NOTIUNI DE ARHITECTURA CALCULATOARELOR

Introducere

In acest capitol se vor studia cateva notiuni legate de modul in care este realizat un calculator si cum este utilizat acesta in indeplinirea unor functii. In lumea specialistilor in calculatoare exista mai multe concepte legate de constructia unui calculator si modul interactiune al acestuia cu lumea exterioara. Printre aceste concepte se numara Arhitectura Calculatorului, Structura si Organizarea Calculatorului. Fiecare dintre ele desemneaza de fapt un anumit nivel de profunzime in descrierea al calculatorului.

Conceptul de arhitectura calculatoarelor se refera la parametrii unui calculator care sunt accesibili unui programator si care pot fi utilizati de acesta in scrierea programelor. Fara a intra in detaliile privind implementarea electronica a acestor componente ale calculatorului, arhitectura calculatorului ofera instrumentele de care are nevoie un programator pentru a implementa software-ul. De numarul si performantele componentelor desemnate de arhitectura calculatorului depind posibilitatile de implementare si performantele software-ului care va fi implementat pe acel calculator.

Conceptul de Structura si Organizarea Calculatorului face un pas mai departe in detalierea constructiei unui calculator si vizeaza implementarea diferitelor componente ale calculatorului cu ajutorul portilor logice elementare. Sunt studiate in aceasta directie modul de implementare al unitatilor aritmetico-logice, registrelor, multiplexoarelor, demultiplexoarelor, codificatoarelor de prioritate, etc. Acest nivel de detaliere nu mai este foarte interesant pentru programator, dar este esential pentru inginerii care se ocupa cu realizarea hardware a calculatorului. Pana la aparitia circuitelor integrate pe scara foarte larga de tip FPGA (Field Programable Gate Array), posibilitatile de alegere la acest nivel erau limitate la alegerea microprocesoarelor utilizate in sistem, a circuitelor integrate comunicare si interfatare etc si realizarea interactiunii intre aceste componente. Odata cu aparitia circuitelor de tip FPGA, un implementator de sisteme de calcul poate recurge la astfel de circuite si sa-si structureze propriul microprocesor in cadrul acestui circuit. Prin programerea circuitelor unui FPGA se pot obtine registre, unitati aritmetico-logice, multiplexoare-demultiplexoare, in fapt toate circuitele necesare unui microprocesor. Ca si nivelul Arhitecturii Calculatoarelor, nici la acest nivel nu se studiaza realizarea calculatoarelor la nivel de circuitelor electronice.

Nivelul de abordare al Arhitecturii Calculatoarelor in cadrul acestui capitol va fi cel necesar intelegerii de catre studenti a notiunilor care sa le permita formarea unei imagini cat mai clare privind principiile de realizare a unui calculator si modalitatile in care acesta poate indeplini diferitele sarcini. Notiunile prezentate in acest capitol vor fi legate de reprezentarea informatiei intr-un calculator numeric, tipuri de arhitecturi de calculatoare, arhitecturi de microprocesoare existente.

Reprezentarea informatiei in calculatorul numeric

Asa cum este binecunoscut, informatia in calculatorul numeric este reprezentata in format binar. Orice informatie este stocata in calculator sub forma unei succesiuni de 1 logic si 0 logic, semnificatia acesteia fiind interpretata in functie de context. In acest sens exista o reprezentare specifica a caracterelor alfa-numerice si o alta reprezentare specifica a informatiei numerice. Alaturi de codificarile comenzilor microprocesorului, acestea sunt categoriile de informatie care permit dezvoltarea marii majoritati a aplicatiilor calculatoarelor.

Reprezentarea caracterelor alfanumerice

Reprezentarea caracterelor alfanumerice in calculatoarelor a evoluat de-a lungul timpului si pe masura ce aplicatiile s-au diversificat s-a realizat si o standardizare a acesteia. In acest sens,

fiecare simbol este reprezentat sub forma unei combinatii binare unice denumita cod intern**.** S-a constatat ca modalitatile de reprezentare a caracterelor alfanumerice in calculatorul numeric trebuie sa indeplineasca un set de reguli, astfel incat acest set de caractere sa poata fi utilizat cat mai usor si in cat mai multe aplicatii [Mancas, D – Arhitectura Calculatoarelor – note de curs]. Aceste primcipii pot si enuntate dupa cum urmeaza:

-Numarul de combinatii de cod din cadrul setului trebuie sa fie suficient de mare astfel incat sa poata fi reprezentate toate simbolurile din setul dorit;

- Dimensiunea codului unui simbol trebuie sa fie corelata cu lungimea unitatii informationale adresabile (UIA). In prezent, cea mai cunoscuta unitate de informatie adresabila este octetul (8 biti) sau byte-ul. Ca urmare, s-au dezvoltat mai intai seturi de caractere pe 8 biti, care puteau reprezenta pana la 256 de caractere. Acestea au devenit insuficiente in timp, astfel incat s-a extins setul de caractere pe 16 biti, permitandu-se astfel reprezentarea a 65536 caractere

- Sa existe o corelatie intre reprezentarea caracterelor si simbolurile bazei de numeratie zecimale. In acest sens, reprezentarea simbolurilor bazei zecimale, in toate seturile de caractere standardizate se face prin coduri binare care se afla exact in succesiunea naturala a acestor caractere.

- Modul de codificare al simbolurilor trebuie sa permita realizarea simpla a unor operatii asupra informatiei reprezentate prin caractere alfanumerice, cum ar fi de exemplu ordonarea alfabetica. In acest sens, seturile de caractere larg utilizate respecta principiul ponderarii prin care o secventa ordonata de caractere alfanumerice corespunde unei secvente crescatoare de coduri binare.

Cele mai utilizate coduri pentru caracterele alfanumerice sunt codurile ASCII, EBCDIC, UNICODE.

Codul ASCII

Denumirea acestui cod provine de la *American Standard Code for Information Exchange*. Acest cod pentru caracterele alfanumerice a aparut in anii ’60, dar a fost standardizat in 1986. Acest cod de caractere alfanumerice respecta principiul ponderarii. Initial a fost un cod pe 7 biti, deci continea 128 de caractere, apoi a fost extins la 8 biti si a devenit codul ASCII-8

Tabelul 1. – Codul ASCII pe 7 biti

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

. Bitul cel mai semnificativ poate avea diverse semnificatii: fie este totdeauna 0, fie este folosit ca bit de paritate fie este utilizat pentru extinderea setului de caractere la 256. Cele doua tetrade ale unui caracter poarta denumiri consacrate – tetrada superioara este denumita *zona* iar cea inferioara *digit*. Varianta pe 7 biti (128 de caractere este prezentata in tabelul 1).

Codul Unicode

A fost creat de consortiul Unicode in scopul de a extinde numarul de caractere, astfel incat sa fie standardizate si caracterele altor limbi decat cele latine. In principiu a fost extins codul ASCII-7, la 16 biti, astfel incat pentru caracterele setului ASCII-7 octetul inferior s-a obtinut prin completarea MSB cu 0 si octetul superior este 00h . Poate codifica 65536 caractere si respecta principiul ponderarii. Este foarte utilizat in prezent de standardele software moderne. Reprezinta implementarea oficiala a standardului ISO/IEC 10646:2003. Primele 256 de caractere ale codului Unicode sunt prezentate in tabelul 2.

Tabelul 2. Primele 256 de caractere ale codului Unicode

0000 NULL	0020 SP	0040 @	0060 `	0080 Ctrl	00A0	00C0 À	00E0 à
0001 SOH	0021 !	0041 A	0061 a	0081 Ctrl	NBS	00C1 Á	00E1 á
0002 STX	0022 "	0042 B	0062 b	0082 Ctrl	00A1 ¡	00C2 Â	00E2 â
0003 ETX	0023 #	0043 C	0063 c	0083 Ctrl	00A2 ¢	00C3 Ã	00E3 ã
0004 EOT	0024 $	0044 D	0064 d	0084 Ctrl	00A3 £	00C4 Ä	00E4 ä
0005 ENQ	0025 %	0045 E	0065 e	0085 Ctrl	00A4 ¤	00C5 Å	00E5 å
0006 ACK	0026 &	0046 F	0066 f	0086 Ctrl	00A5 ¥	00C6 Æ	00E6 æ
0007 BEL	0027 '	0047 G	0067 g	0087 Ctrl	00A6 ¦	00C7 Ç	00E7 ç
0008 BS	0028 (	0048 H	0068 h	0088 Ctrl	00A7 §	00C8 È	00E8 è
0009 HT	0029)	0049 I	0069 i	0089 Ctrl	00A8 ¨	00C9 É	00E9 é
000A LF	002A *	004A J	006A j	008A Ctrl	00A9 ©	00CA Ê	00EA ê
000B VT	002B +	004B K	006B k	008B Ctrl	00AA ª	00CB Ë	00EB ë
000C FF	002C ,	004C L	006C l	008C Ctrl	00AB «	00CC Ì	00EC ì
000D CR	002D -	004D M	006D m	008DCtrl	00AC ¬	00CD Í	00ED í
000E SO	002E .	004E N	006E n	008E Ctrl	00AD -	00CE Î	00EE î
000F SI	002F /	004F O	006F o	008F Ctrl	00AE ®	00CF Ï	00EF ï
0010 DLE	0030 0	0050 P	0070 p	0090 Ctrl	00AF ¯	00D0 Ð	00F0 ð
0011 DC1	0031 1	0051 Q	0071 q	0091 Ctrl	00B0 °	00D1 Ñ	00F1 ñ
0012 DC2	0032 2	0052 R	0072 r	0092 Ctrl	00B1 ±	00D2 Ò	00F2 ò
0013 DC3	0033 3	0053 S	0073 s	0093 Ctrl	00B2 ²	00D3 Ó	00F3 ó
0014 DC4	0034 4	0054 T	0074 t	0094 Ctrl	00B3 ³	00D4 Ô	00F4 ô
0015 NAK	0035 5	0055 U	0075 u	0095 Ctrl	00B4 ´	00D5 Õ	00F5 õ
0016 SYN	0036 6	0056 V	0076 v	0096 Ctrl	00B5 µ	00D6 Ö	00F6 ö
0017 ETB	0037 7	0057 W	0077 w	0097 Ctrl	00B6 ¶	00D7 ×	00F7 ÷
0018 CAN	0038 8	0058 X	0078 x	0098 Ctrl	00B7 ·	00D8 Ø	00F8 ø
0019 EM	0039 9	0059 Y	0079 y	0099 Ctrl	00B8 ¸	00D9 Ù	00F9 ù
001A SUB	003A :	005A Z	007A z	009A Ctrl	00B9 ¹	00DA Ú	00FA ú
001B ESC	003B ;	005B [	007B {	009B Ctrl	00BA º	00DB Û	00FB û
001C FS	003C <	005C \	007C |	009C Ctrl	00BB »	00DC Ü	00FC ü
001D GS	003D =	005D]	007D }	009D Ctrl	00BC ¼	00DD Ý	00FD ý
001E RS	003E >	005E ^	007E ~	009E Ctrl	00BD ½	00DE Þ	00FE þ
001F US	003F ?	005F _	007F DEL	009F Ctrl	00BE ¾	00DF ß	00FF ÿ
					00BF ¿		

Codul EBCDIC

Denumirea sa semnifica *Extended Binary-Coded Decimal Interchange Code* si a fost dezvoltat de firma IBM, fiind folosit de sistemele de calcul ale acestei firme. Si acest cod respecta principiul ponderarii. Reprezentarea caracterelor din cod ASCII difera mult in cod EBCDIC. Marea majoritate a calculatoarelor accepta atat codul ASCII cat si EBCDIC.

Reprezentarea numerelor in calculatoare

Modul de reprezentare a numerelor in calculatoarele numerice a variat in timp in functie de dezvoltarea tehnologiei. S-a urmarit de cele mai multe ori obtinerea unor unitati aritmetico-logice cat mai simple din punctul de vedere al circuitelor componente.

Reprezentarea numerelor negative

O prima problema in reprezentarea numerelor in calculator este reprezentarea valorilor negative. Daca un numar pozitiv este usor de reprezentat intuitiv prin conversia sa in baza doi, pentru numerele negative au existat mai multe variante de reprezentare. Cea mai simpla conventie este reprezentarea in *cod direct*. In cazul acestei conventii, bitul cel mai semnificativ este rezervat ca bit de semn, restul bitilor reprezentand marimea numarului. Bitul de semn prin conventie este 0 pentru numere pozitive si 1 pentru numere negative.

Pentru conversia unui numar intreg in baza 2 se foloseste algoritmul impartirilor succesive. Se imparte numarul succesiv la 2 pana cand rezulta catul 0 si apoi se iau in ordine inversa resturile impartirilor. Spre exemplu pentru numarul 39, reprezentarea in baza 2 se obtine:

39 : 2 = 19 rest 1
19 : 2 = 9 rest 1
9 : 2 = 4 rest 1
4 : 2 = 2 rest 0
2 : 2 = 1 rest 0
1 : 2 = 0 rest 1

Reprezentarea numarului 39 pe 8 biti in cod direct este

$$39 \rightarrow 00100111$$

iar reprezentarea lui –5 in acelasi cod este

$$-39 \rightarrow 10100111.$$

Conform acestei conventii, atat 00000000, cat si 10000000 sunt reprezentari tot ale lui zero.

A doua conventie de reprezentare a numerelor negative este cea in complement fata de 1 sau *codul invers*. Numerele pozitive se reprezinta prin conversia modulului in baza 2 iar cele negative se obtin prin complementarea fata de 1 a numarului pozitiv cu acelasi modul. Conform acestei conventii de reprezentare, pe 8 biti, numarul 39 se reprezinta la fel ca si in cazul codului direct, iar numarul –39 se reprezinta sub forma:

$$-39 \rightarrow 11011000.$$

Si in cadrul acestei conventii numarul zero are doua reprezentari: 00000000 si 11111111. Bitul cel mai semnificativ este interpretat ca bit de semn, fiind 0 pentru numere pozitive si 1 pentru numere negative. In acest sens 10000000 nu reprezinta 128 ci –127.

Cele doua conventii de reprezentare mentionate mai sus prezinita dezavantajul ca necesita circuite mai complicate pentru implementare. Un cod de reprezentare a numerelor care simplifica mult circuitele unitatii aritmetico-logice este *codul complementar*. Acesta foloseste reprezentarea numerelor in negative in complement fata de 2. Numerele pozitive se reprezinta la fel ca si in cazul celorlalte doua coduri, prin conversia valorii in baza doi. Cea mai utilizata versiune a codului complementar este cea cu un bit de semn si cu pondere negativa a acestuia. Conform acestei conventii, bitul cel mai semnificativ este interpretat ca bit de semn si este 0 pentru numere pozitive si 1 pentru numere negative. In plus, pentru o reprezentare pe n biti, se considera ca bitul de semn detine ponderea (-2^n), astfel incat pentru calculul unui numar reprezentat in cod complementar se poate folosi relatia

$$N = 1 \cdot \left(-2^n\right) + \sum_{i=0}^{n-1} \overline{a}_i \cdot 2^i + 1, \qquad (1)$$

in care

$$|N| = \sum_{i=0}^{n-1} a_i \cdot 2^i, \qquad (2)$$

iar $\overline{a}_i$ este complementul fata de 1 al bitului a_i.

Mai practic, calculul reprezentarii unui numar negativ se face in modul urmator: se calculeaza modulul in baza 2 si apoi se calculeaza complementul fata de 2 al modulului, prin urmatoarea regula. Se incepe de la bitul cel mai putin semnificativ si se lasa necomplementati toti bitii pana la primul bit care are valoarea 1, inclusiv. In continuare se complementeaza toti bitii pana la bitul cel mai semnificativ. Astfel, pentru numarul 92 in cod complementar pe 8 biti se obtine urmatoarea reprezentare:

$$92 \rightarrow 01011100$$

iar numarul –92 reprezentat in cod complementar pe 8 biti are reprezentarea

$$-92 \rightarrow 10100100.$$

In cadrul acestei conventii, secventa 10000000 este o exceptie si i se atribuie valoarea –128. Scara de reprezentare a numerelor in acest caz pe cei 8 biti este intre –128 si 127, deci este asimetrica, iar 0 are o singura reprezentare – secventa 00000000.

Reprezentarea numerelor fractionare

Pentru reprezentarea numerelor fractionare se utilizeaza doua conventii: conventia in virgula fixa si conventia in virgula mobila. Fiecare dintre aceste reprezentari prezinta avantaje si dezavantaje, existand aplicatii in care este de preferat una dintre ele sau alta. De obicei insa programele uzuale utilizeaza reprezentarea in virgula mobila.

Reprezentarea in virgula fixa (FXP)

In cazul conventiei in virgula fixa trebuie specificat numarul de biti pe care se reprezinta partea intreaga a numarului si numarul de biti pe care se reprezinta partea fractionara a acestuia. Calculul modulului se face prin conversia numarului in baza 2, iar daca numarul este negativ atunci se reprezinta conform codului complementar prin complementarea modulului fata de 2. Pentru

conversia in baza 2 a unui numar fractionar se face separat conversia partii intregi si conversia partii fractionare. Pentru conversia partii intregi se foloseste algoritmul impartirilor succesive prezentat anterior.

Pentru conversia partii fractionare se utilizeaza algoritmul inmultirilor succesive, astfel: Se retine partea fractionara a numarului, se inmulteste succesiv cu 2 si se retine valoarea cifrei obtinute in stanga virgulei, pana cand se completeaza toti bitii partii fractionare. Spre exemplu reprezentarea numarului 0.637 pe un numar de 8 biti la partea fractionara rezulta

$$0.637 \rightarrow .1010001$$

valoarea obtinuta de fapt prin reprezentare pe 8 biti este 0.6328125, care difera de valoarea convertita. Reprezentarea numerelor pe un numar finit de biti reprezinta una din sursele de erori in calculatoarele digitale.

Pentru reprezentarea numarului 12.435 in virgula fixa cu 8 biti la partea intreaga, din care primul bit de semn si 8 biti la partea fractionara rezulta

$$12.435 \rightarrow 00001100.01101111$$

care reprezinta de fapt valoarea exacta 12.43359375.

Pentru reprezentarea lui –12.435 se face complementarea fata de 2 a lui 12.435, astfel incat se obtine

$$-12.435 \rightarrow 11110011.10010001$$

care este si el o aproximare a lui –12.435, nu este valoarea exacta.

Reprezentarea numerelor in virgula fixa are avantajul ca pasul de reprezentare al numerelor este constant, deci precizia absoluta de reprezentare a numerelor este constanta, dar are dezavantajul unui domeniu mai restrans de reprezentare pe un numar dat de biti iar precizia relativa de reprezentare a numerelor este variabila. Numerele mici in modul se reprezinta cu o eroare relativa mare, iar numerele mari in modul se reprezinta cu o eroare relativa mica. Aceasta face ca acest mod de reprezentare sa duca la o cumulare mai accentuata a erorilor in calculul numeric, fapt care trebuie evitat in aplicatiile tehnice si mai ales in aviatie. Un alt inconvenient este aparitia frecventa a depasirii de format la operatiile de adunare si scadere.

Reprezentarea in virgula mobila

Reprezentarea in virgula mobila implica formatul de reprezentare

bit de semn – exponent – mantisa.

Numarul se calculeaza conform relatiei

$$N = 2^{\exp onent} \cdot mantisa$$

Bitul de semn este ca si in reprezentarile anterioare este 0 pentru numere pozitive si 1 pentru numere negative. Reprezinta de fapt semnul mantisei.

Deoarece atat mantisa cat si exponentul pot fi numere pozitive sau negative, s-a recurs la un artificiu de reprezentare al exponentului, astfel incat sa se evite utilizarea a doi biti de semn. In acest sens exponentul este totdeauna numar intreg si este "oftsetat", astfel incat in reprezentarea din calculator pe pozitia exponentului sa apara totdeauna o valoare pozitiva sau 0. In calculator, valoare reprezentata pe pozitia exponentului poarta denumirea de *caracteristica*, "offsetul" utilizat purtand

denumirea de *exces*. Daca exponentul se reprezinta pe k biti, atunci excesul este 2^{k-1}, iar caracteristica este de fapt suma dintre exponent si exces:

$$C=E+T.$$

Spre exemplu, pentru reprezentarea exponentului 7 pe 5 biti, se foloseste un exces $2^{5-1}=16$, deci caracteristica va fi 23. Pe pozitia rezervata reprezentarii exponentului se va gasi deci caracteristica

$$23 \rightarrow 10111.$$

In acelasi format, pentru reprezentarea exponentului –7, caracteristica va fi –7+16=9, deci pe pozitia rezervata reprezentarii exponentului va apare

$$9 \rightarrow 01001.$$

Domeniul de valori ale exponentului care pot fi reprezentate in acest format vor fi intre –16 si +15.

Mantisa se reprezinta in format binar. Deoarece in virgula mobila reprezentarea unui numar nu este unica, rezulta ca pentru acelasi pot apare numar diverse valori pentru mantisa si exponent. De exemplu

$$2^3 \cdot 0.001011 = 2^1 \cdot 0.1011.$$

Pentru a obtine o reprezentare unica se utilizeaza reprezentarea cu mantisa normalizata, astfel incat prima cifra dupa virgula sa fie totdeauna 1. Astfel, pentru numarul 1,375 se foloseste reprezentarea $2^1 \cdot 0.1011$.

Pentru a castiga un bit de precizie, s-a facut observatia ca oricum primul bit dupa virgula este 1, ca urmare el nu se mai reprezinta, astfel incat reprezentarea mantisei in cazul anterior va fi secventa 011 si nu secventa 1011. Mantisa normalizata reprezentata pe 8 biti va fi deci 01100000 si nu 10110000 cum ar fi de asteptat. Combinatia formata din acel 1 subinteles, punctul zecimal introdus in mod conventional si cifrele care apar in continuarea reprezentarii mantisei poarta numele de signifiant

Pentru numarul 1.375 reprezentat in formatul 1 bit de semn, 5 biti pentru caracteristica si 8 biti pentru mantisa normalizata, si tinand cont si de excesul 16 rezulta reprezentarea

$$1.375 \rightarrow 0\ 10001\ 01100000$$

Signifiantul este

$$s \rightarrow 1.01100000$$

Pentru numarul –1.375, se tine cont ca mantisa este negativa, astfel incat prin complementarea acesteia fata de 2 se obtine 0101, dar care nu mai este normalizata. Pentru normalizare se deplaseaza mantisa cu doua ranguri la stanga, astfel incat exponentul scade cu doua unitati si se obtine caracteristica 15, nu 17 ca in cazul anterior. Deci reprezentarea lui –1.375 in acelasi format va fi

$$-1.375 \rightarrow 1\ 01111\ 01000000.$$

In prezentarea anterioara am ales un numar arbitrar de biti pentru reprezentarea caracteristicii si a mantisei. Pentru o reprezentare unitara a numerelor s-au standardizat numarul de

biti pentru caracteristica si mantisa. Astfel standardul IEEE 754 prevede trei variante de reprezentare in virgula mobila:

- reprezentare in simpla precizie, pe 32 de biti, in format 1 bit de semn + 8 biti pentru caracteristica + 23 biti partea de dupa virgula a signifiantului;

- reprezentare in dubla precizie , in format 1 bit de semn + 11 biti pentru caracteristica + 52 biti partea de dupa virgula a signifiantului;

- reprezentare in precizie extinsa, in format 1 bit de semn + 15 biti pentru caracteristica + 64 biti pentru partea de dupa virgula a signifiantului.

- reprezentare in cuadrupla precizie, in format 1 bit de semn + 15 biti pentru caracteristica + 112 biti pentru partea de dupa virgula a signifiantului.

In cazul reprezentarii in simpla precizie excesul este considerat 127 si exponentul poate lua conform celor prezentate valori intre –127 si 128. Valorile extreme insa nu sunt utilizate pentru reprezentarea normala numerelor conform regulii anterioare, ci pentru extinderea formatului de reprezentare, asa cum va fi prezentat in continuare, astfel incat exponentul poate varia intre –126 si 127. Valorile maxime si minime ale modulului unui numar reprezentabil in simpla precizie sunt $3.4028 \cdot 10^{38}$ si respectiv $1.1754 \cdot 10^{-38}$.

In cazul reprezentarii in dubla precizie excesul este considerat 1023 si exponentul poate lua valori intre –1023 si 1024. Din nou insa valorile extreme sunt utilizate pentru extensia formatului de reprezentare, astfel incat exponentul poate varia de fapt intre –1022 si 1023. Valorile maxime si minime ale modulului unui numar reprezentabil in dubla precizie sunt $1.7976 \cdot 10^{308}$ si respectiv $2.2250 \cdot 10^{-308}$.

In cazul reprezentarii in precizie extinsa si cuadrupla excesul este considerat 16383 si exponentul poate lua valori intre –16383 si 16384. Din nou insa valorile extreme sunt utilizate pentru extensia formatului de reprezentare, astfel incat exponentul poate varia de fapt intre –16382 si 16383. Valorile maxime si minime ale modulului unui numar reprezentabil in precizie cuadrupla sau extinsa sunt $1.1897 \cdot 10^{4932}$ si respectiv $3.3631 \cdot 10^{-4932}$. Diferenta dintre precizia extinsa si precizia cuadrupla este data de numarul de biti ai mantisei.

Atentie! Baza de reprezentare a mantiselor este considerata 2 si nu puteri ale lui 2. Pentru reprezentarea unui numar in calculator, se face mai intai reprezentarea lui in baza 2, dupa care se face normalizarea conform regulilor de mai sus.

Asa cum se poate remarca din cele mentionate mai sus, pot apare doua situatii in care numarul nu mai poate fi reprezentat corect in calculator. Daca reprezentarea este in simpla precizie si modulul numarului depaseste 2^{127}, atunci se spune s-a realizat supradepasire (*overflow*). Daca modulul numarului este sub 2^{-126} atunci se pune ca s-a realizat subdepasire de format (*underflow*). La fel se intampla pentru cazul reprezentarii in dubla precizie si modulul depaseste 2^{1023} (*overflow*), respectiv daca modulul este sub 2^{-1022} (*underflow*). In cazul reprezentarii in precizie extinsa sau cuadruple aceste limite sunt de 2^{16383} pentru supradepasire si respectiv 2^{-16382} pentru subdepasire.

Pentru extinderea si mai mult a domeniului de reprezentare a numerelor mici, s-a recurs la reprezentarea denormalizata a mantisei. Astfel, daca apare caracteristica 0 exponentul este considerat 2^{-126} pentru simpla precizie, 2^{-1022} pentru dubla precizie si respectiv 2^{-16382} pentru precizia extinsa si cuadrupla. Mantisa nu se mai considera normalizata in acest caz si se reprezinta pe bitii corespunzatori, adica 23 biti pentru reprezentarea in simpla precizie, 52 pentru dubla precizie, 64 pentru precizia extinsa si 112 pentru precizie cuadrupla. De data aceasta campurile respective reprezinta chiar mantisa, nu partea de dupa virgula a signifiantului.

Standardul IEEE 754 prevede cazuri speciale si pentru situatia in care caracteristica are toti bitii 1. Conventional, daca toti bitii caracteristicii sunt 1 si mantisa are toti bitii 0, se considera ca aceasta combinatie reprezinta $\pm\infty$. In cazul in care toti bitii caracteristicii sunt 1, dar mantisa contine orice alta valoare decat 0, se considera ca aceasta combinatie nu reprezinta un numar (*NaN*).

In tabelul 2 sunt prezentate semnificatiile reprezentarii numerelor in virgula mobila, conform standardului IEEE 754

Tabelul 3. – Reprezentarea numerelor conform standardului IEEE 754

Semnificatie	Bit de semn	Caracteristica	Mantisa
Numere normalizate	0 sau 1	0< caracteristica< C_{max}	Orice valoare
± 0	0 sau 1	0	0
Numere denormalizate	0 sau 1	0	Orice valoare $\neq 0$
$\pm\infty$	0 sau 1	1111…..1	0
NaN	0 sau 1	1111…..1	Orice valoare $\neq 0$

ARITMETICA ELEMENTARA A CALCULATOARELOR

Desi la nivel utilizator calculatoarele pot efectua cele mai complexe calcule matematice, totusi, la nivelul microprocesorului, si mai precis la nivelul unitatii aritmetico-logice (UAL) calculatoarele nu pot efectua decat operatii foarte simple. Foarte putine sunt microprocesoarele de uz general care au implementate la nivel hardware operatii matematice complexe. Toti algoritmii de calcul sunt descompusi pe baza algoritmilor de metode numerice in operatiile aritmetice simple – adunare, scadere, inmultire si impartire. Calculul unei functii trigonometrice de exemplu se face prin utilizarea dezvoltarii in serie a functiei respective. In acest sens trebuie mentionat ca atunci cand se specifica numarul de operatii pe secunda de care este capabil un microprocesor este referit acest numar de operatii elementare executate de UAL, nu numarul de operatii complexe, cum ar fi calculul unei functii trigonometrice sau inmultirea a doua matrici. Chiar si inmultirea a doua numere intregi reprezentate pe mai multi octeti (byte) presupune destul de multe operatii elementare atunci cand UAL-ul este construit pentru lucrul pe 8 biti (1 byte).

Vom prezenta in continuare cateva notiuni si algoritmi de realizare a operatiilor aritmetice elementare care sunt implementate in microprocesoarele de uz general. Asa cum se va vedea in continuare, in unele cazuri este mai simpla utilizarea codului direct iar in alte cazuri utilizarea codului complementar. Totusi, operatiile trebuiesc realizate utilizand o reprezentare unitara a operanzilor si rezultatelor. Din prezentarea care urmeaza se pot identifica dificultatile care aprar pentru fiecare operatie in parte si cum sunt rezolvate in unele cazuri.

Adunarea si scaderea numerelor

Consideram in continuare numere intregi reprezentate pe 8 biti, din care bitul cel mai semnificativ este bitul de semn. Deci vom avea un bit de semn si 7 biti pentru reprezentarea modulului numarului.

Pentru a realiza operatia de adunare a doua numere reprezentate pe mai multi biti, este necesar mai intai sa fie inteles modul in care se realizeaza adunarea a doua numere pe un bit. Adunarea a doua numere pe un bit este descrisa de tabela de adevar din tabelul 1. In acest tabel apare pe langa suma S=a+b si un bit suplimentar, bitul de transport, care arata ca suma realizata conduce la un rezultat reprezentabil pe mai multi biti. Este cazul similar adunarii a doua numere zecimale de o cifra, atunci cand rezultatul depaseste valoarea 9.

Sinteza coloanelor S si T cu ajutorul functiilor logice conduce la expresiile urmatoare:

Tabelul 1. – Tabela de adevar a operatiei de adunare a doua numere pe un bit

a	b	S=a+b	T
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$$S = a \oplus b \quad (3)$$
$$T = a \cdot b \quad (4)$$

Relatiile (3) si 94) pot fi implementate cu ajutorul schemei logice din figura 1. Acest circuit poarta numele de semisumator elementar. Acesta insa nu este suficient pentru realizarea operatiei de adunare a doua numere reprezentate pe mai multi biti, asa cum ne-am propus.

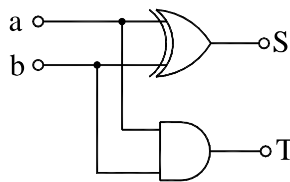


Fig. 1 – Semisumatorul elementar

Pentru a realiza adunarea numerelor pe mai multi biti este necesar sa se tina cont si de transportul care poate apare de la rangurile inferioare catre rangurile superioare. In acest caz, un sumator a doua numere pe un bit va avea trei intrari – numerele a si b si bitul de transport de la rangul inferior T_i. Ca iesiri, va avea tot bitul S care reprezinta suma obtinuta si bitul de transport catre rangul superior notat acum cu T_s. Se va realiza acum de fapt suma a trei numere de un bit. Tabela de adevar pentru acest caz este prezentata in tabelul 2.

Tabelul 2 – Tabela de adevar pentru adunarea a doua numere pe un bit, tinand cont si de bitul de transport de la rangul inferior

a	b	T_i	S	T_s
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Prin sinteza cu ajutorul functiilor logice, se obtin urmatoarele expresii pentru S si T_s .

$$S = a \oplus b \oplus T_i \quad (5)$$
$$T_s = a \cdot b + T_i \cdot (a + b) \quad (6)$$

sau

$$T_s = a \cdot b + T_i \cdot (a \oplus b). \quad (7)$$

Implementarea acestor relatii cu ajutorul portilor logice XOR si NAND poate fi facuta ca in figura 2.

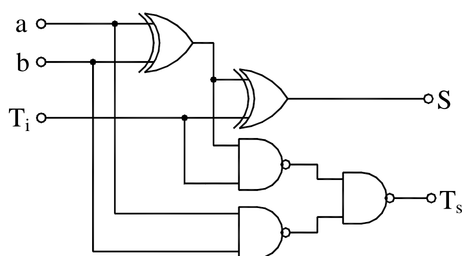


Fig. 2 – Sumatorul elementar pe un bit.

Utilizand astfel de celule sumatoare pe un bit conectate in cascada se poate realiza un sumator de numere binare pe oricati biti. Pentru cazul numerelor cu modulul reprezentat pe 7 biti cum am considerat exemplul nostru, rezulta schema din figura 3.

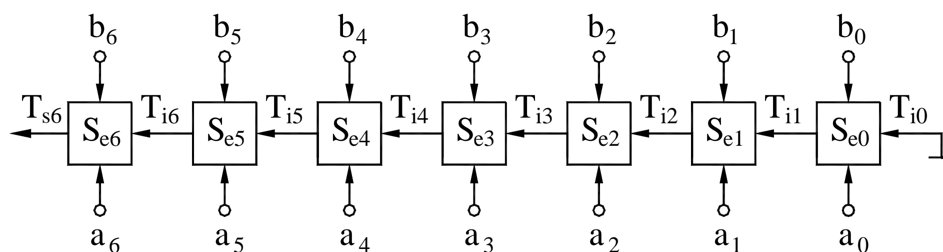


Fig. 3 – Sumator pe 7 biti

Schema din figura 3 are ca dezavantaj important faptul ca pentru obtinerea rezultatului, semnalele de transport trebuie sa se propage de la bitul 0 pana la bitul 6. Deci tinand cont de latenta fiecarui sumator elementar in parte, se va obtine un timp de efectuare a sumarii pe 7 biti egal cu de 7 ori timpul de efectuare a sumarii pe un bit. Deci un sumator in aceasta varianta este foarte lent, mai ales daca se realizeaza sumarea pe 16 sau 32 de biti.

O varianta pentru obtinerea unui sumator mai rapid presupune prelucrarea relatiilor (5) si (7) scrise pentru adunarea pe mai multi biti. De exemplu pentru adunarea pe 4 biti se poate scrie

$$T_{s0} = G_0 + T_{i0} \cdot P_0, \quad (8)$$

$$T_{s1} = G_1 + T_{s0} \cdot P_1, \quad (9)$$

$$T_{s2} = G_2 + T_{s1} \cdot P_2, \quad (10)$$

$$T_{s3} = G_3 + T_{s2} \cdot P_3, \quad (11)$$

in care $G_i = a_i \cdot b_i$, poarta numele de componenta de generare si $P_i = a_i \oplus b_i$ poarta numele de componenta de propagare . In relatiile (8)-(11) termenii G_i si P_i se calculeaza foarte repede cu ajutorul unei singure porti logice fiecare, si acesti termeni nu depind de rangurile adiacente. Daca se inlocuiesc succesiv una in cealalta relatiile (8) – (11) se obtin expresiile bitilor de transport intre ranguri

$$T_{s0} = G_0 + T_{i0} \cdot P_0 \quad (12)$$

$$T_{s1} = G_1 + G_0 \cdot P_1 + T_{i0} \cdot P_0 \cdot P_1 \quad (13)$$

$$T_{s2} = G_2 + G_1 \cdot P_2 + G_0 \cdot P_1 \cdot P_2 + T_{i0} \cdot P_0 \cdot P_1 \cdot P_2 \quad (14)$$

$$T_{s3} = G_3 + G_2 \cdot P_3 + G_1 \cdot P_2 \cdot P_3 + G_0 \cdot P_1 \cdot P_2 \cdot P_3 + T_{i0} \cdot P_0 \cdot P_1 \cdot P_2 \cdot P_3. \quad (15)$$

Se observa in relatiile (12) - (15) ca bitii de transport intre ranguri poate fi calculati fara a se astepta transferul rezultatelor intre ranguri, asa cum este cazul conectarii in cascada a sumatoarelor elementare. Sunt necesari doar termenii G_i , P_i si T_{i0}. G_i si P_i se pot calcula foarte repede simultan

iar bitul T_{i0} este disponibil inca de la inceputul operatiei ca data de intrare. Desi bitii de transport intre ranguri apar si in calculul termenilor S_i conform relatiei (5) ei pot fi calculati foarte repede, fara a fi necesar sa se astepte transferul rezultatelor intre ranguri. Pretul platit insa este cresterea complexitatii schemei sumatorului. Apare un bloc de calcul anticipat al bitilor de transport (BGTA in figura 4) care primeste la intrare bitul de transport T_{i0} si iesirile G_i si P_i corespunzatoare fiecarui rang si furnizeaza la iesire bitii de transport intre ranguri care se utilizeaza in calculul bitilor S_i. Si in acest caz, timpul necesar calculului sumei creste pe masura ce creste lungimea operanzilor deoarece creste complexitatea blocului de calcul anticipat al transporturilor, dar cresterea este mult mai mica. Se poate spune ca si castigul de timp fata de conectarea in cascada a sumatoarelor elementare creste pe masura ce creste lungimea operanzilor.

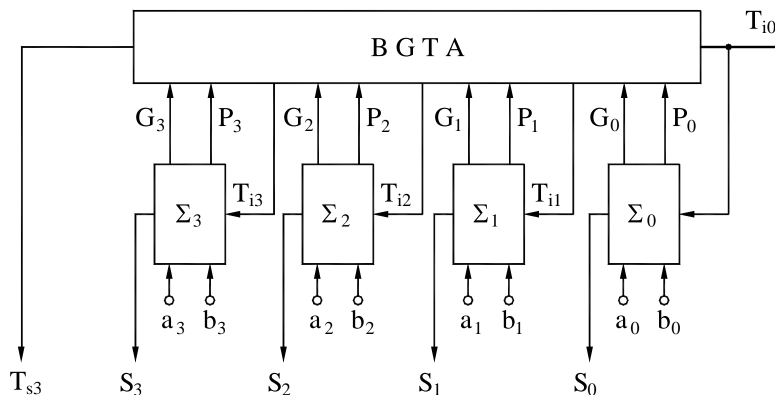


Fig. 4 – Sumator pe patru biti cu calcul anticipat al transporturilor

Ca un compromis la cresterea substantiala a complexitatii BGTA odata cu cresterea numarului de biti al operanzilor se pot conecta in cascada sumatoare cu transport anticipat pe mai putini biti. De exemplu se poate obtine un sumator pe 8 biti din doua sumatoare cu transport anticipat pe 4 biti utilizand schema din figura 5. Evident ca timpul de calcul va fi mai mare decat la utilizarea unui sumator cu un singur BGTA pe 8 biti, dar mai mic decat la conectarea in cascada a 8 sumatoare elementare.

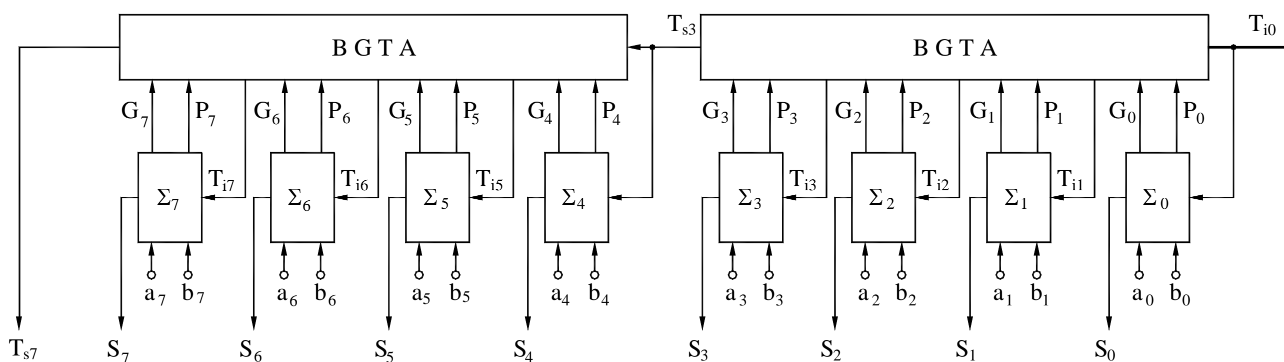


Fig. 5 – Sumator pe 8 biti obtinut prin conexiunea in cascada a doua sumatoare pe 4 biti
cu calcul anticipat al transportului

Circuitele prezentate pana aici pot efectua suma a doua numere fara semn. Ne-am propus sa prezentam modalitatea de rezolvare a problemei adunarii a doua numere intregi cu semn reprezentate in format 1+7. La adunarea a doua numere cu semn reprezentate in cod direct apar unele probleme importante atunci cand cel putin unul din operanzi este negativ. Astfel, Daca unul dintre operanzi este pozitiv si unul este negativ trebuie realizata scaderea modulelor celor doi operanzi si semnul rezultatului ia semnul operandului mai mare in modul. Ca urmare algoritmul de efectuare a adunarii in acest caz este urmatorul:

- se compara modulele operanzilor;
- se atribuie rezultatului semnul operandului mai mare in modul;
- se calculeaza modulul rezultatului ca diferenta intre modulele operanzilor (se scade modulul mai mic din cel mai mare).

Pentru implementarea hardware a acestui algoritm se observa ca mai sunt necesare circuite comparatoare prentru a face comparatia dintre cele doua module, realizarea unor circuite care sa faca diferenta a doua numere si nu suma lor pentru calculul modulului rezultatului si circuite pentru calculul bitului de semn in functie de rezultatul comparatiei de la primul pas. Circuite care fac diferenta a doua numere in format binar se pot realiza, dar ele trebuie implementate alaturi de sumatoare, iar la efectuarea calculelor trebuie sa existe un bloc logic care sa decida daca se face adunarea modulelor numerelor sau scaderea acestora.

Pentru cazul adunarii a doua numere intregi reprezentate in format 1+7 cu semne diferite, nu este posibil sa apara depasire de format. Este posibil insa sa apara depasire de format daca numerele au acelasi semn. Aceasta situatie trebuie sa fie sesizata utilizatorului si complica si mai mult blocul de realizare a sumei a doua numere intregi.

Problema se rezolva mult mai simplu daca se utilizeaza rerezentarea numerelor tot in format 1+7 dar in cod complementar. Atunci, pentru adunarea numerelor, indiferent ca sunt pozitive sau negative se face pur si simplu adunarea reprezentarilor acelor numere. Dam ca exemple operatiile de adunare de mai jos

$$\begin{array}{l} 42 \rightarrow 00101010\ + \\ \underline{56 \rightarrow 00111000} \\ 98 \rightarrow 01100010 \qquad T_{s6} = 0\,,\ T_{s7} = 0. \end{array}$$

Adunarea 42 + (-56) presupune adunarea lui 42 cu –56 reprezentat in complement fata de doi, deci vom obtine:

$$\begin{array}{r} 42 \rightarrow 00101010\ + \\ \underline{-56 \rightarrow 11001000} \\ -14 \rightarrow 11110010 \qquad T_{s6} = 0\,,\ T_{s7} = 0. \end{array}$$

Se observa ca bitul de semna a rezultat 1, deci rezultatul este negativ si pentru a verifica modulul, complementam fata de 2 rezultatul si obtinem 00001110, care este chiar reprezentara lui 14 in format binar.

Adunarea –42 + (-56) presupune adunarea acestor numere reprezentate in complement fata de 2, deci vom obtine:

$$\begin{array}{l} -42 \rightarrow 11010110\ + \\ \underline{-56 \rightarrow 11001000} \\ -98 \rightarrow 10011110 \qquad T_{s6} = 1\,,\ T_{s7} = 1. \end{array}$$

Din nou bitul de semn a rezultat 1, deci rezultatul este negativ si pentru a verifica modulul complementam rezultatul fata de 2 si obtinem 01100010 care este reprezentarea in format binar a numarului 98 obtinuta la prima adunare.

Adunarile prezentate pana aici nu au condus la depasire de format. Si problema depasirii de format se rezolva relativ simplu in cazul utilizarii reprezentarii in complement fata de 2. Astfel, daca se noteaza cu T_{s6} bitul de transport de la rangul 6 spre rangul 7 al operanzilor si cu T_{s7} bitul de transport de la rangul 8 spre octetul superior, atunci se constata ca apare depasire de format in cazul adunarilor in care T_{s6} difera de T_{s7}. Daca se face aceasta verificare pentru adunarile de mai sus se constata ca de fiecare data $T_{s6} = T_{s7}$, asa cum este pus in evidenta la fiecare dintre adunari.

Pentru adunarea 75+67 se obtine

$$\begin{array}{r} 75 \rightarrow 01001011 + \\ 67 \rightarrow 01000011 \\ \hline 10001110 \end{array} \qquad T_{s6} = 1 \text{ , } T_{s7} = 0.$$

Bitul de semn a rezultat 1, deci ar insemna ca rezultatul este negativ, ceea ce este absurd, dar deoarece $T_{s6} \neq T_{s7}$ inseamna ca a aparut o depasire de format. Intr-adevar 75+67=142, care nu se poate reprezenta in format 1+7. Bitul cel mai semnificativ trebuie sa ramana rezervat ca bit de semn.

Pentru adunarea –75+ (-67) se obtine

$$\begin{array}{r} -75 \rightarrow 10110101 + \\ -67 \rightarrow 10111101 \\ \hline 01110010 \end{array} \qquad T_{s6} = 0 \text{ , } T_{s7} = 1.$$

De data aceasta bitul de semn a rezultat 0, ceea ce ar insemna ca rezultatul este pozitiv, ceea ce este din nou fals. Dar din nou $T_{s6} \neq T_{s7}$ deci a aparut o depasire de format la adunarea a doua numere negative.

Se observa ca prin utilizarea unei singure porti “NAND” intre T_{s6} si T_{s7} se poate sesiza daca a aparut depasirea de format, indiferent de semnul celor doi operanzi.

Deci, in cazul utilizarii reprezentarii in cod complementar, toti bitii operanzilor se trateaza la fel, operatiile de adunare se fac si asupra bitilor de semn. Rezultatul apare corect in cod complementar daca $T_{s6} = T_{s7}$ si este afectat de eroare de depasire de format daca $T_{s6} \neq T_{s7}$.

In concluzie, daca se utilizeaza reprezentarea in cod complementar, operatia de adunare necesita o configuratie hardware mult mai simpla decat in cazul in care se utilizeaza reprezentarea in cod direct. Adunarea se poate face in mod unitar, atat pentru operanzi pozitivi cat si negativi pe aceleasi circuite in cazul reprezentarii in cod complementar. In plus si depasirile de format se trateaza in mod unitar atat pentru operanzi pozitivi cat si pentru operanzi negativi.

Alt avantaj in cazul utilizarii reprezentarii in cod complementar a numerelor este ca operatia de scadere poate fi redusa la operatia de adunare a descazutului cu complementul fata de 2 al scazatorului. Deci nu avem nevoie in plus fata de sumator decat de un circuit care sa complementeze fata de 2 scazatorul. Pentru calculul complementului fata de 2 al scazatorului se poate face complementarea acestuia fata de 1, dupa care se aduna 1 la rangul cel mai mic. Complementarea fata de 1 se face foarte usor cu porti “NOT”, astfel incat pentru realizarea operatiei de scadere se poate folosi configuratia din figura 6.

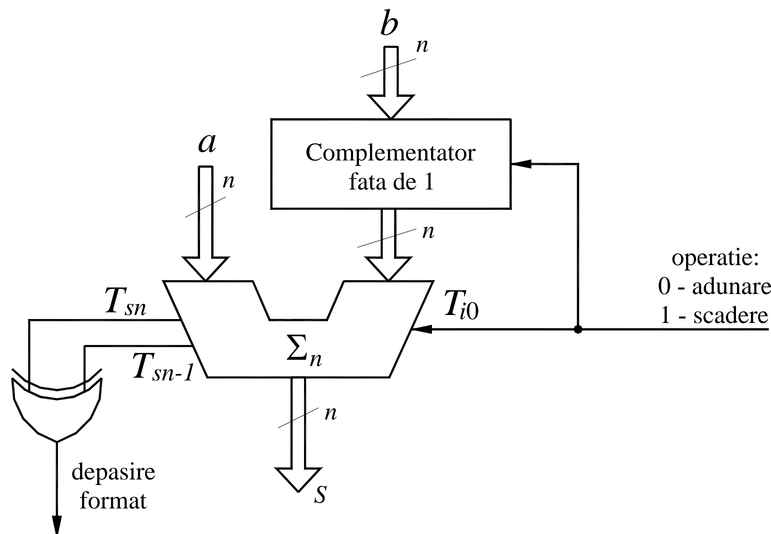


Fig.6 – Sumator-scazator in cod complementar

In figura 6, linia care bareaza sagetile operanzilor, insotita de numarul *n* semnifica faptul ca operanzii sunt vehiculati pe o magistrala paralela cu latimea *n*, cat este lungimea operanzilor.

Blocul complementator fata de 1 este comandat prin bitul de selectie a operatiei executate. Daca acest bit este 0 atunci operandul *b* trece necomplementat si ca urmare operatia executata va fi adunarea;daca este 1, atunci acest bloc va face complementarea fata de 1. Dar pentru a obtine complementul fata de 2 al operandului *b,* in vederea executarii scaderii mai trebuie adaugat 1 la rangul cel mai mic. Acest 1 se adauga direct in blocul sumator-scazator sub forma transportului de la rangul inferior T_{i0}. Bitul de selectie al operatiei pentru scadere este 1 si deci exact cat trebuie pentru ca per total sa se obtina complementul fata de 2 al operandului *b* in vederea realizarii operatiei de scadere *a-b.* In cazuloperatiei de adunare bitul de selectie fiind 0, T_{i0} va fi si el 0 si ca urmare adunarea se va face corect.

La iesire se obtine suma *S* (sau daca s-a selectat operatia de scadere diferenta), si prin compararea bitilor de transport T_{sn-1} si T_{sn} cu ajutorul unui circuit XOR se va obtine si indicatorul (flagul) de depasire de format.

Se observa deci, ca utilizand reprezentarea in cod complementar se obtine o configuratie foarte simpla a sumator-scazatorului. Pe acelasi circuit se pot face atat operatia de adunare cat si cea de scadere cu modificari minime fata de un sumator pe *n* biti.

Am prezentat pana aici operatiile de adunare si scadere a numerelor intregi pe *n* biti. Pentru operatiile in virgula mobila lucrurile se complica intrucat inainte de a face adunarea mantiselor trebuie ca operanzii sa fie adusi la acelasi exponent. Evident ca operanzii vor fi adusi amandoi la exponentul celui mai mare dintre ei. In acest scop, operandul mantisa operandului mai mic se deplaseaza spre dreapta cu un numar de ranguri egal cu diferenta exponentilor. Operatia de adunare (scadere) revine in continuare la adunarea (scaderea) mantiselor. Mai jos este prezentat un exemplu de adunare a doua numere in virgula mobila in format 1+8+23, deci in simpla precizie, considerand excesul 122 si reprezentarea in mantisa normalizata. De asemenea s-a tinut cont de faptul ca in format apare reprezentat signifiantul si nu mantisa.

Operand	Valoare zecimala	Valoare binara	Exponent	Caracteristica	Reprezentare in simpla precizie
a	8,125	1000,001	4	126	0|01111110|000001000….0
b	0,0625	0,0001	-3	119	0|01110111|000000000….0
a+b	8,1875	1000,0011	4	126	0|01111110|000001100….0

Exponentul comun cel mai mare este 4, ca urmare mantisa normalizata a celui de-al doilea operand se deplaseaza cu 7 biti spre dreapta. Se obtin mantisele

$$m_a = 0,100000100\ldots0+$$
$$\underline{m_b = 0,000000010\ldots0}$$
$$m_{a+b} = 0,100000110\ldots0$$

Exponentul rezultatului fiind 4, caracteristica este tot 126, astfel incat reprezentarea rezultatului in simpla precizie este

$$S \to 0\,|\,01111110\,|\,00000110...0$$

Un avantaj important al lucrului in virgula mobila este acela ca in cazul in care apare depasire de format la adunarea mantiselor se poate ajusta exponentul rezultatului astfel incat sa poata fi reprezentat in formatul impus. Evident ca nu trebuie sa apara si depasire de format in cazul ajustarii exponentului.

Pentru operatia de scadere in virgula mobila procedeul este acelasi dar se executa scaderea mantiselor nu adunarea lor.

Circuitele electronice necesare realizarii operatiilor in virgula mobila sunt mai complicate intrucat trebuie sa permita compararea exponentilor si ajustarea acestora, insotita de deplasarea corespunzatoare a mantisei.

Inmultirea numerelor

Pentru a realiza inmultirea numerelor cel mai simplu algortim este cel de tip inmultire succesiva cu cate o cifra – deplasare – adunare. Avantajul reprezentarii binare este ca pot apare doar inmultiri cu 0, caz in care rezultatul este 0 sau inmultire cu 1 cand rezultatul este chiar numarul initial. Acest algoritm este sintetizat mai jos pentru inmultirea numerelor 15 si 9 reprezentate ca numere intregi pe 7 biti.

```
       15 → 0001111 x
         9 → 0001001
       ----------------
              0001111
             0000000
            0000000
           0001111
          0000000
         0000000
        0000000       .
135 → 00000010000111
```

Se observa faptul ca rezultatul inmultirii este reprezentat pe 14 biti (dublul lungimii operanzilor). Daca se doreste ca rezultatul sa fie reprezentat tot pe 7 biti, atunci la inmultire apare foarte frecvent depasire de format, de care trebuie sa tina cont programatorul. In calculator insa registrul in care se stocheaza rezultatul inmultirii are lungime dubla fata de operanzi, si programatorul este cel care gestioneaza cum se va prelucra mai departe rezultatul. Lungimea dubla a registrului in care se stocheaza rezultatul asigura faptul ca niciodata la inmultire nu va apare depasire de format.

In implementarea circuitelor pentru inmultire nu se urmeaza intocmai algortimul prezentat mai sus in sensul ca inmultirea cu zero nu se executa propriu-zis ci pur si simplu se realizeaza o deplasare spre dreapta a registrului rezultatului. Incarcarea deinmultitului se face pe bitii cei mai semnificativi ai registrului - rezultat si apoi succesiv se deplaseaza spre dreapta acest registru si se aduna la bitii cei mai semnificativi deinmultitul daca cifra din inmultitor este 1. Daca cifra din inmultitor este 0 se face numai deplasarea, fara a mai aduna nimic. Pentru inmultirea de mai sus procedeul este sintetizat in continuare:

- Se incarca deinmultitul pe bitii cei mai semnificativi ai rezultatului:

  0001111 0000000

- Ultima cifra a inmultitorului este 1 astfel incat registrul sufera doar o deplasare catre dreapta

  0000111 1000000

- Urmatoarele doua cifre ale inmultitorului sunt 0, deci registrul rezultat se deplaseaza cu doua pozitii spre dreapta fara a se aduna nimic

  0000001 1110000

- Urmatoarea cifra este 1 deci se aduna deinmultitul si se deplaseaza rezultatul cu o pozitie spre dreapta. Se obtine

0000001 1110000+
0001111 .
0010000 1110000 → 0001000 0111000

- Urmatoarele trei cifre ale inmultitorului sunt 0, astfel incat registrul rezultat se deplaseaza spre dreapta cu 3 pozitii fara a mai aduna nimic, deci se obtine

0000001 0000111.

O structura hardware care implementeaza algoritmul de inmultire de mai sus este prezentata in figura 7.

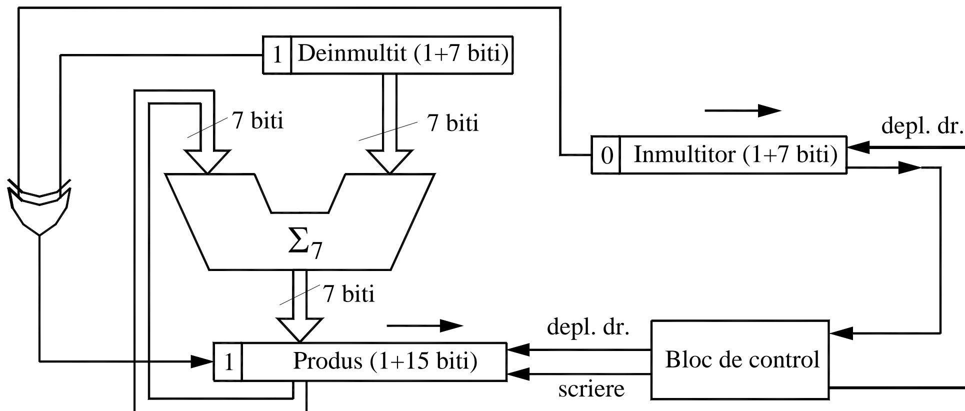


Fig. 7 – Structura hardware a unui bloc de inmultire

Deinmultitul ocupa pe tot parcursul operatiei resistrul cu acelasi nume care are 1+7 biti si nu sufera modificari. Inmultitorul este stocat in registrul corespunzator, tot pe 1+7 biti, dar sufera deplasari succesive spre dreapta. Blocul de control citeste din registrul inmultitor ultima cifra a acestuia si in functie de valoarea ei comanda fie numai deplasarea registrului produs spre dreapta fie adunarea jumatatii superioare a registrului produs cu deinmultitul, stocarea rezultatului tot pe jumatatea superioara a registrului produs si apoi deplasarea spre dreapta a registrului produs.Bitul de semn nu sufera deplasari. Rezultatul se obtine pe 1+15 biti. Bitul de semn rezulta prin aplicarea functiei XOR bitilor de semn ai operanzilor.

Prin aceasta metoda se inmultesc modulele numerelor, ca urmare aceste trebuie reprezentate in cod direct. Daca unul din numere este negativ si se foloseste reprezentarea in cod direct, atunci se pot inmulti direct modulele iar bitul de semn rezulta din bitii de semn ai operanzilor prin functia "XOR". Daca se foloseste reprezentarea in cod complementar, atunci trebuie calculat mai intai bitul de semn utilizand functia "XOR", apoi daca exista operanzi negativi se complementeaza fata de 2 modulele acestora si in final se aplica algoritmul de inmultire prezentat. Daca bitul de semn a rezultat 1 (rezultatul este negativ) atunci in final, modulul rezultat mai trebuie complementat odata fata de 2 astfel incat sa se obtina reprezentarea corecta in complement fata de 2.

Se observa ca algoritmul adunare-deplasare se aplica mai simplu in cazul reprezentarii numerelor in cod direct. Utilizarea acestui algoritm in cazul reprezentarii in cod complementar aduce complicatii si deci se va complica si implementarea schemei electronice. Pentru a evita acest lucru s-a pus la punct un algoritm ce tine cont ca in reprezentarea in cod complementar bitul de semn are pondere negativa (vezi relatia 1).

Considerand inmultitorul reprezentat pe n biti din care bitul cel mai semnificativ este bitul de semn, atunci acesta se poate scrie in cod complementar

$$b = -b_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-1} b_i \cdot 2^i \qquad (16)$$

in care bitul b_{n-1} este bitul de semn. Tinand cont ca

$$b_k \cdot 2^k = 2 \cdot b_k \cdot 2^k - b_k \cdot 2^k = b_k \cdot 2^{k+1} - b_k \cdot 2^k \tag{17}$$

inmultitorul se poate pune sub forma

$$b = \sum_{i=0}^{n-1} (b_{i-1} - b_i) \cdot 2^i \;. \tag{18}$$

Pentru i=0 in relatia (18) apare si b_{-1} care trebuie considerat 0.

Utilizand scrierea inmultitorului in forma (18) se poate deduce urmatorul algoritm pentru inmultirea a doua numere (algoritmul Booth):

- Se incarca deinmultitul in registrul rezultat
- Daca $b_{i-1} = b_i$ se deplaseaza registrul rezultat spre dreapta fara a aduna sau scadea nimic
- Daca b_{i-1} =1 si b_i = 0 se deplaseaza registrul rezultat spre dreapta si se aduna deinmultitul pe rangurile superioare, la fel ca la algoritmul adunare-deplasare
- Daca b_{i-1} =0 si b_i = 1 se deplaseaza registrul rezultat spre dreapta si se scade deinmultitul pe rangurile superioare, analog cazului precedent.

In cazul utilizarii algoritmului Booth nu are importanta semnul operanzilor, rezultatul fiind corect pentru operanzi reprezentati in cod complementar. Implementarea electronica a algoritmului este mult mai simpla decat pentru algoritmul adunare-deplasare.

In cazul inmultirii numerelor in virgula mobila, bitul de semn rezulta de asemenea prin operatia "XOR" intre bitii de semna ai operanzilor, exponentul va fi suma exponentilor iar mantisa va rezulta din produsul mantiselor. Este posibil ca mantisa sa rezulte nenormalizata, astfel incat va fi necesara normalizarea mantisei si ajustarea exponentului. Pentru reprezentarea in cod complementar, inmultirea mantiselor se face utilizand algoritmul Booth prezentat anterior, in care se tine cont si de bitii de semn.

Impartirea numerelor

Ca si in cazul inmultirii, si algoritmul de impartire a numerelor urmeaza algoritmul elementar de impartire, dar implementat in baza 2. In cazul impartirii trebuie facuta insa mai intai verificarea ca impartitorul este diferit de zero. Bitul de semn rezulta ca si in cazul inmultirii prin functia XOR aplicata bitilor de semn ai operanzilor. Vom prezenta in continuare impartirea a doua numere pozitive. Daca reprezentarea este facuta in cod direct atunci algoritmul este acelasi chiar daca cel putin unul din numere este negativ. Daca reprezentarea este in cod complementar, atunci dupa calcularea bitului de semn pentru ambii operanzi trebuie calculat modulul si facuta impartirea modulelor. Daca bitul de semn a rezultat 1, dupa impartire trebuie complementat fata de 2 modulul catului si corectat restul.

Pentru doua numere pozitive vom exemplifica algortimul prin impartirea 87:6 exprimat in baza 2.

Daca deimpartitul este mai mare decat impartitorul prima operatie care se face este alinierea deimpartitului si a impartitorului astfel incat primele lor cifre sa fie 1. Primele *n-k-1 c*ifre ale catului vor fi 0, unde *n* este numarul de biti pe care sunt reprezentate numerele si *k* este numarul de cifre cu care se deplaseaza spre stanga impartitorul. In acest caz se deplaseaza impartitorul cu patru pozitii spre stanga. Deci primele doua cifre ale catului sunt 0. Urmeaza un numar de k+1 operatii de comparatie urmate de scaderile corespunzatoare. Se compara deimpartitul cu impartitorul. Daca este

mai mic urmatoarea cifra a catului este 0 si se scade din deimpartit impartitorul inmultit cu 0. Se deplaseaza spre stanga rezultatul cu o pozitie, completand cu 0 rangul inferior. Se compara noul deimpartit cu impartitorul.

$$87 \rightarrow 1010111$$
$$6 \rightarrow 0000110$$

```
1010111    |1100000
0000000    |0001110=14
10101110
 1100000
 10011100
  1100000
  01111000
   1100000
   00110000
   00000000
    0110000 → 0000011 = 3
```

De data aceasta este mai mare, deci urmatoarea cifra a catului este 1 si se scade din actualul deimpartit impartitorul. Urmeaza o noua deplasare spre stanga a deimpartitului rezultat apoi o noua comparatie urmata de scadere, s.a.m.d. pana cand se completeaza cele k+1 (in cazul de fata 5) pozitii ale catului. Dupa ultima scadere pe pozitia deimpartitului se obtine restul, dar trebuie tinut cont de deplasarea spre stanga facuta la inceput cu k pozitii, astfel incat restul trebuie deplasat spre dreapta cu 4 pozitii. Se obtine astfel rezultatul prezentat

87:6=14 rest 3

Algoritmul se implementeaza utilizand registre de deplasare si registre contor care numara cate deplasari s-au facut in momentul initial. Registrul deimpartitului trebuie sa aiba cu un rang mai mult decat lungimea de reprezentare a modulelor numerelor pentru a nu apare depasire de format la operatia de deplasare spre stanga.

Comparatia a doua numere a si b in calculator se face calculand diferenta a-b si apoi urmarind bitul de semn. Daca bitul de semn rezulta 1 atunci $a<b$, daca rezulta 0 atunci $a \geq b$. Comparatia dintre deimpartit si impartitor presupune deci efectuarea unei scaderi. In consecinta, algoritmul de impartire poate fi pus si sub forma urmatoare, care poarta denumirea de algoritmul cu refacerea restului [Patterson si Hennessy]. Schema hardware de implementare a acestui algoritm este prezentata in figura 7.

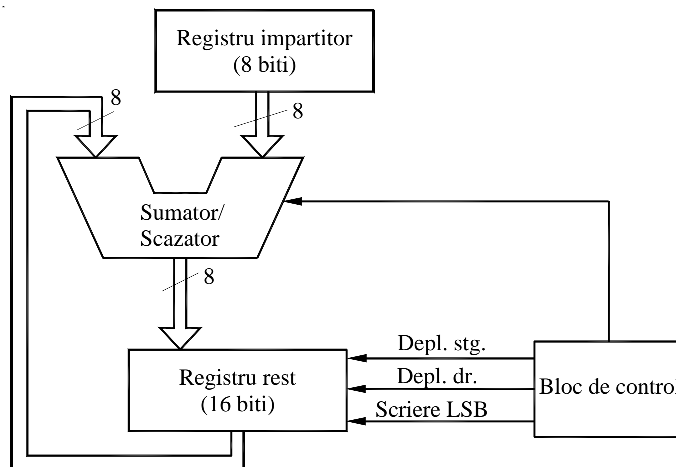


Fig. 8 – Structura hardware pentru algoritmul de impartire cu refacerea restului

Registrul rest este un registru de lungime dubla fata de lungimea operanzilor, pentru exemplul nostru un registru pe 16 biti, din care primul bit este bit de semn. In jumatatea inferioara se stocheaza deimpartitul, iar jumatatea superioara se incarca initial cu 0. Jumatatea superioara va contine in final restul iar jumatatea inferioara catul. Impartitorul este stocat pe toata perioada operatiei in registrul impartitor. Blocul sumator/scazator face diferenta sau suma intre jumatatea superioara a registrului rest si registrul impartitor. Registrul rest poate realiza deplasari spre stanga, bitul cel mai putin semnificativ fiind completat cu 0 sau 1 in functie de semnul rezultatului scaderii dintre jumatatea superioara a registrului rest si registrul impartitor. Jumatatea superioara a registrului rest poate realiza si deplasari spre dreapta cu completarea cu 0 a MSB.

Algoritmul decurge in modul urmator:

- se incarca registrul rest cu 0 in jumatatea superioara si cu deimpartitul in jumatatea inferioara;
- se incarca impartitorul in registrul impartitor;
- se deplaseaza spre stanga registrul rest pentru a tine cont ca primul bit din jumatatea inferioara a fost bitul de semn al deimpartitului (am considerat aici operanzi pozitivi);
- se scade impartitorul din jumatatea superioara a registrului rest. Daca bitul de semn rezulta negativ atunci inseamna ca deimpartitul este mai mic decat impartitorul si cifra catului rezultata la acest pas este zero. Trebuie refacut in aceasta situatie deimpartitul astfel incat se aduna impartitorul la rezultatul din jumatatea superioara a registrului rest, pentru a reface deimpartitul. Se deplaseaza spre stanga registrul rest si se completeaza bitul cel mai putin semnificativ cu 0. Daca bitul de semn rezulta 0 deimpartitul a fost mai mare decat impartitorul si ca urmare cifra catului rezultata la acest pas este 1. Nu e mai reface deimpartitul ci doar se deplaseaza registrul rest spre stanga si se completeaza bitul cel mai putin semnificativ cu 1.
- Se repeta operatiile de la pasul anterior de un numar de ori egal cu lungimea operanzilor, in cazul de fata 8 ori.

In final, asa cum am mentionat, jumatatea inferioara a registrului rest va contine catul si jumatatea superioara ca contine restul. Avantajul acestei metode este acela ca obtine direct si eventualii biti 0 de pe pozitiile cele mai semnificative ale catului, care in cazul anterior erau obtinuti in urma operatiilor aliniere dintre deimpartit si impartitor. De asemenea restul rezulta direct aliniat, fara a mai fi nevoie de operatii de deplasari suplimentare la dreapta pentru aliniere.

Pentru cazul impartirii anterioare, acest algoritm este sintetizat in tabelul 3.

Tabelul 3. – Algoritmul de impartire exemplificat pentru operatia 87:6

Iteratie	Operatii	Registrul rest	Registrul impartitor
0	Incarcare valori initiale	00000000 01010111	00000110
	Deplasare registru rest spre stanga	00000000 10101110	00000110
1	Se scade registrul impartitor din jumatatea superioara a registrului rest	11111010 10101110	00000110
	Bitul de semn 1 – se reface deimpartitul si se deplaseaza registrul rest spre stanga completand LSB cu 0	00000001 01011100	00000110
2	Se scade registrul impartitor din jumatatea superioara a registrului rest	11111011 01011100	00000110
	Analog iteratia 1	00000010 10111000	00000110
3	Se face scaderea	11111100 10111000	00000110
	Analog iteratia 1	00000101 01110000	00000110
4	Se face scaderea	11111001 01110000	00000110

	Analog iteratia 1	00001010 11100000	00000110
5	Se face scaderea	00000100 11100000	00000110
	Bitul de semn 0 – nu se mai reface restul ci doar se deplaseaza registrul rest spre stanga si se completeaza LSB cu 1	00001001 11000001	00000110
6	Se face scaderea	00000011 10000001	00000110
	Analog iteratia 6	00000111 00000011	00000110
7	Se face scaderea	00000001 10000011	00000110
	Analog iteratia 6	00000011 00000111	00000110
8	Se face scaderea	11111101 00000011	00000110
	Analog iteratia 1	00000110 00001110	00000110
	Deplasare jumatate superioara a restului spre dreapta cu o pozitie	00000011 00001110	00000110

Se observa ca la ultima iteratie, dupa deplasarea spre stanga a intregului registru rest spre stanga si completarea LSB cu 0, se face o deplasare dreapta cu o pozitie a jumatatii superioare a registrului rest astfel incat sa se tina cont de faptul ca impartitorul a fost mai mare decat deimpartitul si practic trebuie sa se revina la valoarea deimpartitului care acum este restul final al impartirii.

In cazul lucrului in virgula mobila se procedeaza analog cu cazul inmultirii: bitul de semn rezulta aplicand functia XOR intre bitii de semn ai operanzilor, exponentul este diferenta dintre exponentii deimpartitului si impartitorului si mantisa rezultatului se obtine prin impartirea mantiselor. Inainte de a realiza operatia trebuie facuta si aici verificarea ca impartitorul nu este 0. In finalul operatiei se face daca este necesar normalizarea mantisei.

ARHITECTURA CALCULATOARELOR

Asa cum s-a mentionat introducere, notiunea de arhitectura calculatoarelor are in vedere unitatile functionale ale acestuia care sunt accesibile utilizatorului. In cadrul acestei discipline se studiaza blocurile functionale ale calculatorului si modalitatile de interconectare a lor, din punctul de vedere al utilizatorului, fara a intra in detaliile ce privesc implementarea electronica a blocurilor respective si nici chiar modalitatile de realizare a unor operatii cum ar fi transferul de date intre blocuri, realizarea operatiilor aritmetice si logice, etc. Modalitatea de implementare a acestor operatii il intereseaza mai putin pe utilizatorul obisnuit, el fiind interesat de rezultatele acestor operatii.

Blocurile functionale ale calculatorului au rezultat in urma definirii unor principii care au stat la baza constructiei calculatoarelor. Aceste principii au fost elaborate pe baza scopului in care a fost realizat calculatorul, si anume prelucrarea rapida a unor informatii. La inceput, aceasta prelucrare viza in principal efectuarea rapida a unor calcule matematice. Informatiile ce urmeaza a fi prelucrate trebuie introduse in calculator, iar dupa obtinerea rezultatelor, informatia utila trebuie prezentata utilizatorului. John von Neumann in anul 1945 a elaborat principiile care stau la baza prelucrarii informatiei in interiorul calculatorului. Pe baza acestor principii a rezultat tipul de arhitectura care ii poarta numele. Mult timp aceasta a fost principala arhitectura de calculatoare. Principiile arhitecturii **von Neumann** sunt urmatoarele [Mancas]:

- *informatia in calculatorul digital este codificata binar;*
- *diferentierea intre cuvantul de date si cuvantul instructiune se face prin contextul de utilizare;*
- *cuvintele de date si de comanda sunt depuse in locatii de memorie care se identifica printr-un numar numit **adresa**;*
- *Algoritmul in executie se reprezinta sub forma unei succesiuni de cuvinte de comanda denumite **instructiuni;***

- *Efectuarea calculelor corespunzand algoritmului problemei de rezolvat este univoc determinata de executia secventiala a instructiunilor.*

Primul principiu conform caruia informatia este codificata binar in calculator a aparut ca urmare a nivelului tehnologic al momentului respectiv. Pentru a codifica o informatie trebuie sa existe un set de simboluri cu care sa se faca aceasta codificare. Cel mai simplu set de simboluri este compus din 0 si 1 iar acestea se pot reprezenta foarte usor la nivelul circuitelor electronice prin prezenta sau absenta unei tensiuni. Acest nivel tehnologic nu a fost depasit nici pana astazi, desi a suferit o evolutie extraordinara.

Cel de-al doilea principiu a fost enuntat datorita faptului ca informatia in calculator este de mai multe tipuri. Exista informatia pe care vrea sa o prelucreze utilizatorul, exista modul in care vrea sa prelucreze utilizatorul aceasta informatie, care este tot un tip de informatie, si mai exista informatia pe care o vehiculeaza calculatorul la nivel fizic referitor la starea circuitelor sale si modul cum face utilizarea acestor circuite. Primele doua tipuri de informatie trebuie communicate calculatorului de catre utilizator si vor fi stocate in codificare binara, in conformitate cu primul principiu. Daca este "privita" informatia stocata ea este constituita doar din 0 si 1 si nu se poate spune la o prima "privire" ce tip de informatie este: informatia care se prelucreaza sau modul de prelucrare a acesteia. In functie de modul cum este aranjata aceasta informatie rezulta la "citire", din succesiunea cuvintelor, daca este un tip de informatie sau altul. Acest principiu simplifica foarte mult sarcina constructorului de calculatoare, dar o complica pe cea a utilizatorului, care trebuie sa structureze intr-un format bine precizat informatiile introduse in calculator.

Al treilea principiu enuntat de **von Neumann** se refera la ceea ce se intampla cu informatia introdusa in calculator, atunci cand aceasta nu sufera transformari propriu-zise. Evident ca aceasta informatie se stocheaza. Unitatea de stocare a informatiei poarta numele de **memorie.** Aceasta va stoca atat informatia care urmeaza a fi prelucrata, cat si modul de prelucrare a acesteia, adica **programul.** In cadrul arhitecturii von Neumann, atat programul cat si datele se stocheaza in aceeasi unitate de stocare (memorie). Ulterior, au aparut si alte arhitecturi care stocheaza in memorii diferite datele si programul (arhitectura Harvard). Locatiile de memorie sunt "numerotate" pentru a se putea sti unde a fost stocata o anumita informatie. "Numarul" locatiei de memorie poarta numele de adresa.

Al patrulea principiu se refera la modalitatea de specificare a modului de prelucrare a datelor, adica a **programului.** De la bun inceput, constructia calculatorului a urmarit obtinerea unei flexibilitati cat mai mari in prelucrarea informatiei. Pentru aceasta calculatorul dispune de un set de operatii elementare, care pot fi combinate astfel incat sa se obtina rezultatul dorit. Specificarea operatiei care se va executa la un moment dat se face prin intermediul unor cuvinte de comanda, codificate tot in format binar care poarta numele de **instructiuni.** Acestea se stocheaza impreuna cu datele de prelucrat si cu rezultatele in aceeasi memorie. Din combinarea intr-o succesiune logica a instructiunilor rezulta programul.

Al cincelea principiu asigura faptul ca dintr-un set de date de intrare se va obtine in mod univoc un set de date de iesire, fara a exista ambiguitati. Rezultatele obtinute decurg in mod necesar din succesiunea de instructiuni executata, succesiune care este cea specificata de utilizator.

Aceste principii prezentate au condus la obtinerea unor masinarii (calculatoarele) cu o foarte larga aplicabilitate practica si sunt in esenta principiile care guverneaza si astazi proiectarea si constructia calculatoarelor, chiar daca ele au devenit foarte sofisticate de-a lungul timpului.

Arhitectura elementara a calculatoarului

Am vazut anterior ca informatia cu care interactioneaza calculatorul sufera urmatoarele operatii: este introdusa in calculator, este stocata in memorie, este prelucrata, este prezentata utilizatorului si eventual rezultatele sunt stocate in calculator. Avand in vedere ca trebuie sa existe o unitate de coordonare a activitatii calculatorului, care poarta numele de unitate de control, rezulta o prima schema, foarte generala care reflecta arhitectura unui calculator, prezentata in figura 9.

Informatia este preluata din exterior prin intermediul unitatii de intrare-iesire. Prin aceste unitati de intrare-iesire care pot fi una sau mai multe in cadrul unui calculator sunt preluate atat datele care urmeaza a fi prelucrate cat si programele.

Datele preluate din exterior pot ajunge fie la Unitatea aritmetico-logica, pentru o prelucrare

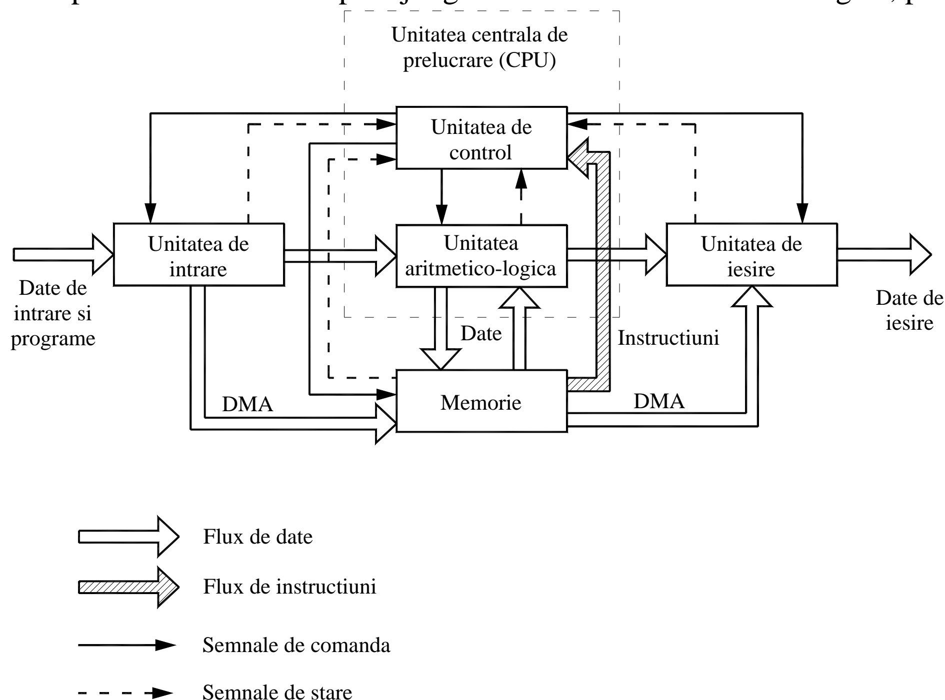


Fig. 9 – Arhitectura generala a unui calculator

imediata, fie pot fi stocate in memorie pentru o prelucrare ulterioara. Stocare in memorie se face pe o cale separata, astfel incat sa nu fie implicata unitatea aritmetico-logica (ALU). Aceasta cale poarta numele de acces direct la memorie si conduce la un castig important de timp deoarece in timpul transferului de date unitatea aritmetico-logica (ALU) poate realiza prelucrarea altor date.Un transfer de date de tip acces direct la memorie poate fi realizat si intre memorie si unitatea de iesire.

Datele care urmeaza a fi prelucrate de unitatea aritmetico-logica pot fi extrase si din memorie, unde au fost in prealabil stocate.

Instructiunile care urmeaza a fi executate sunt extrase din memorie de catre unitatea de control, decodificate si apoi executate. Transferul atat al datelor cat si al instructiunilor in interiorul calculatorului se face de obicei pe magistrale paralele, din aceasta cauza fluxurile da date si de instructiuni sunt figurate cu sageti ingrosate.

Unitatea de control primeste informatii de la toate componentele prin semnalele de stare corespunzatoare si trimite comenzi catre acestea realizand astfel coordonarea functionarii calculatorului. In prezent, unitatea de control de regula se afla pe acelasi chip cu unitatea aritmetico-logica si formeaza unitatea centrala de prelucrare (UCP). Cele doua sunt incluse in cadrul microprocesorului.

Memoria realizeaza stocarea temporara a datelor. Este compusa asa cum am vazut din unitati adresabile de memorie, care cel mai frecvent in prezent sunt compuse din 8 biti (1 octet sau 1 byte), dar pot fi structurate si in alte formate, de exemplu 1, 16, 24, 32, 64 biti, etc. Fiecare unitate adresabila de memorie este identificata prin **adresa.** Numarul total de adrese dintr-o unitate de memorie formeaza dimensiunea acelei memorii. Pe langa dimensiunea memoriei un alt parametru important care o caracterizeaza este timpul de acces. Acesta depinde in principal de distanta la care se afla de unitatea centrala de prelucrare (UCP) si de tehnologia utilizata pentru realizarea memoriei. Mai aproape de UCP sunt plasate memorii de dimensiuni mai reduse, dar foarte rapide, pe cand la distante mai mari sunt plasate memorii mai lente dar care pot ajunge la dimensiuni foarte mari. Mediile pe care se stocheaza informatia in exteriorul calculatorului au evoluat in timp de la

cartele perforate, la benzi magnetice, discuri magnetice, discuri optice etc. In interiorul calculatorului informatia se stocheaza de regula pe diferite dipuri de circuite electronice sau pe unitati de stocare magnetice. In prezent, unitatile magnetice de memorie interna nu mai sunt utilizate.

Dupa distanta la care se afla de UCP si viteza de acces, memoria se imparte **in memorie principala** si **memorie secundara**. Memoria secundara este mai lenta si aflata la distanta mare de UCP, dar poate realiza o stocare permanenta a informatiei (chiar dupa oprirea calculatorului). Poate fi amplasata in interiorul calculatorului sau in exteriorul acestuia. Memoria principala se afla aproape de UCP, este mai rapida, dar dependenta de alimentarea calculatorului. La oprirea calculatorului informatia din aceasta memorie se pierde.

Pentru a creste si mai mult viteza de prelucrare a datelor memoria principala a fost impartita in **memorie operativa,** care se afla pe placa de baza, in apropierea microprocesorului, si **memorie superoperativa** plasata chiar in interiorul procesorului. Informatia accesata foarte des de procesor este stocata in memoria superoperativa, pe cand informatia accesat mai rar este plasata in memoria operativa. Memoria superoperativa are dimensiuni considerabil mai mici decat memoria operativa. Memoria operativa in prezent ajunge in mod uzual la cativa GB, pe cand memoria superoperativa este de ordinul a 256 KB, 512 KB, 1 MB sau 4 MB. Pentru utilizarea memoriei superoperative au fost dezvoltati algoritmi care sa identifice informatia utilizata cel mai des de UCP si sa o aduca in aceasta memorie. Memoria superoperativa mai poarta numele si de **memorie cache.**

Unitatea centrala de prelucrare (UCP)

Unitatea centrala de prelucrare, sau microprocesorul, are o structura interna care sa-i permita preluarea datelor din memorie (sau de la unitatea de intrare), prelucrarea datelor si transmiterea acestora catre memorie sau catre unitatea de iesire. O prima schema structurala a unitatii centrale de prelucrare, care nu surprinde toate componentele acesteia este prezentata in figura 10. Aceasta schema reflecta principalele cai de vehiculare a datelor, dar nu si structura de control.

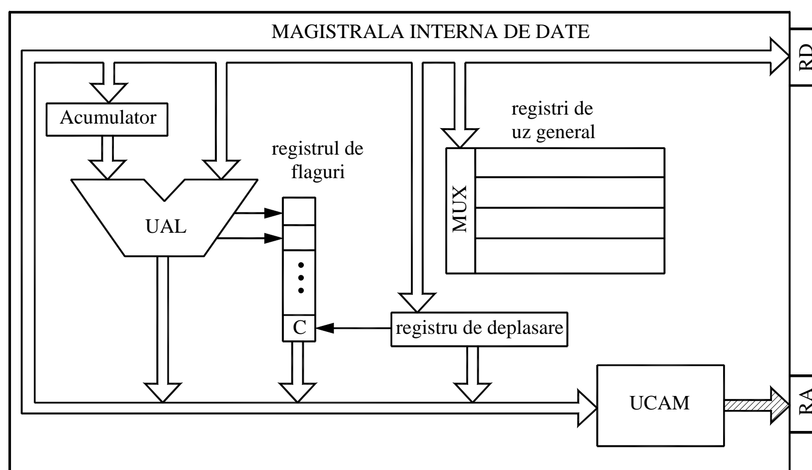


Fig. 10 – Structura simplificata a UCP

UCP-ul comunica cu restul calculatorului prin intermediul a doua registre: Registrul de date (RD) si registrul de adrese (RA). Datele sunt primite din exterior in registrul de date. Odata ajunse aici ele sunt disponibile pe magistrala interna de date si sunt transmise la unitatile functionale ale UCP. Rezultatele obtinute de UCP sunt plasate tot pe magistrala interna de date si ajung tot in RD, de unde vor fi accesibile prin intermediul magistralei externe de date celorlalte unitati componente ale calculatorului.

Pentru a identifica locatia de memorie de unde se citesc datele, sau in care vor fi scrise datele este utilizata unitatea de control al adresarii memoriei (UCAM). Aceasta calculeaza adresa si o plaseaza in RA. De aici, aceasta adresa merge mai departe pe magistrala externa de adrese si

identifica locatia de memorie necesara. Registrul de date este bidirectional. Pot fi primite date din exterior prin registrul de date sau pot fi transmise date catre exterior prin intermediul acestui registru. Registrul de adrese este unidirectional. Doar UCAM plaseaza adrese in RA, care mai departe identifica locatiile de memorie necesare.

Datele pot fi stocate temporar in interiorul UCP in asteptarea prelucrarii in unitati de memorie foarte rapid accesibile pentru UAL, denumite registri. A nu se confunda registrii cu memoria superoperativa. Registrii nu fac parte din memoria superoperativa. Exista registri de uz general in care pot fi stocati operanzi sau rezultate si pot fi utilizati in diverse moduri asa cum se va vedea in continuare. Mai exista doua registre speciale, utilizate de UAL in realizarea operatiilor propriu-zise. Unul din ele poarta numele de *acumulator* si stocheaza unul dintre operanzii folositi de UAL in operatiile realizate si in faza finala a executiei operatiei, rezultatul poate fi stocat tot in acumulator. Nu este obligatoriu ca rezultatul sa fie stocat in acumulator. Unele operatii folosesc in mod implicit ca operand continutul registrului acumulator.

Al doilea registru special care contine operanzi este registrul de deplasare. Acesta are posibilitatea de a executa deplasari stanga-dreapta ale operanzilor. Aceste deplasari sunt utile in anumite situatii. Un exemplu este realizarea rapida a inmultirii sau impartirii unui numar cu puteri ale lui 2. Registrul de deplasare comunica cu unul din bitii registrului de flaguri denumit *carry*. Flagul *carry* stocheaza in unele situatii bitul care iese din registrul de deplasare atunci cand se deplaseaza operandul spre stanga cu o pozitie. De asemenea mai poate stoca bitul de transport catre rangul superior in cazul adunarii sau scaderii a doi operanzi, operatie care se face in UAL. Utilizatorul are acces prin intermediul instructiunilor la registrele de uz general, acumulator si registrul de deplasare atat pentru citire cat si pentru scriere.

Un al treilea registru special, este registrul de flaguri. Acesta stocheaza un numar de biti care indica starea UAL dupa executia unei instructiuni sau stabilesc anumite regimuri de functionare ale procesorului. Intr-un paragraf ulterior se vor da cateva detalii privind continutul registrului de flaguri.

UAL reprezinta unitatea care efectueaza efectiv prelucrarea datelor in microprocesor. Poate executa instructiuni aritmetice, de tipul celor prezentate in paragraful anterior sau poate efectua operatii logice, precum si alte prelucrari de date. Structura sa poate fi foarte complicata la microprocesoarele moderne si nu intram in detalii aici.

Unitatea de control al adresarii memoriei calculeaza adresa la care se afla urmatoarea instructiune care urmeaza a fi executata sau adresa la care (de la care) se va scrie (citi) data accesata la pasul urmator. O structura simplificata a UCAM este prezentata in figura 11.

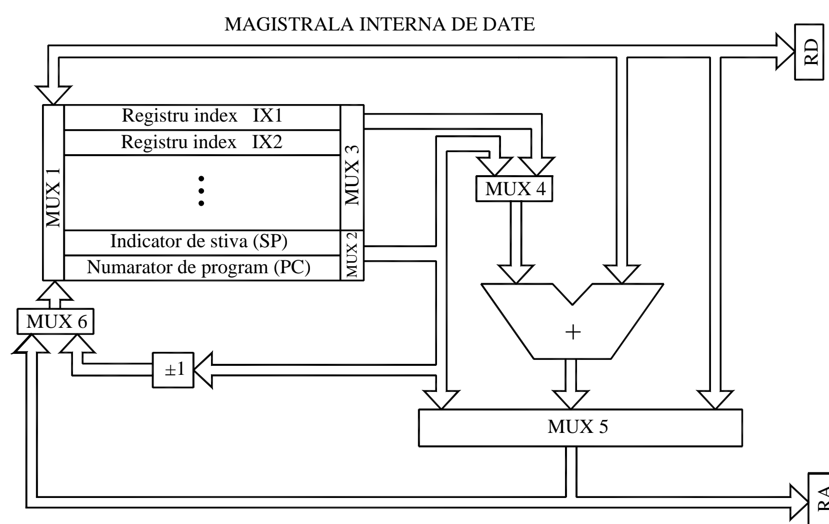


Fig. 11 – Structura simplificata a unitatii de control al adresarii memoriei

Figura 11 arata operatiile care se executa in cadrul UCAM si prefigureaza modurile de adresare a memoriei care vor fi prezentate intr-un paragraf ulterior. UCAM contine o serie de

registri care pot fi accesati pentru citire sau scriere prin intermediul unor multiplexoare si o unitate aritmetico-logica simplificata notata aici cu “+” care calculeaza adresa fizica. In cadrul UCAM exista un numar de registri index care contribuie la calculul adresei in cazul lucrului cu tablouri de date si doi registri foarte importanti denumiti *Indicatorul de stiva* (SP) si *Numaratorul de program* (PC) care uneori mai este notat cu IP (*instruction pointer*).

Numaratorul de program contine totdeauna adresa urmatoarei instructiuni care urmeaza a fi executata. Aceasta ajunge in registrul de adrese prin multiplexorul MUX 5. La inceputul executiei programului PC se incarca cu adresa primei intructiuni (instructiunea de start). Aceasta adresa este citita din memorie si este incarcata prin intermediul multiplexorului MUX1 de pe magistrala interna de date. In cazul in care programul se executa secvential, fara instructiuni de salt, pentru avansarea la urmatoarea instructiune se incrementeaza PC (cu ajutorul blocului ±1). In cazul unei instructiuni de salt, noua adresa poate fi calculata in mai multe moduri. Spre exemplu se insumeaza valoarea din PC cu un deplasament citit din memorie si primit pe magistrala interna de date, sau se poate face salt la o adresa primita direct pe magistrala interna de date. Odata calculata noua adresa, ea ajunge prin multiplexorul MUX5 in registrul de adrese, dar este si stocata in PC pentru a putea fi folosita in cazul in care se continua executia secventiala de la noua adresa.

Pentru accesarea datelor in structuri de tip tablou exista registrele index. Acestea stocheaza adreza bazei tabloului, iar pentru calculul adresei unui element al tabloului se incrementeaza registrul index cu o valoare calculata pe baza pozitiei elementului in tablou. Valoarea incrementului poarta numele de *deplasament*. Nu toate microprocesoarele dispun de registri index.

Indicatorul de stiva contine totdeauna adresa varfului stivei (ultimul element din stiva). Stiva este o structura de date de tip LIFO (ultimul intrat primul iesit – *last in first out*) care este utila in diferite situatii, printre care si cazul utilizarii programarii procedurale pentru apelarea functiilor. De obicei in aceasta structura de date se salveaza valoarea PC atunci cand apare apelul unei functii in programul principal. Pentru a se executa functia trebuie executat saltul la adresa de inceput a functiei, dupa care se executa secvential instructiunile functiei. La reintoarcerea din functie programul principal trebuie sa continue de unde a ramas, astfel incat se incarca din stiva valoarea PC salvata la apelul functiei. Pe langa valoarea PC la intrarea intr-o functie se mai pot salva in stiva si alte date, cum ar fi continutul registrilor de uz general, continutul registrilor index, etc. Toate aceste date vor reconstitui daca este nevoie configuratia registrilor procesorului la intoarcerea din functie.

Unitatea de control al procesorului

Unitatea de control al procesorului coordoneaza functionarea acestuia si in unele cazuri are inclusa si UCAM. O schema functionala a unitatii de control care are inclusa si UCAM este prezentata in figura 12 [Mancas]. Unitatea de control al procesorului, conform figurii 12, realizeaza urmatoarele functii:

- decodifica instructiunile primite;
- calculeaza adresa operanzilor necesari executarii instructiunii;
- calculeaza adresa instructiunii urmatoare ce urmeaza a fi executata;
- primeste semnale de stare de la ALU si celelalte componente si trimite catre acestea semnalele necesare executarii instructiunii.

Instructiunea este citita din memorie si este primita pe magistrala de date in registrul instructiune. Codul instructiunii contine in general doua campuri: campul OPCODE care specifica functia care va fi executata de ALU si ADROP care specifica adresa unuia dintre operanzii utilizati in executarea instructiunii. ADROP nu apare totdeauna in formatul instructiunii. De exemplu o operatie de incrementare a unui operand aflat in registrul acumulator nu are nevoie decat de operandul pe care il gaseste implicit in acumulator, fara a mai fi necesar un al doilea operand. In cazul general o instructiune poate contine mai multe campuri ADROP, dar nu toate procesoarele permit aceasta.

Din registrul instructiune se separa cele doua campuri – OPCODE si ADROP, primul merge in registrul instructiune si al doilea in registrul adresa. OPCODE este decodificat si comunica catre blocului secventiator de control operatia ce urmeaza a fi executata.

Dupa primirea instructiunii in registrul instructiune, in aceasta schema blocul secventiator de

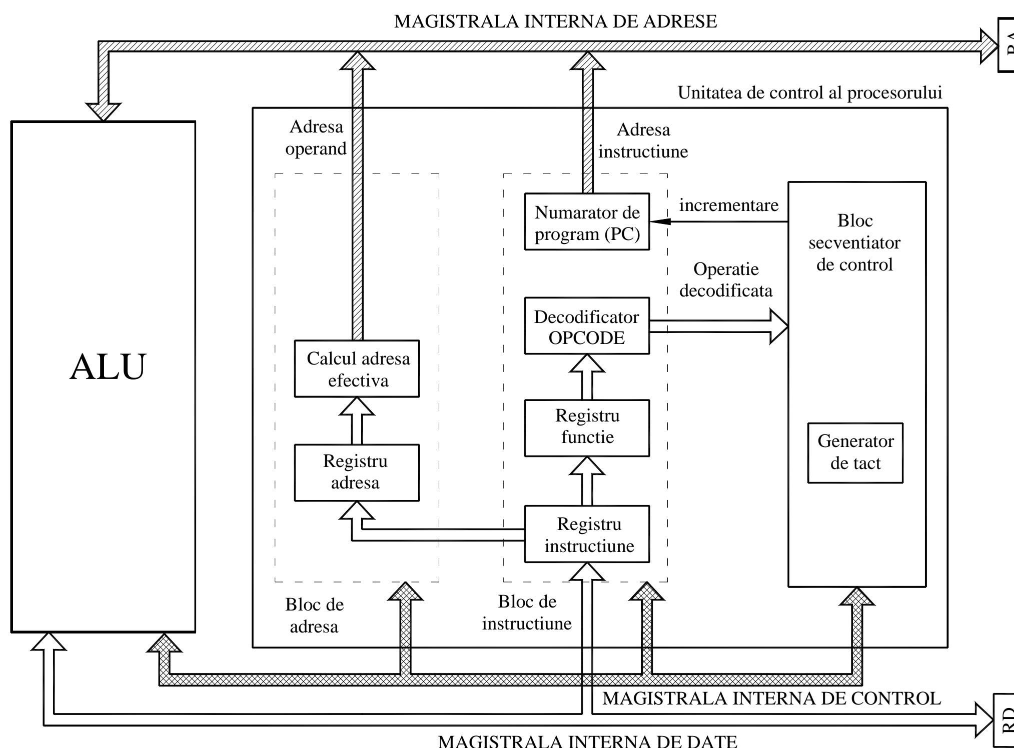


Fig. 12 – Schema simplificata a unitatii de control al procesorului

control incrementeaza numaratorul de program pentru a obtine adresa instructiunii urmatoare. Am vazut atunci cand s-a discutat despre UCAM ca adresa instructiunii urmatoare poate fi obtinuta in mai multe moduri.

Din registrul de adresa se citeste adresa celui de-al doilea operand si se calculeaza adresa efectiva a acestuia. Aceasta adresa este plasata pe magistrala de adrese pentru a accesa cel de-al doilea operand. Al doilea operand se poate afla fie in memorie, fie intr-unul din registri ALU, de aceea magistrala interna de adrese merge si in ALU. Daca al doilea operand se afla in memorie atunci acesta este adus pe magistrala de date in ALU pentru a se executa operatia. Blocul secventiator de control trimite catre ALU semnalele necesare executarii comenzii.

Fazele executarii unei instructiuni [Mancas]

Din cele prezentate anterior s-a putut deduce aproximativ modul de functionare a unui procesor. In cele ce urmeaza vom prezenta sintetizate aceste etape. In mod principial functionarea unui procesor se reduce la un ciclu de forma

repeat
Extrage instructiune din memorie;
Executa instructiune;
end repeat;

Privit mai in detaliu acest ciclu este prezentat in figura 13.

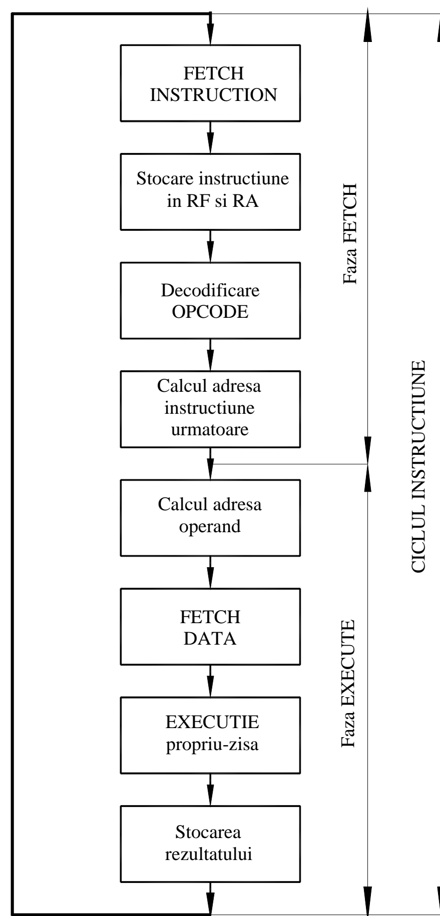


Fig. 13 – Fazele executiei unei instructiuni

Faza de extragere a unei instructiuni din memorie este denumita in limba engleza FETCH. In aceasta faza se mai executa cateva procese care asigura continuitatea functionarii microprocesorului.

Instructiunea este citita din memorie de la adresa continuta in registrul numarator de program (PC). Aceasta actiune poarta numele de FETCH INSTRUCTION (prima etapa din figura 13). Instructiunea este adusa pe magistrala de date in registul instructiune. De aici OPCODE este trimis mai departe in registrul de functie (RF) iar ADROP este trimisa in registrul de adrese (RA) (a doua etapa din figura 13). Urmeaza decodificarea OPCODE in blocul de decodificare (etapa a treia), astfel incat blocul secventiator de control cunoaste acum ce comenzi vor fi date pentru executia instructiunii. Totusi, inainte de a trece la executia propriu-zisa a instructiunii mai trebuiesc parcurse cateva etape. Mai intai este calculata adresa instructiunii urmatoare si este plasata in registrul PC astfel incat sa fie pregatita inceperea executiei instructiunii urmatoare (etapa a patra). In acest moment se considera incheiata faza de FETCH, care dupa cum se poate observa se refera la instructiune.

In general o instructiune are nevoie de doi operanzi. Considerand ca unul dintre ei este plasat in acumulator, este necesara identificarea celui de-al doilea operand. Pentru aceasta, pe baza informatiei continute in registrul de adrese este calculata si adresa la care se afla al doilea operand (etapa a cincea). Dupa aceea urmeaza aducerea din memorie, pe magistrala de date a celui de-al doilea operand. Aceasta operatie poarta numele de FETCH DATA (etapa a sasea). Abia in acest moment se poate trece la executia propriu-zisa a instructiunii (etapa a saptea). Dupa executie, pentru a putea fi folosit, rezultatul trebuie stocat fie in registrele ALU fie in memorie (etapa a opta).

Aceasta a doua faza descrisa este cunoscuta sub numele de faza EXECUTE.

Impreuna, faza FETCH si faza EXECUTE formeaza CICLUL INSTRUCTIUNE. In legatura cu desfasurarea in timp a activitatii microprocesorului mai sunt intalniti doi termeni

importanti. Primul este *starea* care corespunde duratei fizice a unei perioade de tact. Este durata de efectuare a unei actiuni elementare. A nu se confunda aceasta actiune elementara cu etapele prezentate in figura 13, care in general se pot desfasura pe mai multe perioade de tact, deci presupun mai multe stari. Cea de-a doua este notiunea de *ciclu masina* si corespunde unei grupari de mai multe actiuni elementare care duc la indeplinirea unei etape din ciclul instructiune. Se poate considera ca acest ciclu masina corespunde cate uneia dintre etapele din figura 13. Trebuie mentionat ca aceste cicluri masina pot avea lungimi diferite, deci pot presupune parcurgerea unui numar diferit de stari.

Din cele prezentate pana aici se poate constata ca pentru a se executa o instructiune este nevoie de parcurgerea mai multor stari, astfel incat efectuarea unei operatii de catre microprocesor dureaza mai multe cicluri de ceas. Acesta este motivul pentru care, desi microprocesoarele uzuale din prezent lucreaza la frecvente de tact de ordinul GHz, totusi, nu s-a depasit inca nivelul de executie a milioane de operatii de secunda. Din acest punct de vedere, viteza procesoarelor se masoara inca in mii de operatii pe secunda (KOPS) sau milioane de operatii pe secunda (MOPS). Aceste operatii sunt, asa cum am mentionat anterior, operatii elementare, care se executa intr-o instructiune. Executia unei inmultiri intre doi intregi reprezentati pe 8 bytes poate necesita executia chiar a zeci de operatii elementare, deci din acest punct de vedere, viteza de executie a operatiilor pe care le utilizeaza programatorul in limbajele de nivel inalt depinde foarte mult de modul de reprezentare a operanzilor.

Pentru accelerarea executiei instructiunilor, s-a recurs la impartirea procesorului in blocuri functionale specializate in extragerea datelor si instructiunilior din memorie si respectiv in executia propriu-zisa a instructiunilor. In acest mod, faza de executie a unei instructiuni se poate suprapune peste faza fetch a instructiunii urmatoare (figura 14) . Acest procedeu poarta denumirea de prefetching. Pentru utilizator aceasta ar insemna o dublare a vitezei de lucru a procesorului. In realitate nu este chiar o dublare avand in vedere ca mai apar si instructiuni de ramificatie si duratele celor doua faza (executie si fetch) nu sunt perfect egale.

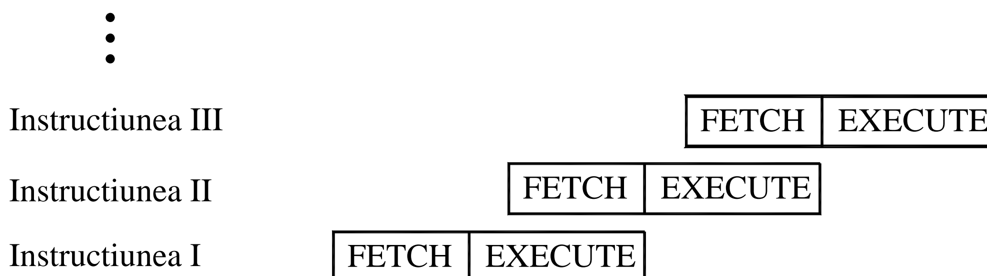


Fig. 14 - Procedeul de prefetching

O structura de microprocesor care utilizeaza acest mecanism este prezentata in figura 15. Un exemplu de procesor care utilizeaza acest procedeu este Intel 80286.

Comunicatia unitatii de executie cu memoria se face numai prin intermediul unitatii de interfata cu memoria (UIM). Aceasta aduce instructiunile si datele din memorie, iar instructiunile le stocheaza intr-o coada de instructiuni , fiind astfel pregatite pentru accesarea rapida de catre unitatea de executie. In perioadele in care nu trebuie sa trasfere date cu memoria, unitatea de interfata cu memoria aduce din memorie instructiuni, imbunatatind astfel timpul de exploatare al magistralelor.

In exemplul din figura 15, UIM contine si o serie de registri pentru structurarea logica a memoriei. Modul de utilizare al acestora va fi prezentat intr-un paragraf urmator. Tot in UIM se afla si numaratorul de program. Calculul adreselor se face tot in UIM, intr-un bloc de calcul simplificat, eliberand astfel unitatea de executie de sarcina calcularii adreselor. Este posibil ca largimea magistralei externe de date sa difere de largimea magistralei interne. Adaptarea intre cele doua magistrale se face tot in UIM.

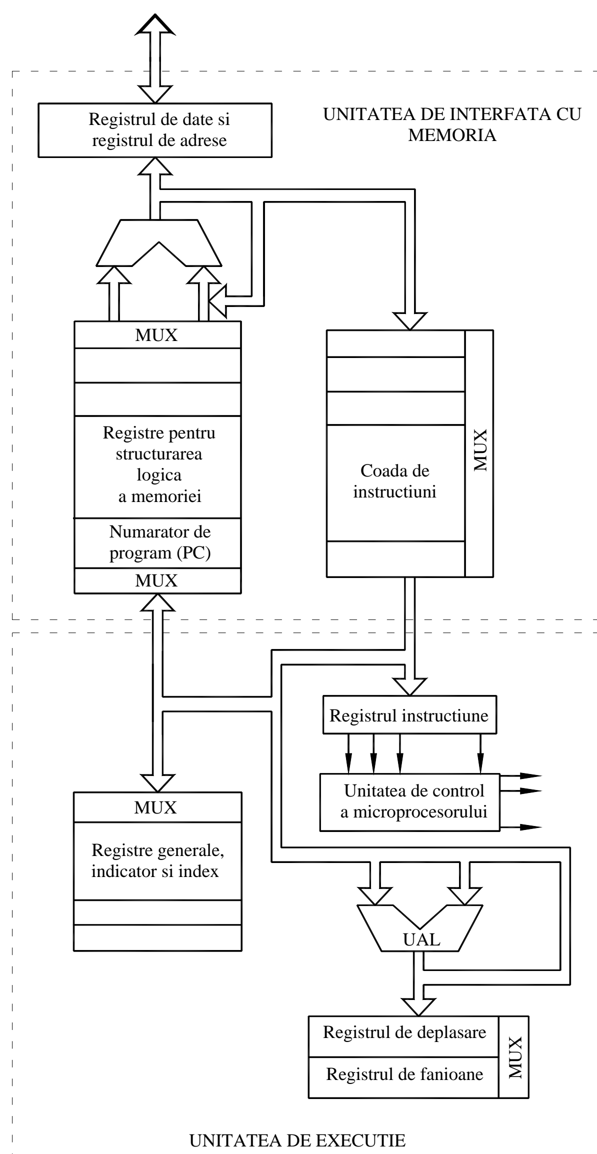


Fig. 15 – Impartirea microprocesurului in unitate de interfata cu memoria si unitate de executie

In unitatea de executie se gaseste registrul de instructiune din care unitatea de control extrage CODOP si il decodifica. Unitatea de control da semnale de comanda atat pentru blocurile din interiorul sau cat si pentru blocurile din UIM. Executia instructiunii se face in UAL, care utilizeaza registrul de deplasare si registrul de fanioane. In unitatea de executie se mai afla o serie de registri in care se pot stoca temporar date (registre generale) sau informatii care permit accesul in structuri de date de tip tablou (registre indicator si index). Intr-o oarecare masura rolurile registrelor sunt interschimbabile.

Arhitectura pipeline

Dupa cum s-a specificat anterior, executia unei instructiuni presupune mai multe faze, fiecare dintre ele executandu-se pe un anumit numar de cicli de ceas. O idee se accelerare a executiei programelor a fost aceea de prefetching in care faxza execute a unei instructiuni se suprapune peste faza fetch a instructiunii urmatoare, acestea fiind executate in blocuri diferite ale microprocesorului. Aceasta idee a fost dusa mai departe si s-a ajuns la impartirea procesului de executie a unei instructiuni in faze care se executa pe un singur ciclu de ceas in blocuri diferite. In acest fel, mai multe instructiuni se afla in executie la momentul curent, in diferite faze ale

procesului de executie. La procesoarele actuale s-a ajuns ca la fiecare ciclu de ceas sa se incheie executia unei instructiuni, astfel incat viteza de executie a uui sir de instructiuni s-a accelerat foarte mult. Totusi, in cazul in care apar instructiuni de salt va fi necesara abandonarea executiei sirului de instructiuni aflat in prelucrare la momentul actual si reluarea sirului de executie de la adresa de salt. Ca urmare, daca procesul de executie al unei instructiuni este impartit in 10 etape care se executa fiecare pe cate un ciclu de ceas in blocuri diferite, viteza de executie in cazul mecanismului *pipeline* nu creste chiar de 10 ori dar oricum, viteza de executie creste considerabil. Un exemplu de executie in mecanism pipeline care ia in consideratie etapele din figura 13 este prezentat in figura 16. Totusi, etapele reale parcurse in mecanism pipeline pot sa nu fie cele din figura 13 si instructiunea sa fie impartita in etape chiar mai mici.

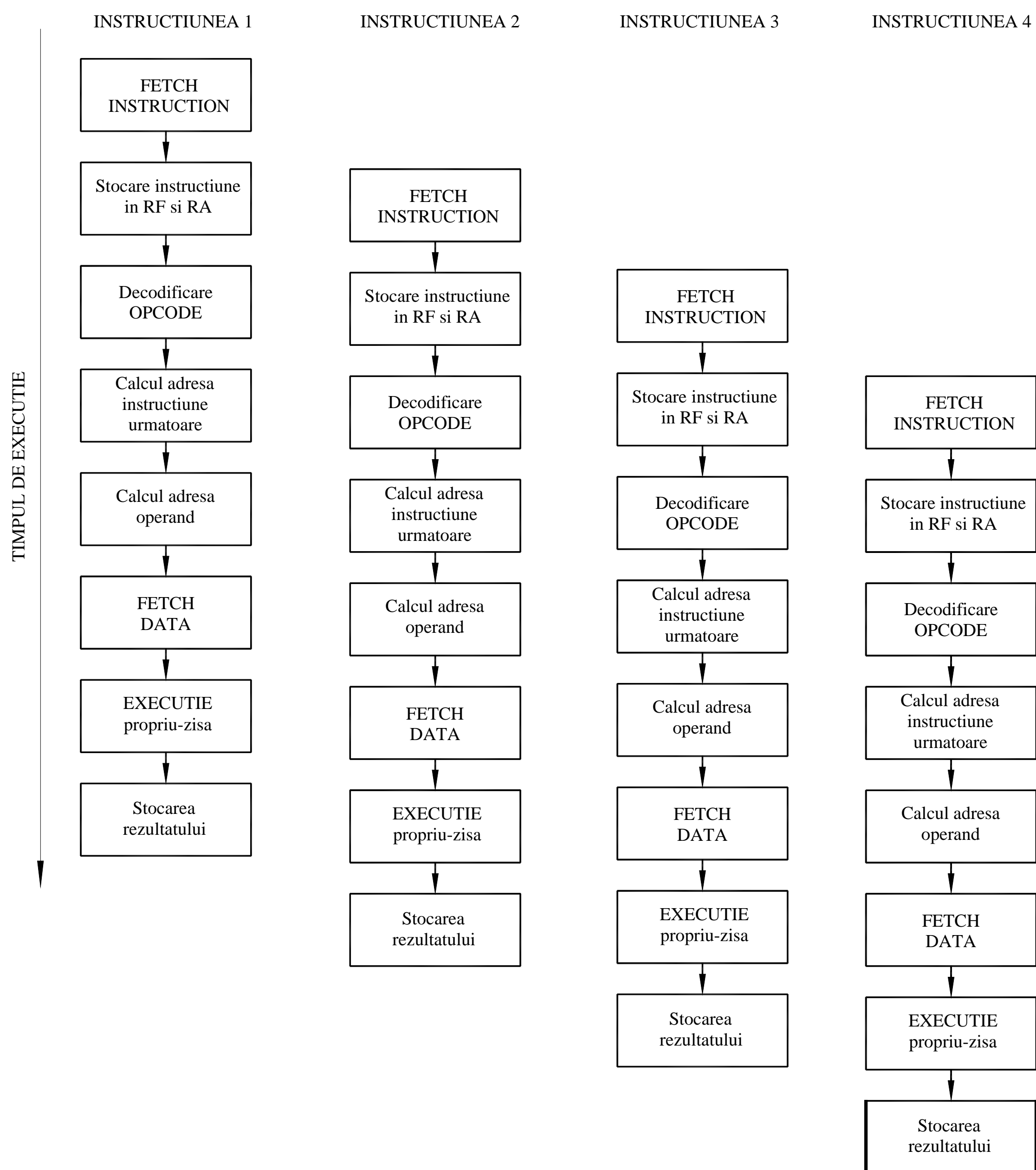


Fig. 16 – Executia instructiunilor in mecanism pipeline

Registrele unui procesor Intel 8086

Setul de registre care pot fi utilizate de programator in cazul unui astfel de microprocesor este prezentat in figura 17.

In figura 17 numaratorul de program si registrul CS apar hasurate. Acestea sunt utilizate in mod implicit de programator, dar nu le poate modifica in mod direct. Registrul CS este completat de asamblor, atunci cand se aloca resurse pentru programul respectiv iar numaratorul de program este gestionat tot de limbajul de asamblare (sau de compilator), fiind incrementat atunci cand se face executia secventiala a instructiunilor sau se calculeaza adresa continuta de el atunci cand se executa o instructiune de salt.

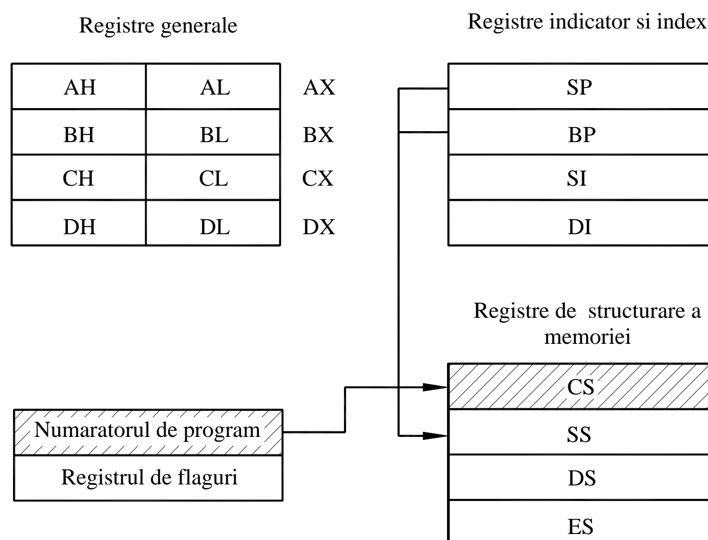


Fig. 17 – Registrele unui microprocesor pe 16 biti (Intel 80286)

Registrele generale sunt in numar de 4 (AX, BX, CX, DX) si sunt registre pe 16 biti. Ele pot fi accesate atat pe ansamblu, cat si pe jumatati (cate un octet), rezultand 8 registre de 8 biti (AH, AL, BH, BL, CH, CL, DH, DL). Acestea sunt registre care contin date, in unele situatii avand roluri bine precizate. De exemplu sunt instructiuni care utilizeaza in mod implicit registrul AX drept acumulator. Registrul BX poate contine si o adresa atunci cand se utilizeaza unele moduri particulare de accesare a memoriei. Registrul CX mai poarta numele si de registru contor, deoarece in afara de operanzi, mai este utilizat si pentru gestionarea operatiilor implicite cu siruri, deplasari si rotatii. Registrul DX poarta numele chiar de registru de date si contine in mod implicit operazi sau rezultate. Aceste registre, in forme evoluate se intalnesc si la microprocesoarele care ii urmeaza lui 80286 in serie (80386, 80486, Pentium). Pot fi insa pe 32 de biti, caz in care sunt denumite EAX, EBX, ECX si EDX sau pe 64 de biti.

Registrele indicator si index sunt registre accesibile doar pe 16 biti si sunt utilizate in operatiile de gestionare a memoriei.

Registrele SP (Stack Pointer) si BP (Base Pointer) sunt utilizate pentru gestionarea stivei. SP contine informatie care contribuie la calculul adresei fizice a varfului stivei iar BP contine informatie care permite calculul adresei fizice a bazei stivei. BP permite si crearea unui punct alternativ de intrare in stiva.

Registrele SI si DI poarta numele de registre index si contin informatie care permite calculul adresei fizice a bazei unui tablou de date fiecare. SI poarta si numele de registru index sursa iar DI de registru index destinatie. Aceste registe sunt multifunctionale. Pot contine si date sau adrese si pot fi folosite chiar ca acumulator pentru unele instructiuni.

Registrele de structurare logica a memoriei sunt tot pe 16 biti si contin informatie care permite calculul adresei de baza a segmentelor de memorie. Un segment de memorie este un bloc

de 64KB. De obicei, unui program i se aloca un segment de cod, unul de stiva, unul de date si in unele cazuri particulare i se mai aloca un segment suplimentar de date (extrasegment). Registrul CS permite caclulul bazei segmentului de cod, SS calculul bazei segmentului de stiva, DS calculul bazei segmentului de date si ES calculul bazei extrasegmentului.

Pentru a accesa stiva (varful sau baza stivei) se folosesc impreuna registrii SS si SP sau SS si BP. De asemenea pentru calculul adresei fizice a unei instructiuni se folosesc impreuna numaratorul de program si CS.

Structura registrului de flaguri pentru un astfel de procesor este prezentata in figura 18. Principalele flaguri sunt:

- A - poarta numele de flag de transport auxiliar. Stocheaza transportul intre jumatatea inferioara (nibble-ul inferior) si cea superioara a rezultatului;

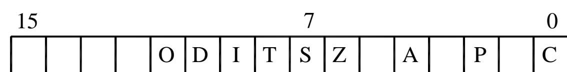


Fig. 18 – Principalele flaguri ale microprocesorului 80286

- C – flag de transport (*carry*) – stocheaza transportul de la bitul cel mai semnificativ al unui operand;
- D – flag de directie – este folosit pentru a indica sensul de parcurgere al unui tablou de elemente;
- I - flag de validare a intreruperilor. Daca acest flag este setat atunci microprocesorul nu accepta intreruperi;
- O – flag de depasire (*overflow*) indica aparitia depasirii de format la efectuarea operatiei;
- P – flag de paritate – indica daca numarul de cifre 1 din rezultatul unei operatii este par sau impar;
- S – flag de semn – indica semnul rezultatului unei operatii;
- T – flag capcana (*trap*) – daca este setatatunci programul se executa pas cu pas. Este utilizat in programele de depanare de tip debugger;
- Z – flag de zero – indica daca rezultatul unei operatii este zero.

Segmentarea si paginarea memoriei. Accesarea memoriei in mod protejat. (Gh. Marian)

Operatiile de segmentare si paginare a memoriei sunt operatii logice de structurare a memoriei, care permit accesarea mai rapida a informatiei in diverse situatii. Aceste operatii nu au nici un efect asupra memoriei fizice, ci doar asupra sistemului logic de accesare amemoriei.

Microprocesorul Intel 8086 are registrul numarator de program pe 16 biti. Aceasta inseamna ca numaratorul de program poate identifica 65536 adrese de memorie (64 KB). Totusi, pentru ca microprocesorul sa poata rula programe mai mari, magistrala sa externa de adrese este pe 20 biti, deci poate accesa 1048576 sau 1MB locatii de memorie. Pentru calculul adresei fizice a instructiunii se procedeaza ca in figura 19.

Se inmulteste continutul registrului CS (code segment) de 16 biti cu 16 si se aduna la valoarea obtinuta valoarea din numaratorul de program. Inmultirea cu 16 a continutului registrului CS este echivalenta cu o deplasare spre stanga a continutului acestuia cu 4 pozitii. Prin adunare la rezultat a registrului PC se obtine o valoare pe 20 de biti care este chiar adresa fizica de la care se va citi instructiunea. La fel se procedeaza si cu calculul adreselor operanzilor sau pentru accesarea stivei, dar de data aceasta sunt folosite registrele DS sau ES respectiv SS pentru stiva, al caror continut se deplaseaza spre stanga cu 4 pozitii.

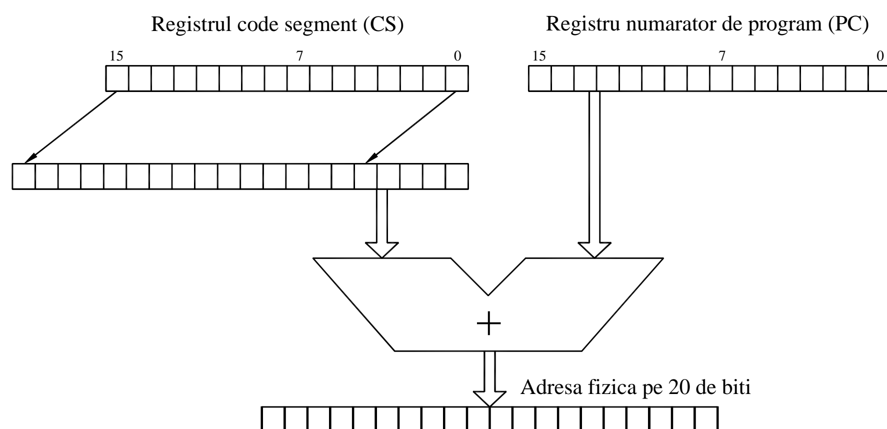


Fig. 19 – Calculul adresei fizice la microprocesorul Intel 8086

Se observa ca 64 KB pot fi accesati foarte repede, prin modificarea doar a valorii din PC, fara a mai fi nevoie de modificarea registrului CS. Aceasta cantitate de memorie a fost denumita *segment de memorie* si era de dimensiuni suficiente pentru programe uzuale de dimensiuni mici spre medii, la data la care a aparut microprocesorul Intel 8086. Era posibila realizarea si de programe mult mai mari, dar trebuiau impartite in blocuri de 64 KB, deci in segmente. Aceasta segmentare a memoriei era utila deoarece fiecare astfel de segment putea contine diferite module ale programului. Pentru rularea unui modul era nevoie de accesarea unui singur segment de memorie. Pentru saltul la un alt modul de program era nevoie de modificarea registrului CS in cadrul unei instructiuni de salt inter-segmente. Gestionarea segmentelor trebuie facuta fie de programator fie de sistemul de operare.

Pentru referirea la zone de memorie mult mai mici care sunt utile de exemplu in cazul executiei unor cicluri de lungime mica sau pentru a calcula mult mai usor adresa operanzilor, s-a definit notiunea de *pagina de memorie*. Lungimea paginii de memorie depinde de formatul instructiunii. Ideea care a condus la definirea paginilor de memorie a fost ca sa se identifice o locatie de memorie direct pe baza informatiei din formatul instructiunii, fara a mai fi nevoie de alte calcule. Daca in formatul instructiunii campul ADROP are k biti, atunci se pot identifica 2^k locatii de memorie. Pentru k=8 rezulta 256 B. O adresa poate fi identificata foarte rapid de exemplu daca sunt utilizati primii n-k biti din PC la care se adauga cei k biti din ADROP prin concatenare. Continutul ADROP va fi de fapt un deplasament pe k biti. Calculul de identificare a adresei se simplifica foarte mult. Se intalnesc notiunile de *pagina curenta* si *pagina relativa.* In cazul paginii curente deplasamentul poate fi doar pozitiv. Pagina curenta incepe la adresa specificata de primii n-k biti din PC urmati de k zerouri. In cazul paginii relative deplasamentul este un intreg cu semn, pagina relativa fiind centrata in jurul valorii din PC (figura 20).

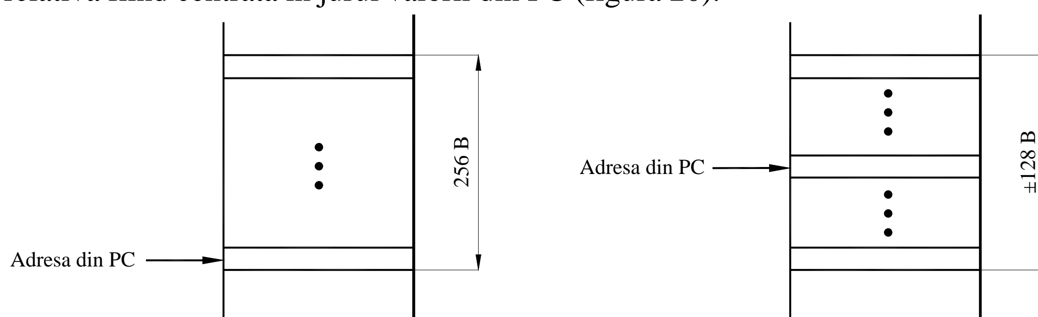


Fig. 20 – Pagina de memorie curenta si pagina relativa.

Un avantaj foarte important al utilizarii segmentelor si paginilor de memorie este obtinerea unor instructiuni cu formate mult mai compacte, astfel incat lungimea programului scade simtitor.

Prezentarea de mai sus a urmarit sa formeze o imagine in legatura cu notiunile de semgent de memorie si pagina de memorie. In cazul procesoarelor actuale mecanismele de accesare a memoriei au evoluat foarte mult dar se bazeaza pe aceste notiuni de segment de memorie si pagina de memorie.

Un mecanism de accesare a memoriei utilizat in prezent este cel al accesarii memoriei in mod protejat utilizat pentru accesarea memoriei virtuale. Memoria virtuala este o zona de memorie care se afla fizic in unitatile de stocare permanenta a informatiei (de exemplu pe hard disk), nu in memoria interna a calculatorului. Accesarea informatiei in cadrul memoriei virtuale se face utilizand adrese virtuale. O adresa virtuala contine un selector si un offset. Mecanismul este asemanator cu cel al segmentarii, dar de data aceasta selectorul contine o adresa a unei zone de memorie in care este stocat un *descriptor*. In calculul adresei fizice intervine informatia continuta in descriptor. Descritporii sunt stocati intr-o tabela de descriptori, care pe langa adresa de baza a blocului de memorie solicitat contine si alte informatii legate de limita segmentului, drepturile de acces, nivelul de privilegiu. Pentru calculul adresei se citeste selectorul, apoi din selector se merge in tabela de descriptori de unde se citeste adresa de inceput a blocului de memorie solicitat si apoi in cadrul acestui bloc de memorie se calculeaza adresa folosind deplasamentul (fig. 20). La accesarea unui descriptor de catre un program se indentifica si dreptul de acces al programului respectiv la zona de memorie solicitata.

Selectorul este un numar pe 16 biti, dar deplasamentul este pe 32 de biti (de data aceasta PC are 32 de biti). Deplasamentul pe 32 de biti poate identifica maxim 4GB de memorie. De obicei din cei 16 biti ai selectorului doar 14 sunt folositi pentru identificarea efectiva a unui bloc de memorie. Rezulta ca pentru identificarea unor locatii de memorie sunt disponibili 46 de biti, adica un volum total de memorie de 64 TB.

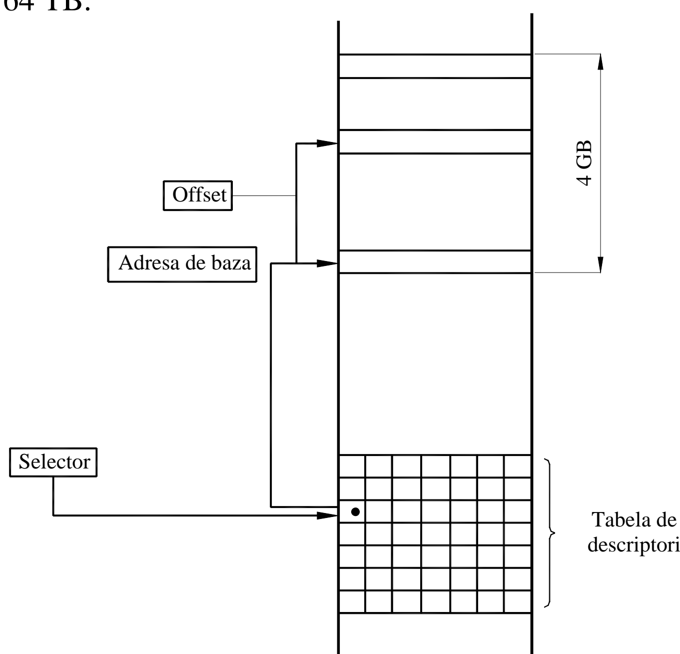


Fig. 21 – Accesarea memoriei in mod protejat

Tehnici de adresarea a memoriei

Odata cu cresterea dimensiunii programelor, pentru a obtine o flexibilitate cat mai mare in specificarea adreselor operanzilor s-au dezvoltat diferite tehnici de adresare a memoriei. Formatul instructiunilor, contine acum un camp ID care identifica modalitatea de accesare a memoriei utilizata de instructiunea respectiva.

Fig. 22 – Formatul unei instructiuni in care este specificat
modul de adresare al memoriei (campul ID)

Principalele modalitati de adresare a memoriei sunt prezentate in continuare.

Adresarea in registru

In acest caz informatia asupra careia actioneaza instructiunea se afla localizata intr-unul din registri microprocesorului. Permite cea mai rapida accesare a informatiei si necesita cel mai scurt format de instructiune. De obicei un octet este suficient pentru a codifica operatia modul de adresare si registrul in care se afla informatia. Cu acest mod de adresare spot localiza doar date (operanzi sau rezultate). O varianta a acestei modalitati de adresare este *adresarea implicita.* In acest caz, operandul este localizat in mod implicit intr-unul din registri (de obicei acumulatorul). Campurile ADROP lipsesc in acest caz din instructiune.

Adresarea imediata

Informatia referita in acest caz se afla in memoria program la octetul imediat urmator dupa OPCOD si ID. Adresa operandului va fi adresa din numaratorul de program incrementata. Octetul cel mai putin semnificativ al constantei se plaseaza primul. Si in cazul acestui mod de adresare se pot identifica doar date si mai precis doar *constante* folosite de programul respectiv. In functie de formatul constantei instructiunea are 2 sau 3 octeti.

Adresarea absoluta (extinsa sau directa)

Informatia este localizata printr-o adresa *completa* care se gaseste in formatul instructiunii, imediat dupa OPCODE si ID. Primul octet care specifica adresa se afla la adresa din numaratorul de program incrementata. Octetul cel mai putin semnificativ se plaseaza primul. In acest mod se pot accesa atat date cat si instructiuni, (de exemplu instructiunea urmatoare unui salt). Adresa specificata in formatul instructiunii nu este adresa fizica, ci doar informatia necesara calculului adresei fizice, asa cum s-a exemplificat in cazul microprocesorului Intel 8086 in figura 18. In cazul in care se identifica informatia in segmentul de date curent, in formatul instructiunii se specifica doar deplasamentul segmentului (in cazul din figura 18 pe 2 octeti). Daca se doreste identificarea unei informatii din cadrul altui segment, in formatul instructiunii trebuie specificata atat adresa de inceput a segmentului respectiv (in cazul din figura 18 tot pe 2 octeti) cat si deplasamentul (tot pe 2 octeti). Daca microprocesorul dispune de o harta de memorie mai mare decat 1 MB atunci specificarea adresei complete necesita mai multi octeti. Formatul instructiunii nu mai este compact, ca in cazurile anterioare, lungimea sa depinzand de harta memoriei procesului respectiv.

Harta memoriei = spatiul de adrese de memorie care poate fi accesat de microprocesorul respectiv. Depinde de numarul de linii de adresa al microprocesorului. Memoria fizica existenta nu este obligatoriu sa acopere intreaga harta de memorie a microprocesorului.

Adresarea scurta

Se aseamana cu adresarea absoluta, dar adresa cautata se afla in mod implicit in pagina zero a hartii de memorie sau a segmentului curent. In acest caz pentru identificarea adresei este necesar doar un deplasament pe 8 biti, care urmeaza dupa OPCODE si ID. Se identifica prin acest procedeu atat date cat si instructiuni. In cazul referirii la pagina zero a hartii de memorie, de obicei se face localizarea unor functii speciale (de exemplu functiile de tratare a intreruperilor).

Adresarea relativa

Informatia cautata in acest caz se afla in pagina curenta a instructiunii aflata in decodificare. Adresa specificata in formatul instructiunii este un deplasament care indica pozitia relativa a informatiei fata de adresa de inceput a paginii curente. Deplasamentul poate fi specificat pe un numar diferit de biti in functie de tipul microprocesorului. De exemplu la microprocesoarele pe 8 biti deplasamentul este de un octet iar la microprocesoarele pe 16 biti deplasamentul poate fi pe un octet sau doi octeti. Este tipica pentru instructiuni de salt intrasegment. Se pot localiza prin aceasta metoda atat date cat si instructiuni.

Adresarea indirecta

Permite o flexibilitate in localizarea informatiilor in memorie. Poate fi adresare indirecta prin registru sau cu memoria. In cazul adresarii indirecte prin registru in instructiune, in campul ADROP se specifica un registru in care se gaseste *adresa informatiei cautate*. Acest mod de adresare poate fi utilizat atat pentru localizare de date si operanzi, cat si pentru localizare de instructiuni. Avantajul acestui mod de adresare este formatul compact al instructiunii care pe langa CODOP si ID contine o adresa foarte scurta, adresa registrului in care se afla adresa informatiei cautate. Timpul de acces la informatie insa este mai mare, deoarece unitatea de interfata cu memoria trebuie citeasca informatia din registru, sa calculeze pe baza acesteia adresa fizica a informatiei cautate si abia apoi se face accesarea memoriei.

In cazul adresarii indirecte cu memoria la adresa specificata in ADROP se gaseste adresa informatiei cautate. Avantajul utilizarii unui astfel de mod de adresare este acela ca se pot face realocari de module de program si de date in memorie cu modificari minime in programul principal. Timpul de accesare al informatiei cautate insa este considerabil mai mare decat in cazul adresarii indirecte in registru. Trebuie facute doua accesari ale memoriei pentru a ajunge la informatia cautata. Prima accesare se face pentru a gasi adresa informatiei si a doua accesare se face pentru a gasi informatia propriu-zisa. In plus, unitatea de interfata cu memoria trebuie sa calculeze doua adrese fizice.

Adresarea indexata

Este utilizata pentru a accesa informatia in structuri de date de tip tablou. Pentru construirea adresei se foloseste un registru index (SI sau DI) si un deplasament specificat in cadrul instructiunii. In unele cazuri registrul index care este utilizat este cunoscut in mod implicit din formatul instructiunii, astfel incat in campul ADROP apare doar deplasamentul in cadrul tabloului. Se poate face astfel identificarea informatiei in cadrul tablourilor cu instructiuni cu formate compacte. Inainte de a accesa datele dintr-un tablou insa, trebuie incarcata adresa bazei tabloului in registrul index corespunzator.

Exista doua tipuri de adresare indexata: *adresarea preindexata* si *adresarea postindexata*.

Adresarea preindexata a fost descrisa practic mai sus. In cazul adresarii postindexate in campul ADROP apare adresa deplasamentului si nu deplasamentul propriu-zis. Si in acest caz se asigura o flexibilitate in ceea ce priveste relocarea modulelor de date in memorie, dar timpul de acces este mai lung, deoarece se face o accesare in plus a memoriei, la fel ca la adresarea indirecta.

Adresarea relativa la baza (bazata)

Acest tip de adresare seamana intru catva cu adresarea indexata. In cadrul segmentului curent de date este utilizata o adresa de baza plasata in registrul BX, fata de care se localizeaza informatia prin intermediul unui deplasament. Pentru calculul adresei fizice sunt utilizate registrele DS, BX si deplasamentul care apare in cadrul instructiunii.

Adresarea bazata indexata

Este o combinatie intre cazurile de adresarea bazata si cea indexata. Pentru calculul adresei informatiei cautate se folosesc registrele DS, BX, SI (sau DI) si un deplasament specificat in formatul instructiunii. In acest fel se localizeaza informatii in tablouri a caror adresa de intrare in tablou a fost definita printr-o adresare bazata.

Adresarea informatiei in stiva

In cazul microprocesoarelor din clasa 80x86, stiva se constituie intr-un segment de memorie separat. Ca urmare, accesarea informatiei din stiva se face utilizand registrul SS care contine adresa bazei segmentului de stiva si registrii SP si BP care contin baza si respectiv varful stivei. Cea mai uzuala metoda de accesare a informatiilor din stiva este accesarea varfului stivei, care este utilizata in mecanismele de apel ale functiilor. Adresa varfului stivei se face utilizand registrii SS si SP. In stiva insa se pot stoca si alte informatii decat numaratorul de program la apelul functiilor. Se pot stoca spre exemplu date in mecanismul de transfer al datelor catre o functie si de la functie catre programul principal. In acest sens se pot utiliza variante de adresare a memoriei prezentate anterior.

Adresarea in stiva directa

Utilizand registrul BP in care se stocheaza adresa bazei stivei de obicei, se poate accesa orice informatie din stiva. Calculul adresei fizice in acest caz se face utilizand registrele SS si BP si eventual un deplasament fata de adresa din BP continut in formatul insructiunii.

Adresarea in stiva indexata

Poate fi folosita in cazul in care in stiva s-a stocat un tablou de elemente si se doreste accesarea acestuia. Pentru calculul adresei unui element al tabloului se folosesc registrele SS, BP, SI sau DI si un deplasament care apare in formatul instructiunii.

Dupa cum s-a prezentat mai sus, accesarea informatiei in stiva se poate face in mai multe moduri, dar asigurarea corectitudinii gestionarii informatiei din stiva revine exclusiv programatorului.

Lucrul cu stiva

Stiva este o zona de memorie cu organizare de tip *ultimul intrat-primul iesit*. Implementarea stivei se poate face fie hardware, caz in care accesul la stiva este mai rapid dar de obicei stiva are dimensiuni mai reduse, sau software, caz in care stiva este localizata in memoria operativa. In acest al doilea caz accesul la stiva dureaza ceva mai mult, dar stiva poate avea dimensiuni considerabil mai mari. Spre exemplu, pentru un procesor 8086 stivei i se aloca un segment de 64 KB de memorie.

Stiva este caracterizata de trei parametri: *adresa de inceput a stivei (baza stivei), adresa varfului stivei* si *dimensiunea stivei*. Pentru cazul prezentat anterior, adresa de inceput a stivei se calculeaza folosind registrul SS (stack segment) si registrul BP (base pointer). Adresa varfului stivei este calculata folosind registrul SS si registrul SP (stack pointer). Adresa bazei este adresa primului element introdus in stiva iar adresa varfului stivei este totdeauna adresa ultimului element introdus in stiva. Dimensiunea stivei este specificata intr-un registru separat. Este specificata de fapt ultima adresa care poate fi utilizata de stiva, denumita *plafonul stivei.* Plafonul stivei este stocat intr-un registru denumit REG1.

Pentru gestionarea stivei mai exista doua flaguri care indica starile de *stiva plina* si *stiva goala*. Flagul de stiva goala este setat atunci cand s-a extras si primul element introdus in stiva.

Previne accesarea in continuare a zonei respective de memorie in sensul extragerii de date din stiva. Continuarea extragerii de date din stiva dupa golirea acesteia ar conduce la accesarea de informatii eronate, care de fapt nu au fost niciodata introduse in stiva si care este posibil sa apartina altor programe decat celui in derulare. Flagul de stiva plina este setat atunci cand zona de memorie alocata stivei a fost umpluta cu date. Varful stivei a atins plafonul acesteia. Introducerea de date in continuare in stiva presupune introducerea de date intr-o zona de memorie in care se pot afla alte date ale programului sau ale altor programe. In acest caz se ajunge la o functionare anormala a calculatorului si chiar la blocarea acestuia.

Operatiile specifice lucrului cu stiva sunt realizate prin intermediul instructiunilor *push* si *pop*. Instructiunea *push* introduce o data in stiva iar instructiunea *pop* extrage o data din stiva. Introducerea datelor in stiva se face intr-un mod specific fiecarui procesor. Spre exemplu la procesorul 8086 la o instructiune *push* sunt introdusi doi octeti in stiva, octetul cel mai semnificativ fiind introdus primul, iar la instructiunea *pop* sunt extrasi doi octeti din stiva. In cazul introducerii in stiva a datelor reprezentate pe mai multi octeti trebuie avuta in vedere ordinea in care se stocheaza in stiva octetii care reprezinta data respectiva.

Derularea unei instructiuni *push* pentru o stiva implementata software, in cazul in care se stocheaza un octet in stiva este urmatoarea:

- Se aduce data care urmeaza a fi stocata in stiva in registrul de date;
- Se compara adresa varfului stivei cu adresa plafonului. Daca sunt egale se seteaza flagul de stiva plina si derularea operatiei inceteaza;
- Se incrementeaza registrul SP ;
- Utilizand registrele SS si SP se calculeaza adresa noului varf al stivei, care este adusa in registrul de adrese;
- Se scrie data din registrul de date in memorie la adresa specificata in registrul de adrese.

Scrierea unei informatii pe doi octeti in stiva presupune de fapt reluarea algoritmului de mai sus pentru fiecare din cei doi octeti stocati in stiva. Se considera ca datele stocate in stiva intre baza stivei si varful stivei sunt date utile programului care vor fi utilizate la un moment ulterior. Deci o data introdusa in stiva este pastrata pana cand va fi utilizata la un moment ulterior.

Extragerea unei date din stiva se face cu ajutorul instructiunii *pop* a carei derulare in cazul extragerii unui octet din stiva este urmatoarea:

- Se compara continutul registrului SP (varful stivei) cu cel al registrului BP (baza stivei). Daca sunt egale atunci stiva este goala. Se seteaza flagul de *stiva goala* si se opreste algoritmul;
- Pe baza registrilor SS si SP se calculeaza adresa fizica a varfului stivei, care este trimisa in registrul de adrese;
- Se citeste de la adresa specificata in registrul de adrese un octet care este adus in registrul de date. Data citita va fi acum disponibila pe magistrala interna de date;
- Se decrementeaza registrul SP, pentru a indica noul varf al stivei.

Dupa cum se poate observa, desi data citita din stiva ramane fizic si in locatia de memorie la care a fost stocata, decrementarea registrului SP indica procesorului ca adresa respectiva de memorie este libera si se poate utiliza pentru introducerea altor date in stiva. Pentru procesor data scoasa din stiva a fost stearsa din zona de memorie in care se afla. Pentru extragerea datelor pe mai multi octeti din stiva, se repeta algoritmul de mai sus pentru fiecare octet al datei respective. De asemenea trebuie avuta in vedere ordinea de stocare a octetilor in stiva atunci cand se face extragerea.

Stiva are doua utilitati foarte importante intr-un calculator in deservirea cererilor de intrerupere si in apelurile de functii. In unele calculatoare este folosita si pentru efectuarea calculelor aritmetice, dar acestea sunt mai putin utilizate.

In cazul aparitiei unei cereri de intrerupere (se vor detalia intr-un paragraf ulterior intreruperile) procesorul termina de executat instructiunea in curs si trece la executia unei proceduri de deservire a cererii de intrerupere care este de fapt tot o functie, dar fara parametri de intrare. Pentru a putea continua executia programului principal la intoarcerea din procedura de deservire a intreruperii, este necesar ca sa fie salvat numaratorul de program si registrul de flaguri (*cuvantul de stare*) din momentul aparitiei intreruperii. In unele cazuri este necesara salvarea si a continutului registrilor microprocesorului. Salvarea acestor date se face in stiva, deoarece se retine semnificatia acestor date prin ordinea de salvare in stiva. La intoarcerea din procedura, se incarca aceste date din stiva in registrii din care proveneau, in ordine inversa celei in care au fost stocate. Se reconstituie astfel configuratia microprocesorului din momentul aparitiei intreruperii. Un proces asemanator se desfasoara atunci cand se apeleaza functii in cadrul unui program.

Apeluri de functii

Mecanismul apelului functiilor a fost prezentat in linii mari deja, in continuare vom sintetiza acest mecanism si vom arata cum stiva poate fi utilizata pentru transferul parametrilor intre programul principal si functia apelata.

Apelul unei functii presupune ca programul principal sa poata continua executia dupa intoarcerea din functie. In acest sens, este strict necesar ca sa se cunoasca adresa de intoarcere din functie si starea microprocesorului inainte de apelul functiei, adica registrul de flaguri. In plus, daca in interiorul functiei se modifica valorile registrilor si valorile aflate in acestia la apelul functiei mai sunt necesare la intoarcerea din functie, este necesar sa se poata reconstitui si valoarea acestor registri. Cel mai simplu mecanism pentru reconstituirea acestor parametri este utilizarea stivei. Mecanismul de apel al functiei este urmatorul:

- se salveaza in stiva datele din registri (daca este necesar);
- se salveaza in stiva registrul de flaguri (RF);
- se salveaza in stiva numaratorul de program (PC). Acesta contine adresa urmatoarei instructiuni din programul principal, calculata in momentul executiei instructiunii curente (instructiunea de apel al functiei – vezi figura 13);
- se executa salt la prima instructiune din functie;
- se executa instructiunile din functie pana la intalnirea instructiunii RET (intoarcere din functie);
- la intalnirea instructiunii RET se incarca din stiva numaratorul de program salvat anterior, asigurandu-se continuarea programului principal;
- se incarca din stiva registrul de flaguri;
- se incarca din stiva datele din registri (daca este necesar);
- se continua executia programului principal.

Salvarea in stiva a numaratorului de program si a registrului de flaguri la apelul functiei se face automat, la fel si incarcarcarea valorilor acestor registri la intoarcerea din stiva. Salvarea datelor din ceilalti registri in stiva si reincarcarea lor ramane in sarcina programatorului.

In cazul apelului functiilor se poate face transferul parametrilor intre programul principal si functie tot prin stiva. Inainte de apelul functiei se incarca in stiva parametrii reali cu care va fi rulata functia. Se incarca de fapt valorile parametrilor de intrare si adresele parametrilor de iesire. Apoi se realizeaza apelul functiei, care va salva RF si PC in stiva. Functia va utiliza parametrii de intrare din stiva prin intermediul registrului BP care permite realizarea de puncte de intrare alternative in stiva si nu SP, a carui gestionare defectuoasa poate conduce la blocarea programului. Rezultatele obtinute de functie vor fi plasate la adresele specificate in stiva, inaintea de apelul functiei. La reintoarcerea

in programul principal se incarca automat PC si RF din varful stivei, astfel incat programul principal poate continua rularea, dar mai trebuie evacuati din stiva parametrii de apel ai functiei. In acest fel ***stiva ramane in aceeasi stare ca si inainte de apelul functiei***. Aceasta este o conditie esentiala pentru buna functionare a programelor care apeleaza functii.

Registrele RF si PC care se salveaza in stiva impreuna cu eventualele registre ce urmeaza a fi reconstituite si parametrii functiei poarta numele de cadrul functiei. Deci la fiecare apel al unei functii se stocheaza in stiva un cadru al functiei respective, care este extras din stiva la reintoarcerea din functie.

Un exemplu de program care apeleaza doua functii este prezentat in figura 23.

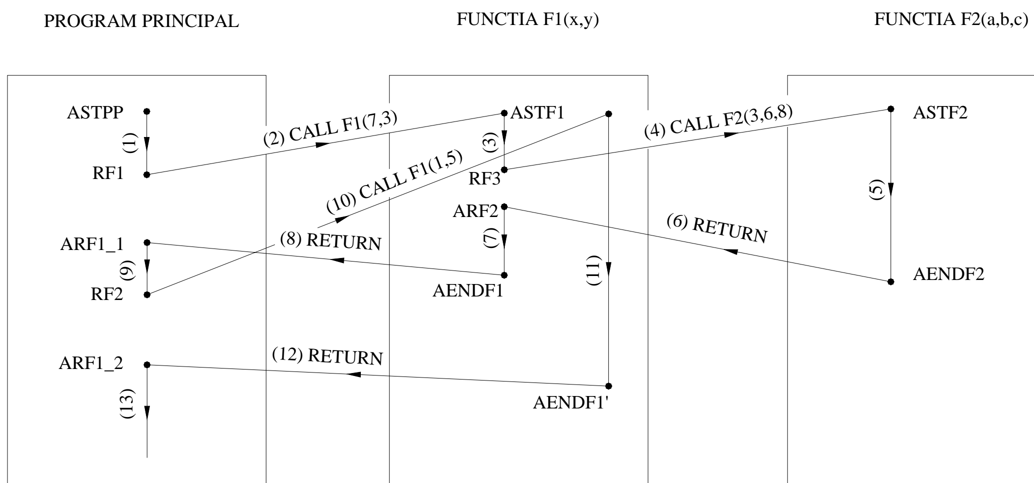


Fig. 23 – Program in a carui executie sunt utilizate doua functii

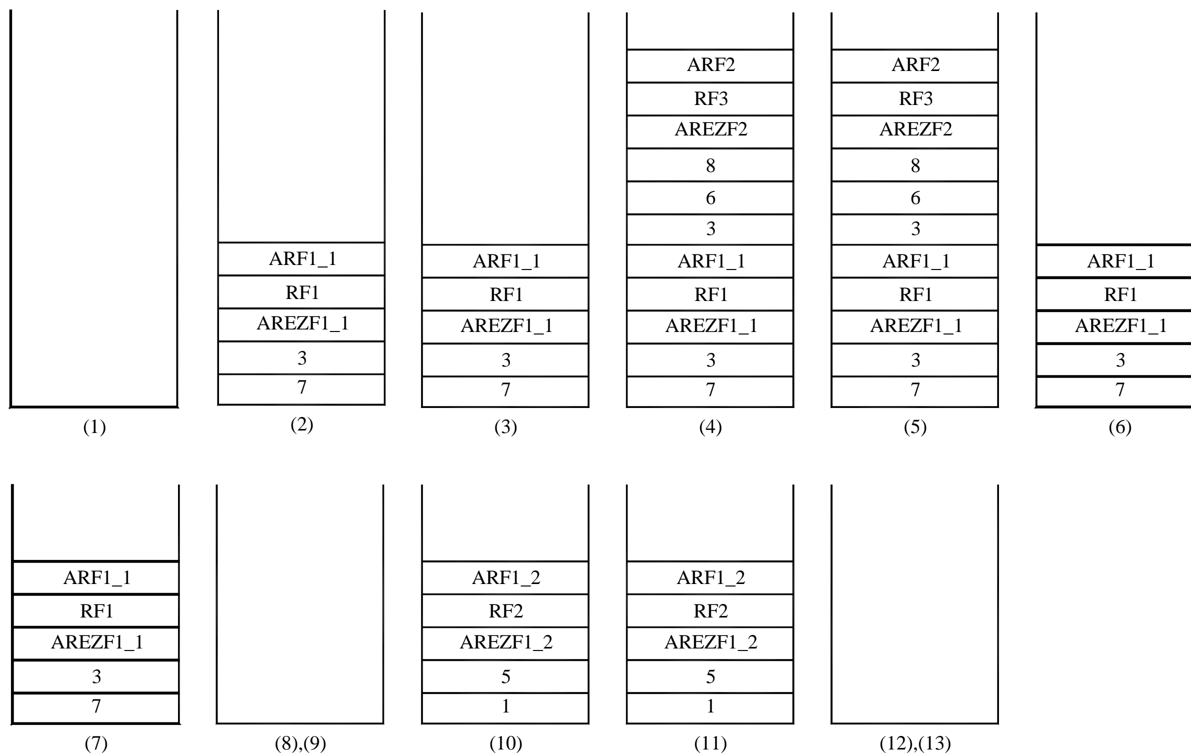


Fig. 24 – Starea stivei in decursul executiei programului din figura 23

Programul principal apeleaza o functie F1(x,y). La primul apel al functiei F1 cu parametrii reali 7 si 3, in functia F1 se intalneste apelul functiei F2(a,b,c). Aceasta este apelata cu parametrii reali 3, 6 si 8. Dupa executia functiei F2 se face intoarcerea in functia F1 care a apelat-o si apoi,

dupa incheierea functiei F1 se face intoarcerea in programul principal. In programul principal se intalneste apoi un al doilea apel al functiei F1 cu parametrii reali 1 si 5. In acest caz in functia F1 nu se mai intalneste apelul la functia F2. Dupa executia celei de-a doua apelari a functiei F1 se face intoarcerea in programul principal si se incheie executia acestuia.

Starea stivei pe parcursul executiei programului este prezentata in figura 24. Notatiile din figurile 23 si 24 au urmatoarele semnificatii:

ASTPP	- adresa de start a programului principal,
RF1	- configuratia registrului de flaguri la primul apel al functiei F1,
AREZF1_1	- adresa la care va fi depus rezultatul functiei F1 dupa primul apel,
ARF1_1	- adresa la care se face intoarcerea dupa primul apel al functiei F1,
ASTF1	- adresa de start a functiei F1,
RF3	- configuratia registrului de flaguri la apelul functiei F2,
ARF2	- adresa la care se face intoarcerea dupa apelul functiei F2,
AREZF2	- adresa rezultatului functiei F2,
ASTF2	- adresa de start a functiei F2,
AENDF2	- adresa finala a functiei F2,
AENDF1	- adresa finala a functiei F1,
RF2	- configuratia registrului de flaguri la al doilea apel al functiei F1,
ARF1_2	- adresa la care se face intoarcerea dupa al doilea apel al functiei F1,
AREZF1_2	- adresa la care va fi depus rezultatul functiei F2 dupa al doilea apel,
AENDF1'	- adresa finala a functiei F1 pe al doilea fir de executie.

In momentul de incepere a executiei programului principal stiva este goala. Se ruleaza programul pana cand se intalneste primul apel al functiei F1(x,y). Pe parcursul executiei primei instructiuni de apel a functiei F1 se calculeaza adresa urmatoarei instructiuni din programul principal, asa cum este prezentat in figura 13. Aceasta adresa este ARF1_1.

In programul principal, inainte de a se face primul apel propriu-zis al functiei F1, sunt salvati in stiva (*push*) parametrii reali ai primului apel, adica 7 si 3, precum si adresa la care se va depune rezultatul primului apel al functiei F1. La executia instructiunii de apel al functiei F1 se salveaza in stiva (tot instructiune *push*), registrul de flaguri in configuratia RF1 si numaratorul de program care va contine adresa de intoarcere dupa primul apel al functiei F1 – ARF1_1. Astfel, la intrarea in functia F1 dupa primul apel, stiva va avea configuratia din figura 24.(2).

Tot pe parcursul executiei primei instructiuni de apel a functiei F1 se incarca in numaratorul de program adresa de start a functiei F1 – ASTF1 si se incepe executia functiei F1. In functia F1 sunt utilizati parametrii 7 si 3 din stiva. Acestia sunt accesati prin intermediul registrului BP si nu SP, pentru a evita gestionarea incorecta a varfului stivei. Accesarea acestor parametrii nu se face prin intermediul instructiunilor de tip *pop*, deoarece s-ar deteriora configuratia stivei, ci prin intermediul unor mecanisme de adresare directa in stiva.

Pe parcursul executiei functiei F1 cu parametrii reali 7 si 3 se intalneste apelul functiei F2(a,b,c) cu parametrii reali 3, 6 si 8. In functia F1, inainte de instructiunea de apel a functiei F2, se salveaza in stiva (*push*) parametrii 3, 6 si 8 precum si adresa la care va fi depus rezultatul functiei F2 – AREZF2. In timpul executiei instructiunii de apel a functiei F2 se calculeaza adresa de intoarcere din functia F2 in functia F1 - ARF2, si se salveaza in stiva (*push*) configuratia registrului de flaguri RF3 si ARF2. In comementul intrarii in functia F2, stiva va avea configuratia din figura 24(4). Aceasta configuratie se mentine si pe parcursul executiei functiei F2 (etapa (5)). Pe parcursul executiei functiei F2, parametrii reali 3, 6 si 8 sunt accesati la fel ca si in cazul functiei F1 prin mecanisme de adresare directa in stiva iar rezultatul este stocat la adresa AREZF2 folosind mecanisme de indirectare.

La terminarea functiei F2 se incarca din stiva in numaratorul de program ARF2 (*pop*) si apoi in registrul de flaguri configuratia RF3 (*pop*). De asemenea sunt extrasi din stiva (fie tot prin instructiune *pop*, fie prin modificarea directa a registrului SP) parametrii reali ai functiei F2 (AREZF2, 8, 6 si 3). Incarcarea numaratorului de program si a registrului de flaguri se face automat, extragerea parametrilor reali din stiva trebuie facuta de catre programator la intoarcerea in

functia F1 din functia F2. Configuratia stivei dupa incheierea functiei F2 este cea din figura 24.(6), care se observa ca este identica cu cea de dinainte de intrarea in functia F2, adica 24.(3).

Mai departe se continua executia primului apel al functiei F1, utilizand parametrii reali asa cum am mentionat anterior si stocand rezultatul functiei F1 la adresa AREZF1_1 utilizand mecanisme de indirectare. La finalul primului apel al lui F1 se incarca automat din stiva in numaratorul de program adresa de intoarcere din primul apel al functiei F1 ARF1_1 (*pop*) si in registrul de flaguri configuratia RF1 (*pop*). Imediat dupa intoarcerea in programul principal trebuie evacuati din stiva si parametrii reali ai primului apel al functiei F1 (AREZF1_1, 3 si 7), operatie care ramane in sarcina programatorului. Evacuarea acestor parametrii din stiva se face la fel ca la intoarcerea din functia F2 in functia F1. Starea stivei ajunge dupa realizarea acestor operatii cea din figura 24.(8), deci stiva ramane goala, la fel ca la inceputul programului principal si inainte de intrarea in functia F1.

In cazul celui de-al doilea apel al functiei F1 cu parametrii reali 1, 5 si AREZF1_2 lucrurile se petrec asemanator, salvand in stiva cadrul functiei F1 format acum din (1, 5, AREZF2, RF2 si ARF1_2). Dupa intrarea a doua oara in functia F1 stiva are configuratia din figura 24.(10). Parametrii reali sunt accesati tot prin mecanisme de accesare directa in stiva iar rezultatul functiei F1 la a doua executie se stocheaza tot prin mecanisme de indirectare la adresa AREZF1_2. La intoarcerea din al doilea apel al functiei F1 se incarca din stiva cadrul functiei F1 (automat ARF1_2 in numaratortul de program si RF2 in registrul de flaguri si prin program parametrii de apel), astfel incat in final stiva ramane goala, la fel ca la intrarea in functia F1 la al doilea apel (figura 24.(12)).

Daca dupa incheierea mecanismului de apel al unei functii stiva nu ramane in aceeasi stare ca inainte de realizarea acestuia, atunci sigur exista erori in programul respectiv.

Tehnici de intrare-iesire

Pentru a-si realiza functiile la bordul aeronavei, un calculator trebuie sa preia date din mediul exterior si sa trimita rezultatele prelucrarilor la elementele de executie sau afisare corespunzatoare. Acest schimb de date cu exteriorul poarta numele de proces de intrare-iesire. Pentru ca datele sa poata fi preluate de calculator trebuie ca ele sa fie prezentate deja in format numeric binar. La iesire, calculatorul ofera rezultatele tot in format numeric binar. Conversia marimilor de intrare in format numeric si a rezultatelor in marimi analogice se realizeaza asa cum s-a vazut in capitolul al doilea cu ajutorul interfetelor de intrare-iesire. Un rol important in aceste interfete il au convertoarele analog-numerice si numeric-analogice prezentate de asemenea in capitolul al doilea, dar nu sun singurele dispozitive care pot realiza conversia unei marimi de intrare in format numeric binar sau a unui rezultat binar intr-o marime utila la iesire.

Transmisia datelor dinspre si spre exterior se face prin locatii speciale de memorie denumite porturi. Fiind niste locatii de memorie, porturile comunica cu microprocesorul prin magistrala de date, magistrala de adrese si magistrala de control. Deci aceste magistrale trebuie sa ajunga si la porturi. Solutiile alese pentru realizarea conexiunii la aceste magistrale sunt foarte diverse in functie de tipul portului. Exista porturi dispuse chiar in memoria operativa a calculatorului, pentru care legaturile la magistrale este chiar legaturile memoriei la magistrale si exista porturi plasate in alte locatii de memorie decat memoria operativa a calculatorului. De obicei aceste locatii de memorie sunt registre care au alocate adrese separate de cele din memoria operativa si microprocesorul comunica cu aceste registre si prin semnale de comanda si control, de obicei cele mai uzuale fiind semnalele de validare a scrierii la porturi, citire de la porturi, semnalele de intrerupere lansate de porturi catre microprocesor, etc.

Un exemplu de utilizare a porturilor de intrare iesire in cazul unei aplicatii simple care preia o tensiune din exterior prin intermediul unui convertor analog-numeric si trimite la iesire tot o tensiune printr-un convertor numeric-analogic este prezentat in figura 25.

Semnalele de comanda trimise de microprocesor in exterior in acest caz sunt reprezentate doar de semnalul de citire/scriere ($R/\overline{W}$). De asemenea porturile semnalizeaza faptul ca sunt gata de comunicatie cu microprocesorul prin intermediul semnalului READY.

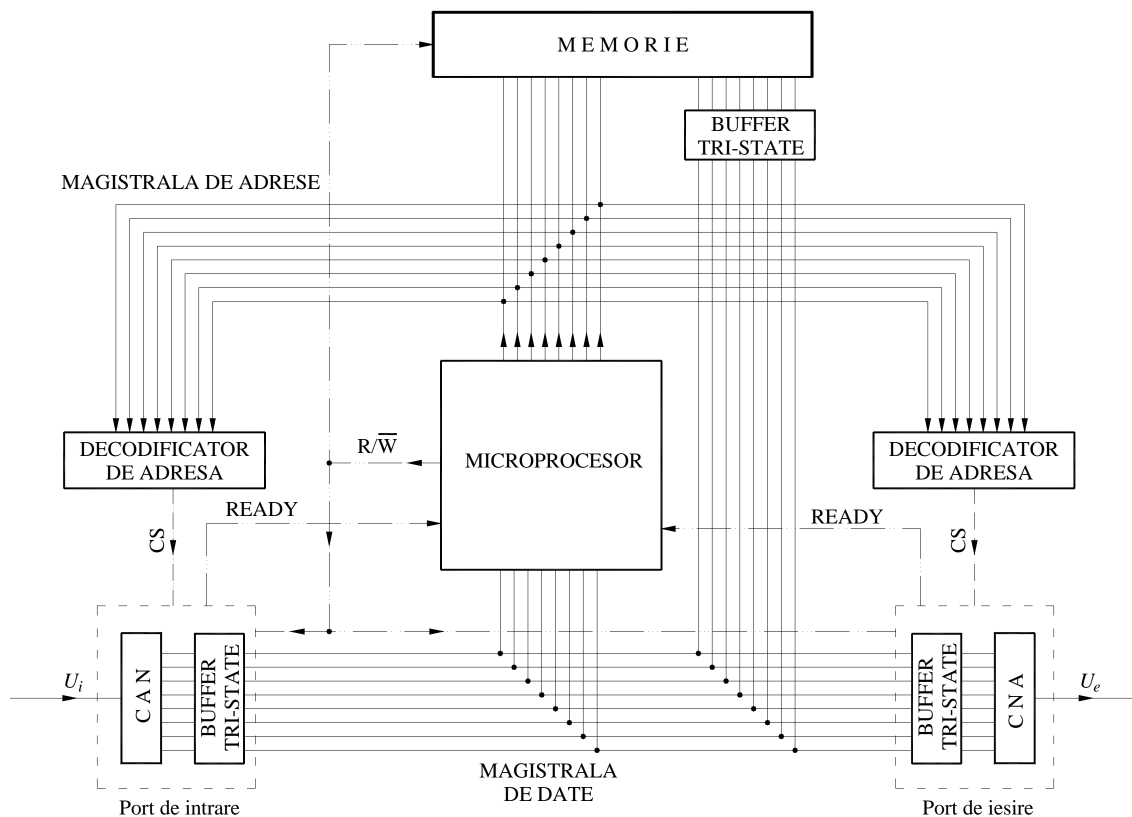


Fig. 25 – Exemplu de aplicatie simpla cu un port de intrare si un port de iesire

La interfata dintre magistrala de date si elementele cuplate la aceasta (memorie si porturile de intrare-iesire) se afla plasate buffere tri-state. Acestea au rolul de a nu permite decat unuia dintre elementele cuplate la magistrala de date sa puna date pe magistrala la un moment dat. Bufferele tri-state au la iesire trei stari: 0 si 1 logic recunoscute de microprocesor si o a treia stare, de impedanta ridicata, in care practic iesirile (intrarile) registrului sunt izolate de magistrala de date. Pentru ca bufferul tri-state sa permita comunicatia portului respectiv sau a memoriei cu magistrala de date trebuie sa primeasca semnalul de comanda, reprezentat aici de semnalul *chip select* (CS). Pentru memorie, semnalul CS se activeaza daca pe magistrala de adrese se afla o adresa din domeniul alocat memoriei. Acesta este obtinut tot de la un decodificator de adresa, dar care nu mai este reprezentat in figura 25. Pentru porturile de intrare-iesire exista cate un decodificator de adresa separat, realizat de obicei cu circuite logice. Atunci cand pe magistrala de adresa apare adresa alocata portului respectiv, decodificatorul de adresa activeaza semnalul CS si permite astfel portului respectiv sa comunice cu magistrala de date prin intermediul bufferului tri-state corespunzator.

In cazul aplicatiei prezentate in figura 25 operatiile de intrare-iesire sunt comandate de catre microprocesor. Pentru o operatie de intrare, trebuie realizata de fapt citirea iesirii CAN –ului. In acest scop, microprocesorul pune pe magistrala de adrese adresa portului de intrare. In acest fel se activeaza semnalul CS al portului de intrare si pe magistrala de date vor apare iesirile buferului tri-state al portului de intrare. Semnalele CS ale memoriei si portului de iesire vor fi dezactivate, astfel incat microprocesorul comunica doar cu portul de intrare prin magistrala de date. In cazul in care CAN-ul a finalizat conversia in curs si in registrul sau de iesire se afla o data disponibila, activeaza semnalul READY, informand astfel ca se poate face citirea. In acest moment microprocesorul activeaza semnalul R (Read) si se realizeaza citirea datei pusa pe magistrala de date de catre portul de intrare. Activarea semnalului R inainte ca semnalul READY sa fie activat de catre portul de intrare poate conduce la citirea unei date eronate, deoarece CAN-ul nu a finalizat inca operatia de conversie curenta iar in registrul sau de iesire se afla un rezultat intermediar, nu cel final.

Pentru o operatie de iesire trebuie scrisa o data in registrul portului de iesire. Lucrurile se petrec asemanator cu cazul operatiei de intrare. Microprocesorul pune pe magistrala de adrese adresa portului de iesire si se activeaza semnalul CS al portului de iesire. Semnalele CS pentru memorie si portul de intrare sunt dezactivate in acest caz. Deci microprocesorul va comunica doar cu portul de iesire. Buferul tri-state permite scrierea in registrul portului de iesire o data. Microprocesorul citeste semnalul READY al portului de iesire si daca acesta este in 1 logic, atunci portul de iesire este pregatit sa preia o data. Microprocesorul pune pe magistrala de date valoarea care trebuie trimisa la portul de iesire si apoi activeaza semnalul $\overline{W}$ (Write), realizand astfel scrierea datei in registrul portului de iesire. Semnalele $R/\overline{W}$ sunt de obicei vehiculate prin aceeasi cale. Citirea si scrierea fiind operatii complementare, pot fi comandate in contratimp prin intermediul aceleiasi linii de semnal. In cazul de fata daca pe linia de comanda se afla 1 logic atunci inseamna ca este comandata o operatie de citire, iar daca se afla 0 logic atunci este comandata o operatie de scriere. Semnalul de scriere apare barat ($\overline{W}$) ceea ce semnifica faptul ca operatia respectiva este activa pe nivel 0 logic (sau nivel scazut).

In cazul de fata scrierea in registrul portului de iesire atunci cand nu este activat semnalul READY al portului de iesire nu are efecte semnificative, deoarece CNA-ul este realizat printr-un sumator cu amplificator operational, deci conversia se face practic instantaneu. In alte cazuri insa scrierea in portul de iesire fara ca semnalul READY al acestuia sa fie activat poate duce la functionarea anormala a aplicatiei.

Din punct de vedere al aplicatiei software implementata pe microprocesor pot exista trei metode de realizare a operatiei de intrare-iesire: metoda de interogare continua (polling), utilizarea intreruperilor si transferul direct cu memoria (*direct memory access* sau DMA). Fiecare dintre ele prezinta avantaje si dezavantaje si se utilizeaza in aplicatii specifice.

Tehnica de interogare continua (polling)

Aceasta tehnica a fost de fapt prezentata anterior. Microprocesorul cand are nevoie de citirea unei date de la portul de intrare trimite adresa acestuia pe magistrala de adrese si asteapta activarea semnalului READY de la portul respectiv. In intervalul de timp de la lansarea operatiei de citire de la porturi si pana cand este activat semnalul READY microprocesorul asteapta, neputand sa desfasoare nici o alta actiune. In cazul in care portul de intrare este lent aceasta inseamna foarte mult timp pierdut pentru microprocesor. Ca urmare aceasta metoda este utilizata foarte rar si pentru porturi accesate cu frecventa foarte redusa. Nu se utilizeaza in aplicatiile care necesita viteza mare de lucru.

Tehnica intreruperilor

Marea majoritate a procesoarelor sunt prevazute cu terminale de intrerupere. In cazul aparitiei unui 1 logic pe un astfel de terminal este activata o “intrerupere”. Aceasta inseamna ca procesorul va incheia executia instructiunii in curs, dupa care va lansa in executie o *procedura de deservire a intreruperii*. Astfel de proceduri de deservire a intreruperilor sunt asemanatoare cu procedurile uzuale, dar nu au parametri de intrare si de iesire. In cazul aplicatiei din figura 25 implementarea acestei tehnici presupune legarea semnalului READY al portului de intrare la unul dintre terminalele de intrerupere. In aceasta situatie finalizarea unei conversii de catre CAN produce activarea semnalului READY si deci activeaza o intrerupere. In cadrul procedurii de deservire a intreruperii se va realiza citirea datei disponibile pe portul de intrare. Deci procesorul poate realiza alte operatii iar finalizarea conversiei de catre CAN va declansa operatia de citire. Ca urmare procesorul nu mai pierde timp in asteptarea finalizarii conversiei, deci in asteptarea datelor de la portul de intrare. Aceasta tehnica permite transferuri mai rapide de date de la porturi si este frecvent utilizata in operatiile de intrare-iesire pentru volume relativ mici de date.

Intreruperile sunt de mai multe tipuri si pot fi clasificate in *intreruperi hardware* si *intreruperi software*. Intreruperea din exemplul anterior este o intrerupere hardware. Exista si

intreruperi lansate de aplicatiile software si care in principiu functioneaza la fel dar lansarea lor nu mai este declansata de un semnal extern ci de indeplinirea unor conditii in aplicatia care le lanseaza.

Intreruperile sunt ierarhizate intr-o anumita ordine de prioritate. In cazul in care apar simultan doua semnale de intrerupere, atunci este deservita intreruperea cu prioritatea cea mai mare. Daca in timpul deservirii unei intreruperi apare o alta intrerupere cu prioritate mai mare atunci se intrerupe executia procedurii intreruperii aflate in desfasurare si se lanseaza in executie procedura de executie a intreruperii mai prioritare aparute. Daca intreruperea aparuta are o prioritate mai mica decat intreruperea pentru care se afla in desfasurare procedura de deservire, atunci a doua intrerupere este ignorata.

Intreruperile sunt clasificate si dupa un alt criteriu in intreruperi *mascabile* si intreruperi *nemascabile*. In unele situatii este necesar ca activitatea microprocesorului sa nu fie intrerupta, indiferent de aparitia unui semnal de intrerupere. Pentru astfel de situatii se activeaza un semnal intern al microprocesorului care conduce la ignorarea tuturor semnalelor de intrerupere. Se spune ca s-au *mascat* intreruperile. Mascarea intreruperilor se face pe cale software de catre programator. Exista insa si intreruperi de prioritate foarte mare care nu pot fi mascate pe aceasta cale. Acestea sunt intreruperile nemascabile.

Tehnica de acces direct la memorie (DMA)

Aceasta tehnica de intrare-iesire este foarte frecvent utilizata pentru volume mari de date, de exemplu trimiterea datelor catre o imprimanta sau citirea datelor de la o unitate optica, etc. Utilizarea sa mai presupune prezenta in schema alaturi de microprocesor a unui *controller de DMA* (figura 26). Acesta va coordona transferul DMA al datelor.

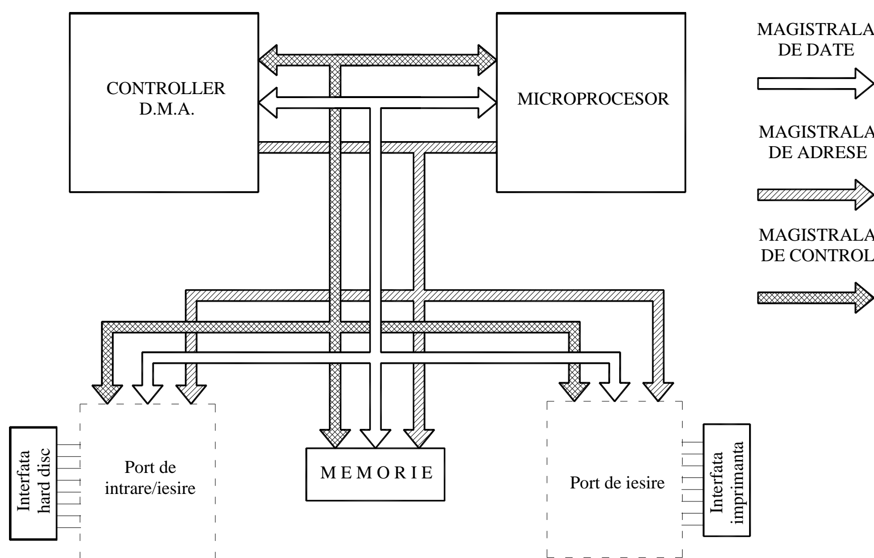


Fig. 26 – Transferul de date prin acces direct la memorie

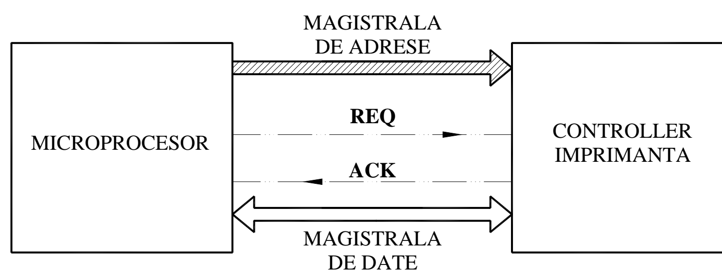
In cazul in care apare necesitatea transferului unui volum mare de date de la memorie la sau de la unul dintre periferice, microprocesorul initializeaza controllerul DMA cu adresa de start a domeniului la (de la) care trebuie sa faca transferul, numarul de octeti si sensul in care se face transferul. Spre exemplu daca adre de trimis un fisier din memorie la imprimanta, specifica adresa de inceput a fisierului in memorie si lungimea fisierului, precum si faptul ca fisierul va fi trimis din memorie la portul de iesire corespunzator imprimantei. In continuare este asteptat un moment in care microprocesorul nu are nevoie de transferuri de date pe magistrala si transfera controlul magistralelor controllerului DMA. Din acest moment microprocesorul nu va mai utiliza magistralele pana cand se va finaliza transferul de date. Controllerul DMA este si el un microprocesor dar cu un ALU simplificat, deoarece nu are alta sarcina decat sa faca transferuri de

date. Acesta controleaza functionarea magistralelor la fel cum o face si microprocesorul, dar el realizeaza doar operatii de citire-scriere. In momentul cand pe magistrala de control i s-a specificat faptul ca i s-a transferat controlul magistralelor, el incepe sa faca transferul de date cerut in momentul initializarii. Cand a terminat transferul de date comunica microprocesorului tot pe magistrala de control faptul ca a incheiat sarcina atribuita si microprocesorul poate utiliza din nou magistralele.

Un astfel de mecanism de transfer este foarte eficient atunci cand microprocesorul este prevazut cu memorie cache. In intervalul de timp in care se face transferul de date microprocesorul poate continua activitatea utilizand instructiunile si datele stocate in memoria cache, deci apare o imbunatatire substantiala a gradului de utilizare a timpului de calcul al microprocesorului. De asemenea, acest mecanism este foarte util in cazul procesoarelor multitasking, cand transferul de date asociat unui task poate fi suprapus peste timpul de calcul asociat unui alt task. Din punct de vedere software, coordonarea transferului DMA este realizata de catre sistemul de operare.

Protocolul handshaking de transfer de date

Acest protocol este utilizat pentru transferul de date intre microprocesor si periferice, mai ales atunci cand frecventele de ceas ale microprocesorului si perifericului respectiv difera foarte mult. Spre exemplu frecventa de ceas a unui microprocesor poate fi in prezent de ordinul 1-2 GHz iar frecventa de ceas a unei imprimante matriceale cu ace poate fi de ordinul a 10 – 100 MHz. Unele dintre semnalele de comanda si de stare sunt de obicei activate pe durata unei perioade de ceas si sunt citite de echipamentul corespondent de obicei pe unul dintre fronturile impuslurilor sale de ceas. Datorita diferentei mari dintre cele doua frecvente, este posibil ca activarea unor semnale de comanda si de stare ale echipamentului mai rapid sa treaca neobservate de echipamentul mai lent, astfel incat apar situatii anormale in transferul de date intre cele doua echipamente.



REQ					
Actiune microprocesor	Depune date pe magistrala de date	Activeaza REQ si asteapta activarea ACK	Sesizeaza activarea ACK si dezactiveaza REQ	Asteapta eventual inceperea unui nou transfer	
ACK					
Actiune controller imprimanta	Asteapta un transfer de date	Sesiseaza activarea REQ si citeste datele	Activeaza ACK si asteapta dezactivarea REQ	Sesizeaza dezactivarea REQ si dezactiveaza ACK	

Fig. 27 – Desfasurarea protocolului handshaking

Protocolul *handshaking* realizeaza sincronizarea transferului de date intre astfel de periferice cu viteze foarte diferite de lucru. In cadrul magistralei de control vor exista doua linii de semnal, denumite REQ (request) si ACK (acknoledge). Linia REQ este controlata de echipamentul care

emite datele si linia ACK este controlata de echipamentul receptor. Daca transferul de date este bidirectional atunci mai este necesara inca o pereche de linii REQ si ACK. Dupa cum se poate observa este necesar ca ambele echipamente intre care se comunica datele sa fie echipamente inteligente, adica sa fie dotate cu un microprocesor sau microcontroller care sa ii coordoneze activitatea. Un astfel de protocol nu poate fi implementat in cazul unor echipamente foarte simple, cum ar fi cele din exemplul din figura 25. In figura 27 este exemplificat transferul intre un microprocesor rapid si o imprimanta lenta.

Atata timp cat nu se desfasoara nici un transfer intre microprocesor si controllerul de imprimanta semnalele REQ si ACK sunt dezactivate. In momentul in care microprocesorul are de transmis date la imprimanta pune adresa controllerului de imprimanta pe magistrala de adrese si datele de transferat pe magistrala de date, dupa care activeaza semnalul REQ, semnalizand astfel controllerului de imprimanta faptul ca are de preluat date de pe magistrala de date.

Cand controllerul de imprimanta sesizeaza activarea semnalului REQ citeste datele de pe magistrala de date, dupa care activeaza semnalul ACK, informand astfel microprocesorul ca a preluat datele de pe magistrala. In acest interval de timp microprocesorul trebuie sa mentina fixe datele pe magistrala de date si adresa controllerului de imprimanta pe magistrala de adrese. Deoarece microprocesorul este foarte rapid, modificarea datelor si adresei dupa o sigura perioada a ceasului sau ar putea face ca datele care trebuie trimise imprimantei pur si simplu sa nu poata fi citite de aceasta, deoarece se modifica cu o viteza prea mare. Rezultatul citirii ar fi eronat. El trebuie sa astepte controllerul de imprimanta sa activeze semnalul ACK. De asemenea o mentinere activa a semnalului REQ doar pe durata unei singure perioade de ceas a microprocesorului ar putea trece neobservata de controllerul de imprimanta.

Trecerea la un nou transfer de date intre microprocesor si controller trebuie facuta atunci cand ambele echipamente au sesizat finalizarea transferului curent, si de asemenea au fost instiintate de faptul ca si echipamentul corespondent a luat la cunostinta de finalizarea transferului curent. Acceptarea trimiterii de noi date atat timp cat este activ semnalul ACK ar conduce la desincronizarea transferului. Activarea semnalului REQ si modificarea datelor si adresei de pe magistrale ar putea trece neobservate de controllerul de imprimanta sau s-ar modifica cu o viteza prea mare, astfel incat controllerul de imprimanta ar pierde din datele transmise. In consecinta, atunci cand microcoprocesorul sesizeaza activarea semnalului ACK dezactiveaza semnalul REQ, instiintand controllerul de magistrala ca din punctul sau de vedere transferul curent s-a incheiat. Microprocesorul trebuie sa aiba confirmarea din partea controllerului de magistrala a faptului ca a primit acest mesaj. Confirmarea este realizata de controllerul de imprimanta in momentul in care a sesizat dezactivarea semnalului REQ prin dezactivarea semnalului ACK. In acest moment transferul curent s-a incheiat si poate incepe unnou transfer de date.

Implementarea acestui protocol poate fi facuta in toate cele trei variante de transferuri de date – tehnica de polling, tehnica intreruperilor sau tehnica de acces direct la memorie.

Arhitecturi CISC si RISC