in CF. Daca COUNT=1 si cei doi biti mai semnificativi ai destinatiei au valori neegale atunci indicatorul overflow devine 1. Daca COUNT<>1, OF e nedefinit.

SAHF

Indicatori afectati: AF,CF,PF,SF,ZF

Descriere: Cei cinci indicatori specificati sint inlocuiti de bitii specifici din AH. (SF)=bit 7, (ZF)=bit 6, (AF)=bit 4, (PF)=bit 2, (CF)=bit 0;(SF):(ZF):X:(AF):X:(PF):X:(CF)=(AH)

SHL/SAL - shift arithmetic left and shift logic left (muta la stinga aritmetic, si muta la stinga logic)

Indicatori afectati: CF,OF,PF,SF,ZF

Descriere: Operandul specificat ca destinatie (stinga) e deplasat la stinga de un numar de ori (COUNT). Acest numar este sau exact 1, specificat de numarul absolut 1, sau este numarul tinut in registrul CL, specificat explicit ca operand. Deplasarea continua pina cind COUNT=0. CF este pierdut. Bitul cel mai semnificativ al destinatiei e deplasat in CF. Bitul cel mai putin semificativ e umplut cu 0. Daca COUNT=1 si cei doi biti mai semnificativi ai destinatiei au valori neegale atunci indicatorul overflow devine 1. Daca COUNT<>1, OF e nedefinit.

SAR - shift arithmetic right (muta la dreapta aritmetic)

Indicatori afectati: CF,OF,PF,SF,ZF

Descriere: Operandul specificat ca destinatie (stinga) e deplasat la dreapta de un numar de ori (COUNT). Acest numar este sau exact 1, specificat de numarul absolut 1, sau este numarul tinut in registrul CL, specificat explicit ca operand. Deplasarea continua pina cind COUNT=0. CF este pierdut. Bitul cel mai putin semnificativ al destinatiei e deplasat in CF. Bitul cel semificativ e umplut cu 0. Daca COUNT=1 si cei doi biti mai semnificativi ai destinatiei au valori neegale atunci indicatorul overflow devine 1. Daca COUNT<>1, OF e nedefinit.

SBB - subtract with borrow (scade cu imprumut)

Indicatori afectati: AF,CF,OF,PF,SF,ZF

Descriere: Operandul sursa este scazut din operandul destinatie (stinga). Daca indicatorul carry era setat, se scade unu din rezultatul de mai sus. Rezultatul inlocuieste operandul destinatie original.

SCAS - scan byte string or scan word string (scaneaza siruri de octeti sau scaneaza siruri de cuvinte)

Indicatori afectati: AF,CF,OF,PF,SF,ZF

Descriere: Elementul de sir specificat de DI in segmentul ES este scazut din valoarea existenta in acumulator, operatia afectind numai indicatorii. DI este incrementat (daca indicatorul direction este zero) sau decrementat (daca (DF)=1) cu 1 pentru octet sau 2 pentru cuvinte.

SHR - shift logic right (muta la dreapta logic)

Indicatori afectati: CF,OF,PF,SF,ZF

Descriere: Operandul specificat ca destinatie (stinga) e deplasat la dreapta de un numar de ori (COUNT). Acest numar este sau exact 1, specificat de numarul absolut 1, sau este numarul tinut in registrul CL, specificat explicit ca operand. Deplasarea continua pina cind COUNT=0. CF este pierdut. Bitul cel mai putin semnificativ al destinatiei e deplasat in CF. Bitul cel mai semificativ e umplut cu 0. Daca COUNT=1 si cei doi biti mai semnificativi ai destinatiei au valori neegale atunci indicatorul overflow devine 1. Daca COUNT<>1, OF e nedefinit.

STC - set carry flag (seteaza indicatorul carry)

Indicatori afectati: CF

Descriere: Indicatorul carry este setat la 1.

STD - set direction flag (seteaza indicatorul directie)

Indicatori afectati: DF

Descriere: Indicatorul directie este setat la 1.

STI - set interrupt flag (seteaza indicatorul intrerupere)

Indicatori afectati: IF

Descriere: Indicatorul intrerupere este setat la 1.

STOS - store byte string or store word string (memoreaza sir de octeti sau sir de cuvinte)

Indicatori afectati: nici unul

Descriere: Octetul (sau cuvintul) din AL (sau AX) inlocuieste continutul octetului sau cuvintului adresat de DI in ES. Apoi DI este incrementat daca indicatorul direction este 0 sau decrementat daca DF=1. Se va schimba valoarea cu 1 pentru octeti si 2 pentru cuvinte.

SUB - subtract (scadere)

Indicatori afectati: AF,CF,OF,PF,SF,ZF

Descriere: Operandul sursa este scazut din operandul destinatie (stinga). Rezultatul inlocuieste operandul destinatie original.

TEST - test, or logical compare (testeaza, sau compara logic)

Indicatori afectati: CF,OF,PF,SF,ZF

Descriere: Cei doi operanzi sint supusi unui "si" logic pentru a afecta indicatorii dar nici unul din operanzi nu este afectat. Indicatorii carry si overflow devin 0.

WAIT – wait (asteapta)

Indicatori afectati: nici unul

Descriere: Nu se efectueaza nici o operatie. WAIT determina intrarea procesorului in starea wait daca pinul TEST nu e asignat. Starea WAIT poate fi intrerupta de o intrerupere externa. Cind aceasta se intimpla locatia de cod salvata e aceea a instructiunii WAIT, astfel incit dupa intoarcerea din intrerupere se revine in starea wait. Starea wait este parasita cind se furnizeaza semnalul TEST. Se reia astfel executia si nu se permit intreruperi pina cind nu se intra in executia instructiunii urmatoare. Instructiunea permite astfel procesorului sa se sincronizeze cu hardware extern.

XCHG – exchange (schimba)

Indicatori afectati: nici unul

Descriere: Exista doua forme pentru instructiunea XCHG, una pentru comutarea continuturilor acumulatorului cu acela al altor registre generale, si una pentru comutarea registrelor cu un operand de tip registru sau memorie. 1) Continutul destinatiei e memorat temporar intr-un registru intern de lucru (temp)=DEST ; 2) Continutul destinatiei e inlocuit de continutul operandului (DEST)=(SRC) 3) Continutul anterior al destinatiei este mutat din registrul de lucru in operandul sursa (SRC)=(temp)

XLAT – translate (translateaza)

Indicatori afectati: nici unul

Descriere: Continutul acumulatorului este inlocuit de octetul din tabela. Adresa de inceput a tabelei a fost mutata in registrul BX. Continutul original a lui AL este numarul de octeti de dupa adresa de start, unde trebuie gasit octetul dorit a fi translatat. El inlocuieste continutul lui AL.

XOR - exclusive or (sau exclusiv)

Indicatori afectati: CF,OF,PF,SF,ZF

Descriere: Fiecare pozitie de bit in operandul destinatie (stinga) devine 0, daca pozitiile corespunzatoare din ambii operanzi sint egale. Daca sint neegale atunci aceea pozitie de bit devine 1. Indicatorii carry si overflow devin 0.

2.3. Extinderea structurii unității centrale la familia 80x86

2.3.1. Unitatea centrală 80x86 din punct de vedere al programatorului

Se vor discuta în acest capitol procesoarele reale 8088/8086, 80188/80186, 80286, şi 80386/80486/80586/Pentium. Dintre componentele hardware ale sistemului de calcul cea mai importantă rămâne unitatea centrală din punct de vedere al programării în limbaj de asamblare.

Cele mai utilizate componente ale unității centrale sunt registrele şi acestea au o importanță deosebită în programarea în limbaj de asamblare. Vom prezenta în continuare, pe larg, modul de utilizare a registrelor unității centrale. Fiecare procesor din familia 80x86 conține un set de registre. Particularitatea acestei familii de procesoare este reprezentată de faptul că un procesor conține un superset de regiştrii ai procesorului precedent. Punctul de plecare îl reprezintă setul de registre al unităților centrale ale procesoarelor 8088, 8086, 80188 şi 80186 deoarece cele patru tipuri de procesoare au acelaşi tip de registre. În cele prezentate în continuare termenul de "8086" se va referi de fapt la oricere dintre aceste procesoare.

Fabricantul acestor procesoare, firma INTEL, împarte registrele unității centrale ale procesorului 8086 în trei categorii:

- registre de uz general,
- registre de segment,
- registre cu destinație specială.

Registrele de uz general sunt cele care pot apărea ca operanzi în operațiile aritmatice, logice şi în instrucțiunile legate de acestea. Deşi aceste registre sunt denumite "de uz general", fiecare dintre ele se utilizează într-un anumit scop implicit dar destinația acestora poate fi schimbată explicit de programator. Registrele segment sunt utilizate pentru accesarea unor blocuri de memorie numite segmente. Registrele de uz special au destinații diverse. Dintre acestea, două prezină o importanță deosebită şi vor fi prezentate pe scurt în continuare.

2.3.2. Registrele de uz general ale unității centrale 8086

Unitatea centrală a procesorului 8086 are opt registre de uz general, de câte 16 biți fiecare, notate: **ax, bx, cx, dx, si, di, bp** şi **sp**. Deşi în calcule se pot folosi oricare din aceste registre, multe instrucțiuni lucrează mai eficient iar altele chiar impun utilizarea unui anumit registru. Din acest motiv, denumirea "de uz general" dată acestor registre nu este chiar potrivită.

Registrul **ax** (registrul Acumulator) este registrul în care au loc majoritatea calculelor aritmetice şi logice. Deşi operațiile aritmetice şi logice pot fi efectuate şi cu ajutorul altor registre, cel mai eficint este să se folosească registrul **ax.**

Registrul **bx** (registrul Bază) are şi el o destinație specială. Acest registru este folosit pentru a stoca adresa indirectă (la acest procesor mai mult ca la procesoarele din familia x86).

Registrul **cx** (registrul Contor) este utilizat de regulă pentru contorizări la bucle sau pentru a stoca dimensiunea şirurilor.

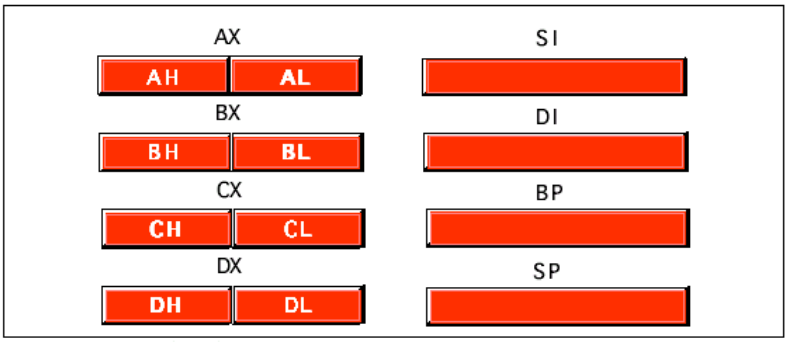
Registrul **dx** (registrul Date) are în general două destinații: el stochează depăşirile pentru anumite operații aritmetice sau stochează adresa portului I/O pa accesarea perifericelor.

Registrele **si** şi **di** (registrul Index Sursă şi registrul Index Destinație) au de asemenea mai multe destinații speciale. Registrele pot fi folosite ca pointer (indicator) la adresarea indirectă a memoriei (similar cu registrul **bx**) sau pot fi folosite în operațiile pe şiruri.

Registrul **bp** (registrul Pointerul – Indicatorul - Bazei) este similar registrului **bx**. El va fi în general utilizat pentru accesarea perametrilor şi a variabilelor locale dintr-o procedură.

Registrul **sp** (registrul Pointer (Indicator) Stivă) are o destinație foarte importantă: el păstrează stiva programului. În mod normal acest registru nu trebuie folosit de programator pentru calcule aritmetice. Funcționarea corectă a celor mai multe programe depinde în mod esențial de utilizarea corectă a acestui registru.

Primele patru registre **ax,bx,cx** şi **dx** ale unității centrale 8086 pot fi folosite de asemenea ca registre pe opt biți. Aceste registre sunt denumite: **al, ah, bl, bh, cl, ch, dl** şi **dh**. Denumirile se referă la partea superioară sau inferioară a registrelor pe 16 biți aşa cum este prezentat în figura următoare.



Este de notat faptul că registrele pe 8 biți nu sunt registre independente. O modificare în registrul **al**, de exemplu, va modifica şi registrul **ax**; la fel şi dacă va fi modificat registrul **ah.** Este evident că şi modificarea registrului **ax** va duce la modificarea registrelor **ah** şi **al**. Este de asemenea de remarcat faptul că modificarea registrului **al** nu va afecta registrul **ah** şi invers.

Registrele **si, di, bp** şi **sp** sunt registre numai pe 16 biți.

2.3.3. Registrele de segment 8086

Procesorul 8086 are patru registre de segment: **cs, ds, es** şi **ss**. Numele lor sunt respectiv: registrul segment de cod (Code Segment), registrul segment de date (Data Segment), registrul segment de date suplimentar (Extra Segment) şi registrul segment de stivă (Stack Segment). Toate aceste registre au dimensiunea de 16 biți şi ele permit selectarea blocurilor (segmentelor) din memoria principală. Un registru segment indică (conține) adresa de început a unui segment de memorie. Segmentul de memorie la 8086 nu poate avea o dimensiune mai mare de 65536 octeți, adică are maximum 64 de Kocteți.

Registrul **cs** indică segmentul de memorie ce conține instrucțiunile maşină ce sunt executate la un moment dat. Deşi un segment are dimensiunea unui segment este

de maximum 64 Kocteţi, programele pot avea dimensiuni mai mari de 64 de Kocteţi. Acest lucru se realizează prin folosirea mai multor segmente şi comutarea între aceste segmente prin schimbarea conţinutului registrului **cs**.

Registrul **ds** indică în general segmentul ce conţine datele globale ale programului. Şi aici putem face aceeaşi observaţie, faptul că datele unui program nu trebuie să se limiteze la maximum 64 de Kocteţi.

Registrul **es** indică un segment suplimentar numit extrasegment. Programele scrise pentru 8086 folosesc adesea acest registru pentru a avea acces la alte segmente atunci când este dificil sau imposibil să se modifice alte registre segment.

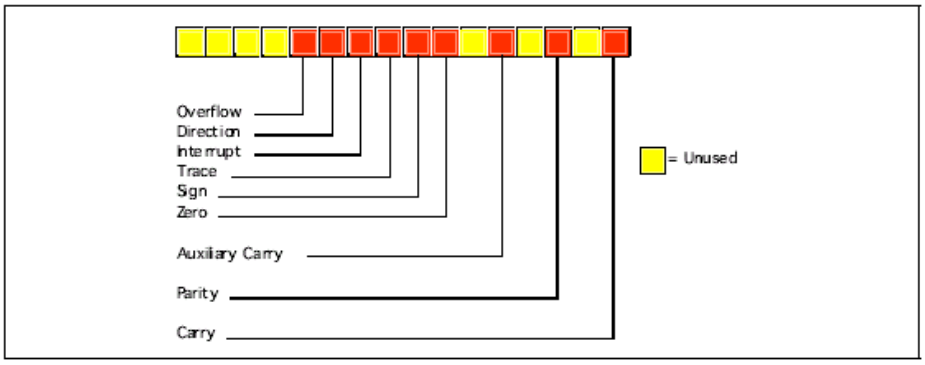
Registrul **ss** indică segmentul unde se află stiva 8086. Stiva reprezintă locul unde 8086 stochează informaţii importante cu privire la starea maşinii, adresele de reîntoarcere din subprograme, parametrii procedurilor şi variabile locale. În general conţinutul registrului segment de stivă nu trebuie modificat din cauză ca multe date importante ale sistemului depind de acesta. De asemenea este posibil să se stocheze date în segmentul de stivă dar acest lucru nu trebuie făcut niciodată deoarece conţinutul stivei reprezintă indicatoare la zone de memorie accesibilă ăi o încercare de a folosi stiva în alte scopuri poate crea probleme considerabile în special cînd folosiţi unităţi centrale mai evoluate cum este, spre exemplu, 80386.

2.3.4. Registrele de uz special

Unitatea centrală a procesorului 8086 are două registre cu destinaţie specială: contorul de program **ip** (instruction pointer) şi registrul bistabililor de condiţii. Aceste registre nu pot fi accesate în acelaşi fel cu celelalte registre ale unităţii centrale 8086. De regulă unitatea centrală conrolează în mod direct aceste registre.

Registrul **ip** este echivalent cu registrul **ip** al procesoarelor x86 – el conţine adresa instrucţiunii curente în execuţie. Registrul **ip** este un registru pe 16 biţi care indică adresa din segmentul de cod curent (cu 16 biţi pot fi selectate 65536 de locaţii de memorie diferite).

Registrul bistabililor de condiţii (sau a fanioanelor de condiţii) este diferit de celelalte registre ale unităţii centrale 8086 care pot memora valor de 8 sau 16 biţi. Registrul bistabililor de condiţii este de fapt o colecţie de bistabile, fiecare dintr acestea ajutând la determinarea stării curente a procesorului. Deşi registrul bistabililor de condiţii are o dimensiune de 16 biţi, 8086 nu foloseşte decât nouă dintre aceştia. Patru fanioane sunt folosite în mod frecvent la programare: zero, carry, sign şi overflow. Aceste fanioane mai sunt denumite şi coduri de condiţii. Registrul bistabililor de condiţii este prezentat mai jos.



2.3.5. Registrele 80286

La 80286 apar modificări consistente la componentele vizibile programatorului în modul protejat. Totuşi nu vom discuta aici despre modul protejat la 80286 pentru că acest mod este folosit doar în cazuri speciale. Cu toate acestea se vor prezenta registrele suplimentare şi bistabilii de stare ce apar în plus în caz că vă veţi întâlnii cu aceştia.

În registrul bistabililor de condiţii la 80286 apar trei bistabili suplimentari. Nivelul privilegiat pentru operaţii I/O are doi biţi (biţii 12 şi 13) şi specifică unul din cele patru nivele de privilegii posibile pentru realizarea operaţiilor I/O. Aceşti doi biţi conţin în general valoarea 00_b când 80286 lucrează în modul real (modul 8086 emulat). Bistabilul NT (nested task) controlează operaţiile realizate de instrucţiune de reîntoarcere din întrerupere (IRET). În mod normal NT este zero în programele ce lucrează în modul real.

În afară de biţii suplimentari din registrul bistabililor de condiţii, 80286 mai are cinci registre suplimentare folosite de sistemul de operare pentru gestionarea memoriei şi a mai multor procese: the machine status word (msw), the global descriptor table register (gdtr), the local descriptor table register (ldtr), the interrupt descriptor table register (idtr) and the task register (tr).

În modul protejat la procesorul 80286 poate fi accesată o memorie mai mare de un megaoctet. Datorită faptului că procesorul este depăşit această metodă este rareori folosită de programatori.

2.3.6. Registrele procesoarelor 80386/80486

La procesorul 80386 a fost extins în mod semnificativ setul de registre. Acesta conţine toate registrele procesorului 80286 (şi implicit 8086) dar are câteva registre suplimentare şi definirea registrelor existente a fost extinsă. Procesorul 80486 nu are registre suplimentare faţă de 80386 dar are definiţi câţiva biţi rămaşi nedefiniţi la 80386.

Cea mai importantă schimbare din punct de vedere al programatorului la procesorul 80386 a fost introducerea setului de registre de 32 de biţi. Registrele **ax, bx, cx, dx, si, di, bp, sp,** registrul bistabililor de condiţii şi **ip** sunt extinse la 32 de biţi. La 80386 aceste registre se numesc **eax, ebx, ecx, edx, esi, edi, ebp, esp, eflags,** şi **eip** pentru a le diferenţia de varianta de 16 biţi (care sunt şi ele disponibile la 80386). Pe lângă registrele de 32 de biţi 80386 are de asemenea două registre segment noi de 16 biţi numite **fs** şi **gs** care permit programatorului să acceseze simultan şase segmente de memorie diferite fără a fi necesară reîncărcarea registrelor segment. Trebuie făcută observaţia că la 80386 registrele de segment au rămas toate pe 16 biţi.

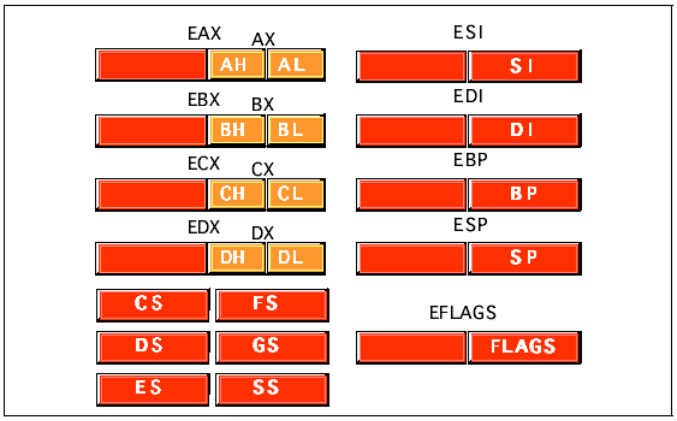
În registrul bistabililor de condiţii nu s-a făcut nici o modificare dar acesta a fost extins la 32 de biţi (**eflag**) şi au fost definiţi biţii 16 şi 17. Bitul 16 este fanionul de începere a depanării (RF) utilizat de registrele de depanare ale lui 80386. Bitul 17 este fanionul pentru modul virtual (VM) care semnalează dacă procesorul lucrează în modul virtual 86 (care simulează un procesor 8086) sau în modul protejat standard. Procesorul 80486 adaugă un al treilea bit în registrul **eflags** pe poziţia 18, fanionul de verificare a alinierii. Împreună cu registrul de control zero (CR0) din 80486, acest fanion forţează o întrerupere (abandon program) atunci când procesorul accesează o dată nealiniată (de exemplu, un cuvânt de la o adresă impară sau un dublu cuvânt de la o adresă care nu este multiplu de patru).

Procesorul 80386 are suplimentar patru registre de control **CR0-CR3.** Aceste registre constituie o extensie a registrului **msw** a lui 80286 (80386 emulează registrul **msw** a lui 80286 pentru compatibilitate dar informațiile apar în realitate în registrele **CRx**). La 80386 şi 80486 aceste registre controlează cuncții cum ar fi gestionarea memoriei paginate, perații de activare/sezactivare a memoriei cache (numai la 80486), opararea în mod protejat şi altele.

Procesoarele 80386/486 au de asemenea opt registre de deapanare suplimentare. Un program de depanare cum sunt Microsoft Codeview sau Turbo Debugger poate utiliza aceste registre pentru a seta puncte de întrerupere cînd se încearcă localizarea unei erori într-un program. Deşi aceste registre nu suntutilizate în programe ele sunt foarte utile în depanatoare pentru găsirea şi eliminarea rapidă a erorilor.

În sfârşit, procesoarele 80386/486 au suplimentar o serie de registre de test care testează funcționarea corectă a procesorului când sistemul este pornit. Cel mai probabil Intel a pus aceste registre pentru testarea imediat după fabricție dar proiectanții de sistem pot folosi avantajul oferit de aceste registre la testul power-on.

Pentru marea majoritate a programatorilor în limbaj de asamblare registrele suplimentare aparute la procesoarele 80386/486/Pentium nu prezintă o prea mare importanță. Oricum, extensia la 32 de biți şi registrele extrasegment sunt destul de folositoare. Pentru programatorii de aplicații, modelul de programare pentru procesoarele 80386/486/Pentium este cel prezentat în figura următoare.



2.3.7. Organizarea memoriei fizice la 80x86

Într-un sistem de calcul Von Neumann unitatea centrală este conectată la memorie prin intermediul unei magistrale. Procesorul 80x86 selectează un anumit element de memorie prin trimiterea unui valori binare pe magistrala de adrese. Din alt punct de vedere memoria reprezintă o matrice de octeți. O structură de date în Pascal care este similară unei memorii va fi:

Memmory : array [0..MaxRAM] of byte;

Valoarea de pe magistrala de adrese corespunde indexului furnizat acestei matrici. De exemplu, scrierea unei date în memorie este echivalent cu:

Memmory[address] := Value_to_write;

Citirea unei date din memorie este echivalentă cu:

Value_read := Memmory[address];

În funcție de tipul unității centrale numarul maxim de locații de memorie (spațiul maxim de adresare) este diferit. De exemplu, 80386 are o magistrala cu 32 de linii de adresă ceea ce înseamnă că poate adresa până la patru gigaocteți de memorie. De asemenea, nu este obligatoriu ca întreg spațiul maxim de adresare să fie acoperit cu memorie fizică existentă în sistem.

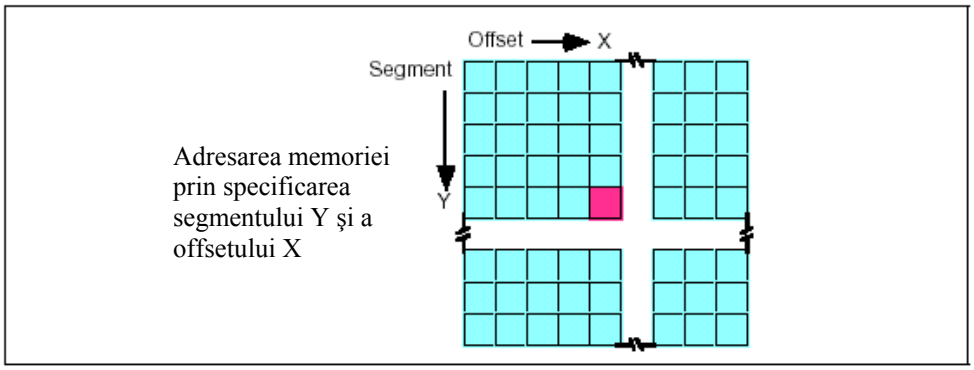
Primul megabit de memorie, de la adresa zero la 0FFFFFh este special pentru 80x86. Acesta corespunde spațiului maxim adresabil la procesoarele 8088, 80186 şi 80188. Cele mai multe programe DOS limitează dimensiunea codului ăi a datelor la acest domeniu. Adresele limitate la acest domeniu se numesc adrese reale după modul real 80x86.

2.3.8. Segmentele la 80x86

Pentru a putea înțelege adresarea memoriei la procesoarele 80x86 trebuie discutat mai înâi mecanismul segmentării. Mecanismul segmentării furnizează un mecanism puternic de gestionare a memoriei, Acesta permite progrmatorilor să partiționeze programele în module care pot opera independent unul de celălalt. Segmentele furnizează de asemenea o cale de implementare simplă a programelor orietate pe obiecte. O altă facilitate a segmentării este aceea că permite simplificarea utilizării în comun a datelor de către două procesoare. În concluzie segmentarea este o facilitate puternică care poate ridica însă unele probleme la realizarea programelor.

Principalele probleme de care trebuie ținut cont la utilizarea segmentării sunt sistemul de operare utilizat şi tipul de procesor. Dacă sistemul de operare DOS impune o anumită limită şi procesoarele care pot face adresarea pe 16 sau 32 de biți ridică unele probleme.

Dacă vom considera memoria ca un vector liniar atunci adresarea poate fi făcută prin furnizarea adresei (indexului) curente în spațiul maxim de adresare. Acest mod de adresare se numeşte adresare liniară. Adresarea segmentată necesită două componente pentru a specifica o locație de memorie: o valoare de segment şi o valoare a ofsetului în segmentul respectiv. Ideal ar fi ca cele două valori să fie independente una de cealaltă. Cel mai simplu mod de a descrie adresarea segmentată este să considerăm o matrice bidimensională. Valoarea segmentului furnizează un indice iar ofsetul celălalt indice din matrice, conform figurii următoare.



Să explicăm care este avantajul uei astfel de structuri. Să presupunem că se scrie un program în care este necesară o rutină care să calculeze funcția SIN(X). Vor fi necesare o serie de variabile temporare care cel mai probabil nu vor fi folosite ca variabile global ci ca variabile locale în interiorul rutinei de calcul a funcțieiSIN(X). În sens larg aceasta este una din facilitățile oferite de segmentare: să poată fi ataşate blocuri de variabile (un segment) la o anumită secțiune de cod. Dacă programul creat conține un segment pentru variabilele locale ale funcției SIN, un segment pentru variabilele locale ale funcției SQRT, este imposibil ca rutina SIN să afecteze datele din segmentul de variabile SQRT aşa cum s-ar putea întâmpla la adresarea liniară. Într-adevăr, cu procesorul 80286 şi următoarele lucrând în modul protejat, unitatea centrală poate ca o rutină să modifice accidental variabilele dintr-un segment diferit.

O adresă completă atunci când se foloseşte adresarea segmentată se compune din adresa de segment şi adresa ofsetului (deplasamentului). O astfel de adresă se scrie: *segment:offset*. La procesoarele 8086 până la 80286 aceste două valori sunt constante pe 16 biți. Îccepând cu procesorul 80386 ofsetul poate fi o constantă pe 16 sau 32 de biți.

Dimensiunea ofsetului limitează valoarea maximă a unui segment. La procesorul 8086 cu un ofset pe 16 biți, segmentul poate avea cel mult 64K (un segment poate fi mai mic decât valoarea sa maximă dar niciodată mai mare). La procesoarele 80386 şi următoarele, ofsetul având 32 de biți rezultă că segmentele pot avea dimensiuni maxime de patru gigaocteți.

Dimensiunea segmentului este de 16 biți la toate procesoarele 80x86 şi deci un singur program poate avea până la 65536 de segmente diferite. Majoritatea programelor au insă în jur de 16 segmente dar acest număr nu reprezintă o limită.

Bineînțeles că, în ciuda faptului că familia procesoarelor 80x86 foloseşte adresarea segmentată, memoria fizică conectată la unitatea centrală este o arie liniară de octeți. Unitatea centrală are funcția de a transforma valoarea furnizată de adresarea segmentată (numită şi adresare logică) în valoarea adresei reale (adresare fizică).

La procesoarele 8086, 8088, 80186 şi 80188 (şi celelalte procesoare care lucrează în modul real), funcția de conversie de la adresa logică (de segment) la cea fizică (reală) este foarte simplă. Unitatea centrală înmulțeşte cu 16 (10h) valoarea conținută de registrul segment şi o adună cu valoarea ofsetului. De exemplu dacă vom considera adresa logică: 1000:1F00. Pentru calculul adresei fizice se înmulțeşte valoarea 1000h cu 10h (16 în baza 10). Înmulîirea în hexazecimal se face extrem de simplu prin adaugarea cifrei zero la deînmulțit: 1000h x 10h = 10000h. La valorea obținută se adună ofsetul şi se obține:

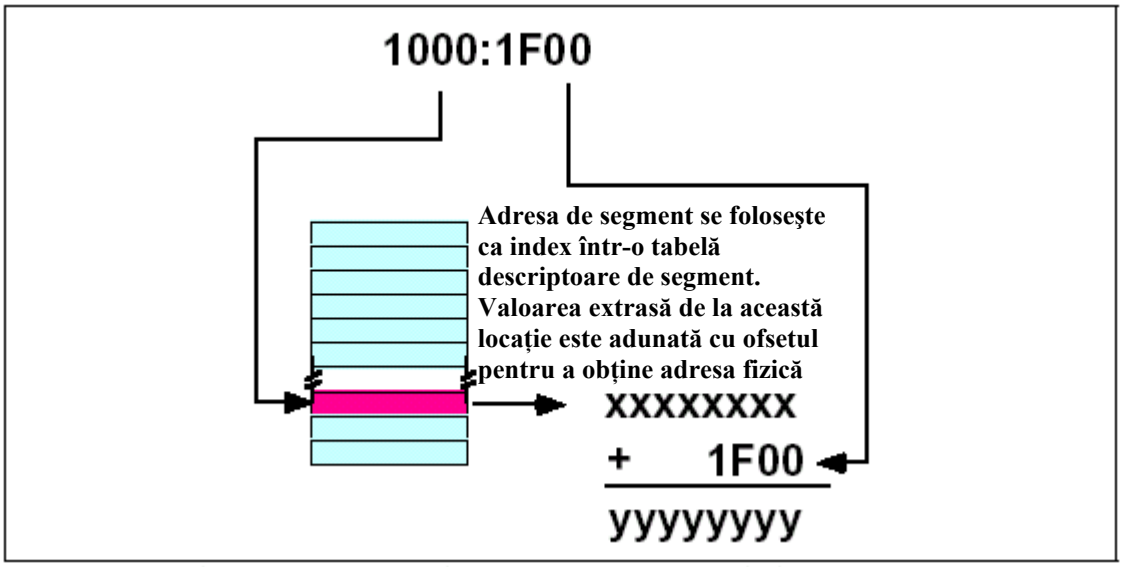
```
 10000h +
  1F00h
 -----------
 11F00h
```

Valoarea 11F00h este cea corespunzătoare adresei fizice (reale, din cauză că memoria este un vector liniar), adică 73472 în baza zece. Din acest mod de calcul este evident că pentru o adresă fizică pot fi mai multe adrese logice în funcție de cum se alege adresa de segment şi cea a ofsetului. De exemplu aceeaşi adresă fizică se obține pentru adresa logică: 1100:0F00.

Firma Intel atunci când a proiectat procesoarele 80286 şi următoarele nu a extins adresarea prin adăugarea unor biți suplimentari la registrele de segment. În schimb a

fost schimbată funcția prin care unitatea centrală calculează adresa fizică. Dacă scrieți programe bazate pe calculul adresei fizice prin înmulțirea cu 16 a adresei de segment şi adunarea ofsetului, aceste programe vor funcționa numai pe procesoare 80x86 care funcționează în modul real şi nu veți avea acces decât la cel mult un megaoctet de memorie (aceeaşi limitare apare dacă veți lucra în modul virtual 86 – V86 – la procesoarele 80386 sau următoarele).

La procesoarele 80286 şi următoarele firma Intel a introdus segmentele în mod protejat. Printre alte schimbări, firma Intel a schimbat complet algoritmul de calcul a adresei fizice pe baza adresei logice. În loc să utilizeze un algoritm, ca cel prezentat mai sus, procesoarele în modul protejat folosesc un tabel de căutare (tabela descriptoare de segment) pentru a calcula adresa fizică. În modul protejat, procesorul 80286 şi următoarele folosesc valoarea din adresa de segment ca index într-o matrice. Conținutul elementului din matrice furnizează (printre altele) adresa de început a segmentului. Unitatea centrală va aduna această valoare la valoarea ofsetului pentru a obține valoarea adresei fizice. Modul de obținere a adresei fizice este ilustrat în figura următoare.



Trebuie reținut faptul că aplicațiile create de programator nu pot modifica direct tabela descriptoare de segment (tabela de căutare). Sistemele de operare în mod protejat (UNIX, Linux, Windows, OS/2 etc.) dirijează această operație.

2.3.9. Adrese normalizate la 80x86

Când se operează în modul real, apare o problemă interesantă (cea amintită anterior). Ne putem referi la un singur obiect din memorie folosind adrese diferite. Dacă reluăm exemplul anterior, adresa 1000:1F00, putem construi şi alte adrese logice care să se refere la aceeaşi adresă fizică. De exemplu: 11F0:0, 1100:F00 şi chiar 1080:1700 corespund toate aceleiaşi adrese fizice şi anume 11F00h. Când se lucrează cu mai multe tipuri de date şi în special atunci când se compară pointerii este convenabil ca atunci când adresele de segment indică obiecte diferite din memorie ca valoarea registrului de segment să fie reprezentată diferit. Este limpede că aceasta nu este întotdeauna cazul procesoarelor 80x86 lucrând în modul real.

Din fericire există o cale simplă de a rezolva problema. Dacă este necesar să se compare două adrese se pot folosi adrese normalizate. Adresele normalizate au o formă specială şi aceasta este întotdeauna unică. Acest lucru se întâmplă în afară de cazul când două valori ale segmentelor normalizate sunt identice şi ele nu se referă la acelaşi obiect din memorie.

Sunt mai multe căi diferite (de fapt 16) pentru a crea adrese normalizate. Prin convenţie, cei mai mulţi programatori (chiar şi de limbaje de nivel înalt) definesc o adresă normalizată astfel:

- adresa de segment poate fi orice valoare pe 16 biţi;
- ofsetul trebuie să fie o valoare cuprinsă în domeniul: 0 ... 0Fh.

Pointerii normalizaţi în felul acesta sunt foarte uşor de convertit la adresa fizică. Singurul lucru pe care-l aveţi de făcut este să adăugaţi singura cifră hexazecimală a ofsetului la sfârşitul valorii segmentului. Forma normalizată a adresei 1000:1F00 este 11F0:0. Adresa fizică se obţine foarte uşor adăugând la sfârşitul adresei de segment 11F0, valoarea 0 a ofsetului, obţinând: 11F00.

Este foarte uşor de a converti o valoare oaecare a unei adrese de segmentate într-o valoare normalizată. Mai întâi se converteşte adresa segmentată la adresa fizică prin înmulţirea cu 16 a valorii adresei de segment şi apoi adunarea la aceasta a valorii ofsetului. Introduceţi apoi simbolul două puncete ":" între ultimile două cifre a rezultatului care trebuie să aibă cinci cifre:

1000:1F00 ⇒ 11F00 ⇒ 11F0:0

Este important de reţinut că adresa normalizată se foloseşte doar la proacesoarele 80x86 ce operează în modul real. În modul protejat nu există o corespondenţă directă între adresa segmentată şi adresa fizică şi deci tehnica descrisă nu poate fi folosită. Atunci când se vorbeşte de adrese normalizate se va subînţelege că procesorul lucrează în modul real.

2.3.10. Registrele de segment la procesoarele 80x86

Atunci când firma Intel a proiectat procesorul 8086, în anul 1976, memoria era o resursă preţioasă. Din acest motiv firma a proiectat setul de instrucţiuni în aşa fel încât să se utilizeze cât mai puţini biţi pentru codificarea acestora. Acest lucru a dus la programe mai mici şi în conseciinţă calculatoarele dotate cu procesoare Intel necesitau mai puţină memorie şi erau mai ieftine. Odată cu scăderea preţului memoriei acest aspect aparent ar părea să devină neimportant. Rămâne totuşi adevărat faptul că programele de dimensiuni mici (şi implicit instrucţiunile scurte) vor fi executate mai repede de către unitatea centrală şi asta va duce la creşterea globală a vitezei de execuţie a programelor. În această idee, firma Intel a dorit evitarea scrierii adresei întregi de 32 de biţi (segment şi ofset) în instrucţiunile ce fac referire la anumite zone de memorie. În mod curent instrucţiunile conţin numai 16 biţi ai adresei de ofset. Pentru a putea realiza acest lucru se fac anumite atribuiri implicite registrelor de segment în aşa fel încât unitatea centrală, în funcţie de context şi de tipul instrucţiunii, să poată determina care anume din registrele de segment este folosit împreună cu adresa de ofset.

Procesoarele 8086 până la 80286 au patru registre de segment: **cs, ds, es** şi **ss**. Procesoarele 80386 şi următoarele au pe lângâ aceste registre de segment, încă două registre de segment suplimentare: **fs** şi **gs**. Registrul de segment **cs** indică segmentul ce conţine codul ce se execută la un moment dat. Unitatea centrală va executa întotdeauna instrucţiunile de la adresa cs:ip. De asemenea în mod implicit unitatea centrală va căuta variabilele aferente programului executat în segmentul de date. Alte variabile sau operaţii se vor executa în segmentul de stivă. Când se accesează aceste zone specifice nu este necesară specificarea registrului de segment utilizat. Pentru accesarea datelor din extrasegmente (**es, fs** sau **gs**) este necesar un singur bit pentru a specifica registrul corespunzător. În setul de instrucţiuni al procesorului doar câteva instrucţiuni de transfer necesită necesită specificarea adresei segmentate în întregime pe 32 de biţi.

Toate aceste lucruri pot părea nişte limitări în utilizarea procesorului. De exemplu, cu ajutorul celor patru registre de segment ale procesorului 8086 nu se pot folosi la un moment dat decât 256 Kiloocteţi (64 Kiloocteţi maxim pentru fiecare segment) de memorie din totalul de un Megaoctet. Problema se rezolvă prin modificarea conţinutului registrelor de segment şi în acest fel poate fi accesată toată memoria disponibilă.

Este evident faptul că instrucţiunile pentru schimbarea conţinutului registrului de segment de la procesoarele 80x86 vor consuma memorie şi un anumit timp pentru execuţie. Cu toate acestea soluţia de a folosi adresarea implicită (fără specificarea registrului de segment) rămâne mai eficientă deoarece pe parcursul unui program necesitatea schimbării segmentului (pentru accesarea datelor din segmente diferite, de exemplu) este destul de puţin frecventă.

2.4. Modurile de adresare la procesoarele 80x86

Existenţa modurilor de adresare permite estimarea posibilităţilor de programare în limbaj de asamblare a unei unităţi centrale. Cu cât modurile de adresare a operanzilor (posibilităţile de accesare a memoriei) sunt mai diversificate cu atât posibilităţile de programare sunt mai extinse şi programele obţinute mai performante.

Procesoarele 80x86 permit accesarea memoriei prin mai multe căi diferite. Modurile de adresare ale procesoarelor 80x86 furnizează un mod flexibil de accesare a memoriei permiţând accesarea simplă a variabilelor, matricilor, înregistrărilor, pointerilor sau a altor tipuri de date complexe. Stăpânirea modurilor de adresare al procesoarelor 80x86 este primul pas în învăţarea programării în limbaj de asamblare.

Când firma Intel a proiectat procesorul 8086, l-a prevăzut cu un set de moduri de adresare a memoriei flexibil dar limitat. La procesorul 80386 au fost adăugate mai multe moduri de adresare dar trebuie reţinut faptul că au fost păstrate toate modurile de adresare a procesoarelor anterioare din motive de compatibilitate. Deşi modurile de adresare noi nu vor putea fi folosite pe procesoarele anterioare (cum ar fi 80286), nici evitarea acestor modur noi de adresare nu este convenabilă dacă programul este scris pentru un procesor 80386 din cauză că se pierd facilităţi importante ce fac programul mai performant. Din acest motiv, prezentarea se va face separat pentru cele două seturi de moduri de adresare pentru evitarea confuziilor.

2.4.1. Modul de adresare a registrelor la procesorul 8086

Cele mai multe instrucțiuni ale procesorului 8086 pot opera cu registrele de uz general. Prin specificarea numelui registrului ca operand într-o instrucțiune se poate avea acces la conținutul acelui registru. Să considerăm instrucțiunea **mov** (move – deplasează, mută):

mov destinație, sursă

Această instrucțiune copie informația din operandul *sursă* în operandul *destinație*. Registrele pe 16 sau 8 biți sunt operanzi valizi pentru această instrucțiune. Singura restricție este reprezentată de faptul că cei doi operanzi trebuie să aibă aceeaşi dimensiune (8 sau 16 biți). Iată câteva exemple:

```
mov   ax, bx  ;Copie valoarea din BX în AX
mov   dl, al  ;Copie valoarea din AL în DL
mov   si, dx  ; Copie valoarea din SI în DX
mov   sp, bp  ; Copie valoarea din BP în SP
mov   dh, cl  ; Copie valoarea din CL în DH
mov   ax, ax  ;Aceasta instrucțiune este posibila dar nu modifică nimic
```

Registrele reprezintă locul cel mai convenabil în care să se păstreze variabilele mai des folosite. În acest fel se evită accesul repetat la memorie şi viteza de execuție a programului creşte iar instrucțiunile folosite vor fi mai scurte. În continuare se vor folosi prescurtările pentru operanzi: *reg* şi *r/m* (registru/memorie) ori de câte ori va fi vorba de unul din registrele de uz general ale procesorului 8086.

În afară de registrele de uz general, multe instrucțiuni ale procesorului 8086 (inclusiv instrucțiunea *mov*) permit folosirea unui registru segment ca operand. Aici avem însă două restricții: în primul rând, registrul **cs** nu poate fi specificat ca operand destinație şi în al doilea rând, doar unul singur dintre operanzi poate fi registru de segment. Asta înseamnă că nu se poate transfera conținutul unui registru segment în altul cu o singură instrucțiune *mov*. Pentru a copia valoarea registrului **cs** în reistrul **ds** se poate folosi o secvență de felul următor:

```
mov   ax, cs
mov   ds, ax
```

Un registru segment nu trebuie folosit niciodată la stocarea datelor întâmplătoare. Aceste registre trebuie să conțină doar adrese de segment. Pentru registre de segment se va folosi prescurtarea *seg* ori de câte ori un registru de segment este permis (sau necesar) ca operand.

2.4.2. Modurile de adresare ale memoriei la procesorul 8086

Procesorul 8086 furnizează 17 căi diferite de acces la memorie. Deşi par destul de multe, din fericire cele mai multe moduri de adresare sunt variante ce derivă una din cealaltă şi din acest motiv sunt foarte uşor de învățat.

Modurile de adresare posibile la familia de procesoare 8086 sunt: numai deplasament, bază, depalsament plus bază, deplasament plus index şi deplasament plus bază plus index. Variaţii ale acestor cinci forme furnizează cele 17 moduri de adresare diferite ale procesorului 8086.

2.4.2.1. Modul de adresare numai prin deplasament

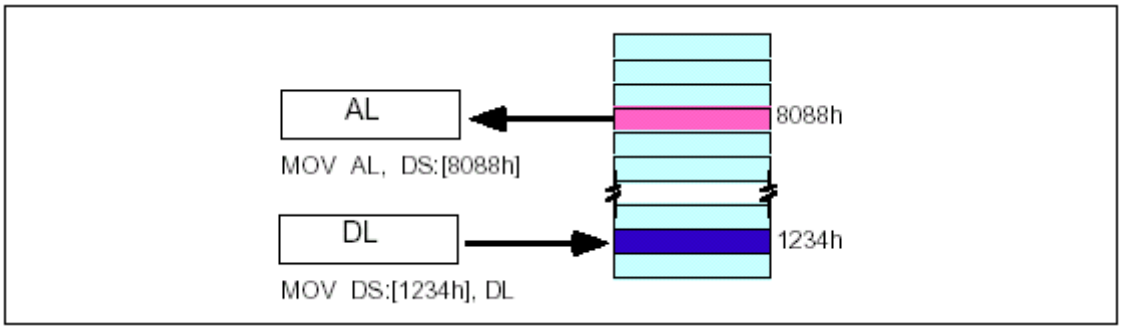
Cel mai utilizat mod de adresare şi cel mai uşor de înţeles este modul de adresare numai prin deplasament (sau direct). Modul de adresare direct constă în specificarea unei valori constante pe 16 biţi care care reprezintă valoarea adresei locaţiei adresate. Instrucţiunea:

mov al, ds:[8088h]

Încarcă registrul **al** cu valoarea conţinută de locaţia de memorie 8088h. De asemenea instrucţiunea:

mov ds:[1234h],dl

stochează valoarea registrului **dl** în locaţia de memorie 1234h.



NOTĂ:

Sintaxa MASM pentru modurile de adresare a memoriei.

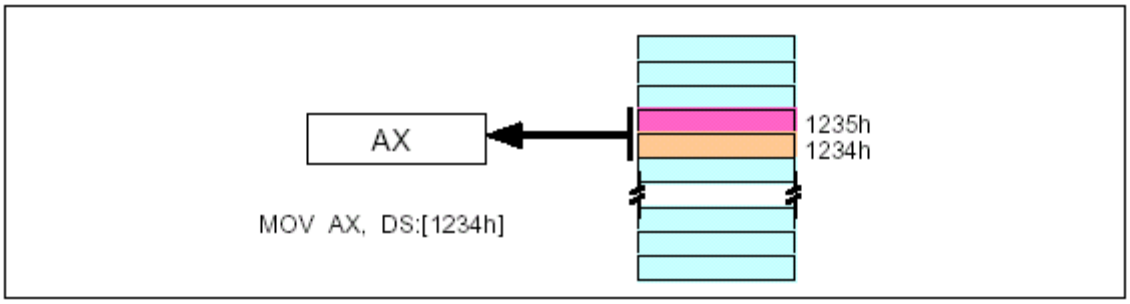
Asamblorul Microsoft MASM foloseşte diferite notaţii pentru adresarea indexată, bazată/indexată şi deplasament plus bazată/indexată. Următoarea listă prezintă combinaţiile care sunt posibile la modurile de adresare 8086:

disp[bx], [bx][disp], [bx+disp], [disp][bx] şi *[disp+bx]*
[bx][si], [bx+si], [si][bx] şi *[si+bx]*
disp[bx][si], disp[bx+si], [disp+bx+si], [disp+bx][si], disp[si][bx], [disp+si][bx], [disp+si+bx], [si+disp+bx], [bx+disp+si], etc.

MASM tratează simbolul "[] la fel ca pe operatorul "+". Acest operator este comutativ la fel ca operatorul "+". Bineînţeles că acestă discuţie se referă la toate modurile de adresare pentru 8086 şi nu numai cele care implică registrele **bx** şi **si**. Aceste registre pot fi înlocuite cu oricare dintre registrele permis a fi utilizate în modurile de adresare descrise mai sus.

Modul de adresare numai prin deplasament (modul direct) este cel mai adecvat pentru accesarea variabilelor simple. Evident că este de preferat să se folosească nume ca "I" sau "J" în loc de "DS:[1234h]" sau "DS:[8088h]" pentru a simplifica lucrurile.

Intel numeşte acest mod de adresare "adresare numai prin deplasament" deoarece se foloseşte o singură constantă de 16 biţi ce reprezintă ofsetul (sau deplasamentul) în codul instrucţiunii *mov*. Din acest punct de vedere acest mod de adresare este foarte asemănător modului de adresare direct de la procesoarele x86 (vezi capitolul anterior). Există însă câteva diferenţe minore. Înainte de toate, deplasamentul reprezintă o anumită distanţă dintre două puncte. La adresarea directă de la procesoarele x86 acest lucru este adevărat considerând deplasamentul faţă de adresa zero. La procesoarele 80x86 deplasamentul este de fapt ofsetul faţă de începutul segmentului (segmentul de date în exemplul nostru). Pentru moment considerăm modul de adresare numai prin deplasament ca un mod de adresare direct. Trebuie reţinut că la procesorul 8086 prin acest mod de adresare se pot accesa şi cuvinte (word – 2 octeţi) iar la 80386 cuvinte duble.



Implicit, toate valorile "numai prin deplasament" furnizează ofsetul în segmentul de date. Dacă doriţi să furnizaţi ofsetul într-un alt segment trebuie să puneţi simbolul segmentului (să prefixaţi adresa) înainte de adresă. De exempu, dacă doriţi să accesaţi locaţia 1234h din extrasegment (**es**) trebuie să folosiţi instrucţiunea *mov* sub forma:

es:[1234h]

iar dacă doriţi să accesaţi locaţia în segmentul de cod (**cs**):

mov ax, cs:[1234h].

Apariţia prefixului "ds:" în exemplele de la început nu reprezintă o specificare a segmentului. Unitatea centrală utilizează segmentul de date (**ds**) implicit. În acele exemple a fost specificat "ds:" numai datorită limitărilor sintactice impuse de asamblorul MASM.

2.4.2.2. Modul de adresare indirectă prin registre

Unităţile centrale ale procesoarelor 80x86 vă permit adresarea indirectă a memoriei prin intermediul registrelor folosind modul de adresare indirectă prin registre.

Sunt patru forme de adresare la 8086 cel mai bine exemplificate prin următoarele instrucțiuni:

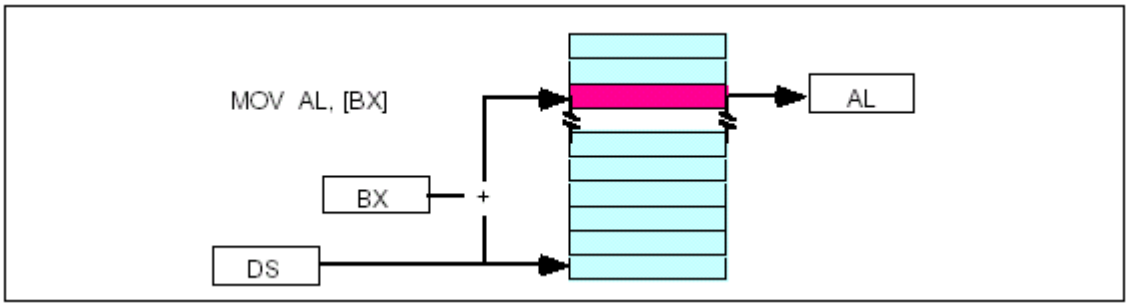
mov al, [bx]
mov al, [bp]
mov al, [si]
mov al, [di]

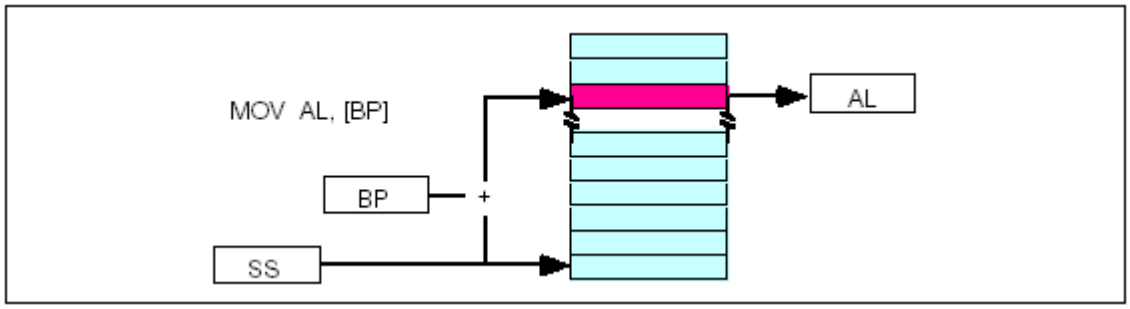
La fel ca modul de adresare x86[bx], aceste patru moduri de adresare vor conține valoarea ofsetului în registrele **bx, bp, si** sau **di**. La utilizarea registrelor [bx], [si] şi [di] registrul **ds** este registrul segment implicit iar pentru [bp] registrul de segment implicit este registrul de segment de stivă (**ss**).

Se poate utiliza de asemenea specificarea explicită a registrului de segment dacă se doreşte accesarea datelor într-un segment diferit de cel implicit. Iată câteva exemple:

mov al, cs:[bx]
mov al, ds:[bp]
mov al, ss:[si]
mov al, es:[di]

Intel se referă la modurile de adresare [bx] şi [bp] ca moduri de adresare bazate iar la registrele **bx** şi **bp** ca registre bază (de fapt **bp** este notația pentru base pointer – indicatorul bazei). La fel, la modurile de adresare ce folosesc [si] si [di] se numesc moduri de adresare indexate (**si** înseamnă source index – index sursă iar **ds**, destination index – index destinație). În orice caz aceste moduri de adresare sunt din punct de vedere funcțional echivalente (lucrează la fel dacă vom înlocui simbolurile [si] sau [di] cu [bx]). Din acest motiv vom numi aceste moduri de adresare ca adresare indirectă prin registre pentru a fi consecvenți. Modul în care funcționează acest mod de adresare este ilustrat în figurile următoare.





2.4.2.3. Modurile de adresare indexate

Adresarea indexată foloseşte următoarea sintaxă:

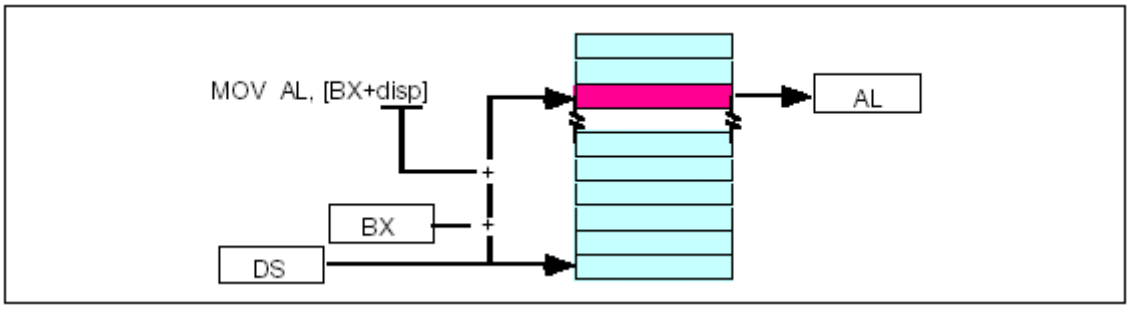
mov al, disp[bx]
mov al, disp[bp]
mov al, disp[si]
mov al, disp[di]

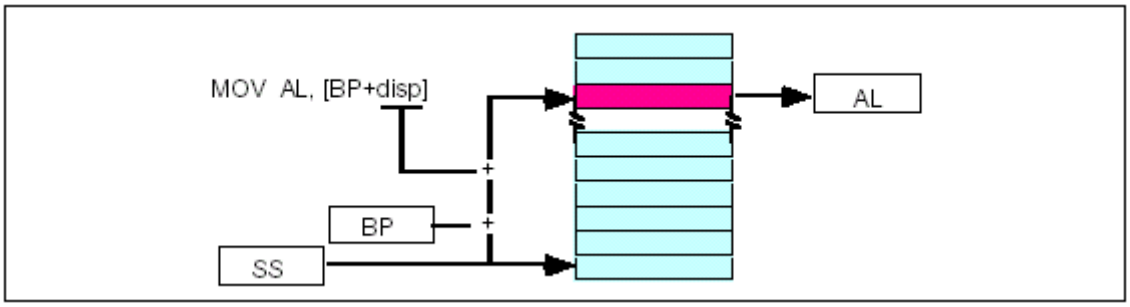
Dacă registrul **bx** conţine 1000h, atunci instrucţiunea mov cl,20h[bx] va încărca registrul **cl** cu conţinutul locaţiei de memorie ds:1020h. De asemenea, dacă registrul **bp** conţine valoarea 2020h, atunci instrucţiunea mov dh,1000h[bp] va încărca registrul **dh** cu valoarea conţinută de locaţia de memorie de la adresa ss:3020.

Ofsetul generat de acest mod de adresare este suma dintre o constantă şi conţinutul registrului specificat. Modurile de adresare care implică registrele **bx**, **si** şi **di** folosesc segmentul de date ca segment implicit iar utilizarea registrului **bp** înseamnă că registrul de segment de stivă **ss** va fi registrul implicit. Şi la acest mod de adresare se poate specifica explicit registrul de segment:

mov al, ss:disp[bx]
mov al, es:disp[bp]
mov al, cs:disp[si]
mov al, ss:disp[di]

Modul de adresare bazat indexat este ilustrat în figurile următoare.





În figura de mai sus putem folosi registrele **si** sau **di** în locul registrului **bx** pentru a obține modurile de adresare pentru [si+disp] sau [di+disp].
(Comparație între adresarea bazată şi adresarea indexată: aici trebuie să ne amintim faptul că Intel numeşte instrucțiunile formate cu **bx** sau **bp** ca instrucțiuni bazate iar cele cu **si** şi **di** ca indexate; din acest motiv apare o confuzie în demumirea modurilor de adresare care trebuie corectată în acest text.)

Există o diferență subtilă între modurile de adresare bazată şi indexată. Amândouă modurile de adresare constau într-un deplasament adunat la conținutul unui registru. Diferența esențială dintre cele două moduri de adresare constă în valoarea relativă a deplasamentului şi cea conținută de registru. În modul de adresare indexat constanta furnizează în mod tipic adresa unei structuri de date specifice iar registrul furnizează un ofset pentru această adresă. În modul de adresare bazat, registrul conține adresa structurii de date iar deplasamentul constant furnizează un index pentru acest punct.

Deoarece adunarea este comutativă cele două moduri de a privi problema sunt echivalente. Totuşi, deoarece Intel permite unul sau doi octeți pentru deplasament este mai rațional să numească acest mod de adresare bazat. Totuşi, de obicei modul de adresare bazat va fi folosit mai mult ca mod de adresare indexat şi prin urmare numele se schimbă.

2.4.2.4. Modul de adresare indexat bazat

Modul de adresare indexat bazat este o combinație între adresarea indirectă prin registre. Acest mod de adresare formează ofsetul prin adunarea conținutului unui registru bază (**bx** sau **bp**) şi conținutul unui registru index (**si** sau **di**). Formele posibile pentru acest mod de adresare sunt:

mov al, [bx][si]
mov al, [bx][di]
mov al, [bp][si]
mov al, [bp][di]

Să presupunem că registrul **bx** conține valoarea 1000h şi registrul **si** conține valoarea 880h. Atunci instrucțiunea:

mov al,[bx][si]

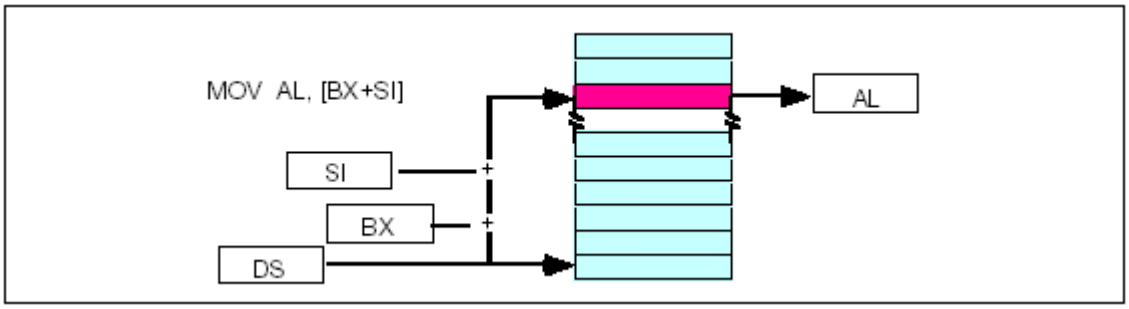
va încărca registrul **al** cu conținutul locației de memorie de la adresa ds:1880h. Şi aici se fac aceleaşi precizări cu privire la registrele segment implicite.

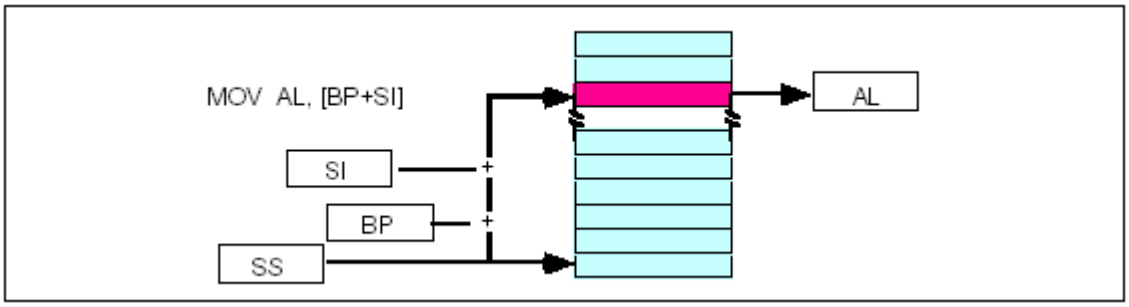
2.4.2.5. Adresare indexată bazată plus deplasament

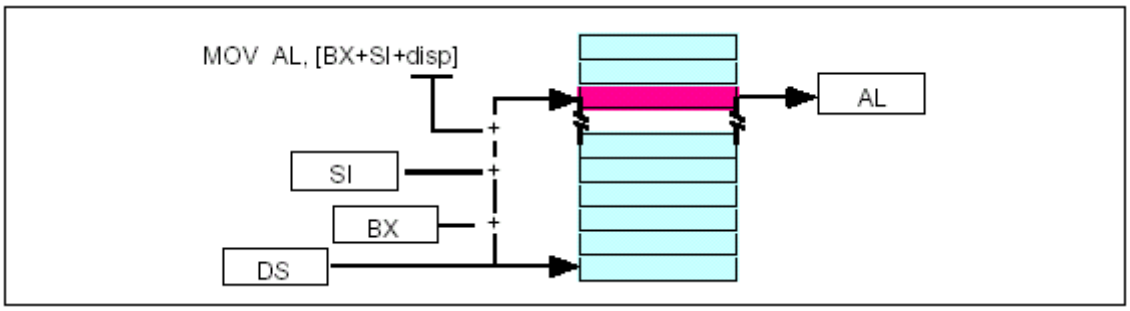
Acest mod de adresare este o modificare a modului de adresare bazat/indexat prin adăugarea unei constante pe 8 biți sau 16 biți. Instrucțiunile următoare reprezintă un exemplu al acestui mod de adresare.

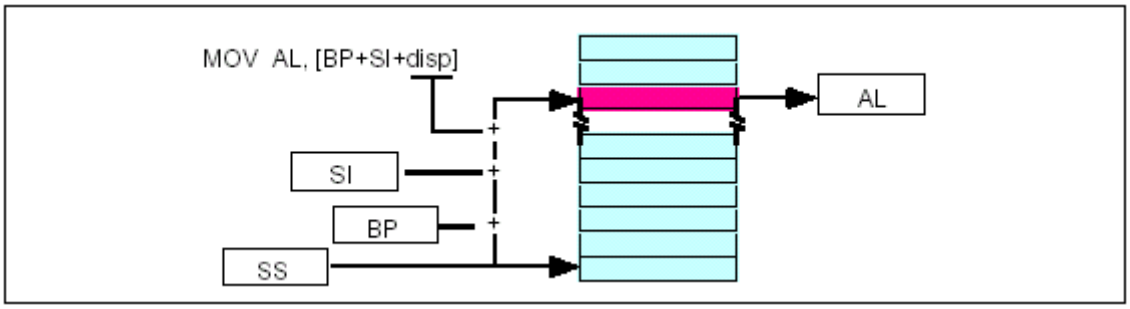
mov al, disp[bx][si]
mov al, disp[bx+di]

mov al, [bp+si+disp]
mov al, [bp][di][disp]









Să presupunem că registrul **bp** conține 1000h, **bx** conține 2000h, **si** conține 120h şi **di** conține 5; atunci instrucțiunea mov al,10h[bx+si] va încărca registrul **al** cu conținutul locației de memorie de la adresa DS:2130; instrucîiunea mov ch,125h[bp+di] încarcă registrul **ch** cu conținutul locației de memorie de la adresa SS:112A şi instrucțiunea mov bx,cs:2[bx][di] încarcă registrul **bx** cu conținutul locației de memorie de la adresa CS:2007.

2.4.2.6. Un mod simplu de a reţine modurile de adresare a memoriei la procesorul 8086

Aşa cum s-a arătat, există 17 moduri de adresare la procesorul 8086: disp, [bx], [bp], [si], [di], disp[bx], disp[bp], disp[si], disp[di], [bx][si], [bx][di], [bp][si], [bx][di], disp[bx][si], disp[bx][di], disp[bp][si] şi disp[bp][di] – fără a ţine cont de diferitele variante sintactice posibile. Toate aceste forme pot fi memorate dacă se cunoaşte care combinaţii cunt valide. Considerăm tabelul următor:

DISP	[BX]	[SI]
	[BP]	[DI]

Tabelul pentru generarea modurilor valide de adresare la procesorul 8086

Dacă alegeţi zero sau unul din oricre din termenii unei coloane şi-l alăturaţi cel puţin uni termen din celelalte coloane, obţineţi un mod de adresare valid la 8086. Iată câteva exemple:

- alegeţi disp din prima coloană, nimic din coloana doi şi [di] din coloana a treia, se obţine: disp[di];
- alegeţi disp, [bx] şi [di] şi se obţine: disp[bx][di]
- săriţi coloana unu şi doi şi alegeţi [si] din coloana trei; se obţine [si];
- săriţi prima coloană, alegeţi [bx] apoi [di] şi se obţine: [bx][di].

De altfel dacă luaţi un mod de adresare care nu poate fi construit cu tabelul de mai sus, atunci acesta nu este legal. De exemplu, modul de adresare disp[dx][si] nu este posibil deoarece [dx] nu există în tabelul de mai sus.

2.4.2.7. Câteva comentarii finale asupra modurilor de adresare la procesorul 8086

Adresa efectivă este ofsetul final obţinut prin calcule la un anumit mod de adresare. De exemplu, dacă registrul **bx** conţine 10h atunci adresa efectivă pentru 10h[bx] este 20h. Vom întâlnii termenul de adresă efectivă în majoritatea discuţiilor despre modul de adresare la procesorul 8086. Este chiar o instrucţiune specială: *load effective addresss* (lea) care calculează adresa efectivă.

Nu toate modurile de adresare necesită acelaşi timp de execuţie. De regulă, cu cât modul de adresare este mai complex cu atât timpul necesar execuţiei instrucţiunii va fi mai mare. De asemenea, deplasamentul, cu excepţia modului de adresare numai deplasament, poate fi un număr cu semn pe 8 sau 16 biţi. Dacă deplasamentul este pe 8 biţi (un număr în domeniul –128 ... +127) instrucţiunea va fi scurtă şi deci mai rapidă. Din acest motiv, la modurile de adresare, de regulă se preferă scrierea valorilor mari în registre în aşa fel încît deplasamentul să fie pe 8 biţi.

Dacă după calculul adreseti efective rezultă o valoare mai mare ca 0FFFFh, unitatea centrală ignoră depăşirea iar reyultatul va fi rotunjit la cei mai puţin semnificativi 16 biţi (wraps around back to zero). De exemplu, dacă registrul **bx** conţine valoarea 10h atunci instrucţiunea mov al,0FFFFh[bx] va încărca registrul **al** cu conţinutul locaţiei de la adresa 0Fh (0FFFFh + 10h = 1000Fh).

2.4.3. Modurile de adresare a registrelor la 80386

Procesorul 80386 şi următoarele furnizează registre pe 32 de biţi. Cele 8 registre de uz general se numesc: **eax, ebx, exc, edx, esi, edi, ebp** şi **esp**. Aceste registre pot fi folosite ca operanzi în numeroare instrucţiuni ale procesorului 80386.

2.4.3.1.Modurile de adresare a memoriei la 80386

Procesorul 80386 a generalizat modurile de adresare la registre. Dacă 8086 permitea utilizarea numai a registrelor **bx** şi **bp** ca registre de bază şi numai a registrelor **di** şi **si** ca registre index, procesorul 80386 permite ca aproape orice registru să fie folosit ca registru de bază sau index. De asemenea se introduce un mod nou de adresare: adresarea indexată scalată care simplifică accesul la elementele unei matrici.

2.4.3.2.Modul de adresare indirectă prin registre

La procesorul 80386 poate fi folosit oricare din registrele de uz general pe 32 de biţi atunci când se foloseşte modul de adresare indirectă prin registre. Simbolurile [eax], [ebx], [ecx], [edx], [esi] şi [edi] furnizează ofsetul pentru registrul segment de date **ds** considerat registru segment implicit. Simbolurile [ebp] şi [esp] folosesc segmentul de stivă ca segment implicit.

Atunci când se rulează programe în modul real pe 16 biţi a lui 80386, ofsetul din registrele pe 32 de biţi trebuie să fie în domeniul 0 ... 0FFFFh. Nu se pot folosi valori mai mari deci nu se pot accesa segmente mai mari de 64k (acest lucru este posibil în modul protejat). De asemenea nu se pot folosi numele pe 16 biţi ale registrelor ci numai cele pe 32 de biţi. În continuare se prezintă exemple de instrucţiuni corecte:

mov al, [eax]
mov al, [ebx]
mov al, [ecx]
mov al, [edx]
mov al, [esi]
mov al, [edi]
mov al, [ebp] ;Foloseşte registrul SS implicit.
mov al, [esp] ; Foloseşte registrul SS implicit.

2.4.3.3.Modurile de adresare indexat, indexat/bazat şi bazat/indexat/deplasament la procesorul 80386

Modul de adresare indexat (indirect prin registru plus deplasament) vă permite să folosiţi un registru pe 32 de bişi şi o constantă. Modul de adresare bazat/indexat vă permite să folosiţi perechi de două registre de 32 de biţi. În sfârşit modul de adresare

deplasament/bazat/indexat vă permite să combinați o constantă cu două registre pe 32 de biți pentru a forma adresa efectivă. Trebuie reținut faptul că ofsetul produs de calculul adresei efective trebuie să rămână pe 16 biți atunci când se lucrează în modul real.

La 80386 termenii de registru de bază şi registru index capătă înțelesuri noi. Când combinăm două registre pe 32 de biți într-un mod de adresare, primul registru este registrul bază iar al doilea este registru index. Acest lucru este adevărat dacă ne referim la numele registrelor. Procesorul 80386 permite utilizarea aceluiaşi registru atât ca registru de bază cât şi ca registru index lucru ce uneori este folositor. Următoarele instrucțiuni prezintă exemple reprezentative pentru diferite moduri de adresare de bază şi indexate:

```
mov al, disp[eax]      ;Moduri de adresare
mov al, [ebx+disp]     ; indexate.
mov al, [ecx][disp]
mov al, disp[edx]
mov al, disp[esi]
mov al, disp[edi]
mov al, disp[ebp] ; Foloseşte registrul SS implicit.
mov al, disp[esp] ; Foloseşte registrul SS implicit.
```

Următoarele instrucțiuni folosesc toate modul de adresare bazat+indexat. Primul registru din cel de-al doilea operand este registrul bazei iar cel de-al doilea este registrul index. Dacă registrul bazei este **esp** sau **ebp** adresa efectivă este relativă le segmentul de stivă. De reținut faptul că alegerea registrului index nu afectează alegerea segmentului implicit.

```
mov al, [eax][ebx]     ;Moduri de adresare
mov al, [ebx+ebx]      ;bazat+indexat.
mov al, [ecx][edx]
mov al, [edx][ebp]     ;Foloseşte DS implicit.
mov al, [esi][edi]
mov al, [edi][esi]
mov al, [ebp+ebx]      ; Foloseşte registrul SS implicit.
mov al, [esp][ecx]     ; Foloseşte registrul SS implicit.
```

Evident că se poate adăuga deplasamentul la modurile de adresare prezentate mai sus pentru a obține modul de adresare bazat+indexat+deplasament. Următoarele exemple sunt reprezentative pentru acest mod de adresare:

```
mov al, disp[eax][ebx]     ;Modul de adresare
mov al, disp[ebx+ebx]      ; bazat indexat.
mov al, [ecx+edx+disp]
mov al, disp[edx+ebp]      ;Foloseşte DS implicit.
mov al, [esi][edi][disp]
mov al, [edi][disp][esi]
mov al, disp[ebp+ebx]      ;Foloseşte SS implicit.
mov al, [esp+ecx][disp]    ;Foloseşte SS implicit.
```

Există o restricție la 80386 legată de registrul **esp**: acest registru poate fi folosit ca registru de bază dar nu poate fi folosit ca registru index.

2.4.3.4.Modul de adresare scalat indexat la procesorul 80386

Modurile de adresare: indexat, bazat/indexat şi bazat/indexat/deplasament descrise până acum sunt cazuri particulare ale adresării indexate scalate ale procesorului 80386. Aceste moduri de adresare sunt utile în particular pentru adresarea elementelor matricilor deşi ele nu sunt destinate numai acestui scop. Aceste moduri vă permit să multiplicați registrul index din modul de adresare cu unu, doi, patru sau opt. Sintaxa generală a acestui mod de adresare este:

*disp[index*n]*
*[base][index*n]*

sau

*disp[base][index*n]*

unde "bază" sau "index" reprezintă oricare din registrele pe 32 de biți ale procesorului 80386 iar "n" este un număr egal cu unu, doi, patru sau opt.

80386 calculează adresa efectivă prin sumarea deplasamentului cu baza şi cu index multiplicat cu n.

De aici rezultă că modurile: indexat, bazat/indexat şi bazat/indexat/deplasament sunt cazuri speciale ale modului de adresare scalat indexat cu "n" egal cu unu. Următoarele perechi de instrucțiuni sunt perfect identice pentru 80386:

*mov al, 2[ebx][esi*1]*	*mov al, 2[ebx][esi]*
*mov al, [ebx][esi*1]*	*mov al, [ebx][esi]*
*mov al, 2[esi*1]*	*mov al, 2[esi]*

Bineînțeles că MASM permite o mulțime de variațiuni la aceste moduri de adresare. Următoarele instrucțiuni ilustrează o mică parte din posibilități:

*disp[bx][si*2], [bx+disp][si*2], [bx+si*2+disp], [si*2+bx][disp],*
*disp[si*2][bx], [si*2+disp][bx], [disp+bx][si*2]*

2.4.3.5.Câteva considerații finale asupra modurilor de adresare a memoriei la 80386

Din cauză că modurile de adresare la procesorul 80386 sunt mult mai coerente ele sunt mult mai uşor de memorat decât modurile de adresare ale procesorului 8086. Pentru programatorii care lucrează cu procesorul 80386 există întotdeauna tentația de a neglija modurile de adresare 8086 şi de a folosi pe cele ale lui 80386 în mod exclusiv. Cu toate acestea, aşa cum se va arăta, modurile de adresare 8086 sunt în realitate mai eficiente decât modurile comparabile ale lui 80386. Aşadar este important să se

cunoască toate modurile de adresare şi să se aleagă modul convenabil pentru o problemă dată.

Când se utilizează modurile de adresare bazat/indexat şi bazat/indexat/deplasament la 80386 fără opţiunea de scalare (asta însemnând să se lase scalarea implicită la "*1"), primul registru care apare în modul de adresare este registrul bază iar cel de-al doilea este registrul index. Acesta este un lucru important din cauză că registrul de segment implicit este ales după registrul bază. Dacă registrul bază este **ebp** sau **esp** atunci registrul segment implicit este registrul de stivă. În toate celelalte cazuri 80386 accesează segmentul de date implicit chiar dacă registrul index este **ebp**. Dacă utilizaţi poeratorul de scalare al indexului ("*n") la un registru, acel registru va fi registrul index indiferent unde apare în modul de adresare:

```
[ebx][ebp]          ;Foloseşte implicit DS.
[ebp][ebx]          ; Foloseşte implicit SS.
[ebp*1][ebx]        ; Foloseşte implicit DS.
[ebx][ebp*1]        ; Foloseşte implicit DS.
[ebp][ebx*1]        ; Foloseşte implicit SS.
[ebx*1][ebp]        ; Foloseşte implicit SS.
es:[ebx][ebp*1]     ; Foloseşte ES.
```

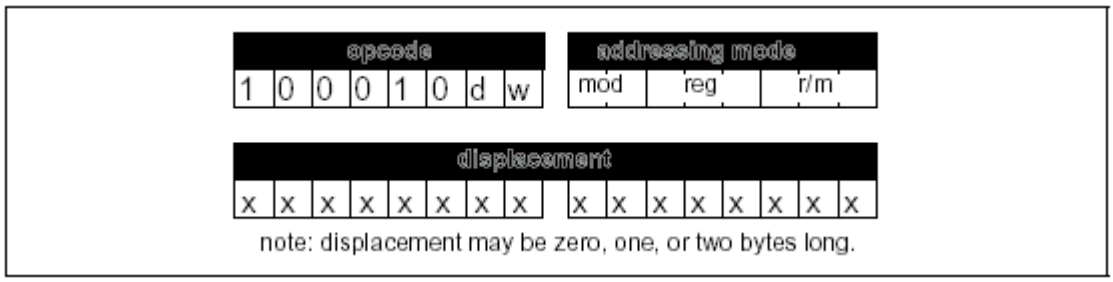
2.5. Instrucţiunea MOV la procesorul 8086

Structura instrucţiunii *mov* (move) care este una dintre cele mai utilizate instrucţiuni este:

mov Destinaţie,Sursă

Instrucţiunea *mov* face o copie a valorii sursei pe care o stochează în destinaţie. Instrucţiunea nu afectează conţinutul sursei ci numai cel al destinaţiei.

Pentru a înţelege complexitatea instrucţiunii *mov* trebuie să studiem modul de codificare a acesteia. În figura următoare este prezentată forma cea mai utilizată pentru codificarea binară a instrucţiunii *mov.*



Instrucţiunea MOV generică

Codul operaţiei se găseşte în primii 8 biţi ai instrucţiunii. Biţii zero şi unu definesc dimensiunea instrucţiunii (8, 16 sau 32 biţi) şi direcţia transferului (acestea sunt simbolizate cu **w** şi **d**). Urmează octetul modului de adresare numit octetul "mod-reg-r/m" de către majoritatea programatorilor. Acest octet care poate acea 256 de valori

diferite şi ele reprezintă combinațiile posibile pentru operanzi la instrucțiunea *mov* generică. Instrucțiunea *mov* generică are trei forme diferite în limbajul de asamblare:

mov reg, memory
mov memory, reg
mov reg, reg

Trebuie reținut faptul că cel puțin unul din operanzi este întotdeauna un registru de uz general. În câmpul *reg* al octetului mod/reg/rm este specificat acest registru de uz general (sau unul din registre în forma a treia de mai sus). Bitul *d* (direcție) din codul operației indică faptul că instrucțiunea va stoca data într-un registru (d = 1) sau în memorie (d = 0).

Biții din câmpul *reg* permite alegerea unui registru din 8 posibile. 8086 are 8 registre de 8 biți şi 8 registre de 16 biți de uz general. 80386 mai are suplimentar 8 registre de 32 de biți de uz general. Modul de decodificare a tipului de registru de către unitatea centrală este prezentat în tabelul următor:

reg	w=0	16 bit mode w=1	32 bit mode w=1
000	AL	AX	EAX
001	CL	CX	ECX
010	DL	DX	EDX
011	BL	BX	EBX
100	AH	SP	ESP
101	CH	BP	EBP
110	DH	SI	ESI
111	BH	DI	EDI

Pentru a diferenția registrele de 16 sau 32 de biți procesoarele 80386 şi următoarele folosesc un prefix special la codul operației la instrucțiunile ce folosesc registre de 32 de biți. În rest codificarea instrucțiunii este aceeaşi la cele două tipuri de instrucțiuni.

Câmpul *r/m* în conjuncție cu câmpul *mod* stabileşte modul de adresare. Codificarea câmpului *mod* este următoarea:

MOD	Meaning
00	The r/m field denotes a register indirect memory addressing mode or a base/indexed addressing mode (see the encodings for r/m) *unless* the r/m field contains 110. If MOD=00 and r/m=110 the mod and r/m fields denote displacement-only (direct) addressing.
01	The r/m field denotes an indexed or base/indexed/displacement addressing mode. There is an eight bit signed displacement following the mod/reg/rm byte.
10	The r/m field denotes an indexed or base/indexed/displacement addressing mode. There is a 16 bit signed displacement (in 16 bit mode) or a 32 bit signed displacement (in 32 bit mode) following the mod/reg/rm byte .
11	The r/m field denotes a register and uses the same encoding as the *reg* field

Câmpul *mod* selectează între transferul registru la registru şi transferul între registru şi memorie. De asemenea se selectează dimensiunea deplasamentului (zero, unu, doi sau patru biţi) folosiţi în instrucţiunile pentru modurile de adresare a memoriei. Dacă MODD=00 atunci este selectat unul din modurile de adresare fără deplasament (indirect prin registre sau bazat/indexat). Trebuie reţinut cazul special când MOD=00 şi r/m=110 care în mod normal ar corespunde modului de adresare [bp]. 8086 utilizează această codificare pentru modul de adresare numai deplasament. Asta înseamnă că nu există un mod de adresare [bp] adevarat la 8086.

Pentru a înţelege de ce nu putem utiliza în programe modul de adresare [bp] să privim la MOD=01 şi MOD=10 din tabelul de mai sus. Aceste configuraţii activează modurile de adresare disp[reg] şi disp[reg][reg] iar acestea nu sunt aceleaşi cu modul de adresare [bp]. Să considerăm următoarele instrucţiuni:

mov al, 0[bx]
mov ah, 0[bp]
mov 0[si], al
mov 0[di], ah

Aceste instrucţiuni, ce folosesc modurile de adresare indexată, realizează aceleaşi operaţii ca şi dublurile lor cu adresare indirectă cu registre (obţinute prin îndepărtarea deplasamentului în instrucţiunile de mai sus). Singura diferenţă reală între cele două forme este aceea că modul de adresare indexat este de un octet (dacă MOD=01) şi de doi octeţi (dacă MOD=10) pentru a reţine deplasamentul lui zero. Din cauză că ele sunt mai lungi atunci aceste instrucţiuni vor dura mai mult (execuţia va fi mai lentă).

Aceste trăsături ale procesorului 8086 – de a furniza două sau mai multe căi pentru a realiza acelaşi lucru – apar în întregul set de instrucţiuni. Deşi există mai multe forme MASM selectează automat forma cea mai bună (adecvată). Dacă scrieţi instrucţiunile de mai sus şi le asamblaţi cu MASM veţi vedea că se generează modul de adresare indirect pentru toate instrucţiunile cu excepţia instrucţiunii mov ah,0[bp]. Asamblorul va încerca întotdeauna să emită numai deplasamente de un octet pentru aceste instrucţiuni sunt mai scurte şi mai rapide decât instrucţiunile cu deplasament pe doi octeţi (dan – inex zero). Trebuie notat faptul că MASM nu vă cere să introduceţi 0[bp] ci trebuie să introduceţi numai [bp] iar MASM va furniza zero automat.

Dacă MOD nu este egal cu 11b, câmpul r/m codifică modul de adresare al memoriei în felul următor:

R/M	Addressing mode (Assuming MOD=00, 01, or 10)
000	[BX+SI] or DISP[BX][SI] (depends on MOD)
001	[BX+DI] or DISP[BX+DI] (depends on MOD)
010	[BP+SI] or DISP[BP+SI] (depends on MOD)
011	[BP+DI] or DISP[BP+DI] (depends on MOD)
100	[SI] or DISP[SI] (depends on MOD)
101	[DI] or DISP[DI] (depends on MOD)
110	Displacement-only or DISP[BP] (depends on MOD)
111	[BX] or DISP[BX] (depends on MOD)

Aceste explicații justifică faptul că instrucțiunile procesoarelor Intel sunt de tip CISC (Complex Instruction Set Computer).

2.6. Comentarii finale asupra instrucțiunilor MOV

Sunt câteva lucruri importante ce trebuie reținute în legătură cu instrucțiunea *mov*. Mai întâi de toate nu se poate face transferul direct de la memorie la memorie. Pentru a putea face acest transfer este necesar un grup de două instrucțiuni, una pentru transferul conținutului memoriei într-un registru şi una pentru transferul conținutului registrului în memorie. Un alt fapt important ce trebuie reținut în legătură cu instrucțiunea *mov* este faptul că există mai multe instrucțiuni *mov* diferite care realizează acelaşi lucru. De asemenea sunt mai multe moduri de adresare diferite ce pot fi folosite pentru accesarea aceleiaşi locații de memorie. Dacă doriți să scrieți cel mai scurt program posibil în limbaj de asamblare trebuie să cântăriți tot timpul care dintre instrucțiuni este cea mai convenabilă.

Discuția din acest capitol s-a făcut pentru instrucțiunea *mov* generică pentru a vedea cum procesorul 80x86 codifică modurile de adresare la memorie şi registre la acest tip de instrucțiune. Alte forme ale instrucțiunii *mov* vă permit transferul datelor între registrele de uz general de 16 biți şi registrele segment 80x86 sau încărcarea registrelor sau a locațiilor de memorie cu o constantă. Aceste variante ale instrucțiunii *mov* au alte coduri operație.

De asemenea sunt mai multe instrucțiuni *mov* suplimentare la procesorul 80386 care vă permit încărcarea registrelor de uz special ale acestuia ce nu au fost prezentate aici. Trebuie reamintite aici şi instrucțiunile pe şiruri ale procesoarelor 80x86 ce realizează transferuri memorie la memorie care pot fi un bun substituent pentru instrucțiunile *mov*.

2.7. Câteva instrucțiuni suplimentare

Instrucțiunile: **LEA** (load effective address). **LES** (load *es* and general purpose register), **ADD** (addition) şi **MUL** (multiply) ca şi instrucțiunea **MOV** prezentată anterior se dovedesc folositoare pentru accesarea diferitelor tipuri de date.

Instrucțiunea **LEA** are următoarea formă:

$$\texttt{lea}\ reg_{16},\ \texttt{memory}$$

unde reg_{16} este un registru de uz general pe 16 biți. *Memory* este o locație de memorie reprezentată de un octet mod/reg/rm (cu excepția faptului că trebuie să fie o locație de memorie şi nu poate fi un registru).

Această instrucțiune încarcă registrul de 16 biți cu ofsetul locației specificate de operandul *memory*. Instrucțiunea: lea ax,1000h[bx][si] de exemplu, va încărca registrul **ax** cu adresa locației de memorie specificată de 1000h[bx][si], care este desigur valoarea dată de 1000h+bx+si. Instrucțiunea este foarte folositoare pentru obținerea adresei unei variabile. Daca aveti o variabilă "I" undeva în memorie, instrucțiunea: *lea bx,I* va încărca registrul **bx** cu adresa (ofsetul) variabilei "I".

Instrucțiunea **LES** are următoarea formă:

$$\texttt{les}\ reg_{16},\ \texttt{memory}_{32}$$

Instrucțiunea încarcă registrul **es** şi unul din registrele de uz general de la adresa de memorie specificată. Trebuie notat faptul că adresa de memorie poate fi specificată cu octetul mod/reg/rm dar cal la instrucțiunea **lea** trebuie să fie o locație de memorie şi nu un registru.

Instrucțunea **les** încarcă registrul de uz general specificat cu cuvântul de la adresa specificată ăi registrul **es** cu următorul cuvânt din memorie. Această instrucțiune este companionul instrucțiunii **lds** (care încarcă registrul **ds**) şi sunt singurele instrucțiuni pe 32 de biți la maşinile pre-80386.

Instrucțiunea **add** la fel ca la x86 adună două valori. Instrucțiunea poate avea mai multe forme dar acum ne interesează următoarele cinci forme:

```
add reg, reg
add reg, memory
add memory, reg
add reg, constant
add memory, constant
```

Toate aceste instrucțiuni adună cel de-al doilea operand la primul lăsind rezultatul în primul operand. De exemplu *add bx,5* calculează bx:=bx + 5.

Ultima instrucțiune este **mul** (multiply), instrucțiune ce are un singur operand şi are forma:

```
mul reg/memory
```

Sunt mai multe detalii importante în ceea ce priveşte instrucțiunea **mul** pe care acest capitol le ignoră. Locația de memorie sau registrul sunt de 16 biți. În acest caz instrucțiunea calculează dx:ax := ax*reg/mem. Pentru această instrucțiune nu avem modul imediat de adresare.

2.8. Structura unui program în limbaj de asamblare

Programul în limbaj de asamblare este alcătuit din mai multe linii sursă. 0 linie sursă este alcătuită din următoarele elemente:

<Eticheta> <Mnemonicul> <Operanzii> <Comentariul>

<Eticheta> este un nume simblic asociat unei adrese (locații) de memorie.
<Mnemonicul> se referă la denumirea codului operației unei instrucțiuni, de exemplu add, mov etc. Acest atom lexical poate fi însă şi o directivă (pseudo - operație), de exemplu **.code**, **.data**, **dosseg** etc. Directivele servesc la efectuarea anumitor acțiuni de către asamblor, în timp ce instrucțiunile permit realizarea anumitor operațiuni. <Operanzii> însoțesc de regulă aceste categorii enumerate mai sus (instrucțiuni, directive). De pildă, instructiunea ***mov*** posedă doi operanzi:

mov ax, @data

iar directiva de mai jos, un singur operand:

.stack 200h

<Comentariul>, ultimul din atomii lexicali este constituit dintr-o înşiruire de cuvinte text explicativ - precedat de separatorul punct şi virgulă.

Cei patru atomi lexicali nu trebuie să fie toţi prezenţi pe o aceeaşi linie de program Eticheta poate fi izolată pe o linie, mnemonicul pe alta, el find urmat de operanzi. Comentariul poate să se întindă pe mai multe linii, dar fiecare va trebui să înceapă cu separatorul punct şi virgulă. Atomii lexicali sunt despărţiţi prin blancuri sau tab-uri sau CR. Blank-urie, tab-urile, punctul şi virgula se numesc separatori.

2.8.1. Directivele de segmentare

Un program este alcătuit din cel puţin un segment de cod, unul de date şi unul de stivă.

Sub sistemul de operare DOS sunt posibile şase modele de memorie. Prin model de memorie se înţelege de fapt un mod de dispunere în memoria RAM a segmentelor ce alcătuiesc un program. În tabelul 2.1 se prezintă aceste modele (s-a notat cu LP, LD şi LS lungimea segmentului de cod – program , date respectiv stivă).

Tabelul 2.1.

Modelul	**Lungimea diferitelor componente**
foarte mic (tiny)	LP + LD + LS < 64ko
mic (small)	LP < 64ko şi LD + LS < 64ko
mediu (medium)	64 ko ≤ LP < 1 Mo şi LD + LS < 1 Mo
compact (compact)	LP < 64 ko şi 64 ko ≤ LD + LS < 1 Mo
mare (large)	64 ko ≤ LP < 1Mo şi 64ko ≤ LD + LS < 1Mo
foarte mare (huge)	idem large, doar punctatorii vor fi normalizaţi

Pentru macroasamblorul MASM sau TASM, directivele simplificate de segmentare sunt: **.code**, **.data** şi **.stak** pentru segmentele de cod (program), date şi respectiv, stivă.

Exemplul 1 prezintă un program în limbaj de asamblare cu directive simplificate de segmentare.

Exemplul 1.

```
;Un program care nu face nimic...
;El arata cum se stabilesc segmentele simplificat
;Programul nu poate fi lansat in executie din cauza ca nu are
;apel functie DOS de iesire din program
      dosseg
      .model small
      .stack 200h
      .data
      .code
end
```

Acest program, deşi este corect scris, nu poate fi lansat în execuţie deoarece nu conţine funcţia de reîntoarcere în sistemul de operare DOS. Sistemul de operare DOS, atunci când lansează o aplicaţie în execuţie, predă controlul acesteia, iar la terminarea aplicaţiei, reluarea controlului de către sistemul de operare se face prin apelul funcţiei

sistem 4Ch. Apelul unei funcţii sistem DOS se face prin lansarea înreruperii 21h după ce, în prealabil, numărul funcţiei a fost înscris în registrul ***ah***. Acest lucru se arată în exemplul 2.

Un segment începe după directiva corespunzătoare a acestuia şi se termină la apariţia următoarei directive.

Exemplul 2.

```
;Un program care nu face nimic...
;El arata cum se stabilesc segmentele simplificat
;Programul poate fi lansat in executie din cauza ca are
;apelul functiei DOS 4Ch de iesire din program
      dosseg
      .model small
      .stack 200h
      .data
      .code
Start:
      mov ah,4ch
      int 21h
end Start
```

În exemplul 2, pentru stiva se rezervă 512 octeţi iar începutul programului este marcat de eticheta “Start”.

Pentru obţinerea fişierului executabil vom folosi turboasamblorul TASM.EXE.

La lansarea în execuţie a programului TASM.EXE se afişează:

```
Turbo Assembler Version 3.2 Copyright (c) 1988, 1992 Borland International
Syntax: TASM [options] source [,object] [,listing] [,xref]
/a,/s Alphabetic or Source-code segment ordering
/c Generate cross-reference in listing
/dSYM[=VAL] Define symbol SYM = 0, or = value VAL
/e,/r Emulated or Real floating-point instructions
/h,/? Display this help screen
/iPATH Search PATH for include files
/jCMD Jam in an assembler directive CMD (eg. /jIDEAL)
/kh# Hash table capacity # symbols
/l,/la Generate listing: l=normal listing, la=expanded listing
/ml,/mx,/mu Case sensitivity on symbols: ml=all, mx=globals, mu=none
/mv# Set maximum valid length for symbols
/m# Allow # multiple passes to resolve forward references
/n Suppress symbol tables in listing
/os,/o,/op,/oiObject code: standard, standard w/overlays, Phar Lap, or IBM
/p Check for code segment overrides in protected mode
/q Suppress OBJ records not needed for linking
/t Suppress messages if successful assembly
/uxxxx Set version emulation, version xxxx
/w0,/w1,/w2 Set warning level: w0=none, w1=w2=warnings on
/w-xxx,/w+xxx Disable (-) or enable (+) warning xxx
/x Include false conditionals in listing
/z Display source line with error message
/zi,/zd,/zn Debug info: zi=full, zd=line numbers only, zn=none
```

Programul TASM.EXE sa foloseşte în modul limie de comandă unde se precizează: opţiunile, numele fişierului sursă şi opţional numele fişierului obiect, listă şi referinţe încrucişate. Pentru a asambla fişierul din exemplul 2 se foloseşte linia de comandă (numele fişierului sursă este tdan2.asm):

TASM /l /zi /os tdan2

se obține un fişier în cod obiect relocabil “tdan2.obj” şi un fişier listă “tdan2.lst”. Fişierul listă este prezentat în continuare.

```
Turbo Assembler    Version 3.2       04/12/07 09:55:58       Page 1
tdan2.ASM

 1                        ;Un program care nu face nimic...
 2                        ;El arata cum se stabilesc   segmentele
simplificat
 3                        ;Programul poate fi lansat   in executie din
      cauza ca are
 4                        ;apelul functiei DOS 4Ch de iesire din program
 5                              dosseg
 6    0000                      .model small
 7    0000                      .stack 200h
 8    0000                      .data
 9    0000                      .code
 10   0000                 Start:
 11   0000 B4 4C                mov ah,4ch
 12   0002 CD 21                int 21h
 13                       end Start

Turbo Assembler    Version 3.2       04/12/07 09:55:58       Page 2
Symbol Table

Symbol Name              Type  Value

??DATE                   Text  "04/12/07"
??FILENAME               Text  "tdan2      "
??TIME                   Text  "09:55:58"
??VERSION                Number 0314
@32BIT                   Text  0
@CODE                    Text  _TEXT
@CODESIZE                Text  0
@CPU                     Text  0101H
@CURSEG                  Text  _TEXT
@DATA                    Text  DGROUP
@DATASIZE                Text  0
@FILENAME                Text  TDAN2
@INTERFACE               Text  00H
@MODEL                   Text  2
@STACK                   Text  DGROUP
@WORDSIZE                Text  2
START                    Near  _TEXT:0000

Groups & Segments        Bit Size Align Combine Class

DGROUP                         Group
 STACK                         16 0200 Para      Stack       STACK
 _DATA                         16 0000 Word      Public DATA
_TEXT                    16 0004 Word      Public CODE
```

Pentru obținerea programului în cod obiect direct executabil se folosește editorul de legături TLINK.EXE. Acest program poate elabora atât programme cu extensia **.com** (cu dimensiunea maxima de 64ko) cât şi programme cu extensia **.exe** (care pot avea dimensiuni mai mari de 64ko). Lansarea în execuție a programului duce la afişarea textului următor.

```
Turbo Link Version 3.0 Copyright (c) 1987, 1990 Borland International
Syntax: TLINK objfiles, exefile, mapfile, libfiles
@xxxx indicates use response file xxxx
Options: /m = map file with publics
 /x = no map file at all
 /i = initialize all segments
 /l = include source line numbers
 /s = detailed map of segments
 /n = no default libraries
 /d = warn if duplicate symbols in libraries
 /c = lower case significant in symbols
 /3 = enable 32-bit processing
 /v = include full symbolic debug information
 /e = ignore Extended Dictionary
 /t = create COM file
 /o = overlay switch
 /ye = expanded memory swapping
 /yx = extended memory swapping
```

Linia de comandă pentru obținerea fişierului de tip .exe este:

TLINK tdan2 /v

se generează un fişier cu extensia .exe în cod obiect direct executabil şi un fişier text cu extensia .map ce conține informații despre alocarea memoriei:

```
Start Stop Length Name Class

 00000H 00003H 00004H _TEXT CODE
 00004H 00004H 00000H _DATA DATA
 00010H 0020FH 00200H STACK STACK

Program entry point at 0000:0000
```

Pentru obținerea unui unui fişier direct executabil cu extensia .com se adaugă opțiunea /t. Programele în format **.exe** diferă structural de cele în format **.com**. Pentru a exemplifica acest lucru vom folosi acelaşi program scris pentru obținerea formatului **.exe** (exemplul 3) şi pentru obținerea formatului **.com** (exemplul 4). Prin analiza celor două exemple ne putem da seama de structura fiecărui tip de program.

Exemplul 3. Program scris în scopul linkeditării în format .exe

```
;Program care determina numarul de unitati de disc din sistem si
;afiseaza acest numar

      stiva segment STACK 'STACK'
      dw    100H dup (?)
stiva ends
code  SEGMENT
```

```
      assume cs:code,   ds:code
;
;     Datele programului
;
text1 db    'In configuratie sunt $'
text2 db    ' unitati de disc flexibil$'
;
;     Codul programului

inceput:
      mov   ax,code
      mov   ds,ax
      mov   ah,9
      mov   dx,offset text1
      int   21H         ;afisarea primului text
                        ;ds:dx indica adresa de inceput a textului
                        ;textul se incheie cu caracterul $
      int 11H                 ;determinare configuratie
      and al,0C0H       ;se indica bitii care indica
                        ;numarul unitatii de disc
;

;     Construieste codul ASCII al numarului de unitati de disc
;
      mov cl,6
      shr al,cl
      add al,1 + '0'
      mov dl,al         ;salvare caracter
;
;     tiparire numar de unitati de disc
;
      mov ah,2
      int 21H           ;apel DOS pentru afisarea caracterului
                        ;din registrul dl

;
;     Tiparire sfarsit text
;
      mov ah,9
      mov dx,offset text2
      int 21H
;
;     Sfarsit executie program format EXE
;
      mov ax,4c00h
      int 21H
code  ends
      end   inceput
```

Exemplul 4. Program scris în scopul linkeditării în format .com (opțiunea /t)

```
;Modul de realizare a exemplului 3 sub forma de program COM
;

code  SEGMENT
      assume cs:code,   ds:code
      org   100H
```

```
start:
      jmp   short inceput

;
;     Datele programului
;
text1 db    'In configuratie sunt $'
text2 db    ' unitati de disc flexibil$'
;
;     Codul programului

inceput:
      mov   ah,9
      mov   dx,offset text1
      int   21H         ;afisarea primului text
                        ;ds:dx indica adresa de inceput a textului
                        ;textul se incheie cu caracterul $
      int 11H           ;determinare configuratie
      and al,0C0H       ;se indica bitii care indica
                        ;numarul unitatii de disc
;

;     Construieste codul ASCII al numarului de unitati de disc
;
      mov cl,6
      shr al,cl
      add al,1 + '0'
      mov dl,al         ;salvare caracter
;
;     tiparire numar de unitati de disc
;
      mov ah,2
      int 21H           ;apel DOS pentru afisarea caracterului
                        ;din registrul dl

;
;     Tiparire sfarsit text
;
      mov ah,9
      mov dx,offset text2
      int 21H
;
;     Sfarsit executie program format COM
;
      mov ax,4c00h
      int 21H
      code ends
      end start
```

Un program cu directive de segmentare complete este prezentat în exemplul 5.

Exemplul 5.

```
;program cu directive de segmentare complete
;evident ca programul nu face nimic dar poate fi rulat
;pentru ca are apelul functiei DOS de iesire din program
```

```
;Prima data definim segmentul de stiva.
;In acest segment se rezerva 512 octeti care sunt initializati cu zero
;cuvintul scris intre ghilimele da indicatii linkeditorului cum sa
;grupeze datele

      Stiva SEGMENT para public stack 'stiv'
            db 200h dup(0)
      Stiva ENDS

;Vom defini mai departe doua segmente de date diferite

      Date1 SEGMENT word public 'data' ;primul segment de date
      param1      db 5
      Date1 ENDS

      Date2 SEGMENT word public 'data' ;al doilea segment de date
      param2      db 9
      Date2 ENDS

;Si in sfirsit segmentul de cod
      Cod SEGMENT word public 'code'
;     ASSUME cs:Cod

;Inceputul programului
Start:
      mov ax,Date1
      mov ds,ax ;Adresarea segmentului Date1
      ASSUME ds:Date1

      mov ax,Date2
      mov es,ax ;Adresarea segmentului Date2
      ASSUME es:Date2

;     mov ax,Stiva ;Aceasta secventa de fapt nu e necesara...
;     mov ss,ax    ;Adresarea segmentului de stiva

      ASSUME ss:Stiva

;Programul propriu-zis

      mov al,param1
      mov al,param2

      ;Aici evident se pune secventa de iesire
      mov ah,4ch
      int 21h

      Cod Ends
      end Start
```

Acest stil de programare oferă o mai mare flexibilitate în privința amplasării în memorie a segmentelor.

Se remarcă amplasarea directivelor **SEGMENT** şi **ENDS** care delimitează un segment al cărui nume apare în fața ambelor directive. Directivele pot fi scrise cu litere mai sau mici. Segmentul este o grupare logică de elemente, adresate prin intermediul unui registru de segment. Elementele ce îl alcatuiesc pot fi instrucțiuni şi/sau date. Cuvântul rezervat **PARA** (provine de la « paragraph ») este un operand al directivei

SEGMENT, prin care se anunță că implantarea segmentului se va face la o adresă multiplu de 16. În schimb **_TEXT** şi **_DATE** vor fi alineate la adrese multiplu de doi (operandul **WORD**). În sfârşit cuvântul rezervat **PUBLIC** va face cunoscute denumirile acestor segmente în exteriorul programului (util pentru linkeditor).

Directiva **ASSUME** asociază un registru de segment la segmentul respectiv.

2.8.2. Directivele pentru definirea datelor

Se permit următoarele forme de definire a datelor:

- definirea unui octet (*define byte*, **db** sau **DB**) ;
- definirea unui cuvânt (*define word*, **dd** sau **DD**);
- definirea unui dublu cuvânt (*define double word*, **dw** sau **DW**);
- definirea unui cuvânt de 6 octeți (*define float*, **df** sau **DF**);
- la fel (*define pointer*, **dp** sau **DP**);
- definirea unui cuv-nt de 8 octeți (*define quad word*, **dq** sau **DQ**);
- definirea unei zone de 10 octeți (*define ten bytes*, **dt** sau **DT**).

2.8.3. Concluzii privind limbajul de asamblare

Un program scris in limbaj de asamblare urmareste topica urmatoare:

- Controlul segmentarii si adresarii (SEGMENT/ENDS, ASSUME, GROUP) incluzind incarcarea registrelor de segment, a segmentelor de cod si diverse consideratii asupra instructiunilor de sir (MOVS, MOVSB).
- Definirea etichetelor (LABEL).
- Definirea procedurilor (PROC, ENDP).
- Legarea programelor (NAME, END, PUBLIC, EXTRN).
- Controlul numaratorului de instructiuni (ORG).

Segmentul este cea mai mica unitate de memorie relocabila şi el are cel mult 64 ko.

Fiecare bloc este continu (nu sint permise gauri in segment) dar segmentele pot fi imprastiate prin memorie.

Puteti defini cite segmente doriti dar trebuie sa fie definit cel putin unul pe modul de asamblare (daca se omite instructiunea de definire a segmentului, asamblorul asigneaza automat numele ??SEG). Fiecare instructiune si fiecare data din programul dumneavoastra trebuie sa apartina unui segment. Nu exista nimic care sa impiedice amestecarea codului si a datelor in segmente.

Citeva exemple practice de segmentare sint:

- un segment pentru date globale;
- un segment pentru date locale;
- un segment pentru stiva;
- un segment pentru programul principal;
- un segment pentru subrutinele reentrante;
- un segment pentru vectorii de intrerupere;

- un segment pentru rutinele de intreruperi.

Un segment fizic consta din cel mult 65535 (64K) octeti incepind de la o adresa absoluta divizibila prin 16. O astfel de adresa se numeste limita de paragraf. Cum un segment logic nu e necesar sa inceapa la o limita de paragraf, nu e necesar ca segmentele logice sa corespunda celor fizice.
Cum fiecare segment incepe in anumite paragrafe, registrele de 16 biti (CS, DS, ES, SS) sunt folosite sa pastreze numerele paragrafelor unde incep segmentele.

La executie, fiecare referinta la memorie necesita doua componente pentru adresarea fizica propriu-zisa:

1. Valoarea bazei segmentului de 16 biti ce este continuta intr-unul din registrele CS, DS, SS, ES.
2. O adresa efectiva de 16 biti care da offset-ul memoriei.

Adresa propriu-zisa (fizică) este calculata :

adresa fizică (20 biti) = 16 * (adresa de baza a segmentului) + adresa efectiva (offset)

Posibilitatile operanzilor

Operanzii din memorie pot fi adresati direct cu o adresa de offset de 16 biti, sau indirect cu baza (BX sau BP) si/sau indexul (SI sau DI) adunate la un deplasament optional de 8 sau 16 biti.

Rezultatul unei operatii cu doi operanzi poate fi pus direct in memorie sau registru. Operatiile cu un singur operand se aplica oricarui operand, exceptie constantele imediate.

Legatura segmentarii cu modulele de asamblare

Modulul de asamblare poate rezulta din:

- parte de segment
- un segment
- parti din mai multe segmente
- mai multe segmente

si orice combinatie a acestora in functie de utilizarea directivelor SEGMENT/ENDS. Dupa asamblare se pot combina din nou cu ajutorul programului LINK.

Controlul segmentarii si adresarea

In timpul executiei orice instructiune si variabila se gaseste in cadrul unui anumit segment. Daca nu i-ati dat un nume asamblorul creaza automat unul, ??SEG. Pentru a numi segmentul, a controla alinierea si continuitatea lui se folosesc directivele SEGMENT, ENDS:

[nume-seg] SEGMENT [tip aliniere][tip combinare]['nume clasa']

.
.
.
[nume-seg] ENDS

unde:

tip aliniere specifica la ce tip de limita va fi locatat segmentul:

1. PARA - (automat) - la o adresa divizibila cu 16 (ceea mai putin semnificativa cifra egala cu 0) ;
2. BYTE - oriunde ;
3. WORD - la o adresa limita de tip cuvint (cel mai putin semnificativ bit 0) ;
4. PAGE - la o adresa limita de tip pagina (ultimele doua cifre hexa 0) ;
5. INPAGE - intreg segmentul ocupa mai putin de 256 octeti si acesta, locatat nu trebuie sa depaseasca limita paginii.

tip combinare specifica cum acest segment poate fi combinat cu alte segmente pentru legare si locatare:

1. necombinabil
2. PUBLIC - specifica ca acest segment va fi concatenat cu altele, de acelasi nume la legare (LINK)
3. COMMON - specifica ca acest segment si toate segmentele de acelasi nume care se leaga impreuna vor incepe la aceeasi adresa, astfel suprapunindu-se. Lungimea este cea corespunzatoare lungimii segmentului maxim legat.
4. AT expresie - specifica ca acest segment va fi locatat la un paragraf evaluat de expresia data (ex: AT 4444H inseamna o adresa absoluta de memorie. Expresia poate fi orice expresie valida rezultind o constanta, dar nu se permit referinte inainte).
5. STACK - specifica ca acest segment este o parte din stiva de executie, adresata de tip LIFO. Aceste segmente sint puse inaintea memoriei mari si cresc descrescator. Memoria alocata segmentului de stiva este suma alocarilor pentru fiecare segment individual.
6. MEMORY - specifica ca acest segment trebuie locatat deasupra tuturor segmentelor legate impreuna. Daca se intilnesc mai multe segmente de tip MEMORY numai primul este tratat ca atare, restul sint tratate ca segmente COMMON.

'nume clasa' specifica un nume de clasa pentru segment:

- CODE
- CONST
- DATA
- STACK
- MEMORY

Segmente inlantuite

Segmentele nu sint niciodata inlantuite fizic, totusi este permis sa se codifice o portiune a unui segment, se porneste si se sfirseste altul, apoi se revine la codificarea primului. Asamblorul de fapt concateneaza a doua portiune a segmentului la prima.

Nu se permite suprapunerea segmentelor ci doar inlantuirea lor lexicala.

Directiva ASSUME

Directiva ASSUME construieste o legatura simbolica intre:

- definirea instructiunilor in timpul asamblarii si datele in segmentele logice si
- evenimentele de executare a instructiunilor cu registrele de segment.

Cu alte cuvinte, ASSUME e o promisiune data asamblorului ca instructiunile si datele sint adresate in timpul executiei prin intermediul anumitor registre. Incarcarea actuala si manipularea valorilor este responsabilitatea programatorului. ASSUME permite asamblorului sa verifice ca fiecare data si instructiune este adresata corespunzator.

Directiva LABEL

Directiva LABEL creeaza un nume pentru locatia curenta de asamblare, data sau instructiune.

nume LABEL tip

unde :

nume este asignat atributelor urmatoare:

- segment - segmentul curent de asamblare;
- offset - offset-ul in cadrul segmentului curent;
- tip - operandul lui LABEL.

tip poate fi :

- NEAR sau FAR daca urmeaza cod executabil. Eticheta poate fi utilizata in JMP sau CALL dar nu in MOV sau alte instructiuni de manipulare a datelor ;
- BYTE, WORD, DWORD, nume de structura sau nume de inregistrare daca urmeaza date. Se poate indexa un identificator declarat cu LABEL daca directiva asigneaza un tip: BYTE, WORD. In acest caz numele este o variabila si e valid in MOV dar nu in JMP sau CALL.

Principalele utilizari ale lui LABEL sint:

- a accesa variabile (tablouri) prin BYTE sau WORD dupa cum e nevoie ;
- definirea unei etichete de tip FAR ;
- furnizarea unei etichete de tip NEAR, existente, care are valoare de segment si offset determinate, calitatea de FAR, putind fi astfel accesate si din alte segmente.

Proceduri (directivele PROC/ENDP)

Limbajul de asamblare furnizeaza proceduri pentru a implementa conceptul de subrutina.

```
nume PROC [NEAR/FAR]
        .
        .
        .
        RET
        .
        .
        .
nume ENDP
```

unde "nume" este un identificator care trebuie sa apara in PROC si ENDP. Se asigneaza tipul NEAR sau FAR dupa cum este specificat, implicit este NEAR. Trebuie specificat FAR daca procedura va fi apelata din cod cu alta valoare ASSUME CS:. Tipul procedurii determina daca RET este asamblat NEAR sau FAR.

Apelul unei proceduri

Cind se apeleaza o procedura NEAR, numaratorul de instructiuni este salvat (IP) in stiva si se transmite controlul primei instructiuni din procedura.

Cind se apeleaza o procedura FAR, registrul CS si apoi IP sint salvate in stiva si se transfera controlul. Se permit puncte de intrare multiple intr-o procedura, tipul acestor puncte poate fi diferit.

Intoarcerea din proceduri

O procedura se termina cind se specifica instructiunea RET (din rutina de intreruperi IRET). Pot apare mai multe RET-uri in procedura si nu e necesar ca ultima instructiune sa fie RET. Intoarcerea dintr-o procedura FAR va pune virful stivei in IP si urmatorul cuvint in CS; la o procedura NEAR virful stivei se pune in IP.

Daca procedura utilizeaza stiva pentru memorarea unor date temporare, aceste date trebuie sa fie descarcate inainte de intoarcerea din procedura.

Directive pentru legarea programelor NAME/END, PUBLIC, EXTRN

Utilizind LINK si LOCATE se pot lega si reloca in vederea executiei mai multe module de program intr-unul singur.

Pentru a identifica referintele simbolice intermodulare se pot utiliza directivele:

- NAME - asigneaza un nume modulului obiect generat de asamblor
- PUBLIC - specifica simboli definiti in acest modul de asamblare ai caror atribute sint facute disponibile altor module in faza de legare
- EXTRN - specifica simboli definiti in alte module de asamblare ale caror atribute sint necesare acestui modul in faza de legare

O buna programare urmareste declararea etichetelor externe si a variabilelor intr-un fisier INCLUDE pentru fiecare modul asamblat care contine declaratii EXTRN pentru toti simbolii declarati PUBLIC in el. Fisierul INCLUDE ar trebui sa contina perechi SEGMENT PUBLIC/ENDS pentru fiecare segment si intre ele o directiva EXTRN listind variabilele (cu tipurile lor) pentru acest segment.

END [nume eticheta]

exista un singur END intr-un fisier sursa si trebuie sa fie ultima instructiune.
Daca exista, "nume eticheta" este folosit ca adresa de start pentru executia programului. Daca sint mai multe module de legat impreuna, numai unul poate specifica o adresa de start. Acest modul este modulul principal.

Numaratorul de instructiuni ($) si directiva ORG

Numaratorul de instructiuni contine o valoare (reprezentata simbolic prin ($)) care spune asamblorului ce offset in segmentul curent are urmatoarea instructiune sau data de asamblat.

Directiva ORG poate fi utilizata pentru a incarca cu o anumita valoare numaratorul de instructiuni

ORG expresie

unde "expresie" este evaluata modulo 65536 si trebuie sa nu contina nici o referinta inainte. Directiva nu poate avea eticheta.

Utilizarea operatorului SHORT

Cind saltul este in interiorul segmentului si deplasamentul relativ al lui este in gama -128 si 127 de octeti si scopul (tinta) etichetei nu a fost inca definita, poti salva un octet prin codificarea operatorului SHORT inaintea etichetei.

PTR este util, ca in exemplele urmatoare:

- incrementarea unui octet sau cuvint din memorie:

INC BYTE PTR [BX]
INC WORD PTR [SI]

- muta o valoare imediata intr-un octet sau cuvint din memorie:

MOV WORD PTR [DI],99
MOV BYTE PTR [DI],99

- salt cu doua nivele de indirectare:

JMP DWORD PTR [BX]

Operatorii HIGH si LOW

Se mai numesc si operatori de izolare a octetului. Accepta ca argument un numar sau o expresie de tip adresa, HIGH intoarce octetul mai semnificativ; LOW cel mai putin semnificativ.

Ierarhia operatorilor

Clasele de operatori in ordine descrescatoare de precedenta sint:

1) expresii cu paranteze, paranteze drepte la expresii, paranteze rotunde la expresii, (.) in structuri, LENGTH, SIZE, WIDTH, MASK.
2) PTR, OFFSET, SEG, TYPE, THIS si "nume:" (suprapunere de segmente).
3) HIGH, LOW.
4) /, *, MOD, SHL, SHR
5) +, -
6) relationali : EQ, NE, LT, LE, GT, GE
7) NOT logic
8) AND logic
9) OR, XOR logic
10) SHORT

Directiva EQU

Unui simbol i se poate asigna o valoare in timpul asamblarii utilizind EQU. Formatul este:

nume EQU expresie

Prezentăm în continuare citeva exemple de programe scrise în limbaj de asamblare.

Exemplul 6.

```
;Program care prezinta un mod de rezervare a stivei

stiva segment stack
      dw 100 dup(0)     ;initializare cu zero a unei zone
```

```
                        ;de 100 x 2 octeti
varf_stiva label word   ;asociere nume pentru aceasta adresa
stiva ends
initializare_stiva segment
      assume cs:initializare_stiva, ss:stiva
start:            mov ax,stiva      ;actualizare registru segment stiva
            mov ss,ax
            mov sp,offset varf_stiva ;actualizare indicator varf stiva
;
;incheiere program
      mov ax,4c00h
      int 21h
initializare_stiva ends

end start
```

Exemplul 7.

```
;Varianta cu directive de segmentare complete
;Programul realizeaza suma a doua numere
;Programul arata definirea si apelul unei proceduri
;
StkSegment  segment     para stack 'stack'
            db 512 dup(?)
StkSegment  ends
;
DataSegment segment     word 'data'
Lista1            db 6,8,?
Lista2            db 10,35,?
DataSegment ends
;
CodSegment  segment word 'code'
      assume cs:CodSegment,ds:DataSegment,ss:StkSegment
;Procedura de adunare a doua valori reprezentate pe cate un octet
;depune rezultatul in memorie peste elementul din fiecare lista
;notat cu ? (locatie de memorie neinitializata)
;
adsub proc
      mov al,[si]
      add al,[si+1]
      mov [si+2],al
      ret
adsub endp
;
;Programul principal
;Punctul de intrare
;
StartProgram:
      mov ax,DataSegment
      mov ds,ax
      mov si,offset Lista1
      call adsub
      lea si,Lista2
      call adsub
      mov ax,4c00h
      int 21h
CodSegment  ends
end StartProgram
```

Exemplul 8.

```
;Program pentru sortarea in ordine descrescatoare a unui sir cu 10
;numere.
;dimensiunea maxima a numerelor de ordonat este de un octet. Numarul
;maxim de numere ale sirului este de 255.
;
;Sumarul utilizarii registrelor
;AH - pastreaza valoarea maxima a numerelor din sir + 1
;AL - prima valoare din sir de comparat
;BX - contor in sir - BH este nefolosit si este initializat cu zero
;iar BL arata pozitia in sir cu care se compara AH
;DH - contine numarul maxim de termeni ai sirului de comparat
;DL - contine cea de-a doua valoare din sir cu care se compara AL
;CH - stocheaza temporar valoarea DL
;CL - stocheaza temporar valoarea BL
;
dosseg ;directiva de segmentare pentru sistemul de operare DOS

.model small      ;model mic, lungime program < 64 Ko si lungime date
                  ;si lungime stiva (impreuna) < 64 Ko. Compilatorul
                  ;va da un mesaj de eroare daca sunt depasite aceste
                  ;dimensiuni.

.stack 200  ;pentru stiva sunt rezervati 200 de octeti

.data       ;zona de date
sir_numere  db 5,1,7,2,8,0,3,6,4,9 ;sirul de numere ce urmeaza a fi
                                   ;ordonat
index_sir   db 0  ;zona de memorie unde se pastreaza indexul curent
nr_max_valori     db 10 ;numarul maxim de valori pentru sir_numere

.code
start:                  ;inceput program

;Sortarea sirului se face astfel: se citeste primul numar si se
;compara cu urmatoarele. Daca se gaseste un numar mai mic acestea se
;inverseaza. La sfirsit numarul este memorat in sir si se trece la
;urmatorul, si asa mai departe.

      mov ax,@data ;se initializeaza segmentul de date
      mov ds,ax
      mov ah,[nr_max_valori] ;AH va pastra numarul maxim de termeni ai
                        ;sirului
      mov dh,ah         ;La fel si registrul DH
      inc ah            ;Din motive de comparare in AH vom avea numarul
                        ;maxim de termeni + 1
      mov si,offset sir_numere ;in SI se pune offsetul sirului de
                        ;numere
valoare_noua:
      mov bh,0    ;BH este initializat la zero pt. ca este folosit
                  ;la adresarea sirului
      mov bl,[index_sir]  ;BL contine pozitia primului numar din sir
      mov al,[bx][si]   ;AL prima valoare din sir
      inc bx            ;se trece la urmatoarea valoare din sir
      cmp dh,bl         ;se verifica daca nu s-au comparat toate
                        ;valorile
```

```
      je sfirsit        ;s-au comparat toate valorile si s-a terminat
                        ;programul deci fac salt la sfirsit
      mov [index_sir],bl  ;memorez pozitia primei valori de comparat.
                        ;Aceasta va fi comparata succesiv cu valorile
                        ;urmatoare din sir
      dec bx            ;pentru ca urmeaza o bucla se decrementeaza
                        ;valoarea
comparare:
      inc bx            ;se trece la urmatoarea valoare din sir
      cmp ah,bl         ;se verifica daca nu s-au comparat toate
                        ;valorile
                        ;din sir
      je valoare_noua   ;daca da, se trece la o alta valoare de
                        ;comparat
                        ;in registrul AL
      mov dl,[bx][si]   ;DL contine valoarea cu care se compara AL,
                        ;valoarea
                        ;urmatoare din sir
      cmp al,dl         ;se compara cele doua valori din sir
      jb comparare      ;daca in AL este o valoare mai mica decit in DL
                        ;atunci se trece la urmatoarea valoare de
                        ;comparat
      mov cl,bl         ;daca nu atunci de inverseaza cele doua valori
                        ;in memorie si in registrele AL si DL. Aici se
                        ;stocheaza temporar indexul in sir
      mov ch,dl         ;se stocheaza temporar valoarea din DL
      mov [bx][si],al   ;se memoreaza valoarea mai mare in locul celei
                        ;mai mici
      mov bl,[index_sir]  ;se determina adresa valorii mai mici
      dec bl            ;aceasta a fost incrementata deci se
                        ;decrementeaza
      mov [bx][si],dl   ;se memoreaza valoarea mai mica
      mov bl,cl         ;se reface indexul sirului (unde s-a ramas cu
                        ;compararea valorilor
      mov dl,al         ;se inverseaza cele doua valori si in registre
      mov al,ch
      jmp comparare     ;se continua compararea
sfirsit:                ;programul s-a terminat si ne reintoarcem in
                        ;sistemul de operare
      mov ax,4c00h      ;iesire din program
      int 21h

end start
```

2.9. Scrierea aplicațiilor Windows în limbaj de asamblare

Aplicațiile în limbaj de asamblare sub Windows pot fi realizate în mai multe feluri. Prezentăm în continuare includerea unor secvențe în limbaj de asamblare în programele scrise pentru Visual Basic.

2.9.1. Includerea limbajului de asamblare în programele Visual Basic

Vom explica în continuare noțiunile de bază ale amestecării limbajului de asamblare cu Visual Basic. Nu se explică utilizarea limbajului de asamblare. Codul utilizat aici este realizat cu NASM 0.97 care se poate prelua gratuit de pe Internet.

Avertisment: depanarea este extrem de dificila, mediul de programare nefiind în acest caz de nici un folos.

Realizarea programului în limbaj de asamblare.
De ce aveți nevoie:

- Visual Basic sau Custom Control Edition – gratuit – de pe site Microsoft;
- NASM – gratuit de pe site NASM;
- Editor de text.

Pentru a utiliza NASM cu Vizual Basic trebuie să construiți un DLL cu NASM iar după aceeea veți utiliza acest DLL cu Visual Basic (VB3 nu permite utilizarea DLL-urilor). Utilizarea DLL-ului se face în acelaşi fel ca DLL-urile sistemului Windows ca user32.dll sau gdi32.dll.

Pentru a face un DLL cu NASM se parcurg trei pasi:

- se scrie codul cu NASM;
- după scrierea codului se linkeditează cu linker-ul Visul Basic;
- apoi se foloseşte cu Visual Basic.

De exemplu vom face un DLL pantru adunarea a doi întregi (Dwords):

Codul în limbaj de asamblare este:

```
        SEGMENT code USE32
        GLOBAL _DllMain        ;Just a small routine that gets called.
_DllMain:       mov     eax, 1 ;Dont worry about this.
                        retn    12

;Sub addLongs (ByRef number1 As Long, ByVal number2 As Long)
        GLOBAL addLongs
addLongs:
        enter   0, 0
        mov     eax, [ebp+8]    ;pointer to number1
        mov     ecx, [eax]      ;ecx = number1
        add     ecx, [ebp+12]   ;ecx = number1 + number2
        mov     [eax], ecx      ;number1 = number1 + number2
        leave
        retn 8           ;return, with 8 bytes of arguments (2 DWords)


        ENDS
```

Prima linie spune asamblorului NASM că este vorba de un program Windows de 32 de biți. Această linie trebuie să fie în toate DLL-urile înainte de orice cod.

A doua linie spune linkeditorului că **_DLLMain** va fi un nume global. Linkeditorul va permite ca acest nume să fie apelat de Visual Basic. Trebuie să declarați toate procedurile dumneavoastră ca GLOBAL altfel ele nu vor fi văzute de Visual Basic.

A treia linie are numele (eticheta) numită **_DllMain** şi astfel linkeditorul va şti unde este **_DllMain**.

Cele două linii de cod pe care le are **_DllMain** fac registrul **eax** egal cu 1, scoate 12 octeți pentru argumentele sale şi se reântoarce la apelant. Este o rutină specială şi nu trebuie să vă faceți griji în legătură cu ea.

Prima linie a celei de-a doua rutină începe cu **;** (punct şi virgulă) care arată că este vorba de un comentariu şi vă arată cum trebuie să o apelați în Visual Basic. Aceasta înseamnă că locația de memorie actuală este transferată ca referință pe cînd celălalt argument este transferat ce valoare. Prima variabilă este trimisă ca referință şi deci ea poate fi schimbată pentru că avem locația acesteia. A doua variabilă nu poate fi schimbată de această rutină deoarece avem numai valoarea ei.

A doua linie (a celei de-a doua rutine) arată că **addLongs** este o etichetă pe care o va vedea şi Visual Basic.

A treia linie arată unde este **addLongs**.

A patra linie salvează registrul **ebp** şi îl setează la începutul stivei **call** (acest lucru este necesar numai dacă vreți să utilizați argumente). **EBP + 8** va fi locația primului argument. Din acuză că Windows 9x este pe 32 de biți, fiecare argument trebuie să aibă 32 de biți, adică 4 octeți, iar argumentele sunt unul după celălalt, al doilea argument va fi la **EBP + 12**, al treilea la **EBP + 16**, următorul la **EBP + 20** şi aşa mai departe.

A cincea linie setează **EAX** la primul argument. Din cauză că primul argument este transferat ca referință, **EAX** va fi egal cu adresa locației de memorie unde este stocat prima variabilă.

A şasea linie face registrul **ECX** egal cu valoarea primului argument.

A şaptea linie adună valoarea celui de-al doilea argument la **ECX**, rezultatul fiind reținut în **ECX**.

A opta linie va seta prima variabilă la valoarea **ECX**. În acest fel "numărul1" utilizat în Visual Basic va avea valoarea "numărul1" + "numărul2".

Penultima linie va anula ceea ce a făcut ENTER. Aceasta trebuie folosită dacă s-a folosit ENTER.

Ultima linie marchează sfârşitul rutinei şi se reîntoarce la apelant (Visual Basic) şi arată de asemenea numărul de octeți ai argumentului care se trimit acestuia.

ENDS arată sfîrşitul fişierului

După aceasta se face compilarea cu NASM.

Se poate folosi un fişier BAT numit "MakeDLL.bat" unde se trec toate argumentele pentru NASM:

C:\Nasm\NasmW.exe –f coff myDLL.asm
C:\VB5\Link.exe /dll /export:addLongs /entry:DllMain myDll.o
del myDLL.exp
del myDLL.lib
del myDLL.o

Prima linie face compilarea în format COFF acceptat de linkeditorul Visual Basic. Daca doriti listingul adaugați " –l myDLL.lst" la sârşitul primei linii.

A doua linie va linkedita formatul COFF ".o" într-un fişier ".dll" care poate fi folosit cu VB. Primul argument "/dll" arata ca trebuie facut un fişier DLL. Fara acest argument va fi facut un fişier EXE. Argumentul "/export:addLongs" arată că numele "addLongs" va fi vizibil în VB. Trebuie făcut aceasta pentru orice rutină din DLL altfel VB nu va fi capabil să le găsească. Următorul argument "/entry:DllMain" arată unde

este rutina “DLLMain”. Acest argument trebuie folosit întotdeauna. Ultimul argument “myDLL.o” arată care este fişierul de linkeditat.

Apar nişte mesaje la prelucrare dar acestea nu se vor lua în considerare.

Utilizarea DLL-ului creat în VB.

DLL-ul trebuie să fie în acelaşi director cu programul VB.

```
Option Explicit

Private Declare Sub addLongs Lib "samples\ASM\myDLL" (ByRef number1 As
Long , ByVal number2 As Long)


Private Sub Form_Click()
    Dim x As Long, y As Long
    x = 200
    y = 5
    Print "x = "; x
    Print "y = "; y
    addLongs x, y
    'If it reached this line, then its probably perfect.
    Print "Added y to x, so now x = "; x
    'The answer better be 205, otherwise you stuffed something up!
End Sub
```

A doua linie declară rurtina care este folosită în VB. Trebuie făcute aceste declaraţii pentru fiecare rutină pe care o folosiţi chiar dacă ele sunt toate în acelaşi DLL. (Aici ea este declarată ca “Private” ceea ce inseamnă că numai această formă VB va putea s-o acceseze. Puteţi s-o puneţi într-un modul separat (“.bas”) astfel încât toate formele din proiectul dumneavoastră VB s-o poată accesa. În acest caz va trebui să eliminaţi cuvântul “Private”. Evident trebuie să schimbaţi directorul (relativ la directorul VB) pantru a arăta unde este DLL-ul pe calculator, dacă acesta este în altă parte.

Rezultatul afişat trebuie să fie 205

Puteţi face aproape orice se poate face în DOS în DLL-ul creat cu excepţia utilizării întreruperilor şi trebuie ţinut cont că se programează în modul protejat. Dacă nu aţi programat niciodată în modul protejat asm nu vă faceţi griji. Tot ce aveţi de făcut este să ţineţi cont de faptul că segmantele nu sunt de 64k, ele sunt enorme şi din acest motiv nu trebuie să utilizaţi decât un singur segment, acest lucru vă permite accesul la megaocteţi de memorie! (Windows lucrează în modul protejat).

Se pot face multe lucruri în DLL ce nu pot făcute înVB cum ar fi:

- să vă construiţi propriile rutine BitBlt sau StretchBlt
- să scrieţi propriile rutine sistem DLL
- utilizarea octetului inferior dintr-un întreg
- să folosiţi o locaţie de memorie ca variabilă
- să folosiţi conversia ASCII a datelor binare sau în virgulă mobilă
- sa realizaţi înmulţiri sa împărţiri prin deplasări
- să optimizaţi codul pentru un anumit tip de calculator
- să optimizaţi buclele
- şi multe altele…

CAPITOLUL 3

PROGRAMAREA MICROPROCESORULUI TMS 320F240

Microprocesorul Texas Instruments TMS 320F240 posedă un set complex de instrucțiuni, puternic şi flecxibil care permite utilizarea acestuia şi ca procesor de semnal. Pentru programare se foloseşte cross-asamblorul TASM care permite obținerea fişierelor în cod obiect direct executabile, fişiere ce sunt transferate în memoria sistemului cu microcontroler TMS 320F2040 şi lansate în execuție. Depanarea se face direct în sistem prin legătura ce se poate stabili prin interfața serială sau prin intermediul dispozitivelor de tip JTAG.

3.1. Setul de instrucțiuni a procesoarelor Texas Instruments C5X/C2XX

Categoriile de instrucțiuni:

- instrucțiuni pentru folosirea acumulatorului şi a memoriei;
- instrucțiuni pentru Registrele Auxiliare şi Poinerul pentru pagina de date;
- instrucțiuni pentru registrul T, registrul P şi înmulțire;
- instrucțiuni de salt şi apel subprogram;
- operații I/O şi cu memoria de date;
- instrucțiuni de control;

Tipuri individuale de instrucțiuni:

ABS Valoarea absolută a acumulatorului;
ADD Adună cu acumulatorul;
ADDC Adună cu acumulatorul şi carry;
ADDS Adună la partea inferioară a acumulatorului cu suprimarea extensiei de semn;
ADDT Adună la partea superioară a acumulatorului cu deplasarea specificată în registrul T;
ADRK Adunnare imediată a registrului auxiliar cu valoare pe 8 biți;
AND Operație ŞI cu acumulatorul;
APAC Adună registrul P cu acumulatorul;

APL	Operație ŞI cu dată din memorie cu DBMR (Dynamic Bit Manipulation Register) sau constantă pe 16 biți;
B	Salt necondiționat;
BACC	Salt la locația specificată de acumulator;
BANZ	Salt dacă registrul auxiliar nu este zero;
BCND	Salt condiționat;
BIT	Test bit;
BITT	Test bit specificat de registrul TREG2;
BLDD	Mutarea unui bloc din memoria de date în memoria de date;
BLPD	Mutarea unui bloc din memoria de program în memoria de date;
CALA	Apelul unui subprogram din locația specificată de acumulator;
CALL	Apel necondiționat de subprogram;
CC	Apel condiționat de subprogram;
CLRC	Şterge bitul de control;
CMPL	Complementează acumulatorul;
CMPR	Compară registrul auxiliar cu registrul ARCR;
DMOV	Încarcă o dată în memoria de date;
IDLE	Aşteaptă o întrerupere;
IN	Citeşte o dată din port;
INTR	Întrerupere software;
LACC	Încarcă acumulatorul cu deplasare;
LACL	Încarcă partea mai puțin semnificativă a acumulatorului şi şterge partea mai semnificativă;
LACT	Încarcă acumulatorul cu deplasarea specificată de registrul TREG1 a datei încărcate;
LAR	Încarcă registrul auxiliar;
LDP	Încarcă pointerul de pagină a memoriei de date;
LPH	Încarcă partea cea mai semnificativă a registrului P;
LST	Încarcă registrul de stare;
LT	Încarcă TREG0
LTA	Încarcă TREG0 şi adună produsul precedent;
LTD	Încarcă TREG0, adună produsul precedent şi salvează data;
LTP	Încarcă TREG0 şi memorează registrul P în acumulator;
LTS	Încarcă TREG0 şi scade produsul precedent;
MAC	Înmulțeşte şi adună;
MACD	Înmulțeşte şi adună cu salvarea datei;
MAR	Înmulțeşte registrul auxiliar;
MPY	Înmulțeşte (cu registrul T, memorează produsul în registrul P)
MPYA	Înmulțeşte şi adună produsul precedent;
MPYS	Înmulțeşte şi scade produsul precedent;
MPYU	Înmulțire fără semn;
NEG	Neagă acumulatorul;
NMI	Întrerupere nemascabilă;
NOP	Nu execută nici o operație;
NORM	Normalizează conținutul acumulatorului;
OPL	Operația SAU cu DBMR sau cu data pe 16 biți;
OR	Operația SAU cu acumulatorul;
OUT	Trimitere dată la port;

PAC	Încarcă acumulatorul cu registrul P register;
POP	Încarcă din vârful stivei partea mai puțin semnificativă a acumulatorului;
POPD	Încarcă din vârful stivei o dată în memorie;
PSHD	Depune din vârful stivei o dată din memorie;
PUSH	Depune partea mai puțin semnificativă a acumulatorului în stivă;
RET	Reîntoarcere din subprogram;
RETC	Reîntoarcere condiționată din subprogram;
ROL	Roteşte acumulatorul stânga;
ROR	Roteşte acumulatorul dreapta;
RPT	Repetă instrucțiunea următoare de un număr de ori specificat de o valoare din memoria de date;
SACH	Salvează partea mai semnificativă a acumulatorului cu deplasare;
SACL	Salvează partea mai puțin semnificativă a acumulatorului;
SAR	Salvează registrul auxiliar;
SBRK	Scade din registrul auxilia o dată pe 8 biți;
SETC	Setează bitul de control;
SFL	Deplasează acumulatorul stânga;
SFR	Deplasează acumulatorul dreapta;
SPAC	Scade registrul P din acumulator;
SPH	Salvează partea cea mai semnificativă a registrului P;
SPL	Salvează partea cea mai puțin semnificativă a registrului P
SPLK	Salvează o valoare pe 16 biți;
SPM	Setează modul de deplasare pentru registrul P;
SQRA	Ridică la pătrat şi adună produsul precedent;
SQRS	Ridică la pătrat şi scade produsul precedent;
SST	Salvează registrul de stare;
SUB	Scade din acumulator;
SUBB	Scade din acumulator cu împrumut;
SUBC	Scădere condițională;
SUBS	Scădere din partea mai puțin semnificativă a acumulatorului, fără extensie de semn;
SUBT	Scădere din acumulator cu deplasare specificată de registrul T;
TBLR	Citire tabel;
TBLW	Scriere tabel;
TRAP	Întrerupere software;
XOR	SAU exclusiv cu acumulatorul;
ZALR	Scrie zero în partea mai puțin semnificativă a acumulatorului şi încarcă partea mai semnificativă prin rotunjire.

Sintaxa instrucțiunilor

INSTRUCȚIUNI REFERITOARE LA ACUMULATOR ŞI MEMORIE

ABS ADD ADDC ADDS ADDT AND CMPL LACC LACL LACT NEGNORM OR ROL ROR SACH SACL SFL SFR SUB SUBB SUBC SUBS SUBT XOR ZALR

ABS Absolute Value of Accumulator

```
            [label]         ABS
ADD         Add to Accumulator with Shift
            Direct: [label]                 ADD   dma [,shift ]
            Indirect:       [label]                 ADD   {ind} [,shift [, next ARP]]
            Short Immediate:[label]                 ADD   #k
            Long Immediate:[label]                  ADD   #lk [,shift ]
ADDC        Add to Accumulator with Carry
            Direct: [label]         ADDC dma
            Indirect:       [label]         ADDC {ind} [, next ARP]
ADDS        Add to Accumulator with Sign-Extension Suppressed
            Direct: [label]         ADDS dma
            Indirect:       [label]         ADDS {ind} [, next ARP]
ADDT        Add to Accumulator with Shift Specified by T Register
            Direct: [label]         ADDT dma
            Indirect:       [label]         ADDT {ind} [, next ARP]
AND         AND With Accumulator
            Direct: [label]                 AND   dma
Indirect:   [label]                 AND   {ind}  [, next ARP]
            Long Immediate:[label]                  AND   #lk  [, shift]
CMPL        Complement Accumulator
            [label]         CMPL
LACC        Load Accumulator With Shift
            Direct: [label]         LACC dma, [,shift1 ]
            Indirect:       [label]         LACC {ind} [,shift1 [ next ARP]]
            Immediate:[label]           LACC    #lk [,shift2 ]
                    where shift1 <= 16
                    and shift2 <= 15
LACL        Load Accumulator and Clear High Accumulator
            Direct: [label]         LACL dma
            Indirect:       [label]         LACL  {ind} [ ,next ARP]
            Immediate:[label]       LACL #k
LACT        Load Accumulator With Shift Specified by TREG1
            Direct: [label]         LACT dma
            Indirect:       [label]         LACT  {ind} [ next ARP]
NEG         Negate Accumulator
            [label]         NEG
NORM        Normalize Contents of Accumulator
            [label]         NORM        {ind}
OR          OR With Accumulator
            Direct: [label]                 OR      dma
            Indirect:       [label]                 OR      {ind} [, next ARP]
            Long Immediate:[label]                  OR      #lk [,shift ]
ROL         Rotate Accumulator Left
            [label]         ROL
ROR         Rotate Accumulator Right
            [label]         ROR
SACH        Store High Accumulator With Shift
            Direct: [label]         SACH dma
```

Indirect: [label] SACH {ind} [, next ARP]

SACL Store Low Accumulator With Shift

Direct: [label] SACL dma

Indirect: [label] SACL {ind} [, next ARP]

SFL Shift Accumulator Left

[label] SFL

SFR Shift Accumulator Right

[label] SFR

SUB Subtract from Accumulator with Shift

Direct: [label] SUB dma [,shift1]

Indirect: [label] SUB {ind} [,shift1 [, next ARP]]

Short Immediate:[label] SUB #k

Long Immediate:[label] SUB #lk [,shift2]

SUBB Subtract from Accumulator with Borrow

Direct: [label] SUBB dma

Indirect: [label] SUBB {ind} [, next ARP]

SUBC Conditional Subtract

Direct: [label] SUBC dma

Indirect: [label] SUBC {ind} [, next ARP]

SUBS Subtract from Low Accumulator with Sign-Extension Suppressed

Direct: [label] SUBS dma

Indirect: [label] SUBS {ind} [, next ARP]

SUBT Subtract from Accumulator with Shift Specified by TREG1

Direct: [label] SUBT dma

Indirect: [label] SUBT {ind} [, next ARP]

XOR Exclusive-OR with Accumulator

Direct: [label] XOR dma

Indirect: [label] XOR {ind} [, next ARP]

Long Immediate:[label] XOR #lk [,shift]

ZALR Zero Low Accumulator and Load High Accumulator with Rounding

Direct: [label] ZALR dma

Indirect: [label] ZALR {ind} [, next ARP]

INSTRUCȚIUNI REFERITOARE LA REGISTRELE AUXILIARE ŞI LA POINTERUL DE PAGINĂ

ADRK CMPR LAR LDP MAR SAR SBRK

ADRK Add to Auxiliary Register Short Immediate

[label] ADRK constant

CMPR Compare Auxiliary Register with Auxiliary Register ARCR

[label] CMPR constant

LAR Load Auxiliary Register

Direct: [label] LAR AR, dma

Indirect: [label] LAR AR, {ind} [, next ARP]

Short Immediate:[label] LAR AR, #k

Indirect: [label] LAR AR, #lk

LDP Load Data Memory Page Pointer

Direct: [label] LDP dma
Indirect: [label] LDP {ind} [, next ARP]
Short Immediate:[label] LDP #k

MAR Modify Auxiliary Register
Direct: [label] MAR dma
Indirect: [label] MAR {ind} [, next ARP]

SAR Store Auxiliary Register
Direct: [label] SAR AR, dma
Indirect: [label] SAR AR, {ind} [, next ARP]

SBRK Subtract From Auxiliary Register Short Immediate
[label] SBRK #k

INSTRUCȚIUNI DE ÎNMULȚIRE ŞI REFERITOARE LA REGIŞTRII T ŞI P

APAC LPH LT LTA LTD LTP LTS MAC MACD MPY MPYA MPY MPYU PAC SPAC SPH SPL SPLK SPM SQRA SQRS

APAC Add P Register to Accumulator
[label] APAC

LPH Load Product High Register
Direct: [label] LPH dma
Indirect: [label] LPH {ind} [, next ARP]

LT Load TREG0
Direct: [label] LT dma
Indirect: [label] LT {ind} [, next ARP]

LTA Load TREG0 and Accumulate Previous Product
Direct: [label] LTA dma
Indirect: [label] LTA {ind} [, next ARP]

LTD Load TREG0, Accumulate Previous Product, and Move Data
Direct: [label] LTD dma
Indirect: [label] LTD {ind} [, next ARP]

LTP Load T Register and Store P Register in Accumulator
Direct: [label] LTP dma
Indirect: [label] LTP {ind} [, next ARP]

LTS Load TREG0 and Subtract Previous Product
Direct: [label] LTS dma
Indirect: [label] LTS {ind} [, next ARP]

MAC Multiply and Accumulate
Direct: [label] MAC pma, dma
Indirect: [label] MAC pma, {ind} [, next ARP]

MACD Multiply and Accumulate With Data Move
Direct: [label] MACD pma, dma
Indirect: [label] MACD pma, {ind} [, next ARP]

MPY Multiply
Direct: [label] MPY dma
Indirect: [label] MPY {ind} [, next ARP]
Short Immediate:[label] MPY #k
Long Immediate:[label] MPY #lk

MPYA Multiply and Accumulate Previous Product
Direct: [label] MPYA dma
Indirect: [label] MPYA {ind} [, next ARP]

MPYS Multiply and Subtract Previous Product
Direct: [label] MPYS dma
Indirect: [label] MPYS {ind} [, next ARP]

MPYU Multiply Unsigned
Direct: [label] MPYU dma
Indirect: [label] MPYU {ind} [, next ARP]

PAC Load Accumulator with P Register
[label] PAC

SPAC Subtract P Register from Accumulator
[label] SPAC

SPH Store High P Register
Direct: [label] SPH dma
Indirect: [label] SPH {ind} [, next ARP]

SPL Store Low P Register
Direct: [label] SPL dma
Indirect: [label] SPL {ind} [, next ARP]

SPLK Store Parallel Long Immediate
Direct: [label] SPLK #lk,dma
Indirect: [label] SPLK #lk,{ind} [, next ARP]

SPM Set P Register Output Shift Mode
[label] SPM constant

SQRA Square and Accumulate Previous Product
Direct: [label] SQRA dma
Indirect: [label] SQRA {ind} [, next ARP]

SQRS Square and Subtract Previous Product
Direct: [label] SQRS dma
Indirect: [label] SQRS {ind} [, next ARP]

IN STRUCȚIUNI DE SALT ŞI APEL SUBPROGRAME

B BACC BANZ BCND CALA CALL CC RET RETC TRAP

B[D] Branch Unconditionally
[label] B[D] pma [,{ind} [, next ARP]]

BACC[D] Branch to Address Specified by Accumulator
[label] BACC[D]

BANZ[D] Branch on Auxiliary Register Not Zero
[label] BANZ[D] pma [,{ind} [, next ARP]]

BCND[D] Branch Conditionally
[label] BCND[D] pma [, cond1] [,cond2] [,..]
Operands: 0 <= pma <= 65535
Conditions: ACC=0 EQ
ACC != 0 NEQ
ACC <0 LT
ACC<= 0 LEQ

		ACC > 0	GT
		ACC >=0	GEQ
		C=0	NC
		C=1	C
		OV=0	NOV
		OV=1	OV
		-BIO low	BIO
		TC=0	NTC
		TC=1	TC
	Unconditionally	UNC	

CALA[D] Call Subroutine Indirect

[label] CALA[D]

CALL[D] Call Subroutine

[label] CALL[D] pma [,{ind} [, next ARP]]

CC[D] Call Conditionally

[label] CC[D] pma [, cond1] [,cond2] [,..]

Operands: 0 <= pma <= 65535

Conditions:	ACC=0		EQ
	ACC != 0	NEQ	
	ACC <0		LT
	ACC<= 0	LEQ	
	ACC > 0	GT	
	ACC >=0	GEQ	
	C=0	NC	
	C=1	C	
	OV=0	NOV	
	OV=1	OV	
	-BIO low	BIO	
	TC=0	NTC	
	TC=1	TC	
Unconditionally	UNC		

RET[D] Return From Subroutine

[label] RET[D]

RETC[D] Return From Subroutine Conditionally

[label] RETC[D] [, cond1] [,cond2] [,..]

Conditions:	ACC=0		EQ
	ACC != 0	NEQ	
	ACC <0		LT
	ACC<= 0	LEQ	
	ACC > 0	GT	
	ACC >=0	GEQ	
	C=0	NC	
	C=1	C	
	OV=0	NOV	
	OV=1	OV	
	-BIO low	BIO	
	TC=0	NTC	
	TC=1	TC	

Unconditionally UNC

TRAP Software Interrupt

[label] TRAP

OPERAȚII I/O ŞI CU MEMORIA DE DATE

APL BLDD BLPD CLRC DMOV IN OPL OUT SETC TBLR TBLW

APL AND Data Memory Value with DBMR or Long Constant

Direct: [label] APL [#lk,] dma

Indirect: [label] APL [#lk,] {ind} [, next ARP]

BLDD Block Move From Data Memory to Data Memory

General Syntax: [label] BLDD src, dst

All valid cases have the gereral syntax:

Direct K/DMA: [label] BLDD #addr, dma

Indirect K/DMA: [label] BLDD #addr, {ind} [, next ARP]

Direct DMA/K: [label] BLDD dma,#addr

Indirect DMA/K: [label] BLDD {ind} , #addr[, next ARP]

Direct BMAR/DMA: [label] BLDD BMAR, dma

Indirect BMAR/DMA: [label] BLDD BMAR, {ind} [, next ARP]

Direct DMA/BMAR: [label] BLDD dma, BMAR

Indirect DMA/BMAR:[label] BLDD {ind}, BMAR [, next ARP]

BLPD Block Move From Program Memory to Data Memory

General Syntax: [label] BLPD src, dst

All valid cases have the general syntax:

Direct K/DMA: [label] BLPD #pma, dma

Indirect K/DMA: [label] BLPD #pma, {ind} [, next ARP]

Direct BMAR/DMA: [label] BLPD BMAR, pma

Indirect BMAR/DMA:[label] BLPD BMAR, {ind} [, next ARP]

CLRC Clear Control Bit

[label] CLR Ccontrol Bit

Operands: STO, ST1 bit (from: { C, CNF, HM, INTM, OVM, TC, SXM, XF})

DMOV Data Move in Data Memory

Direct: [label] DMOV dma

Indirect: [label] DMOV {ind} [, next ARP]

IN Input Data From Port

Direct: [label] IN dma, PA

Indirect: [label] IN {ind}, PA [, next ARP]

OPL OR With DBMR or Long Immediate

Direct: [label] OPL [.#lk,] dma

Indirect: [label] OPL [.#lk,] {ind} [, next ARP]

OUT Output Data to Port

Direct: [label] OUT dma, PA

Indirect: [label] OUT {ind}, PA [, next ARP]

SETC Set Control Bit

[label] SETC control bit

control bit: ST0 or ST1 bit (from:{C, CNF, HM, INTM, OVM, SXM, TC, XF})

TBLR Table Read
Direct: [label] TBLR dma
Indirect: [label] TBLR {ind} [, next ARP]

TBLW Table Write
Direct: [label] TBLW dma
Indirect: [label] TBLW {ind} [, next ARP]

INSTRUCȚIUNI DE CONTROL

BIT BITT CLRC IDLE INTR LST NMI NOP POP POPD PSHD PUSH RPT SETC SST

BIT Test Bit
Direct: [label] BIT dma, bit code
Indirect: [label] BIT {ind}, bit code [, next ARP]

BITT Test Bit Specified by TREG2
Direct: [label] BITT dma
Indirect: [label] BITT {ind} [, next ARP]

CLRC Clear Control Bit
[label] CLRC control bit
Operands: STO, ST1 bit (from: { C, CNF, HM, INTM, OVM, TC, SXM, XF})

IDLE Idle Until Interrupt
[label] IDLE

INTR Soft Interrupt
[label] INTR k

LST Load Status Register
Direct: [label] LST #n, dma
Indirect: [label] LST #n, {ind} [, next ARP]

NMI Nonmaskable Interrupt
[label] NMI

NOP No Operation
[label] NOP

POP Pop Top of Stack to Low Accumulator
[label] POP

POPD Pop Top of Stack to Data Memory
Direct: [label] POPD dma
Indirect: [label] POPD {ind} [, next ARP]

PSHD Push Data Memory Value Onto Stack
Direct: [label] PSHD dma
Indirect: [label] PSHD {ind} [, next ARP]

PUSH Push Low Accumulator Onto Stack
[label] PUSH

RPT Repeat Instructions as Specified by Data Memory Value
Direct: [label] RPT dma
Indirect: [label] RPT {ind} [, next ARP]
Short Immediate: [label] RPT #k

Long Immediate: [label] RPT #lk

SETC Set Control Bit

[label] SETC control bit

control bit: ST0 or ST1 bit (from:{C, CNF, HM, INTM, OVM, SXM, TC, XF})

SST Store Status Register

Direct: [label] SST #n, dma

Indirect: [label] SST #n, {ind} [, next ARP]

NOTĂ:

În adresarea directă un cuvânt de instrucțiune conține cei mai puțin semnificativi 7 biți ai adresei de memorie de date. Acest câmp este concatenat cu cei nouă biți conținuți de registrul pointerului la pagina memoriei de date (DP) obținându-se 16 biți ai adresei memoriei de date. Rezultă că în modul de adresare directă, memoria de date este paginată, conținând în total 512 pagini, fiecare pagină conținând 128 de cuvinte de 16 biți. Registrul DP poate fi modificat cu instrucțiunile LST şi LDP.

Adresarea indirectă permite accesarea memoriei prin intermediul registrelor auxiliare. În acest mod de adresare, adresa operandului instrucțiunii este conținută de registrul auxiliar selectat. Există opt regiştrii auxiliari (AR0 – AR7) care permit o adresare indirectă flexibilă şi puternică. Pentru a selecta un anumit registru auxiliar, registrul pointer ARP este încărcat cu o valoare de la zero la şapte pentru indicarea registrului AR0, respectiv AR7.

3.2. Turbo-Asamblorul (TASM)

Introducere

TASM este un cross-asamblor pentru mediul MS-DOS. Realizează asamblarea codului sursă scris într-un dialect adecvat (în general foarte apropiat de limbajul de asamblare al producătorului). Codul obiect rezultat poate fi transferat microprocesorului prin intermediul memoriei PROM sau prin alte metode.

Apelare

TASM poate fi apelat astfel (câmpurile opționale sunt puse între paranteze pătrate iar câmpurile simbolice cu italice):

tasm [-*opțiune*] *fişier_src* [*fişier_obiect* [*fişier_lst*[*fişier_exp*[*fişier_sym*]]]]

unde opțiunea poate fi:

- table -> opțiune care specifică versiunea tabelului de instrucțiuni folosită
- ttable -> tabel (alternativă la opțiunea de mai sus)
- aamask -> controlul asamblării (opțiuni pentru verificarea erorilor)
- c -> fişierul obiect va fi scris ca un bloc continuu
- dmacro -> defineşte un macro (sau numai un nume de macro)
- e -> afişează liniile sursă după expandarea macro
- ffillbyte -> umple întregul spațiu de memorie cu fillbyte (valoare hexa)
- i -> ignoră literele mari în simboluri

- k -> generează fişiere obiect tip DSK
- l[al] -> crează tabelul etichetelor în listing
- p[lines] -> paginează fişierul listing (numărul de linii pe pagină implicit = 60)
- q -> dezactivează fişierul listing
- rkb -> setează dimensiunea buffer-ului de citire în kocteți (implicit 2 kocteți)
- s -> scrie un fişier cu tabela simbolurilor
- y -> timpul de asamblare

paramerii fişier sunt:

- fişier_src -> numele fişierului sursă
- fişier_obj -> numele fişierului obiect
- fişier_lst -> numele fişierului listing
- fişier_exp -> numele fişierului de export (numai dacă este utilizată directiva EXPORT)
- fişier_sym -> numele fişierului tabelei de simboluri (numai dacă opțiunea "-s" a fost utilizată sau s-au utilizat directivele SYM/AVSYM)

Numele fişierului sursă trebuie specificat obligatoriu. Dacă nu se specifică numele fişierului sursă atunci se afişează un help sumar.

Implicit numele pentru celelalte fişiere (dacă ele nu sunt specificate) sunt generate din numele fişierului sursă la care se adaugă extensia corespunzătoare. Extensia folosită în acest caz este:

Extensia	Tip fişier
.OBJ	Fişier obiect
.LST	Fişier listing
.EXP	Fişier export simboluri
.SYM	Fişier tabelă de simboluri

TASM nu are o tabelă internă a setului de instrucțiuni pentru asamblare. Definirea instrucțiunilor se face prin citirea unui fişier la rularea TASM. TASM determină care este tabelul de instrucțiuni care va fi folosit din câmpul opțiunii "-table", care conține un număr zecimal de trei cifre, prezentat mai jos.

De exemplu pentru a asambla codul din fişierul sursă numit "source.asm", trebuie să introducem comanda:

tasm –203 source.asm (pentru limbaj de asamblare TMS320C2xx)

Numele fişierului care conține tabela de instrucțiuni pentru exemplul de mai sus va fi: "TASM203.TAB", deci în afară de numărul 203 prezent în opțiune, numele fişierului este format prin adăugarea în fața numărului "TASM" şi extensia ".TAB".

Este posibil să proiectăm tabele ale căror nume să conțină litere şi nume. De exemplu apelarea fişierului cu tabelul instrucțiunilor numit "TASMF206.TAB" se face cu comanda:

tasm –tf206 source.asm

Fiecare opțiune trebuie precedată de liniuță (semnul minus). Numele fişierelor nu pot fi scrise în fața opțiunilor.

Descrierea opţiunilor:

a – controlul asamblării – TASM poate furniza o verificare suplimentară a erorii. Dacă se specifică "-a" fără nici o cifră după aceea, atunci toate metodele de verificare sunt folosite. Dacă se specifică o cifră atunci se foloseşte o mască pentru a determina care verificare a erorilor se va face. Biţii măştii sunt definiţi astfel:

Bit	Opţiune	Descriere
0	-a1	Verifică utilizarea indirectărilor aparent ilegale
1	-a2	Verifică datele nefolosite în argumente
2	-a4	Verifică simbolurile multiple
3	-a8	Verifică operatorii non-unari la începutul expresiei

Se pot folosi şi combinaţii ale biţilor de mai sus. De exemplu "-a5" va valida verificarea indirectărilor ilegale şi a simbolurilor multiple.

Indirectarea ilegală se aplică microprocesoarelor care folosesc parantezele în jurul unui argument pentru a indica indirectarea. Chiar dacă este legal să punem încă un rând de paranteze în jurul expresiei, TASM nu va accepta acest lucru dacă nu este specificat clar în tabelul instrucţiunilor şi dacă verificarea respectivă este validată.

Datele neutilizate dintr-un argumentse aplică cazurilor când este nevoie în argument de un singur octet dar argumentul conţine mai mulţi octeţi. O adresă de 16 biţi utilizată în adresarea imediată necesită un singur octet. Dacă sunt folosiţi mai mulţi este generat un mesaj de eroare.

Pentru ca aceste verificări să se facă ori de câte ori se lansează TASM, se adaugă în AUTOEXEC.BAT linia:

SET TASMOPTS = -a

c – scrierea într-un bloc continuu – dacă această opţiune este specificată, atunci toţi octeţii de la primul la ultimul, din fişierul codului obiect, vor fi definiţi. În mod normal dacă contorul de program (PC) sare mai departe pentru că s-a întâlnit o directivă .ORG, octeţii săriţi nu vor avea nici o valoare atribuită (sunt într-o stare necunoscută) în fişierul obiect. Cu această opţiune activată nu se scrie nimic în fişierul obiect nimic până la sfârşitul asamblării iar atunci se scrie întregul bloc. Această opţiune este folositoare atunci când generăm cod care va fi pus în PROM şi toţi octeţii trebuie să aibă valori cunoscute. Această opţiune se foloseşte în conjuncţie cu opţiunea "-f" pentru a ne asigura că toţi octeţii neutilizaţi vor avea o valoare cunoscută.

d – definirea unui macro – macrourile sunt definite în liniile de comandă ale fişierului sursă pentru a asambla diferitele linii cu directiva IFDEF. Utilizarea opţiunii este o cale convenabilă pentru generarea diferitelor versiuni ale codului obiect dintr-un singur fişier sursă.

e – expandarea sursei - în mod normal TASM afişează numai liniile din fişierul sursă. Dacă se folosesc macrodefiniţii (cu directive DEFINE), pentru a vedea liniile acestora în listing se foloseşte această opţiune.

f – umplerea memoriei – Tasm foloseşte o imagine a memoriei de 64 kocteţi chiar şi atunci când procesorul nu poate folosi atâta. Folosind opţiunea "-fxx" atunci această imagine din memorie este umplută cu xx_H. Sunt necesare aproximativ 2 secunde pentru iniţializarea memoriei.

i – ignoră literele mari din simboluri - în mod normal TASM face deosebire între literele mari şi mici. Dacă nu dorim acest lucru se foloseşte opţiunea "-i".

k – generează format obiect DSK – formatul obiect destinat utilizării cu aplicația Pathway 2xx DSK.

l – tabelul de simboluri – generează un tabel al simbolurilor în fişierul listing. Simbolurile din macro nu sunt afişate.

Două sufixe pot fi utilizate opțional cu opțiunea "-l".

Sufix	Descriere
l	utilizează forma lungă a listingului
a	afişează toate simbolurile (inclusiv cele locale)

Sufixul se foloseşte imediat după opțiune.

Exemple:

-l -> afişează simbolurile nelocale în forma scurtă;
-la -> afişează toate simbolurile în forma scurtă;
-ll -> afişează simbolurile nelocale în forma lungă;
-lal -> afişează toate simbolurile în forma lungă.

p – paginarea fişierului listing – această opțiune determină ca fişierul listing să aibă un antet şi un subsol după fiecare grup de 60 de linii. Dacă dorim alt număr de linii pe pagină atunci acest lucru se dă explicit.

Exemplu:

TASM –203 –p56 source.asm

q – dezactivează fişierul listing – această opțiune suprimă fişierul listing chiar dacă s-a întâlnit o directivă .LIST.

r – setează dimensiunea bufferului de citire – această opțiune modifică dimensiunea implicită (2 kocteți) a bufferului de citire.

După ***r*** urmează o cifră hexazecimală care dă dimensiunea bufferului (exemplu: "-r8" indică un buffer de 8 kocteți iar "-rf" indică un buffer de 15 kocteți). Trebuie notat că bufferul de citire ocupă aceeaşi zonă de memorie ca simbolurile şi macro. De obicei creşterea bufferului de citire este necesară dacă sunt utilizate directive INCLUDE. Dimensiunea de 8 kocteți de buffer poate fi suficientă pentru cele mai multe asamblări dar programele cu multe simboluri pot să nu permită această valoare. De altfel reducând bufferul la 1 koctet se poate creşte memoria disponibilă pentru stocarea simbolurilor (dacă acest lucru este necesar).

s – validează generarea fişierului de simboluri – dacă această opțiune este setată atunci va fi generat un fişier de simboluri la sfârşitul asamblării. Formatul acestui fişier este: un simbol pe linie, fiecare simbol începând în prima coloană şi este urmat de un blanc şi patru valori hexa reprezentând valoarea simbolului.

Exemplu:

```
label1    FFFE
label2    FFFF
label3    1000
```

Fişierul de simboluri poate fi generat şi de directiva SYM.

t – numele tabelei – variantă alternativă pentru a specifica tabela de instrucțiuni. Această opțiune este folositoare când tabelul începe cu un caracter nezecimal. De exemplu tabelul F8 poate fi selectat astfel:

TASM –tf8 source.asm

Se va citi tabelul de instrucțiuni din fişierul:

TASMF8.TAB

y – validează măsurarea timpului de asamblare – dacă opțiunea este validată se va genera timpul de asamblare şi numărul de linii asamblate/secundă la sfârşitul asamblării.

Variabilele de mediu

Mediul TASM poate fi personalizat utilizând următoarele variabile de mediu:

TASMTABS – specifică calea de căutare pentru tabelele cu instrucțiuni destinate TASM

Exemplu:

SET TASMTABS = C:\TASM

dacă tabela de instrucțiuni se găseşte în directorul TASM

TASMOPTS – opțiunile ce se vor folosi la execuția TASM

Exemplu:

SET TASMOPTS = -203 –k

Codurile de ieşire

Cod ieşire	Semnificație
0	Terminare normală, fără erori de asamblare
1	Terminare normală, cu erori de asamblare
2	Terminare anormală, memorie insuficientă
3	Terminare anormală, eroare la acces fişier
4	Terminare anormală, eroare generală

Codul de ieşire 2 este însoțit de mesajele de eroare la consolă.

Formatul fişierului sursă

Structura generală:

etichetă operație operand comentariu

toate câmpurile sunt opționale. Câmpurile sunt separate de unul sau mai multe spații sau TAB. Lungimea maximă a liniei are 255 caractere.

Câmpul etichetă - dacă primul caracter al liniei este alfabetic atunci se consideră începutul etichetei. Caracterele care urmează sunt considerate ca aparținând etichetei cu excepția caracterului spațiu, TAB sau ":" când se consideră că este sfârşitul etichetei. Lungimea maximă a etichetei este de 32 de caractere. În mod normal etichetele diferă între ele dacă se folosesc caractere mari şi mici (cu excepția cazului când se foloseşte opțiunea "-i").

Câmpul operației – conține un mnemonic. Poate începe în orice coloană cu excepția primei coloane. Nu are importanță dacă se folosesc litere mari sau mici.

Câmpul operandului – poate include expresii şi/sau simboluri speciale ce descriu modul de adresare utilizat.

Câmpul comentariu - începe cu caracterul ";" restul caracterelor după acesta fiind ignorat de TASM

Linii cu mai multe comenzi – mai multe instrucțiuni pot fi scrise pe o linie separate cu "\" (backslash). Prima coloană după "\" este considerată coloana 1 a noii instrucțiuni şi deci aici va fi eticheta (dacă există). Acest mod de scriere este folositor la construcția macro.

Expresii

Expresiile pot fi construite cu mai multe elemente:

1. simboluri
2. constante
3. simbolul contorului de locații
4. operatori
5. paranteze

Simboluri – reprezintă valori numerice. Simbolurile locale încep cu o literă sau cu prefixul implicit al simbolurilor locale "_". Valaorea simbolului este limitată la precizia de 32 de biți (32 caractere).
Constantele numerice - încep cu un număr. Cele hexa trebuie să înceapă cu "0" dacă prima cifră este o literă. Această condiție nu este necesară dacă se foloseşte în fața cifrei hexa simbolul "$".

Baza de numerație este stabilită de prefixul sau sufixul numărului.

Baza de numerație	Sufix	Prefix
2	B sau b	%
8	O sau o	@
10	D sau d	nimic
16	H sau h	$

Prefixele pot introduce ambiguități. Simbolurile "%" şi "$" au utilizări alternative: "%" pentru operația modulo şi "$" pentru simbolul contorului de locații. Ambiguitatea este rezolvată studiind contextul. Caracterul "%" este interpretat ca modulo numai dacă este în poziția necesară pentru un operator binar. La fel dacă după "$" este un caracter hexa valid atunci se consideră număr hexa, altfel se consideră contor de locații.
Constantele caracter – sunt caractere unice între ghilimele (ghilimelele de la sfârşit sunt opționale). Aceste constante reprezintă valoarea ASCII a caracterului. Caracterele netipăribile nu pot fi folosite.
Constantele şir – sunt constante formate din unul sau mai multe caractere între ghilimele. Constantele şir nu sunt permise în expresii. Ele pot fi folosite numai în directivele asamblor TITLE, BYTE şi TEXT.

Caracterele netipăribile permise aici sunt:

Caracter netipăribil	Descriere
\n	Linie nouă
\r	Retur de car
\b	Un caracter la stânga (backspace)
\t	TAB
\f	Formfeed
\\	Backslash
\"	Ghilimele
\ooo	Valoarea octală a caracterului de tipărit

Simbolul contorului de locații – valoarea curentă a contorului de program PC poate fi folosită în expresii utilizând simbolul "$". Se poate folosi orinde sunt permise şi constantele numerice. Dacă este urmat de o cifră hexa atunci simbolul se va considera constantă hexa. Se poate folosi şi simbolul "*" dar acesta nu este preferat din cauza ambiguității cu operatorul de înmulțire.

Operatorii – operatorii posibil de utilizat în expresii sunt:

Operator	Tip	Descriere
+	Aditiv	adunare
-		scădere
*	Multiplicativ	înmulțire
/		împărțire
%		modulo
<<		rotație logică stânga
>>		rotație logică dreapta
~	Unar	inversarea bitului (complement față de 1)
-		negație unară
=	Relațional	egal
==		egal
!=		diferit
<		mai mic
>		mai mare
<=		mai mic sau egal
>=		mai mare sau egal
&	Binar	ŞI binar
|		SAU binar
^		SAU EXCLUSIV binar

Sintaxa este similară cu cea din limbajul C cu următoarele observații:

1. Precedarea operatorului nu are efect. Evaluarea se face de la stânga la dreapta cu excepția grupării în paranteze.
2. Toate evaluările se fac pe 32 de biți cu semn.
3. Ambii operatori "=" şi "= =" se pot folosi pentru verificarea egalității.

Operatorii relaționali întorc valoarea "1" dacă relația este adevărată şi valoarea "0" dacă este falsă. Sunt folosiți 32 de biți cu semn.

Este bine să se indice ordinea operațiilor cu paranteze pentru a păstra portabilitatea din cauză că TASM nu evaluează operațiile ca alte asambloare.

Exemplu:

1+2*3+4

va fi evaluat de TASM astfel:

(((1+2)*3)+4)=13

regulile tipice de precedență impun evaluarea lui (2*3) mai întâi, astfel:

1+(2*3)+4=11

Pentru a fi siguri că se obține ordinea dorită de evaluare a operațiilor folosiți parantezele cât mai mult.

Exemple de expresii valide:

(0f800H+tab)
(label_2 >> 8)
(label_3 << 8) & $f000
$ + 4
010010000100100b +'a'
(base + ((label_4 >> 5) & (mask << 2))

Directivele asamblorului

Cele mai multe directive asamblor au un format similar cu instrucțiunile mașină. Există două tipuri de directive de asamblare – unele care se aseamănă cu funcțiile preprocesor din limbajul C şi altele care se aseamănă mai mult cu directivele tradiționale asamblor.

Directivele de tip preprocesor C sunt invocate cu "#" în primul caracter al liniei urmat de directivă (exact ca în limbajul C). Sunt acceptate atât caracterele mari cât şi cele mici.

ADDINSTR – poate fi utilizată pentru a defini o instrucțiune suplimentară pentru a fi folosită la asamblarea cu TASM. Formatul este:

[etichetă] .ADDINSTR inst args opcode nbytes modop class shift binar

câmpurile sunt separate cu spații exact cum trebuie să apară în fişierul de definiții a instrucțiunii.

LOCK – această directivă duce la avansarea contorului de instrucțiuni cu un număr specificat de octeți fără atribuirea vreunei valori locațiilor sărite.

Formatul este:

[etichetă] .BLOCK expr

Exemple:

```
word1  .BLOK    2
byte1  .BLOK    1
buffer .BLOK    80
```

BYTE – este folosită la atribuirea unei valori adresate de contorul de locații (locația curentă).

Formatul este:

[etichetă] .BYTE expr[,expr ...]

Numai octetul cel mai puțin semnificativ al expresiei este folosit.

Exemple:

```
label1 .BYTE   10010110B
       .byte   'a'
       .byte   0
       .byte   100010110b,'a',0
       .byte   "Hello",10,13,"World"
```

CHK – determină calculul unei sume de control care va fi depusă în locația curentă. Punctul de început al sumei de control este dat în argument.

Formatul este:

.CHK start_addr

Suma de control este calculată ca o sumă aritmetică simplă începând de la start_addr până la adresa (exclusiv) a directivei CHK. Cel mai puțin semnificatv octet este memorat.

CODES/NOCODES – se foloseşte pentru a comuta succesiv generarea codului în fişierul de ieşire. Cu NOCODES activat liniile sursă sunt trimise în listingul de ieşire fără a se genera cod. Este utilă pentru comentariile lungi.

DB – este o alternativă a directivei BYTE.

DW – este o alternativă a directivei WORD.

DEFINE – este una din cele mai puternice directive care permite substituția unor şiruri cu argumente opționale (macro). Formatul este următorul:

#DEFINE macro_label [(arg_list)] [macro_definition]

macro_label := şirul care va fi expandat când este găsit în fişierul sursă
arg_list := şir opțional cu variabile pentru substituția variabilelor din macro
macro_def := şirul care apare în locul macro_label în textul sursă

Exemplu:

```
#DEFINE      MLABEL
```

Notați că nu s-a specificat şirul de substituție. Scopul unei directive de acest fel este în mod tipic de a defini o variabilă în scopul controlului unor secvențe de asamblare condiționată (IFDEF sau IFNDEF).

Un alt exemplu:

```
# DEFINE VAR1_LO (VAR1 & 255)
```

Această instrucțiune va determina înlocuirea şirului "VAR1_LO" din programul sursă cu "(VAR1 & 255)"

Reguli asociate cu lista de argumente:

1. Utilizați maximum 10 argumente
2. Fiecare argument trebuie să aibă maximum 15 caractere.

De notat că aceste macro pot fi definite de asemenea în linia de coman TASM, utilizând opțiunea "-d".

DEFCONT – se utilizează pentru a adăuga linii la ultimul macro început cu o directivă DEFINE. Furnizează o metodă convenabilă de a defini macrouri lungi care depăşesc o linie.

Exemplu:

```
#DEFINE      ADD (xx,yy) clc
#DEFCONT     \lda   xx
#DEFCONT     \ldc   yy
#DEFCONT     \sta   xx
```

DS – această directivă se comportă similar cu directiva .ORG. Poate fi utilizată pentru a identifica adresa unde vor fi plasate datele în Spațiul de Date a lui TMS320C2xx.

EJECT – schimbarea paginii şi generarea unui header în fişierul de listare. Nu are efect dacă modul de paginare este dezactivat (PAGE/NOPAGE).

Formatul:

.EJECT

ELSE – se utilizează opțional cu IFDEF, IFNDEF şi IF pentru a desemna un bloc alternativ de asamblat în cazul în care blocul imediat după IFDEF, IFNDEF şi IF nu este asamblat.

Exemple:

```
#IFDEF label1
lda byte1
sta byte2
#ENDIF

#ifdef label1
lda byte1
#else
lda byte2
#endif

#ifndef label1
lda byte2
#else
lda byte1
#endif

#if ($>=100h)

;generează o instrucțiune invalidă pentru a produce o eroare atunci ;când depăşim
;granița de 4k octeți
#endif
```

END – această directivă trebuie să fie ultima în fişierul sursă. Forțează scrierea ultimei înregistrări în fişierul obiect.

Format:

[label] .END

ENDIF – această directivă trebuie să urmeze întotdeauna după o directivă IFDEF, IFNDEF sau IF şi semnifică sfârşitul blocului condițional.

ENTRY – este utilizată pentru identificarea punctului de intrare (punct de start) în Spațiul de Program a programului pentru TMS320C2xx.

Exemplu:

```
        .ps 8000h     ;poziționare PC
        .entry        ;defineşte punctul de intrare în program
start:  nop           ;această instrucțiune va fi punctul de intrare
```

EQU – este folosităpentru a atribui o valoare unei variabile. În această situație variabilele pot fi folosite în expresii drept constante literale.

Format:

```
etichetă .EQU expr
```

Exemplu:

```
MASK        .EQU  0F0H
;
lda     IN_BYTE
and MASK
sta OUT_BYTE
```

O formă alternativă a lui EQU este “=”. Exemplul anterior este echivalent cu:

MASK = $F0

sau

MASK =0F0H

sau

MASK = $F0

Este necesar un spațiu după denumire dar nu neapărat după "=".

EXPORT – este folosită pentru a scrie simbolurile într-un fişier de ieşire. Numele fişierului de ieşire este dat de opțiunea "-s". Simbolurile sunt scrise ca egalități (utilizând directive .EQU) fişierul rezultat putând fi inclus într-o asamblare subsecvențială. Această facilitate poate ajuta la eliminarea unor deficiențe ale TASM date de linkeditor.

Format:

[etichetă] .EXPORT etichetă

Exemplu:

Fişierul sursă:

```
.EXPORT read_byte
.EXPORT write_byte
.EXPORT open_file
```

Fişierul rezultat:

```
read_byte     .EQU  $1243
write_byte    .EQU  $12AF
open_file     .EQU  $1301
```

IFDEF – folosită pentru asamblarea opțională a unui bloc de instrucțiuni.

Forma:

#IFDEF macro_label

Când este apelată, lista de macro_labels (stabilite pe baza directivelor DEFINE) sunt căutate. Dacă eticheta este găsită, în fişierul de intrare sunt sărite liniile de după IFDEF până este întâlnită o directivă ENDIF sau ELSE.

Liniile sărite apar totuşi în fişierul listing dar semnul "Î" ca apărea imediat după PC curent şi nu se generează cod obiect (acest lucru este aplicabil şi la directivele IFDEF, IFNDEF şi IF).

IFNDEF – este opusa directivei IFDEF. Blocul de instrucțiuni următor directivei este asamblat numai dacă macro_label nu este definită.

Forma:

 # IFNDEF macro_label

Când este apelată, lista de macro_labels (stabilite pe baza directivelor DEFINE) sunt căutate. Dacă eticheta nu este găsită, în fişierul de intrare sunt asamblate liniile de după IFNDEF până este întâlnită o directivă ENDIF sau ELSE.

IF – este utilizată pentru asamblarea opțională a unui bloc de instrucțiuni în funcție de valoarea dată de expresie.

Format:

#IF expr

Dacă expresia evaluată este diferită de zero blocul următor directivei IF este asamblat (până se întâlneşte o directivă ENDIF sau ELSE).

INCLUDE – citeşte şi asamblează fişierul sursă indicat. Directiva poate avea până la şase nivele. Permite o cale convenabilă de păstrare a definiţiilor comune, declaraţiilor sau subprogramelor.

Format:

#INCLUDE nume_fişier

Numele fişierului trebuie inclus între ghilimele duble.

Exemple:

```
#INCLUDE "macros.h"
#include "equates"
#include "subs.asm"
```

LIST/NOLIST – aceste directive pot fi folosite alternativ pentru a lista sau a suprima listarea în fişierul listing.

Format:

```
.LIST
.NOLIST
```

ORG – setează contorul de instrucţiuni (contorul de program PC) la valoarea dorită.

Format:

[etichetă] .ORG expr

Exemplu:

Pentru a genera cod începând cu adresa 1000H:

```
start    .ORG 1000H
```

Expresia poate conţine referiri la valoarea curentă a pointerului de instrucţiuni permiţând diferite manipulări de date. De exemplu pentru a alinia pointerul de instrucţiuni peste 256 de octeţi se poate folosi:

```
ORG (($+0FFH) & 0FF00H)
```

Directiva ORG poate fi folosită de asemenea pentru a rezerva spaţiu fără a desemna valoarea:

```
.ORG $+8
```

O formă alternativă a directivei ORG este "*=" sau "$=". Exemplul anterior este echivalent cu:

```
* = * + 8
$ = $ + 8
```

PAGE/NOPAGE – este folosită pentru listarea în mod pagină sau continuu.

Format:

.PAGE

.NOPAGE

PS – se comportă ca şi directiva .ORG. Se foloseşte pentru poziţionarea contorului de program în Spaţiul Program a lui TMS320C2xx.

SET – permite schimbarea valorii unei variabile existente.

Format:

variabilă .SET expresie

Utilizarea directivei SET trebuie evitată pentru că ea poate duce la erori de fază între pasu 1 şi pasul 2 a asamblării.

SYM – directiva poate fi utilizată pentru a genera un fişier cu tabela simbolurilor.

Formatul:

.SYM [fisier_simboluri]

Exemplu:

.SYM "symbol.map"

.SYM

Formatul fişierului SYM este de un simbol pe linie, fiecare simbol începe în prima coloană este urmat de un spaţiu şi apoi patru cifre hexa reprezentând valoarea simbolului.

Exemplu de format:

```
label1  FFFE
label2  FFFF
label3  1000
```

TEXT – permite folosirea unui şir ASCII căruia i se va aloca adresa curentă a pointerului de instrucţiuni.

Formatul este:

[etichetă] .TEXT "şir"

Valoarea ASCII a fiecărui caracter din şir este atribuită locaţiei următoare în mod succesiv. Sunt admise secvenţe speciale:

Caracter netipăribil	Descriere
\n	Linie nouă
\r	Retur de car
\b	Un caracter la stânga (backspace)
\t	TAB
\f	Formfeed
\\	Backslash
\"	Ghilimele
\ooo	Valoarea octală a caracterului de tipărit

Exemple:

```
message1    .TEXT     "Disk I/O error"
message2    .text     "Enter file name"
            .text     "abcdefg\n\r"
            .text     "I said \"NO\""
```

TITLE – se foloseşte pentru definirea unui titlu de către utilizator care va apărea la începutul fiecărei pagini (dacă este activ PAGE).

Formatul:

.TITLE "string"

Şirul nu trebuie să depăşească 80 de caractere.

Exemple:

```
.TITLE      "Controller version 1.1"
.title      "This is the title of the assembly"
.title      ""
```

WORD – permite atribuirea unei valori următoarelor două locaţii începând de la valoarea curentă a pinterului de instrucţiuni.

Formatul:

[etichetă] .WORD expr

Cel mai puţin semnificativ octet este pus primul şi după aceea cel mai semnificativ (cu excepţia cazului când se foloseşte directive MSFIRST).

Exemple:

```
data_table  .WORD     (data_table+1)
            .word     $1234
```

```
.Word      (('x'-'a') << 2)
.Word      12,55,32
```

Formatul fişierului obiect

TASM poate genera fişiere obiect care pot fi încărcate în Pathway 2xx DSK.

Acest format este orientat pe linii şi utilizează numai caractere ASCII tipăribile cu excepţia returului de car de la sfârşitul fiecărei linii. Sunt trei tipuri diferite de linii pentru formatul DSK.

Primul tip de linie în format DSK conţine numai informaţiile header inclusiv numele fişierului. De exemplu:

K_D203_1.01_xf.dsk

Următoarea linie conţine informaţii despre punctul de intrare pentru program şi semnalează acest lucru începând linia cu caracterul "1". De exemplu:

FE078FE0F

În acest format pentru punctul de intrare, informaţiile sunt următoarele: "1" începe linia şi indică faptul că acestă linie conţine informaţii legate de punctul de intrare. Următoarele patru caractere, în acest caz 8FE0, indică adresa hexa a punctului de intrare a programului. Caracterul "7" care urmează după adresă este un separator şi indică faptul că urmează suma de control pe patru caractere. În sfârşit caracterul "F" indică faptul că linia s-a terminat şi urmează retur de car şi linie nouă.

Ultimul tip de linie în format obiect DSK este linia care conţine programul/datele rezultate în urma asamblării. Formatul este prezentat în continuare.

Fiecare linie începe cu caracterul "9". Următoarele patru caractere reprezintă adresa hexa unde sunt plasate codul/datele. Urmează un caracter cu rol de separator. Dacă acest caracter este "M" acest lucru arată că următoarele patru caractere hexa sunt date care vor fi încărcate în Spaţiul de Date a dispozitivului 2xx. Dacă separatorul este "B" acest lucru arată că următoarele patru caractere hexa trebuie încărcate în Spaţiul de Program a dispozitivului 2xx. Separatorul se repetă până se întâlneşte caracterul "7" în locul separatorului. După caracterul "7" ca separator urmează o sumă de control hexa de patru caractere. În sfârşit caracterul "F" indică faptul că urmează retur de car şi linie nouă. Iată două exemple:

90300MBABEMDEADMB00B74976F

98FE0BBC04BBF0AB0000BBF0B7BE89F

Caracterul folosit pentru a indica sfârşitul fişierului de cod obiect este ":".

Mesajele de eroare

Mesajul de eroare	Descriere
Binary operator where value expected	S-au întâlnit doi operatori binari unul după celălalt fără o valoare între ei (lipseşte valoarea)
Cannot malloc for label storage	Memorie insuficientă pentru stocarea simbolurilor (vezi LIMIT|RI)
Duplicate label	Verificarea simbolurilor multiple validată prin opţiunea "-a"

Filname too short	Numele fişierului din linia de comandă are mai puţin de trei caractere. Această limitare este impusă pentru a nu confunda o opţiune cu numele unui fişier.
Heap overflow on label definition	TASM nu poate aloca memorie pentru a stoca variabile
Invalid operand	Nu există indirectare pentru această instrucţiune. Primul caracter a unui operand este o paranteză stângă pentru instrucţiunile care nu specifică explicit acest format. Unele micro utilizează parantezele pentru a semnala indirectarea dar punerea unei perechi de paranteze la o expresie este un lucru valabil (cu atât mai mult cu cât interesează evaluarea expresiei). Testul în acest caz este dat numai dacă opţiunea "-a4" este selectată (vezi secţiunea CONTROLUL ASAMBL|RII)
Invalid token where value expected	Doi operatori binari unul după celălalt nu sunt permişi
Label too long	Etichetele sunt limitate la 31 de caractere
Label value misaligned	Valoarea simbolului pare a avea o valoare diferită în cel de-al doilea pas faţă de cea calculată în primul pas. Acest lucru este dat în general de modul de adresare în pagina zero la versiunea TASM 6502. Simbolurile care sunt utilizate ca operanzi în instrucţiuni nu pot fi utilizate pentru modul de adresare în pagina zero. Modul de adresare în pagina zero trebuie întotdeauna definit înainte de a fi utilizat ca operand.
Label not found	Un simbol utilizat într-o expresie nu este găsit în tabela de simboluri
Label must pre-exist for SET	Directiva SET nu poate fi aplicată decât unui simbol existent
Label table overflow	S-au întâlnit prea multe simboluri
List file open error	TASM nu poate deschide fişierul specificat
Macro expansion too long	Expresia macro rezultată într-o linie depăşeşte lungimea maximă
Maximum number of macros exceeded	S-au întâlnit prea multe directive DEFINE
No END directive before EOF	Fişierul sursă nu conţine directive END. Nu e critic dar s-ar putea ca în fişierul obiect ultima înregistrare să se piardă
No files specified	TASM a fost apelat fără specificarea fişierului sursă

No such label yet defined	O directivă SET a fost întâlnită pentru o variabilă care nu a fost încă definită
No indirection for this instruction	S-a folosit o expresie între paranteze. Acest lucru poate însemna o încercare de indirectare într-un loc nepotrivit
Non-unary operator at start of expresion	Un operator binar (ca de exemplu "*") a fost găsit la începutul expresiei. Unele micro utilizează "*" ca operator de indirectare. Chiar dacă este un operator legitim în expresie, pot apărea ambiguități. Dacă un mod particular de instrucțiune/adresare nu permite indirectarea şi un "*" este plasat în fața expresiei asociate, asamblorul va semnala această eroare. Vezi opțiunea "-a8" în CONTROLUL ASAMBL|RII.
Object file open error	TASM nu poate deschide fişierul obiect specificat
Range of argument exceeded	Valoarea unui argument depăşeşte domeniul valid pentru modul de adresare al istrucțiunii curente
Range of relative branch exceeded	O instrucțiune de salt depăşeşte domeniul maxim
Source file open error	TASM nu poate deschide fişierul sursă specificat
Unrecognized directive	O instrucțiune care începe cu "." sau "#" are un mnemonic care nu este definit ca directivă
Unrecognized instruction	O instrucțiune are un cod operație care nu este definit
Unrecognized argument	O instrucțiune are un operand care nu e definit
Unknown token	A fost găsit un caracter nepotrivit la analiza unei expresii
Unused data in MS byte of argument	O instrucțiune sau o directivă utilizează cel mai puțin semnificativ octet al unui argument şi pierde cel mai semnificativ octet dar acesta nu este zero
Unknown option Flag	Invalid option flag has been specified on the command line. Apelați TASM fără nici o opțiune în linia de comandă pentru a vedea opțiunile valide.

Erori şi limitări

Limitări şi specificații	
Numărul maxim de simboluri	2000
Lungimea maximă a simbolurilor	32 caractere
Spațiul maxim de adresare	64 kocteți (65536 octeți)

Numărul maxim de directive INCLUDES imbricate	4
Lungimea maximă a titlului	79 caractere
Lungimea maximă a liniei sursă	255 caractere
Lungimea maximă după expandarea macro	255 caractere
Lungimea maximă a expresiilor	255 caractere
Lungimea maximă a căilor de căutare	79 caractere
Lungimea maximă a liniei de comandă	127 caractere
Numărul maxim de instrucțiuni (pe tabel)	600
Numărul maxim de macro	1000
Numărul maxim de argumente ale macro	10
Lungimea maximă a argumentului macro	16 caractere
Dimensiunea Heap (pt. simboluri, macro şi buffere)	20000 octeți
Necesar de memorie	160K

Erori

1. Expresiile nu au priorități la execuție şi deci rezultatul poate fi imprevizibil dacă nu se utilizează parantezele pentru a stabili ordinea de calcul.
2. Prima pagină din listing nu va arăta titlul definit de utilizator (definit prin directive TITLE).
3. TASM nu va genera mesaje de eroare pentru expresii formate incorect.

3.3. Exemple de programe în limbaj de asamblare, pentru microprocesorul TMS 320F240

În această secțiune vor fi prezentate programe demonstrative care să ilustreze modul de programare a unității centrale DSP TMS320F240, Texas Instruments. Programele au fost realizate cu ajutorul sistemul de dezvoltare al firmei White Mountain, DSP (WMDSP) Pathway 24x.

PROGRAMUL 1.

Este un program simplu care să arate care este structura generală a unui program scris în limbaj de asamblare.

```
;Acest program este realizat pentru a testa elementele limbajului de asamblare
;In program se aduna la o lucatie 40h pantru a genera un semnal rampa

                .nolist
                .include    "..\\..\\..\\include\\pathway.inc"

;tabela de intreruperi
;nu folosesc deocamdata intreruperile dar tabela trebuie initializata

                .ps 0fe00h                  ; starting address for this section is
                                            ; 0fe00h in Program Space (CNF = 1)

                b           0000h           ; (00h) Hardware Reset
                b           Phantom_ISR     ; (02h) Interrupt Level 1
                b           Phantom_ISR     ; (04h) Interrupt Level 2
                b           Phantom_ISR     ; (06h) Interrupt Level 3
                b           Phantom_ISR     ; (08h) Interrupt Level 4
                b           Phantom_ISR     ; (0Ah) Interrupt Level 5
                b           Phantom_ISR     ; (0Ch) Interrupt Level 6
```

```
                b               Phantom_ISR     ; (0Eh) Reserved
                b               Phantom_ISR     ; (10h) User-defined Software Interrupt
                b               Phantom_ISR     ; (12h) User-defined Software Interrupt
                b               Phantom_ISR     ; (14h) User-defined Software Interrupt
                b               Phantom_ISR     ; (16h) User-defined Software Interrupt
                b               Phantom_ISR     ; (18h) User-defined Software Interrupt
                b               Phantom_ISR     ; (1Ah) User-defined Software Interrupt
                b               Phantom_ISR     ; (1Ch) User-defined Software Interrupt
                b               Phantom_ISR     ; (1Eh) User-defined Software Interrupt
                b               Phantom_ISR     ; (20h) User-defined Software Interrupt
                b               Phantom_ISR     ; (22h) TRAP instruction vector
                b               Phantom_ISR     ; (24h)     Nonmaskable interrupt (NMI)
                b               Phantom_ISR     ; (26h) Reserved
                b               Phantom_ISR     ; (28h) User-defined Software Interrupt
                b               Phantom_ISR     ; (2Ah) User-defined Software Interrupt
                b               Phantom_ISR     ; (2Ch) User-defined Software Interrupt
                b               Phantom_ISR     ; (2Eh) User-defined Software Interrupt
                b               Phantom_ISR     ; (30h) User-defined Software Interrupt
                b               Phantom_ISR     ; (32h) User-defined Software Interrupt
                b               Phantom_ISR     ; (34h) User-defined Software Interrupt
                b               Phantom_ISR     ; (36h) User-defined Software Interrupt
                b               Phantom_ISR     ; (38h) User-defined Software Interrupt
                b               Phantom_ISR     ; (3Ah) User-defined Software Interrupt
                b               Phantom_ISR     ; (3Ch) User-defined Software Interrupt
                b               Phantom_ISR     ; (3Eh) User-defined Software Interrupt

                .list

;programul principal

inceput_memorie .EQU 310h
sfarsit_memorie .EQU 330h

;definesc o variabila temporara
                .ds 300h
temp            .word   0


                .ps 0fe50h
                .entry

;initializari
;initializez registrul de pagina din memoria de date
;initializez registrele cu zona de memorie unde scriu datele

                        ldp #06h                ;registrul de pagina din Memoria Date
                        lar ar0,#sfarsit_memorie        ;adresa maxima de memorie
                        lar ar1,#inceput_memorie        ;adresa de inceput
                        mar *,ar1               ;registrul ar5 este cel curent

;prima varianta
;Bucla:
;
;               lacl temp   ;incarc acululatorul cu continutul memoriei
;               add #40h    ;adun 40h
;               sacl temp   ;memorez rezultatul
;               b Bucla     ;continui

;varianta a doua mai perfectionata
                splk #0,*   ;stochez la adresa AR5 valoarea 0
bucla:
                lacl #40h   ;incarc acumulatorul cu 40h
                add *+      ;adun la acumulator valoarea adresata de AR5 si incrementez AR5
                cmpr 2      ;verific daca nu s-a depasit zona de memorie alocata
```

```
                bcnd cont,NTC       ;daca nu s-a depasit, continui
                lar ar1,#inceput_memorie        ;o iau de la capat
                lacl #0     ;incarc acumulatorul cu zero ca s-o ia de la capat
cont:
                sacl *      ;stochez noua valoare
                b bucla     ;continui

Phantom_ISR:
                            b Phantom_ISR

.end
```

PROGRAMUL 2.

Testarea conversiei analog numerice.

```
;Acest program este demonstrativ pentru utilizarea ADC
;Citirile se fac simultan pe cele doua ADC pe intrarea analogica ADCIN7 (pin 35 conector
;P13 MC-BUS primary Pathway) pentru ADC1 si pe intrarea analogica ADCIN15 (pin 36 conector
;P13 MC-BUS primary Pathway) pentru ADC2
;Intreruperile se genereaza cu ADC la sfarsitul conversiei.
;Pornirea conversiei se face pe eveniment GPT 1 la atingerea perioadei
;Citirile se fac la atingerea perioadei de 0.625ms (32 esantioane/perioada)
;Numaratorul GPT1 are factor de prescalare 1 si este incarcat in numarator cu 12499
;Timp = 12500/20 000 000= 0.625ms
;-----------------


                .nolist
                .include    "..\\..\\..\\include\\pathway.inc"

;tabela de intreruperi

                .ps 0fe00h                      ; starting address for this section is
                                                        ; 0fe00h in Program Space (CNF = 1)

                b           0000h                       ; (00h) Hardware Reset
                b           Phantom_ISR                 ; (02h) Interrupt Level 1
                b           Phantom_ISR                 ; (04h) Interrupt Level 2
                b           Phantom_ISR                 ; (06h) Interrupt Level 3
                b           Phantom_ISR         ; (08h) Interrupt Level 4
                b           Phantom_ISR         ; (0Ah) Interrupt Level 5
                b           Intr_ADC            ; (0Ch) Interrupt Level 6 (Intrerupere ADC)
                b           Phantom_ISR         ; (0Eh) Reserved
                b           Phantom_ISR                 ; (10h) User-defined Software Interrupt
                b           Phantom_ISR                 ; (12h) User-defined Software Interrupt
                b           Phantom_ISR         ; (14h) User-defined Software Interrupt
                b           Phantom_ISR         ; (16h) User-defined Software Interrupt
                b           Phantom_ISR         ; (18h) User-defined Software Interrupt
                b           Phantom_ISR         ; (1Ah) User-defined Software Interrupt
                b           Phantom_ISR         ; (1Ch) User-defined Software Interrupt
                b           Phantom_ISR         ; (1Eh) User-defined Software Interrupt
                b           Phantom_ISR         ; (20h) User-defined Software Interrupt
                b           Phantom_ISR                 ; (22h) TRAP instruction vector
                b           Phantom_ISR                 ; (24h)     Nonmaskable interrupt (NMI)
                b           Phantom_ISR         ; (26h) Reserved
                b           Phantom_ISR         ; (28h) User-defined Software Interrupt
                b           Phantom_ISR         ; (2Ah) User-defined Software Interrupt
                b           Phantom_ISR         ; (2Ch) User-defined Software Interrupt
                b           Phantom_ISR         ; (2Eh) User-defined Software Interrupt
                b           Phantom_ISR         ; (30h) User-defined Software Interrupt
                b           Phantom_ISR         ; (32h) User-defined Software Interrupt
                b           Phantom_ISR         ; (34h) User-defined Software Interrupt
                b           Phantom_ISR         ; (36h) User-defined Software Interrupt
```

```
                b               Phantom_ISR     ; (38h) User-defined Software Interrupt
                b               Phantom_ISR     ; (3Ah) User-defined Software Interrupt
                b               Phantom_ISR     ; (3Ch) User-defined Software Interrupt
                b               Phantom_ISR     ; (3Eh) User-defined Software Interrupt

                .list

;datele
                .ds 300h
sfarsit_memorare .word 0    ;aici se scrie 1 cand s-a umplut zona 310h-350h

;programul principal

                .ps 0fe50h                      ; adresa de inceput a programului
                                                        ; 0fe50h in Program Space
                .entry                          ; definesc punctul de intrare in program

start:
                setc    INTM                            ; INTM = 1, dezactivez intreruperile globale
                clrc    sxm                             ; nu folosesc extensia de semn

;Initializare ADC
;Programez timer GPT1 numarare periodica cu perioada de 0.625ms, atingerea perioadei
;lanseaza ADC
;---------------
                ldp #0e8h                       ;registrul de pagina => registrele managerului
                                                        ;de evenimente
;programez registrul GPTCON, pentru semnificatia bitilor vezi documentatia
                splk #0000000100000000b,GPTCON
;programez registrul T1PER cu perioada timerului
                splk #12499,T1PR
;incarc valoarea in timer
                splk #0,T1CNT
;lansez numaratorul prin programare T1CON
                splk #1001000001000100b,T1CON   ;pentru semnificatie vezi documentatia

;Programez ADC (1) si (2): lansare conversie de catre GPT1, generare intrerupere la terminarea
;conversiei

                ldp     #224                            ;DP -> pagina registrilor ADC

                splk #1101101101111110b,ADCTRL1 ;primul registru control ADC
                splk #0000010000000110b,ADCTRL2 ;al doilea registru control ADC

;------------------
;Intreruperile

;--------------
                ldp #0e8h                       ;registrul de pagina => registrele managerului
                                                        ;de evenimente
                splk #0,EVIMRA                  ;maschez toate intreruperile managerului de
                splk #0,EVIMRB                  ;evenimente EV
                splk #0,EVIMRC

                        ldp             #0
                splk    #0ffffh, IFR            ;sterg intreruperile in asteptare
                splk    #0030h, IMR                     ;activez intreruperile Level 6 si Level 5<-pt. monitor
                clrc    INTM                            ; Enable global interrupts
;--------------
;pregatesc memorarea datelor citite
                lar ar0,#3a0h           ;zona maxima de memorie pana la care stochez datele ADC2
                lar ar1,#310h           ; inceputul zonei de memorie unde stochez datele ADC1
                lar ar2,#360h           ; inceputul zonei de memorie unde stochez datele ADC2
```

```
program_principal:
                        nop
                        nop
                        nop
                        b program_principal

;Rutina citire analogica
;Datele citite sunt stocate in memorie
citire_analogica:
                        ldp             #0e0h           ; DP -> 0x7000 - 0x707f
                        lacc    ADCFIFO1                ; citesc data convertita din FIFO1
                        mar *,ar1       ; registrul curent ar1
                        sacl *+
                        lacc    ADCFIFO2                ; citesc data convertita din FIFO2
                        mar *,ar2       ; registrul curent ar2
                        sacl *+
                        cmpr 2
                        bcnd citire_analogica1,NTC
                        ldp #6
                        lacl #1
                        sacl sfarsit_memorare
citire_analogica1:
                                ret

;Intreruperea ADC
Intr_ADC:
                        ldp             #0e0h                           ; DP -> 0x7000 - 0x707f (Event Manager)
                        lacc    SYSIVR                          ; Acc = Peripheral Vector Address Offset
                        sub             #0004h                  ; 0x004 = ADC int
                        bcnd    Intr_ADC1, NEQ          ; intreruperea n-a fost ceruta de ADC
                        ldp             #6
                        lacc            sfarsit_memorare
                        bcnd    Intr_ADC1, NEQ                  ;s-a terminat memorarea
                        call citire_analogica
Intr_ADC1:
                        clrc INTM
                        ret

;intrerupere neasteptata - raman aici
Phantom_ISR:
                                b Phantom_ISR           ;intrerupere neprevazuta

.end
```

PROGRAMUL 3.

Program pentru testarea întreruperilor.

```
;Acest program este demonstrativ pentru utilizarea intreruperilor
;Voi folosi un numarator de uz general care sa aiba perioada de 0,1 secunde.
;Daca folosesc un factor de prescalare la numarator de 1/128 atunci numaratorul trebuie sa aiba
;perioada de 15625 considerand frecventa de ceas a CPU egala cu 20MHz.
;Astfel perioada va fi:
;                                       T=128*15625/20 000 000 = 0,1 secunde
;La fiecare zecime de secunda managerul de evenimente va genera o intrerupere pe care o
;folosesc pentru realizarea ceasului
;Pentru activarea intreruperilor trebuie validat: fanionul INTM=0, registrul IFR si
;registrul EVIMRx, x=A, B sau C din managerul de evenimente
;Numaratorul este programat in mod numarare directa continua (SPRU161B.PDF - pag. (2-20) 59)
;In registrul perioadei se inscrie 15624 din cauza ca numarul de impulsuri numarate este
;TxPR+1 impulsuri prescalate (divizate). In GPTCON fanionul directiei trebuie sa fie 1.
;Intrarea TMRDIR este ignorata in acest mod.
;Programul merge bine si masurarea timpului este foarte precisa.
```

```
                .nolist
                .include    "..\\..\\..\\include\\pathway.inc"

;tabela de intreruperi
;Se foloseste timerul de uz general 1 care genereaza o intrerupere in grupul A al managerului
;de evenimente care este conectata la INT 2 (Level 2) a CPU.

                .ps 0fe00h                      ; starting address for this section is
                                                        ; 0fe00h in Program Space (CNF = 1)

                b           0000h                       ; (00h) Hardware Reset
                b           Phantom_ISR                 ; (02h) Interrupt Level 1
                b           Intr_zsecunda       ; (04h) Interrupt Level 2 - tratez intreruperea la
                                                        ; o secunda generata de timer GP 1
                b           Phantom_ISR                 ; (06h) Interrupt Level 3
                b           Phantom_ISR         ; (08h) Interrupt Level 4
                b           Phantom_ISR         ; (0Ah) Interrupt Level 5
                b           Phantom_ISR                 ; (0Ch) Interrupt Level 6
                b           Phantom_ISR         ; (0Eh) Reserved
                b           Phantom_ISR                 ; (10h) User-defined Software Interrupt
                b           Phantom_ISR                 ; (12h) User-defined Software Interrupt
                b           Phantom_ISR         ; (14h) User-defined Software Interrupt
                b           Phantom_ISR         ; (16h) User-defined Software Interrupt
                b           Phantom_ISR         ; (18h) User-defined Software Interrupt
                b           Phantom_ISR         ; (1Ah) User-defined Software Interrupt
                b           Phantom_ISR         ; (1Ch) User-defined Software Interrupt
                b           Phantom_ISR         ; (1Eh) User-defined Software Interrupt
                b           Phantom_ISR         ; (20h) User-defined Software Interrupt
                b           Phantom_ISR                 ; (22h) TRAP instruction vector
                b           Phantom_ISR                 ; (24h)     Nonmaskable interrupt (NMI)
                b           Phantom_ISR         ; (26h) Reserved
                b           Phantom_ISR         ; (28h) User-defined Software Interrupt
                b           Phantom_ISR         ; (2Ah) User-defined Software Interrupt
                b           Phantom_ISR         ; (2Ch) User-defined Software Interrupt
                b           Phantom_ISR         ; (2Eh) User-defined Software Interrupt
                b           Phantom_ISR         ; (30h) User-defined Software Interrupt
                b           Phantom_ISR         ; (32h) User-defined Software Interrupt
                b           Phantom_ISR         ; (34h) User-defined Software Interrupt
                b           Phantom_ISR         ; (36h) User-defined Software Interrupt
                b           Phantom_ISR         ; (38h) User-defined Software Interrupt
                b           Phantom_ISR         ; (3Ah) User-defined Software Interrupt
                b           Phantom_ISR         ; (3Ch) User-defined Software Interrupt
                b           Phantom_ISR         ; (3Eh) User-defined Software Interrupt

                .list

;programul principal

;datele
                .ds 300h

zsecunda  .word 0
secunda   .word     0
minut               .word 0
ora                 .word 0

                .ps 0fe50h                      ; adresa de inceput a programului
                                                        ; 0fe50h in Program Space
                .entry                          ; definesc punctul de intrare in program

start:
                setc        INTM                        ; INTM = 1, dezactivez intreruperile globale
                clrc        sxm                         ; nu folosesc extensia de semn
```

```
;initializarea timerului GP 1

                ldp #0e8h                       ;registrul de pagina => registrele managerului
                                                        ;de evenimente

;programez registrul GPTCON, pentru semnificatia bitilor vezi documentatia

                splk #0000000000111111b,GPTCON

;programez registrul T1PER cu perioada timerului

                splk #15624,T1PR

;incarc valoarea in timer

                splk #0,T1CNT

;acum pregatesc intreruperile si lansez numaratorul mai tarziu prin scriere T1CON
;maschez toate intreruperile in afara de intreruperea la perioada a timerului GP 1 din
;registrii de mascare a EV

                splk #00000000010000000b,EVIMRA
                splk #0,EVIMRB
                splk #0,EVIMRC
                splk #0ffffh,EVIFRA             ;sterg eventualele intreruperi in asteptare

;maschez toate intreruperile in afara de Level 2 (timer) si Level 5 (pentru monitor) din IMR

                ldp #0
                splk #0ffffh,IFR        ;sterg eventualele intreruperi in asteptare
                splk #0012h,IMR         ;validez intreruperile

;lansez numaratorul prin programare T1CON
                ldp #0e8h                       ;registrul de pagina => registrele managerului
                                                        ;de evenimente
                splk #1001011101000100b,T1CON   ;pentru semnificatie vezi documentatia

                clrc INTM                       ;validez intreruperile globale

bucla_principala:
                        nop
                        nop
                        nop
                        ldp #6
                        lacc secunda
                        sub #60
                        bcnd un_minut,EQ
                        b bucla_principala
un_minut:
                        lacc #0
                        sacl secunda
                        lacc minut
                        add #1
                        sacl minut
                        sub #60
                        bcnd o_ora,EQ
                        b bucla_principala
o_ora:          lacc #0
                        sacl minut
                        lacc ora
                        add #1
                        sacl ora
                        sub #24
                        bcnd o_zi,EQ
```

```
                        b bucla_principala
o_zi:                   lacc #0
                        sacl ora
                        b bucla_principala

;rutina de intrerupere

Intr_zsecunda:
                ldp #6
                lacc zsecunda
                add #1
                sacl zsecunda
                sub #10
                bcnd o_secunda,EQ

Intr_zsecunda_iesire:
                ldp #0e8h                       ;registrul de pagina => registrele managerului
                                                        ;de evenimente
;               splk #0ffffh,IFR                ;sterg eventualele intreruperi in asteptare
                splk #00080h,EVIFRA             ;ACHIT INTRERUPEREA !
                clrc    INTM                            ;reactivez intreruperile si ma reintorc
                ret
o_secunda:
                lacc #0
                sacl zsecunda
                lacc secunda
                add #1
                sacl secunda
                b Intr_zsecunda_iesire

;intrerupere neasteptata - raman aici
Phantom_ISR:
                        b Phantom_ISR   ;intrerupere neprevazuta

.end
```

PROGRAMUL 4.

Testare PWM.

```
;Acest program nu este unul PWM propriu-zis. Se va genera cu ajutorul GP timer 2 un semnal
;cu factor de umplere 50% pentru (simetric) a carui frecventa sa poata fi modificata.
;Pentru aceasta se va programa timerul GP 2 in modul numarare continuu sus/jos.
;Relatia de calcul a duratei active a impulsului este:
;                       TxPR - TxCMPR
;Daca se ia TxCMPR = TxPR div 2 atunci se obtine un semnal simetric cu perioada TxPR
;Frecventa impulsurilor este data de variatia vitezei de rotatie a motorului care este
;intre 30 rot/min si 3000 rot/min adica 1ntre fmot= 0.5Hz si 50Hz.
;Frecventa de comanda este data de relatia f=6720 * fmot deci f= 3360Hz si 336000Hz.
;Raportul intre frecventa minima si cea maxima este 100.
;Calculez factorul de prescalare.
;Ceasul CPU are 20MHz. Rezulta ca valoarea ce trebuie scrisa in registrul perioadei pentru
;a obtine 3360Hz la iesire este: fCPU/(3360 * 2) = 2977 (aproximativ 3359Hz)
;iar pentru a obtine 336000Hz este: fCPU/(336000 * 2) = 30 (333 333Hz).
;Inmultesc cu 2 din cauza ca o perioada a semnalului generat este de 2 ori mai mare decat
;perioada inscrisa in numarator. Perioada este de fapt 2 x (TxPR+1).
;Cum pot calcula aceste valori, rezulta factor de prescalare = 1
;Iesirea compare/PWM folosita este cea a timerului GPT2: T2PWM/T2CMP/IOPB4 care se gaseste
;la pinul 13 a conectorului P13 (MC-BUS primary) (Pathway).
;Nu folosesc intreruperile.


                .nolist
                .include    "..\\..\\..\\include\\pathway.inc"
```

```
;tabela de intreruperi
;nu folosesc deocamdata intreruperile dar tabela trebuie initializata

                .ps 0fe00h                      ; starting address for this section is
                                                        ; 0fe00h in Program Space (CNF = 1)

                b               0000h                   ; (00h) Hardware Reset
                b               Phantom_ISR             ; (02h) Interrupt Level 1
                b               Phantom_ISR             ; (04h) Interrupt Level 2
                b               Phantom_ISR             ; (06h) Interrupt Level 3
                b               Phantom_ISR     ; (08h) Interrupt Level 4
                b               Phantom_ISR     ; (0Ah) Interrupt Level 5
                b               Phantom_ISR             ; (0Ch) Interrupt Level 6
                b               Phantom_ISR     ; (0Eh) Reserved
                b               Phantom_ISR             ; (10h) User-defined Software Interrupt
                b               Phantom_ISR             ; (12h) User-defined Software Interrupt
                b               Phantom_ISR     ; (14h) User-defined Software Interrupt
                b               Phantom_ISR     ; (16h) User-defined Software Interrupt
                b               Phantom_ISR     ; (18h) User-defined Software Interrupt
                b               Phantom_ISR     ; (1Ah) User-defined Software Interrupt
                b               Phantom_ISR     ; (1Ch) User-defined Software Interrupt
                b               Phantom_ISR     ; (1Eh) User-defined Software Interrupt
                b               Phantom_ISR     ; (20h) User-defined Software Interrupt
                b               Phantom_ISR             ; (22h) TRAP instruction vector
                b               Phantom_ISR             ; (24h)     Nonmaskable interrupt (NMI)
                b               Phantom_ISR     ; (26h) Reserved
                b               Phantom_ISR     ; (28h) User-defined Software Interrupt
                b               Phantom_ISR     ; (2Ah) User-defined Software Interrupt
                b               Phantom_ISR     ; (2Ch) User-defined Software Interrupt
                b               Phantom_ISR     ; (2Eh) User-defined Software Interrupt
                b               Phantom_ISR     ; (30h) User-defined Software Interrupt
                b               Phantom_ISR     ; (32h) User-defined Software Interrupt
                b               Phantom_ISR     ; (34h) User-defined Software Interrupt
                b               Phantom_ISR     ; (36h) User-defined Software Interrupt
                b               Phantom_ISR     ; (38h) User-defined Software Interrupt
                b               Phantom_ISR     ; (3Ah) User-defined Software Interrupt
                b               Phantom_ISR     ; (3Ch) User-defined Software Interrupt
                b               Phantom_ISR     ; (3Eh) User-defined Software Interrupt

                .list

;programul principal

;definesc variabile inmultire
                .ds 300h

ct_f_min        .word   2977    ;constanta pentru frecventa minima la iesire
ct_f_max        .word 30        ;constanta pentru frecventa maxima la iesire

                .ps 0fe50h
                .entry

;initializari
;sterg extensia de semn

                        clrc sxm                ;nu folosesc extensia de semn

;validarea iesirii comparare/PWM a timerului GPT2
;Pentru aceasta trebuie scris 1 in OCRA[12]

                ldp     #0e1h                   ; DP -> 0x7090 - 0x70ff
                splk    #1000h, OCRA            ; activarea pinului se face in registrul OCRA
```

```
;programarea numaratorului GPT2
                ldp     #232
                splk    #0000000001001000b, GPTCON
                ldp     #6
                lacl    ct_f_max                ;incarc acumulatorul cu ct. de frecventa
                ldp     #232                            ; DP -> 0x7400 - 0x747f (Event Manager)
                sacl    T2PR                            ;programez perioada
                sfr                                     ;acumulator = acumulator div 2 (pt. comparare)
                sacl    T2CMP
;programez GPT2 si lansez numaratoarea
                splk    #1010100001000010b, T2CON

start:
                nop                                     ;aici nu fac nimic deocamdata...
                nop
                nop
                b start

Phantom_ISR:
                        b Phantom_ISR

.end
```

PROGRAM 5.

Program pentru testarea QEP - unității de citire a impulsurilor codate în cuadratură.

```
;Acest program este demonstrativ pentru utilizarea QEP
;Conectez la intrarea QEP un TIRO
;Citesc QEP in registrul GPTimer 3. Continutul registrului imi da pozitia iar sensul de numarare
;imi da sensul de rotatie
;Testez acum citirea in intreruperi. Intreruperea este data de RTI la 15.63ms
;(RTIPS2..RTIPS0 = 100)
;Memorez datele in zona de memorie de la 310h pe 64 octeti ca sa vad variatia vitezei
;Rezultatul masuratorii se imparte la patru pentru a obtine numarul real de impulsuri pe durata
;de 15.63 ms

                .nolist
                .include    "..\\..\\..\\include\\pathway.inc"

;tabela de intreruperi

                .ps 0fe00h                      ; starting address for this section is
                                                        ; 0fe00h in Program Space (CNF = 1)

                b               0000h                   ; (00h) Hardware Reset
                b               Intr_monitor    ; (02h) Interrupt Level 1 (intrerupere la 15.63ms)
                b               Phantom_ISR             ; (04h) Interrupt Level 2
                b               Phantom_ISR             ; (06h) Interrupt Level 3
                b               Phantom_ISR     ; (08h) Interrupt Level 4
                b               Phantom_ISR     ; (0Ah) Interrupt Level 5
                b               Phantom_ISR             ; (0Ch) Interrupt Level 6
                b               Phantom_ISR     ; (0Eh) Reserved
                b               Phantom_ISR             ; (10h) User-defined Software Interrupt
                b               Phantom_ISR             ; (12h) User-defined Software Interrupt
                b               Phantom_ISR     ; (14h) User-defined Software Interrupt
                b               Phantom_ISR     ; (16h) User-defined Software Interrupt
                b               Phantom_ISR     ; (18h) User-defined Software Interrupt
                b               Phantom_ISR     ; (1Ah) User-defined Software Interrupt
                b               Phantom_ISR     ; (1Ch) User-defined Software Interrupt
                b               Phantom_ISR     ; (1Eh) User-defined Software Interrupt
```

```
                b               Phantom_ISR     ; (20h) User-defined Software Interrupt
                b               Phantom_ISR             ; (22h) TRAP instruction vector
                b               Phantom_ISR             ; (24h)     Nonmaskable interrupt (NMI)
                b               Phantom_ISR     ; (26h) Reserved
                b               Phantom_ISR     ; (28h) User-defined Software Interrupt
                b               Phantom_ISR     ; (2Ah) User-defined Software Interrupt
                b               Phantom_ISR     ; (2Ch) User-defined Software Interrupt
                b               Phantom_ISR     ; (2Eh) User-defined Software Interrupt
                b               Phantom_ISR     ; (30h) User-defined Software Interrupt
                b               Phantom_ISR     ; (32h) User-defined Software Interrupt
                b               Phantom_ISR     ; (34h) User-defined Software Interrupt
                b               Phantom_ISR     ; (36h) User-defined Software Interrupt
                b               Phantom_ISR     ; (38h) User-defined Software Interrupt
                b               Phantom_ISR     ; (3Ah) User-defined Software Interrupt
                b               Phantom_ISR     ; (3Ch) User-defined Software Interrupt
                b               Phantom_ISR     ; (3Eh) User-defined Software Interrupt

                .list

;programul principal

;datele
                .ds 300h
numar_impuls_QEP .word 0
sens                    .word 0
contor_intrerupere .word 1  ;aici trebuie initializat cu 1 ca sa sar prima intrerupere
sfarsit_memorare .word 0    ;cand nu mai memorez datele pun 1

;program
                .ps 0fe50h                      ; adresa de inceput a programului
                                                        ; 0fe50h in Program Space
                .entry                          ; definesc punctul de intrare in program

start:
                setc    INTM                            ; INTM = 1, dezactivez intreruperile globale
                clrc    sxm                             ; nu folosesc extensia de semn
                setc    xf                              ;aprind LED xf (daca e stins)

;prima data trebuie sa comut pinii care sunt utilizati in comun de functia primara (aici
;CAPx/QEPx) si pinii porturilor I/O

                ldp             #0e1h                   ; DP -> 0x7090 - 0x70ff

                ; activez pinii QEP1 (bit 4 = 1) si QEP2 (bit 5 = 1)
                splk    #0030h, OCRB    ; activarea pinilor se face in registrul OCRB

                ; initializez CAPFIFO stivele unitatii de captura (sterg toti bitii)
                ldp             #0e8h                   ; DP -> 0x7400 - 0x747f (Event Manager)
                splk    #00ffh, CAPFIFO

                ; setez registrul de control al GPTimer
                splk #0,GPTCON

                ; configurare GPTimer3
                splk    #0FFFFh, T3PR   ; setez perioada GPTimer3
                splk    #00000h, T3CNT  ; set contor GPTimer3
                splk    #1101100001110000b,T3CON ;registrul de control GPTimer3

                ; initializare CAPCON
                splk    #0110000000000000b, CAPCON
                splk    #1110000000000000b, CAPCON

                ;initializarea timerului de timp real IRT
```

```
                ldp #0e0h                       ;registrul de pagina => registrele sistem
                splk #01000100b,RTICR           ;intrerupere la 15.63 ms, validez intreruperile

                ;maschez toate intreruperile in afara de Level 1 (IRT) si Level 5 (pentru monitor) din IMR

                ldp #0          ;pagina zero de memorie
                splk #0ffffh,IFR        ;sterg eventualele intreruperi in asteptare
                splk #0011h,IMR         ;validez intreruperile

                clrc    INTM                    ; activare intreruperi
                lar ar0,#350h                   ; sfarsitul zonei de memorie unde stochez datele
                lar ar1,#310h                   ; inceputul zonei de memorie unde stochez datele
                mar *,ar1                       ; registrul ar1 este registrul curent

Program_principal:
                        nop
                        nop
                        b Program_principal

;Rutina de citire a contorului QEP (impulsuri codate in cuadratura - TIRO)
Citire_QEP:
                        ldp     #0e8h                   ; DP -> 0x7400 - 0x747f (Event Manager)
                        lacl    T3CNT                   ; citesc continutul numaratorului T3CNT
                        ldp     #6
                        sacl    numar_impuls_QEP        ;memorez numar impulsuri
                        ldp     #0e8h                   ; DP -> 0x7400 - 0x747f (Event Manager)
                        bit     GPTCON, 0               ; verific stare GPT3: numara sus sau jos
                        bcnd Numara_sus,TC              ;valoarea bitului testat este copiata in bit TC
Numara_jos:
                        ldp     #6
                        lacl    #0
                        sacl    sens
                        lacl    numar_impuls_QEP
                        neg
                        sacl    numar_impuls_QEP
                        b Iesire_QEP
Numara_sus:
                        ldp     #6
                        lacl    #1
                        sacl    sens
Iesire_QEP:
                        ldp             #0e8h                   ; DP -> 0x7400 - 0x747f (Event Manager)
                        splk    #00000h, T3CNT          ; sterg contor GPTimer3
                        ret

;Rutina de memorare valori citite de la QEP pentru a vedea corectitudinea
Memorare_date:
                        ldp #6
                        lacl numar_impuls_QEP
                        sacl *+
                        cmpr 2
                        bcnd cont_mem,NTC
                        lacl sfarsit_memorare
                        add #1
                        sacl sfarsit_memorare
cont_mem:
                        ret

;Intrerupere IRT
Intr_monitor:
                ldp #6
                lacc contor_intrerupere
                sub #1
                bcnd Intr_monitor1,EQ           ;nu ma intereseaza prima intrerupere
```

```
                call Citire_QEP
                ldp #6
                lacc sfarsit_memorare
                cc Memorare_date,EQ             ;cat timp sfarsit_memorare=0 pot memora
                clrc INTM
                ret
Intr_monitor1:
                lacc #5                         ;o valoare oarecare diferita de 1
                sacl contor_intrerupere
                clrc INTM
                ret

;intrerupere neasteptata - raman aici
Phantom_ISR:
                        b Phantom_ISR       ;intrerupere neprevazuta

.end
```

PROGRAM 6.

Salvarea şi restaurarea regiştrilor de stare ai CPU.

```
;Testez salvarea si restaurarea registrilor de stare ai CPU

                .nolist
                .include    "..\\..\\..\\include\\pathway.inc"

;tabela de intreruperi
;nu folosesc deocamdata intreruperile dar tabela trebuie initializata

                .ps 0fe00h                      ; starting address for this section is
                                                        ; 0fe00h in Program Space (CNF = 1)

                b           0000h                   ; (00h) Hardware Reset
                b           Phantom_ISR             ; (02h) Interrupt Level 1
                b           Phantom_ISR             ; (04h) Interrupt Level 2
                b           Phantom_ISR             ; (06h) Interrupt Level 3
                b           Phantom_ISR     ; (08h) Interrupt Level 4
                b           Phantom_ISR     ; (0Ah) Interrupt Level 5
                b           Phantom_ISR             ; (0Ch) Interrupt Level 6
                b           Phantom_ISR     ; (0Eh) Reserved
                b           Phantom_ISR             ; (10h) User-defined Software Interrupt
                b           Phantom_ISR             ; (12h) User-defined Software Interrupt
                b           Phantom_ISR     ; (14h) User-defined Software Interrupt
                b           Phantom_ISR     ; (16h) User-defined Software Interrupt
                b           Phantom_ISR     ; (18h) User-defined Software Interrupt
                b           Phantom_ISR     ; (1Ah) User-defined Software Interrupt
                b           Phantom_ISR     ; (1Ch) User-defined Software Interrupt
                b           Phantom_ISR     ; (1Eh) User-defined Software Interrupt
                b           Phantom_ISR     ; (20h) User-defined Software Interrupt
                b           Phantom_ISR             ; (22h) TRAP instruction vector
                b           Phantom_ISR             ; (24h)     Nonmaskable interrupt (NMI)
                b           Phantom_ISR     ; (26h) Reserved
                b           Phantom_ISR     ; (28h) User-defined Software Interrupt
                b           Phantom_ISR     ; (2Ah) User-defined Software Interrupt
                b           Phantom_ISR     ; (2Ch) User-defined Software Interrupt
                b           Phantom_ISR     ; (2Eh) User-defined Software Interrupt
                b           Phantom_ISR     ; (30h) User-defined Software Interrupt
                b           Phantom_ISR     ; (32h) User-defined Software Interrupt
                b           Phantom_ISR     ; (34h) User-defined Software Interrupt
                b           Phantom_ISR     ; (36h) User-defined Software Interrupt
                b           Phantom_ISR     ; (38h) User-defined Software Interrupt
                b           Phantom_ISR     ; (3Ah) User-defined Software Interrupt
                b           Phantom_ISR     ; (3Ch) User-defined Software Interrupt
```

```
                b                       Phantom_ISR         ; (3Eh) User-defined Software Interrupt

                .list

;programul principal

;definesc variabile inmultire
                .ds 300h
RStare0         .word   0           ;Stochez registrul de stare 0
RStare1         .word 0     ;Stochez registrul de stare 1

                .ps 0fe50h
                .entry

;initializari

                        ldp #06h            ;registrul de pagina din Memoria Date
                        setc sxm            ;folosesc extensia de semn

start:
                        clrc INTM           ;activez intreruperile
                        lar ar0,#0300h      ;acestea sunt registrele pe care le folosesc in program
                        lar ar1,#0350h      ;               -"-
                        mar *,ar0           ;registrul implicit este ar0
                        lar ar6,#RStare0    ;registrul folosit pentru salvarea registrilor de stare
                        ldp #0e0h           ;registrul de pagina
                        mar *,ar6           ;pregatesc salvarea registrilor de stare
                        setc INTM           ;dezactivez intreruperile
                        sst #0,*+           ;salvez ST0
                        sst #1,*-           ;salvez ST1
                        setc INTM
                        ldp #6              ;schimb registrul de pagina sa vad daca va fi restaurat
                        lst #0,*+           ;restaurez ST1
                        lst #1,*-           ;restaurez ST0
;aici ar trebui sa fie din nou ar0 registru implicit si DP la 0e0h
;OK asa se si intampla...
                        b start

Phantom_ISR:
                        b Phantom_ISR

.end
```

CAPITOLUL 4

PROGRAMAREA MICROCONTROLERELOR DE TIP PIC12, PIC16 ŞI PIC 18

Unitățile centrale de tip de tip RISC PIC12, PIC16 şi PIC18 au un set de 35 de instrucțiuni cu lungimea de 14 biți. Programarea acestora se face cu ajutorul mediului de programare MPLAB furnizat gratuit de firma Microchip. Programul obținut în cod obiect absolut este în format Intel Hex specific programatoarelor cu ajutorul cărora programul este înscris în memoria microcontrolerului.

Programarea microcontrolerelor PIC poate fi făcută prin intermediul interfeței ICSP (In-Circuit Serial Programming). Această interfață conține 5 linii dintre care pe două linii se transmit datele în format serial şi semnalul de ceas a acestora iar pe celelalte trei sunt aplicate tensiunea de alimentare, tensiunea de programare şi masa (potențialul de referință). Toate aceste linii sunt comune cu liniile microcontrolerului pe care sunt în mod obişnuit semnale ale perifeicelor.

Programarea unui microcontroler presupune mai întâi scrierea programului sursă într-un limbaj de nivel înalt (Pascal, Basic, C etc) sau in limbaj de asamblare, compilarea acestuia şi scrierea programului în memoria Flash a microcontrolerului.

Toate programatoarele destinate microcontrolerelor necesită ca progrmele direct executabile, care vor fi înscrise în memoria de program a microcontrolerului, să fie sub un format special denumit format hexazecimal. Această denumire provine din faptul că aceste fişiere conțin codul program sub formă hexazecimală, scris cu caractere ASCII.

4.1. Organizarea memoriei microcontrolerelor PIC

Datorită faptului că microcontrolerele din familiile PIC12, PIC16 şi PIC18 au aceeaşi structură a unității centrale, diferențele apărând datorită perifericelor existente şi a memoriei folosite, vom prezenta în continuare structura unui microcontroler simplu, utilizat pe scară largă, microcontrolerul PIC16F84A.

Memoria microcontrolerului se compune din:

- memorie flash - unde se scrie programul;
- eeprom - memorie de date - date importante pentru program;
- ram - date temporare in execuția programului.

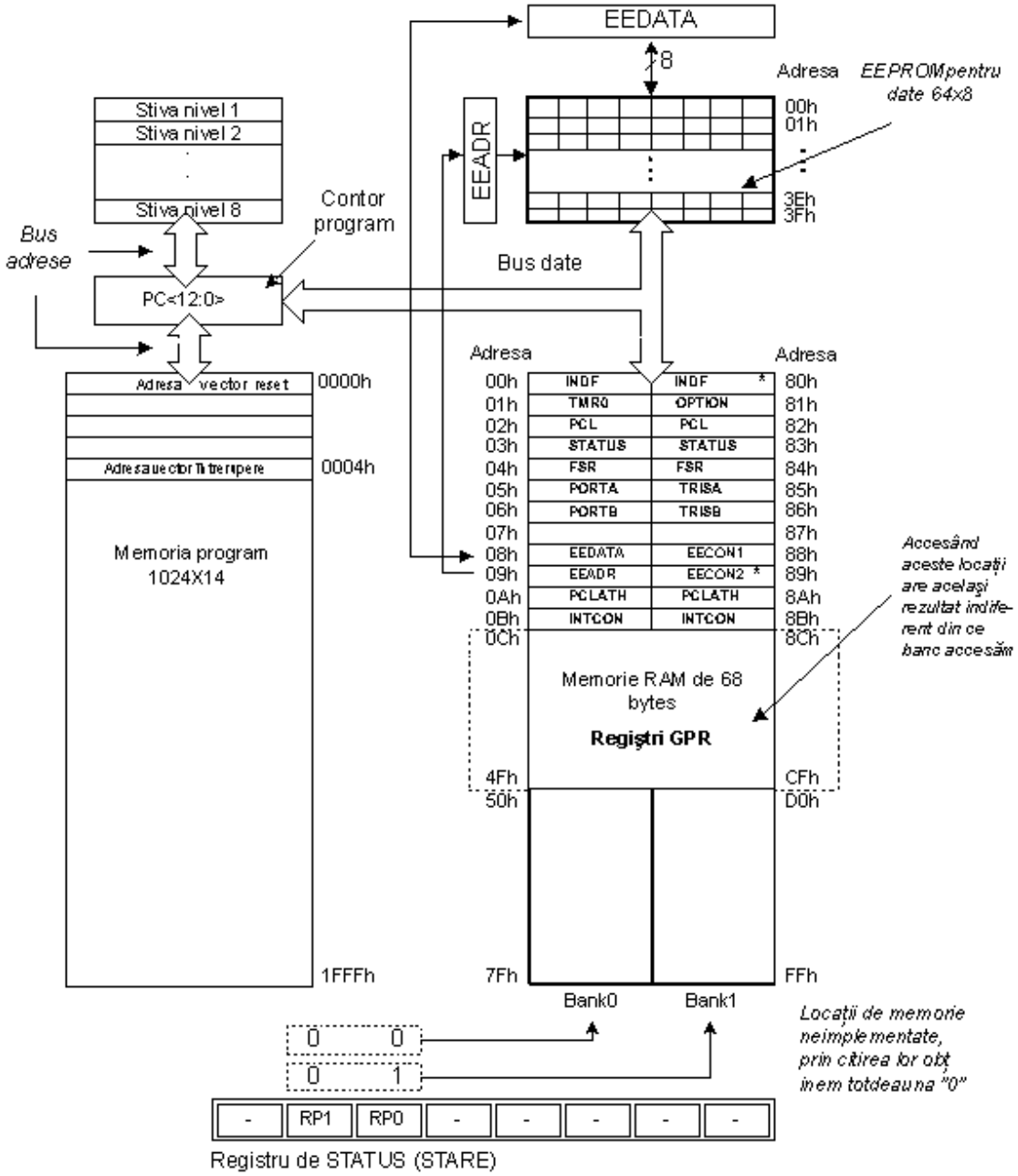
Registre:

- registrul de lucru w;

- registrul de stare (status) - conține biții de stare;
- GPR (General Purpose Registers) - registre de uz general;
- SFR SpecialFunction Registers - registre cu funcție specială;

Stiva este separată şi are opt nivele.

PIC16F84 are două blocuri separate de memorie, unul pentru date şi celălalt pentru programe. Memoria EEPROM şi regiştrii GPR în memoria RAM constituie blocul de date iar memoria FLASH constituie un blocul de programe.



Organizarea memoriei microcontrolerului PICF84

4.1.1. Memoria program

Memoria de program este o memorie flash. Mărimea memoriei program este de 1024 locații cu lățime de 14 biți unde locațiile zero şi patru sunt rezervate pentru reset şi pentru vectorul întrerupere.

4.1.2. Memoria de date

Memoria de date constă din memoriile EEPROM şi RAM. Memoria EEPROM constă din 64 de locații de opt biți a căror conținut nu este pierdut în timpul opririi sursei de alimentare. EEPROM-ul nu este direct adresabil, dar este accesat indirect prin regiştrii EEADR şi EEDATA. Pentru că memoria EEPROM este folosită curent la memorarea unor parametri importanți (de exemplu, o temperatură dată în regulatoarele de temperatură), există o procedură strictă de scriere în EEPROM ce trebuie urmată pentru a preveni scrierea accidentală. Memoria RAM pentru date ocupă un spațiu într-o hartă a memoriei de la locația 0x0C la 0x4F ceea ce înseamnă 68 de locații. Locațiile memoriei RAM sunt de asemenea denumite registre GPR care este o abreviere General Purpose Registers-Registre cu Scop General. Registrele GPR pot fi accesate indiferent de ce banc este selectat la un moment.

4.1.3. Registrele SFR

La microcontrolerul 16F64A registrele ce ocupă primele 12 locacii în bancurile 0 şi 1 sunt registre cu funcții speciale asociate cu unele blocuri ale microcontrolerului. Acestea sunt numite Special Function Registers - Registre cu Funcții Speciale.

4.1.4. Bancuri de Memorie

În afară de această diviziune în 'lungime' a regiştrilor SFR şi GPR, harta memoriei este de asemenea împărțită în 'adâncime' în zone numite 'bancuri'. Selectarea unuia din bancuri se face de biții RPO şi RP1 din registrul STATUS de stare.

Exemplu:

```
        bcf STATUS, RP0
```

Instrucțiunea BCF şterge bitul RP0 (RP0=0) în registrul STATUS şi astfel setează bancul 0.

```
        bsf STATUS, RP0
```

Instrucțiunea BSF setează bitul RP0 (RP0=1) în registrul STATUS şi astfel setează bancul 1.

Cu ajutorul macrocomenzilor, selecția dintre două bancuri devine mai clară şi programul mult mai inteligibil.

```
BANK0 macro
      Bcf STATUS, RP0   ;Select memory bank 0
```

```
        Endm

BANK1 macro
      Bsf STATUS, RP0   ;Select memory bank 1
      Endm
```

NOTĂ:

Locațiile 0Ch - 4Fh sunt regiştri cu scop general (GPR) ce sunt folosiți ca memorie RAM. Când sunt accesate locațiile 8Ch - CFh în Bancul 1, accesăm de fapt exact aceleaşi locații în Bancul 0. Cu alte cuvinte, când trebuie accesat unul din regiştrii GPR, nu trebuie ținut cont de banc.

4.1.5. Contorul de Program

Contorul de program (PC) este un registru de 13 biți ce conține adresa instrucțiunii ce se execută. Prin incrementarea sau schimbarea sa (ex. în caz de salturi) microcontrolerul execută instrucțiunile de program pas-cu-pas.

4.1.6. Stiva

PIC16F84 are o stivă de 13 biți cu 8 nivele, sau cu alte cuvinte, un grup de 8 locații de memorie, de 13 biți lățime, cu funcții speciale. Rolul său de bază este de a păstra valoarea contorului de program după un salt din programul principal la o adresă a unui subprogram. Pentru ca un program să ştie cum să se întoarcă la punctul de unde a s-a produs un salt la un subprogram, trebuie să salveze valoarea contorului programului în stivă. Când se produce saltul dintr-un program într-un subprogram, contorul programului este salvat în stivă (de exemplu la execuția instrucțiunii CALL). Când se execută o instrucțiune ca RETURN, RETLW sau RETFIE ce se găseşte la sfârşitul unui subprogram, contorul programului este extras din stivă, în aşa fel încât programul principal să poată continua execuția din punctul în care a fost întrerupt la apariția apelului de subprogram. Aceste operații de plasare într-o şi luare dintr-o stivă a contorului de program sunt numite PUSH şi POP, la fel cu instrucțiunile similare ale unor microcontrolere mai mari.

4.1.7. Registrul STATUS (ADRESA: 03h, 83h)

Registrul STATUS conține starea aritmetică ALU (C, DC, Z), starea RESET (TO, PD) şi biții pentru selectarea bancului de memorie (IRP, RP1, RP0). Considerând că selecția bancului de memorie este controlată prin acest registru, el trebuie să fie prezent în fiecare banc. Registrul STATUS poate fi o destinație pentru orice instrucțiune, cu oricare alt registru. Dacă registrul STATUS este o destinație pentru instrucțiunile ce afectează biții Z, DC or C, atunci scrierea în aceşti trei biți nu este posibilă.

R/W-0	R/W-0	R/W-0	R - 1	R - 1	R/W-x	R/W-x	R/W-x
IRP	RP1	RP0	/TO	/PD	Z	DC	C
Bit 7	bit 6	bit 5	bit 4	Bit 3	bit 2	bit 1	bit 0

R = bit de citire; W = bit de scriere U = bit neimplementat, citit ca zero; n = valoare la resetul power-on; '1' = bitul este setat; '0' = bitul este resetat; x = valoarea bitului este nu este cunoscută

Bit 0 C (Carry bit) - Transfer.
Bit care este afectat de operațiile de adunare, scădere şi transfer.
1 = s-a produs un transfer din bitul cel mai semnificativ al rezultatului;
0 = transferul nu s-a produs.
Bitul C este afectat de instrucțiunile: addwf, addlw, sublw, subwf.

Bit 1 DC (Digit Carry bit) - DC transfer.
Bit afectat de operațiile de adunare, scădere şi transfer. Spre deosebire de bitul C, acest bit reprezintă transferul între biții mediani ai rezultatului. Este setat la adunare când se produce un transport de la bitul 3 la bitul 4, sau de scădere când se produce împrumutul de la bitul 4 de către bitul 3, sau transfer în ambele direcții.
1 = transfer produs la al patrulea bit al rezultatului;
0 = transferul nu s-a produs.
Bitul DC este afectat de instrucțiunile: addwf, addlw, sublw, subwf.

Bit 2 Z (Zero bit) - indică un rezultat egal cu zero.
Acest bit este setat atunci când rezultatul unei operații logice sau aritmetice esecutate este zero.
1 = rezultatul este egel cu zero;
0 = rezultat diferit de zero.

Bit 3 /PD (Power Down bit)
Bit ce este setat când microcontrolerul este alimentat şi începe să funcționeze, după fiecare reset obişnuit şi după executarea instrucțiunii CLRWDT. Instrucțiunea SLEEP resetează acest bit când microcontrolerul intră în regimul de consum redus. Setarea lui repetată este posibilă prin reset sau prin oprirea şi pornirea sursei.
Setarea poate fi triggerată de asemenea de un semnal de la pinul RB0/INT, de o schimbare la portul RB, la terminarea scrierii în EEPROM-ul de date intern şi de watchdog.
1 = după ce sursa a fost pornită;
0 = executarea instrucțiunii SLEEP.

Bit 4 /TO (Time Out bit) - depăşirea (overflow) a watchdog-ului
Bitul este setat după pornirea sursei de alimentare şi execuția instrucțiunilor: CLRWDT şi SLEEP. Bitul este resetat când watchdog-ul ajunge la sfârşit semnalând că ceva nu este în ordine.
1= depăşirea-oveflow nu s-a produs;
0= depăşirea-overflow s-a produs.

Bit 6:5 RP1:RP0 (Register Bank Select bits) - biți de selectare a bancului de regiştri

Aceşti doi biți reprezintă partea superioară a adresei la adresarea directă. Pentru că instrucțiunile ce adresează memoria direct au doar şapte biți de adresă mai este necesar de încă un bit pentru a adresa cei 256 octeți câți are PIC16F84. Bitul RP1 nu este folosit, dar este lăsat pentru extinderi viitoare ale acestui microcntroler.
01= primul banc
00= bancul zero

Bit 7 IRP (Register Bank Select bit) - bit de selectare a bancului de regiştri. Bit al cărui rol este de a fi al optulea bit la adresarea indirectă a RAM-ului intern.
1= bancul 2 şi 3
0= bancul 0 şi 1 (de la 00h la FFh)

4.1.8. Registrul OPTION (ADRESA: 81h)

Registrul OPTION este un registru în care se poate scrie şi care se poate citi şi care conține diferiți biți de configurare pentru circuitul de prescalare TMR0/WDT, întreruperea externă INT, TMR0 şi the weak pull-ups on PORTB.

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
/RBPU	INTEDG	TOCS	TOSE	PSA	PS2	PS1	PS0
bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

R = bit de citire; W = bit de scriere U = bit neimplementat, citit ca zero; n = valoare la resetul power-on; '1' = bitul este setat; '0' = bitul este resetat; x = valoarea bitului este nu este cunoscută

Bit 0:2 PS0, PS1, PS2 (Prescaler Rate Select bits) - Bit Selecție Rată Prescaler
Aceşti trei biți definesc valoarea constantei de prescalare a contorului de temporizare (timer) TMR0.

Biții	TMR0	WDT
000	1:2	1:1
001	1:4	1:2
010	1:8	1:4
011	1:16	1:8
100	1:32	1:16
101	1:64	1:32
110	1:128	1:64
111	1:256	1:128

Bit 3 PSA (Prescaler Assignment bit) - Bit de Asignare Prescaler. Bit ce asignează prescalerul între TMR0 şi watchdog.
1= prescalerul este asignat watchdog-ului
0= prescalerul este asignat timer-ului liber (free-run) TMR0.

Bit 4 T0SE (TMR0 Source Edge Select bit) - Bit Selecție a Frontului Sursei TMR0.
Dacă este permis de a se triggera TMR0 prin impulsurile de la pinul RA4/T0CKI, acest bit determină dacă aceasta va fi la frontul descrescător sau crescător al unui semnal.
1= front crescător
0= front descrescător

Bit 5 TOCS (TMR0 Clock Source Select bit) - Bit Selecție Sursă Ceas TMR0.
Acest pin permite timerului liber (free-run) să incrementeze starea lui fie de la oscilatorul intern la fiecare ¼ a ceasului oscilatorului, fie prin impulsuri externe la pinul RA4/T0CKI.
1= impulsuri externe
0= ceas intern 1/4

Bit 6 INTEDG (Interrupt Edge Select bit) - Bit de Selecție a Frontului Întrerupere.
Dacă întreruperea este activată este posibil ca acest bit să determine frontul la care o întrerupere va fi activată la pinul RB0/INT.
1= front crescător
0= front descrescător

Bit 7 /RBPU (PORTB Pull-up Enable bit) - Bit Enable-Activare Pull-up PORTB.
Acest bit porneşte şi opreşte rezistorii interni 'pull-up'- de ieşire la portul B.
1= Rezistori oprire "pull-up"
0= Rezistori pornire "pull-up"

4.1.9. Registrul INTCON (ADRESA: 0Bh, 8Bh)

Registrul INTCON este un registru ce poate fi scris şi citit şi care conține diferiți biți de validare pentru toate sursele de intrerupere.

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF
bit 7	Bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

R = bit de citire; W = bit de scriere U = bit neimplementat, citit ca zero; n = valoare la resetul power-on; '1' = bitul este setat; '0' = bitul este resetat; x = valoarea bitului este nu este cunoscută

Bit 7 GIE (Global Interrupt Enable bit) - bit de validare globală a întreruperilor.
1 = validează toate întreruperile nemascate
0 = dezactivează toate întreruperile

Bit 6 EEIE (EE Write Complete Interrupt Enable bit) - bit de validare a întreruperii de scriere a memoriei EEPROM
1 = activează EE Write Complete Interrupts
0 = dezactivează EE Write Complete Interrupts

Bit 5 T0IE (TMR0 Owerflow Interrupt Enable bit) - bit de validare a întreruperii de depăşire al temporizatorului TMR0
1 = validează întreruperile TMR0
0 = dezactivează întreruperile TMR0

Bit 4 INTE (RB0/INT External Interrupt Enable bit) - bit de validare a întreruperii externe
1 = validează întreruperea externă RB0/INT
0 = dezactivează întreruperea externă RB0/INT

Bit 3 RBIE (RB Port Change Interrupt Enable bit) - bit de validare a înreruperii la schimbare produsă la port RB
1 = validează întreruperea
0 = invalidează întreruperea

Bit 2 T0IF (TMR0 Overflow Interrupt Flag bit) - fanionul de semnalizare a întreruperii de depăşire a TMR0
1 = registrul TMR0 a fost depăşit (fanionul trebuie şters prin software)
0 = registrul TMR0 nu a fost depăşit

Bit 1 INTF (RB0/INT External Interrupt Flag bit) - fanion de semnalizare a întreruperii externe
1 = întreruperea externă RB0/INT s-a produs (fanionul trebuie şters prin program)
0 = întreruperea externă RB0/INT nu s-a produs

bit 0 RBIF (RB Port Change Interrupt Flag bit) - fanion de semnalizare a întreruperii de apariţie a unei schimbări la portul RB
1 = la cel puţin unul din pinii RB7 - RB4 a apărut o schimbare de stare (trebuie şters prin program)
0 = la nici unul din pinii RB7 - RB4 nu a apărut o schimbare de stare

4.1.10. PCL şi PCLATH

Contorul de program (PC) indică adresa instrucţiunii ce urmează a fi executată. PC are o lăţime de 13 biţi. Cel mai puţin semnificativ octet este registrul PCL. Acest registru poate fi scris şi citit. Cel mai semnificativ octet este registrul PCH. Acest registru conţine biţii PC<12:8> şi nu poate fi scris şi citit direct. Dacă valoarea contorului de program (PC) este modificată sau un test de condiţie este adevărat, instrucţiunea necesită două cicluri. Al doilea ciclu este executat ca o instrucţiune NOP. Toate actualizările registrului PCH se fac prin intermediul registrului PCLATH.

4.1.11. Memoria de date EEPROM

Memoria de date se adresează în mod indirect prin intermediul regiştrilor cu funcţii speciale. Sunt patru regiştrii SFR pentru scrierea şi citirea memoriei EEPROM: EECON1, EECON2 (registru neimplementat fizic), EEDATA şi EEADR.

EEDATA conţine cei opt biţi de date pentru scriere sau citire iar EEADR conţine adresa locaţiei de memorie EEPROM ce va fi accesata. PIC16F84A are 64 de octeţi de memorie EEPROM adresabili în plaja 0h la 3Fh.

Memoria de date EEPROM permite atât scrierea cât şi citirea. Un octet scris sterge în mod automat locaţia înainte de a scrie data (erase before write).

4.1.12. Registrul EECON1 (ADRESA: 88h)

U-0	U-0	U-0	R/W-0	R/W-x	R/W-0	R/S-0	R/S-0
-	-	-	EEIF	WRERR	WREN	WR	RD
bit 7	Bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

R = bit de citire; W = bit de scriere U = bit neimplementat, citit ca zero; n = valoare la resetul power-on; '1' = bitul este setat; '0' = bitul este resetat; x = valoarea bitului este nu este cunoscută

Biţii 7-5 Neimplementaţi - sunt citiţi ca '0'

Bit 4 EEIF (EEPROM Write Operation Interrupt Flag bit) - fanion de semnalizare a întreruperii de scriere a EEPROM
1 = operaţia de scriere terminată (bitul trebuie şters prin program)
0 = operaţa de scriere nu s-a terminat sau nu a început

Bit 3 WRERR (EEPROM Error Flag bit) - fanion de eroare a operaţiei de scriere în EEPROM
1 = operaţia de scriere s-a terminat prematur (orice reset /MCLR sau orice reset WDT pe durate operării normale)
0 = operaţia de scriere s-a terminat

Bit 2 WREN (EEPROM Write Enable bit) - bit de validare a scrierii în EEPROM
1 = permite cicluri de scriere
0 = inhibă scrierea în memoria EEPROM

Bit 1 WR (Write Control bit) - bit de control a scrierii
1 = iniţiază un ciclu de scriere. Bitul este şters de hardware o dată ce scrierea este completă. Bitul WR poate fi numai setat (nu şi şters - resetat) prin program.

Bit 0 RD (Read Control bit) - bit de control a citirii
1 = iniţiază o citire din EEPROM şi este şters de hardware. Bitul RD poate fi numai setat (nu şi şters - resetat) prin program.
0 = nu se iniţiază o citire din EEPROM

4.1.13. Citirea memoriei EEPROM

Pentru a citi o dată din memoria EEPROM utilizatorul trebuie să scrie adresa în registrul EEADR şi să seteze bitul de control RD (EECON1<0>). Data este disponibilă în următorul ciclu, si deci va putea fi citită de următoarea instrucţiune, în registrul

EEDATA care reţine această dată până la următoarea citire sau scriere din/în memoria EEPROM.
Exemplu, citire din memoria EEPROM:

```
BCF         STATUS, RP0       ;Bank 0
MOVLW       CONFIG_ADDR
MOVWF       EEADR             ;Address to read
BSF         STATUS, RP0       ;Bank 1
BSF         EECON1, RD              ;EE Read
BCF         STATUS, RP0       ;Bank 0
MOCVF EEDATA, W               ;W = EEDATA
```

4.1.14. Scrierea în memoria de date EEPROM

Pentru a scrie într-o locaţie a memoriei EEPROM utilizatorul trebuie să scrie adresa locaţiei de memorie în registrul EEADR şi data în registrul EEDATA. Dupa aceasta, utilizatorul trebuie să urmărească secvenţa specifică pentru a iniţia ciclul de scriere.
Exemplu, scrierea în memoria EEPROM:

```
BSF         STATUS, RP0 ;Bank 1
BCF         INTCON, GIE       ;Disable INTs.
BSF         EECON1, WREN      ;Enable Write
MOVLW 55h               ;
.....................................................
MOVWF  EECON2           ;Write 55h
MOVLW AAh               ;
MOVWF EECON2            ;Write AAh
BSF         EECON1, WR        ;Set WR bit begin write
BSF         INTCON, GIE       ;Enable INTs.
.....................................................
```

Secventa obligatorie

Scrierea nu este iniţiată dacă secvenţa de mai sus nu este realizată exact (se scrie 55h în EECON2, se scrie Aah în EECON2 şi apoi se setează bitul WR) pentru fiecare octet scris. Este recomandat ca întreruperile să fie dezactivate pe parcursul acestei secvenţe de cod.

În plus, bitul WREN din registrul EECON1 trebuie setat pentru a permite scrierea. Acest mecanism previne scrierea accidentală în memoria de date EEPROM ce poate apărea datorită execuţiei neaşteptate a unei secvenţe de program (de exemplu programe pierdute). Utilizatorul trebuie să ţină permanent bitul WREN şters cu excepţia cazurilor când se face înnoirea conţinutului memoriei EEPROM. Bitul WREN nu este şters de către hardware.

După ce secvenţa de scriere a fost iniţiată, ştergerea bitului WREN nu va afecta ciclul de scriere.

La completarea ciclului de scriere bitul WR este şters de către hardware şi fanionul de scriere completă în EE (EEIF) este setat. Utilizatorul poate activa întreruperile sau poate testa prin program acest bit. Bitul EEIF trebuie şters prin program.

4.1.15. Verificarea scrierii

În funcţie de aplicaţie, un obicei bun în programare poate să ceară verificarea datelor scrise in memoria EEPROM. Metoda de verificare prezentată în exemplul de mai jos trebuie folosită atunci când memoria EEPROM este folosită la limitele de stres. Altfel, eşuarea scrierii în memoria EEPROM va fi dată de bitul WRERR din registrul EECON1 care întoarce valoarea unu în caz de eroare.
Exemplu de verificare a scrierii în memoria EEPROM:

```
      BCF         STATUS, RP0 ;Bank 0
      ;                       ;Any code can go here
      MOVF        EEDATA,W    ;Must be in Bank 0
      BSF         STATUS, RP0 ;Bank 1
READ
      BSF         EECON1, RD  ;ZES Read the value written
      BCF         STATUS, RP0 ;Bank 0
                              ;
                              ;Is the value written (in W reg)
and read
                              ;(in EEDATA) the same?
                              ;
      SUBWF EEDATE,W      ;
      BTFSS       STATUS, Z   ;Is difference 0?
      GOTO        WRITE_ERR   ;NO, Write error
```

4.1.16. Harta memoriei RAM

Adresa registrului (File Address)	Location	Location	Adresa registrului (File Address)
00h	INDF Indirect addr.(1)	INDF Indirect addr.(1)	80h
01h	TMR0	OPTION_REG	81h
02h	PCL	PCL	82h
03h	STATUS	STATUS	82h
04h	FSR	FSR	84h
05h	PORTA	TRISA	85h
06h	PORTB	TRISB	86h
07h	-	-	87h
08h	EEDATA	EECON1	88h
09h	EEADR	EECON2(1)	89h
0Ah	PCLATH	PCLATH	8Ah
0Bh	INTCON	INTCON	8Bh
0Ch 4Fh	68 REGIŞTRII DE UZ GENERAL	ACEIAŞI REGIŞTRII CU CEI DIN BANK-UL 0	8Ch CFh
50h 7Fh	LIBER	LIBER	D0h FFh
	Bank 0	Bank 1	

(1) NU REPREZINTĂ UN REGISTRU FIZIC.

Prezentarea sumară a registrelor

Addr	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on Power-on RESET	Details on page
Bank 0											
00h	INDF	Uses contents of FSR to address Data Memory (not a physical register)								---- ----	11
01h	TMR0	8-bit Real-Time Clock/Counter								xxxx xxxx	20
02h	PCL	Low Order 8 bits of the Program Counter (PC)								0000 0000	11
03h	STATUS(2)	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1xxx	8
04h	FSR	Indirect Data Memory Address Pointer 0								xxxx xxxx	11
05h	PORTA(4)	—	—	—	RA4/T0CKI	RA3	RA2	RA1	RA0	---x xxxx	16
06h	PORTB(5)	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0/INT	xxxx xxxx	18
07h	—	Unimplemented location, read as '0'								—	—
08h	EEDATA	EEPROM Data Register								xxxx xxxx	13,14
09h	EEADR	EEPROM Address Register								xxxx xxxx	13,14
0Ah	PCLATH	—	—	—	Write Buffer for upper 5 bits of the PC(1)					---0 0000	11
0Bh	INTCON	GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x	10
Bank 1											
80h	INDF	Uses Contents of FSR to address Data Memory (not a physical register)								---- ----	11
81h	OPTION_REG	RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	1111 1111	9
82h	PCL	Low order 8 bits of Program Counter (PC)								0000 0000	11
83h	STATUS (2)	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1xxx	8
84h	FSR	Indirect data memory address pointer 0								xxxx xxxx	11
85h	TRISA	—	—	—	PORTA Data Direction Register					---1 1111	16
86h	TRISB	PORTB Data Direction Register								1111 1111	18
87h	—	Unimplemented location, read as '0'								—	—
88h	EECON1	—	—	—	EEIF	WRERR	WREN	WR	RD	---0 x000	13
89h	EECON2	EEPROM Control Register 2 (not a physical register)								---- ----	14
0Ah	PCLATH	—	—	—	Write buffer for upper 5 bits of the PC(1)					---0 0000	11
0Bh	INTCON	GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x	10

Legend: x = unknown, u = unchanged. - = unimplemented, read as '0', q = value depends on condition

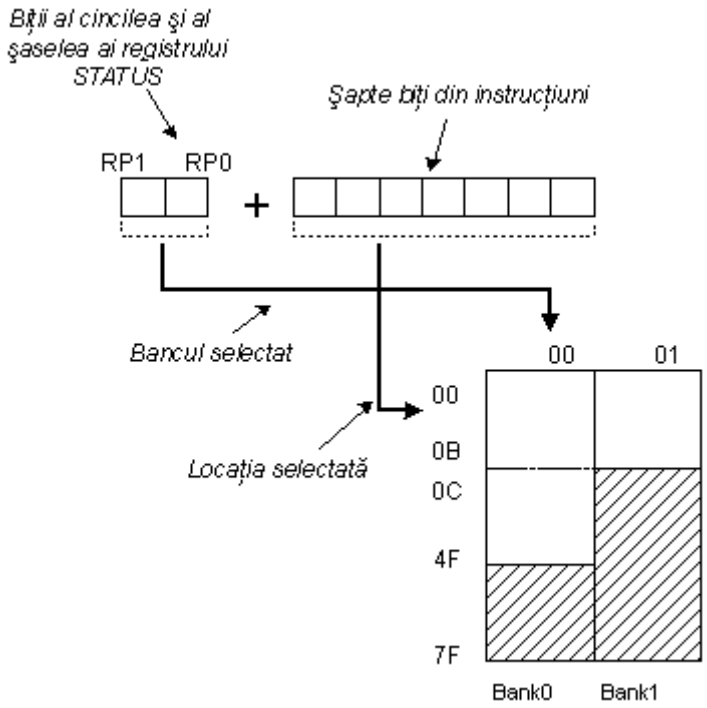
Notă:

1. Cel mai semnificativ octet al contorului de program nu este direct accesibil. PCLATH este un registru slave pentru PC<12:8>. Conținutul registruluin PCLATH poate fi transferat în octetul cel mai semnificativ al contorului de program dar conținutul PC<12:8> nu poate fi transferat în PCLATH.
2. Biții de stare TO şi PD din registrul STATUS nu sunt afectati de /MCLR Reset.
3. Alte initializări RESET (care nu sunt determinate de cuplarea sursei de alimentare) include RESET extern prin intermediul /MCLR şi Watchdog Timer Reset.
4. La orice RESET al dispozitivului, aceşti pini sunt sunt configurați ca intrări.
5. Aceasta este valoarea ce va fi în latch-ul portului de ieşire.

4.1.17. Moduri de adresare

Locațiile de memorie RAM pot fi accesate direct sau indirect.

Adresarea Directă



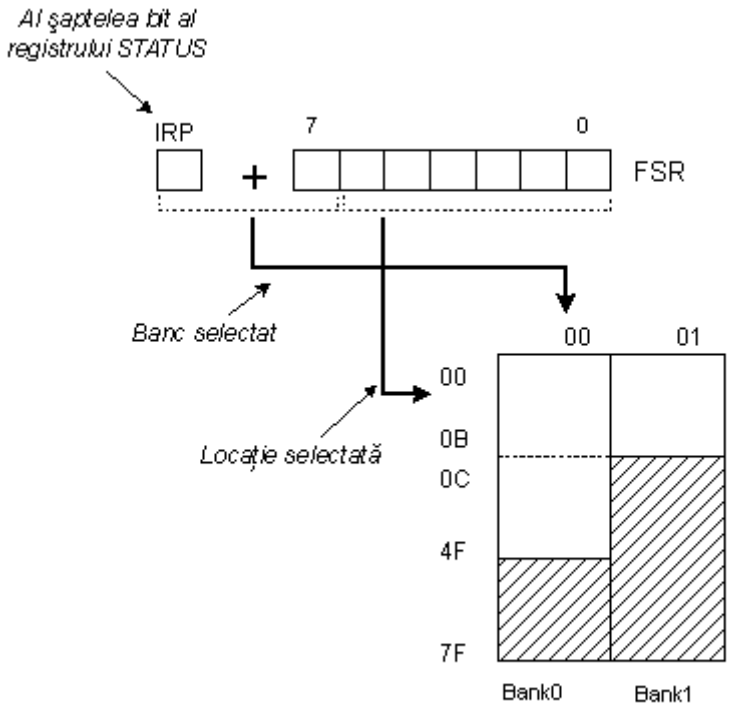
Adresarea Directă

Adresarea Directă se face printr-o adresă de 9 biţi. Această adresă este obţinută prin adăugarea la cei şapte biţi ai adresei directe a unei instrucţiuni a doi biţi (RP1, RP0) din registrul STATUS după cum se arată în figura următoare. Orice acces la regiştrii SFR poate fi un exemplu de adresare directă.

```
Bsf STATUS, RP0 ;Bankl
movlw 0xFF      ;w=0xFF
movwf TRISA     ;address of TRISA register is taken from
                ;instruction movwf
```

Adresarea Indirectă

Adresarea indirectă spre deosebire de cea directă nu ia o adresă dintr-o instrucţiune ci o creează cu ajutorul bitului IRP din registrul STATUS şi a registrului FSR. Locaţia adresată este accesată prin registrul INDF care de fapt conţine o adresă indicată de FSR. Cu alte cuvinte, orice instrucţiune care foloseşte INDF ca registrul al ei, în realitate accesează datele indicate de registrul FSR. Să spunem, de exemplu, că un registru cu scop general (GPR) la adresa 0Fh conţine o valoarea 20. Prin scrierea unei valori 0Fh în registrul FSR vom obţine un registru indicator la adresa 0Fh, iar prin citirea din registrul INDF, vom obţine valoarea 20, ceea ce înseamnă că am citit din primul registru valoarea lui fără accesarea lui directă (dar prin FSR şi INDF). Se pare că acest tip de adresare nu are nici un avantaj faţă de adresarea directă, dar există unele nevoi în timpul programării ce se pot rezolva mai simplu doar prin adresarea indirectă.



Adresarea Indirectă

Un asemenea exemplu poate reprezenta trimiterea unui set de date prin comunicaţia serială, lucrând cu bufere şi indicatoare (ce vor fi discutate în continuare într-un capitol cu exemple), sau, un alt exemplu este şteargerea unei părţi a memoriei RAM (16 locaţii) ca în următorul exemplu.

```
       Movlw 0x0C        ;initialization of starting address
       Movwf FSR         ;FSR indicates address 0x0C
LOOP   clrf INDF         ;INDF = 0
       incf FSR          ;address = initial address + 1
       btfss FSR,4       ;are all locations erased
       goto loop         ;no, go through a loop again
CONTINUE
       :                 ; yes, continue with program
```

Ştergând datele din registrul INDF se scrie în memorie la adresa dată de registrul FSR valoarea zero ce reprezintă operaţia NOP (no operation- nu se face nimic).

4.2. Porturile microcontrolerului

Unii pini ai porturilor I/O sunt multiplexati şi au o funcţie alternativă asociată cu unul din perifericele dispozitivului. În general, când perifericul este activat, aceşti pini nu mai pot fi folosiţi ca pini I/O de uz general ci pentru funcţia destinată la perifericul ce este activat.

Registrul PORTA (ADRESA 05h)

Registrul PORTA este registrul buffer de ieşire a portului A. Atunci când se citeşte furnizează starea pinilor portului dacă aceştia au fost configuraţi ca intrări (în registrul TRISA) iar când se scrie, se scrie în registrul buffer PORTA care se transmite la pini dacă aceştia au fost configuraţi ca ieşiri (în registrul TRISA). Operaţia de scriere în registrul PORTA este o operaţie read-modify-write, adică se citesc pinii portului, se modifică valorile şi se scrie în latch port.

-	-	-	RA4/T0CKI	RA3	RA2	RA1	RA0
Bit 7	Bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

Registrul TRISA (ADRESA: 85h)

Registrul TRISA este destinat pentru stabilirea direcţiei pinilor portului A. Dacă se scrie 0 într-un bit al acestui registru, pinul corespunzător este setat ca ieşire iar dacă se scrie 1 este setat ca intrare.

-	-	-	TRISA4	TRISA3	TRISA2	TRISA1	TRISA0
Bit 7	Bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

Exemplu pentru iniţializarea portului A:

```
BCF         STATUS, RP0 ;
CLRF        PORTA ;Initialize PORTA bz clearing output data latches
BSF         STATUS, RP0 ;Select Bank 1
MOWLW 0X0F              ;Value used to initialize data direction
MOWWF TRISA             ;Set RA<3>0> as imputs RA4 as output
                        ;TRISA<7:5> are always read as '0'
```

Registrul PORTB (ADRESA 06h)

Registrul PORTB este registrul buffer de ieşire a portului B. Atunci când se citeşte, furnizează starea pinilor portului dacă aceştia au fost configuraţi ca intrări (în registrul TRISB) iar când se scrie, se scrie în registrul buffer PORTB care se transmite la pini dacă aceştia au fost configuraţi ca ieşiri (în registrul TRISB). Operaţia de scriere în registrul PORTB este o operaţie read-modify-write, adică se citesc pinii portului, se modifică valorile şi se scrie în latch port.

Fiecare pin al portului B are o sarcină internă cuplată la sursa de alimentare. Cu ajutorul unui singur bit de control (bitul /RBPU - bitul 7 din registrul OPTION) sarcinile pot fi conectate dacă acest bit este şters. Sarcinile sunt decuplate automat dacă pinii portului sunt configuraţi ca ieşire. De asemenea sarcina este decuplată la resetul de alimentare al dispozitivului (Power-On Reset).

Patru din pinii portului B, RB7:RB4 au asociată o întrerupere la schimbare (vezi registrul INTCON). Numai pinii configuraţi ca intrări permit această întrerupere.

O astfel de întrerupere poate scoate dispozitivul din modul SLEEP. Utilizatorul, în subprogramul de servire a întreruperii poate şterge întreruperea în modul următor:

a) orice scriere a PORTB. Această acţiune va încheia condiţia de schimbare;
b) ştergerea fanionului RBIF.

O condiție de schimbare va continua să seteze fanionul RBIF. Citind PORTB se va încheia condiția de schimbare şi fanionul RBIF va fi şters.

RB7(1)	RB6(2)	RB5	RB4	RB3	RB2	RB1	RB0/INT
Bit 7	Bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

(1) Serial programming data
(2) Serial programming clok

Registrul TRISB (ADRESA: 86h)

Registrul TRISB este destinat pentru stabilirea direcției pinilor portului B. Dacă se scrie 0 într-un bit al acestui registru, pinul corespunzător este setat ca ieşire iar dacă se scrie 1 este setat ca intrare.

TRISB7	TRISB6	TRISB5	TRISB4	TRISB3	TRISB2	TRISB1	TRISB0
Bit 7	Bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

Exemplu, inițializarea portului B:

```
BCF         STATUS, RP0 ;
CLRF        PORTB       ;Initialize  PORTB  bz  clearing  output  data
latches
BSF         STATUS, RP0 ;Select Bank 1
MOVLW 0xCF              ;Value used to initialize data direction
MOPVWF      TRISB       ;Set RB<3:0> as inputs, RB<5:4> as outputs
                        ;RB<7:6> as inputs
```

4.3. Setul de instrucțiuni a unităților centrale de tip RISC PIC12, PIC16 şi PIC18

Fiecare instrucțiune reprezintă un cuvânt de 14 biți compus dintr-un cod de instrucțiune (OPCODE) care specifică tipul de instrucțiune şi unul sau mai mulți operanzi. Setul de instrucțiuni al microcontrolerelor PIC este prezentat în tabelul 4.1 care cuprinde instrucțiunile orientate pe octet (byte-oriented), pe bit (bit-oriented) şi operațiile bazate pe numere şi de control (literal and control operations).

Pentru instrucțiunile orientate pe octet 'f' reprezintă simbolizarea unui registru (a file register) folosit de instrucțiune iar 'd' este folosit pentru a simboliza destinația. Simbolurile de registru vor arăta care din regiştrii sunt folosiți de instrucțiune.

Simbolul pentru destinație arată unde se plasează rezultatul operației. Dacă 'd' este zero atunci rezultatul este plasat în registrul W. Dacă 'd' este unu atunci rezultatul este plasat în registrul specificat de instrucțiune.

La instrucțiunile orientate pe bit, 'b' reprezintă simbolul unui bit care indică bitul afectat de operație pe cînd 'f' simbolizează adresa registrului unde este localizat bitul. Pentru opeații cu constante (literal = număr) şi de control 'k' reprezintă o constantă sau un număr pe 8 sau 11 biți.

TABELUL 4.1. Setul de instrucțiuni.

Mnemonic, operands		Description
BYTE-ORIENTED FILE REGISTER OPERATIONS		
ADDWF	f, d	Add W and f
ANDWF	f, d	AND W with f
CLRF	f	Clear f
CLRW	-	Clear W
COMF	f, d	Complement f
DECF	f, d	Decrement f
DECFSZ	f, d	Decrement f, Skip if 0
INCF	f, d	Increment f
INCFSZ	f, d	Increment f, Skip if 0
IORWF	f, d	Inclusive OR W with f
MOVF	f, d	Move f
MOVWF	f	Move W to f
NOP	-	No Operation
RLF	f, d	Rotate Left f through Carry
RRF	f, d	Rotate Right f through Carry
SUBWF	f, d	Subtract W from f
SWAPF	f, d	Swap nibbles in f
XORWF	f, d	Exclusive OR W with f
BIT-ORIENTED FILE REGISTER OPERATIONS		
BCF	f, b	Bit Clear f
BSF	f, b	Bit Set f
BTFSC	f, b	Bit Test f, Skip if Clear
BTFSS	f, b	Bit Test f, Skip if Set
LITERAL AND CONTROL OPERATIONS		
ADDLW	k	Add literal and W
ANDLW	k	AND literal with W
CALL	k	Call subroutine
CLRWDT	-	Clear Watchdog Timer
GOTO	k	Go to address
IORLW	k	Inclusive OR literal with W
MOVLW	k	Move literal to W
RETFIE	-	Return from interrupt
RETLW	k	Return with literal in W
RETURN	-	Return from Subroutine
SLEEP	-	Go into standby mode
SUBLW	k	Subtract W from literal
XORLW	k	Exclusive OR literal with W

ADDLW Add Literal and W
Sintaxa: [etichetă] ADDLW k
Operand: $0 \leq k \leq 255$
Operația: $(W) + k \rightarrow (W)$
Bistabili de stare modificați: C, DC, Z
Descriere: la conținutul registrului W este adunată valoarea pe opt biți (literalul) *k* iar rezultatul este plasat în registrul W.

ADDWF Add W and f
Sintaxa: [etichetă] ADDWF f,d
Operanzi: $0 \le f \le 127$
$d \in [0,1]$
Operația: (W) + (f) → (destinație)
Bistabili de stare modificați: C, DC, Z
Descriere: se adună conținutul registrului W cu registrul *f*. Dacă *d* este *0*, rezultatul este stocat în registrul W iar dacă *d* este *1* rezultatul este stocat în registrul *f*

ANDLW AND Literal with W
Sintaxa: [etichetă] ANDLW k
Operand: $0 \le k \le 255$
Operația: (W) .AND. (k) → (W)
Bistabili de stare modificați: Z
Descriere: se efectuează operația ŞI între conținutul registrului cu valoarea numerică pe opt biți (literalul) *k*. Rezultatul este plasat în registrul W.

ANDWF AND W with f
Sintaxa: [etichetă] ANDWF f,d
Operanzi: $0 \le f \le 127$
$d \in [0,1]$
Operația: (W) .AND. (f) → (destinație)
Bistabili de stare modificați: Z
Descriere: se efectuează operația ŞI între registrul W şi registrul *f*. Dacă *d* este *0* atunci rezultatul este stocat în registrul W iar dacă *d* este *1* rezultatul este stocat în registrul *f*.

BCF Bit Clear f
Sintaxa: [etichetă] BCF f,b
Operanzi: $0 \le f \le 127$
$0 \le b \le 7$
Operația: 0 → (f<b>)
Bistabili de stare modificați: nici unul.
Descriere: bitul *b* din registrul *f* este şters.

BSF Bit Set f
Sintaxa: [etichetă] BSF f,b
Operanzi: $0 \le f \le 127$
$0 \le b \le 7$
Operația: 1 → (f<b>)
Bistabili de stare modificați: nici unul
Descriere: bitul *b* din registrul *f* este setat.

BTFSS Bit Test f, Skip if Set
Sintaxa: [etichetă] BTFSS f,b
Operanzi: $0 \le f \le 127$
$0 \le b < 7$
Operația: sare dacă (f<b>) = 1
Bistabili de stare modificați: nici unul.
Descriere: dacă bitul *b* din registrul *f* este zero, se execută instrucțiunea următoare. Dacă bitul *b* din registrul *f* este unu atunci instrucțiunea următoare este sărită şi se execută o instrucțiune NOP obținându-se o instrucțiune de doi ciclii (2TCY).

BTFSC Bit Test, Skip if Clear
Sintaxa: [etichetă] BTFSC f,b
Operanzi: $0 \le f \le 127$
$0 \le b \le 7$

Operația: sare dacă (f<b>) = 0
Bistabili de stare modificați: nici unul.
Descriere: dacă bitul *b* din registrul *f* este unu, se execută instrucțiunea următoare. Dacă bitul *b* din registrul *f* este zero atunci instrucțiunea următoare este sărită şi se execută o instrucțiune NOP obținându-se o instrucțiune de doi ciclii (2TCY).

CALL Call Subroutine
Sintaxa: [etichetă] CALL k
Operand: $0 \leq k \leq 2047$
Operația: (PC)+ 1→ TOS,
k → PC<10:0>,
(PCLATH<4:3>) → PC<12:11>
Bistabili de stare modificați: nici unul.
Descriere: apelează o subrutină. Mai întâi adresa de reîntoarcere (PC+1) este salvată în stivă. Cei unsprezece biți ai adresării imediate sunt încărcați în biții PC <10:0>. Biții cei mai semnificativi ai registrului PC sunt încărcați din PCLATH. Instrucțiunea CALL este o instrucțiune de două cicluri.

CLRF Clear f
Sintaxa: [etichetă] CLRF f
Operand: $0 \leq f \leq 127$
Operația: 00h → (f)
1 → Z
Bistabili de stare modificați: Z
Descriere: Conținutul registrului *f* este şters iar bistabilul Z este setat.

CLRW Clear W
Sintaxa: [etichetă] CLRW
Operand: nici unul.
Operația: 00h → (W)
1 → Z
Bistabili de stare modificați: Z
Descriere: regitrul W este şters iar bistabilul Z este setat.

CLRWDT Clear Watchdog Timer
Sintaxa: [etichetă] CLRWDT
Operand: nici unul
Operația: 00h → WDT
0 → WDT prescaler,
1 → TO
1 → PD
Bistabili de stare modificați: TO, PD
Descriere: instrucțiunea CLRWDT resetează Watchdog Timer. De asemenea este resetat prescaler-ul WDT. Bistabilii de stare TO şi PD sunt setați.

COMF Complement f
Sintaxa: [etichetă] COMF f,d
Operanzi: $0 \leq f \leq 127$
$d \in [0,1]$
Operația: (f) → (destinație)
Bistabili de stare modificați: Z
Descriere: conținutul registrului *f* este complementat. Dacă *d* este *0* atunci rezultatul este stocat în registrul W iar dacă *d* este *1* rezultatul este stocat în registrul *f*.

DECF Decrement f
Sintaxa: [etichetă] DECF f,d

Operanzi: $0 \le f \le 127$
$d \in [0,1]$
Operația: (f) - 1 → (destinație)
Bistabili de stare afectați: Z
Destinație: Decrementează registrul *f*. Dacă *d* este *0* atunci rezultatul este stocat în registrul W iar dacă *d* este *1* rezultatul este stocat în registrul *f*.

DECFSZ Decrement f, Skip if 0
Sintaxa: [etichetă] DECFSZ f,d
Operanzi: $0 \le f \le 127$
$d \in [0,1]$
Operația: (f) - 1 → (destinație);
sare dacă rezultatul = 0
Bistabili de stare afectați: nici unul
Descriere: Conținutul registrului *f* este decrementat. Dacă *d* este *0* atunci rezultatul este stocat în registrul W iar dacă *d* este *1* rezultatul este stocat în registrul *f*. Dacă rezultatul este *1*, instrucțiunea următoare este executată. Dacă rezultatul este *0* atunci se execută o instrucțiune NOP obținâdu-se o instrucțiune de două cicluri (2TCY).

GOTO Unconditional Branch
Sintaxa: [etichetă] GOTO k
Operands: $0 \le k \le 2047$
Operația: k → PC<10:0>
PCLATH<4:3> → PC<12:11>
Bistabili de stare afectați: nici unul
Descriere: instrucțiunea GOTO este o instrucțiune de salt necondiționat. Valoarea imediată de unsprezece biți este incărcată în biții PC <10:0>. Cei mai semnificativi biți ai registrului PC sunt încărcați din PCLATH<4:3>. GOTO este o instrucțiune de două cicluri.

INCF Increment f
Sintaxa: [etichetă] INCF f,d
Operands: $0 \le f \le 127$
$d \in [0,1]$
Operația: (f) + 1 → (destinație)
Bistabili de stare afectați: Z
Descriere: conținutul registrului *f* este incrementat. Dacă *d* este *0* atunci rezultatul este stocat în registrul W iar dacă *d* este *1* rezultatul este stocat în registrul *f*.

INCFSZ Increment f, Skip if 0
Sintaxa: [etichetă] INCFSZ f,d
Operands: $0 \le f \le 127$
$d \in [0,1]$
Operația: (f) + 1 → (destinație),
sare dacă rezultatul = 0
Bistabili de stare afectați: Nici unul
Descriere: Conținutul registrului *f* este incrementat. Dacă *d* este *0* atunci rezultatul este stocat în registrul W iar dacă *d* este *1* rezultatul este stocat în registrul *f*. Dacă rezultatul este *1* atunci se execută instrucțiunea următoare. Dacă rezultatul este *0* atunci se execută o instrucțiune NOP obținându-se o instrucțiune pe două cicluri (2TCY).

IORLW Inclusive OR Literal with W
Sintaxa: [etichetă] IORLW k
Operand: $0 \le k \le 255$
Operația: (W) .OR. k → (W)
Bistabili de stare afectați: Z

Descriere: se realizează operația OR între conținutul registrului W şi valoarea pe opt biți (literalul) *k*. Rezultatul este plasat în registrul W.

IORWF Inclusive OR W with f
Sintaxa: [etichetă] IORWF f,d
Operands: $0 \leq f \leq 127$
$d \in [0,1]$
Operația: (W) .OR. (f) $\rightarrow$ (destination)
Bistabili de stare afectați: Z
Descriere: se realizează operația SAU între registrul W cu registrul *f*. Dacă *d* este *0* atunci rezultatul este stocat în registrul W iar dacă *d* este *1* rezultatul este stocat în registrul *f*.

MOVF Move f
Sintaxa: [etichetă] MOVF f,d
Operanzi: $0 \leq f \leq 127$
$d \in [0,1]$
Operația: (f) $\rightarrow$ (destinație)
Bistabili de stare afectați: Z
Descriere: conținutul registrului *f* este trensferat la destinație în funcție de valoarea lui *d*. Dacă d = 0 atunci destinația este registrul W iar dacă d = 1 atunci destinația este însuşi registrul *f*. Situația în care d = 1 este utilă la testarea registrului *f* când în urma operației se poziționează bistabilul Z.

MOVLW Move Literal to W
Sintaxa: [etichetă] MOVLW k
Operand: $0 \leq k \leq 255$
Operația: k $\rightarrow$ (W)
Bistabili de stare afectați: nici unul
Descriere: valoarea pe opt biți este încărcată în registrul *f*.

MOVWF Move W to f
Sintaxa: [etichetă] MOVWF f
Operand: $0 \leq f \leq 127$
Operația: (W) $\rightarrow$ (f)
Bistabili de stare afectați: nici unul
Descriere: conținutul registrului W este transferat în registrul *f*.

NOP No Operation
Sintaxa: [etichetă] NOP
Operand: nici unul
Operația: nici o operație.
Bistabili de stare afectați: nici unul
Descriere: nu se efectuează nici o operație.

RETFIE Return from Interrupt
Sintaxa: [etichetă] RETFIE
Operand: Nici unul
Operația: TOS $\rightarrow$ PC,
1 $\rightarrow$ GIE
Bistabili de stare afectați: nici unul

RETLW Return with Literal in W
Sintaxa: [etichetă] RETLW k
Operand: $0 \leq k \leq 255$
Operația: k $\rightarrow$ (W);
TOS $\rightarrow$ PC

Bistabili de stare afectați: nici unul
Descriere: registrul W este încărcat cu valoarea pe opt biți (literalul) *k*. Contorul de program este încărcat cu valoarea din vârful stivei (adresa de reîntoarcere). Inctrucțiunea durează două cicluri.

RETURN Return from Subroutine
Sintaxa: [etichetă] RETURN
Operand: Nici unul
Operația: TOS → PC
Bistabili de stare afectați: nici unul
Descriere: reîntoarcere din subprogram. Vârful stivei (TOS – Top of Stack) este încărcat în registrul contor de program PC. Inctrucțiunea durează două cicluri.

RLF Rotate Left f through Carry
Sintaxa: [etichetă] RLF f,d
Operanzi: $0 \le f \le 127$
$d \in [0,1]$
Operația: vezi descrierea.
Bistabili de stare afectați: C
Descriere: conținutul registrului *f* este rotit spre stânga un bit prin bistabilul Carry. Dacă *d* este *0* atunci rezultatul este stocat în registrul W iar dacă *d* este *1* rezultatul este stocat în registrul *f*.

RRF Rotate Right f through Carry
Sintaxa: [etichetă] RRF f,d
Operands: $0 \le f \le 127$
$d \in [0,1]$
Operația: vezi descrierea.
Bistabili de stare afectați: C
Descriere: conținutul registrului *f* este rotit spre dreapta un bit prin bistabilul Carry. Dacă *d* este *0* atunci rezultatul este stocat în registrul W iar dacă *d* este *1* rezultatul este stocat în registrul *f*.

SLEEP
Sintaxa: [etichetă] SLEEP
Operand: nici unul
Operația: 00h → WDT,
0 → WDT prescaler,
1 → TO,
0 → PD
Bistabili de stare afectați: TO, PD
Descriere: bitul de stare Power-Down PD este şters. Bitul de stare Time-Out este setat. Watchdog Timer şi prescaler-ul sunt şterşi. Procesorul este pus în modul SLEEP cu oscilatorul oprit.

SUBLW Subtract W from Literal
Sintaxa: [etichetă] SUBLW k
Operand: $0 \le k \le 255$
Operația: k - (W) → (W)
Bistabili de stare afectați: C, DC, Z
Descriere: conținutul registrului W este scăzut (prin metoda complementului față de doi) din valoarea de opt biți (literalul) *k*. Rezultatul este pus în registrul W.

SUBWF Subtract W from f
Sintaxa: [etichetă] SUBWF f,d
Operanzi: $0 \le f \le 127$
$d \in [0,1]$

Operația: (f) - (W) → (destinație)
Bistabili de stare afectați: C, DC, Z
Descriere: conținutul registrului W este scăzut (prin metoda complementului față de doi) din valoarea registrului *f*. Dacă *d* este *0* atunci rezultatul este stocat în registrul W iar dacă *d* este *1* rezultatul este stocat în registrul *f*.
Bistabilul C este 1 cind rezultatul este pozitivsau zero si zero cind rezultatul este negativ.

SWAPF Swap Nibbles in f
Sintaxa: [etichetă] SWAPF f,d
Operanzi: $0 \le f \le 127$
$d \in [0,1]$
Operația: (f<3:0>) → (destinație<7:4>),
(f<7:4>) → (destinație<3:0>)
Bistabili de stare afectați: nici unul
Descriere: cei mai semnificativi patru biți ai registrului *f* sunt schimbați cu cei mai puțin semnificativi patru biți (nibble) ai registrului *f*. Dacă *d* este *0* atunci rezultatul este stocat în registrul W iar dacă *d* este *1* rezultatul este stocat în registrul *f*.

XORLW Exclusive OR Literal with W
Sintaxa: [etichetă] XORLW k
Operand: $0 \le k \le 255$
Operația: (W) .XOR. k → (W)
Bistabili de stare afectați: Z
Descriere: se execută operația XOR (SAU EXCLUSIV) între valoarea registrului W şi valoarea pe opt biți (literalul) *k*. Rezultatul este plasat în registrul W.

XORWF Exclusive OR W with f
Sintaxa: [etichetă] XORWF f,d
Operanzi: $0 \le f \le 127$
$d \in [0,1]$
Operația: (W) .XOR. (f) → (destinație)
Bistabili de stare afectați: Z
Descriere: se execută operația XOR (SAU EXCLUSIV) între valoarea registrului W şi valoarea registrului *f*. Dacă *d* este *0* atunci rezultatul este stocat în registrul W iar dacă *d* este *1* rezultatul este stocat în registrul *f*.

Notă: pentru a menține compatibilitatea cu produsele PIC16CXX viitoare nu trebuie folosite instrucțiunile OPTION şi TRIS.

4.4. Exemple de programme în limbaj de asamblare

În acest paragraf se vor prezenta citeva programme simple, scrise în limbaj de asamblare, pentru microcontrolerele PIC12, PIC16 sau PIC18.

4.4.1. Inițializarea unei zone de memorie RAM

Un exemplu de adresare indirectă folosită la şteargerea unei părți a memoriei RAM (16 locații) ca în următorul exemplu.

```
      Movlw OxOC        ; initialization of starting address
      Hovwf FSR         ; FSR indicates address OxOC
LOOP  clrf  INDF        ; INDF = 0
      Incf  FSR         ; address = initial address + 1
      Btfss FSR,4       ; are all locations erased
```

```
      Goto  loop        ; no, go through a loop again
CONTINUE
      :                 ; yes, continue with program
```

Ştergând datele din registrul INDF se scrie în memorie la adresa dată de registrul FSR valoarea zero ce reprezintă operaţia NOP (no operation- nu se execută nimic).

4.4.2. Salvarea şi restaurarea regiştrilor (echivalentul instrucţiunilor PUSH şi POP)

Datorită simplităţii şi folosirii frecvente, aceste părţi ale programului pot fi făcute ca macro-uri. În următorul exemplu, conţinuturile regiştrilor W şi STATUS sunt memorate în variabilele W_TEMP şi STATUS_TEMP înainte de rutina de întrerupere. La începutul rutinei PUSH trebuie să verificăm bancul selectat în prezent pentru că W_TEMP and STATUS_TEMP nu se găsesc în bancul 0. Pentru schimbul de date între aceşti regiştri, instrucţiunea SWAPF se foloseşte în loc de MOVF pentru că nu afectează starea biţilor registrului STATUS.

Exemplul este un program asamblor pentru următorii paşi

1. Testarea bancului curent
2. Stocarea registrului W indiferent de bancul curent
3. Stocarea registrul STATUS în bancul 0
4. Executarea rutinei de întrerupere pentru procesul de întrerupere (ISR)
5. Restaurează registrul STATUS
6. Restaurează registrul W

Dacă mai sunt şi alte variabile sau regiştri ce trebuie stocaţi, atunci ei trebuie să fie păstraţi după stocarea registrului STATUS (pasul 3), şi aduşi înapoi înainte ca registrul STATUS să fie restaurat (pasul 5).

Macro-urile realizate, pot fi folosite pentru scrierea de noi macro-uri.

```
push  macro
      movwf W_Temp            ; W_Tem p <- W
      swapf W_Temp,F          ; Swap them
      BANK1                   ; macro for switching to Bankl
      swapf OPTION_REGJ,W     ; W <- OPTION_REG
      movwf Option_Temp       ; Option_Temp <- W
      BANKO                   ; macro for switching to Bank0
      swapf STATUS,W          ; W <- STATUS
      movwf Stat_Temp         ; Stat_Temp <- W
      endm                    ; End of push macro
pop   macro
      swapf Stat_Temp,W       ; W <- Stat_Temp
      movwf STATUS            ; STATUS <- W
      BANK1                   ; Macro for switching to Bankl
      swapf Option_Temp,W     ; W <- Option Temp
      movwf OPTION_REG        ; OPTIOW_REG <- W
      BANKO                   ; Macro for switching to BankO
      swapf W_Temp,W          ; W <- W_Temp
      endm                    ; End of a pop macro
```

4.4.3. Testarea conţinutului unui registru

Se testează dacă o locaţie de memorie este egală cu o anumită valoare.

```
      movf  timp_100us,w;se citeste locaţia de memorie timp_100us in w
      sublw D'10'       ;se testeaza daca valoarea este egala cu 10
      btfsc STATUS, Z
      goto  msecunda    ;este egala cu 10
      goto  MAIN1       ;nu este egala cu 10
```

4.4.4. Conversie binar-ASCII

Se converteşte un număr binar într-un numar ASCII.

```
      movlw b'00000111' ;un numar oarecare intre 0 si 9
      call  TABLE       ;se apeleaza subprogramul de conversie
      movwf r0          ;caracterul ASCII corespunzator
                        ;numarului este salvat in memoria r0
      goto  main

TABLE
      addwf PCL,1 ; Number 0-9
      DT              "0123456789"
;directiva DT (Define Table) genereaza un sir de instructiuni
;retlw k, cite una pentru fiecare termen al sirului
;directiva este folosita la generarea tabelelor de date pentru
;unitatile centrale din familia PIC 12/16
```

4.4.5. Afişarea unui şir pe un display LCD

Se prezintă un program complet de afişare a unui şir pe un display LCD.

```
      list      p=16F877A    ; list directive to define processor
      #include <p16F877A.inc>; processor specific variable definitions

      __CONFIG _CP_OFF & _WDT_OFF & _BODEN_OFF & _PWRTE_ON & _XT_OSC &
_WRT_OFF & _LVP_ON & _DEBUG_OFF & _CPD_OFF

; '__CONFIG' directive is used to embed configuration data within .asm
;file.
; The lables following the directive are located in the respective
;.inc file.
; See respective data sheet for additional information on
;configuration word.

;***** VARIABLE DEFINITIONS
w_temp        EQU     0x70        ; variable used for context saving
status_temp   EQU     0x71        ; variable used for context saving

      CBLOCK      0x20        ; RAM in bank0
;variabile necesare pentru afisare LCD
      BUFFER0,BUFFER1
;variabile pentru temporizari
            XX,YY
      ENDC
```

```
;Resursele folosite:
;PORTB la care se folosesc pinii RB2 la RB7: RB7:RB4 magistrala de
;date la D7:D4 a LCD, RB3 pinul RS a LCD, RB2 pinul enable a LCD,
;deoarece nu se fac citiri din LCD pinul R/W este pus la masa. Acest
;pin poate fi pus la RB1 si RB1 comutat la zero pentru scriere
;in LCD sau in unu pentru citire din LCD. De asemenea D3:D0 a LCD se
;leaga la masa.
;Temporizarea este facuta soft.

;*******************************************************************
            ORG     0x000             ; processor reset vector
            clrf    PCLATH            ; ensure page bits are cleared
            goto    main              ; go to beginning of program


            ORG     0x004        ; interrupt vector location
            movwf   w_temp       ; save off current W register contents
            movf  STATUS,w       ; move status register into W register
            movwf status_temp    ; save off contents of STATUS register


; isr code can go here or be located as a call subroutine elsewhere


            movf    status_temp,w   ; retrieve copy of STATUS register
            movwf STATUS     ; restore pre-isr STATUS register contents
            swapf   w_temp,f
            swapf   w_temp,w      ; restore pre-isr W register contents
            retfie                ; return from interrupt


;PROGRAMUL PRINCIPAL

main
      CALL  INIT_LCD    ; Setup LCD
      MOVLW "T"   ; Setup message
      CALL  WR_LCD_DATA
      MOVLW "e"
      CALL  WR_LCD_DATA
      MOVLW "s"
      CALL  WR_LCD_DATA
      MOVLW "t"
      CALL  WR_LCD_DATA
      MOVLW " "
      CALL  WR_LCD_DATA
      MOVLW "L"
      CALL  WR_LCD_DATA
      MOVLW "C"
      CALL  WR_LCD_DATA
      MOVLW "D"
      CALL  WR_LCD_DATA
main1
      goto main1

;SUBPROGRAMELE
;+++++++++++++++++++++++++++  LCD  +++++++++++++++++++++++++++++++++++
;Subprogramele pentru LCD sunt alcatuite din:
```

```
;Subprogram INIT_LCD
;subprogramul de initializare a LCD
;in acest subprogram se seteaza iesirile PORTB RB7-RB2 ca iesiri iar
;RB1-RB0 ca intrari
;nota: pentru LCD sunt necesare numai iesirile RB7-RB2

;Subprogram FUNCTION_INIT - necesar INIT_LCD
;in acest subprogram se transmite comanda 0x33 = 00110011
;si apoi comanda 0x32 = 00110010 catre LCD
;acest lucru seteaza afisarea cu date pe 4 biti

;Subprogram FUNCTION_SET - necesar INIT_LCD
;seteaza LCD cu date pe 4 biti si 2 rinduri

;Subprogram ENTRY_MODE
;seteaza LCD sa miste cursorul spre dreapta

;Subprogram DISPLAY_CTRL
;seteaza LCD: Display ON, Cursor OFF, pilpiire OFF

;Subprogram WR_LCD_DATA
;subprogramul scrie o data in LCD. Data in registrul w
;transmite succesiv mai intii cei mai semnificativi 4 biti
;si apoi cei mai putini semnificativi 4 biti

;Subprogram WR_LCD_CMD
;subprogramul scrie o comanda in LCD. Comanda in registrul w.
;transmite succesiv mai intii cei mai semnificativi 4 biti
;si apoi cei mai putini semnificativi 4 biti

;Subprogramul WR_LCD_CMD2
;similar cu WR_LCD_CMD dar cu pemporizari mai mari
;se foloseste pentru comenzi ce necesita asteptari mai lungi

;Subprogram MASK_BIT_CMD
;in aceasta subrutina se seteaza RS si E pentru comenzi
;si se trimite cuvintul spre LCD
;de asemenea se comuta E
;Denumirea e improprie pentru ca de fapt nu se mascheaza nimic

;Subprogramul MASK_BIT_DATA
;in aceasta subrutina se seteaza RS si E pentru date
;si se trimite cuvintul spre LCD
;de asemenea se comuta E
;Denumirea e improprie pentru ca de fapt nu se mascheaza nimic

;Subprogramul CLEAR_SCREEN
;sterge afisajul LCD

;Subprogramul RETURN_HOME
;duce cursorul home

;Subprogramul DISPLAY_ON
;seteaza display LCD on

;Subprogramul SET_POSITION
;seteaza pozitia in DDRAM
```

```
;Subprogramul DELAY15U
;intirziere 15,05 microsecunde

;Subprogramul DELAY45U
;intirziere ~45 microsecunde

;Subprogramul DELAY617U
;intirziere 617,2 microsecunde

;Subprogramul DELAY5M
;intirziere ~5 milisecunde

;=====================================================================

#DEFINE     LCD_DATA_BUS      PORTB ; RB7:RB4 = Data bus
#DEFINE     LOW_RS      BCF PORTB,3 ; RB3 = RS pin
#DEFINE     HIGH_RS     BSF PORTB,3
#DEFINE     LOW_E BCF PORTB,2 ; RB2 = enable pin
#DEFINE     HIGH_E      BSF PORTB,2

    CONSTANT      FUNCTION_VALUE  = B'00101000' ; LCD command
;Functie: 4 biti 2 linii
    CONSTANT      ENTRY_VALUE     = B'00000110' ; LCD command
;cursorul se misca spre dreapta
    CONSTANT      DISPLAY_VALUE   = B'00001100' ; LCD command
;Control display: Display ON, Cursor OFF, pilpiire OFF

;=====================================================================
; aceste comenzi sunt preluate din Carte PIC capitolul 6
; comenzile LCD

;
;     CONSTANT    LCDEM8      = b’00110000’     ;mod 8 biti, 1 linie
;     constant          = b'00111000'     ;mod 8 biti, 2 linii
;     CONSTANT    LDCEM4      = b’00100000’     ;mod 4 biti, 1 linie
;     CONSTANT          = b'00101000'     ;mod 4 biti, 2 linii
;     CONSTANT    LCDDZ       = b’10000000’     ;scrie 0 in DDRAM
;trebuie dat inainte adresa din DDRAM unde se scrie

; comenzi standard pentru initializarea LCD

;     CONSTANT    LCD2L = b’00101000’     ;Functie: 4 biti 2 linii
;=============FUNCTION_VALUE din programul folosit aici
;     CONSTANT    LCDCONT     = b’00001100’;Control display: Display ON
                                    ;Cursor OFF, pilpiire OFF
;=============DISPLAY_VALUE din programul folosit aici
;     CONSTANT    LCDSH = b’00101000’     ;Display mode: AutoInc cursor
                                    ;NoDisplayAutoShift
                                    ;se observa ca este acelasi cu
LCD2L
; Comenzi LCD standard

;     CONSTANT    LCDCLR      = b’00000001’     ;sterge display,
                                              ;reseteaza cursorul
;     CONSTANT    LCDCH = b’00000010’     ;cursor home
;     CONSTANT    LCDCR = b’00000110’     ;cursorul se misca spre
                                          ;dreapta
;=============ENTRY_VALUE din programul folosit aici
```

```
;     CONSTANT    LCDCL = b'00000100';cursorul se misca spre stinga
;     CONSTANT    LCDSL = b'00011000';deplasare continut display spre
                                     ;stinga
;     CONSTANT    LCDSR = b'00011100';deplasare continut display spre
                                     ;dreapta
;de fapt aici LCDL1 este adresa de inceput a primei linii iar LCDL2
;este adresa de inceput a celei de-a doua linii. Vezi mai jos.
;     CONSTANT    LCDL1 = b'10000000'     ;selecteaza linia 1
;     CONSTANT    LCDL2 = b'11000000'     ;selecteaza linia 2

;=====================================================================
;Adresa DDRAM
;Afisajul LCD are 8x80 de pixeli si afiseaza 2x16 caractere.
;Memoria DDRAM are 80 de octeti corespunzator pixelilor. 80/5=16
;Caracterele sunt afisate in matrici de 5x7 pixeli sau 5x10.
;Pe doua linii caracterele nu pot fi afisate decit ca 5x7 pixeli.
;(ultimul rind din matrice este pentru cursor)
;Caracterele sunt scrise succesiv in memoria DDRAM
;Adresele memoriei DDRAM sunt intre 80h si A7h pentru prima linie sau
;C0h la E7 pentru a doua linie.
;Structura adresei este: 1, linie (un bit), adresa (6 biti).
;80h = 10000000b - adresa 0 prima linie
;A7h = 10100111b - adresa 39 prima linie
;C0h = 11000000b - adresa 0 a doua linie
;E7h = 11100111b - adresa 39 a doua linie
;in memoria DDRAM se scriu coduri ASCII - de ce sunt atunci 40 de
;adrese pe fiecare linie?
;
;Distinctia dintre coduri ASCII de afisat (date) si comenzi se face de
catre registrul RS
;0 = instructiuni (comenzi)
;1 = date
;

INIT_LCD
;subprogramul de initializare a LCD
;in acest subprogram se seteaza iesirile PORTB RB7-RB2 ca iesiri iar
;RB1-RB0 ca intrari
;nota: pentru LCD sunt necesare numai iesirile RB7-RB2
      BSF   STATUS,RP0  ; Bank 1
      MOVLW B'00000011' ; RB7-RB2 as output
      MOVWF TRISB
      BCF   STATUS,RP0  ; Bank 0
      CALL  DELAY5M     ; Delay > 15 millisecond
      CALL  DELAY5M
      CALL  DELAY5M
      CALL  DELAY5M
      CALL  FUNCTION_INIT     ; 4 bit initialization
      CALL  FUNCTION_SET      ; date pe 4 biti si 2 rinduri
      CALL  ENTRY_MODE  ;cursor spre dreapta
      CALL  DISPLAY_CTRL      ; Display ON, Cursor OFF, pilpiire OFF
      CALL  CLEAR_SCREEN      ; Clear display
      RETURN

FUNCTION_INIT
;in acest subprogram se transmite comanda 0x33 = 00110011
;si apoi comanda 0x32 = 00110010 catre LCD
;acest lucru seteaza afisarea cu date pe 4 biti
```

```
      MOVLW 0x33
      CALL  WR_LCD_CMD2
      MOVLW 0x32
      GOTO  WR_LCD_CMD

FUNCTION_SET
;seteaza LCD cu date pe 4 biti si 2 rinduri
      MOVLW FUNCTION_VALUE
      GOTO  WR_LCD_CMD

ENTRY_MODE
;seteaza LCD sa miste cursorul spre dreapta
      MOVLW ENTRY_VALUE
      GOTO  WR_LCD_CMD

DISPLAY_CTRL
;seteaza LCD: Display ON, Cursor OFF, pilpiire OFF
      MOVLW DISPLAY_VALUE
      GOTO  WR_LCD_CMD

WR_LCD_DATA
;subprogramul scrie o data in LCD. Data in registrul w.
;transmite succesiv mai intii cei mai semnificativi 4 biti
;si apoi cei mai putini semnificativi 4 biti
      MOVWF BUFFER0           ; Write high nibble
      CALL  MASK_BIT_DATA     ; seteaza si RS si E care se
;transmit la LCD pentru data
      HIGH_E
      SWAPF BUFFER0,0   ; Write low nibble
      CALL  MASK_BIT_DATA
      HIGH_E
      CALL  DELAY45U    ; Delay 40 microsecond
      RETURN

WR_LCD_CMD
;subprogramul scrie o comanda in LCD. Comanda in registrul w.
;transmite succesiv mai intii cei mai semnificativi 4 biti
;si apoi cei mai putini semnificativi 4 biti
      MOVWF BUFFER0           ; Write high nibble
      CALL  MASK_BIT_CMD      ; seteaza si RS si E care se
;transmit la LCD pentru comanda
      HIGH_E
      SWAPF BUFFER0,0   ; Write low nibble
      CALL  MASK_BIT_CMD
      CALL  DELAY45U    ; Delay 40 microsecond
      HIGH_E
      RETURN

WR_LCD_CMD2
;similar cu WR_LCD_CMD dar cu pemporizari mai mari
;se foloseste pentru comenzi ce necesita asteptari mai lungi
      MOVWF BUFFER0     ; Write high nibble
      CALL  MASK_BIT_CMD
      CALL  DELAY5M
      HIGH_E
      SWAPF BUFFER0,0   ; Write low nibble
      CALL  MASK_BIT_CMD
      CALL  DELAY45U    ; Delay > 100 microsecond
```

```
      CALL  DELAY45U
      CALL  DELAY45U
      HIGH_E
      RETURN

MASK_BIT_CMD
;in aceasta subrutina se seteaza RS si E pentru comenzi
;si se trimite cuvintul spre LCD
;de asemenea se comuta E
      MOVWF BUFFER1
      LOW_RS
      BCF   BUFFER1,3   ; For RS = 0
      BSF   BUFFER1,2   ; For E  = 1
      MOVFW BUFFER1
      MOVWF LCD_DATA_BUS
      NOP
      LOW_E       ; Enable pulse
      RETURN

MASK_BIT_DATA
;in aceasta subrutina se seteaza RS si E pentru date
;si se trimite cuvintul spre LCD
;de asemenea se comuta E
      MOVWF BUFFER1
      HIGH_RS
      BSF   BUFFER1,3   ; For RS  = 1
      BSF   BUFFER1,2   ; For E   = 1
      MOVFW BUFFER1
      MOVWF LCD_DATA_BUS
      NOP
      LOW_E       ; Enable pulse
      RETURN

CLEAR_SCREEN
;sterge afisajul LCD
      MOVLW B'00000001' ; Clear LCD
      CALL  WR_LCD_CMD
      CALL  DELAY5M
      RETURN

RETURN_HOME
;duce cursorul home
      MOVLW B'00000010'       ; Return home
      CALL  WR_LCD_CMD
      CALL  DELAY5M
      RETURN

DISPLAY_ON
;seteaza display LCD on
      MOVLW B'00001100'       ; Turn-on display
      GOTO  WR_LCD_CMD

SET_POSITION
;seteaza pozitia in DDRAM
      GOTO  WR_LCD_CMD  ; Set DDRAM position
```

```
;+++++++++++++++++ DELAY SUBROUTINES ++++++++++++++++++++++++++++++++
DELAY15U
;intirziere 15,05 microsecunde
      MOVLW .25   ; Tdelay 15.05 microsecond
      MOVWF XX
      DECFSZ      XX,1
      GOTO  $-1
      RETURN

DELAY45U
;intirziere ~45 microsecunde
      MOVLW .74   ; Delay ~45 microsecond
      MOVWF XX
      DECFSZ      XX,1
      GOTO  $-1
      RETURN

DELAY617U
;intirziere 617,2 microsecunde
      CLRF  XX    ; Tdelay = 617.2 microsecond
      MOVLW .4
      MOVWF YY
      DECFSZ      XX,1
      GOTO  $-1
      DECFSZ      YY,1
      GOTO  $-3
      RETURN

DELAY5M
;intirziere ~5 milisecunde
      CLRF  XX    ; Delay ~5 millisecond
      MOVLW .33
      MOVWF YY
      DECFSZ      XX,1
      GOTO  $-1
      DECFSZ      YY,1
      GOTO  $-3
      RETURN


            END                     ; directive 'end of program'
```

BIBLIOGRAFIE

1. Arsinte Radu - Arhitecturi paralele si procesoare de semnal, Editura "Politehnica" Timidoara, 2000;
2. Athanasiu Irina, Panoiu Alexandru, - Microprocesoarele 8086, 286, 386, Editura TEORA, Bucuresti, 1992;
3. Andronescu Gh., - Sisteme Digitale, Editura MatrixRom, Bucuresti, 2002;
4. Baluta Gheorghe, - Circuite logice si structuri numerice. Proiectare si aplicatii. Editura MatrixRom, Bucuresti, 2002;
5. Belega, D., - Placa de dezvoltare TMS320C5X DSK în aplicatii, Editura Politehnica, Timişoara, 2002;
6. Blakeslee Thomas, Proiectarea cu circuite logice MSI si LSI standard, EdituraTehnica, Bucuresti, 1988;
7. Bogdanov Ivan, - Microprocesorul in comanda actionarilor electrice, Editura FACLA, Timisoara, 1989;
8. Borcoci A. si col. - Arhitectura Microprocesoarelor Media publishing 1995;
9. Capatina Octavian, - Proiectarea cu microcalculatoare integrate, Editura Dacia, Cluj, 1992;
10. Cristian Lupu, Stefan Stancescu, - Microprocesoare Circuite Proiectare., Editura Militara, Bucuresti, 1986;
11. Dancea Ioan, - Microprocesoare. Arhitectura interna, programare, aplicatii. Editura Dacia, Cluj-Napoca, 1979;
12. Davidoviciu A., s.a., - Minicalculatoarele si microcalculatoarele in conducerea proceselor industriale, Editura Tehnica, Bucuresti, 1983;
13. Douglas F. Elliott, - Handbook of Digital Signal Processing: Engineering Applications, Academic Press, 1987;
14. Dragu Ion, Iosif Ion-Mihail, - Prelucrarea numerica a semnalelor discrete in timp, Editura Militara, Bucuresti, 1985;
15. Driscoll F.F., Coughlin R.F., Villanucci R.S. - Data Acquisition and Process Control , Prentice Hall, 2000;
16. Dumortier Dominique, - Digital/Analog and Analog/Digital conversion Handbook, Motorola, 1990;
17. Graham., I., King, T. - The Transputer Handbook, Prentice Hall Ltd., 1989;
18. Grover Dale, John R. - Digital Signal Processing and the Microcontroller, Prentice Hall, 1999;
19. IEEE DSP Committee, - Programs for Digital Signal Processing, IEEE Press, 1979;
20. Ionescu D. - Codificare si coduri, Editura Tehnica, Bucuresti, 1981;
21. Lim J. S., Oppenheim A. V., - Advanced Topics in Signal Processing, Pretice Hall, 1988;

22. Lupu C., s.a. - Microprocesoare. Aplicatii. Editura Militara, Bucuresti 1982;
23. Lupu Eugen, Miclea Tiberiu, Arsinte Radu - Procesoare de semnal - Generația TMS320C2X - prezentare si aplicatii – Ediura Promedia, Cluj-Napoca, 1995;
24. Marinescu D. Naicu S., - Microcontrolerul 80C32. Manual de utilizare. Editura Tehnica, Bucuresti, 1998;
25. Marven, C., Ewers, G. *A Simple Approach to Digital Signal Processin*g, Texas Instruments, 1993.
26. Musca Ghe. - Programare in Limbaj de Asamblare Editura TEORA 1997,
27. Nigel P., - Cook A First Course In Digital Electronics, Pretice Hall, New Jersey,1999;
28. Oppenheim A. V., Schafer R. W., - Digital Signal Processing, Prentice-Hall, 1975;
29. Oppenheim A. V., Schafer R. W., - Discrete-Time Signal Processing, Prentice Hall, 1989;
30. Pop Eugen, s.a. - Metode in prelucrarea numerica a semnalelor, Editura FACLA, Timisoara, 1989;
31. Proakis, G., Manolakis, D. G. - *Digital Signal Processing. Principles, Algorithms and Application*s, 3rd Edition, Prentice-Hall, 1996;
32. Puiu-Berizințu M., Rotar Dan – An Optimal Control Method of the PWM Inverter used in Electrical Drives with Induction Motor - MIPRO'99 CONFERENCE, IEEE Region 8, CROAȚIA 1999.
33. Puiu Berizințu Mihai, Rotar Dan – Using DSP for PWM Inverter Command by the Generatrix Wave Sampling Principle, Conferința Națională de Acționări Electrice "CNAE 2000", Iaşi, 12-14 octombrie 2000, publicată în Buletinul Institutului Politehnic Iaşi, Tomul XLVI (L), Fasc. 5, ISSN 0258-9109, pp. 72-77
34. Radu O., Sandulescu Gh., - Filtre numerice. Aplicatii, Editura Tehnica, Bucuresti, 1979;
35. Rotar Dan - Harmonic analysis based on microcomputers, Efficiency, Cost, Optimization, Simulation and Environmental Aspects of Energy Systems and Processes Congress ECOS98, ISBN 2-905-267-29-1, Nancy, France, pp. 1173-1180, 1998.
36. Rotar Dan - Protection of the Microcomputer-based Pulse-Width Modulated Inverters, 17th International Conference on COMPUTERS IN TECHNICAL SYSTEMS, Proceedings Volume 2, ISBN 953-6042-57-6, pp. 67-70, CROAȚIA 1998.
37. Rotar Dan - Microcomputer-based electrical drives command with sound card, Conferința Națională de Acționări Electrice CNAE98, Craiova, 1998, pp. 165-168.
38. Rotar Dan, Ababei Ştefan - Determinarea consumului energetic prin contorizare numerică, Conferința Națională de Energetică Industrială, Bacău, 1998, Editura Plumb, ISBN 973-9362-16-8, pp. 170-173.
39. Rotar Dan – Sisteme de măsură digitale a energiei electrice – Probleme de management şi conservare a energiei, Craiova, ISBN 973-0-00917-1, pp. 21-28, 1999
40. Rotar Dan – Programarea DSP, Conferința Națională de Energetică Industrială CNEI 2000 MILENIUM, 10-11 noiembrie 2000, Bacău, Editura ALMA MATER, ISBN 973-99703-4-6, pp. 84-87
41. Rotar Dan – Regulator numeric pentru procesorul digital de semnal TMS320F240, Conferința Națională de Energetică Industrială CNEI 2000 MILENIUM, 10-11 noiembrie 2000, Bacău, Editura ALMA MATER, ISBN 973-99703-4-6, pp. 88-91

42. Rotar Dan, Ababei Stefan, Sorin Popa, Communication system for dsp and PC compatible computer, Romanian Academy, Branch office of Iasi, MCOM-8, 2002, ISSN 1224-7480, pp. 413-418.
43. Rotar Dan, Ababei Stefan, DSP solution for experimental transistor characteristics plots, Romanian Academy, Branch office of Iasi, MCOM-8, 2002, ISSN 1224-7480, pp. 419-421.
44. Rotar Dan, Petru Livinţi, Ababei Stefan, Digital filtering with digital signal processing controller, Romanian Academy, Branch office, MCOM-9 vol. 2, 2003, ISSN 1224-7480, pp. 207-210.
45. Sheingold Danil H., Analog-digital conversion handbook, Analog Device, 1992;
46. Somnea Dan, Vladut Teodor, - Programarea in assembler. Editura Tehnica, Bucuresti, 1992;
47. Stanasila Octavian, - Notiuni si tehnici de matematica discreta, Editura Stiintifica si Enciclopedica, Bucuresti, 1985;
48. Stanomir D., Stanasila O., - Metode matematice in teoria semnalelor, Editura Tehnica, Bucuresti 1980;
49. Suciu Marcel, Popescu Dumitru, Ionescu Traian, - Microprocesoare, microcalculatoare si roboti in automatizari industriale, Editura Tehnica, Bucuresti, 1986
50. Stearns S. D., - *Digital Signal Analysi*s, Hayden Book, 1975;
51. Sztojanov I., s.a. - De la poarta TTL la microprocesor vol I, II, Editura Tehnica, Bucuresti, 1987;
52. Tanase Ady, Gaitan V., - Familia de procesoare pentru prelucrarea numerica a semnalelor ADSP-21, Editura MatrixRom, Bucuresti, 2004;
53. Toderean, G., s.a. - Transputere si procesoare de semnal, Ed. Microinformatica, Cluj- Napoca, 1993;
54. Thomas L.Floyd - Digital Fundamentals, Pretice Hall, New Jersey,2000;
55. Toacse Ghe., - Introducere in microprocesoare, Editura Stiintifica si Enciclopedica, Bucuresti, 1986;
56. Tocci Ronald J., Neal S. Widmer - Digital Systems, Pretice Hall, New Jersey,1998;
57. Toma L., - Sisteme de achiziþie şi prelucrarea numericã a semnalelor, Editura.de Vest, Timisoara.1996;
58. Zoican Sorin, - Joint Time-Frequency Adaptive Echo Canceller with Digital Signal Processor, Proceedings of MELECON'2000, May 29-31, 2000, Cyprus ., pag. 295-300;
59. Zoican Sorin, - Effect of LMS Decorrelated Algorithm on the Adaptive Filter Performance, International Conference on Telecommunications, ITC2000, May 22-25, 2000, Acapulco, Mexico, pag.195-200 ;
60. Zoican Sorin, Zoican Roxana, - Enhanced Voice Activity Detector Algorithm for Wireless Communication, International Conference Communication2000, Military Technical Academy, december 2000, Bucharest;
61. Wakerly John F., - Digital Design, Pretice Hall, New Jersey, 2000;
62. *** - TMS320C24x DSP Controllers - Reference Set: Vol.1, Texas Instruments Inc, 1997;
63. *** - TMS320C24x DSP Controllers - Reference Set: Vol.2, Texas Instruments Inc, 1997;
64. *** - PIC 16F87x – Users Manual, Microchip, 2002.