

# **Limbajul de asamblare al microprocesorului Rabbit 3000. Setul de instrucțiuni**

## **1. Scopul lucrării**

Scopul acestei lucrări de laborator este familiarizarea studenților cu setul de instrucțiuni al limbajului de asamblare al microprocesorului Rabbit 3000, precum și realizarea și verificarea primelor programe scrise în limbaj de asamblare.

## **2. Introducere în limbajul de asamblare Rabbit**

Limbajul de asamblare a fost mult timp unul dintre favoriții programatorilor pentru mai multe motive. Privind în urmă, unele dintre primele computere puteau fi programate doar în limbaj de asamblare. Chiar și unele mașini industriale care erau considerate puternice pentru timpul lor foloseau tehnici de programare primitive conform standardelor de azi. De exemplu în 1974, Raytheon RDS 500 a fost proiectat inițial să urmărească traiectoriile rachetelor, dar trebuia programat instrucțiune cu instrucțiune, utilizând comutatoare de bit pe panoul frontal. Imediat după ce computerele personale au început să fie populare, au început să apară compilatoare pentru limbajele de nivel înalt și programatorii puteau programa aceste mașini în Basic, C, Pascal, Fortran, etc.

Programatorii au descoperit că anumite lucruri sunt cel mai bine făcute folosind limbaj de asamblare. În unele cazuri, limbajele de nivel înalt nu ofereau programatorilor nivelul de control dorit, în timp ce în alte cazuri ținând cont de viteza ceasului în acele vremuri anumite lucruri se produceau prea lent în comparație cu cele făcute în limbaj de asamblare. Chiar și limbaje de nivel înalt precum C și limbaje „simple” precum Basic permiteau programatorilor să manipuleze biți și octeți și să introducă cod în limbaj de asamblare între declarații de nivel înalt. Mai mult, capacitatea limitată de memorie disponibilă în asemenea sisteme necesita cod eficient forțând programatorii să utilizeze limbaj de asamblare.

Chiar dacă s-au schimbat multe de atunci, programatorii ce lucrează cu microcontrolere sunt de acord că unele lucruri sunt cel mai bine făcute cu ajutorul limbajului de asamblare. Chiar dacă viteza procesoarelor s-a mărit drastic în ultimele 3 decenii majoritatea programatorilor vor fi de acord că cel mai eficient cod din punct de vedere al timpului de execuție și al utilizării memoriei este cel scris în limbaj de asamblare. Ori de câte ori avem limitări din punct de vedere a timpului programatorii se vor gândi dacă să recurgă la limbaj de asamblare. De exemplu, atunci când utilizează un microcontroler pe 8 biți pentru a implementa secvențele de conectare la un modem, programatorul se poate baza doar pe cod scris în limbaj de asamblare pentru a genera și detecta tonurile în fereastra de timp necesară.

În continuare vom vedea o descriere a setului de instrucțiuni Rabbit.

## 2.1. Introducere în setul de instrucțiuni al microprocesorului Rabbit 3000

Instrucțiunile de asamblare Rabbit sunt folosite pentru:

- Încărcarea datelor în regiștrii specifici sau în locații de memorie. Aceste instrucțiuni sunt folosite pentru a încărca date dintr-o locație de memorie sau registru sursă într-un registru (în general acumulatorul), pentru a efectua unele operații, și pentru a transfera rezultatele înapoi într-un registru sau locație de memorie.
- Schimbarea conținutului anumitor regiștrii. Atunci când operațiile afectează conținutul anumitor regiștrii și programatorul dorește să salveze conținutul acestor regiștrii, programatorul poate utiliza aceste instrucțiuni de interschimbare pentru a lăsa regiștrii originali în pace lucrând cu un set alternativ de regiștrii, iar atunci când lucrul s-a terminat, conținutul original al regiștrilor poate fi restaurat. Atunci când regiștrii sunt schimbați conținutul regiștrilor originali nu va fi afectat de către operații.
- Plasarea datelor pe stivă. Pe lângă memorarea adreselor de revenire din subrutine și a parametrilor, stiva este un loc bun pentru stocarea temporară a datelor. Sunt oferite mai multe instrucțiuni pentru a memora datele pe stivă și a le prelua înapoi într-un registru dată. Programatorul trebuie să fie atent să preia toate datele salvate pe stivă altfel putând apare o condiție de depășire a stivei.
- Efectuarea de operații aritmetice și booleene. Aproape toate operațiile aritmetice pe 8 biți implică acumulatorul, în timp ce operațiile pe 16 biți implică perechi de regiștrii de 8 biți. Procesul Rabbit are chiar și o instrucțiune de înmulțire.
- Testarea și manipularea individuală a biților. Aceste instrucțiuni sunt necesare pentru operațiile de I/O atunci când este necesar să știm dacă un bit a fost setat de o intrare externă sau când programul trebuie să pornească sau să oprească un dispozitiv extern prin manipulare unui bit de port. În plus pot fi testați biți ai regiștrilor interni sau a locațiilor de memorie. De exemplu biții din registrul de stare pot fi testați pentru a determina dacă rezultatul unei operații aritmetice sau dacă o operație aritmetică anterioară a necesitat transport sau împrumut.
- Copierea blocurilor de memorie. Unele instrucțiuni Rabbit permit programatorului copierea unor întregi blocuri de memorie folosind doar 4 instrucțiuni.
- Salturi către alte secțiuni ale codului. Programele scrise în limbaj de asamblare pot lua decizii de salt, în general prin testarea rezultatelor unor operații. Diferite instrucțiuni permit programatorilor să testeze anumite condiții logice și să ia decizii de salt în mod corespunzător.

Chiar dacă setul de instrucțiuni Rabbit derivă din setul de instrucțiuni Z80, au mai fost adăugate și alte instrucțiuni noi microprocesorului Rabbit 3000.

Instrucțiunile limbajului de asamblare și denumirile regiștrilor nu depind de folosirea literelor mari sau mici. Totuși uneori este recomandat să folosim un anumit tip de scriere pentru a face codul mai ușor de citit. De exemplu, perechea de regiștrii HL poate fi

interpretată greșit de către cititor dacă apare scrisă cu litere mici: hl. Unele fonturi nu permit diferențierea între litera l și cifra 1.

Instrucțiunile limbajului de asamblare Rabbit sunt alcătuite dintr-un opcod urmat de 0 sau mai mulți operanzi. Opcodul reprezintă instrucțiunea în timp ce operanzii sunt datele. Operanzii pot lua diferite forme. De exemplu, ei pot fi o adresă pe 16 biți sau un singur bit. Tabelul 2.1 prezintă modul de reprezentare al operanzilor și rezultatul operațiilor asupra indicatorilor (flag-urilor) procesorului. Descrierile instrucțiunilor vor folosi abrevierile operanzilor din acest tabel.

Tabelul 2.1. Operanzii folosiți în setul de instrucțiuni Rabbit

Operand	Semnificație
<b>b</b>	Selectarea bitului: 000=bit 0, 001=bit 1, 010=bit 2, 011=bit 3, 100=bit 4, 101=bit 5, 110=bit 6, 111=bit 7
<b>cc</b>	Codul de selectare a condiției: 00=NZ, 01=Z, 10=NC, 11=C
<b>d</b>	Deplasament pe 7 biți (cu semn). Exprimat în complement față de 2.
<b>dd</b>	Registrul cuvânt selectează destinația: 00=BC, 01=DE, 10=HL, 11=SP
<b>dd'</b>	Registrul cuvânt selectează alternativa: 00=BC', 01=DE', 10=HL'
<b>e</b>	Deplasament pe 8 biți (cu semn) adăugat la PC
<b>f</b>	Codul de selectare a condiției : 000=NZ (non zero), 001=Z (zero) 010=NC (non transport), 011=C (transport) 100=LZ (zero logic), 101=LO (1 logic) 110=P (semnul plus), 111=M (semnul minus)
<b>m</b>	Cel mai semnificativ octet pentru o constantă pe 16 biți
<b>mn</b>	Constantă pe 16 biți
<b>n</b>	Constantă pe 8 biți sau cel mai puțin semnificativ octet al unei constante pe 16 biți
<b>r , g</b>	Selectare registru octet: 000=B, 001=C, 010=D, 011=E, 100=H, 101=L, 111=A
<b>ss</b>	Selectare registru cuvânt (sursă): 00=BC, 01=DE, 10=HL, 11=SP
<b>v</b>	Selectare adresă de restart: 010=0x0020, 011=0x0030, 100=0x0040, 101=0x0050, 111=0x0070
<b>xx</b>	Selectare registru cuvânt: 00=BC, 01=DE, 10=IX, 11=SP
<b>yy</b>	Selectare registru cuvânt: 00=BC, 01=DE, 10=IY, 11=SP
<b>zz</b>	Selectare registru cuvânt: 00=BC, 01=DE, 10=HL, 11=AF

Instrucțiunile limbajului de asamblare Rabbit sunt împărțite în următoarele grupuri:

- Încărcare și salvare
  - Încărcare dată imediată
  - Încărcare și Salvare la adresă imediată
  - Încărcare și Salvare indexată pe 8 biți
  - Încărcare și Salvare indexată pe 16 biți
  - Transfer registru – registru
- Instrucțiuni de interschimbare
- Instrucțiuni de manipulare a stivei

- Operații aritmetice și logice
  - Operații aritmetice și logice pe 8 biți
  - Operații aritmetice și logice pe 16 biți
- Setare, Resetare și Testare la nivel de bit pe 8 biți
- Incrementare și Decrementare pe 8 biți
- Operații rapide cu acumulatorul pe 8 biți
- Deplasări și Rotații pe 8 biți
- Prefixele instrucțiunilor
- Instrucțiuni de mutare la nivel de bloc
- Instrucțiuni de control – Salturi (jump) și Apeluri (call)
- Instrucțiuni diverse

Celor familiarizați cu programarea în limbaj de asamblare, anumite grupuri le vor părea familiare. Majoritatea procesoarelor au instrucțiuni de „încărcare și salvare” și „aritmetice și logice”. Alte grupuri vor părea familiare entuziaștilor Z80: „instrucțiuni de interschimbare”. Totuși, procesorul Rabbit are un număr de instrucțiuni unice, cum ar fi cele găsite în grupurile „instrucțiuni de mutare la nivel de bloc” sau „prefixe ale instrucțiunilor”.

Fiecare descriere a instrucțiunilor va fi sub forma unui tabel cu următorul format:

Instrucțiune	Clk	A	I	S Z V C	Operație
--------------	-----	---	---	---------	----------

Coloana **Instrucțiune** va conține mnemonica instrucțiunii și formatul opcodului.

Coloana **Clk** va indica numărul de cicli mașină necesari execuției instrucțiunii.

Coloana **A** indică ce efect are asupra instrucțiunii prefixul ALTD. Tabelul următor prezintă cheile pentru coloana **A**.

Simbol	Descriere
<b>F</b>	<b>ALTD selectează flag-uri alternative</b>
<b>R</b>	<b>ALTD selectează regiștrii destinație alternativă</b>
<b>SP</b>	<b>Operația ALTD este un caz special</b>

Coloana **I** indică ce efect au prefixele IOI și IOE asupra instrucțiunilor. Tabelul următor prezintă cheile pentru coloana **I**.

Simbol	Descriere
<b>S</b>	<b>IOI și IOE afectează sursa</b>
<b>D</b>	<b>IOI și IOE afectează destinația</b>

Coloanele **S**, **Z**, **V** și **C** corespund flagurilor Semn, Zero, Depășire (Overflow) și Transport (Carry). Acestea se găsesc în registrul Flagurilor (F). Flagul Depășire mai este referit ca și flagul LV. Tabelul următor prezintă cheile pentru simbolurile utilizate în coloana flagurilor.

Simbol	Descriere
*	Flag afectat
-	Flag neafectat
0	Flagul este resetat
1	Flagul este setat
V	Este memorată depășire aritmetică
L	Este memorat rezultat logic

### 2.1.1. Încărcare dată imediată

Instrucțiunile ce aparțin acestui mod de adresare încarcă o constantă în registrul sau registrul pereche destinație. Este numit modul de adresare imediată deoarece constanta ce trebuie încărcată urmează imediat după opcodul instrucțiunii de încărcare. Tabelul 2.2 listează instrucțiunile acestui grup.

Tabel 2.2. Încărcare dată imediată

Instrucțiune	Clk	A	I	S	Z	V	C	Operație
LD IX, mn	8			-	-	-	-	IX = mn
LD IY, mn	8			-	-	-	-	IY = mn
LD dd, mn	6	r		-	-	-	-	dd = mn
LD r,n	4	r		-	-	-	-	r = n

Următoarele instrucțiuni arată cum este încărcată o dată imediată în regiștrii pe 8 și 16 biți:

```
ld a, 5           ; registrul a ia valoarea 5 (zecimal)
ld ix, 0x1234     ; registrul ix ia valoarea 1234h (hexazecimal)
```

Dacă programatorul programează o buclă care se execută întotdeauna de un număr fix de ori, un registru poate fi setat pe post de contor și poate fi inițializat prima dată prin utilizarea unei instrucțiuni de încărcare imediată. Următoarele instrucțiuni arată cum se poate face asta:

```
#define COUNTER 240           ; contor constant (zecimal)
ld BC, COUNTER               ; încarcă valoarea contorului
```

### 2.1.2. Încărcare și stocare la adresă imediată

În acest mod de adresare, unul dintre operanzi este un registru, în timp ce celălalt operand este extras din memorie. În funcție de instrucțiune, o adresă pe 16 biți este utilizată pentru a pointer datele sursă sau destinație. Acest set de instrucțiuni conține numele de

adresă imediată deoarece adresa pe 16 biți urmează imediat după opcodul instrucțiunii. Tabelul 2.3 listează instrucțiunile acestui grup.

Tabel 2.3. Încărcare și stocare la adresă imediată

Instrucțiune	Clk	A	I	S	Z	V	C	Operație
LD (mn), A	10		d	-	-	-	-	(mn) = A
LD A, (mn)	9	r	s	-	-	-	-	A = (mn)
LD (mn), HL	13		d	-	-	-	-	(mn) = L; (mn+1) = H
LD (mn), IX	15		d	-	-	-	-	(mn) = IXL; (mn+1) = IXH
LD (mn), IY	15		d	-	-	-	-	(mn) = IYL; (mn+1) = IYH
LD (mn), ss	15		d	-	-	-	-	(mn) = ssL; (mn+1) = ssH
LD HL, (mn)	11	r	s	-	-	-	-	L = (mn); H = (mn+1)
LD IX, (mn)	13		s	-	-	-	-	IXL = (mn); IXH = (mn+1)
LD IY, (mn)	13		s	-	-	-	-	IYL = (mn); IYH = (mn+1)
LD dd, (mn)	13	r	s	-	-	-	-	ddL = (mn); ddH = (mn+1)

Instrucțiunile de mai sus sunt utile pentru realizarea operațiilor simple bazate pe pointeri. Pointerul pe 16 biți poate memora sau scoate unul sau doi octeți în/din memorie (sau de la porturile de I/O, dacă acestea sunt mapate în memorie). De exemplu, dacă un program trebuie să folosească mai multe variabile, acestea pot fi memorate în locații de memorie și rezultatele operațiilor pot fi stocate în variabile.

Următoarele instrucțiuni ilustrează niște exemple pentru acest grup de instrucțiuni:

```
#define BUFFER_SIZE 64          ; spațiul alocat pentru buffer
char  bytestoread               ; numărul de octeți de citit
int   bytcounter               ; numărul de octeți ce au fost citați
ld    a, (bytestoread)          ; aflare număr octeți de citit
ld    (bytcounter), hl          ; update contor octeți
ld    ix, (sp+2+BUFFER_SIZE)    ; ix = buffer
```

### 2.1.3. Încarcă și Salvează indexat pe 8 biți

În forma lor cea mai simplă, aceste instrucțiuni sunt utile pentru realizarea operațiilor bazate pe pointeri. De exemplu, dacă un program trebuie să adune un set de valori stocate în memoria RAM, programatorul poate seta o pereche de regiștri să poarte locația de start a tabelii în RAM, să citească valorile una câte una și să le adune la registrul destinație.

Într-o aplicație mai complexă de adresare indexată, conținutul unui registru index este adunat la un deplasament pentru a calcula adresa operandului. Acest lucru este obținut de instrucțiunile „IX + d” și „IY + d” de mai jos, unde „d” este deplasamentul pe 8 biți.

Programatorii în limbaj C pot vedea modelul „index cu deplasament” ca o structură unde registrul index pointează la începutul structurii, iar deplasamentul este folosit pentru a pointa elementele din cadrul structurii. Tabelul 2.4 listează instrucțiunile indexate pe 8 biți.

Tabel 2.4. Încărcare și stocare indexată pe 8 biți

Instrucțiune	Clk	A	I	S	Z	V	C	Operație
LD A,(BC)	6	r	s	-	-	-	-	A = (BC)
LD A, (DE)	6	r	s	-	-	-	-	A = (DE)
LD (BC), A	7		d	-	-	-	-	(BC) = A
LD (DE), A	7		d	-	-	-	-	(DE) = A
LD (HL), n	7		d	-	-	-	-	(HL) = n
LD (HL), r	6		d	-	-	-	-	(HL) = r = B,C,D,E,H,L,A
LD r, (HL)	5	r	s	-	-	-	-	r = (HL)
LD (IX+d), n	11		d	-	-	-	-	(IX+d) = n
LD (IX+d), r	10		d	-	-	-	-	(IX+d) = r
LD r, (IX+d)	9	r	s	-	-	-	-	r = (IX+d)
LD (IY+d), n	11		d	-	-	-	-	(IY+d) = n
LD (IY+d), r	10		d	-	-	-	-	(IY+d) = r
LD r, (IY+d)	9	r	s	-	-	-	-	r = (IY+d)

Un exemplu simplu de adresare indexată ar fi un program care adună o serie de întregi aflați în locații consecutive din RAM. Programatorul poate seta un registru pe 16 biți pentru a pointa la începutul tabelii, iar apoi să încarce fiecare întreg în acumulator. Programul poate aduna întregii individuali la rezultatul pe 32 de biți aflat în RAM.

Următoarele instrucțiuni ilustrează încărcările și salvările indexate pe 8 biți.

```
ld          a, (iy+8)          ;
ld          (ix+8), b          ;
```

#### 2.1.4. Încărcări și salvări indexate pe 16 biți

Încărcările și salvările indexate pe 16 biți sunt asemănătoare celor pe 8 biți, cu excepția faptului că sursa sau destinația este un registru pe 16 biți sau două locații consecutive de memorie.

Deoarece aceste instrucțiuni necesită citirea adresei din memorie, adunarea unui deplasament și stocarea conținutului într-un registru pe 16 biți, au nevoie de un număr relativ mare de cicli de ceas pentru execuție. Tabelul 2.5 prezintă instrucțiunile indexate pe 16 biți.

Tabelul 2.5. Încărcări și salvări indexate pe 16 biți

Instrucțiune	Clk	A	I	S	Z	V	C	Operație
LD (HL+d), HL	13		d	-	-	-	-	(HL+d) = L; (HL+d+1) = H
LD HL, (HL+d)	11	r	s	-	-	-	-	L = (HL+d); H = (HL+d+1)
LD (SP+n), HL	11			-	-	-	-	(SP+n) = L; (SP+n+1) = H
LD (SP+n), IX	13			-	-	-	-	(SP+n) = IXL; (SP+n+1) = IXH
LD (SP+n), IY	13			-	-	-	-	(SP+n) = IYL; (SP+n+1) = IYH
LD HL, (SP+n)	9	r		-	-	-	-	L = (SP+n); H = (SP+n+1)
LD IX, (SP+n)	11			-	-	-	-	IXL = (SP+n); IXH = (SP+n+1)
LD IY, (SP+n)	11			-	-	-	-	IYL = (SP+n); IYH = (SP+n+1)
LD (IX+d), HL	11		d	-	-	-	-	(IX+d) = L; (IX+d+1) = H
LD HL, (IX+d)	9	r	s	-	-	-	-	L = (IX+d); H = (IX+d+1)
LD (IY+d), HL	13		d	-	-	-	-	(IY+d) = L; (IY+d+1) = H
LD HL, (IY+d)	11	r	s	-	-	-	-	L = (IY+d); H = (IY+d+1)

Următoarele instrucțiuni utilizează încărcări și salvări indexate pe 16 biți:

ld hl, (ix+4) ;  
ld (ix+2), hl ;

### 2.1.5. Copieri registru - registru

Acest mod de adresare mai este numit și adresare prin regiștri, deoarece toți operanzii sunt regiștri ai procesorului. Așa cum implică și numele, aceste instrucțiuni copie conținutul unui registru în altul – oricare din regiștrii pe 8 biți poate fi copiat în oricare alt registru pe 8 biți. Acest lucru este necesar deoarece majoritatea operațiilor matematice și logice sunt efectuate folosind un acumulator (registru A sau perechea de regiștri HL). Așa cum se poate observa în tabelul 2.6, anumiți regiștri pe 16 biți pot fi de asemenea copiați în alți regiștri pe 16 biți.

Deoarece copierile registru – registru se produc în interiorul procesorului și nu este necesară extragerea operanzilor di memorie, aceste instrucțiuni durează puțini cicli de ceas.



Tabelul 2.6. Copieri registru - registru

Instrucțiune	Clk	A	I	S	Z	V	C	Operație
LD r,g	2	R		-	-	-	-	$r = g$ (r, g any B,C,D,E,H,L,A)
LD A, EIR	4	fr		*	*	-	-	$A = EIR$
LD A, IIR	4	fr		*	*	-	-	$A = IIR$
LD A, XPC	4	R		-	-	-	-	$A = MMU$
LD EIR, A	4			-	-	-	-	$EIR = A$
LD IIR, A	4			-	-	-	-	$IIR = A$
LD XPC, A	4			-	-	-	-	$XPC = A$
LD HL, IX	4	R		-	-	-	-	$HL = IX$
LD HL, IY	4	R		-	-	-	-	$HL = IY$
LD IX, HL	4			-	-	-	-	$IX = HL$
LD IY, HL	4			-	-	-	-	$IY = HL$
LD SP, HL	2			-	-	-	-	$SP = HL$
LD SP, IX	4			-	-	-	-	$SP = IX$
LD SP, IY	4			-	-	-	-	$SP = IY$
LD dd <sup>l</sup> , BC	4			-	-	-	-	$dd^l = BC(dd^l, BC^l, DE^l, HL^l)$
LD dd <sup>l</sup> , DE	4			-	-	-	-	$dd^l = DE(dd^l, BC^l, DE^l, HL^l)$

De exemplu, să considerăm registrul IIR: pointează o tabelă cu vectorii de întreruperi specifici întreruperilor generate intern. Atunci când un programator dorește să utilizeze întreruperile generate intern, registrul IIR nu poate fi încărcat imediat cu o valoare – nu există un asemenea opcod. În schimb, valoarea imediată poate fi încărcată în acumulator, iar conținutul acumulatorului poate fi copiat în registrul IIR.

### 2.1.6. Instrucțiuni de interschimbare

UCP conține un set alternativ de regiștri, unde perechile de regiștri AF, HL, BC și DE au corespondenții AF', HL', BC' și DE'. În mod normal, programatorul nu folosește regiștrii alternativi, deoarece aceștia sunt utilizați de Dynamic C în scop propriu.

După cum indică și numele, instrucțiunile de „interschimbare” înlocuiesc conținutul regiștrilor pe 16 biți cu conținutul regiștrilor speciali alternativi. De exemplu, instrucțiunea „EX DE', HL” schimbă conținutul perechii de regiștri HL cu cel al perechii alternative DE'. Există două cazuri speciale:

- Instrucțiunea „EXX” realizează trei schimbări cu o singură instrucțiune: BC, DE și HL

- Instrucțiunea „EX AF, AF'” tratează acumulatorul și registrul flagurilor ca o pereche de regiștri și le înlocuiesc cu regiștri alternativi.
- Prefixul ALTD permite instrucțiunilor să acceseze direct regiștrii alternativi, fără a interschimba toti regiștrii. Tabelul 2.7 listează diferitele instrucțiuni de interschimbare.

Tabelul 2.7. Instrucțiunile de interschimbare

Instrucțiune	Clk	A	I	S	Z	V	C	Operație
EX (SP), HL	15	r		-	-	-	-	H <-> (SP+1); L <-> (SP)
EX (SP), IX	15			*	*	-	-	IXH <-> (SP+1); IXL <-> (SP)
EX (SP), IY	15			*	*	-	-	IYH <-> (SP+1); IYL <-> (SP)
EX AF, AF'	2			-	-	-	-	AF <-> AF'
EX DE', HL	2	s		-	-	-	-	if (!ALTD) then DE' <-> HL else DE' <-> HL'
EX DE', HL'	4	s		-	-	-	-	DE' <-> HL'
EX DE, HL	2	s		-	-	-	-	if (!ALTD) then DE <-> HL else DE <-> HL'
EX DE, HL'	4	s		-	-	-	-	DE <-> HL'
EXX	2			-	-	-	-	BC <-> BC'; DE <-> DE'; HL <-> HL'

Figura 2.1 ilustrează modul cum sunt interschimbate perechile de regiștri.

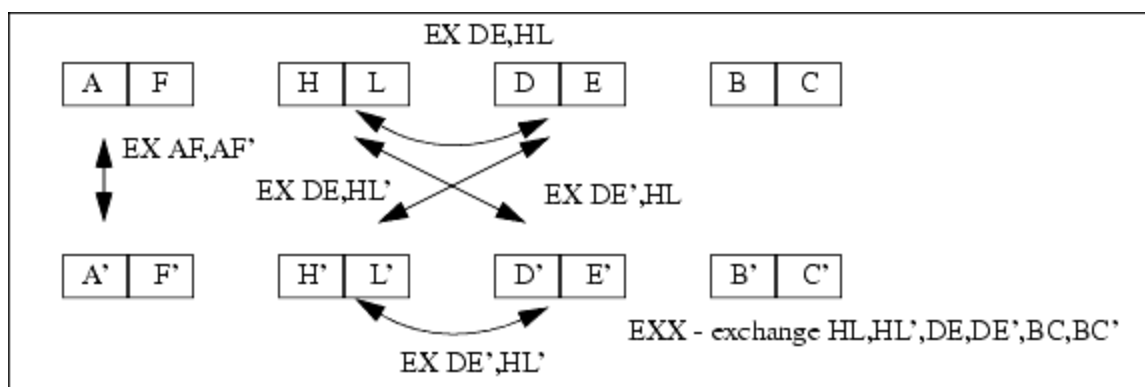


Figura 2.1. Instrucțiunile de interschimbare

### 2.1.7. Instrucțiuni de manipulare a stivei

Se presupune că cititorul știe ce este o stivă și cum funcționează ea. Tabelul 2.8 prezintă instrucțiunile de manipulare a stivei. Prima instrucțiune din acest grup (ADD SP, d) adună la stivă un deplasament pe un octet. Acesta este un mod rapid pentru programator de a muta pointerul stivei înainte cu până la 255 de octeți, fără a modifica conținutul stivei. Deplasamentul este întotdeauna pozitiv. Principalul motiv pentru această instrucțiune ar fi rezervarea unui bloc de memorie pe stivă.

Celelalte instrucțiuni din acest grup fie salvează pe stivă regiștri pe 16 biți, fie îi încarcă de pe stivă.

Aceste instrucțiuni permit programatorului să salveze (push) date și variabile pe stivă, iar apoi să le citească înapoi (pop) când este necesar. Trebuie să avem grijă atunci când se lucrează cu stiva și cu subrutine și întreruperi, deoarece acestea folosesc stiva pentru a reține adresele de revenire. De exemplu, dacă programul principal trebuie să transmită date unei subrutine prin intermediul stivei, programul apelant poate salva datele pe stivă. Atunci când subrutina este apelată, nu poate citi datele folosind pointerul la stivă curent, deoarece apelul subrutinei a alterat conținutul stivei și al pointerului la stivă – pointerul la stivă trebuie ajustat pentru a putea citi datele relevante corect, iar apoi trebuie ajustat din nou pentru a pointera adresa de revenire din programul apelant.

În Dynamic C este responsabilitatea rutinei apelante să restaureze pointerul stivei, odată ce rutina apelată s-a terminat. Rutina apelată accesează valorile transmise folosind adresare indexată relativă la SP.

Tabelul 2.8. Instrucțiunile de manipulare a stivei

Instrucțiune	Clk	A	I	S	Z	V	C	Operație
ADD SP,d	4	f		-	-	-	*	SP = SP + d -- d=0 to 255
POP IP	7			-	-	-	-	IP = (SP); SP = SP+1
POP IX	9			-	-	-	-	IXL = (SP); IXL = (SP+1); SP = SP+2
POP IY	9			-	-	-	-	IYL = (SP); IYL = (SP+1); SP = SP+2
POP zz	7	r		-	-	-	-	zzL = (SP); zzH = (SP+1); SP=SP+2 -- zz = BC,DE,HL,AF
PUSH IP	9			-	-	-	-	(SP-1) = IP; SP = SP-1
PUSH IX	12			-	-	-	-	(SP-1) = IXL; (SP-2) = IXL; SP = SP-2
PUSH IY	12			-	-	-	-	(SP-1) = IYL; (SP-2) = IYL; SP = SP-2
PUSH zz	10			-	-	-	-	(SP-1) = zzH; (SP-2) = zzL; SP=SP-2 -- zz = BC,DE,HL,AF

#### 2.1.8. Operații aritmetice și logice pe 8 biți

Microprocesorul Rabbit efectuează operațiile aritmetice și logice tipice întâlnite și la alte procesoare pe 8 biți: adunări, scăderi, complementări, AND, OR, XOR. Nu se regăsesc instrucțiuni de înmulțire sau împărțire în acest grup, chiar dacă grupul de instrucțiuni pe 16 biți permite înmulțiri ce implică regiștri pe 16 biți.

Există și instrucțiuni speciale de adunare și scădere, care țin cont de bitul de transport.

Aceste instrucțiuni folosesc acumulatorul și un al doilea operand ce poate fi o valoare imediată, un alt registru pe 8 biți sau o locație de memorie pointată de regiștri HL, IX sau IY. Tabelul 2.9 prezintă operațiile aritmetice și logice pe 8 biți.

Tabelul 2.9. Operațiile aritmetice și logice pe 8 biți

Instrucțiune	Clk	A	I	S	Z	V	C	Operație
ADC A,(HL)	5	fr	s	*	*	V	*	$A = A + (HL) + CF$
ADC A,(IX+d)	9	fr	s	*	*	V	*	$A = A + (IX+d) + CF$
ADC A,(IY+d)	9	fr	s	*	*	V	*	$A = A + (IY+d) + CF$
ADC A,n	4	fr		*	*	V	*	$A = A + n + CF$
ADC A,r	2	fr		*	*	V	*	$A = A + r + CF$
ADD A,(HL)	5	fr	s	*	*	V	*	$A = A + (HL)$
ADD A,(IX+d)	9	fr	s	*	*	V	*	$A = A + (IX+d)$
ADD A,(IY+d)	9	fr	s	*	*	V	*	$A = A + (IY+d)$
ADD A,n	4	fr		*	*	V	*	$A = A + n$
ADD A,r	2	fr		*	*	V	*	$A = A + r$
AND (HL)	5	fr	s	*	*	L	0	$A = A \& (HL)$
AND (IX+d)	9	fr	s	*	*	L	0	$A = A \& (IX+d)$
AND (IY+d)	9	fr	s	*	*	L	0	$A = A \& (IY+d)$
AND n	4	fr		*	*	L	0	$A = A \& n$
AND r	2	fr		*	*	L	0	$A = A \& r$
CP (HL)	5	f	s	*	*	V	*	$A - (HL)$
CP (IX+d)	9	f	s	*	*	V	*	$A - (IX+d)$
CP (IY+d)	9	f	s	*	*	V	*	$A - (IY+d)$
CP n	4	f		*	*	V	*	$A - n$
CP r	2	f		*	*	V	*	$A - r$
OR (HL)	5	fr	s	*	*	L	0	$A = A   (HL)$
OR (IX+d)	9	fr	s	*	*	L	0	$A = A   (IX+d)$
OR (IY+d)	9	fr	s	*	*	L	0	$A = A   (IY+d)$
OR n	4	fr		*	*	L	0	$A = A   n$
OR r	2	fr		*	*	L	0	$A = A   r$
SBC (IX+d)	9	fr	s	*	*	V	*	$A = A - (IX+d) - CY$
SBC (IY+d)	9	fr	s	*	*	V	*	$A = A - (IY+d) - CY$
SBC A,(HL)	5	fr	s	*	*	V	*	$A = A - (HL) - CY$
SBC A,n	4	fr		*	*	V	*	$A = A - n - CY$ (cout if $(r - CY) > A$ )
SBC A,r	2	fr		*	*	V	*	$A = A - r - CY$ (cout if $(r - CY) > A$ )
SUB (HL)	5	fr	s	*	*	V	*	$A = A - (HL)$
SUB (IX+d)	9	fr	s	*	*	V	*	$A = A - (IX+d)$
SUB (IY+d)	9	fr	s	*	*	V	*	$A = A - (IY+d)$
SUB n	4	fr		*	*	V	*	$A = A - n$
SUB r	2	fr		*	*	V	*	$A = A - r$
XOR (HL)	5	fr	s	*	*	L	0	$A = [A \& \sim(HL)]   [\sim A \& (HL)]$
XOR (IX+d)	9	fr	s	*	*	L	0	$A = [A \& \sim(IX+d)]   [\sim A \& (IX+d)]$
XOR (IY+d)	9	fr	s	*	*	L	0	$A = [A \& \sim(IY+d)]   [\sim A \& (IY+d)]$
XOR n	4	fr		*	*	L	0	$A = [A \& \sim n]   [\sim A \& n]$
XOR r	2	fr		*	*	L	0	$A = [A \& \sim r]   [\sim A \& r]$

Următoarele instrucțiuni ilustrează operațiile aritmetice și logice pe 8 biți:

```

#define      BIT_MASK    0xA5
cp    a, b                ;
and   BIT_MASK            ;
cp    0x01                ;
xor   0x80                ;

```

### 2.1.9. Operații aritmetice și logice pe 16 biți

Aceste instrucțiuni, prezentate în tabelul 2.10, sunt asemănătoare cu cele pe 8 biți, cu o singură diferență: deoarece procesorul Rabbit are doar regiștri pe 8 biți, instrucțiunile matematice pe 16 biți vor folosi ca operanzi perechi de regiștri.

Există o instrucțiune de înmulțire ce folosește ca operanzi regiștrii BC și DE, și memorează rezultatul în regiștrii HL și BC. Mai există diferite instrucțiuni de incrementare, decrementare și rotire care implică perechi de regiștri.

Tabelul 2.10. Operațiile aritmetice și logice pe 16 biți

Instrucțiune	Clk	A	I	S	Z	V	C	Operație
ADC HL,ss	4	fr		*	*	V	*	HL = HL + ss + CF -- ss=BC, DE, HL, SP
ADD HL,ss	2	fr		-	-	-	*	HL = HL + ss
ADD IX,xx	4	f		-	-	-	*	IX = IX + xx -- xx=BC, DE, IX, SP
ADD IY,yy	4	f		-	-	-	*	IY = IY + yy -- yy=BC, DE, IY, SP
ADD SP,d	4	f		-	-	-	*	SP = SP + d -- d=0 to 255
AND HL,DE	2	fr		*	*	L	0	HL = HL & DE
AND IX,DE	4	f		*	*	L	0	IX = IX & DE
AND IY,DE	4	f		*	*	L	0	IY = IY & DE
BOOL HL	2	fr		*	*	0	0	if (HL != 0) HL = 1, set flags to match HL
BOOL IX	4	f		*	*	0	0	if (IX != 0) IX = 1
BOOL IY	4	f		*	*	0	0	if (IY != 0) IY = 1
DEC IX	4			-	-	-	-	IX = IX - 1
DEC IY	4			-	-	-	-	IY = IY - 1
DEC ss	2	r		-	-	-	-	Ss = ss - 1 (ss= BC,DE, HL, SP)
INC IX	4			-	-	-	-	IX = IX + 1
INC IY	4			-	-	-	-	IY = IY + 1
INC ss	2	r		-	-	-	-	Ss = ss + 1 (ss= BC,DE, HL, SP)
MUL	12			-	-	-	-	HL:BC = BC * DE, signed 32 bit result. DE unchanged
OR HL,DE	2	fr		*	*	L	0	HL = HL   DE -- bitwise or
OR IX,DE	4	f		*	*	L	0	IX = IX   DE
OR IY,DE	4	f		*	*	L	0	IY = IY   DE
RL DE	2	fr		*	*	L	*	{CY,DE} = {DE,CY} -- left shift with CF
RR DE	2	fr		*	*	L	*	{DE,CY} = {CY,DE}
RR HL	2	fr		*	*	L	*	{HL,CY} = {CY,HL}

RR IX	4	f		*	*	L	*	{IX,CY} = {CY,IX}
RR IY	4	f		*	*	L	*	{IY,CY} = {CY,IY}
SBC HL,ss	4	fr		*	*	V	*	HL=HL-ss-CY (cout if (ss-CY)>hl)

#### 2.1.10. Setare, resetare și testare pe 8 biți

Instrucțiunile de manipulare a biților sunt folosite pentru a seta sau reseta biții, precum și pentru a testa starea acestora. Să considerăm un port paralel pe 8 biți care utilizează anumiți biți ca ieșiri; pentru a schimba starea unui singur bit, conținutul portului poate fi citit, un bit poate fi modificat folosind aceste instrucțiuni și apoi conținutul poate fi scris înapoi la port. Această secvență este adesea numită ca operație de citire-modificare-scriere.

Instrucțiunile din acest grup, prezentate în tabelul 2.11, permit manipularea biților în anumiți regiștri pe 8 biți, sau în locații de memorie pointate de perechile de regiștri HL, IX sau IY.

Tabelul 2.11. Setare, resetare și testare pe 8 biți

Instrucțiune	Clk	A	I	S	Z	V	C	Operație
BIT b, (HL)	7	f	s	-	*	-	-	(HL) & bit
BIT b, (IX+d)	10	f	s	-	*	-	-	(IX+d) & bit
BIT b, (IY+d)	10	f	s	-	*	-	-	(IY+d) & bit
BIT b, r	4	f		-	*	-	-	r & bit
RES b, (HL)	10		d	-	-	-	-	(HL) = (HL) & ~ bit
RES b, (IX+d)	13		d	-	-	-	-	(IX+d) = (IX+d) & ~ bit
RES b, (IY+d)	13		d	-	-	-	-	(IY+d) = (IY+d) & ~ bit
RES b, r	4	r		-	-	-	-	r = r & ~ bit
SET b, (HL)	10		b	-	-	-	-	(HL) = (HL)   bit
SET b, (IX+d)	13		b	-	-	-	-	(IX+d) = (IX+d)   bit
SET b, (IY+d)	13		b	-	-	-	-	(IY+d) = (IY+d)   bit
SET b, r	4	r		-	-	-	-	r = r   bit

În tabelul de mai sus, „bit” este o valoare între 0 și 7, unde 7 este cel mai semnificativ bit.

Următoarele instrucțiuni ilustrează diferite operații de setare, resetare și testare:

# define ON\_BIT        4        ; bit pentru pornirea motorului

set ON\_BIT, a        ; pornire motor

bit 7, (hl) ; testare bit ocupat  
res 7, (hl) ; ștergere bit ocupat

#### 2.1.11. Incrementare și decrementare pe 8 biți

Aceste instrucțiuni operează asupra locațiilor de memorie pointate de perechile de regiștri HL, IX sau IY, precum și asupra regiștrilor pe 8 biți ai procesorului. O incrementare sau decrementare este întotdeauna mai rapidă decât o adunare cu 1 sau o scădere cu 1, deoarece nu trebuie încărcată nici o dată imediată. Spre deosebire de instrucțiunile pe 16 biți INC/DEC, cele pe 8 biți modifică indicatorii de stare (flagurile). Instrucțiunile de incrementare și decrementare pe 8 biți sunt prezentate în tabelul 2.12.

Tabelul 2.12. Instrucțiunile de incrementare și decrementare pe 8 biți

Instrucțiune	Clk	A	I	S	Z	V	C	Operație
DEC (HL)	8	f	b	*	*	V	-	(HL) = (HL) - 1
DEC (IX+d)	12	f	b	*	*	V	-	(IX+d) = (IX+d) - 1
DEC (IY+d)	12	f	b	*	*	V	-	(IY+d) = (IY+d) - 1
DEC r	2	fr		*	*	V	-	r = r - 1
INC (HL)	8	f	b	*	*	V	-	(HL) = (HL) + 1
INC (IX+d)	12	f	b	*	*	V	-	(IX+d) = (IX+d) + 1
INC (IY+d)	12	f	b	*	*	V	-	(IY+d) = (IY+d) + 1
INC r	2	fr		*	*	V	-	r = r + 1

#### 2.1.12. Operații rapide cu acumulatorul pe 8 biți

Aceste instrucțiuni sunt considerate rapide deoarece ele nu încarcă date imediate și nu folosesc pointeri pentru a pointa operanzii – totul se petrece chiar în acumulator. Tabelul 6.13 prezintă instrucțiunile acestui grup.

Tabelul 2.13. Operații rapide cu acumulatorul pe 8 biți

Instrucțiune	Clk	A	I	S	Z	V	C	Operație
CPL	2	r		-	-	-	-	A = ~A
NEG	4	fr		*	*	V	*	A = 0 - A
RLA	2	fr		-	-	-	*	{CY,A} = {A,CY}
RLCA	2	fr		-	-	-	*	A = {A[6,0], A[7]}; CY = A[7]
RRA	2	fr		-	-	-	*	{A,CY} = {CY,A}
RRCA	2	fr		-	-	-	*	A = {A[0], A[7,1]}; CY = A[0]

### 2.1.13. Deplasări și rotații pe 8 biți

Aceste instrucțiuni permit deplasări sau rotații ale biților din regiștri sau locații de memorie. Înainte de a trece mai departe, este important să distingem între deplasări și rotații:

- O deplasare se produce atunci când biții dintr-un registru sau o locație de memorie sunt deplasați cu o poziție, la stânga sau la dreapta. Bitul deplasat afară va fi încărcat în indicatorul de transport (carry), în timp ce un 0 va intra în partea opusă. Există o situație specială (SRA), când bitul cel mai semnificativ rămâne neschimbat.
- O rotire diferă de o deplasare deoarece biții sunt deplasați afară prin carry și apoi intră în capătul opus (vezi figura 2.2). Sunt cazuri speciale despre care bit intră înapoi, și anume bitul carry sau bitul deplasat afară.

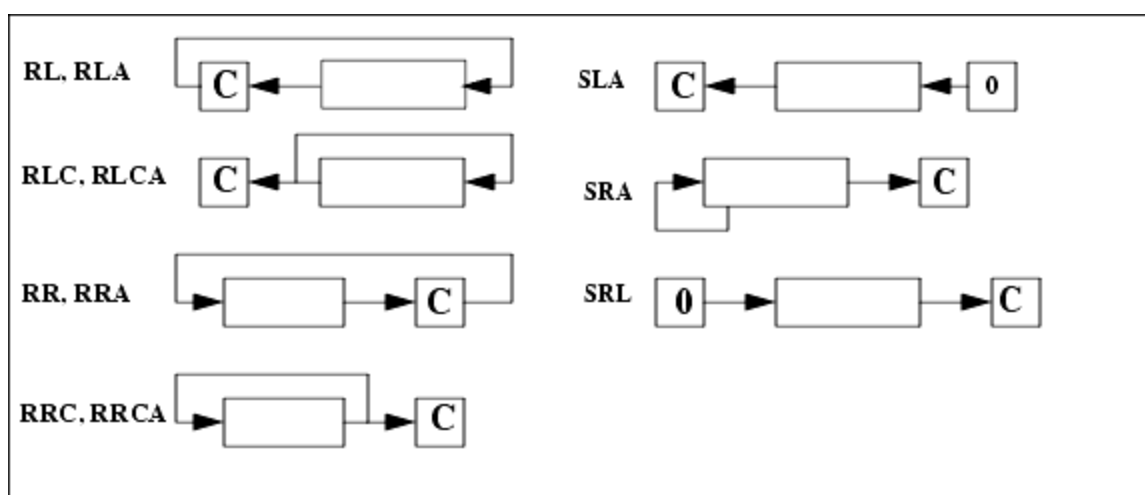


Figura 2.2. Deplasări și rotații pe 8 biți

Sunt multe motive pentru care programatorii vor să deplaseze biții:

- O deplasare la stânga are același efect ca o înmulțire cu 2, în timp ce o deplasare la dreapta semnifică o împărțire la 2. Aceasta este cea mai rapidă cale de a obține rezultatele. Un program poate necesita recepția unui octet, inversarea ordinii biților și transmiterea mai departe a rezultatului. Odată ce octetul a fost recepționat și verificat pentru corectitudine, este foarte simplu să deplasăm biții afară prin carry, apoi să rotim acei biți în alt registru sau locație de memorie. Acest lucru va determina inversarea ordinii biților la destinație. În cazul împărțirii la 2 se pierde din rezoluție, dar când se lucrează cu numere întregi este inevitabilă o reducere a preciziei.
- Uneori, programatorii folosesc portul paralel pentru a transmite date seriale, în special atunci când portul serial este deja utilizat. Atunci când se transmit date serial, programatorul va memora în general octetul ce trebuie transmis într-un registru paralel, apoi va deplasa fiecare bit ce trebuie transmis. Similar, la recepție, fiecare bit recepționat va fi deplasat astfel încât să se obțină tot octetul.

Tabelul 2.14 ilustrează instrucțiunile grupului deplasare și rotire.



Tabelul 2.14. Deplasări și rotații pe 8 biți

Instrucțiune	Clk	A	I	S	Z	V	C	Operație
RL (HL)	10	F	b	*	*	L	*	$\{CY, (HL)\} = \{(HL), CY\}$
RL (IX+d)	13	F	b	*	*	L	*	$\{CY, (IX+d)\} = \{(IX+d), CY\}$
RL (IY+d)	13	F	b	*	*	L	*	$\{CY, (IY+d)\} = \{(IY+d), CY\}$
RL r	4	fr		*	*	L	*	$\{CY, r\} = \{r, CY\}$
RLC (HL)	10	F	b	*	*	L	*	$(HL) = \{(HL)[6,0], (HL)[7]\}; CY = (HL)[7]$
RLC (IX+d)	13	F	b	*	*	L	*	$(IX+d) = \{(IX+d)[6,0], (IX+d)[7]\}; CY = (IX+d)[7]$
RLC (IY+d)	13	F	b	*	*	L	*	$(IY+d) = \{(IY+d)[6,0], (IY+d)[7]\}; CY = (IY+d)[7]$
RLC r	4	fr		*	*	L	*	$r = \{r[6,0], r[7]\}; CY = r[7]$
RR (HL)	10	F	b	*	*	L	*	$\{(HL), CY\} = \{CY, (HL)\}$
RR (IX+d)	13	F	b	*	*	L	*	$\{(IX+d), CY\} = \{CY, (IX+d)\}$
RR (IY+d)	13	F	b	*	*	L	*	$\{(IY+d), CY\} = \{CY, (IY+d)\}$
RR r	4	fr		*	*	L	*	$\{r, CY\} = \{CY, r\}$
RRC (HL)	10	F	b	*	*	L	*	$(HL) = \{(HL)[0], (HL)[7,1]\}; CY = (HL)[0]$
RRC (IX+d)	13	F	b	*	*	L	*	$(IX+d) = \{(IX+d)[0], (IX+d)[7,1]\}; CY = (IX+d)[0]$
RRC (IY+d)	13	F	b	*	*	L	*	$(IY+d) = \{(IY+d)[0], (IY+d)[7,1]\}; CY = (IY+d)[0]$
RRC r	4	fr		*	*	L	*	$r = \{r[0], r[7,1]\}; CY = r[0]$
SLA (HL)	10	F	b	*	*	L	*	$(HL) = \{(HL)[6,0], 0\}; CY = (HL)[7]$
SLA (IX+d)	13	F	b	*	*	L	*	$(IX+d) = \{(IX+d)[6,0], 0\}; CY = (IX+d)[7]$
SLA (IY+d)	13	F	b	*	*	L	*	$(IY+d) = \{(IY+d)[6,0], 0\}; CY = (IY+d)[7]$
SLA r	4	fr		*	*	L	*	$r = \{r[6,0], 0\}; CY = r[7]$
SRA (HL)	10	F	b	*	*	L	*	$(HL) = \{(HL)[7], (HL)[7,1]\}; CY = (HL)[0]$
SRA (IX+d)	13	F	b	*	*	L	*	$(IX+d) = \{(IX+d)[7], (IX+d)[7,1]\}; CY = (IX+d)[0]$
SRA (IY+d)	13	F	b	*	*	L	*	$(IY+d) = \{(IY+d)[7], (IY+d)[7,1]\}; CY = (IY+d)[0]$
SRA r	4	fr		*	*	L	*	$r = \{r[7], r[7,1]\}; CY = r[0]$
SRL (HL)	10	F	b	*	*	L	*	$(HL) = \{0, (HL)[7,1]\}; CY = (HL)[0]$
SRL (IX+d)	13	F	b	*	*	L	*	$(IX+d) = \{0, (IX+d)[7,1]\}; CY = (IX+d)[0]$
SRL (IY+d)	13	F	b	*	*	L	*	$(IY+d) = \{0, (IY+d)[7,1]\}; CY = (IY+d)[0]$
SRL r	4	fr		*	*	L	*	$r = \{0, r[7,1]\}; CY = r[0]$

#### 2.1.14. Prefixele instrucțiunilor

Procesorul Rabbit are două spații I/O: regiștrii I/O interni și regiștrii I/O externi. Prefixele instrucțiunilor IOI și IOE pot fi folosite pentru a genera cod care să acceseze regiștrii I/O interni sau externi, în loc să acceseze memoria. Folosind aceste prefixe, orice adresă de memorie pe 16 biți va fi decodificată ca o adresă I/O internă sau externă:

- Prefixul IOI determină următoarea instrucțiune să acceseze un port I/O intern în locul memoriei. Din moment ce perifericele I/O interne ocupă primii 256 de octeți ai spațiului I/O intern, octetul superior al adresei pe 16 biți este ignorat și se va utiliza doar octetul mai puțin semnificativ pentru a accesa portul de I/O intern.

Următoarea instrucțiune oferă un exemplu de utilizare al prefixului IOI.

```

        xor    a                                ; port e biții 1..7 intrări, 0 ieșire
        ld     (PEFRShadow), a                  ; update registru umbră
ioi     ld     (PEFR), a                        ; setare registru funcții

        ld     a, 0x01
        ld     (PEDDRShadow), a                ; update registru umbră
ioi     ld     (PEDDR), a                      ; setare registru direcție date

```

- Prefixul IOE determină instrucțiunea următoare să acceseze un port I/O extern în locul memoriei. Din moment ce perifericele I/O externe pot fi mapate în 64K, pentru accesarea porturilor I/O externe se utilizează o adresă pe 16 biți. Implicit, scrierile sunt inhibitate pentru operațiile cu porturi I/O externe, și sunt adăugate 15 stări de wait pentru accesările I/O.

Următoarea instrucțiune oferă un exemplu de utilizare al prefixului IOE.

```

#define     EXT_ADDRESS    0xFC00              ; adresa externă

        ld     hl, EXT_ADDRESS                 ; setare pointer
ioe     ld     (hl), a                        ; scrierea datelor la portul extern

```

- Prefixul ALTD determină instrucțiunea imediat următoare să modifice flagurile alternative, sau să utilizeze regiștri alternativi ca destinație a datelor, sau ambele. Folosirea acestei instrucțiuni permite programatorului să acceseze direct setul alternativ de regiștri sau flagurile alternative, fără a necesita interschimbarea tuturor regiștrilor.

Următoarea instrucțiune oferă un exemplu pentru folosirea prefixului ALTD:

```
altd    ex     de, hl                        ; de = index (era în hl')
```

Prefixele instrucțiunilor sunt prezentate în tabelul 2.15.

Tabelul 2.15. Prefixele instrucțiunilor

Instrucțiune	Clk	A	I	S	Z	V	C	Operație
ALTD	2			-	-	-	-	Destinație registru alternativ pentru următoarea instrucțiune
IOE	2			-	-	-	-	Prefix I/O extern
IOI	2			-	-	-	-	Prefix I/O intern

#### 2.1.15. Instrucțiuni de mutare a blocurilor

Aceste instrucțiuni, prezentate în tabelul 2.16, sunt folosite pentru a copia (sau muta) rapid blocuri de date dintr-o parte în alta a memoriei. Funcționează în modul următor:

- BC este setat ca și contor – cu numărul de octeți ce trebuie copiați
- HL este setat să pointeze blocul sursă
- DE este setat să pointeze blocul destinație

Instrucțiunile diferă între ele prin următoarele:

**LDD:**

- Setează DE să pointeze adresa destinație
- Setează HL să pointeze adresa sursă
- Este copiat un octet de date din (HL) în (DE)
- HL și DE sunt decrementate

**LDDR:**

- Setează DE să pointeze la adresa cea mai mare a blocului destinație
- Setează HL să pointeze la adresa cea mai mare a blocului sursă
- Se copie un bloc de date de la (HL) la (DE), dimensiunea blocului fiind precizată de BC
- HL și DE sunt decrementate pe parcursul operației

**LDI:**

- Setează DE să pointeze adresa destinație
- Setează HL să pointeze adresa sursă
- Este copiat un octet de date din (HL) în (DE)
- HL și DE sunt incrementate

**LDIR:**

- Setează DE să pointeze la adresa cea mai mică a blocului destinație
- Setează HL să pointeze la adresa cea mai mică a blocului sursă
- Se copie un bloc de date de la (HL) la (DE), dimensiunea blocului fiind precizată de BC
- HL și DE sunt incrementate pe parcursul operației.

Tabelul 2.16. Instrucțiuni de mutare a blocurilor

Instrucțiune	Clk	A	I	S	Z	V	C	Operație
LDD	10		D	-	-	*	-	(DE)=(HL);BC=BC-1;DE=DE-1;HL=HL-1
LDDR	6+7i		D	-	-	*	-	if {BC != 0} repeat:
LDI	10		D	-	-	*	-	(DE)=(HL);BC=BC-1;DE=DE+1;HL=HL+1
LDIR	6+7i		D	-	-	*	-	if {BC != 0} repeat:

## 2.1.16. Instrucțiuni de control – salturi și apeluri

Instrucțiunile din cadrul acestui grup afectează execuția programului și pot fi împărțite în următoarele categorii:

- Salturi necondiționate: aceste instrucțiuni pornesc execuția din altă parte a codului. După cum se poate observa din tabelul 2.17.a, aceste instrucțiuni pornesc execuția programului de la o adresă fixă ( $PC = mn$ ), sau de la o adresă pointată de regiștri HL, IX sau IY. Instrucțiunea LJP este specială deoarece permite saltul la o adresă calculată din XMEM.

Tabelul 2.17.a. Instrucțiunile de salt necondiționat

Instrucțiune	Clk	A	I	S	Z	V	C	Operație
JP mn	7			-	-	-	-	$PC = mn$
JP (HL)	4			-	-	-	-	$PC = HL$
JP (IX)	6			-	-	-	-	$PC = IX$
JP (IY)	6			-	-	-	-	$PC = IY$
JR e	5			-	-	-	-	$PC = PC + e$ (dacă $e=0$ , se execută următoarea instrucțiune)
LJP xpc, mn	10			-	-	-	-	$XPC = xpc$ ; $PC = mn$

- Salturile condiționate: aceste instrucțiuni, prezentate în tabelul 2.17.b, determină un salt dacă una din condițiile date este îndeplinită.

Tabelul 2.17.b. Instrucțiunile de salt condiționat

Instrucțiune	Clk	A	I	S	Z	V	C	Operație
JP f, mn	7			-	-	-	-	if {f} $PC = mn$
JR cc, e	5			-	-	-	-	If {cc} $PC = PC + e$

- Apelurile către subrutine pot fi făcute către o adresă pe 16 biți, folosind instrucțiunea CALL, sau către o adresă calculată din XMEM, folosind instrucțiunea LCALL, după cum se poate observa în tabelul 2.17.c.

Tabelul 2.17.c. Apelarea subrutinelor

Instrucțiune	Clk	A	I	S	Z	V	C	Operație
CALL mn	12			-	-	-	-	$(SP-1) = PCH$ ; $(SP-2) = PCL$ ; $PC = mn$ ; $SP = SP-2$
LCALL xpc, mn	19			-	-	-	-	$(SP-1) = XPC$ ; $(SP-2) = PCH$ ; $(SP-3) = PCL$ ; $XPC = xpc$ ; $PC = mn$ ; $SP = (SP-3)$

- Revenirea din subrutine se poate face necondiționat, folosind instrucțiunea RET, sau folosind condițiile flagurilor cu instrucțiunea RET f. LRET este utilizată pentru reveniri din subrutine aflate în spațiul de memorie XMEM. Aceste instrucțiuni sunt prezentate în tabelul 2.17.d.

Tabelul 2.17.d. Instrucțiuni de revenire

Instrucțiune	Clk	A	I	S	Z	V	C	Operație
RET	8			-	-	-	-	PCL = (SP); PCH = (SP+1); SP = SP+2
RET f	8/2		-		-	-	-	if {f} PCL = (SP); PCH = (SP+1); SP = SP+2
LRET	13			-	-	-	-	PCL = (SP); PCH = (SP+1); XPC = (SP+2); SP = SP+3;

Rutinele de tratare a întreruperilor (ISR) prezintă un caz special:

- Spre deosebire de alte procesoare, procesorul Rabbit folosește o instrucțiune RET față de RETI pentru a reveni dintr-o întrerupere. Instrucțiunea RETI este prezentată în tabelul 2.17.e și pare a nu folosi la nimic.

În majoritatea cazurilor, o rutină de tratare a întreruperii ar trebui să se termine după cum urmează:

ipres ; refacerea priorității întreruperii  
ret ; revenirea la programul întrerupt

Un programator nu are de făcut nimic deosebit pentru a utiliza această instrucțiune, adăugarea la sfârșitul rutinei de tratare a întreruperii a instrucțiunii RET fiind suficientă.

Tabelul 2.17.e. Instrucțiunea de revenire din întrerupere

Instrucțiune	Clk	A	I	S	Z	V	C	Operație
RETI	12			-	-	-	-	IP = (SP); PCL = (SP+1); PCH = (SP+2); SP = SP+3

- Decrementează și sari dacă nu e zero, prezentată în tabelul 2.17.f. Această instrucțiune este utilă în special la realizarea buclelor. Orice bloc de cod ce precede această instrucțiune poate fi repetat, cu condiția să îndeplinească următoarele condiții:

- blocul de cod poate fi executat de maxim 256 de ori (B = 0 la început);
- întregul bloc de cod ce va fi repetat trebuie să se încadreze într-o pagină de 256 de octeți, deoarece instrucțiunea face un salt relativ de maxim 256 de octeți față de poziția curentă.

Tabelul 2.17.f. Instrucțiunea DJNZ

Instrucțiune	Clk	A	I	S	Z	V	C	Operație
DJNZ j	5	R		-	-	-	-	B = B-1; if {B != 0} PC = PC+j

- Instrucțiunea de Reset (RST). Această instrucțiune, prezentată în tabelul 2.17.g, salvează PC curent pe stivă și apoi pornește execuția programului de la vectorul v din tabela de întreruperi. Salvarea valorii curente a PC pe stivă îi spune întreruperii de unde trebuie reluat programul după tratarea întreruperii.

După cum se poate observa din tabel, sunt disponibile RST 10, RST 18, RST 20, RST 28 și RST 38. Instrucțiunea RST 0x28 este specială deoarece transferă execuția programului către nucleul de depanare Dynamic C. Aceasta este singura instrucțiune de reset folosită de Dynamic C.

Tabelul 2.17.g. Instrucțiunea de reset

Instrucțiune	Clk	A	I	S	Z	V	C	Operație
RST v	10			-	-	-	-	(SP-1) = PCH; (SP-2) = PCL; SP = SP-2; PC = {R, v} v = 10, 18, 20, 28, 38

#### 2.1.17. Instrucțiuni diverse

Următoarele instrucțiuni nu se încadrează în nici unul din grupurile precedente; o descriere sumară a fiecărei instrucțiuni este prezentată în tabelul 2.18.

Tabelul 2.18. Instrucțiuni diverse

Instrucțiune	Clk	A	I	S	Z	V	C	Operație
CCF	2	f		-	-	-	*	CF = ~CF
IPSET 0	4			-	-	-	-	IP = {IP[5:0],00}
IPSET 1	4			-	-	-	-	IP = {IP[5:0],01}
IPSET 2	4			-	-	-	-	IP = {IP[5:0],10}
IPSET 3	4			-	-	-	-	IP = {IP[5:0],11}
IPRES	4			-	-	-	-	IP = {IP[1:0],IP[7:2]}
LD A, EIR	4	fr		*	*	-	-	A = EIR
LD A, IIR	4	fr		*	*	-	-	A = IIR
LD A, XPC	4	f		-	-	-	-	A = MMU
LD EIR, A	4			-	-	-	-	EIR = A
LD IIR, A	4			-	-	-	-	IIR = A
LD XPC, A	4			-	-	-	-	XPC = A
NOP	2			-	-	-	-	Nici o operație

POP IP	7			-	-	-	-	IP = (SP); SP = SP+1
PUSH IP	9			-	-	-	-	(SP-1) = IP; SP = SP-1
SCF	2	F		-	-	-	1	CF = 1

### 3. Echipamente și dispozitive folosite

Pentru buna desfășurare a lucrării de laborator se vor folosi următoarele dispozitive și resurse software:

- Modul RCM3365	10
- Placă de bază pentru modulul RCM3365	10
- Sursă de alimentare 12V	10
- Cablu serial pentru programarea RCM3365	10
- Compilator Dynamic C	10

### 4. Teme

1. Realizați un program scris în limbaj de asamblare care să realizeze incrementarea conținutului unei locații de memorie de fiecare dată când este lansat în execuție.
2. Realizați un program scris în limbaj de asamblare care la fiecare rulare să decrementeze conținutul unei locații de memorie.
3. Realizați un program în limbaj de asamblare care să umple o zonă de memorie RAM cu o constantă.
4. Scrieți un program în limbaj de asamblare care să copieze conținutul unei zone de memorie de la o adresă la altă adresă de memorie
5. Scrieți un program în limbaj de asamblare Rabbit care să realizeze căutarea unui anumit octet într-un șir de octeți predefinit (căutarea unui caracter într-un șir de caractere) și să depună în memorie octetul căutat și numărul de apariții al acestuia. Declararea unui șir și a lungimii lui se realizează în felul următor:  
sir: db "Acesta este un sir de caractere"  
lung equ \$ - sir
6. Scrieți un program în limbaj de asamblare care să realizeze sortarea în ordine crescătoare (descrescătoare) a unui șir de caractere.
7. Scrieți un program în limbaj de asamblare Rabbit care să caute un grup de octeți (un subșir) într-un șir de octeți predefinit și să depună în memorie grupul de octeți căutat și numărul de apariții al acestuia.