

Programare in Limbaje de Asamblare  
CPU: Z80, b-bit

# MICROPROCESOARE

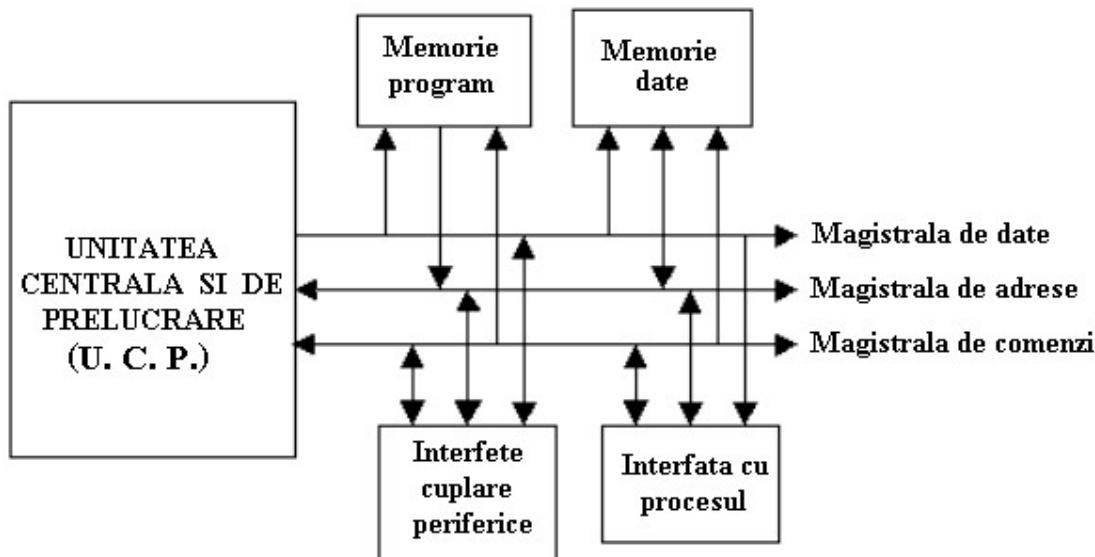
## Prezentarea generală a structurii unui microcalculator

Procesorul reprezintă realizarea cu cel mai mare impact asupra evoluției ulterioare a societăților industrializate. Ușurința implementării diverselor aplicații, flexibilitatea privind conducerea proceselor, modularizarea aplicațiilor, fiabilitatea ridicată, au contribuit la o utilizare pe scară largă a microprocesoarelor și implicit la o scădere a prețului de cost a aplicațiilor respective.

În acest context microprocesorul este dispozitivul electronic care însoțește aproape orice echipament de calcul sau instalație tehnologică. Rafinarea microprocesoarelor a aparut ca o necesitate obiectivă datorită faptului că cerințele impuse sistemelor de calcul au evoluat și ele de asemenea într-o manieră naturală. O scurtă trecere în revistă a procesoarelor dezvăluie dinamica evoluției acestora și creativitatea specialiștilor în acest domeniu. Reorganizarea modului de procesare împreună cu implementarea diferitelor mecanisme de calcul paralel au contribuit la creșterea puterii de prelucrare a procesoarelor.

Piața structurilor de calcul cunoaște o mare diversitate și pune în evidență faptul că specialiștii din acest domeniu sunt într-o continuă căutare de noi soluții care să răspundă provocărilor prezentului și viitorului.

Schema bloc a unui microcalculator realizata în jurul unui microprocesor este dată în cele ce urmează.



**U. C. P.** este componenta care guvernează activitatea întregului sistem executând programe care se află în memoria program (ROM, PROM, EPROM, EEPROM). Prin execuția programului aflat în memoria program U.C.P.-ul procesează date cu scopul îndeplinirii unor cerințe impuse într-un anumit domeniu de activitate.

**Memoria program** este utilizată de regulă pentru stocarea programelor care sunt executate de către U.C.P. În cazul PC-urilor, memoria program (EPROM) conține un program ce

permite încărcarea unui alt program care la rândul său încarcă sistemul de operare. Memoriile program au evoluat de la memorii ROM care sunt scrise de către fabricant o singura dată, până la memorii EEPROM ce permit ștergerea și programarea de mai multe ori pe cale electrică de către utilizator.

**Memoria de date** permite stocarea datelor care se pot modifica în timpul execuției programului. Memoria de date este de două tipuri: memoria RAM – statică care nu necesită refreșarea informației stocate și memorii RAM – dinamice care necesită refreșarea informației la un interval de aproximativ de 2ms.

**Interfețele pentru cuplarea periferice** permit conectarea dispozitivelor externe la U.C.P. în vederea comunicării cu operatorul uman și respectiv a editării unor documente privind anumite mărimi din proces.

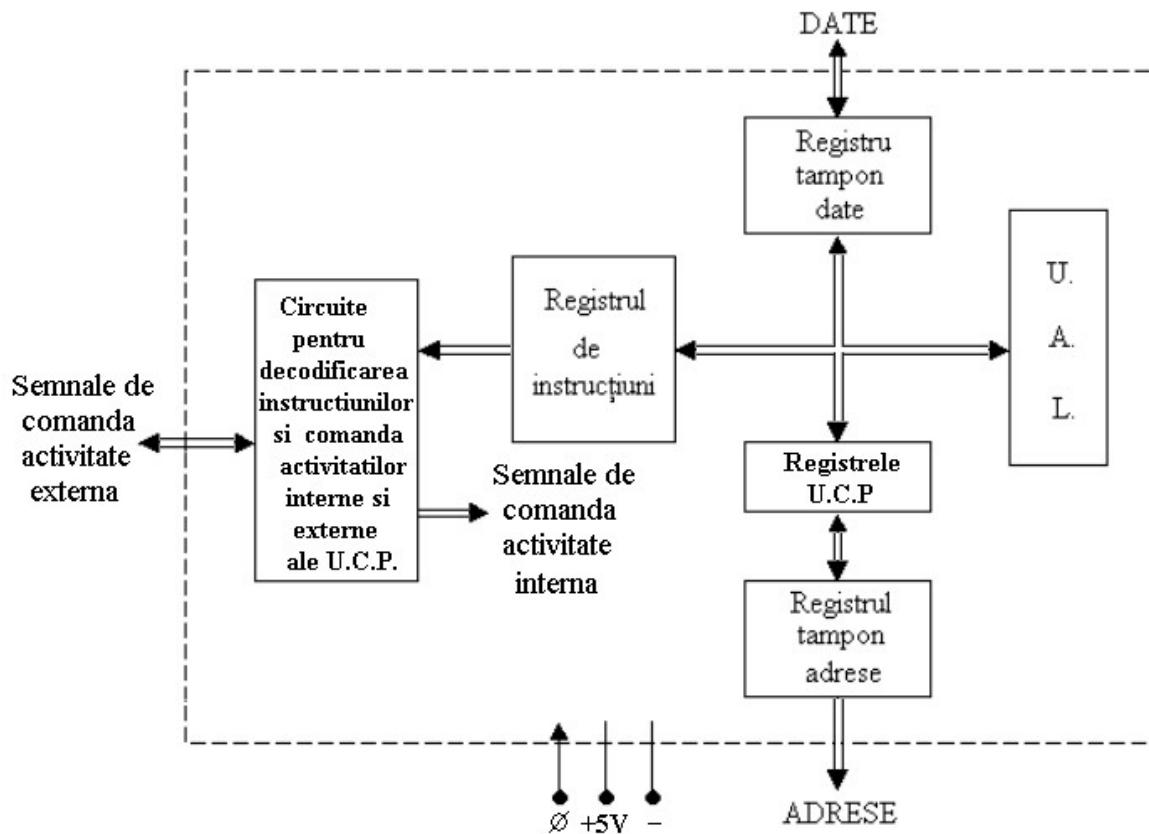
**Interfața de proces** este utilizată pentru conectarea procesului la microcalculator.

**Magistralele microsistemului** permit transferul de informații între dispozitivele electronice ale microcalculatorului.

### Unitatea centrală și de prelucrare a microprocesorului Z80

Micropresorul Z80 reprezintă unul dintre cele mai performante procesoare organizate pe 8 biți, fiind realizat într-o structură unitară și disponând de un set de instrucțiuni evoluat, comparativ cu celealte procesoare pe 8 biți.

Structura internă este dată în figura de mai jos.



**Circuitele pentru decodificarea instrucțiunilor** reprezintă componența de bază a unui microprocesor având rolul de a interpreta (decodifica) instrucțiunile reținute în registrul de instrucțiuni și de a genera acele semnale de comandă (interne și externe) care permit rezolvarea cerinței specificată de către instrucțiunea care se execută la momentul curent.

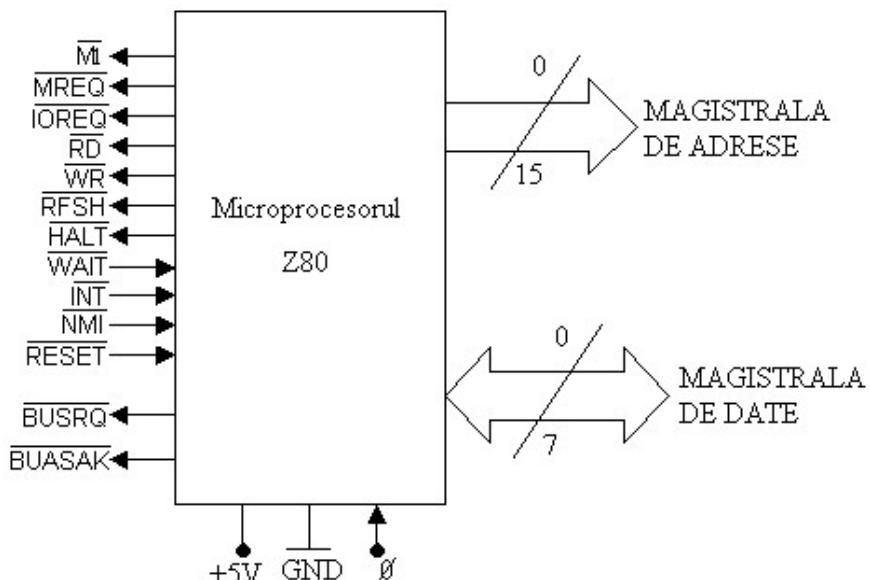
**Registrul de instrucțiuni** menține, un timp bine precizat, codul instrucțiunii în curs de execuție la intrările circuitului pentru decodificarea instrucțiunilor.

**Registrele unității centrale** sunt memorii rapide care au rolul de a mări viteza de execuție a instrucțiunilor și de a crește flexibilitatea în ceea ce privește implementarea algoritmilor. Registrele unității centrale mai conțin și informații cum ar fi adresa de refreshare (registrul R), adresa memoriei stivă (registrul SP), partea high a adresei tabelei unde se află adresele rutinelor de tratare a întreruperilor (registrul I), etc.

**Registrele tampon de date și adrese** au rolul de a menține adresele și datele pe magistralele de adrese și date un interval de timp bine definit.

**Unitatea aritmetico-logică** (U.A.L.) reprezintă o altă componentă importantă a microprocesorului care are rolul de a executa operații aritmetice și logice între operanzi ce se pot afla în registrele unității centrale sau în memoria sistemului.

### Descrierea conexiunilor externe ale microprocesorului Z80



M1 - indică extragerea unui cod de operație numit și OPCOD sau aşa cum este cunoscut și sub denumirea de ciclul FETCH.

**Obs.: M1 + IOREQ** indică un ciclu de achitare a unei întreruperi

MREQ - este activ pe zero logic și indică lucrul cu memoria. Semnalizează faptul că pe magistrala de adrese se află o adresă de citire sau scriere din/în memoria externă.

IOREQ - permite lucrul cu dispozitivele de intrare/ieșire. Indică plasarea pe octetul mai puțin semnificativ al magistralei de adrese a unei adrese care va fi utilizată într-o operație de citire sau scriere la dispozitivele de I-E.

*RD* - un semnal cu ajutorul căruia se realizează citirea informației din memorie sau la un periferic. Se utilizează acest semnal pentru a valida cu ajutorul lui plasarea datelor pe magistrala de date.

*WR* - permite scrierea informației în memorie sau la un periferic de ieșire. Semnalează faptul că pe magistrala de date se găsesc datele ce trebuie scrise în memoria externă sau la dispozitivul periferic adresat.

*RFSH* - semnal cu ajutorul căruia se realizează refresharea memoriei RAM dinamice. Pe perioada când RFSH este activ se activează și MREQ, iar pe liniile A<sub>0</sub> – A<sub>6</sub> se află adresa de reîmprospătare a memoriei RAM dinamică.

*HALT* - indică oprirea U.C.P., timp în care se așteaptă de regulă o întrerupere nemascabilă sau una mascabilă, dar validată în prealabil, pentru a putea ieși din starea HALT.

**Obs.:** în timpul opririi se execută instrucțiuni de tip NOP (No operation) cu scopul de a se refresha RAM-ul dinamic.

*WAIT* - semnal prin care se poate realiza dialogul cu perifericele mai lente prin introducerea unor stări de așteptare. În timpul stărilor de așteptare nu se generează semnale de refresh.

*INT* - cerere de întrerupere mascabilă care este validată numai dacă se execută instrucțiunea EI (Enable Interrupt). Cererea poate fi invalidată executând instrucțiunea DI (Disable Interrupt).

*NMI* - permite întreruperea activității microprocesorului în orice moment de timp (întrerupere nemascabilă). În momentul apariției unei întreruperi nemascabile se suspendă execuția programului și se face saltul la adresa 0066<sub>H</sub>, unde se află rutina de tratare a întreruperii nemascabile.

*RESET* - este un semnal de resetare a microprocesorului care trebuie ținut pe zero logic cel puțin 3 perioade de ceas pentru a reseta toate elementele componente ale microprocesorului. După reset avem următoarea stare: PC (Program Counter) = 0000<sub>H</sub>; IF1=0, IF2=0; I=00<sub>H</sub>, R=00<sub>H</sub>; modul de întrerupere setat este modul zero.

*BUSRQ* - este un semnal de cerere de magistrală trimis de o altă unitate de procesare care este tratat la sfârșitul fiecărui ciclu mașină.

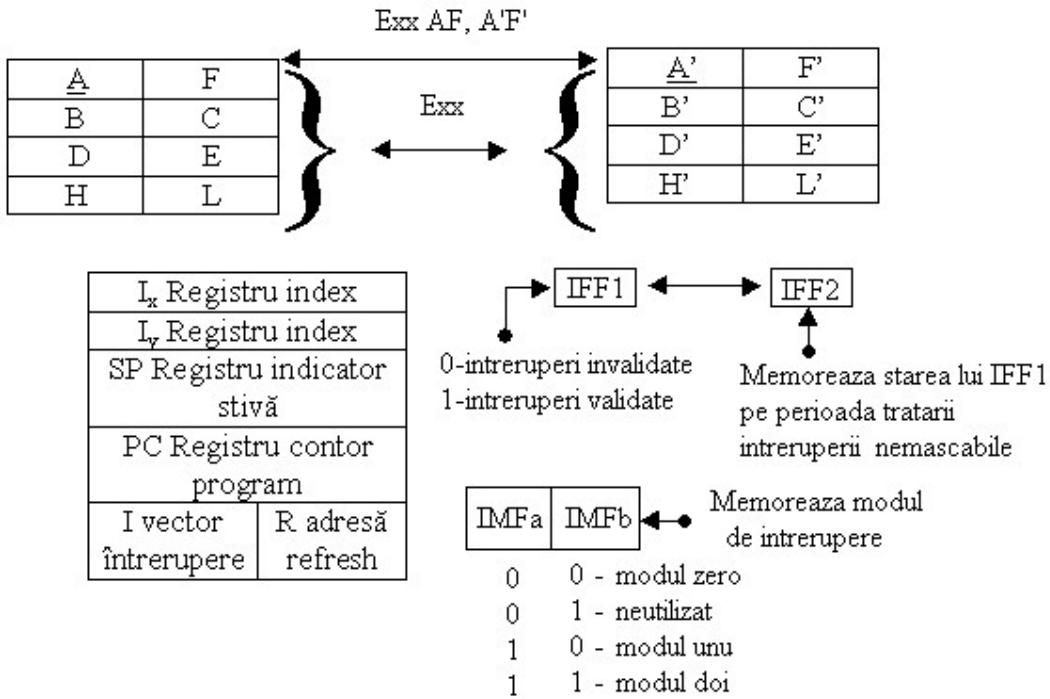
*BUSAK* - este un semnal prin care microprocesorul anunță unitatea de calcul care a cerut magistralele, faptul că acestea pot fi preluate.

$\emptyset$  - clock-ul U.C.P. care permite desfășurarea operațiilor executate de microprocesor. Domeniul clock-ului  $\emptyset$  este [2...10] MHz.

*Magistralele de adrese și de date* permit adresarea și transferul de informații de la și către dispozitivele microsistemu.

## Descrierea registrelor U.C.P.

Unitatea Centrală de Prelucrare cu Z80 conține registre organizate pe 8 sau 16 biți ce permit creșterea flexibilității U.C.P.-ului și a vitezei de lucru a microprocesorului Z80.



**A** - registrul acumulator este un registru des utilizat de către U.A.L. Informația conținută în acesta poate fi atât sursă cât și destinație în cadrul operațiile executate de microprocesor.

**BC, DE, HL** – sunt regitre de câte 16 biți, care pot utiliza și ca regiștre de 8 biți (**B, C, D, E, H, L**). Aceste regitre sunt folosite atât în operațiile logice și aritmetice, cât și în operațiile prin care se realizează transfer de informație de la memorie la microprocesor și invers.

**I<sub>x</sub>, I<sub>y</sub>** - registre de index care permit memorarea adresei de bază în cazul unor instrucțiuni care folosesc adresarea relativă.

**SP** - indicator de stivă care este de fapt un regisztr pe 16 biți ce permite lucrul cu memoria de tip stivă, memorie utilizată în cazul instrucțiunilor de tip CALL sau în cazul tratării intreruperilor. Spațiul rezervat memoriei de tip stivă nu trebuie modificat de către programele în curs de execuție.

**PC** - contor de program este un regisztr pe 16 biți care memorează adresa instrucțiunii în curs de execuție.

**I** - regisztr pe 8 biți care conține cei mai semnificativi 8 biți ai adresei tabelei cu adresele rutinelor de tratare a intreruperilor.

**F** - regisztr care memorează indicatorii de condiție rezultați în urma efectuării operațiilor.

Regisztrul F are următoarea structură:

S	Z	*	H	*	P/V	N	C
---	---	---	---	---	-----	---	---

unde:

**S** - indicator de semn ( $S = 1$  dacă rezultatul operației executate este negativ);

**Z** - indicator de zero ( $Z = 1$  dacă rezultatul unei operații este zero);

**C** - indicator de transport care se setează cu 1, dacă operația a produs un transport la cel mai semnificativ bit al operandului sau rezultatului;

**P/V** - indicator de paritate/depășire (pentru operații logice avem paritate, iar pentru aritmetice avem depășire). În cazul parității dacă  $P/V=1$ , atunci rezultatul are un număr par de biți, iar în cazul depășirii dacă  $P/V=1$ , atunci rezultatul a depășit limitele intervalului  $[-128, 127]$ ;

**H** - indicator de transport auxiliar, fiind utilizat numai de către instrucțiunile de ajustare zecimală;

**N** - indicator de adunare sau scădere care este utilizat de asemenea de către instrucțiunile de ajustare zecimală;

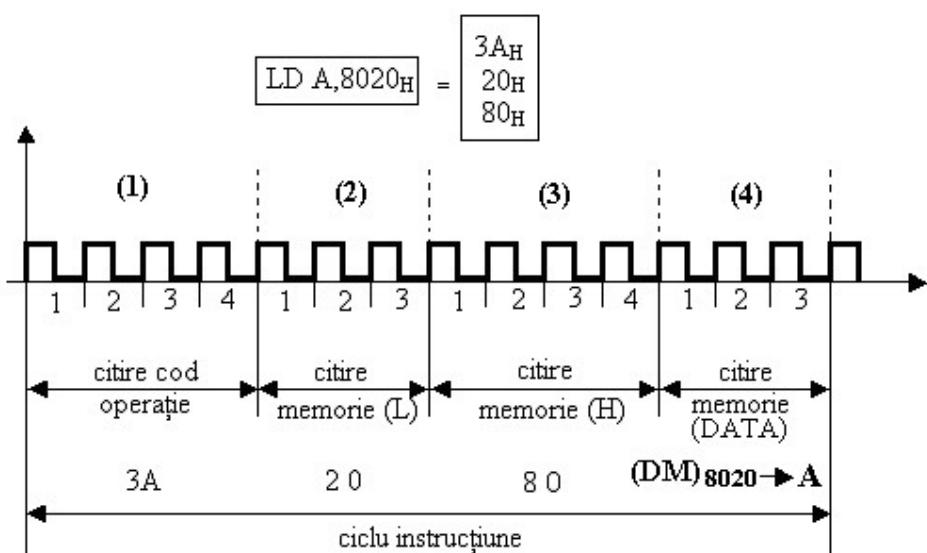
**IF1, IF2** bistabile care permit lucrul cu intreruperile mascabile și nemascabile;

**IMFa, IMFb** - bistabile care memorează unul dintre cele 3 moduri de tratare a intreruperilor.

### Descrierea ciclurilor mașină

Ciclul instrucțiune reprezintă timpul de execuție a unei instrucțiuni și este format la rândul său din cicluri mașină. Un ciclu mașină este o succesiune de mai multe stări  $T$  ( $T_1, T_2, \dots, T_n$ ), unde o stare reprezintă perioada de ceas a microsistemului.

Structura generală a unui ciclu instrucțiune este dată în figura următoare:

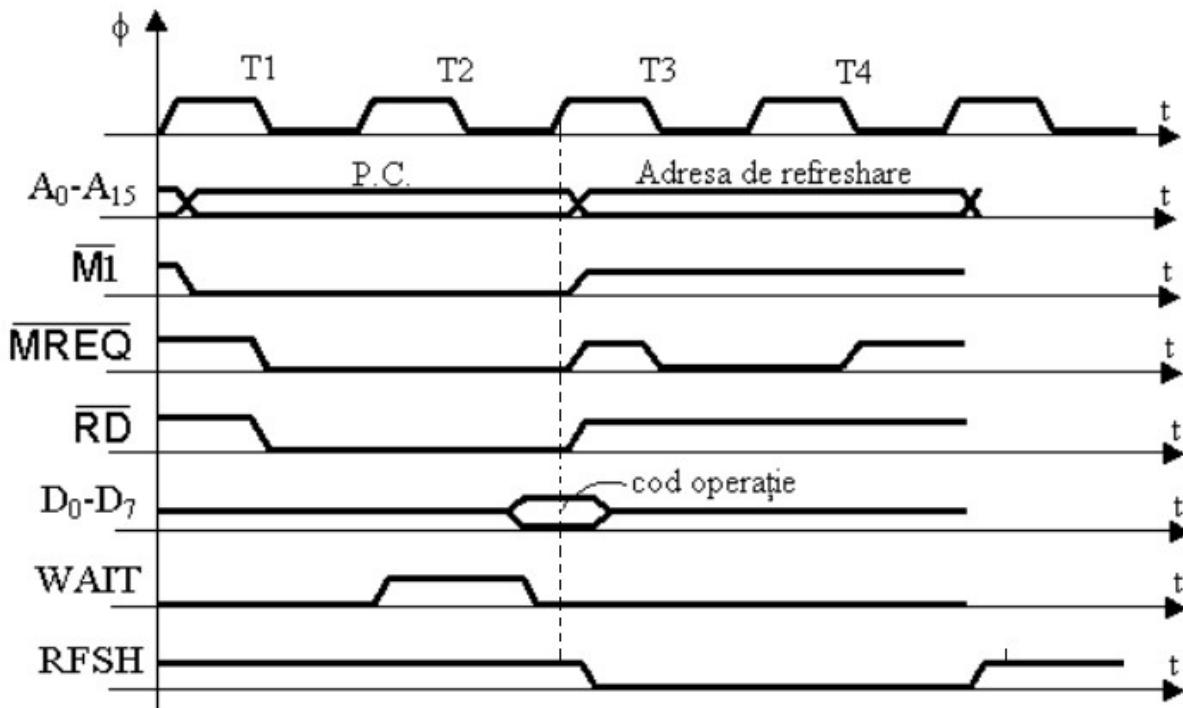


Ciclul instrucțiune prezentat corespunde instrucțiunii  $LD\ A,(8020_H)$  și conține patru cicluri mașină. Primul ciclu este un ciclu *FETCH* (extragere cod operație), urmat de două cicluri mașină necesare extragerii adresei de memorie de unde se va face transferul. Ultimul ciclu corespunde executării proprie-zise a instrucțiunii de transfer a datei din memorie în registrul A.

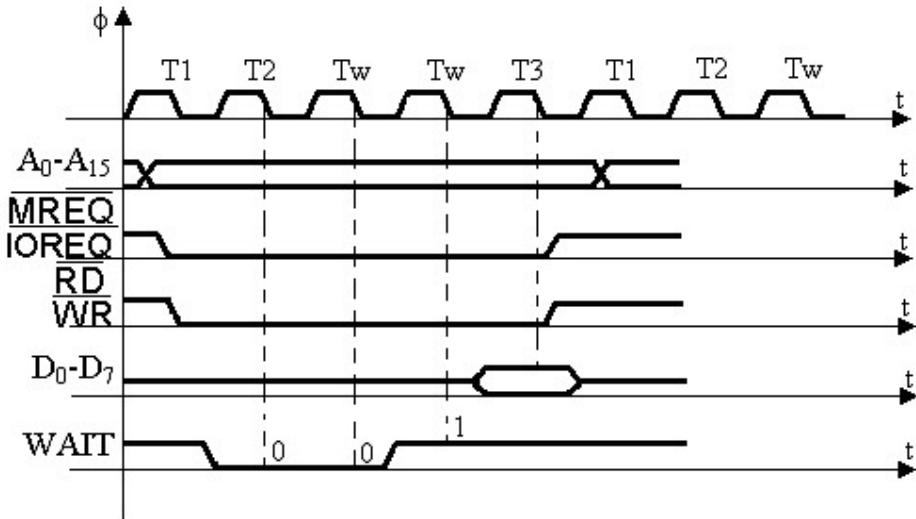
Microprocesorul Z80 dispune de cicluri de citire din memorie (extragere cod operație, citire operand, citire date din memorie) și cicluri de scriere în memorie care se referă la transferul informației de la microprocesor în memoria *RAM* statică sau dinamică a microsistemu. De asemenea Z80 dispune de cicluri de citire/scriere din/în porturi de intrare/ieșire care sunt similare cu ciclurilor de citire/scriere din/în memorie. În cadrul acestor cicluri se activează  $\overline{IOREQ}$  în loc de  $\overline{MREQ}$ .

În cazul comunicării cu dispozitivele mai lente se utilizează tehnica de *WAIT* necesară transferului de date între perifericele cu timp de acces mai mare și microprocesor.

**Ciclul de extragere cod operație.** Pentru extragerea codului operație se plasează pe magistrala de adrese continutul registrului PC, corespunzător instrucțiunii în curs de execuție și se activează semnalul  $\overline{M1}$ , deoarece se extrage un cod de operație. După activarea lui  $\overline{M1}$  se activează  $\overline{MREQ}$  și  $\overline{RD}$  și se reține informația pe frontul crescător al perioadei  $T3$ . După extragerea codului operație urmează refresharea memoriei RAM dinamică pe durata perioadelor de ceas  $T3$  și  $T4$  prin plasarea adresei de refreshare pe magistrala de adrese și activarea semnalelor  $\overline{MREQ}$  și  $\overline{RFSH}$ .



## Cicluri de scriere/citire în/din memorie sau port de ieșire/intrare cu stari WAIT



Pe frontul căzător al perioadei  $T2$  se testează  $WAIT$ , iar în cazul în care este găsit pe 0 logic se mai introduce încă o stare de aşteptare  $Tw$ . Pe frontul căzător al lui  $Tw$  se testează din nou  $WAIT$  până când acesta este pe 1 logic. După ce se trece  $WAIT$  pe 1 logic se continuă desfășurarea normală a ciclului mașină.

Transferul datelor are loc pe frontul căzător al perioadei  $T3$ , moment în care se realizează tranziția semnalelor de citire/scriere din 0 în 1 logic. În funcție de semnalul care se activează ( $\overline{MREQ}$ ,  $\overline{IOREQ}$ ,  $\overline{RD}$ ,  $\overline{WR}$ ) microprocesorul execută una dintre operațiile date în tabelul următor.

$\overline{MREQ}$	$\overline{IOREQ}$	$\overline{RD}$	$\overline{WR}$	Operația
0	1	0	1	Citire memorie
0	1	1	0	Scriere memorie
1	0	0	1	Citire port
1	0	1	0	Scriere port

## Modurile de adresare ale microprocesorului Z80

Microprocesorul Z80 dispune de un set de instrucțiuni mult mai performant decât celelalte microprocesoare de 8 biți. Are în total 158 tipuri de instrucțiuni care includ și cele 78 de instrucțiuni ale microprocesorului INTEL 8080 la nivel de cod obiect. Dintre instrucțiunile puternice ale microprocesorului Z80 se pot reaminti și următoarele: transfer și comparare la nivel de blocuri de date și prelucrare la nivel de bit.

Instrucțiunile microprocesorului se grupează în următoarele tipuri de instrucțiuni:

- instrucțiuni de transfer pe 8 biți;
- instrucțiuni de transfer pe 16 biți;

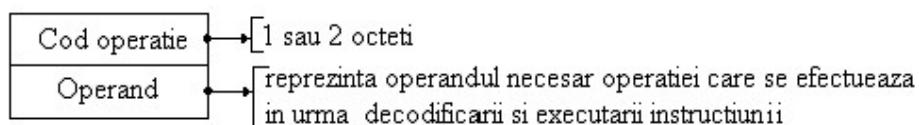
- instrucțiuni de transfer pe blocuri și căutări în interiorul blocurilor de memorie;
- instrucțiuni specifice operațiilor aritmetice și logice pe 8 biți;
- instrucțiuni corespunzătoare operațiilor de comandă a U.C.P.;
- instrucțiuni de rotație și deplasare;
- instrucțiuni de salt;
- instrucțiuni apel de subroutines și revenire în programul principal;
- instrucțiuni specifice operațiilor de intrare/ieșire.

**Observație:** Toate aceste tipuri de instrucțiuni se prezintă, în detaliu, în cadrul laboratorului aferent acestei discipline.

Modurile de adresare ale microprocesorului Z80 permit un transfer eficient între registre, memoria externă și dispozitivele periferice. În cele ce urmează se prezintă modurile de adresare ale microprocesorului Z80.

### Adresare imediată

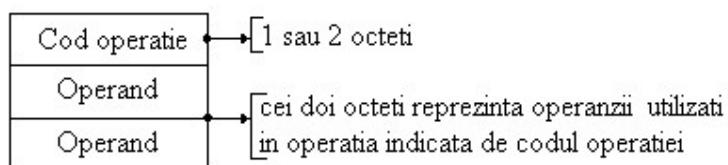
Structura instrucțiunii



Exemplu: LD A, 05    **efect**     $A \leftarrow 05_H$   
              ADD A, 30    **efect**     $A \leftarrow A + 30_H$

### Adresare imediată extinsă

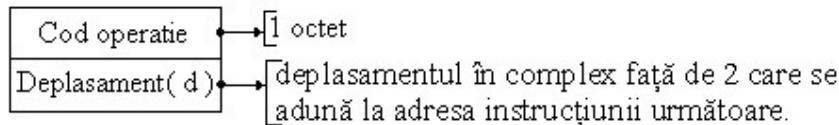
Structura instrucțiunii:



Exemplu: LD HL, nn    **efect**     $HL \leftarrow nn$

### Adresare relativă

Structura instrucțiunii:



Exemplu: JR Z, e      **efect:** dacă  $Z = 0$ , atunci continuă  
                              : dacă  $Z = 1$ , atunci  $(PC + d) \rightarrow PC$

### Adresare extinsă

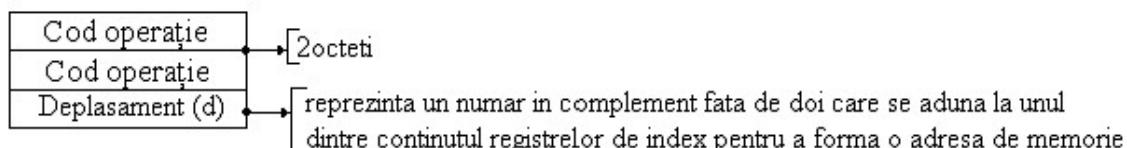
Structura instrucțiunii:



Exemplu: JP nn      **efect: salt la adresa nn**  
LD (nn),A      **efect: transferă data din registrul A în locația de memorie la adresa nn;**

### Adresare indexată

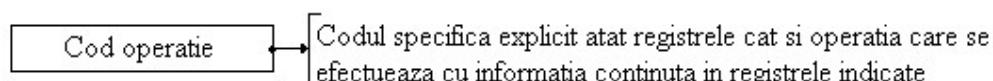
Structura instrucțiunii:



Exemplu: LD (Ix+d),n      **efect:  $OM_{(Ix+d)} \leftarrow n$**

### Adresare la registru

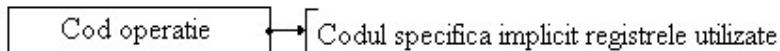
Structura instrucțiunii:



Exemplu: LD B,C      **efect:  $B \leftarrow C$**

### Adresare implicită

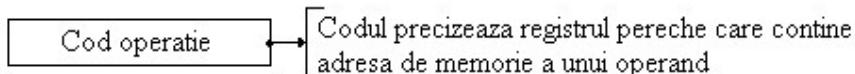
Structura instrucțiunii:



Exemplu: XOR A      *efect*     $A \leftarrow A \oplus A$

### Adresare indirectă cu registre

Structura instrucțiunii:



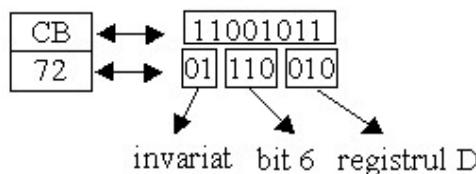
Exemplu: ADD A,(HL)      *efect*     $A \leftarrow A + DM_{(HL)}$

### Adresare pe bit

Structura instrucțiunii:



Exemplu: BIT 6,D      *efect*    testeaza bitul 6 al regisztrului D si modifica in mod corespunzator flagul Z



## Programarea în limbajul de asamblare al processorului Z80

Programarea într-un anumit limbaj de asamblare este strâns corelată cu modul de operare al unității de procesare utilizată, necesitând în unele situații și cunoștințe privind hard-ul microsistemului. Limbajul de asamblare presupune o gestionare riguroasă a resurselor procesorului, comparativ cu majoritatea limbajelor de nivel care nu au acces direct la resursele hard ale platformei de calcul.

Diferențele care există între un limbaj de asamblare și un limbaj de nivel înalt se pot concretiza atât în avantaje cât și în dezavantaje și anume:

- limbajul de asamblare permite accesul la toate resursele unui microprocesor, dând posibilitatea utilizatorului să realizeze aplicații care să fie performante atât din punct de vedere al spațiului de memorie folosit cât și din punct de vedere al vitezei de execuție;
- programarea necesită o organizare riguroasă, deoarece manipularea variabilelor de lucru presupune și cunoașterea locației de memorie la care se află acele variabile. Un alt aspect

care trebuie subliniat se referă la faptul că, dacă în cazul unui limbaj de nivel înalt adresarea unui element dintr-un vector se face prin nume și indice, în cazul limbajului de asamblare se face printr-o adresare relativă care presupune și operații prin care se determină adresa respectivă. Registrele prin care se adresează respectivul element nu trebuie folosite și de alte instrucțiuni sau dacă sunt folosite trebuie salvate la acel moment ca apoi să se refacă. Toate aceste operații necesită o abordare riguroasă și coerentă în ceea ce privește realizarea programelor într-un limbaj de asamblare. Abordare impune ca programatorul să cunoască starea tuturor resurselor procesorului (registrele acestuia, alocarea memorie, sistemul de intrerupere etc) și implicațiile modificării acestor resurse asupra bunei funcționări a întregului microsistem.

Realizarea și rularea unui program în asamblare presupune parcurgerea următoarelor etape:

- conceperea algoritmului;
- scrierea programului în limbajul de asamblare corespunzător procesorului Z80;
- translarea fișierului text care conține programul sursă, în fișiere obiect relocabile utilizând *un assembler*;
- linke-ditarea fișierelor obiect într-un singur fișier obiect executabil, prin reevaluarea atât a legăturilor interne cât și a legăturilor externe. Aceasta etapă de lucru se realizează cu ajutorul unui program denumit linkeditor;
- încărcarea programului în memoria de lucru a microsistemului;
- rularea programului încărcat în memorie și evident verificarea rezultatului obținut.

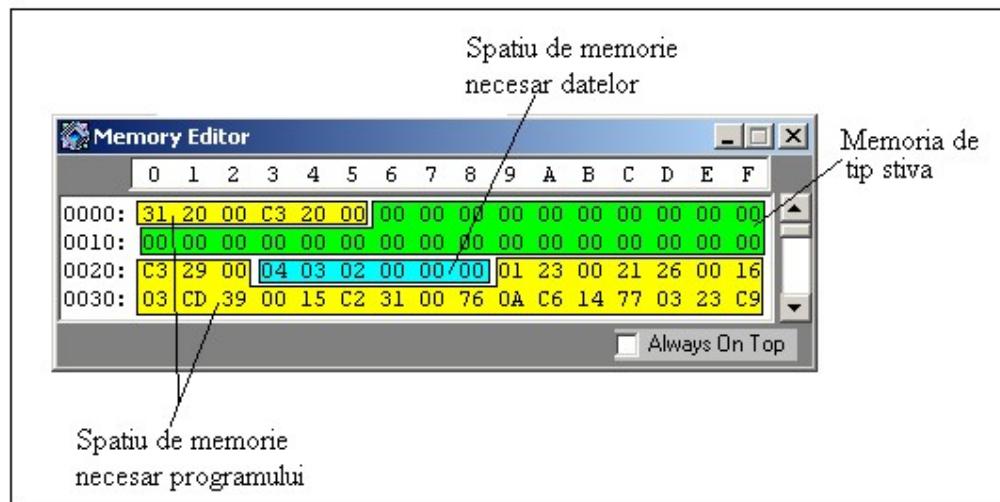
În cele ce urmează se ilustrează etapele menționate mai sus, considerând un exemplu simplu de calcul al sumei unei constante la un vector și apoi memorarea aceluia vector la o anumită zonă de memorie. Programul realizat și editat cu un editor de fișiere text este dat mai jos și este însotit de comentarii, astfel că nu mai necesită și alte explicații suplimentare.

## EXEMPLU DE PROGRAM REALIZAT IN ASAMBLARE Z80

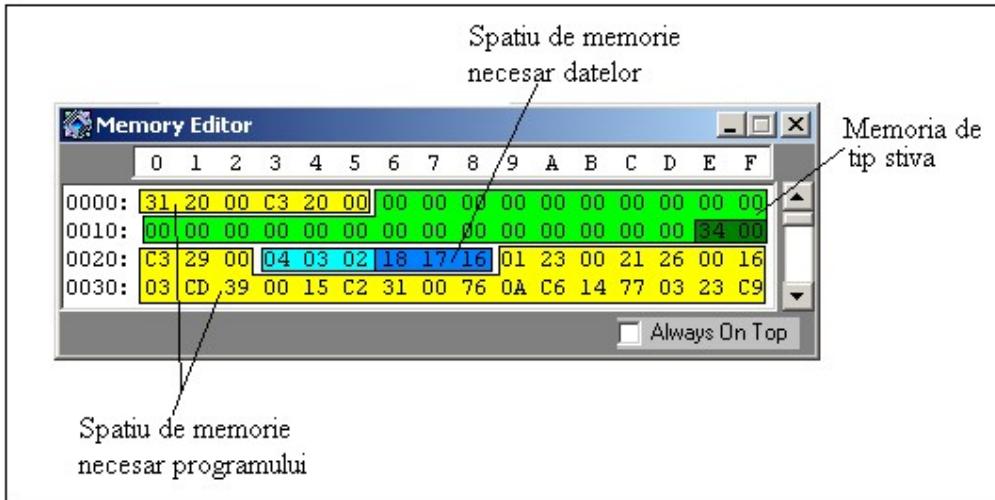
Fisierul <u>Prog . asm</u>	Fisierul <b>Prog . Ist</b>	Fisierul <b>Prog . obj</b>
; Ex. de pr. în asamblare Z80 ; Se setează indicatorul de stivă LD SP,0020H JP ST	0001 0000 ; Ex. de pr. în asamblare Z80 0002 0000 ; Se setează indicatorul de stivă <b>0003 0000 31 20 00 LD SP,0020H</b> <b>0004 0003 C3 20 00 JP ST</b>	<b>0000 31 20 00</b> <b>0003 C3 20 00</b> 0006 00 0007 00 0008 00
; Se stabilește adresa unde ; se plasează codul programului .ORG 0020H ST: JP LP1	0005 0006 ; Se stabilește adresa unde 0006 0006 ; se plasează codul programului <b>0007 0020 .ORG 0020H</b> <b>0008 0020 C3 29 00 ST: JP LP1</b>	0009 00 000A 00 000B 00 000C 00 000D 00 000E 00
; Se definesc datele de intrare LD1: .DB 04H .DB 03H .DB 02H	0009 0023 ; Se definesc datele de intrare <b>0010 0023 04 LD1: .DB 04H</b> <b>0011 0024 03 .DB 03H</b> <b>0012 0025 02 .DB 02H</b>	000F 00 0010 00 0011 00 0012 00 0013 00 0014 00
; Se definește spațiul destinație SD1: .DB 00H	0013 0026 ; Se definește spațiul destinație <b>0014 0026 00 SD1: .DB 00H</b>	0015 00 0016 00

.DB 00H	0015 0027 00	.DB 00H	0017 00
.DB 00H	0016 0028 00	.DB 00H	0018 00
; Se seteaza constanta CT1	0017 0029 ; Se seteaza constanta CT1	0019 00	0019 00
CT1 .EQU 14H	0018 0029 CT1 .EQU 14H	001A 00	001A 00
LP1: LD BC,LD1	0019 0029 01 23 00 LP1: LD BC,LD1	001B 00	001B 00
LD HL,SD1	0020 002C 21 26 00	001C 00	001C 00
LD D,03H;	0021 002F 16 03	001D 00	001D 00
LP2: CALL R1	0022 0031 CD 39 00 LP2: CALL R1	001E 00	001E 00
DEC D	0023 0034 15	001F 00	001F 00
JP NZ,LP2	0024 0035 C2 31 00	0020 C3 29 00	
HALT	0025 0038 76	HALT	
; Rutina care aduna la data	0026 0039 ; Rutina care aduna la data	0023 04	
; din memorie de la adresa (BC)	0027 0039 ; din memorie de la adresa (BC)	0024 03	
; contanta CT1 si o depune	0028 0039 ; contanta CT1 si o depune	0025 02	
; la adresa de memorie (HL)	0029 0039 ; la adresa de memorie (HL)	0026 00	
R1: LD A,(BC)	0030 0039 0A R1: LD A,(BC)	0027 00	
ADD A,CT1	0031 003A C6 14	0028 00	
LD (HL),A	0032 003C 77	0029 01 23 00	
INC BC	0033 003D 03	002C 21 26 00	
INC HL	0034 003E 23	002F 16 03	
RET	0035 003F C9	0031 CD 39 00	
.END	0036 0040	0034 15	
		0035 C2 31 00	
		0038 76	
		0039 0A	
		003A C6 14	
		003C 77	
		003D 03	
		003E 23	
		003F C9	

Spatiul de memorie necesar executiei programului, inaintea rularii acestuia



Spatiu de memorie necesar executiei programului, dupa rularea acestuia



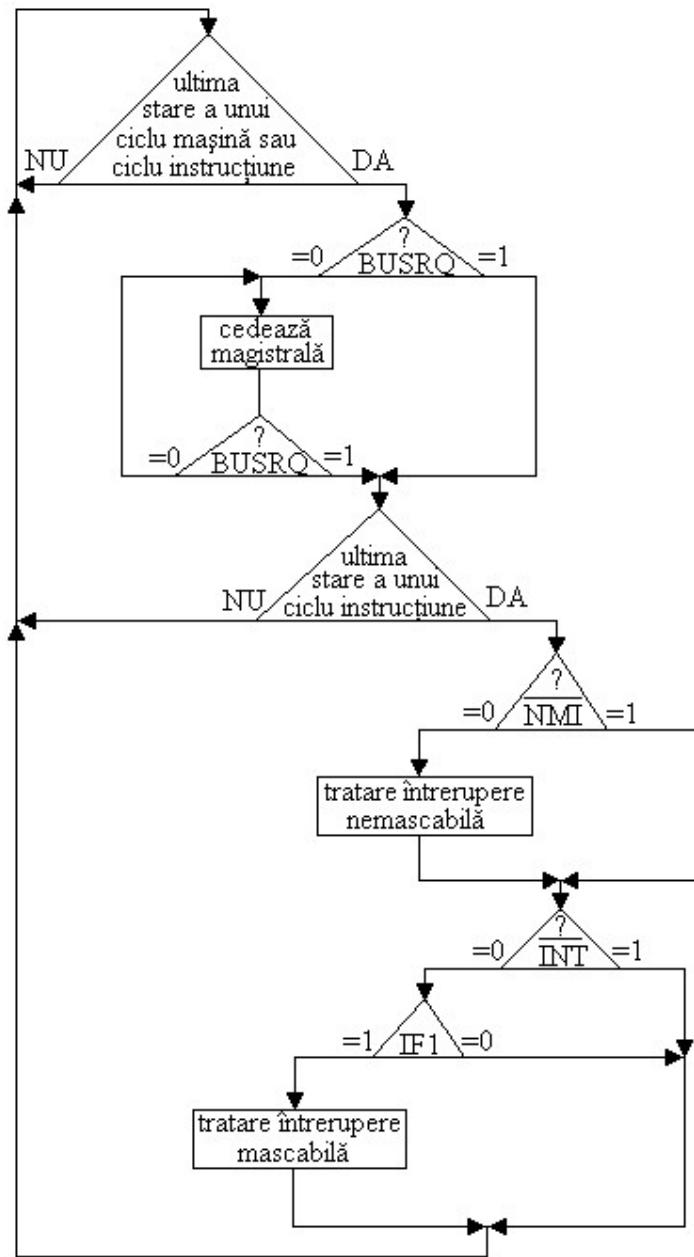
În cadrul programului s-au utilizat pe langă instrucțiunile specifice microprocesorului Z80 și directive specifice limbajului de asamblare. Directivele sunt *elemente ale limbajului de programare prin care se stabilesc secțiunile de date și de program și elementele prin care se definesc constantele și variabilele de lucru*. Aceste directive oferă un cadru riguros și comod în ceea ce privește programarea într-un limbaj de asamblare și sunt necesare inițializării variabilelor și rezervării unor zone de memorie.

Directivele utilizate în cadrul exemplului de mai sus sunt:

- Directiva **.ORG** prin care se stabilește adresa unde se placează codul programului  
Directiva **.DB** prin care se definesc datele de intrare sau se rezervă spațiu de lucru  
Directiva **.EQU** prin care se setează valoarea unei anumite constante

## STUDIUL CERERILOR DE MAGISTRALĂ, A ÎNTRERUPERILOR ȘI STAREA HALT

Aceste moduri de lucru ale microprocesorului contribuie la creșterea flexibilității microprocesorului deoarece cererile de magistrală și întreruperile permit comunicarea într-un mod asincron cu alte dispozitive și la inițiativa acestora. Prin utilizarea acestor facilități se degrevează microprocesorul de anumite operații care trebuie făcute ciclic și la intervale de timp relativ mici. Având în vedere faptul că atât cererile de magistrală cât și întreruperile sunt generate de către dispozitive externe, microprocesorul le tratează într-o ordine bine stabilită. Organograma prin care se ilustrează modul de tratare a acestor evenimente este dată în figura următoare.

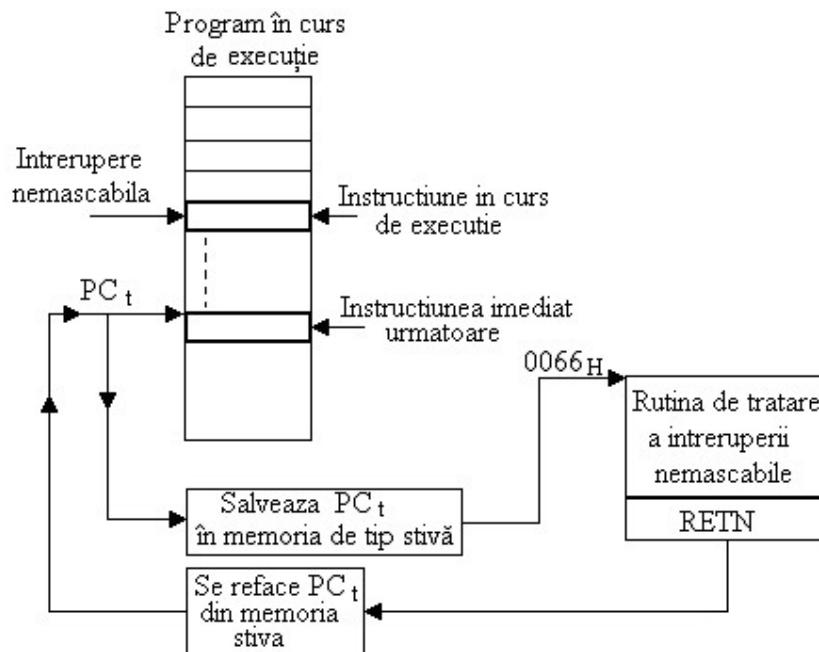
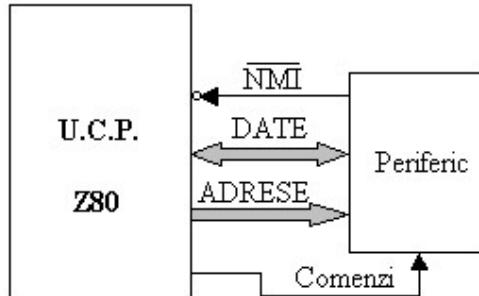


Cerile de magistrală sunt prioritare și se tratează la sfârșitul ciclurilor mașină. Întreruperile nemascabile sunt preluate în orice moment și se tratează la sfârșitul instrucțiunii în curs de execuție, iar întreruperile mascabile au prioritatea cea mai mică și sunt preluate și tratate la sfârșitul instrucțiunii în curs de execuție, numai dacă acestea (întreruperile nemascabile) au fost activate în prealabil (IF1=1).

## ÎNTRERUPERILE MICROPROCESORULUI Z80

**Întreruperile nemascabile** sunt prioritare față de întreruperile mascabile și mai puțin prioritare față de o cerere de magistrală. La apariția unei întreruperi, microprocesorul salvează PC-ul instrucțiunii imediat următoare în memoria de tip stivă și trece la tratarea

întreruperii nemascabile prin execuția rutinei de tratare a încreruperii care se află la adresa  $0066_H$ . Modul de conectarea a unui periferic care generează o intrerupere nemascabilă și tratarea acesteia sunt ilustrate în figurile următoare



**Observatie:** Cererea de încrerupere nemascabilă este reținută în orice moment, chiar și în situația în care microprocesorul are cedate magistralele, dar se tratează numai la sfârșitul instrucțiunii în curs de execuție.

- **Încreruperile mascabile** sunt preluate și tratate la sfârșitul execuției instrucțiunii curente numai dacă acestea au fost activate în prealabil. Microprocesorul Z80 are trei moduri de tratare a încreruperilor nemascabile, moduri care se stabilesc cu ajutorul următoarelor instrucțiuni: IM0, IM1, IM2. Modul zero (IM0) este compatibil cu modul de tratare a intreruperii, corespunzător microprocesorului I8080. Modul unu (IM1) poate fi utilizat de periferice care nu aparțin familiei Z80. Modul doi (IM2) este specific familiei de periferice Z80. Se menționează faptul că activarea încreruperilor se face cu instrucțiunea EI, iar dezactivarea acestora cu instrucțiunea DI.

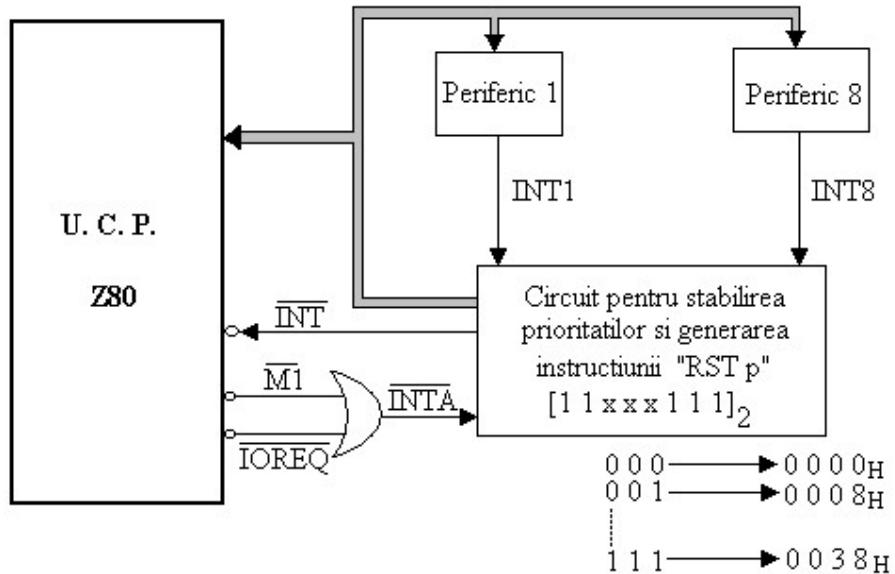
**Observație:** La resetarea microprocesorului îintreruperile mascabile sunt dezactivate, iar modul setat este modul zero.

## Modul 0 de intrerupere

Modul zero este compatibil cu modul de îintrerupere al microprocesorului 8080. Perifericul care generează o îintrerupere plasează pe magistrala de date codul instrucțiunii  $RST\ p$  (RESTART  $p$ ), unde  $p = 00_H, 08_H, 10_H, 18_H, \dots, 38_H$ , sau codul unei instrucțiuni CALL adr.. În cazul plasării instrucțiunii  $RST\ p$  se salvează PC-ul instrucțiuni imediat următoare și se trece la execuția instrucțiunilor de la adresa  $0000_H$  sau  $0008_H \dots 0038_H$ , în funcție de valoarea lui  $p$ . Dacă programul permite tratarea îintruperii în spațiul de adresă de 8 octeți, atunci la adresa respectivă nu este necesară o instrucțiune de salt necondiționat la o altă adresă, unde se găsește adevărata rutină de tratare a îintruperilor generată de perifericul respectiv.

În cazul utilizării controlerelor de îintrerupere, cum ar fi circuitul integrat 8259, acestea generează o instrucțiune CALL, deci permite plasarea rutinelor de tratare a îintruperilor oriunde în memoria microsistemuui.

Modul de generare și tratare a unei îintruperi în cazul în care perifericul plasează instrucțiunea  $RST\ p$  este ilustrat în figura următoare.



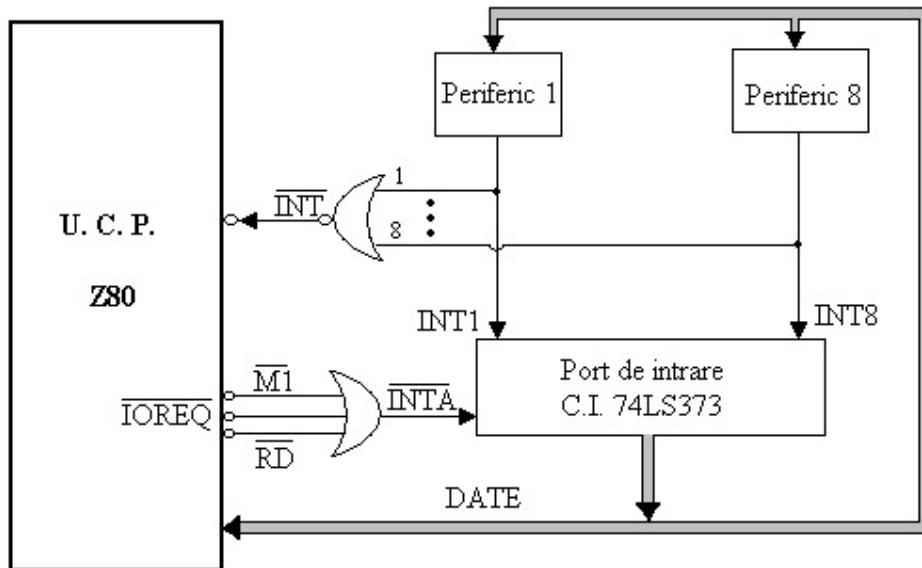
În momentul generării unei îintruperi de către cel puțin un periferic, circuitul specializat generează o îintrerupere către microprocesor, iar acesta răspunde prin semnalul  $\overline{INTA} = \overline{M1} + \overline{IOREQ}$ , moment în care circuitul specializat pune pe magistrala de date instrucțiunea  $RST\ p$  corespunzătoare perifericului cu cea mai mare prioritate. Microprocesor tratează îintreruperea prin rularea programului corespunzător perifericului care a generat respectiva îintrerupere. Un circuit specializat care rezolvă atât problema priorităților, cât și a generării codului instrucțiunii  $RST\ p$  este circuitul I8214.

Dacă se utilizează controlerul de intrerupere CI 8259, atunci rutinele de tratare a intreruperilor pot fi plasate oriunde în memoria microsistemu, deoarece acestea sunt memorate într-o tabelă, iar accesul la adresa rutinei se face prin intermediul a doi octeți generați de controler.

## Modul 1 de intrerupere

Dacă intreruperile sunt validate, iar la sfârșitul instrucției în curs de execuție pinul INT se află la ‘0’ logic, atunci microprocesorul salvează PC instrucțiunii imediat urmatoare și trece la tratarea intreruperii, executând programul care se află la adresa 0038H. Întreruperea mascabilă-modul 1 este similară intreruperii nemarcabile cu excepția faptului că aceasta (intreruperea nemascabilă) se trasează cu ajutorul unui program aflat la adresa 0066H. Acest mod de intrerupere se poate utiliza și în cazul când există mai multe periferice care generează o intrerupere, urmând ca prioritatea să se stabilească prin interogare.

Schimbul de principiu în cazul modului 1 de intrerupere prin care se tratează intreruperi de la 8 periferice este dată în figura următoare.

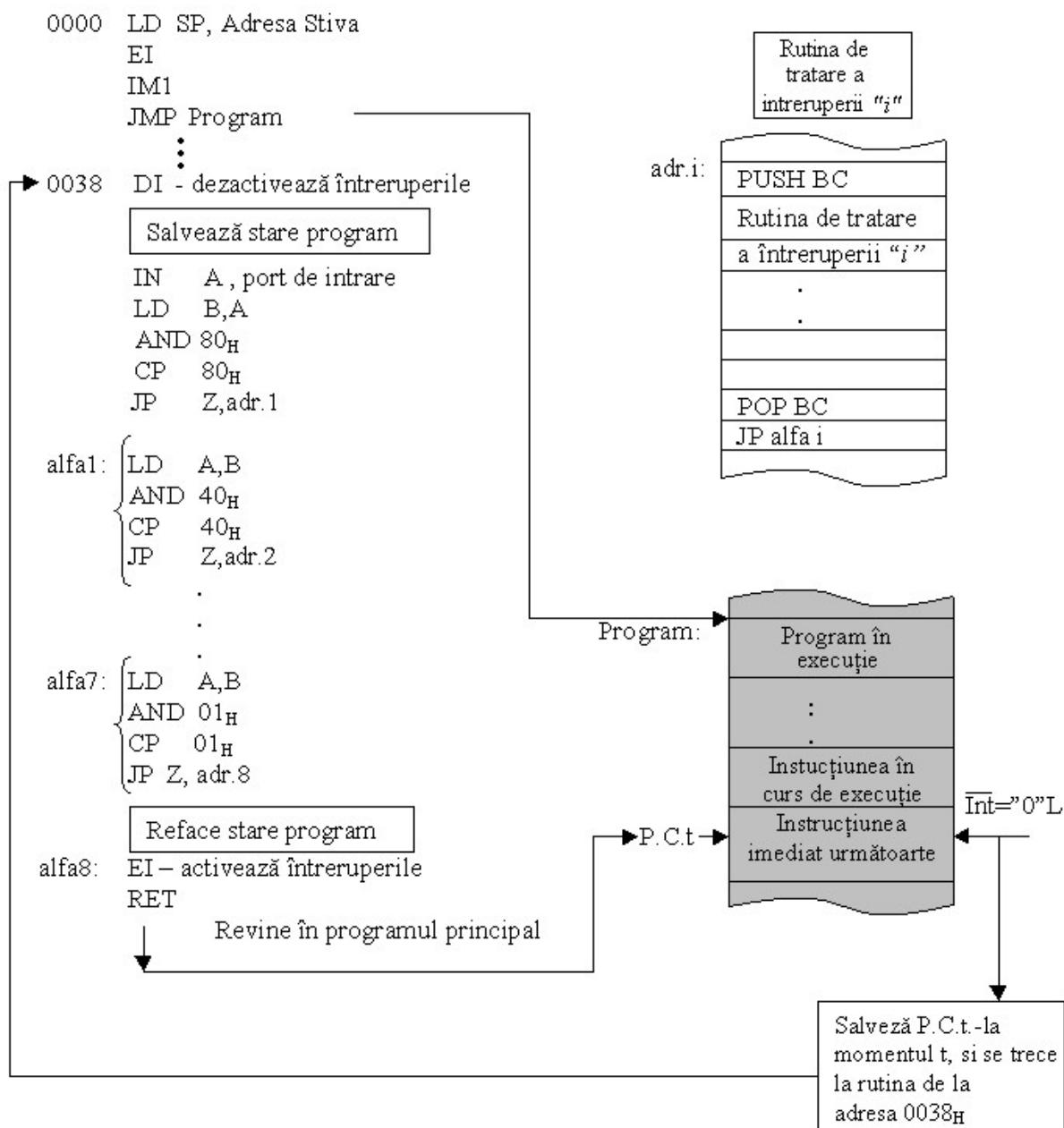


Dialogul se desfășoară după protocolul următor:

- un periferic trimite pe linia  $INT_i$  o cerere de intrerupere care are drept efect generarea unui semnal egal cu unu logic și plasarea acestuia pe una din intrările portului 74LS373. Dacă cel puțin unul din semnalele  $INT_i$  este unu logic, atunci semnalul  $\overline{INT}$  devine zero logic, moment în care se anunță procesorul că cel puțin un periferic a generat o intrerupere.

- microprocesorul ia cunoștință de întrerupere și salvează contorul program corespunzător instrucțiunii imediat următoare, trecând apoi la execuția programului de la adresa 0038<sub>H</sub>. În cadrul acestui program se citește data de la perifericul de intrare și se trece la tratarea întreruperii perifericului cu prioritate maximă, ca apoi să se revină în programul principal sau să se trateze întreruperea imediat următoare în ordinea priorităților. Achitarea întreruperii trebuie să o facă rutina de tratare a întreruperilor prin resetarea cererii (INT i = 0).

Programul prin care se realizează tratarea întreruperilor este dat mai jos.

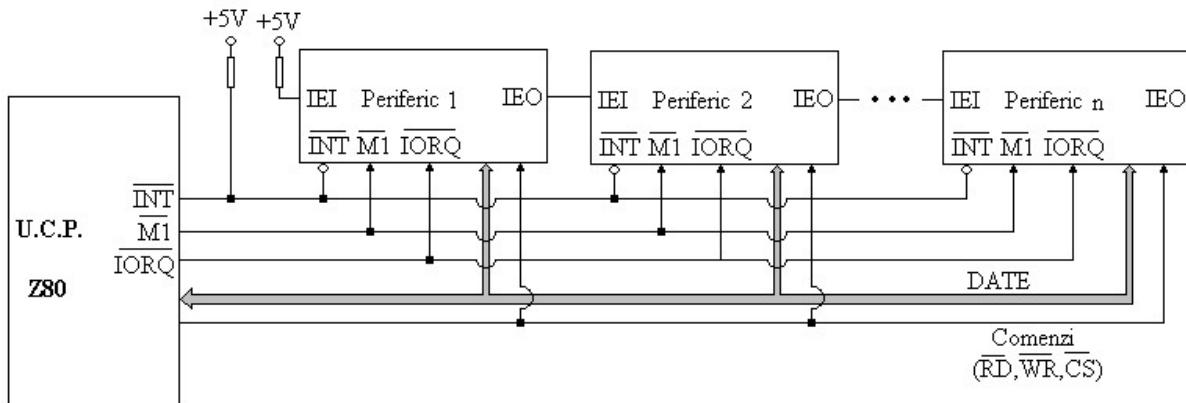


În cadrul acestui mod de întrerupere se simplifică hardul în ceea ce privește stabilirea priorităților, dar crește durata privind determinarea perifericului care a generat întreruperea, deoarece aceasta (stabilirea priorităților) se face prin program.

## Modul 2 de întrerupere

Acest mod de întrerupere este specific perifericilor din familia micropresorului Z80 și anume PIO-Z80; SIO-Z80 ; CTC-Z80 . Se poate lucra în acest mod numai după execuția instrucțiunile *EI* , *IM2* . În cadrul acestui mod se pot deservi până la 128 de periferice a căror prioritate se stabilește prin poziționarea lor într-o ordine care corespunde priorităților stabilite aprioric.

Modul de conectare a perifericilor din familia Z80 care utilizează modul 2 de întrerupere este dat în figura de mai jos.



Fiecare periferic poate genera o întrerupere dacă sunt îndeplinite urmatoarele condiții :

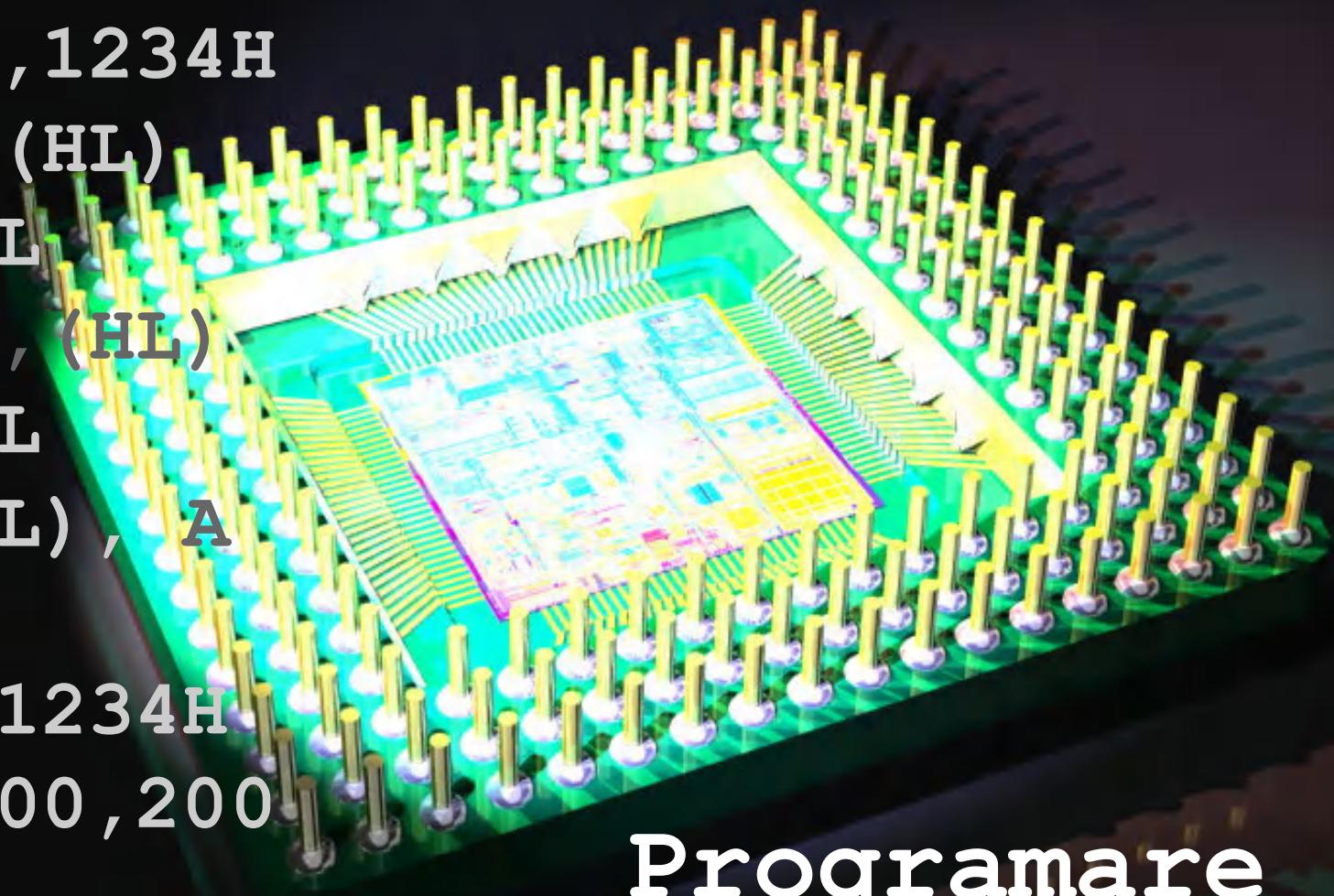
- a fost programat să lucreze în modul de întrerupere corespunzător modului 2 oferit de micropresorul Z80.
- intrarea IEI (intrare de activare a intreruperilor) este la 1 logic.
- registrul care conține octetul inferior al adresei de memorie unde se află adresa rutinei de tratare a intreruperilor a fost încărcat cu valorile corespunzătoare ;

Dacă sunt îndeplinite condițiile menționate mai sus, în cazul apariției unei întreruperi se parcurg următoarele etape :

- se citește vectorul de întrerupere corespunzător perifericului care a generat întreruperea activându-se semnalele *M1* și *IREQ* ;
- se salvează PC instructiunii imediat următoare în memoria de tip stivă și se extrage din tabela cu adrese, adresa rutinei de tratare a intreruperii asociată perifericului ;
- se trece la execuția rutinei de tratare a intreruperii și apoi se revine în programul aflat în curs de execuție.

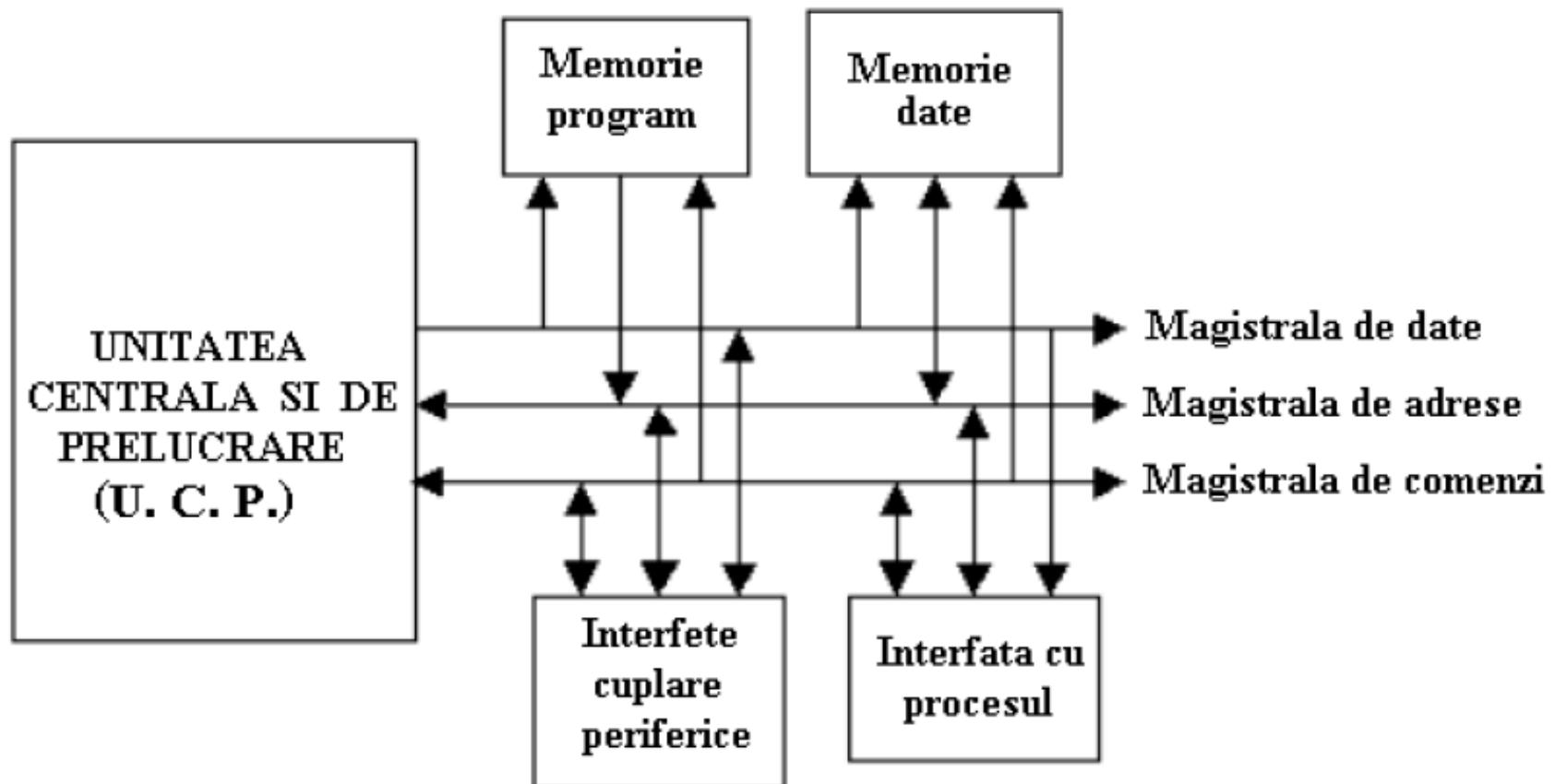
Fiecare periferic din familia lui Z80 dispune de o ieșire de inhibare a celorlalte periferice (IEO). De fapt prin acest mecanism se stabilește prioritatea în cazul apariției mai

```
.ORG 100H  
LD HL,1234H  
LD A, (HL)  
INC HL  
ADD A, (HL)  
INC HL  
LD (HL), A  
.END  
.ORG 1234H  
.DB 100,200
```



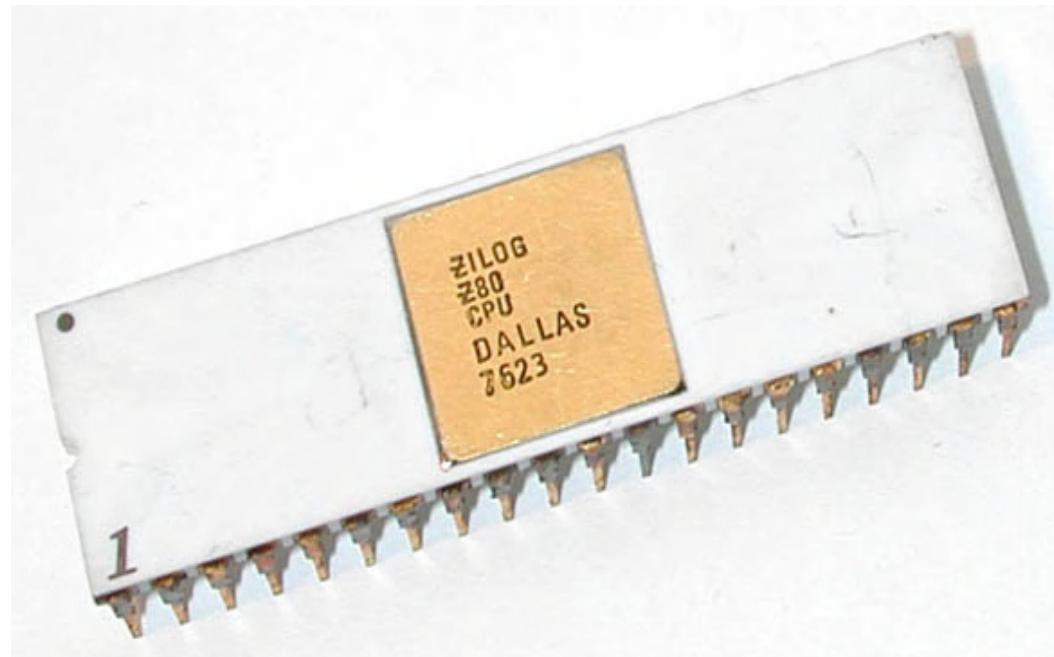
# Programare în Limbaje de Asamblare

# MICROPROCESOARE

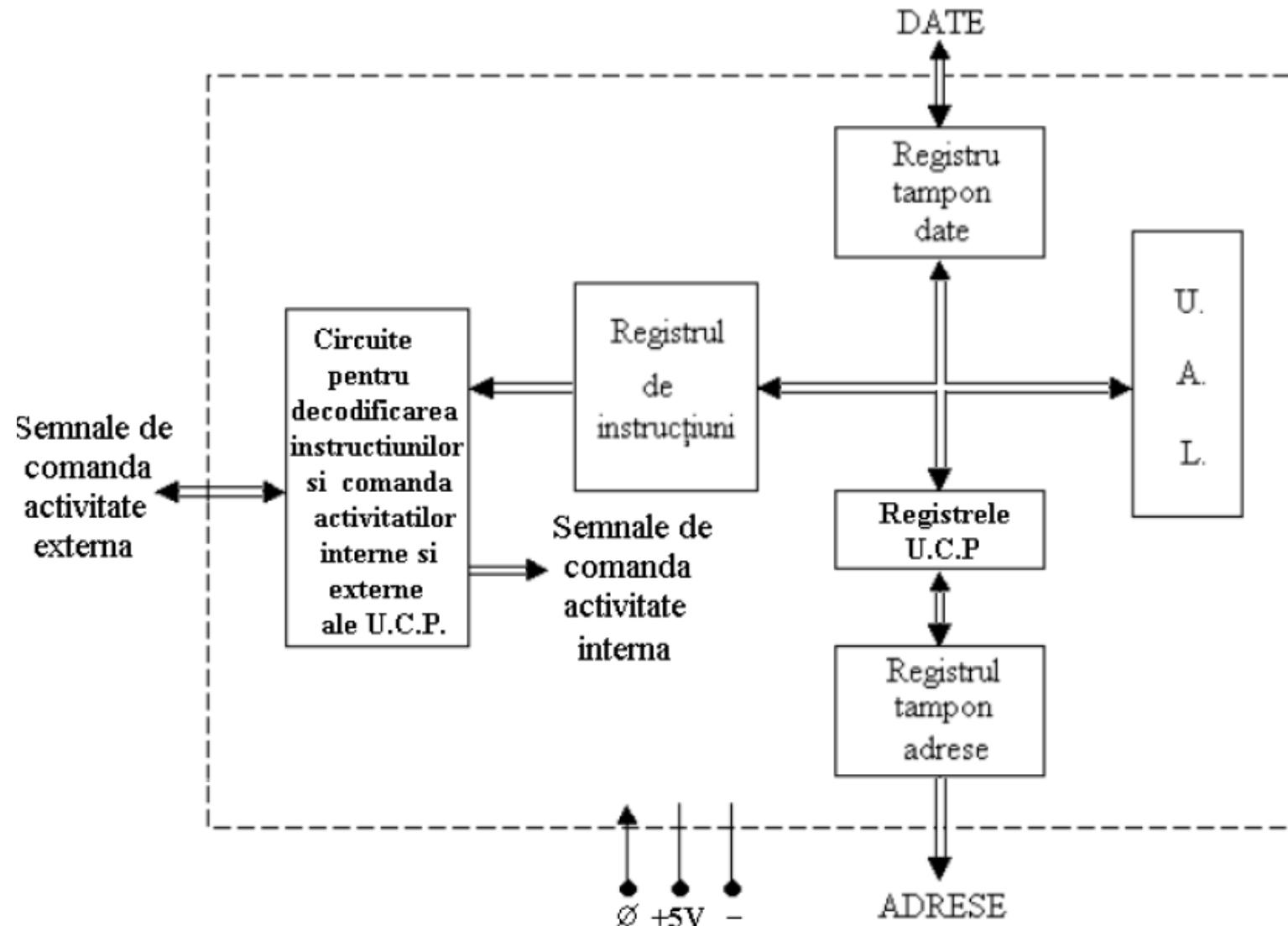


# Unitatea centrală și de prelucrare a micropresorului Z80

- Micropresorul Z80 reprezintă unul dintre cele mai performante procesoare organizate pe 8 biți, fiind realizat într-o structură unitară și disponând de un set de instrucțiuni evoluat, comparativ cu celealte procesoare pe 8 biți.



# Unitatea centrală și de prelucrare a microprocesorului Z80



# **Unitatea centrală și de prelucrare a microprocesorului Z80**

**Circuitele pentru decodificarea instrucțiunilor** reprezintă componenta de bază a unui microprocesor având rolul de a interpreta (decodifica) instrucțiunile reținute în registrul de instrucțiuni și de a genera acele semnale de comandă (interne și externe) care permit rezolvarea cerinței specificată de către instrucțiunea care se execută la momentul curent.

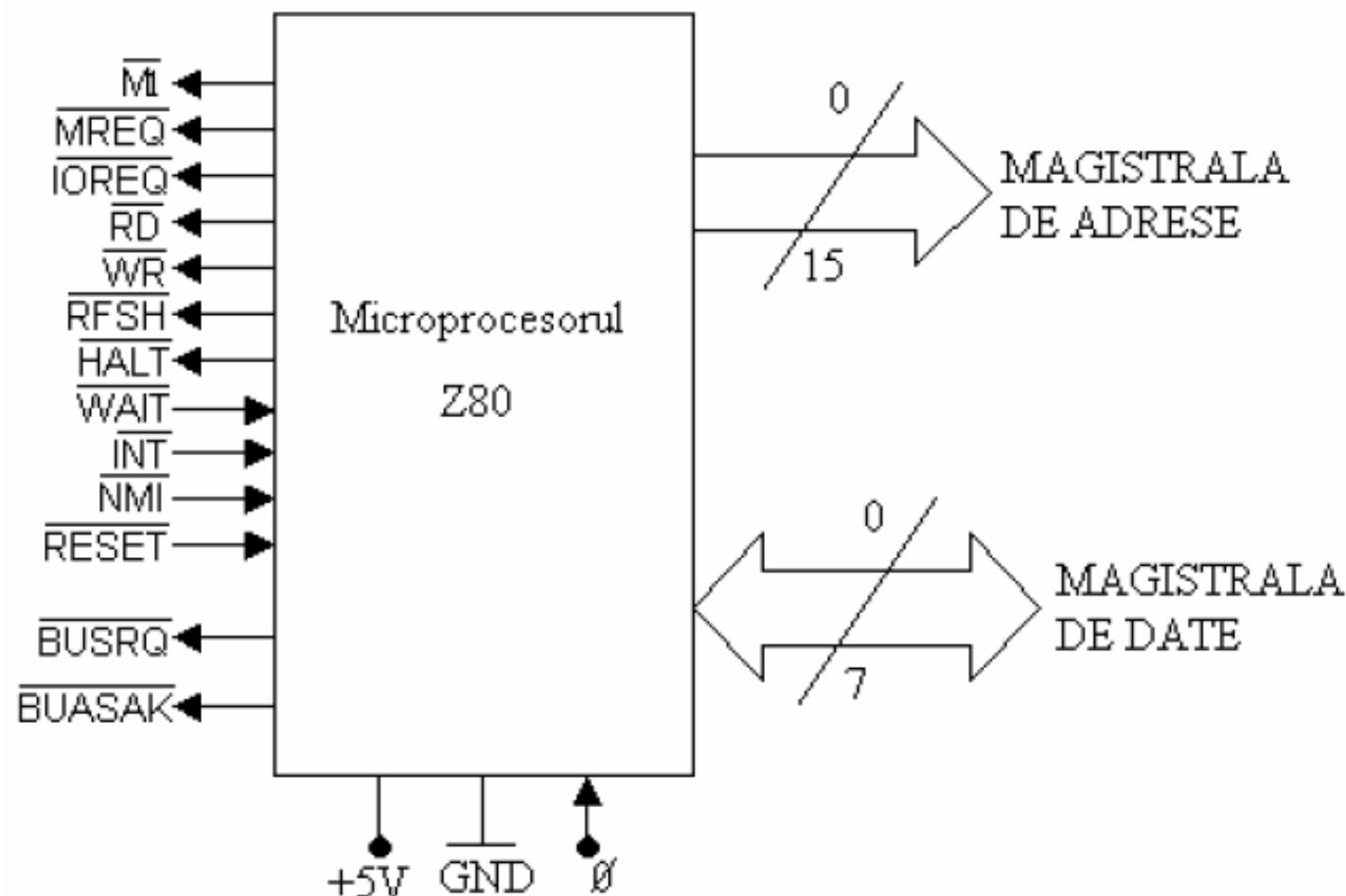
**Registrul de instrucțiuni** menține, un timp bine precizat, codul instrucțiunii în curs de execuție la intrările circuitului pentru decodificarea instrucțiunilor.

**Registrele unității centrale** sunt memorii rapide care au rolul de a mări viteza de execuție a instrucțiunilor și de a crește flexibilitatea în ceea ce privește implementarea algoritmilor.

**Registrele tampon de date și adrese** au rolul de a menține adresele și datele pe magistralele de adresa și date un interval de timp bine definit.

**Unitatea aritmetico-logică** (U.A.L.) reprezintă o altă componentă importantă a microprocesorului care are rolul de a executa operații aritmetice și logice între operanzi ce se pot afla în registrele unității centrale sau în memoria sistemului.

# Descrierea conexiunilor externe ale microprocesorului Z80



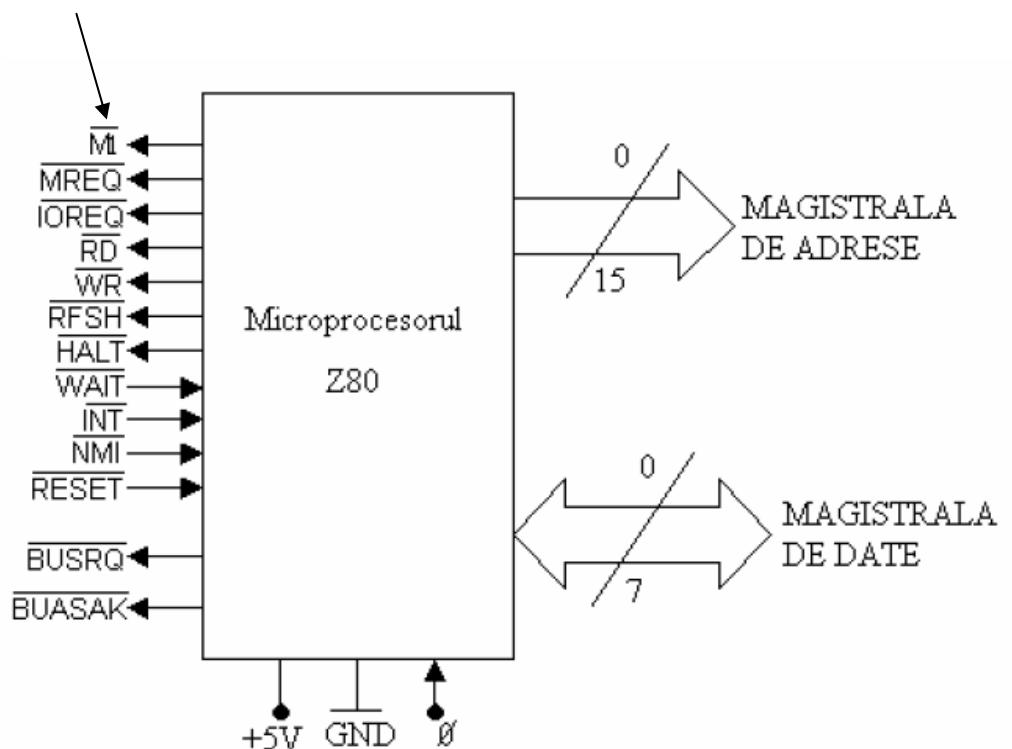
# Descrierea conexiunilor externe ale microprocesorului Z80

**M1** – activ pe 0 logic indica extragerea unui cod de operatie numit si OPCODE (ciclu fetch)

**MREQ** – activ pe zero logic si indica lucrul cu memoria, pe magistrala avem o adresa de citire sau scriere in / din memoria interna sau externa

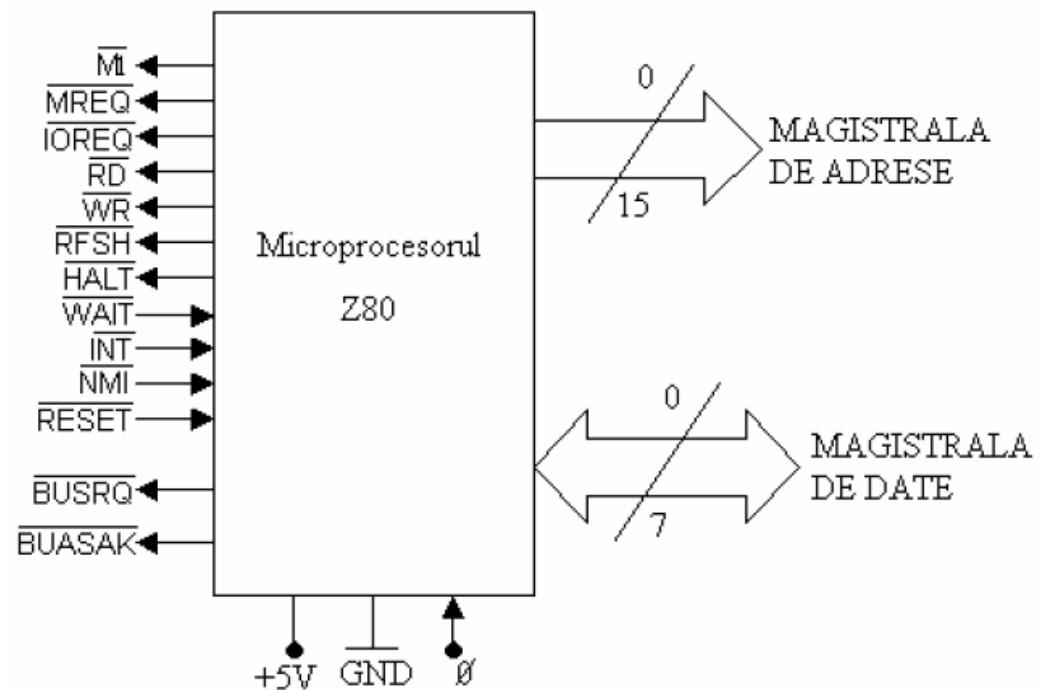
**IREQ** – activ pe 0 logic permite lucrul cu dispozitivele de IO

**RD** – activ pe 0 logic realizeaza citirea datelor din memorie sau de la un periferic



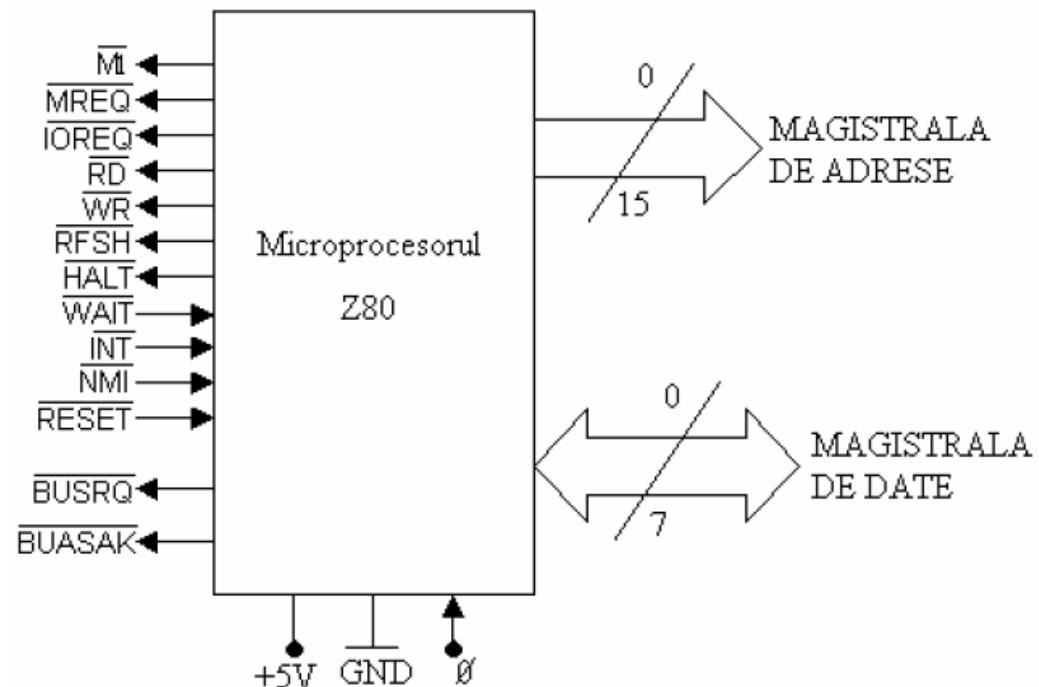
# Descrierea conexiunilor externe ale microprocesorului Z80

- **WR** – activ pe 0 logic realizeaza scrierea in memorie sau la un periferic
- **RFSH** – active pe 0 logic realizeaza refresh-ul memoriei RAM dinamice
- **HALT** – activ pe 0 logic determina oprirea UCP (se executa NOP pentru refresh-ul memoriei RAM)
- **WAIT** – semnal care realizeaza dialogul cu perifericele mai lente prin introducerea unor stari de asteptare



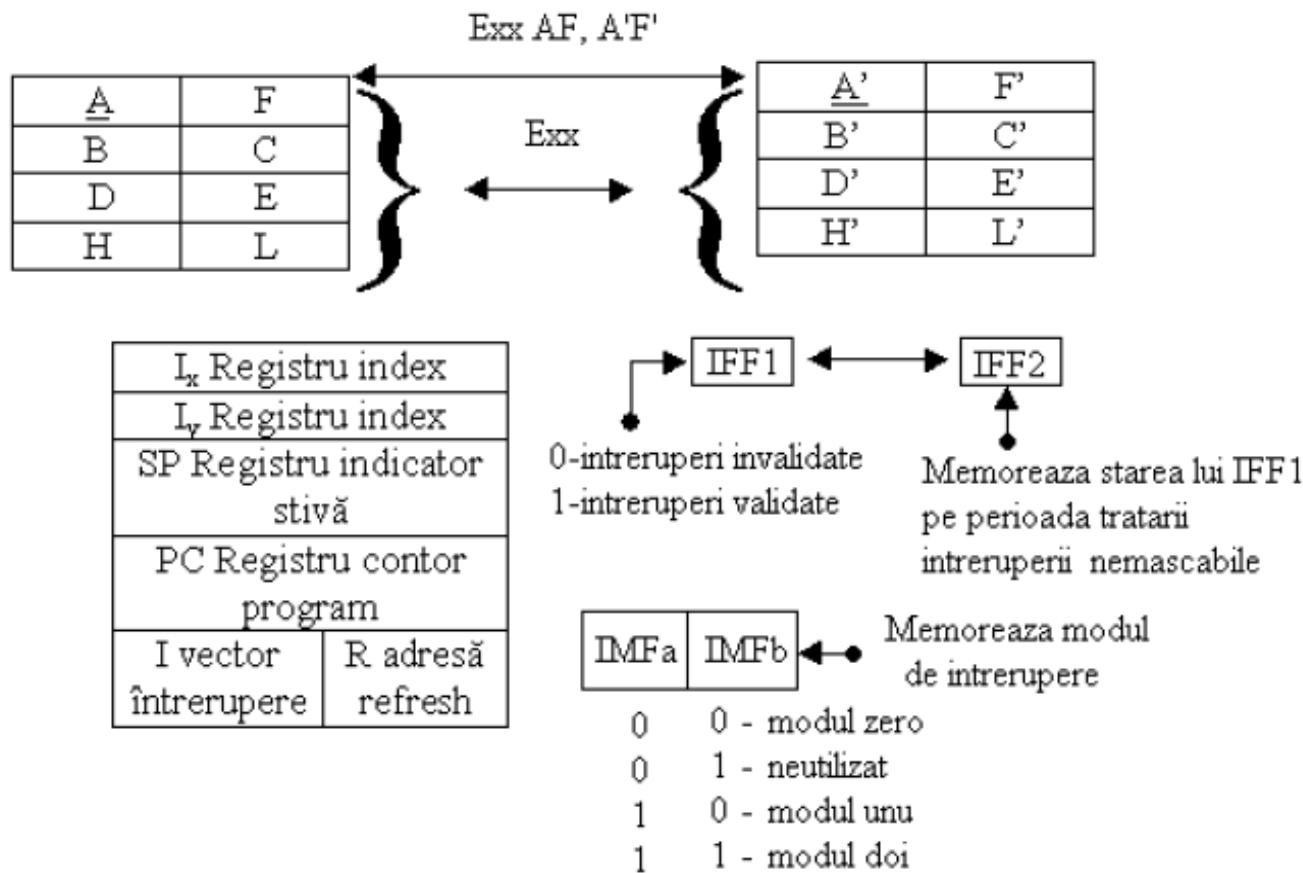
# Descrierea conexiunilor externe ale microprocesorului Z80

- **INT** – cerere de intrerupere care este validata prin instructiunea EI si invalidate prin instructiunea DI
- **NMI** – permite intreruperea activitatii procesorului in orice moment de timp
- **RESET** – semnal de resetare a MPU care trebuie tinut pe 0 logic cel putin 3 perioade ceas pentru a reseta toate subsistemele MPU
- **BUSRQ** – este un semnal de cerere de magistrala trimis de o alta unitate de procesare
- **BUSAK** – semnal care anunta unitatea care a cerut magistrala ca aceasta este disponibila



# Descrierea registrelor U.C.P.

- Unitatea Centrală de Prelucrare cu Z80 conține registre organizate pe 8 sau 16 biți ce permit creșterea flexibilității U.C.P.-ului și a vitezei de lucru a microprocesorului Z80.



# Descrierea registrelor U.C.P.

- **A** - registrul acumulator, este un registru des utilizat de către U.A.L. Informația conținută în acesta poate fi atât sursă cât și destinație în cadrul operațiilor executate de către microprocesor.
- **BC, DE, HL** – sunt registre de câte 16 biți, care pot fi utilizati și ca registre de 8 biți (**B, C, D, E, H, L**). Aceste registre sunt folosite atât în operațiile logice și aritmetice, cât și în operațiile prin care se realizează transfer de informație de la memorie la microprocesor și invers.
- **IX, IY** - registre de index, care permit memorarea adresei de bază în cazul unor instrucțiuni care folosesc adresarea relativă.
- **F** - registru care memorează indicatorii de condiție rezultați în urma efectuării operațiilor.

# Descrierea registrelor U.C.P.

- Registrul F (FLAG) are următoarea structură:

S	Z	*	H	*	P/V	N	C
---	---	---	---	---	-----	---	---

- **S** - indicator de semn ( $S = 1$  dacă rezultatul operației executate este negativ);
- **Z** - indicator de zero ( $Z = 1$  dacă rezultatul unei operații este zero);
- **C** - indicator de transport care se setează cu 1 dacă operația a produs un transport la cel mai semnificativ bit al operandului sau rezultatului;
- **P/V** - indicator de paritate/depășire
- **H** - indicator de transport auxiliar, fiind utilizat numai de către instrucțiunile de ajustare zecimală;
- **N** - indicator de adunare sau scădere care este utilizat de asemenea de către instrucțiunile de ajustare zecimală;

# Descrierea registrelor U.C.P.

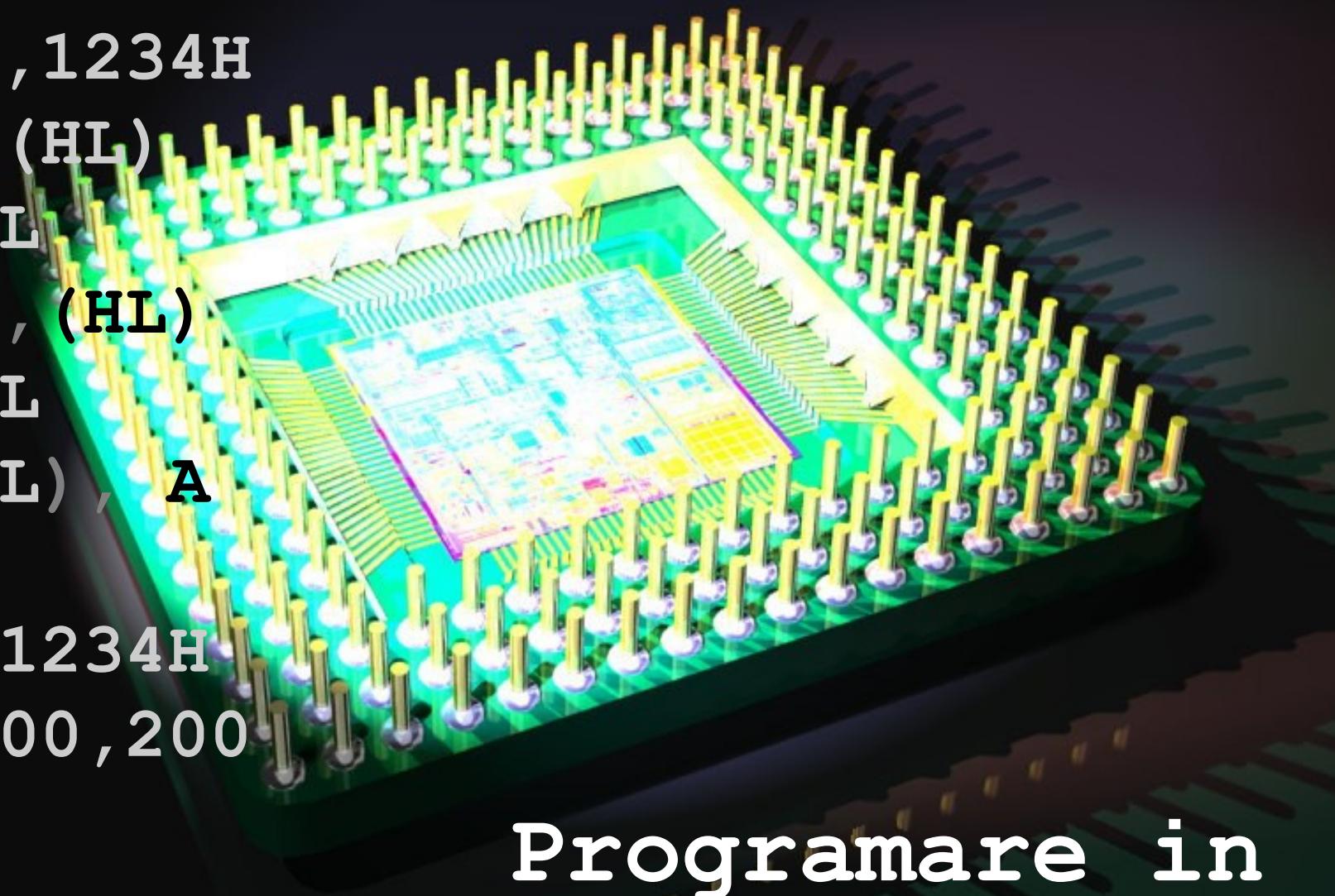
- **IF1, IF2** bistabile care permit lucrul cu întreruperile mascabile și nemascabile;
- **IMFa, IMFb** - bistabile care memorează unul dintre cele 3 moduri de tratare a întreruperilor.
- **SP** - indicator de stivă, care este de fapt un registru pe 16 biți ce permite lucrul cu memoria de tip stivă, memorie utilizată în cazul instrucțiunilor de tip CALL sau în cazul tratării întreruperilor.
- **PC** - contor de program, este un registru pe 16 biți care memorează adresa instrucțiunii în curs de execuție.
- **I** - registru pe 8 biți care conține cei mai semnificativi 8 biți ai adresei tabelei cu adresele rutinelor de tratare a întreruperilor.

**0100110101110101101100011101000111  
01010110110101100101011100110110001  
10010000001110000011001010110111001  
11010001110010011101010010000001100  
00101110100011001010110111001110100  
01101001011001010010000000100001**

**sau altfel spus ...**

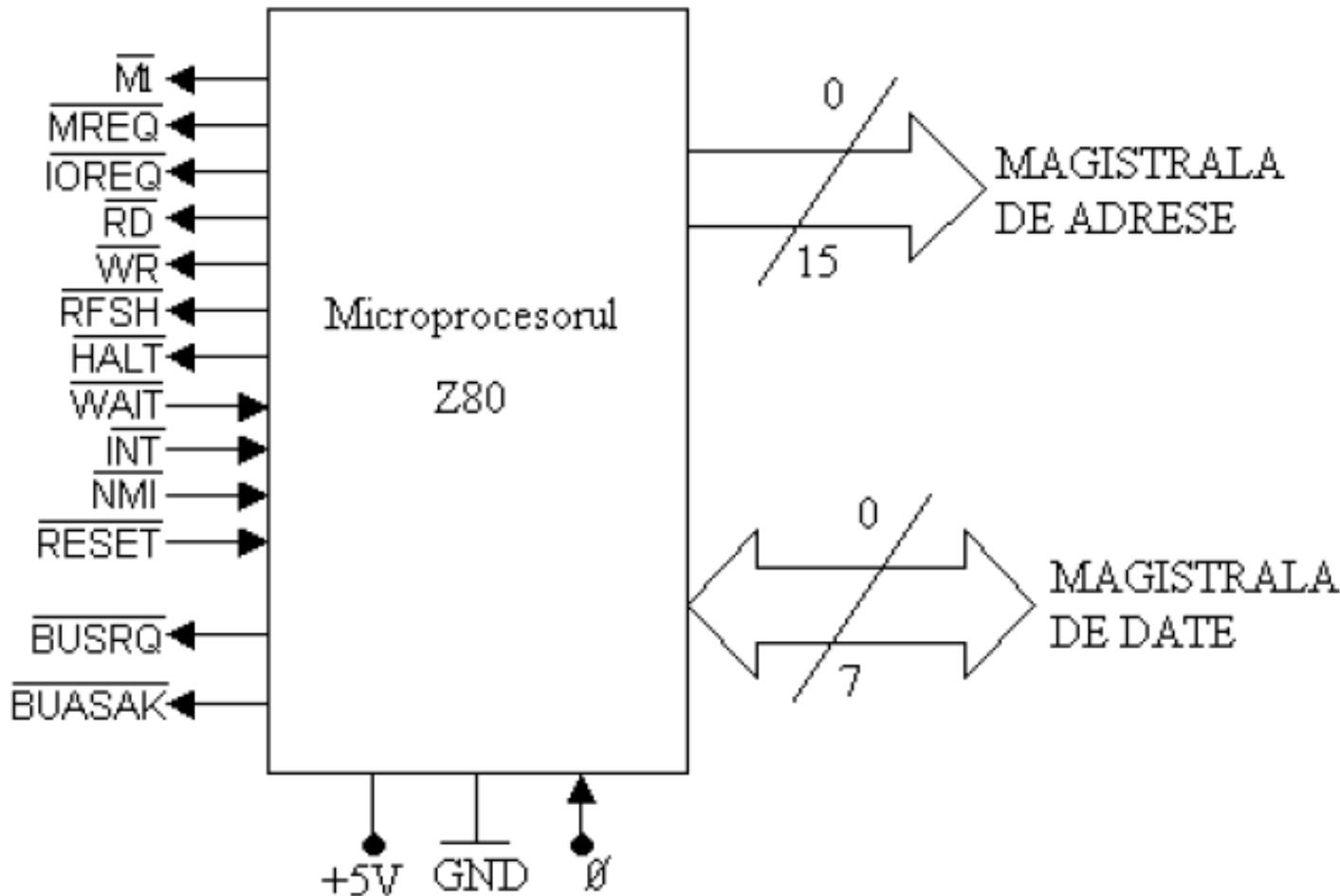
**Multumesc pentru atentie !**

```
.ORG 100H  
LD HL,1234H  
LD A, (HL)  
INC HL  
ADD A, (HL)  
INC HL  
LD (HL), A  
.END  
.ORG 1234H  
.DB 100,200
```

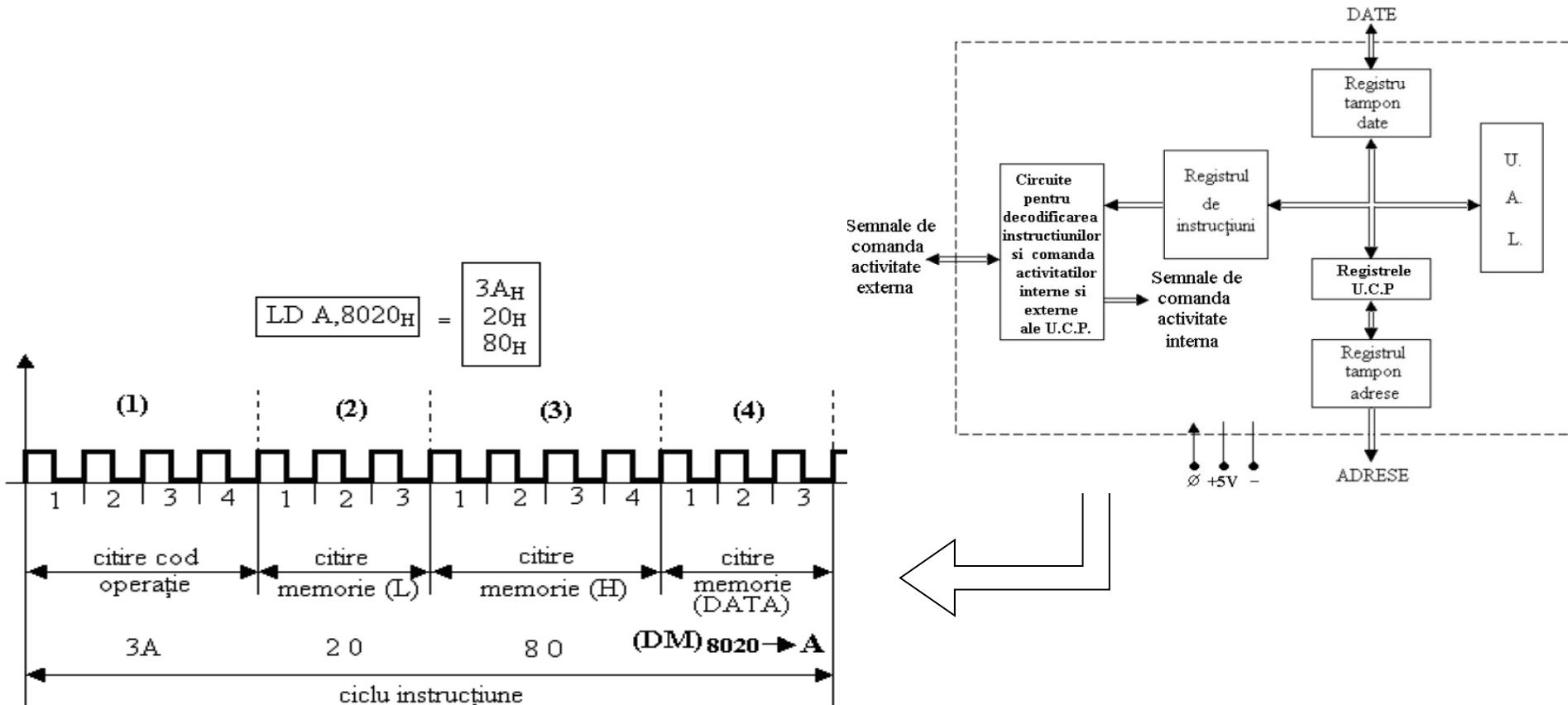


Programare in  
Limba de Asamblare

# Conexiunile externe ale procesorului Z80



# Descrierea ciclurilor mașină



## Ciclul instrucțiune

→ timpul de execuție a unei instrucțiuni și este format la rândul său din cicluri mașină.

## Ciclu mașină

→ succesiune de mai multe stări  $T$  ( $T_1, T_2, \dots, T_n$ ), unde o stare reprezintă perioada de ceas a microsistemului.

# Descrierea ciclurilor masină

Microprocesorul Z80 dispune de :

- cicluri de citire din memorie (extragere cod operație, citire operand, citire date din memorie)
- cicluri de scriere în memorie, care se referă la transferul informației de la microprocesor în memoria *RAM* statică sau dinamică a microsistemului.
- cicluri de citire/scriere din/în porturi de intrare/ieșire, care sunt similare ciclurilor de citire/scriere din/în memorie. În cadrul acestor cicluri se activează *IOREQ* în loc de *MREQ*.

În cazul comunicării cu dispozitivele mai lente se utilizează tehnica de *WA/T* necesară transferului de date între perifericile cu timp de acces mai mare și microprocesor.

# Ciclul de extragere al codului de operatie

Etapele extragerii codului operație :

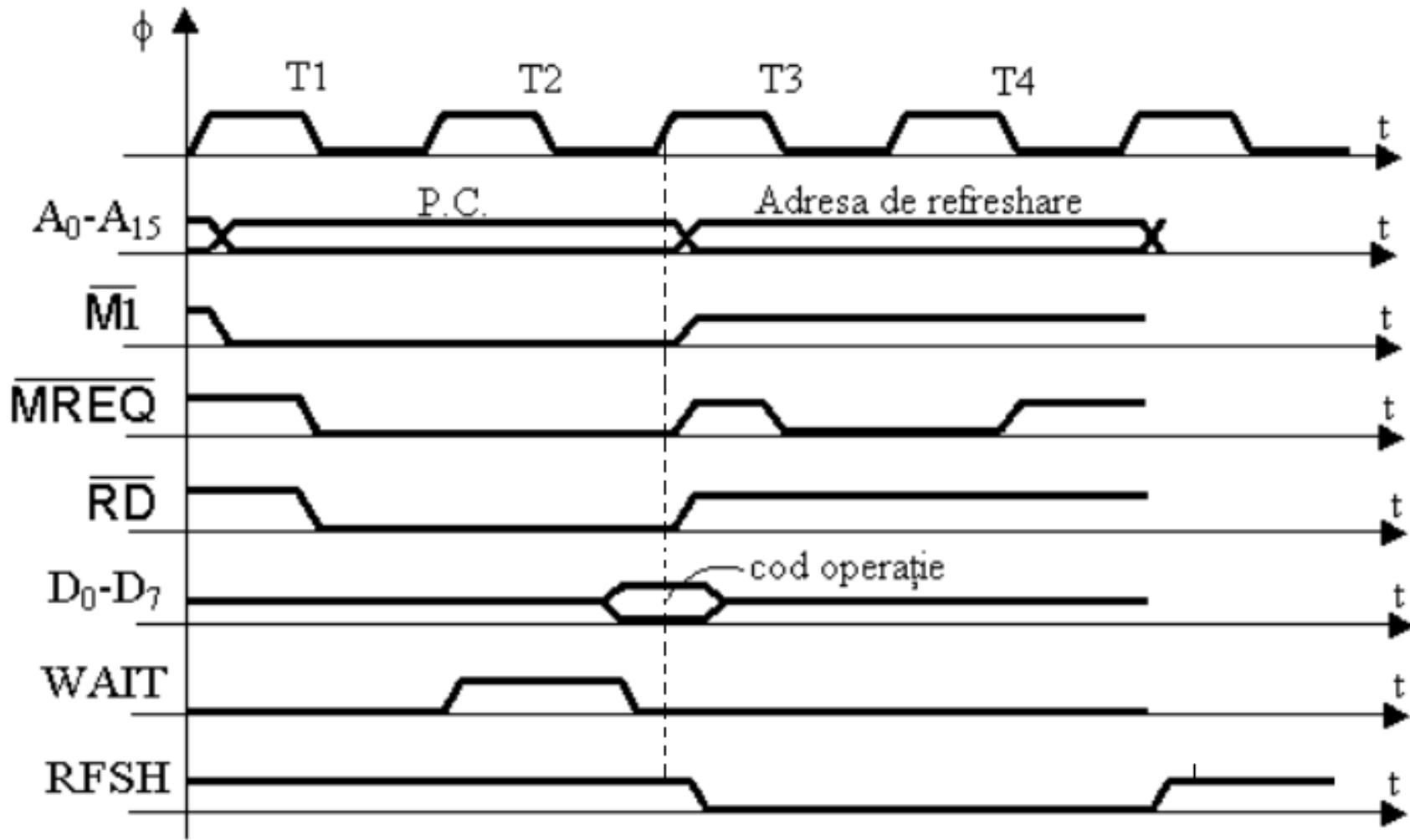
- se plasează pe magistrala de adrese continutul registrului PC, corespunzător instrucțiunii în curs de execuție,
- se activează semnalul  $M1$ , deoarece se extrage un cod de operație,
- se activează  $MREQ$  și  $RD$  și se reține informația pe frontul crescător al perioadei de  $CLK$ .

După extragerea codului operație urmează refreshul memoriei RAM dinamică pe durata a doua perioadă de  $CLK$ .

Seventa de refresh sse realizeaza astfel :

- Se plaseaza adresa de refresh pe magistrala de adrese,
- Se activeaza semnalele  $MREQ$  și  $RFSH$ .

# Ciclul de extragere al codului de operatie



# Cicluri de scriere/citire în/din memorie sau port I/O

Etapele etapelor de citire/scriere în memorie sau porturi I/O:

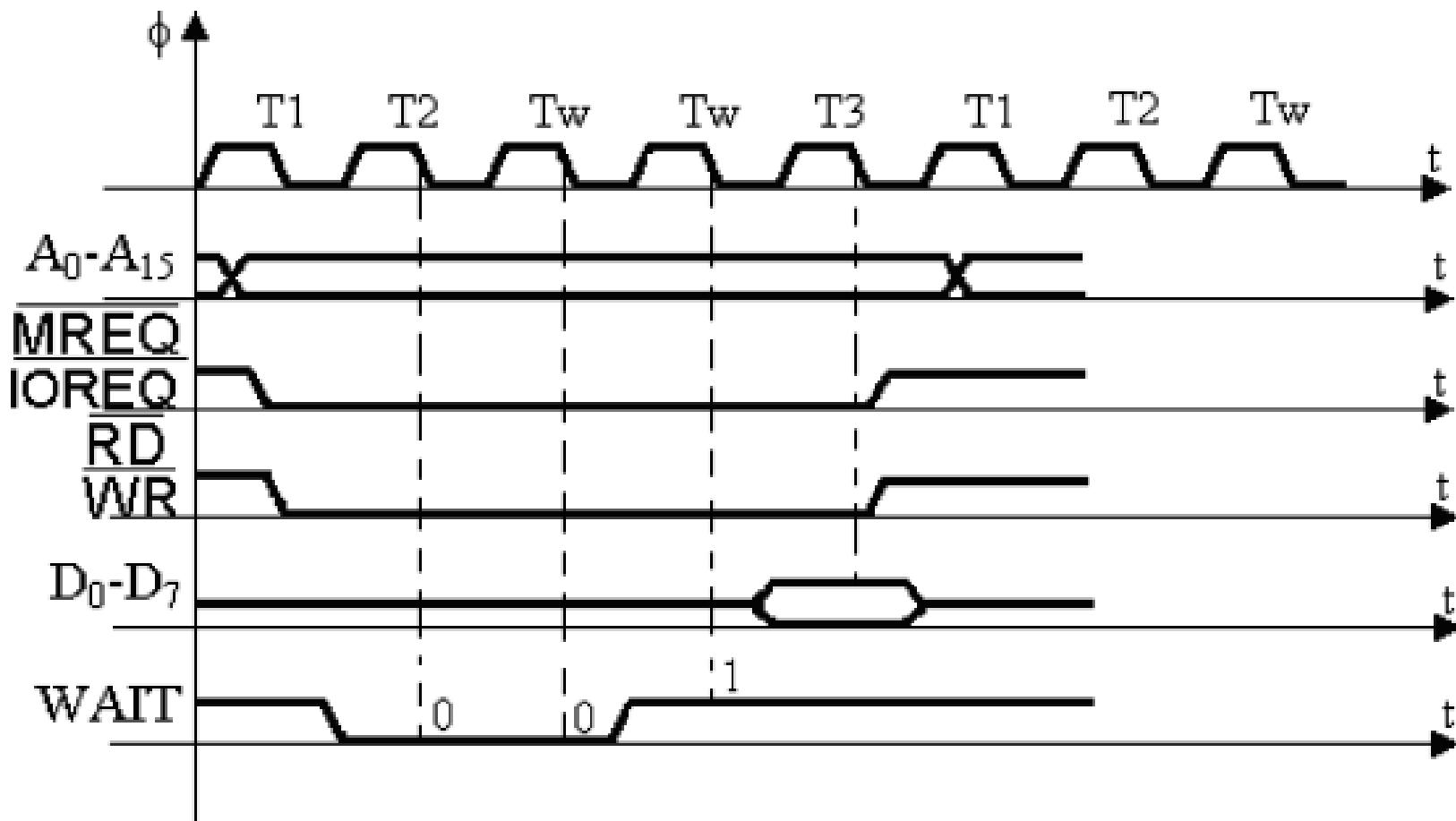
- se testează *WAIT* pe frontul căzător al perioadei de *CLK*,
  - în cazul în care este găsit pe 0 logic se mai introduce încă o stare de așteptare
    - » se testează din nou *WAIT* pe frontul căzător al stării de așteptare până când acesta este pe 1 logic,
    - » după ce se trece *WAIT* pe 1 logic se continuă desfășurarea normală a ciclului mașină.
- transferul datelor are loc pe frontul căzător al perioadei de *CLK* urmatoare
  - realizează tranziția semnalelor de citire/scriere din 0 în 1 logic.

# Cicluri de scriere/citire în/din memorie sau port I/O

În funcție de semnalul care se activează (*MREQ*, *IOREQ*, *RD*, *WR*) microprocesorul execută una dintre operațiile date în tabelul următor.

<i>MREQ</i>	<i>IOREQ</i>	<i>RD</i>	<i>WR</i>	Operația
0	1	0	1	Citire memorie
0	1	1	0	Scriere memorie
1	0	0	1	Citire port
1	0	1	0	Scriere port

# Cicluri de scriere/citire în/din memorie sau port I/O



# **Detalii privind programarea în limbaje de asamblare**

- **Programarea** într-un anumit **limbaj de asamblare** :
  - strâns corelată cu **modul de operare** al **unității de procesare** utilizată,
  - necesită **cunoștințe** privind **hardware-ul** microsistemului.
- **Limbajul de asamblare** presupune :
  - **gestionare riguroasă a resurselor** procesorului,
  - **nivel de abstractizare** scazut comparativ cu majoritatea limbajelor de nivel care nu au **acces direct la resursele hardware** ale platformei de calcul.

# **Detalii privind programarea în limbaje de asamblare**

Diferențele care există între un limbaj de asamblare și un limbaj de nivel înalt Limbajul de asamblare :

- permite accesul la toate resursele unui microprocesor,
- realizarea de aplicații performante
  - » spațiului de memorie folosit,
  - » viteza de execuție;
- programarea necesită o organizare riguroasă,
  - manipularea variabilelor de lucru presupune și cunoașterea locației de memorie la care se află acele variabile.
  - programatorul să cunoască starea tuturor resurselor procesorului
    - » registrii acestuia,
    - » alocarea memoriei,
    - » sistemul de întrerupere,
    - » implicațiile modificării acestor resurse asupra bunei funcționări a întregului microsistem.

# **Detalii privind programarea în limbaje de asamblare**

**Ex.:**

## **Adresarea unui element dintr-un vector**

- În un limbaj de nivel înalt se realizează prin nume și indice,
- În un limbaj de asamblare se realizează printr-o adresare relativă care presupune și operații prin care se determină adresa respectivă.

Registrele prin care se adresează respectivul element nu trebuie folosite și de alte instrucțiuni sau dacă sunt folosite trebuie salvate la acel moment ca apoi să se refacă.

# **Detalii privind programarea în limbaje de asamblare**

Realizarea și rularea unui program în asamblare presupune parcurgerea următoarelor etape:

- **conceperea** algoritmului;
- **scrierea programului** în limbajul de asamblare corespunzător procesorului;
- **translarea fișierului** text care conține programul sursă, în **fișiere obiect relocabile** utilizând *un assembler*;
- **linke-ditarea** fișierelor obiect într-un singur fișier obiect executabil, prin reevaluarea atât a legăturilor interne cât și a legăturilor externe;
- **încărcarea programului** în memoria de lucru a microsistemului;
- **rularea programului** încărcat în memorie,
- **verificarea rezultatului** obținut.

# **Detalii privind programarea în limbaje de asamblare**

Prezentarea unei aplicații ce ilustrează etapele menționate anterior, considerând un exemplu simplu de calcul al sumei unei constante la un vector și apoi memorarea acelui vector la o anumită zonă de memorie.

# Detalii privind programarea in limbaje de asamblare – Exemplu

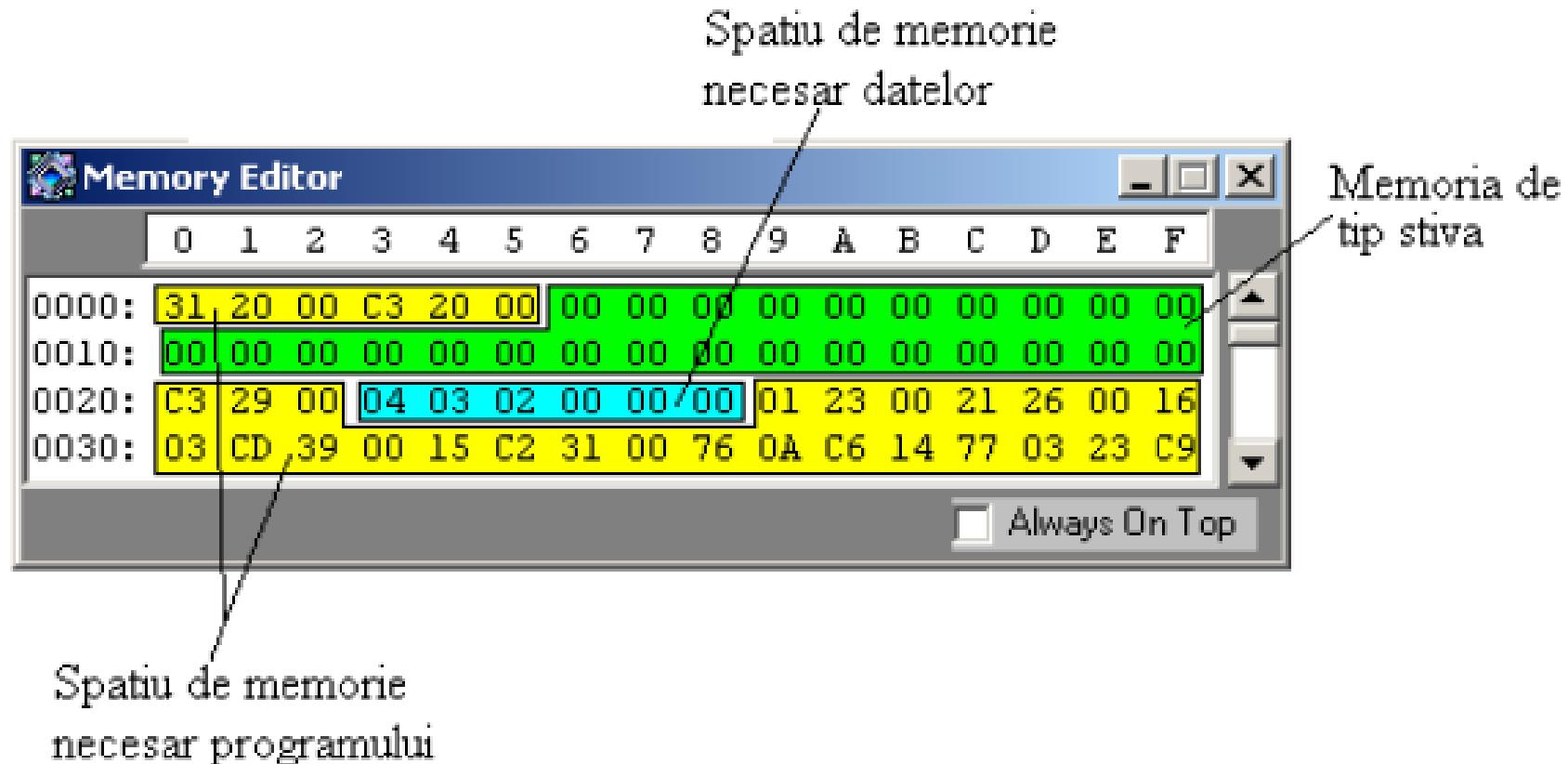
Fisierul <u>Prog . asm</u>	Fisierul <u>Prog . lst</u>	Fisierul <u>Prog . obj</u>
; Ex. de pr. in asamblare Z80 ; Se seteaza indicatorul de stiva LD SP,0020H JP ST	0001 0000 ; Ex. de pr. in asamblare Z80 0002 0000 ; Se seteaza indicatorul de stiva 0003 0000 31 20 00 LD SP, 0020H 0004 0003 C3 20 00 JP ST	0000 31 20 00 0003 C3 20 00
; Se stabileste adresa unde ; se plaseaza codul programului .ORG 0020H ST: JP LP1	0005 0006 ; Se stabileste adresa unde 0006 0006 ; se plaseaza codul programului 0007 0020 .ORG 0020H 0008 0020 C3 29 00 ST: JP LP1	0009 00 000A 00 000B 00 000C 00 000D 00 000E 00
; Se definesc datele de intrare LD1: .DB 04H .DB 03H .DB 02H	0009 0023 ; Se definesc datele de intrare 0010 0023 04 LD1: .DB 04H 0011 0024 03 .DB 03H 0012 0025 02 .DB 02H	000F 00 0010 00 0011 00 0012 00 0013 00 0014 00
; Se defineste spatiul destinatie SD1: .DB 00H .DB 00H .DB 00H	0013 0026 ; Se defineste spatiul destinatie 0014 0026 00 SD1: .DB 00H 0015 0027 00 .DB 00H 0016 0028 00 .DB 00H	0015 00 0016 00 0017 00 0018 00 0019 00 001A 00

# Detalii privind programarea in limbaje de asamblare - Exemplu

<pre> ; Se seteaza constanta CT1 CT1 .EQU 14H LP1: LD BC,LD1       LD HL,SD1       LD D,03H; LP2: CALL R1       DEC D       JP NZ,LP2       HALT     </pre>	<pre> 0017 0029 ; Se seteaza constanta CT1 0018 0029 CT1 .EQU 14H 0019 0029 01 23 00 LP1: LD BC,LD1 0020 002C 21 26 00 LD HL,SD1 0021 002F 16 03 LD D,03H; 0022 0031 CD 39 00 LP2: CALL R1 0023 0034 15 DEC D 0024 0035 C2 31 00 JP NZ,LP2 0025 0038 76 HALT     </pre>	<pre> 001B 00 001C 00 001D 00 001E 00 001F 00 0020 C3 29 00 0023 04 0024 03 0025 02 0026 00 0027 00 0028 00 0029 01 23 00 002C 21 26 00 002F 16 03 0031 CD 39 00 0034 15 0035 C2 31 00 0038 76 0039 0A 003A C6 14 003C 77 003D 03 003E 23 003F C9     </pre>
<pre> ; Rutina care aduna la data ; din memorie de la adresa (BC) ; contanta CT1 si o depune ; la adresa de memorie (HL) </pre>	<pre> 0026 0039 ; Rutina care aduna la data 0027 0039 ; din memorie de la adresa (BC) 0028 0039 ; contanta CT1 si o depune 0029 0039 ; la adresa de memorie (HL)     </pre>	<pre> R1: LD A,(BC)       ADD A,CT1       LD (HL),A       INC BC       INC HL       RET       .END     </pre>
	<pre> 0030 0039 0A 0031 003A C6 14 0032 003C 77 0033 003D 03 0034 003E 23 0035 003F C9 0036 0040     </pre>	<pre> R1: LD A,(BC)       ADD A,CT1       LD (HL),A       INC BC       INC HL       RET       .END     </pre>

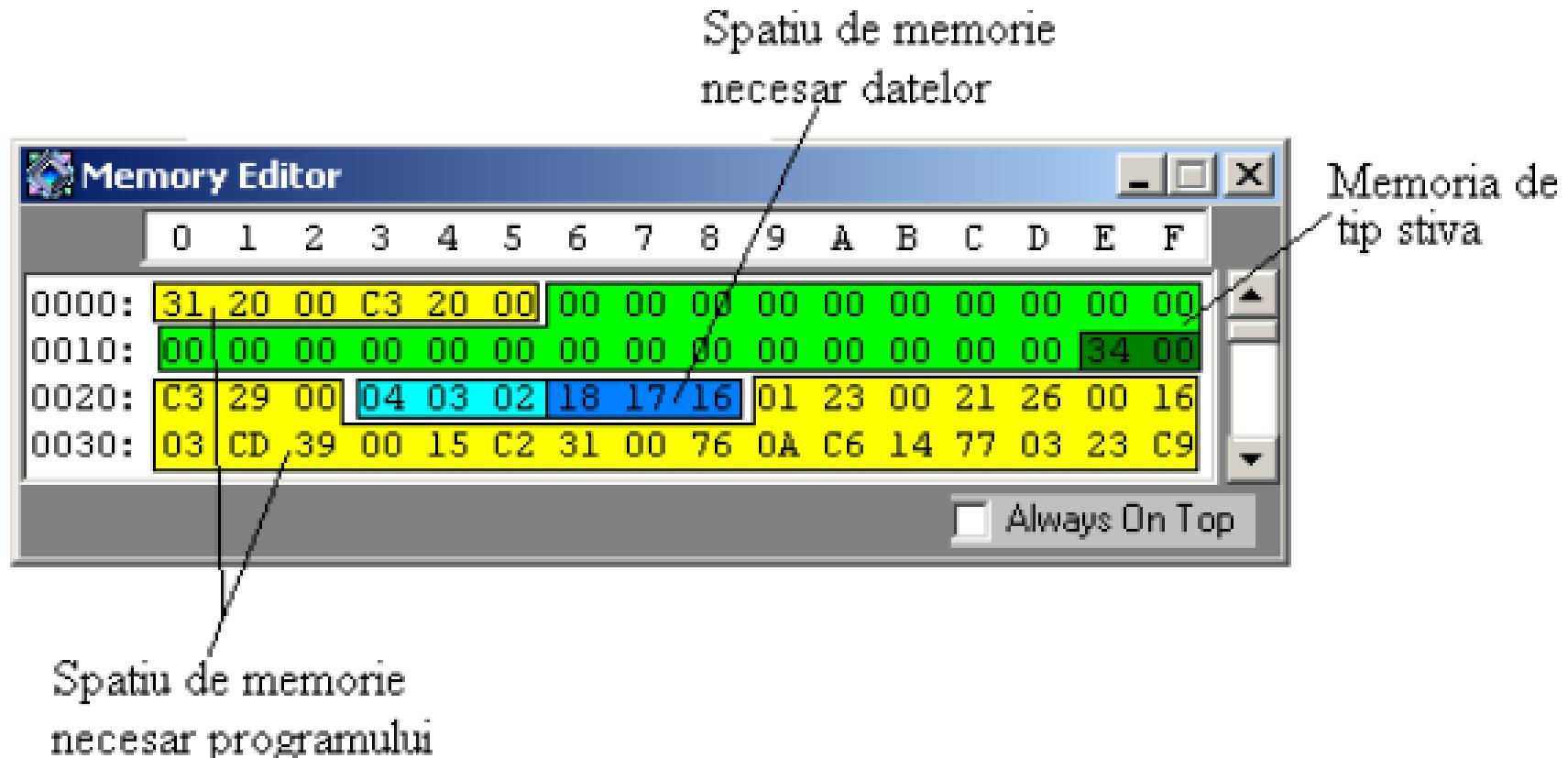
# Detalii privind programarea in limbaje de asamblare - Rezultate

Spatiul de memorie (RAM+ROM) necesare executiei programului inainte de executie



# Detalii privind programarea in limbaje de asamblare - Rezultate

Spatiul de memorie (RAM+ROM) necesare executiei programului dupa executie

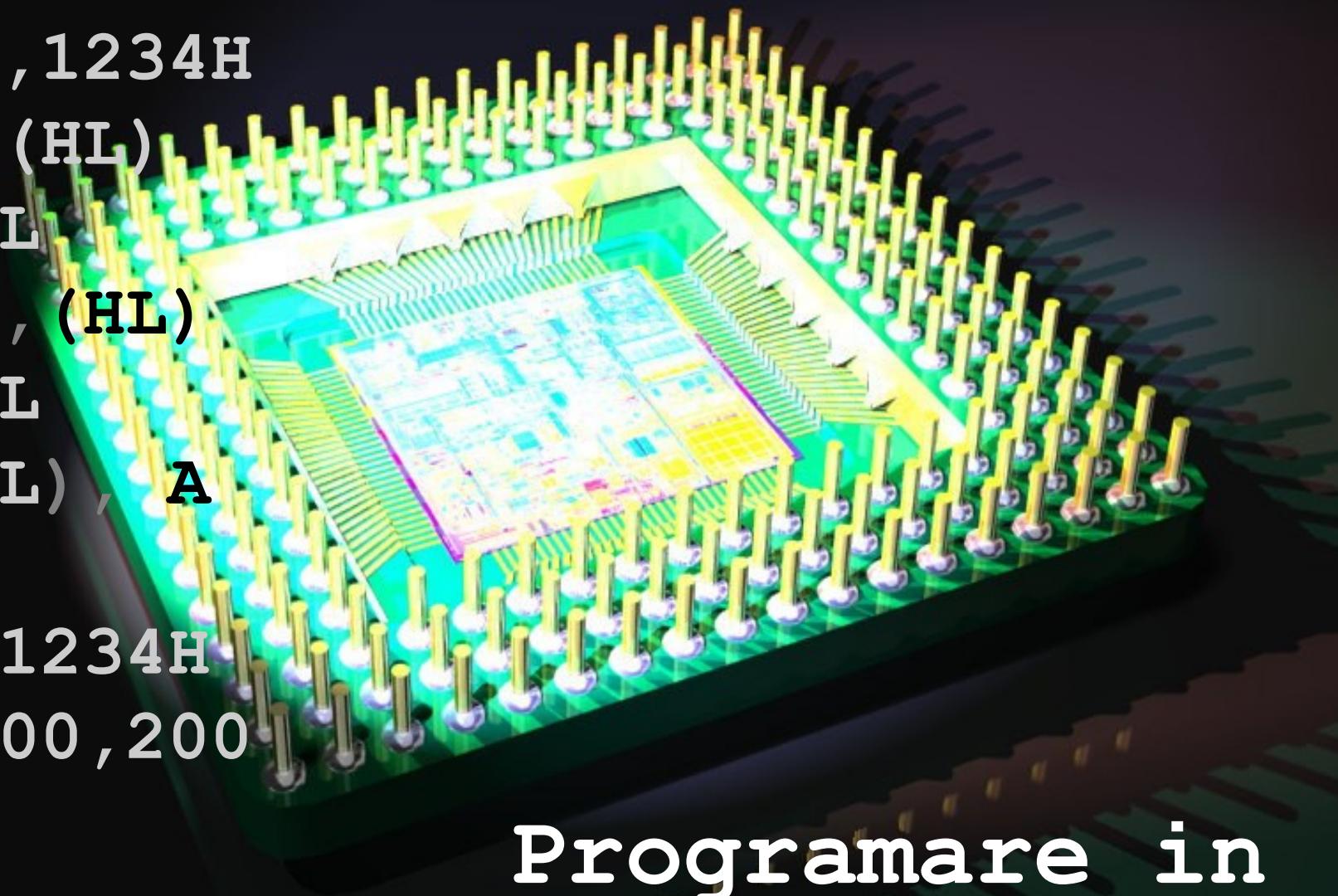


**010011010111010101101100011101000111  
01010110110101100101011100110110001  
10010000001110000011001010110111001  
11010001110010011101010010000001100  
00101110100011001010110111001110100  
0110100101100101001000000100001**

**sau altfel spus ...**

**Multumesc pentru atentie !**

```
.ORG 100H  
LD HL,1234H  
LD A, (HL)  
INC HL  
ADD A, (HL)  
INC HL  
LD (HL), A  
.END  
.ORG 1234H  
.DB 100,200
```



Programare in  
Limbaș de Asamblare

# **Modurile de adresare ale micropresorului Z80**

**Microprocesorul Z80 dispune de un set de instructiuni mult mai performant decât celelalte microprocesoare de 8 biti.**

- **158 tipuri de instructiuni** care includ și
- **78 de instructiuni** ale micropresorului **INTEL 8080** la nivel de cod obiect.

**Instructiuni puternice ale micropresorului Z80:**

- **transfer si comparare la nivel de blocuri** de date
- **prelucrare la nivel de bit.**

# Modurile de adresare ale micropresorului Z80

Instructiunile microprocesorului se grupează în următoarele tipuri de instructiuni:

- instructiuni de transfer pe 8 biti;
- instructiuni de transfer pe 16 biti;
- instructiuni de transfer pe blocuri și căutări în interiorul blocurilor de memorie;
- instructiuni specifice operatiilor aritmetice și logice pe 8 biti;
- instructiuni corespunzătoare operatiilor de comandă a U.C.P.;
- instructiuni de rotatie și deplasare;
- instructiuni de salt;
- instructiuni apel de subrutine și revenire în programul principal;
- instructiuni specifice operatiilor de intrare/iesire.

# Modurile de adresare ale micropresorului Z80

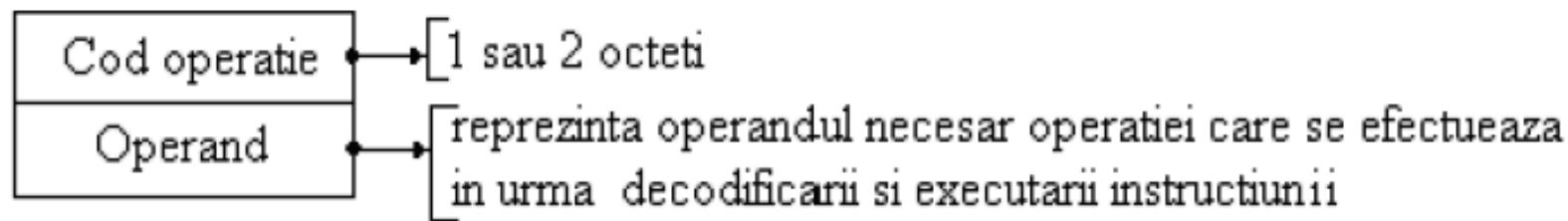
Modurile de adresare ale micropresorului Z80 permit un **transfer eficient între registre, memoria externă și dispozitivele periferice.**

În cele ce urmează se prezintă modurile de adresare ale micropresorului Z80 insotite de exemple care să surprinda specificul fiecarui mod de adresare.

# Modurile de adresare ale microp procesorului Z80

## Adresare imediată

### Structura instrucțiunii

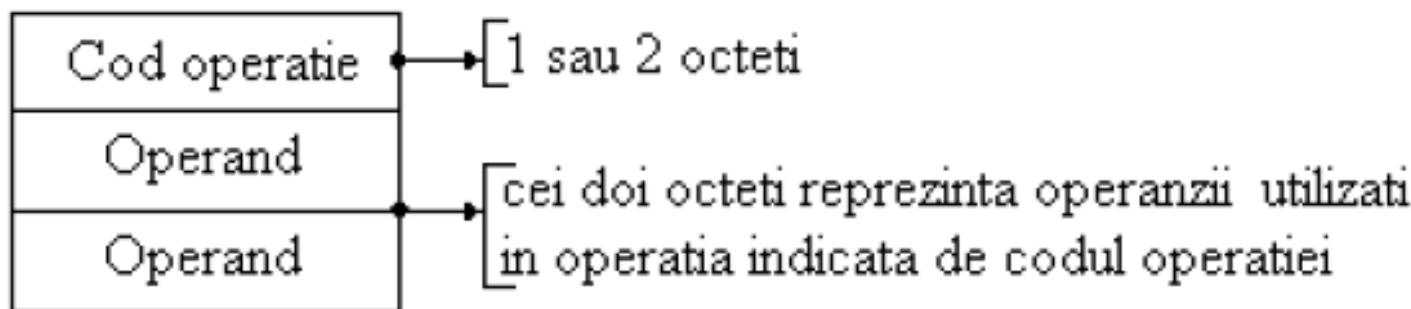


Exemplu: LD A, 05     *efect*     $A \leftarrow 05_H$   
          ADD A, 30    *efect*     $A \leftarrow A + 30_H$

# Modurile de adresare ale microp procesorului Z80

## Adresare imediată extinsă

Structura instrucțiunii:

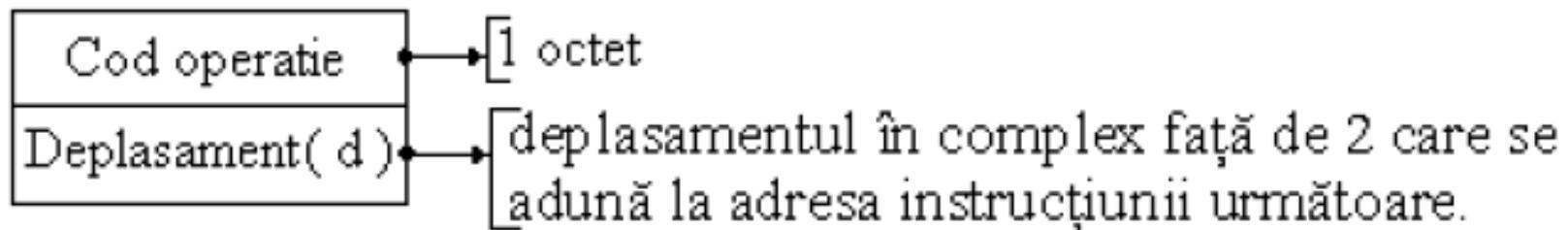


Exemplu: LD HL, nn    *efect*     $HL \leftarrow nn$

# Modurile de adresare ale microp procesorului Z80

## Adresare relativă

Structura instrucțiunii:

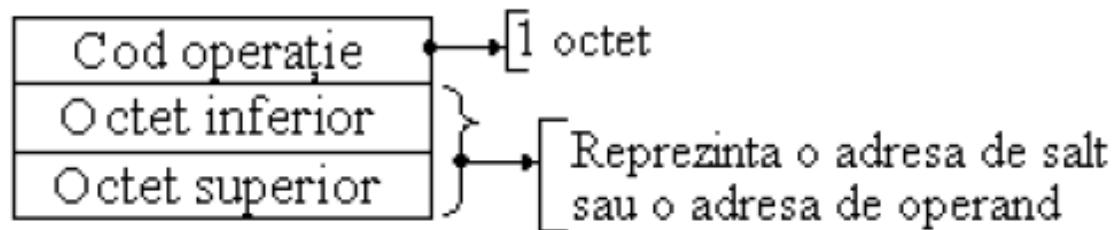


Exemplu:    JR Z, e              *efect: dacă Z = 0, atunci continuă*  
    : dacă Z = 1, atunci  $(PC + d) \rightarrow PC$

# Modurile de adresare ale microp procesorului Z80

## Adresare extinsă

Structura instrucțiunii:

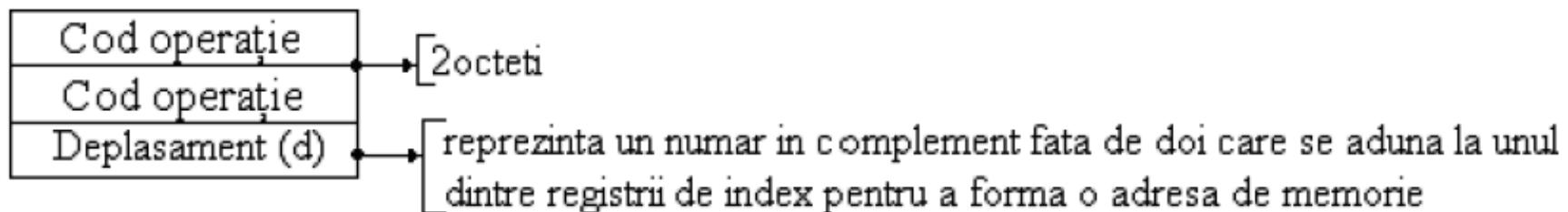


Exemplu:    JP nn                **efect: salt la adresa nn**  
                LD (nn),A            **efect: transferă data din registrul A în locația de memorie de la adresa nn;**

# Modurile de adresare ale microp procesorului Z80

## Adresare indexată

Structura instrucției:

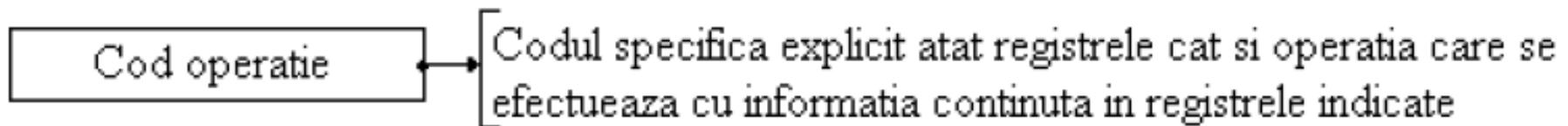


Exemplu: LD (Ix+d),n      *efect: OM<sub>(Ix+d)</sub> ← n*

# Modurile de adresare ale microp procesorului Z80

## Adresare la regisztr

Structura instrucțiunii:

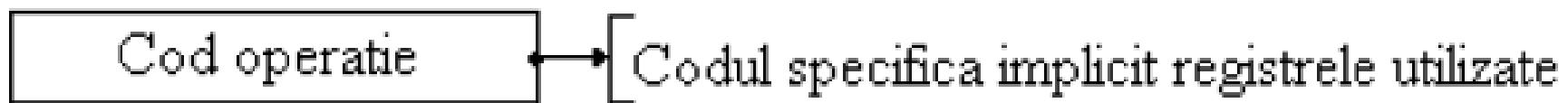


Exemplu: LD B,C      *efect*    **B** ← **C**

# Modurile de adresare ale microp procesorului Z80

## Adresare implicită

Structura instrucțiunii:

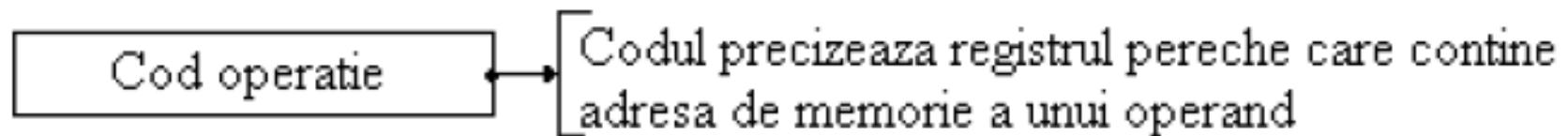


Exemplu:    XOR A                *efect*     $A \leftarrow A \oplus A$

# Modurile de adresare ale microp procesorului Z80

## Adresare indirectă cu registre

Structura instrucțiunii:

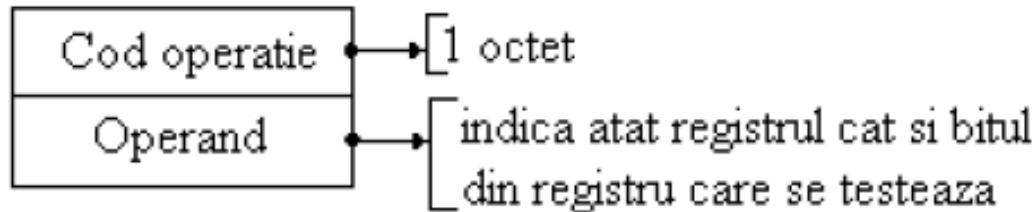


Exemplu:    ADD A,(HL)              *efect*     $A \leftarrow A + M_{(HL)}$

# Modurile de adresare ale microp procesorului Z80

## Adresare pe bit

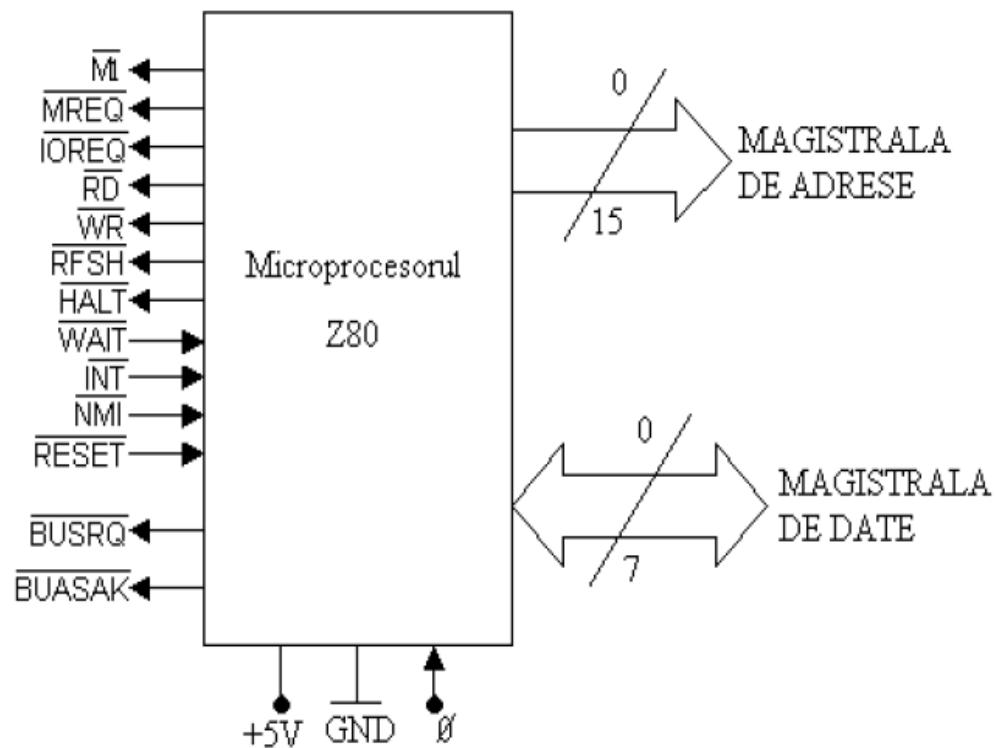
Structura instrucțiunii:



Exemplu: BIT 6,D      *efect* testeaza bitul 6 al regisztrului D si modifica in mod corespunzator flagul Z

# Studiul cererilor de magistrală

Acest mod de lucru ale microprocesorului contribuie la **cresterea flexibilității** microprocesorului deoarece cererile de magistrală și întreruperile permit **comunicarea într-un mod asincron cu alte dispozitive și la initiativa acestora.**

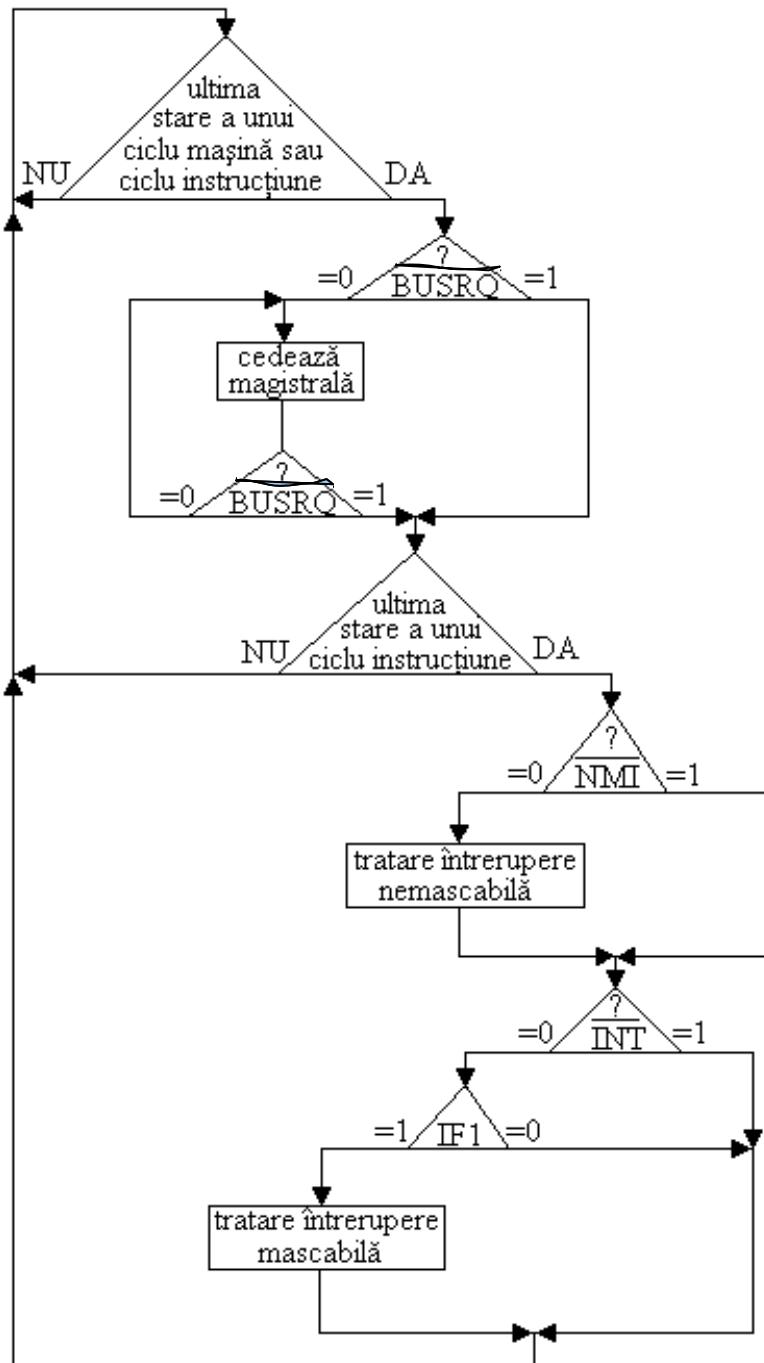


# Studiul cererilor de magistrală

Avantaje :

- se degrevează microprocesorul de anumite **operatii** care trebuie făcute **ciclic**,
- se degrevează microprocesorul de anumite **operatii** care trebuie făcute **la intervale de timp relativ mici**,
- microprocesorul le **tratează într-o ordine bine stabilită**.

**Organograma** prin care se ilustrează modul de tratare a **evenimentelor generate de dispozitivele periferice** (cererile de magistrală și intreruperile) este dată în figura următoare.



# Studiul cererilor de magistrală

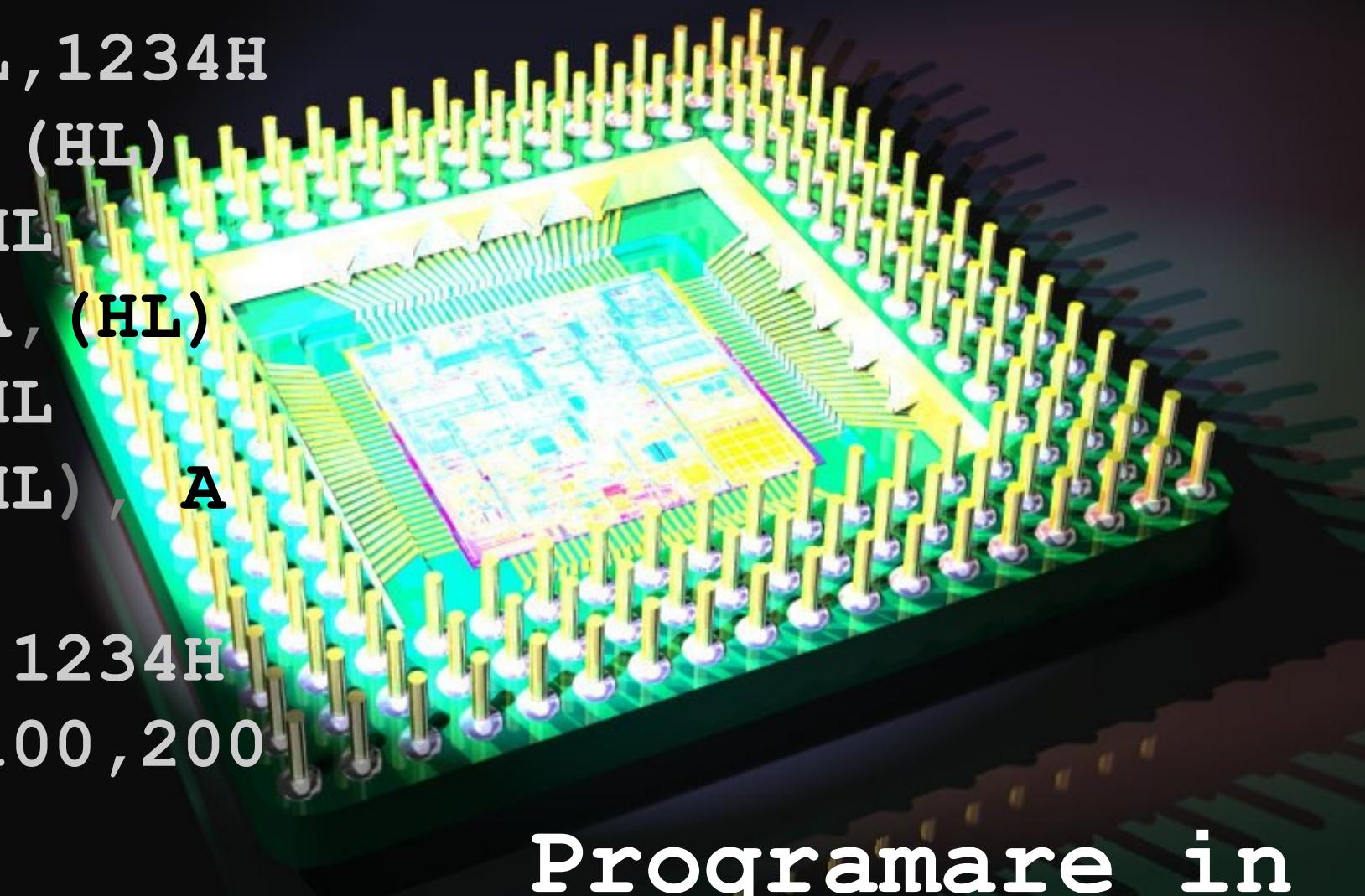
- **Cererile de magistrală** sunt prioritare și se tratează la sfârșitul ciclurilor masină.
- **Întreruperile nemascabile** sunt preluate în orice moment și se tratează la sfârșitul instructiunii în curs de execuție,
- **Întreruperile mascabile** au prioritatea cea mai mică și sunt preluate și tratate la sfârșitul instructiunii în curs de execuție, numai dacă acestea (întreruperile nemascabile) au fost activate în prealabil ( $IF1=1$ ).

**0100110101110101011000111010001110101  
01101101011001010111001101100011001000  
00011100000110010101101110011101000111  
00100111010100100000011000010111010001  
10010101101110011101000110100101100101  
0010000000100001**

**sau altfel spus ...**

**Multumesc pentru atentie !**

```
.ORG 100H  
LD HL,1234H  
LD A, (HL)  
INC HL  
ADD A, (HL)  
INC HL  
LD (HL), A  
.END  
.ORG 1234H  
.DB 100,200
```



Programare in  
Limba de Asamblare

# DMA (Direct Memory Access)

## DMA :

- Ofera un **mod eficient si avantajos de transfer de date** pe magistrala unui sistem intre un **port de interfata** al unui periferic si **unitatea de memorie** a sistemului
- **Degreveaza** procesorul de efectuarea unor **transferuri de date** care pot mari **latenta** in executia programului

# DMA (Direct Memory Access)

## Fara DMA:

- UCP a procesorului **intervine** prin utilizarea **registrelor interne** ca **memorie intermediara**
- **Transfer** periferic/memorie in **2 faze**:
  - Transfer port/memorie in accumulatorul procesorului
  - Transfer din accumulator in memorie/port

# DMA (Direct Memory Access)

**Solutia** pentru sistemele cu Z80:

- circuitul specializat **LSI8257** pentru **acces direct la memorie (DMA)**
- **elimina interventia procesorului** de pe magistrala sistemului
- **dirijeaza transferurile** periferic-memorie prin **generarea simultana de semnale de R/W** pentru memorie si port
- genereaza **adrese de memorie la locatii successive** pentru transferuri pe sir de caractere
- genereaza **semnale de selectie** pentru port

# DMA cu LSI8257

## Caracteristici:

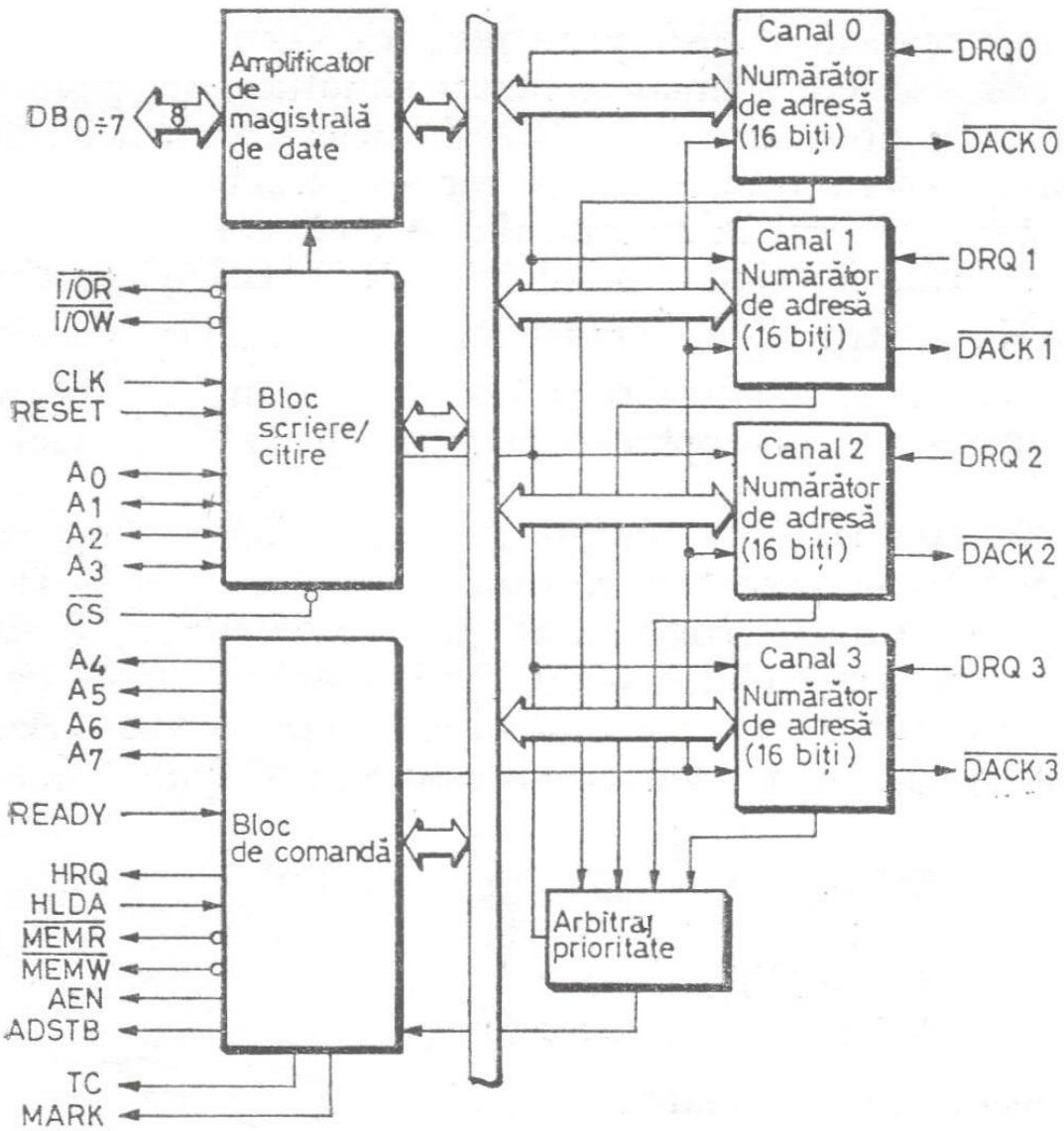
- 4 canale DMA cu operare simultana
- accesul la magistrala se realizeaza dialogand cu procesorul folosind semnalele BUSRQ/BUSAQ
- transferuri successive in bloc
- transferuri octeti individuali
- viteza maxima corespunde transferului unui octet la fiecare 4T, unde T este perioada semnalului de sincronizare a procesorului ( $T \sim 0.32\text{us} \rightarrow f \sim 800\text{kHz}$ )

# DMA cu LSI8257

## Arhitectura internă:

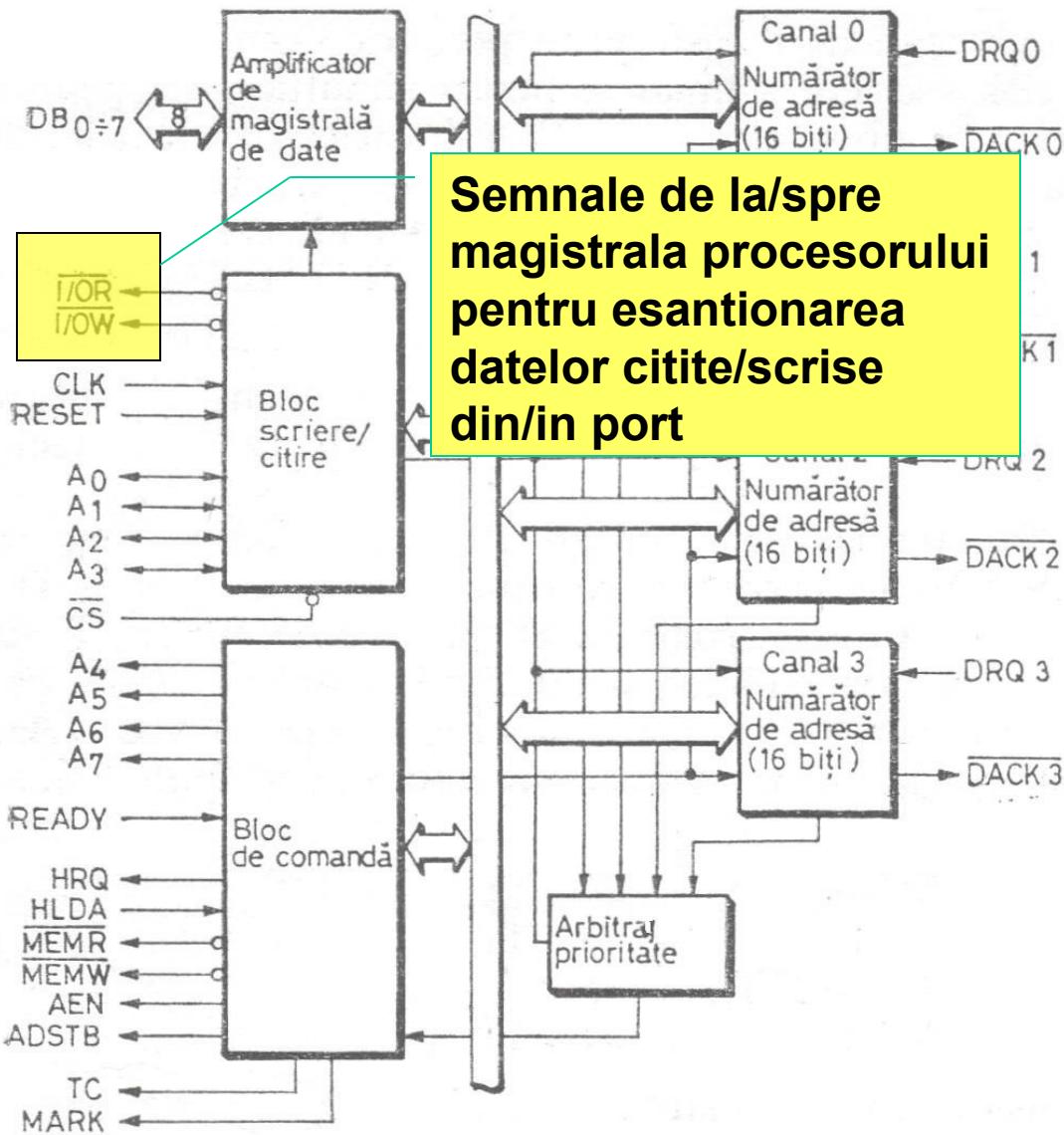
- **magistrala internă** pentru transferuri de date
- **circuite de interfata :**
  - circuit de **interfata cu procesorul** sistemului
  - circuite pentru **dialogul de preluare/eliberaare magistrala internă**
  - circuite de **interfata cu perifericele si registrele de canal**
  - circuite pentru **stabilirea prioritatilor intre canale**

# DMA cu LSI8257



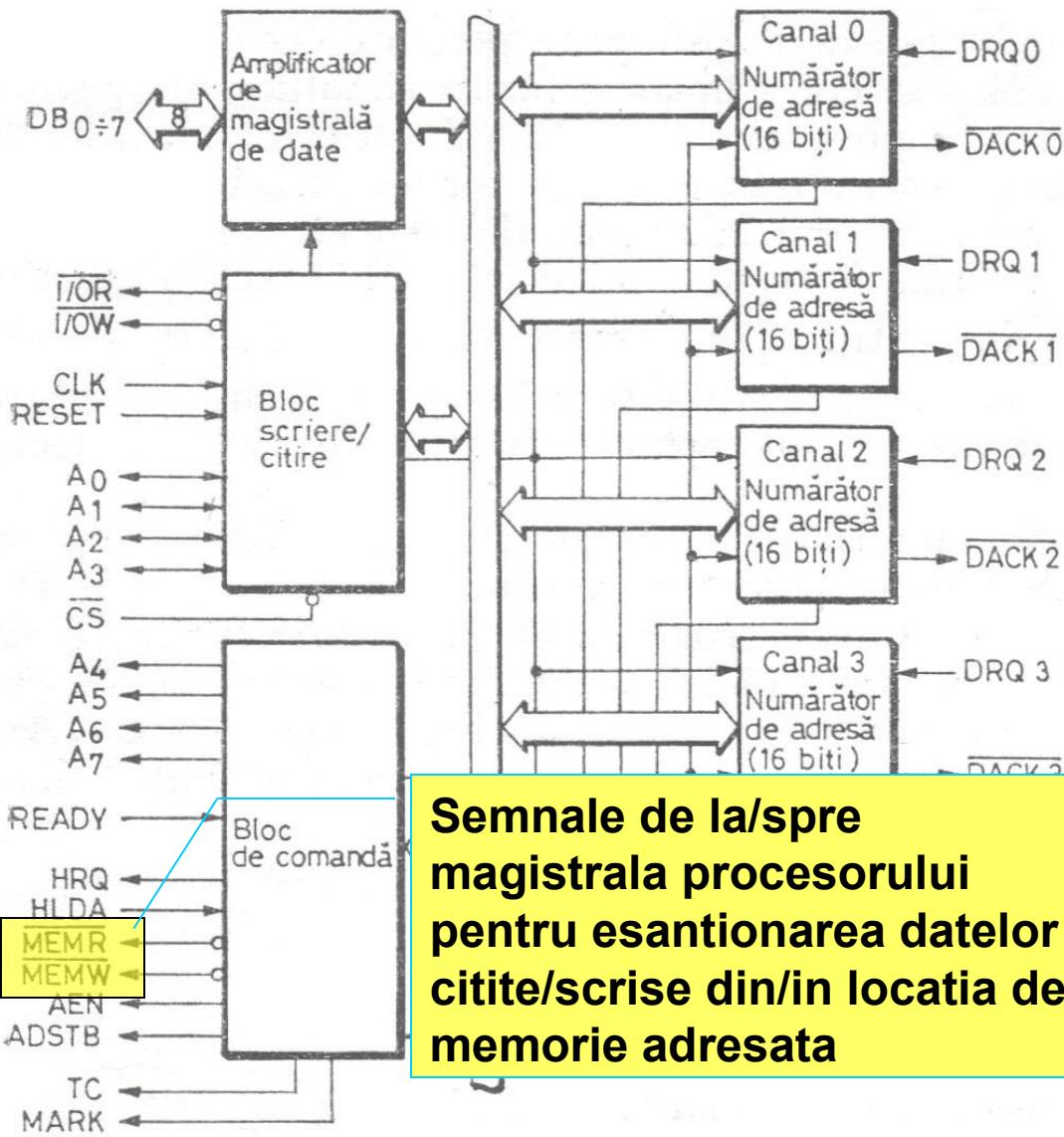
I/OR	1	40	A7
I/OW	2	39	A6
MEMR	3	38	A5
MEMW	4	37	A4
MARK	5	36	TC
READY	6	35	A3
HLDA	7	34	A2
ADSTB	8	33	A1
AEN	9	8257	A0
HRQ	10	32	VCC
CS	11	30	D0
CLK	12	29	D1
RESET	13	28	D2
DACK 2	14	27	D3
DACK 3	15	26	D4
DRQ 3	16	25	DACK 0
DRQ 2	17	24	DACK 1
DRQ 1	18	23	D5
DRQ 0	19	22	D6
GND	20	21	D7

# DMA cu LSI8257



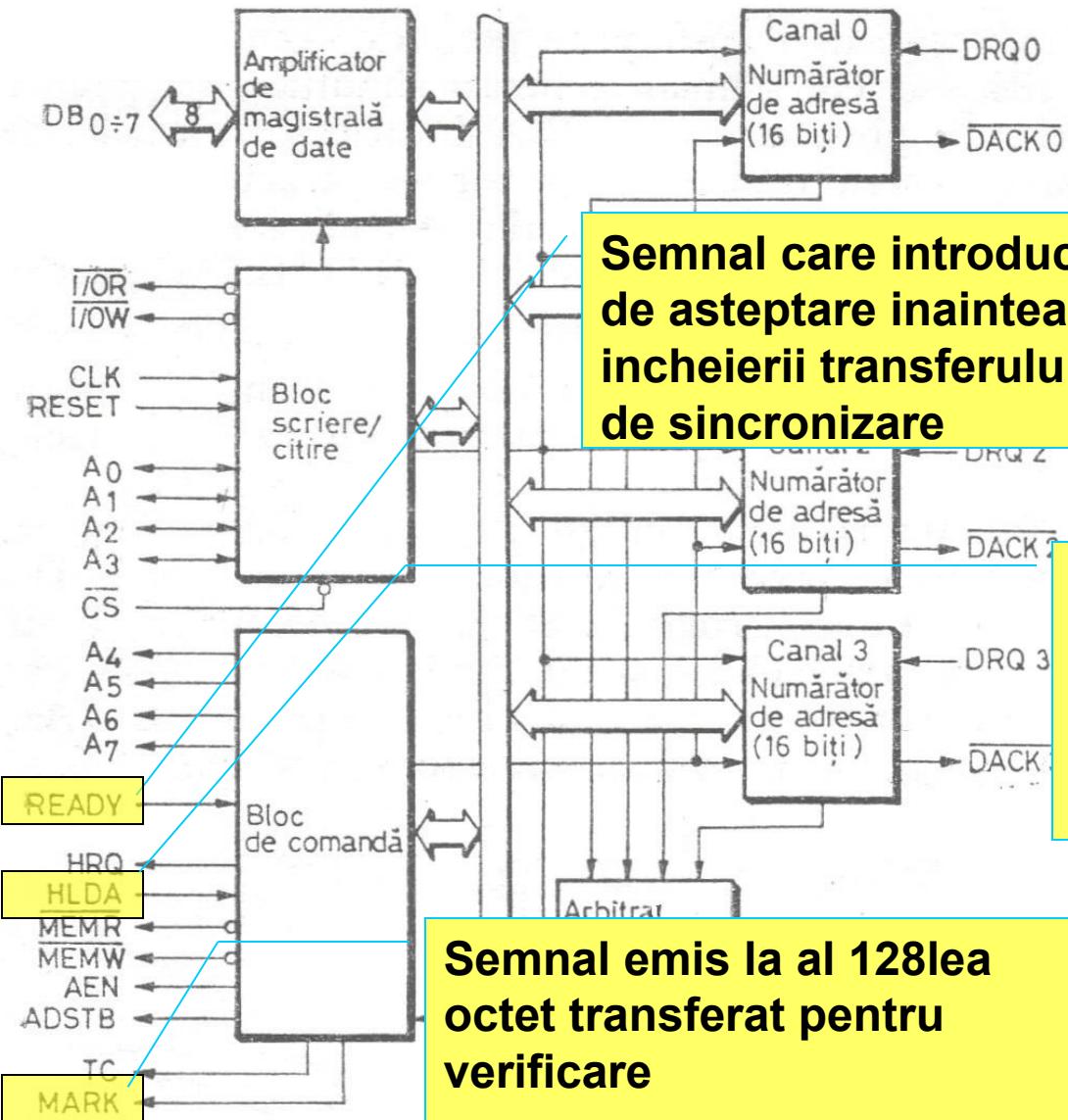
I/OR	1	40	A7
I/OW	2	39	A6
MEMR	3	38	A5
MEMW	4	37	A4
MARK	5	36	TC
READY	6	35	A3
HLDA	7	34	A2
ADSTB	8	33	A1
AEN	9	8257	A0
HRQ	10	32	VCC
CS	11	30	D0
CLK	12	29	D1
RESET	13.	28	D2
DACK 2	14	27	D3
DACK 3	15	26	D4
DRQ 3	16	25	DACK 0
DRQ 2	17	24	DACK 1
DRQ 1	18	23	D5
DRQ 0	19	22	D6
GND	20	21	D7

# DMA cu LSI8257



I/OR	1	40	A7
I/OW	2	39	A6
MEMR	3	38	A5
MEMW	4	37	A4
MARK	5	36	TC
READY	6	35	A3
HLDA	7	34	A2
ADSTB	8	33	A1
AEN	9	8257	A0
HRQ	10	32	VCC
CS	11	30	D0
CLK	12	29	D1
RESET	13	28	D2
ACK 2	14	27	D3
ACK 3	15	26	D4
DRQ 3	16	25	DACK 0
DRQ 2	17	24	DACK 1
DRQ 1	18	23	D5
DRQ 0	19	22	D6
GND	20	21	D7

# DMA cu LSI8257

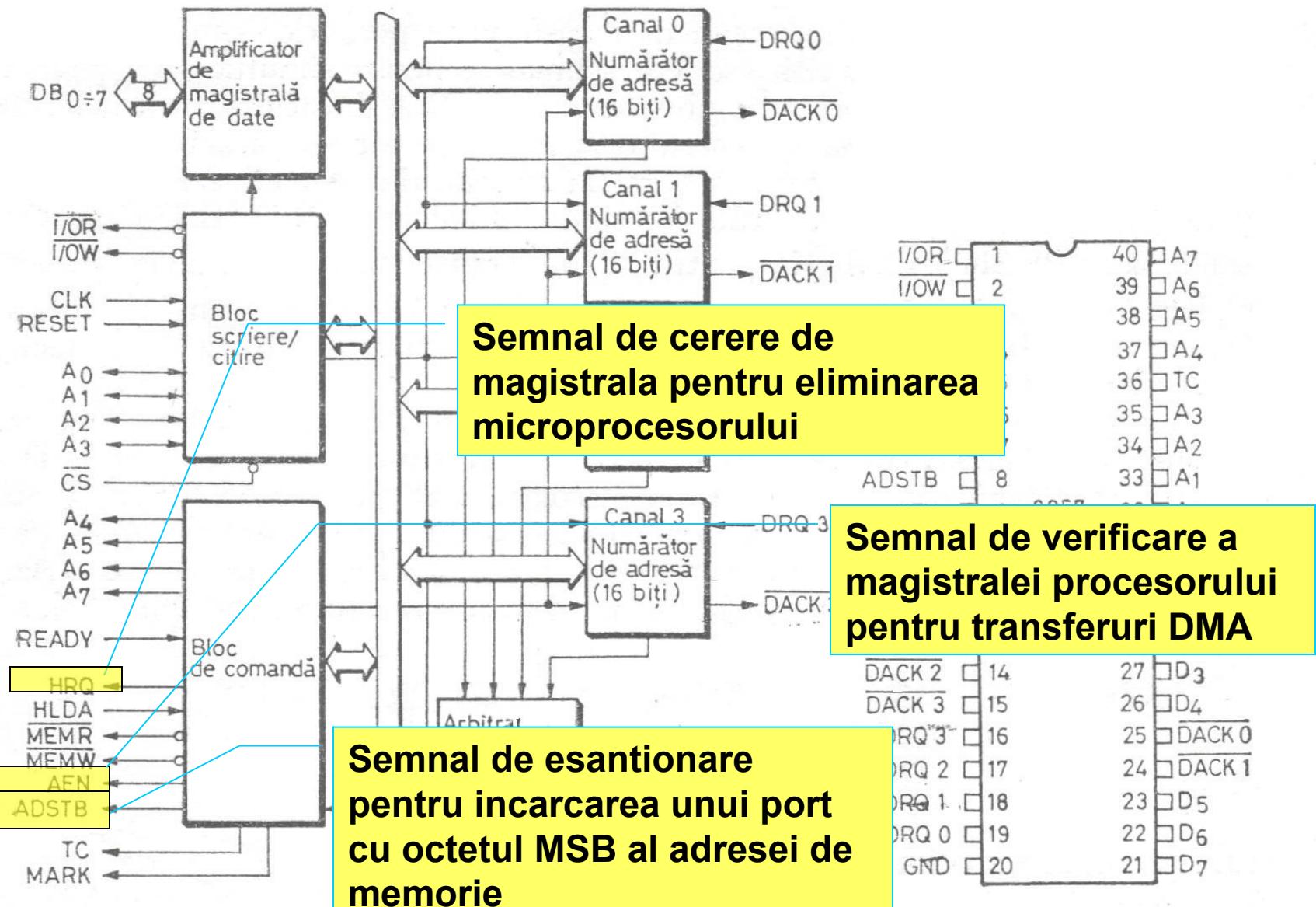


40	A7
39	A6
38	A5
37	A4
36	TC
35	A3

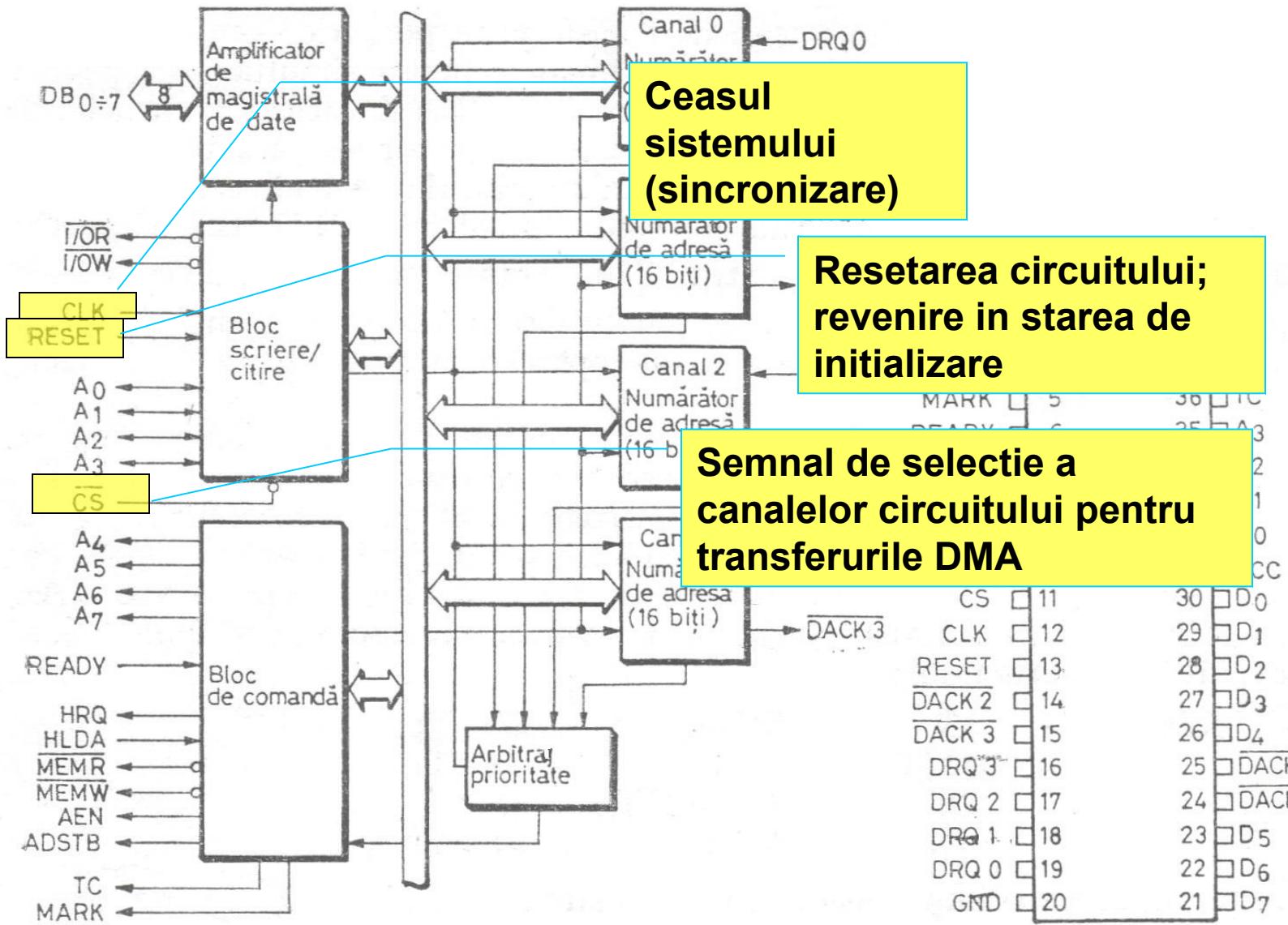
**Semnal primit de la magistrala procesorului pentru a anunta eliberarea acestiei pentru transferurile DMA**

DACK 2	□ 14	27	□ U3
DACK 3	□ 15	26	□ D4
DRQ 3	□ 16	25	□ DACK 0
DRQ 2	□ 17	24	□ DACK 1
DRQ 1	□ 18	23	□ D5
DRQ 0	□ 19	22	□ D6
GND	□ 20	21	□ D7

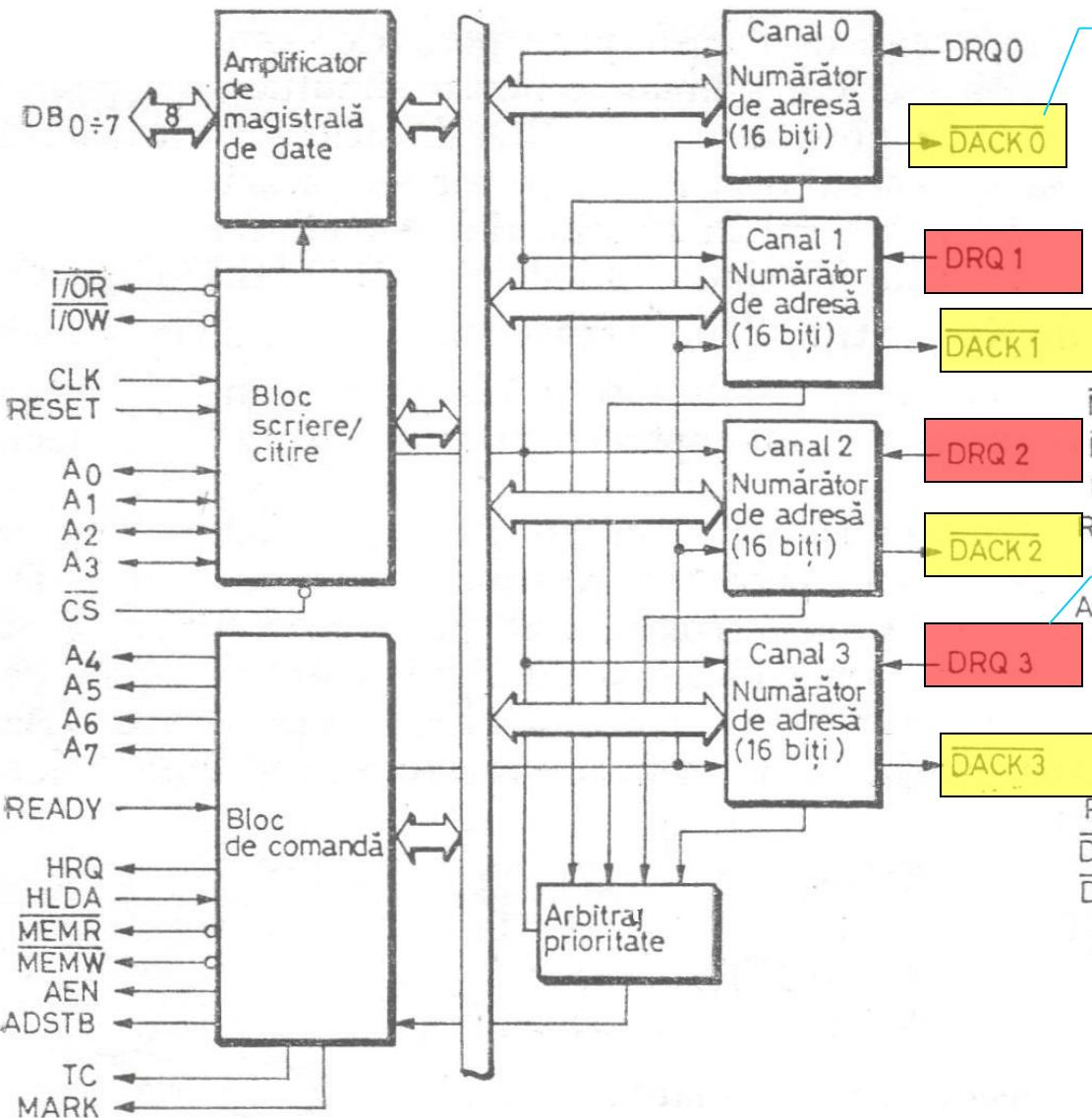
# DMA cu LSI8257



# DMA cu LSI8257



# DMA cu LSI8257



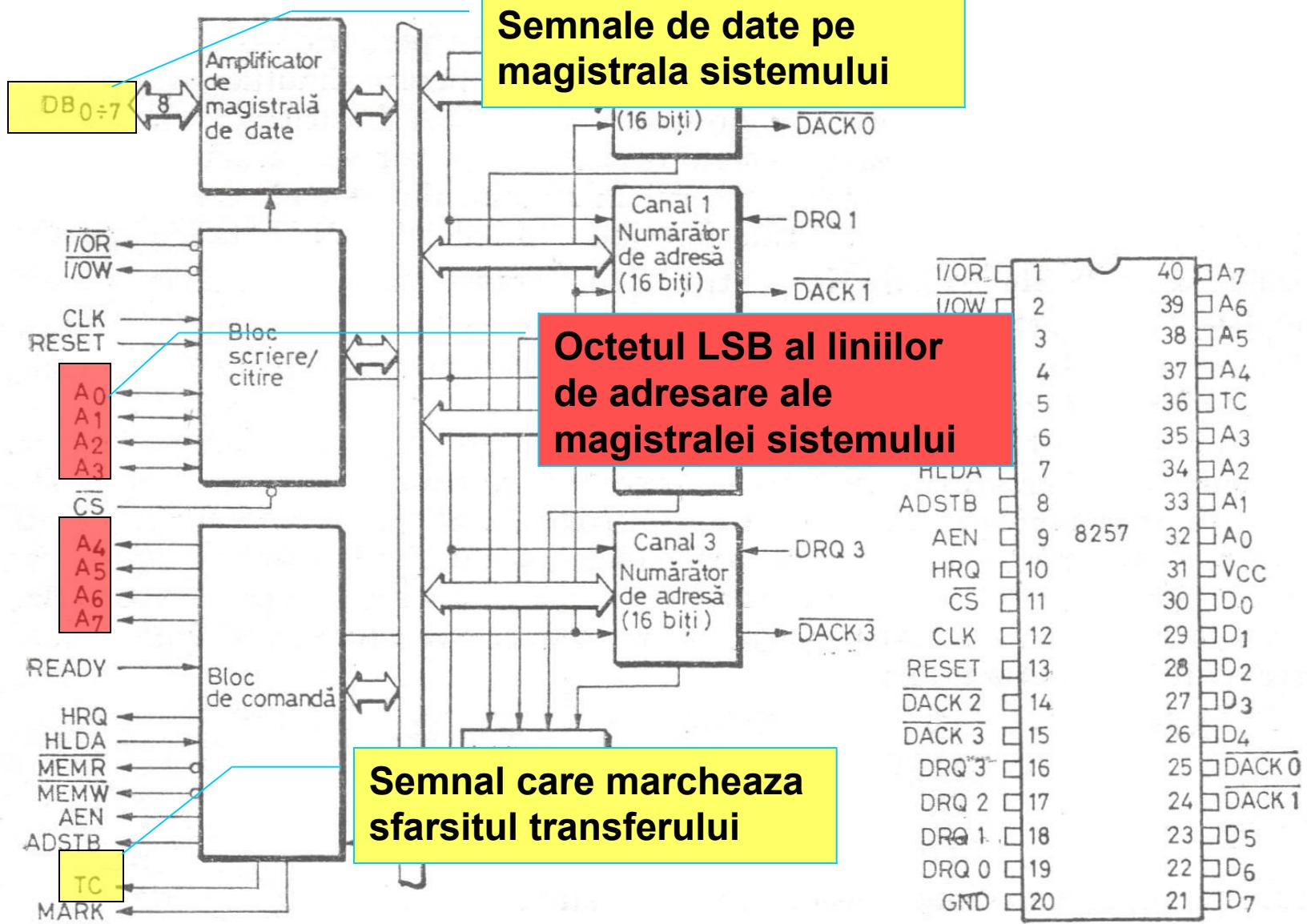
**Semnale de confirmare a obținerii magistralei și incepere transfer (confirmare)**

I/OR	1	A7
I/OW	2	A6
MEMR	3	A5
MEMW	4	A4
MARK	5	TC
READY	6	A3
HLDA	7	
ADSTB	8	
AB	9	
HRQ	10	
C	11	
CU	12	

**Semnale care semnalizează circuitului prezenta datelor pentru transfer DMA (cerere)**

RESET	13	28	D2
DACK 2	14	27	D3
DACK 3	15	26	D4
DRQ 3	16	25	DACK 0
DRQ 2	17	24	DACK 1
DRQ 1	18	23	D5
DRQ 0	19	22	D6
GND	20	21	D7

# DMA cu LSI8257



# DMA cu LSI8257

## Registrele interne ale LSI8257

### Registrele de canal:

- cate unul pentru fiecare canal al circuitului
- mentin **adresa de RAM** de la care incepe transferul
- contin un **counter** al octetilor transferati
- **monitorizeaza** generarea **semnalelor I/OR, I/OW, MEMR, MEMW**

### Registrul de stare:

- contine informatie privind **canalele care au finalizat transferurile**, sfarsit de numarare -> Terminal Count
- contine informatie care **semnaleaza startul / restartul unui transfer** -> Update

# DMA cu LSI8257

## Registrele interne ale LSI8257

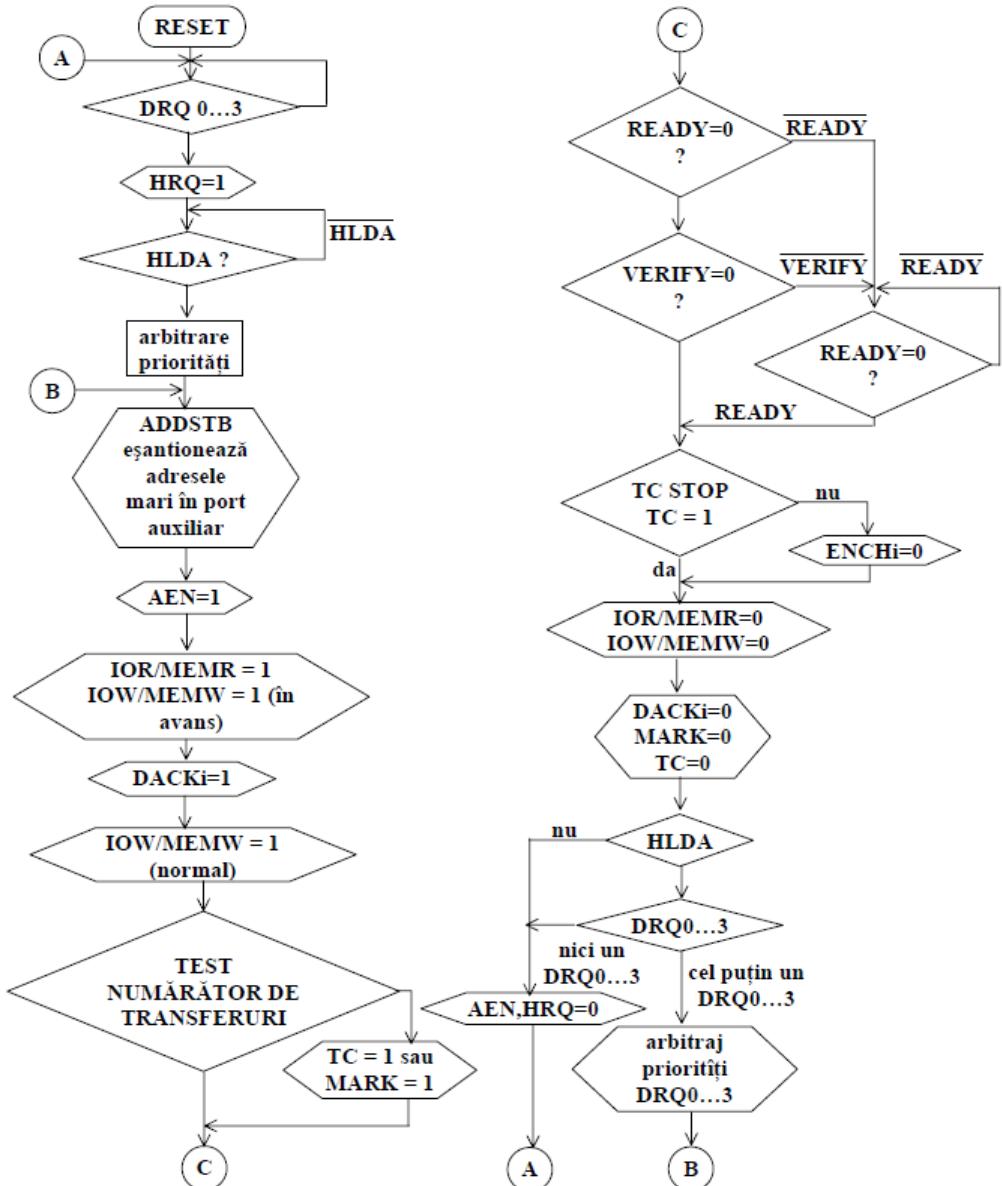
### Registrul de mod:

- Asigura **validarea canalelor** in lucru
- Contine **optiuni privind desfasurarea dialogului** pe magistrala:
  - **Setarea prioritatii** intre canale (fixa/circulara) -> Rotating Priority
  - **Generarea de semnale anticipate** de scriere pentru compensarea timpilor de acces mai mari ai participantilor la transfer -> Extended Write
  - **Oprirea / Continuarea unui transfer DMA** -> Terminal Count Stop
  - **Inlantuirea de comenzi** pentru **transferuri DMA multiple** -> Autoload

# Transferuri DMA cu LSI8257

## Transferul octet cu octet:

1. Perifericul cere transfer prin emiterea semnalului DRQ pe unul din canale
2. Daca este prioritara si este validat canalul incepe transferul lansand HOLD
3. La receptia HLDA se lanseaza DACK pe canalul corespunzator si se selecteaza registrul corespunzator
4. Se genereaza semnalele de esantionare in perechi (I/OR, MEMW) sau (I/OW, MEMR)
5. Dupa transfer se inhiba DACK, se elibereaza magistrala si se inhiba HOLD

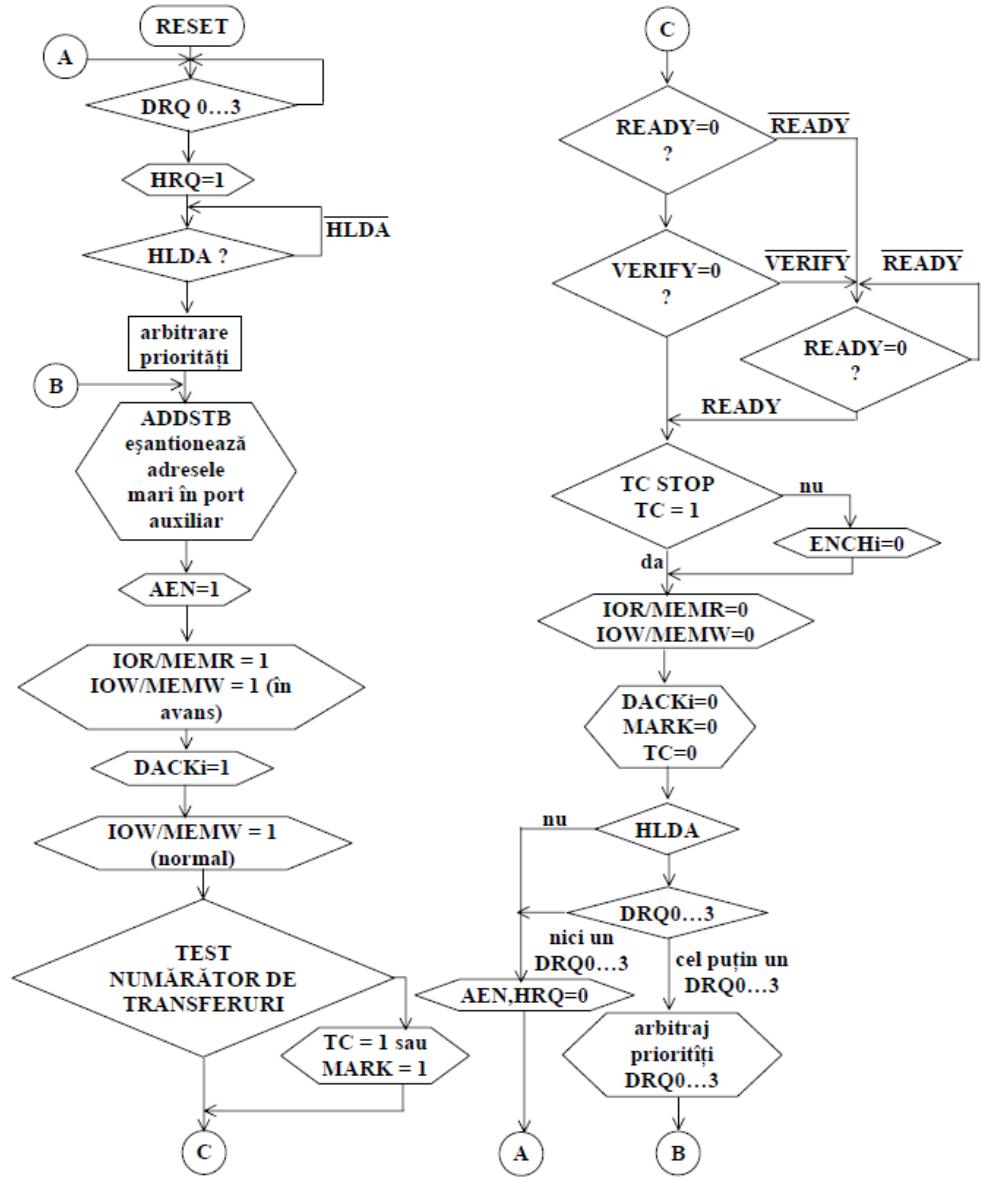


# Transferuri DMA cu LSI8257

## Transferul in mod burst(bloc):

1. se pastreaza DRQ si HLDA in aceeasi stare dupa un transfer,
2. se incrementeaza adresa locatiei de memorie,
3. se reiau operatiile pe octet,
4. nu se mai realizeaza deconectare/reconectare intermedia.

\* Pentru **transferuri simultane** pe mai multe canale se realizeaza o **comutare a semnalelor DRQ, DACK** in functie de **schema de prioritate** programata.

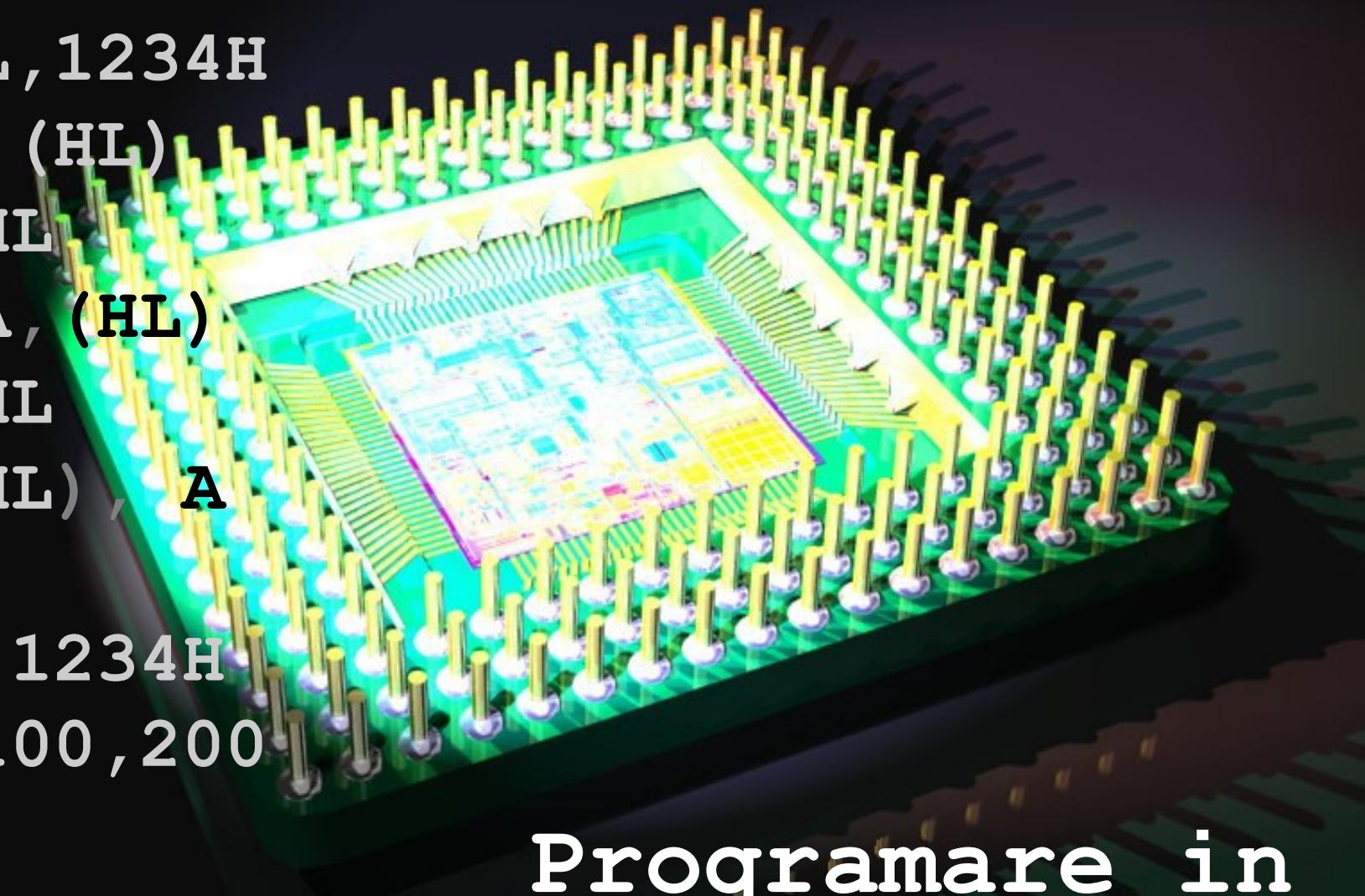


**0100110101110101011000111010001110101  
01101101011001010111001101100011001000  
00011100000110010101101110011101000111  
0010011101010010000001100001011101001  
10010101101110011101000110100101100101  
0010000000100001**

**sau altfel spus ...**

**Multumesc pentru atentie !**

```
.ORG 100H  
LD HL,1234H  
LD A, (HL)  
INC HL  
ADD A, (HL)  
INC HL  
LD (HL), A  
.END  
.ORG 1234H  
.DB 100,200
```



Programare in  
Limba de Asamblare

# Quicksort - Detalii generale

Premise:

- Necesitatea sortarii elementelor unui vector cu dimensiune data in ordine ascendentă;
- Se impune parcurgerea si compararea elementelor vectorului pentru a determina relatia de ordine din vector;

# Quicksort - Detalii generale

## Algoritm:

1. La fiecare iteratie se alege un element si se imparte vectorul in doua parti :

- Prima parte contine elementele mai mari decat elementul selectat (la dreapta);
- A doua parte contine elemente mai mici decat elementul selectat (la stanga);
- Elementele egale cu elementul selectat pot apartine oricarei parti.

2. Cele doua parti sunt apoi sortate recursiv in acelasi mod;

3. Algoritmul continua pana cand toate partile contin un element sau nu contin elemente.

**OBS:** Din rationamente practice recursivitatea trebuie oprita in momentul in care o parte contine un numar mic de elemente !

# Quicksort – Detalii generale

## Parametri de intrare:

- Adresa de inceput a vectorului;
- Adresa ultimului sau element;
- Adresa de inceput a stivei.

## Rutine utile:

- Selectie valoare curenta;
- Comparare elemente;
- Parcursere inainte/inapoi in vector;
- Interschimbare elemente.

Obs: Vectorul poate ocupa orice dimensiune atata timp cat nu se suprapune peste stiva, avand astfel libertate in teste de benchmarking!

# Quicksort – Detalii generale

Observatii privind algoritmul :

- Va imparti la fiecare iteratie vectorul in doua parti
- Elementul selectat va servi drept pivot si astfel cel mai bine ar fi sa se considere un element central
- Desi nu poate fi ales cu exactitate (din cauza dimensiunii vectorului) se impune utilizarea unei aproximari (ex.: selectia elementului median dintre primul, elementului central si ultimului element)

# Quicksort – Detalii implementare

## Procedura:

- Se considera intregul vector
- Se utilizeaza elementul median intre primul element, elementul central si elementul final pentru a imparti vectorul
- Se plaseaza elementul median la inceputul vectorului si se imparte vectorul in doua parti
- Se opereaza apoi recursiv pe fiecare dintre parti separandu-le din nou in alte parti pana cand o parte are 1 element sau 0 elemente

## Reguli de selectie pentru elementul median:

1. Daca vectorul are un numar impar de elemente se alege centrul (ex.: dim = 11 atunci med = 6)
2. Daca vectorul are un numar par de elemente si adresa de baza este para se alege elementul care se afla in centru spre inceputul vectorului (ex.: dim = 12 atunci med = 6)
3. Daca vectorul are un numar par de elemente si adresa de baza este impara se alege elementul care se afla in centru spre finalul vectorului (ex.: dim = 12 atunci med = 7)

# Quicksort – Detalii implementare

## Conditii de lucru :

- Adresa de inceput a vectorului va fi stocata in registrul BC
- Adresa ultimului element va fi stocata in registrul DE
- Pivotul va fi retinut in registrul HL
- Adresa de inceput a stivei SP = 00f0h

## Conditii de iesire :

Vectorul a fost sortat crescator incepand cu  
elementele cele mai mici la adresa de inceput a  
vectorului.

# Quicksort – Exemplu

Vector:

[ 2Bh, 57h, 1Dh, 26h, 22h, 2Eh, 0ch, 44h, 17h,  
4Bh, 37h, 27h ]

Dimensiune vector: 0Ch

Executie:

La iteratia 1: elementul median intre 2Bh, 27h si  
2Eh este 2Bh si nu se face interschimbare  
fiind pe prima pozitie

La finalul iteratiei 1 vectorul este : 27h, 17h, 1Dh,  
26h, 22h, 0Ch, 2Bh, 44h, 2Eh, 4Bh, 37h, 57h

# Quicksort - Exemplu

Prima partitie care contine elementele mai mici decat 2Bh este :

27h, 27h, 1Dh, 26h, 22h, 0Ch

A doua partitie care contine elemente mai mari decat 2Bh este:

44h, 2Eh, 4Bh, 37h, 57h

Se reia recursiv sortarea pe prima partitie:

- Se selecteaza valoarea mediana intre 27h, 1Dh si 0Ch

# Quicksort – Exemplu

## Stare pre-executie(cod, date, stiva)

**Z80 Simulator IDE**

**Assembler - qsort.asm**

**Memory Editor**

**Simulation Log Viewer**

**Program location: C:\\_and\_work\cirea\_facultate\Programare\_In\_Limbaj\_Asamblare\curs\qsort.asm**

**Main registers:**

A	FF
B	FF
C	FF
D	FF
E	FF
H	FF
L	FF

**Alternate registers:**

A'	FF
B'	FF
C'	FF
D'	FF
E'	FF
H'	FF
L'	FF

**Last instruction:** Next | Clock cycles counter: 0 | Instructions counter: 0

**16-bit registers:**

IX	FFFF	SP	FFFF
IY	FFFF	PC	0000

**Main F register:**

7	SF	1
6	ZF	1
5	YF	1
4	HF	1
3	XF	1
2	PF	1
1	NF	1
0	CF	1

**Alternate F register:**

7	SF	1
6	ZF	1
5	YF	1
4	HF	1
3	XF	1
2	PF	1
1	NF	1
0	CF	1

**Special registers:**

I	00	R	00
---	----	---	----

**Interrupt control:**

IFF1	0	IFF2	0	IM	0
NMI		INT		RESET	

**Code View:**

```

0001; QSORT
0002; utilizeaza: bc->primul element, de->ultimul element,
0003;           hl -> adresa pivotului
0004;           call qsort
0005; modifica abcdehl
0006;
0007ld sp,00f0h
0008jp prog
0009qsort:    ld hl,0 ; resetam adresa pivotului
0010        push hl ; salvam in stiva adresa pivotului
0011        ld h,b ; incarcam in hl adresa primului element
0012        ld l,c
0013        or a ; setam CF = 0 dar alternativ se utilizeaza ccf (modifica foar bitul
0014        sbc hl,de; scadem DE din HL testam daca pivot e in vector
0015        jp c,pivot_test ;daca pivot <j se face salt la urmatorul element
0016        pop bc
0017        ld a,b
0018        or c
0019        ret z ; capatul stivei
0020        pop de
0021        jp qsortloop
0022pivot_test: push de ;salvam in stiva contorul j
0023        push bc ;salvam in stiva contorul i

```

**Memory View:**

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0000:	31	F0	00	C3	49	00	21	00	E5	60	69	B7	ED	52	DA	
0010:	1A	00	C1	78	B1	C8	D1	C3	0A	D5	C5	0A	67	0B	13	
0020:	03	0A	BC	DA	20	00	1B	1A	6F	7C	BD	DA	26	00	E5	62
0030:	6B	B7	ED	42	DA	41	00	67	1A	02	7C	12	B1	C3	20	
0040:	00	E1	E1	C5	44	4D	C3	0A	00	01	53	00	11	5P	00	CD
0050:	06	00	7C	2B	57	1D	26	22	2E	0C	44	17	4B	37	27	00
0060:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0070:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0080:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0090:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00A0:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00B0:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00C0:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00D0:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00E0:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

**Log View:**

NO.	PC	Instruction	A	SZYHXPNC	B	C	D	E	H	L
1	0000	LD SP,00F0H	FF	11111111	FF	FF	FF	FF	FF	FF
2	0003	JP 0049H	FF	11111111	FF	FF	FF	FF	FF	FF
3	0049	LD BC,0053H	FF	11111111	00	53	FF	FF	FF	FF
4	004C	LD DE,005EH	FF	11111111	00	53	00	5E	FF	FF
5	004F	CALL 0006H	FF	11111111	00	53	00	5E	FF	FF
6	0006	LD HL,0000H	FF	11111111	00	53	00	5E	00	00
7	0009	PUSH HL	FF	11111111	00	53	00	5E	00	00
8	000A	LD H,B	FF	11111111	00	53	00	5E	00	00
9	000B	LD L,C	FF	11111111	00	53	00	5E	00	53
10	000C	OR A	FF	10101100	00	53	00	5E	00	53

# Quicksort – Exemplu

## Stare post-executie (cod, date, stiva)

**Program location:** C:\\_and\_work\ore\_facultate\Programare\_In\_Limbaj\_Asamblare\curs\

**Main registers:**

A	00
B	00
C	00
D	00
E	5E
H	00
L	00

**Alternate registers:**

A'	FF
B'	FF
C'	FF
D'	FF
E'	FF
H'	FF
L'	FF

**Last instruction:** HALT

**Next:** HALT

**Clock cycles counter:** 6892

**Instructions counter:** 985

**16-bit registers:**

IX	FFFF	SP	00FO
IY	FFFF	PC	0052

**Special registers:**

I	00	R	09
---	----	---	----

**Interrupt control:**

IFF1	01	IFF2	01	IM	00
NMI		INT		RESET	

**Memory Editor:**

Address	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000:	31	F0	00	C3	49	00	21	00	00	E5	60	69	B7	ED	52	DA
0010:	1A	00	C1	78	B1	C2	D1	C3	0A	00	D5	C5	0A	67	0B	13
0020:	03	0A	BC	DA	20	00	18	1A	6F	7C	BD	DA	26	00	E5	62
0030:	6B	B7	ED	42	41	00	00	6A	77	1A	02	7C	12	C1	33	20
0040:	00	E1	C5	44	4D	C3	0A	00	01	53	00	11	5E	00	CD	
0050:	06	00	76	DC	17	10	22	26	27	2B	2E	37	44	4B	57	00
0060:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0070:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0080:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0090:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00A0:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00B0:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00C0:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00D0:	**	**	**	**	**	**	**	**	**	**	**	**	**	**	**	**
00E0:	**	**	**	**	**	**	**	**	**	**	**	**	**	**	**	**
00F0:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

**Simulation Log Viewer:**

No.	PC	Instruction	A	S	Z	Y	H	X	P	N	C	D	E	H	L
1	0000	LD SP,00F0H	FF	11	11	11	11	11	FF						
2	0003	JP 0049H	FF	11	11	11	11	11	FF						
3	0049	LD BC,0053H	FF	11	11	11	11	11	00	53	00	FF	FF	FF	FF
4	004C	LD DE,005EH	FF	11	11	11	11	11	00	53	00	5E	FF	FF	FF
5	004F	CALL 0006H	FF	11	11	11	11	11	00	53	00	5E	FF	FF	FF
6	0006	LD HL,0000H	FF	11	11	11	11	11	00	53	00	5E	00	00	00
7	0009	PUSH HL	FF	11	11	11	11	11	00	53	00	5E	00	00	00
8	000A	LD H,B	FF	11	11	11	11	11	00	53	00	5E	00	00	00
9	000B	LD L,C	FF	11	11	11	11	11	00	53	00	5E	00	53	00
10	000C	OR A	FF	10	01	10	11	00	00	53	00	5E	00	53	00
11	000D	SWP HL,DE	FF	10	11	11	11	11	00	53	00	5E	FF	FF	FF

**Code View:**

```

0001; QSORT
0002; utilizeaza bc->primul element, de->ultimul element,
0003; hl -> adresa pivotului
0004; call qsort
0005; modifica abcdehl
0006;
0007; ld sp,00f0h
0008; jp prog
0009; qsort: ld hl,0 ; resetam adresa pivotului
0010; push hl ; salvam in stiva adresa pivotului
0011; qsloop: ld h,b ; incarcam in hl adresa primului element
0012; ld l,c
0013; or a ; setam CF = 0 dar alternativ se utilizeaza ccf (modifica foar bitul CF)
0014; sbc hl,de; scadem DE din HL testam daca pivotul e in vector
0015; jp c,pivot_test ;daca pivot <j se face salt la urmatorul element
0016; pop bc
0017; ld a,b
0018; or c
0019; ret z ; capatul stivei
0020; pop de
0021; jp qsloop
0022; pivot_test: push de ;salvam in stiva contorul j
0023; push bc ;salvam in stiva contorul i

```

**Log View:**

Line	PC	Instruction	Value
0045	0045	0038 67	ld h,a ; se incarca in h elementul din partitie
0046	0046	0039 1A	ld a,(de) ; se incarca in a elementul din partitie
0047	0047	003A 02	ld (bc),a ; se intreschimba valorile intre cele doua elemente
0048	0048	003B 7C	ld a,h
0049	0049	003C 12	ld (de),a
0050	0050	003D E1	pop hl ; se restaureaza valoarea pivotului din partitie
0051	0051	003E C3 20 00	left_part ; salt la noua partitionare pt partitie
0052	0052	0041 E1	contor_test: pop hl ; daca i>j se restaureaza pivotul
0053	0053	0042 E1	pop hl ; se extrage BC
0054	0054	0043 C5	push bc ; stiva = i - hi
0055	0055	0044 44	ld b,h
0056	0056	0045 4D	ld c,l ;bc=lo,de=j
0057	0057	0046 C3 0A 00	qsloop ; salt la urmatorul element
0058	0058	0049 01 53 00	prog: ld bc,startv
0059	0059	004C 11 5E 00	ld de,endv
0060	0060	004F CD 06 00	call qsort
0061	0061	0052 76	halt
0062	0062	0053 2B	startv: .db 2bh
0063	0063	0054 57	.db 57h
0064	0064	0055 1D	.db 1dh
0065	0065	0056 26	.db 26h
0066	0066	0057 22	.db 22h
0067	0067	0058 2E	.db 2eh
0068	0068	0059 0C	.db 0ch
0069	0069	005A 44	.db 44h
0070	0070	005B 17	.db 17h
0071	0071	005C 4B	.db 4bh
0072	0072	005D 37	.db 37h

# Quicksort - Exemplu

Observatii:

- Timpul de executie este relativ mare datorita recursivitatii;
- Datorita arhitecturii simple a procesorului recursivitatea se implementeaza utilizand intensiv stiva care trebuie plasata atent in memorie pentru a evita overflow-ul;
- In timpul executiei registrii preiau valori diferite si este util sa utilizam stiva pentru a mentine valorile de interes disponibile ulterior;
- Varianta implementata nu trateaza si optimizarea la nivelul alegerii pivotului.

# Quicksort - Exemplu

Cerinte si optimizare de cod:

1. Sa se inlocuiasca instructiunea care pune 0 pe bitul de stare CF (carry flag) util la scaderea pe 16 biti pentru testul de existenta a pivotului in vector cu instructiunea **ccf** (clear carry flag). Analizati diferentele la modul de setare al bitilor din registrul de stare F.
2. Sa se inlocuiasca instructiunile de transfer de 16 biti necesare testului de final de vector

**ld h,d**

**ld l,e**

cu instructiunea de interschimbare de 16 biti, **ex de,h1**.

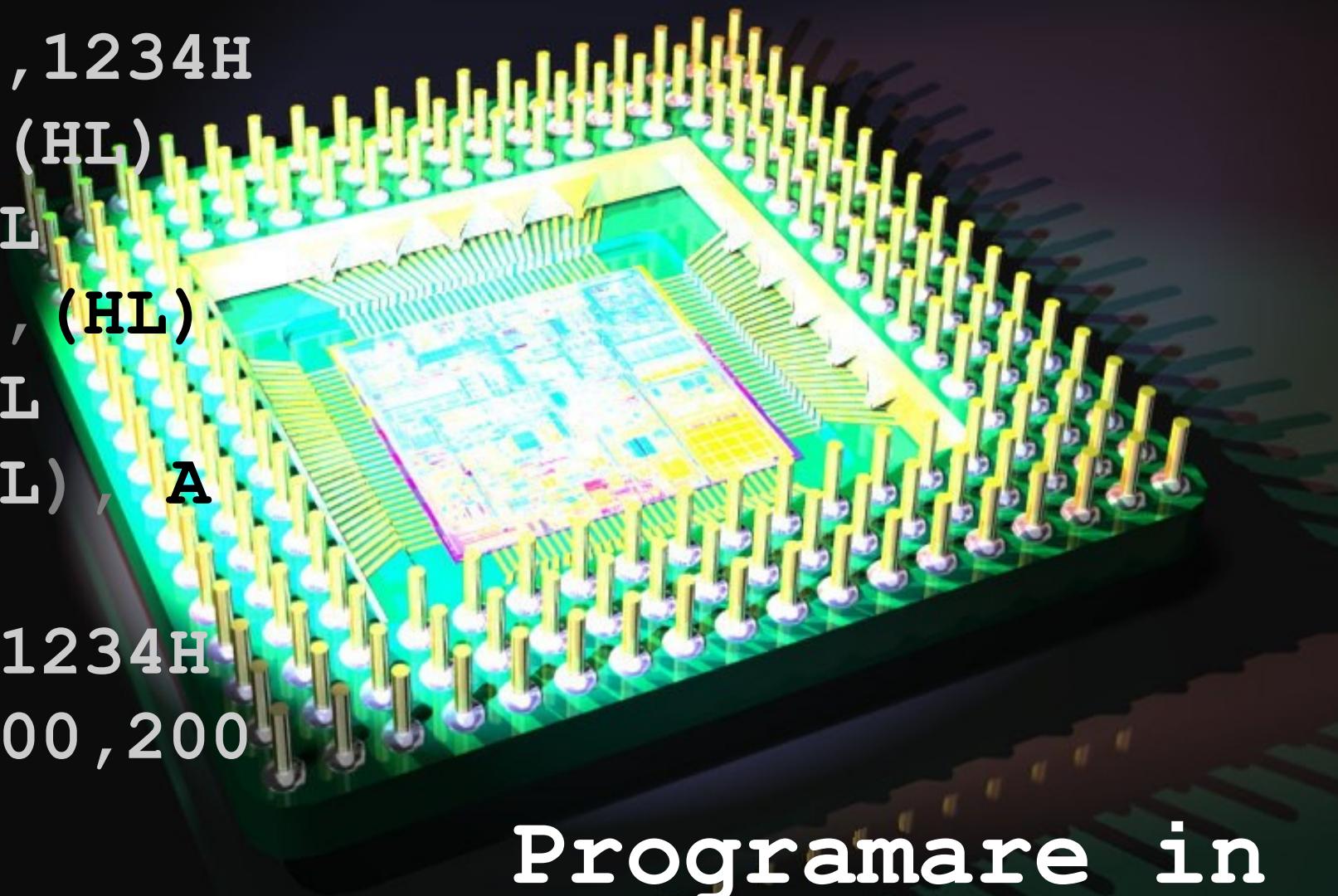
Analizati modul in care se executa transferul si influenta asupra timpului de executie.

**0100110101110101011000111010001110101  
01101101011001010111001101100011001000  
00011100000110010101101110011101000111  
00100111010100100000011000010111010001  
10010101101110011101000110100101100101  
0010000000100001**

**sau altfel spus ...**

**Multumesc pentru atentie !**

```
.ORG 100H  
LD HL,1234H  
LD A, (HL)  
INC HL  
ADD A, (HL)  
INC HL  
LD (HL), A  
.END  
.ORG 1234H  
.DB 100,200
```



Programare in  
Limbaș de Asamblare

# Coduri detectoare si coduri corectoare de erori

In **comunicatiile de date** scopul metodelor de **detectare** a erorilor este acela de a permite celui care primeste un mesaj transmis printr-un mediu cu zgomot, sa recunoasca daca mesajul a fost sau nu corupt.

S-au dezvoltat **doua tipuri de coduri**:

- **coduri corectoare de erori** - este transmis un bloc de date impreuna cu informatie redundanta pentru ca receptorul sa detecteze caracterul lipsa
- **coduri detectoare de erori** - este transmis un bloc de date impreuna cu informatie redundanta pentru depistarea aparitiei unor erori, fara a cunoaste exact eroarea

# Coduri detectoare de erori

**Transmitatorul** construieste o valoare (checksum) care depinde de continutul mesajului si o adauga acestuia.

**Receptorul** va utiliza aceeasi functie pentru a calcula suma de control, valoare pe care o va compara cu cea receptionata.

Se disting la verificare doua cazuri:

- mesajul a fost alterat
- suma de control transmisa a fost modificata.

Receptorul nu poate face deosebire intre aceste doua cazuri si va cere retransmisia blocului.

# Coduri detectoare de erori

Ex:

Sa luam ca functie suma octetilor din mesaj modulo 256.

Mesaj : 4 26 5

Mesaj transmis : 4 26 5 35

Mesaj corect receptionat : 6 26 5 35

Mesaj incorect receptionat : 6 26 7 35

# Coduri detectoare de erori - CRC

Idea algoritmului CRC este de a considera mesajul ca un singur numar binar pe care sa-l impartim repetat cu o alta valoare fixata, suma de control fiind restul impartirii.

Sirul de biti este interpretat ca un polinom cu coeficienti cu 1 si 0. Un cadru de k biti este vazut ca o lista de coeficienti pentru un polinom de grad k-1. Bitul cel mai semnificativ este coeficientul lui  $x^{k-1}$ .

Exemplu:

$23=17(\text{hexa})=10111(\text{binar}):$

$$\begin{aligned}1*x^4 + 0*x^3 + 1*x^2 + 1*x^1 + 1*x^0 &= x^4 + x^2 \\&\quad + x^1 + x^0\end{aligned}$$

# Coduri detectoare de erori - CRC

Aritmetica polinomiala mod 2 este aritmetica binara mod 2 fara transport. Adunarile si scaderile sunt identice cu "sau exclusiv" - XOR.

$$\begin{array}{r} 10011011 \\ +11001010 \\ \hline 01010001 \end{array}$$

Inmultirea este foarte simpla:

$$\begin{array}{r} 1101 \\ \times 1011 \\ \hline 1101 \\ 1101. \\ 0000.. \\ 1101... \\ \hline 1111111 \end{array}$$

# Coduri detectoare de erori - CRC

Impartirea este putin mai complicata deoarece trebuie sa stim cind un numar este mai mic ca altul: X este mai mare sau egal cu Y daca pozitia bitului 1 cel mai din stanga al lui X este aceeasi sau mai mare ca pozitia bitului 1 cel mai din stanga al lui Y.

$$\begin{array}{r|l} 101011000110 & | 1010011 \\ 1010011 \downarrow & | 100010 = catul \\ \hline 0001010 & \\ 0000000 \downarrow & \\ \hline 0010100 & \\ 0000000 \downarrow & \\ \hline 0101001 & \\ 0000000 \downarrow & \\ \hline 1010011 & \\ 1010011 \downarrow & \\ \hline 0000000 & \\ 0000000 \downarrow & \\ \hline 0000000 = restul & \end{array}$$

# Coduri detectoare de erori - CRC

Pentru a calcula CRC, emitorul si receptorul stabilesc un impartitor: acesta se mai numeste si **polinom generator**.

Bitul cel mai semnificativ cit si cel mai putin semnificativ trebuie sa fie 1.

# Coduri detectoare de erori - CRC

**Algoritmul pentru calculul CRC este urmatorul:**

1. Fie  $r$  gradul polinomului generator  $G(x)$ . Se adauga  $r$  biti la capatul mai putin semnificativ al mesajului care va avea acum  $n+r$  biti.
2. Se imparte acest sir la polinomul generator.
3. Restul este adaugat la capatul cel mai putin semnificativ al mesajului, care apoi se transmite.

Receptorul poate alege una din urmatoarele metode:

1. Separa mesajul si suma de control. Calculeaza suma de control pentru mesaj (dupa ce adauga  $r$  0-uri) si compara cele doua sume de control.
2. **Calculeaza suma de control a intregului mesaj si verifica daca obtine ca rezultat 0.**

# Coduri detectoare de erori - CRC

Exemple de polinoame utilizate:

$$\text{CRC-12} = x^{12} + x^{11} + x^3 + x^2 + x^1 + 1$$

$$\text{CRC-16} = x^{16} + x^{15} + x^2 + 1$$

$$\text{CRC-CCITT} = x^{16} + x^{12} + x^5 + 1$$

*Ethernet* =

$$\begin{aligned} & x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + \\ & x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1 \end{aligned}$$

# Coduri detectoare de erori – CRC

## Implementare assembly Z80

Programul dezvoltat genereaza o suma de control CRC utilizand polinomul generator  $x^4 + x^2 + x^1 + x^0$ .

### ***Structura programului:***

*Rutina ICRC16 initializeaza suma CRC cu 0 si polinomul generator cu valoarea data.*

*Rutina CRC16 realizeaza updateul sumei CRC pentru octetul curent.*

*Rutina GCRC16 returneaza suma CRC intr-un registru.*

*Programul principal calculeaza CRC pentru un octet si apoi verifica corectitudinea CRC generata.*

# Coduri detectoare de erori – CRC

## Implementare assembly Z80

Procedura:

- Rutina ICRC16 initializeaza suma CRC si polinomul cu 1 pe pozitiile in care sunt prezente puteri ale lui X
- Rutina CRC16 calculeaza suma CRC pentru octetul curent si aduna cu valoarea anterioara
  - Se shifteaza data si suma CRC la stanga de 8 ori si daca XOR intre bitul cel mai semnificativ al CRC si bitul datei este 1, se calculeaza XOR intre polinom si CRC
  - Suma CRC este salvata in memorie
- Rutina GCRC16 extrage suma CRC din memorie in registrul HL

# Coduri detectoare de erori – CRC

## Implementare assembly Z80

```
0014 0006          ; rutina de calcul a CRC (se apeleaza pentru fiecare octet)
0015 0006          CRC16: ; salveaza starea registrilor procesorului in stiva
0016 0006 F5        push af
0017 0007 C5        push bc
0018 0008 D5        push de
0019 0009 E5        push hl
0020 000A          ; bucla de generare a sumei CRC
0021 000A 06 08      ld b,8       ; numarul de biti dintr-un octet
0022 000C ED 5B 40 00   ld de,(POLY) ; DE retine polinomul generator
0023 0010 2A 3E 00   ld hl,(CRC)  ; HL retine suma CRC
0024 0013 4F          CRCLP:    ld c,a       ; salveaza valoarea datei in C
0025 0014 E6 80        and 10000000b ; preia bitul 7 al valorii
0026 0016 AC          xor h        ; XOR intre bitul 7 al datei si bitul 15 al CRC
0027 0017 67          ld h,a       ; incarca in H data
0028 0018 29          add hl,hl   ; se shifteaza CRC la stanga
0029 0019 30 06      jr nc,CRCLP1 ; salt daca bitul 7 al XOR a fost 0
0030 001B          ; bit 7 a fost 1 si astfel facem XOR intre CRC si polinom
0031 001B 7B          ld a,e       ; preia octetul LSB al polinomului
0032 001C AD          xor l        ; XOR cu LSB al CRC
0033 001D 6F          ld l,a       ; incarca in L valoarea din a
0034 001E 7A          ld a,d       ; preia octetul MSB al polinomului
0035 001F AC          xor h        ; XOR cu MSB al CRC
0036 0020 67          ld h,a       ; se incarca in H valoarea din A
0037 0021 79          CRCLP1:   ld a,c       ; se restaura valoare datei
0038 0022 17          rla          ; rotate left accumulator with carry(rotate)
0039 0023 10 EE        djnz CRCLP   ; cat timp mai sunt biti de testat reia testul
0040 0025 22 3E 00   ld (CRC),hl  ; salveaza suma CRC in memorie
0041 0028          ; restaura starea registrilor si se revine
0042 0028 E1          pop hl
0043 0029 D1          pop de
0044 002A C1          pop bc
0045 002B F1          pop af
0046 002C C9          ret
```

# Coduri detectoare de erori – CRC

## Implementare assembly Z80

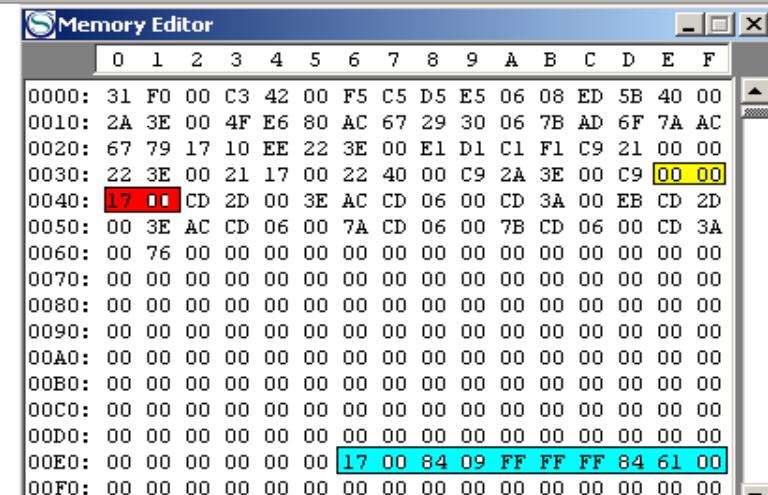
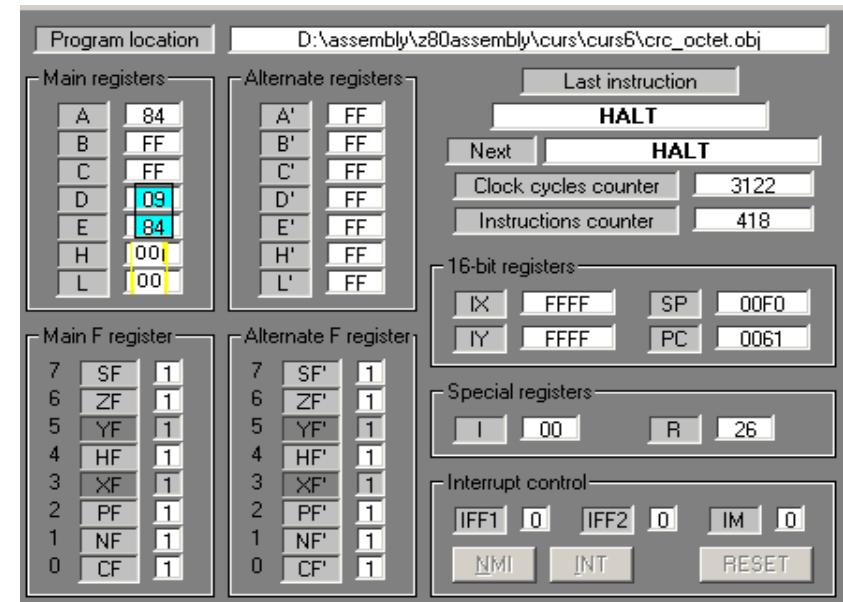
```
0048 002D          ; rutina de initializare a CRC si a polinomului generator
0049 002D 21 00 00  ICRC16: ld hl,0h
0050 0030 22 3E 00      ld (CRC),hl ; se initializeaza suma CRC cu 0
0051 0033 21 17 00      ld hl,10111b ; se scrie valoarea hex corespunzatoare polinomului
0052 0036 22 40 00      ld (POLY),hl ; se salveaza in memorie polinomul
0053 0039 C9          ret
0054 003A
0055 003A          ; rutina care preia valoarea CRC
0056 003A 2A 3E 00  GCRC16: ld hl,(CRC) ; extrage din memorie in registrul HL valoarea CRC
0057 003D C9          ret
0058 003E          ; defineste spatiu pentru CRC si polinom (2 octeti)
0059 003E          CRC: .ds 2
0060 0040          POLY: .ds 2
0061 0042
0062 0042          ; programul principal
0063 0042          ; ex: genereaza CRC pentru numarul ... (preluat de la user) si verifica
0064 0042          ; la verificare suma CRC pentru valoare + suma CRC generata pt ea trebuie sa fie 0
0065 0042          PROG: ; generare suma de control
0066 0042 CD 2D 00      call ICRC16    ; se initializeaza CRC si polinomul
0067 0045 DB 12          in a,(12h)   ; se incarca valoarea pt care dorim sa calculam suma CRC
0068 0047 CD 06 00      call CRC16    ; calculeaza suma CRC
0069 004A CD 3A 00      call GCRC16   ; extrage suma CRC din memorie
0070 004D EB          ex de,hl    ; scrie suma in registrul DE
0071 004E          ; verificare suma de control
0072 004E CD 2D 00      call ICRC16   ; se initializeaza CRC si polinomul
0073 0051 3E AC          ld a,0ach   ; se incarca valoarea pt care vrem sa verificam CRC
0074 0053 CD 06 00      call CRC16    ; se calculeaza CRC pt valoare
0075 0056 7A          ld a,d     ; se incarca in A MSB al CRC calculat pt valoarea din A
0076 0057 CD 06 00      call CRC16    ; se calculeaza CRC
0077 005A 7B          ld a,e     ; se incarca in A LSB al CRC calculat pt valoarea din A
0078 005B CD 06 00      call CRC16    ; se calculeaza CRC
0079 005E CD 3A 00      call GCRC16   ; se extrage din memorie CRC pt verificare
0080 0061 76          halt
```

# Coduri detectoare de erori – CRC

## Implementare assembly Z80

Harta memoriei inainte de executie si dupa calculul CRC pentru octetul 4ch dat de user

```
0000: 31 F0 00 C3 42 00 F5 C5 D5 E5 06 08 ED 5B 40 00
0010: 2A 3E 00 4F E6 80 AC 67 29 30 06 7B AD 6F 7A AC
0020: 67 79 17 10 EE 22 3E 00 E1 D1 C1 F1 C9 21 00 00
0030: 22 3E 00 21 17 00 22 40 00 C9 2A 3E 00 C9 00 00
0040: 00 00 CD 2D 00 DB 12 CD 06 00 CD 3A 00 EB CD 2D
0050: 00 3E AC CD 06 00 7A CD 06 00 7B CD 06 00 CD 3A
0060: 00 76 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0070: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0090: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

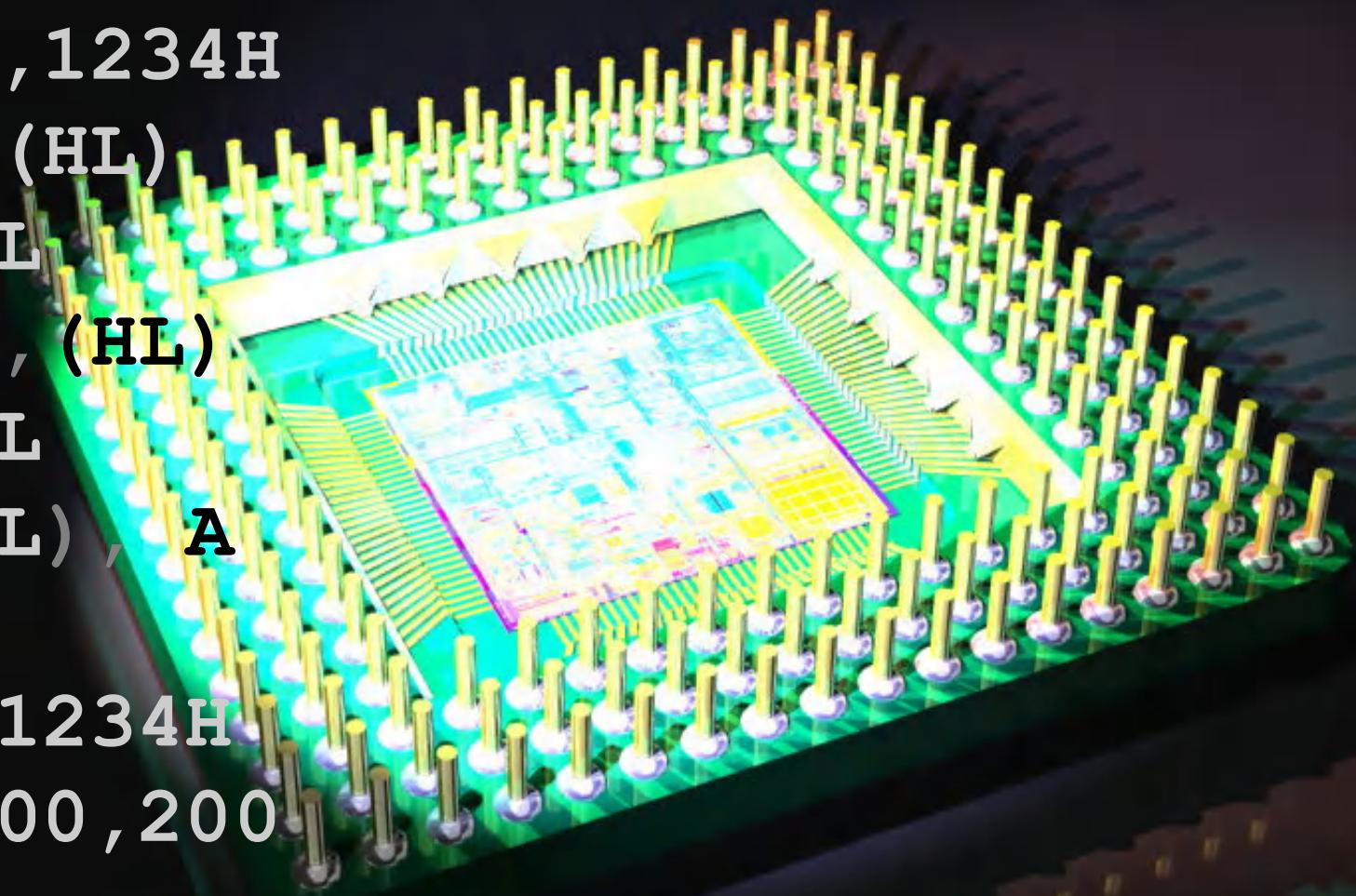


**0100110101110101011000111010001110101  
01101101011001010111001101100011001000  
00011100000110010101101110011101000111  
00100111010100100000011000010111010001  
10010101101110011101000110100101100101  
0010000000100001**

**sau altfel spus ...**

**Multumesc pentru atentie !**

```
.ORG 100H  
LD HL,1234H  
LD A, (HL)  
INC HL  
ADD A, (HL)  
INC HL  
LD (HL), A  
.END  
.ORG 1234H  
.DB 100,200
```



Programare în  
Limbaje de Asamblare

# **Impartirea a doua numere reprezentate pe 16 biti**

Aplicatia propusa imparte doi operanzi pe 16 biti si returneaza catul si restul impartirii.

## **Detalii de implementare:**

- Rutina pentru impartiri cu numere cu semn (SDIV)
- Rutina pentru impartiri cu numere fara semn (UDIV)

## **Conditii si premise:**

Daca impartitorul este 0 CF este setat pe 1 si catul si restul sunt setati apoi 0

Daca impartitorul este diferit de 0 CF este setat pe 0.

# Impartirea a doua numere reprezentate pe 16 biti

## Procedura:

1. Se testeaza daca operanzii au semn
2. Se determina astfel care este semnul catului
3. Se calculeaza valoare absoluta a operanzilor negativi
4. Se realizeaza o impartire fara semn utilizand un algoritm **shift-and-subtract**
  1. Se shifteaza catul si deimpartitul la stanga
  2. Se plaseaza un bit de 1 in cat de cate ori scaderea s-a realizat cu succes
5. Se testeaza daca operanzii sunt negativi si astfel in caz afirmativ se neaga catul sau restul corespunzator.

# Impartirea a doua numere reprezentate pe 16 biti

Rezultate:

- Bitul CF este 0 daca rezultatul impartirii este valid si 1 daca avem un impartitor egal cu 0.

Exemple:

Date: D = 03E0h

I = 00B6h

Rezultat: Q = 0005h

R = 0052h

CF = 0

# **Impartirea a doua numere reprezentate pe 16 biti**

Aplicatia propusa imparte doi operanzi pe 16 biti si returneaza catul si restul impartirii.

## **Detalii de implementare:**

- Rutina pentru impartiri cu numere cu semn (SDIV)
- Rutina pentru impartiri cu numere fara semn (UDIV)

## **Conditii si premise:**

Daca impartitorul este 0 CF este setat pe 1 si catul si restul sunt setati apoi 0

Daca impartitorul este diferit de 0 CF este setat pe 0.

# Impartirea a două numere reprezentate pe 16 biti – Cod sursa

```
; Program de realizare a operatiei de impartire pe 16 biti
; Imparte doi operanzi de 16 biti si returneaza catul si restul impartirii
; Aplicatia acopera cazul operanzilor cu semn dar si a operanzilor fara semn

; Date de intrare : Deimpartit in reg. HL (D)
;                   Impartitor in reg. DE (I)

; Date de iesire : Catul in reg. HL (Q)
;                   Restul in reg. DE (R)
; Daca impartitorul este diferit de 0 atunci CF = 0 si rezultatul este normal
; Daca impartitorul este 0, CF = 1 si atat catul cat si restul vor fi 0.
;   ld sp,0300h
;   jp PROG
;   .org 0300h
;-----
; datele de lucru
SQ: .ds 1          ; semnul catului
SR: .ds 1          ; semnul restului
; rutina de realizare a impartirii cu numere cu semn
; se determina semnul lui Q efectuand XOR intre octetii MSB ai
; D si I. Q are semn pozitiv daca D si I au acelasi semn si negativ altfel.
; R are acelasi semn cu D.
SDIV:  ld a,h      ; preia MSB al deimpartitului
       ld (SR),a    ; salveaza ca semn pt rest
       xor d        ; xor cu MSB al deimpartitului pt a det semnul catului
       ld (SQ),a    ; retine semnul catului

; extrage valoarea absoluta a impartitorului
ld a,d      ; se incarca MSB al impartitorului
or a        ; se sumeaza cu a
jp P,CHKDE ; salt daca impartitorul e pozitiv
sub a      ; scade impartitorul din 0
sub e
ld e,a      ; incarca in E valoarea din A
sbc a,a    ; se propaga transportul (a=ffh daca se realizeaza imprumut)
sub d
ld d,a
```

# Impartirea a două numere reprezentate pe 16 biti – Cod sursa

```
; extrage valoarea absoluta a deimpartitului
CHKDE: ld a,h      ; se incarca MSB al deimpartitului
       or a        ; se sumeaza cu a
       jp P,DODIV ; salt daca deimpartitul e pozitiv
       sub a       ; scade deimpartitul din 0
       sub l
       ld l,a
       sbc a,a    ; se propaga imprumutul (a=ffh daca se realizeaza imprumut)
       sub h       ; se scade octetul MSB al deimpartitului
       ld h,a     ; incarcam in H continutul lui A
; se imparte valorile absolute determinante anterior
DODIV: call udiv    ; apelul rutinei de calcul pt numere fara semn
       ret c       ; iesire daca se realizeaza impartire la 0
       ; se neaga catul daca este negativ
       ld a,(SQ)
       or a
       jp P,DOREM ; salt daca catul e pozitiv
       sub a       ; scade catul din 0
       sub l
       ld l,a
       sbc a,a    ; propaga imprumutul
       sub h
       ld h,a
DOREM: ; neaga restul daca este negativ
       ld a,(SR)
       or a
       ret P      ; revenire daca restul este pozitiv
       sub a       ; scade restul din 0
       sub e
       ld e,a
       sbc a,a    ; propaga imprumutul (borrow)
       sub d
       ld d,a
       ret
;-----
```

# Impartirea a două numere reprezentate pe 16 biti – Cod sursa

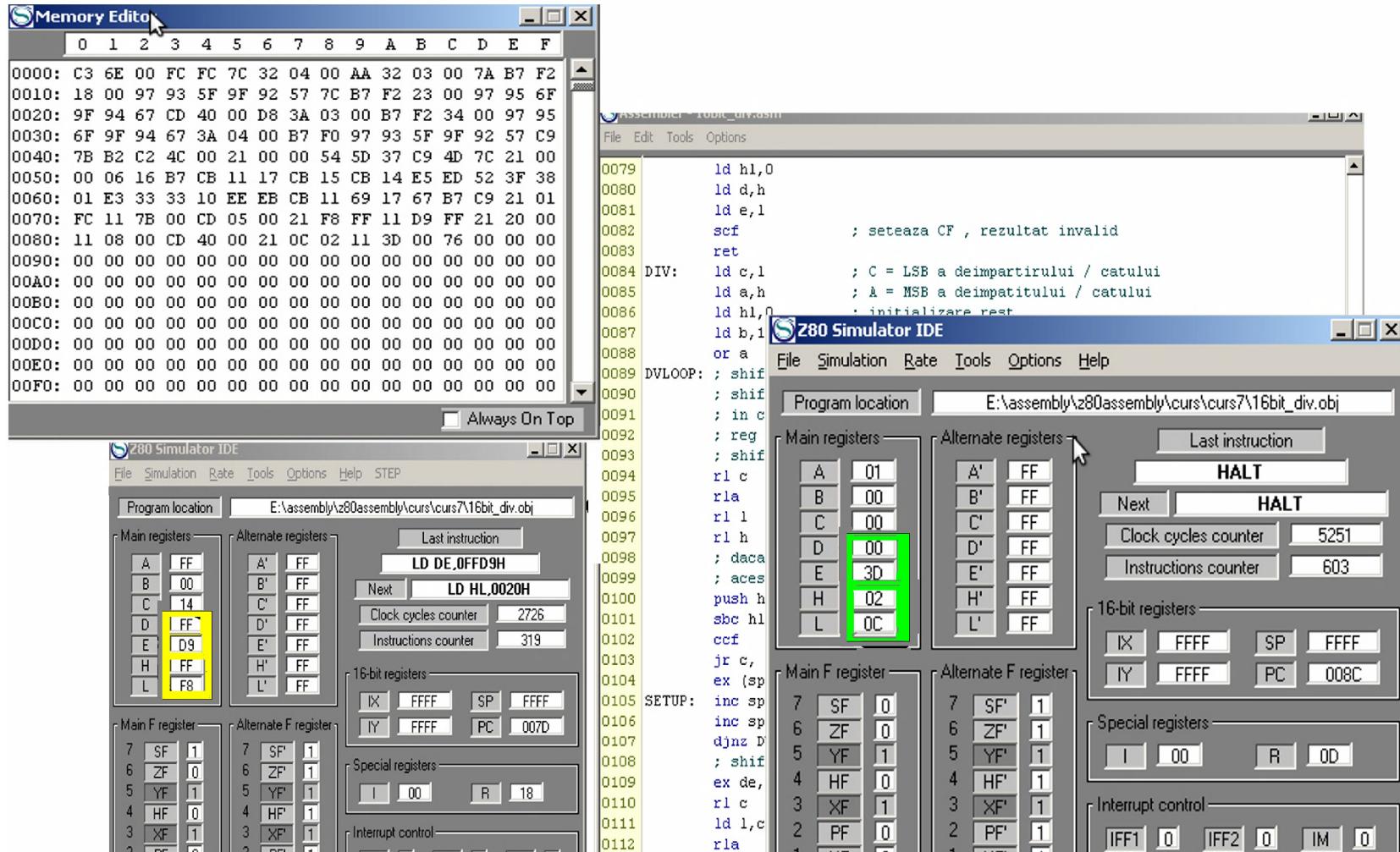
```
; rutina de realizare a impartirii cu numere fara semn
UDIV:    ; se verifica impartirea la 0
        ld a,e
        or d
        jp nz,DIV    ; salt la DIV daca a nu este 0
        ld hl,0
        ld d,h
        ld e,l
        scf      ; seteaza CF , rezultat invalid
        ret

DIV:    ld c,l      ; C = LSB a deimpartirului / catului
        ld a,h      ; A = MSB a deimpartitului / catului
        ld hl,0      ; initializare rest
        ld b,16h     ; nr de biti ai deimpartitului
        or a        ; resteaaza bitul de CF
DVLOOP: ; shifteaza urmatorul bit al catului in bitul 0 al deimpartitului
        ; shifteaza urmatorul cel mai semnificativ bit al deimpartitului
        ; in cel mai putin semnificativ bit al restului
        ; reg BC va retine atat dweimpartitul dar si catul
        ; shifteaza la stanga (CF->C, C in A, A in L si L in H)
        rl c        ; carry, urmatorul bit al catului este pus pe 0
```

# Impartirea a două numere reprezentate pe 16 biti – Cod sursa

```
rla      ; shifteaza bitii care raman dupa prima operatiune
rl l
rl h      ;resteaza carry flag din moment ce HL = 0
; daca restul este mai mare / egal cu imparitorul urmatorul bit al catului este 1
; acest bit se va transfera in CF
push hl    ; salveaza restul curent
sbc hl,de  ; scade imparitorul din rest
ccf       ; indica daca operatia a fost cu succes
jr c, SETUP ; salt daca restul >= deimpartitul
ex (sp),hl ; restaureaza valoarea restului
SETUP: inc sp     ; se extrage restul de pe stiva
inc sp
djnz DVLOOP ; continua pana cand toti bitii au fost testati
; shifteaza ultimul bit al carry in cat
ex de,hl   ; plasam in DE restul
rl c       ; copie carry flag in registrul C
ld l,c    ; L = LSB al catului
rla
ld h,a    ; H = MSB al catului
or a      ; reseteaza carry pt. ca rezultat valid
ret
;-----
PROG: ; exemplu impartire numere fara semn si cu semn
; ex1 - numere signed
ld hl, -1023   ; deimpartitul
ld de, 123     ; imparitorul
call SDIV    ; catul = -1023/123 = -8 (ffff8h) si restul = -39 (ffd9h)
; ex2 - numere unsigned
ld hl, 32      ; deimpartitul
ld de, 8       ; imparitorul
call UDIV    ; catul = 524 (20ch) si restul = 61 (3dh)
halt
.end
```

# Impartirea a două numere reprezentate pe 16 biti – Rezultatele executiei

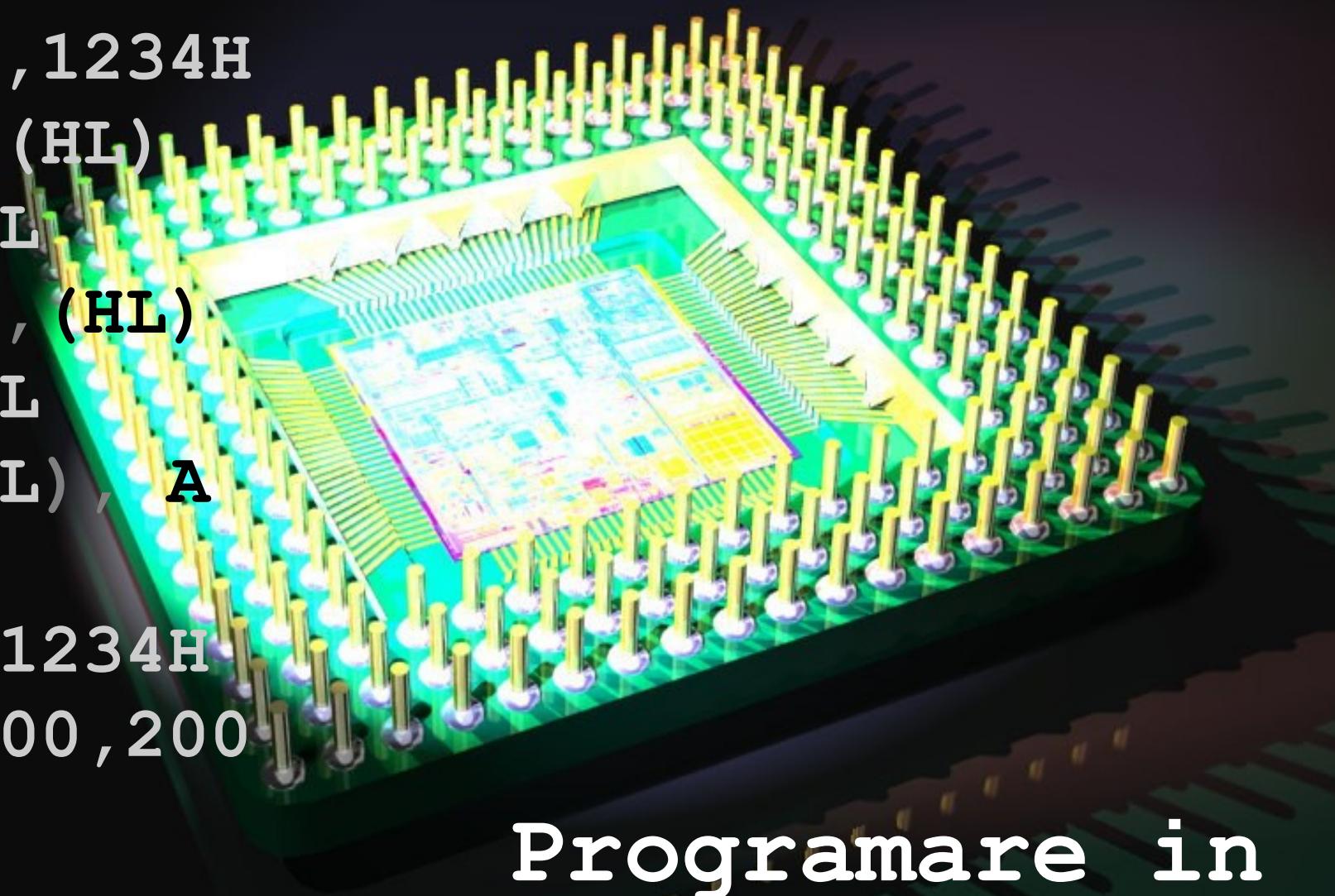


**01001101011101010110001110100011  
01010110110101100101011100110110001  
10010000001110000011001010110111001  
11010001110010011101010010000001100  
00101110100011001010110111001110100  
01101001011001010010000000100001**

**sau altfel spus ...**

**Multumesc pentru atentie !**

```
.ORG 100H  
LD HL,1234H  
LD A, (HL)  
INC HL  
ADD A, (HL)  
INC HL  
LD (HL), A  
.END  
.ORG 1234H  
.DB 100,200
```



Programare in  
Limba de Asamblare

# Real-Time Clock (RTC)

Un **RTC este un dispozitiv de temporizare** sub forma unui circuit integrat pentru a mentine valoarea curenta a timpului.

RTC se regaseste in toate dispozitivele care necesita **mentinerea precisa a valorii curente a timpului**.

Beneficii:

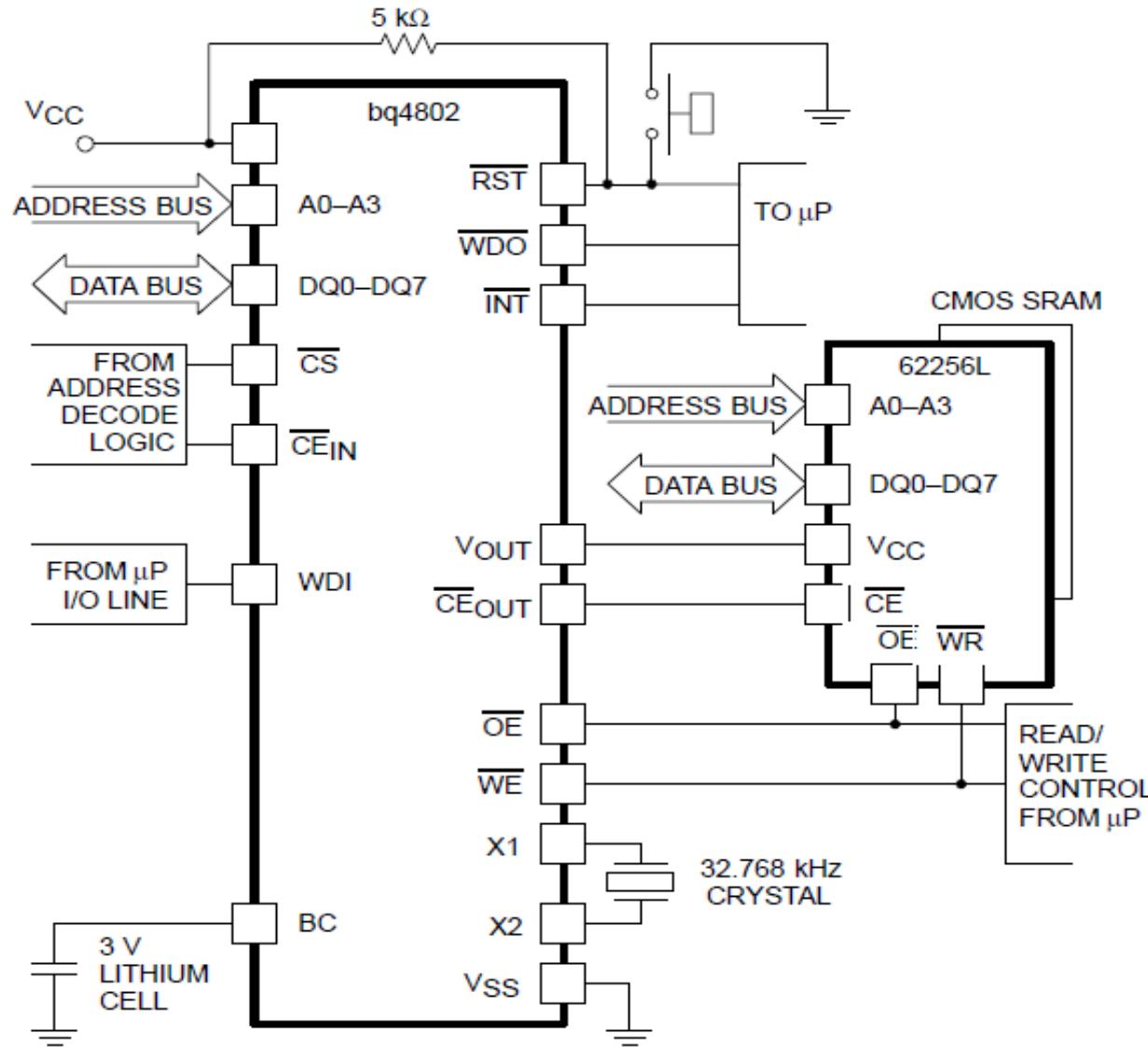
- periferic care are un **consum redus de putere**;
- degreveaza sistemul principal de **taskurile critice cu restrictii temporale**;
- este mai precis decat alte metode de masurare a timpului (timere, bucla PLL care sunt sincronizate la frecventa CPU).

# Real-Time Clock (RTC)

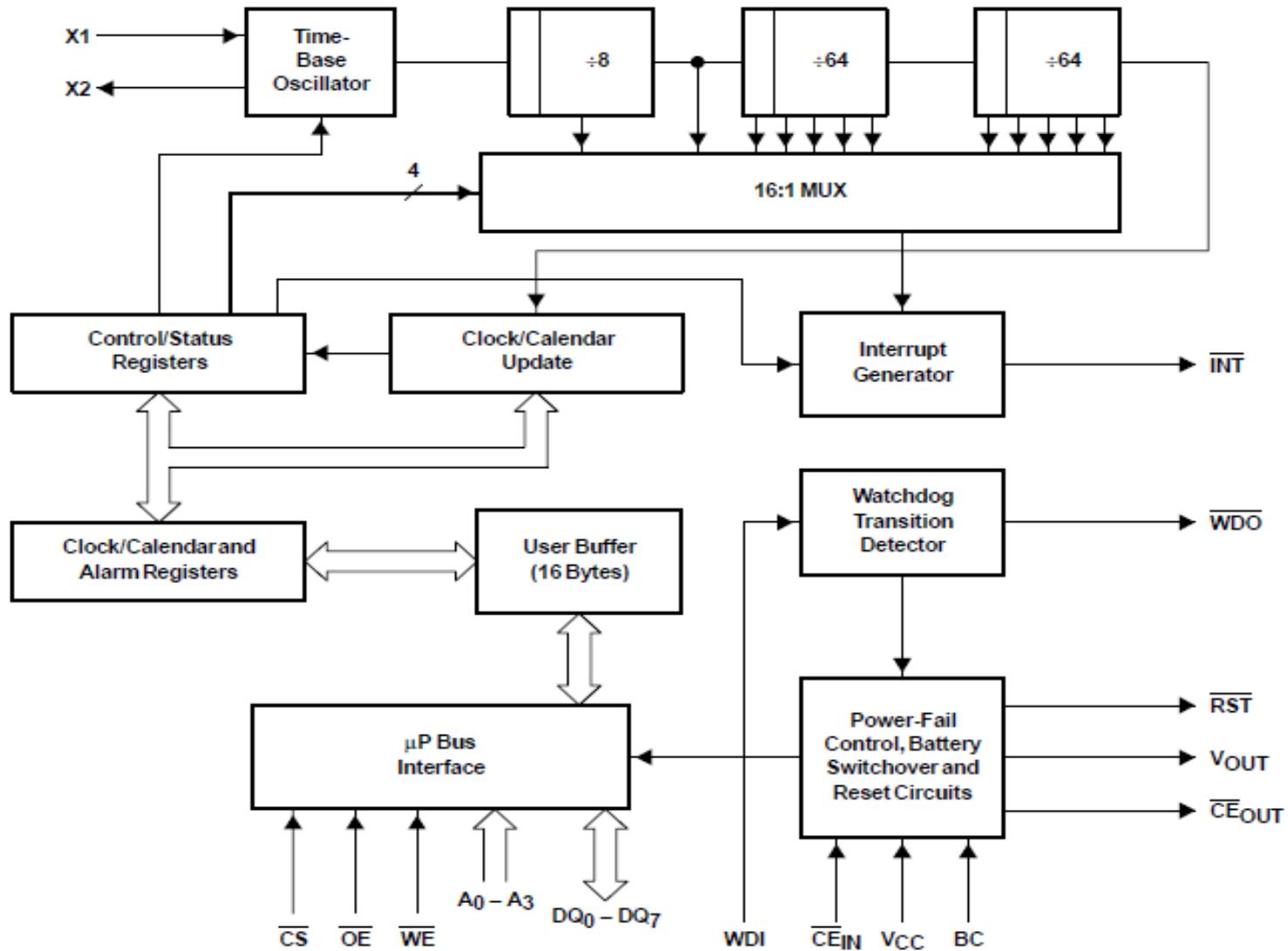
## **Exemple de utilizare:**

- ceasuri electronice si alte dispozitive de temporizare sau de acces controlat
- data loggere, puncte de vazare publice, ATM, faxuri
- receivere GPS (se compara timpul curent preluat de la RTC si momentul cand a primit ultimul pachet cu informatie valida si daca diferența e mai mica decat o valoare impusa pentru timp, datele primite pot fi utilizate la pozitionare, altfel datele sunt eliminate)

# Structura generica a unei aplicatii tipice cu circuit RTC



# Arhitectura interna generica a unui circuit RTC



# Implementarea unui RTC utilizand z80 si circuitul CTC

- Se propune dezvoltarea unei aplicatii utilizand z80 si perifericul CTC (Counter Timer Circuit) ce face parte din familia de semiconductoare a uP z80.
- Aplicatia va mentine un ceas de 24 de ore si un calendar bazata pe intreruperile de la CTC.

# Implementarea unui RTC utilizand z80 si circuitul CTC

## Algoritm:

#1. se seteaza valorile de configurare pentru circuitul CTC si se initializeaza circuitul pentru functionarea bazata pe intreruperi

- Seteaza canalul circuitului CTC
- Se scrie cuvantul de comanda pt CTC
- Se scrie constanta de timp

#2. se seteaza o data initiala de referinta (baza) si se initializeaza astfel unitatile (s, min, h, zi, luna, an)

#3. se trateaza intreruperile de CTC si se incrementeaza fiecare unitate la aparitia unei intreruperi (se decrementeaza temporizarea interna a CTC, definind astfel intervele de timp precise)

#4. se testeaza unitatile de timp: test daca anul este bisect, test zile luna februarie, test zi de final a lunii curente (vector cu numar zile/luna)

#5. se testeaza ceasul realizand un test pentru o data curenta (se va porni ceasul pana va ajunge la data impusa)

#6. testarea se poate extinde la functionarea continua

# Implementarea unui RTC utilizand z80 si circuitul CTC

- **Subrutine dezvoltate:**
  - **CLOCK** : returneaza adresa de baza pentru mentinerea variabilelor (parametrilor ceasului)
  - **ICLK** : initializeaza variabilele ceasului si intreruperile asociate
  - **CLKINT** : updateaza ceasul dupa fiecare intrerupere

# Implementarea unui RTC utilizand z80 si circuitul CTC

- Descrierea procedurii de lucru:
  - CLOCK : incarca adresa de baza a variabilelor ceasului in HL. Variabilele ceasului sunt memorate in urmatoarea ordine : pulsuri, s, min, h, zile, luni, LSB al anului, MSB al anului
  - ICLK : se initializeaza CTC, sistemul de intreruperi si variabilele ceasului. Valoarea arbitrara de inceput este ora 00:00:00, 1 Ianuarie 1980.
  - CLKINT: decrementeaza numarul de pulsuri ramas si updateaza restul variabilelor daca este necesar. Limitari apr la secunde si minute (<60) si la ore (<24).

Ziua din luna trebuie sa fie  $\leq$  cu ultima zi din luna curenta (se va seta un vector cu numarul de zile din fiecare luna)

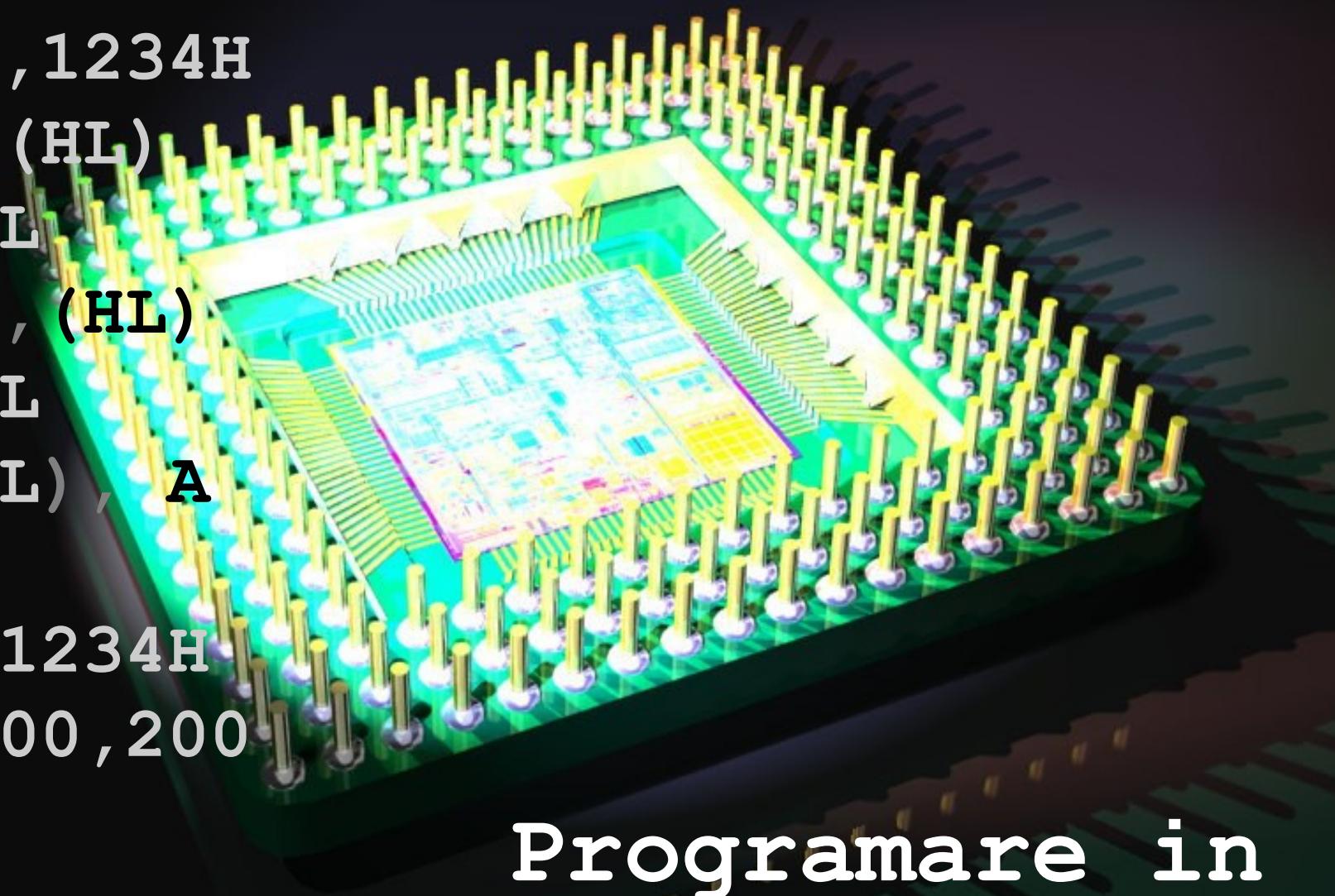
Anul se testeaza pentru a verifica daca este bisect si in cazurile in care apare carry (la comparatii) se reinitializeaza variabilele corespunzator tipului lor

**010011010111010101101100011101000111  
01010110110101100101011100110110001  
10010000001110000011001010110111001  
11010001110010011101010010000001100  
00101110100011001010110111001110100  
0110100101100101001000000100001**

**sau altfel spus ...**

**Multumesc pentru atentie !**

```
.ORG 100H  
LD HL,1234H  
LD A, (HL)  
INC HL  
ADD A, (HL)  
INC HL  
LD (HL), A  
.END  
.ORG 1234H  
.DB 100,200
```



Programare in  
Limba de Asamblare

# Intreruperi (INT/NMI)

**Intreruperile** sunt **evenimente asincrone** generate de **periferice** sau de **conditii interne** de executie care determina procesorul sa opreasca executia taskului current in favoarea **executiei actiunii** corespunzatoare **perifericului** care a generat intreruperea.

**Procesorul Z80** poate trata intreruperi **mascabile** (3 moduri, **mod0**, **mod1**, **mod2**) si **nemascabile**.

# Intreruperi (INT/NMI)

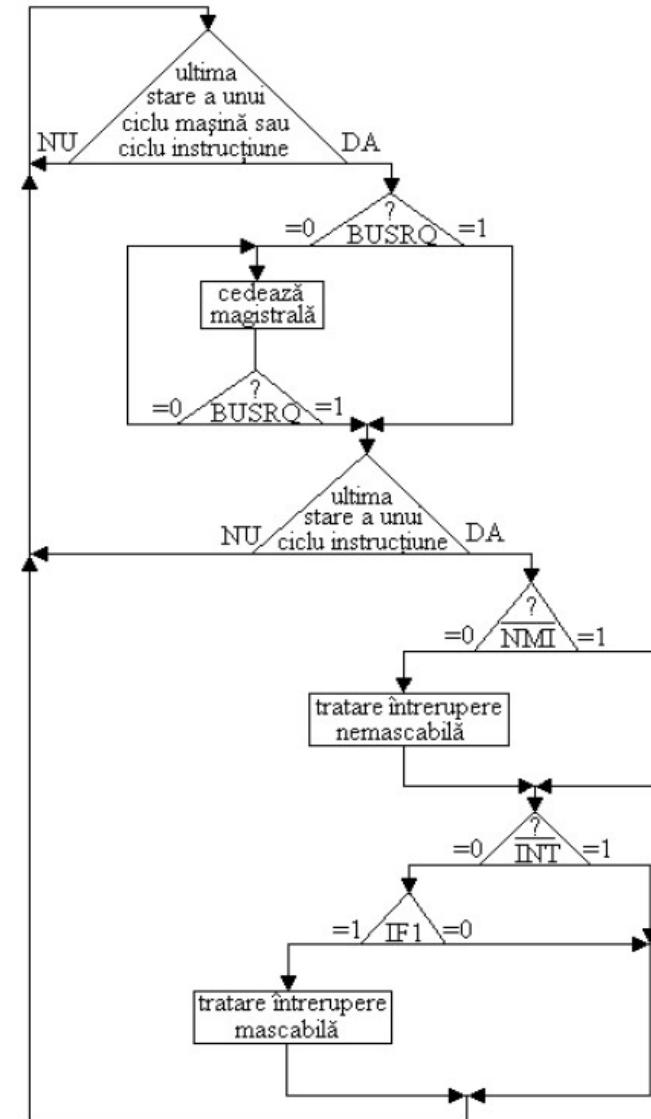
## Avantajele INT/NMI:

- Aceste **moduri de lucru** ale uP contribuie la **creșterea flexibilității** uP deoarece impreuna cu cererile de magistrală (**BUSRQ**), îñtreruperile permit **comunicarea** **într-un mod asincron** cu alte **dispozitive** și **la inițiativa acestora**.
- Prin utilizarea acestor facilități se **degrevează** uP de anumite **operații** care trebuie făcute **ciclic** și la **intervale de timp relativ mici**.

# Intreruperi (INT/NMI)

Având în vedere faptul că atât cererile de magistrală (**BUSRQ**) cât și îintreruperile (**INT/NMI**) sunt generate de către dispozitivele externe, uP le tratează într-o ordine bine stabilită.

**Organograma** prin care se ilustrează modul de tratare a acestor evenimente.



# Intreruperi (INT/NMI)

- **Cererile de magistrală (BURQ)** sunt prioritare și se tratează la **sfârșitul ciclurilor mașină**.
- **Întreruperile nemascabile (NMI)** sunt preluate în orice moment și se tratează la **sfârșitul instrucțiunii** în curs de execuție,
- **Întreruperile mascabile (INT)** au prioritatea cea mai mică și sunt preluate și tratate la **sfârșitul instrucțiunii** în curs de execuție, numai dacă acestea (întreruperile nemascabile) au fost **activate în prealabil (IF1=1)**.

# Intreruperi (INT/NMI) la uP Z80

## Intreruperi Mascabile Vs. Intreruperi Nemascabile

**Diferenta** intre aceste doua tipuri consta in faptul ca **intreruperile namascabile nu pot fi neglijate** de catre **processor** si sunt **tratate** atunci **cand apar** in timp ce **intreruperile mascabile** pot fi neglijate in tratare daca au fost **dezactivate**.

# Intreruperi (INT/NMI) la uP Z80

## Intreruperi Mascabile Vs. Intreruperi Nemascabile

### Întreruperile nemascabile :

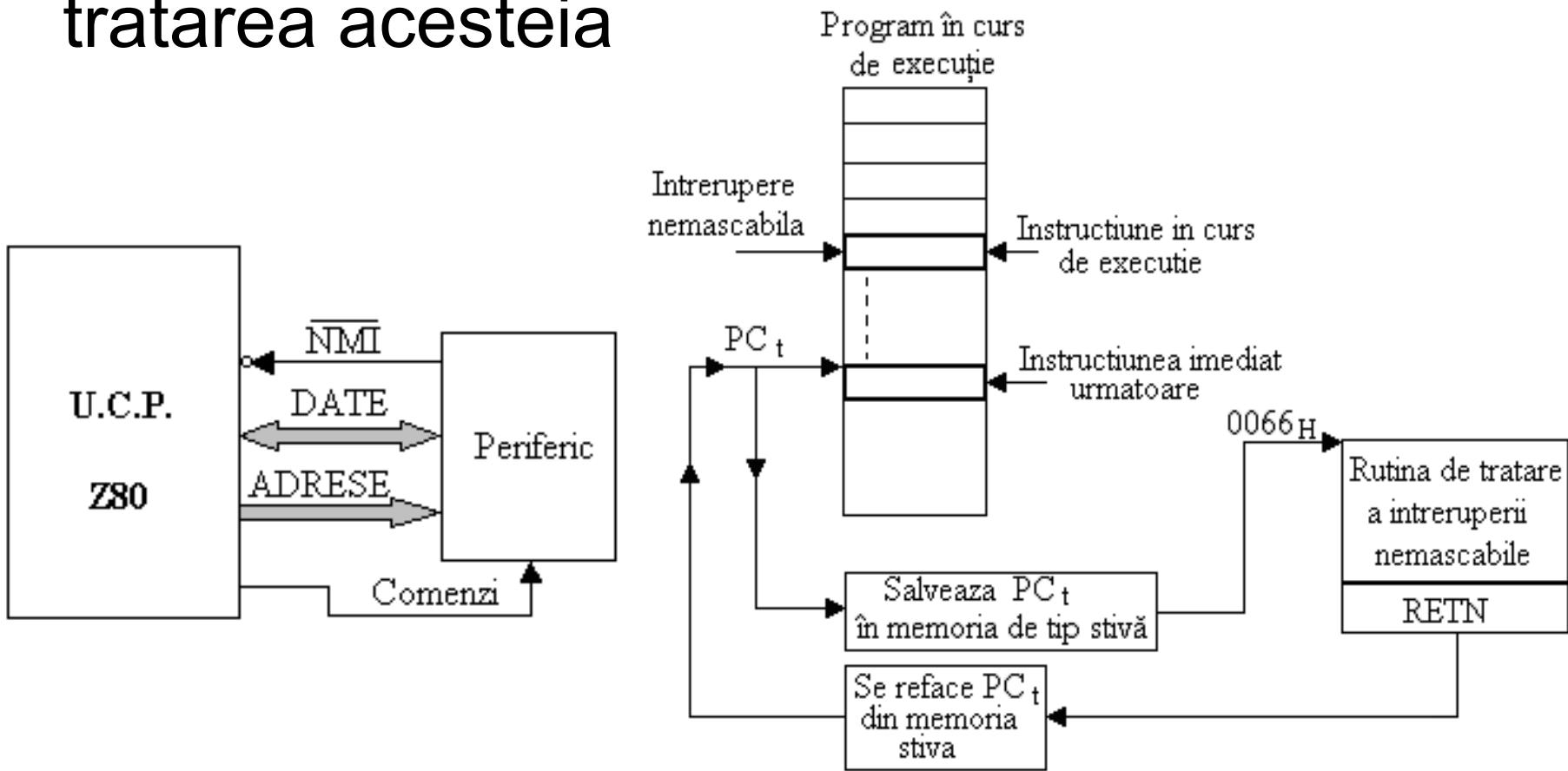
- sunt **prioritare** față de îնtreruperile **mascabile** ;
- mai puțin prioritare față de o **cerere de magistrală(BUSRQ)**.

### Reactia uP z80 :

La apariția unei înntreruperi, microprocesorul salvează PC-ul instrucțiunii imediat următoare în memoria de tip stivă și trece la tratarea înntreruperii nemascabile prin execuția **rutinei de tratare a înntreruperii** care se află la adresa 0066H.

# Intreruperi (NMI) la uP Z80

**Modul de conectare a unui periferic care generează o intrerupere nemascabilă și tratarea acesteia**

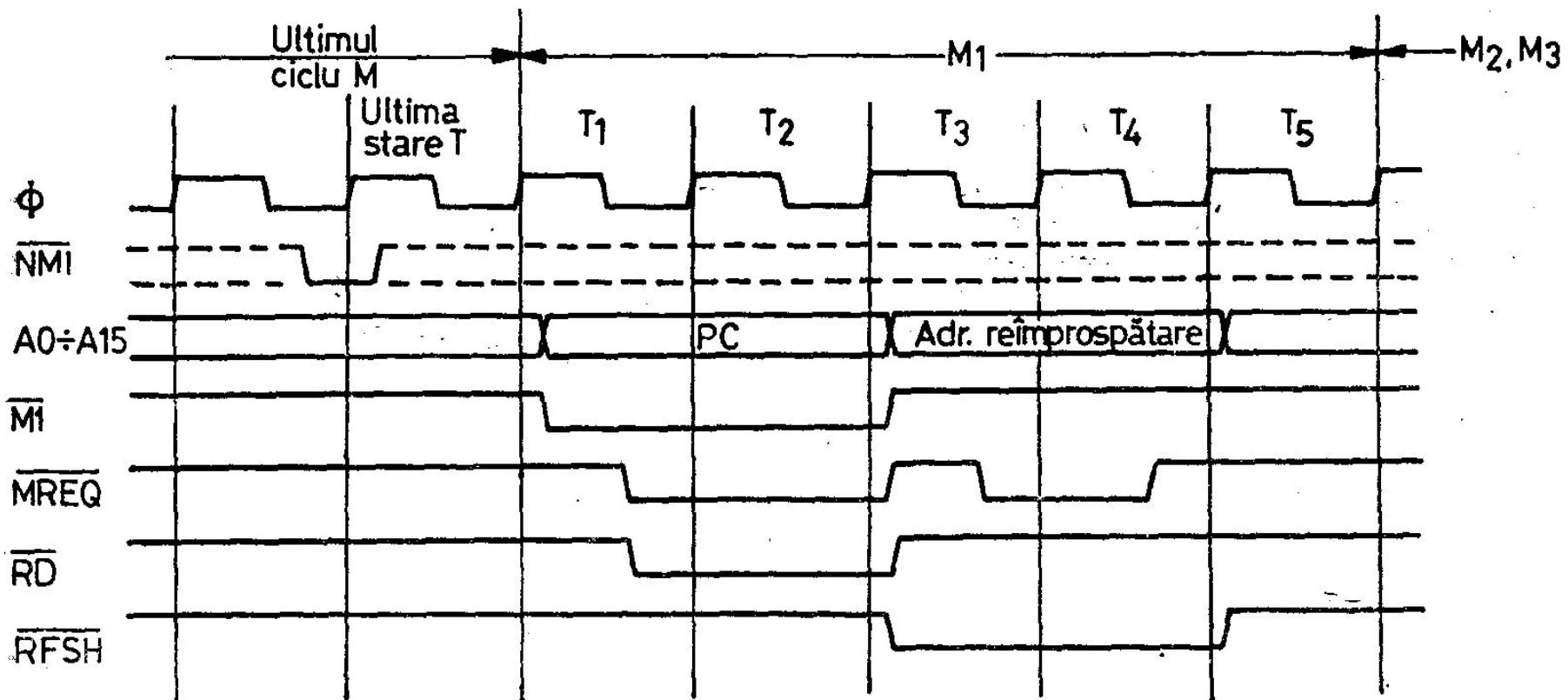


# Intreruperi (NMI) la uP Z80

Cererea de **întrerupere nemascabilă (NMI IRQ)** este **reținută** în orice moment, chiar și în situația în care microprocesorul are cedate magistralele, dar **se tratează numai la sfârșitul instrucțiunii** în curs de execuție.

# Ciclul de achitare intreruperi (NMI) la uP Z80

Ciclul de cerere-achitare (IRQ-IACK) a  
intreruperii nemascabile



# Ciclul de achitare intreruperi (NMI) la uP Z80

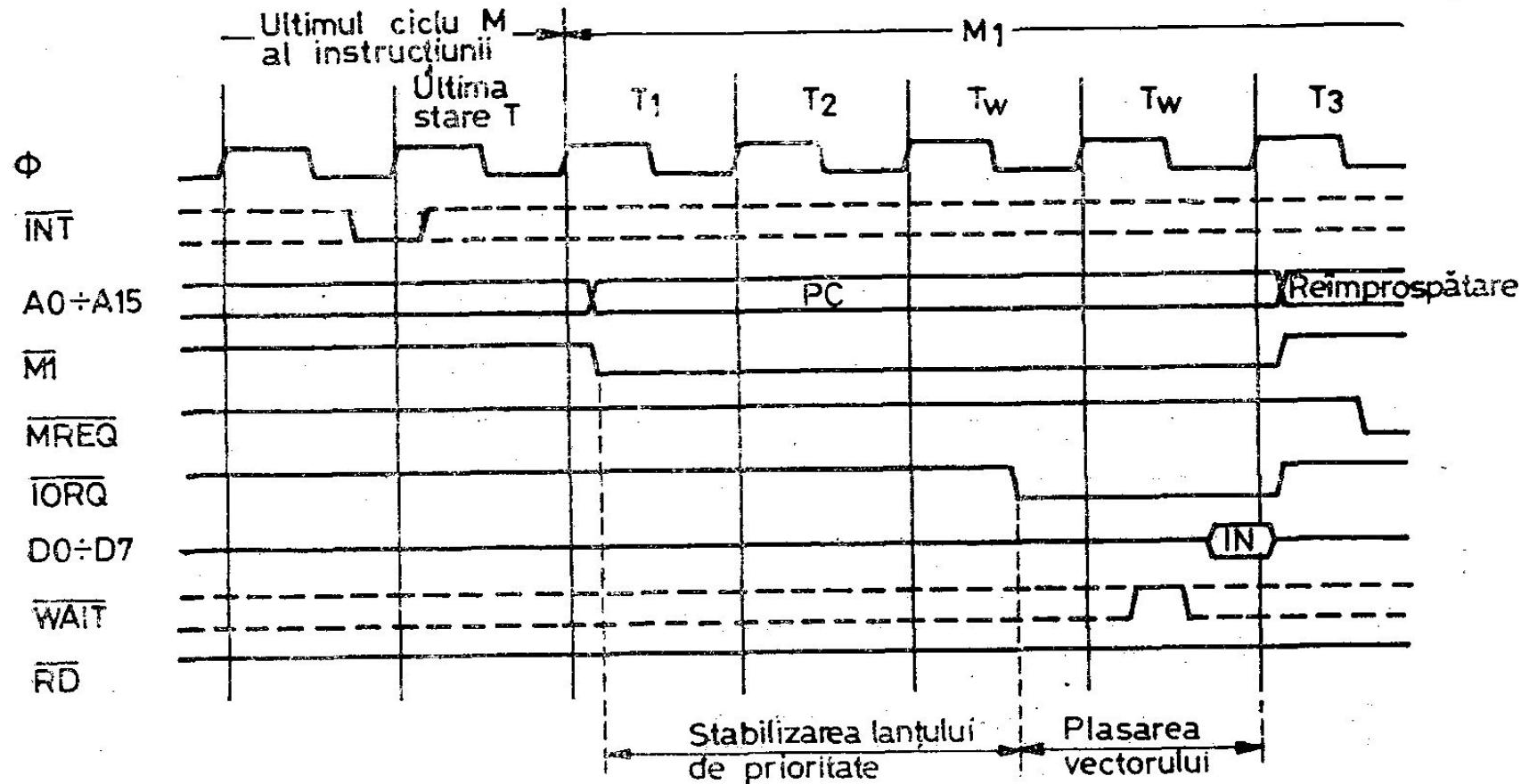
- Semnalul NMI se esantioneaza in acelasi timp cu INT pe frontul pozitiv al CLK, in ultima stare T a instructiunii in curs de executie.
- NMI este prioritara fata de INT si nu se poate masca prin program.

**Context de cerere-achitare intern :**

1. Un impuls pe linia NMI determina **setarea unui bistabil** intern, care este **testat la sfarsitul fiecarei instructiuni**.
2. **Raspunsul uP la NMI este asemenator cu un ciclu de citire din memorie**, diferența constand in faptul ca in acest caz uP **ignora codul** plasat de memorie pe magistrala de date si **executa o instructiune de restart** la adresa **rutinei de tratare a intreruperii** (plasata la adresa 0066h) .

# Ciclul de achitare intreruperi (INT) la uP Z80

Ciclul de cerere-achitare (IRQ-IACK) a  
intreruperii mascabile



# Ciclul de achitare intreruperi (INT) la uP Z80

## Semnalul INT :

- este esantionat la **finalul** fiecarei **instructiuni**, pe **frontul pozitiv** al ultimei stari T.
- daca e activ va fi **acceptat** doar daca **bistabilul intern de validare**, IFF1, comandat prin soft este pe 1 si daca semnalul de cerere magistrala **BUSRQ** nu e pe 0.

Daca intreruperea se accepta se va declansa un **ciclu M1(fetch) special**, care va da in mod unic dispozitivul periferic care a generat intreruperea si caruia ii va cere sa plaseze pe magistrala de date un **vector de 8 biti**.

# Intreruperi (INT) la uP Z80

**Întreruperile mascabile sunt preluate și tratate la sfârșitul execuției instrucțiunii curente numai dacă acestea au fost activate în prealabil.**

**Microprocesorul Z80 are trei moduri de tratare a întreruperilor mascabile**, moduri care se stabilesc cu ajutorul următoarelor instrucțiuni: **IM0, IM1, IM2**.

**Modul doi (IM2) este specific familiei de periferice Z80.**

- **activarea** întreruperilor se face cu instrucțiunea **EI**, iar **dezactivarea** acestora cu instrucțiunea **DI**.
- la **resetarea** microprocesorului **întreruperile mascabile sunt dezactivate**, iar modul setat este **modul zero**.

**Modul unu (IM1) poate fi utilizat de perifericele care nu aparțin familiei Z80.**

# Intreruperi mod0 (INT IM0) la uP Z80

**Modul zero** este compatibil cu modul de întrerupere al microprocesorului 8080.

**Context de executie** (cerere si achitare – IRQ si IACK) :

**#1. Perifericul care generează o**  
întrerupere plasează pe magistrala de date codul instrucțiunii **RST p** (**RESTART p**), unde **p = 00H, 08H, 10H, 18H, ..., 38H**, sau codul unei instrucțiuni **CALL adr.**

# Intreruperi mod0 (INT IM0) la uP Z80

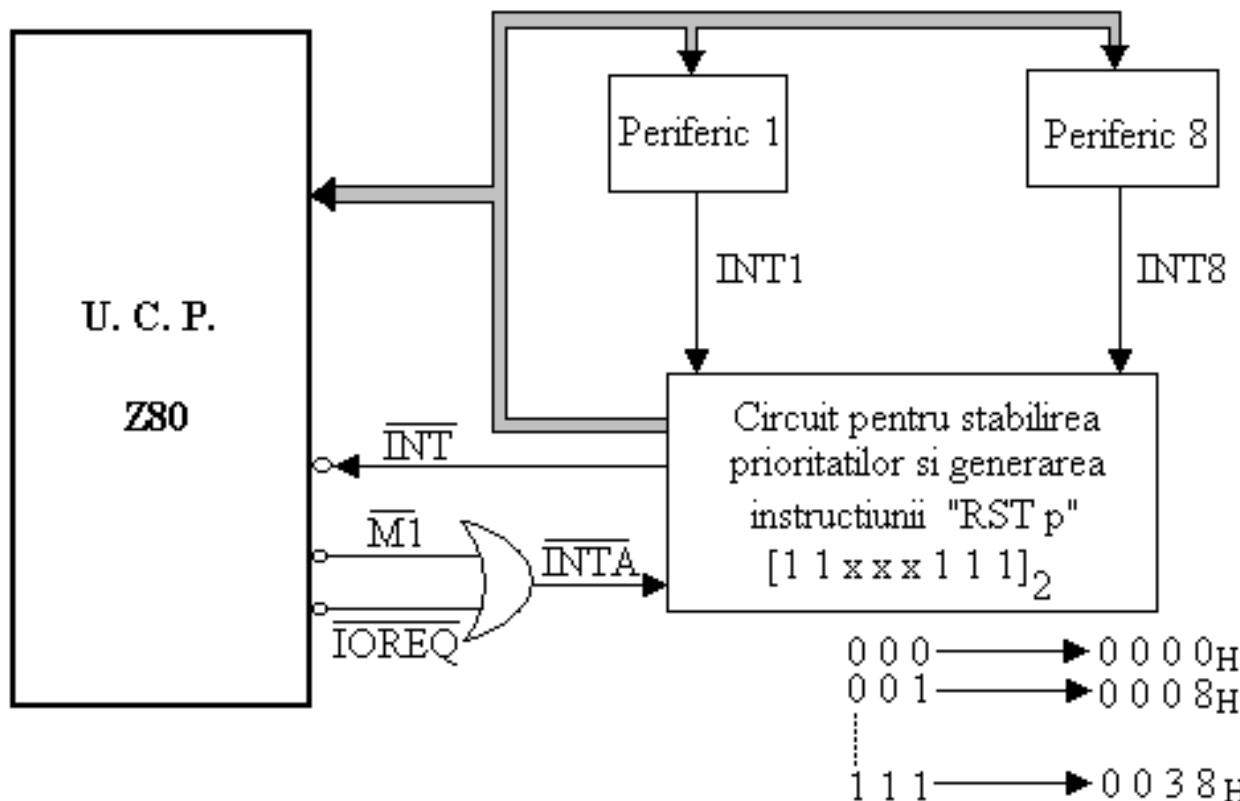
**Context de executie** (cerere si achitare – IRQ si IACK) :

**#2.** În **cazul** plasării instrucțiunii **RST p** se **salvează** **PC-ul** instrucțiuni imediat următoare și se trece la **execuția instrucțiunilor** de la adresa **0000H** sau **0008H ... 0038H**, în funcție de **valoarea lui p**.

**#3.** Dacă programul permite **tratarea intreruperii** în **spațiul de adresă de 8 octeți**, atunci la adresa respectivă **nu este necesară o instrucțiune de salt** necondiționat la o altă adresă unde se găsește adevărata **rutină de tratare a intreruperilor** generată de perifericul respectiv.

# Intreruperi mod0 (INT IM0) la uP Z80

**Modul de generare și tratare a unei  
întreruperi în cazul în care perifericul  
plasează instrucțiunea RST p.**



# Intreruperi mod0 (INT IMO) la uP Z80

## Pasii ciclului de achitare a intreruperii (IACK) :

- p1. in momentul generării unei intreruperi de către cel puțin un periferic,
- p2. circuitul specializat generează o intrerupere către uP,
- p3. uP răspunde prin **semnalul de achitare  $IOREQ+M1=INTA$** ,
- p4. in acest moment circuitul specializat pune pe **magistrala de date** instrucțiunea ***RST p*** corespunzătoare **perifericului** cu cea mai **mare prioritate** care a generat intrerupere (**INT**).

# Intreruperi mod0 (INT IM0) la uP Z80

**Pasii ciclului de achitare a intreruperii (IACK) :**

- p5. uP tratează Întreruperea prin rularea rutinei de tratare (ISR) asociate perifericului care a generat respectiva Întrerupere.
- p6. un **circuit specializat** auxiliar rezolvă problema priorităților cât și a generării codului instrucțiunii **RST p** (ex.: circuitul I8214).

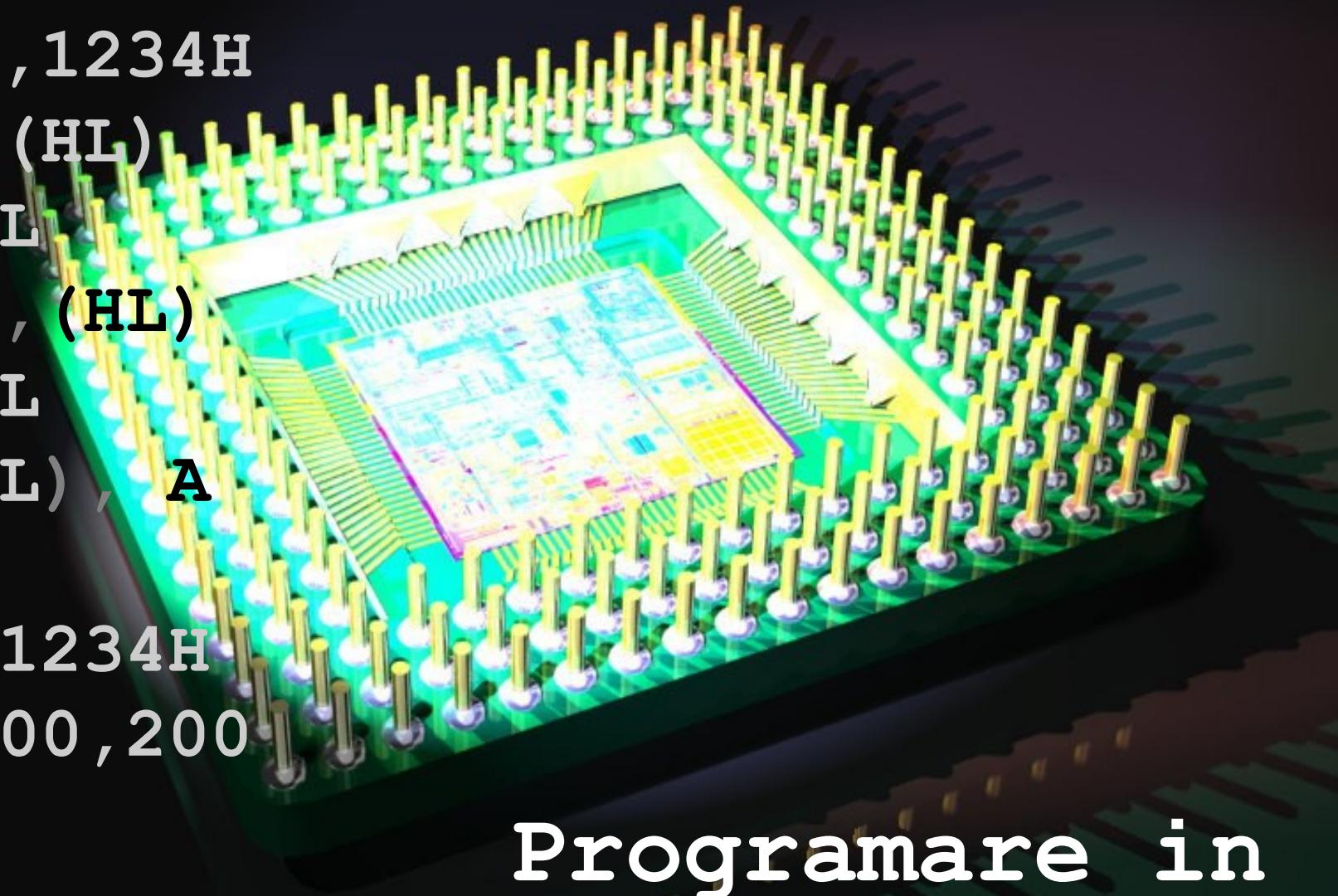
Dacă se utilizează **controlerul de întrerupere CI 8259**, atunci **rutinele de tratare a întreruperilor** pot fi plasate **oriunde în memoria microsistemului**, deoarece acestea sunt memorate într-o **tabelă**, iar accesul la adresa **rutinei** se face prin intermediul a **doi octeți generați de controler**.

**010011010111010101101100011101000111  
01010110110101100101011100110110001  
10010000001110000011001010110111001  
11010001110010011101010010000001100  
00101110100011001010110111001110100  
0110100101100101001000000100001**

**sau altfel spus ...**

**Multumesc pentru atentie !**

```
.ORG 100H  
LD HL,1234H  
LD A, (HL)  
INC HL  
ADD A, (HL)  
INC HL  
LD (HL), A  
.END  
.ORG 1234H  
.DB 100,200
```



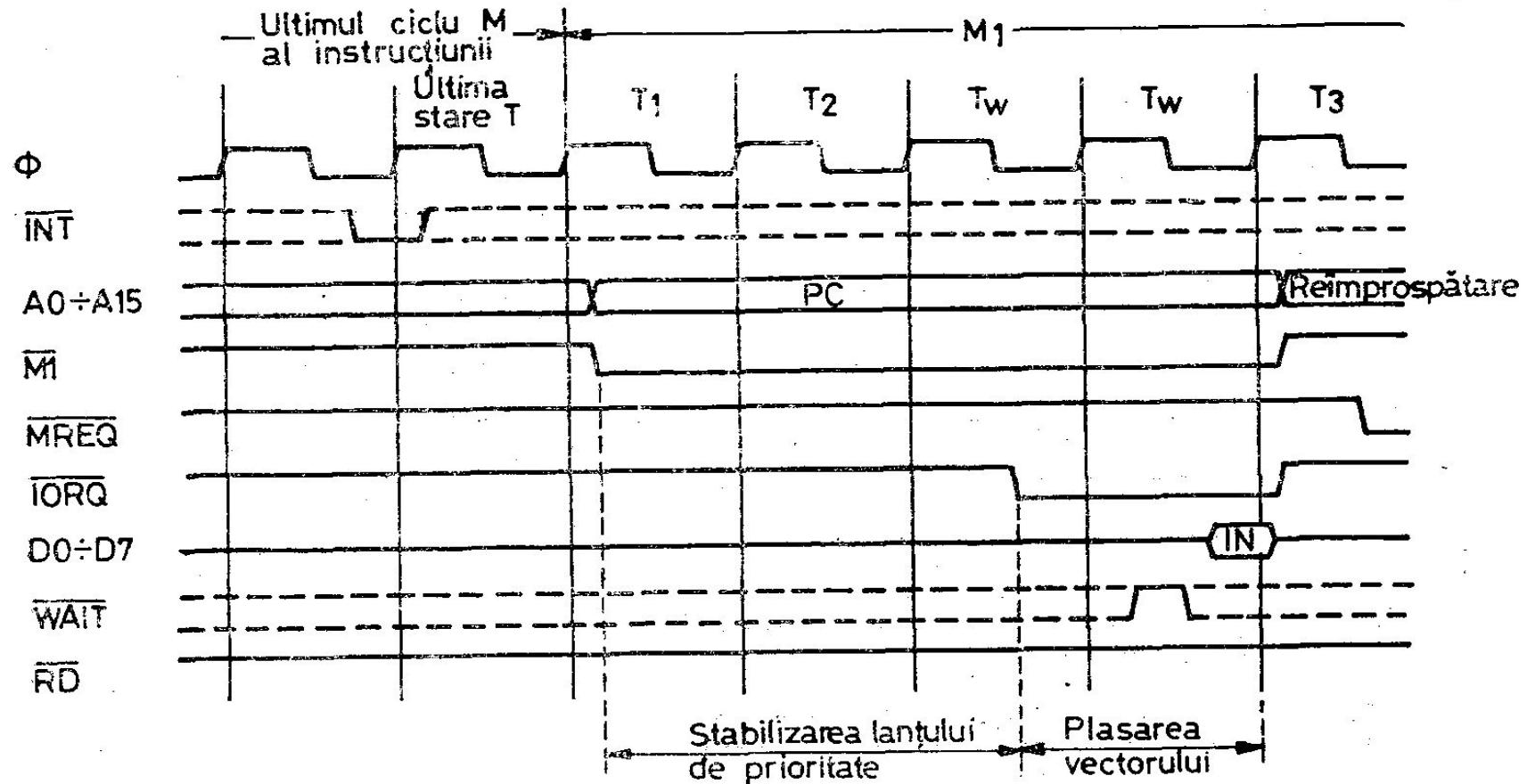
Programare in  
Limba de Asamblare

# Intreruperi mod1 (INT IM1) la uP Z80

- **Întreruperile mascabile sunt preluate și tratate la sfârșitul execuției instrucțiunii curente numai dacă acestea au fost activate în prealabil.**
- Microprocesorul Z80 are trei moduri de tratare a **întreruperilor mascabile**, moduri care se stabilesc cu ajutorul instrucțiunilor: **IM0, IM1, IM2**.
- **Modul unu (IM1)** poate fi utilizat de perifericele care nu aparțin familiei Z80.

# Intreruperi mod1 (INT IM1) la uP Z80

Ciclul de cerere-achitare (IRQ-IACK) a intreruperii mascabile



# Intreruperi mod1 (INT IM1) la uP Z80

## Semnalul INT :

- este esantionat la **finalul** fiecarei **instructiuni**, pe **frontul pozitiv** al ultimei stari T.
- daca e activ va fi **acceptat** doar daca **bistabilul intern de validare**, IFF1, comandat prin soft este pe 1 si daca semnalul de cerere magistrala **BUSRQ** nu e pe 0.

Daca intreruperea se accepta se va declansa un **ciclu M1(fetch) special**, care va da in mod unic dispozitivul periferic care a generat intreruperea si caruia ii va cere sa plaseze pe magistrala de date un **vector de 8 biti**.

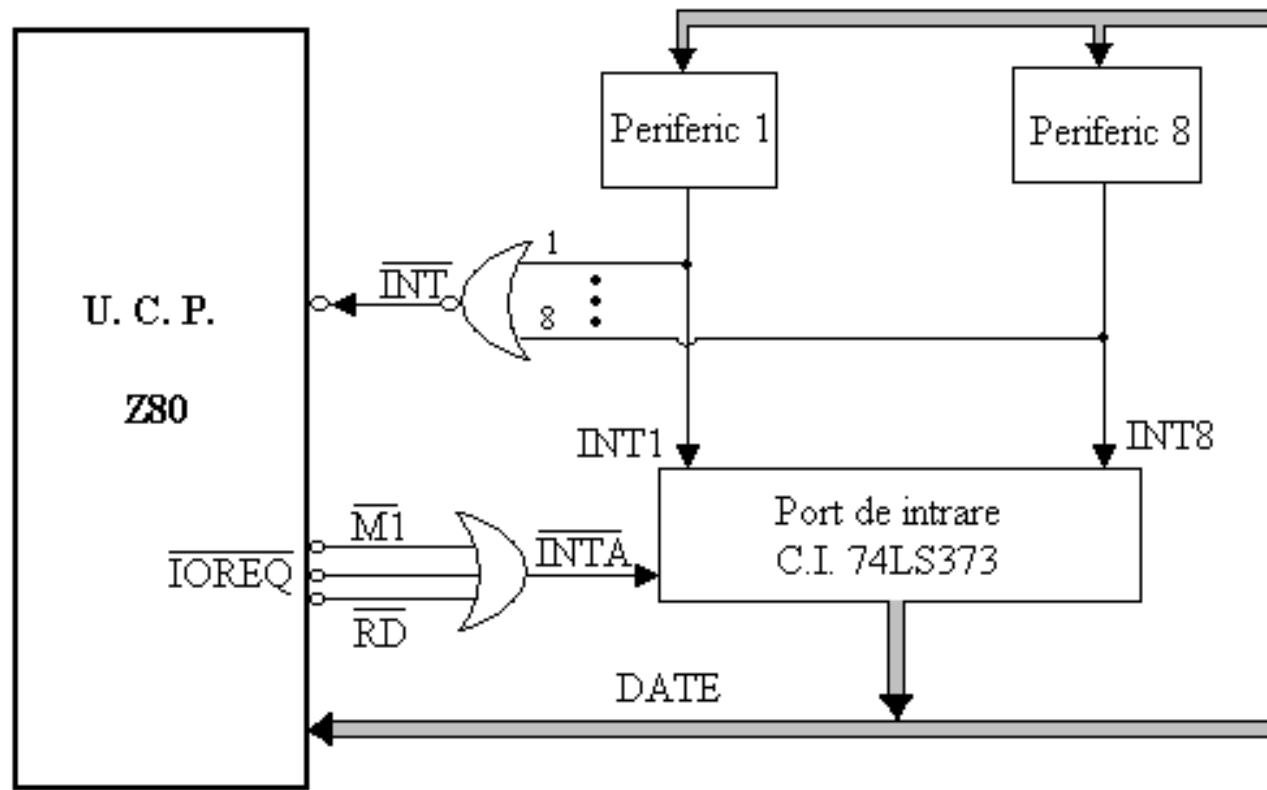
# Intreruperi mod1 (INT IM1) la uP Z80

**Context de executie** (cerere si achitare – IRQ si IACK) :

- #1. se testeaza daca **întreruperile** sunt **validate**, si daca la **sfârșitul instrucțiunii** în curs de executie **pinul INT** se afla la '0' logic,
- #2. uP **salveaza PC-ul** instructiunii imediat urmatoare,
- #3. uP trece la **tratarea intreruperii** executand **rutina de tratare** care se afla la **adresa 0038H**.
- #4. Acet **mod de intrerupere** se poate utiliza și în cazul când există mai **multe periferice** care generează o **întrerupere**, urmând ca **prioritatea** să se stabilească prin **interrogare**.

# Intreruperi mod1 (INT IM1) la uP Z80

Schema de principiu în cazul **modului 1 de intrerupere** prin care se tratează intreruperi de la 8 periferice



# Intreruperi mod1 (INT IM1) la uP Z80

**Dialogul cu perifericele se desfășoară după protocolul următor:**

- > un periferic trimite pe linia *INT i* o cerere de intrerupere (IRQ),
- > se generează un semnal egal cu ‘1’ logic și plasează pe una din intrările portului 74LS373.
- > se testează dacă cel puțin unul din semnalele *INT i* este ‘1’ logic, atunci semnalul *INT l* devine ‘0’ logic, moment în care se anunță uP că cel puțin un periferic a generat o intrerupere.

# Intreruperi mod1 (INT IM1) la uP Z80

**Dialogul cu perifericele se desfasoara dupa protocolul următor:**

**uP :**

- > ia **cunoștință** de intrerupere (**IACK**),
- > **salvează PC** corespunzător **instrucțiunii** imediat **următoare**,
- > **trece la executia rutinei de tratare a intreruperii de la adresa 0038H.**

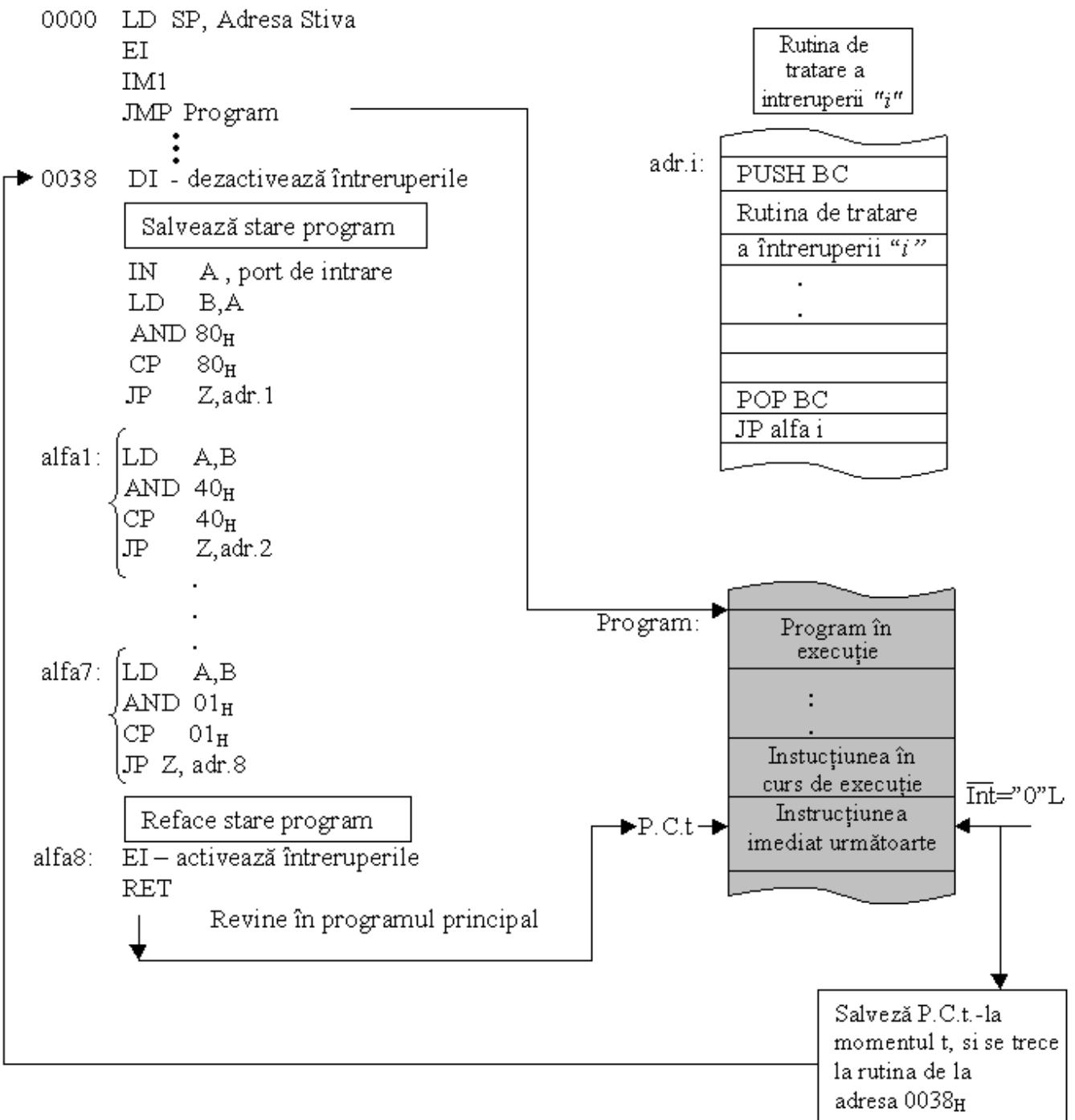
**În cadrul rutinei de tratare a intreruperii (ISR):**

- > **se citește data de la perifericul de intrare,**
- > **se trece la tratarea intreruperii** perifericului cu **prioritate maximă**,
  - > **se revine în programul principal sau,**
  - > **se trateaza intreruperea** imediat urmatoare in **ordinea prioritatilor**.

Achitarea intreruperii trebuie sa o faca rutina de tratare a intreruperilor prin resetarea cererii (INT i =0).

# Intreruperi mod1 (INT IM1) la uP Z80

Organograma prin care se realizeaza tratarea intreruperilor de mod 1.



# Intreruperi mod1 (INT IM1) la uP Z80

În cadrul acestui **mod de întrerupere** :

- se **simplifică structura hardware** în ceea ce privește **stabilirea priorităților**,
- **crește durata** privind **determinarea perifericului** care a generat **întreruperea**, deoarece aceasta (**stabilirea priorităților**) se face **prin program**.

# Intreruperi mod1 (INT IM1) la uP Z80

## Aplicatii – Circuit comunicatii seriale

### Z8440AB1 (ST Micro)

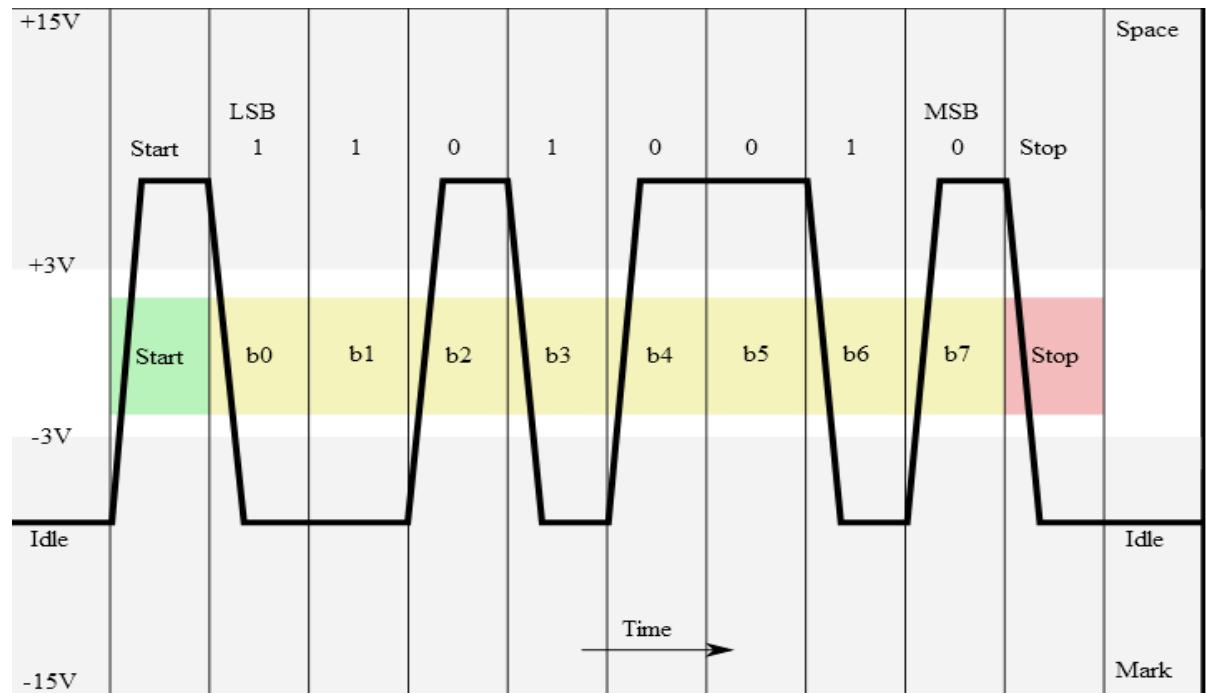
#### Circuitul Z8440AB1

- se utilizeaza in aplicatiile de **comunicatii de date** asigurand **interfatarea independenta a doua canale**,
- implementeaza functii de circuit **UART** (Universal Asynchronous Receiver Transmitter) dar si **USART** (Universal Synchronous/Asynchronous Receiver Transmitter).

# Specifcul comunicatiilor seriale (Protocol RS-232)

In comunicatie  
**RS232:**

- datele se transmit serial,
- transmisie pe un singur fir,
- **codificate prin nivelele de tensiune fata de masa.**



Transmisia caracterului ASCII "K" – 4bH

**Nivelul logic 1 este codificat ca o tensiune negativa de -12V, 0 logic codificat printr-o tensiune pozitiva de +12V.**

# Specificul comunicatiilor seriale (Protocol RS-232)

**Standardul RS232** defineste si semnale “de handshake” pentru controlul fluxului de date.

**DTR – Data Terminal Ready** – este iesire din portul de comunicatie DTE si intrare pentru DCE.

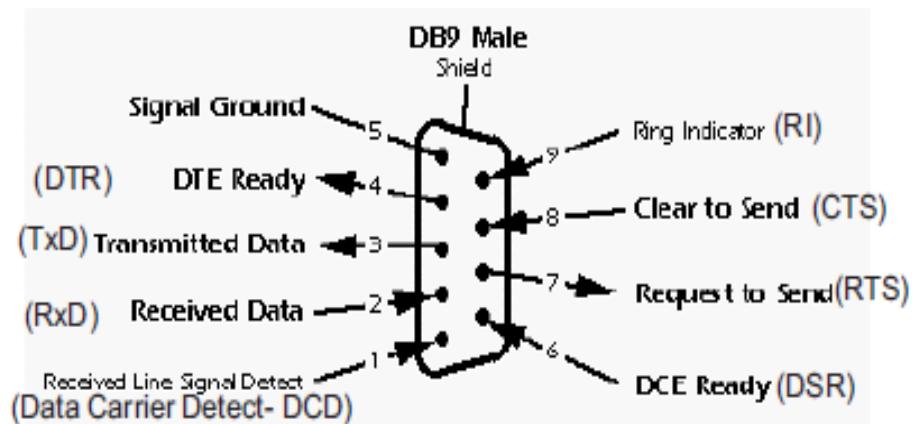
**DSR – Data Set Ready** – este raspunsul la DTE – intrare pentru echipamentul DTE si iesire din DCE (indica modem alimentat si pregatit de emisie)

**RTS – Request to Send** – cerere de transmisie (iesire din DTE, intrare pentru DCE)

**CTS – Clear to Send** (raspunsul DCE la o cerere de emisie. E activ cand modemul are loc in bufferul de receptie)

**RI** – ring indicator este specific modemurilor pentru linii telefonice. Se activeaza la detectarea tonului de apel. Este iesire din DCE si intrare pentru DTE.

**DCD – Data Carrier Detect** (prezenta purtatoare) – este de asemenea specific modemurilor si indica faptul ca modem-ul a intrat in dialog cu perechea lui de la celalalt capat al liniei de comunicatie.



# Intreruperi mod1 (INT IM1) la uP Z80

## Aplicatii in comunicatii seriale

Aplicatia propusa se refera la implementarea unui **Terminal de comunicatii** ce implementeaza urmatorul mecanism de comunicatii parametrizat:

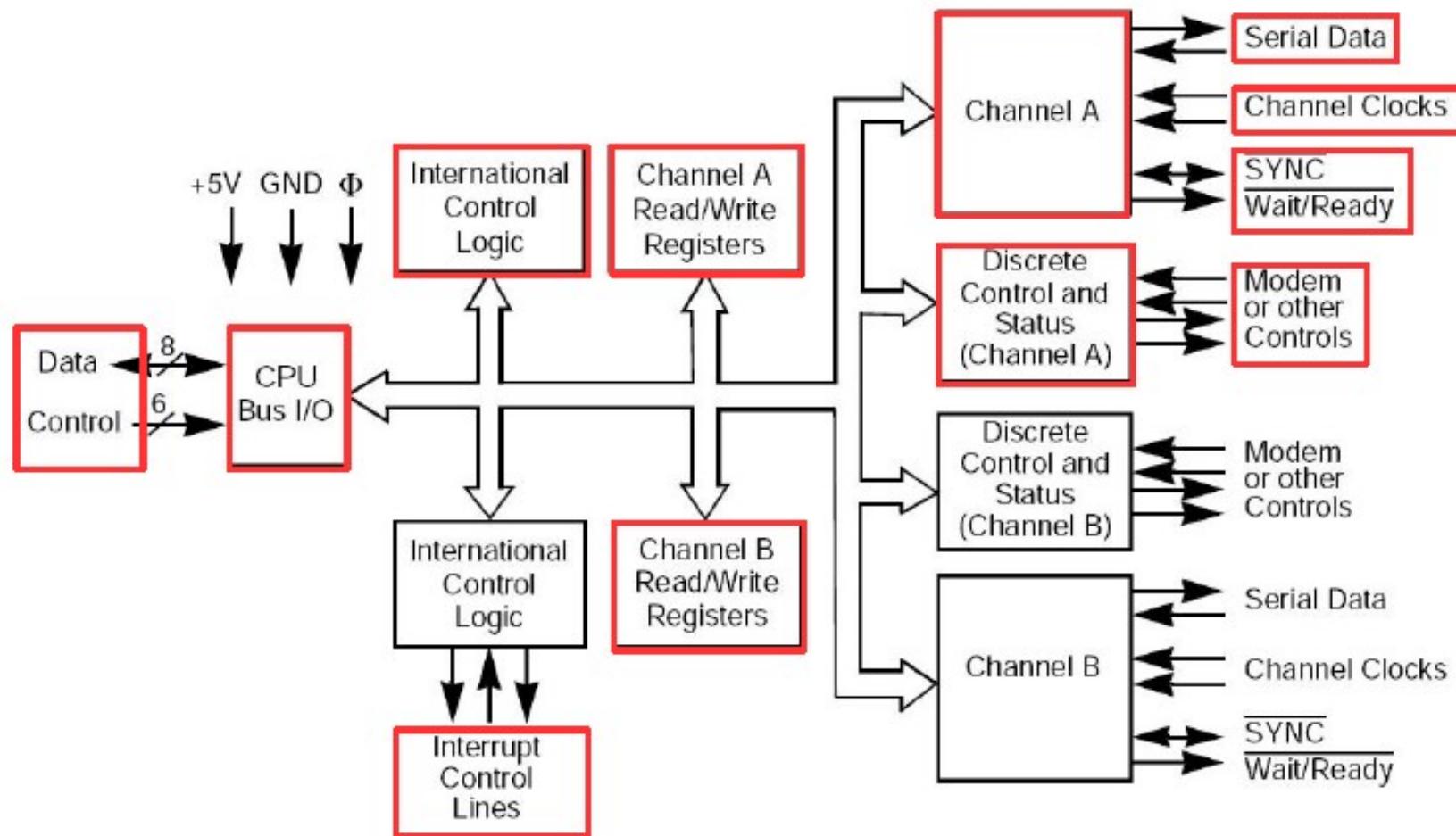
- Circuitul se programeaza pentru Terminal mode **RS232 asincron** (nu se transmite si semnalul de CLK)
- **Baudrate**: 9600 Baud/sec
- **Biti de stop**: 1
- **Biti de start**: 1
- **Dimensiunea caracterului**: 8 biti

Pentru conectarea la dispozitivul cu care comunica sistemul cu Z80 in mod Terminal va utiliza o linie **NULL-MODEM**. (configuratie TX-RX, RX-TX, GND-GND).

# Intreruperi mod1 (INT IM1) la uP Z80

## Aplicatii in comunicatii seriale

### Schema bloc a circuitului Z8440AB1



# Intreruperi mod1 (INT IM1) la uP Z80

## Aplicatii in comunicatii seriale

**Detalii de implementare:**

1. **Initializarea circuitului Z8440AB1**
2. **Implementarea mecanismului de intreruperi**
  - a. **Definirea tablei de vectori de intreruperi**
  - b. **Initializarea circuitului CTC pt CLK de RX si TX dar fara a fi transmis pe linia seriala**
3. **Initializarea uP**
4. **Managementul pe RX/TX al canalelor circuitului Z8440AB1**
  - a. Implementarea mecanismelor de **control al fluxului**
  - b. **Definirea rutinelor de tratarea a intreruperilor de receptie / transmisie**
5. **Implementarea rutinei de transmisie a unui caracter**

# Intreruperi mod1 (INT IM1) la uP Z80

## Aplicatii in comunicatii seriale

Subrutinele necesare in proiectarea unei aplicatii :

```
0001 0000      ; Implementarea unui sistem pentru comunicatii seriale utilizand z80 si Z8440AB1 (ST Micro)
0002 0000      ; codul introduce doar rutinele necesare dezvoltarii intr-un sistem real nu si un program
0003 0000      ; de test intrucat acesta depinde de contextul de proiectare a aplicatiei
0004 0000
0005 0000      ; Definirea adresei hardware pentru porturile circuitului de comunicatie seriala
0006 0000      INIT:
0007 0000      SERIAL_A_D .equ 4h ; adresa port date
0008 0000      SERIAL_A_C .equ 6h ; adresa port control
0009 0000
0010 0000      ; Setarea canalului 0 pentru circuitul CTC utilizat la generarea CLK pt RX si TX
0011 0000      CH0      .equ 0h
0012 0000
0013 0000      ; La receptia unui caracter circuitul genereaza o intrerupere catre uP
0014 0000      ; acesta face un salt in memorie la adresa ISR aflata la RX_CHA_AVAILABLE
0015 0000      ; se va tine cont si de conditiile speciale de transmisie (ex:buffer full)
0016 0000      ; pentru care am creat o rutina SPEC_RX_CONDITION
0017 0000
0018 0000      INT_VEC:
0019 0000          .org 0Ch
0020 000C 63 00      .defw RX_CHA_AVAILABLE
0021 000E          .org 0Eh
0022 000E 91 00      .defw SPEC_RX_CONDITION
0023 0010
```

# Intreruperi mod1 (INT IM1) la uP Z80

## Aplicatii in comunicatii seriale

```
0023 0010
0024 0010 ; Initializarea circuitului
0025 0010 ; configurarea pentru a genera intrerupere
0026 0010 ; pentru toate caracterele receptionate
0027 0010
0028 0010 SERIAL_A_RESET:
0029 0010 ;set up TX and RX:
0030 0010 3E 30 ld a,00110000b ;scrive in reg WR0 al circuitului: reset eroare, selecteaza WR0
0031 0012 D3 06 out (SERIAL_A_C),A
0032 0014 3E 18 ld a,018h ;scrive in WR0: reset canal
0033 0016 D3 06 out (SERIAL_A_C),A
0034 0018 3E 04 ld a,004h ;scrive in WR0: selecteaza WR4
0035 001A D3 06 out (SERIAL_A_C),A
0036 001C 3E 44 ld a,44h ;scrive 44h in WR4: CLKx16,1 bit de stop, fara paritate
0037 001E D3 06 out (SERIAL_A_C),A
0038 0020 3E 05 ld a,005h ;scrive in WR0: selecteaza WR5
0039 0022 D3 06 out (SERIAL_A_C),A
0040 0024 3E E8 ld a,0E8h ;semnal DTR activ, TX 8biti, BREAK inactiv, TX activ, RTS inactiv
0041 0026 D3 06 out (SERIAL_A_C),A
0042 0028 3E 01 ld a,01h ; scrive in WR0: selecteaza WR1
0043 002A D3 06 out (SERIAL_A_C),A
0044 002C 3E 18 ld a,00011000b ;activeaza INT pe receptia tuturor caracterelor
0045 002E ;depasirea bufferului este o conditie speciala tratata
0046 002E D3 06 out (SERIAL_A_C),A
0047 0030
0048 0030 SERIAL_A_EI:
0049 0030 ;activeaza receptia pe canalul A al circuitului
0050 0030 3E 03 ld a,003h ;scrive in WR0: selecteaza WR3
0051 0032 D3 06 out (SERIAL_A_C),A
0052 0034 3E C1 ld a,0C1h ;RX pe 8biti, RX activ, auto enable inactiv
0053 0036 D3 06 out (SERIAL_A_C),A
0054 0038 ;canalul A are bufferul RX activ
0055 0038 C9 ret
0056 0039
```

# Intreruperi mod1 (INT IM1) la uP Z80

## Aplicatii in comunicatii seriale

```
0057 0039      ; Configurarea circuitului CTC pentru semnalul CLK la RX si TX
0058 0039
0059 0039      INI_CTC:
0060 0039          ; initializare canal CH0
0061 0039          ; valoarea din A va fi scrisa in registrul de control al CTC
0062 0039 3E 07    ld A,00000111b ; intreruperi dezactivate, timer activ, prescaler=16,
0063 003B          ; porneste timerul dupa incarcarea constantei,
0064 003B          ; reset software activ;
0065 003B D3 00    out (CH0),A   ; scrie in reg de control al CTC
0066 003D 3E 02    ld A,2h       ; definirea constantei de timp
0067 003F D3 00    out (CH0),A   ; se incarca in canalul 0
0068 0041          ; T00 va da la iesire un ceas de
0069 0041          ; frecventa=CLK/2/16/(constanta_timp)/2
0070 0041          ; care va da de fapt baudrateul de 9600 bits/sec
0071 0041
0072 0041          ; dupa initializarea circuitului Z8440AB1 si CTC
0073 0041          ; se initializeaza procesorul pentru operare in modul 1 de intrerupere
0074 0041
0075 0041          INT_INI:
0076 0041 3E 00    ld a,0
0077 0043 ED 47    ld i,a       ; incarca in registrul I val 0
0078 0045 ED 56    im 1        ; seteaza modul de intrerupere
0079 0047 FB       ei          ; activeaza intreruperile
0080 0048
```

# Intreruperi mod1 (INT IM1) la uP Z80

## Aplicatii in comunicatii seriale

```
0081 0048      ; Rutina pentru controlul fluxului de date in comunicatie
0082 0048      ; aceasta rutina are rolul de a determina daca al doilea nod de comunicatie
0083 0048      ; este pregatit pentru receptia caracterelor
0084 0048      ; se va utiliza testul semnalului RTS (Ready To Send)
0085 0048
0086 0048      A_RTS_OFF:
0087 0048 3E 05      ld a,005h ;scrie in WR0: selecteaza WR5
0088 004A D3 06      out (SERIAL_A_C),A
0089 004C 3E E8      ld a,0E8h ;DTR activ, TX pe 8biti, BREAK inactiv, TX activ, RTS inactiv
0090 004E D3 06      out (SERIAL_A_C),A
0091 0050 C9      ret
0092 0051      A_RTS_ON:
0093 0051 3E 05      ld a,005h ;scrie in WR0: selecteaza WR5
0094 0053 D3 06      out (SERIAL_A_C),A
0095 0055 3E EA      ld a,0EAh ;DTR activ, TX 8biti, BREAK inactiv, TX activ, RTS activ
0096 0057 D3 06      out (SERIAL_A_C),A
0097 0059 C9      ret
0098 005A
0099 005A      ; In cazul in care bufferul de receptie este plin
0100 005A      ; se apeleaza rutina de dezactivare a receptiei
0101 005A
0102 005A      SERIAL_A_DI:
0103 005A      ;dezactiveaza bufferul RX al canalului A al circuitului
0104 005A 3E 03      ld a,003h ;scrie in WR0: selecteaza WR3
0105 005C D3 06      out (SERIAL_A_C),A
0106 005E 3E C0      ld a,0C0h ;RX pe 8biti, auto enable inactiv, RX inactiv
0107 0060 D3 06      out (SERIAL_A_C),A
0108 0062 C9      ret
0109 0063
```

# Intreruperi mod1 (INT IM1) la uP Z80

## Aplicatii in comunicatii seriale

```
0110 0063          ; implementarea rutinelor de tratare a intreruperilor
0111 0063          ; la receptia unui caracter se apeleaza rutina RX_CHA_AVAILABLE
0112 0063          ; LA APELUL RUTINEI DE TRATARE A INTRERUPERII SE IMPUNE SALVAREA
0113 0063          ; CONTEXTULUI CURENT AL RPOCESROULUI (REGISTRRII AF, HL, DE ,BC)
0114 0063          ; PENTRU A CONSERVA VALORILE CURENTE DUPA REVENIREA DIN RUTINA
0115 0063
0116 0063          RX_CHA_AVAILABLE:
0117 0063 F5          push AF ;salveaza contextul procesorului (reg. AF, BC, DE, HL)
0118 0064 C5          push BC
0119 0065 D5          push DE
0120 0066 E5          push HL
0121 0067 CD 48 00      call A_RTS_OFF
0122 006A DB 04          in A,(SERIAL_A_D) ;citeste caracterul din reg RX im reg A
0123 006C              ;testeaza caracterele receptionate:
0124 006C FE 0D          cp 0Dh ;testeaza daca ultimul caracter primit este CR
0125 006E CA 81 00      jp z,RX_CR
0126 0071 FE 08          cp 8h ;testeaza daca ultimul caracter primit este BS
0127 0073 CA 84 00      jp z,RX_BS
0128 0076 D3 04          out (SERIAL_A_D),A
0129 0078              ;AICI CARACTERELE PRIMITE POT FI PRELUCRATE
0130 0078 CD A1 00      call TX_EMP ;elibereaza bufferul de transmisie
0131 007B CD 94 00      call RX_EMP ;elibereaza bufferul de receptie
0132 007E C3 87 00      jp EO_CH_AV
0133 0081
0134 0081          RX_CR:
0135 0081              ;actiune realizata la receptia CR
0136 0081 C3 87 00      jp EO_CH_AV
0137 0084
0138 0084          RX_BS:
0139 0084              ;actiune realizata la receptia BS
0140 0084 C3 87 00      jp EO_CH_AV
0141 0087
0142 0087          EO_CH_AV:
0143 0087 FB          ei ; activeaza intreruperile mascabile
0144 0088 CD 51 00      call A_RTS_ON ;o noua receptie
0145 008B F1          pop AF ;restaureaza contextul procesorului
0146 008C C1          pop BC
0147 008D D1          pop DE
0148 008E E1          pop HL
0149 008F ED 4D          reti
0150 0091
```

# Intreruperi mod1 (INT IM1) la uP Z80

## Aplicatii in comunicatii seriale

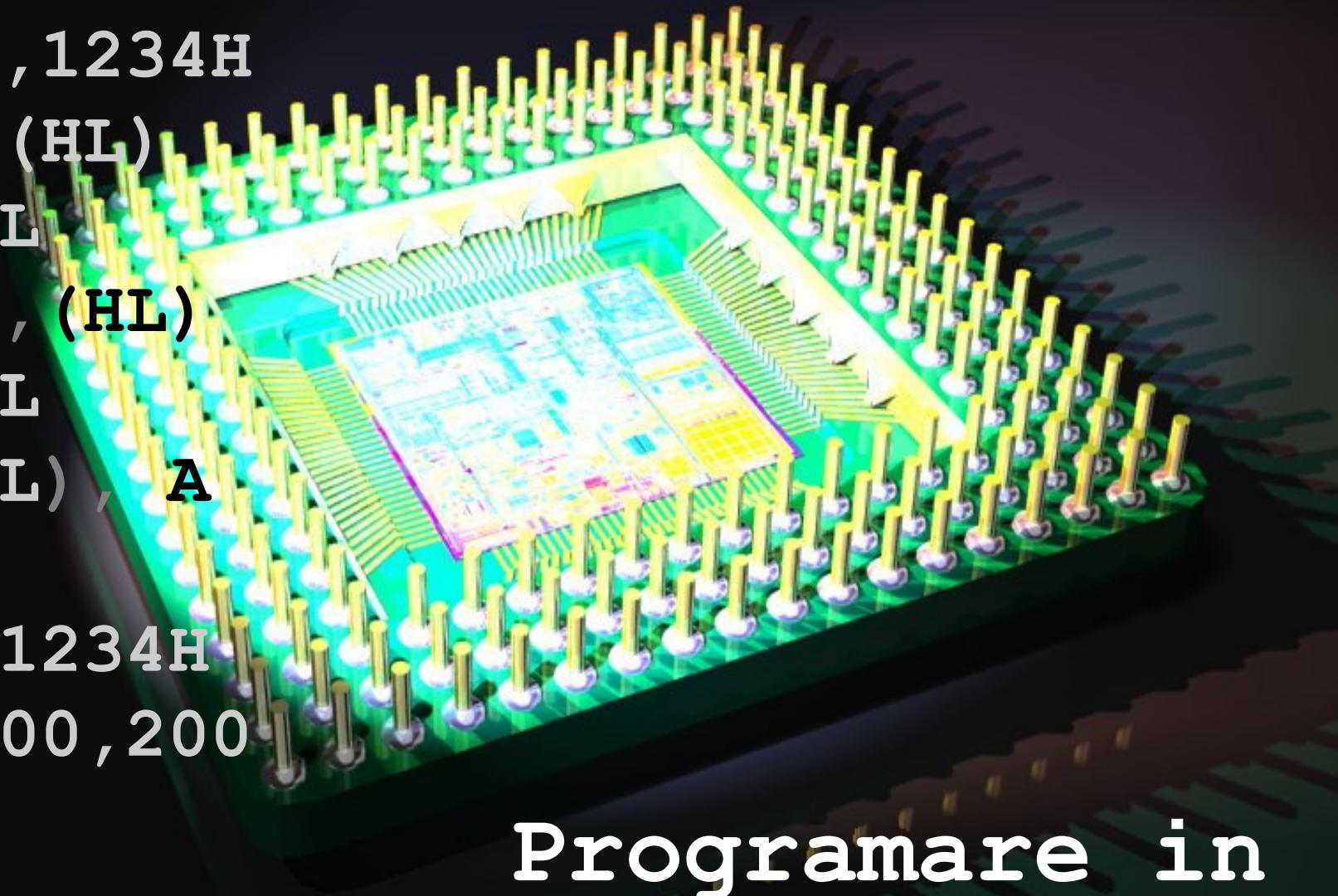
```
0150 0091
0151 0091          SPEC_RX_CONDITION:
0152 0091 C3 00 00      jp 0000h
0153 0094
0154 0094          ; rutina care elibereaza bufferul de receptie dupa citirea caracterelor
0155 0094
0156 0094          RX_EMP:
0157 0094          ;testeaza daca bufferul este gol
0158 0094 97          sub a           ;elibereaza reg a, scrie in WR0: selecteaza RR0
0159 0095 D3 06          out (SERIAL_A_C),A
0160 0097 DB 06          in A, (SERIAL_A_C)    ;citeste bufferul RRx
0161 0099 CB 47          bit 0,A
0162 009B C8          ret z           ;daca mai sunt caractere ne preluate din bufferul de receptie
0163 009C DB 04          in A, (SERIAL_A_D)    ;citeste caracterele ramase
0164 009E C3 94 00      jp RX_EMP
0165 00A1
0166 00A1          ; rutina de transmitere a unui caracter la al doilea nod
0167 00A1          ; transmiterea se realizeaza golind bufferul de transmisie
0168 00A1
0169 00A1          TX_EMP:
0170 00A1          ; testeaza daca bufferul de TX este gol
0171 00A1 97          sub a ;elibereaza reg A, scrie in WR0: selecteaza RR0
0172 00A2 3C          inc a ;selecteaza RR1
0173 00A3 D3 06          out (SERIAL_A_C),A
0174 00A5 DB 06          in A, (SERIAL_A_C) ;citeste din RRx
0175 00A7 CB 47          bit 0,A
0176 00A9 CA A1 00      jp z,TX_EMP
0177 00AC C9          ret
```

**010011010111010101101100011101000111  
01010110110101100101011100110110001  
10010000001110000011001010110111001  
11010001110010011101010010000001100  
00101110100011001010110111001110100  
0110100101100101001000000100001**

**sau altfel spus ...**

**Multumesc pentru atentie !**

```
.ORG 100H  
LD HL,1234H  
LD A, (HL)  
INC HL  
ADD A, (HL)  
INC HL  
LD (HL), A  
.END  
.ORG 1234H  
.DB 100,200
```



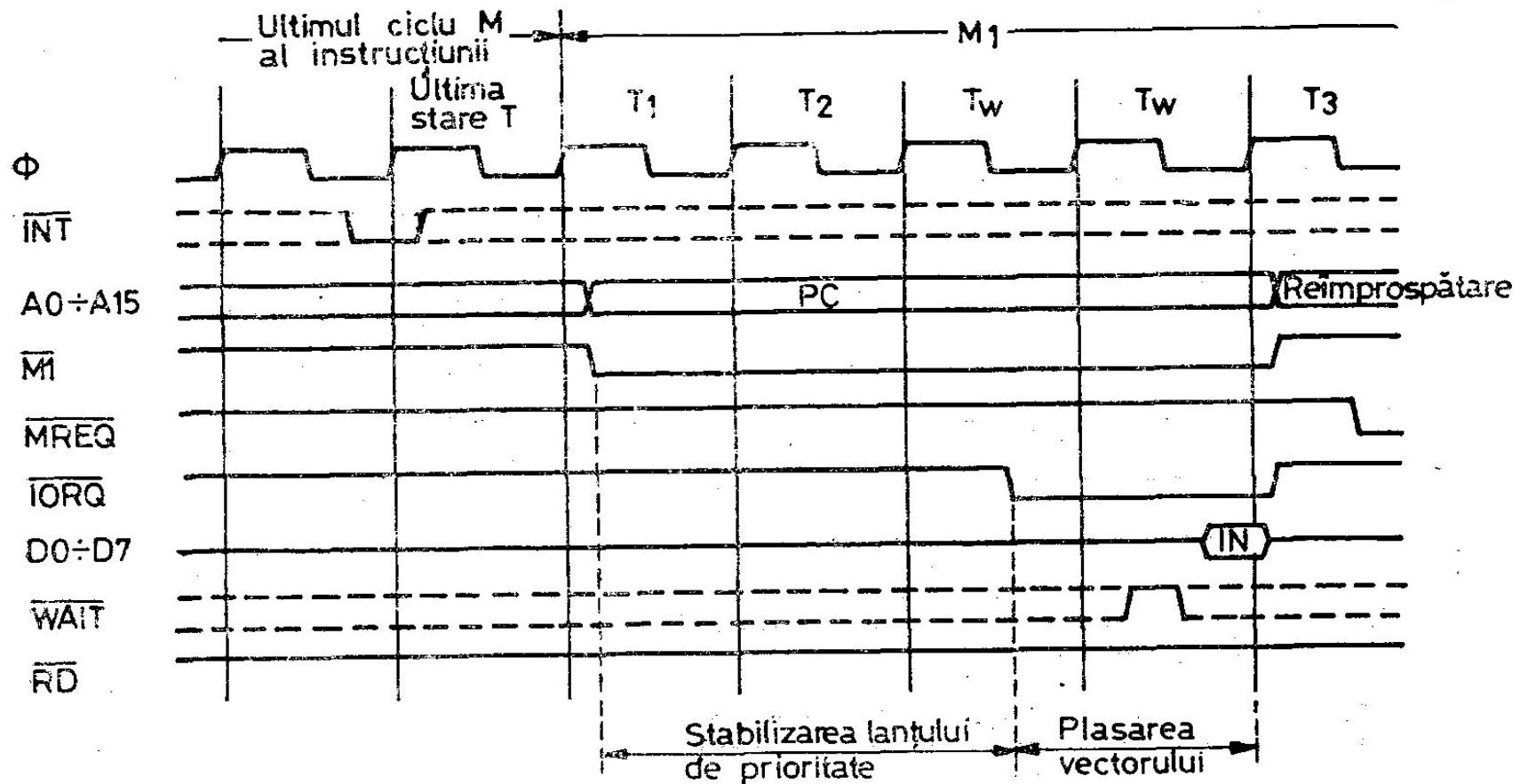
Programare in  
Limba de Asamblare

# Intreruperi mod2 (INT IM2) la uP Z80

- Întreruperile mascabile sunt preluate și tratate la sfârșitul execuției instrucțiunii curente numai dacă acestea au fost activate în prealabil.
- Microprocesorul Z80 are trei moduri de tratare a întreruperilor mascabile, moduri care se stabilesc cu ajutorul instrucțiunilor: **IM0**, **IM1**, **IM2**.
- Modul unu (**IM2**) poate fi utilizat de perifericele care aparțin familiei Z80.

# Intreruperi mod2 (INT IM2) la uP Z80

## Ciclul de cerere-achitare (IRQ-IACK) a intreruperii mascabile



# Intreruperi mod2 (INT IM2) la uP Z80

## Semnalul INT :

- este esantionat la **finalul** fiecarei **instructiuni**, pe **frontul pozitiv** al ultimei stari T.
- daca e **activ** va fi **acceptat** doar daca **bistabilul intern de validare**, IFF1, comandat prin soft este pe 1 si daca semnalul de cerere magistrala **BUSRQ** nu e pe 0.

Daca intreruperea se accepta se va declansa un **ciclu M1(fetch) special**, care va da in mod unic dispozitivul periferic care a generat intreruperea si caruia ii va cere sa plaseze pe magistrala de date un **vector de 8 biti**.

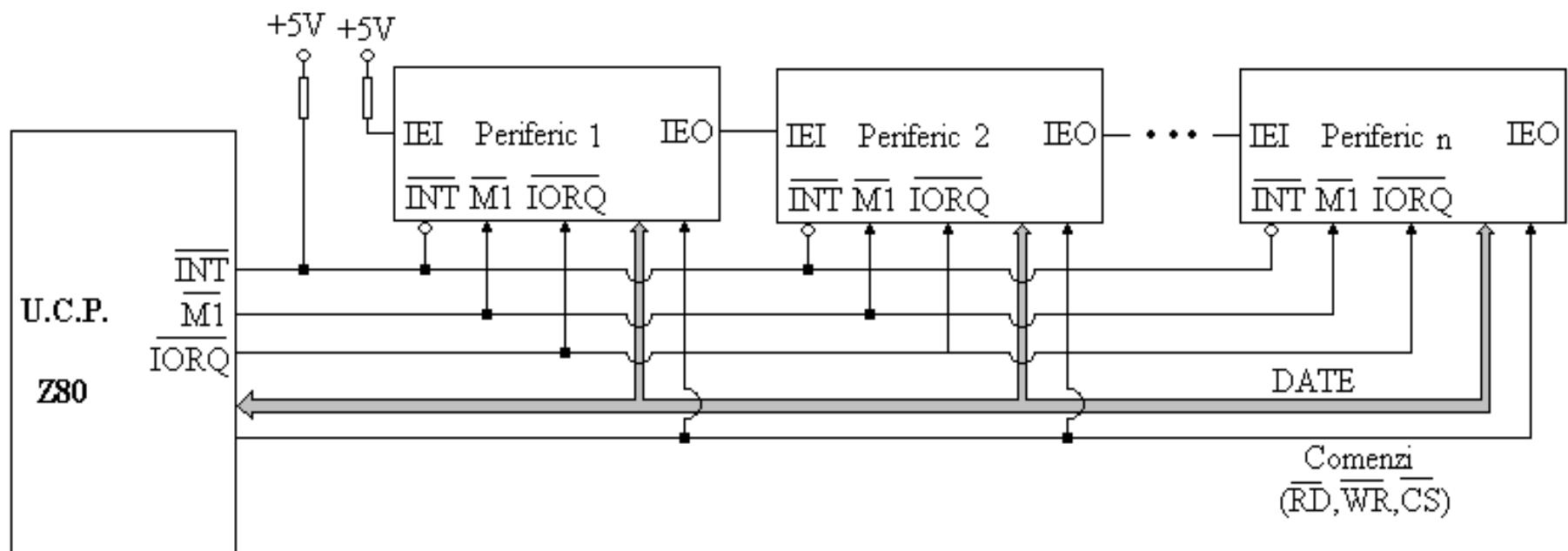
# Intreruperi mod2 (INT IM2) la uP Z80

## Specific mod intrerupere 2:

- este specific **perifericelor din familia uP Z80** și anume PIO-Z80; SIO-Z80 ; CTC-Z80.
- se poate lucra în acest mod numai după **execuția instrucțiunilor EI, IM2**.
- în cadrul acestui mod se pot **deservi până la 128 de periferice** a căror **prioritate** se stabilește prin **poziționarea lor** într-o **ordine** care corespunde priorităților **stabilite aprioric**.

# Intreruperi mod2 (INT IM2) la uP Z80

Modul de **conectare** a perifericelor din familia uP Z80 care utilizeaza modul 2 de intrerupere



# Intreruperi mod2 (INT IM2) la uP Z80

**Fiecare periferic poate genera o intrerupere dacă sunt îndeplinește urmatoarele condiții :**

- **a fost programat** să lucreze în modul de intrerupere corespunzător **modului 2** oferit de uP Z80,
- **intrarea IEI** (intrare de activare a intreruperilor) este la **1 logic**,
- **registrul** care conține **octetul inferior** al **adresei de memorie** unde se află adresa **rutinei de tratare a intreruperilor** a fost încărcat cu valorile corespunzătoare.

# Intreruperi mod2 (INT IM2) la uP Z80

**Context de executie** (cerere si achitare – IRQ si IACK) :

- #1. se citește **vectorul de întrerupere** corespunzător **perifericului** care a **generat întreruperea**,
- #2. se activeaza semnalele **M1** și **IOREQ**,
- #3. se **salveaza PC-ul** instructiunii imediat urmatoare **în memoria de tip stivă**,
- #4. se **extragă** din tabela cu adrese **adresa rutinei de tratare a intreruperii** asociată **perifericului**,
- #5. se **executa** rutina de tratare a **întreruperii (ISR)**,
- #6. se **revine** **în programul** aflat **în curs de execuție**

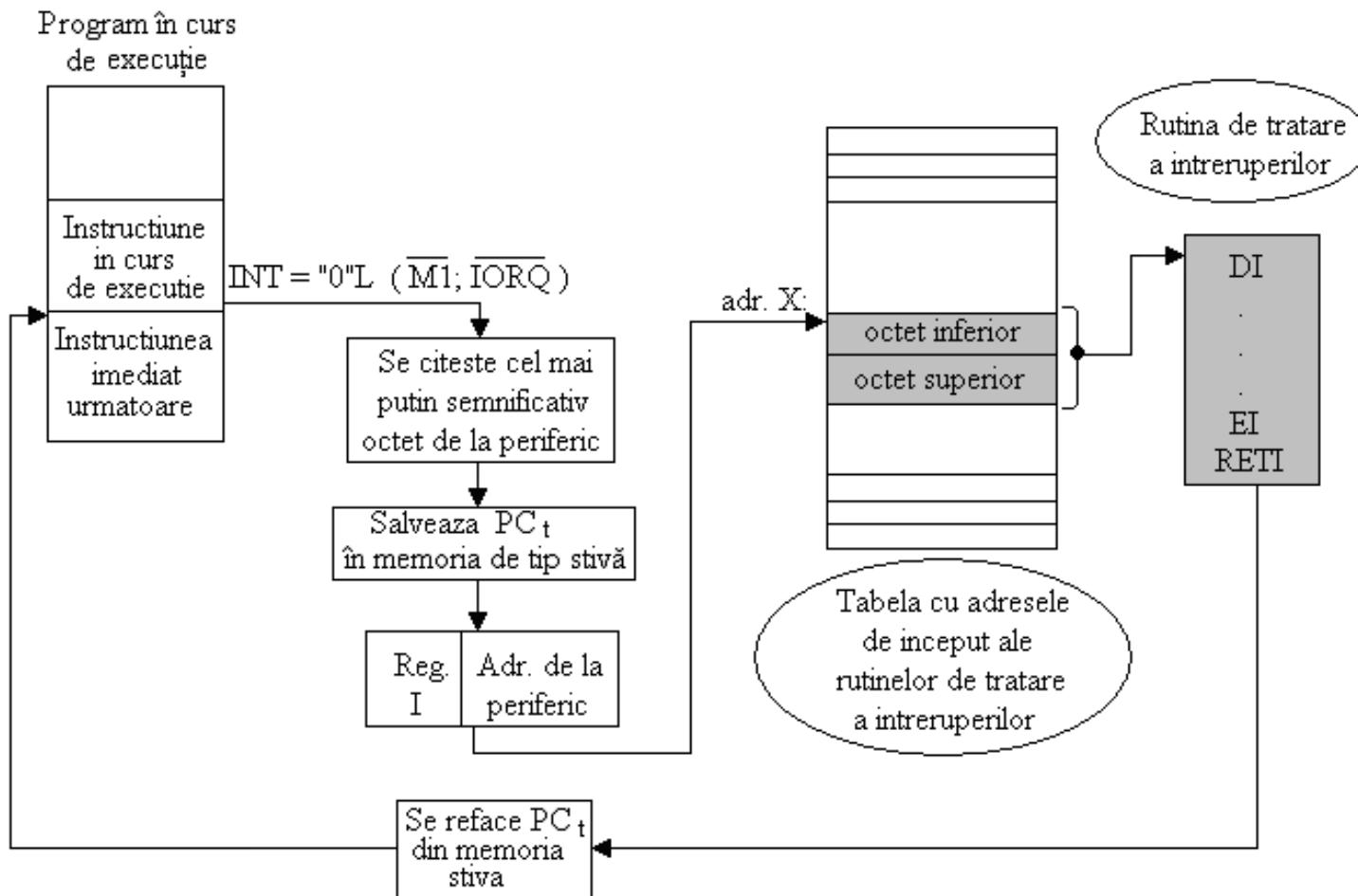
# Intreruperi mod2 (INT IM2) la uP Z80

**Context de executie** (cerere si achitare – IRQ si IACK) :

- **Fiecare periferic** din familia lui Z80 dispune de o **iesire de inhibare** a celorlalte periferice (IEO).
- **Mecanismul de inhibare** se stabilește **prioritatea** în cazul apariției mai **multor cereri de intrerupere**.
  - Când un **periferic** a generat o **întrerupere** acesta **genereaza un semnal 0 logic la iesirea IEO**, semnal care **inhibă toate perifericele** plasate fizic după acesta.

# Intreruperi mod2 (INT IM2) la uP Z80

## Modul de tratare a intreruperilor mascabile de mod 2



# Intreruperi mod2 (INT IM2) la uP Z80

## Observatii:

1. Prin utilizarea acestui **mod de intrerupere**, atat **tabela** cât și **rutinele de tratare a intreruperilor** pot fi plasate **oriunde în memorie**.
2. **Starea HALT** reprezintă starea în care intră **uP Z80** în urma execuției instrucțiunii HALT.
  1. pe durata instrucțiunii HALT se execută instrucțiuni NOP;
  2. scoaterea uP din HALT se poate face prin **resetarea** lui, sau printr-o **întrerupere nemascabilă sau mascabilă**;
  3. **intreruperea mascabilă** are efect numai dacă înainte de intrarea uP în stare HALT, **întreruperile mascabile** au fost **activate**.

# Intreruperi mod2 (INT IM2) la uP Z80

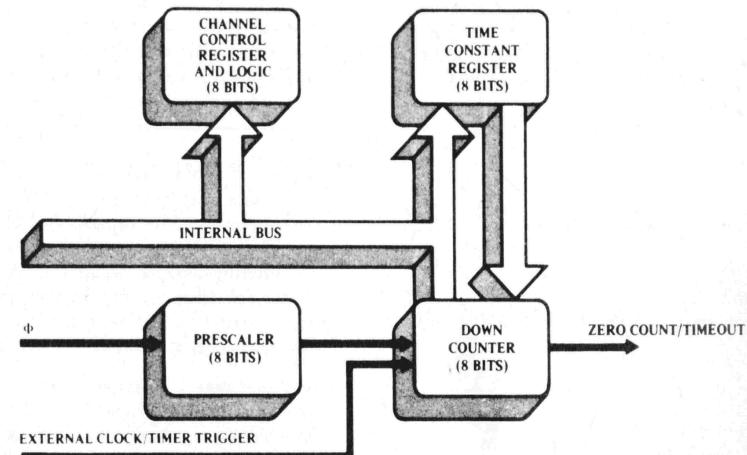
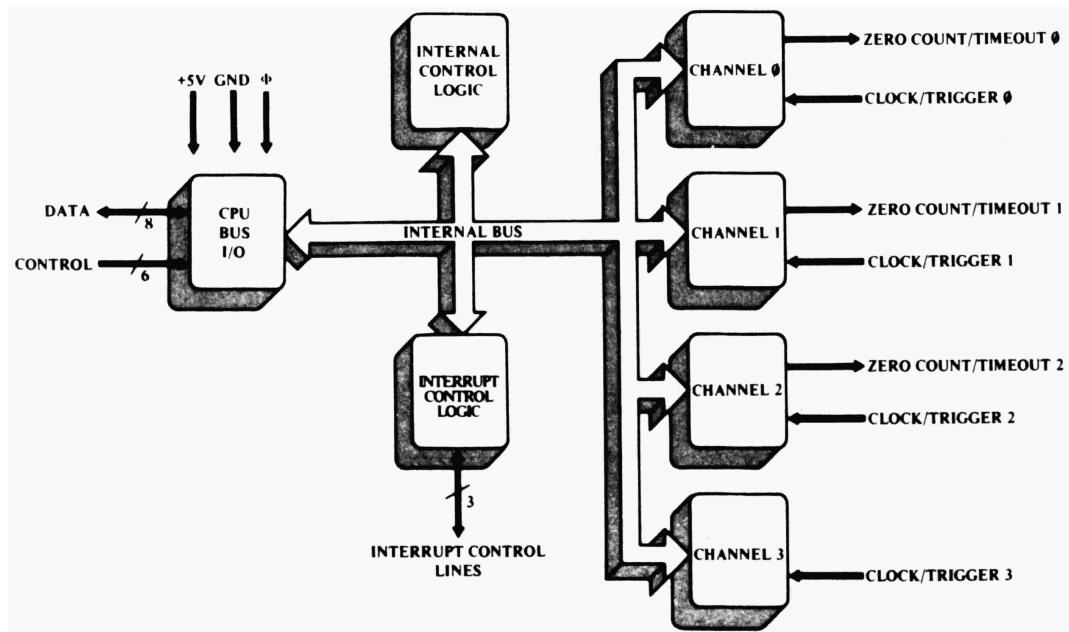
## Aplicatii – System state signal cu circuitul Z84C30 (ZiLOG)

Aplicatia propusa implementeaza un **mecanism de semnalizare a activitatii sistemului** cu uP z80 utilizand un **LED** care trebuie sa **pulseze** la fiecare **t secunde**. Rutina de aprindere a LED ului va fi de fapt rutina de tratare a intreruperii de timer generate de circuitul Z84C30 la fiecare t secunde.

**Circuitul Z84C30** contine **4 canale timer/counter** aplicatia necesitand doar lucrul cu 2 canale (**canalul 2 si canalul 3**).

# Aplicatii – System state signal cu circuitul Z84C30 (ZiLOG)

Diagrama bloc si schema interna a unui canal a circuitului Z84C30



# Aplicatii – System state signal cu circuitul Z84C30 (ZiLOG)

## Detalii de implementare:

1. Se initializeaza circuitul Z84C30,
2. Se implementeaza mecanismul de intreruperi, definire vectori de intrerupere,
3. Se implementeaza rutina de intrerupere pentru pulsatia LEDului.

# Aplicatii – System state signal cu circuitul Z84C30 (ZiLOG)

```
0001 0000      ; Aplicatie ce implementeaza un system state signal utilizand un LED
0002 0000      ; care va pulsa (heartbeat) la fiecare t secunde
0003 0000      ; aplicatia utilizeaza intreruperile generate de circuitul timer Z84C30
0004 0000      ; codul prezentat contine doar subruteinele care pot fi considerate in implementarea reala
0005 0000
0006 0000      ; initializarea porturilor de control pentru circuitul temporizator Z84C30
0007 0000      CH0 .equ 0h
0008 0000      CH1 .equ 1h
0009 0000      CH2 .equ 2h
0010 0000      CH3 .equ 3h
0011 0000
0012 0000      ; de fiecare data cand numaratorul canalului 3 ajunge la 0
0013 0000      ; genereaza o intrerupere si uP face un salt la adresa de memorie
0014 0000      ; unde se afla rutina de tratare a intreruperii, CT3_ZERO
0015 0000
0016 0000      ; tabela vectorului de intrerupere pt circuitul Z84C30
0017 0000      .org 16h
0018 0016 39 00      .defw CT3_ZERO
0019 0018
0020 0018      ; initializarea circuitului Z84C30
0021 0018      ; se configureaza canalele
0022 0018
0023 0018      ; nu se utilizeaza canalele 0 si 1 si se seteaza pe starea de hold
0024 0018      ; canalul 2 va fi utilizat ca divizor de frecventa pentru semnalul
0025 0018      ; de CLK a uP (5MHz), divizare cu un factor de 256*256.
0026 0018
0027 0018      ; iesirea TO2(output) a canalului 2 se conecteaza cu TRG3(trigger)
0028 0018      ; a canalului 3 care realizeaza o noua divizare cu AFh.
0029 0018      ; in acest mod numaratorul canalului 3 va ajunge la 0 cu o frecventa
0030 0018      ; de aproximativ 0.44Hz, deci intreruperea va fi generata la fiecare
0031 0018      ; 2.3 secunde.
nn32 nn18
```

# Aplicatii – System state signal cu circuitul Z84C30 (ZiLOG)

```
0033 0018      INI_Z84C30:  
0034 0018          ;initializeaza canalul CH 0 si 1 trimitand o valoarea de comanda  
0035 0018 3E 03      ld A,00000011b ; intreruperi dezactivate, timer oactiv,  
0036 001A          ; prescaler=16, start timer la incarcarea valorii constantei,  
0037 001A          ; software reset activ  
0038 001A D3 00      out (CH0),A ; CH0 este in starea hold  
0039 001C D3 01      out (CH1),A ; CH1 este in starea hold  
0040 001E  
0041 001E          ;initializarea canalului CH2  
0042 001E          ;CH2 divizeaza semnalul CLK al uP cu (256*256)  
0043 001E          ; oferind un semnal de ceas la iesirea T02.  
0044 001E          ; T02 este conectat la intrarea trigger(delansare) TRG3 a canalului 3.  
0045 001E 3E 27      ld A,00100111b ; int dezactivate, timer activ, prescaler=256, fara start extern,  
0046 0020          ; start dupa incarcarea valorii constantei, constanta urmeaza  
0047 0020          ; reset software -> informatia se va trimite ca un cuvant de comanda  
0048 0020 D3 02      out (CH2),A  
0049 0022 3E FF      ld A,0FFh       ; constanta are valoarea 256  
0050 0024 D3 02      out (CH2),A       ; se incarca valoarea pt canalul 2  
0051 0026          ; T02 genereaza un semnal cu f= CPU_CLK/(256*256)  
0052 0026  
0053 0026          ;initializarea canalului CH3  
0054 0026          ;intrarea TRG a CH3 primeste semnalul de la iesirea T02 a canalului 2  
0055 0026          ;CH3 divizeaza semnalul de ceas de la T02 cu AFh  
0056 0026          ;CH3 va genera intrerupere la ~2.3s uP executand ISR CT3_ZERO (aprinderea LED)  
0057 0026 3E C7      ld A,11000111b ; int activate, timer activ, prescaler=X, fara start extern,  
0058 0028          ; start dupa incarcarea valorii constantei, constanta urmeaza  
0059 0028          ; reset software -> informatia se va trimite ca un cuvant de comanda  
0060 0028 D3 03      out (CH3),A  
0061 002A 3E AF      ld A,0AFh        ; definirea constantei de divizare AFh  
0062 002C D3 03      out (CH3),A       ; se incarca pt canalul 3  
0063 002E 3E 10      ld A,10h         ; vectorul de int definit in bit 73, bit 21 nu conteaza, bit 0 = 0  
0064 0030 D3 00      out (CH0),A       ; incarca pt canalul 0  
0065 0032
```

# Aplicatii – System state signal cu circuitul Z84C30 (ZiLOG)

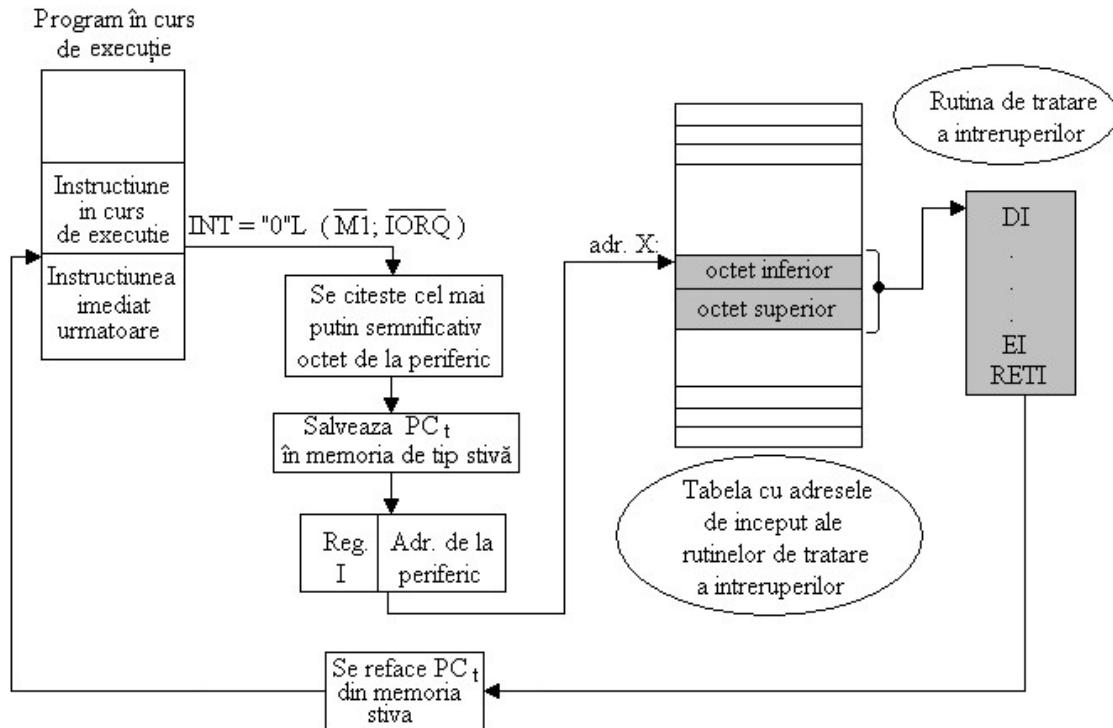
```
0065 0032
0066 0032      ; se initializeaza uP
0067 0032      INT_INI:
0068 0032 3E 00      ld A,0
0069 0034 ED 47      ld I,A ; incarca registrul I cu 0
0070 0036 ED 5E      im 2   ;setarea modului de intrerupere 2
0071 0038 FB        ei     ;activarea intreruperilor
0072 0039
0073 0039      ; definirea rutinei de tratare a intreruperilor
0074 0039      ; codul rutinei aprinde / stinge LEDul la expirarea numaratorului canalului 3
0075 0039      CT3_ZERO:
0076 0039          ;aprinde LED
0077 0039          ; salveaza contextul procesorului (starea registrilor AF, BC, DE, HL)
0078 0039 F5        push AF
0079 003A C5        push BC
0080 003B D5        push DE
0081 003C E5        push HL
0082 003D          ; se adreseaza perifericul care comanda LEDul
0083 003D          ; se scrie o valoare de comanda
0084 003D          ; se scrie pe portul de iesire conectat la perifericul
0085 003D          ; de comanda a LEDului
0086 003D          ; se reface contextul initial al procesorului
0087 003D F1        pop AF
0088 003E C1        pop BC
0089 003F D1        pop DE
0090 0040 E1        pop HL
0091 0041 FB        ei     ; reactiveaza intreruperile
0092 0042 ED 4D        reti   ; revenire din rutina de tratare a intreruperilor
```

**010011010111010101101100011101000111  
01010110110101100101011100110110001  
10010000001110000011001010110111001  
11010001110010011101010010000001100  
00101110100011001010110111001110100  
0110100101100101001000000100001**

**sau altfel spus ...**

**Multumesc pentru atentie !**

multor cereri de intrerupere. Când un periferic a generat o intrerupere acesta generează un semnal 0 logic la iesirea IEO, semnal care inhibă toate perifericele plasate fizic după acesta. Modul de tratare a intreruperii este ilustrat în figura următoare.



Prin utilizarea acestui mod de intrerupere atât tabela cât și rutinele de tratare a intreruperilor pot fi plasate oriunde în memorie.

## Starea HALT

Starea HALT reprezintă starea în care intră microprocesorul Z80 în urma execuției instrucțiunii HALT care are codul 76. Trebuie specificat faptul că pe durata instrucțiunii HALT se execută instrucțiuni NOP.

Scoaterea microprocesorului din această stare se poate face prin resetarea lui, sau printr-o intrerupere nemascabilă sau mascabilă. Întreruperea mascabilă are efect numai dacă înainte de intrarea microprocesorului în stare HALT, intreruperile mascabile au fost activate.

# **Programare in Limbaje de Asamblare**

## **DSP: TI TMS 32- & 64-bit**

## DSP Laborator 1

**L1 :** Descrierea platformei de dezvoltare DSProto32 cu TMS320C32. Familiarizarea cu mediul de lucru si de programare in limbaj de asamblare DSProto32.

- Setul de instructiuni. Tipuri de date, reprezentare interna, moduri de adresare.
- Exemplificare tipuri de instructiuni.
- Analiza executiei unui program si incarcarea registrilor.

# FAMILIA PROCESOARELOR DE SEMNAL TMS320

## 1.1. Introducere

Procesoarele de semnal, numite în literatura de specialitate DSP-uri (*DSP = Digital Signal Processor*), sunt sisteme de calcul programabile de tip „single-chip”, destinate prelucrării complexe a semnalelor digitale. Deși se numesc procesoare, ele înglobează într-un singur circuit integrat, principalele subsisteme componente ale unui sistem de calcul (unitate centrală, subsistem de memorie, subsistem de intrare/ieșire etc.), realizând funcții complexe de transfer și de prelucrare a datelor.

Pentru efectuarea unor prelucrări în timp real asupra datelor, procesoarele de semnal lucrează la frecvențe mari și dispun de un set complex de instrucțiuni, putând astfel executa zeci de milioane de operații în virgulă mobilă pe secundă (*MFLOPS = Million Floating-point Operations per Second*). Totodată, structura internă paralelă permite efectuarea mai multor operații simultan, ceea ce crește considerabil puterea de calcul a DSP-ului.

Evoluția în timp a procesoarelor de semnal a fost condiționată de dezvoltarea tehnologiei de fabricație a circuitelor integrate digitale. În general, tendințele dezvoltării tehnologice au în vedere următoarele aspecte:

- mărirea densității de integrare, prin creșterea numărului de tranzistoare pe unitatea de suprafață a chip-ului. Aceasta permite creșterea numărului unităților funcționale integrate și a complexității acestora;
- creșterea frecvenței interne de lucru a procesoarelor, ce se poate face explicit prin îmbunătățirea calității tranzistoarelor și, implicit, prin micșorarea dimensiunii tranzistoarelor și a distanțelor dintre ele;
- realizarea mai multor unități paralele de prelucrare și de mărire a dimensiunii cuvintelor prelucrate și memorate;
- micșorarea consumului energetic și a puterii disipate, ce tind să crească o dată cu creșterea frecvenței de lucru a procesoarelor;
- păstrarea compatibilității software cu modelele anterioare din aceeași familie de procesoare.

Datorită performanțelor în timp real și a capacitatii de prelucrare mare, procesoarele de semnal sunt utilizate într-o gamă largă de aplicații, enumerate succint în tabelul 1.1.

Tab. 1.1. Domenii de utilizare a procesoarelor de semnal

Prelucrare de semnale	Grafică/Imagini	Instrumentație
Filtrare digitală Convoluție Corelație Transformări Hilbert Transformări Fourier rapide Filtrare adaptivă Generare de semnale Fereștre de timp	Transformări 3D/ Renderizare Recunoaștere de forme Transmisie/Compresie imagini Recunoașterea formelor Îmbunătățire imagini  Scanare cod de bare Stații de lucru grafice	Analiză Spectrală Generare de funcții Potrivire de şabloane  Simulații seismice Filtrare digitală
<b>Recunoaștere voce</b>	<b>Comandă</b>	<b>Domeniul militar</b>
Voice mail Vocodere	Controlul discului  Comandă servo	Securitatea comunicațiilor  Procesare semnale radar
Recunoaștere vocală Verificare amprentă vocală Îmbunătățirea vorbirii Sinteză vocală Text-to-speech Rețele neurale	Comanda roboților  Imprimante laser  Comanda motoarelor  Filtre Kalman	Procesare sonar  Procesare imagini  Instrumente de navigație Ghidare rachete Modemuri în radio-frecvență
<b>Telecomunicații</b>		<b>Locomotie</b>
Suprimarea ecului Transcodere ADPCM Multiplexarea canalelor Modemuri  Codare/Decodare DMTF Criptare date Comunicații în spectru larg	FAX  Telefonie celulară  Telefoane inteligente  Interpolare  Video conferințe	Securitatea comunicațiilor  Analiza vibrațiilor  Sisteme anti-coliziune Control adaptiv al rulării GPS  Comenzi vocale Radio digital
<b>Bunuri de larg consum</b>	<b>Industria</b>	<b>Domeniul medical</b>

Detectoare de radar	Robotică	Proteze auditive
Instrumente de putere	Comandă numerică	Monitorizare pacienți
Audio/TV digitale	Securitatea accesului	Echipamente cu ultrasunete
Sintetizatoare muzicale	Monitorizare surse de putere	Echipamente de diagnostic
Jucării inteligente	Inspectie vizuală	Proteze
Roboți telefonici	CAM	

Pentru obținerea unei valori optime a raportului *performanță/cost*, structura internă a unui DSP trebuie adaptată specificului aplicației în care acesta se utilizează. În plus, performanțele obținute cu un anumit procesor de semnal depend esențial de componentele software (programe utilitare și de aplicații) oferite de fabricant.

De aceea, firmele producătoare au în vedere două aspecte majore:

- asigurarea unei game largi de produse și de soluții complete cu DSP-uri, adaptându-se permanent la cerințele pieței;
- crearea suportului software pentru dezvoltarea aplicațiilor cu DSP-uri.

De exemplu, firma *Texas Instruments* (TI) a devenit liderul pieței de procesoare de semnal digital, începând cu anul 1982, când a introdus pe piață procesorul *TMS32010*. În timp, firma a dezvoltat o gamă foarte largă de produse, având în producție mai mult de 100 de tipuri de DSP-uri, utilizate în domenii diverse, cum ar fi: comunicații, calculatoare, produse de larg consum, conducerea proceselor industriale, instrumentație, aplicații militare etc. Pentru utilizarea eficientă a procesoarelor, firma TI oferă un pachet software amplu, sub formă de documentații, aplicații diverse și biblioteci de programe, colaborând cu peste 250 de firme producătoare de software și hardware.

Procesoarele de semnal digital din gama *TMS320C3x* produse de firma *Texas Instruments* sunt realizate în tehnologie CMOS și lucrează cu numere în virgulă mobilă pe 32 de biți. Datorită unei magistrale interne extinse și a unui set puternic de instrucțiuni, DSP-ul poate executa până la 60 MFLOPS. Procesorul dispune de un înalt grad de paralelism în execuție, ce permite efectuarea simultană de până la 11 operații.

Procesoarele de semnal din familia *TMS320* pot fi împărțite în două mari clase: procesoare de virgulă fixă (cum sunt cele din generațiile 'C20x, 'C24x, 'C5x, 'C54x și 'C62x) și, respectiv, procesoare de virgulă mobilă (corespunzătoare generațiilor 'C3x, 'C4x, 'C67x).

### 1.3. Generația procesoarelor *TMS320C3x*

Procesoarele *TMS320C3x* reprezintă prima generație de procesoare în virgulă mobilă pe 32 de biți, având performanțe de 33-60 MFLOPS și de 16.67-30 MIPS. Arhitectura C3x este proiectată pentru obținerea unei platforme eficiente de tip compilator. Astfel, datorită compilatorului C optimizat și setului de instrucțiuni paralele, procesoarele din această generație sunt ușor de utilizat și permit realizarea unor produse de înaltă calitate într-un timp scurt, fără un efort deosebit.

Unitatea centrală (*UC*) are o unitate aritmetică independentă de multiplicare în virgulă fixă și mobilă, precum și o unitate aritmetico-logică (*UAL*). Procesorul oferă o interfață de

memorie flexibilă, ce permite accesul pe 8, 16 și 32 de biți și poate adresa un spațiu de memorie de 16 Mcuvinte de 32 de biți.

Controlerul de acces direct la memorie (DMA = Direct Memory Access) posedă o magistrală proprie de date și funcționează în paralel cu UC. Controlerul DMA poate accesa atât memoria internă a DSP-ului sau registrele periferice asignate la adrese de memorie, cât și memoria externă.

Procesoarele 'C3x, prin intermediul a 7 magistrale interne, pot transfera simultan patru cuvinte în fiecare ciclu mașină: un cuvânt de cod, doi operanzi pentru unitatea centrală și un transfer DMA de date.

Cel mai reprezentativ DSP al acestei generații este *TMS320C32*, a cărui schemă bloc este ilustrată în figura 1.2.

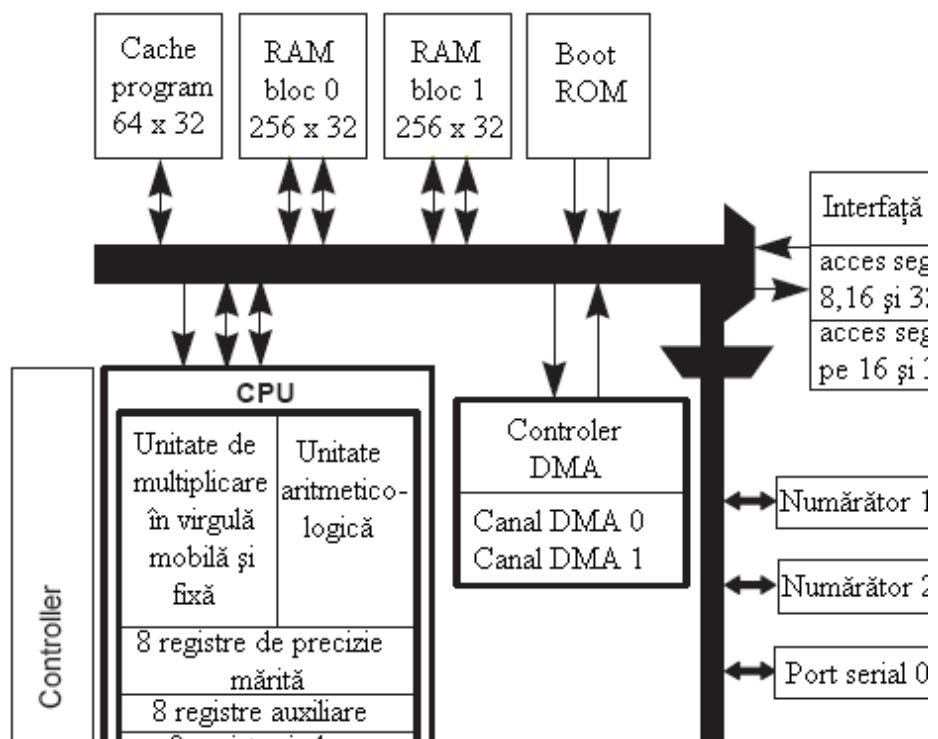


Fig. 1.2. Schema bloc a procesorului de semnal *TMS320C32*

Unul dintre avantajele acestui DSP este prețul, fiind cel mai ieftin procesor în virgulă mobilă pe 32 de biți. Codul mașină este compatibil cu cel al procesoarelor 'C30 și 'C31.

De asemenea, 'C32 poate rula în regim de consum redus, în două moduri. Primul mod reduce frecvența ceasului intern, permitând execuția în continuare a instrucțiunilor, iar al doilea mod suspendă execuția instrucțiunilor și pune procesorul în stare de așteptare.

Pe lângă cele menționate, procesorul *TMS320C32* se caracterizează prin:

- ciclul instrucțione de 33 ns, 40 ns și 50 ns;
- 2 canale DMA cu priorități configurabile;
- moduri de funcționare cu consum redus;
- memorie cache pentru program de 64 cuvinte;
- 2 numărătoare pe 32 de biți;
- port serial;
- boot ROM;
- 2 blocuri de memorie RAM cu dublu acces, de câte 256 cuvinte x 32 de biți.

Acest procesor poate fi utilizat într-o gamă largă de domenii, cum ar fi: industria automobilelor, procesare audio, control industrial, comunicații de date, echipamente de birou (copiatoare, imprimante laser, echipamente multifuncționale) etc.

## STRUCTURA HARDWARE A PROCESORULUI DE SEMNAL TMS320C32

### Structura internă

Familia de procesoare în virgulă mobilă pe 32 de biți *TMS320C3x* conține trei modele: 'C30, 'C31 și 'C32. Acestea pot efectua în paralel operații aritmetico-logice și de înmulțire cu numere întregi sau în virgulă mobilă, într-un singur ciclu mașină, atingând viteze de până la 30 MIPS și 60 MFLOPS.

Din punct de vedere hardware, performanțele procesoarelor *TMS320C3x* sunt date de următoarele elemente caracteristice, ce permit implementarea mai ușoară a limbajelor de programare de nivel înalt:

- arhitectura bazată pe registre;
- memorii interne cu acces dual și memorie cache de program;
- spațiu mare de adresare și moduri de adresare puternice;
- interfață pentru conectarea în sisteme multiprocesor;
- sistem de întreruperi multiple;
- porturi seriale și numărătoare/temporizatoare;
- acces direct la memorie, prin controller DMA intern.

Procesorul de semnal *TMS320C32* conține mai multe unități, ce vor fi detaliate în subcapitolele următoare.

Schema bloc detaliată a procesorului de semnal *TMS320C32* este ilustrată în figura 2.1.

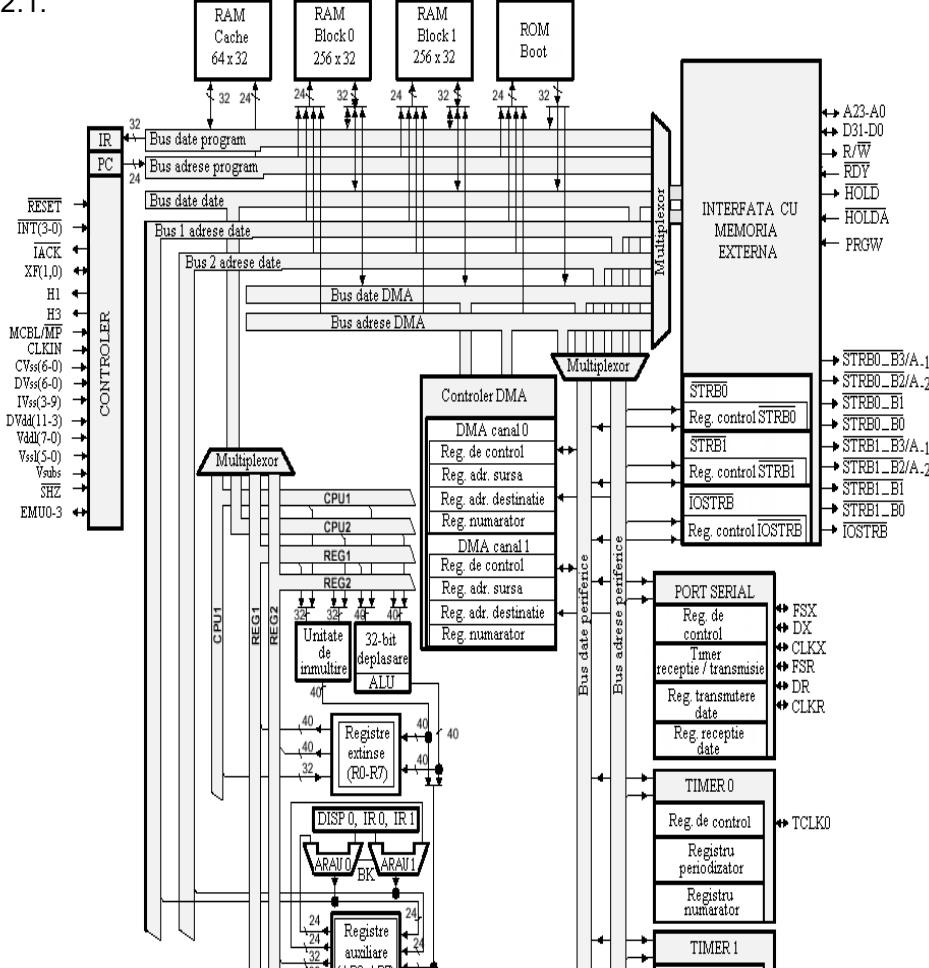


Fig. 2.1. Schema bloc detaliată a procesorului TMS320C32

Din figură se observă că procesorul conține următoarele elemente: o unitate centrală de prelucrare (UCP), cu 8 registre generale (R0-R7) și 8 registre auxiliare (AR0-AR7), o memorie cache pentru program cu capacitatea de 64 cuvinte de 32 de biți, două blocuri de memorie internă RAM cu dublu acces, de câte 256 cuvinte de 32 de biți, două numărătoare/temporizatoare, utilizate la măsurarea intervalelor de timp și la numărarea trenurilor de impulsuri, un port serial, precum și un controler DMA intern cu 2 canale, ce suportă operații concurente de intrare / ieșire.

Procesorul are mai multe magistrale interne, pe care se pot transfera simultan până la patru cuvinte în fiecare ciclu mașină: un cuvânt de cod, doi operanzi pentru unitatea centrală de la memoriile cu acces dual, precum și un transfer DMA de date.

O prezentare în detaliu a procesorului TMS320C32, în componentele sale hardware și software, depășește spațiul acestei cărți. Totuși, dezvoltarea aplicațiilor necesită o cunoaștere aprofundată a componentei hardware și presupune consultarea permanentă a documentației complete a procesorului, pusă la dispoziția utilizatorilor de către fabricant [29].

## Unitatea centrală de prelucrare

Structura internă a unității centrale de prelucrare (*UCP sau CPU = Central Processing Unit*) este dată în figura 2.2 și conține următoarele elemente:

- unități de înmulțire numere întregi și numere reale în virgulă mobilă;
- unitate aritmetico-logică (*ALU = Arithmetic Logic Unit*);
- unitate de deplasare la stânga/dreapta pe 32 biți;
- magistrale interne (CPU1/CPU2 și REG1/REG2);
- unități aritmetice auxiliare de adrese (*ARAU = Auxiliary register arithmetic unit*);
- fișierul cu registre al CPU.

Cele patru magistrale de date interne ale UCP (CPU1, CPU2, REG1 și REG2) sunt conectate la magistrala de date a procesorului, prin intermediul unui multiplexor. Ele pot transfera doi operanzi de la memorii și doi operanzi de la registrele UCP, permitând într-un singur ciclu, operații paralele de înmulțire și adunare/scădere.

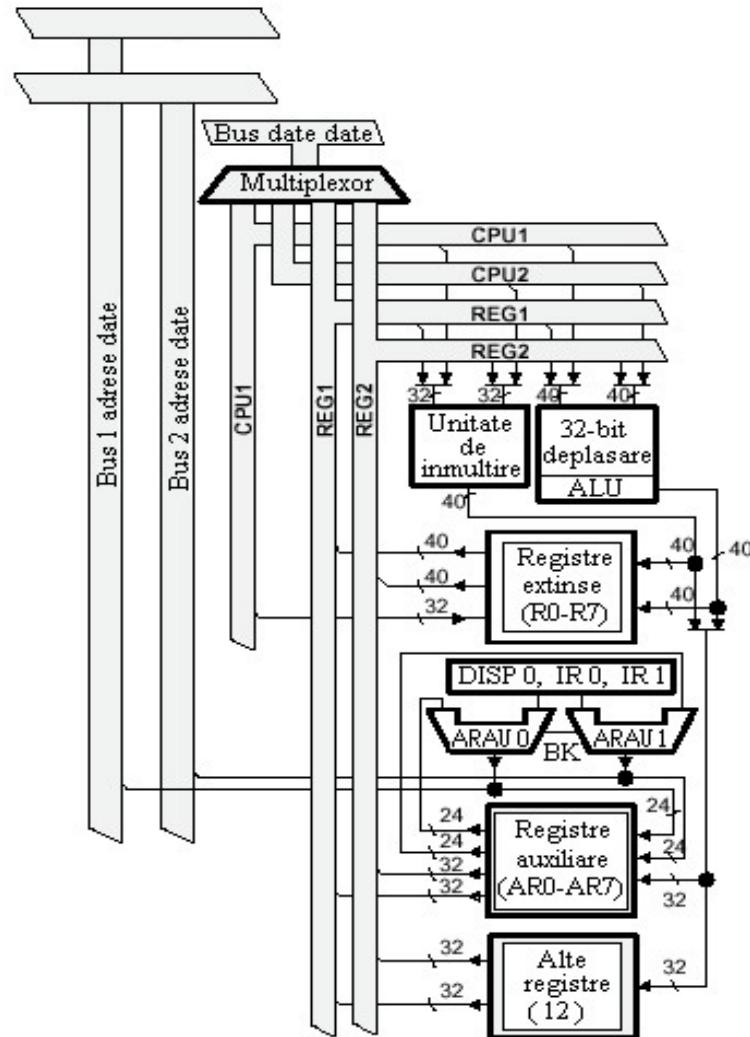


Fig. 2.2. Structura unității centrale de prelucrare

### **Unitatea de înmulțire**

Unitatea de înmulțire execută într-un singur ciclu mașină înmulțiri cu numere întregi reprezentate pe 24 de biți și cu numere reale reprezentate pe 32 de biți. Implementarea paralelă permite realizarea unei înmulțiri și a unei operații ALU într-un singur ciclu instrucție, la o perioadă a ceasului de 20 ns. Când unitatea de înmulțire realizează înmulțiri în virgulă mobilă, intrările sunt numere reale pe 32 de biți, iar rezultatul se reprezintă pe 40 de biți. Pentru înmulțiri în virgulă fixă, intrările sunt numere întregi pe 24 de biți, iar rezultatul reprezintă număr întreg pe 32 de biți.

### **Unitatea aritmetico-logică**

Unitatea aritmetico-logică (ALU) efectuează, într-un singur ciclu mașină, operații cu numere întregi reprezentate pe 32 de biți, operații logice cu numere binare reprezentate pe 32 de biți și cu numere reale reprezentate pe 40 de biți. De asemenea, ALU realizează conversia numerelor întregi în numere reale și invers. Rezultatele operațiilor sunt întotdeauna păstrate pe 32 de biți pentru numere întregi și pe 40 de biți pentru numere reale. Tot într-un singur ciclu mașină se pot deplasa spre stânga sau spre dreapta biții unui registru pe 32 de biți, cu un număr de poziții predefinit.

### Unități aritmetice auxiliare pentru adrese

Unitatea centrală de prelucrare dispune de două unități aritmetice auxiliare (ARAU0 și ARAU1), ce pot genera două adrese diferite într-un singur ciclu. Unitățile ARAU lucrează în paralel cu unitatea de înmulțire și cu unitatea aritmetico-logică. Ele suportă adresări cu deplasamente, cu registrele index (IR0 și IR1), precum și adresarea circulară și adresarea în ordinea inversă a bițiilor.

### Registrele unității centrale de prelucrare

Procesorul TMS320C32 dispune de 28 de registre, numite registre primare, prezentate succint în continuare. Ele sunt grupate într-un fișier de registre multiport, care este strâns cuplat cu CPU. Toate registrele primare pot fi utilizate de unitatea de înmulțire și de ALU și pot fi folosite ca registre generale de lucru.

De asemenea, registrele primare au și funcții speciale. De exemplu, cele opt registre de precizie mărită ( $R_0, \dots, R_7$ ) se pot utiliza la păstrarea rezultatului în virgulă mobilă cu precizie mărită. Registrele auxiliare ( $AR_0, \dots, AR_7$ ) suportă o varietate de moduri de adresare indirectă și pot fi folosite ca registre generale pentru operații cu numere întregi pe 32 de biți sau ca registre logice. Restul regisrelor sunt utilizate pentru funcții de sistem: adresare, managementul stivei, memorarea stărilor procesorului, întreruperi, adrese și contor pentru bucle.

**1. Registrele de precizie mărită ( $R_7 \div R_0$ )** sunt în număr de 8 și se utilizează la operații cu numere întregi pe 32 de biți sau cu numere reale pe 40 de biți.

Când se lucrează cu operanzi în virgulă mobilă, registrele de precizie mărită folosesc toți cei 40 de biți ( $bit_{39} \div bit_0$ ), după cum se poate observa în figura 2.3.

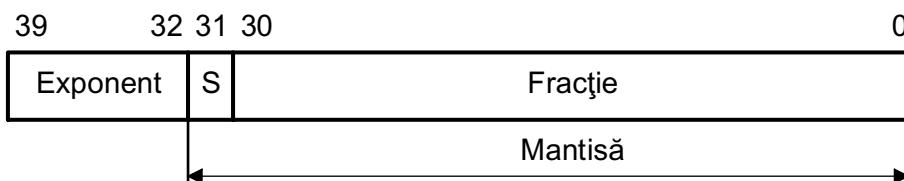


Fig. 2.3. Memorarea numerelor în virgulă mobilă, în registrele  $R_7 \div R_0$

În acest caz, registrele de precizie mărită au două regiuni separate și distincte:

- cei mai semnificativi 8 biți ( $bit_{39} \div bit_{32}$ ) sunt dedicați memorării exponentului (e) corespunzător numărului în virgulă mobilă;
- restul bițiilor ( $bit_{31} \div bit_0$ ) sunt utilizati pentru memorarea mantisei numărului în virgulă mobilă, unde  $bit_{31}$  este bit de semn (S), iar  $bit_{30} \div bit_0$  reprezintă fracția (f).

Dacă operanții sunt numere întregi, cu sau fără semn, atunci se folosesc doar 32 de biți ( $bit_{31} \div bit_0$ ), restul bițiilor rămânând nemodificați (figura 2.4). Acest lucru este valabil pentru toate operațiile de deplasare.

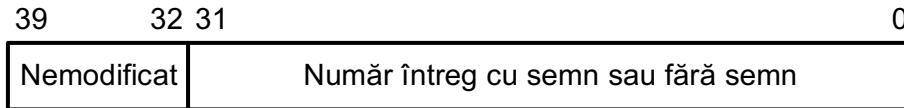


Fig. 2.4. Memorarea numerelor în virgulă fixă, în registrele R7 ÷ R0

**2. Registrele auxiliare (AR7 ÷ AR0)** pe 32 de biți sunt în număr de 8 și pot fi accesate de CPU și modificate de cele două unități ARAU. Funcția principală a regastrelor auxiliare este de a genera adrese pe 24 de biți. De asemenea, ele pot fi folosite în adresarea indirectă, ca registre contor pentru ciclurile repetitive sau ca registre generale pe 32 de biți, putând fi modificate de unitatea de înmulțire sau de ALU.

**3. Registrul indicator pagină de date (DP)** pe 32 de biți este utilizat la formarea adresei fizice de memorie. Registrul DP (*DP = Data-page register*) este încărcat cu instrucțiunea LDT. Cei mai puțin semnificativi 8 biți (*LSB = Least significant bit*) ai indicatorului de pagină sunt folosiți de modul de adresare direct. Ei indică pagina fizică de memorie utilizată la momentul curent. Paginile de memorie au lungimea de 64 Kcuvinte. Deoarece adresa este generată pe 24 de biți, rezultă că procesorul TMS320C32 poate accesa în total 256 pagini de memorie.

**4. Registrele de index (IR0, IR1)** pe 32 de biți, conțin valori index, ce pot fi utilizate de ARAU pentru calculul unei adrese de memorie indexate.

**5. Registrul dimensiune bloc (BK)** pe 32 de biți este folosit de ARAU în cazul adresării circulare. Registrul BK (*BK = Block-size register*) specifică dimensiunea blocului de date utilizat ca registru circular.

**6. Registrul indicator de stivă (SP)** pe 32 de biți specifică adresa vârfului stivei. Stiva funcționează ca o memorie FIFO, iar registrul SP (*SP = Stack pointer register*) indică întotdeauna adresa ultimului element introdus în stivă. Orice operație de adăugare în stivă este precedată de o incrementare a registrului SP, iar orice operație de extragere din stivă este urmată de o decrementare a SP. Doar cei mai puțin semnificativi 24 de biți sunt folosiți ca adresă de memorie pentru vârful stivei.

Registrul SP este utilizat de fiecare dată când se dorește salvarea contextului programului aflat în execuție, la tratarea întreruperilor, în timpul executării pas cu pas a programului, la apeluri de proceduri, la întoarceri din proceduri, precum și de instrucțiunile PUSH și POP.

**7. Registrul de stare (ST)** pe 32 de biți conține informații globale despre starea UCP. În funcție de rezultatul operațiilor (0, negativ etc.), unele instrucțiuni (cum ar fi cele de încărcare și de salvare a regastrelor, cele aritmetice și logice etc.) modifică valoarea indicatorilor de condiție ai registrului ST (*ST = Status register*).

De asemenea, este permisă încărcarea acestui regisztr cu valoarea unui operand sursă, indiferent care ar fi starea bițiilor din ST. În urma unei astfel de operații, conținutul regisztrului

ST va fi identic cu cel al operandului sursă. Acest lucru permite salvarea și restaurarea registrului ST.

La inițializarea sistemului, registrul ST are valoarea  $00000000H$ .

**8. Registrul de validare a întreruperilor CPU/DMA (IE)** pe 32 de biți permite validarea sau invalidarea întreruperilor CPU sau DMA. Biții de validare a întreruperilor CPU se află pe pozițiile  $bit_{11} \div bit_0$ , iar cei pentru validarea întreruperilor DMA se găsesc pe pozițiile  $bit_{31} \div bit_{16}$ .

Prin setarea bițiilor registrului IE (*IE = Interrupt-enable register*), întreruperile respective sunt validate, iar prin resetarea lor (poziționare pe 0), se indică faptul că întreruperea corespunzătoare este invalidată.

**9. Registrul indicatorilor de întreruperi (IF)** pe 32 de biți semnalează care întreruperi au fost activate de către unitățile slave. Dacă un bit al registrului IF (*IF = Interrupt flag register*) are valoarea 1, atunci acesta indică apariția unei întreruperi generate de perifericul corespunzător bitului setat. De asemenea, biți pot fi setați prin program, pentru a provoca o întrerupere. Dacă un bit al registrului IF este resetat (valoare 0), atunci întreruperea respectivă se anulează.

La inițializarea procesorului, conținutul bițiilor registrului IF este  $00000000H$ .

Cei mai semnificativi 16 biți ai registrului IF ( $bit_{31} \div bit_{16}$ ) reprezintă indicatorul tabelei vectorilor de întreruperi și de execuție pas cu pas (*ITTP = Interrupt-trap table pointer*). Vectorul *reset* al procesorului TMS320C32, ca și în cazul celorlalte procesoare din familia lui, se află la adresa  $000000H$ . Totuși, tabela de vectori pentru întreruperi și execuție pas cu pas a unui program se poate situa și la o altă adresă de memorie, aceasta fiind stabilită de utilizator prin ITTP.

**10. Registrul indicatorilor de intrare/ieșire (IOF)** pe 32 de biți controlează funcțiile pinilor externi: *XF0* și *XF1*. Acești pini pot fi configurați pentru operații de intrare și ieșire, putând fi scriși și citiți. La inițializarea procesorului, biții corespunzători din registrul IOF (*I/O flag register*) devin 0.

**11. Registrul contor pentru bucle (RC)** pe 32 de biți este utilizat pentru execuția repetitivă a unui bloc de program. Registrul RC (*RC = Repeat-counter register*) specifică de câte ori se repetă un bloc de instrucțiuni într-o buclă repetitivă.

Dacă *RC* conține valoarea *N*, bucla va fi executată de  $(N+1)$  ori.

**12. Registrele de adrese pentru structuri repetitive (RS, RE)** sunt pe 32 de biți. RS (*RS = Repeat start-address register*) și RE (*RE = Repeat end-address register*) conțin adresa de început, respectiv adresa de sfârșit ale blocului de memorie program care se execută în buclă. Dacă *RE* < *RS*, atunci programul execută o singură dată codul blocului de instrucțiuni dintre adresele indicate de *RS* și *RE*.

Procesorul mai are două registre, care nu aparțin unității centrale de prelucrare., ci unității de control. Ele sunt utilizate pentru citirea și decodificarea instrucțiunilor.

**13. Registrul numărător de program (PC)** pe 32 de biți conține adresa de memorie de unde se va citi opcodul următoarei instrucțiuni, în timpul ciclului de citire a instrucțiunii (fetch). Adresa este furnizată magistralei de adrese program pe 24 de biți.

Deși registrul PC (*Program-counter register*) nu face parte din registrele CPU, totuși acesta poate fi modificat prin instrucțiuni program.

**14. Registrul instrucțiune (IR)** pe 32 de biți memorează opcodul instrucțiunii curente, în timpul etapei de decodificare a instrucțiunii. Acest regisztr este folosit de unitatea de decodificare a instrucțiunii și nu este accesibil unității centrale de prelucrare.

## Registrul de stare

Registrul de stare conține informații globale despre starea unității centrale de prelucrare, sub forma indicatorilor de condiții și control.

De regulă, operațiile aritmetice și logice, în funcție de rezultatul obținut, setează indicatorii de condiție ai regisztrului ST. De asemenea, setarea indicatorilor se face și în cazul instrucțiunilor de încărcare în registre și de salvare în memorie.

Asupra regisztrului de stare se pot efectua operații de salvare și de restaurare. Prin operațiile de încărcare (restaurare) a regisztrului ST, biți aceștia sunt înlocuiți de cei ai operandului sursă, indiferent de starea bițiilor operandului sursă. După o astfel de instrucțiune, conținutul lui ST va fi identic cu conținutul operandului sursă.

La inițializarea sistemului, în regisztrul ST se scrie valoarea **00000000H**.

În figura 2.5. se prezintă structura regisztrului de stare ST. Toți biții de interes (indicatorii de condiții și control) se pot citi (*R = read*) și scrie (*W = write*), cu excepția bitului PRGW, care poate fi doar citit.

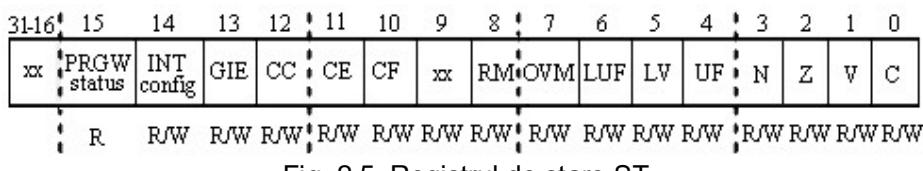


Fig. 2.5. Registrul de stare ST

Din figură, se poate observa că cei mai puțin semnificativi 8 biți sunt indicatori de condiții, iar biți *bit<sub>15</sub>* – *bit<sub>10</sub>* și *bit<sub>8</sub>* reprezintă indicatori de control. Restul bițiilor sunt rezervați, valorile citite fiind 0.

Semnificația indicatorilor de condiții și de control, specificați în regisztrul de stare, este dată în tabelul 2.1.

Tab. 2.1. Indicatorii regisztrului de stare

<b>C</b>	Carry Flag – indicator de transport.
<b>V</b>	Overflow Flag – indicator care este setat dacă numărul întreg rezultat este în afara intervalului de reprezentare (operand cu semn sau fără semn) sau dacă numărul real obținut are exponentul mai mare decât 127. Orice operație care nu produce condițiile menționate șterge indicatorul V.
<b>Z</b>	Zero Flag – indicator de zero.
<b>N</b>	Negative Flag – indicator de rezultat negativ.
<b>UF</b>	Underflow Flag – indicator ce semnalează faptul că exponentul numărului real rezultat este mai mic sau egal cu -128. Indicatorul UF este șters dacă operația aritmetică nu generează un exponent mai mic sau egal cu -128.
<b>LV</b>	Latched Overflow Flag - indicator care este setat de fiecare dată când este setat și indicatorul V.
<b>LUF</b>	Latched Underflow Flag- indicator care este setat de fiecare dată când este setat și indicatorul UF.
<b>OVM</b>	Overflow Mode Flag – acest indicator este afectat doar de operațiile cu numere întregi. Dacă OVM=0 (modul “overflow” este dezactivat), atunci rezultatele operațiilor care conduc la depășiri nu sunt tratate în mod special. Dacă OVM=1, rezultatele operațiilor cu numere întregi care dau depășire în direcția pozitivă sunt trunchiate la 7FFFFFFFh, iar cele care dau depășire în direcție negativă sunt trunchiate la 80000000h.
<b>RM</b>	Repeat Mode Flag – Dacă RM=1, atunci registrul PC este modificat de instrucțiunile de repetare bloc sau repetare instrucțiune.
<b>CE</b>	Cache Enable –activează sau dezactivează memoria cache de instrucțiuni. Dacă CF=1, atunci memoria cache de instrucțiuni este validată și va funcționa ca o stivă LRU ( <i>Least recently used</i> ). Dacă CF=0, memoria cache este dezactivată și nu va fi nici modificată, nici actualizată.
<b>CF</b>	Cache Freeze – activează sau dezactivează cache-ul de instrucțiuni. Dacă CF=1, se “îngheată” memoria cache (cache-ul nu mai este actualizat), inclusiv algoritmul LRU. Dacă și CE=1, atunci se pot citi instrucțiuni din cache, însă nu se pot face modificări în cache. La reset, acest bit este pus în 0 și apoi, după reset, este trecut în 1. Dacă CF=0, memoria cache este automat actualizată prin citiri din memoria externă.

<b>CC</b>	Cache Clear – șterge memoria cache.
<b>GIE</b>	Global Interrupt Enable – activează sau dezactivează întreruperile procesorului.
<b>INT config</b>	Interrupt Configuration – setează semnalele externe de întrerupere INT3-INT0 pentru activare pe palier sau pe front. INT config = 0 – semnalele de întrerupere externe INT3-INT0 sunt configurate pentru activare pe palier. INT config = 1 – întreruperile externe sunt active pe front.
<b>PRGW status</b>	Program Width Status – indică starea pinului extern PRGW. Dacă pinul PRGW este în 1, atunci bitul PRGW trece în 1, indicând faptul că memoria externă este organizată pe cuvinte de 16 biți. Procesorul va efectua 2 cicluri de fetch pentru încărcarea unei instrucțiuni pe 32 biți. Dacă PRGW = 0, se indică organizarea memoriei în cuvinte de 32 biți.

Instrucțiunile procesorului pot testa indicatorii de condiții, în operații cu numere fără semn sau cu semn, precum și individual, după cum se arată în tabele 2.2 ÷ 2.5.

Tab. 2.2. Compararea a două numere fără semn

Condiție	Cod	Descriere	Indicatori
LO	00001	Mai mic	C
LS	00010	Mai mic sau egal	C OR Z
HI	00011	Mai mare	¬C AND ¬Z
HS	00100	Mai mare sau egal	¬C
EQ	00101	Egal	Z
NE	00110	Diferit	¬Z

Tab. 2.3. Compararea a două numere cu semn

Condiție	Cod	Descriere	Indicatori
LT	00111	Mai mic	N
LE	01000	Mai mic sau egal	N OR Z
GT	01001	Mai mare	¬N AND ¬Z
GE	01010	Mai mare sau egal	¬N
EQ	00101	Egal	Z
NE	00110	Diferit	¬Z

Tab. 2.4. Compararea cu zero

Condiție	Cod	Descriere	Indicatori
Z	00101	Zero	Z
NZ	00110	Diferit de zero	¬Z
P	01001	Pozitiv	¬N AND ¬Z
N	00111	Negativ	N

NN	01010	Non-negativ	~N
----	-------	-------------	----

Tab. 2.5. Compararea raportată strict la indicatorii de condiții

Condiție	Cod	Descriere	Indicator
NN	01010	Non-negativ	~N
N	00111	Negativ	N
NZ	00110	Non-zero	~Z
Z	00101	Zero	Z
NV	01100	Non-overflow	~V
V	01101	Overflow	V
NUF	01110	Non-underflow	~UF
UF	01111	Underflow	UF
NC	00100	Non-carry	~C
C	00001	Carry	C
NLV	10000	Non-latched overflow	~LV
LV	10001	Latched overflow	LV
NLUF	10010	Non-latched floating-point underflow	~LUF
LUF	10011	Latched floating-point underflow	LUF
ZUF	10100	Zero sau floating-point underflow	Z sau UF

## Organizarea memoriei interne

Procesorul TMS320C32 poate adresa un spațiu total de memorie de 16 Mcuv. pe 32 de biți, ce poate fi utilizat pentru memorarea programelor, a datelor și a informațiilor despre unitățile I/O. Memoria internă a procesorului conține unități RAM și ROM, în care se pot stoca programe sau date sub formă de tabele, coeficienți etc.

În figura 2.6 se arată modul de organizare a memoriei interne a procesorului TMS320C32. Blocurile 1 și 2 de memorie RAM au fiecare 256 cuvinte de 32 de biți și permit două accesări într-un singur ciclu mașină. De aceea, ele se numesc memorii RAM cu acces dual (DARAM). Aceste memorii pot păstra și programe și date.

De asemenea, un program încărcator de programe (numit *boot-loader*), aflat în memoria ROM de 4Kcuvinte de 32 de biți, permite încărcarea unui program sau a unor date după inițializarea sistemului, de la un port serial sau de la o memorie externă organizată pe 1, 2, 4, 8, 16 sau 32 de biți. Folosind un singur pin extern (*MCBL / MP*), se pot configura primele 1000H cuvinte de memorie pentru a accesa memoria ROM internă a procesorului sau memoria RAM externă.

Memoria ROM, la fel ca și memoria RAM internă, permite 2 accesări de către CPU într-un singur ciclu mașină.

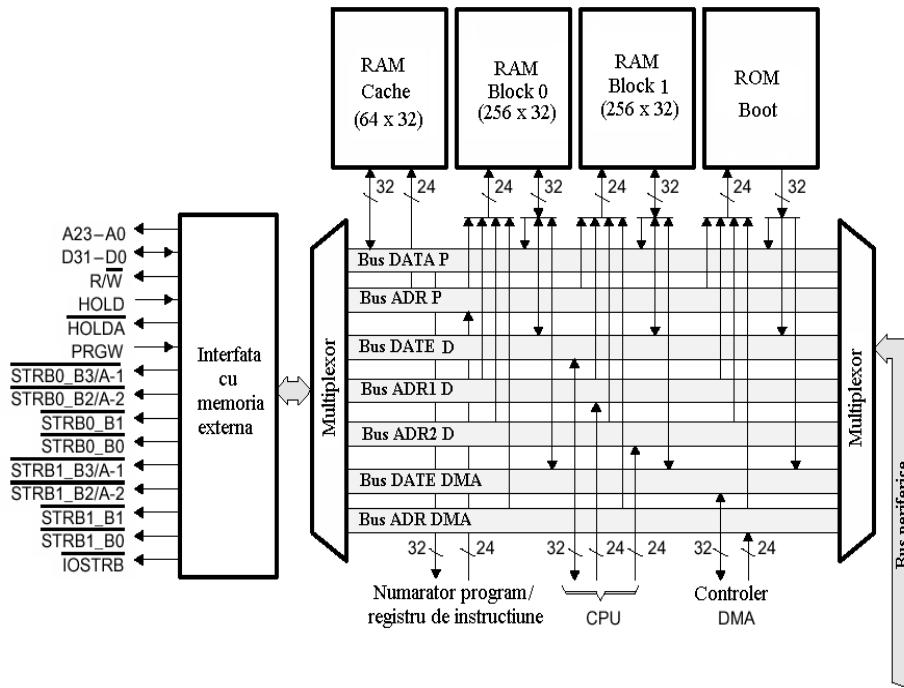


Fig. 2.6. Organizarea memoriei interne a procesorului *TMS320C32*

Multe dintre performanțele procesorului *TMS320C32* se datorează magistrelor interne multiple, ce permit paralelismul în execuție. Astfel, magistralele interne separate pentru program, date și transfer DMA permit executarea în paralel a unor operații de extragere cod instrucție, citire/scriere de date și transferuri pe canale DMA.

Procesorul are 7 magistrale interne: 2 magistrale de program (Bus DATA P și Bus ADR P), 3 magistrale de date (Bus DATE D, Bus ADR1 D și Bus ADR2 D) și 2 magistrale DMA (Bus DATE DMA și Bus ADR DMA).

Aceste magistrale conectează blocurile funcționale ale procesorului și permit executarea mai multor operații în paralel. De exemplu, într-un singur ciclu mașină, unitatea centrală de prelucrare poate accesa 2 date dintr-un bloc RAM și poate efectua o extragere cod instrucție din memoria externă, în timp ce controller-ul DMA poate realiza un transfer de date cu celălalt bloc de memorie RAM.

Registrul PC pe 32 de biți este conectat la magistrala de adrese program (Bus ADR P), prin cei mai puțin semnificativi 24 de biți ai săi. Registrul instrucție IR este conectat la magistrala de date program de 32 de biți. Pe aceste magistrale se poate aduce, în fiecare ciclu mașină, codul unei instrucții pe 32 de biți din unităile de memorie internă sau externă.

Magistralele de adrese pentru transferul datelor pe 24 de biți (Bus ADR1 D și Bus ADR2 D) și magistrala de date de 32 de biți (Bus DATE D) permit două accesări la memoria de date, în fiecare ciclu mașină. Magistrala Bus DATE D transportă date către CPU, prin cele 4 magistrale ale CPU: magistralele CPU1 și CPU2 pot transporta, într-un ciclu mașină, doi operanzi de date la unitatea multiplicativă, respectiv la ALU, iar magistralele REG1 și REG2 pot transporta simultan, într-un ciclu mașină, două date cu registrele din fișierul de registre.

Canalele DMA pot efectua transferuri folosind magistrala de adrese (Bus ADR DMA) pe 24 de biți și magistrala de date (Bus DATE DMA) pe 32 de biți. Aceste magistrale permit

controller-ului DMA să efectueze accesări la memorie în paralel cu transfer de informații pe magistralele de date și de program.

Magistrala externă poate fi folosită pentru implementarea unui sistem de dezvoltare cu DSP, conectându-se diverse componente în exteriorul procesorului. Interfața pentru memoria externă are posibilitatea de a adresa independent date din memorii externe, organizate pe 8, 16 sau 32 de biți.

## Memoria cache

Procesorul TMS320C32 dispune de o memorie cache pentru instrucțiuni, de dimensiune 64 cuvinte de 32 de biți. Conceptul de memorie cache reduce mult numărul de accesări la memoria externă procesorului, care durează mai multe stări de ceas. Astfel, programele pot fi memorate și în memorii externe mai lente, dar mai ieftine. De asemenea, memoria cache eliberează magistralele externe, ce pot fi astfel folosite de către canalele DMA, pentru transferuri de date.

Memoria cache operează automat, fără intervenția utilizatorului, folosindu-se un algoritm LRU pentru actualizarea conținutului ei. Memoria cache este organizată în set asociativ pe 2 căi, fiind împărțită în două segmente de câte 32 de cuvinte.

Memoria externă va fi și ea împărțită în segmente de câte 32 locații de memorie. Corespunzător, adresa de memorie pe 24 de biți este împărțită în 2 câmpuri: adresa de start a segmentului (SSA = Segment start address) pe 19 biți și, respectiv, numărul cuvântului în interiorul segmentului pe 5 biți, ca în figura 2.7.

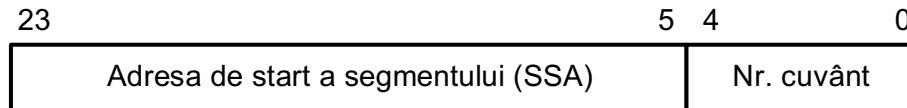


Fig. 2.7. Câmpurile adresăi de memorie pentru algoritmul memoriei cache

Câmpul SSA joacă rol de etichetă și va fi comparat cu valorile SSA (etichetele) memorate de cele două segmente de memorie cache. De aceea, fiecare segment de memorie cache are propriul registru SSA pe 19 biți, ce indică segmentul de memorie care se găsește deja memorat și în memoria cache.

Arhitectura memoriei cache este ilustrată în figura 2.8.

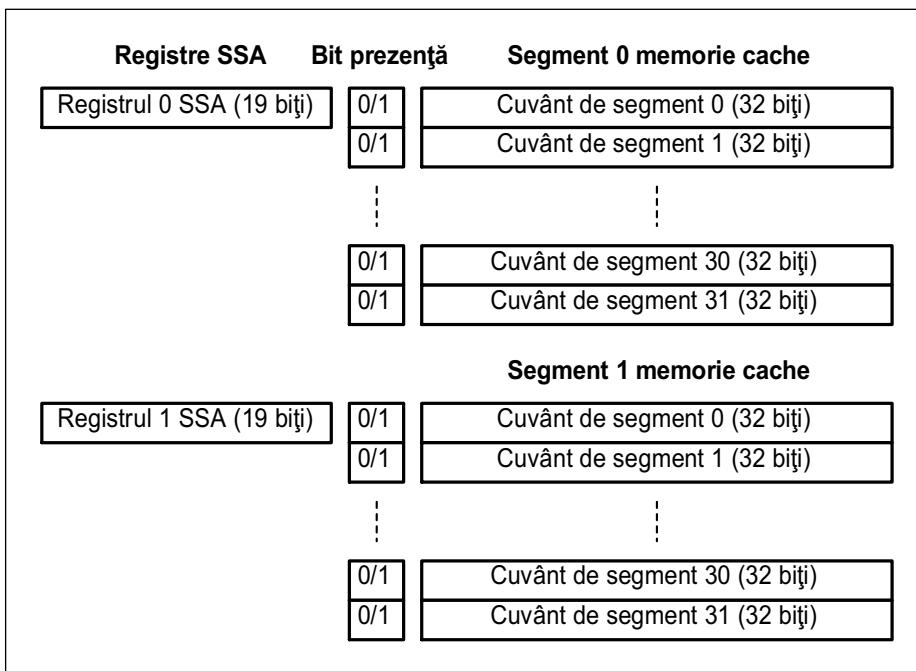


Fig. 2.8. Arhitectura memoriei cache

Din figură se observă că, pentru fiecare cuvânt din cele 2 segmente de memorie cache, se păstrează câte un bit de prezență, care indică dacă acel cuvânt din segmentul de memorie se găsește deja memorat și în memoria cache.

Când unitatea centrală de prelucrare dorește un cuvânt instrucțiune din memoria externă, algoritmul memoriei cache verifică dacă respectivul cuvânt de instrucțiune se găsește deja adus în memoria cache.

Pentru aceasta, algoritmul memoriei cache compară câmpul SSA din adresa de memorie cu valorile registrelor SSA ale memoriei cache. Dacă există o egalitate, se verifică bitul de prezență corespunzător cuvântului dorit din segment (dat de câmpul număr cuvânt din adresă) și, dacă acesta este 1, înseamnă că data dorită se găsește în memoria cache și va fi citită de aici. Dacă una dintre condiții nu se respectă, înseamnă că instrucțiunea dorită nu se găsește în memoria cache și va fi citită din memoria externă. Simultan, ea va fi scrisă și în memoria cache.

### Harta de memorie a procesorului **TMS320C32**

Harta de memorie depinde de modul de lucru al procesorului, determinat de semnalul  $MCBL/\overline{MP}$ : “modul microprocesor” ( $MCBL/\overline{MP}=0$ ), respectiv “modul microcalculator” ( $MCBL/\overline{MP}=1$ ). Harta de memorie este similară pentru cele două moduri, fiind ilustrată în figura 2.9. Se observă că, în ambele moduri de lucru ale procesorului, există mai multe zone de memorie rezervate. De asemenea, blocurile de memorie RAM 0 și 1 se găsesc mapate în aceleași zone de memorie consecutive, începând de la adresele de memorie  $87FE00H$ , respectiv  $87FF00H$ .

Dacă procesorul lucrează în “modul microprocesor”, programul încărcațor de programe (boot-loader) nu se află mapat în harta de memorie a procesorului 'C32. Locațiile de adresă  $0H-7FFFFFFH$  sunt accesate prin portul corespunzător memoriei externe (cu  $STRB0$  activ), la adresa  $000000H$  fiind vectorul de întrerupere *RESET*.

În cazul când se lucrează în “modul microcalculator”, memoria ROM internă a procesorului, ce conține programul boot-loader, este mapată în harta de memorie la adresele  $0-0FFFH$ , iar locațiile  $1000H-7FFFFFFH$  sunt accesate prin intermediul portului pentru memoria externă ( $STRB0$  activ).

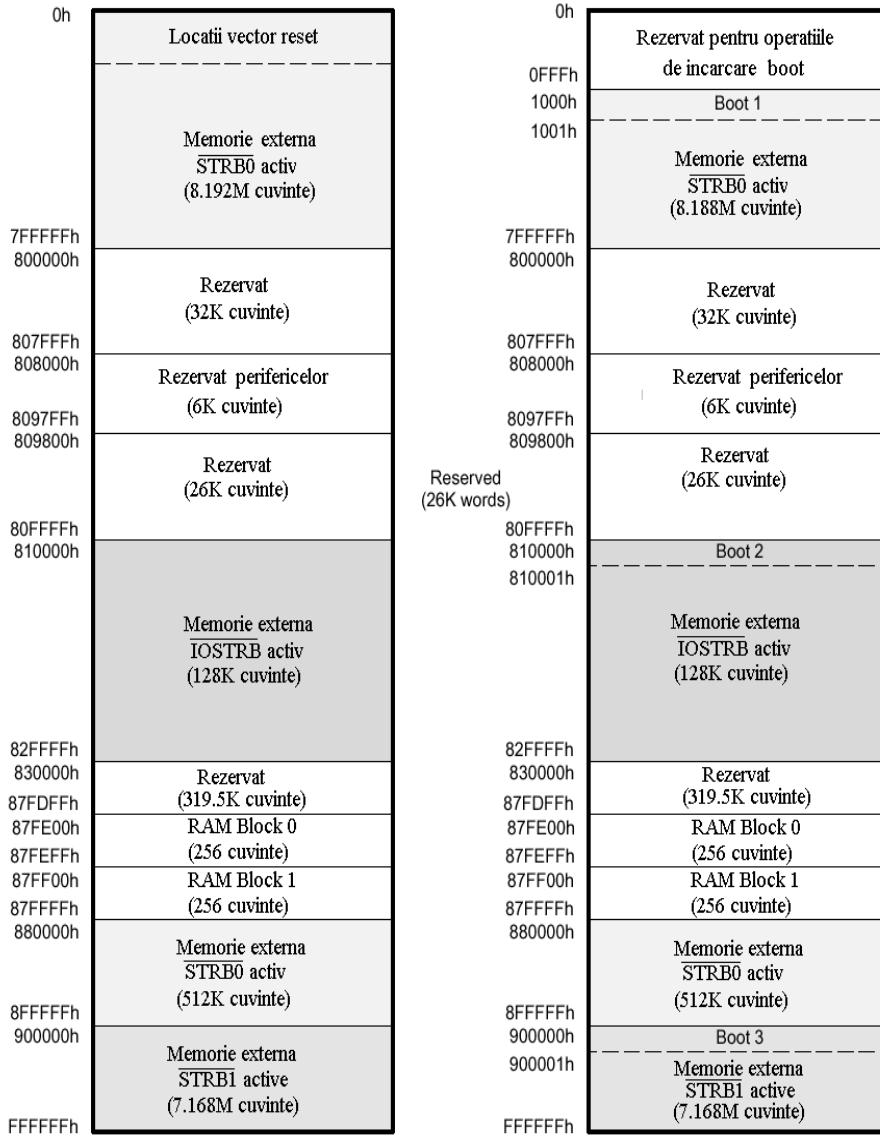


Fig. 2.9. Harta memoriei corespunzătoare procesorului TMS320C32  
**Harta adreselor de memorie asignate porturilor I/O**

Pe lângă harta spațiului de memorie, trebuie cunoscute și adresele de memorie asignate perifericelor și registrelor de control corespunzătoare magistralelor externe. Acestea formează harta adreselor de memorie asignate porturilor I/O și registrelor de control pentru magistralele externe, care este ilustrată în figura 2.10.

808000h	Reg. de control DMA0
808004h	Reg. adrese sursa DMA0
808006h	Reg. adrese destinatie DMA0
808008h	Reg. numarator DMA0
808010h	Reg. de control DMA1
808014h	Reg. adrese sursa DMA1
808016h	Reg. adrese destinatie DMA1
808018h	Reg. numarator DMA1
808020h	Reg. de control Timer 0
808024h	Reg. numarator Timer 0
808028h	Reg. periodizator Timer 0
808030h	Reg. de control Timer 1
808034h	Reg. numarator Timer 1
808038h	Reg. periodizator Timer 1
808040h	Reg. de control port serial
808042h	Control port serial FSX/DX/CLKX
808043h	Control port serial FSR/DR/CLKR
808044h	Control timer R/X port serial
808045h	Numarator timer R/X port serial
808046h	Periodizator timer R/X port serial
808048h	Transmitere data port serial
80804Ch	Receptie data port serial
808060h	Control bus <u>IOSTRB</u>
808064h	Control bus <u>STRB0</u>
808068h	Control bus <u>STRB1</u>
8097FFh	

Fig. 2.10. Harta adreselor de memorie asignate perifericelor

Din figură se observă că procesorul are rezervat un spațiu de memorie pentru registrele perifericelor, începând de la adresa  $808000H$ . Pentru fiecare registru de periferic este asignată o adresă de memorie, iar aceste adrese nu sunt consecutive.

### Sistemul de întreruperi

Procesorul TMS320C32 suportă 4 întreruperi externe, noteate  $\overline{INT3}$ , ...,  $\overline{INT0}$ , un număr predefinit de întreruperi interne și un semnal extern nemascabil de  $\overline{RESET}$ . Acestea pot fi folosite pentru întreruperea unităților master ale procesorului: unitatea centrală de prelucrare și canalele controller-ului DMA.

Atunci când CPU acceptă o cerere de întrerupere, el răspunde cu un semnal specific pe pinul  $\overline{IACK}$ , pentru a semnala, în exterior, acceptarea întreruperii externe.

Întreruperile pot fi validate sau invalidate prin intermediul registrului IE, iar starea întreruperilor validate poate fi monitorizată prin registrul IF.

### REGISTRUL DE VALIDARE A ÎNTRERUPERILOR CPU/DMA (IE)

Registrul IE este un registru pe 32 de biți utilizat atât la validarea întreruperilor CPU prin intermediul bițiilor  $bit_{11} \div bit_0$ , cât și pentru validarea întreruperilor generate către cele două canale DMA, folosind biți  $bit_{31} \div bit_{16}$ . Configurația bițiilor registrului IE este ilustrată în figura 2.11.

Dacă un bit este încărcat cu valoarea 1, atunci întreruperea corespunzătoare bitului este validată, iar dacă valoarea este 0, întreruperea este invalidată. La inițializarea sistemului, registrul IE este încărcat cu valoarea  $00000000H$ , toate întreruperile fiind invalide.

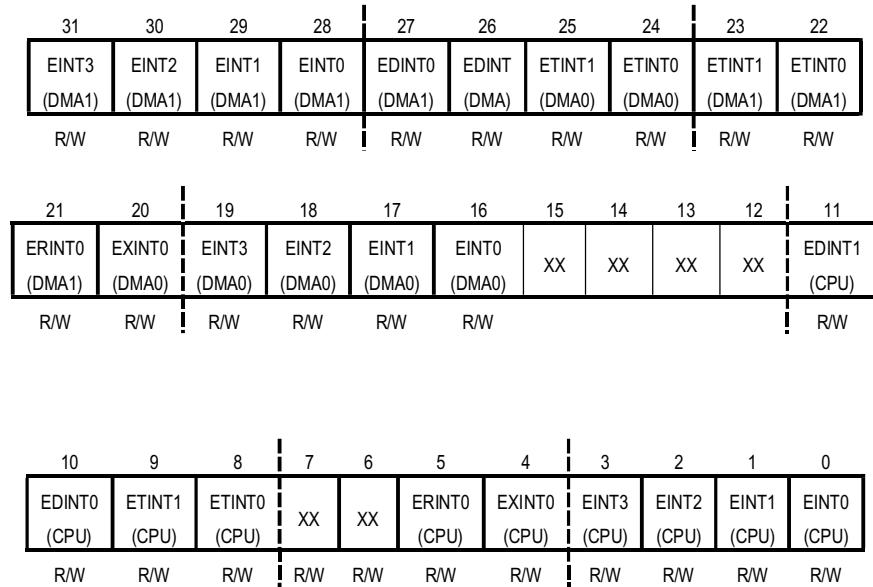


Fig. 2.11. Registrul de validare a întreruperilor

Semnificațiile indicatorilor registrului IE sunt date în tabelul 2.6.

Tab. 2.6. Indicatorii registrului de activare a întreruperilor

Bit	Denumire	Semnificație
0	<b>EINT0 (CPU)</b>	Validare întrerupere externă 0 a CPU
1	<b>EINT1 (CPU)</b>	Validare întrerupere externă 1 a CPU
2	<b>EINT2 (CPU)</b>	Validare întrerupere externă 2 a CPU
3	<b>EINT3 (CPU)</b>	Validare întrerupere externă 3 a CPU
4	<b>EXINT0 (CPU)</b>	Validare întreruperea transmisie serială
5	<b>ERINT0 (CPU)</b>	Validare întrerupere receptie serială
6	<b>xx</b>	Utilizat numai de procesorul TMS320C30
7	<b>xx</b>	Utilizat numai de procesorul TMS320C30
8	<b>ETINT0 (CPU)</b>	Validare întrerupere timer 0 către CPU
9	<b>ETINT1 (CPU)</b>	Validare întrerupere timer 1 către CPU
10	<b>EDINT0 (CPU)</b>	Validare întrerup. controller DMA0 către CPU
11	<b>EDINT1 (CPU)</b>	Validare întrerup. controller DMA1 către CPU
16	<b>EINT0 (DMA0)</b>	Validare întrerupere externă 0 către DMA0
17	<b>EINT1 (DMA0)</b>	Validare întrerupere externă 1 către DMA0
18	<b>EINT2 (DMA0)</b>	Validare întrerupere externă 2 către DMA0
19	<b>EINT3 (DMA0)</b>	Validare întrerupere externă 3 către DMA0
20	<b>EXINT0 (DMA0)</b>	Valid. întrerup. de transmisie serială prin DMA0
21	<b>ERINT0 (DMA1)</b>	Valid. întrerup. de receptie serială prin DMA1
22	<b>ETINT0 (DMA1)</b>	Validare întrerupere timer 0 către DMA1
23	<b>ETINT1 (DMA1)</b>	Validare întrerupere timer 1 către DMA1
24	<b>ETINT0 (DMA0)</b>	Validare întrerupere timer 0 către DMA0
25	<b>ETINT1 (DMA0)</b>	Validare întrerupere timer 1 către DMA0

<b>26</b>	<b>EDINT1 (DMA0)</b>	Validare întrerupere contr. DMA1 către DMA0
<b>27</b>	<b>EDINT0 (DMA1)</b>	Validare întrerupere contr. DMA0 către DMA1
<b>28</b>	<b>EINT0 (DMA1)</b>	Validare întrerupere externă 0 către DMA1
<b>29</b>	<b>EINT1 (DMA1)</b>	Validare întrerupere externă 1 către DMA1
<b>30</b>	<b>EINT2 (DMA1)</b>	Validare întrerupere externă 2 către DMA1
<b>31</b>	<b>EINT3 (DMA1)</b>	Validare întrerupere externă 3 către DMA1

Întreruperile validate pentru canalele DMA permit sincronizarea transferului de date între periferice și memoria DSP-ului, fără intervenția unității centrale. De asemenea, se poate observa faptul că și cele două canale DMA pot să-și sincronizeze activitatea, prin setarea adecvată a bițiilor 26 și 27.

Multiplele variante de întreruperi, de la un periferic spre toate unitățile master, permit implementarea unui sistem foarte flexibil și dinamic, optimizându-se astfel transferurile de date între periferice și memorie.

### Registrul indicatorilor de întreruperi CPU (IF)

Registrul indicatorilor de întreruperi CPU (IF) pe 32 de biți semnalează care întreruperi au fost activate de către unitățile slave. Biți acestui registru se setează în momentul în care un periferic intern sau extern generează o întrerupere deja validată. De asemenea, un bit se poate seta și prin program, cauzând o întrerupere care apoi se tratează, ca și cum această întrerupere ar fi declanșată de un periferic.

Dacă un bit al registrului IF are valoarea 1, atunci acesta indică apariția unei întreruperi CPU generate de perifericul corespunzător bitului setat. Dacă într-un bit al registrului IF se înscrie valoarea zero, atunci întreruperea corespunzătoare este ştearsă. La inițializarea procesorului, conținutul bițiilor registrului IF este **0000000H**.

Registrul indicatorilor de întreruperi CPU este ilustrat în figura 2.12, iar semnificația bițiilor acestuia este dată în tabelul 2.7.

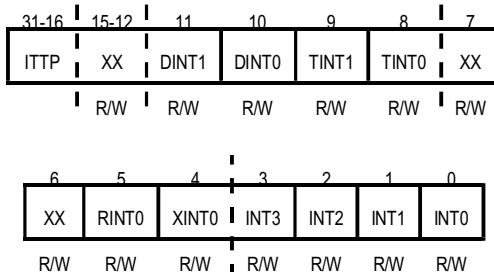


Fig. 2.12. Registrul indicatorilor de întreruperi CPU

Tab. 2.7. Semnificația indicatorilor de întreruperi

<b>INT0</b>	Indicator pentru întreruperea externă 0
<b>INT1</b>	Indicator pentru întreruperea externă 1
<b>INT2</b>	Indicator pentru întreruperea externă 2

<b>INT3</b>	Indicator pentru întreruperea externă 3
<b>XINT0</b>	Indicator pentru întreruperea de transmisie serială
<b>RINT0</b>	Indicator pentru întreruperea de recepție serială
<b>TINT0</b>	Indicator pentru întreruperea generată de timer 0
<b>TINT1</b>	Indicator pentru întreruperea generată de timer 1
<b>DINT0</b>	Indicator pentru întreruperea generată de DMA0
<b>DINT1</b>	Indicator pentru întrerupere generată de DMA1
<b>ITTP</b>	Indicatorul tablei de întreruperi și execuție pas cu pas

### Tabela vectorilor de întreruperi și execuție pas cu pas

Cei mai semnificativi 16 biți ai registrului IF ( $bit_{31} \div bit_{16}$ ) reprezintă indicatorul tablei vectorilor de întreruperi și de execuție pas cu pas (ITTP). Aceasta permite realocarea în memorie a tablei vectorilor.

Vectorul *RESET* al procesorului *TMS320C32*, ca și în cazul celorlalte procesoare din familia lui, se află la adresa *000000H*. Totuși, tabela de vectori pentru întreruperi și de execuție pas cu pas (*trap*) a unui program se poate situa și la o altă adresă de memorie, fiind stabilită de utilizator prin ITTP.

Grupul de 16 biți ITTP stabilește adresa de start a tablei vectorilor de întreruperi și de execuție pas cu pas. Adresa de bază pe 24 de biți a tablei este calculată prin deplasare la stânga cu 8 poziții binare, a bițiilor dați de ITTP. Valoarea obținută se numește adresă de bază efectivă și se notează cu EA [ ITTP ].

În figura 2.13 se prezintă modul de calcul al adresei de bază efectivă a tablei vectorilor de întreruperi și de execuție pas cu pas.

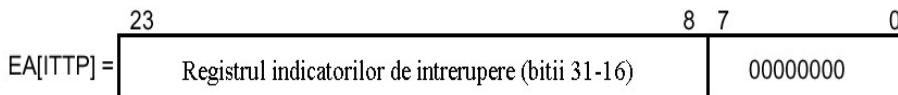


Fig. 2.13. Calculul adresei de bază efectivă a tablei ITTP

Adresa unei anumite intrări în tabelă este dată prin adunarea offset-ului vectorului de întrerupere respectiv, la adresa de bază efectivă. La această intrare se găsește memorată adresa de început a subretinei de tratare a întreruperii respective sau de execuție pas cu pas. De asemenea, cunoscând valoarea offset-ului, intrarea în tabelă se poate calcula și prin concatenarea câmpului ITTP cu valoarea offset-ului pe 8 biți.

În figura 2.14 este ilustrat modul de ocupare a tablei cu vectorii de întreruperi și de execuție pas cu pas. Se observă că adresele de memorie pentru fiecare intrare în tabelă sunt calculate în funcție de adresa de bază efectivă a tablei și de offset-ul vectorului respectiv, în interiorul tablei.

Pentru întreruperile vectorizate, offset-ul unei întreruperi este dat de numărul vectorului de întrerupere. Când o întrerupere CPU validată în registrul IE este generată de

către un periferic, procesorul determină vectorul de întrerupere corespunzător și calculează intrarea în tabela vectorilor de întreruperi. De la această locație de memorie se citește adresa de început a subruteinei de tratare a întreruperii respective.

EA (ITTP) + 00h	Rezervat
EA (ITTP) + 01h	INT0
EA (ITTP) + 02h	INT1
EA (ITTP) + 03h	INT2
EA (ITTP) + 04h	INT3
EA (ITTP) + 05h	XINT0
EA (ITTP) + 06h	RINT0
EA (ITTP) + 07h	Rezervat
EA (ITTP) + 08h	Rezervat
EA (ITTP) + 09h	TINT0
EA (ITTP) + 0Ah	TINT1
EA (ITTP) + 0Bh	DINT0
EA (ITTP) + 0Ch	DINT1
EA (ITTP) + 0Dh	Rezervat
EA (ITTP) + 1Fh	
EA (ITTP) + 20h	TRAP0
	:
	:
	:
EA (ITTP) + 3Bh	TRAP27
EA (ITTP) + 3Ch	TRAP28 (rezervat)
EA (ITTP) + 3Dh	TRAP29 (rezervat)
EA (ITTP) + 3Eh	TRAP30 (rezervat)
EA (ITTP) + 3Fh	TRAP31 (rezervat)

Fig. 2.14. Tabela vectorilor de întreruperi și de execuție pas cu pas

# **CAPITOLUL 3**

## **SETUL DE INSTRUCȚIUNI AL PROCESORULUI DE SEMNAL TMS320C32**

### **3.1. Introducere**

În general, realizarea unui program în limbaj de asamblare pentru un anumit procesor presupune cunoștințe aprofundate de hardware și software, atât despre procesorul propriu-zis, cât și despre sistemul de calcul în care acesta este utilizat. Astfel, programatorul trebuie să cunoască modul în care procesorul lucrează cu unitățile de memorie și porturile I/O, atât cele interne, cât și cele externe plasate în sistemul de calcul respectiv. De asemenea, sunt necesare cunoștințe privind structurile de date cu care operează procesorul, modurile lor de utilizare, precum și setul de instrucții. În plus, experiența programării în limbaj de asamblare este un factor, ce contribuie într-o mare măsură la obținerea unor programe utilizator performante.

Programarea în limbajul de asamblare specific procesorului *TMS320C32* necesită, pe lângă noțiunile hardware prezentate în capitolul anterior, și cunoștințe software privind tipurile de date cu care lucrează procesorul, modurile de adresare și setul de instrucții, ce vor fi aprofundate în acest capitol. Trebuie menționat faptul că noțiunile sunt prezentate într-o formă concisă, realizarea unui program în limbaj de asamblare presupunând o consultare continuă a documentației oferite de firmă.

Setul de instrucțiuni conține, pe lângă instrucțiunile de bază de interes general, și instrucțiuni aritmetice, orientate pe procesarea paralelă a semnalelor. De aceea, procesorul *TMS320C32* se utilizează în aplicații ce implică un volum mare de calcul. Toate instrucțiunile ocupă în memorie un singur cuvânt, iar majoritatea instrucțiunilor au nevoie pentru execuție doar de un singur ciclu mașină. În total, setul de instrucțiuni conține 113 instrucțiuni, împărțite în următoarele clase funcționale:

- instrucțiuni de transfer;
- instrucțiuni de prelucrare cu 2 operanzi;
- instrucțiuni de prelucrare cu 3 operanzi;
- instrucțiuni paralele de transfer și de prelucrare;
- instrucțiuni de control;
- instrucțiuni de interblocare.

### 3.2. Tipuri de date

În general, datele numerice pot fi reprezentate în virgulă fixă sau în virgulă mobilă, necesitând unități de prelucrare diferite în procesor.

Procesorul de semnal *TMS320C32* poate opera cu trei tipuri de date numerice:

- numere întregi fără semn (*unsigned integer*)
- numere întregi cu semn, numite pe scurt numere întregi (*signed integer* sau *integer*)
- numere reale reprezentate în virgulă mobilă (*floating-point*).

Numerele întregi cu sau fără semn sunt reprezentate în virgulă fixă și pot fi în format scurt sau în format simplă precizie, iar numerele reale sunt reprezentate în virgulă mobilă, putând fi în format scurt, precizie simplă sau precizie extinsă.

Arhitectura procesorului permite realizarea operațiilor în virgulă mobilă, la aceeași viteză cu cele în virgulă fixă, fără a necesita o tratare specială a acestor numere. În plus, sunt eliminate problemele specifice operațiilor cu numere întregi, cum ar fi depășirea, alinierea operanzilor etc. De aceea, aceste DSP-uri se folosesc la aplicații care cer un volum de calcul foarte mare cu numere reale.

### 3.2.1. Reprezentarea numerelor în virgulă fixă

Numerele întregi sunt reprezentate în virgulă fixă, prin cuvinte binare pe  $N$  biți, putându-se astfel reprezenta  $2^N$  numere diferite. Intervalul de numere ce pot fi reprezentate pe  $N$  biți diferă însă în funcție de tipul operandului.

Dacă operandul este număr întreg fără semn, considerat implicit număr natural, atunci toate cele  $2^N$  combinații binare vor reprezenta numere naturale, iar intervalul de reprezentare pe  $N$  biți este:  $I_1 = [0, 2^N-1]$ .

Dacă operandul este număr întreg cu semn, atunci trebuie generată o regulă de reprezentare a semnului, care nu este unică. Există mai multe coduri de reprezentare a numerelor negative, cum ar fi: codul semn-magnitudine, codul complement față de 1 sau codul complement față de 2.

Toate codurile de reprezentare a operanzilor întregi cu semn au o serie de caracteristici comune:

- semnul (S) reprezintă bitul cel mai semnificativ din combinația binară ( $bit_{N-1}$ ), având semnificația:  $S = 0$  număr pozitiv și  $S = 1$  număr negativ;
- numerele pozitive au aceeași reprezentare în toate codurile;
- numerele negative sunt reprezentate diferit, însă codurile trebuie să respecte regula de negare aritmetică a oricărui număr  $A$ :  
$$(-A) = A$$

În codul complement față de 2, reprezentarea binară a unui număr negativ ( $-A$ ) se obține din reprezentarea numărului pozitiv  $A$ , negând logic toți biții și adunând 1. De exemplu, pentru  $N = 8$ , pornind de la numărul întreg pozitiv  $A = 85 = 01010101_b$ , se poate obține reprezentarea binară a numărului negativ  $-A$ :

$$-A = -85 = 10101011_b$$

Codul complement față de 2 are două avantaje majore față de celelalte două coduri de reprezentare:

- valoarea 0 are reprezentare unică (+0), față de reprezentarea dublă în celelalte două coduri (- 0 și respectiv + 0). Din acest motiv, intervalul de reprezentare în complement față de 2 este mai mare cu 1, în zona numerelor negative;

- rezultatul operațiilor aritmetice simple se obține direct, fără a fi necesare alte corecții efectuate asupra sa.

Datorită acestor avantaje, codul complement față de 2 este implementat în majoritatea microprocesoarelor.

Pentru codul complement față de 2, intervalul de reprezentare pe  $N$  biți a numerelor întregi cu semn este:  $I_2 = [-2^{N-1}, 2^{N-1}-1]$ .

În figura 3.1 sunt ilustrate intervalele de reprezentare în virgulă fixă pe  $N$  biți a numerelor întregi fără semn și a celor cu semn, reprezentate în complement față de 2.

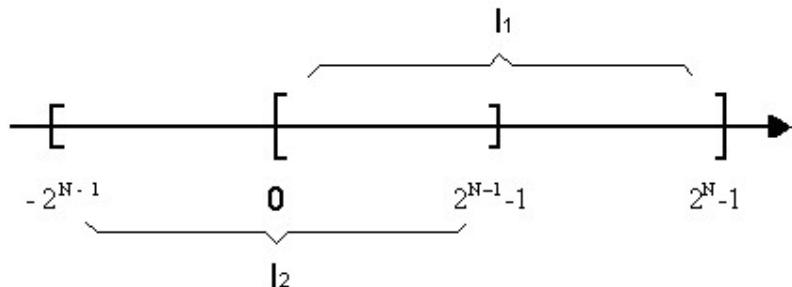


Fig. 3.1. Intervalele de reprezentare în virgulă fixă pe  $N$  biți

Operațiile cu numere în virgulă fixă sunt realizate de aceeași unitate aritmetico-logică, indiferent dacă operanții sunt întregi cu semn sau fără semn. Trebuie precizat că ALU operează cu numere binare, iar semnificația acestora, de operanți cu sau fără semn, este cunoscută doar de programator.

De exemplu, pentru  $N = 8$ , intervalele de reprezentare a numerelor în virgulă fixă pe 8 biți sunt:  $I_1 = [0, 255]$ , respectiv  $I_2 = [-128, 127]$ . În aceste condiții, operandul binar pe 8 biți  $M = 10110100_b$  poate reprezenta numărul întreg fără semn 180 sau numărul întreg cu semn -76.

În funcție de tipul operanților, programatorul testează indicatorii de condiții la sfârșitul operației, pentru interpretarea rezultatului.

Procesorul de semnal *TMS320C32* poate opera cu numere în virgulă fixă, reprezentate în două tipuri de formate: formatul scurt, în care numerele sunt reprezentate pe 16 biți ( $N = 16$ ), respectiv formatul simplă precizie, în care numerele sunt reprezentate pe 32 de biți ( $N = 32$ ).

**Numerele întregi fără semn în format scurt** sunt reprezentate pe 16 biți, rezultând intervalul de reprezentare  $I_1 = [0, 2^{16}-1]$ . Un operand imediat fără semn în format scurt va fi indicat în câmpul instrucțiunii pe 16 biți. Însă, procesorul TMS320C32 memorează operanții în virgulă fixă, în registrele proprii, pe 32 de biți și toate operațiile de virgulă fixă sunt realizate pe 32 de biți.

De aceea, operanții fără semn în format scurt sunt utilizati pe 32 de biți, completându-se cei mai semnificativi 16 biți cu 0, după cum se arată în figura 3.2.

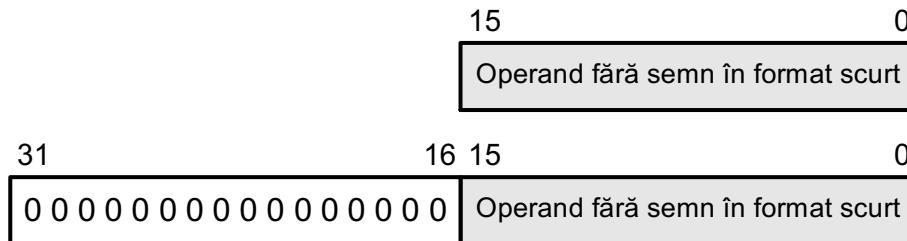


Fig. 3.2. Reprezentarea numerelor întregi fără semn în format scurt

**Numerele întregi cu semn în format scurt** sunt reprezentate tot pe 16 biți, în complement față de 2, pentru care intervalul de reprezentare este  $I_2 = [-2^{15}, 2^{15}-1]$ .

Similar, un operand imediat cu semn în format scurt va fi indicat în câmpul instrucțiunii pe 16 biți, însă, utilizarea sa de către procesor se va face pe 32 de biți, completându-se cei mai semnificativi 16 biți cu valoarea semnului (S), după cum se arată în figura 3.3.

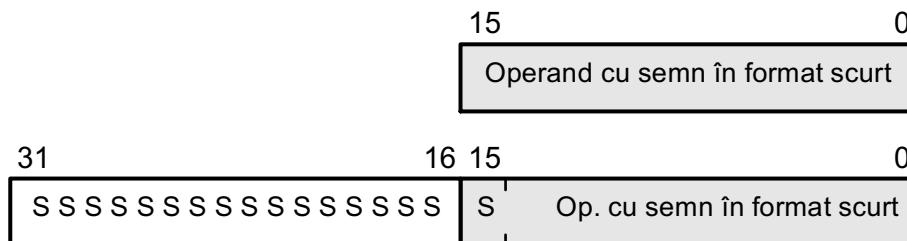


Fig. 3.3. Reprezentarea numerelor întregi cu semn în format scurt

**Numerele întregi în format simplă precizie** sunt reprezentate pe 32 de biți. Intervalele de reprezentare a numerelor în virgulă fixă pe 32 de biți sunt:  $I_1 = [0, 2^{32}-1]$  pentru operanzi fără semn, respectiv  $I_2 = [-2^{31}, 2^{31}-1]$  pentru operanzi cu semn în complement față de 2.

### 3.2.2. Reprezentarea numerelor în virgulă mobilă

Spre deosebire de reprezentarea în virgulă fixă, formatul în virgulă mobilă permite reprezentarea numerelor reale foarte mici ( $\approx \pm 10^{-38}$ ) sau foarte mari ( $\approx \pm 10^{38}$ ).

Numerele reale sunt reprezentate în virgulă mobilă, în formatul:  $M \cdot B^E$ , în care  $M$  reprezintă mantisa,  $B$  este baza, iar  $E$  reprezintă exponentul. Baza are valoare fixată, fiind în general aleasă  $B = 2$ .

Mantisa este un număr în virgulă fixă cu semn, ce poate fi reprezentată diferit, în funcție de formatul de virgulă mobilă utilizat, în cod semn-magnitudine (formatul standard IEEE 754) sau complement față de 2 (formatul procesorului TMS320C32). Exponentul este un număr întreg cu semn, exprimat în complement față de 2. Pentru majoritatea microprocesoarelor, exponentul este pe maxim 8 biți, de unde rezultă valorile maxime și minime de reprezentare a numerelor reale.

Deoarece baza are valoare fixată și cunoscută, un număr real exprimat în virgulă mobilă poate fi reprezentat printr-o pereche de numere în virgulă fixă:  $(M, E)$ . Astfel, operațiile cu numere reale, care sunt realizate în unități aritmetice speciale de virgulă mobilă, se reduc la un set de operații de virgulă fixă.

Pentru procesorul de semnal TMS320C32, formatul general al numerelor în virgulă mobilă conține trei câmpuri, fiind ilustrat în figura 3.4: câmpul exponent (E), bitul de semn (S) și câmpul fracție (F). Bitul de semn și câmpul fracție formează câmpul mantisă.

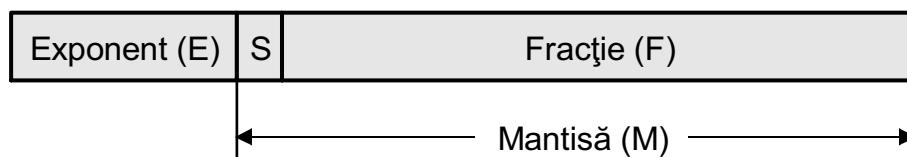


Fig. 3.4. Formatul general al numerelor în virgulă mobilă

Formula generală de calcul al unui număr real  $x$ , reprezentat în virgulă mobilă în formatul specific procesorului TMS320C32, este:

$$x = S\bar{S}.F_2 \cdot 2^E,$$

unde  $S$  = bitul de semn,  $\bar{S}$  = valoarea negată a bitului de semn,  $F_2$  = valoarea binară a câmpului fractie, iar  $E$  = valoarea zecimală echivalentă a câmpului exponent.

Comparând acest format cu formatul general al numerelor reprezentate în virgulă mobilă, se constată că mantisa este un număr binar, care are parte întreagă (formată din doi biți  $S\bar{S}$ ) și parte fracționară ( $F_2$ ):

$$M = S\bar{S}.F_2$$

Mantisa ( $M$ ) reprezintă un număr cu semn în complement față de 2, normalizat. În reprezentarea normalizată, în afară de bitul de semn ( $S$ ), pentru partea întreagă este utilizat un bit suplimentar, care reprezintă totdeauna valoarea negată a bitului de semn ( $\bar{S}$ ). Se obține astfel o precizie mărită cu un bit.

Dacă  $S = 0$  (numărul real este pozitiv), atunci cei doi biți ai mantisei din fața punctului binar sunt  $S\bar{S} = 01$ . În acest caz, partea întreagă a mantisei va fi egală cu 1, iar mantisa  $M$  va fi un număr pozitiv,  $M = 01.F_2 \in [1, 2)$ . Dacă valoarea binară a fractiei este zero ( $F_2 = 0b$ ), atunci  $M = 1$ . Se observă că, pentru  $E = S = F = 0b$ ,  $x = 1$ .

Dacă  $S = 1$  (numărul real este negativ), atunci cei doi biți ai mantisei din fața punctului binar sunt  $S\bar{S} = 10$ . În acest caz, partea întreagă a mantisei va fi egală cu -2 (complement față de 2), iar mantisa  $M$  va fi un număr negativ,  $M = 10.F_2 \in [-2, -1)$ . Dacă valoarea binară a fractiei este zero ( $F_2 = 0b$ ), atunci  $M = -2$ . De asemenea, se observă că, pentru  $E = F = 0b$  și  $S = 1$ ,  $x = -2.0$ .

În orice bază, înmulțirea unui număr cu baza  $B$  la o putere  $k$  ( $B^k$ ) se obține prin deplasarea virgulei  $k$  poziții la dreapta sau la stânga, în funcție de valoarea lui  $k$ , eventual completând numărul cu zerouri.

Deoarece mantisa  $M$  este un număr binar, înmulțirea ei cu  $2^E$  se obține prin deplasarea virgulei la dreapta (dacă exponentul este pozitiv) sau la stânga (dacă exponentul este negativ), un număr de poziții egal cu valoarea exponentului  $E$ .

În funcție de dimensiunile exponentului ( $E$ ) și ale fractiei ( $F$ ), procesorul TMS320C32 poate opera cu numere în virgulă mobilă,

reprezentate în trei tipuri de formate: format scurt, precizie simplă sau precizie extinsă.

**Numerele reale în format scurt** sunt reprezentate în virgulă mobilă pe 16 biți în complement față de 2, astfel: 4 biți pentru exponent, 1 bit de semn și 11 biți pentru câmpul fracție, după cum se arată în figura 3.5.

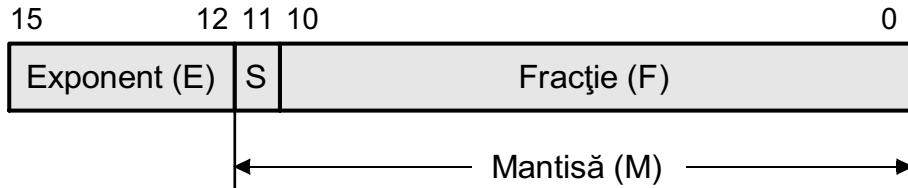


Fig. 3.5. Reprezentarea numerelor în virgulă mobilă în format scurt

Exponentul este operand cu semn în complement față de 2 pe 4 biți, având valori în intervalul [-8, 7]. Valorile maxime pozitive și negative se obțin pentru  $E = 7$ , iar valorile minime pozitive și negative se obțin pentru  $E = -7$ , ilustrate în tabelul 3.1. Valoarea  $E = -8$  este rezervată pentru reprezentarea numărului real 0.0 ( $E=-8$ ,  $S=F=0$ ), a cărui reprezentare în format scurt este 8000h.

Tab. 3.1. Valorile maxime și minime pentru virgula mobilă în format scurt

Număr real	Reprezentare în format scurt (hexa)	Valoare reală (binar)	Valoare reală (zecimal)
maxim pozitiv	77FFh	01.1111111111 <sub>2</sub> · 2 <sup>7</sup>	255.94
minim pozitiv	9000h	01.00000000000 <sub>2</sub> · 2 <sup>-7</sup>	0.0078125
minim negativ	9FFFh	10.1111111111 <sub>2</sub> · 2 <sup>-7</sup>	-0.0078163
maxim negativ	7800h	10.00000000000 <sub>2</sub> · 2 <sup>7</sup>	-256

Când un operand imediat este indicat în format scurt în câmpul instrucțiunii, el este memorat în registrele procesorului de 32 de biți, aliniat la stânga, pe pozițiile celor mai semnificativi 16 biți.

**Numerele reale în format simplă precizie** sunt reprezentate în virgulă mobilă pe 32 de biți, astfel: 8 biți pentru exponent, 1 bit de semn și 23 de biți pentru câmpul fractie, după cum se arată în figura 3.6. Exponentul este operand cu semn în complement față de 2 pe 8 biți, cu valori în intervalul [-128, 127]. Valoarea minimă  $E = -128$  este rezervată pentru reprezentarea numărului real 0.0 ( $E = -128$ ,  $S=F=0$ ), a cărui reprezentare în format simplă precizie este 80000000h.

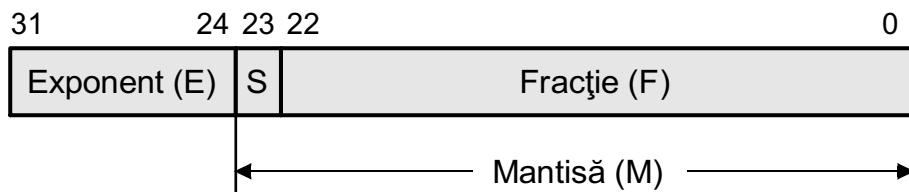


Fig. 3.6. Reprezentarea în virgulă mobilă în format simplă precizie

Valorile maxime pozitive și negative se obțin pentru  $E = 127$ , iar valorile minime pozitive și negative se obțin pentru  $E = -127$ , fiind ilustrate în tabelul 3.2.

Tab. 3.2. Valori maxime și minime pentru virgula mobilă în simplă precizie

Număr real	Format simplă precizie (hexa)	Valoare reală (binar)	Valoare reală (zecimal)
maxim pozitiv	7F7FFFFFFh	01.111111111111111111111111111111 <sub>2</sub> · 2 <sup>127</sup>	3.4028234 · 10 <sup>38</sup>
minim pozitiv	81000000h	01.000000000000000000000000000000 <sub>2</sub> · 2 <sup>-127</sup>	5.8774717 · 10 <sup>-39</sup>
minim negativ	81FFFFFFh	10.111111111111111111111111111111 <sub>2</sub> · 2 <sup>-127</sup>	-5.8774724 · 10 <sup>-39</sup>
maxim negativ	7F800000h	10.000000000000000000000000000000 <sub>2</sub> · 2 <sup>127</sup>	-3.4028236 · 10 <sup>38</sup>

**Numerele reale în format precizie extinsă** sunt reprezentate în virgulă mobilă pe 40 de biți, astfel: 8 biți pentru exponent, 1 bit de semn și 31 de biți pentru câmpul fractie, după cum se arată în figura 3.7. Aceste numere pot fi memorate în registrele de precizie mărită  $R7 \dots R0$  și sunt exprimate pe 10 cifre hexa.

Similar formatului simplă precizie, exponentul este tot operand cu semn în complement față de 2 pe 8 biți, cu valori în intervalul [-128, 127]. Valoarea minimă  $E=-128$  este rezervată pentru reprezentarea numărului real 0.0 ( $E = -128$ ,  $S = F = 0$ ), a cărui reprezentare în format precizie extinsă este 8000000000h.

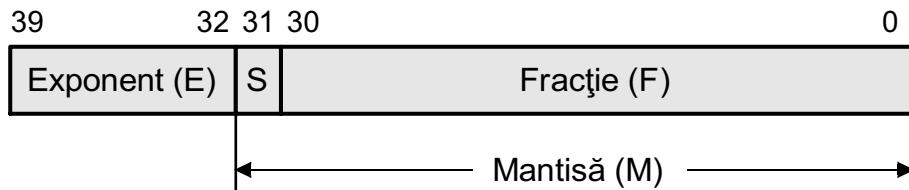


Fig. 3.7. Reprezentarea în virgulă mobilă în format precizie extinsă

De asemenea, valorile maxime pozitive și negative se obțin pentru  $E = 127$ , iar cele minime pozitive și negative se obțin pentru  $E = -127$ , fiind ilustrate în tabelul 3.3.

Tab. 3.3. Valori maxime și minime pentru virgula mobilă în precizie extinsă

Număr real	Format simplă precizie (hexa)	Valoare reală (zecimal)
maxim pozitiv	7F7FFFFFFFh	$3.4028236684 \cdot 10^{38}$
minim pozitiv	8100000000h	$5.8774717541 \cdot 10^{-39}$
minim negativ	81FFFFFFFh	$-5.8774717568 \cdot 10^{-39}$
maxim negativ	7F80000000h	$-3.4028236692 \cdot 10^{38}$

Valoarea reală în binar este similară formatului anterior, fractia fiind însă reprezentată pe 31 de biți. Deoarece câmpul fractie are dimensiune mai mare, numerele reprezentate vor avea precizie mai mare față de formatul simplă precizie.

### 3.3. Moduri de adresare

În general, instrucțiunile unui procesor sunt flexibile, putând fi folosite cu mai multe tipuri de operanzi. Un operand se poate afla într-un registru al procesorului, în câmpul instrucțiunii sau într-o locație de memorie, caz în care operandul poate fi accesat în mai multe feluri. Modul de adresare al unei instrucțiuni indică procesorului unde se găsește operandul necesar instrucțiunii și procedeul prin care acesta poate fi obținut. Modul de adresare este inclus în cuvântul de cod, fiind precizat implicit de utilizator prin sintaxa instrucțiunii.

Procesorul de semnal *TMS320C32* suportă cinci moduri de adresare, ce permit accesul la date din registre proprii, din memorie sau din cuvintele de cod ale instrucțiunilor: adresarea la registru, adresarea directă, cea indirectă, imediată și respectiv adresarea relativă la registrul PC. De asemenea, procesorul dispune de două moduri de adresare suplimentare, utilizate în algoritmii de prelucrare a semnalelor, în filtre sau în calculul *FFT*: adresarea circulară și adresarea în ordinea inversă a bițiilor. Modurile de adresare vor fi prezentate succint, fiind însăși de exemple.

**1. Adresarea la registru** se utilizează în cazul când operandul se găsește într-unul din registrele unității centrale. Acest mod de adresare este cel mai rapid, deoarece operandul se află deja adus în procesor, fiind aproape de unitățile de prelucrare. De aceea, este indicat ca adresarea la registru să se utilizeze cât mai frecvent, mai ales că procesorul *TMS320C32* conține un număr considerabil de registre, ce permit rezolvarea unor probleme punctuale fără a mai apela la date din memoria externă.

*Exemplu:*      *ABSF R1*                          ;  $R1 = |R1|$

Instrucțiunea calculează modulul numărului real din registrul R1, care apoi este memorat tot în registrul R1 ( $|R1| \rightarrow R1$ ).

**2. Adresarea directă** indică faptul că operandul se găsește în memorie, într-o pagină de date, și specifică adresa operandului pe 24 de biți, pentru accesul în mod pagină. Adresa se calculează prin concatenarea celor mai puțin semnificativi 8 biți ai registrului indicator al paginii de date DP, cu cei

mai puțin semnificativi 16 biți ai cuvântului instrucție de 32 de biți, după cum se arată în figura 3.8.

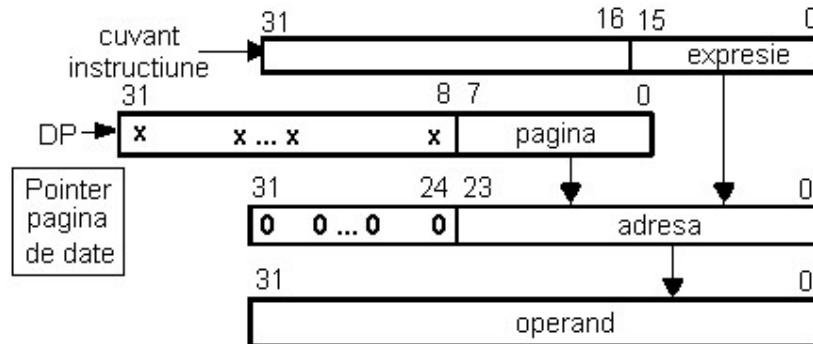


Fig. 3.8. Calculul adresei operandului prin adresarea directă

Procesorul poate accesa un spațiu mare de memorie, de 16 Mcuvinte de 32 de biți, prin schimbarea valorii registrului indicator de pagină DP. Astfel, se pot apela 256 pagini de memorie a către 64 Kcuvinte.

Adresarea directă se indică în câmpul instrucției, utilizându-se următoarea sintaxă: @expr, unde expr reprezintă deplasamentul în interiorul paginii.

*Exemplu:* Instrucția de adunare a două valori întregi, între un operand din memorie și registrul R7, rezultatul depunându-se în R7. Execuția instrucției este ilustrată în figura 3.9.

ADDI @0BCDEh, R7

	Inainte de execuție	Dupa execuție
R7	00 0000 0000	00 1234 5678
DP	8A	8A
Data din memorie	8ABCDEh 1234 5678	8ABCDEh 1234 5678

Fig. 3.9. Exemplu de adresare directă

**3. Adresarea indirectă** specifică adresa unui operand din memorie pe 24 de biți, prin conținutul unui registrul auxiliar, eventual împreună cu un

deplasament, ce poate fi specificat în câmpul instrucțiunii sau printr-un registru de index.

În tabelul 3.4 sunt prezentate variantele utilizate în cazul adresării indirecte. Deplasamentul (*depl*) poate fi dat printr-un număr întreg fără semn reprezentat pe 8 biți și indicat în câmpul instrucțiunii sau prin conținutul regiselor index *IR0* sau *IR1*. Dacă deplasamentul nu este specificat, atunci acesta este considerat implicit 1.

Tab. 3.4. Tipuri de adresare indirectă

Tip adresare indirectă	Operații	Descriere
$*+ARn(depl)$	$adr=(ARn)+depl$	Adresa este calculată prin adunare
$*-ARn(depl)$	$adr=(ARn)-depl$	Adresa este calculată prin scădere
$*++ARn(depl)$	$adr=(ARn)+depl$ $(ARn)=(ARn)+depl$	Adresa se calculează prin adunare, iar apoi se modifică și <i>ARn</i>
$*--ARn(depl)$	$adr=(ARn)-depl$ $(ARn)=(ARn)-depl$	Adresa se calculează prin scădere, iar apoi se modifică și <i>ARn</i>
$*ARn++(depl)$	$adr=(ARn)$ $(ARn)=(ARn)+depl$	Adresa este dată de <i>ARn</i> , iar apoi se modifică <i>ARn</i> prin adunare
$*ARn--(depl)$	$adr=(ARn)$ $(ARn)=(ARn)-depl$	Adresa este dată de <i>ARn</i> , iar apoi se modifică <i>ARn</i> prin scădere
$*ARn++(depl)%$	$adr=(ARn)$ $(ARn)=circ[(ARn)+$ $+depl]$	Adresa este dată de <i>ARn</i> , iar apoi se modifică <i>ARn</i> circular prin adunare
$*ARn--(depl)%$	$adr=(ARn)$ $(ARn)=circ[(ARn)-$ $-depl]$	Adresa este dată de <i>ARn</i> , iar apoi se modifică <i>ARn</i> circular prin scădere

Flexibilitatea adresării indirecte se datorează faptului că unitățile aritmetice ARAU modifică regisrele auxiliare în paralel cu operațiile efectuate de CPU. Tipul de adresare indirectă este specificat de un câmp

de 5 biți ai respectivei instrucții, iar registrele auxiliare ( $AR_n$ ) sunt codificate în instrucție conform reprezentării binare a numărului  $n$ . De exemplu, registrul  $AR_3$  este codificat prin valoarea binară  $011_2$ .

*Exemplu:*                    ABSI \*-AR5(1), R5  
                                   || STI    R1,\*AR2-(IR1)

În acest exemplu, procesorul execută două operații cu numere întregi într-o singură perioadă de ceas, prin adresare indirectă. Astfel, se calculează valoarea absolută a conținutului unei locații de memorie (indicată prin adresare indirectă), care se depune în registrul  $R5$ , apoi se depune registrul  $R1$  în memorie, la o altă adresă specificată tot prin adresare indirectă. În urma execuției instrucției, adresarea indirectă modifică doar conținutul registrului  $AR2$ , după cum se poate observa în figura 3.10.

	Înainte de execuție		Dupa execuție	
R1	00 0000 0042	66	R1 00 0000 0042	66
R5	00 0000 0000		R5 00 0000 0035	53
AR2	80 98FF		AR2 80 98F0	
AR5	80 99E2		AR5 80 99E2	
IR1	0F		IR1 0F	

	Memoria		
8098FF	2	2	8098FF 42 66
8099E1	0FFFFFFFCB	-53	8099E1 0FFFFFFFCB -53

Fig. 3.10. Exemplu de adresare indirectă

**4. Adresarea imediată** se utilizează atunci când valoarea operandului este specificată în câmpul instrucției. În acest caz, operandul reprezintă o valoare imediată pe 16 biți (*short*) sau 24 de biți (*long*), conținută în cei mai puțin semnificativi biți ai cuvântului instrucție. În funcție de tipul de date

cu care lucrează instrucțiunea, operandul poate fi: număr întreg cu semn, întreg fără semn sau număr real în virgulă mobilă.

*Exemplu:* SUBI 1, R0

Inainte de executie	Dupa executie
R0 [ 00 0000 0000 ]	R0 [ 00 FFFF FFFF ]

Fig. 3.11. Exemplu de adresare imediată

**5. Adresarea relativă la registrul PC** este folosită în operațiile de salt, prin care se modifică registrul *PC*. În acest caz, se adună deplasamentul dat de cei 16 sau 24 de biți mai puțin semnificativi ai instrucțiunii, la registrul *PC*. Asamblorul, în funcție de tipul instrucțiunii, calculează deplasamentul după cum urmează:

- dacă operația de salt este de tip standard, deplasamentul este egal cu [*etichetă* – (*adresa instrucțiunii* + 1)];
- dacă operația de salt este de tip întârziat (*delayed branch*), deplasamentul este egal cu [*etichetă* – (*adresa instrucțiunii* + 3)].

Deplasamentul este un număr întreg cu semn pe 16 sau 24 de biți, în cele mai puțin semnificativi biți ai cuvântului instrucțiune. Deplasamentul va fi adunat la registrul *PC* în etapa de decodificare a instrucțiunii. Deoarece registrul *PC* se incrementează în etapa de fetch, deplasamentul se va aduna la noua valoare incrementată a lui *PC*.

Modul de adresare pe 24 de biți este folosit de instrucțiunile de control, ca de exemplu: *BR*, *BRD*, *CALL*, *RPTB* și *RPTBD*. Valoarea bitului 24 al cuvântului instrucțiune determină tipul saltului:  $D = 0$  pentru salt standard și, respectiv,  $D = 1$  pentru salt întârziat.

În funcție de instrucțiunea curentă, noua valoare a registrului *PC* se formează adunând deplasamentul cu valoarea actuală a registrului *PC*, un exemplu fiind indicat în figura 3.12.

*Exemplu:* BNZD 24h

; If Z=0 then 24h+PC+3 → PC  
; else PC+1 → PC

Inainte de executie	Dupa executie
PC [ 0050 ]	PC [ 0077 ]
Z [ 0 ]	Z [ 0 ]

Fig. 3.12. Exemplu de adresare relativă la registrul PC

**6. Adresarea circulară** este utilizată de algoritmii de prelucrare digitală a semnalelor, cum ar fi convoluția și corelația, ce necesită un buffer de memorie circular. Locația curentă în buffer este dată de un indicator (pointer).

În cazul convoluției sau corelației, buffer-ul circular se comportă ca o fereastră alunecătoare, ce conține datele cele mai recent achiziționate din proces. La următoarea achiziție, noua dată este suprascrisă peste cea mai veche dată înregistrată.

Adresa de memorie în care se suprascrie noua dată se obține prin incrementarea indicatorului, în sens contrar acelor de ceasornic. Când pointer-ul indică sfârșitul buffer-ului, acesta este redirecționat către începutul buffer-ului.

Modul de utilizare a unui buffer circular cu 8 locații de memorie este ilustrat în figurile 3.13 și 3.14. Buffer-ul este ilustrat atât printr-o reprezentare logică (circulară), cât și prin cea fizică (locații de memorie). Deoarece lungimea buffer-ului este de 8, a noua valoare a vectorului (val. 8) se va înscrie peste valoarea zero. Procesorul *TMS320C32* permite implementarea mai multor buffere circulare, dar acestea trebuie să aibă aceeași dimensiune.

Pentru implementarea unui buffer circular trebuie specificate următoarele:

- dimensiunea buffer-ului circular pe maxim 16 biți, memorată în registrul *BK*; rezultă că dimensiunea buffer-ului trebuie să fie mai mică sau egală cu  $64\text{ Kcuv}$ ;
- alinierarea adresei de start a buffer-ului cu un câmp de  $K$  biți, ceea ce înseamnă că cei mai puțin semnificativi  $K$  biți ai adresei buffer-ului circular trebuie să fie 0.

Valoarea lui  $K$  se determină astfel încât să fie satisfăcută condiția:  $2^K > R$ , unde  $R$  este dimensiunea buffer-ului circular, iar  $K$  este numărul celor mai puțin semnificativi biți ai adresei buffer-ului circular, poziționați pe zero. În tabelul 3.5 este exemplificată relația dintre adresa de început și lungimea buffer-ului circular.

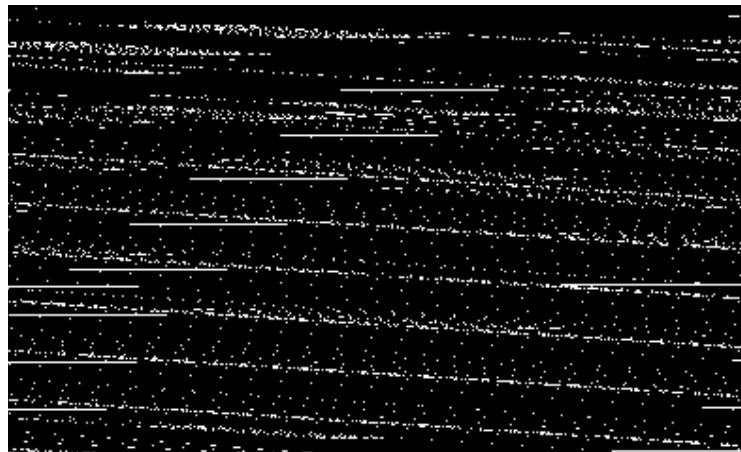


Fig. 3.13. Buffer circular de lungime 8 cu 3 valori înscrise

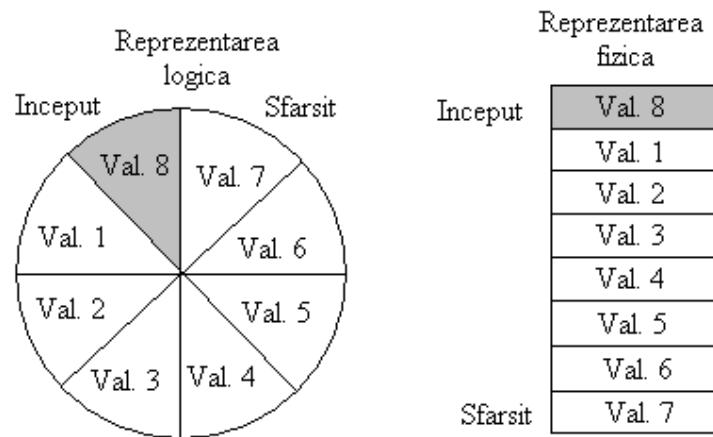


Fig. 3.14. Buffer circular de lungime 8 cu 9 valori înscrise

Pentru implementarea unui buffer circular trebuie specificate următoarele:

- dimensiunea buffer-ului circular pe maxim 16 biți, memorată în registrul  $BK$ ; rezultă că dimensiunea buffer-ului trebuie să fie mai mică sau egală cu  $64\text{ Kcuv}$ ;
- alinierea adresei de start a buffer-ului cu un câmp de  $K$  biți, ceea ce înseamnă că cei mai puțin semnificativi  $K$  biți ai adresei buffer-ului circular trebuie să fie 0.

Valoarea lui  $K$  se determină astfel încât să fie satisfăcută condiția:  $2^K > R$ , unde  $R$  este dimensiunea buffer-ului circular, iar  $K$  este numărul celor mai puțin semnificativi biți ai adresei buffer-ului circular, poziționați pe zero. În tabelul 3.5 este exemplificată relația dintre adresa de început și lungimea buffer-ului circular.

Tab. 3.5. Alinierea adresei de start a buffer-ului circular

Lungimea buffer-ului	Valoarea registrului $BK$	Valoarea lui $K$	Adresa de început a buffer-ului circular
31	31	5	xxxxxxxxxxxxxxxxxxxx00000 <sub>2</sub>
32	32	6	xxxxxxxxxxxxxxxxxxxx000000 <sub>2</sub>
1024	1024	11	xxxxxxxxxxxxxx0000000000 <sub>2</sub>

În cazul adresării circulare, câmpul *index* se referă la numărul  $K$  de biți 0 ai registrului auxiliar utilizat, iar *pasul* reprezintă valoarea care se adună sau se scade din valoarea registrului auxiliar.

Pentru o utilizare corectă a modului de adresare circular, trebuie îndeplinite următoarele condiții:

- *pasul* folosit trebuie să fie mai mic sau egal cu dimensiunea blocului. Pasul este considerat ca fiind un număr întreg fără semn. Dacă se utilizează un registru index (*IR0* sau *IR1*) ca increment sau decrement al pasului, atunci și registru index este tratat ca un întreg fără semn;
- prima dată când se adresează buffer-ul circular de memorie, registrul auxiliar trebuie să indice adresa unui element din buffer.

Algoritmul pentru calculul adresei în cazul adresării circulare este următorul:

- dacă  $0 \leq (\text{index} + \text{pas}) < BK$  atunci  $\text{index} = \text{index} + \text{pas};$
- dacă  $(\text{index} + \text{pas}) \geq BK$  atunci  $\text{index} = \text{index} + \text{pas} - BK;$
- dacă  $(\text{index} + \text{pas}) < 0$  atunci  $\text{index} = \text{index} + \text{pas} + BK.$

**7. Adresarea în ordinea inversă a bițiilor (Bit-Reversed)** este utilizată de algoritmii de prelucrare digitală a semnalelor, cum ar fi transformata Fourier rapidă (*FFT*) și permite accesarea datelor dintr-un buffer, într-o altă ordine decât cea în care sunt datele aranjate în buffer-ul respectiv.

Procesorul *TMS320C32* poate implementa mai rapid ultima etapă din cadrul algoritmului *FFT*, utilizând acest mod de adresare. Atunci când se calculează transformata Fourier rapidă a unei secvențe de date consecutive de lungime  $2^Y$ , rezultatul obținut are valorile plasate în memorie, în ordinea inversă a bițiilor. Pentru a obține vectorul rezultat cu valori ordonate corect, trebuie interschimbate anumite locații de memorie, pierzându-se astfel timp și memorie pentru ordonarea pozițiilor vectorului obținut.

Prin modul de adresare în ordinea inversă a bițiilor, CPU poate accesa direct datele, în ordine inversă a bițiilor, obținându-se vectorul rezultat, fără a mai fi nevoie de ordonarea valorilor și apoi de accesarea secvențială a acestora.

Pentru adresarea corectă în ordinea inversă a bițiilor, baza adresei trebuie să localizeze zona de memorie dată de dimensiunea tabelei transformatei Fourier rapidă. Similar adresării circulare, trebuie îndeplinite condițiile:

- adresa de bază trebuie aliniată cu un număr de  $K$  biți de valoare 0, după aceeași regulă  $2^K > R$ , unde  $R$  este dimensiunea tabelei FFT, iar  $K$  este numărul de biți 0 de pe pozițiile cele mai puțin semnificative ale adresei de început ale tabelei;
- dimensiunea tabelei FFT trebuie să fie mai mică sau egală cu  $64\text{ Kcuv}$  (16 biți).

De exemplu, dacă în memorie există un vector de lungime 16 cuvinte:

$$X = (x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}, x_{13}, x_{14}, x_{15}),$$

atunci accesul la elementele vectorului, utilizând modul de adresare în ordinea inversă a bițiilor, se poate realiza cu o adresă pentru care trebuie inversați doar cei mai puțin semnificativi 4 biți ai acesteia  $(b_3b_2b_1b_0)_2$ . Astfel, valoarea de pe poziția a doua a noului vector ordonat este  $x_8$  de pe poziția de la adresa  $b_3b_2b_1b_0 = 1000_2$ , care s-a obținut prin inversarea bițiilor adresei poziției a doua a lui  $X$ ,  $b_3b_2b_1b_0 = 0001_2$ .



### 3.4. Instrucțiuni de transfer

Limbajul de asamblare al procesorului TMS320C32 conține 13 instrucțiuni de transfer. Operațiile efectuate cu acest set de instrucțiuni sunt:

- încărcarea unui cuvânt din memorie într-un registru (load);
- salvarea unui cuvânt dintr-un registru în memorie (store);
- manipularea datelor din stivă.

**Observații:** trei dintre aceste tipuri de instrucțiuni pot opera condiționat.

În tabelul următor se prezintă instrucțiunile de încărcare și de stocare.

Tab. 3.6. Instrucțiuni de transfer

Instrucțiune	Descriere
<b>LDE</b>	Încarcă exponentul unei valori în virgulă mobilă din memorie într-un registru.
<b>LDF</b>	Încarcă o valoare în virgulă mobilă din memorie într-un registru.
<b>LDF cond</b>	Încarcă , dacă este îndeplinită condiția, o valoare în virgulă mobilă din memorie într-un registru.
<b>LDI</b>	Încarcă o valoare întreagă din memorie într-un registru
<b>LDI cond</b>	Încarcă , dacă este îndeplinită condiția, o valoare întreagă din memorie într-un registru.
<b>STF</b>	Salvează o valoare în virgulă mobilă dintr-un registru în memorie.
<b>STF cond</b>	Salvează, dacă este îndeplinită condiția, o valoare în virgulă mobilă dintr-un registru în memorie.
<b>STI</b>	Salvează o valoare întreagă dintr-un registru în memorie.
<b>STI cond</b>	Salvează, dacă este îndeplinită condiția, o valoare întreagă dintr-un registru în memorie.
<b>POP</b>	Încarcă o valoare întreagă din stiva.
<b>PUSH</b>	Salvează o valoare întreagă în stivă.
<b>POPF</b>	Încarcă o valoare în virgulă mobilă din stivă
<b>PUSHF</b>	Salvează o valoare în virgulă mobilă în stivă.
<b>LDM</b>	Încarcă o mantisă în virgulă mobilă.
<b>LDP</b>	Încarcă pointer-ul cu pagina de date curentă.

### Exemple de instrucții de transfer

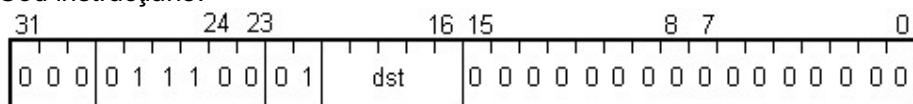
#### Instrucțiuie de refacere a unui registru cu date din memoria stivă -POP

Sintaxa: *POP dst*

Operația efectuată:  $*SP \rightarrow dst$

Indicatorii care pot fi afectați:  $UF=0, N, Z, V=0$

Cod instrucție:



Data de la locația de memorie indicată de adresa conținută în *SP* este transferată în registrul *dst* de dimensiune egală cu 32 de biți. După efectuarea transferului, registrului *SP* se decrementează cu o unitate.

**Exemplu:** *POP R3*

<i>Înainte de execuție</i>	<i>După execuție</i>
R3 [ 00 0000 12DA ] 4,826	R3 [ 00 FFFF ODA4 ] -62,044
SP [ 809856 ]	SP [ 809855 ]
N [ 0 ]	N [ 1 ]

#### *Memoria*

809856h [ FFFF0DA4 ] -62,044    809856h [ FFFF0DA4 ] -62,044

Fig. 3.15. Ilustrarea modului de execuție a instrucției *POP*

În urma execuției instrucției, data (număr întreg cu semn) de la adresa 809856 indicată de *SP* a fost transferată în *R3*, iar conținutul lui *SP* s-a decrementat cu o unitate.

**Instrucțiu de salvare a unei date în memoria stivă -PUSH**

Sintaxa:

*PUSH src*

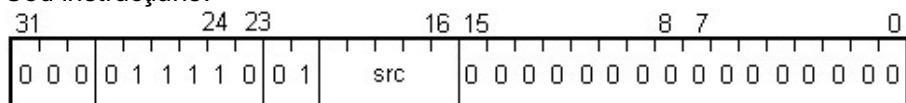
Operația efectuată:

*src → \*++SP*

Indicatorii care pot fi afectați:

*nici un indicator nu este modificat*

Cod instrucție:



După incrementarea adresei conținute în SP, numărul întreg aflat la locația indicată de *src* este transferat la locația din memoria stivă indicată de SP.

**Exemplu:** *PUSH R6*

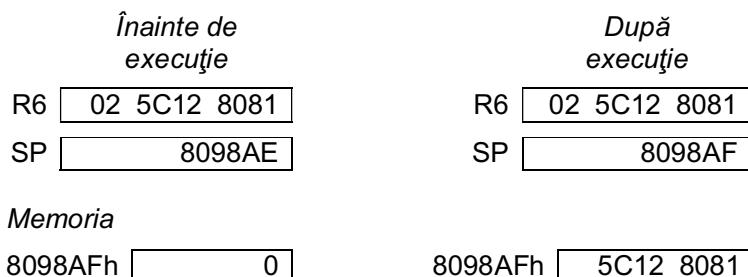


Fig. 3.16. Ilustrarea modului de execuție a instrucției *PUSH*

După execuția instrucției, cei mai puțini semnificativi 32 de biți ai datei aflate în *R6* au fost transferați la locația de memorie  $8098AF_H$  care este indicată de conținutul actual al registrului *SP*.

### **3.5. Instrucții de prelucrare cu doi operanzi**

Procesorul *TMS320C32* suportă 35 de operații aritmetice și logice cu 2 operanzi. Într-o accepție unanim recunoscută, cei 2 operanzi sunt sursa și destinația. Operandul sursă poate fi un cuvânt de memorie, un registru sau o parte din cuvântul instrucție. Operandul destinație este

întotdeauna un registru. Aceste instrucțiuni asigură operații aritmetice și logice cu operanzi întregi, în virgulă mobilă sau în valori logice.

Tab. 3.7. Instrucțiuni de prelucrare cu doi operanzi

Instrucțiune	Descriere
<b>ABSF</b>	Întoarce valoarea absolută a unui număr în virgulă mobilă
<b>NORM</b>	Normalizează o valoare în virgulă mobilă
<b>ABSI</b>	Calculează valoarea absolută a unui întreg
<b>ADDC</b>	Adună doi operanzi întregi utilizând și flag-ul de transport C (Carry)
<b>ADDF</b>	Adună două valori în virgulă mobilă
<b>ADDI</b>	Adună două valori întregi
<b>AND</b>	Și logic
<b>ANDN</b>	Și logic complementat
<b>ASH</b>	Deplasare aritmetică
<b>CMPF</b>	Comparație a două valori în virgulă mobilă
<b>CMPI</b>	Comparație a două valori întregi
<b>FIX</b>	Conversia unei valori din virgulă mobilă în întreg
<b>FLOAT</b>	Conversie din întreg în virgulă mobilă
<b>LSH</b>	Deplasare logică la stânga
<b>MPYF</b>	Înmulțire a două valori în virgulă mobilă
<b>MPYI</b>	Înmulțire a două valori întregi
<b>NEGB</b>	Negarea unei valori întregi cu "împrumut"
<b>NEGF</b>	Negarea unei valori în virgulă mobilă
<b>NEGI</b>	Negarea unei valori întregi
<b>NOT</b>	Negare logică pe biți
<b>OR</b>	Sau logic
<b>RND</b>	Rotunjirea unei valori în virgulă mobilă
<b>ROL</b>	Rotație spre stânga
<b>ROLC</b>	Rotație spre stânga cu transport (carry)
<b>ROR</b>	Rotație spre dreapta
<b>RORC</b>	Rotație spre dreapta cu transport (carry)
<b>SUBB</b>	Scăderea a două valori întregi cu împrumut
<b>SUBC</b>	Scăderea condiționată a două numere întregi
<b>SUBF</b>	Scăderea a două valori în virgulă mobilă

Cap. 3. Setul de instrucțiuni al procesorului de semnal TMS320C32

<b>SUBI</b>	Scăderea a două numere întregi
<b>SUBRB</b>	Scăderea a două numere întregi utilizând biții inversați cu transport
<b>SUBRF</b>	Scăderea a două numere reale utilizând biții inversați
<b>SUBRI</b>	Scăderea a două numere întregi utilizând biții inversați
<b>TSTB</b>	Test pe câmpuri de biți
<b>XOR</b>	Sau exclusiv

### Exemple de instrucțiuni de prelucrare cu 2 operanzi

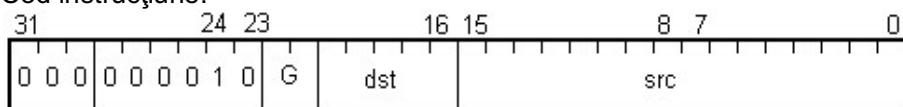
#### Instrucțiune de adunare cu transport a două numere întregi - ADDC

Sintaxa:  $ADDC \ src, dst$

Operația efectuată:  $src + dst + C \rightarrow dst$

Indicatorii care pot fi afectați:  $LV, N, Z, V, C$

Cod instrucțiune:



Instrucțiunea *ADDC* realizează adunarea a două numere întregi specificate prin *src* și *dst*, la care se adaugă și bitul de transport *C* (carry). Rezultatul este încărcat în registrul CPU indicat prin *dst*. Prin *G* (bit<sub>22</sub>, bit<sub>21</sub>) se stabilește modul de adresare, și anume: (00)-adresare la registru; (01)-adresare directă; (10)-adresare indirectă; (11)-adresare imediată.

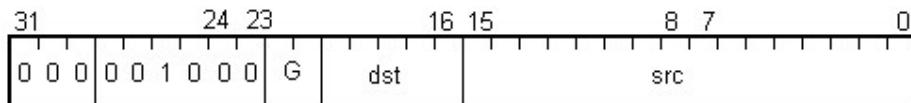
**Exemplu:** *ADDC R1,R5*

<i>Înainte de execuție</i>		<i>După execuție</i>			
R1	00 FFFF 5C25	-41.947	R1	00 FFFF 5C25	-41.947
R5	00 FFFF 019E	-65.122	R5	00 FFFF 5DC4	-107.068
C	1		C	0	

Fig. 3.17. Ilustrarea modului de execuție a instrucțiunii *ADDC*

**Instrucția de comparare a două numere în virgulă mobilă-CMPF**

Sintaxa: *CMPF src, dst*  
 Operația efectuată: *dst-src*  
 Indicatorii care pot fi afectați: *LUF, LV, UF, N, Z, V*  
 Cod instrucție:



Instrucția *CMPF* realizează scăderea a două numere reale specificate prin *src* și *dst* și modifică indicatorii în funcție de rezultat. Numerele aflate la locațiile indicate de *dst* și *src* nu sunt modificate. Prin G (bit<sub>22</sub>, bit<sub>21</sub>) se stabilește modul de adresare, și anume: (00)-adresare la registru; (01)-adresare directă; (10)-adresare indirectă; (11)-adresare imediată.

**Exemplu:** *CMPF \*+AR4, R6*

<i>Înainte de execuție</i>			<i>După execuție</i>		
R6	07 0C80 0000	1.405e+02	R6	07 0C80 0000	1.405e+02
AR4	80 98F2		AR4	80 98F2	
Z	0		Z	1	

*Memoria*

8098F3h 070C8000 1.405e+02    8098F3h 070C8000 1.405e+02

Fig. 3.18. Ilustrarea modului de execuție a instrucției *CMPF*

Efectul vizibil al instrucției apare doar în modificarea indicatorului de zero care devine Z=1 datorită faptului că cele două numere indicate de *\*+AR4* și *R6* sunt egale.

**Instrucțiune de conversie a unui număr real într-un număr întreg-FIX**

Sintaxa:

*FIX src, dst*

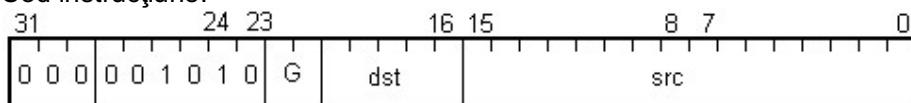
Operația efectuată:

*fix(src) → dst*

Indicatorii care pot fi afectați:

*LV, UF=0, N, Z, V*

Cod instrucțiune:



Instrucțiunea determină partea întreagă a numărului real aflat la locația specificată de *src* și stochează rezultatul la locația indicată de *dst*. Prin *G* (bit<sub>22</sub>, bit<sub>21</sub>) se stabilește modul de adresare, și anume: (00)-adresare la registru; (01)-adresare directă; (10)-adresare indirectă; (11)-adresare imediată.

**Exemplu:** *FIX R1, R2*

<i>Înainte de execuție</i>		<i>După execuție</i>	
R1	0A 2820 0000	1.3454e+3	R1 0A 2820 0000 1.3454e+3
R2	00 0000 0000		R2 00 0000 0541 1345

Fig. 3.19. Ilustrarea modului de execuție a instrucțiunii *FIX*

**Instrucțiune de rotunjire a unui număr real -RND**

Sintaxa:

*RND src, dst*

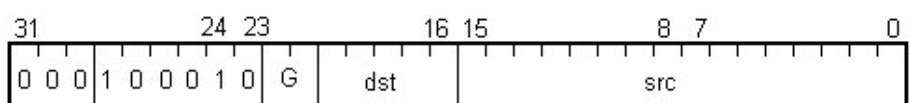
Operația efectuată:

*rotunjire(src) → dst*

Indicatorii care pot fi afectați:

*LUF, LV, UF, N, V*

Cod instrucțiune:



Rezultatul rotunjirii numărului real aflat la locația indicată de *src* este transferat în locația indicată de *dst*. Prin *G* (*bit*<sub>22</sub>, *bit*<sub>21</sub>) se stabilește modul de adresare, și anume: (00)-adresare la registru; (01)-adresare directă; (10)-adresare indirectă; (11)-adresare imediată.

**Exemplu:** *RND R5, R2*

<i>Înainte de execuție</i>	<i>După execuție</i>
R2 <span style="border: 1px solid black; padding: 2px;">00 0000 0000</span>	R2 <span style="border: 1px solid black; padding: 2px;">07 33C1 6F00</span> 179.7556
R5 <span style="border: 1px solid black; padding: 2px;">07 33C1 6EEF</span> 179.755599	R5 <span style="border: 1px solid black; padding: 2px;">07 33C1 6EEF</span> 179.755599

Fig. 3.20. Ilustrarea modului de execuție a instrucției *RND*

Valoarea 179.755599 conținută în R5 este rotunjită la valoarea 179.7556 și apoi transferată în R2.

### 3.6. Instrucții de prelucrare cu 3 operanzi

În comparație cu instrucțiunile pe 2 operanzi care au un singur operand sursă (sau contor de deplasare) și un operand destinație, instrucțiunile pe 3 operanzi pot avea 2 operanzi sursă (sau un operand sursă și un operand contor de deplasare) și un operand destinație. *Operandul sursă poate fi un cuvânt de memorie sau un registru, în timp ce operandul destinație este întotdeauna un registru.*

Tab. 3.8. Instrucții de prelucrare cu trei operanzi

<b>Instrucțiu</b> n	<b>Descriere</b>
<b>ADDC3</b>	Adunare cu transport
<b>ADDI3</b>	Adunarea numerelor în virgulă mobilă
<b>ADDI3</b>	Adunarea numerelor întregi
<b>AND3</b>	Și logic
<b>ANDN3</b>	Și logic negat
<b>ASH3</b>	Deplasare aritmetică
<b>CMPF3</b>	Comparare de numere în virgulă mobilă

<b>CMPI3</b>	Comparare de numere întregi
<b>LSH3</b>	Deplasare logică la stânga
<b>MPYF3</b>	Înmulțirea numerelor în virgulă mobilă
<b>MPYI3</b>	Înmulțirea numerelor întregi
<b>OR 3</b>	Sau logic
<b>SUBB3</b>	Scăderea numelor întregi cu împrumut
<b>SUBF3</b>	Scădere numere în virgulă mobilă
<b>SUBI3</b>	Scădere numere întregi
<b>TSTB3</b>	Test pe câmpuri de biți
<b>XOR3</b>	Sau exclusiv

### Exemple de instrucțiuni de prelucrare cu 3 operanzi

#### Instrucțiunea de deplasare aritmetică - ASH3

Sintaxa: ASH3 count, src, dst

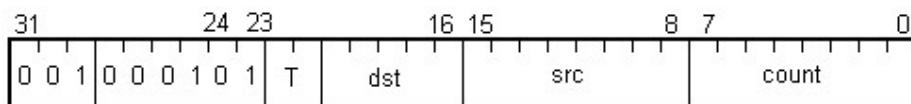
Operația efectuată:

Dacă  $(\underline{\text{count}} \geq 0)$ , atunci numărul specificat de src este deplasat la stânga cu un număr de biți egal cu numărul indicat de count și apoi încărcat în locația indicată de dst;

Dacă  $(\underline{\text{count}} < 0)$ , atunci numărul specificat de src este deplasat la dreapta cu un număr de biți egal cu modulul numărului indicat de count și apoi încărcat în locația indicată de dst;

Indicatorii care pot fi afectați: LV, UF, N, Z, V, C

Cod instrucțiune:



Prin  $T$  ( $\text{bit}_{22}, \text{bit}_{21}$ ) se stabilește modul de adresare, iar cei mai puțin semnificativi biți ai instrucțiunii specifică numărul pozițiilor cu care se deplasează numărul indicat de src.

**Exemplul 1:** ASH3      \*AR3-(1), R5, R0

<i>Înainte de execuție</i>	<i>După execuție</i>
R0 [ 00 0000 0000 ]	R0 [ 00 02B0 0000 ]
R5 [ 00 0000 02B0 ]	R5 [ 00 0000 02B0 ]
AR3 [ 80 9921 ]	AR3 [ 80 9920 ]
<i>Memoria</i>	
809921h [ 10 ] 16	809921h [ 10 ] 16

Fig. 3.21. Ilustrarea modului de execuție a instrucției ASH3 (exemplul 1)

În urma execuției instrucției, numărul întreg cu semn din registrul R5 este deplasat la stânga cu 16 poziții (16 reprezintă numărul aflat la adresa 809921h, adresă specificată prin \*AR3-(1)) și apoi stocat în R0.

**Exemplul 2:** ASH3      R1,R3, R5

<i>Înainte de execuție</i>	<i>După execuție</i>
R1 [ 00 FFFF FFF8 ] -8	R1 [ 00 FFFF FFF8 ] -8
R3 [ 00 FFFF CB00 ]	R3 [ 00 FFFF CB00 ]
R5 [ 00 0000 0000 ]	R5 [ 00 FFFF FFBC ]

Fig. 3.22. Ilustrarea modului de execuție a instrucției ASH3 (exemplul 2)

Data conținută în registrul R3 este deplasată la dreapta cu 8 ( $|R1|$ ) poziții și stocată în registrul R5.

**Instrucțiunea de deplasare logică - LSH3**

Sintaxa:                    *LSH3 count, src, dst*

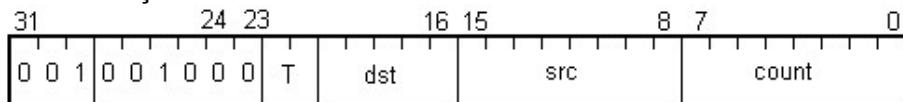
Operația efectuată:

Dacă ( $count \geq 0$ ), atunci numărul specificat de *src* este deplasat la stânga cu un număr de biți egal cu numărul indicat de *count* și apoi încărcat în locația indicată de *dst*;

Dacă ( $count < 0$ ), atunci numărul specificat de  $src$  este deplasat la dreapta cu un număr de biți egal cu modulul numărului indicat de  $count$  și apoi încărcat în locația indicată de  $dst$ ;

Indicatorii care pot fi afectați:  $UF=0, N, Z, V=0$

Cod instrucție:



Prin intermediul biților  $T$  se stabilește modul de adresare, iar cei mai puțin semnificativi biți ai instrucției specifică numărul pozițiilor cu care se deplasează numărul indicat de  $src$ .

**Exemplu:**  $LSH3 \quad R4, R7, R2$

<i>Înainte de execuție</i>		<i>După execuție</i>	
R2	00 0000 0000	R2	02 AC00 0000
R4	00 0000 0018	24	R4 00 0000 0018 24
R7	00 0000 02AC	R7	00 0000 02AC
N	0	N	1

Fig. 3.23. Ilustrarea modului de execuție a instrucției  $LSH3$

În urma execuției instrucției, numărul binar din registrul  $R7$  este deplasat la stânga cu 24 de pozitii (24 reprezintă numărul aflat în  $R4$ ) și apoi stocat în  $R2$ .

### 3.7. Instrucții paralele de transfer și prelucrare

Cele 13 instrucții pentru operații paralele existente în setul de instrucții al procesorului *TMS320C2x* fac posibilă implementarea unui înalt grad de paralelism. Unele dintre instrucțiunile DSP-ului pot fi tratate ca

perechi și se pot executa în paralel. Aceste instrucțiuni permit următoarele operații:

- încărcare paralelă a registrelor;
- operații aritmetice paralele;
- instrucțiuni aritmetice/logice utilizate în paralel cu o instrucțiune de stocare.

Fiecare instrucțiune dintr-o anumită pereche este introdusă ca o instrucțiune separată, iar cea de-a doua instrucțiune din pereche trebuie să fie precedată de două bare verticale ( || ).

### Instrucțiuni de încărcare paralelă

Tab. 3.9. Instrucțiuni de încărcare paralelă

Instrucțiune	Descriere
LDF    LDF	Încărcare valori reale
LDI    LDI	Încărcare valori întregi

### Instrucțiuni dedicate operațiilor aritmetice executate în paralel cu instrucțiuni de stocare

Tab. 3.10. Instrucțiuni paralele (înmulțire și stocare)

Instrucțiune	Descriere
ABSF    STF	Valoarea absolută a unui număr în virgulă mobilă și salvarea unei valori în virgulă mobilă
ABSI    STI	Valoarea absolută a unui întreg și salvarea unui întreg
ADDF3    STF	Adunare a două numere reale și salvarea unui număr real
ADDI3	Adunare a doi întregi și salvarea unui întreg

Cap. 3. Setul de instrucții al procesorului de semnal TMS320C32

<b>   STI</b>	
<b>AND3</b> <b>   STI</b>	ȘI logic și salvarea unui întreg
<b>ASH3</b> <b>   STI</b>	Deplasare aritmetică și salvare a unui întreg
<b>FIX</b> <b>   STI</b>	Conversie din real în întreg și salvarea unui întreg
<b>FLOAT</b> <b>   STF</b>	Conversie din întreg în real și salvarea unei valori reale
<b>LDF</b> <b>   STF</b>	Încarcă o valoare reală și salvează o valoare reală
<b>LDI</b> <b>   STI</b>	Încarcă un întreg și salvează un întreg
<b>LSH3</b> <b>   STI</b>	Deplasare logică și salvarea unui întreg
<b>MPYF3</b> <b>   STF</b>	Înmulțire de numere reale și salvarea unui număr real
<b>MPYI3</b> <b>   STI</b>	Înmulțire de numere întregi și salvarea unui întreg
<b>NEGF</b> <b>   STF</b>	Negarea unei valori reale și salvarea unei valori reale
<b>NEGI</b> <b>   STI</b>	Negarea unui întreg și salvarea unui întreg
<b>NOT</b> <b>   STI</b>	Negarea și salvarea unei valori întregi
<b>OR3</b> <b>   STI</b>	SAU logic (cu 3 operanzi) și salvarea unui întreg
<b>STF</b> <b>   STF</b>	Salvare numere reale (2 operații în paralel)
<b>STI</b> <b>   STI</b>	Salvare numere întregi
<b>SUBF3</b> <b>   STF</b>	Scădere numere reale și salvare numere reale
<b>SUBI3</b> <b>   STI</b>	Scădere întregi și salvare valoare întreagă
<b>XOR3</b> <b>   STI</b>	SAU EXCLUSIV și salvare valoare întreagă

### Instrucțiuni de înmulțire și adunare sau scădere

Tab. 3.11. Instrucțiuni paralele (înmulțire și adunare sau scădere)

Instrucțiune	Descriere
<b>MPYF3</b>    <b>ADDF3</b>	Înmulțire și adunare numere reale
<b>MPYF3</b>    <b>SUBF3</b>	Înmulțire și scădere numere reale
<b>MPYI3</b>    <b>ADDI3</b>	Înmulțire și adunare numere întregi
<b>MPYI3</b>    <b>SUBI3</b>	Înmulțire și scădere numere întregi

### Exemple de instrucțiuni paralele de transfer și prelucrare

#### Instrucțiune paralelă de adunare a două numere reale și stocare - ADDF3 // STF

Sintaxa:  $ADDF3\ src2,src1,dst1$   
 $// STF\ src3,dst2$

Operația efectuată:  $src2+src1 \rightarrow dst1$   
 $// src3 \rightarrow dst2$

Indicatorii care pot fi afectați:  $LUF, LV, UF, N, Z, V, C$

Cod instrucțiune:

31	24	23	16	15	8	7	0
1 1	0 0 1 1 0	dst1	src1	src3	dst2		src2

Instrucțiunea *addf3/stf* realizează în paralel, într-un singur ciclu, atât adunarea a două numere reale, cât și transferul unui număr real indicat de sursa *src3* către destinația *dst2*. Instrucțiunea citește simultan informația indicată de toate sursele la început și o înscrie la sfârșitul ciclului instrucțiune. Dacă *src2* și *dst2* fac referire la aceeași locație, atunci locația *src2* este citită înainte de a fi încărcată *dst2*.

Din codul instrucțiunii rezultă faptul că acestea conțin toate informațiile referitoare la locațiile surselor și destinațiilor.

**Exemplu:**    *ADDF3 \*+AR3(IR1),R2,R5*  
                   || *STF R4,\*AR2*

	<i>Înainte de execuție</i>		<i>După execuție</i>	
R2	07 0C80 0000	1.405e+02	R2 07 0C80 0000	1.405e+02
R4	05 7B40 0000	62.8125	R4 05 7B40 0000	62.8125
R5	00 0000 0000		R5 08 2020 0000	320.25
AR2	80 98F3		AR2 80 98F3	
AR3	80 9800		AR3 80 9800	
IR1	0A5		IR1 0A5	

#### *Memoria*

8098A5h	733C000	179.75	8098A5h	733C000	179.75
8098F3h	0		8098F3h	57B4000	62.8125

Fig. 3.24. Ilustrarea modului de execuție a instrucției *ADDF3 || STF*

După execuția instrucției nu a fost afectat nici un indicator de condiție. În cele ce urmează, se va face referire la acest aspect numai dacă s-a modificat conținutul a cel puțin unui indicator.

#### **Instrucțiunea paralelă *ȘI LOGIC bit cu bit si stocare - AND3 // STI***

Sintaxa:                    *AND3 src2, src1, dst1*  
                               // *STI src3, dst2*

Operația efectuată:      *src2 AND src1 → dst1*  
                               // *src3 → dst2*

Indicatorii care pot fi afectați: *UF=0, N este cel mai semnificativ bit al rezultatului, Z=1 dacă rezultatul este zero, V=0;*

Cod instrucție:

31	24 23	16 15	8 7	0
1 1	0 1 0 0 0	dst1 src1 src3	dst2	src2

Instrucția *and3//sti* realizează în paralel, într-un singur ciclu, atât operația logică și bit cu bit a două numere întregi, cât și transferul unui număr întreg indicat de sursa *src3* către destinația *dst2*. Instrucția citește simultan informația indicată de toate sursele la început și o înscrive la sfârșitul ciclului. Ca și în cazul instrucției anterioare, dacă *src2* și *dst2* face referire la aceeași locație, atunci locația *src2* este citită înainte de a fi încărcată *dst2*.

**Exemplu:** AND3 \*+AR1(IR0),R4,R7  
|| STI R3,\*AR2

	<i>Înainte de execuție</i>		<i>După execuție</i>
IR0	00 0000 0008		IR0 00 0000 0008
R3	00 0000 0035	53	R3 00 0000 0035
R4	00 0000 A323		R4 00 0000 A323
R7	00 0000 0000		R7 00 0000 0003
AR1	80 99F1		AR1 80 99F1
AR2	80 983F		AR2 80 983F

#### Memoria

8099F9h	5C53		8099F9h	5C53
80983Fh	0		80983Fh	35 53

Fig. 3.25. Ilustrarea modului de execuție a instrucției AND3 || STI

#### Instrucție paralelă de conversie a unui număr întreg într-un număr real reprezentat în virgulă mobilă și stocare - FLOAT // STF

Sintaxa: *FLOAT src1, dst1*

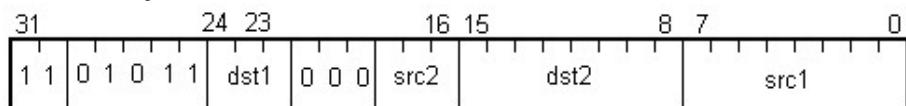
// *STF src2, dst2*

Operația efectuată: *float(src1) → dst1*

// *src2 → dst2*

Indicatorii care pot fi afectați: *UF=0, N, Z, V=0*

Cod instrucțiune:



Instrucțiunea FLOAT//STF realizează în paralel, într-un singur ciclu, conversia unui număr întreg într-un număr real reprezentat în virgulă mobilă și transferă un număr real indicat de sursa *src2* către destinația *dst2*. Instrucțiunea citește simultan informația indicată de cele două surse (*src1*, *src2*) la început și o înscrive la sfârșitul ciclului. Dacă *src1* și *dst2* fac referire la aceeași locație, atunci locația *src1* este citită înainte de a fi încărcată *dst2*.

**Exemplu:**    *FLOAT    \*+AR2(IR0),R6*  
                  || STF    *R7,\*AR1*

<i>Înainte de execuție</i>	<i>După execuție</i>		
R6 [00 0000 0000]	R6	[07 2E00 0000]	1.740e+02
R7 [03 4C20 0000]	R7	[03 4C20 0000]	12.7578125
AR1 [80 9933]	AR1	[80 9933]	
AR2 [80 98C5]	AR2	[80 98C5]	
IR0 [8]	IR0	[8]	

#### *Memoria*

8098CDh	[0AE]	174	8098CDh	[0AE]	174
809933h	[0]		809933h	[034C2000]	12.7578125

Fig. 3.26. Ilustrarea modului de execuție a instrucțiunii *FLOAT || STF*

#### **Instrucțiune paralelă de înmulțire și scădere numere reale - MPYF3||SUBF3**

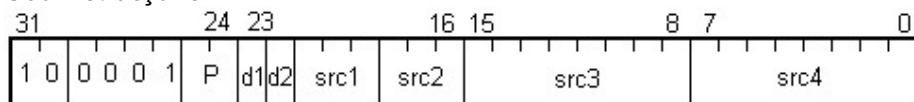
Sintaxa:              *MPYF3 src1, src2, d1*  
                          || *SUBF3 src3, src4, d2*

Operația efectuată:  $src1 \times src2 \rightarrow d1$

$\parallel src4 - src3 \rightarrow d2$

Indicatorii care pot fi afectați: LUF, LV, UF, N, Z, V

Cod instrucție:



Instrucția  $MPYF3||SUBF3$  realizează în paralel, într-un singur ciclu, atât înmulțirea a două numere reale ( $src1 \times src2$ ), cât și scăderea a altor două numere reale ( $src4 - src3$ ). Instrucția citește simultan informația indicată de toate sursele la început și o înscrive la sfârșitul ciclului.

**Exemplu:**  $MPYF3 R5, *++AR7(IR1), R0$   
 $\parallel SUBF3 R7, *AR3-(1), R2$

sau

$MPYF3 *++AR7(IR1), R5, R0$   
 $\parallel SUBF3 R7, *AR3-(1), R2$

	<i>Înainte de execuție</i>		<i>După execuție</i>
R0	00 0000 0000	R0	04 6718 0000 28.8867188
R2	00 0000 0000	R2	05 E300 0000 -3.925e+01
R5	03 4C00 0000	R5	03 4C00 0000 1.275e+01
R7	07 33C0 0000	R7	07 33C0 0000 1.7975e+2
AR3	80 98B2	AR3	80 98B2
AR7	80 9904	AR7	80 9904
IR1	8	IR1	8

#### Memoria

80990Ch	1110000	2.25e+00	80990Ch	1110000	2.25e+00
8098B2h	70C8000	1.405e+02	8098B2h	70C8000	1.405e+02

Fig. 3.27. Ilustrarea modului de execuție a instrucției  $MPYF3 || SUBF3$

În cazul instrucțiunilor paralele, două dintre sursele *src1*, *src2*, *src3*, *src4* trebuie să fie registre, iar celelalte două trebuie să fie locații de memorie. Această restricție se datorează faptului că DSP-ul dispune doar de două magistrale de adrese pentru registre și de două magistrale de adrese pentru memorie.

**Instrucție paralelă de înmulțire și scădere numere întregi - MPYI3||SUBI3**

Sintaxa:

*MPYI3 src1, src2, d1*  
|| *SUBI3 src3, src4, d2*

Operația efectuată:

*src1 x src2 → d1*  
|| *src4 – src3 → d2*

Indicatorii care pot fi afectați:

*LV, UF, N, Z, V*

Cod instrucție:

31	24	23	24	23	16	15	8	7	0		
1	0	0	0	1	1	P	d1d2	src1	src2	src3	src4

Instrucția *mpyi3||subi3* realizează în paralel, într-un singur ciclu, atât înmulțirea a două numere întregi (*src1 x src2*), cât și scăderea a altor două numere întregi (*src4 – src3*). Instrucția citește simultan informația indicată de toate sursele la începutul ciclului și o înscrive la sfârșitul ciclului.

**Exemplu:**    *MPYI3 R2,\*++AR0(1),R0*  
                   || *SUBI3 \*AR5—(IR1),R4,R2*  
                   sau  
                   *MPYI3 \*++AR0(1),R2,R0*  
                   || *SUBI3 \*AR5—(IR1),R4,R2*

<i>Înainte de execuție</i>			<i>După execuție</i>		
R0	00 0000 0008		R0	00 0000 1324	4900
R2	00 0000 0032	50	R2	00 0000 0320	800
R4	00 0000 07D0	2000	R4	00 0000 07D0	2000
AR0	80 98E3		AR0	80 98E4	
AR5	80 99FC		AR5	80 99F0	
IR1	0C		IR1		0C

*Memoria*

8098E4h	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>62</td></tr></table>	62	98	8098E4h	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>62</td></tr></table>	62	98
62							
62							
8099FCh	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>4B0</td></tr></table>	4B0	1200	8099FCh	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>4B0</td></tr></table>	4B0	1200
4B0							
4B0							

Fig. 3.28. Ilustrarea modului de execuție a instrucțiunii *MPYI3 || SUBI3***3.8. Instrucțiuni de control**

Instrucțiunile de control sunt acele instrucțiuni care afectează derularea programului în sensul că, după executarea instrucțiunii aflate la adresa *PC*, nu se mai execută instrucțiunea aflată la adresa (*PC+1*). Există posibilitatea repetării unui bloc de instrucțiuni cu ajutorul instrucțiunilor dedicate: *RPTB* pentru un bloc de instrucțiuni și, respectiv, *RPTS* pentru o singură linie de cod. Este suportată atât varianta standard, cât și cea cu întârziere (cu un singur ciclu) a comenzi de salt condiționat. De asemenea, există instrucțiuni de control care pot efectua operații condiționate.

Tab. 3.12. Instrucțiuni de control

Instrucțiune	Descriere
<b>Bcond</b>	Salt condiționat
<b>B condD</b>	Salt condiționat întârziat
<b>BR</b>	Salt necondiționat (standard)
<b>BRD</b>	Salt necondiționat întârziat
<b>CALL</b>	Apel de subrutină
<b>CALL cond</b>	Apel de subrutină condiționat
<b>DB cond</b>	Decrementare și salt condiționat (standard)
<b>DB condD</b>	Decrementare și salt condiționat întârziat
<b>IDLE</b>	Stop până la apariția unei întreruperi
<b>NOP</b>	Ciclu instrucțiune fără operație
<b>RETI cond</b>	Întoarcere condiționată din rutina de întrerupere
<b>RETS cond</b>	Întoarcere condiționată din subrutină
<b>RPTB</b>	Repetare bloc instrucțiuni

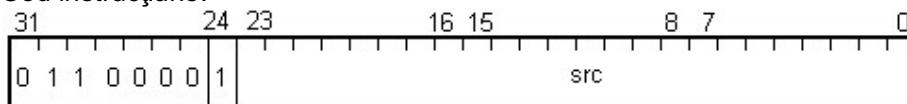
<b>RPTS</b>	Repetare instrucțiune
<b>SWI</b>	Întrerupere software
<b>TRAP</b>	Oprire în vederea depanării
<b>IACK</b>	Acceptare întrerupere

### Exemple de instrucțiuni de control

#### Instrucțiunea de salt necondiționat - BRD

Sintaxa:  $BRD \ src$   
 Operația efectuată:  $src \rightarrow PC$   
 Indicatorii care pot fi afectați: *nici un indicator nu este modificat*

Cod instrucțiune:



Instrucțiunea realizează saltul necondiționat la adresa indicată de *src*. Prin intermediu bitului 24 se poate realiza un salt necondiționat întârziat cu trei instrucțiuni, dacă valoarea acestuia este 1.

**Exemplu:**  $BRD \ 2Ch$



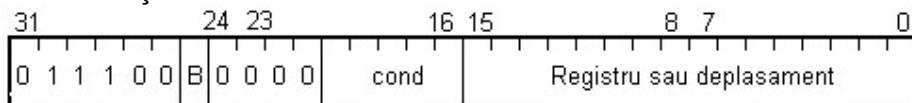
Fig. 3.29. Illustrarea modului de execuție a instrucțiunii *BRD*

#### Instrucțiunea de apel condiționat – CALLcond

Sintaxa:  $CALLcond \ src$   
 Operația efectuată:  
 Dacă expresia *cond* este adevărată, atunci *PC-ul* instrucțiunii imediat următoare se salvează în stivă și numărul specificat de *src* este trecut în *PC* ( $src \rightarrow PC$ )

Indicatorii care pot fi afectați: *nici un indicator nu este modificat*

Cod instrucțiiune:



Prin intermediul *bitului 25* se specificază modul de adresare relativă ( $B=1$ ) sau modul de adresare la registru ( $B=0$ ). În funcție de modul de adresare, cei mai puțin semnificativi biți indică deplasamentul sau registrul care conține adresa la care se face saltul. Condiția se setează în urma execuției unei instrucții de comparare care este efectuată înaintea execuției instrucționii CALL.

**Exemplu:** CALLNZ R5

<i>Înainte de execuție</i>		<i>După execuție</i>	
R5	00 0000 0789	R5	00 0000 0789
PC	0123	PC	0789
SP	809835	SP	809836
Z	0	Z	0

*Memoria*

809836h 124

Fig. 3.30. Ilustrarea modului de execuție a instrucționii CALLNZ

În acest caz, PC-ul instrucționii imediat următoare este 124.

#### **Instrucție de revenire condiționată din intrerupere – RETIcond**

Sintaxa:

*RETIcond*

Operația efectuată:

*Dacă cond este adevărată atunci*

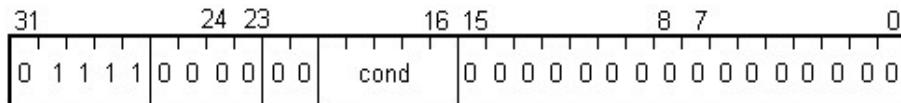
*\*SP-- → PC*

*1 → ST(GIE)*

*altfel*

*continuă*

Indicatorii care pot fi afectați: *nici un indicator nu este modificat*  
Cod instrucțione:



Data de la locația de memorie indicată de adresa conținută în *sp* este transferată în registrul *pc* numai dacă condiția este adevărată. În caz contrar, se continuă execuția programului. După efectuarea transferului, registrul *sp* se decrementează cu o unitate, iar bitul corespunzător întreruperilor este setat cu 1 (întreruperile sunt din nou validate).

**Exemplu:** *RETINZ*

<i>Înainte de execuție</i>	<i>După execuție</i>
PC      0456	PC      0123
SP      809830	SP      80982F
ST      0	ST      2000

#### *Memoria*

809830h	123	809830h	123
---------	-----	---------	-----

Fig. 3.31. Ilustrarea modului de execuție a instrucției *RETINZ*

După execuția instrucției, se reface *PC* cu adresa aflată la adresa 809830h, iar bitul *G/E* al registrului *ST* se setează cu 1.

#### **Instrucțiunea de repetare a unei singure instrucții - RPTS**

Sintaxa:

*RPTS src*

Operația efectuată:

*src → RC*

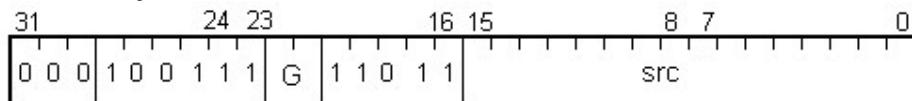
*1 → ST(RM)*

*1 → S*

*următorul PC → RS și RE*

Indicatorii care pot fi afectați: *nici un indicator nu este modificat*

Cod instrucțune:



Instrucțunea permite repetarea unei alte instrucțuni de un număr de ori mai mare cu unu decât numărul indicat de *src*, număr transferat în prealabil în registrul RC.

**Exemplu:** RPTS AR5

	<i>Înainte de execuție</i>		<i>După execuție</i>
AR5	00 00FF		00 00FF
PC	0123		0124
RC	0		OFF
RE	0		124
RS	0		124
ST	0		100

Fig. 3.32. Ilustrarea modului de execuție a instrucției RPTS

#### Instrucțunea de tratare a unei intreruperi provocate și conditionate - TRAPcond

Sintaxa:

*TRAPcond N*

Operația efectuată:

$0 \rightarrow ST(GIE)$

*Dacă cond este adevărată atunci*

*următorul PC  $\rightarrow *++SP$*

*vectorul TRAP N  $\rightarrow PC$*

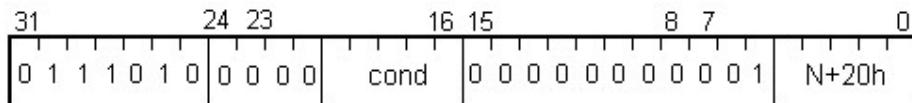
*altfel*

*setează ST(GIE) la valoarea inițială și continuă*

Dacă expresia *cond* este adevărată, atunci se tratează întreruperea provocată prin intermediul instrucției TRAPcond. Procedura de tratare a întreruperii este cea cunoscută, și anume: se salvează PC-ul instrucției imediat următoare în stivă și numărul specificat de TRAP N este trecut în PC. Trebuie specificat faptul că N este un număr cuprins între 0 și 31.

Indicatorii care pot fi afectați: *nici un indicator nu este modificat*

Cod instrucție:



Instrucția dezactivează sistemul de întreruperi prin setarea bitului GIE al registrului de stare – ST ( $0 \rightarrow ST(GIE)$ ) și apoi face saltul la instrucția de la adresa aflată la locația de memorie TRAP N.

**Exemplu:** TRAPZ 16

<i>Înainte de execuție</i>		<i>După execuție</i>	
PC	0123	PC	0010
SP	809870	SP	809871
ST	0	ST	0
Z	0	Z	0

#### *Memoria*

Trap V.16 [ ] 10                            809871h [ ] 124

Fig. 3.33. Ilustrarea modului de execuție a instrucției TRAPZ

În acest caz, PC-ul instrucției imediat următoare este 124.

### 3.9. Instrucțiuni pentru funcționarea în regim de consum redus

Acest grup de instrucțiuni de control constă dintr-un număr de 3 instrucțiuni care afectează modul de funcționare în regim de consum redus. Instrucțiunea **IDLE2** permite funcționarea în regim de consum extrem de redus. Instrucțiunea de divizare a frecvenței de tact prin 16 - **LOOPPOWER** reduce frecvența ceasului de intrare. Instrucțiunea de revenire la frecvența normală **MAXSPEED** determină reluarea execuției operațiilor la viteza maximă.

### 3.10. Instrucțiuni cu interblocare

Cele cinci instrucțiuni din acest grup permit comunicarea multi-procesor și utilizarea de semnale externe, implementând un puternic sistem de sincronizare. Acestea asigură, de asemenea, integritatea comunicației permitând operații foarte rapide.

Tab. 3.13. Instrucțiuni cu interblocare

Instrucțiune	Descriere
<b>LDFI</b>	Încarcă o valoare în virgulă mobilă într-un registru, cu interblocare
<b>LDII</b>	Încarcă o valoare întreagă, cu interblocare
<b>STFI</b>	Salvează în memorie o valoare în virgulă mobilă, cu interblocare
<b>STII</b>	Salvează în memorie o valoare întreagă, cu interblocare
<b>SIGI</b>	Semnal interblocare

### 3.11. Instrucțiuni ilegale

Procesorul *TMS320C32* nu are un mecanism de detecție a instrucțiunilor ilegale. Încărcarea unui cod instrucțiune incorrect duce la execuția unei operații nedefinite. Utilizarea corectă a instrumentelor

### Cap. 3. Setul de instrucționi al procesorului de semnal TMS320C32

software asigură generarea de cod lipsit de instrucționi ilegale. Operații ilegale se pot obține numai în următoarele condiții:

- utilizarea incorectă a instrumentelor software;
- erori în codul aflat în memoria ROM;
- erori în memoria RAM.

## **CAPITOLUL 4**

### **SISTEMUL DE DEZVOLTARE DSProto 32 CU PROCESORUL DE SEMNAL TMS 320C32**

#### **4.1. Prezentare generală**

Pentru dezvoltarea unor aplicații cu ajutorul procesorului de semnal TMS320c32 s-a utilizat o platformă dedicată domeniului audio. Platforma numită DSProto32 a fost achiziționată de la firma Dicon Lab din Florida și permite implementarea aplicațiilor privind prelucrarea semnalului vocal în vederea obținerii unor efecte speciale și, de asemenea, se poate utiliza și în aplicații de recunoaștere a semnalului vocal.

Platforma de dezvoltare DSProto32 are următoarele caracteristici (figura 4.1):

- este echipată cu procesorul de semnal TMS320c32 care este cel mai performant procesor din familia TMS320c3x;
- frecvența procesorului este de 50 Mhz ceea ce permite, de exemplu, executarea unei operații paralele de înmulțire și de adunare a unor numere reale într-un singur ciclu mașină;
- conține o memorie SRAM având o capacitate de 128kx32biti;
- are o memorie EPROM de 64kx8biti;

- dispune de un circuit specializat cs4218 (codec audio stereo), utilizat la achiziția semnalului audio și la refacerea acestuia;
- are un port paralel necesar comunicării cu un calculator PC. Dezvoltarea aplicațiilor pe această platformă presupune o legătura online cu un PC, legătură realizată prin intermediul portului paralel;
- conține, evident, și circuite pentru decodificarea spațiului de adresare.

Pentru a permite dezvoltarea hardware, platforma DSProto32 are doi conectori care conțin magistralele de date, adrese și comenzi. Aceștia pot fi utilizați și la adăugarea de noi dispozitive electronice.

Placa de dezvoltare este însotită de un soft specializat care facilitează dezvoltarea de programe scrise în asamblare sau într-un limbaj de nivel înalt, cum ar fi limbajul C. Pe lângă asamblorul și linkeditorul de programe, placa de dezvoltare mai este însotită și de programe utilitare ce permit transferul de informații din memoria placii pe hardul calculatorului și invers. Aceste programe sunt foarte utile în situația în care trebuie să se memoreze înregistrări martor, obținute prin intermediul sistemului de achiziție disponibil pe placa de dezvoltare.

Pe lângă cele menționate, cei de la *Dicon Lab* mai pun la dispoziția utilizatorilor și o bibliotecă matematică scrisă în limbajul de asamblare al procesorului de semnal TMS320c32. Printre cele mai importante programe se numără și funcțiile de calcul ale logaritmului natural, ale sinusului, ale cosinusului, ale arctangentei etc și, de asemenea, funcțiile dedicate operațiilor cu matrici (adunare, înmulțire). Având în vedere faptul că procesorul de semnal TMS320c32 este utilizat în aplicații care necesită un volum mare de calcul (analiza spectrală, analiza de corelație), placa de dezvoltare este însotită și de un program de calcul al transformatei Fourier rapide, care facilitează implementarea aplicațiilor din domeniul recunoașterii semnalului vocal.

În concluzie, trebuie subliniat următorul aspect: disponând de aceste instrumente, atât hard cât și soft, și de un set puternic de instrucțiuni, implementarea unei aplicații complexe se realizează fără un efort deosebit, chiar dacă programul se scrie în limbajul de asamblare al procesorului de semnal TMS320c32.

Pentru a pune în evidență oportunitățile oferite de platforma DSProto32, în capitolul următor se prezintă succint și circuitul specializat CS4218, dedicat conversiei semnalului analogic în semnal numeric și invers.

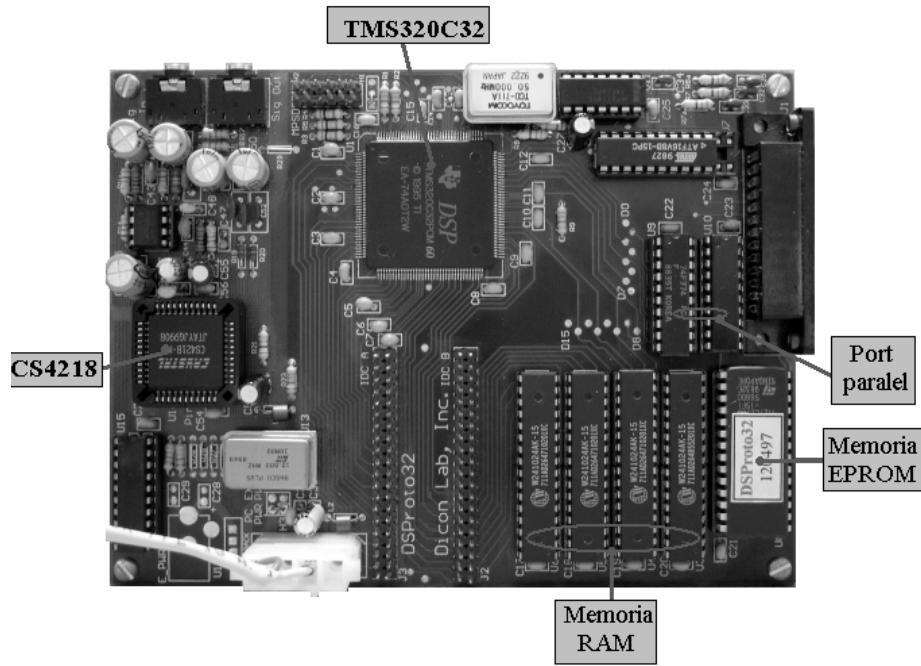


Fig. 4.1. Platforma DSProto32 realizată cu procesorul TMS320c32

## 4.2. Circuitul programabil specializat CS4218 (codec audio stereo)

Circuitul CS4218 este realizat în tehnologie CMOS și utilizat în domeniul procesoarelor multimedia și în aplicații audio dedicate. Acest circuit conține câte 2 convertoare A/D și D/A performante pe 16 biți, având o rată de conversie de până la 54kHz pentru fiecare canal. Circuitul dispune de 4 intrări audio, de 2 ieșiri audio și de filtre digitale programabile, atât pentru semnalul de intrare, cât și pentru cel de ieșire. Transferul de date între circuit CS4218 și DSP se realizează prin intermediul interfeței seriale. Schema bloc structurală a circuitului ce pune în evidență elementele componente ale acestuia este dată în figura următoare.

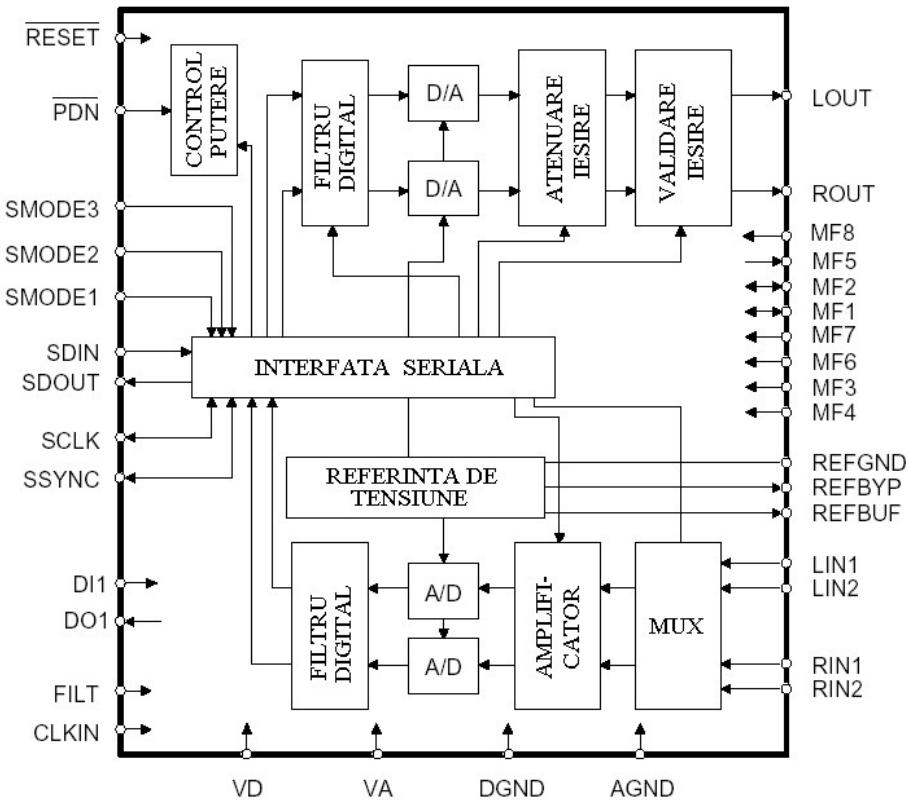


Fig. 4.2. Circuitul specializat CS4218 (Codec audio stereo)

Semnificația pinilor externi circuitului, într-o prezentare succintă, este:

- pini pentru alimentarea, atât a structurilor digitale, cât și a structurilor analogice: **VD**, **DGND**, **VA**, **AGND**;
- intrări analogice: **LIN1**, **LIN2**, **RIN1**, **RIN2**;
- ieșiri analogice: **ROUT**, **LOUT**, **REFBYP**, **REFGND**, **REFBUF**;
- semnale pentru comunicația cu interfața serială: **SDIN**, **SDOUT**, **SCLK**, **SSYNC**, **SMODE1**, **SMODE2**, **SMODE3**;

- pini suplimentari cu funcții diferite care depind de modul de lucru al circuitului;
- pin de reset: *RESET*;
- pin de clock general: *CLKIN*;
- intrare și ieșire digitală: *DI1 și DO1*.

Placa de dezvoltare DSProto32 conține subroutines dedicate circuitului CS4218, astfel că, realizarea de aplicații cu acesta nu necesită un efort considerabil. De altfel, politica companiei *Texas Instruments* este de a pune la dispoziția utilizatorilor toate facilitățile, astfel încât, cât mai multe aplicații să se realizeze cu ajutorul procesoarelor TMS320xxx. Acest aspect este foarte important, dacă se are în vedere faptul că un procesor de semnal presupune un efort considerabil pentru a putea fi asimilat și utilizat.

### **4.3. Componenta software pentru dezvoltarea de aplicații**

Realizarea unei aplicații și rularea acesteia pe platforma DSProto32 necesită într-o ultimă etapă și testarea programului, verificând astfel dacă prescripțiile de proiectare sunt respectate. Acest demers privind corectarea și verificarea programului presupune un soft specializat care să ofere instrumente de încărcare a programului în memoria SRAM a platformei DSProto32 și de lansare în execuție a programului realizat. Firma DiCon Lab livrează platforma de dezvoltare împreună cu un soft dedicat DSProto32 Debugger care oferă, pe lângă cele menționate, și alte facilități, specifice utilitărelor de depanare, privind citirea memoriei SRAM, scrierea unei zone de memorie etc.

Interfața programului de lucru cu platforma DSProto32 este sugestivă și oferă o manieră lejeră de operare cu procesorul *TMS320C32*. În figura următoare, se prezintă interfața corespunzătoare programului DSProto32 Debugger care ilustrează și modul de operare.

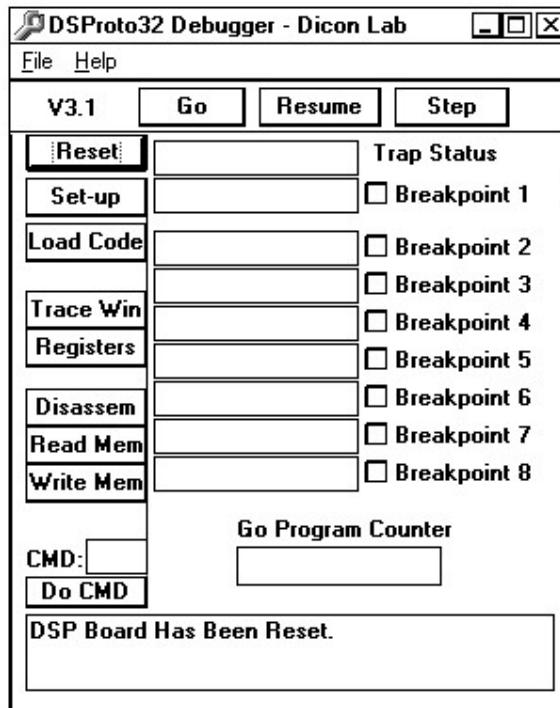


Fig. 4.3. Interfața de lucru cu programul DSProto32 Debugger

Prin intermediul interfeței prezentate în figura 4.3 se pot stabili următoarele comenzi de operare:

- *Reset* platformă DSProto32. Calculatorul la care este conectată platforma transmite prin intermediul portului paralel un semnal de inițializare (PC INIT) ce are drept scop trecerea ieșirii Q a unui bistabil de tip *D* în starea *low*, ieșire conectată, la rândul ei, la pinul de reset al procesorului TMS320C32. La inițializare, se execută programul din memoria boot EPROM care, pe lângă faptul că realizează o configurare a resurselor interne, interpretează și comenziile primite de la calculator prin intermediul portului paralel;
- *Set-up* stabilește directorul de lucru și numele fișierului care se încarcă pentru a fi executat (figura 4.4a);
- *Load Code* încarcă programul obținut în urma asamblării și linkeditării, în memoria SRAM, în vederea executării acestuia;

- *Registers* permite citirea regiszrelor în cazul în care s-a întrerupt programul din execuție;
- *Go* lansează în execuție programul încărcat la adresa specificată prin comanda *Set-up*;
- *Read Mem și Write Mem* (figura 4.4b) permit citirea unei zone de memorie sau înscrierea unui bloc de memorie cu o valoare specificată;
- *Disassem* dezasamblează programul de la o anumită locație de mem;
- *CMD și Do CMD* sunt utilizate pentru adăugarea de noi comenzi și pentru lansarea acestora în execuție;
- *Trace Win* utilizată pentru a genera oprirea programului în cazul când contorul de program are o valoare specificată apriori;
- *Break point* puncte de oprire;
- *Step* relansarea în execuție a programului după o oprire;
- *Resume* salvarea contextului programului (în cazul când s-a ajuns într-un punct de oprire) necesar reînceperii execuției programului.

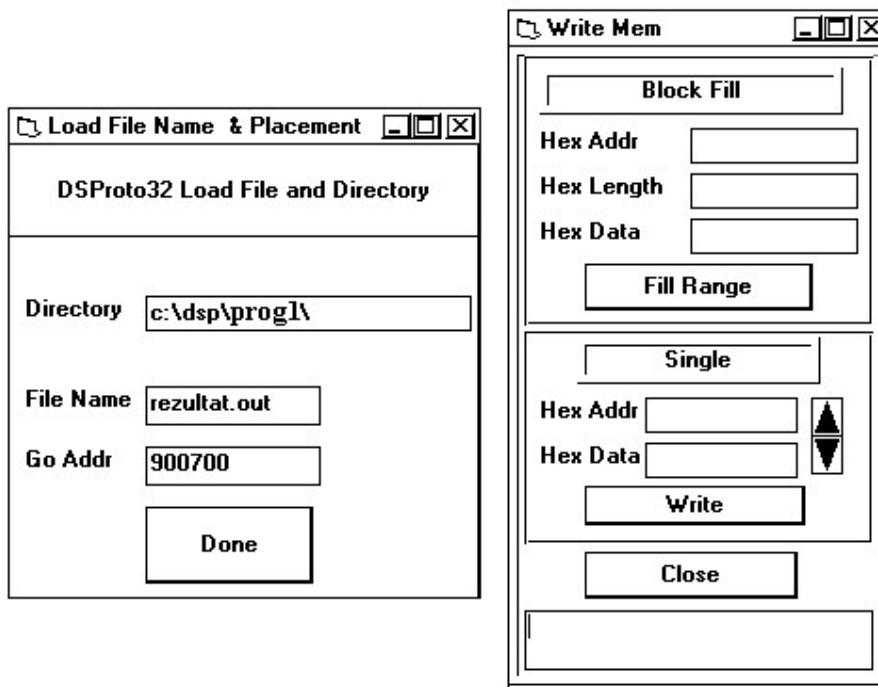


Fig. 4.4a  
Fig. 4.4b  
Ferestre de lucru ale interfeței programului DSProto32 Debugger

**Observație.** Ilustrarea unor comenzi cum ar fi *Read Mem*, *Write Mem* și *Disassem* se va face în capitolul următor, în secțiunea în care se prezintă programele scrise în asamblare.

Trebuie subliniat faptul că, harta memoriei SRAM de pe placa de dezvoltare are următoarea configurație:

- *900000-9000ff tabela vectorilor de intrerupere;*
- *900100-9004ff o copie a programului boot, utilizată în cazul resetării prin acționarea butonului de reset a interfeței reprezentate în figura 4.3;*
- *900500-9005ff memoria stivă;*
- *900600-90067f rezervat pentru variabilele utilizate în cazul opririi programului;*
- *900680-9006ff rezervat pentru variabilele programului de generare a unor efecte speciale.*

**Observație:** Aplicațiile prezentate în această lucrare utilizează spațiul de memorie de la adresa 900700, până la adresa 91FFFF.

#### 4.4. Programarea în limbaj de asamblare

Programarea într-un limbaj de asamblare este strâns corelată cu modul de operare al unității de procesare utilizată, necesitând și cunoștințe privind hard-ul platformei de lucru. Limbajul de asamblare presupune o gestionare riguroasă a resurselor procesorului, comparativ cu majoritatea limbajelor de nivel înalt ce nu au acces direct la resursele hard ale platformei de calcul.

Diferențele care există între un limbaj de asamblare și un limbaj de nivel înalt se pot concretiza atât în avantaje, cât și în dezavantaje:

- limbajul de asamblare permite accesul la toate resursele unui microprocesor, dând posibilitatea utilizatorului să realizeze aplicații care să fie performante atât din punct de vedere al spațiului de memorie folosit, cât și din punct de vedere al vitezei de execuție;

- programarea necesită o organizare riguroasă, deoarece manipularea variabilelor de lucru presupune și cunoașterea locației de memorie la care se află acele variabile.
- un alt aspect care trebuie subliniat se referă la faptul că, dacă în cazul unui limbaj de nivel înalt adresarea unui element dintr-un vector se face prin nume și indice, în cazul limbajului de asamblare, aceasta se face printr-o adresare relativă ce presupune și operații de determinare a adresei respective. Registrele utilizate la adresarea respectivului element nu trebuie folosite și de alte instrucțiuni sau, dacă sunt folosite, ele trebuie salvate la acel moment, ca apoi să poată fi refăcute. Toate aceste operații necesită o abordare riguroasă și coerentă în ceea ce privește realizarea programelor într-un limbaj de asamblare. Abordarea impune ca programatorul să cunoască starea tuturor resurselor procesorului (registrele acestuia, alocarea memoriei, sistemul de întreruperi etc) și implicațiile modificării acestor resurse asupra bunei funcționări a întregului sistem.

Pornind de la considerentele menționate mai sus, programul asambilor *TMS320C3X Assembler* oferă mecanisme prin care se pot realiza programe complexe fără a depune un efort considerabil. Această afirmație este susținută de faptul că procesorul *TMS320C32* dispune de un set de instrucțiuni foarte puternic (înmulțirea în paralel cu adunarea numerelor reale, instrucțiuni repetitive etc) și de faptul că limbajul de asamblare oferă mecanisme facile privind partajarea spațiului de memorie, setarea constantelor și a elementelor unui vector, utilizarea macro-urilor etc.

Realizarea și rularea unui program în limbajul de asamblare al procesorului *TMS320C32*, presupune parcurgerea următoarelor etape:

- conceperea algoritmului;
- scrierea programului în limbajul de asamblare corespunzător procesorului *TMS320C32*;
- translatarea fișierului text ce conține programul sursă, în fișiere obiect relocabile, utilizând *TMS320C3X Assembler*;

- linkeditarea fișierelor obiect într-un singur fișier obiect executabil prin reevaluarea, atât a legăturilor interne, cât și a celor externe. Această etapă de lucru se realizează cu ajutorul unui program numit linkeditor;
- încărcarea programului în memoria SRAM a platformei DSProto32, care se realizează cu programul DSProto32 Debugger;
- rularea programului încărcat în memoria SRAM și, evident, verificarea rezultatului obținut.

În cele ce urmează, se ilustrează etapele menționate considerând un exemplu simplu de calcul al maximului dintre două numere reale predefinite. Programul (*ex1.asm*) realizat și editat cu un editor de fișiere text este dat mai jos și este însoțit de comentarii, astfel că nu mai necesită și alte explicații suplimentare.

```
*****
*
*      PROGRAM EX. nr 1. DSP C32
* Se calculează maximumul dintre două numere reale
*
*****



.sect ".prog"

*****
*
*      Declararea variabilelor globale
*
*****



mem_addr    .set    0900000h      ;Adresa de început SRAM
mem_addrh   .set    090h          ;Cel mai semnificativ octet
                                ;al adresei de început SRAM
.bss rez,1      .bss rez,1      ;Se rezervă spațiu de memorie
                                ;pentru rezultat

.data

*****
```

```
* Se initializează numerele      *
*                                     *
*****  
A1      .float   2.0
A2      .float   3.0
Adr     .word    rez ; adr. de mem. unde se depune rezultatul

.text

*****  

*                                     *
*       Program principal          *
*                                     *
*****  
  
ldi 9005h,sp ;Se set. adr. mem. stivă: SP=900500h
lsh 8,sp
call init ;Se initializează procesorul TMS320C32
ldp mem_addr,dp ;Se încarcă în reg. DP valoarea
;0900000h  
  
* Se încarcă în registrul auxiliar ar0 adresa
* din memorie unde se află rezultatul  
  
ldi @Adr,ar0  
  
* Se încarcă în registrele r0 si r1 constantele k1 si k2  
  
ldf @A1,r0
ldf @A2,r1
cmpf r1,r0
bge stop
ldf r1,r0  
  
* Se salvează rezultatul  
  
stop: stf r0,*ar0  
  
* Se așteaptă într-o buclă până se resetează procesorul  
  
repeta:
      br repeta  
  
* Sfârșit program
```

\*\*\*\*\* initializare C32 \*\*\*\*\*

```
init:  
    and      0h,ie      ;Se dezactivează toate intreruperile  
    and      0h,if      ;Se sterge registrul if  
    ldi      mem_addrh,ar0  
    lsh      24,ar0  
    or       ar0,if  
    or       0800h,st    ;activare memorie cache  
    or       02000h,st   ;se set. bitul corespunzător  
                      ;intr. globale  
    retsu  
                      ;se revine din subrutină
```

După realizarea programului și editarea acestuia, urmează translatarea fișierului text în fișiere obiect relocabile, utilizând următoarea comandă:

```
asm3x -l ex1.asm | more
```

unde: *ex1.asm* reprezintă fișierul text sursă, iar argumentul “*l*” din linia de comandă specifică asamblorului faptul că acesta trebuie să creeze și un fișier listing care să ofere informații suplimentare cu privire la rezultatul obținut și, eventual, informații cu privire la erorile care pot să apară în program. În urma execuției comenții de mai sus, pe lângă fișierul *ex1.lst*, se mai obține și fișierul *ex1.obj*, care conține blocuri de program și de date relocabile. Dacă nu se specifică numele fișierelor ce urmează să se obțină, atunci numele acestora sunt identice cu cele ale fișierului cu extensia *asm*.

Fișierul *ex1.lst* este prezentat în cele ce urmează și conține patru câmpuri: (c1)- indică numărul liniei din fișierul text (cod program, definire date, comentarii etc; (c2)-numărul liniei din interiorul blocului obiect; (c3)-opciodul instrucțiunii sau valoarea numerică a datelor utilizate; (c4)-linia programului sursă care se regăsește și în fișierul sursă *ex1.asm*.





<i>(c1)</i>	<i>(c2)</i>	<i>(c3)</i>	<i>(c4)</i>
0001			***** *****
0002			* PROGRAM EX. nr 1. DSP C32 *
0003			* Se calculează maximul dintre două numere reale *
0004			***** *****
0005			
0006 000000			.sect ".prog"
0008			***** *****
0009			* Declararea variabilelor globale *
0010			***** *****
0011			
0012 00900000	mem_addr	.set	0900000h ;Adresa de început SRAM
0013 00000090	mem_addrh	.set	090h ;Cel mai semnificativ octet
0014			;al adresei de început SRAM
0015 000000 00000001		.bss rez,1	;Se rezervă spațiu de memorie
0016			;pentru rezultat
0017 000000		.data	
0018			***** *****
0019			* se initializează numerele *
0020			***** *****
0021 000000 01000000	A1	.float	2.0
0022 000001 01400000	A2	.float	3.0
0023 000002 00000000	Adr	.word	rez ;adresa de mem. unde se depune rezultatul

---

## Cap. 4 Sistemul de dezvoltare DSProto 32 cu procesorul de semnal TMS320c32

---

```
0024 000000          .text
0025
0026          ****
0027          *      Program principal      *
0028          ****
0029 000000 08749005    ldi  9005h,sp    ;Se setează adresa memoriei stivă: SP=900500h
0030 000001 09f40008    lsh  8,sp
0031 000002 6200000c    call init       ;Se initializează procesorul TMS320C32
0032 000003 08700090    ldp  mem_addr,dp  ;Se încarcă în reg. DP valoarea 0900000h
0033
0034 000004 08280002    * Se încarcă în registrul auxiliar ar0 adresa
0035
0036 000005 07200000    * din memorie unde se află rezultatul
0037 000006 07210001    ldi  @Adr,ar0
0038 000007 04000001    ldf  @A1,r0
0039 000008 6a0a0001    ldf  @A2,r1
0040 000009 07000001    cmpf r1,r0
0041
0042 00000a 1440c000    bge stop
0043
0044
0045 00000b
0046 00000b 6000000b    ldf  r1,r0
0047
0048
0049 00000c          * Se salvează rezultatul
stop:   stf r0,*ar0
0043
0044
0045 00000b
0046 00000b 6000000b    * Se așteaptă într-o buclă până se
0047
0048
0049 00000c          * resetează procesorul
repeta:
0046 00000b 6000000b    br repeta
0047
0048
0049 00000c          * Sfârșit program
init:
```

```
0050          ***** initializare C32 *****
0051
0052 00000c 02f60000    and   0h,ie      ;Se dezactivează toate intreruperile
0053 00000d 02f70000    and   0h,if      ;Se sterge registrul if
0054 00000e 08680090    ldi    mem_addrh,ar0
0055 00000f 09e80018    lsh    24,ar0
0056 000010 10170008    or     ar0,if
0057 000011 10750800    or     0800h,st    ;activare memorie cache
0058 000012 10752000    or     02000h,st   ;se set. bitul coresp. intr. glob.
0059 000013 78800000    retsu           ;se revine din intrerupere
```

Symbol Table

.bss	.bss	00000000	.data	.data	00000000
.prog	.prog	00000000	.text	.text	00000000
A1	.data	00000000	A2	.data	00000001
Adr	.data	00000002	init	.text	0000000c
mem_addr	.set	00900000	mem_addrh	.set	00000090
repeta	.text	0000000b	rez	.bss	00000000
stop	.text	0000000a			

[13 Symbols]







Trebuie menționat faptul că, după generarea listingului cu cele patru câmpuri, asamblorul pune la dispoziția utilizatorului un tabel cu simboluri utilizate la stabilirea legăturilor între modulele programului.

Pentru linkeditarea fișierelor obiect se utilizează fișierul *ex1.obj*, ce reprezintă fișierul de intrare pentru următoarea linie de comandă:

```
Ink3x -o rezultat.out -m ex1.map com.cmd ex1.obj
```

Linkeditarea fișierelor obiect necesită și un fișier de comenzi *com.cmd*, care are rolul de a stabili adresele de memorie unde se încarcă programul (.text) și unde se rezervă spațiu de memorie, atât pentru datele inițializate (.data), cât și pentru cele neinițializate (.bss).

**Fișierul de comenzi *com.cmd*.**

```
.prog 900700h  
.text  
.data  
.bss 910000h
```

**Observație.** Dacă pentru anumite directive (.text, .data) nu se specifică adresa, atunci acestea se setează automat la adresa imediat următoare spațiului de memorie ocupat de blocul specificat de precedenta directivă.

Fișierul *ex1.map* rezultat în urma linkeditării conține configurația memoriei în ceea ce privește spațiul ocupat de: instrucțiunile programului, datele inițializate și respectiv de datele neinițializate.

**Fișierul *ex1.map***

Memory Configuration					
Section	Run Address	Load Address	Size(hex)	Size(decimal)	
.text	0x00900700	0x00900700	0x00000014	00000020	
data	0x00900714	0x00900714	0x00000003	00000003	
.bss	0x00910000	0x00910000	0x00000001	00000001	
.prog	0x00900700	0x00900700	0x00000000	00000000	

Global Symbol Table			
Name	Value	Name	Value
[0 symbols]			

Toate aceste date vin în ajutorul programatorului, oferindu-i informații complete, care îi permit urmărirea variabilelor în timpul rulării programului.

Dispunând de fișierul *rezultat.out* care conține programul la nivel de cod obiect, se poate trece la ultima etapă care constă în încărcarea programului în memoria SRAM a platformei DSProto32 și în rularea acestuia.

După încărcarea programului în memoria SRAM a platformei DSProto32 și după dezasamblarea acestuia, rezultatul obținut este prezentat în figura 4.5, iar în figura 4.6 este dată zona de memorie alocată programului.

```

Disassembler
Hex Addr 900700 Print

900700:ldi 09005h,sp
900701:lsb 08h,sp
900702:call 090070ch
900703:ldp 090h
900704:ldi @0716h,ar0
900705:ldf @0714h,r0
900706:ldf @0715h,r1
900707:cmpf r1,r0
900708:bge 090070ah
900709:ldf r1,r0
90070A:stf r0,*ar0
90070B:br 090070bh
90070C:and 00h,ie
90070D:and 00h,if
90070E:ldi 090h,ar0
90070F:lsb 018h,ar0
900710:or ar0,if
900711:or 0800h,st
900712:or 02000h,st
900713:retsu
900714:addc r0,r0
900715:addc *+ar0(00h),r0
900716:absi r0,ir0
900717:undefined
900718:undefined

```

Fig. 4.5. Programul obținut în urma dezasamblării

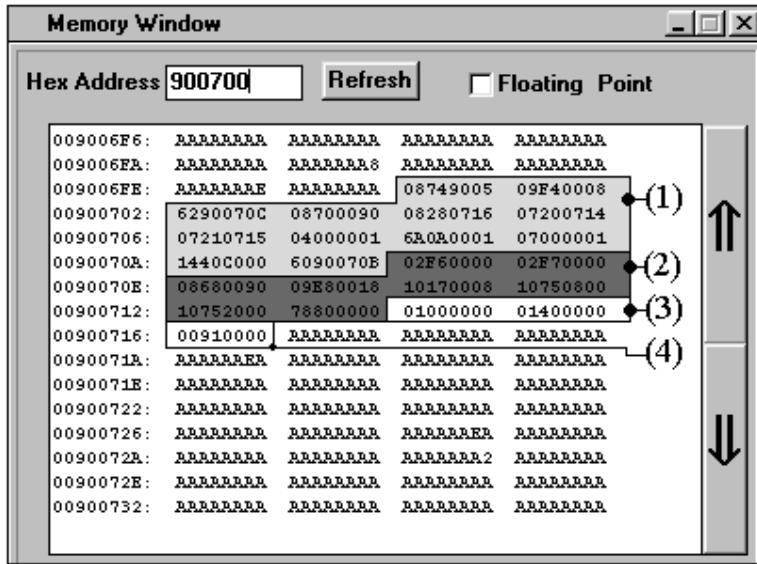


Fig. 4.6. Zona de memorie ocupată de program

Se poate observa faptul că programul este încărcat la adresa  $00900700h$ , adresă specificată prin intermediul fișierului de comenzi *com.cmd*. Prima zonă de memorie (1) corespunde programului principal, iar cea de-a doua zonă (2) este rezervată subroutinei de inițializare a procesorului *TMS320C32*. Imediat după aceste blocuri de memorie urmează, așa cum indică și fișierul de comenzi *com.cmd*, zona de memorie (3) corespunzătoare datelor inițializate care conține cele două numere reale. La locația (4) din figura 4.6 se află adresa unde se va depune maximul dintre cele două numere reale după rulare.

După încărcarea programului urmează rularea acestuia, iar maximul numerelor aflate la adresele de memorie  $00900714h$  și  $00900715h$  (figura 4.7) se găsește la adresa  $00910000h$  (figura 4.8).

**Observație.** Programul DSProto32 Debugger prin comanda *Read Mem* permite citirea datelor din memorie și afișarea acestora în format număr real cu exponent.

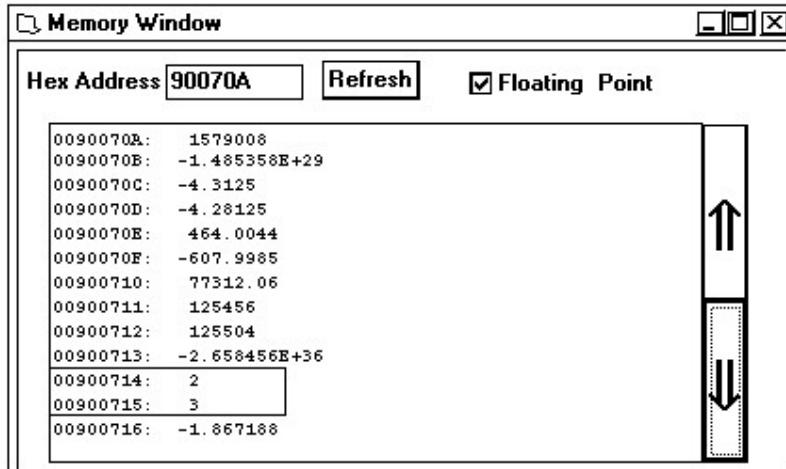


Fig. 4.7. Zona de memorie ocupată de variabilele de lucru

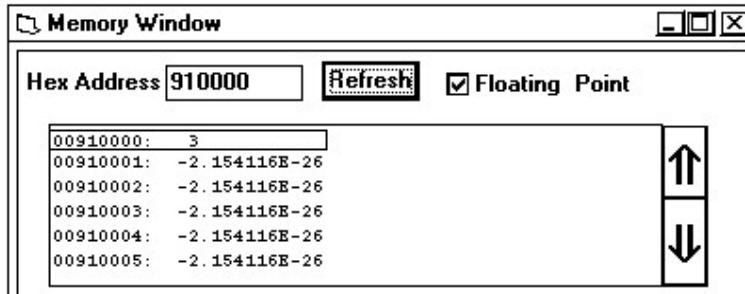


Fig. 4.8. Zona de memorie ocupată de variabila rezultată

#### 4.5. Directive ale limbajului de asamblare

Deși programul dezasamblat prezentat în paragraful 4.4 (vezi figura 4.5), reprezintă același program ca și cel din fișierul ex1.asm, prezentat tot în paragraful 4.4, totuși există diferențe privind anumite **elemente prin care se stabilesc secțiunile de date și de program și elemente prin care se definesc constantele și variabilele de lucru**. Aceste elemente

de control se **numesc directive** și oferă un cadru riguros și comod în ceea ce privește programarea într-un limbaj de asamblare.

În cele ce urmează, se prezintă o parte dintre cele mai uzuale directive care se regăsesc și în cadrul aplicațiilor ce se vor prezenta în această lucrare.

#### 4.5.1. Directive prin care se definesc secțiunile de program

Din această categorie fac parte directivele: **.data**, **.bss**, **.text** cu rolul următor:

- directiva **.data** specifică asamblorului faptul că începe o nouă secțiune de date prin care se initializează variabilele utilizate în cadrul programului;
- directiva **.bss** indică asamblorului faptul că urmează o nouă secțiune de date dedicată variabilelor neinitialize prin care se rezervă o anumită zonă de memorie;
- directiva **.text** are rolul de a informa asamblorul despre faptul că urmează o secțiune de program care conține cod executabil.

Acste directive se pot intercala în program, iar linkeditorului îi revine sarcina de a concatena secțiunile din program definite prin aceeași directivă, într-o singură secțiune de program, astfel că utilizatorul poate folosi o directivă în orice loc din program.

Pe lângă cele menționate, asamblorul permite definirea unor secțiuni de program personalizate, utilizând directivele **.usect** și **.sect** care sunt similare directivelor **.bss** și, respectiv, **.data** sau **.text**. Prin intermediul acestor noi directive se poate realiza o structurare adecvată a programului realizat și o administrare eficientă a spațiului de memorie.

```
.sect "numele secțiunii"  
.simbol .usect "numele secțiunii", nr. de cuvinte rezervate
```

#### 4.5.2. Directive prin care se inițializează constantele și variabilele de lucru

Printre cele mai cunoscute directive utilizate la inițializarea constantelor și a variabilelor se regăsesc următoarele:

- directiva **.byte** inițializează, în secțiunea curentă, o succesiune de cuvinte cu octeții din dreapta acesteia. În exemplul următor: {strx: .byte 10,-1,"abc",'a'}, se inițializează zona de memorie curentă cu următoarele cuvinte: 0000000A; 000000FF; 00000061; 00000062; 00000063; 00000061. Se observă faptul că pentru fiecare octet asamblorul rezervă câte un cuvânt pe 32 de biți.
- directiva **.equ** având sintaxa: *simbol* .equ *val* atribuie valoarea *val* variabilei *simbol*. În exemplul următor: {PI .equ 3.141592654}, simbolul PI va fi utilizat ca o constantă a cărei valoare este 3.141592654.
- directiva **.set** este identică cu directiva **.equ** și este folosită cu preponderență la setarea adreselor de lucru.
- directivele: **.int**, **.long**, **.word** permit inițializarea în secțiunea curentă a unui cuvânt sau a mai multor cuvinte cu date reprezentate pe 8, 16 sau 32 de biți. În exemplul următor,

```
apvcl    .word    1,1,1,2,2,2,3,3,3,4,4,4,5,5,5,6,6,6,7,7,7  
adapvcl .word    apvcl
```

zona de memorie de la adresa: *apvcl* este inițializată cu cele 23 de valori specificate în dreapta directivei, iar la adresa *adapvcl* se va afla adresa de început a zonei de memorie rezervată cu directiva precedentă. Prin acest mod de atribuire se poate adresa orice locație de memorie care este inițializată cu valori cunoscute.

- directiva **.float** inițializează o constantă sau elementele unui vector cu o valoare reală. Constantele se pot inițializa atât cu valori reprezentate fără exponent, cât și cu valori reprezentate cu exponent, așa cum reiese și din exemplul următor:

<i>cta</i>	.float	2.0
<i>ctmax</i>	.float	1.0e30

#### 4.5.3. Directive care permit referirea la alte surse

Realizarea aplicațiilor complexe presupune utilizarea unor simboluri globale care să reprezinte aceeași mărime pentru mai multe module de program. De asemenea, este util să se definească o unitate de program numită *macro* care apoi să fie utilizată asemenea unei funcții. În cele ce urmează, se prezintă directivele care răspund cerințelor menționate:

- directivele **.include** și **.copy**, având sintaxa: **.include "nume fisier"** și **.copy "nume fisier"**, permit citirea unor fișiere surse și includerea acestora în fișierul curent și în locul unde s-a utilizat respectiva directivă. Prin utilizarea acestor directive se pot include tabele și vectori care se află în alte fișiere, astfel că programul principal devine mai aerisit;
- directiva **.def symbol 1 [, ... , symbol n ]** identifică un simbol sau mai multe simboluri definite în modulul curent și utilizate în alte module;
- directiva **.global symbol 1 [, ... , symbol n ]** identifică unul sau mai multe simboluri globale;
- directiva **.ref symbol 1 [, ... , symbol n ]** identifică unul sau mai multe simboluri globale utilizate în modulul curent;
- directiva **.mllib [ "] filename[ "]** definește o librărie de macrouri.

*În concluzie, se poate aprecia faptul că setul de instrucțiuni al procesorului de semnal, împreună cu directivele recunoscute de limbajul de asamblare, constituie elemente solide care pot susține implementarea lejeră a unor aplicații complexe.*

## CAPITOLUL 5

### APLICAȚII ÎN CALCULUL VECTORIAL

În cadrul acestui capitol se vor prezenta aplicații privind operații asupra vectorilor, urmărind atât ilustrarea principiilor programării în limbajul de asamblare al procesorului de semnal TMS320C32, cât și evidențierea performanțelor obținute cu acest procesor, în ceea ce privește puterea de procesare a datelor. Exemplele sunt însotite și de o scurtă prezentare a problemelor abordate, iar acolo unde este cazul se prezintă rezultatele obținute în urma rulării programului.

#### 5.1. Sumarea ponderată a doi vectori

Așa cum reiese și din titlu, în cadrul acestui paragraf se prezintă un program în care se determină suma ponderată a elementelor a doi vectori. Dacă se notează cu  $x$  și  $y$  cei doi vectori, atunci suma ponderată cu numerele  $k_1$  și respectiv  $k_2$  este dată de relația următoare:

$$z(i) = k_1 \cdot x(i) + k_2 \cdot y(i), \quad \text{unde } i = 1 \dots 5. \quad (5.1)$$

De asemenea, în cadrul programului, se exemplifică adresarea indirectă prin intermediul registrelor auxiliare și se pune în evidență maniera de lucru cu instrucțiunea de repetare a unui bloc de instrucțiuni.

În cele ce urmează, este prezentat listingul programului, însorit de comentarii care ușurează atât înțelegerea algoritmului, cât și modul de operare al instrucțiunilor.

```
*****
*
*      PROGRAM nr. 1 PENTRU DSP C32
*
* Se dau doi vectori de numere reale având fiecare câte
* cinci elemente. Vectorii sunt memorati la adresele V_X
* si V_Y. Să se calculeze z(i)=k1*x(i)+k2*y(i)
* unde i=1,5 , iar k1 si k2 sunt constante definite
*
*****
.sect ".prog"
*****
*
*      Declararea variabilelor globale
*
*****
mem_addr      .set    0900000h ;Adresa de inceput SRAM
mem_addrh     .set    090h      ;Cel mai semnificativ
                           ;octet al adresei de
                           ;inceput SRAM

.bss rez,5      ;Se rezerva spatiu de memorie
                  ;pentru vectorul rezultat

.data
*****
*      Se initializeaza valorile vectorilor V_X si V_Y
*****
V_X           .float 1.25          ;vectorul 1 (5 elemente)
              .float 1.5
              .float 1.75
              .float 2.0
              .float 2.25

V_Y           .float 4.25          ;vectorul 2 (5 elemente)
              .float 4.5
              .float 4.75
```

---

```

        .float 5.0
        .float 5.25

*****
* se inițializează constantele *
*****

k1          .float   2.0
k2          .float   3.0

* Se setează adresele zonelor de memorie
* corespunzătoare vectorilor sursă și destinație

Advx        .word    V_X      ; adresa de început a vect. 1
Advy        .word    V_Y      ; adresa de început a vect. 2
Advz        .word    rez      ; adresa bufferului cu val.
                           ;vect. de ieșire

.text

*****
*       Program principal      *
*****


ldi    9005h,sp           ;Se setează adresa memoriei
                           ;stiva: SP= 900500h
lsh    8,sp
call   init                ;Se inițializează
                           ;procesorul TMS320C32
ldp   mem_addr,dp          ;Se încarcă în reg. DP
                           ;valoarea 0900000h
*
* Se încarcă în registrele auxiliare adresele din memorie
* unde se află vectorii sursă și vectorul destinație
*
ldi @Advx,ar7
ldi @Advy,ar6
ldi @Advz,ar5

*
* Se încarcă în registrele r0 si r1 constantele k1 si k2
*
ldf @k1,r0
ldf @k2,r1

```

```
*  
* Se stab. nr. de repetări(4+1) pentru  
* următoarea secvență de program  
*  
    ldi 4,rc  
    rptb const  
    ldf *ar7++,r2  
||     ldf *ar6++,r3  
        mpyf r0,r2  
        mpyf r1,r3  
        addf r3,r2  
const:   stf r2,*ar5++  
  
*  
* Se așteaptă într-o buclă până se resetează procesorul  
*  
  
repeta:  
    br repeta  
  
* Sfârșit program  
init:  
***** Subrutina de initializare C32 *****  
    and    0h,ie      ;Se dezactivează toate intreruperile  
    and    0h,if      ;Se șterge registrul if  
  
    ldi    mem_addrh,ar0  
    lsh    24,ar0  
    or     ar0,if  
    or     0800h,st    ;activare memorie cache  
    or     02000h,st    ;se setează bitul corespunzător  
                      ;întreruperii globale  
    retsu           ;se revine din subrutină
```

Pentru o ilustrare a modului de realizare a unui program în asamblare și a modului de rulare a acestuia pe platforma prezentată în capitolul 4, se descriu în cele ce urmează etapele obținerii programului executabil.

Fișierul executabil în format hexa se obține prin convertirea fișierului text de mai sus în fișiere obiect și apoi linkeditarea acestora, utilizând următoarele linii de comandă:

```
asm3x -l prog1.asm | more
lnk3x -o rezultat.out -m prog1.map com.cmd
prog1.obj
```

Fișierul *com.cmd* stabilește zona de memorie unde se va încărca programul executabil care este de fapt fișierul *rezultat.out*. Fișierul de comenzi *com.cmd* are următoarea structură:

```
.prog 900700h
.text
.data
.bss 910000h
```

În urma linkeditării se obține fișierul *prog1.map* care conține configurația memoriei în ceea ce privește spațiul ocupat de:

- instrucțiunile programului, stabilite prin directiva *.text*;
- datele inițializate care s-au stabilit prin directiva *.data*;
- datele neinițializate definite prin directiva *.bss*.

Conținutul fișierului care indică secțiunile de program și dimensiunea acestora în cuvinte a câte 32 de biți fiecare este dat mai jos.

```
Memory Configuration
Section Run Address Load Address Size(hex) Size(decimal)
-----
.text 0x00900700 0x00900700 0x00000019 00000025
.data 0x00900719 0x00900719 0x0000000f 00000015
.bss 0x00910000 0x00910000 0x00000005 00000005
.prog 0x00900700 0x00900700 0x00000000 00000000

Global Symbol Table
Name Value Name Value
-----
[0 symbols]
```

După încărcarea programului în memoria SRAM a platformei DSProto32 și după dezasamblarea acestuia, se obține de fapt același

program (figura 5.1), dar care nu mai conține simboluri. Toate adresele sunt adrese fizice, iar registrele se încarcă cu valori bine precizate, valori corelate cu poziția programului în spațiul fizic de memorie.

În figura 5.2 se prezintă zona de memorie alocată programului unde se poate observa secțiunea alocată instrucțiunilor (1 - programul principal, 2 – subrutina de initializare) și secțiunea de date care conține, la rândul ei, zona de memorie 3 în care se află cei doi vectori și cele două constante și zona de memorie 4 în care sunt salvate adresele de început ale blocurilor de memorie corespunzătoare celor trei vectori: doi sursă și unul destinație.

```

Disassembler
Hex Addr 900700 Print

900700:ldi 09005h,sp
900701:lsh 08h,sp
900702:call 0900711h
900703:ldp 090h
900704:ldi @0725h,ar7
900705:ldi @0726h,ar6
900706:ldi @0727h,ar5
900707:ldf @0723h,r0
900708:ldf @0724h,r1
900709:ldi 04h,rc
90070A:rptb 090070fh
90070B:ldf *ar7++,r2 || ldf *ar6++,r3
90070C:mpyf r0,r2
90070D:mpyf r1,r3
90070E:addf r3,r2
90070F:stf r2,*ar5++(01h)
900710:br 0900710h
900711:and 00h,ie
900712:and 00h,if
900713:ldi 090h,ar0
900714:lsh 018h,ar0
900715:or ar0,if
900716:or 0800h,st
900717:or 02000h,st
900718:retsu

```

Fig. 5.1. Programul încărcat în memoria de lucru

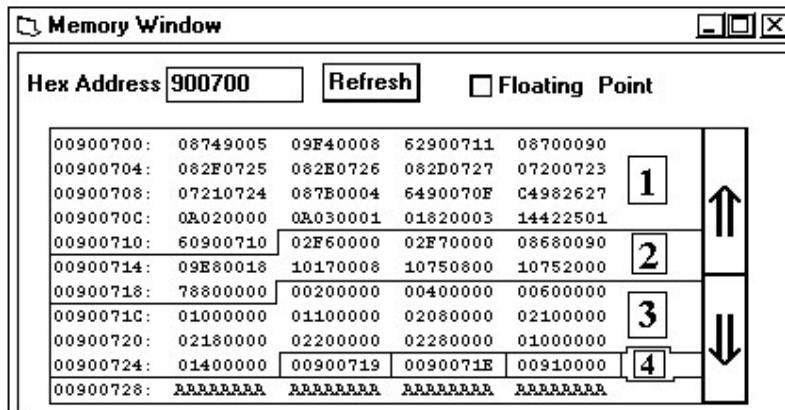


Fig. 5.2. Memoria de lucru

Dispunând de facilitatea programului DSProto32 Debugger de a afișa datele din memorie și în format - număr real cu exponent, în figurile următoare se prezintă datele de intrare și de ieșire din zonele de memorie respective, în format zecimal cu virgulă.

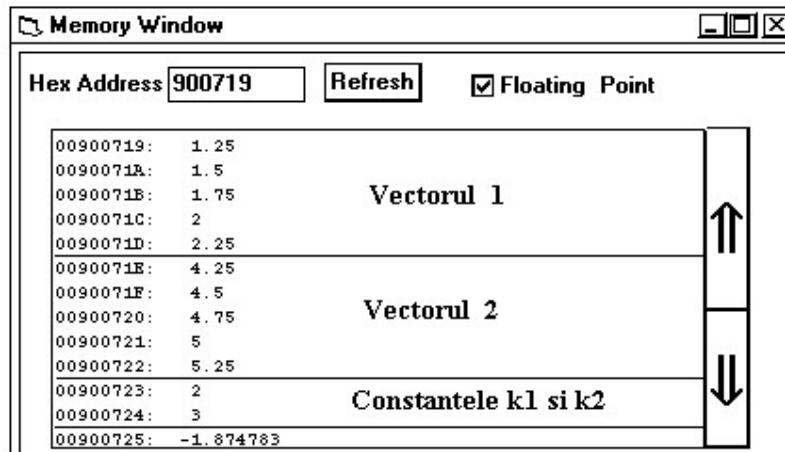


Fig. 5.3. Variabilele de lucru

Memory Window	
Hex Address	Refresh
00910000:	15.25
00910001:	16.5
00910002:	17.75
00910003:	19
00910004:	20.25
00910005:	1.92593E-34
00910006:	1.001221
00910007:	1.001221

Fig. 5.4. Rezultatul obținut în urma executării programului

## DSP Laborator 5

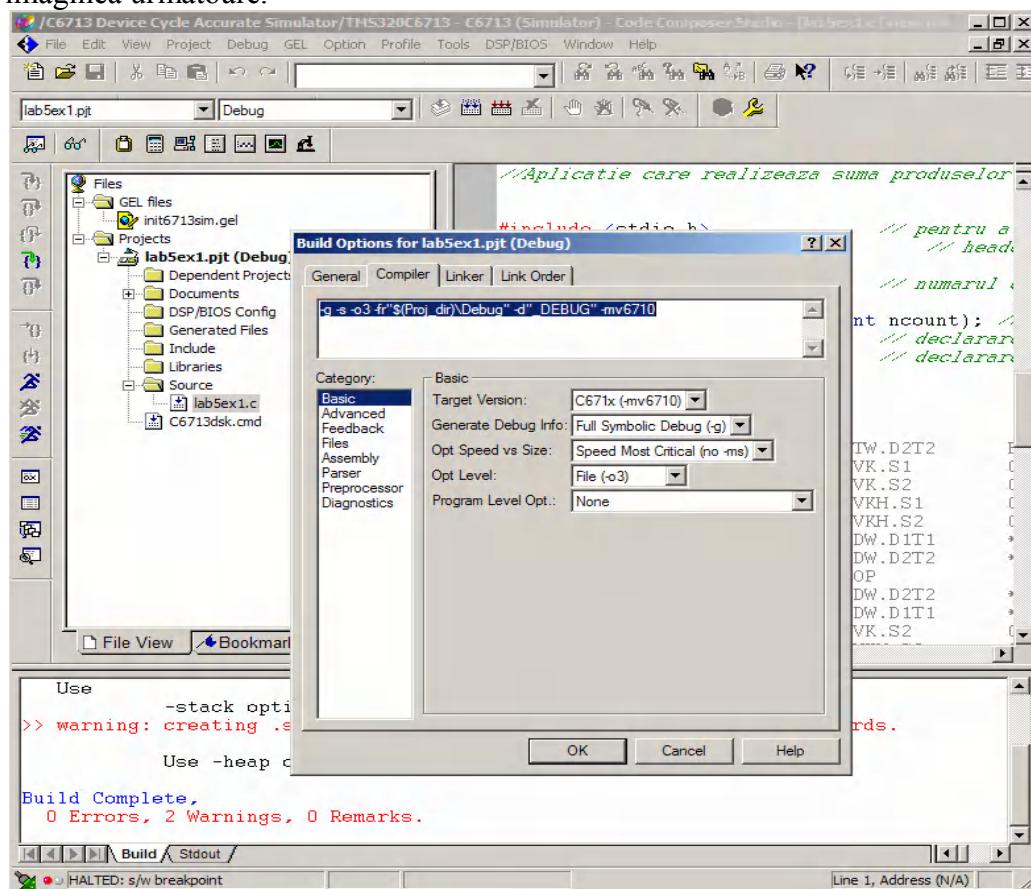
**L5 :** Descrierea hardware a platformei de dezvoltare TMS320C6713 DSK si a mediului de programare CodeComposer. Familiarizarea cu mediul de programare. Descrierea subsistemelor DSP. BSL si CSL.

- Documentatii DSK.
- Set de instructiuni.
- Tiparul de dezvoltare a aplicatiilor.
- Suma produselor elementelor a doi vectori in C.

### Ex1. Suma produselor elementelor a doi vectori in C

Tinand cont ca descrierea hardware si software a platformei a fost realizata sub forma unor anexe in continuare se prezinta o aplicatie simpla dezvoltata utilizand Code Composer Studio 3.1 pentru platforma cu DSP TMS320C6713. Aplicatia de fata isi propune sa calculeze suma produselor componentelor a doi vectori de lungime data. In cazul de fata se considera doi vectori cu 4 componente si in cele ce urmeaza se prezinta detalii de configurare a unui proiect si de scriere a codului.

Toate aplicatiile care se vor dezvolta in cadrul laboratorului vor fi integrate in proiecte. Astfel pentru aplicatia de fata se considera un proiect numit lab5ex1. Dupa crearea proiectului (pentru targetul TMS320C6713) acesta se configureaza utilizand meniul Project → Build Options dupa cum se poate observa in imaginea urmatoare.



In acest moment putem modifica optiunile care se vor transmite liniei de compilare a fisierelor proiectului, optiunile de linkeditare, optiunile de optimizare si optiuni generale privind ordinea de

compilare. Aceste optiuni modificate in fereastra IDE-ului se vor regasi in fisierul pjt asociat proiectului.

Pentru a prezenta modul in care trebuie configurat un proiect corespunzator pentru targetul TMS320C6713 este redat in continuare continutul fisierului pjt pentru exemplul considerat care va fi considerat template pentru proiectele viitoare.

```
; Code Composer Project File, Version 2.0 (do not modify or remove this line)

[Project Settings]
ProjectDir="D:\school_stuff_and_work\ore_facultate\DSP\laboratoare\lab_tms320c6713
\lab1\lab5ex1\
ProjectType=Executable
CPUFamily=TMS320C67XX
Tool="Compiler"
Tool="CustomBuilder"
Tool="DspBiosBuilder"
Tool="Linker"
Config="Debug"
Config="Release"

[Source Files]
Source="lab5ex1.c"
Source="lab5ex1.h"
Source="C6713dsk.cmd"

["Compiler" Settings: "Debug"]
Options=-g -s -o3 -fr"${Proj_dir}\Debug" -d"_DEBUG" -mv6710

["Compiler" Settings: "Release"]
Options=-o3 -fr"${Proj_dir}\Release" -mv6700

["Linker" Settings: "Debug"]
Options=-c -m".\Debug\lab5ex1.map" -o".\Debug\lab5ex1.out" -w -x -l"rts6700.lib"

["Linker" Settings: "Release"]
Options=-c -m".\Release\lab5ex1.map" -o".\Release\lab5ex1.out" -w -x

["lab5ex1.h" Settings: "Debug"]
ExcludeFromBuild=true

["lab5ex1.h" Settings: "Release"]
ExcludeFromBuild=true

["C6713dsk.cmd" Settings: "Debug"]
LinkOrder=1

["C6713dsk.cmd" Settings: "Release"]
LinkOrder=1
```

Dupa ce proiectul a fost creat si configurat se pot adauga fisierele sursa pentru acesta. In acest context acestea pot fi creat apriori sau direct in cadrul proiectului. In cazul de fata se adauga 3 fisiere noi proiectului creat. Doua dintre ele sunt fisierele sursa lab5ex1.h si lab5ex1.c, iar al treilea fisier este un fisier de comanda a linkerului. Acest fisier C6713dsk.cmd are rolul de a seta limitele de memorie si sectiunile definite in memorie pentru vectorii de intreuperi, pentru sectiunea de variabile globale, pentru stiva etc. Fisierul de comanda a linkerului care va fi utilizat in fiecare proiect dezvoltat se poate observa in continuare.

```

/*C6713dsk.cmd  Linker command file*/

MEMORY
{
    IVECS: org=0h,           len=0x220
    IRAM:   org=0x0000220, len=0x0002FDE0 /*internal memory*/
    SDRAM:  org=0x80000000, len=0x00100000 /*external memory*/
    FLASH:  org=0x90000000, len=0x00020000 /*flash memory*/
}

SECTIONS
{
    .EXT_RAM :> SDRAM
    .vectors :> IVECS    /*in vector file*/
    .text     :> IRAM     /*Created by C Compiler*/
    .bss      :> IRAM
    .cinit    :> IRAM
    .stack    :> IRAM
    .sysmem   :> IRAM
    .const    :> IRAM
    .switch   :> IRAM
    .far      :> IRAM
    .cio      :> IRAM
    .csldata  :> IRAM
}

```

Dupa configurarea proiectului acesta se compileaza utilizand Project→ Build si outputul compilarii se poate observa in fereastra Build a IDE-ului. Avand posibilitatea de a vizualiza cod mixt C si asm putem executa aplicatia in modul Run si sa obtinem rezultatul in fereastra Stdout sau in mod pas cu pas setand un cursor la instructiunea de interes si apoi se executia analizand incarcarea registrilor si puterea paralelismului instructiunilor DSP. In figura urmatoare este redata executia aplicatiei.

The screenshot shows the Code Composer Studio interface. On the left, the project tree for 'lab5ex1.pjt (Debug)' is visible, containing files like 'lab5ex1.c', 'lab5ex1.h', and 'C6713dsk.cmd'. The main window displays the C source code for a dot product function:

```

//Aplicatie care realizeaza suma produselor elementelor a doi vectori de dimensiune data

#include <stdio.h>           // pentru a utiliza printf
#include "lab5ex1.h"          // header care contine definitiile vectorilor

#define count 4                // numarul de componente al vectorilor

int dotp(short *a, short *b, int ncount); //prototipul functiei
short x[count] = {x_array};    // declararea primului vector
short y[count] = {y_array};    // declararea vectorului 2

main()
{
    00007A60      main:   STW.D2T2    B3,*SP-[0x4]
    00007A64  01BC94F6  MVK.S1     0xffff85b0,A3
    00007A68  0242D82A  ||       MVK.S2     0xffff85b0,B4
    00007A6C  01800069  ||       MVKH.S1    0x0000,A3
    00007A70  0200006A  ||       MVKH.S2    0x0000,B4
    00007A74  020C2265  LDW.D1T1    *+A3[0x1],A4
    00007A78  029002E7  ||       LDW.D2T2    *+B4[0x1],B5
    00007A7C  00000000  ||       NOP
    00007A80  021002E7  LDW.D2T2    *+B4[0x0],B4
    00007A84  028C0264  ||       LDW.D1T1    *+A3[0x0],A5
    00007A88  01BD662A  MVK.S2     0x7acc,B3
    00007A8C  0180006A  MVKH.S2    0x0000,B3
    00007A90  02000000  ||       NOP
    00007A94  02908C93  MDY.M2X    B5,A4,B5
    00007A98  01949001  ||       MDYH.MIX   A4,B5,A3
    00007A9C  00000000  ||       NOP
    00007AA0  0210B001  MDYH.M1X   A5,B4,A4
    00007AA4  0FFF6C10  ||       B.S1      printf
    00007AA8  01947079  ADD.L1X   A3,B5,A3
    00007AAC  02149C92  MDY.M2X   B4,A5,B4
    00007AB0  018C8078  ADD.L1    A4,A3,A3
    00007AB4  020C907B  ADD.L2X   B4,A3,B4

```

The assembly code shows the generated code for the 'main' function, including memory moves (MVK, MVKH), loads (LDW), and arithmetic operations (ADD). Below the assembly is the build log:

```

lab5ex1.pjt - Debug ----
[lab5ex1.c] "C:\CCStudio_v3.1\CC6000\cgtools\bin\cl6x" -g -s -o3 -f
[Linking...] "C:\CCStudio_v3.1\CC6000\cgtools\bin\cl6x" -@Debug.lk
<Linking>
>> warning: creating .stack section with default size of 400 (hex)
  Use
  -stack option to change the default size.
>> warning: creating .symmem section with default size of 400 (hex)
  Use -heap option to change the default size.
Build Complete,
  0 Errors, 2 Warnings, 0 Remarks.

```

The bottom status bar indicates the build was successful.

Codul aplicatiei :

### lab5ex1.c

```

//Aplicatie care realizeaza suma produselor elementelor a doi vectori de dimensiune data

#include <stdio.h>           // pentru a utiliza printf
#include "lab5ex1.h"          // header care contine definitiile vectorilor

#define count 4                // numarul de componente al vectorilor

int dotp(short *a, short *b, int ncount); //prototipul functiei
short x[count] = {x_array};    // declararea primului vector
short y[count] = {y_array};    // declararea vectorului 2

main()
{
    int result = 0;           // initializarea sumei produselor

    result = dotp(x, y, count); // se apeleaza functia
    printf("rezultat = %d \n", result); //se afiseaza rezultatul pe stdout
}

int dotp(short *a, short *b, int ncount)
//functia de calcul al sumei de produse
{
    int sum = 0;              //initializarea locala a sumei
    int i;

```

```
for (i = 0; i < ncount; i++)
    sum += a[i] * b[i];           //se calculeaza suma produselor
return(sum);                   //returneaza valoarea
}
```

lab5ex1.h

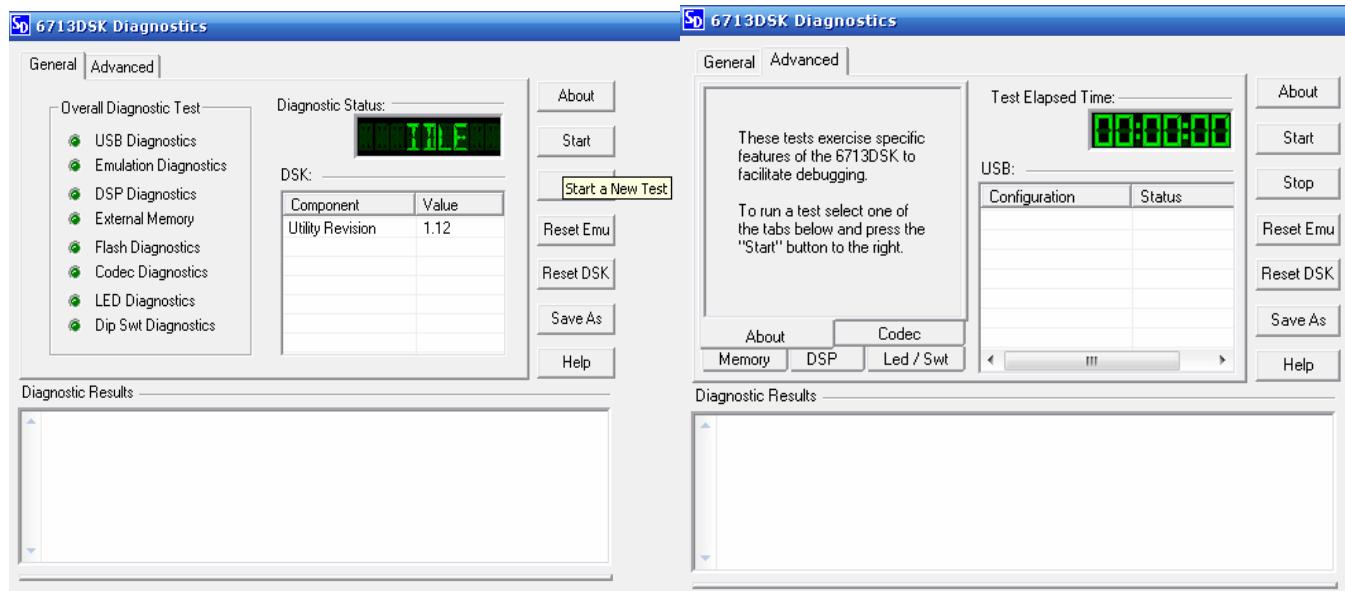
```
//Vectorii considerati
```

```
#define x_array 1,2,3,4
```

```
#define y_array 0,2,4,6
```

# Dezvoltarea aplicatiilor cu TMS320C6713 utilizand Code Composer Studio v 3.1

Pentru inceput pentru a realiza conexiunea cu placa de dezvoltare cu TMS320C6713 se deschide utilitarul 6713 DSK Diagnostics Utility v3.1 pentru a realiza verificarea statusului placii de dezvoltare, respectiv verificarea portului USB de conexiune, diagnosticarea DSP si a Emulatorului, a memoriilor externe si flash precum si a modulului Codec si a LED-urilor si switch-urilor DIP. Testele efectuate pot fi implicit parametrizate sau avansate pentru testare online de catre utilizator pentru fiecare subsistem mentionat mai sus.



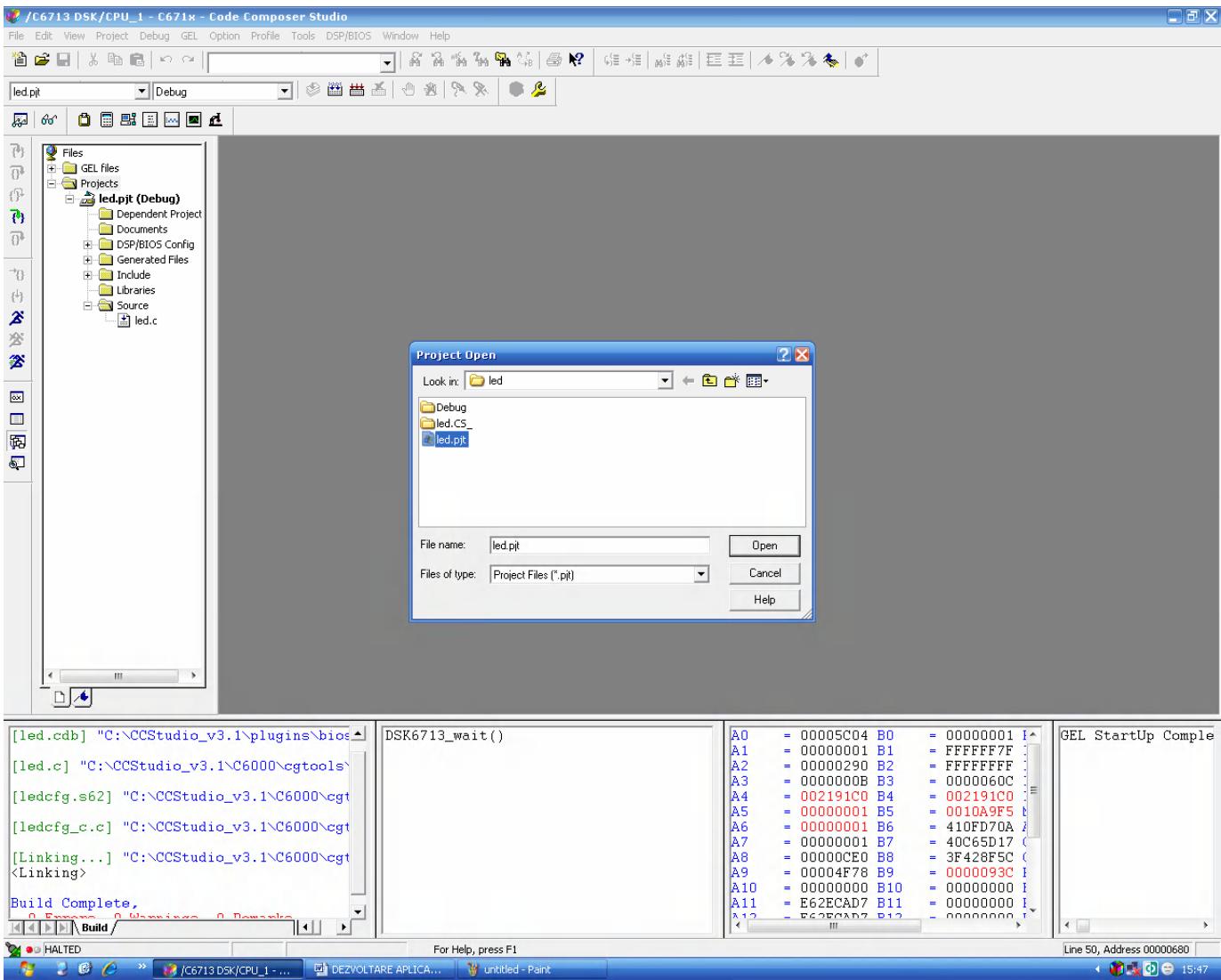
Dupa realizarea diagnozei daca s-au obtinut rezultatele bune, adica toate subsistemele au statusul conform cu parametrii impusi la initializare, se deschide aplicatia Code Composer Studio v 3.1 pentru a dezvolta aplicatii.

## Dezvoltarea de aplicatii cu Code Composer Studio

Se deschide mediul de programare si se acceseara din bara de meniu optiunea Debug→Connect (Alt+C), acum se stabileste comunicarea cu placa de dezvoltare.

Pentru inceput se prezinta modul de executare a unei aplicatii (unui proiect) anterior dezvoltat.

Se acceseara din bara de meniu optiunea Project→Open si se alege fisierul cu extensia .pj1 dorit.



Pentru a accesa sursele scrise in limbajul C alegem din fereastra de navigare din stanga accesul la sursa executabila a proiectului (care contine void main()). Apoi se deschide fisierul de baza. In continuare trebuie compilate si linkeditate toate fisierele incluse (din folderul Include) precum si utilizarea librariilor specific pentru lucrul cu compilatorul specific placii de dezvoltare pentru DSP. Se acceseaza meniul Project→Build (F7) se obtine un output ca urmatorul(exemplu):

----- led.pjt - Debug -----

```
[led.cdb] "C:\CCStudio_v3.1\plugins\bios\gconfig" led.cdb
```

```
[led.c] "C:\CCStudio_v3.1\C6000\cgtools\bin\cl6x" -g -q -fr"./Debug" -i". -i"C:/CCStudio_v3.1/c6000/dsk6713/include" -d"_DEBUG" -d"CHIP_6713" -ml3 -mv6710 -@"Debug.lkf" "led.c"
```

```
[ledcfg.s62] "C:\CCStudio_v3.1\C6000\cgtools\bin\cl6x" -g -q -fr"./Debug" -i". -i"C:/CCStudio_v3.1/c6000/dsk6713/include" -d"_DEBUG" -d"CHIP_6713" -ml3 -mv6710 -@"Debug.lkf" "ledcfg.s62"
```

```
[ledcfg_c.c] "C:\CCStudio_v3.1\C6000\cgtools\bin\cl6x" -g -q -fr"./Debug" -i". -i"C:/CCStudio_v3.1/c6000/dsk6713/include" -d"_DEBUG" -d"CHIP_6713" -ml3 -mv6710 -@"Debug.lkf" "ledcfg_c.c"
```

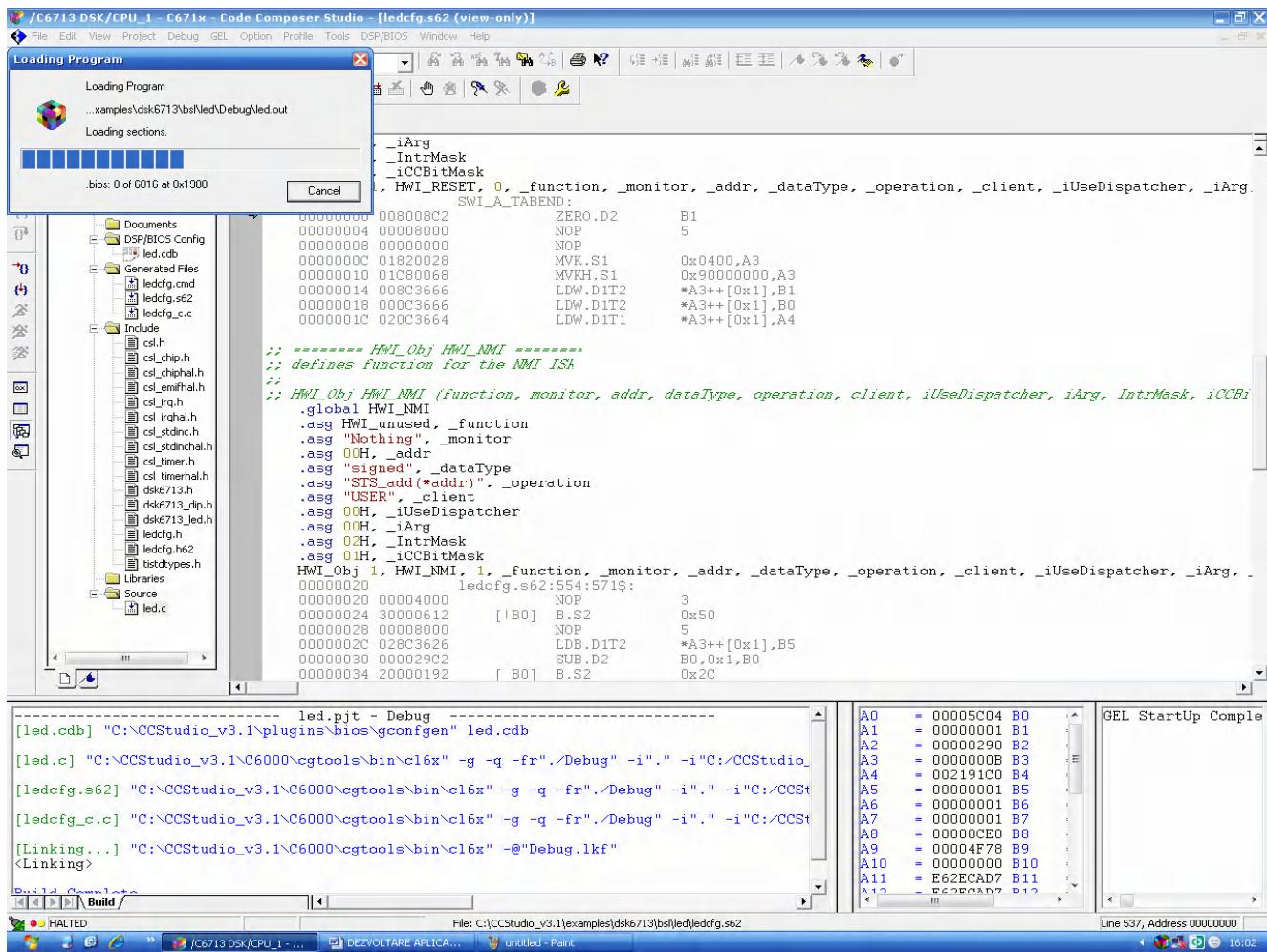
[Linking...] "C:\CCStudio\_v3.1\C6000\cgtools\bin\cl6x" -@"Debug.lkf"

<Linking>

Build Complete,

0 Errors, 0 Warnings, 0 Remarks.

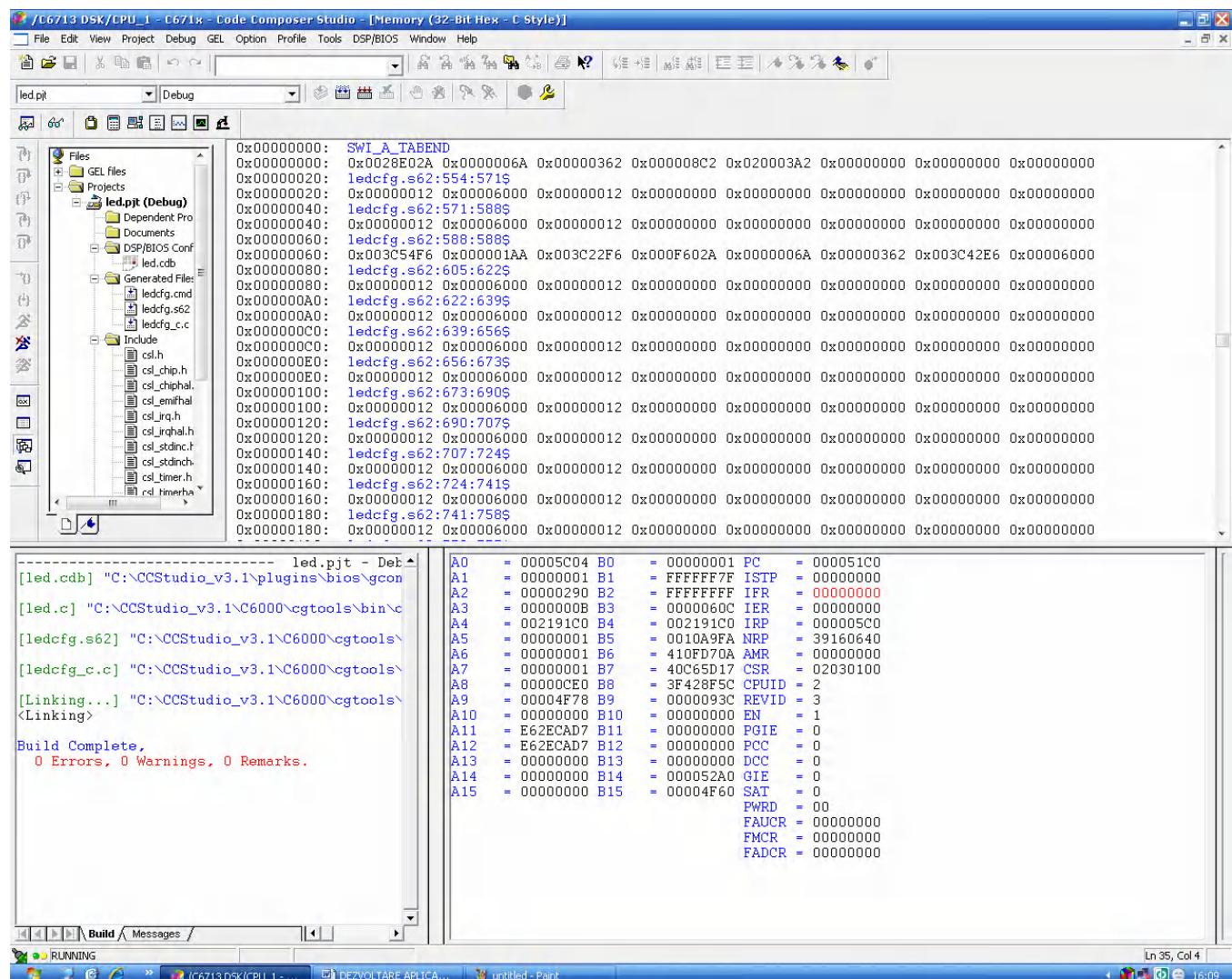
Apoi dupa etapa de compilare se trimit fisierul executabil cu extensia .out catre memoria placii dezvoltare utilizand File→Load Program...(Ctrl+L) si din fisierul Debug creat in directorul curent al proiectului fisierul creat in acea locatie dupa compilare.



Dupa ce s-a incarcat programul in memorie se trece la executie. Se accesaza meniul Debug→Go Main pentru a selecta locul de intrare in program in momentul executiei, iar apoi Debug→Run si se incepe executia programului.

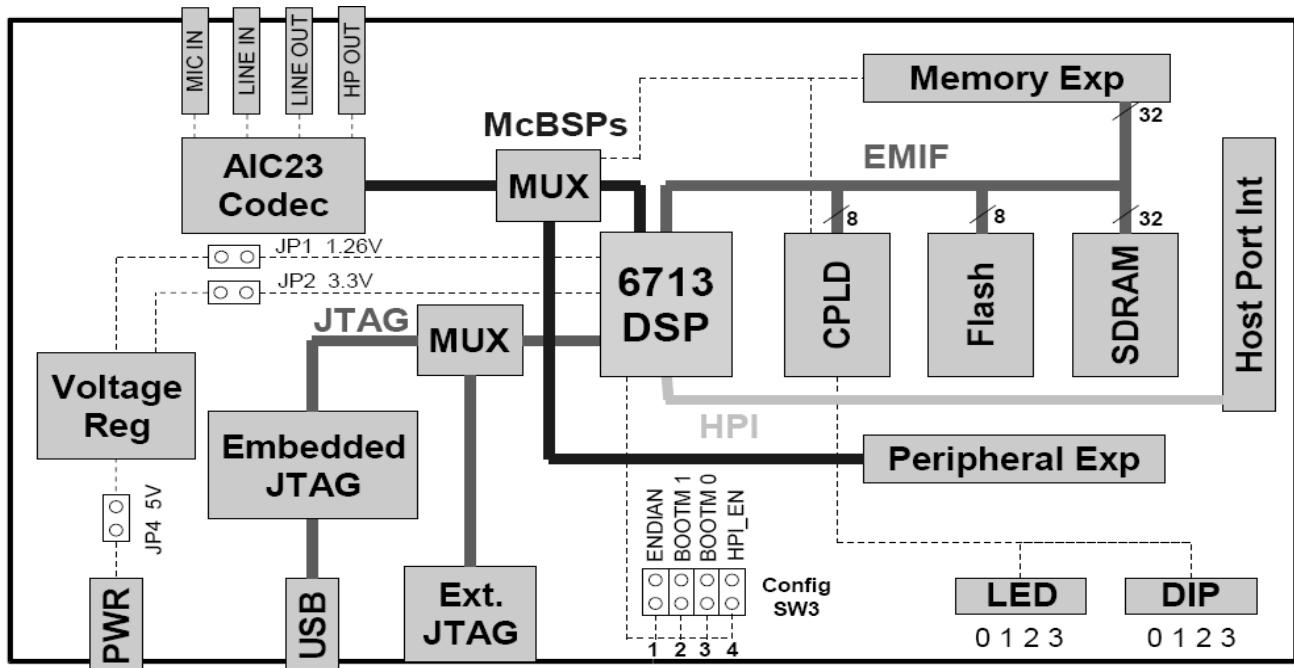
In timpul executiei avem posibilitatea de a vizualiza harta memoriei , starea si continutul registrilor procesorului. O alta facilitate oferita de mediul de programare este posibilitatea de a vizualiza in interiorul codului sursa C si a instructiunilor in limbaj de asamblare care corespund instructiunilor limbajului de nivel inalt. Aceasta facilitate este utila la rularea pas cu pas a instructiunilor fie in C si in

paralel in asamblare pentru studiul incarcarii registrelor extinse, auxiliare si de baza precum si a utilizarii unitatilor functionale din structura DSP-ului si anume ALU, unitatile de inmultire etc.



## Descrierea platformei de lucru cu DSP TMS320C6713

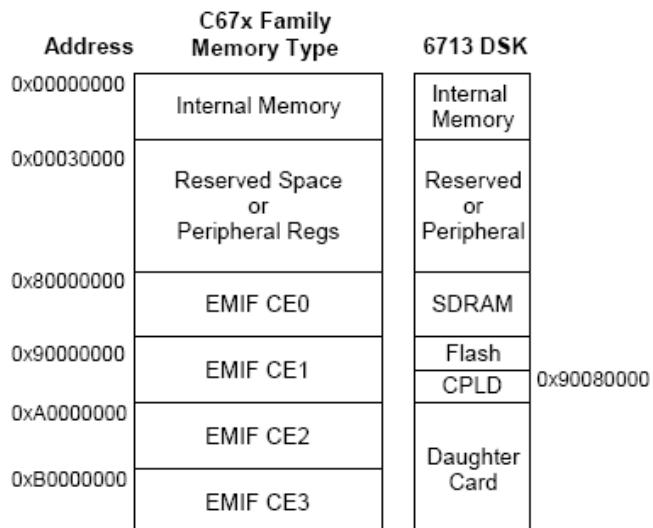
Pentru a doua parte a laboratorului pentru dezvoltarea aplicatiilor de procesare digitala a semnalelor s-a ales o placă de dezvoltare bazată pe DSP TMS320C6713 de la Texas Instruments (Spectrum Digital) datorită posibilităților multiple și a facilităților hardware oferite de aceasta în dezvoltarea aplicațiilor de prelucrare audio real-time și a altor aplicatii de uz general. Diagrama bloc a modulului DSK este redată în continuare fiind însorită și de o descriere a resurselor hardware de care dispune placa, dar și o descriere a performanțelor DSP-ului.



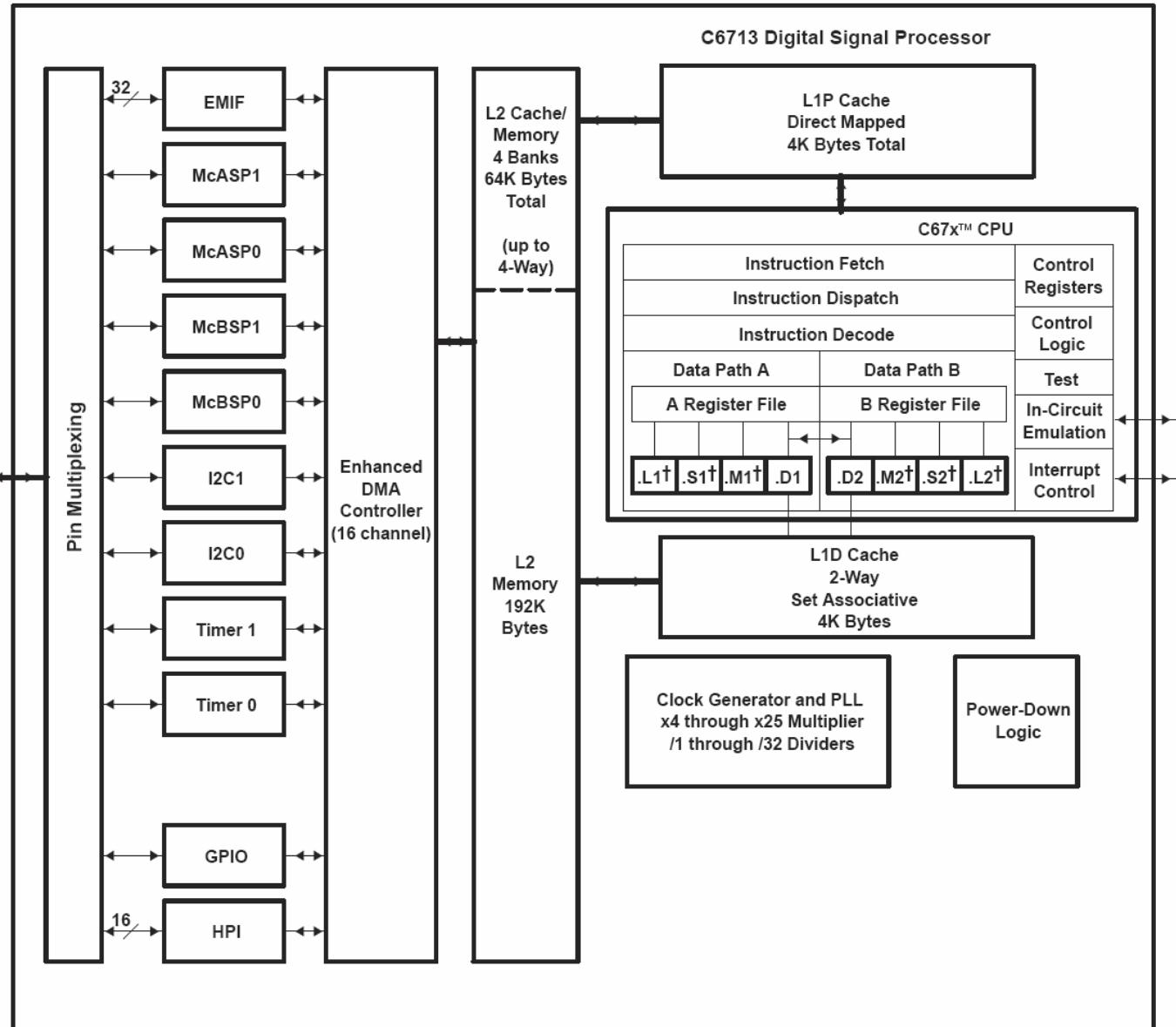
Principalele caracteristici, dispozitive on-board ale DSK și harta memoriei:

- DSP TMS320C6713 de 225MHz
- Codec Stereo AIC23
- 16 Mb SDRAM

- 512 Kb Flash
- 4 LED-uri și switch-uri DIP
- Opțiuni de boot configurabile
- Conector de extensie standard
- Emulator JTAG
- Sursă de alimentare de +5V



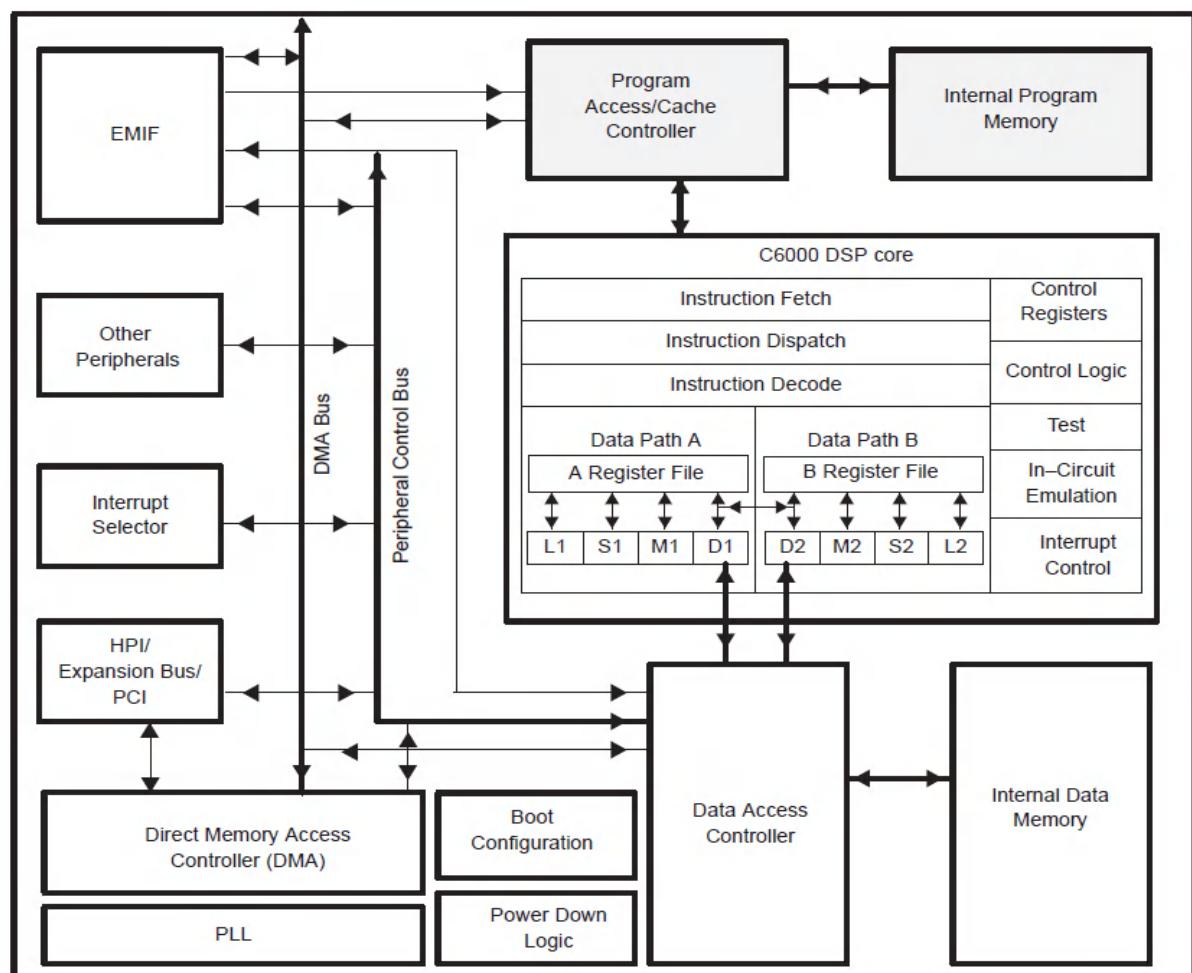
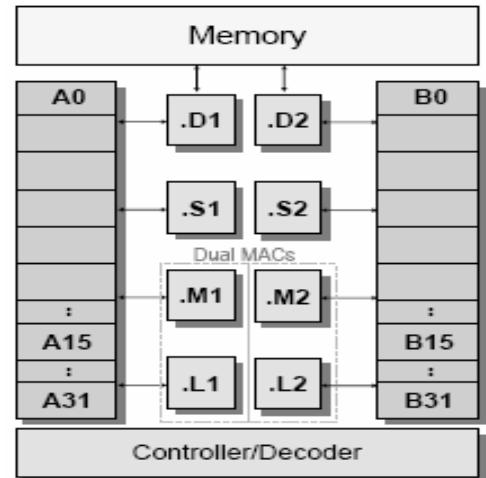
Datorită faptului că C6713 se bazează pe arhitectura VLIW de înaltă performanță dezvoltată de TI, DSK dedicat este o alegere excelentă pentru aplicațiile care implică lucrul multicanal și multifuncție. Operând la 225MHz DSP C6713 realizează 1350 MFLOPS ( Million Floating Point Operations Per Second), 1800 MIPS (Million Instructions Per Second) și 450 MMACS(Million Multiply Accumulate Cycles per Second). În continuare este prezentată arhitectura procesorului TMS320C6713 și se vor prezenta aspecte funcționale și de implementare care îl fac optim pentru implementarea aplicațiilor.



Acest CPU utilizează instrucțiuni VLIW cu dimensiune de 256 biți alimentând cu până la 8 instrucțiuni de 32 de biți cele 8 unități funcționale într-un ciclu de ceas. Pachetele de fetch au dimensiune fixă de 256 de biți, pachetele de execuție însă, au dimensiuni variabile, dar nu depășesc 256 biți, unitățile funcționale fiind alimentate cu instrucțiuni doar în cazul în care nu execută alte operațiuni în acel ciclu de ceas, acesta fiind un mecanism eficient de economisire a resurselor de memorie. Nucleul DSP bazat pe C67x CPU prezintă o unitate de instruction fetch din memoria cache de program L1, cu mapare directă. Pipeline-ul are fazele divizate în 3 etape (stagi): fetch, decode și execute. Etapa de fetch a pipeline-ului conține 4 faze : generarea adresei de program (PG), transmiterea adresei de program (PS), faza de așteptare acces program, când are loc o citire a memoriei (PW) și primirea pachetului de

program fetch de către CPU (PR). Fazele etapei de decode a pipeline-ului sunt instruction dispatch când pachetele de fetch sunt divizate în pachete de execuție(DP) și instruction decode pentru execuția în unitățile funcționale (DC). Etapa de execuție a pipeline-ului conține 10 faze. Cele două căi de date din CPU, data path A și data path B, fiecare conținând câte 4 unități funcționale. Fiecare set de unități funcționale conține un fișier registru (register file) și 4 unități. Un set conține unitățile funcționale .L1, .S1, .M1, .D1, iar cealaltă cale de date conține unitățile .D2, .M2, .S2, .L2, iar cele două fișiere registru conțin 16 registre de 32 de biți, pentru un total de 32 registre de 32 de biți pentru uz general.

Unitatea funcțională de tip .L este responsabilă de efectuarea operațiilor aritmetice și de comparare, operații logice pe biți, normalizare și operații de conversie în virgulă mobilă; unitatea funcțională de tip .S este utilizată la efectuarea operațiilor aritmetice în virgulă fixă, operații de shiftare, logice și comparații în virgulă mobilă, salturi, conversii, generare de constante, operații cu valori absolute și transferul de date în registre; unitatea funcțională de tip .M efectuează operațiile de multiplicare în virgulă fixă, mobilă sau mod mixt; unitatea funcțională de tip .D e responsabilă de calculul adreselor pentru modurile de adresare liniară și circulară specifice DSP și de operațiile de Load/Store.

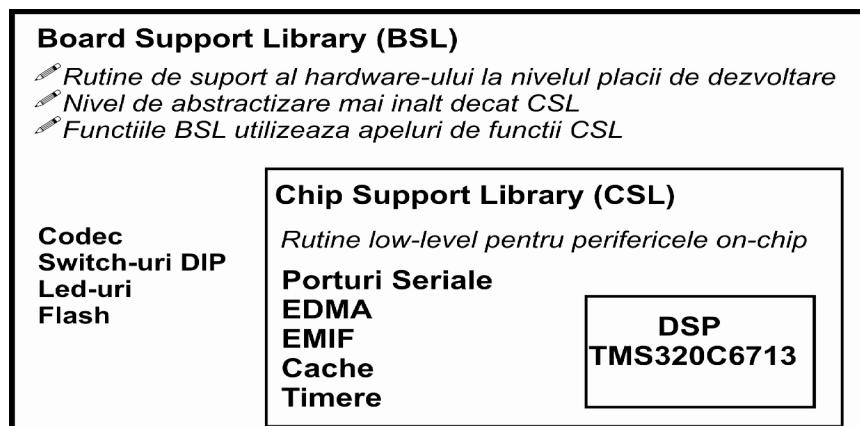


TMS320C6713 suportă un număr ridicat de periferice care sporesc funcționalitatea modului de dezvoltare. Astfel controller-ul DMA efectuează transferul de date între adrese ale locațiilor din memorie fără intervenția CPU și prezintă 4 canale programabile și un canal auxiliar. Controller-ul EDMA are aceeași funcție de bază ca și controller-ul DMA, dar spre deosebire de acesta prezintă 16 canale programabile, precum și un spațiu în RAM pentru stocarea configurațiilor multiple pentru transferurile viitoare. Portul HPI (host port interface) este un port paralel prin care un procesor gazdă poate accesa spațiul de memorie al CPU. Dispozitivul host are acces mai ușor întrucât este master pentru această interfață. Host-ul și CPU pot transfera date prin intermediul memoriei externe sau interne, în plus host-ul are acces direct la perifericele mapate în memorie. Magistrala de extensie (expansion bus) este un înlocuitor pentru HPI cât și o extensie pentru EMIF. EMIF oferă o interfață generică pentru extensii pentru SDRAM, SBSRAM, SRAM, ROM/Flash și dispozitive I/O. Extensia oferă două arii distințe de funcționalitate, host port și I/O port, care pot exista în același sistem. Chip-ul C6713 prezintă două timere de 32 de biți de uz general utilizate pentru evenimente temporizate, numărarea evenimentelor exterioare asincrone, generarea de pulsuri, generarea de întreruperi către CPU și emiterea de evenimente de sincronizare către controller-ele DMA/EDMA. Porturile McASP (2) conțin fiecare câte 8 pini pentru transfer serial care pot fi alocati pentru citire sau scriere, suportând TDM pe fiecare pin și transmisii de date de 192 KHz stereo. În plus pentru transmisie McASP poate fi programat pentru a emite date codate S/PDIF, IEC60958, AES-3, CP-430 pe canele multiple simultan, cu o singură memorie RAM care conține implementarea completă a datelor utilizator și câmpurile de status ale canalelor de transmisie. McASP oferă o interfață generică pentru ADC I2S multicanal, DAC și Codec-uri. În ceea ce privește McASP se va face o descriere mai amănunțită ulterior când se vor considera doar acele aspecte necesare în dezvoltarea aplicațiilor din cadrul laboratorului. Portul McBSP se bazează pe portul serial standard al dispozitivelor din gama DSP ale TI. Acest port are capacitatea de a asigura un buffer, pentru eșantioanele seriale, în memorie, automat prin intermediul controller-elor DMA/EDMA, având capacitați multicanal compatibile cu standardele de rețea T1, E1, SCSA și MVIP. McBSP oferă o interfață comună pentru portul de control SPI, pentru codec-urile de mare viteză cu TDM, pentru codec-urile AC97 și pentru EEPROM Serial.

Detaliile constructive ale arhitecturii au fost prezentate succint pentru a descrie modulul de dezvoltare ales pentru implementarea aplicațiilor din cadrul laboratorului. În continuare sunt descrise acele elemente specifice software de interfațare cu dispozitivele implicate în execuția aplicației și modul în care acestea oferă mecanisme de lucru eficient cu hardware-ul. Atenția se va concentra asupra primului nivel de abstractizare și anume kernelul DSP/BIOS, apoi pe funcțiile specifice BSL și CSL, care oferă o interfață de programare, EDMA, McASP și nu în ultimul rând McBSP.

## **Librăriile BSL/CSL și dezvoltarea de aplicații**

Următorul nivel prezentat este cel al interfețelor de programare C pentru dispozitivele de pe placă de dezvoltare, aici incluzând librăriile BSL și CSL.



Librăria CSL (chip support library) oferă o interfață în limbajul C pentru configurarea și controlul perifericelor din arhitectura bazată pe DSP (on-chip). Este compusă din module discrete care sunt construite (compilate) și arhivate într-un fișier librărie. Fiecare modul este destinat unui singur periferic, exceptie făcând doar unele module care oferă suport generic de programare pentru toate perifericele, cum ar fi, modulul de cerere de intrerupere IRQ (care conține API-ul pentru managementul intreruperilor) și modulul CHIP care determină setarea globală a chip-ului. Utilizarea CSL oferă un acces ușor la periferice prin ușurința de dezvoltare a aplicațiilor și timpul scurt de dezvoltare, oferă portabilitate, un nivel incipient de abstractizare hardware și un nivel de standardizare și compatibilitate între dispozitive. CSL pune la dispoziție un protocol de comunicație standard pentru programarea perifericelor on-chip, prin intermediul unor tipuri de date și macrouri specifice, însesnind configurarea și operarea cu perifericele, un manager generic de resurse, util în aplicațiile care utilizează periferice cu mai multe canale, în special în aplicațiile de procesare audio și nu în ultimul rând o descriere simbolică a perifericelor. În ceea ce privește arhitectura CSL, aceasta a fost proiectată astfel încât fiecare periferic este acoperit de un singur modul API, astfel avem un API pentru EDMA, un API pentru McBSP și.a.m.d. Cele mai importante module, care vor fi utilizate și în dezvoltarea aplicațiilor de laborator, din librăria CSL și fișierele header C corespunzătoare sunt următoarele: CACHE (modul cache, csl\_cache.h), CSL (modul de nivel înalt de inițializare a librăriei CSL, csl.h), EDMA (modul Enhanced Direct Memory Acces, csl\_edma.h), EMIF (modul external memory interface, csl\_emif.h), IRQ (modul controller de intreruperi, csl\_irq.h), McASP (modul multi-channel audio serial port, csl\_mcasp.h), McBSP (modul multi-channel buffered serial port, csl\_mcbsp.h), etc.

Funcțiile modului CACHE sunt utilizate pentru managementul cache-ului de date și program. Arhitectura cache-ului se bazează pe două nivele L1 și L2, descrise în prezentarea DSK.

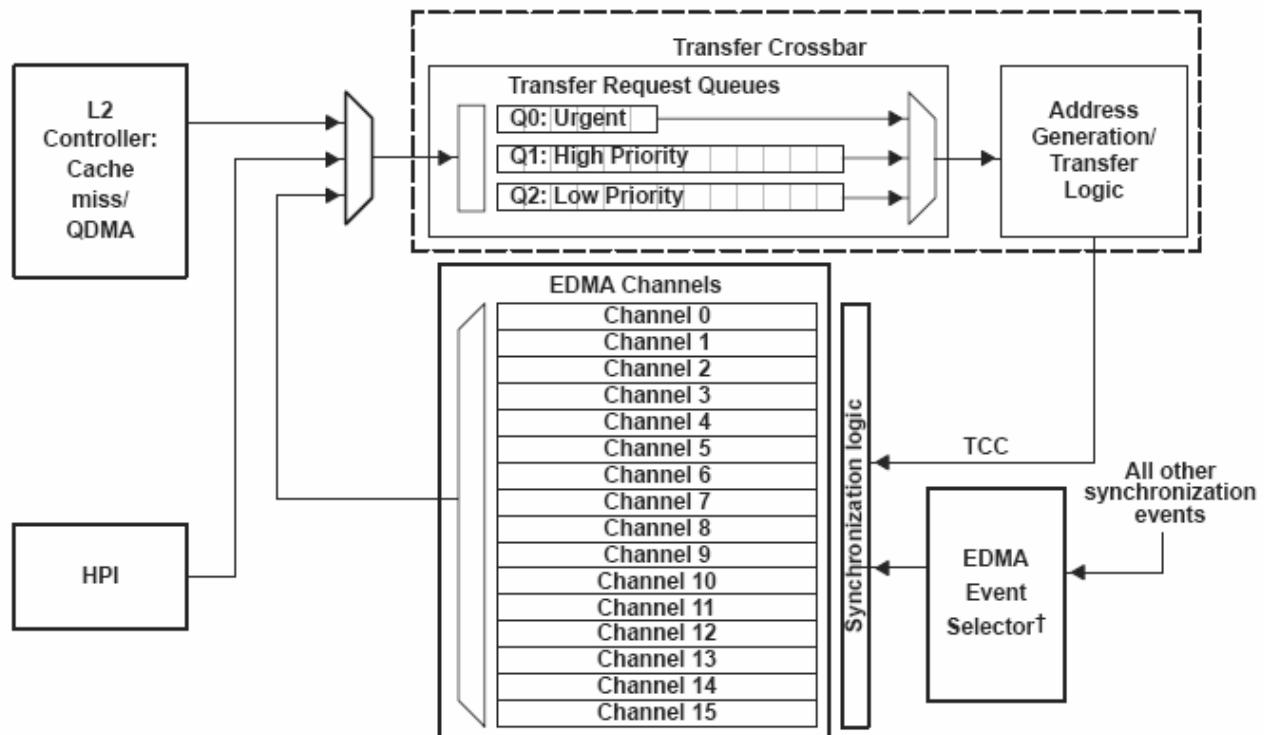
Librăria BSL oferă o interfață în limbajul C pentru configurarea și controlul tuturor dispozitivelor de pe placa de dezvoltare, DSK. Librăria este alcătuită din module discrete compilate și arhivate într-un fișier librărie. Granularitatea modulelor este proiectată astfel încât fiecare dispozitiv este acoperit de un singur API, exceptie făcând modulul portului I/O care este divizat în două module API, modulul LED și modulul DIP. Utilizarea BSL oferă caracteristici care ușurează dezvoltarea aplicațiilor și mențin portabilitatea, standardizarea și compatibilitatea între dispozitive. BSL are importanță și un mare impact în dezvoltarea și testarea algoritmilor.

## Controller-ul EDMA

Controller-ul enhanced DMA (EDMA) al TMS320C6713 este un motor de transfer de date cu eficiență sporită. Pentru maximizarea lărgimii de bandă, minimizarea interferențelor în transfer arhitectura acestui motor de transfer a fost proiectată conform unor obiective de performanță ridicate. Controller-ul EDMA poate fi considerat backbone-ul arhitecturii cu două nivele de cache, fiind utilizat pentru transferuri sincrone de date în background, transferuri inițiate de CPU, servirea portului HPI și servirea cache-ului. Acestea sunt principalele aplicații care ne interesează în contextul dezvoltării aplicațiilor propuse. Pentru început se face o descriere a arhitecturii EDMA-ului, mecanismele specifice de tratare a cererilor de transfer, alocarea cozilor de priorități, interacțiunea transferurilor multiple și rezolvarea problemelor de transfer. Apoi se va studia funcționalitatea EDMA, ISR ale CPU și un exemplu direct de transfer, în servirea perifericelor.

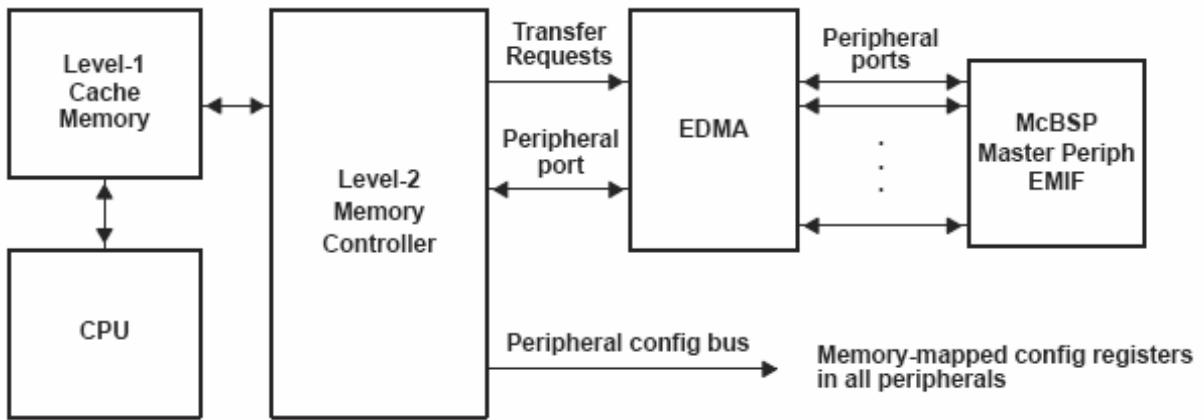
Controller-ul EDMA este un motor de transfer eficient capabil să trateze până la 8 bytes în fiecare ciclu EDMA, rezultând astfel 1800 Mbytes/s transfer de vîrf la frecvența CPU de 225 MHz. Motorul EDMA realizează toate transferurile între memoria cache L2, memoria externă, conectată prin intermediul interfeței de memorie externă EMIF și periferice. Aceste transferuri includ transferuri inițiate de CPU sau declanșate de eveniment, de accesuri ale perifericelor master, de servirea cache-ului și de accesuri la memoria non-cacheable. Datorită modalităților eficiente prin care transferurile de date interacționează, EDMA fiind proiectată să suporte transferuri multiple de mare viteză simultan, este posibilă crearea unui sistem eficient și prin care se maximizează lărgimea de bandă.

Fiecare transfer de date este inițiat printr-o cerere de transfer (transfer request - TR) care conține toate informațiile necesare pentru efectuarea transferului, incluzând adresa sursă, adresa destinație, prioritatea



transferului, numărul de elemente transmise. Cererile de transfer sunt sortate în cozi (buffere FIFO) cărora le sunt asociate priorități. Elementul curent în coadă, TR curent, este transferat în registrele cozii controllerului EDMA, care efectueză efectiv mutarea datelor definite de cererea de transfer. Toată procedura prin care cererea de transfer este propusă, configurarea cozilor după priorități și arbitrarea transferurilor au loc la viteza de lucru a EDMA și anume viteza de lucru a CPU, excepție făcând transferurile către periferice care au loc la viteza de lucru a perifericelor, care conțin porturi cu buffere de date. Emitenții de cereri de transfer aparțin DSP-ului și anume controller-ul de memorie cache L2, canalele EDMA și perifericele master.

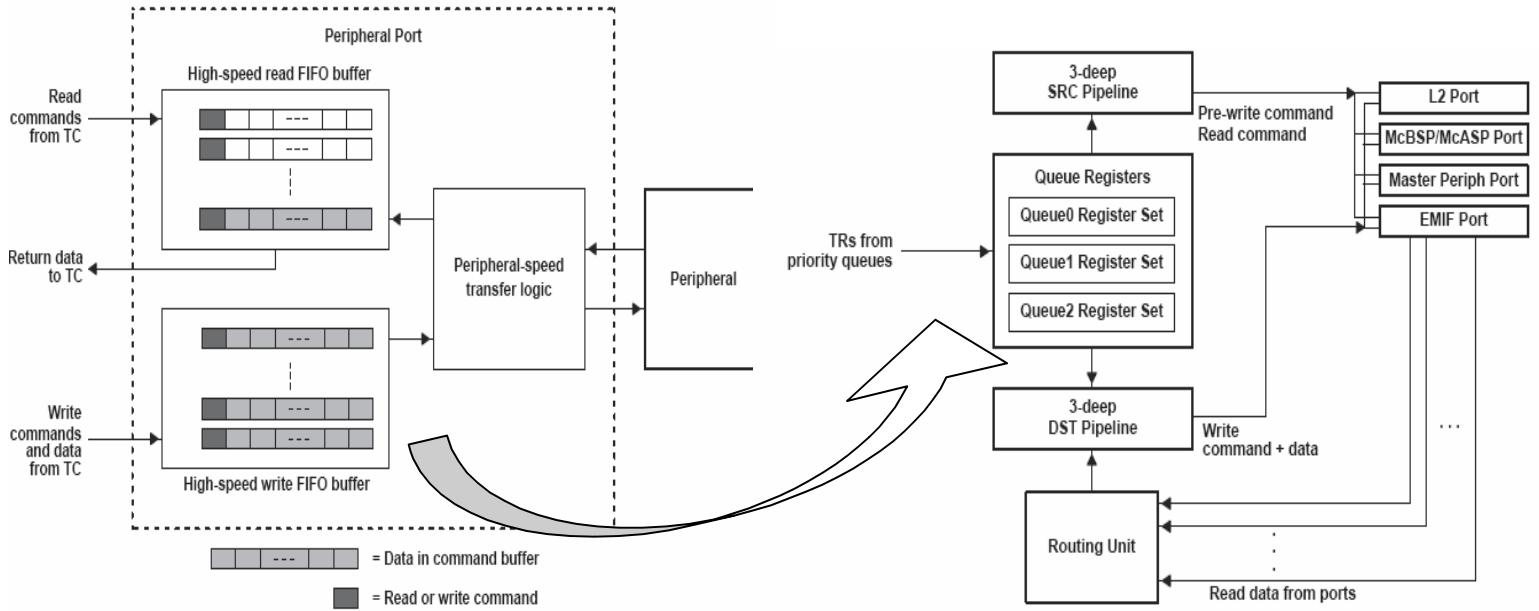
Controller-ul de memorie cache L2 are rolul de a trata accesurile de date la CPU, propune cererile de transfer pentru QDMA (quick DMA) și menține coerentă memoriei cache L1 și L2, toate liniile de comunicație între blocul CPU și restul dispozitivelor trecând prin acesta.



Controller-ul L2 direcționează cererile QDMA și accesurile externe la memorie către EDMA, cererile și accesurile la cache L2 către memoria L2, iar accesurile la registrele de control mapate în memorie către magistrala de configurare a perifericelor. Controller-ul L2 tratează toate cererile de transfer cache prin introducerea lor în coada de prioritate maximă (urgent priority queue) care este rezervată în acest sens. Transferurile QDMA pot fi setate pentru priorități mari sau scăzute funcție de anumiți biți din registrul de opțiuni de transfer QDMA.

QDMA-ul oferă unul din cele mai eficiente moduri de a muta date, suportând aproape toate modurile de transfer ca și EDMA, dar supune cererile de transfer mai rapid. În ceea ce privește inițializarea transferului QDMA, acesta necesită doar 1/5 cicli CPU pentru supunerea cererii, depinzând de numărul de registri care trebuie configurați. Toate transferurile QDMA sunt supuse (submitted) utilizând sincronizarea cadrelor (frame synchronization 1D) sau sincronizarea blocurilor (block synchronization 2D) și similar canalelor EDMA, QDMA poate prezenta priorități programabile la nivele joase de transfer. Cele 16 canale EDMA pot fi configurate cu ajutorul unor valori stocate într-un parameter RAM (PaRAM), fiecare canal corespunzând unui eveniment de sincronizare specific sau unui eveniment de sincronizare programat în selectorul de evenimente al EDMA. Structura bazată pe RAM a EDMA-ului oferă un grad sporit de flexibilitate, fiecare canal având un set complet de parametrii accesibil prin magistrala de configurare periferice, transferul parametrilor fiind independent între canale. Selectorul de evenimente al EDMA realizează selectia evenimentului asociat cu fiecare canal EDMA, permitând astfel fiecarui canal să fie sincronizat cu orice eveniment EDMA printr-un set de registre propriu. Perifericul (Master Peripherals) prezent în alcătuirea TMS320C6713 DSP este HPI (Host Port Interface), acesta realizând transferuri de date cu orice locație a hărții de memorie a DSP-ului cu prioritate maximă, fixată, fără intervenția utilizatorului printr-o conexiune directă la EDMA. Emitenții de cereri de transfer la EDMA sunt conectați cu controller-ul de transferuri prin intermediul magistralei de cerere de transfer, care se bazează pe priorități, dar cu tempi de arbitraj scăzuți. Controller-ul de transferuri este componenta EDMA care procesează cererile de transfer și execută mutarea efectivă a datelor. Aceasta se bazează pe cozi de așteptare, care introduc și trei nivele de priorități, Q0 (urgent, rezervat pentru cererile de transfer ale cache-ului), Q1 (prioritate nivel înalt), Q2 (prioritate nivel scăzut) și care se bazează și pe trei seturi de registre pentru monitorizarea evoluției transferului. Porturile perifericelor implicate în trafic de date de mare viteză (McASP/McBSP, HPI, EMIF, Controller L2) acceptă comenzi de la controller-ul de transferuri, accesând direct perifericele, eliberând EDMA-ul pentru tratarea altor transferuri până la primirea răspunsului unui periferic, oferind astfel posibilitatea transferurilor simultane de la și către periferice cu priorități diferite. În figură sunt prezentate modul în care porturile

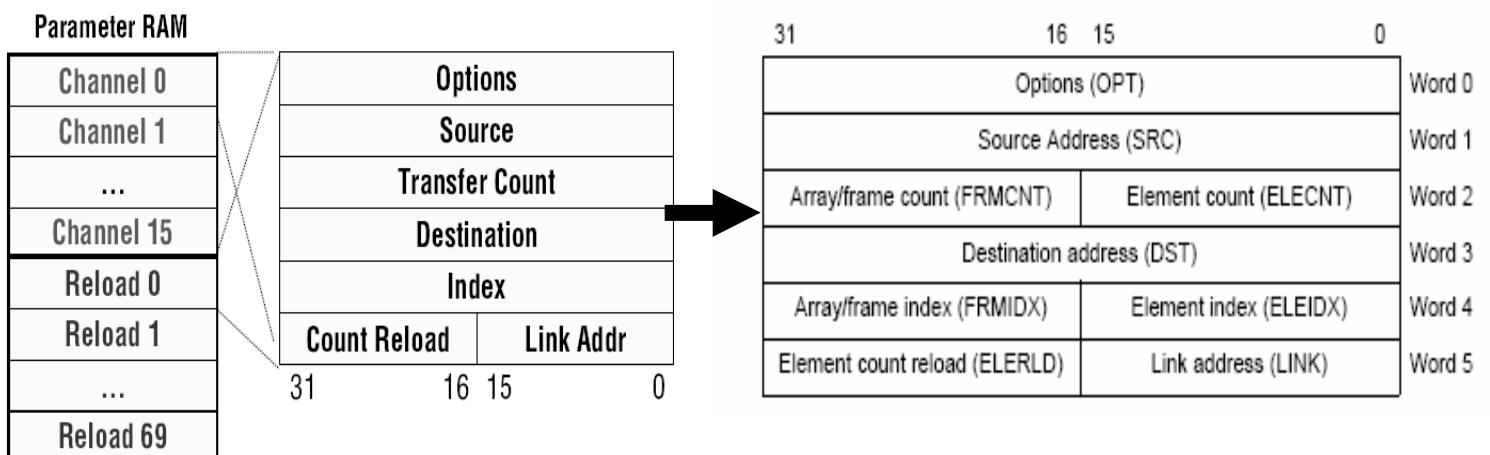
perifericelor acceptă comenzi de la controller-ul de transferuri și configurația seturilor de registre ale cozilor, magistrala de comenzi și unitatea de dirijare a comenzilor.



Pentru efectuarea unui transfer controller-ul de transferuri transmite comenzi către porturile sursă și destinație pentru efectuarea citirilor/scrierilor. Controller-ul transmite comenzi porturilor pentru efectuarea transferurilor de date însă datele sunt mutate doar când portul este liber, activitatea controller-ului de transfer nefiind perturbată de timpii de aşteptare întrucât cozile care primesc cereri de transfer de la porturi diferite le tratează simultan, iar în cazul în care se emit mai multe comenzi pentru același port acestea sunt arbitrate în concordanță cu prioritatea. Pentru inițierea transferului controller-ul de transfer al datelor emite o comandă pipeline-ului sursei sau destinației. Există 3 comenzi pre-write, read și write, protocolul bazându-se pe confirmări, procesul de arbitrage a pipeline-urilor fiind efectuat individual de către controller. După primirea confirmării de la destinație datele sunt citite la frecvență maximă a sursei în bufferul de comandă și apoi transmis, împreună cu o comandă write a portului, unității de dirijare a EDMA-ului pentru a fi transmis ulterior destinației. Un aspect important este cel al cunoașterii momentului și modului în care cererile de transfer sunt supuse (submitted), pentru a implementa mecanismul de scheduling al traficului de date în sistem. Tiparul unei cereri de transfer este comun conținând aceleași informații esențiale: adresa sursă și destinație, numărul de elemente transmise și relația între elementele din sursă și destinație (fixed, increment, decrement, indexed).

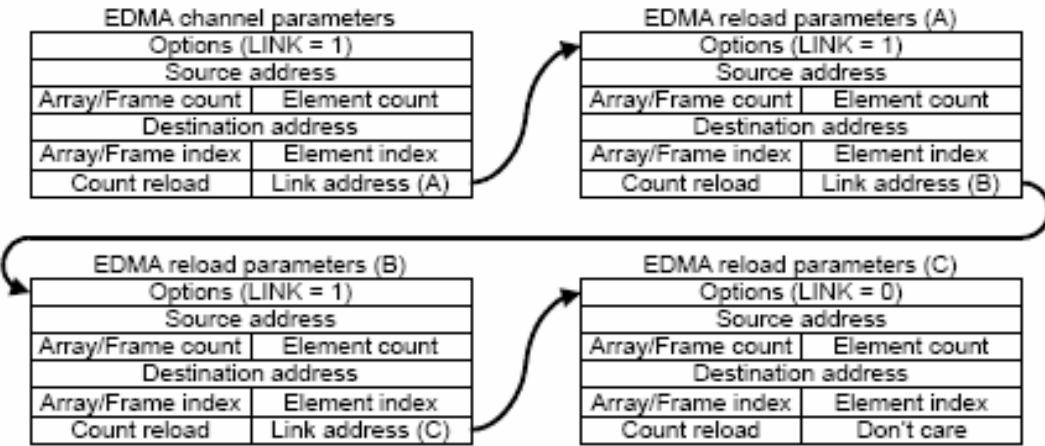
Alocarea cozilor de priorități necesită o atenție sporită în implementare întrucât se dorește prevenirea ca un emitent de cerere de transfer să inunde cozile de priorități, fiind astfel necesară limitarea numărului de cereri de transfer active la momentul curent, supuse de către un emitent. Un aspect care rezultă din această problemă a alocării este problema interacțiunii transferurilor și a arbitrării transferurilor și managementul acestora în vederea maximizării performanțelor obținute. Transferurile în interiorul EDMA necesită arbitrage la nivelul nodurilor de emitenti de cereri de transfer date, în cozile de priorități și în timpul procesării active în registrele cozilor. Transferurile se fac prin intermediul cuvintelor de 8-, 16- sau 32 de biți sau prin intermediul blocurilor de date 1D sau 2D. Astfel al nivelul nodurilor de emitere de cereri de transfer dacă un emitent de cerere este inactiv, nu se mai emit cereri, în caz contrar cererile de transfer sunt supuse în ordinea sosirii, în cazul simultaneității utilizându-se un model round-robin de supunere a cererilor. În cazul cozilor de priorități arbitragea este simplă, procesarea cererilor

supuse unei singure cozi fiind tratate serial, în ordinea sosirii; în cazul regiszrelor cozilor este o abordare mai specială întrucât fiecare nivel de prioritate posedă un set de regisztri de coadă care când sunt gata de servire a unui transfer, extrag o cerere de transfer din coada de priorități și începe procesarea. În anumite circumstanțe un transfer de prioritate scăzută va apărea ca precedând un alt transfer pe care utilizatorul îl creditase cu o prioritate mai ridicată, datorită unei serii de scenarii cum ar fi, blocarea porturilor, transferurilor multiple de prioritate ridicată, blocarea cererilor de transfer sau paralelismul de citire/scriere. Efectele acestor scenarii negative pot fi minimizate sau chiar evitate prin setarea potrivită a fluxului de date din sistem. Acest lucru se poate efectua utilizând un scheduling potrivit, dând o definire corectă a traficului de date din sistem (generarea cererilor de transfer, stagiile operațiilor de transfer, porturile sursă/ destinație, duratele transferurilor și latențele, periodicitate și deadline-uri) și sintetizând informațiile obținute în analiza traficului din sistem în implementări care să țină cont de durata transferurilor și interferențe, de partajarea largimii de bandă a porturilor periferice, planificarea temporizată și să fie în concordanță cu prioritățile și prioritatea în alocare. În continuare sunt tratate posibile aplicări ale EDMA-ului, în ceea ce privește transferurile sincrone de date de background, transferurile inițiate de CPU, servirea portului host și servirea cache-ului, subliniind aspectele funcționale. Astfel transferurile sincrone de date de background sunt configurate cu ajutorul unei memorii RAM de parametri on-chip (PaRAM), fiecare din cele 16 canale disponibile corespunzând unui eveniment de sincronizare specific care declanșează transferul. PaRAM-ul este alcătuit din parametrii canalelor și un set de parametri reîncărcați ai canalelor, după cum se poate observa în figura următoare. Fiecare canal EDMA conține următorii parametri care trebuie să fie configurați pentru un transfer valid: Options (setările de configurare a transferului), Source address (locația de memorie de unde sunt transferate elementele), Destination address (locația de memorie unde sunt transferate elementele), Transfer Count (care combină Array/frame count + Element count , care redau numărul de array-uri dintr-un frame , respectiv numărul de elemente dintr-un array), Index (Array/frame index + Element index, ce redau offset-ul utilizat la calculul adresei de început a fiecărui array sau frame, respectiv spatierea între adresele elementelor dintr-un frame), Count Reload (valoarea de reîncărcare pentru numărul de elemente din frame), Link Address (adresa din PaRAM a parametrilor care să fie încărcați la finalizarea transferului curent).



În cazul transferurilor de date inițiate de CPU transferul efectiv are loc în timpul operării sistemului printr-un set de registre mapate în memorie, referite ca registre ale QDMA, care după cum am precizat anterior este un canal EDMA adițional care este sincronizat de către CPU. Parametrii QDMA sunt identici cu cei ai EDMA cu excepția faptului că nu prezintă Element Count și Link Address.

Servirea portului host este realizată fără intervenția utilizatorului, printr-un canal care nu este configurabil direct, în schimb servirea cache-ului prezintă o altă abordare întrucât controller-ul de cache L2 inițiază transferurile prin EDMA. EDMA servește în acest caz cache miss-urile, data flush-urile către locația de memorie fizică și accesurile la memoria noncacheable, acestă funcționalitate neavând capacitatea de a fi programată. În contextul funcționalității implementate de EDMA putem considera transferul blocurilor de date între diferite locații de memorie (canalele putând fi configurate astfel încât să acceseze orice locație din harta de memorie a dispozitivului DSP), servirea continuă a portului serial McBSP și realizarea unei paginări a programului/datelor din memoria externă în memoria internă L2 SRAM. Toate accesurile la memoria externă trec prin interfață de memorie externă EMIF, aceasta oferind o interfață generică pentru SDRAM, sync-burst SRAM și alte memorii asincrone. Porturile McBSP sunt singurele periferice on-chip care necesită uzul EDMA. Fiecare McBSP are un registru de primire date, un registru de transmitere date (care sunt registre mapate în memorie), un semnal de eveniment de recepție și un semnal de eveniment de transmisie (evenimentele fiind setate în momentul în care datele sunt transferate prin semnalul de eveniment de recepție sau transmisie).

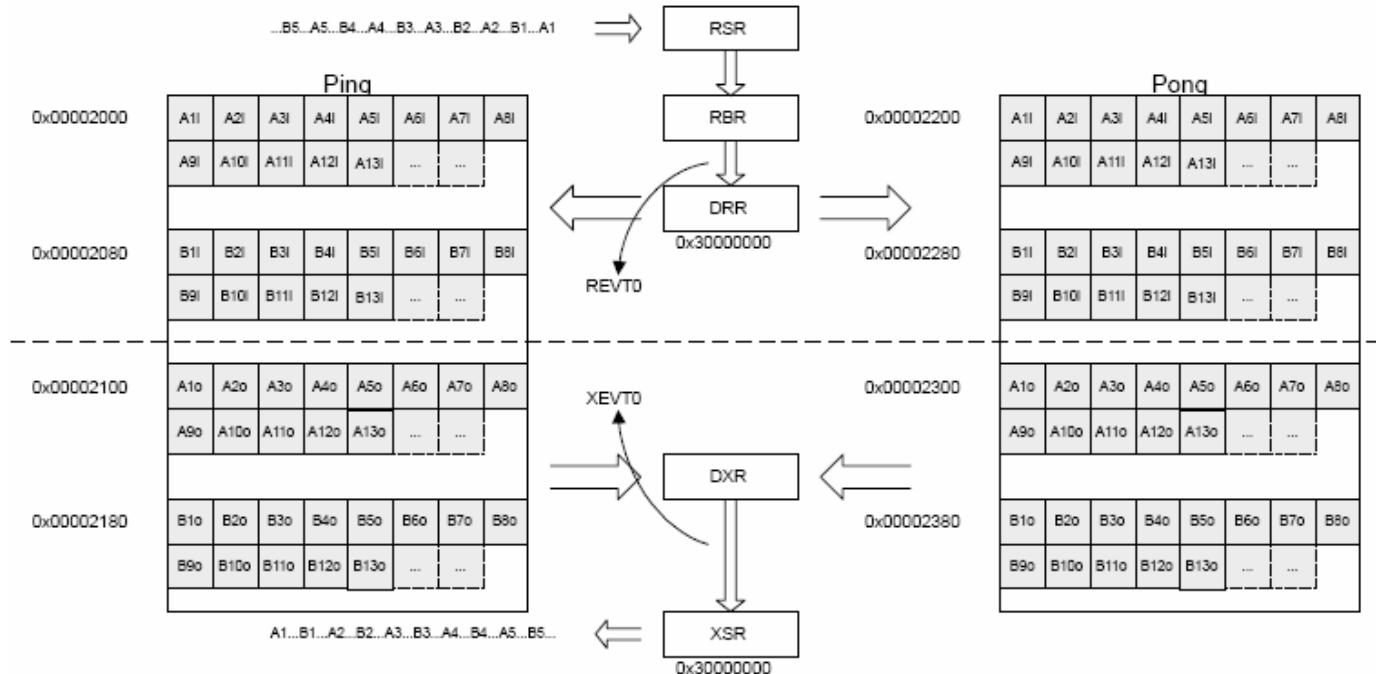


În ceea ce privește sincronizarea elementelor și array-urilor, totul se bazează pe recepția evenimentelor de sincronizare, când un subframe este transmis, dimensiunea acestuia fiind definită de a doua dimensiune a transferului. Sincronizarea frame-urilor și blocurilor are loc cu primirea unui eveniment de sincronizare pentru care se realizează un transfer valid. Un alt aspect important este legat de transferurile dimensionate, întrucât transferurile efectuate de către EDMA au dimensiune, fie unidimensionale (1D), fie bidimensionale (2D). În cazul transferurilor 1D frame-urile sunt alcătuite dintr-un număr de elemente individuale, suportându-se transferuri de căte un element (sincronizat) odată sau căte un frame (sincronizat) odată. Pentru transferurile 2D fiecare bloc de date pentru transmisie are două dimensiuni și anume, numărul de array-uri din bloc și numărul de elemente dintr-un array, transferurile efectuându-se fie căte un array (sincronizat) odată, fie căte un frame (sincronizat) odată. Transferurile de orice tip necesită și un update al adresei, realizat fie prin parametrii setați ai unui canal al EDMA-ului, fie prin logica de generare/transfer de adresă. Linking-ul transferurilor este o capacitate importantă a EDMA, care se realizează prin primirea unei adrese de legătură (link address) și prin setarea opțiunii LINK=1, atunci un canal EDMA încarcă o nouă intrare din PaRAM și începe execuția noului transfer. În figură este surprins un scenariu posibil.

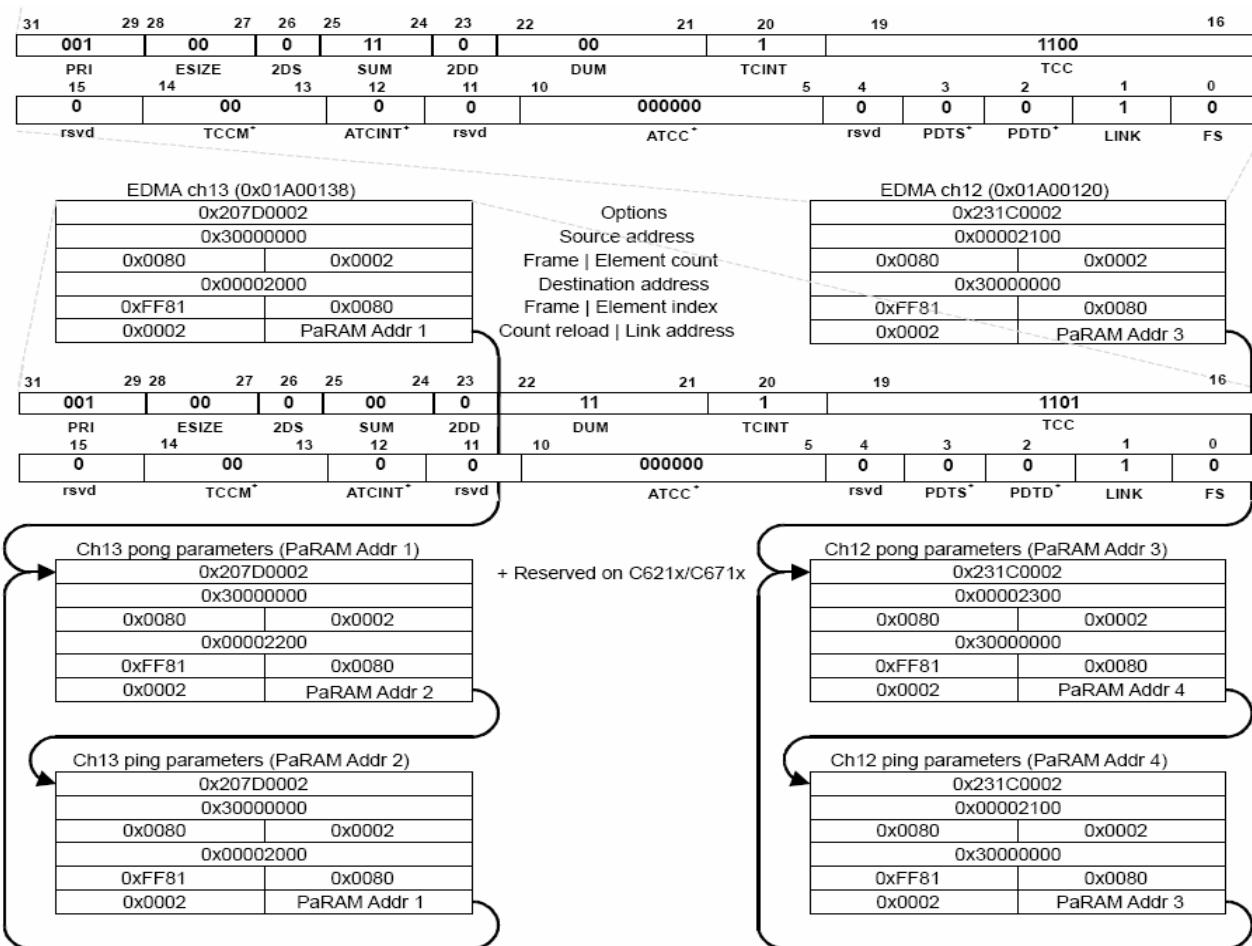
O altă funcționalitate a EDMA este înlănțuirea care oferă posibilitatea ca un transfer EDMA să fie sincronizat de finalizarea altui transfer EDMA (un scenariu posibil fiind în cazul în care un canal EDMA

primește un frame de date de intrare și apoi automat declanșează un frame de date de ieșire care să fie transmis imediat după) sau posibilitatea ca un canal EDMA să se auto-sincronizeze prin traversarea propriei liste înlanțuite de transferuri.

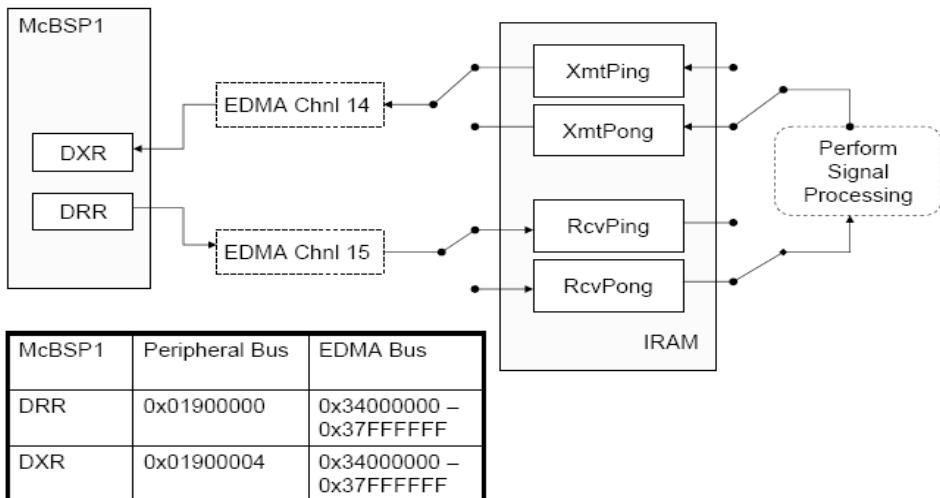
Întrucât unele din aplicațiile care se vor dezvolta impun transferuri utilizând EDMA este necesara prezentarea unui scenariu de bază pentru un model de transfer de date specific pentru acestă arhitectură în ceea ce privește servirea perifericelor (în speță McBSP). O capacitate importantă a EDMA, amintită și mai sus, este abilitatea de a servi perifericele fără intervenția CPU. Prin intermediul unei inițializări corecte canalele EDMA pot fi configurate pentru a servi continuu perifericele on-chip și off-chip în timpul operării dispozitivului, fiecare eveniment disponibil EDMA-ului având propriul său canal dedicat, canalele operând simultan. Programând un canal EDMA pentru servirea unui periferic presupune cunoașterea modului de prezentare a datelor către DSP. Datele sunt oferite mereu alături de un eveniment de sincronizare, fie sub forma unui element pe eveniment (non-bursting), fie mai multe elemente pe eveniment (bursting). Am ales pentru exemplificare, modelul cu buffere ping-pong (ping-pong buffering). Deși configurația de operare continuă pentru servirea McBSP permite EDMA să servească perifericul continuu, apar anumite restricții către CPU. Din moment ce bufferele de intrare și ieșire sunt umplute/golite în continuu, pentru ca CPU să proceseze datele, transferurile trebuie să se realizeze la viteza de lucru a EDMA. Datele de recepție ale EDMA trebuie stocate în memorie înainte ca CPU să le acceseze, iar CPU trebuie să ofere datele de ieșire înainte ca EDMA să le transfere. Această provocare devine mai pronunțată datorită faptului că avem un sistem cu două niveluri de cache. O tehnică simplă care permite activității CPU să se distanțeze de activitatea EDMA rezidă în implementarea buffer-elor ping-pong. Aceasta presupune că avem seturi multiple (de obicei două) de buffere de date pentru toate fluxurile de date de intrare/ieșire. În timp ce EDMA transferă date în și din bufferele ping, CPU manipulează datele din bufferele pong. Când activitățile CPU și EDMA se încheie, operează invers, EDMA scriind peste vechile date de intrare și transferând noile date de ieșire. Un exemplu de prelucrare ping-pong pentru date ale McBSP este prezentat în continuare.



Utilizând acest model de prelucrare în momentul în care un transfer este finalizat, canalul respectiv încarcă intrarea pentru celălalt transfer și astfel transferul de date continuă. Configurarea canalului de transmisie , parametrii EDMA pentru transferul ping-pong și seturile de parametrii de configurare ping-pong sunt descrise în continuare. Canalul EDMA este încărcat inițial cu parametrii de configurare ping. Adresa de legătură pentru intrarea ping (intrarea în tabela parametrilor) este setată cu offset-ul PaRAM a setului de parametri pong și invers. Opțiunile canalului, valorile de numărare și valorile index sunt identice între parametrii ping și pong pentru fiecare canal, diferind doar adresa de legătură și adresa buffer-ului din memoria internă. Pentru a utiliza tehnica bufferelor ping-pong este nevoie de a semnala CPU-ul momentul în care acesta poate începe accesarea noului set de date (se impune deci o sincronizare cu CPU-ul). După terminarea procesării unui buffer de intrare (ping) CPU-ul așteaptă ca EDMA-ul să-și încheie operarea înainte de a trece la bufferul alternativ (pong).



Următoarea diagramă este un schelet de proiectare care poate fi utilizat la prelucrarea semnalelor utilizând buffere duble, pentru procesarea blocurilor de date. Se realizează o initializare a două canale EDMA, de la McBSP la un buffer intern și de la un alt buffer intern la McBSP. După transferul unui bloc de date este emisă o intrerupere.



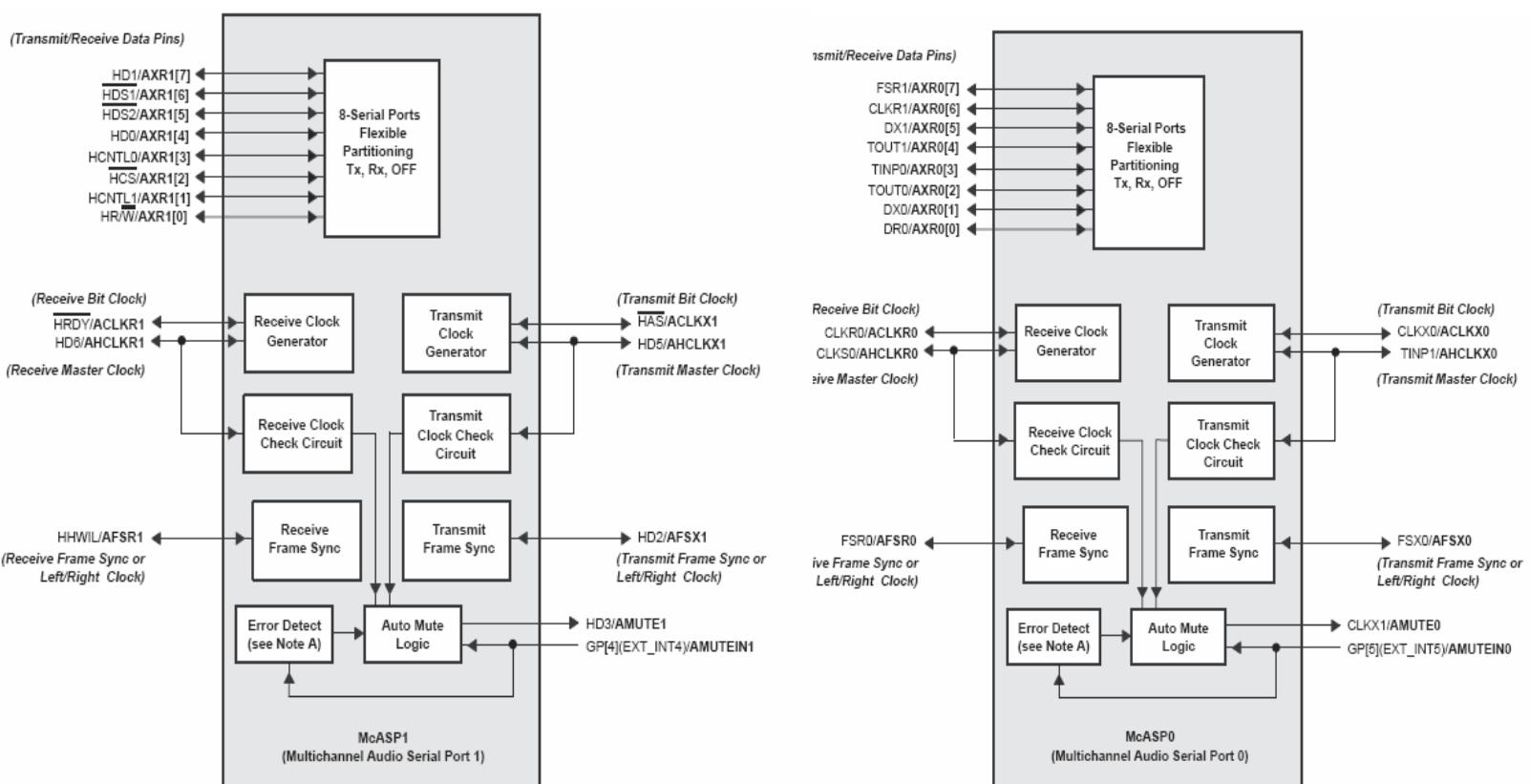
Motorul de transfer de date EDMA, este componenta primară a arhitecturii cu două nivele de cache. Efectuează servirea cache-ului, a host port-ului și a transferurilor de date programate de utilizator. Toate canalele EDMA, plus un set de registre QDMA sunt programabile pentru a efectua transferuri de date în background față de operarea CPU, a cărui implicare este minimă. Astfel CPU-ul se concentrează doar pe partea de procesare a datelor, managementul fluxurilor de intrare/ ieșire de la și către DSP fiind asigurat, în background, de către EDMA.

## Porturile McASP și McBSP

Portul serial audio multicanal McASP funcționează ca un port audio generic optimizat pentru aplicații audio multicanal. Acesta este util pentru TDM (time division multiplexing), pentru implementarea protocoalelor I2S și transmisii digitale audio (DIT – digital audio interface transmission). McASP este alcătuit din secțiuni de transmitere și recepționare care pot opera sincronizate, sau complet independent cu CLK separat, cu CLK de bit și sincronizări de cadre, utilizând moduri de transmisie cu diferite formate de streaming. Modulul mai conține și 16 serializatoare care pot fi activate individual pentru transmisie/recepție, cu posibilitatea operării și configurării ca pini GPIO (general purpose input/output). Deși arhitectura McASP este interesantă de studiat (dar depășește cadrul laboratorului) și importantă în dezvoltare, se vor prezenta în continuare doar acele aspecte de funcționalitate și caracteristici necesare în dezvoltarea de aplicații în cadrul laboratorului. McASP include două module de generare de CLK pentru transmisie și recepție, permitând transferuri cu rate variabile. Datorită faptului că prezintă module de Tx și Rx diferențiate care conțin CLK programabil și generator de sincronizare pentru cadre, permite emisiei de semnale TDM cu 2, 32, 384 cuante de timp până la 32 de biți și capacitatea de formatare a datelor pentru manipulare la nivel de bit.

Pinii de date seriale sunt asignabili individual, cu posibilitatea conectării la ADC audio, DAC, codecuri, receiver audio cu interfață digitală (DIR) și componente de transmisie la nivelul nivelului fizic al protocolului de comunicație, S/PDIF și varietăți ale formatului I2S. La nivelul sistemului poate fi configurat în diverse tipuri de conexiuni, decoder audio digital, amplificator digital, encoder audio digital, procesor digital cu 16 canale utilizând 8 ADC bi-canali. În continuare este prezentat succint modul de operare al McASP. Prima etapă este pregătirea și inițializarea, care se desfășoară pe parcursul a mai mulți pași. Primul pas este inițializarea secțiunii de transmitere/recepție, și se configerează

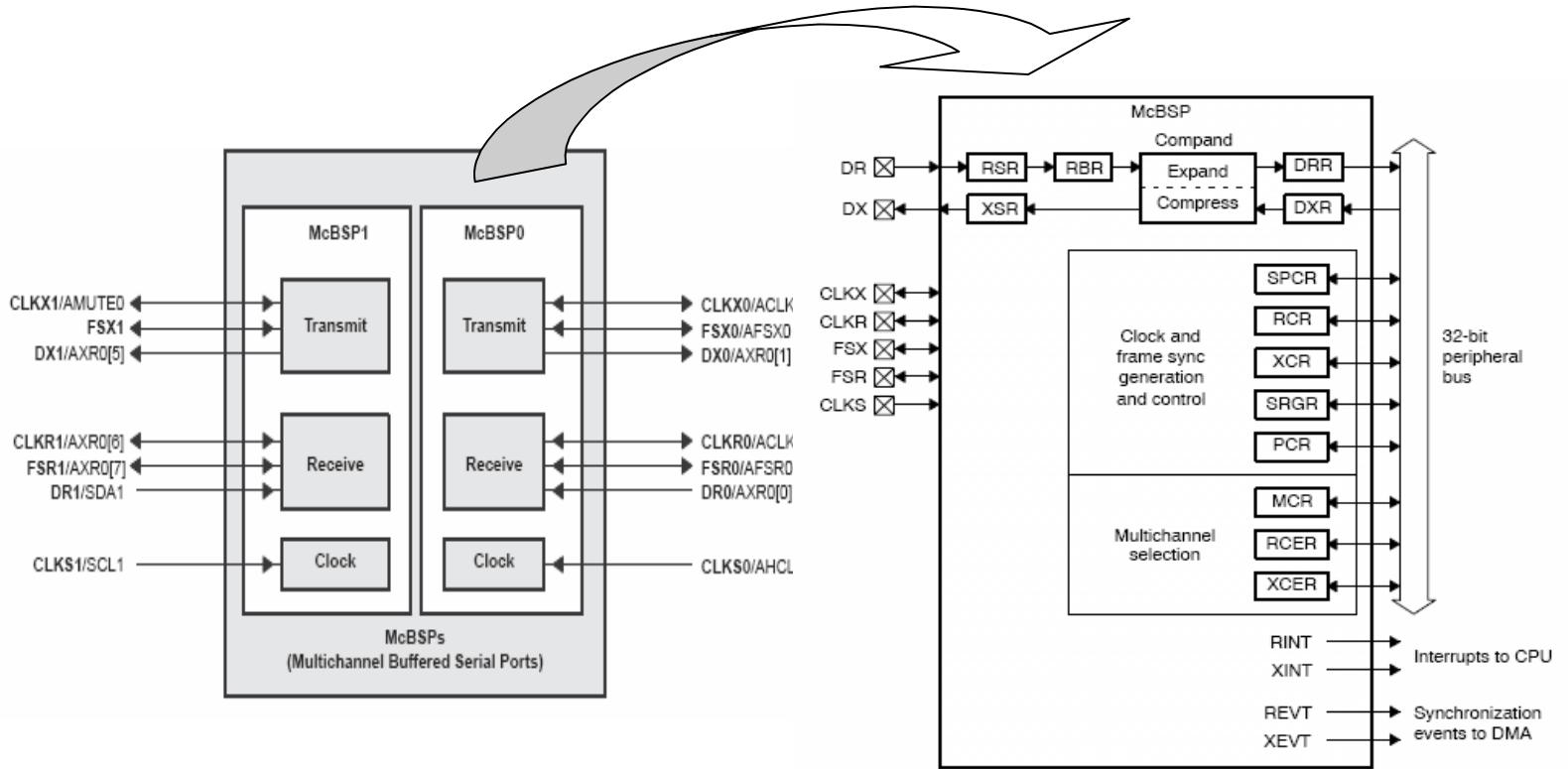
registrele McASP ; primul se configurează registrul de management de power down și emulare, apoi se configurează registrele de recepție și registrele de transmisie, care trebuie să fie active pentru a asigura sincronizarea, configurarea registrelor de serializare și registrele globale și cele pentru operarea DIT. Al treilea pas este pornirea CLK-urilor seriale, urmat apoi de pregătirea achiziției de date, care necesită ca în cazul în care EDMA este utilizat pentru servirea McASP pregătirea achiziției datelor dorite și pornirea EDMA, până la revenirea McASP din reset. În cazul în care pentru servirea McASP este folosită o întrerupere CPU, se declanșează întreruperi și pentru transmisie și recepție. Următorul pas este activarea serializatoarelor, verificându-se apoi dacă toate bufferele de transmisie sunt servite și se eliberează mașinile de stare din reset. Apoi se eliberează din reset și generatoarele de sincronizare pentru cadre, transferul efectiv McASP începând în timpul primului semnal de sincronizare cadru. Operațiile de transmisie și recepție se pot realiza fie sincron, fie asincron, existând mai multe moduri de transfer (în rafală – Burst transfer mode, TDM transfer mode, digital audio interface transmit (DIT) transfer mode). Transmisia și recepția datelor se fac ținând cont de anumite flaguri și de conținutul anumitor registre din structura McASP, verificându-se elemente ca statusul data ready și generarea de evenimente sau întreruperi. Transferurile suportate de McASP se pot realiza fie prin intermediul portului de date (data port - DAT), fie prin magistrala de configurare (configuration bus - CFG), fie prin utilizarea CPU pentru servirea McASP prin intermediul întreruperilor, fie prin utilizarea EDMA pentru servirea transferurilor de date ale McASP, acesta fiind scenariul tipic. Structura generică a celor două module McASP din alcătuirea TMS320C6713, este prezentată în figură.



Unitatea de formatare a datelor pentru McASP suportă mai multe formate seriale de date. În ceea ce privește fluxul de transmisie date sau recepție date cele două unități de transmisie și recepție suportă formatarea serială a datelor și există astfel posibilitatea de a forma datele în formatul slot (sau time slot cu până la 32 de biți), formatul word, cu left/right alignment și LSB/MSB. Din punct de vedere al

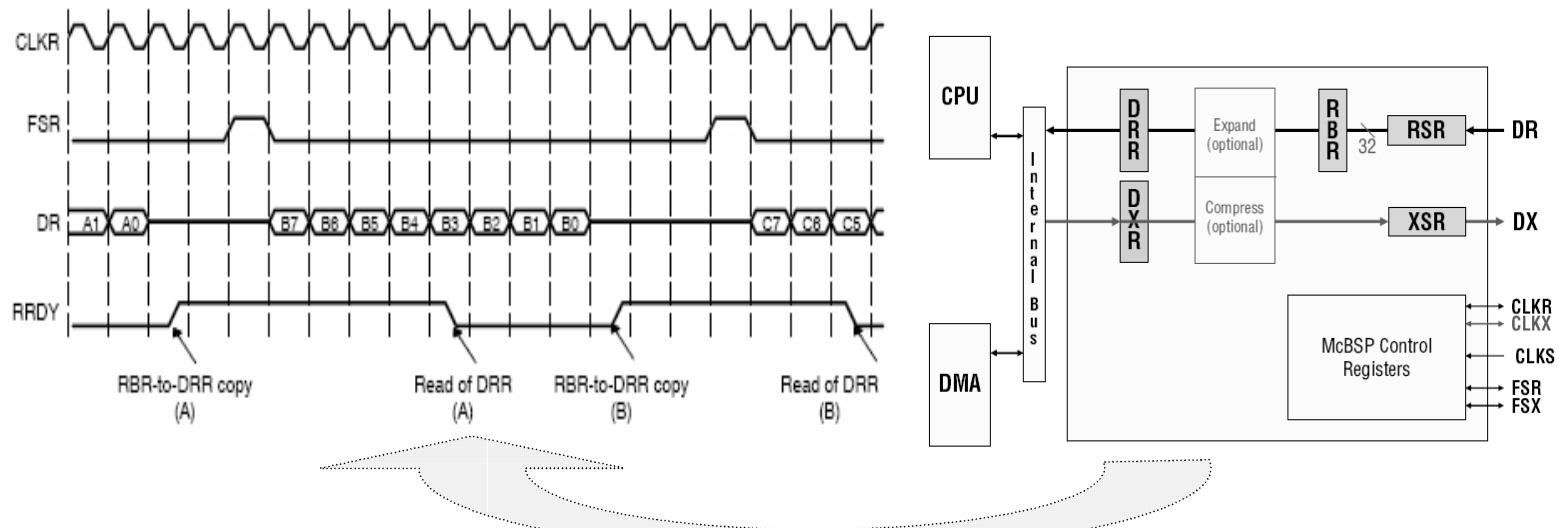
întreruperilor sunt generate de evenimente ce determină declanșarea îնtreruperii prin recunoașterea unui cadru de sincronizare de transmisie sau recepție, relativ la status-ul unor registre, în ambele direcții, având îнtreruperi de transmisie date și îнtrerperi de recepție date. Erorile generează îнtreruperi, acest lucru fiind semnalizat de registrul de status pentru receptia/transmisia de date. Acestea sunt determinate de evenimente ca overrun-ul receptorului, analog underrun-ul transmisiei, de receptia/transmiterea unui cadru de sincronizare neașteptat, eroare a CLK de receptie/transmisie sau eroare EDMA la transmisie/recepție. Întreruperilor generice le sunt atașate anumite proprietăți cum ar fi, proprietatea de Active IRQ, proprietatea de Outstanding IRQ și Serviced IRQ. Mecanismul de detecție a erorilor specific McBSP detectează situațiile de overrun, underrun, sincronizări de cadre timpurii/întârziate, erori DMA și starea de intrare externă mută.

În continuare este descris modul de operare a portului serial multicanal McBSP aparținând procesorului TMS320C6713. Portul McBSP oferă următoarele funcții : comunicare full-duplex ; registre de date dublu-bufferate, ce permit operarea cu un flux continuu de date, CLK și framing independent pentru Rx și Tx ; interfață directă pentru codec-urile uzuale și pentru chip-urile de interfață analogică (AIC) și alte dispozitive ADC sau DAC conectate ; CLK intern pentru transferurile de date. În plus portul are următoarele capacitați de interfațare directă cu unități de framing T1/E1, MVIP, SCSA, H.100 ; interfață pentru dispozitivele compatibile IOM-2, AC97, I2S și SPI (serial peripheral interface). McBSP este capabil să efectueze operații de Rx și Tx cu până la 128 de canale, cu date de dimensiuni variabile, opțiuni de LSB sau MSB first și posibilitatea de a programa polaritatea pentru CLK-urile de date și pentru sincronizarea cadrelor. Arhitectura interfeței McBSP este redată în continuare.

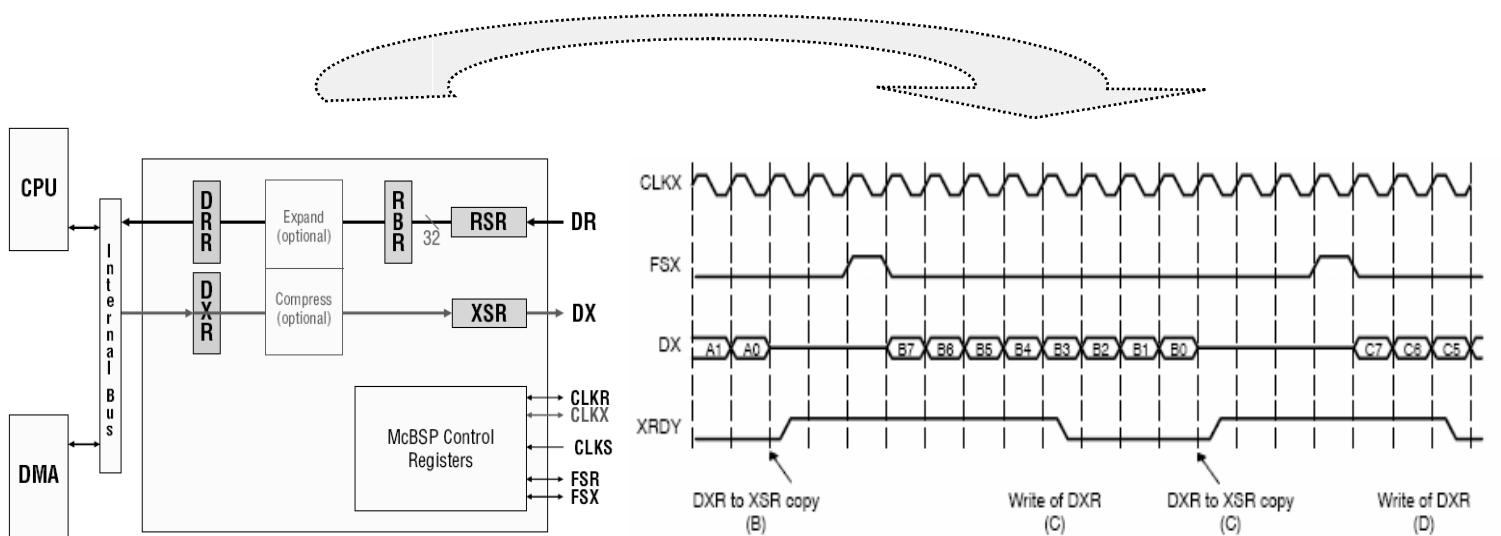


McBSP este alcătuit din două căi de transfer, o cale de date și o cale de control care sunt conectate cu dispozitivele externe, pinii de Tx și RX fiind separați, dispozitivele conectate la McBSP comunicând cu acesta prin intermediul unor registre de control de 32 de biți accesibile prin magistrala periferică internă.

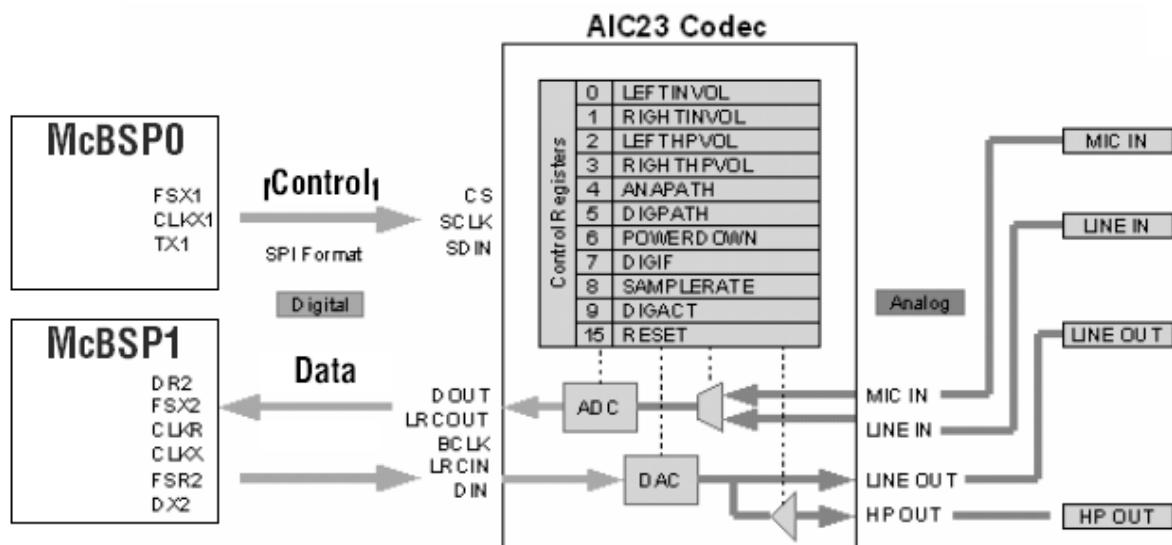
Datele sunt transferate către dispozitivele care interfațează cu McBSP prin intermediul pinului de transmisie DX și datele recepționate pe pinul de recepție DR. Informația de control (ceasul și sincronizarea cadrelor) este transmisă prin intermediul mai multor registre, CLKS, CLKX, CLKR, FSX și FSR. CPU comunică cu MCBSP utilizând registre de control 32 de biți prin magistrala periferică internă. Fie CPU, fie controller-ul EDMA citește datele recepționate în registrul de recepție date DRR și scrie datele pentru transmisie în registrul de transmisie DXR, care prin proprietatea de shiftare către pinii DR, respectiv DX determină posibilitatea de a executa mutarea datelor interne și comunicațiile externe simultan. În continuare este redat modul de operare standard al McBSP. În timpul unui transfer serial în mod tipic apar perioade de inactivitate ale portului serial între pachete sau transferuri. Semnalul de sincronizare a cadrelor pentru Rx și TX este activat pentru fiecare transfer. Când McBSP nu este în starea de reset și a fost configurat pentru o operație specifică, un transfer poate fi inițiat programând caracteristica de fază pentru un cadru pentru un număr de elemente de cadru necesare. Operația de recepție este redată în formele de undă de mai jos și se analizează starea și conținutul regisrelor implicate în acest transfer de date serial.



Odată ce semnalul de sincronizare de recepție cadru FSR, trece în starea activă este detectat pe primul front căzător al ceasului de recepție, CLKR. Datele care se află pe pinul DR sunt apoi shiftate în registrul de recepție shiftare RSR după o întârziere. Conținutul RSR este apoi copiat în RBR la sfârșitul fiecărui element pe frontul crescător al CLK. RBR nu este umplut cu datele anterioare, apoi are loc o copiere RBR în DRR. Acest lucru indică că registrul de recepție date DRR, este pregătit cu datele de citit de către CPU sau de către controller-ul EDMA. Operația de transmisie este redată în continuare. Odată ce are loc sincronizarea cadrelor valoarea din registrul de transmisie este shiftat și transmis pinului DX după o întârziere setată de un bit auxiliar. După fiecare copiere din DXR în XSR un bit de stare este activat pe frontul căzător al CLK următor, indicând faptul că registrul de transmisie date poate fi scris cu următoarele date care trebuie transmise. Bitul de stare amintit este resetat după ce CPU sau EDMA au efectuat scrierile.

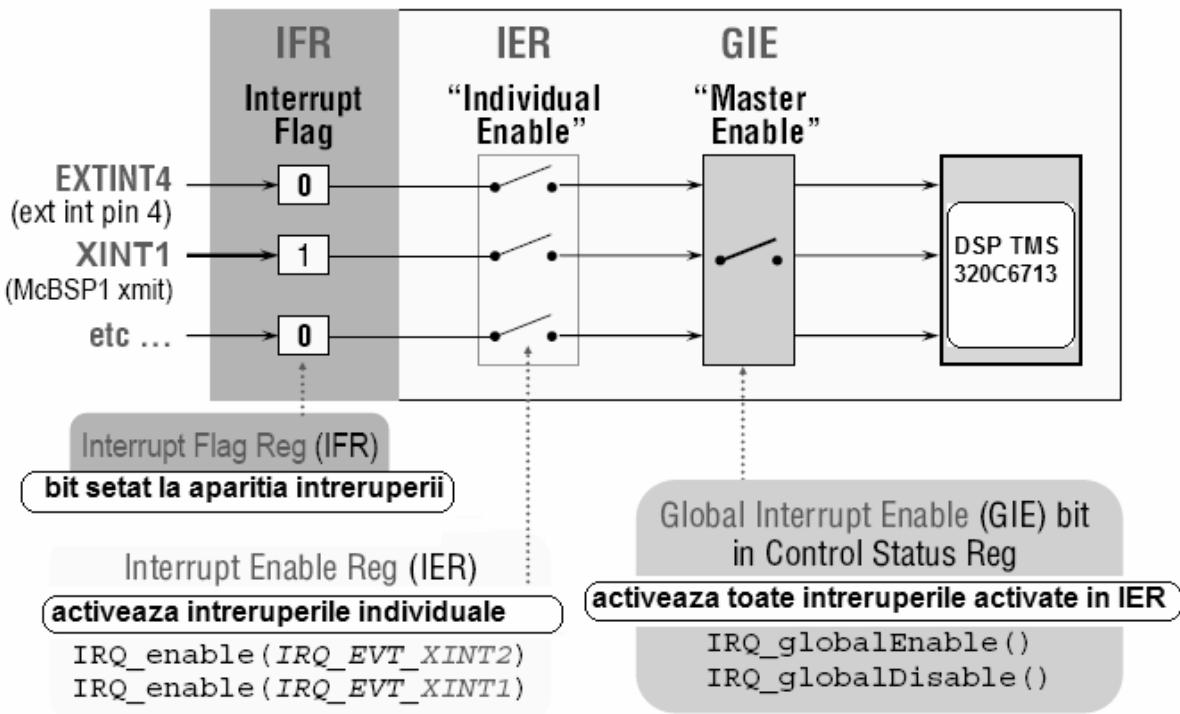


Din punct de vedere funcțional în arhitectura platformei de dezvoltare apar două unități McBSP, McBSP0 și McBSP1. McBSP0 este dedicat pentru programarea registrelor de control ale codec-ului AIC23 în timp ce McBSP1 este utilizat pentru a realiza transferul de date către și de la convertoarele de 24 de biți A/D și D/A, aceste operațiuni putând fi efectuate la frecvențe programabile (8k, 1k, 24k, 32k, 44.1k, 48k, 96k). Figura următoare este diagramea funcțională a modului de interfațare și operare a McBSP cu codec-ul AIC23, precum și o diagramă bloc ce sintetizează doar componentele necesare pentru aplicații audio.



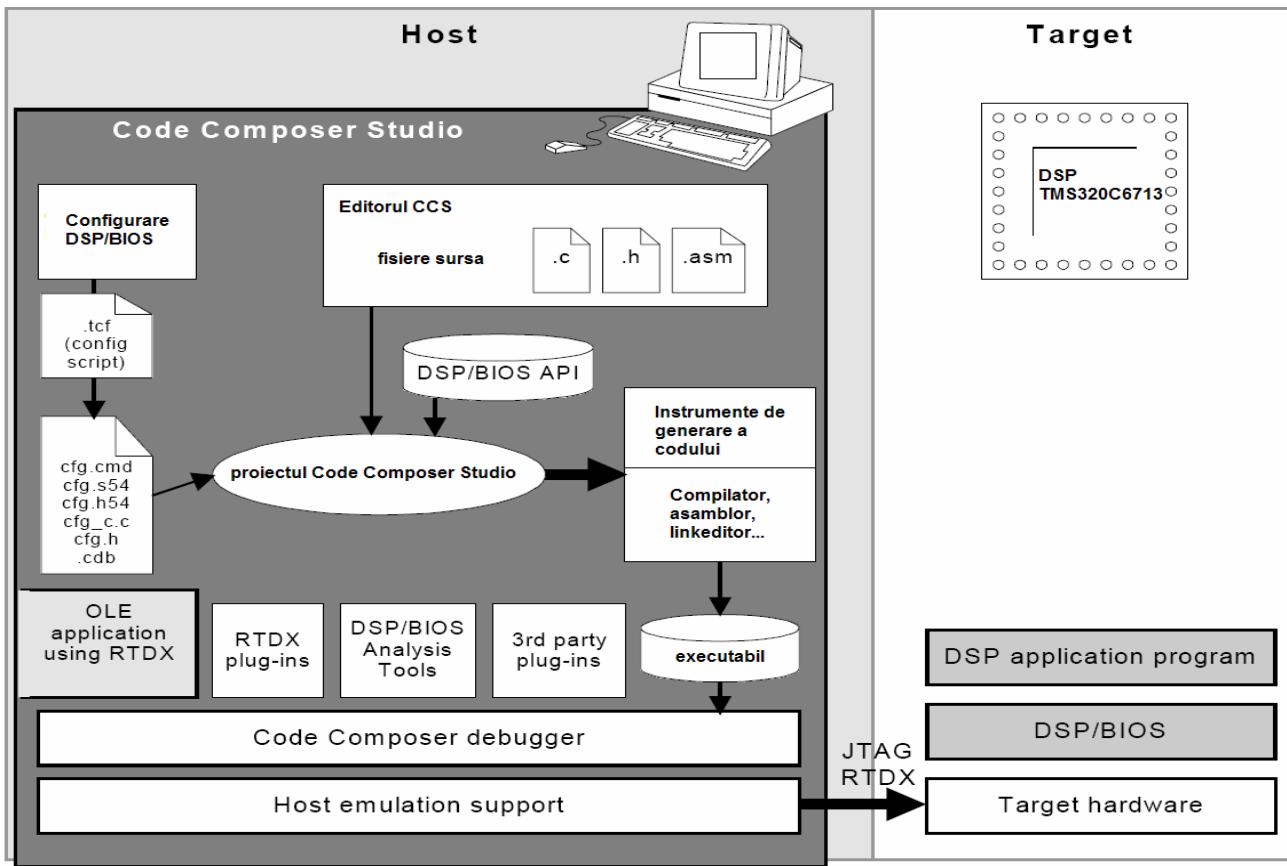
Ultimul aspect legat de comunicarea cu perifericele în sensul dezvoltării aplicațiilor este lucrul cu întreruperi. Astfel la momentul apariției unui eveniment exterior asincron care modifică starea uneia din perifericele din componența sistemului, DMA, HPI, timere, pini externi etc., se setează un flag specific de întreruperi într-un registrul de stare a execuției. În acest moment dacă întreruperile au fost activate, CPU se oprește din execuția curentă, dezactivează întreruperile global, resetează flagul de întrerupere din registrul de stare, salvează adresa de revenire, determină tipul întreruperii și cine a generat și apelează ISR. Rutina de tratare a întreruperilor ISR, salvează contextul curent al sistemului (operări efectuată la nivelul DSP/BIOS), execută codul de tratare pentru întrerupere, restaurează contextul sistemului și continuă prelucrarea din locația premergătoare apariției întreruperii (ultimele

două etape sunt efectuate la nivelul dispecerului din componența DSP/BIOS). Un scenariu de apariție și tratare al unei întreruperi este descris în continuare, fiind prezentate și mecanismele specifice pentru arhitectura de dezvoltare considerată.



## Sistemul DSP/BIOS

DSP/BIOS este un kernel real-time scalabil care implementează primul nivel de abstractizare al platformei de dezvoltare cu DSP TMS320C6713. Acesta a fost proiectat pentru a susține dezvoltarea de aplicații care necesită o planificare și o sincronizare real-time a task-urilor, comunicația gazdă – target (calculator gazdă unde se dezvoltă aplicația – platforma de dezvoltare cu DSP), precum și implementarea unui sistem de instrumentație real-time. Kernelul oferă specificul unui multi-threading preemptiv, un prim nivel de abstractizare al hardware-ului, mijloace de analiză real-time a performanțelor aplicațiilor și instrumente de configurare.



Kernelul DSP/BIOS a fost proiectat pentru a minimiza cerințele de resurse de memorie și CPU de la target, prin implementarea modulară și suportul unui API pentru asigurarea configurației și optimizării modulelor dezvoltate. Structurile de date interne ale kernelului sunt optimizate, iar dimensiunea codului redusă, prin posibilitatea de a configura static și de a combina entitățile în programe executabile. Fiecare API corespunzător modulelor este modularizat, librările fiind optimizate pentru a necesita un număr minim de cicli de ceas la execuție, o mare portiune fiind implementată în assembly. Comunicația între gazdă, pe care rulează instrumentele de analiză DSP/BIOS și target (placa de dezvoltare) se realizează într-o buclă de fundal (background thread), datele de instrumentație (loguri și fișiere trace-file de date) fiind formatare de gazdă. Rularea în background a instrumentelor de analiză ale kernel-ului nu interferează cu execuția curentă a taskurilor program. Dezvoltarea de aplicații este susținută de un puternic set de caracteristici ale kernelului care prin structurile de date și mecanismele de execuție implementate determină facilități importante. Astfel un program poate crea sau distruge obiecte funcție de contextul de execuție în care se află, utilizând fie obiecte create dinamic, fie obiecte create static.

Kernelul oferă un model de execuție bazat pe threaduri, implementând tipuri distincte de threaduri funcție de situație, cu posibilitatea de a controla prioritățile și caracteristicile de comunicare și interblocaj ale threadurilor.

Astfel sunt suportate threaduri specifice întreruperilor hardware, întreruperilor software, taskurilor utilizator, funcțiilor idle și funcțiilor periodice. Sunt implementate deosemenea și structuri de realizare a comunicației și sincronizării inter-thread, cum ar fi semafoarele, mailbox-urile și resource locks (echivalentul mutex-urilor din arhitecturile moderne de SO). Din punctul de vedere al subsistemului de I/O sunt suportate două modele de implementare, pentru a suporta performanțe maxime. Astfel un prim model I/O este cel bazat pe pipe-uri care sunt utilizate în comunicarea target/host și situații simple de intercomunicare între threaduri când un thread scrie date în pipe, iar un altul citește datele din pipe. Al

doilea model I/O implementat este cel bazat pe stream-uri (fluxuri), care este mai complex și este dezvoltat pentru a susține driverele. Managementul erorilor este asigurat de primitive de sistem de nivel scăzut, care suportă implementarea de tipuri de date generice și managementul utilizării memoriei, pentru a minimiza creșterea cererii de resurse de memorie și CPU. Un ultim aspect legat de prezentarea kernelului DSP/BIOS se referă la instrumentul de configurare, care generează cod pentru declararea statică a obiectelor utilizate în programul dezvoltat. Scripturile de configurare pot fi create și modificate pentru a include instrucțiuni de salt, ciclare și testare a parametrilor din linia de comandă. În continuare sunt prezentate componentele DSP/BIOS, cu implicații directe în generarea programului și mediului de debugging al IDE de dezvoltare CCS.

Pentru început sunt descrise modulele DSP/BIOS și instrumentele de configurare și analiză. Astfel după cum am menționat anterior API-ul DSP/BIOS este împărțit în module. Programele aplicație utilizează DSP/BIOS-ul prin apelul de funcții din modulele API, care oferă interfețe apelabile în C sau sub forma unor macro-uri optimizate în assembly. În componența DSP/BIOS distingem module pentru funcții atomare în assembly pentru operațiuni low-level (ATM module), un manager de buffere cu lungime fixă (BUF module), funcții dependente de arhitectura targetului, un manager de CLK, o interfață device driver, un manager de setare globală (GBL module), un manager de I/O general (GIO module), un manager de comunicație a hostului (HST module), manager de întreruperi hardware (HWI module), un manager de mailbox, pentru comunicarea inter-thread (MBX module), un manager de segmente de memorie, un manager de funcții periodice, un manager pentru setarea transferului de date real-time (RTDX module), un manager de stream-uri I/O (SIO module), un manager de întreruperi software (SWI module), un manager de servicii ale sistemului, un manager de multitasking (TSK module), precum și alte module cu funcționalități diverse.

Instrumentul de configurare a DSP/BIOS oferă posibilitatea optimizării aplicației dezvoltate prin crearea și stabilirea proprietăților obiectelor în mod static și nu la momentul execuției, îmbunătățindu-se astfel performanțele de execuție și consumul de resurse al aplicației. Există metode prin care un număr relativ mare de parametri pot fi setați de către librăria real-time a kernelului în momentul execuției, obiectele create fiind utilizate în apelurile API ale DSP/BIOS și incluzând întreruperi software, task-uri, stream-uri I/O și jurnale de evenimente (event logs) toate acestea compactate într-un program executabil.

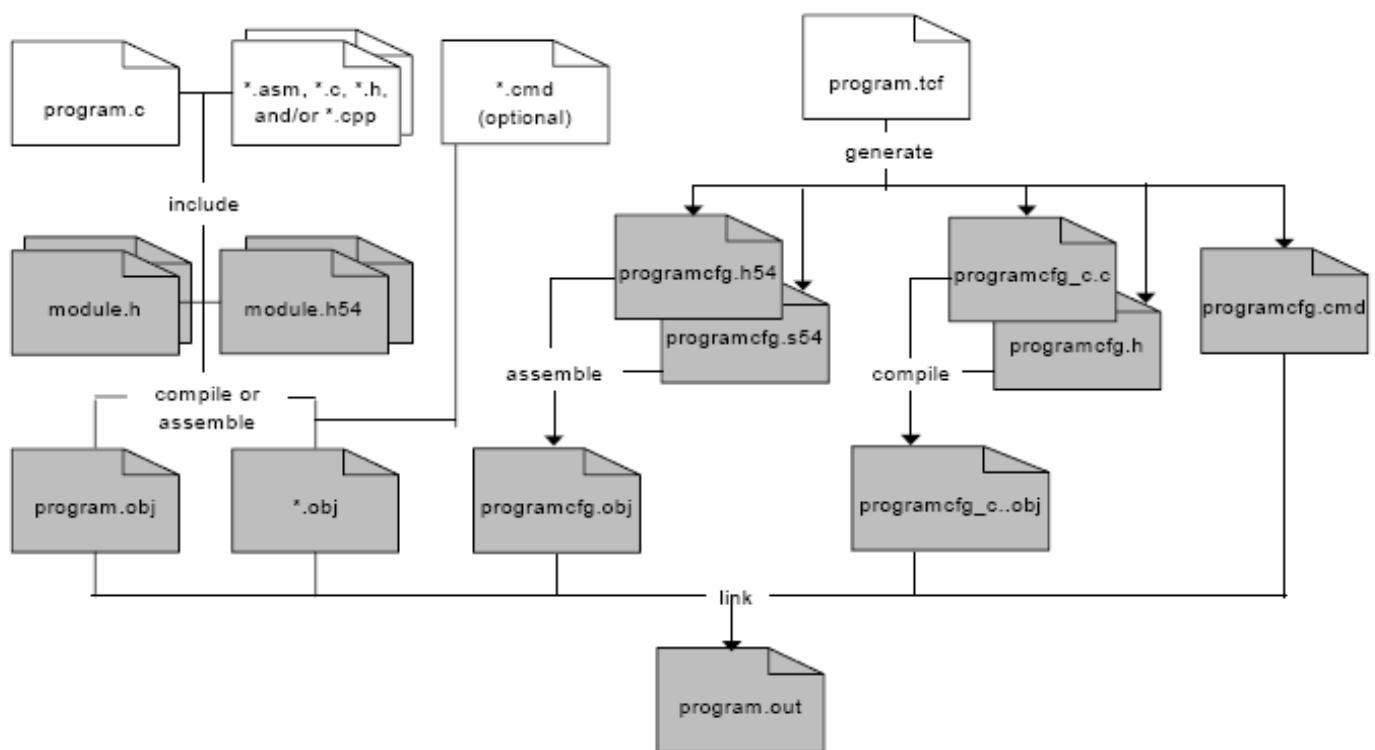
Instrumentul de analiză a kernelului DSP/BIOS complementează caracteristicile mediului de programare CodeComposerStudio oferind capacitați avansate de analiză a execuției aplicației, în mod independent și fără a afecta performanțele de execuție în real-time a aplicației. Ca metode de analiză a execuției aplicației se pot aminti Program Tracing, prin care se analizează fluxul dinamic de control în timpul execuției prin consultarea logurilor de evenimente; Performance Monitoring, prin care se urmărește analiza utilizării resurselor target-ului, cum ar fi încărcarea procesorului și temporizările și File Streaming prin care se realizează legarea obiectelor I/O de fișierele host rezidente.

În continuare este descris succint procesul de generare a programelor cu DSP/BIOS, urmărind ce fișiere sunt generate de către componente kernel-ului și cum sunt ele utilizate. Vor fi descrise aspecte cu privire la ciclul de dezvoltare a aplicației; aspecte privind configurarea statică a aplicațiilor DSP/BIOS; crearea dinamică a obiectelor DSP/BIOS; modul de creare și fișierele utilizate la crearea programelor, precum și aspectele privind compilarea și link-editarea lor; se va prezenta deosemenea pe scurt și utilizarea DSP/BIOS cu RTSL și apelul funcțiilor utilizator de către kernel și apelul API-ului în aplicațiile dezvoltate.

DSP/BIOS-ul suportă cicluri de dezvoltare iterative pentru programe, având posibilitatea de a crea framework-ul de bază pentru o aplicație și simularea acestuia înainte de implementarea efectivă a algoritmilor pentru DSP. Utilizatorul poate schimba cu ușurință prioritățile thread-urilor și tipurile de

thread-uri program pentru diverse funcții. Aspectul de interes maxim în dezvoltare pe lângă configurarea statică și crearea dinamică a obiectelor, discutate anterior, se referă la diagrama funcțională a modului de alcătuire a unei aplicații sub DSP/BIOS.

Fișierele program sunt descrise în continuare. Astfel program.c este sursa care conține funcția main, care mai poate conține și alte fișiere .c sau.h adiționale și fișiere .asm opționale. Fișierul module.h este fișierul header pentru API-ul DSP/BIOS-ului pentru programe C/C++, iar module.h54 este fișierul header API DSP/BIOS pentru programele în assembly. Fișierul obiect obținut în urma compilării sau asamblării fișierelor sursă este program.obj, care poate fi însotit și de alte fișiere obiect. Un alt element necesar în crearea unei aplicații este fișierul (fișierele opțional/e) de comenzi pentru linker care pot conține secțiuni adiționale pentru program care nu sunt definite de către configurația DSP/BIOS-ului.

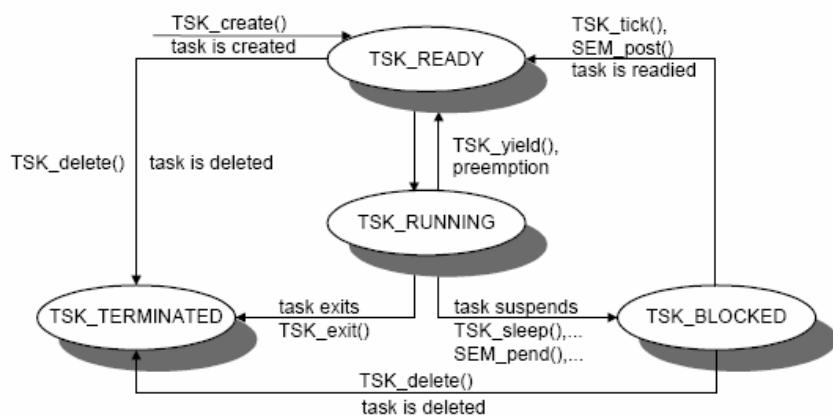


După cum am precizat anterior kernelul DSP/BIOS oferă mijloace implicate dar și explicate de a realiza analiza în real-time a aplicațiilor dezvolte. Aceste mecanisme au fost proiectate pentru a avea un impact minimal asupra performanțelor real-time ale aplicației în execuție. Analiza real-time este analiza datelor achiziționate de la sistem în timpul operării în timp real, în intenția de a determina dacă sistemul operează în limitele constrângerilor impuse, dacă împlinește obiectivele de performanță impuse și dacă este flexibil și extensibil pentru dezvoltări ulterioare. În acest context se pune problema unui debugging real-time, eficient în timpul execuției aplicației contrar specificului debugging-ului ciclic de la aplicațiile obișnuite care nu sunt supuse unor constrângerii temporale, hardware sau software. Astfel analiza se bazează pe codul de instrumentație implementat de kernel și care conține mecanisme cum sunt menegerul de evenimente, preluate din logurile de execuție, managerul de statistică a obiectelor (conținând informații despre SWI, HWI și TSK) și managerul canalului de I/O gazdă. De interes în dezvoltarea de aplicații este studiul tipurilor de thread-uri suportate de kernalul DSP/BIOS și comportamentul lor în funcție de priorități în timpul execuției programului. Cele mai multe aplicații real-time destinate DSP trebuie să efectueze un număr de funcții distincte (threaduri) în același timp, ca

răspuns la o serie de evenimente exterioare cum ar fi disponibilitatea datelor sau prezentă unui semnal de control. Kernelul DSP/BIOS oferă posibilitatea de a structura aplicațiile sub forma unei colecții de threaduri fiecare implementând o funcție modularizată. Programul multithread rulează pe un singur procesor permitând thread-urilor cu prioritate mai mare să suspende threadurile cu prioritate mai scăzută (preemptivitate) oferind diferite mijloace de interacțiune între thread-uri, bloaje, comunicare și sincronizare. DSP/BIOS oferă suport pentru câteva tipuri de thread-uri cu priorități diferite, fiecare având caracteristici de execuție și preemptive distincte, cum ar fi (în ordinea descrescătoare a priorităților): întreruperile hardware (HWI), ce includ funcții CLK (specifice întreruperilor timer-ului încorporat), întreruperi software (SWI), ce includ funcții PRD(periodice, multiplu de întreruperi ale timer-ului), taskuri (TSK) și thread-ul de background(IDL – idle thread). Înteruperile hardware sunt declanșate ca urmare a apariției unor evenimente externe asincrone mediului DSP. O funcție HWI (numită adesea ISR) este executată după declanșarea unei întreruperi hardware pentru a efectua un task critic care este supus unui deadline. Înteruperile software sunt declanșate, spre deosebire de cele hardware, prin apelul în interiorul programului a funcțiilor SWI, neavând aceleași constrângeri critice ca și cele hardware. Task-urile au o prioritate mai mică decât a întreruperilor dar mai mare decât a thread-ului de background, ele având posibilitatea de a-și amâna execuția până când resursele necesare sunt disponibile, kernelul DSP/BIOS oferind structuri specifice care sunt utilizate pentru comunicarea și sincronizarea inter task-uri, cozi, semafoare și mailbox-uri. Thread-ul de background execută bucla de idle și are cea mai scăzută prioritate și care după revenirea din main determină apelul de start-up al fiecărui modul al kernelului și intrarea în buclă, apelându-se funcții specifice pentru obiectele IDL(obiectele specifice modulului idle al kernelului). Întrucât înteruperile sunt un aspect important în proiectarea aplicației propuse, dar și în general în dezvoltare, se impune o descriere a înteruperilor hardware și software și a modului specific de implementare a managerelor de înterupere din arhitectura kernelului DSP/BIOS. Înteruperile hardware asigură execuția procesarilor critice pe care o aplicație trebuie să le execute, ținând cont de anumite constrângeri, la apariția unor evenimente externe asincrone. Modulul HWI al kernelului DSP/BIOS se ocupă cu managementul înteruperilor hardware. Ca implementare înteruperile hardware pot fi scrise fie în C, fie în assembly fie combinând cele două limbaje. Funcțiile HWI sunt de obicei scrise în assembly, pentru eficiență, pentru scrierea acestora în C fiind necesară intervenția dispecerului HWI. În ceea ce privește configurarea înteruperilor managerul modulului HWI conține un obiect HWI pentru fiecare întrerupere hardware a DSP-ului. Funcțiile specifice modulului se referă la activarea sau dezactivarea înteruperilor, la conectarea dispecerului HWI, la operațiunile de intrare și ieșire din ISR hardware, precum și o funcție de restaurare a stării înteruperilor hardware. Un alt aspect legat de tratarea înteruperilor hardware se referă la managementul contextului și înteruperii între două întrerupere, suportul fiind asigurat de dispecerul HWI și prin execuția unor operații la nivel de sistem, apelul planificatoarelor pentru managerele modulelor SWI și TSK să se efectueze la momentul potrivit și dezactivarea înteruperilor individuale în timpul execuției unei ISR. Execuția înteruperilor software este controlată de managerul SWI. Atunci când o aplicație face un apel la un API specific care poate declanșa sau propune o întrerupere software, managerul SWI planifică pentru execuție funcția corespunzătoare înteruperii software, utilizând obiecte SWI. O rutină SWI poate fi suspendată preemptiv de o întrerupere HWI și poate suspenda preemptiv task-uri utilizator. Ceea ce este important și util de știut în dezvoltarea de aplicații este faptul că un handler SWI este executat până la final, în cazul în care nu este întrerupt de o întrerupere hardware sau suspendat preemptiv de un handler SWI cu prioritate mai mare. Foarte important de menționat este faptul că deseori înteruperile software pot fi apelate în interiorul unei ISR HWI, codul de declanșare a înteruperii software trebuind inclusă în funcțiile specifice HWI de intrare sau ieșire din ISR, fie sub invocarea dispecerului. Obiectele SWI pot fi create dinamic sau distruse dinamic în momentul execuției

programului, printr-un apel către o funcție specifică de creare/distrugere, fie static la configurare. Apelul de funcție pentru crearea unui obiect SWI poate fi efectuat doar de la nivelul de task și nu de către un obiect HWI sau alt obiect SWI. Un aspect foarte important și care are un rol important în dezvoltarea de aplicații este utilizarea mailbox-ului unui obiect SWI, cu implicație directă la implementarea funcțiilor de procesare a bufferelor de date primite pe line-in la prelucrarea primară audio. Astfel fiecare obiect SWI are un mailbox de 32 de biți care este utilizat fie pentru a determina declanșarea întreruperii software sau pentru a constitui valori pentru evaluarea în funcția SWI.

Task-urile sunt obiecte ale kernelului DSP/BIOS care sunt supervizate de modulul TSK, având prioritate mai mică decât întreruperile și mai mare decât a task-ului de background. Modulul TSK planifică dinamic task-urile și realizează preempția acestora bazându-se pe priorități și starea curentă a execuției. Obiectele TSK pot fi create fie dinamic prin funcții specifice fie static. În continuare sunt redate variațiile în modul de execuție a task-urilor.



Ultimul aspect descris în acest context este thread-ul de background (idle loop) care rulează continuu când nu au loc întreruperi hardware sau software sau nu se execută task-uri. Orice alt thread poate realiza suspendarea preemptivă a buclei idle în orice moment. Managerul IDL permite utilizatorului să insereze funcții care să fie executate pe parcursul executării buclei, ca thread de background.

Următorul subsistem și aspect în descrierea arhitecturii kernelului DSP/BIOS este memoria și funcțiile de nivel scăzut (low-level) ale kernelului pentru real-time și multitasking. Aceste funcții sunt integrate în următoarele module ale kernelului, MEM, BUF (pentru managementul alocării de segmente de memorie de dimensiune fixă(BUF) sau variabilă(MEM)), SYS (managementul serviciilor de sistem diverse) și QUE (pentru managementul structurilor de coadă, specifice și utile în implementarea elementelor de excludere mutuală sau în transmiterea mesajelor între threaduri). Managerul de segmente de memorie al modulului MEM se ocupă de menevrare a segmentelor de memorie cu identificatori corespunzători zonelor de memorie, oferind desemnează un set de funcții pentru alocarea și eliberarea dinamică a blocurilor de memorie de dimensiune variabilă. Kernelul DSP/BIOS oferă suportul necesar, printr-un set puternic de funcții, pentru configurarea segmentelor de memorie (inserare segment, redenumire, ștergere, schimbare proprietăți segment), posibilitatea activării/dezactivării alocării dinamice a memoriei, posibilitatea definirii segmentelor de către utilizator prin fișiere de comenzi pentru linker, obținerea status-ului segmentelor de memorie și mecanisme eficiente pentru reducerea fragmentării memoriei. Serviciile de sistem oferite de modulul SYS oferă funcționalități diverse în ceea ce privește managementul activității sistemului prin mecanisme de oprire (halt) a execuției programului și mecanisme de tratare a erorilor.

Un ultim aspect în descrierea kernelului DSP/BIOS se referă la studiul implementării operațiilor de I/O, prin cele două modele de implementare, pipe-uri și stream-uri. Astfel la nivelul aplicație operațiunile I/O pot fi tratate sub forma stream-urilor, pipe-urilor sau obiectelor canal (de transfer date) gazdă, fiecare tip de obiect având un modul dedicat pentru managementul datelor de I/O. Un flux este un canal de transfer date prin care are loc transferul de date între programul aplicație și un dispozitiv I/O. Acest canal având ca moduri de acces read-only (pentru intrare) sau write-only (pentru ieșire), oferind o interfață simplă și generică pentru toate dispozitivele I/O, degrevând aplicația de a cunoaște detaliile de implementare și operare a fiecărui dispozitiv. Un aspect important de menționat în acest context este natura asincronă a fluxurilor I/O, bufferele de date fiind intrări sau ieșiri în concurență cu prelucrările efectuate. În timp ce o aplicație procesează buffer-ul curent, un nou buffer de intrare este umplut, iar cel anterior este golit. Acest management eficient al bufferelor I/O permite minimizarea operațiunilor de copiere efectuate de către stream-uri, acestea realizând de fapt un transfer de pointeri către date și nu date, astfel reducându-se overhead-ul și permățând programelor aplicație să împlinească constrângerile real-time. Un program generic preia un buffer cu date de intrare, procesează datele conținute și apoi trimitе la ieșire datele procesate, într-o secvență continuă care se încheie de obicei la terminarea programului. Pipe-urile oferă o structură software consistentă utilizată pentru operațiunile de I/O între DSP și toate tipurile de dispozitive periferice real-time. De menționat este faptul că toate operațiunile I/O tratează secvențial cadrele de date care au o lungime fixă, deși aplicațiile pot depune în aceste cadre date de lungime variabilă, dar mai mică decât lungimea maximă a cadrului. Un lucru important despre pipe-uri este faptul că acestea pot suporta și transferul de date între două thread-uri aplicație. Ultimul mod de transfer date este cel al canalelor de transfer date pentru host care sunt utilizate pentru transferul de date între target și host, fiind configurate static pentru intrare sau ieșire și implementate utilizând un obiect pipe. Pentru a marca principala diferență între cele două modele de transfer al datelor se poate preciza faptul că modelul bazat pe pie-uri suportă comunicația de nivel scăzut, în timp ce modelul bazat pe stream-uri suportă operațiunile I/O de nivel înalt independente de dispozitiv. Aceste specificații au avut rolul de a crea o imagine de ansamblu asupra primului nivel de abstractizare a hardware-ului în vederea dezvoltării aplicațiilor propuse. Asupra aspectului operațiunilor I/O se va insista ulterior la descrierea efectivă a aplicațiilor întrucât sunt elemente specifice de implementare, care se vor baza pe cele prezentate mai sus.

Elementele prezentate până acum au pus în evidență mediul de dezvoltare ales pentru dezvoltarea de aplicații de prelucrare digitală a semnalului. Avantajele hardware sunt clare și au fost descrise pe parcurs, suportul software (DSP/BIOs, CSL/BSL și CodeComposerStudio) fiind deosebit de mare ajutor și eficiență pentru dezvoltare.

## DSP Laborator 6

Aplicatii simple de prelucrare a datelor.

- Factorialul unui numar.
- Looping audio utilizand pooling sau intreruperi.

In lucrarea de laborator propusa se vor dezvolta o serie de aplicatii simple. Ca suport pentru dezvoltarea de aplicatii s-a dezvoltat un template de proiect care a fost configurat pentru a permite dezvoltarea de aplicatii. Acest template contine toate optiunile activate in ceea ce priveste lucru cu librariile BSL, CSL si RTS. Pentru a sintetiza optiunile de configurare ale proiectului este redat in continuare continutul fisierului .pjf al proiectului template. Pentru a adauga noi optiuni in functie de aplicatie se utilizeaza meniul Build Options a proiectului unde se pot executa modificarile intr-o interfata grafica asupra acelorasi optiuni redate mai jos.

```
; Code Composer Project File, Version 2.0 (do not modify or remove this line)

[Project Settings]
ProjectDir="D:\DSP_DEV\template_dezvoltare\
ProjectType=Executable
CPUFamily=TMS320C67XX
Tool="Compiler"
Tool="CustomBuilder"
Tool="DspBiosBuilder"
Tool="Linker"
Config="Debug"
Config="Release"

[Source Files]
Source="Factfunc.asm"
Source="Factorial.c"
Source="C6713dsk.cmd"

["Compiler" Settings: "Debug"]
Options=-g -fr"$(Proj_dir)\Debug" -i"." -i"$(Install_dir)\c6000\dsk6713\include"
-d"_DEBUG" -d"CHIP_6713" -mv6710

["Compiler" Settings: "Release"]
Options=-O3 -fr"$(Proj_dir)\Release" -mv6700

["Linker" Settings: "Debug"]
Options=-q -c -m".\Debug\templateC6713.map" -o".\Debug\templateC6713.out" -w -x
-i"$(Install_dir)\c6000\dsk6713\lib" -l"dsk6713bsl.lib"
-l"cs16713.lib" -l"rts6700.lib"

["Linker" Settings: "Release"]
Options=-c -m".\Release\templateC6713.map" -o".\Release\templateC6713.out" -w -x

["C6713dsk.cmd" Settings: "Debug"]
LinkOrder=1

["C6713dsk.cmd" Settings: "Release"]
LinkOrder=1
```

In dezvoltarea de aplicatii se va utiliza fie arhiva cu templateul fie se va crea un proiect nou cu optiunile de mai sus active. Templateul nu va contine niciun fisier sursa predefinit, insa va avea atasat fisierul de configurare a linkerului C6713dsk.cmd care are rolul de a defini harta de memorie a sistemului cu adresele de inceput si dimensiunile sistemelor de memorie existente in sistem (IRAM, SDRAM, FLASH). In acelasi fisier se definesc si sectiunile de memorie alocate pentru diferitele

componente (vectori de intrerupere, stiva, memoria de system, codul executabil, entry pointuri pentru program, date de initializare specifice CSL). Continutul fisierului de configurare a linkerului este dat in continuare.

```
/*C6713dsk.cmd  Linker command file*/

MEMORY
{
    IVECS:      org=0h,           len=0x220
    IRAM:       org=0x00000220,   len=0x0002FDE0 /*internal memory*/
    SDRAM:      org=0x80000000,   len=0x00100000 /*external memory*/
    FLASH:      org=0x90000000,   len=0x00020000 /*flash memory*/
}

SECTIONS
{
    .EXT_RAM :> SDRAM
    .vectors :> IVECS /*in vector file*/
    .text      :> IRAM /*Created by C Compiler*/
    .bss       :> IRAM
    .cinit     :> IRAM
    .stack     :> IRAM
    .sysmem   :> IRAM
    .const     :> IRAM
    .switch   :> IRAM
    .far       :> IRAM
    .cio       :> IRAM
    .csldata  :> IRAM
}
```

Pentru prima aplicatie propusa de calcul a factorialului unui numar se impune adaugarea unor noi fisiere in proiectul template. Astfel se adauga sursa .c care contine programul principal si un fisier .asm care contine implementarea in limbaj de asamblare a codului functiei de calcul a factorialului unui numar.

```
Codul C : Factorial.c

//Factorial.c Programul principal de calcul al factorialului unui numar n.

#include <stdio.h>

void main()
{
    short n=7;                      //seteaza valoarea
    short result;                   //rezultatul de la functia factfunc

    result = factfunc(n);          //apel catre rutina asm
    printf("factorialul lui %d este %d \n", n,result); //se afiseaza rezultatul
}
```

Rutina in limbaj de asamblare pentru calculul factorialului este redata in continuare.

```
;Factfunc.asm  Functie de calcul a factorialului unui numar implementata in asm

        .def    _factfunc           ;numele functiei de calcul apelata din C
_factfunc: MV     A4,A1           ;setam numarul de pasi in registrul A1
            SUB    A1,1,A1           ;decrementam numarul de pasi
LOOP:    MPY    A4,A1,A4         ;produsul se acumuleaza in A4
            NOP                  ;se adauga o intariziere la inmultire
            SUB    A1,1,A1           ;se decrementeaza contorul
            [A1]   B     LOOP           ;salt la LOOP daca A1 nu e 0
```

```

NOP      5          ; intarziere implementata cu NOP
B       B3          ; revenire din rutina
NOP      5          ; intarziere implementata cu NOP
.end

```

Dupa ce proiectul a fost compilat si programul incarcat in memoria flash a DSP ului se poate rula programul pas cu pas in mod debug sau free run cand rezultatul este redirectat catre fluxul de iesire STDOUT. Mediul de programare CodeComposer ofera posibilitatea de a analiza codul si de a rula la nivel de instructiune de asamblare utilizand optiunea Mixed Code. Astfel putem optimiza codul pe portiuni pentru sporirea vitezei/timpului de executie, peste optiunile configurabile ale compilatorului. In continuare este redat codul in mod mixed util in debugging.

```

//Factorial.c Programul principal de calcul al factorialului unui numar n.

#include <stdio.h>

void main()
{
00007A60          main:
00007A60 01BCD4F6           STW.D2T2      B3,*SP--[0x6]
00007A64 00002000           NOP          2
short n=7;           //seteaza valoarea
00007A68 018003A8           MVK.S1      0x0007,A3
00007A6C 01BD42D4           STH.D2T1      A3,*+SP[0xA]
00007A70 00002000           NOP          2
short result;         //rezultatul de la functia factfunc

result = factfunc(n);   //apel catre rutina asm
00007A74 00001C10           B.S1        Factfunc.asm:4:13$
00007A78 023D42C4           LDH.D2T1      *+SP[0xA],A4
00007A7C 01BD442A           MVK.S2      0x7a88,B3
00007A80 0180006A           MVKH.S2     0x0000,B3
00007A84 00002000           NOP          2
00007A88          RL0:
00007A88 023D62D4           STH.D2T1      A4,*+SP[0xB]
00007A8C 00002000           NOP          2
printf("factorialul lui %d este %d \n", n,result); //se afiseaza rezultatul
00007A90 02BD42C6           LDH.D2T2      *+SP[0xA],B5
00007A94 023D62C6           LDH.D2T2      *+SP[0xB],B4
00007A98 0FFF9013           B.S2        printf
00007A9C 01C92BA8  ||       MVK.S1      0xfffff9257,A3
00007AA0 01800068           MVKH.S1     0x0000,A3
00007AA4 01BC22F4           STW.D2T1      A3,*+SP[0x1]
00007AA8 02BC42F6           STW.D2T2      B5,*+SP[0x2]
00007AAC 023C62F7           STW.D2T2      B4,*+SP[0x3]
00007AB0 01BD5C2A  ||       MVK.S2      0x7ab8,B3
00007AB4 0180006A           MVKH.S2     0x0000,B3
}
00007AB8          RL1:
00007AB8 01BCD2E6           LDW.D2T2      *++SP[0x6],B3
00007ABC 00006000           NOP          4
00007AC0 000C0362           B.S2        B3
00007AC4 00008000           NOP          5

```

Este important de sesizat faptul ca DSP-ul fiind o arhitectura de tip VLIW (very long instruction word) pachetele de instructiuni pot avea o dimensiune pana la 256 de biti intrucat pot contine instructiuni adresate celor 8 unitati functionale, unitatea functională fiind specificată în mnemonica instructiunii (ex: LDW.D2 – încarcă un cuvânt de date utilizând unitatea functională D2; B.S1 – salt necondiționat la adresa de inceput a codului executabil al funcției factfunc utilizând unitatea functională S1).

In cazul executiei aplicatiei putem vizualiza continutul curent al registrilor interni ai DSP-ului si putem astfel valida executia aplicatiei.

---

A0	= 00000000	B0	= 00008314	PC	= 00007C60	EN	= 1
A1	= 00000000	B1	= 00000001	ISTP	= 00000000	PGIE	= 0
A2	= 00000001	B2	= 00000000	IFR	= 00000400	PCC	= 0
A3	= 00007BC0	B3	= 00005E20	IER	= 00000000	DCC	= 0
A4	= FFFFFFFF	B4	= 00007EB0	IRP	= 000005C0	GIE	= 0
A5	= 00000000	B5	= 000000BE	NRP	= 18EC01CC	SAT	= 0
A6	= 00000001	B6	= 00000000	AMR	= 00000000	PWRD	= 00
A7	= 00000002	B7	= 000085F4	CSR	= 02030100	FAUCR	= 00000000
A8	= 00000000	B8	= 00000004	CPUID	= 2	FMCR	= 00000000
A9	= 00000001	B9	= 00000000	REVID	= 3	FADCR	= 00000000
A10	= 00000000	B10	= 00008500				
A11	= 449F375E	B11	= 00000000				
A12	= 00000000	B12	= 00000000				
A13	= 00000000	B13	= 905B2F8F				
A14	= 449F375E	B14	= 00000220				
A15	= FFFFFFFF	B15	= 000089E8				

---

### Cerinta1 :

Sa se implementeze suma  $n+(n-1)+(n-2)+\dots+1$  utilizand o functie in limbaj de asamblare. Sa se testeze utilizand un program de test in C.

;SUMFUNC.ASM Rutina in asm pentru calculul  $n+(n-1)+\dots+1$

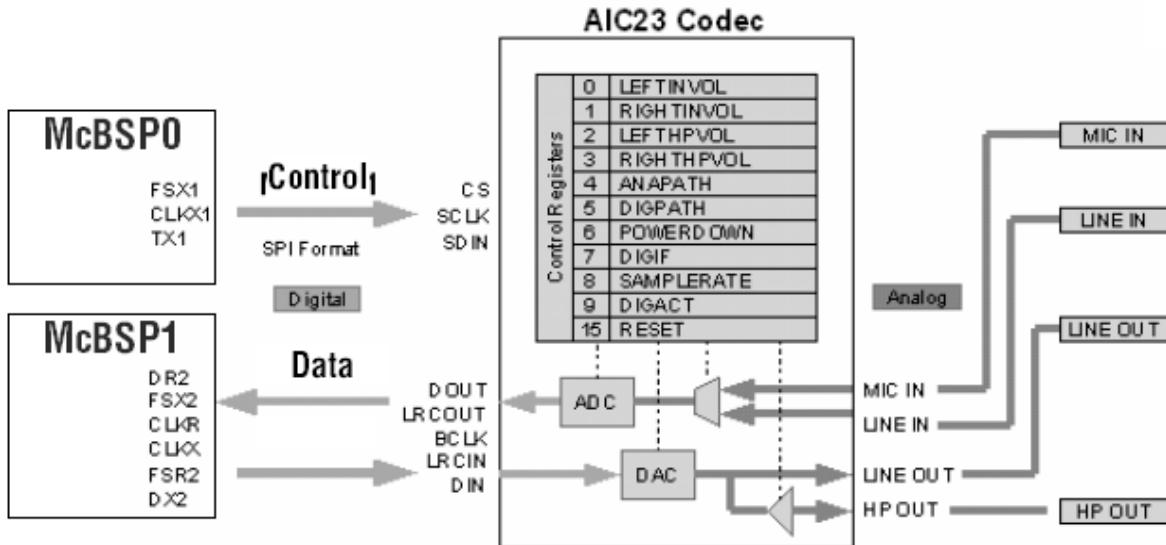
```

.def      _sumfunc          ; numele functiei care se va apela din C
_sumfunc: MV    .L1  A4,A1          ; seteaza n ca un contor de bucla
              SUB   .S1  A1,1,A1          ;decrementeaza n
LOOP:      ADD   .L1  A4,A1,A4        ;suma se acumuleaza in A4
              SUB   .S1  A1,1,A1          ;decrementeaza contorul
[A1]       B     .S2   LOOP           ; salt la eticheta loop daca A1!=0
              NOP    5                  ;intarzieri pentru sincronizare unitati functionale
              B     .S2   B3             ;revenire la rutina apelanta
              NOP    5                  ;intarzieri pentru sincronizare unitati functionale
.end

```

Looping audio utilizand pooling sau intreruperi.

A doua aplicatie propusa se refera la looping audio utilizand pooling sau intreruperi. Ambele cazuri vor fi analizate comparativ. Pentru a realize interfata cu semnalul audio placa de dezvoltare cu DSP TMS320C6713 DSK contine un codec audio TLV320AIC23B. Acesta este un codec stereo de inalta performanta. Converteoarele ADC/DAC din structura acestui codec utilizeaza tehnologia multibit sigma-delta, suportand transferuri intre 8kHz si 96kHz. Intrucat pentru dezvoltarea aplicatiei propuse se utilizeaza codecul AIC23 se vor utiliza functii din libraria BSL. Pentru a prelua informatia de la codec DSP ul utilizeaza un port serial multicanal buffered McBSP. Portul McBSP ofera urmatoarele functii : comunicare full-duplex ; registre de data dublu-bufferate, ce permit operarea cu un flux continuu de date, CLK si framing independent pentru Rx si Tx ; interfață directă pentru codec-urile uzuale și pentru chip-urile de interfață analogică (AIC) și alte dispozitive ADC sau DAC conectate ; CLK intern pentru transferurile de date. Din punct de vedere functional in arhitectura platformei de dezvoltare apar două unitati McBSP, McBSP0 și McBSP1. McBSP0 este dedicat pentru programarea regisrelor de control ale codec-ului AIC23 in timp ce McBSP1 este utilizat pentru a realiza transferul de date către și de la convertoarele de 24 de biți A/D și D/A, aceste operațiuni putând fi efectuate la frecvențe programabile (8k, 1k, 24k, 32k, 44.1k, 48k, 96k). Figura următoare este diagrama



funcțională a modului de interfațare și operare a McBSP cu codec-ul AIC23, precum și o diagramă bloc ce sintetizează doar componentele necesare pentru aplicații audio.

Intrucat utilizam McBSP se impune integrarea libreriei de functii CSL, aceasta oferind o interfata de programare a subsistemului McBSP. La compilarea aplicatiilor dezvoltate mediul de programare va realiza satisfacerea dependentelor la nivel de fisiere header utilizand informatia data linkerului privind librariile de functii utilizate si caile de includere a fisierelor din fisierul de configurare a proiectului. Astfel sectiunile de include si documents vor fi populate la build cu headerele necesare fiind disponibile si vizualizarii.

Intrucat am ales implementarea utilizand doua tehnici codul sursa contine macrouri care sa separe codul asociat fiecarei abordari. In continuare sunt redate componentele principale ale aplicatiei.

#### Test\_codec.c

```
// Aplicatie de test a codecului AIC23
// se propune lucrul in mod polling sau cu intreruperi pe achizitia semnalului
// Semnalul de iesire va fi cel aplicat la intrare..

#include <csl.h> // headere functii suport chip
#include <csl_mcbsp.h> // headere functii pt interafatare McBSP
#include "dsk6713_aic23.h" //fisierul de suport pentru codec din DSK

#define INTERRUPT_ON // activare mod transfer cu intreruperi
#define POLLING_ON // activare mod transfer cu polling

Uint32 fs=DSK6713_AIC23_FREQ_8KHZ; //seteaza rata de esantionare

#ifdef INTERRUPT_ON
interrupt void c_int11() //rutina de tratare a intreruperilor
{
    short sample_data;

    sample_data = input_sample(); //date de intrare
    output_sample(sample_data); //date de iesire
    return;
}
#endif

void main()
{
#ifdef POLLING_ON
    short sample_data;

    comm_poll(); //se initializeaza codecul, DSK si McBSP
#endif
}
```

```

#define INTERRUPT_ON
    comm_intr();           //initializare codec, DSk , McBSp si activare intreruperi
#endif
    while(1){
#endif
    #ifdef POLLING_ON
        sample_data = input_sample(); //esantion de intrare
        output_sample(sample_data);   //esantion de iesire
#endif
    }
}

```

### **Codec\_interface.c**

```

/* functii de interfata pentru lucrul cu codecul audio */

#include "codec_interface.h"

extern Uint32 fs;                      //frecventa de esantionare

void c6713_dsk_init()                  //initializarea perifericelor DSP
{
DSK6713_init();                      //apel la initializarea componentelor
specifice BSL-EMIF,PLL

hAIC23_handle=DSK6713_AIC23_openCodec(0, &config); //handle la codec
DSK6713_AIC23_setFreq(hAIC23_handle, fs); //seteaza rata de esantionare
MCBSP_config(DSK6713_AIC23_DATAHANDLE,&AIC23CfgData); //configurarea codecului

MCBSP_start(DSK6713_AIC23_DATAHANDLE, MCBSP_XMIT_START | MCBSP_RCV_START |
            MCBSP_SRGR_START | MCBSP_SRGR_FRAMESYNC, 220); //declansarea startului
codecului
}

void comm_poll()                      //communicatie si transfer utilizand polling (interrogare)
{
    poll=1;                           //setam 1
    c6713_dsk_init();                //initializarea DSP si codecului
}

void comm_intr()                     //comunicatie si transfer utilizand intreruperi
{
    poll=0;                           //setam 0
    IRQ_globalDisable();             //dezactivam intreruperile
    c6713_dsk_init();                //initializarea DSP si codec
    CODECEventId=MCBSP_getXmtEventId(DSK6713_AIC23_codecdathandle); //transmitem
e utlizand McBSP1

    IRQ_map(CODECEventId, 11);       //mapeaza transmisia la intrerupere INT11
    IRQ_reset(CODECEventId);        //reseteaza intreruperea
    IRQ_globalEnable();             //activarea globala a intreruperilor
    IRQ_nmiEnable();                //activeaza intreruperile nemascabile
    IRQ_enable(CODECEventId);       //activeaza intreruperile pe transmisie

    output_sample(0);               //declanseaza intreruperea scriind un esantion la iesire
}

void output_sample(int out_data)     //scrierea unui esantion la iesire
{
    AIC_data.uint=0;                //se sterge informatia din structura de date a AIC
    AIC_data.uint=out_data;         //se scrie informatia care trebuie scrisa la iesire

    //daca codecul este pregatit de transmisie
    if (poll) while(!MCBSP_xrdy(DSK6713_AIC23_DATAHANDLE));
}

```

```

// scrie informatia la iesire
MCBSP_write(DSK6713_AIC23_DATAHANDLE,AIC_data.uint); }

Uint32 input_sample()           //citirea unui esantion pe intrare
{
//daca codecul a receptionat date
    if (poll) while(!MCBSP_rrdy(DSK6713_AIC23_DATAHANDLE));
        //citeste datele de intrare
        AIC_data.uint=MCBSP_read(DSK6713_AIC23_DATAHANDLE);
    return(AIC_data.uint); }

/* header pentru functiile de interfata cu codecul AIC23 */

#include "dsk6713.h"
#include "dsk6713_aic23.h"

#define LEFT  0           //defineste canalul stanga
#define RIGHT 1          //defineste canalul dreapta
union {
    Uint32 uint;
    short channel[2];
} AIC_data;

extern far void vectors();      //vectori de intrerupere

static Uint32 CODECEventId, poll;

// Setarea configuratiei canalului de date al McBSP
MCBSP_Config AIC23CfgData = {
    MCBSP_FMKS(SPCR, FREE, NO)           |
    MCBSP_FMKS(SPCR, SOFT, NO)           |
    MCBSP_FMKS(SPCR, FRST, YES)          |
    MCBSP_FMKS(SPCR, GRST, YES)          |
    MCBSP_FMKS(SPCR, XINTM, XRDY)         |
    MCBSP_FMKS(SPCR, XSYNCERR, NO)         |
    MCBSP_FMKS(SPCR, XRST, YES)          |
    MCBSP_FMKS(SPCR, DLB, OFF)           |
    MCBSP_FMKS(SPCR, RJUST, RZF)          |
    MCBSP_FMKS(SPCR, CLKSTP, DISABLE)     |
    MCBSP_FMKS(SPCR, DXENA, OFF)          |
    MCBSP_FMKS(SPCR, RINTM, RRDY)          |
    MCBSP_FMKS(SPCR, RSYNCERR, NO)          |
    MCBSP_FMKS(SPCR, RRST, YES),           |

    MCBSP_FMKS(RCR, RPHASE, SINGLE)       |
    MCBSP_FMKS(RCR, RFRLEN2, DEFAULT)     |
    MCBSP_FMKS(RCR, RWDLEN2, DEFAULT)     |
    MCBSP_FMKS(RCR, RCOMPAND, MSB)        |
    MCBSP_FMKS(RCR, RFIG, NO)              |
    MCBSP_FMKS(RCR, RDATDLY, 0BIT)        |
    MCBSP_FMKS(RCR, RFRLEN1, OF(0))       |
    MCBSP_FMKS(RCR, RWDLEN1, 32BIT)        |
    MCBSP_FMKS(RCR, RWDREVRS, DISABLE),   |

    MCBSP_FMKS(XCR, XPHASE, SINGLE)       |
    MCBSP_FMKS(XCR, XFRLEN2, DEFAULT)     |
    MCBSP_FMKS(XCR, XWDLEN2, DEFAULT)     |
    MCBSP_FMKS(XCR, XCOMPAND, MSB)        |
    MCBSP_FMKS(XCR, XFIG, NO)              |
    MCBSP_FMKS(XCR, XDATDLY, 0BIT)        |
    MCBSP_FMKS(XCR, XFRLEN1, OF(0))       |
    MCBSP_FMKS(XCR, XWDLEN1, 32BIT)       |
}

```

```

MCBSP_FMKS (XCR, XWDREVRS, DISABLE), |  

MCBSP_FMKS (SRGR, GSYNC, DEFAULT) |  

MCBSP_FMKS (SRGR, CLKSP, DEFAULT) |  

MCBSP_FMKS (SRGR, CLKSM, DEFAULT) |  

MCBSP_FMKS (SRGR, FSGM, DEFAULT) |  

MCBSP_FMKS (SRGR, FPER, DEFAULT) |  

MCBSP_FMKS (SRGR, FWID, DEFAULT) |  

MCBSP_FMKS (SRGR, CLKGDV, DEFAULT), |  

MCBSP_MCR_DEFAULT, |  

MCBSP_RCER_DEFAULT, |  

MCBSP_XCER_DEFAULT, |  

MCBSP_FMKS (PCR, XIOEN, SP) |  

MCBSP_FMKS (PCR, RIOEN, SP) |  

MCBSP_FMKS (PCR, FSXM, EXTERNAL) |  

MCBSP_FMKS (PCR, FSRM, EXTERNAL) |  

MCBSP_FMKS (PCR, CLKXM, INPUT) |  

MCBSP_FMKS (PCR, CLKRM, INPUT) |  

MCBSP_FMKS (PCR, CLKSSTAT, DEFAULT) |  

MCBSP_FMKS (PCR, DXSTAT, DEFAULT) |  

MCBSP_FMKS (PCR, FSXP, ACTIVEHIGH) |  

MCBSP_FMKS (PCR, FSRP, ACTIVEHIGH) |  

MCBSP_FMKS (PCR, CLKXP, FALLING) |  

MCBSP_FMKS (PCR, CLKRP, RISING) |  

}; |  

/* Valorile de configurare pentru codecul audio ce se scriu in registrii de  

control ai AIC23 */  

DSK6713_AIC23_Config config = { \  

    0x0017, /* Set-Up Reg 0      Control volum canal stanga intrare */ \  

        /* LRS      0      volum simultan setat stanga / dreapta: inactiv */ \  

        /* LIM      0      intrare canal stanga mut : inactiv */ \  

        /* XX       00     rezervat */ \  

        /* LIV      10111   volum intrare canal stanga: 0 dB */ \  

    } \  

    0x0017, /* Set-Up Reg 1      Control volum canal dreapta intrare */ \  

        /* RLS      0      volum simultan setat stanga / dreapta: inactiv */ \  

        /* RIM      0      intrare canal dreapta mut : inactiv */ \  

        /* XX       00     rezervat */ \  

        /* RIV      10111   volum intrare canal dreapta: 0 dB */ \  

    } \  

    0x01f9, /* Set-Up Reg 2      Control volum canal stanga casca */ \  

        /* LRS      1      volum simultan setat stanga / dreapta: activ */ \  

        /* LZC      1      detectie treceri prin 0 canal stanga: activ */ \  

        /* LHV      1111001   volum casca stanga: 0 dB */ \  

    } \  

    0x01f9, /* Set-Up Reg 3      Control volum canal dreapta casca */ \  

        /* RLS      1      volum simultan setat stanga / dreapta: activ */ \  

        /* RZC      1      detectie treceri prin 0 canal dreapta: activ */ \  

        /* RHV      1111001   volum casca dreapta: 0 dB */ \  

    } \  

    0x0011, /* Set-Up Reg 4      Controlul caii audio analogice */ \  

        /* X        0      rezervat */ \  

        /* STA     00     atenuarea tonurilor auxiliare: -6 dB */ \  

        /* STE     0      tonuri auxiliare : inactiv */ \  

        /* DAC     1      DAC: selectat */ \  

        /* BYP     0      bypass: inactiv */ \  

        /* INSEL   0      selectie intrare ADC: linie */ \  

        /* MICM   0      microfon mut: inactiv */ \  

        /* MICB   1      microfon boost: activ */ \  

    } \  

    0x0000, /* Set-Up Reg 5      Controlul caii digitale audio */ \  

        /* XXXXX   00000   rezervat */ \  

        /* DACM    0      DAC mute: inactiv */ \  

        /* DEEMPF  0      control faza: inactiv */ \  

        /* ADCHP   0      ADC filtru trece-sus: inactiv */ \  

    } \  

    0x0000, /* Set-Up Reg 6      Control Power down */ \  


```

```

/* X      0      rezervat */
/* OFF    0      alimentare dispozitiv: activ */
/* CLK    0      semnal de ceas : activ */
/* OSC    0      oscilator: activ */
/* OUT    0      iesire: activ */
/* DAC    0      DAC: activ */
/* ADC    0      ADC: activ */
/* MIC    0      microfon: activ */
/* LINE   0      line in: activ */

0x0043, /* Set-Up Reg 7      Format interfata digitala audio */
/* XX     00      rezervat */
/* MS     1       mod master/slave : master */
/* LRSWAP 0      DAC inversare stanga/dreapta: inactiv */
/* LRP    0       mod transmisie MSB/LSB */
/* IWL    00      dimensiune data intrare: 16 bit */
/* FOR    11      format date: format DSP */

0x0081, /* Set-Up Reg 8      Control rata de esantionare */
/* X      0      rezervat */
/* CLKOUT 1      divizor iesire CLK: 2 (MCLK/2) */
/* CLKIN  0      divizor intrare CLK: 2 (MCLK/2) */
/* SR,BOSR 00000 selector mod CLK (USB/normal): USB */
/* USB/N  1      rata de esantionare: ADC 48 kHz DAC 48 kHz */

0x0001  /* Set-Up Reg 9      Activare interfata digitala */
/* XX..X  00000000 rezervat */
/* ACT    1      activ */

};

/* handle pentru codecul audio */
DSK6713_AIC23_CodecHandle hAIC23_handle;
/* functie de initializare a subsistemelor placii */
void c6713_dsk_init();
/* functie de activare a pollingului pt transferuri */
void comm_poll();
/* activarea transferurilor cu intreruperi */
void comm_intr();
/* scrierea esantionului la iesire */
void output_sample(int);
/* citirea esantionului de pe intrare */
Uint32 input_sample();

```

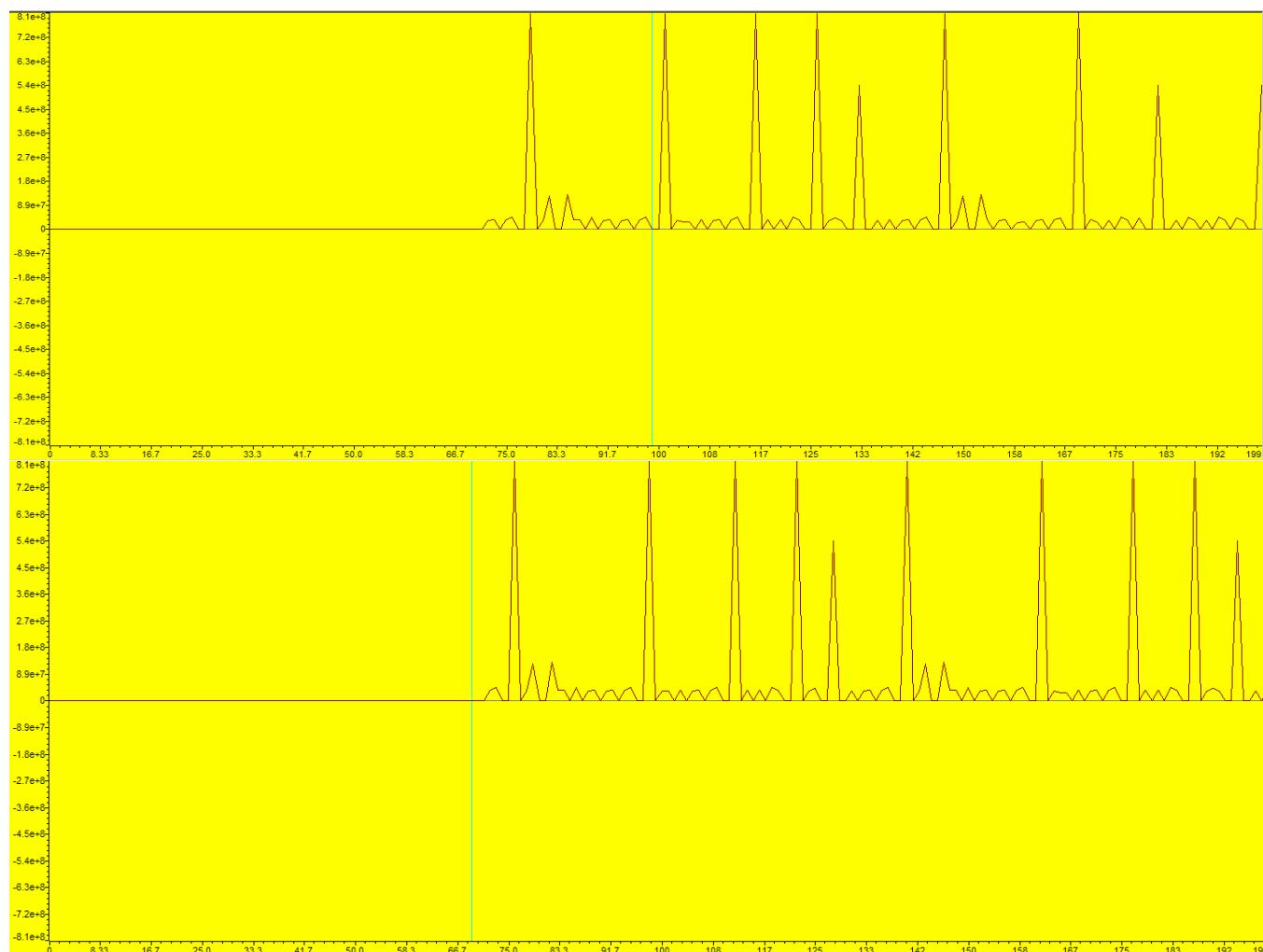
In continuare este redată starea registrilor interni ai DSP precum și starea registrilor specific McBSP.

A0 = 00000000 B0 = 000008314 PC = 00001DE0	DR0 = 00000000
A1 = 00000000 B1 = 00000001 ISTP = 00000000	DXR0 = 00000000
A2 = 00000001 B2 = 00000000 IFR = 00000000	SPCR0 = 00000000
A3 = 00007BC0 B3 = 00005E20 IER = 00000000	RCR0 = 00000000
A4 = FFFFFFFF B4 = 00007EB0 IRP = 000005C0	XCR0 = 00000000
A5 = 00000000 B5 = 000000BE NRP = 18EC01CC	SRGRO = 20000001
A6 = 00000001 B6 = 00000000 AMR = 00000000	MCR0 = 00000000
A7 = 00000002 B7 = 000085F4 CSR = 02030100	RCERO = 00000000
A8 = 00000000 B8 = 00000004 CPUID = 2	XCERO = 00000000
A9 = 00000001 B9 = 00000000 REVID = 3	PCR0 = 00000000
A10 = 00000000 B10 = 00008500 EN = 1	DRR1 = 00000000
A11 = 449F375E B11 = 00000000 PGIE = 0	DXR1 = 00000000
A12 = 00000000 B12 = 00000000 PCC = 0	SPCR1 = 00000000
A13 = 00000000 B13 = 905B2F8F DCC = 0	RCR1 = 00000000
A14 = 449F375E B14 = 00000220 GIE = 0	XCR1 = 00000000
A15 = FFFFFFFF B15 = 000089E8 SAT = 0	SRGR1 = 20000001
	PWRD = 00
	FAUCR = 00000000
	FMCR = 00000000
	FADCR = 00000000

In timpul executiei incarcarea registrilor se modifica si transferurile intre codec si DSP au loc.

A0	= 000029EC	B0	= 00000000	PC	= 00000248	DRR0	= 00000000
A1	= 00000001	B1	= 00000000	ISTP	= 00000000	DXR0	= 0000108D
A2	= 00000001	B2	= 00000000	IFR	= 00000400	SPCR0	= 00C31000
A3	= 01B7C100	B3	= 000002D0	IER	= 00000000	RCR0	= 00000000
A4	= 00000014	B4	= 00000014	IRP	= 000005C0	XCRO	= 00010040
A5	= 00000220	B5	= 00000013	NRP	= 18EC01CC	SRGR0	= 20001363
A6	= 00000020	B6	= 02030100	AMR	= 00000000	MCR0	= 00000000
A7	= 00000001	B7	= 00008000	CSR	= 02030100	RCERO	= 00000000
A8	= 00000000	B8	= 00000009	CPUID	= 2	XCERO	= 00000000
A9	= 00002980	B9	= 00000460	REVID	= 3	PCR0	= 00000A0A
A10	= 00000006	B10	= 00008500	EN	= 1	DRR1	= 00000000
A11	= 00000000	B11	= 00000000	PGIE	= 0	DXR1	= 00000000
A12	= 00000000	B12	= 00000000	PCC	= 0	SPCR1	= 00C30001
A13	= 00000000	B13	= 905B2F8F	DCC	= 0	RCR1	= 000000A0
A14	= 449F375E	B14	= 00000220	GIE	= 0	XCR1	= 000000A0
A15	= FFFFFFFF	B15	= 00002430	SAT	= 0	SRGR1	= 20000001
				PWRD	= 00	MCR1	= 00000000
				FAUCR	= 00000000	RCER1	= 00000000
				FMCR	= 00000000	XCER1	= 00000000
				FADCR	= 00000000	PCR1	= 00000003

In timpul executiei programului putem vizualiza continutul bufferului de intrare si a celui de iesire privind blocurile de date care vor fi scrise acolo. Utilizand Graph (View) putem vizualiza semnalul in domeniul timp sau il putem analiza utilizand diferite moduri de reprezentare (ex: transformate FFT).



**Cerinta 2:**

Sa se implementeze pornind de la aplicatia de looping un efect de echo. Acest efect consta in combinarea valorii de pe intrare cu o valoare ponderata a intrarii si apoi scrierea la iesire. Se recomanda rearea unui buffer in care sa se retine valorile care se vor prelua pe intrare si se vor procesa. Deasemenea, se considera si o constanta de amplificare a semnalului ce va fi scris la iesire.

Nota : Aplicatiile dezvoltate in cadrul laboratorului pot fi cu usurinta portata catre platforma cu TMS320C6416 DSP. Diferentele care apar se refera la antetul functiilor din BSL si CSL.

## DSP Laborator 7

- Implementarea functiei echo si echo cu efecte (amplitudine, intarziere si fading).
- Implementarea efectelor pe voce utilizand filtre trece jos.
- Implementarea unei scheme de bruiaj a unui semnal vocal.

### Implementarea functiei echo si echo cu efecte (amplitudine, intarziere si fading)

Prima aplicatie propusa in lucrarea de laborator presupune implementarea unui efect echo parametrizabil, care sa determine crearea unor noi efecte. Parametrizarea se refera la amplitudinea efectului pe semnalul util (modularea amplitudinii efectului), intarzierea se refera la modul de prelucrare al bufferului de intrare (index in vectorul de esantioane), iar faddingul se refera la crestere/scadere graduala a nivelului unui semnal audio pe unul din canale.

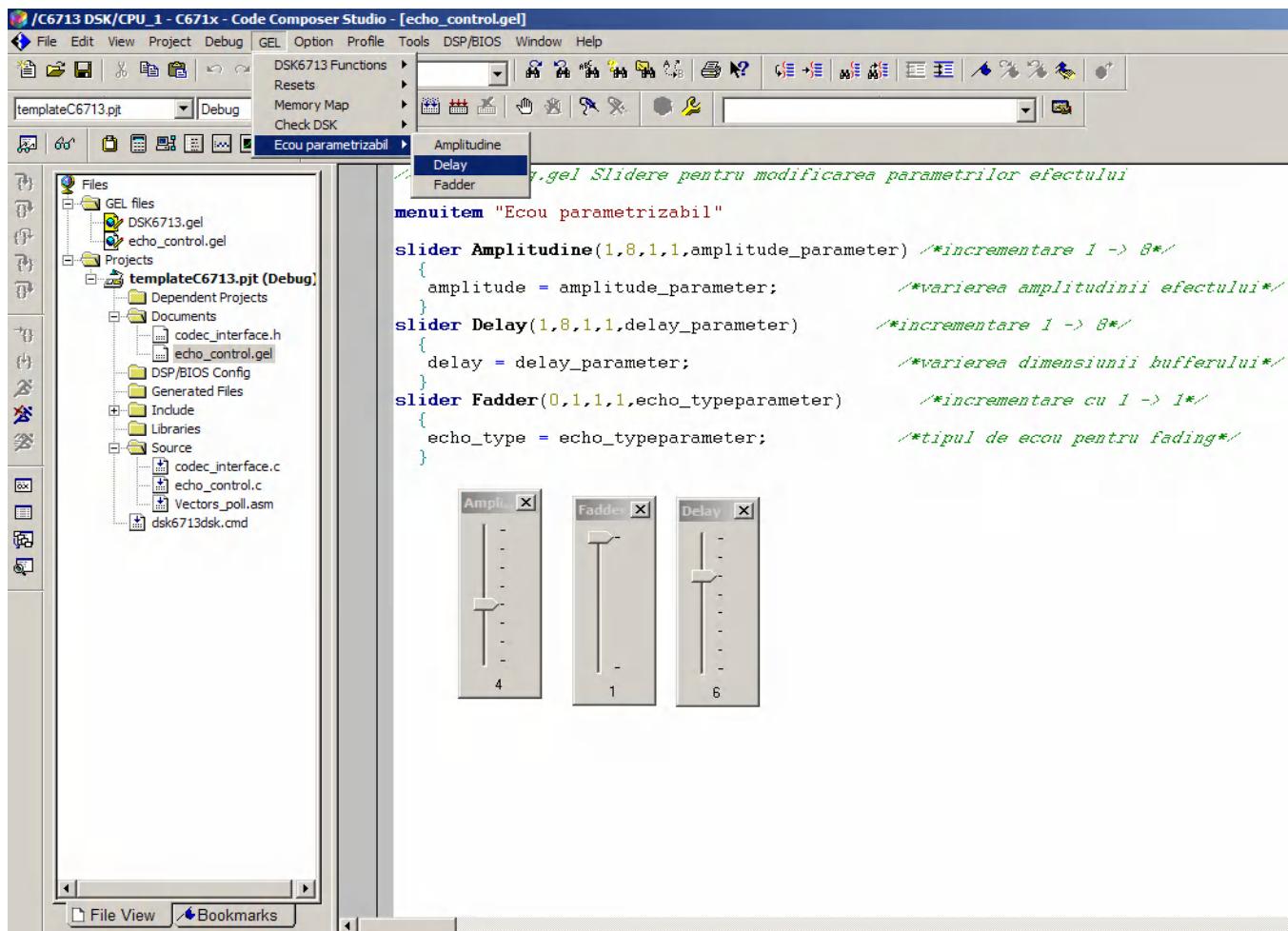
Pentru a permite utilizarea in timp real a efectelor s-a implementat o interfata GUI simpla utilizand limbajul GEL specific CodeComposer IDE. GEL (General Extension Language) este un limbaj interpretat care permite scrierea de functii pentru configurarea mediului de programare CodeComposer Studio IDE precum si pentru accesul la procesorul targetului sau la un simulator. Limbajul GEL permite automatizarea taskurilor intr-un mod asemanator macrourilor utilizand o sintaxa apropiata cu limbajul C. In acest caz limbajul GEL a fost utilizat pentru a implementa un GUI pentru a controla anumite valori de control pentru configurarea codecului de pe targetul cu DSP. Astfel, fisierul gel asociat aplicatiei de fata contine definirea a 3 slider: primul determina variația amplitudinii celui mai vechi esantion, al doilea slider este utilizat pentru a seta dimensiunea bufferului functie de intarzierea data si al treilea slider creaza efectul de fading.

Continutul fisierului GEL asociat aplicatiei (echo\_control.gel) este redat in continuare.

```
//Echo_fading.gel Slidere pentru modificarea parametrilor efectului
menuitem "Ecou parametrizabil"

slider Amplitudine(1,8,1,1,amplitude_parameter) /*incrementare 1 -> 8*/
{
    amplitude = amplitude_parameter;           /*varierea amplitudinii efectului*/
}
slider Delay(1,8,1,1,delay_parameter)        /*incrementare 1 -> 8*/
{
    delay = delay_parameter;                  /*varierea dimensiunii bufferului*/
}
slider Fadder(0,1,1,1,echo_typeparameter)     /*incrementare cu 1 -> 1*/
{
    echo_type = echo_typeparameter;          /*tipul de ecou pentru fading*/
}
```

Dupa compilarea proiectului si inainte de rulare se impune incarcarea fisierului GEL. Acest lucru se realizeaza accesand meniul File->Load GEL selectand fisierul GEL din proiect. Apoi se activeaza elementele proiectate in fisierul gel accesand GEL->Ecou Parametrizabil si activand fiecare slider dupa modul prezentat in figura ce urmeaza.



Un context de utilizare este redat in continuare. Se seteaza sliderul pentru amplitudine la pozitia 5 si sliderul de intarziere (delay) la pozitia 3. Din moment ce intarzierea nu este egala cu delay\_flag dimensiunea bufferului este schimbată. Bufferul are o noua dimensiune de bufferlength =  $1000 * 3 = 3000$ . Sliderul delay poate lua valori intre 1-8 determinand dimensiuni ale bufferului de 1000...8000. Pentru a introduce o intarziere mai mare intre esantionul cel mai nou si cel mai vechi se marea valoarea intarzierii. Sliderul fadder adauga semnalului audio un efect de fading iesirea fiind cel mai recent esantion.

Aplicatia ruleaza in mod polling (esantionele din bufferul de receptie ale codecului sunt transmise portului McBSP(1 intrucat McBSp0 este utilizat la controlul codecului), procesorul prelucreaza informatia de la McBSP si trimite apoi prin intermediul McBSP informatia in bufferul de iesire al codecului).

Pentru a lucra in mod polling, dar si in modul bazat pe intreruperi trebuie sa definim fisierele cu vectorii de intrerupere (INT, NMI, RESET) si definitiile adreselor pentru vectorii de intreruperi pentru intreruperi nemascabile sau mascabile. Fisierul template pentru lucrul in mod polling este redat in continuare.

\*Vectors\_poll.asm Fisierul de definire vectori de intrerupere pentru operarea in mod polling

```

.global _vectors
.global _c_int00
.global _vector1
.global _vector2
.global _vector3
.global _vector4
.global _vector5

```

```

.global _vector6
.global _vector7
.global _vector8
.global _vector9
.global _vector10
.global _vector11
.global _vector12
.global _vector13
.global _vector14
.global _vector15

.ref _c_int00 ;adresa de inceput

VEC_ENTRY .macro addr
    STW    B0,*--B15
    MVKL   addr,B0
    MVKH   addr,B0
    B      B0
    LDW    *B15++,B0
    NOP    2
    NOP
    NOP
.endm

_vec_dummy:
    B      B3
    NOP    5

.sect ".vecs"
.align 1024

_vectors:
_vector0: VEC_ENTRY _c_int00 ;RESET
_vector1: VEC_ENTRY _vec_dummy ;NMI
_vector2: VEC_ENTRY _vec_dummy
_vector3: VEC_ENTRY _vec_dummy
_vector4: VEC_ENTRY _vec_dummy
_vector5: VEC_ENTRY _vec_dummy
_vector6: VEC_ENTRY _vec_dummy
_vector7: VEC_ENTRY _vec_dummy
_vector8: VEC_ENTRY _vec_dummy
_vector9: VEC_ENTRY _vec_dummy
_vector10: VEC_ENTRY _vec_dummy
_vector11: VEC_ENTRY _vec_dummy
_vector12: VEC_ENTRY _vec_dummy
_vector13: VEC_ENTRY _vec_dummy
_vector14: VEC_ENTRY _vec_dummy
_vector15: VEC_ENTRY _vec_dummy

```

Codul sursa este redat in continuare.

```

//Echo_control.c Efecte de ecou cu fading
//3 slidere pentru controlul efectelor: dimensiune buffer, amplitudine, fading

#include "DSK6713_AIC23.h" //fisier de suport codec
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ; // setarea ratei de esantionare
short input, output;
short buffer[8000]; //dimensiunea maxima a bufferului
short bufferlength = 1000; //dimensiunea initiala a bufferului
short i = 0; //index buffer
short delay = 3; //determina dimensiunea bufferului
short delay_flag = 1; //flag ce marcheaza schimbarea dimenii bufferului
short amplitude = 5; //controlul amplitudinii cu sliderul

```

```

short echo_type = 1;                                //activare fader (1 / 0 dezactivare)

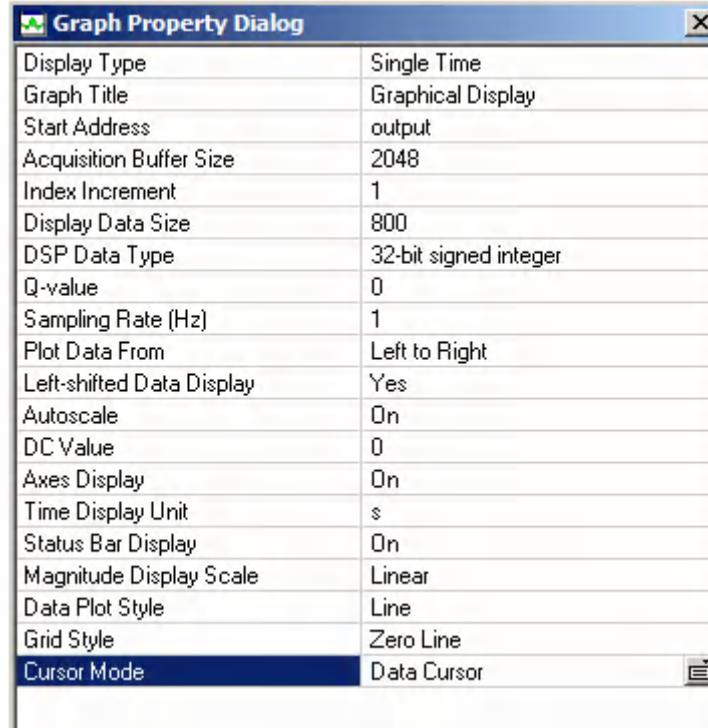
main()
{
    comm_poll();                                     //initializare DSK, codec, McBSP
    while(1)                                         //ciclu infinit
    {
        short new_count;                            //numarator pentru buffer

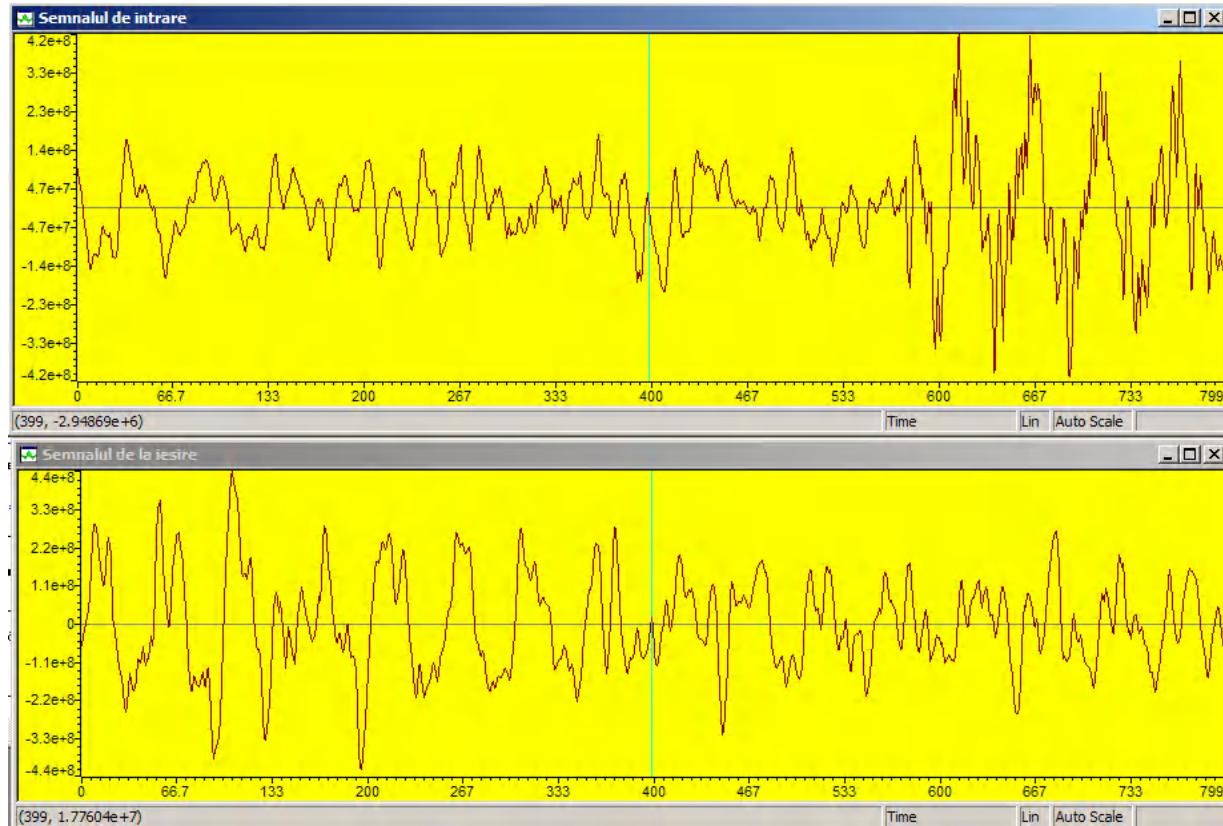
        output=input+0.1*amplitude*buffer[i];       // esantion nou + esantion vechi
        if (echo_type == 1)                          // daca fading este activ
        {
            new_count = (i-1) % bufferlength;      //locatia anterioara din buffer
            buffer[new_count] = output;           //stocheaza cea mai recenta iesire
        }
        output_sample(output);                     //scrise la iesire esantionul intarziat

        input = input_sample();                   //cel mai nou esantion de intrare
        if (delay_flag != delay)                 //daca intarzierea s-a modificat
        {
            delay_flag = delay;                //reinitializeaza pentru modificarile ulterioare
            bufferlength = 1000*delay;         //noua dimensiune a bufferului
            i = 0;                           // reinitializarea indexului
        }
        buffer[i] = input;                      //stocheaza esantionul de intrare
        i++;                                //incrementeaza indexul bufferului
        if (i == bufferlength) i=0;           //daca s-a ajuns la finalul bufferului index=0
    }
}

```

Semnalul de intrare cat si cel de iesire pot fi vizualizate cu instrumente specifice din CodeComposer Studio accesand View->Graph->Time-Frequency si selectand variabilele care sa fie plotate din fereastra de configurare.





### Cerinte :

1. Sa se modifice aplicatia proiectata pentru operarea in modul bazat pe intreruperi. Se impune setarea adresei vectorului de intrerupere pentru codec si scrierea rutinei de tratare a intreruperii corespunzatoare.
2. Sa se modifice dimensiunea maxima a bufferului astfel incat la setarea utilizand slider valoarea maxima pentru buffer sa fie de 500.

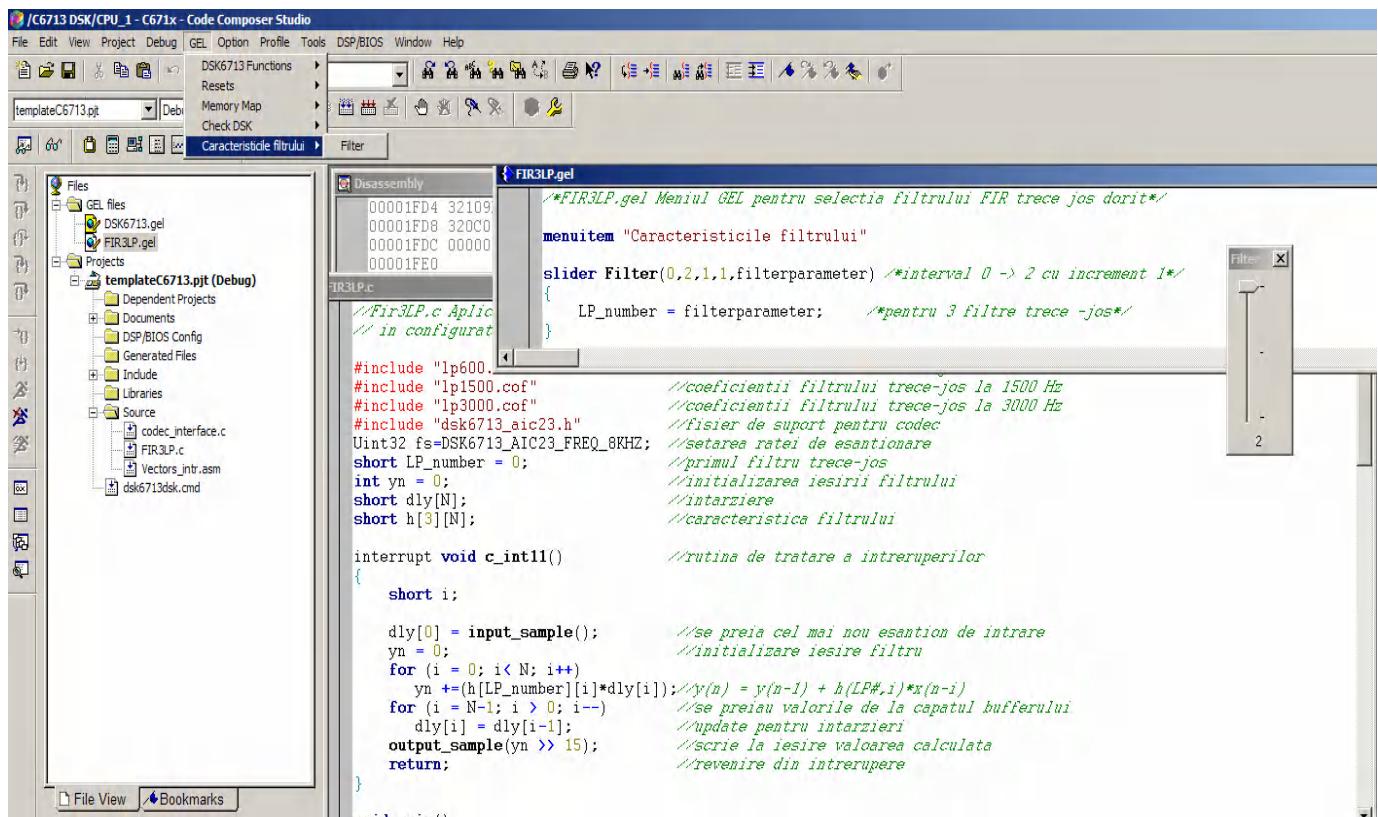
### Implementarea efectelor pe voce utilizand filtre trece jos

Pentru implementarea aplicatiei am ales utilizarea a trei filtre trece jos cu frecvențe de tăiere diferite. Pentru a genera cele trei filtre s-a utilizat MATLAB în sensul determinării seturilor de coeficienți pentru cele trei filtre cu frecvențe de tăiere de 600Hz, 1500Hz și respectiv 3000Hz. În aplicatie putem seta filtrul de ieșire dorit scriind o valoare corespunzătoare în variabila LP\_number care se poate seta utilizând un slider creat cu GEL. Astfel pentru filtrul 0, h[0][i] este hlp600[i] (i crește în bucla din main()) adică adresa primului set de coeficienți. Coeficienții sunt stocati în lp600.cof (81 de coeficienți pentru filtrul trece-jos cu frecvență de tăiere de 600Hz cu expresia analitică prezentată în sursă). Celelalte două filtre, filtrul 1 și filtrul 2 au valori de 1500Hz și respectiv 3000Hz frecvență de tăiere și pot fi cuplate schimbând sliderul LP\_number pe poziția 1 sau 2.

Fisierul GEL pentru selectorul de filtru este reprezentat în continuare împreună cu configurația curentă a mediului de programare.

```
/*FIR3LP.gel Meniul GEL pentru selectia filtrului FIR trece jos dorit*/
menuitem "Caracteristicile filtrului"

slider Filter(0,2,1,1,filterparameter) /*interval 0 -> 2 cu increment 1*/
{
    LP_number = filterparameter;      /*pentru 3 filtre trece-jos*/
}
```



Aplicatia lucreaza in modul bazat pe intreruperi, deoarece atunci cand portul serial McBSP primeste date de la codec acesta genereaza intrerupere catre DSP si DSP preia apoi informatia de pe magistrala de date de la McBSP, o prelucreaza si apoi o trimite inapoi McBSP. Informatia din bufferul de receptie McBSP este scrisa in bufferul de iesire al codecului AIC23.

Codul aplicatiei este redat in continuare.

```
//Fir3LP.c Aplicatie de implementare a efectului echo utilizand 3 filtre FIR
// in configuratie trece-jos avand latime de banda diferita
```

```
#include "lp600.cof"           //coeficientii filtrului trece-jos la 600 Hz
#include "lp1500.cof"          //coeficientii filtrului trece-jos la 1500 Hz
#include "lp3000.cof"          //coeficientii filtrului trece-jos la 3000 Hz
#include "dsk6713_aic23.h"      //fisier de suport pentru codec
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ; //setarea ratei de esantionare
short LP_number = 0;           //primul filtru trece-jos
int yn = 0;                   //initializarea iesirii filtrului
short dly[N];                //intarzire
short h[3][N];               //caracteristica filtrului

interrupt void c_int11()        //rutina de tratare a intreruperilor
{
    short i;

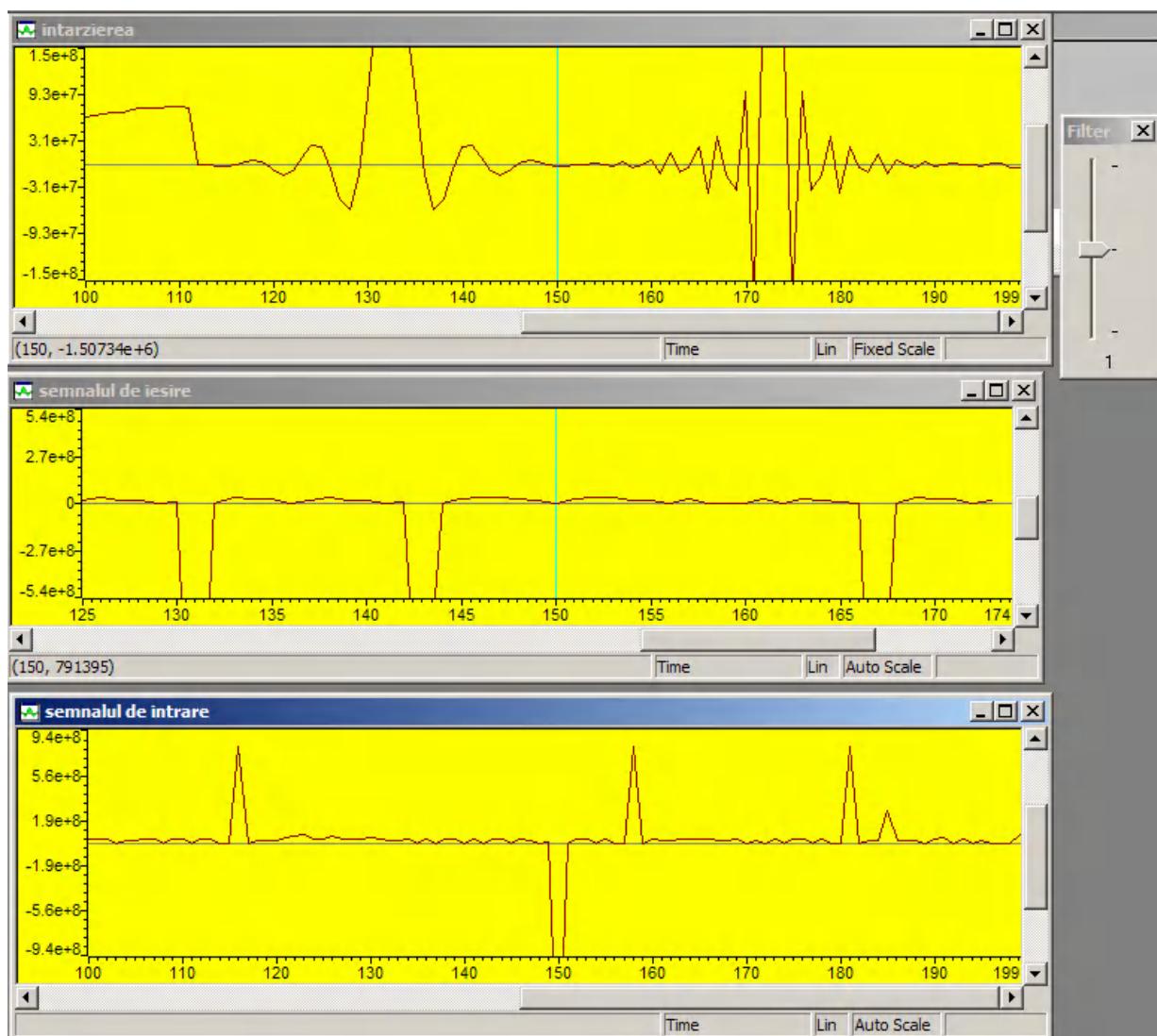
    dly[0] = input_sample();      //se preia cel mai nou esantion de intrare
    yn = 0;                      //initializare iesire filtru
    for (i = 0; i < N; i++)
        yn +=(h[LP_number][i]*dly[i]); //y(n) = y(n-1) + h(LP#,i)*x(n-i)
    for (i = N-1; i > 0; i--)    //se preiau valorile de la capatul bufferului
        dly[i] = dly[i-1];        //update pentru intarzieri
    output_sample(yn >> 15);   //scrise la iesire valoarea calculata
    return;                      //revenire din intrerupere
}
```

```

void main()
{
    short i;
    for (i=0; i<N; i++)
    {
        dly[i] = 0; //initializarea bufferului
        h[0][i] = hlp600[i]; //se preiau coeficientii FTJ la 600 Hz
        h[1][i] = hlp1500[i]; //se preiau coeficientii FTJ la 1500 Hz
        h[2][i] = hlp3000[i]; //se preiau coeficientii FTJ la 3000 Hz
    }
    comm_intr(); //initializare DSK, codec, McBSP
    while(1); //ciclu infinit
}

```

In timpul executiei aplicatiei putem analiza continutul bufferelor de intrare si de iesire.



Continutul fisierului cu coeficienti pentru filtrul trece jos cu frecventa de tajere 3000Hz este redat in continuare.

```

//LP3000 coeficientii filtrului trece jos
#define N 81 //dimensiunea filtrului

short hlp3000[N]=
{
0,-9,17,-16,0,24,
-41,35,0,-49,80,-66,
0,87,-140,113,0,-144,

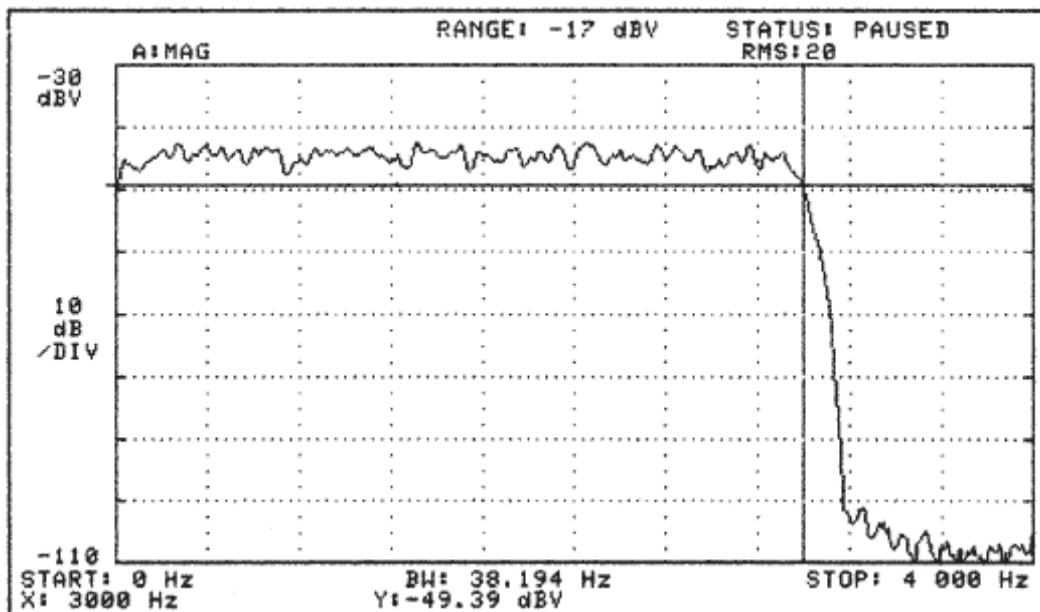
```

```

229,-182,0,228,-361,285,
0,-355,562,-446,0,565,
-905,731,0,-984,1653,-1424,
0,2428,-5186,7365,24576,7365,
-5186,2428,0,-1424,1653,-984,
0,731,-905,565,0,-446,
562,-355,0,285,-361,228,
0,-182,229,-144,0,113,
-140,87,0,-66,80,-49,
0,35,-41,24,0,-16,
17,-9,0
};

```

Utilizand un osciloscop se poate observa raspunsul in frecventa a filtrului trece jos cu o largime de banda de 3000Hz utilizand coeficientii utilizati in aplicatia dezvoltata.



Ideea de baza a aplicatiei este de a pune in evidenta prin selectia unui filtru cum anumite frecvente sunt filtrate din semnalul util de pe intrare. Astfel de exemplu prin selectia filtrului FTJ 3000Hz componentele semnalului vocal peste 3000Hz vor fi suprimate. Diferentele intre frecventele de taiere ale filtrorelor se pot distinge usor ruland aplicatia pentru un semnal vocal aplicat pe intrarea line-in.

#### Cerinta:

1. Sa se rescrie aplicatia utilizand un filtru cu comportament dat de ecuatia  $y(n) = 5*y(n-1) + 3*h(LP\#,i)*x(n-i)*x(n-i)$ , la o frecventa de esantionare de 16KHz, preluarea datelor realizandu-se continuu prin polling.

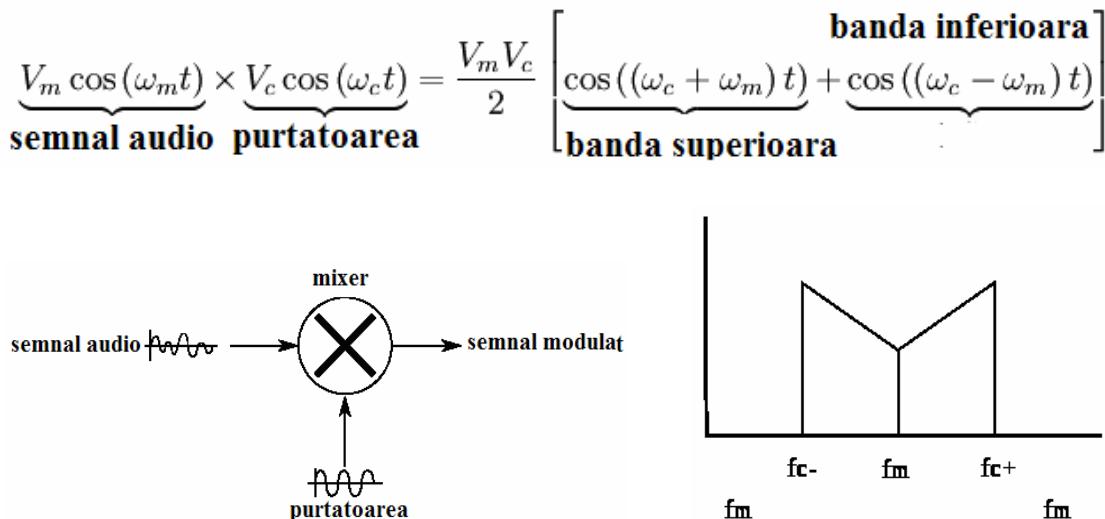
#### Implementarea unei scheme de bruiaj a unui semnal vocal

In aplicatia de fata se propune implementarea unei scheme de bruiaj a vocii si apoi de extragere a semnalului vocal din semnalul bruiat. Algoritmul implementat utilizeaza elemente simple cum ar fi filtrarea si modulatia. Pentru a obtine semnalul util ne-bruiat iesirea cu semnalul bruiat de la prima placă se aplica la intrarea celei de a doua placi. Algoritmul utilizat in aplicatia de fata este denumit in literatura de specialitate *inversarea frecventei*. Aceasta preia un semnal audio intr-o anumita gama de valori reprezentata de banda 0.3 si 3kHz si il filtreaza cu un filtru trece jos. Inversarea frecventei se obtine multiplicand (moduland) semnalul audio filtrat cu o purtatoare, determinand o inversare in spectrul de frecventa in ceea ce priveste banda superioara si cea inferioara. Astfel in banda

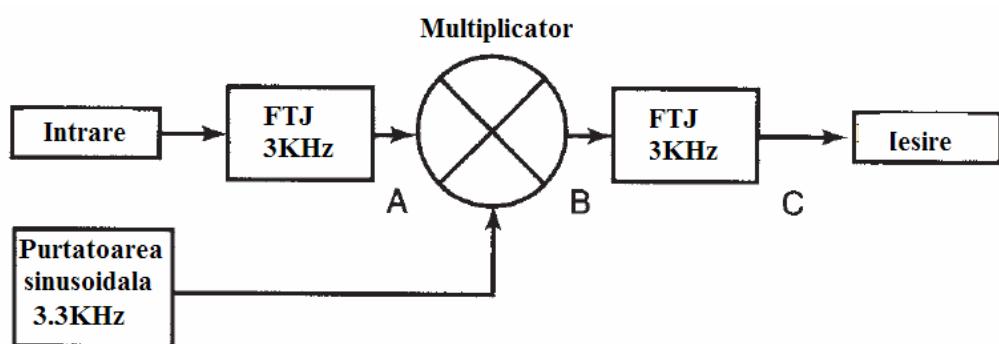
inferioara, care reprezinta gama de valori sesizabile auzului, tonurile de frecventa joasa devin tonuri de frecventa inalta si invers.

In figura care urmeaza este redat regula generala a formarii semnalului modulat in amplitudine cu purtatoarea care da frecventa si apoi o schematizare functionala a implementarii algoritmului relativ la problema aleasa in aplicatie.

Modul de generare al semnalului si spectrul sunt prezentate in prima instanta.



Particularizand schema generala pentru problema noastra avem urmatoarea schema functionala.



In continuare este redat codul aplicatiei si sunt discutate aspecte privind implementarea algoritmului.

```
//Scrambler.c Aplicatie de bruiaj / de-bruiaj a semnalului audio

#include "dsk6713_aic23.h"           //fisier de suport al codecului
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ; // setare rata de esantionare
#include "sine160.h"                 // valorile semnalului sinusoidal
#include "LP114.coF"                  // fisier cu coeficientii filtrului

// prototipurile functiilor de filtrare / modularare
short filtmodfilt(short data);
short filter(short inp, short *dly);
short sinemod(short input);
static short filter1[N],filter2[N];
short input, output;

// implementarea functiilor
```

```

short filtmodfilt(short data)           //filtrare si modulare
{
    data = filter(data,filter1); //cel mai nou esantion se plaseaza in primul filtru
    data = sinemod(data);      //modulatie sinusoidală
    data = filter(data,filter2); //filtrare utilizand un filtru trece jos(filt. 2)
    return data;
}

short filter(short inp,short *dly) //implementarea FIR
{
    short i;
    int yn;
//plaseaza cel mai nou esantion in partea inferioara a bufferului
    dly[N-1] = inp;
    yn = dly[0] * h[N-1];           //y(0)=x(n-(N-1))*h(N-1)
    for (i = 1; i < N; i++)        //ciclu pentru celelalte esantioane
    {
        yn += dly[i] * h[N-(i+1)]; //y(n)=x[n-(N-1-i)]*h[N-1-i]
        dly[i-1] = dly[i];         //update pentru intarzieri
    }
    yn = (yn >>15);             //iesirea filtrelor
    return yn;                   //returneaza y(n) la momentul n
}

short sinemod(short input)           //generarea sinusoidei (modulatia)
{
    static short i=0;

    input=(input*sine160[i++])>>11; // (intrarea)*(semnal sinusoidal)
    if(i>= NSINE) i = 0;           //daca s-a ajuns la finalul tabelei de generare sinus
    return input;                  //returneaza semnalul modulat
}

interrupt void c_int11(){           // rutina de tratare a intreruperii
    input=input_sample();          // preia de pe intrare esantionul curent
    filtmodfilt(input);           //proceseaza esantionul
    output=filtmodfilt(input);    //scrie in bufferul de iesire semnalul prelucrat
    output_sample(output);         //scrie la iesirea codecului
}

void main()
{
    short i;

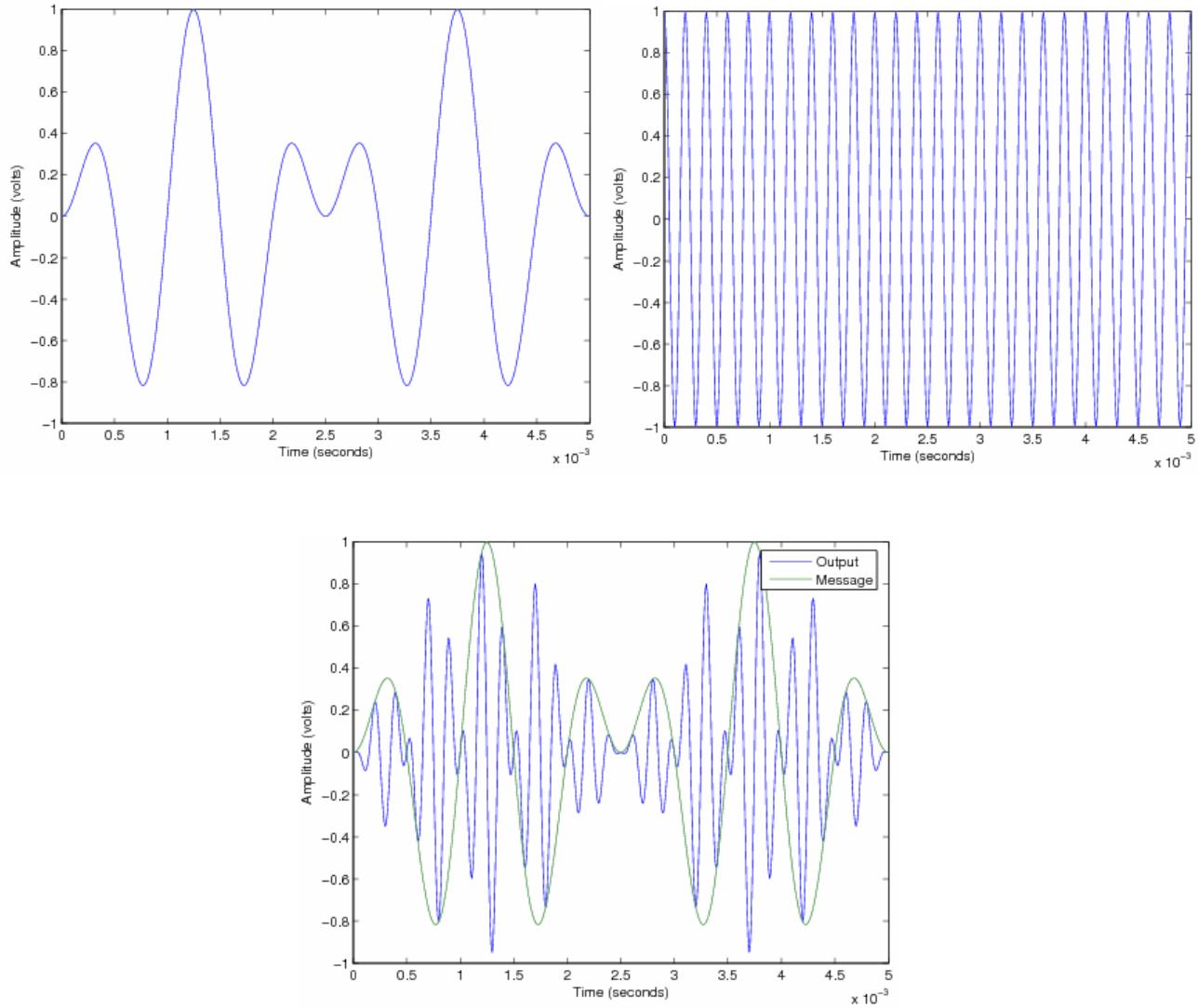
    comm_intr();                  //initializare DSK utilizand intreruperi
    for (i=0; i< N; i++)
    {
        filter1[i] = 0;           //initializare buffer primul filtru
        filter2[i] = 0;           //initializare buffer al 2 lea filtru
    }
    while(1);                    // loop
}

```

Functia filtmodfilt realizeaza apelul functiei care implementeaza primul FTJ. Iesirea filtrata devine astfel intrarea pentru mixer (modulator/multiplicator). Functia sinemod moduleaza (multiplica) semnalul filtrat cu valorile sinusoidei cu frecventa 3.3KHz. Aceasta operatie determina aparitia celor doua benzi de frecvente, superioara si inferioara. Semnalul modulat este apoi filtrat din nou astfel incat doar frecventele din banda inferioara sa fie mentinute si celelalte din banda de frecvente inalte sa fie suprimate.

S-a creat un buffer definit in `lp114.conf` pentru mentinerea coeficientilor FTJ. Pentru intarzierea esantioanelor se utilizeaza alte doua buffere, cate unul pentru fiecare filtru FTJ. Esantioanele sunt salvate in buffer astfel incat cel mai nou esantion se afla la capatul bufferului si cel mai vechi la inceputul bufferului. Semnalul purtator sinusoidal este generat dupa un tabel de valori aflat in `sine160.h` avand o frecventa de  $f = 3.3\text{Khz}$ . Utilizand iesirea de la placa ca intrare la o a doua placa care ruleaza aceeasi aplicatie semnalul vocal este extras din cel bruiat.

Descrierea algoritmului este sustinuta in continuare prin reprezentarea graficelor semnalelor de intrare, a purtatoarei si a semnalului modulat parametrizate pentru un caz oarecare.



## **Observatii privind implementarea aplicatiilor de laborator pe platforma cu DSP TMS320C6416.**

Datorita faptului ca apar anumite diferente arhitecturale intre platforma cu TMS320C6713 si TMS320C6416 am ales prezentareaa numitor detalii de proiectare si de configurare a proiectelor pentru platforma TMS320C6416 intrucat aplicatiile din laborator au fost prezentate in mare parte pentru prima platforma. Desi exista doua templateuri dezvoltate pentru fiecare din cele doua platforme in continuare sunt descrise elementele specifice platformei cu TMS320C6416 utile in dezvoltarea aplicatiilor ulterioare.

Prima indicatie se refera la fisierul de configurare a proiectului, care ar trebui sa aiba urmatoarele optiuni activate.

```
["Compiler" Settings: "Debug"]
Options=-g -fr"${Proj_dir}\Debug" -i"." -i"${Install_dir}\c6000\dsk6416\include"
-d" DEBUG" -d"CHIP_6416" -mv6400 --mem_model:data=far

["Compiler" Settings: "Release"]
Options=-O3 -fr"${Proj_dir}\Release" -mv6400

["Linker" Settings: "Debug"]
Options=-q -c -m".\Debug\templateC6416.map" -o".\Debug\templateC6416.out" -x -
-i"${Install_dir}\c6000\dsk6416\lib"      -l"dsk6416bsl.lib"     -l"csl6416.lib"      -
-l"rts6400.lib"

["Linker" Settings: "Release"]
Options=-c -m".\Release\templateC6416.map" -o".\Release\templateC6416.out" -w -x
```

Al doilea aspect se refera la modificarea configuratiei canalului de date pentru McBSP0 care controleaza (trimite cuvinte de control catre codecul AIC23). Astfel in fisierul ce implementeaza functiile de interfata cu codec (codec\_interface.h) se adauga noi variabile de setare pentru McBSP dupa cum urmeaza.

```
MCBSP_MCR_DEFAULT,           // setare pt multi channel control register
MCBSP_RCERE0_DEFAULT,        //setare receiver channel enable register0
MCBSP_RCERE1_DEFAULT,        // setare receiver channel enable register1
MCBSP_RCERE2_DEFAULT,        // setare receiver channel enable register2
MCBSP_RCERE3_DEFAULT,        // setare receiver channel enable register3
MCBSP_XCERE0_DEFAULT,         // setare transmitter channel enable register0
MCBSP_XCERE1_DEFAULT,         // setare transmitter channel enable register1
MCBSP_XCERE2_DEFAULT,         // setare transmitter channel enable register2
MCBSP_XCERE3_DEFAULT,         // setare transmitter channel enable register3
```

# DSP Laborator 8

## Generarea semnalelor uzuale :

- Generarea unui semnal dreptunghiular
- Generarea unui semnal rampă
- Generarea unui semnal sinusoidal

### Generarea unui semnal dreptunghiular

Prima aplicatie din lucrarea de laborator isi propune generarea unui semnal dreptunghiular utilizand codecul audio AIC23. Generarea efectiva a semnalului se bazeaza pe un tabel de valori (Look Up Table – LUT) a carui valori se construiesc in programul principal. In main() se creeaza un LUT buffer cu dimensiunea 64 si se incarca cu valori. Prima jumata a valorilor sunt  $2^{15}-1 = 32767$  si a doua jumata se initializeaza cu  $-2^{15} = -32768$ . La aparitia unei intreruperi de la codec, la fiecare Ts, o valoare din LUT este scrisa la iesire. Dupa ce o valoare din LUT a fost scrisa la iesire (dupa intrerupere) se revine in programul principal si se asteapta o noua intrerupere care va determina scrierea urmatoarei valori la iesire. In momentul in care s-a atins finalul bufferului, indexul acestuia este reinitializat.

Codul aplicatiei este redat in continuare.

```
//Generarea unui semnal dreptunghiular utilizand un table de look-up

#include "dsk6713_aic23.h"           //header suport pentru codec
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ; //setarea ratei de esantionare

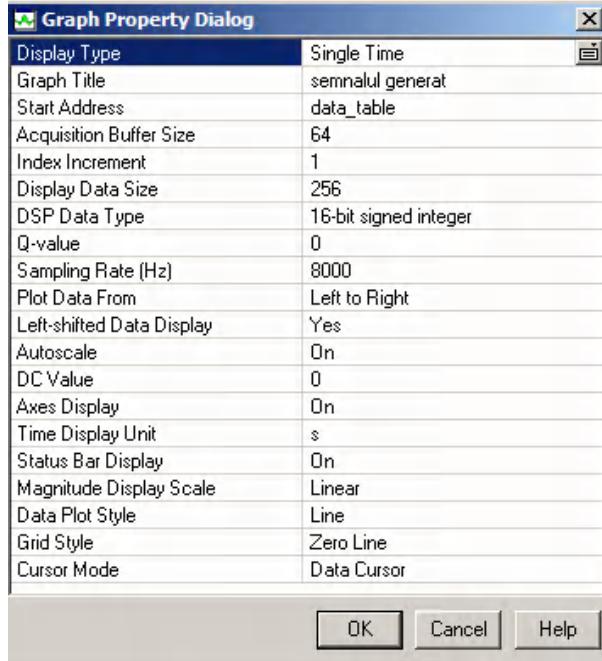
#define table_size (int)0x40          //dimensiunea tabelei de look-up
short data_table[table_size];       //tabela de look-up
int i;                             // contor buffer

interrupt void c_int11()            //rutina de tratare a intreruperilor
{
    // valoarea de iesire la fiecare perioada de esantionare
    output_sample(data_table[i]);
    if (i < table_size) ++i;         //se parcurge tabela pana la dimensiunea maxima
    else i = 0;                      //se initializeaza tabela daca s-a ajuns la final
    return;                          //se revine din rutina de tratare a intreruperilor
}

main()
{
    for(i=0; i<table_size/2; i++)   //seteaza prima jumata a bufferului
        data_table[i] = 0x7FFF;       //cu valoarea maxima ( $2^{15}-1$ )
    for(i=table_size/2; i<table_size; i++) //seteaza a doua jumata a bufferului
        data_table[i] = -0x8000;      //cu valoarea maxima  $-(2^{15})$ 

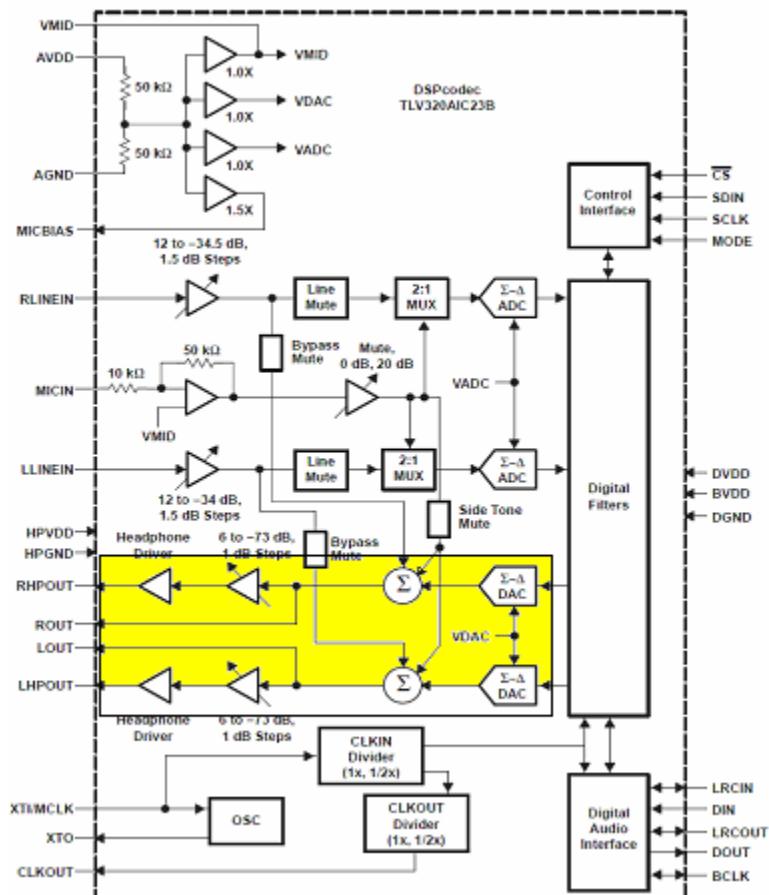
    i = 0;                          //reinitializeaza counterul
    comm_intr();                   //initializare DSk, codec si McBSP
    while (1);                     // bucla infinita
}
```

Pentru a vizualiza semnalul care va fi scris la iesire se poate utiliza instrumentul de graphing setand urmatorii parametri pentru vizualizare.



Semnalul generat prezinta o amplitudine de aproximativ 6V p-p (peak-to-peak). De remarcat ca datele valide de intrare pentru codec sunt intre  $-2^{15}$  si  $2^{15}-1$  si daca schimbam valorile din prima jumata a tablei din 0x7FFF in 0x8000 semnalul dreptunghiular nu va mai fi generat.

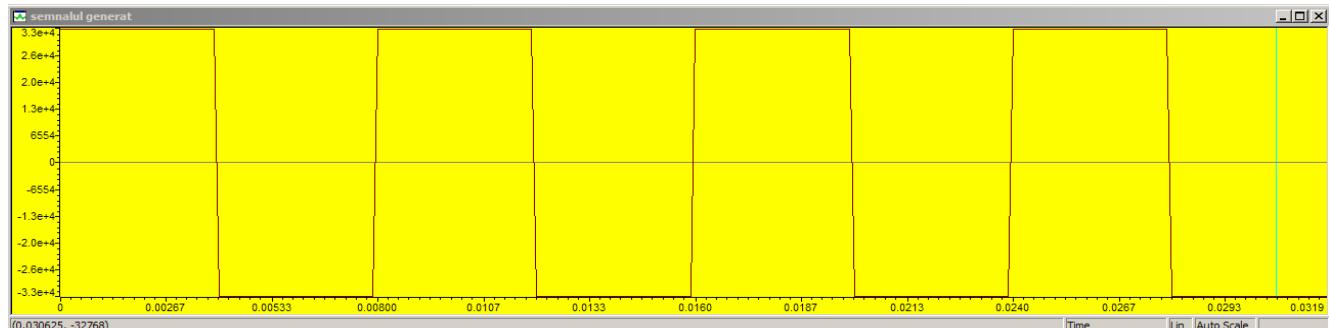
Marind numarul de puncte in tabelul LUT ( $> 0x40 = 64$ ) apare un efect de incarcare / descarcare mai pronuntat datorita condensatorului de iesire din etajul de iesire al codecului AIC23.



In cazul de fata 64 de puncte (0x40) determina o frecventa fundamentala de  $8\text{KHz}/64 = 125\text{Hz}$ . In cazul in care dublam numarul de puncte se va dubla perioada semnalului dreptunghiular si efectul descarcarii va fi mai pronuntat.

Solutia ar fi marirea perioadei de esantionare si verificarea caracteristicilor semnalului rezultat.

In cazul parametrilor considerati in aplicatie se obtine o forma de unda reprezentata in figura urmatoare.



### Cerinte:

1. Sa se dubleze numarul de puncte al LUT si sa se mareasca frecventa de esantionare la 16, respectiv 24KHz pentru cazul in care aplicatia preia continuu esantioane de la codecul audio si scrie la iesirea acestuia.

### Generarea unui semnal rampa

Aplicatia de generare a unui semnal de tip rampa utilizeaza un tabel de lookup (LUT) de dimensiune 1024 (0x400). In programul principal tabelul LUT este incarcat cu valori 0x0, 0x20, 0x40 ... (in zecimal 0, 32, 64, ..., 32736). Semnalul generat in acest mod va avea pantă (negativa) determinata de faptul ca codecul AIC23 opereaza cu format de date in complement fata de 2. Codul asociat generarii semnalului rampa este redat in continuare.

```
// Aplicatie de generare a unui semnal rampa
// Semnalul se genereaza utilizand un tabel de lookup

#include "dsk6713_aic23.h"      // headere de suport pentru codec
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ;    //setarea ratei de esantionare

#define table_size (int)0x400 //seteaza dimensiunea tabelului LUT la 1024
short data_table[table_size];   //vectorul unidimensional ce reprezinta LUT
int i;                         // index de lucru

interrupt void c_int11() //rutina de tratare a intreruperilor
{
    output_sample(data_table[i]); //valoarea ramplei la fiecare perioada de esantionare
    if (i < table_size-1) i++; //daca s-a atins finalul LUT
    else i = 0;                //se reinitializeaza indexul

    return;                    //se revine din rutina de tratare a intreruperii
}

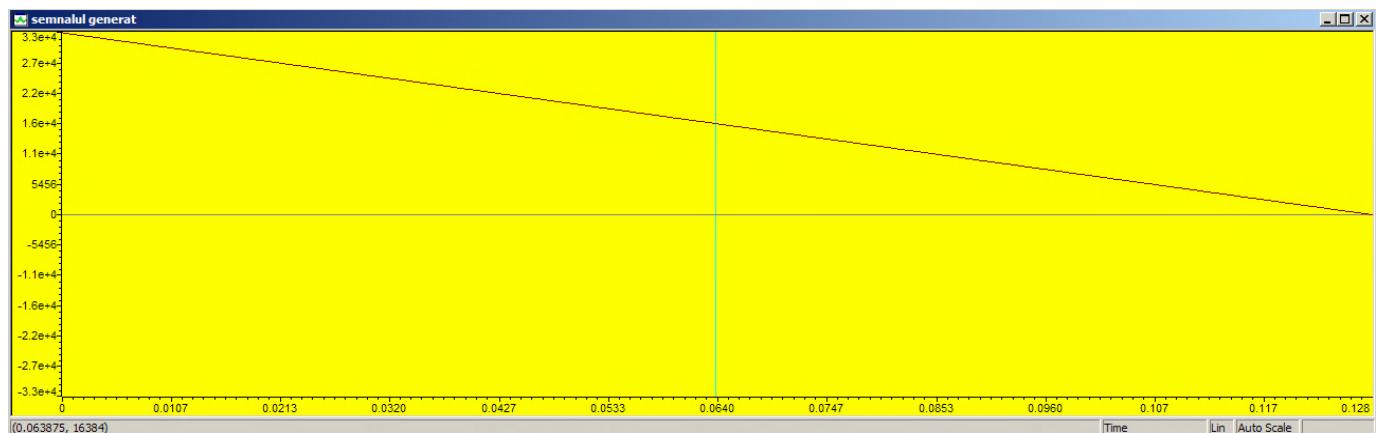
void main()
{
    for(i=0; i < table_size; i++)
    {
        data_table[i] = 0x0;      //elibereaza locatiile bufferului
        data_table[i] = i * 0x20; //genereaza valorile 0,32,64,96,...,32736
    }
    i = 0;                      //reinitializeaza indexul

    comm_intr();               //initializarea DSK, codec si McBSP
    while (1);                 //bucla infinita
}
```

Pentru a activa optiunile de vizualizare se configureaza instrumentul de graphing cu urmatorii parametri.

Graph Property Dialog	
Display Type	Single Time
Graph Title	semanul generat
Start Address	data_table
Acquisition Buffer Size	1024
Index Increment	1
Display Data Size	1024
DSP Data Type	16-bit signed integer
Q-value	0
Sampling Rate (Hz)	8000
Plot Data From	Right to Left
Left-shifted Data Display	Yes
Autoscale	On
DC Value	0
Axes Display	On
Time Display Unit	s
Status Bar Display	On
Magnitude Display Scale	Linear
Data Plot Style	Line
Grid Style	Zero Line
Cursor Mode	Data Cursor

Se poate observa cum semnalul generat are panta negativa.



Acest exemplu se bazeaza pe incarcarea unor valori intr-un tabel si apoi scrierea lor la iesirea codecului la fiecare perioada de esantionare apoi reinitializand bufferul la atingerea extremitatii.

### Cerinta :

1. Sa se modifice valoarea de parametrizare pentru LUT astfel incat rampa sa aiba o panta pozitiva, modificati numarul de puncte al reprezentarii si analizati rezultatele.
2. Sa se rescrie aplicatia de generare a semnalului rampa astfel incat incepand cu o valoare initiala 0, valoarea de iesire sa fie majorata cu 0x20 la fiecare perioada de esantionare (nu la aparitia intreruperilor de la codecul AIC23), astfel incat valorile de iesire sa fie 0, 32, 64, ..., 32736. Dupa obtinerea semnalului se cere si generarea unui semnal cu panta pozitiva.

### Generarea unui semnal sinusoidal

Aplicatia de generare a unui semnal sinusoidal utilizeaza generarea formei de unda conform unui tabel. Se considera astfel 8 puncte cu valori pentru semnalul sinusoidal si un factor de amplificare. La fiecare intrerupere generata de codec se scrie in bufferul de iesire al acestuia valoarea curenta din tabel ponderata cu factorul de amplificare.

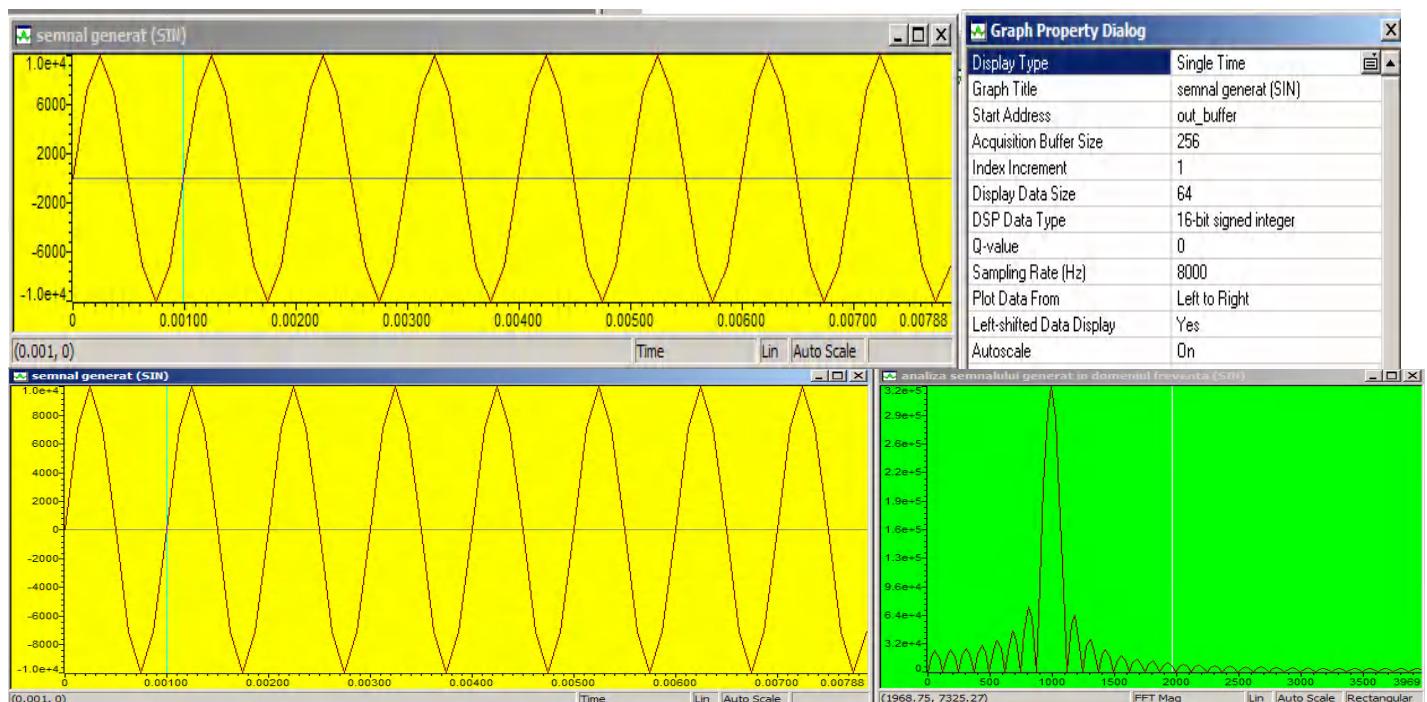
Se testeaza la fiecare pas daca s-a atins capatul tabelului si daca da se reinitializeaza contorul corespunzator. Programul principal ruleaza in bucla si asteapta intreruperi de la codecul AIC23. Pentru a permite vizualizarea semnalului generat la iesire s-a ales utilizarea unui buffer care se va alimenta permanent cu valorile scrise la iesirea codecului. Codul aplicatiei de generare a semnalului sinusoidal este listat in continuare.

```
//Aplicatie de generare a unui semnal sinusoidal utilizand o implementare cu tabel de
valori
#include "DSK6713_AIC23.h"           //fisier header de suport pentru codecul audio
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ;   //seteaza rata de esantionare

int loop = 0;                           //index tabel
short gain = 10;                      //factor
amplificare
short sine_table[8]={0,707,1000,707,0,-707,-1000,-707}; //valorile sinusoidei
short out_buffer[256];                 //bufferul de iesire
const short BUFFERLENGTH = 256;        //dimensiunea bufferului de iesire
int i = 0;                            //index buffer

interrupt void c_int11() //rutina de tratare a intreruperilor
{
    output_sample(sine_table[loop]*gain); //scrise la iesire valorile sinusoidei
    out_buffer[i] = sine_table[loop]*gain; //scrise in buffer valorile sinusoidei
    i++;                                //incrementeaza indexul de buffer
    if(i==BUFFERLENGTH) i=0;             //daca s-a ajuns la capatul bufferului reinitializeaza
    if (loop < 7) ++loop;              //verificare daca s-a ajuns la final tabel sinus
    else loop = 0;                     //reinitializeaza indexul de tabel
    return;                             //revenire din ISR
}
void main()
{
comm_intr();                         //initializare DSK, codec si McBSP (operare cu intreruperi)
    while(1);                         //cicleaza
}

Semnalul generat impreuna cu setarile pentru vizualizare precum si analiza comparativa in domeniul timp si domeniul frecventa sunt redate in continuare. Sinusoida generata are o frecventa de 1KHz.
```



**Cerinte :**

1. Sa se scrie o aplicatie de generare a unui semnal sinusoidal de 1KHz bazat pe un tabel de valori. Aplicatia trebuie sa scrie la fiecare perioada de esantionare o valoare la iesirea codecului AIC23 doar atunci cand switchul DIP 0 este activat. In plus in timpul generararii sinusoidei LED 0 sa fie comandat. In cazul in care switchul nu este activ iesirea sa fie 0.

## **Observatii privind implementarea aplicatiilor de laborator pe platforma cu DSP TMS320C6416.**

Datorita faptului ca apar anumite diferente arhitecturale intre platforma cu TMS320C6713 si TMS320C6416 am ales prezentareaa numitor detalii de proiectare si de configurare a proiectelor pentru platforma TMS320C6416 intrucat aplicatiile din laborator au fost prezentate in mare parte pentru prima platforma. Desi exista doua templateuri dezvoltate pentru fiecare din cele doua platforme in continuare sunt descrise elementele specifice platformei cu TMS320C6416 utile in dezvoltarea aplicatiilor ulterioare.

Prima indicatie se refera la fisierul de configurare a proiectului, care ar trebui sa aiba urmatoarele optiuni activate.

```
["Compiler" Settings: "Debug"]
Options=-g -fr"${Proj_dir}\Debug" -i"." -i"${Install_dir}\c6000\dsk6416\include"
-d" DEBUG" -d"CHIP_6416" -mv6400 --mem_model:data=far

["Compiler" Settings: "Release"]
Options=-O3 -fr"${Proj_dir}\Release" -mv6400

["Linker" Settings: "Debug"]
Options=-q -c -m".\Debug\templateC6416.map" -o".\Debug\templateC6416.out" -x -
-i"${Install_dir}\c6000\dsk6416\lib"      -l"dsk6416bsl.lib"     -l"csl6416.lib"      -
-l"rts6400.lib"

["Linker" Settings: "Release"]
Options=-c -m".\Release\templateC6416.map" -o".\Release\templateC6416.out" -w -x
```

Al doilea aspect se refera la modificarea configuratiei canalului de date pentru McBSP0 care controleaza (trimite cuvinte de control catre codecul AIC23). Astfel in fisierul ce implementeaza functiile de interfata cu codec (codec\_interface.h) se adauga noi variabile de setare pentru McBSP dupa cum urmeaza.

```
MCBSP_MCR_DEFAULT,           // setare pt multi channel control register
MCBSP_RCERE0_DEFAULT,        //setare receiver channel enable register0
MCBSP_RCERE1_DEFAULT,        // setare receiver channel enable register1
MCBSP_RCERE2_DEFAULT,        // setare receiver channel enable register2
MCBSP_RCERE3_DEFAULT,        // setare receiver channel enable register3
MCBSP_XCERE0_DEFAULT,         // setare transmitter channel enable register0
MCBSP_XCERE1_DEFAULT,         // setare transmitter channel enable register1
MCBSP_XCERE2_DEFAULT,         // setare transmitter channel enable register2
MCBSP_XCERE3_DEFAULT,         // setare transmitter channel enable register3
```

## DSP Laborator 9

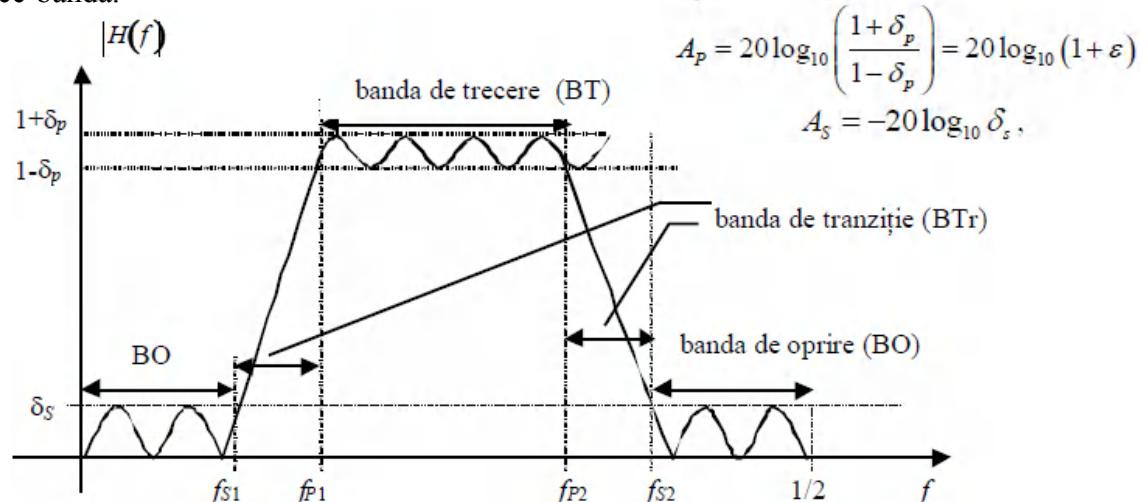
- Sinteză filtrelor digitale
  - Implementarea filtrelor FIR
- Implementarea Transformatei Fourier Rapida

### Sinteză filtrelor digitale

In lucrarea de fata se aduce in discutie implementarea filtrelor digitale utilizand platforma cu DSP. Inainte de a trece la partea de proiectare si implementare se impune o prezentare sumara a specificului filtrelor digitale si a acelor aspecte necesare in proiectare. Filtrele digitale sunt sisteme discrete care scaleaza si/sau defazeaza in mod selectiv componentelete spectrale ale semnalului discret de la intrare, oferind la iesire un semnal discret. Filtrarea este utilizata pentru a imbunatati calitatea semnalului, pentru a extrage informatii sau pentru a separa mai multe semnale combinate.

Un prim pas in proiectarea unui filtru digital este elaborarea specificatiilor filtrului. Acest process are mai multe etape:

1. Specificarea caracteristicilor filtrului, in contextul raspunsului in frecventa la nivelul frecventelor de margine din banda de tranzitie si banda de oprire. De exemplu problema se concretizeaza in parametrizarea coeficientilor din caracteristica de amplitudine a unui filtru trece-banda:



2. Determinarea parametrilor filtrului pentru a satisface conditiile impuse la proiectare.
3. Implementarea filtrului utilizand o structura specifica.

Ultimul pas se va concretiza de-a lungul lucrarii cand vom utiliza doua structuri de tip filtru FIR pentru a sintetiza filtre de uz general cu comportament FTB, FOB.

### Implementarea FIR

Filtrele FIR (Finitie Impulse Response – Filtre cu raspuns finit la impuls) sunt filtre digitale de convolutie la care semnalul de intrare este inmultit cu coeficientii sau functia pondere a filtrului (raspunsul la impuls) printr-un produs de convolutie.

Ecuatia specifica filtrului de tip FIR pentru implementarea discreta este redata in continuare.

$$y(n) = \sum_{i=0}^{N-1} h(i)x(n-i)$$

unde  $h[i]$  sunt coeficientii raspunsului la impuls al filtrului si  $N-1$  este ordinul filtrului. Functia de transfer in discret este redata de urmatoare relatie :

$$H(z) = \sum_{k=0}^{M-1} h[k]z^{-k}$$

unde  $h[k]$  sunt coeficientii raspunsului la impuls al filtrului,  $M-1$  este ordinul filtrului.

Filtrele FIR sunt specifice domeniului discret si nu pot fi obtinute prin transformarea filtrelor analogice. S-au determinat mai multe metode de aproximare a filtrelor FIR cele mai frecvent folosite fiind metoda ferestrelor de timp (ce utilizeaza Tf. Fourier inversa), metoda esantionarii in frecventa (ce utilizeaza raspunsul in frecventa) si metoda optimala (criteriu de proiectare optimal pentru minimizarea erorii maxime de aproximare).

In cele ce urmeaza sunt redate aspectele de proiectare si implementare a filtrelor FIR utilizand platforma cu DSP.

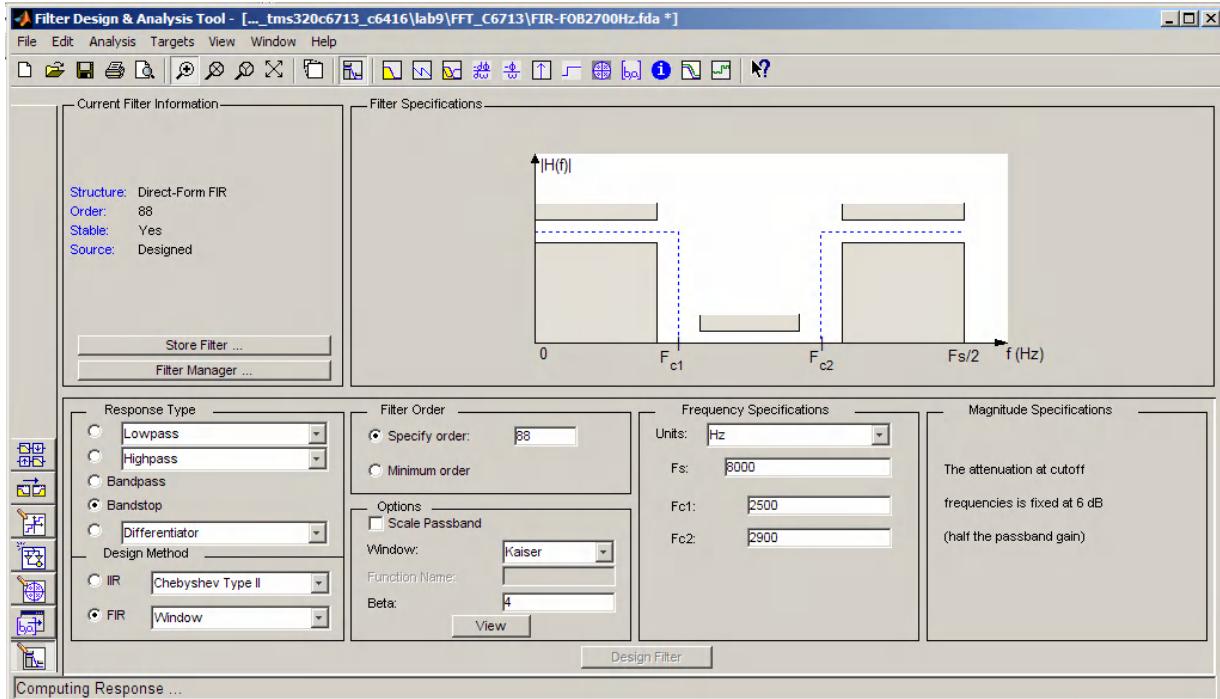
Prima etapa propusa este proiectarea filtrului FIR utilizand instrumentul FilterDesign din MATLAB. Pentru aplicatia de fata se doreste sinteza a doua filtre FIR in configuratie FOB si FTB (Filtru Opreste Banda si Filtru Trece Banda). Pentru proiectarea filtrului FIR se va utiliza metoda reprezentarii cu fereastra Kaiser. Intrucat reprezentarea filtrului impune o dezvoltare in serie, in functia de transfer se utilizeaza o functie fereastra cu anumita amplitudine pentru a neglaja coeficientii din afara ei. Astfel cu cat fereastra este mai mare cu atat aproximarea este mai precisa.

Aceasta abordare care utilizeaza ferestre este utilizata intrucat se elimina oscilatiile de amplitudine mare asigurand o trunchiere graduala catre dezvoltarea seriei de reprezentare a semnalului. O masura a performantei filtrului o reprezinta factorul de riplu care reda o comparatie intre varful lobului extrem si lobului central din reprezentarea in frecventa a semnalului filtrat.

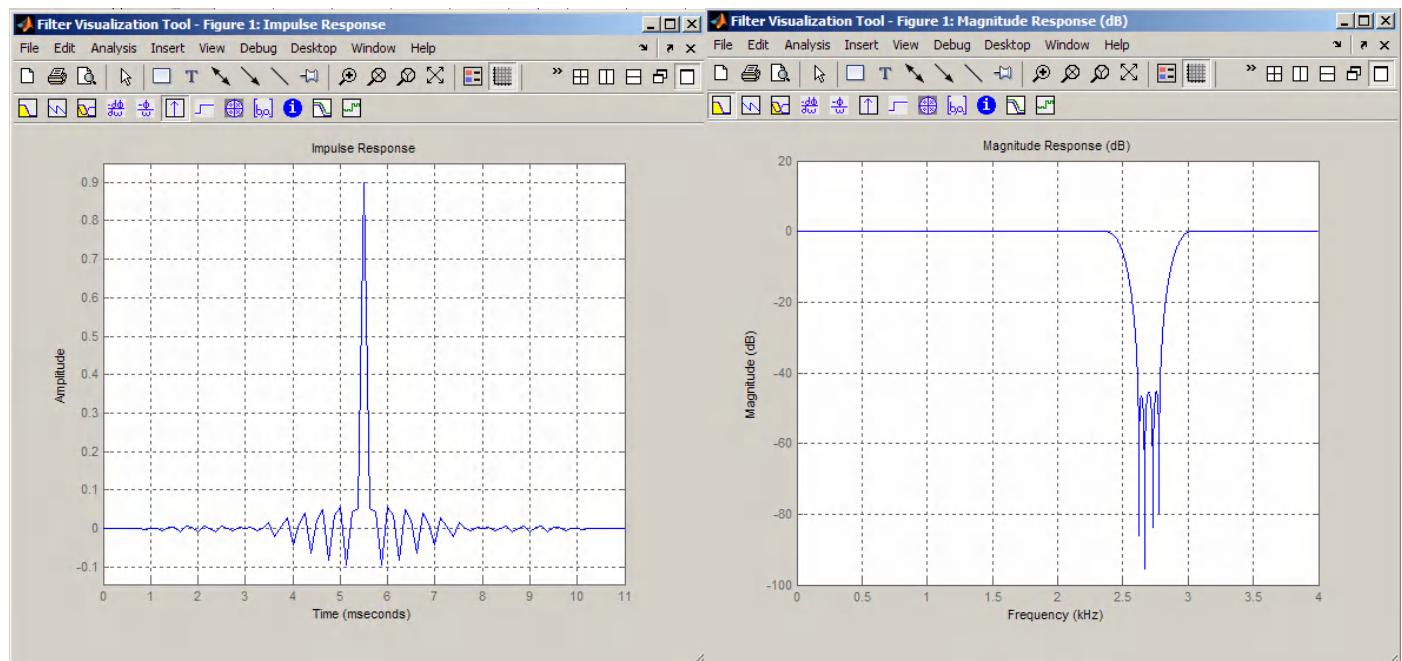
Fereastra Kaiser se bazeaza pe aproximările in timp discret ale unei functii cu suport finit in timp dar cu energie minim localizata in afara unui interval de frecventa selectat. Fereastra Kaiser este definita de un parametru  $\beta$  prin intermediul caruia se poate ajusta banda de tranzitie si nivelul spectrului. De obicei parametrul  $\beta$  ia valori in intervalul [4,9]. Pe masura ce valoarea lui  $\beta$  creste atenuarea minima in banda de oprire a filtrului, crescand si banda de tranzitie.

---

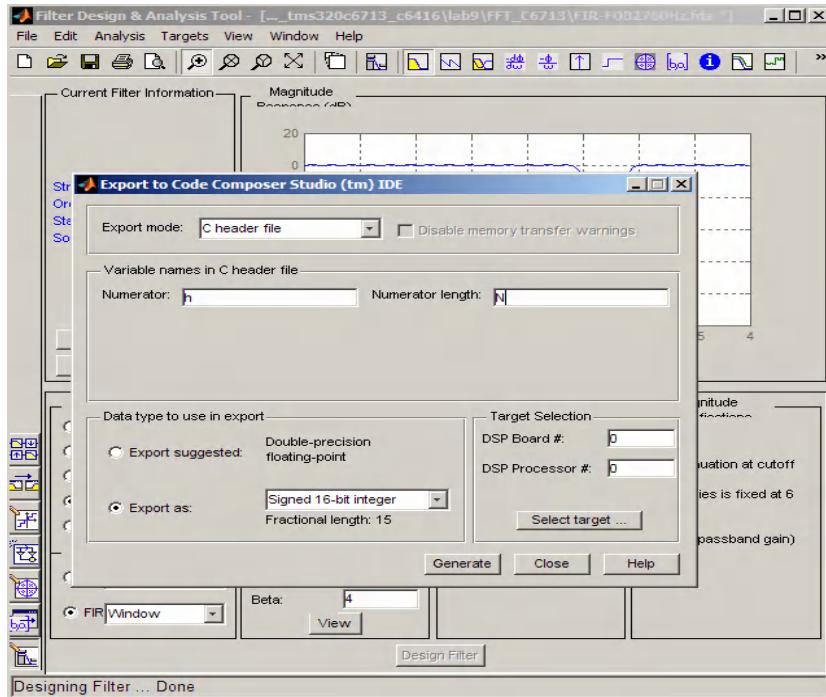
Pentru etapa de proiectare se deschide MATLAB si se apeleaza din command-line *fdatool* si o fereastra pentru proiectare va aparea.



Se parametrizeaza filtrul proiectat la valorile impuse in figura si se poate observa cum in fereastra initiala se pot stabili specificatiile filtrului si vizualiza caracteristica de freceventa generica a filtrului resultant. Pentru aplicatia propusa s-au sintetizat 2 filtre FIR. Primul filtru FIR este in configuratie FOB centrat la o frecventa de 2700Hz. Astfel se seteaza tipul si ordinul filtrului (Bandstop (Filtru Opreste Banda) ordin 88), frecventele de taiere  $F_{c1}=2500$ Hz si  $F_{c2}=2900$ Hz si in final se seteaza tipul de fereastra pentru reprezentare (fereastra Kaiser cu  $\beta=4$ ). Dupa ce s-a generat filtrul parametrizat cu valorile dorite (Design Filter) putem analiza caracteristica amplitudine-frecventa si raspunsul la impuls al filtrului utilizand instrumentul de vizualizare a filtrelor.



Dupa ce am finalizat proiectarea si analiza filtrului propus se impune sa generam coeficientii filtrului pentru aplicatia dorita. Astfel putem genera din *fdatool* al MATLAB parametri sub forma unui header direct pentru mediul de programare Code Composer Studio IDE utilizand generatorul de cod integrat (in *fdatool* meniu Targets->Code Composer Studio IDE) dupa cum este sugerat in figura urmatoare.



Fisierul generat va contine coeficientii filtrului care se vor utiliza la implementarea in cadrul aplicatiei propuse si are formatul prezentat in continuare. Acest fisier header generat (fob2700Hz.h) a fost inclus in proiectul dezvoltat pentru aplicatia curenta si redenumit in bs2700.cof ajustand si tipul de date generat pentru a fi potrivit aplicatiei.

```
/*
 * Filter Coefficients (C Source) generated by the Filter Design and Analysis Tool
 *
 * Generated by MATLAB(R) 7.7 and the Signal Processing Toolbox 6.10.
 *
 * Generated on: 18-Dec-2010 14:47:28
 *
 */

/*
 * Discrete-Time FIR Filter (real)
 * -----
 * Filter Structure   : Direct-Form FIR
 * Filter Length     : 89
 * Stable            : Yes
 * Linear Phase      : Yes (Type 1)
 */

/* General type conversion for MATLAB generated C-code */
#include "tmwtypes.h"
/*
 * Expected path to tmwtypes.h
 * C:\Program Files\MATLAB\R2008b\extern\include\tmwtypes.h
 */
/*
 * Warning - Filter coefficients were truncated to fit specified data type.
 * The resulting response may not match generated theoretical response.
 * Use the Filter Design & Analysis Tool to design accurate
 * int16 filter coefficients.
 */
const int N = 89;
const int16_T h[89] = {
    -14,     23,     -9,     -6,      0,      8,     16,     -58,      50,
    44,    -147,    119,     67,    -245,    200,     72,    -312,     257,
    53,    -299,    239,     20,    -165,     88,      0,     105,    -236,
    33,     490,    -740,    158,    932,   -1380,    392,    1348,   -2070,
    724,    1650,   -2690,   1104,   1776,   -3122,   1458,   1704,   29491,
    1704,    1458,   -3122,   1776,   1104,   -2690,   1650,     724,   -2070,
```

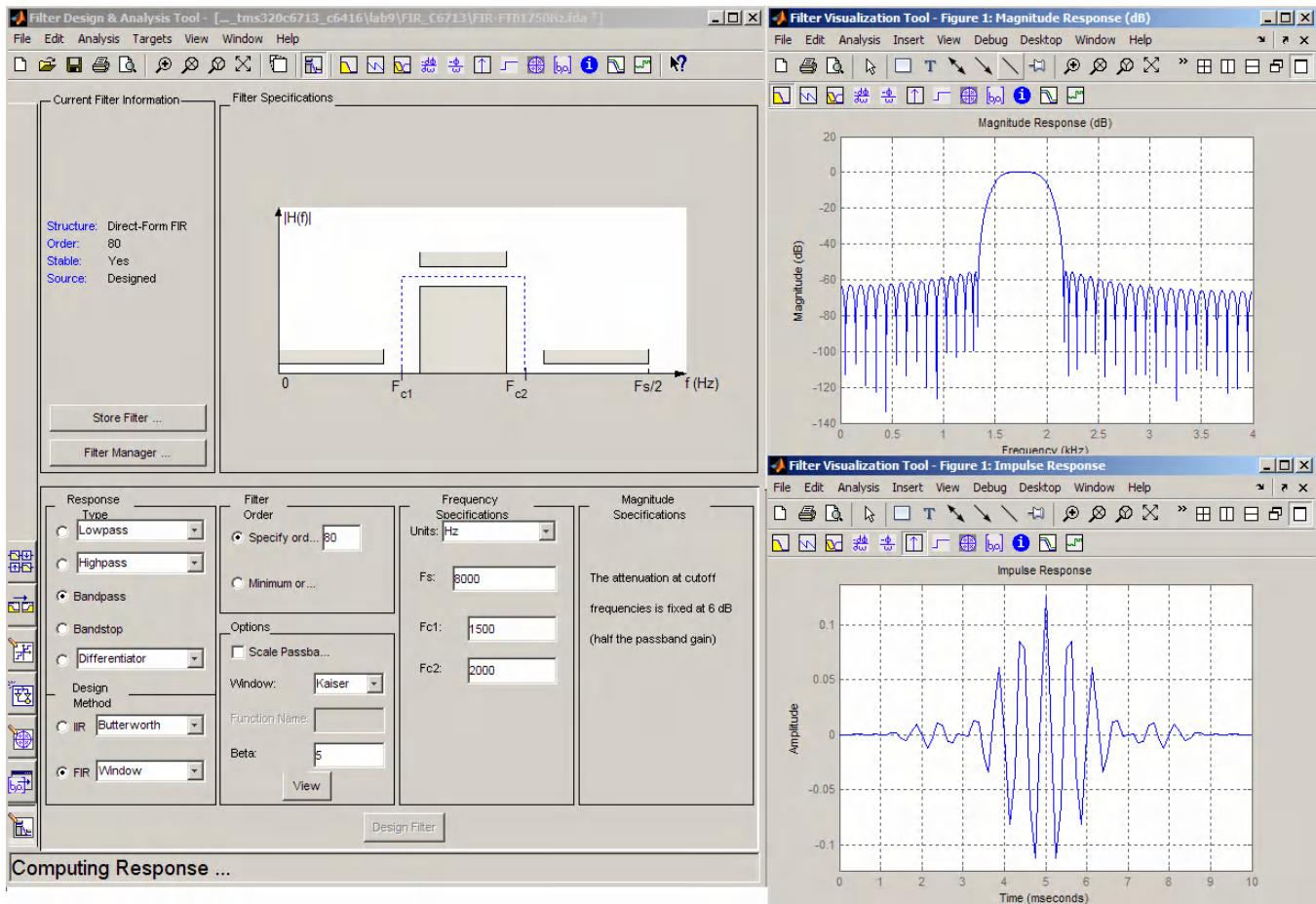
```

1348,      392,     -1380,      932,      158,      -740,      490,      33,      -236,
105,        0,       88,      -165,      20,       239,      -299,      53,       257,
-312,       72,      200,      -245,      67,       119,      -147,      44,       50,
-58,        16,       8,        0,       -6,       -9,        23,      -14
};

;

```

Al doilea filtru FIR implementat este un filtru centrat la o frecventa de 1750Hz in configuratie trece-banda, FTB. Detaliile de sinteza sunt redate in continuare impreuna cu caracteristica amplitudine-frecventa si raspunsul la impuls.



Dupa etapa de proiectare si analiza se genereaza fisierul header care va contine coeficientii filtrului FIR si se adauga proiectului pentru platforma cu DSP. Acest fisier header generat (FTB1750Hz.h) a fost inclus in proiectul dezvoltat pentru aplicatia curenta si redenumit in bp1750.cof ajustand si tipul de date generat pentru a fi potrivit aplicatiei.

In continuare sunt prezentate aspecte de proiectare ale aplicatiei si de implementare pe platforma cu DSP a filtrului digital FIR. Pornind de la ecuatia specifica a unui filtru FIR se poate reda distributia initiala a coeficientilor si esantioanelor in memorie pentru conditiile initiale.

i	Coefficienti	Esantioane
0	$h(0)$	$x(n)$
1	$h(1)$	$x(n - 1)$
2	$h(2)$	$x(n - 2)$
.	.	.
.	.	.
.	.	.
$N - 1$	$h(N - 1)$	$x(n - (N - 1))$

Coefficientii filtrului se gasesc in vectorul  $h$  astfel incat primul coefficient  $h[0]$  se afla la prima locatie a bufferului (adresa de memorie inferioara) si ultimul coefficient  $h[N-1]$  se afla la ultima locatie din bufferul coefficientilor (adresa de memorie superioara). Esantioanele intarziate sunt dispuse in buffer astfel incat cel mai nou esantion  $x(n)$  se situeaza la inceputul bufferului pentru esantioane si cel mai vechi esantion  $x(n-(N-1))$  se afla la finalul bufferului.

La momentul  $n$  cel mai nou esantion  $x(n)$  este preluat de la convertorul ADC al codecului si stocat la inceputul bufferului. La acest moment iesirea filtrului va fi obtinuta din ecuatia filtrului sub forma :

$$y[n] = h[0]x[n] + h[1]x[n-1] + \dots + h[N-2]x[n-(N-2)] + h[N-1]x[n-(N-1)]$$

Intarzirea se updateaza la fiecare pas astfel incat  $x[n-k] = x[n+1-k]$  si contribuie la determinarea iesirii la pasul de esantionare urmator,  $y[n+1]$ . Toate esantioanele sunt updateate in afara de cel mai nou esantion ( $x[n-1]=x[n]$  si  $x[n-(N-1)] = x[n-(N-2)]$ ). In continuare este redata o sinteza a modului in care se updateaza la fiecare moment bufferele de intrare si coefficientii care se utilizeaza in calculul iesirii.

Coeficienti	Esantioane		
	Pasul n	Pasul n+1	Pasul n+2
$h(0)$	$x(n)$	$x(n + 1)$	$x(n + 2)$
$h(1)$	$x(n - 1)$	$x(n)$	$x(n + 1)$
$h(2)$	$x(n - 2)$	$x(n - 1)$	$x(n)$
.	.	.	.
.	.	.	.
.	.	.	.
$h(N - 3)$	$x(n - (N - 3))$	$x(n - (N - 4))$	$x(n - (N - 5))$
$h(N - 2)$	$x(n - (N - 2))$	$x(n - (N - 3))$	$x(n - (N - 4))$
$h(N - 1)$	$x(n - (N - 1))$	$x(n - (N - 2))$	$x(n - (N - 3))$

Astfel se observa simplu ca iesirea la momentul  $n+1$  este

$$y[n+1] = h[0]x[n] + h[1]x[n-1] + \dots + h[N-2]x[n-(N-3)] + h[N-1]x[n-(N-2)]$$

si iesire la momentul  $n+2$  este data de relatia

$$y[n+2] = h[0]x[n] + h[1]x[n-1] + \dots + h[N-1]x[n-(N-3)].$$

In continuare este redat codul sursa pentru implementarea filtrelor FIR in configuratie FOB si FTB utilizand parametrii generati in MATLAB in faza de proiectare si analiza.

```
//Fir.c Filtru FIR (Finite Impulse Response)

#include "dsk6713_aic23.h" // header suport pentru codec

#define FIR_BPF

#ifndef FIR_BPF
#include "bp1750.cof" //fisierul care contine coefficientii filtrului FTB
#endif

#ifndef FIR_BSF
#include "bs2700.cof" //fisierul care contine coefficientii filtrului FOB
#endif

Uint32 fs=DSK6713_AIC23_FREQ_8KHZ; //setare rata de esantionare
int yn = 0; //initializarea iesirii filtrului
short dly[N]; //esantioane de intarziere

interrupt void c_int11() //rutina de tratare a intreruperilor
{
    short i =0;
```

```

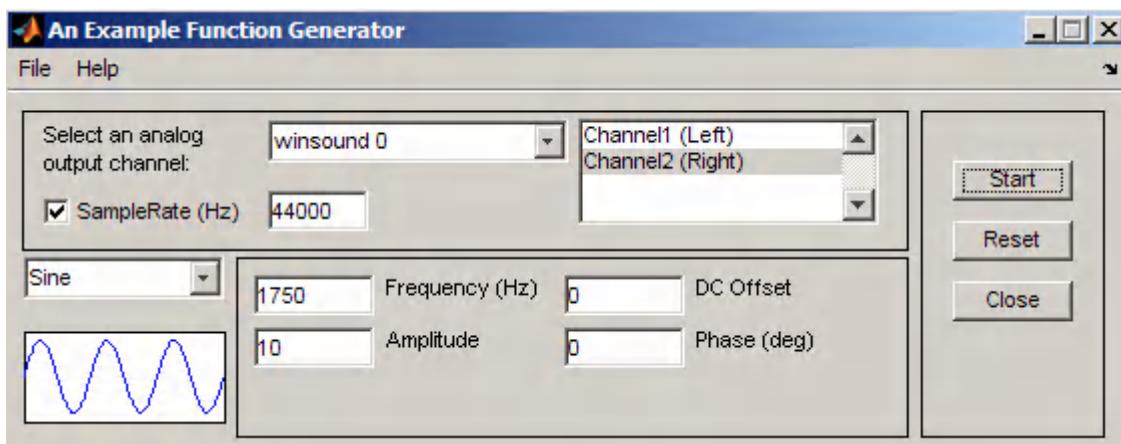
dly[0]=input_sample(); //preia cel mai nou esantion de intrare
yn = 0; //initializarea iesirii filtrului
for (i = 0; i< N; i++)
    yn += (h[i] * dly[i]); //y(n) += h(i)* x(n-i)
for (i = N-1; i > 0; i--)
    dly[i] = dly[i-1]; //se incepe cu finalul bufferului
    //refresh intarziere
output_sample((yn >> 15)); //scalarea esantionului de iesire al filtrului
return;
}

void main()
{
    comm_intr(); //initializare DSK, codec, McBSP
    while(1); //ciclu infinit
}

```

Pentru a testa aplicatia dezvoltata se va utiliza instrumentul FunctionGenerator din Toolboxul MATLAB Data Aquisition. Pentru a selecta configuratia de operare FTB sau FOB se defineste variabila simbolica corespunzatoare (`#define FIR_BPF` sau `#define FIR_BSF`) pentru a include parametrii corespunzatori.

Pentru generatorul de semnal se stabeleste urmatoarea configuratie de operare.



Semnalul se va genera prin intermediul placii de sunet a sistemului host si se alege un semnal sinusoidal parametrizat ca in figura de mai sus.

**Pentru primul caz in care se sintetizeaza un filtru FIR configuratie FTB** (Filtru Trece Banda) se aleg frecvente de generare a sinusoidei in gama [1000Hz , 2500Hz] centrate pe 1750Hz. Se poate observa comportamentul de filtru trece-banda intrucat la generarea unei sinusoide de 1750Hz semnalul audio va trece de filtru si va fi redat la iesirea codecului AIC23 in timp ce pentru frecvente spre limitele gamei de valori semnalul audio va fi suprimat (frecventele respective vor fi suprimate de filtru).

**Pentru al doilea caz in care se sintetizeaza un filtru FIR configuratie FOB** (Filtru Opreste Banda) se seteaza o frecventa de generare a sinusoidei in gama [2000Hz, 3000Hz] centrate pe 2700Hz. Se poate observa comportamentul de filtru opreste-banda intrucat la generarea unor frecvente mai mari sau mai mici de 2700Hz semnalul scris la iesirea AIC23 este sesizabil fiind suprimata (amplitudine minima) componenta la 2700Hz.

#### Cerinte:

1. Sa se proiecteze un filtru FIR FOB de ordin 80 centrata pe o frecventa de 3500Hz, cu frecvente de taiere  $F_{c1} = 3000\text{Hz}$  si  $F_{c2} = 4000\text{Hz}$ , utilizand o reprezentare cu fereastra Kaiser cu beta 6, sa se verifice implementarea pe platforma cu DSP si sa se analizeze rezultatele pe o plaja de valori de frecvente [2000Hz, 5000Hz] generate de placa de sunet.
2. Sa se proiecteze un filtru FIR FTJ de ordin 12 cu o frecventa de esantionare de 8000Hz si o frecventa de taiere de 3000Hz. Sa se implementeze pe platforma cu DSP filtrul digital si sa se

testez utilizand generatorul de semnal din MATLAB sau un semnal audio aplicat pe linia line-in a codecului AIC23. Sa se observe ce se intampla cu componentelete de frecvențe peste 3000Hz.

## Implementarea Transformatei Fourier Rapida

Transformata Fourier Rapida (FFT) este numele generic dat unei clase de algoritmi rapizi de calcul al Transformantei Fourier Discrete pentru semnale cu suport finit. Pentru o recapitulare a noțiunilor teoretice utile în analiza Fourier se prezintă un scurt breviar teoretic util apoi în implementarea aplicatiei propuse.

Definirea transformantei Fourier discrete implică parcurgerea următorilor pași:

1. Fie  $x(t)$  un semnal de durată finită  $\tau$  (fig FFT a)), al cărui model spectral  $X(\omega)$  este dat în fig. FFT c. S-a admis că  $x(t)$  are caracteristica spectrală limitată la frecvența maximă  $\omega_M$  (fig. FFT c)
2. Este posibilă construcția unui *semnal periodic*,  $x_\tau(t)$ , repetând pe  $x(t)$  la fiecare interval de timp  $\tau$  (fig. FFT b)). Deci:

$$x_\tau(t) = \sum_{m=-\infty}^{\infty} x(t-m\tau)$$

Semnalul periodic  $x_\tau(t)$  se modelează cu seria Fourier complexă (fig. FFT d), astfel:

$$x_\tau(t) = \sum_{i=-\infty}^{\infty} \underline{A}_i e^{j i \omega_0 t},$$

unde:  $\omega_0 = \frac{2\pi}{\tau}$  și

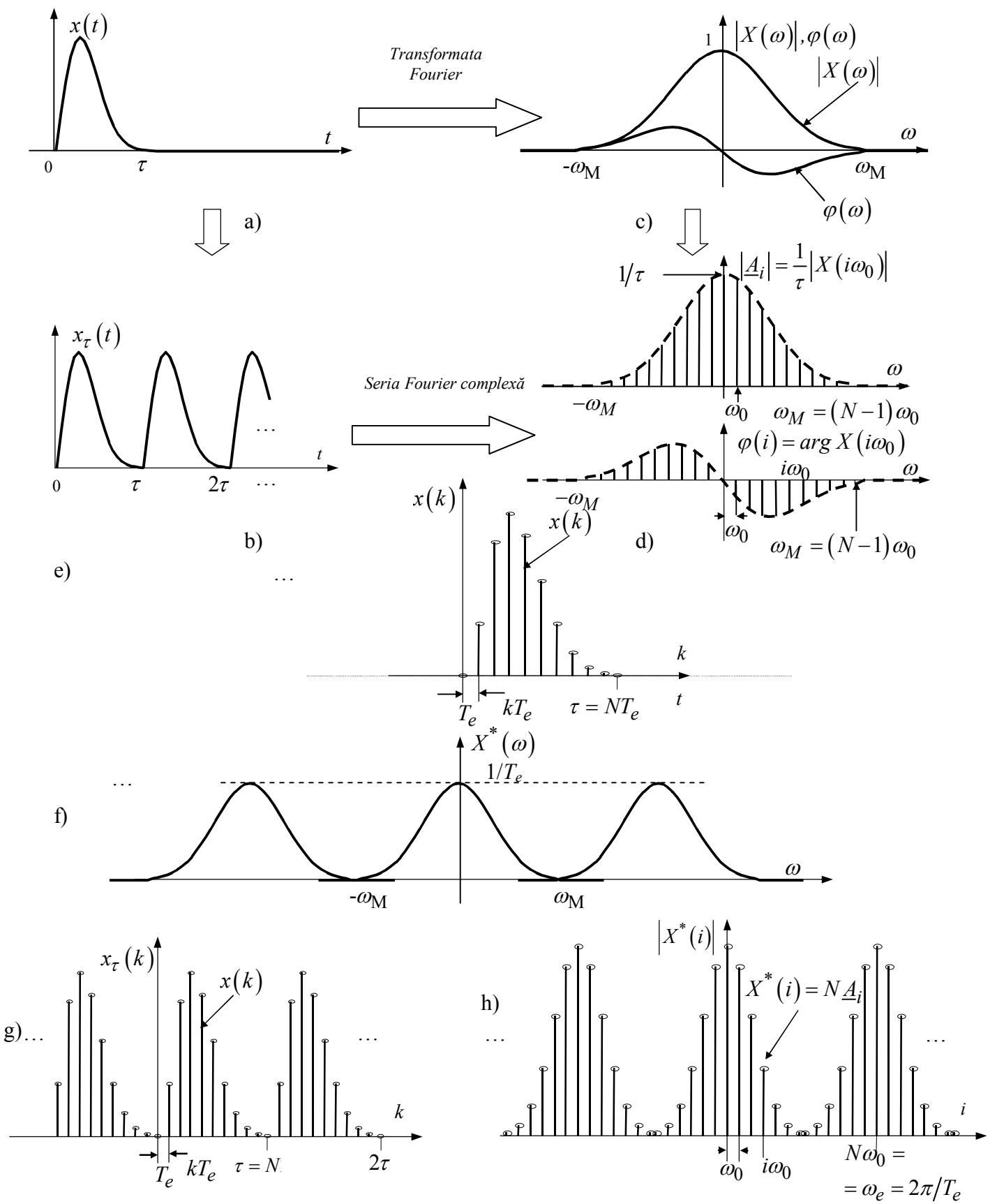
$$\underline{A}_i = \frac{1}{\tau} \int_0^\tau x_\tau(t) \cdot e^{-j i \omega_0 t} dt = \frac{1}{\tau} \int_0^\tau x(t) e^{-j i \omega_0 t} dt = \frac{1}{\tau} \int_{-\infty}^{\infty} x(t) e^{-j i \omega_0 t} dt,$$

căci  $x(t)=0$  pentru  $t < 0$  și  $t > \tau$ . Din ultima relație rezultă:

$$\underline{A}_i = \frac{1}{\tau} X(i\omega_0)$$

iar expresia  $x_\tau(t) = \sum_{i=-\infty}^{\infty} \underline{A}_i e^{j i \omega_0 t}$  devine:

$$x_\tau(t) = \frac{1}{\tau} \sum_{i=-\infty}^{\infty} X(i\omega_0) \cdot e^{j i \omega_0 t}$$



FFT Caracteristica spectrală a semnalului esantionat

Se observă că spectrul SFC al semnalului  $x_\tau(t)$  este obținut prin eşantionarea caracteristicii spectrale  $X(\omega)$  a semnalului  $x(t)$ , cu perioada de eşantionare  $\omega_0 = 2\pi/\tau$ , efectuând o modificare de scară cu  $1/\tau$  (fig. FFT c și d)). Expresia analitică a semnalului neperiodic  $x(t)$  este:

$$x(t) = \begin{cases} \frac{1}{\tau} \cdot \sum_{i=-\infty}^{\infty} X(i\omega_0) \cdot e^{j i \omega_0 t}, & 0 \leq t \leq \tau \\ 0, & \text{în rest} \end{cases}$$

3. Fie  $x^*(t) = x(kT_e) \equiv x(k)$  semnalul eşantionat, cu perioada de eşantionare  $T_e = 1/2f_M$ , unde  $f_M = \omega_M/2\pi$  (v. fig. FFT e). Alegerea perioadei de eşantionare la valoarea  $T_e = 1/2f_M$  (adică, la limita impusă de teorema lui Shannon:  $\omega_e = 2\omega_M$ ) implică valabilitatea relației  $2\omega_M = N\omega_0$  (într-adevăr,  $2\omega_M = N\omega_0 = N \frac{2\pi}{\tau} = N \frac{2\pi}{NT_e} = \omega_e$ ).

Caracteristica spectrală a semnalului eşantionat este dată în fig. FFT f (pentru simplificarea desenului, s-a renunțat la reprezentarea caracteristicii de fază). În banda de bază, această caracteristică este:

$$X^*(\omega) = \frac{1}{T_e} \cdot \sum_{i=-\infty}^{\infty} X(\omega - i\omega_e) \Big|_{i=0} = \frac{1}{T_e} \cdot X(\omega),$$

unde: 
$$X(\omega) = F\{x(t)\} = \int_{-\infty}^{\infty} x(t) \cdot e^{-j\omega t} dt = \int_0^{\tau} x(t) \cdot e^{-j\omega t} dt$$

În această integrală se discretizează timpul  $t$  cu pasul de eşantionare  $T_e$ , rezultând:

$$N = \tau/T_e$$

intervale de discretizare. Se obține:

$$X(\omega) = T_e \cdot \sum_{k=0}^{N-1} x(k) \cdot e^{-j\omega k T_e}$$

și relația devine:

$$X^*(\omega) = \sum_{k=0}^{N-1} x(k) \cdot e^{-j\omega k T_e} *$$

4. Fie acum  $x_\tau(k)$  un semnal periodic, construit repetând pe  $x(k)$  la fiecare interval de timp  $\tau = NT_e$  (v fig. FFT g). Vom discretiza axa frecvențelor  $\omega$  din caracteristica spectrală  $X^*(\omega)$ , utilizând pasul  $\omega_0$  (vezi fig. 5.24, h). Folosind relațiile anterioare și

$$2\omega_M = N \cdot \omega_0$$

relația \* devine:

$$\begin{aligned} X^*(i\omega_0) \equiv X^*(i) &= \sum_{k=0}^{N-1} x(k) \cdot e^{-ji\omega_0 k T_e} = \sum_{k=0}^{N-1} x(k) \cdot e^{-ji \frac{2\pi}{\tau} k T_e} = \\ &= \sum_{k=0}^{N-1} x(k) \cdot e^{-ji \frac{2\pi}{NT_e} k T_e} = \sum_{k=0}^{N-1} x(k) \cdot e^{-jik \frac{2\pi}{N}}, \quad i = \overline{0, N-1} \end{aligned}$$

$$\text{Deci: } X^*(i) = \sum_{k=0}^{N-1} x(k) \cdot e^{-jik\frac{2\pi}{N}}, \quad i = 0, 1, 2, \dots, N-1$$

Să înlocuim  $X(\omega)$  din  $X^*(\omega) = \frac{1}{T_e} \cdot X(\omega)$ , în  $x(t) = \begin{cases} \frac{1}{\tau} \cdot \sum_{i=-\infty}^{\infty} X(i\omega_0) \cdot e^{jia_0 t}, & 0 \leq t \leq \tau \\ 0, & \text{în rest} \end{cases}$ . Cum  $X(\omega)$  este limitată la frecvența  $\omega_M$ , se obține:

$$(5.79) \quad x(t) = \begin{cases} \frac{1}{\tau} \cdot \sum_{i=-\infty}^{\infty} X(i\omega_0) e^{jia_0 t} = \frac{T_e}{\tau} \cdot \sum_{i=0}^{N-1} X^*(i\omega_0) e^{jia_0 t}, & 0 \leq t \leq \tau \\ 0, & \text{în rest} \end{cases}$$

În relația anterioară se discretizează timpul  $t$  cu pasul  $T_e$ , punând  $t = k \cdot T_e$ , unde valorile lui  $k$  corespund intervalului  $[0, \tau]$ :  $k = 0, 1, 2, \dots, N-1$ . Rezultă:

$$x(kT_e) \equiv x(k) = \frac{T_e}{NT_e} \cdot \sum_{i=0}^{N-1} X^*(i\omega_0) e^{ji\frac{2\pi}{\tau} kT_e} = \frac{1}{N} \cdot \sum_{i=0}^{N-1} X^*(i) e^{jki\frac{2\pi}{NT_e} kT_e}, \quad k = 0, 1, \dots, N-1$$

sau

$$x(k) = \frac{1}{N} \cdot \sum_{i=0}^{N-1} X^*(i) e^{jki\frac{2\pi}{N}}, \quad k = 0, 1, \dots, N-1$$

Făcând corespondență cu seria Fourier complexă, rezultă

$$\underline{A}_i = X_i^* / N$$

Fie acum:

$$\underline{u} = e^{j\frac{2\pi}{N}}$$

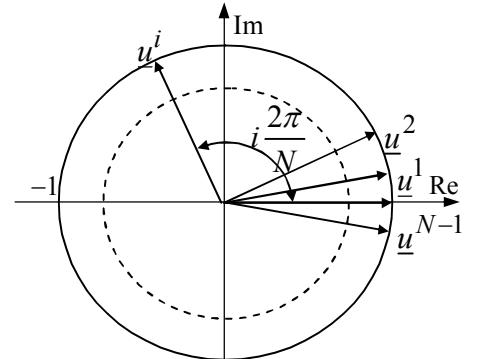
numărul complex de modul unitar și de argument  $\frac{2\pi}{N}$ , reprezentat ca

vector, și  $\underline{u}^i$ ,  $k = 0, 1, \dots, N-1$ , steaua simetrică a vectorilor de modul unitar. Relațiile anterioare devin respectiv:

$$X^*(i) = F_d \{x(i)\} = \sum_{k=0}^{N-1} x(k) \cdot u^{-ik}, \quad i = \overline{0, N-1}$$

$$x(k) = F_d^{-1} \{X^*(k)\} = \frac{1}{N} \cdot \sum_{i=0}^{N-1} X^*(i) \cdot \underline{u}^{ik}, \quad k = \overline{0, N-1}$$

Calculul transformantei Fourier discrete porneste de la dezvoltarea relaiei  $X^*(i) = F_d \{x(i)\} = \sum_{k=0}^{N-1} x(k) \cdot u^{-ik}$ ,  $i = \overline{0, N-1}$  pentru  $i = 0, 1, 2, \dots, N-1$ , se obțin  $N$  relații algebrice, care pot fi scrise sub forma matricială urmatoare :



Steaua vectorilor unitari  
 $\underline{u}^i$ ,  $i = \overline{0, N-1}$

$$\begin{bmatrix} X^*(0) \\ X^*(1) \\ X^*(2) \\ \dots \\ X^*(N-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \underline{u^{-1}} & \underline{u^{-2}} & \dots & \underline{u^{-(N-1)}} \\ 1 & \underline{u^{-2}} & \underline{u^{-4}} & \dots & \underline{u^{-2(N-1)}} \\ \dots & \dots & \dots & \dots & \dots \\ 1 & \underline{u^{-(N-1)}} & \underline{u^{-2(N-1)}} & \dots & \underline{u^{-(N-1)^2}} \end{bmatrix} \cdot \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ \dots \\ x(N-1) \end{bmatrix}$$

Pornind de la particularitățile matricei din relația anterioara (identitatea liniilor și coloanelor având același indice;  $\underline{u}^N = 1$ , etc.) a fost dezvoltat un algoritm de calcul rapid al valorilor  $X^*(k)$ ,  $k = \overline{0, N-1}$ . Transformata Fourier discretă astfel calculată se numește Transformata Fourier Rapidă sau FFT.

Aplicatia propusa ce implementeaza FFT preia semnalul pe intrarea line-in a codecului si aplica asupra lui transformarea in 256 de puncte. In programul principal se apeleaza o functie de calcul a FFT generica, coeficientii utilizati in reprezentare fiind obtinuti prin program. Amplitudinea semnalului transformat FFT este scrisa la iesirea codecului.

Aplicatia va utiliza 3 buffere si anume, `samples` (va contine datele care se vor transfera), `buffer` (utilizat pentru a scrie la iesire datele procesate si deasemenea pentru a prelua noi esantioane de la intrare), `ibuffer` (utilizat pentru afisarea semnalului primit pe intrarea codecului AIC23), `obuffer` (utilizat pentru a afisa semnalul transformat) si `x1` (contine amplitudinea semnalului transformat).

La fiecare aparitie a unei intreruperi de codec o valoare din buffer este trimisa la iesirea codecului AIC23 si o un nou esantion de intrare este preluat de la codec si stocat in buffer. Indexul `buffercount` la acest buffer este utilizat pentru a seta un `flag` cand bufferul este plin. Cand bufferul s-a umplut continutul este copiat in alt buffer (`samples`) care va fi utilizat la apelul functiei care implementeaza FFT. In acest moment amplitudinea semnalului procesat FFT continut in bufferul `x1`, poate fi copiat in bufferul principal (`buffer`) pentru a fi scris la iesire.

In cazul nostru procesarea are loc la achizitia unui cadru intreg de date (buffer intreg).

Codul aplicatiei este redat in continuare impreuna cu descrierea pasilor din algoritm si functia care implementeaza transformata.

```
//FFT256c.c Aplicatie de calcul a FFT unui semnal primit pe intrarea codecului

#include "dsk6713_aic23.h"                                // header suport pt codec
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ;    // rata de esantionare

#include <math.h>
#define PTS 256                                         // numarul de puncte pentru FFT
#define PI 3.14159265358979
typedef struct {float real,imag;} COMPLEX;
void FFT(COMPLEX *Y, int n);                            // prototipul functiei de calcul al FFT
float buffer[PTS];                                      //buffer de transfer principal
float ibuffer[PTS];                                     // buffer in vizualizare intrare
float obuffer[PTS];                                    // buffer out vizualizare iesire
float x1[PTS];                                         //buffer intermediar
short i;                                                 //variabila index
short buffercount = 0;                                  //numarul de esantioane noi in bufferul de I/O
short flag = 0;                                         //genereaza intrerupere la umplerea bufferului
COMPLEX w[PTS];                                         //constantele de ponderare stocate in w
COMPLEX samples[PTS];                                   //buffer primar de lucru

interrupt void c_int11()                                //rutina de tratare a intreruperilor
{
    output_sample((short)(buffer[buffercount])); //scrise la iesire continutul bufferului I/O
    if (buffercount >= PTS)                      //daca bufferul este plin
    {
        buffercount = 0;                           //reinitializeaza indexul bufferului
    }
}
```

```

        flag = 1;                                     //seteaza flagul de buffer plin
    }
//scrie iesirea in bufferul de iesire

    obuffer[buffercount]=(float) ((short) (buffer[buffercount]));
if (buffercount >= PTS)                                //daca bufferul este plin
{
    buffercount = 0;                                 //reinitializeaza indexul bufferului
}

buffer[buffercount]=(float) ((short)input_sample()); //citeste intrarea in bufferul de I/O
if (buffercount >= PTS)                                //daca bufferul este plin
{
    buffercount = 0;                                 //reinitializeaza indexul bufferului
    flag = 1;                                       //seteaza flagul de buffer plin
}
}
//scrie intrarea in bufferul de intrare

    ibuffer[buffercount]=(float) ((short)input_sample());
if (buffercount >= PTS)                                //daca bufferul este plin
{
    buffercount = 0;                                 //reinitializeaza indexul bufferului
}
}

main()
{
for (i = 0 ; i<PTS ; i++)           // calculeaza componentele complexe
{
    w[i].real = cos(2*PI*i/512.0); //componenta reala
    w[i].imag =-sin(2*PI*i/512.0); //componenta imaginara
}
comm_intr();                                //initializare DSK, codec, McBSP

while(1)                                    //ciclu infinit
{
    while (flag == 0) ;                    //asteapta pana cand bufferul de I/O este plin
    flag = 0;                           //reseteaza flagul
    for (i = 0 ; i < PTS ; i++)          //inverseaza bufferele
    {
        samples[i].real=buffer[i]; //buffer cu date noi
        buffer[i] = x1[i];           //datele procesate se scriu in bufferul de I/O
    }
    for (i = 0 ; i < PTS ; i++)
        samples[i].imag = 0.0;         //componentele imaginare se anuleaza

    FFT(samples,PTS);                  //apeleaza functia generatoare

    for (i = 0 ; i < PTS ; i++)          // calculeaza amplitudinea
    {
        x1[i] = sqrt(samples[i].real*samples[i].real
                      + samples[i].imag*samples[i].imag)/16;
    }
    x1[0] = 32000.0;                   //referinta varf negativ
}                                         // end while(1)
}

//FFT.c  Functie de calcul al FFT  (Fast Fourier Transform)

#define PTS 256                               //numarul de puncte pentru FFT
typedef struct {float real,imag;} COMPLEX;
extern COMPLEX w[PTS];                      //constantele de rotire stocate in w

void FFT(COMPLEX *Y, int N)                //vector esantioane de intrare, numar de puncte
{
    COMPLEX temp1,temp2;                     //variabile de stocare temporare
    int i,j,k;                            //variabile contor de bucla
    int upper_leg, lower_leg;               //indeci pt componente superioara si inferioara a semnalului
    int leg_diff;                          //diferenta intre limita superioara si cea inferioara a formei semnalului
    int num_stages = 0;                    //numarul de iteratii al FFT
    int index, step;                      //index/pas pentru constantele de ponderare w
    i = 1;                                //log(base2)din N puncte = nr de iteratii
do
{
    num_stages +=1;
    i = i*2;
}while (i!=N);
leg_diff = N/2;                            //diferenta intre limita superioara si limita inferioara
step = 512/N;                             //pas intre valori
for (i = 0;i < num_stages; i++) //pentru FFT in N puncte

```

```

{
    index = 0;
    for (j = 0; j < leg_diff; j++)
    {
        for (upper_leg = j; upper_leg < N; upper_leg += (2*leg_diff))
        {
            lower_leg = upper_leg+leg_diff;
            temp1.real = (Y[upper_leg]).real + (Y[lower_leg]).real;
            temp1.imag = (Y[upper_leg]).imag + (Y[lower_leg]).imag;
            temp2.real = (Y[upper_leg]).real - (Y[lower_leg]).real;
            temp2.imag = (Y[upper_leg]).imag - (Y[lower_leg]).imag;
            (Y[lower_leg]).real = temp2.real*(w[index]).real
                -temp2.imag*(w[index]).imag;
            (Y[lower_leg]).imag = temp2.real*(w[index]).imag
                +temp2.imag*(w[index]).real;
            (Y[upper_leg]).real = temp1.real;
            (Y[upper_leg]).imag = temp1.imag;
        }
        index += step;
    }
    leg_diff = leg_diff/2;
    step *= 2;
}
j = 0;
for (i = 1; i < (N-1); i++) //rescventiarea datelor prin inversarea bitilor
{
k = N/2;
while (k <= j)
{
    j = j - k;
    k = k/2;
}
j = j + k;
if (i<j)
{
    temp1.real = (Y[j]).real;
    temp1.imag = (Y[j]).imag;
    (Y[j]).real = (Y[i]).real;
    (Y[j]).imag = (Y[i]).imag;
    (Y[i]).real = temp1.real;
    (Y[i]).imag = temp1.imag;
}
}
return;
}

```

Pentru testarea aplicatiei se aplica la intrarea line-in a codecului AIC23 o sinusoida de 2000Hz utilizand generatorul de semnal din MATLAB (Data Aquisition Toolbox) prin intermediul placii de sunet a PC host si apoi se analizeaza reprezentarea in domeniul timp a amplitudinii semnalului transformat. Distributia varfurilor negative si pozitive din semnalul transformat reda informatii privind perioada de esantionare si frecventa de infasurare (folding). Astfel varfurile negative se afla la 256 de punct distanta (TS=32ms) interval care reda frecventa de esantionare.

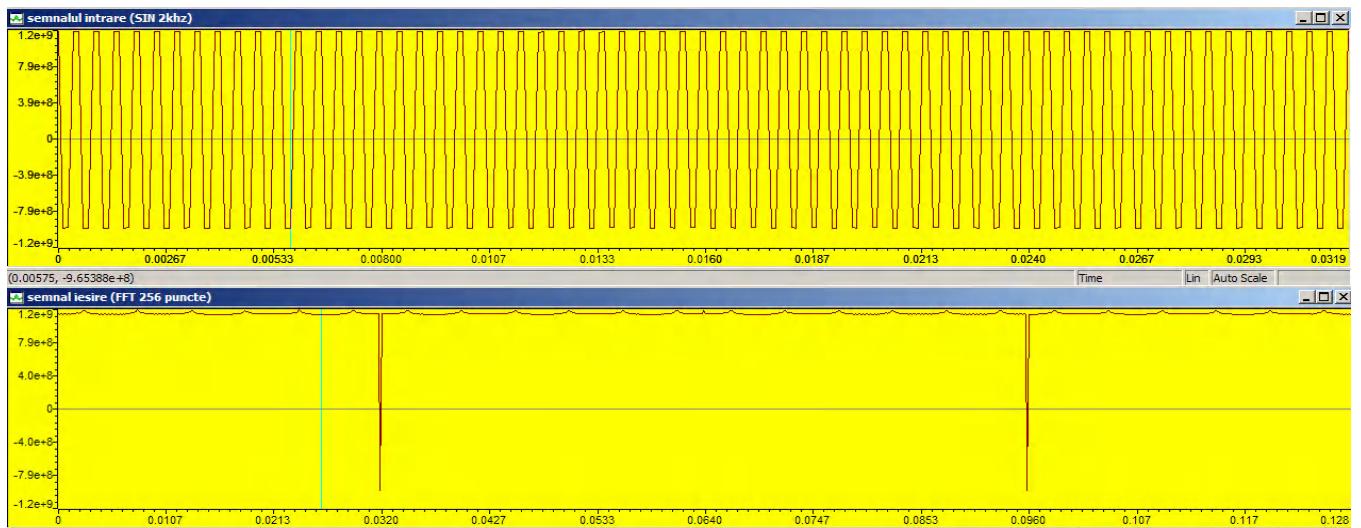
Pentru vizualizare se utilizeaza instrumentul de graphing cu parametrii setati in figura urmatoare.

Graph Property Dialog	
Display Type	Single Time
Graph Title	semnalul intrare (SIN 2khz)
Start Address	ibuffer
Acquisition Buffer Size	256
Index Increment	1
Display Data Size	256
DSP Data Type	32-bit signed integer
Q-value	0
Sampling Rate (Hz)	8000
Plot Data From	Left to Right
Left-shifted Data Display	Yes
Autoscale	On
DC Value	0
Axes Display	On
Time Display Unit	s
Status Bar Display	On
Magnitude Display Scale	Linear
Data Plot Style	Line
Grid Style	Zero Line
Cursor Mode	Data Cursor

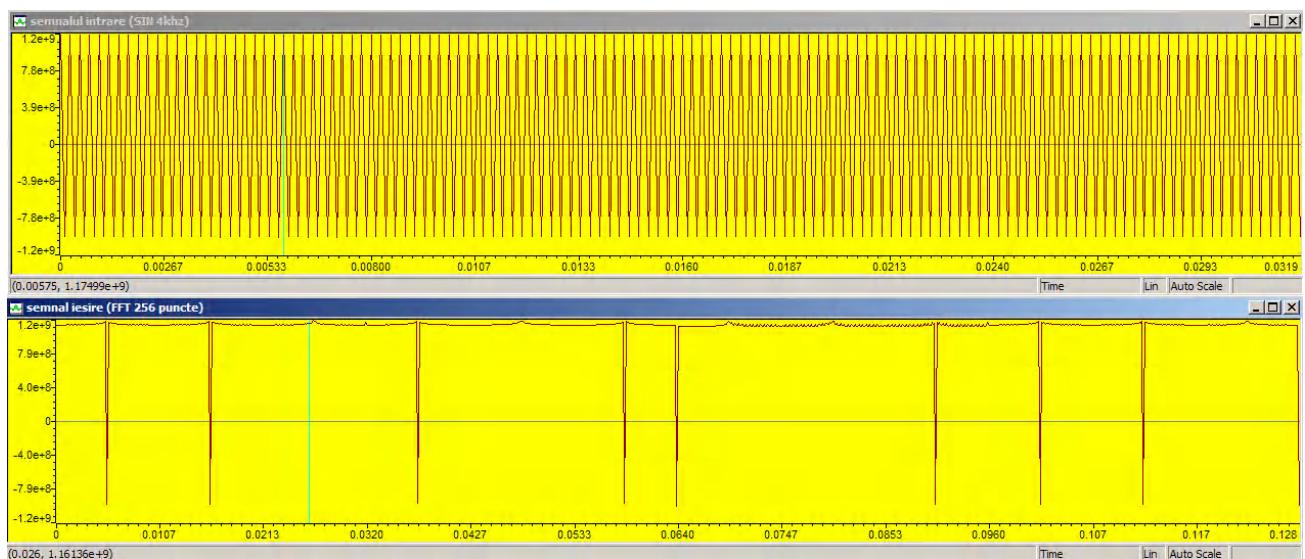
  

Graph Property Dialog	
Display Type	Single Time
Graph Title	semnal iesire (FFT 256 puncte)
Start Address	obuffer
Acquisition Buffer Size	256
Index Increment	1
Display Data Size	1024
DSP Data Type	32-bit signed integer
Q-value	0
Sampling Rate (Hz)	8000
Plot Data From	Left to Right
Left-shifted Data Display	Yes
Autoscale	On
DC Value	0
Axes Display	On
Time Display Unit	s
Status Bar Display	On
Magnitude Display Scale	Linear
Data Plot Style	Line
Grid Style	Zero Line
Cursor Mode	Data Cursor

Semnalul de intrare (sinusoida de 2KHz) si semnalul de iesire transformat utilizand FFT sunt redate in continuare.



Daca modificam frecventa semnalului de intrare la 4000Hz se poate observa cum semnalul transformat are in reprezentarea in domeniul timp alt aspect reflectand modificarile in frecventa ale semnalului de intrare.



Concluzionand se poate afirma ca Transformata Fourier Rapida este un algoritm eficient utilizat pentru a converti un semnal din domeniul timp in echivalentul sau in domeniul frecventa si are aplicabilitate in aplicatii de recunoastere vocala sau filtrari adaptive.

## **Observatii privind implementarea aplicatiilor de laborator pe platforma cu DSP TMS320C6416.**

Datorita faptului ca apar anumite diferente arhitecturale intre platforma cu TMS320C6713 si TMS320C6416 am ales prezentareaa numitor detalii de proiectare si de configurare a proiectelor pentru platforma TMS320C6416 intrucat aplicatiile din laborator au fost prezentate in mare parte pentru prima platforma. Desi exista doua templateuri dezvoltate pentru fiecare din cele doua platforme in continuare sunt descrise elementele specifice platformei cu TMS320C6416 utile in dezvoltarea aplicatiilor ulterioare.

Prima indicatie se refera la fisierul de configurare a proiectului, care ar trebui sa aiba urmatoarele optiuni activate.

```
["Compiler" Settings: "Debug"]
Options=-g -fr"${Proj_dir}\Debug" -i"." -i"${Install_dir}\c6000\dsk6416\include" -d"_DEBUG" -d"CHIP_6416" -mv6400 --mem_model:data=far

["Compiler" Settings: "Release"]
Options=-o3 -fr"${Proj_dir}\Release" -mv6400

["Linker" Settings: "Debug"]
Options=-q -c -m".\Debug\templateC6416.map" -o".\Debug\templateC6416.out" -x -i"${Install_dir}\c6000\dsk6416\lib" -l"dsk6416bsl.lib" -l"csl6416.lib" -l"rts6400.lib"

["Linker" Settings: "Release"]
Options=-c -m".\Release\templateC6416.map" -o".\Release\templateC6416.out" -w -x
```

Al doilea aspect se refera la modificarea configuratiei canalului de date pentru McBSP0 care controleaza (trimite cuvinte de control catre codecul AIC23). Astfel in fisierul ce implementeaza functiile de interfata cu codecul (codec\_interface.h) se adauga noi variabile de setare pentru McBSP dupa cum urmeaza.

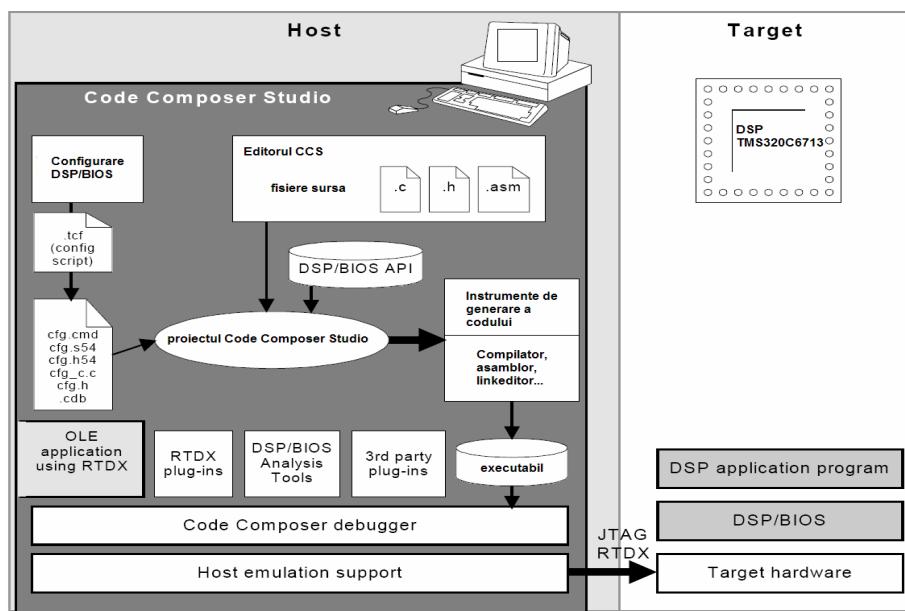
```
MCBSP_MCR_DEFAULT,           // setare pt multi channel control register
MCBSP_RCERE0_DEFAULT,        //setare receiver channel enable register0
MCBSP_RCERE1_DEFAULT,        // setare receiver channel enable register1
MCBSP_RCERE2_DEFAULT,        // setare receiver channel enable register2
MCBSP_RCERE3_DEFAULT,        // setare receiver channel enable register3
MCBSP_XCERE0_DEFAULT,        // setare transmitter channel enable register0
MCBSP_XCERE1_DEFAULT,        // setare transmitter channel enable register1
MCBSP_XCERE2_DEFAULT,        // setare transmitter channel enable register2
MCBSP_XCERE3_DEFAULT,        // setare transmitter channel enable register3
```

# DSP Laborator 10

- **Aplicatii utilizand nucleul de sistem de operare DSP/BIOS**
  - Descrierea nucleului sistemului de operare DSP/BIOS.
  - Aplicatie comanda LED.
  - Aplicatie de generare semnal sinusoidal.
- **Aplicatii utilizand TMS320C6713/TMS320C6416 si MATLAB/Simulink**
  - Generarea efectelor de echo si reverb pe un semnal util pe line-in utilizand interfata Simulink cu TMS320C6713/ TMS320C6416.

## Descrierea nucleului sistemului de operare DSP/BIOS

DSP/BIOS este un kernel real-time scalabil care implementează primul nivel de abstractizare al platformei de dezvoltare cu DSP TMS320C6713 dar este prezent și la DSP TMS320C6416. Acesta a fost proiectat pentru a susține dezvoltarea de aplicații care necesită o planificare și o sincronizare real-time a task-urilor, comunicația gazdă – target (calculator gazdă unde se dezvoltă aplicația – platformă de dezvoltare cu DSP), precum și implementarea unui sistem de instrumentație real-time. Kernelul oferă specificul unui multi-threading preemptiv, un prim nivel de abstractizare al hardware-ului, mijloace de analiză real-time a performanțelor aplicațiilor și instrumente de configurare.



Kernelul DSP/BIOS a fost proiectat pentru a minimiza cerințele de resurse de memorie și CPU de la target, prin implementarea modulară și suportul unui API pentru asigurarea configurației și optimizării modulelor dezvoltate. Structurile de date interne ale kernelului sunt optimizate, iar dimensiunea codului redusă, prin posibilitatea de a configura static și de a combina entitățile în programe executabile. Fiecare API corespunzător modulelor este modularizat, librăriile fiind optimizate pentru a necesita un număr minim de cicli de ceas la execuție, o mare porțiune fiind implementată în assembly. Comunicația între gazdă, pe care rulează instrumentele de analiză DSP/BIOS și target (placa de dezvoltare) se realizează într-o buclă de fundal (background thread), datele de instrumentație (loguri și fișiere trace-file de date) fiind formatare de gazdă. Rularea în background a instrumentelor de analiză ale kernel-ului nu interferează cu execuția curentă a taskurilor program. Dezvoltarea de aplicații este susținută de un puternic set de caracteristici ale kernelului care prin structurile de date și mecanismele de execuție implementate determină facilități importante. Astfel un program poate crea sau distruge obiecte funcție de contextul de

execuție în care se află, utilizând fie obiecte create dinamic, fie obiecte create static. Kernelul oferă un model de execuție bazat pe threaduri, implementând tipuri distințe de threaduri funcție de situație, cu posibilitatea de a controla prioritățile și caracteristicile de comunicare și interblocaj ale threadurilor.

Astfel sunt suportate threaduri specifice întreruperilor hardware, întreruperilor software, taskurilor utilizator, funcțiilor idle și funcțiilor periodice. Sunt implementate deasemenea și structuri de realizare a comunicației și sincronizării inter-thread, cum ar fi semafoarele, mailbox-urile și resource locks (echivalentul mutex-urilor din arhitecturile moderne de SO). Din punctul de vedere al subsistemului de I/O sunt suportate două modele de implementare, pentru a suporta performanțe maxime. Astfel un prim model I/O este cel bazat pe pipe-uri care sunt utilizate în comunicarea target/host și situații simple de intercomunicare între threaduri când un thread scrie date în pipe, iar un altul citește datele din pipe. Al doilea model I/O implementat este cel bazat pe stream-uri (fluxuri), care este mai complex și este dezvoltat pentru a susține driverele. Managementul erorilor este asigurat de primitive de sistem de nivel scăzut, care suportă implementarea de tipuri de date generice și managementul utilizării memoriei, pentru a minimiza creșterea cererii de resurse de memorie și CPU. Un ultim aspect legat de prezentarea kernelului DSP/BIOS se referă la instrumentul de configurare, care generează cod pentru declararea statică a obiectelor utilizate în programul dezvoltat. Scripturile de configurare pot fi create și modificate pentru a include instrucțiuni de salt, ciclare și testare a parametrilor din linia de comandă. În continuare sunt prezentate componentele DSP/BIOS, cu implicații directe în generarea programului și mediului de debugging al IDE de dezvoltare CCS.

Pentru început sunt descrise modulele DSP/BIOS și instrumentele de configurare și analiză. Astfel după cum am menționat anterior API-ul DSP/BIOS este împărțit în module. Programele aplicație utilizează DSP/BIOS-ul prin apelul de funcții din modulele API, care oferă interfețe apelabile în C sau sub forma unor macro-uri optimizate în assembly. În componența DSP/BIOS distingem module pentru funcții atomare în assembly pentru operațiuni low-level (ATM module), un manager de buffere cu lungime fixă (BUF module), funcții dependente de arhitectura targetului, un manager de CLK, o interfață device driver, un manager de setare globală (GBL module), un manager de I/O general (GIO module), un manager de comunicație a hostului (HST module), manager de întreruperi hardware (HWI module), un manager de mailbox, pentru comunicarea inter-thread (MBX module), un manager de segmente de memorie, un manager de funcții periodice, un manager pentru setarea transferului de date real-time (RTDX module), un manager de stream-uri I/O (SIO module), un manager de întreruperi software (SWI module), un manager de servicii ale sistemului, un manager de multitasking (TSK module), precum și alte module cu funcționalități diverse.

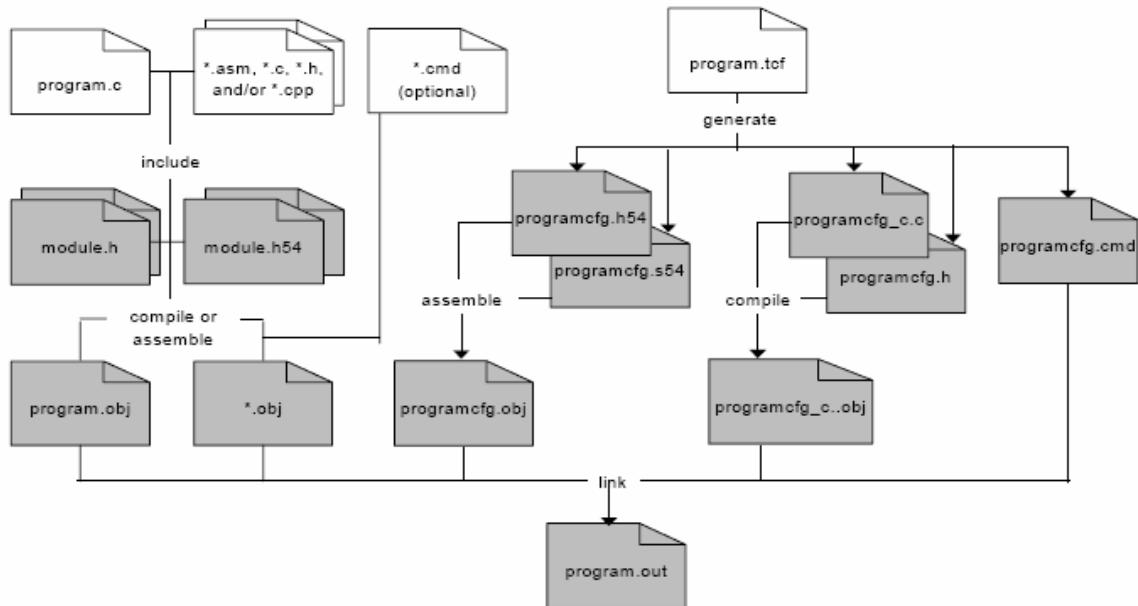
Instrumentul de configurare a DSP/BIOS oferă posibilitatea optimizării aplicației dezvoltate prin crearea și stabilirea proprietăților obiectelor în mod static și nu la momentul execuției, îmbunătățindu-se astfel performanțele de execuție și consumul de resurse al aplicației. Există metode prin care un număr relativ mare de parametri pot fi setați de către libraria real-time a kernelului în momentul execuției, obiectele create fiind utilizate în apelurile API ale DSP/BIOS și incluzând întreruperi software, task-uri, stream-uri I/O și jurnale de evenimente (even logs) toate acestea compactate într-un program executabil.

Instrumentul de analiză a kernelului DSP/BIOS complementează caracteristicile mediului de programare CodeComposerStudio oferind capacitați avansate de analiză a execuției aplicației, în mod independent și fără a afecta performanțele de execuție în real-time a aplicației. Ca metode de analiză a execuției aplicației se pot aminti Program Tracing, prin care se analizează fluxul dinamic de control în timpul execuției prin consultarea logurilor de evenimente; Performance Monitoring, prin care se urmărește analiza utilizării resurselor target-ului, cum ar fi încărcarea procesorului și temporizările și File Streaming prin care se realizează legarea obiectelor I/O de fișierele host rezidente.

În continuare este descris succint procesul de generare a programelor cu DSP/BIOS, urmărind ce fișiere sunt generate de către componentele kernel-ului și cum sunt ele utilizate. Vor fi descrise aspecte cu privire la ciclul de dezvoltare a aplicației; aspecte privind configurarea statică a aplicațiilor DSP/BIOS; crearea dinamică a obiectelor DSP/BIOS; modul de creare și fișierele utilizate la crearea programelor,

precum și aspectele privind compilarea și link-editarea lor; se va prezenta deasemenea pe scurt și utilizarea DSP/BIOS cu RTSL și apelul funcțiilor utilizator de către kernel și apelul API-ului în aplicațiile dezvoltate. DSP/BIOS-ul suportă cicluri de dezvoltare iterative pentru programe, având posibilitatea de a crea framework-ul de bază pentru o aplicație și simularea acestuia înainte de implementarea efectivă a algoritmilor pentru DSP. Utilizatorul poate schimba cu ușurință prioritățile thread-urilor și tipurile de thread-uri program pentru diverse funcții. Aspectul de interes maxim în dezvoltare pe lângă configurarea statică și crearea dinamică a obiectelor, discutate anterior, se referă la diagrama funcțională a modului de alcătuire a unei aplicații sub DSP/BIOS.

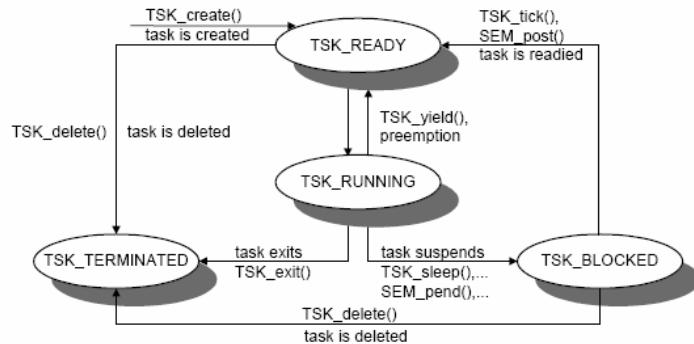
Fișierele program sunt descrise în continuare. Astfel program.c este sursa care conține funcția main, care mai poate conține și alte fișiere .c sau.h adiționale și fișiere .asm optionale. Fișierul module.h este fișierul header pentru API-ul DSP/BIOS-ului pentru programe C/C++, iar module.h54 este fișierul header API DSP/BIOS pentru programele în assembly. Fișierul obiect obținut în urma compilării sau asamblării fișierelor sursă este program.obj, care poate fi însoțit și de alte fișiere obiect. Un alt element necesar în crearea unei aplicații este fișierul (fișierele optional/e) de comenzi pentru linker care pot conține secțiuni adiționale pentru program care nu sunt definite de către configurația DSP/BIOS-ului.



După cum am precizat anterior kernelul DSP/BIOS oferă mijloace implicite dar și explicate de a realiza analiza în real-time a aplicațiilor dezvoltate. Aceste mecanisme au fost proiectate pentru a avea un impact minimal asupra performanțelor real-time ale aplicației în execuție. Analiza real-time este analiza datelor achiziționate de la sistem în timpul operării în timp real, în intenția de a determina dacă sistemul operează în limitele constrângerilor impuse, dacă împlineste obiectivele de performanță impuse și dacă este flexibil și extensibil pentru dezvoltări ulterioare. În acest context se pune problema unui debugging real-time, eficient în timpul execuției aplicației contrar specificului debugging-ului ciclic de la aplicațiile obișnuite care nu sunt supuse unor constrângerii temporale, hardware sau software. Astfel analiza se bazează pe codul de instrumentație implementat de kernel și care conține mecanisme cum sunt managerul de evenimente, preluate din logurile de execuție, managerul de statistică a obiectelor (conținând informații despre SWI, HWI și TSK) și managerul canalului de I/O gazdă. De interes în dezvoltarea de aplicații este studiul tipurilor de thread-uri suportate de kerneleul DSP/BIOS și comportamentul lor în funcție de priorități în timpul execuției programului. Cele mai multe aplicații real-time destinate DSP trebuie să efectueze un număr de funcții distincte (threaduri) în același timp, ca răspuns la o serie de evenimente exterioare cum ar fi disponibilitatea datelor sau prezentă unui semnal de control. Kernelul DSP/BIOS oferă posibilitatea de a structura aplicațiile sub forma unei colecții de threaduri fiecare implementând o funcție modularizată. Programul multithread rulează pe un singur procesor permitând thread-urilor cu

prioritate mai mare să suspende threadurile cu prioritate mai scăzută (preemptivitate) oferind diferite mijloace de interacțiune între thread-uri, blocaje, comunicare și sincronizare. DSP/BIOS oferă suport pentru câteva tipuri de thread-uri cu priorități diferite, fiecare având caracteristici de execuție și preemptive distincte, cum ar fi (în ordinea descrescătoare a priorităților): întreruperile hardware (HWI), ce includ funcții CLK (specifice întreruperilor timer-ului încorporat), întreruperi software (SWI), ce includ funcții PRD(periodice, multiplu de întreruperi ale timer-ului), taskuri (TSK) și thread-ul de background(IDL – idle thread). Întreruperile hardware sunt declanșate ca urmare a apariției unor evenimente externe asincrone mediului DSP. O funcție HWI (numită adesea ISR) este executată după declanșarea unei întreruperi hardware pentru a efectua un task critic care este supus unui deadline. Întreruperile software sunt declanșate, spre deosebire de cele hardware, prin apelul în interiorul programului a funcțiilor SWI, neavând aceleași constrângeri critice ca și cele hardware. Task-urile au o prioritate mai mică decât a întreruperilor dar mai mare decât a thread-ului de background, ele având posibilitatea de a-și amâna execuția până când resursele necesare sunt disponibile, kernelul DSP/BIOS oferind structuri specifice care sunt utilizate pentru comunicarea și sincronizarea inter task-uri, cozi, semafoare și mailbox-uri. Thread-ul de background execută bucla de idle și are cea mai scăzută prioritate și care după revenirea din main determină apelul de start-up al fiecarui modul al kernelului și intrarea în buclă, apelându-se funcții specifice pentru obiectele IDL(obiectele specifice modulului idle al kernelului). Întrucât întreruperile sunt un aspect important în proiectarea aplicațiilor propuse, dar și în general în dezvoltare, se impune o descriere a întreruperilor hardware și software și a modului specific de implementare a managerelor de întreruperi din arhitectura kernelului DSP/BIOS. Întreruperile hardware asigură execuția procesărilor critice pe care o aplicație trebuie să le execute, ținând cont de anumite constrângeri, la apariția unor evenimente externe asincrone. Modulul HWI al kernelului DSP/BIOS se ocupă cu managementul întreruperilor hardware. Ca implementare întreruperile hardware pot fi scrise fie în C, fie în assembly fie combinând cele două limbaje. Funcțiile HWI sunt de obicei scrise în assembly, pentru eficiență, pentru scrierea acestora în C fiind necesară intervenția dispecerului HWI. În ceea ce privește configurarea întreruperilor managerul modulului HWI conține un obiect HWI pentru fiecare întrerupere hardware a DSP-ului. Funcțiile specifice modulului se referă la activarea sau dezactivarea întreruperilor, la conectarea dispecerului HWI, la operațiunile de intrare și ieșire din ISR hardware, precum și o funcție de restaurare a stării întreruperilor hardware. Un alt aspect legat de tratarea întreruperilor hardware se referă la managementul contextului și întreruperii între două întreruperi, suportul fiind asigurat de dispecerul HWI și prin execuția unor operații la nivel de sistem, apelul planificatoarelor pentru managerale modulelor SWI și TSK să se efectueze la momentul potrivit și dezactivarea întreruperilor individuale în timpul execuției unei ISR. Execuția întreruperilor software este controlată de managerul SWI. Atunci când o aplicație face un apel la un API specific care poate declansa sau propune o întrerupere software, managerul SWI planifică pentru execuție funcția corespunzătoare întreruperii software, utilizând obiecte SWI. O rutină SWI poate fi suspendată preemptiv de o întrerupere HWI și poate suspenda preemptiv task-uri utilizator. Ceea ce este important și util de știut în dezvoltarea de aplicații este faptul că un handler SWI este executat până la final, în cazul în care nu este întrerupt de o întrerupere hardware sau suspendat preemptiv de un handler SWI cu prioritate mai mare. Foarte important de menționat este faptul că deseori întreruperile software pot fi apelate în interiorul unei ISR HWI, codul de declanșare a întreruperii software trebuind inclusă în funcțiile specifice HWI de intrare sau ieșire din ISR, fie sub invocarea dispecerului. Obiectele SWI pot fi create dinamic sau distruse dinamic în momentul execuției programului, prin un apel către o funcție specifică de creare/distrugere, fie static la configurare. Apelul de funcție pentru crearea unui obiect SWI poate fi efectuat doar de la nivelul de task și nu de către un obiect HWI sau alt obiect SWI. Un aspect foarte important și care are un rol important în dezvoltarea de aplicații este utilizarea mailbox-ului unui obiect SWI, cu implicație directă la implementarea funcțiilor de procesare a bufferelor de date primite pe line-in la prelucrarea primară audio. Astfel fiecare obiect SWI are un mailbox de 32 de biți care este utilizat fie pentru a determina declanșarea întreruperii software sau pentru a constitui valori pentru evaluarea în funcția SWI.

Task-urile sunt obiecte ale kernelului DSP/BIOS care sunt supervizate de modulul TSK, având prioritate mai mică decât întreruperile și mai mare decât a task-ului de background. Modulul TSK planifică dinamic task-urile și realizează preempția acestora bazându-se pe priorități și starea curentă a execuției. Obiectele TSK pot fi create fie dinamic prin funcții specifice fie static. În continuare sunt redate variațiile în modul de execuție a task-urilor.



Ultimul aspect descris în acest context este thread-ul de background (idle loop) care rulează continuu când nu au loc întreruperi hardware sau software sau nu se execută task-uri. Orice alt thread poate realiza suspendarea preemptivă a buclei idle în orice moment. Managerul IDL permite utilizatorului să insereze funcții care să fie executate pe parcursul executării buclei, ca thread de background.

## Dezvoltarea unei aplicații de comanda a LED urilor utilizator de pe placă de dezvoltare

Prima aplicatie propusa se refera la comanda LEDurilor prezente pe placă de dezvoltare cu DSP. Aplicatia determina pulsatia ledurilor la frecvențe diferite utilizand mecanismele de planificare si configurare a executiei taskurilor concurente ale sistemului de operare DSP/BIOS. Codul aplicatiei este simplu si se rezuma la implementarea unor functii simple de pulsatie a ledurilor utilizand apeuri de functii din libraria BSL si un apel de initializare a placii de dezvoltare utilizand DSK6713\_init() .

Codul aplicatiei este redat in continuare:

```

//Bios_4LED      Aplicatie care determina pulsatia ledurilor la frecvențe
//diferite utilizand mecanismele de planificare si configurare a executiei
//taskurilor concurente ale sistemului de operare DSP/BIOS

#include "bios_4ledcfg.h"                                //generat de fisierul .cdb de configurare a DSP/BIOS

void blinkLED0()
{
    DSK6713_LED_toggle(0);                            //pulseaza LED0 (50ms)
}

void blinkLED1()
{
    DSK6713_LED_toggle(1);                            //pulseaza LED1 (100ms)
}

void blinkLED2()
{
    DSK6713_LED_toggle(2);                            ///pulseaza LED2 (200ms)
}

void blinkLED3()
{
    DSK6713_LED_toggle(3);                            //pulseaza LED3 (400ms)
}

void main()
{
    DSK6713_init();                                  //initializare DSK
}

```

Pentru implementarea taskurilor propuse se creeaza un fisier de configurare pentru sistemul de operare DSP/BIOS, bios\_4led.cdb, care va contine parametrizarea corespunzatoare pentru executie a taskurilor concurente, asigurand si considerand mecanisme de protectie a accesului la resurse partajate si parametri

de periodizare. Fisierul .cdb poate fi editat utilizand o interfata grafica sau direct in cod, avantajul interfetei fiind ca fiecare proprietate are si o descriere utila in parametrizare. In continuare sunt redate doua moduri de a parametriza executia nucleului DSP/BIOS, in mod grafic si cod.

```

//!
//# c6xix.cdb 4.90.270

object IRAM :: MEM {
    param iComment :: "This object defines space for the DSP's on-chip memory"
    param iIsUsed :: 1
    param iId :: 0
    param iDelUser :: "USER"
    param iDelMsg :: "ok"
    param base :: 0
    param len :: 196608
    param iAllocHeap :: 1
    param iHeapSize :: 32768
    param iUserHeapId :: 0
    param iHeapId :: @segment_name
    param iReqHeapCount :: 3
    param space :: "code/data"
    param iIsModifiable :: 1
}

object CACHE_L2 :: MEM {
    param iComment :: "Generated by Cache Settings in GBL"
    param iIsUsed :: 1
    param iId :: 0
    param iDelUser :: "MEM"
    param iDelMsg :: "L2 Cache cannot be deleted by user"
    param base :: 196608
    param iAllocHeap :: 0
    param iHeapSize :: 65536
    param iUserHeapId :: 0
    param iHeapId :: @segment_name
    param iReqHeapCount :: 0
    param space :: "Cache"
    param iIsModifiable :: 0
}

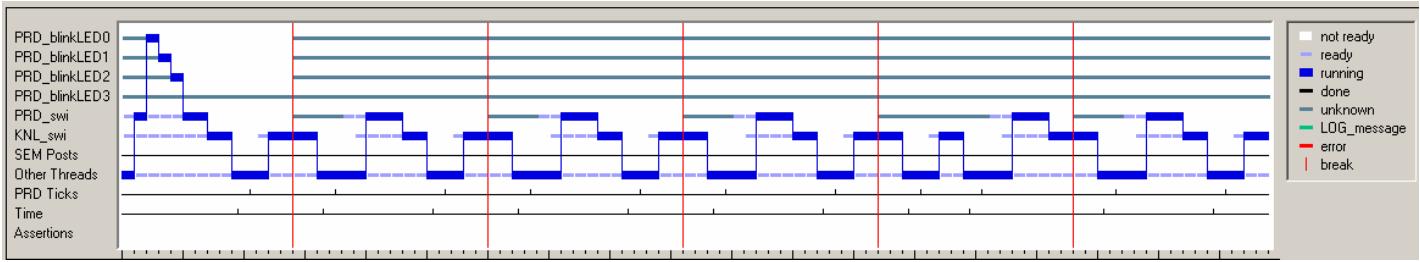
class Module {
    prop Visible :: 1
    prop Writable :: 1
    prop IsConfMod :: (@gNumOf > 0 {1} else {0})
    prop Nog :: 0
    prop IsDirty :: ($a = self.gDirty, self.gDirty = 0, $a)
    prop dataSize :: 0
    prop error :: # ("Error: ", self.name, $1)
    prop warning :: ("Warning ...", self.name, $1)
    prop minBit :: ($a = 0, while (($! & (1 << $a)) && $a < 32) (++$a), $a)
    prop name :: ("<unnamed module>")
    prop numBit :: ($a = $b = 0, while ($a < 32) {if ($! & (1 << $a)) (++$b)}, $b)
}

```

De interes pentru aplicatia dezvoltata este parametrizarea campului Scheduling intrucat aici setam functiile periodice in executie si putem seta perioada (tick sau ms), modul de periodizare (continuu/one-shot), parametrii functiei periodice si numele acestora din codul aplicatiei.

PRD_blinkLED0 properties	
Property	Value
comment	<add comments here>
period (ticks)	50
mode	continuous
function	blinkLED0
arg0	0x00000000
arg1	0x00000000
period (ms)	50.0

Pentru a analiza executia se impune utilizarea instrumentelor specific DSP/BIOS. Astfel in prima instantă se deschide DSP/BIOS-ExecutionGraph unde putem vizualiza comutarea de context intre taskuri si tranzitia taskurilor de-a lungul executiei intre starile descrise mai sus. Modulul TSK are rolul de a implementa o planificare preemptiva functie de prioritatile stabilite apriori.



Vizualizarea se poate realiza cu o fereastra de timp variabila intre 2048 – 16384 de evenimente. Se poate observa momentul in care taskurile periodice continue implementate in aplicatie realizeaza tranzitia din starea Ready in Running intrucat aceastea nu acceseaza resurse partajate si nu trebuie astfel sa astepte finalizarea altor taskuri dependente.

Un alt instrument util este DSP/BIOS-MessageLog care are rolul de a prezenta modul de comunicare interproces la nivelul kernelului sistemului de operare prin mesaje. Din mesaje se poate extrage informatie privind modulul care superviseaza acel task, adresa functiei care implementeaza taskul si starea curenta. Si in acest caz putem remarca comutarea de context.

```
Log Name: Execution Graph Details
95124 SWI: begin KNL_swi [TSK scheduler] (0x115c4)
95125 SWI: end KNL_swi [TSK scheduler] (0x115c4) state = done
95126 CLK: current time = 13518 (0x000034ce)
95127 PRD: tick count = 13518 (0x000034ce)
95128 SWI: post PRD_swi (0x11598)
95129 SWI: post KNL_swi [TSK scheduler] (0x115c4)
95130 SWI: begin PRD_swi (0x11598)
95131 PRD: end
95132 SWI: end PRD_swi (0x11598) state = done
95133 SWI: begin KNL_swi [TSK scheduler] (0x115c4)
95134 SWI: end KNL_swi [TSK scheduler] (0x115c4) state = done
95135 CLK: current time = 13519 (0x000034cf)
95136 PRD: tick count = 13519 (0x000034cf)
95137 SWI: post KNL_swi [TSK scheduler] (0x115c4)
95138 SWI: begin KNL_swi [TSK scheduler] (0x115c4)
95139 SWI: end KNL_swi [TSK scheduler] (0x115c4) state = done
102363 SWI: begin KNL_swi [TSK scheduler] (0x115c4)
102364 SWI: end KNL_swi [TSK scheduler] (0x115c4) state = done
102365 CLK: current time = 23570 (0x00005c12)
102366 PRD: tick count = 23570 (0x00005c12)
102367 SWI: post PRD_swi (0x11598)
102368 SWI: post KNL_swi [TSK scheduler] (0x115c4)
102369 SWI: begin PRD_swi (0x11598)
102370 PRD: end
102371 SWI: end PRD_swi (0x11598) state = done
102372 SWI: begin KNL_swi [TSK scheduler] (0x115c4)
102373 SWI: end KNL_swi [TSK scheduler] (0x115c4) state = done
102374 CLK: current time = 23571 (0x00005c13)
102375 PRD: tick count = 23571 (0x00005c13)
102376 SWI: post KNL_swi [TSK scheduler] (0x115c4)
102377 SWI: begin KNL_swi [TSK scheduler] (0x115c4)
102378 SWI: end KNL_swi [TSK scheduler] (0x115c4) state = done
109610 SWI: begin KNL_swi [TSK scheduler] (0x115c4)
109611 SWI: end KNL_swi [TSK scheduler] (0x115c4) state = done
109612 CLK: current time = 33148 (0x0000817c)
109613 PRD: tick count = 33148 (0x0000817c)
109614 SWI: post PRD_swi (0x11598)
```

Ultimul instrument de analiza util este DSP/BIOS-CPU Load Graph care reda incarcarea DSP de-a lungul executiei aplicatiei. In cazul de fata procesorul nu este incarcat intrucat aplicatia este foarte simpla si nu presupune calcule intensive. Astfel, procentual procesorul ajunge la o valoare maxima de 2.91% in incarcare.



## Dezvoltarea unei aplicatii de generare a unui semnal sinusoidal utilizand DSP/BIOS

In cele ce urmeaza este descrisa o aplicatie care implementeaza doua taskuri periodice care au perioade de executie diferite, utilizatorul avand posibilitatea de a oferi un input care sa determine o comutare de context in interiorul nucleului sistemului de operare DSP/BIOS. Aplicatia implementeaza o functie care genereaza o sinusoida la 50 de ms pe iesirea codecului audio AIC23, la operare in mod polling care va fi considerata primul task periodic. A doua functie determina pulsatia LED 0 si functie de starea DIP switchurilor comanda sau nu LED 3 la 200ms. Freamente din codul aplicatiei sunt redate in continuare.

```
void sinegen()
{
    if (DSK6713_DIP_get(2) == 0)           //daca switchul 2 este apasat
    {
        DSK6713_LED_on(2);                //activeaza ledul 2
        while(++sine_on<5000)             //genereaza semnal sinusoidal
        {
            //se scrie la iesire pe ambele canale semnalul amplificat
            output_sample(sine_table[loop]*gain);
            if (++loop > 7) loop = 0;
        }
        sine_on=0;
    }
    DSK6713_LED_off(2);
}

void blinkLED0()           // functie de lucru cu LED 0 functie de starea switchurilor
{
    DSK6713_LED_toggle(0);

    if (DSK6713_DIP_get(3) == 0)
        DSK6713_LED_on(3);
    else
        DSK6713_LED_off(3);
}

void main()
{
    comm_poll();
    DSK6713_LED_init();      // initializare LED DSK
    DSK6713_DIP_init();     // initializare DIP switchuri DSK
}
```

Parametrizarea taskurilor periodice din fisierul de configurare bios\_sine\_ctrl.cdb este redata in continuare impreuna cu analiza comutarilor de context intre taskuri in interiorul DSP/BIOS si a incarcarii CPU.

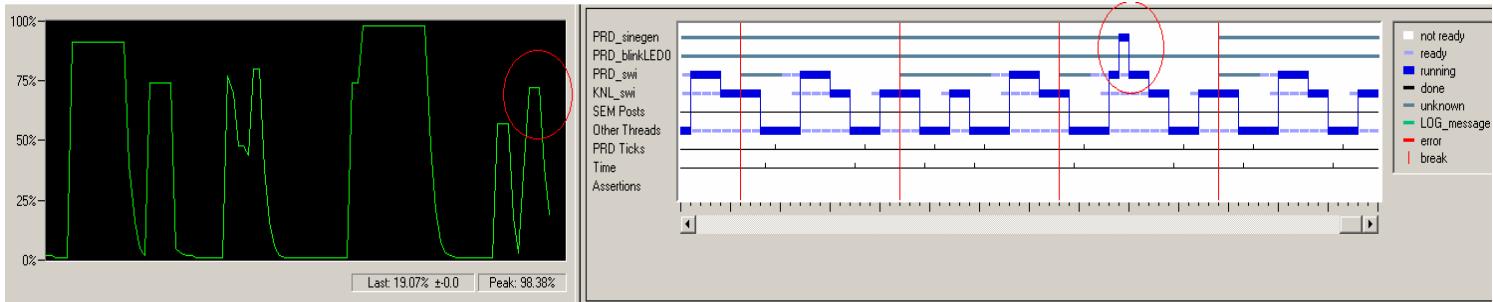
```
object PRD_sinegen :: PRD {
    param iComment :: "<add comments here>"
    param iIsUsed :: 1
    param iId :: 0
    param iDelUser :: "USER"
    param iDelMsg :: "ok"
    param period :: 50
    param mode :: "continuous"
    param function :: @_sinegen
    param arg0 :: 0
    param arg1 :: 0
    param Order :: 1
    param iPri :: 0
}

object PRD_blinkLED0 :: PRD {
    param iComment :: "<add comments here>"
    param iIsUsed :: 1
    param iId :: 0
    param iDelUser :: "USER"
    param iDelMsg :: "ok"
```

```

param period :: 200
param mode :: "continuous"
param function :: @_blinkLED0
param arg0 :: 0
param arg1 :: 0
param Order :: 2
param iPri :: 0
}

```



Se poate observa cum CPU lucrand in mod polling scrie esantioane la fiecare 50 ms (frecventa de planificare in executie (stare running) pentru taskul `sinegen()`) la iesirea codecului AIC23, acest lucru fiind marcat de incarcarea medie de aproape 80% a CPU.

#### Cerinta:

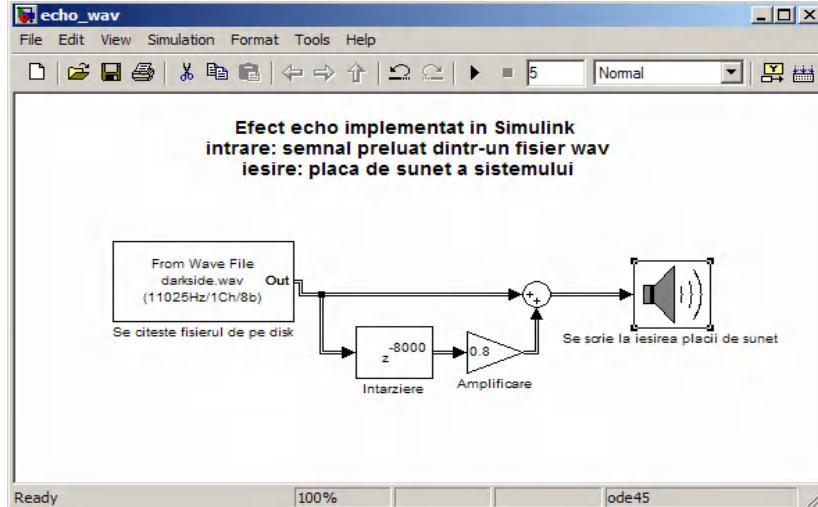
1. Sa se modifice perioada de executie a taskului de generare a sinusoidei la 2s si se adauge inca 3 taskuri periodice de comanda a LEDurilor la perioade de 20ms. Sa se observe modul de planificare al taskurilor, comutarea de context si incarcarea procesorului.

#### **Generarea efectelor de echo si reverb pe un semnal util de la microfon/sursa de semnal audio utilizand interfata MATLAB/Simulink cu TMS320C6713/ TMS320C6416**

In a doua parte a lucrarii de laborator se propune implementarea unui efect de echo utilizand MATLAB/Simulink si targetul de interfata cu Code Composer Studio si DSP TMS320C6713/TMS320C6416. In prima faza s-a implementat un model Simulink care reda structura unui modul de echo (intarziere progresiva propagata prin esantioanele de intrare). Aceast exemplu ne va fi util la implementarea interfetei utilizand DSP si Simulink.

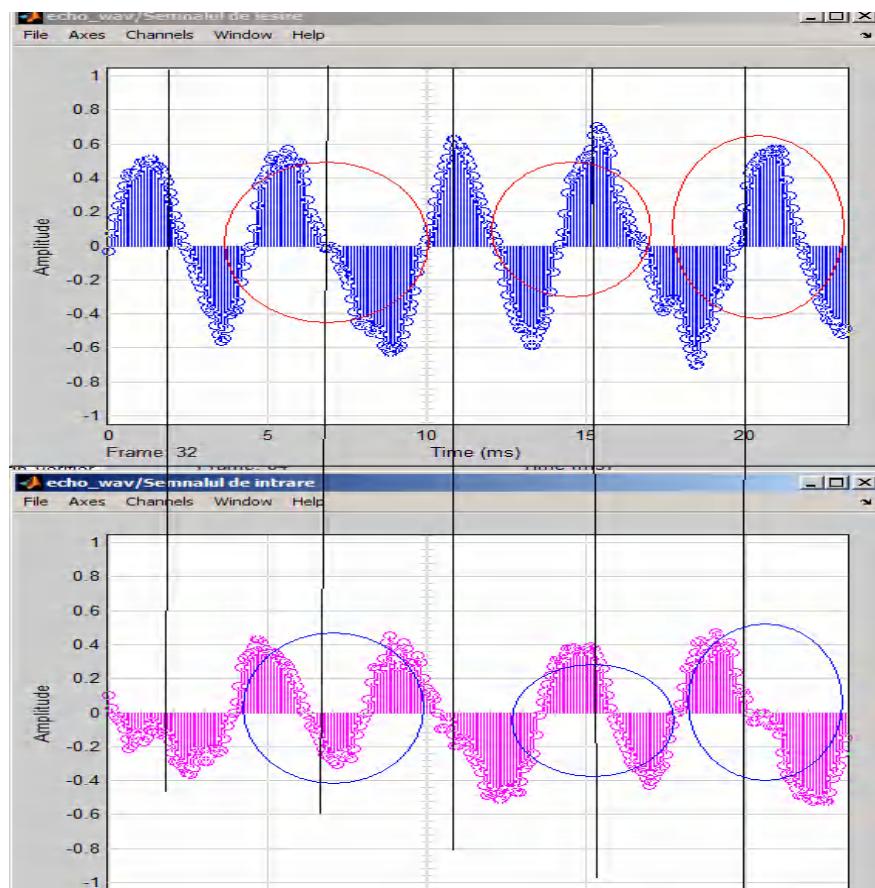
In continuare este redata o scurta analiza a fenomenului de echo. Astfel atunci cand o unda sonora atinge extremitatea mediului de propagare are un comportament aparte. Prin extremitatea mediului se poate intelege un perete sau o alta suprafață care ar putea determina reflexia unei sonore. Aceasta reflexie este deseori numita echo. S-a demonstrat ca daca reflexia are loc la aproximativ 17m de sursa de semnal, semnalul sonor va ajunge în mai mult de 0.1s în apoi la sursa și cum perceptia sunetului ramane în memoria umană aproximativ 0.1s va exista o intarziere între perceptia sunetului original și a sunetului reflectat.

Modelul Simulink pentru implementarea echoului este redat în continuare împreună cu valorile de parametrizare.

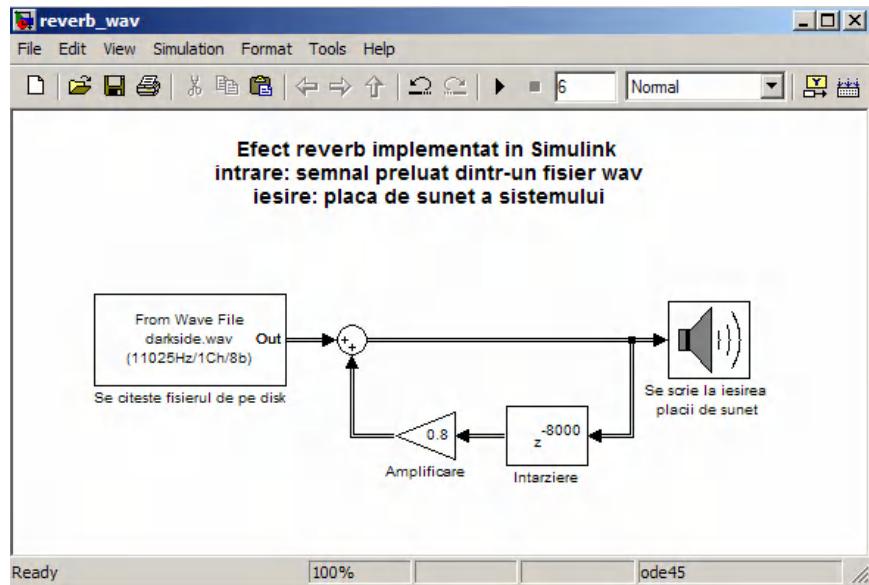


Dupa cum se poate observa efectul este implementat sub forma unei intarzieri a esantioanelor citite din fisierul de pe disk, de fapt o sumare intre semnalul util si semnalul intarziat cu n esantioane si amplificat de m ori. Pentru blocul de intrare (fisierul wav de pe disk) se citesc date formataate double, cu un numar de 256 de esantioane / cadru de iesire si 256 de esantioane citite minim la fiecare acces la fisierul wav. Blocul de intarzire determina o intarzire a semnalului util de 8000 de esantioane, semnalul intarziat fiind apoi amplificat de 0.8 ori. In ceea ce priveste iesirea, semnalul modificat este scris la iesirea line-out a placii de sunet a sistemului host cu o durata a cozii de asteptare pentru redare real-time egala cu 2 si o intarzire initiala de 0.1s.

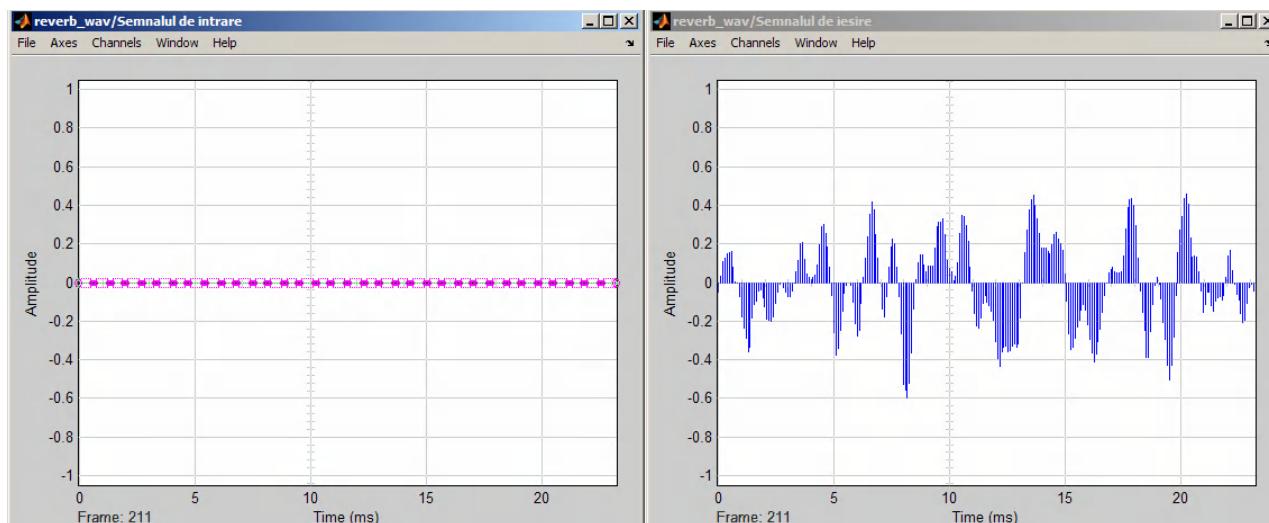
Se poate observa in urmatoarea reprezentare steme a semnalelor cum iesirea este intarziata in timp.



Pornind de la aceleasi premise ca si la efectul de echo, daca unda sonora atinge o suprafata si se reflecta in mai putin de 17m va reveni la sursa in mai putin de 0.1s. Din moment ce semnalul original este inca in memorie, deci nu apar intarzieri, cele doua semnale se vor combina intr-o unda cu o durata prelungita. Pentru implementarea efectului de reverb se utilizeaza structura urmatoare.



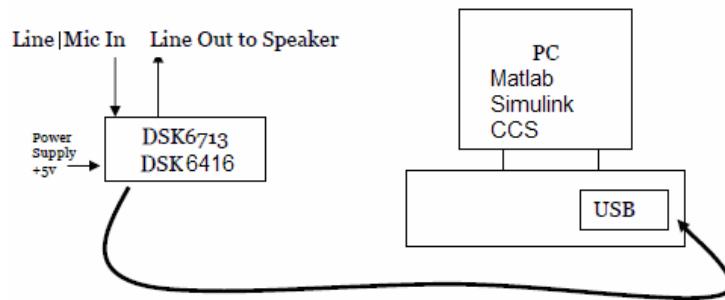
Modulul de implementare al efectului de reverb se parametrizeaza asemanator cu efectul de echo. Un aspect interesant al efectului de reverb, se refera la combinarea semnalului util cu cel intarziat pentru a prelungi propagarea semnalului original. Acest lucru se poate observa in continuare cand la finalul redarii din fisierul sursa semnalul de intrare este nul insa semnalul de iesire este inca activ.



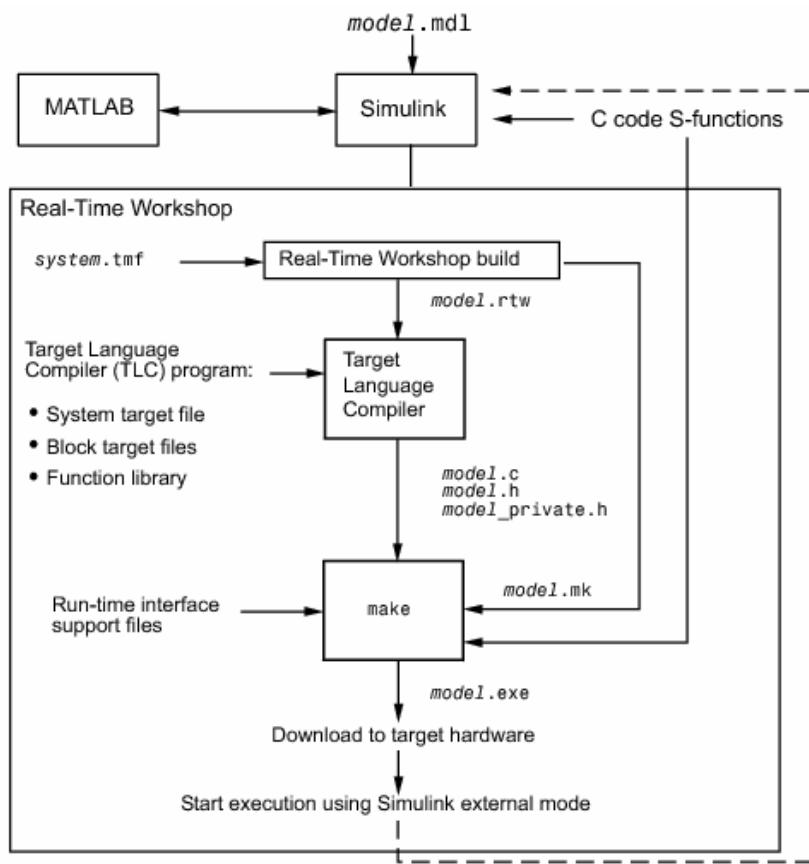
### Cerinte:

1. Sa se parametrizeze efectul de echo astfel incat intarzierea sa fie de 2 ori mai mica si amplitudinea efectului de 10 ori mai mare. Analizati rezultatele.
2. Sa se parametrizeze efectul de echo astfel incat intarzierea sa fie de 2 ori mai mare si amplitudinea efectului de 8 ori mai mare. Analizati rezultatele.

In cele ce urmeaza se propune interfatarea cu platforma DSP pentru generarea efectelor implementate in Simulink. In figura urmatoare este redata modalitatea de conectare a platformei cu DSP la calculatorul ce ruleaza MATLAB/Simulink.



Se conecteaza platforma cu DSP si in modelul simulink se acceseaza meniul Tools→Real-Time Workshop→Build Model, pentru a realiza legatura dintre Simulink si platforma. Legatura este implementata sub forma unui plugin care acceseaza un link low-level care permite accesul la functionalitatea de nivel scazut exportata de driverul placii. Instrumentul Real-Time Workshop (RTW) permite interfatarea cu o gama larga de platforme embedded cu DSP sau MCU. Arhitectura instrumentului Real-Time Workshop (RTW) este redata in continuare.



RTW genereaza cod ANSI C sau C++ optimizat, portabil si customizabil din modelul Simulink pentru a crea implementari de sine statatoare ale modelelor Simulink si care pot opera in timp real sau nu. Dupa ce codul C a fost generat de RTW acesta este preluat de Targetul pentru TI6000, un instrument ce permite generarea de cod executabil pe platformele cu DSP de la Texas Instruments din familia C6000, transformand codul de limbaj de nivel superior in cod masina pentru procesorul utilizat.

Logica de conectare a platformei cu DSP si Simulink este data de instrumentul Link for Code Composer Studio al MATLAB, de fapt un plugin care asigura legatura intre functiile de nivel superior din MATLAB

si functii de nivel scazut de acces la memoria si registrii platfromei cu DSP. Arhitectura Linkului Code Composer Studio are 2 componente (Automation Interface si Project Generator) si se bazeaza pe obiecte cu un comportament asemanator paradigmiei OOP. Automation Interface (Interfata de automatizare) este o colectie de metode ce utilizeaza API-ul Code Composer Studio pentru a asigura comunicarea intre MATLAB si Code Composer Studio utila in debugging, profiling si configurarea proiectelor. Project Generator (generatorul de proiecte) este o colectie de metode care utilizeaza functii API ale Code Composer Studio pentru a crea proiecte si pentru a genera cod pentru RTW.

In cele ce urmeaza este redat analiza executiei aplicatiei dupa construirea modelului Simulink. Dupa ce se converteste modelul in cod, acesta este integrat automat intr-un proiect un CCS si apoi compilat cu compilatorul DSP care va genera cod nativ executabil pe platforma.

Outputul in timpul conversiei din model Simulink in cod (proiect) CCS este redat in continuare din linia de comanda a MATLAB:

```
### Connecting to Code Composer Studio(tm) ...
### Generating code into build directory: C:\Documents and Settings\Cristian Axenie\My Documents\MATLAB\echo_ccslink
### Invoking Target Language Compiler on echo.rtw
### Loading TLC function libraries

.....
### Initial pass through model to cache user defined code
.
### Caching model source code
.....
### Writing header file echo_types.h
### Writing header file echo.h
.
### Writing header file MW_c6xxx_csl.h
### Writing source file MW_c6xxx_csl.c
### Writing header file MW_c6xxx_bsl.h
.
### Writing source file MW_c6xxx_bsl.c
### Writing source file echo.c
### Writing header file echo_private.h
.
### Writing source file echo_data.c
### Writing source file echo_main.c
### Writing header file echo_main.h
.
### TLC code generation complete.
### Creating project ...
### Building Code Composer Studio(tm) project...
### Build complete
### Downloading COFF file
### Downloaded: echo.out
```

Dupa generarea codului se porneste automat o instanta de CCS si se compileaza proiectul generat dupa care se downloadeaza fisierul executabil echo.out in memoria program a DSP si se porneste executia programului. Se conecteaza pe intrarea line-in semnalul audio de la o sursa de semnal/microfon si la iesirea line-out se poate sesiza semnalul procesat si LED 0 aprins daca se apasa DIP switchul 0.

### Cerinte :

1. Sa se creeze un model Simulink pentru implementarea efectului de reverb utilizand RTW si configurarile de la aplicatie de implementare a echoului. Se va utiliza ca modul de generare a efectului modelul de la primul punct, cand intrarea era preluata dintr-un fisier de pe disk si scrisa la iesirea placii de sunet a sistemului.
2. Sa se parametrizeze cele doua modele create astfel incat amplitudinea efectelor sa fie dublata si intarzierea triplata. Cele doua modele vor fi integrate in acelasi model si comutarea se va face manual printr-un switch din model.

## **Observatii:**

1. Intrucat Link for Code Composer Studio din MATLAB necesita Code Composer Studio v.3.3 se impune instalarea noii versiuni si efectuarea de mici modificari la nivelul driverelor inregistrate in mediul IDE. Astfel a fost creat un script (ccs\_fix\_matlab\_simulink\_link.bat) care sa fie rulat dupa instalarea versiunii Code Composer Studio v.3.3 si care realizeaza toate modificarile necesare pentru a activa Link for Code Composer Studio in Simulink.

### **ccs\_fix\_matlab\_simulink\_link.bat**

```
echo Prepare Code Composer Studio v3.3 for Linking with Matlab/Simulink R2007b
echo off
C:
IF EXIST C:\CCStudio_v3.1\DosRun.bat GOTO PREP
echo Change already applied !
goto END
:PREP
move C:\CCStudio_v3.1 C:\CCStudio_v3.1_OLD_VER
echo The Code Composer Studio v.3.1 link to Matlab was disabled !
echo Updating drivers in Code Composer Studio v.3.3 ...
copy C:\CCStudio_v3.1_OLD_VER\drivers\6713DSKDiag.exe C:\CCStudio_v3.3\drivers
copy C:\CCStudio_v3.1_OLD_VER\drivers\6416DSKDiag.exe C:\CCStudio_v3.3\drivers
copy C:\CCStudio_v3.1_OLD_VER\drivers\c6416dsk_11.exe C:\CCStudio_v3.3\drivers
copy C:\CCStudio_v3.1_OLD_VER\drivers\c6713dsk.exe C:\CCStudio_v3.3\drivers
copy C:\CCStudio_v3.1_OLD_VER\drivers\sdgo6416dsk_11.drv C:\CCStudio_v3.3\drivers
copy C:\CCStudio_v3.1_OLD_VER\drivers\sdgo6713dsk.drv C:\CCStudio_v3.3\drivers
copy C:\CCStudio_v3.1_OLD_VER\drivers\sdgo6416dsk_11.xml C:\CCStudio_v3.3\drivers
copy C:\CCStudio_v3.1_OLD_VER\drivers\sdgo6713dsk.xml C:\CCStudio_v3.3\drivers
copy C:\CCStudio_v3.1_OLD_VER\drivers\import\dsk6416.ccs C:\CCStudio_v3.3\drivers\import
copy C:\CCStudio_v3.1_OLD_VER\drivers\import\dsk6713.ccs C:\CCStudio_v3.3\drivers\import
echo Code Composer Studio Updated !
echo You can now use the Code Composer Studio v3.3 Link in Matlab/Simulink R2007b
:END
```

2. Pentru adaugarea Targetului TI6000 specific platformei de lucru in laborator (C6713/C6416) se acceseaza meniul Simulink→TI Target C6000→C6000 Target Preferences.

## **Observatii privind implementarea aplicatiilor de laborator pe platforma cu DSP TMS320C6416.**

Datorita faptului ca apar anumite diferente arhitecturale intre platforma cu TMS320C6713 si TMS320C6416 am ales prezentareaa numitor detalii de proiectare si de configurare a proiectelor pentru platforma TMS320C6416 intrucat aplicatiile din laborator au fost prezентate in mare parte pentru prima platforma. Desi exista doua templateuri dezvoltate pentru fiecare din cele doua platforme in continuare sunt descrise elementele specifice platformei cu TMS320C6416 utile in dezvoltarea aplicatiilor ulterioare.

Prima indicatie se refera la fisierul de configurare a proiectului, care ar trebui sa aiba urmatoarele optiuni activate.

```
["Compiler" Settings: "Debug"]
Options=-g -fr"$(Proj_dir)\Debug" -i"." -i"$(Install_dir)\c6000\dsk6416\include" -d"_DEBUG" -d"CHIP_6416" -mv6400 --mem_model:data=far

["Compiler" Settings: "Release"]
Options=-O3 -fr"$(Proj_dir)\Release" -mv6400

["Linker" Settings: "Debug"]
Options=-q -c -m".\Debug\templateC6416.map" -o".\Debug\templateC6416.out" -x -i"$(Install_dir)\c6000\dsk6416\lib" -l"dsk6416bsl.lib" -l"csl6416.lib" -l"rts6400.lib"
```

```
["Linker" Settings: "Release"]
Options==c -m".\Release\templateC6416.map" -o".\Release\templateC6416.out" -w -x
```

Al doilea aspect se refera la modificarea configuratiei canalului de date pentru McBSP0 care controleaza (trimite cuvinte de control catre codecul AIC23). Astfel in fisierul ce implementeaza functiile de interfata cu codecul (`codec_interface.h`) se adauga noi variabile de setare pentru McBSP dupa cum urmeaza.

```
MCBSP_MCR_DEFAULT,           // setare pt multi channel control register
MCBSP_RCERE0_DEFAULT,        //setare receiver channel enable register0
MCBSP_RCERE1_DEFAULT,        // setare receiver channel enable register1
MCBSP_RCERE2_DEFAULT,        // setare receiver channel enable register2
MCBSP_RCERE3_DEFAULT,        // setare receiver channel enable register3
MCBSP_XCERE0_DEFAULT,        // setare transmitter channel enable register0
MCBSP_XCERE1_DEFAULT,        // setare transmitter channel enable register1
MCBSP_XCERE2_DEFAULT,        // setare transmitter channel enable register2
MCBSP_XCERE3_DEFAULT,        // setare transmitter channel enable register3
```