Cuprins

Capitolul 1

Arhitectura unui sistem de calcul

Pentru început să trecem în revistă organizarea generală a unui sistem actual de calcul.

1.1 Organizarea de bază

Structura şi compunerea logică a programelor care pot fi prelucrate de o maşină de calcul au fost elaborate anterior definirii structurii hard; ele se bazează pe teza lui Church[1], enunţată în prima parte a anilor ′30. Ulterior ea a fost completată de Turing.

Definiţia 1.1 *O metodă (procedură) M – asociată rezolvării unei probleme – se numeşte "efectivă" (sau "mecanică") dacă:*

- M se poate exprima printr-un număr finit de comenzi (instrucţiuni), fiecare comandă putând fi definită pe baza unui alfabet finit de simboluri.

- M poate produce rezultatul dorit într-un număr finit de paşi;

- M poate fi descrisă în mod logic, fără a folosi mijloace externe.

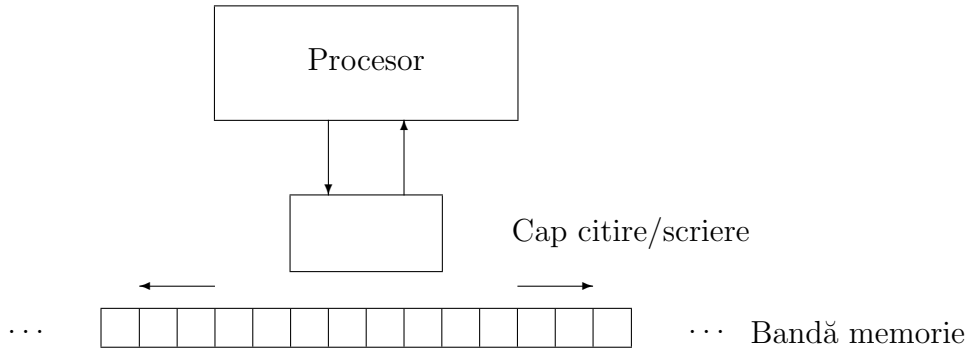
În 1933 Barwise stabileşte criteriile pe care trebuie să le satisfacă un computer:

- Nu va stoca răspunsurile la toate problemele posibile.
- Va rezolva numai probleme pentru care i s-a dat o procedură de rezolvare.
- Timpul necesar rezolvării unei probleme este finit.

Turing, la rândul lui, aduce în anii ′40 o serie de precizări de ordin formal asupra posibilităţilor de calcul pe care le poate avea un computer.

Maşina definită de el (*Maşină Turing*) este modelul matematic al calculatoarelor de mai târziu.

[1]Pe scurt enunţul acesteia este: *O funcţie $f : \mathcal{N} \longrightarrow \mathcal{N}$ este efectiv calculabilă dacă şi numai dacă este recursivă.*



O maşină se compune dintr-un procesor (generator de instrucţiuni), o bandă de memorie care se poate mişca în ambele sensuri, şi un cap de citire/scriere fix.

La fiecare tact, banda se mişcă cu o poziţie spre stânga sau dreapta, permiţând executarea unei comenzi.

O astfel de comandă (sau *instrucţiune*) este reprezentată în procesor sub forma unui quadruplu (S_h, T_i, O_j, S_k), cu următoarea semnificaţie:

Dacă starea procesorului este S_h şi simbolul citit pe bandă este T_i, atunci maşina efectuează operaţia O_j şi trece procesorul în starea S_k.

sau - ca o variantă formalizată:

$$\textbf{if } stare = S_h \textbf{ and } input = T_i \textbf{ then } \; output := O_j \textbf{ and } stare := S_k$$

Operaţiile permise pe o maşină Turing sunt:

1. $O_j = T_j$: simbolul T_j este scris pe bandă în locul lui T_i;
2. $O_j = R$: banda se deplasează spre dreapta cu o locaţie;
3. $O_j = L$: banda se deplasează spre stânga cu o locaţie;
4. $O_j = H$: calculul se opreşte.

Se demonstrează că orice calculator este echivalent din punct de vedere al posibilităţilor de calcul cu o maşină Turing, iar aceasta – conform Tezei lui Church – poate aborda orice problemă ce admite o rezolvare algoritmică.

Nu vom prezenta detalii ale acestor aserţiuni, ele constituind subiectul altor domenii de cercetare, mult mai generale şi fără multă legătură cu ceea ce vrem să prezentăm în această carte.

1.2 Structura generală a maşinilor de calcul

Structura funcţională a unei maşini de calcul (sau *Computer*) în forma actuală a fost definită în 1947 de von Neumann (1903 - 1957), modificările ulterioare fiind nesemnificative.

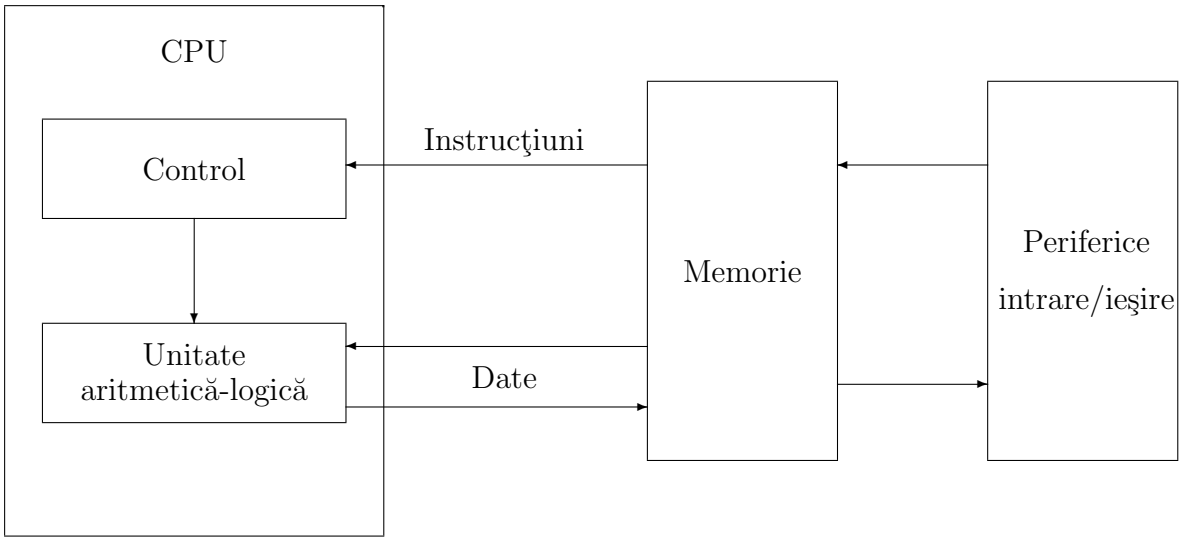
În principal ea conţine trei componente de bază:

1. O unitate centrală de prelucrare (CPU) compusă din: o unitate de calcul (aritmetică şi logică) şi o unitate de control.

   Rolul ei principal este de a localiza şi executa comenzi.

2. Una sau mai multe unităţi de stocare a datelor (memorii); aici sunt păstrate datele şi programele calculatorului (programe care formează componenta de software sau programe utilizator).

3. Periferice (intrare/ieşire): unităţi de transfer a informaţiei spre şi dinspre utilizator.



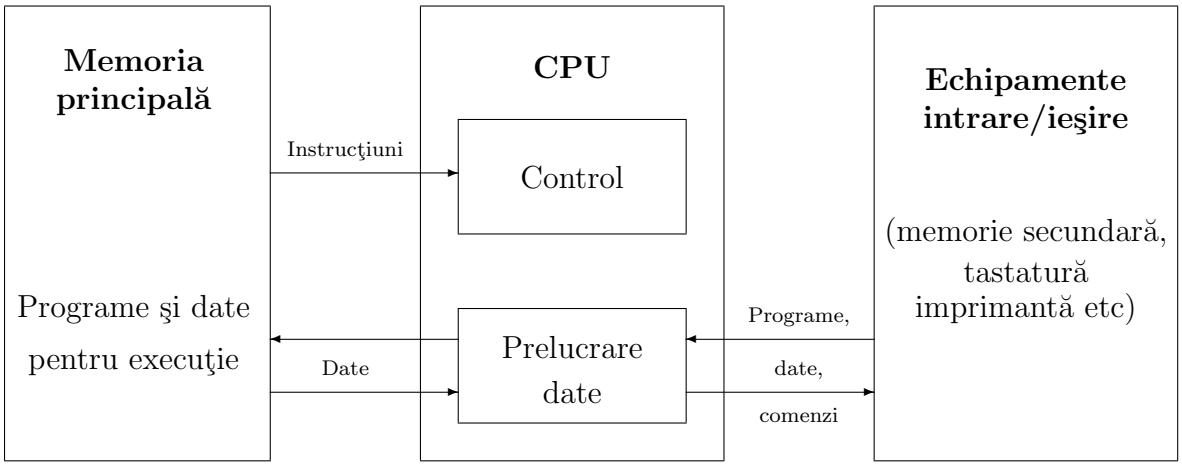
Aceste componente hard comunică între ele printr-un sistem de magistrale. Principalul soft inclus este *sistemul de operare*, care asigură majoritatea funcţiilor de prelucrare ale sistemului.

Sunt trei tipuri majore de instrucţiuni folosite de un calculator: de *transfer de date*, de *prelucrare date* şi de *control al programelor*.

Definirea setului de instrucţiuni şi modul lor de prelucrare caracterizează în general puterea unui calculator.

Pe baza acestei structuri au fost construite calculatoarele din prima generaţie (anii $'40 - '50$).

Structura unui astfel de calculator (cum au fost ENIAC, EDVAC, IAS, UNIVAC) este

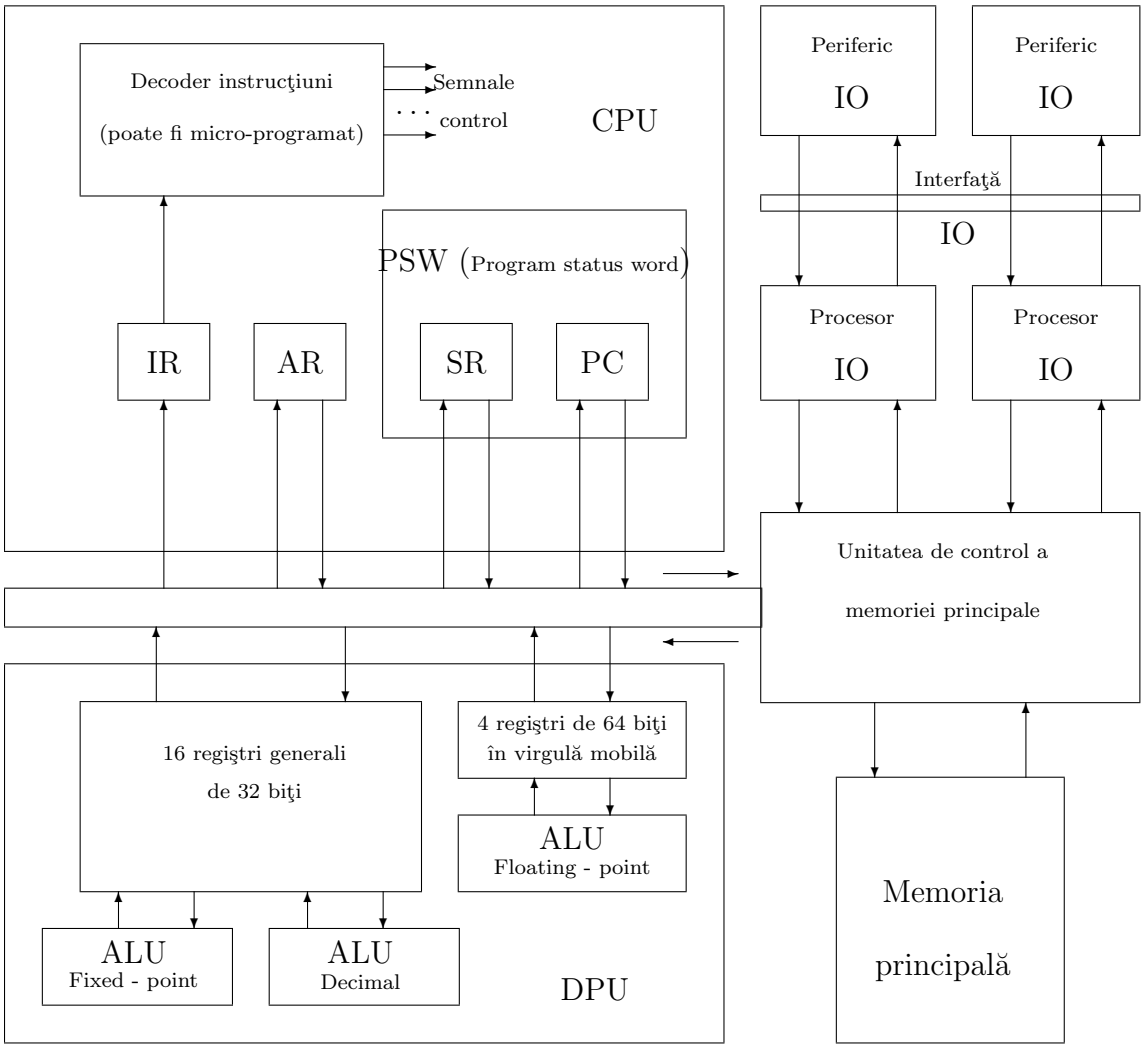


Impactul cel mai mare asupra dezvoltării sistemelor de calcul l-au avut însă calculatoarele din generaţia a treia, reprezentantul specific fiind $IBM/360$ (existent şi în dotarea Universităţii Bucureşti, începând cu anul 1967).

Schema generală a structurii unui astfel de calculator este prezentată pe pagina următoare.

Notaţiile folosite (devenite standard odată cu generaţia a treia de calculatoare):

- DPU - Unitatea de prelucrare a datelor
- CPU - Unitatea de control a programului
- ALU - Unitatea aritmetică şi logică
- IO - Intrare / ieşire
- SR - Registru de stări
- IR - Registru de instrucţiuni
- AR - Registru de calcul pentru instrucţiuni aritmetice
- PC - Registru de control al programului

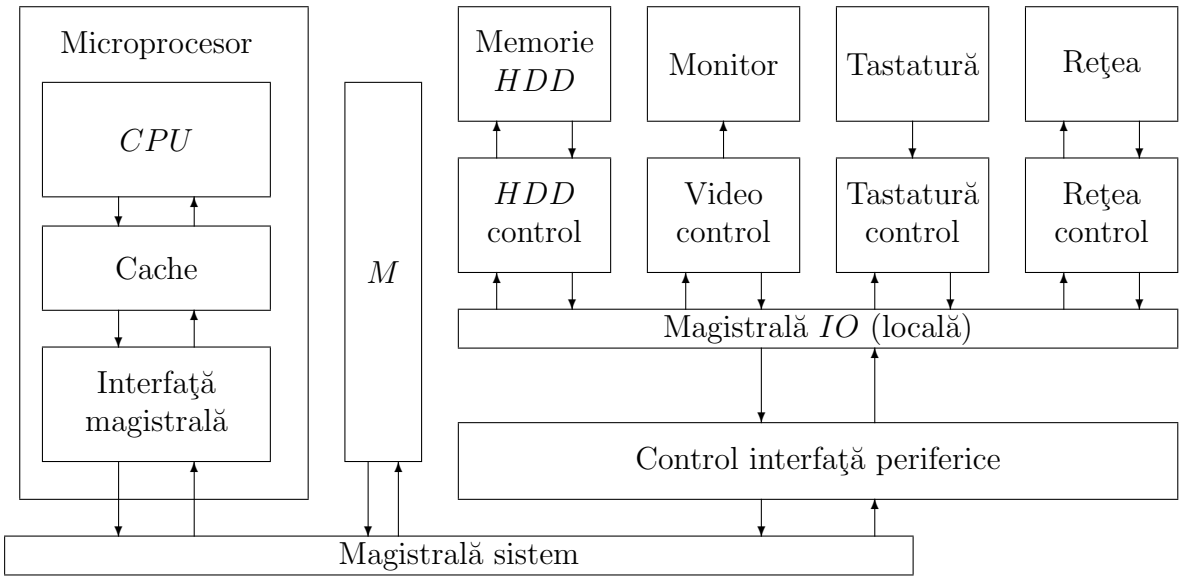


Caracteristici ale calculatoarelor din generaţia 3 ($IBM/360 - IBM/390$)

- Introduc ideea de standardizare a construcţiei, pentru a accepta limbaje de programare universale.
- Constituie baza arhitecturii calculatoarelor actuale.
- Introduc unităţile de informaţie: octet (byte), cuvânt (word), cuvânt dublu, registru.
- Introduc ideea de micro - programare.

Sistemul actual uzual de calcul este computerul, tratat ca unitate independentă (*PC* sau staţie de lucru) dedicată unui singur utilizator.

Structura lui este:



Principalul element hard este un *microprocesor* realizat pe un chip, care conţine o versiune a arhitecturii definite de von Neumann.

El asigură un CPU şi este responsabil cu aducerea, decodificarea şi executarea instrucţiunilor.

Datele şi instrucţiunile au un format standard de grupuri formate din 32 biţi (*cuvinte*); acestea sunt unităţile de bază prelucrate de computer.

CPU este caracterizat printr-un set de circa 200 instrucţiuni care realizează transferul şi prelucrarea datelor precum şi operaţii de control al programelor (cu o structură rămasă în general aceeaşi în timp).

CPU poate fi suplimentată cu alt chip (interior sau exterior) numit *co-procesor*, care implementează funcţii specializate, cum ar fi prelucrarea de interfeţe grafice.

Rolul memoriei principale M este de a stoca programe şi date prelucrate de CPU.

Ea este o memorie cu acces aleator (RAM - random access memory), formată din o secvenţă liniară de componente (de obicei grupuri de 8 biţi, numite *bytes* sau *octeţi*).

Fiecare byte are asociat o adresă unică, care permite CPU să citească sau să schimbe (scrie) conţinutul (pe baza unor instrucţiuni de încărcare sau memorare).

Memoria principală M este completată de o memorie secundară, mai mare dar mai lentă (din punct de vedere al accesului), de obicei situată pe hardiscuri (HDD).

Harddiscurile formează o componentă a sistemului de intrare/ieşire (IO).

De asemenea, o memorie intermediară, numită *cache* poate fi inserată între CPU şi M.

Deci un calculator conţine mai multe tipuri de memorie, care pot fi ierarhizate în registrele CPU: *memoria cache, memoria principală* şi *memoria secundară*.

Această structură complexă rezultă din faptul că cele mai rapide componente de memorie sunt şi cele mai scumpe; deci ierarhizarea asigură CPU cu un acces rapid la un număr mare de date, la un preţ relativ scăzut.

Scopul unui sistem IO este de a permite utilizatorului să comunice cu calculatorul.

Componentele IO sunt ataşate la calculatorul gazdă prin intermediul unor porturi IO, a căror funcţie este de a controla transferul de date între componenta de intrare/ieşire şi memoria principală.

O astfel de componentă are asignată un set de adrese tip memorie care permit utilizarea de instrucţiuni de intrare/ieşire ce pot fi implementate aproape identic cu instrucţiunile de încărcare respectiv stocare.

Dar deoarece operaţiile de intrare/ieşire sunt foarte lente, CPU are nevoie de un timp mult mai lung pentru a accesa un cuvânt aflat în sistemul IO decât unul din memoria M.

Componentele tradiţionale de intrare/ieşire sunt *tastatura* respectiv *ecranul de monitor*, adaptate în special pentru procesarea informaţiei de tip text.

Adăugarea unei componente punctuale, tip *mouse* permite introducerea ecranului în lista componentelor de intrare; el asigură comunicarea utilizator - computer via imagini grafice. Interfeţe audio pentru generarea şi recunoaşterea sunetelor extind aria de lucru a calculatorului în sistemele multimedia.

1.3 Micro-controllere

Reducerea impresionantă a dimensiunilor şi scăderea costurilor componentelor principale ale unui calculator a permis construirea unor computere specializate pentru anumite probleme, numite *micro-controllere*.

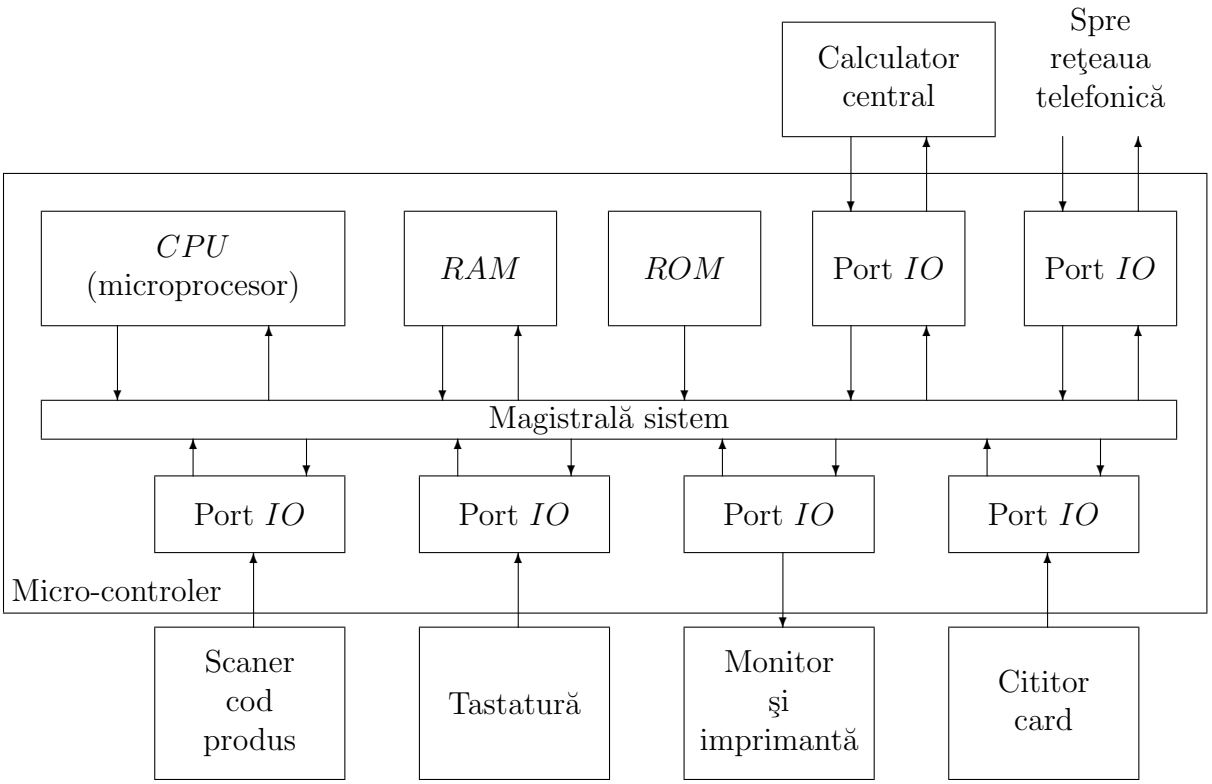
Ele au circuite de control specifice, sau sunt lipsite total de componenta logică (cum ar fi de exemplu circuitul de control al unei maşini de spălat automate).

Programele sunt stocate în memoria *ROM* ("read only memory"), care formează o parte a memoriei principale folosite de micro-controller.

Acesta este construit direct în obiectul controlat, într-o zona adesea invizibilă şi inaccesibilă utilizatorului.

Un micro-controller programat să efectueze operaţiile unei aplicaţii poate înlocui circuitele de control specifice acelei aplicaţii, adesea cu o reducere de cost apreciabilă.

Majoritatea calculatoarelor actual operabile sunt micro-controllere construite în diverse aparate utilizate în cele mai variate medii.



Schema de mai sus descrie una din primele aplicaţii ale unui micro-controller: un punct de vânzare terminal care înlocuieşte casa într-un supermarket.

Micro-controllerul folosit are structura unui computer convenţional.

El este construit în jurul unei magistrale de sistem de care sunt ataşate un microprocesor (cu rol de CPU), cel puţin un chip ROM pentru stocarea programului şi cel puţin un chip RAM pentru lucru şi stocarea datelor.

Toate perifericele IO sunt conectate la sistem folosind porturi IO cu interfeţe standard.

Perifericele tipice unei astfel de aplicaţii sunt: *tastatura*, o *imprimantă* pentru eliberarea chitanţelor, un *monitor*, un *scaner* pentru codurile de bare ale produselor şi un *cititor de card* (credit sau debit).

Acest ultim periferic necesită o conectare la o reţea telefonică pentru autorizarea cardului.

Ultima componentă este o legătură la un calculator central, folosit pentru informaţii de preţ, controlul stocului de produse etc.

1.4 Ce înseamnă "Arhitectura unui Calculator"

Termenul de "*Arhitectură a calculatorului*" a fost folosit prima oară de firma IBM în 1964.

Necesitatea lui a apărut deoarece în acea perioadă tehnologia hardware permitea o avansare rapidă a proceselor de calcul, înlocuind frecvent o structură cu alta; tehnologia software nu avea posibilitatea de a ţine pasul, orice nouă structură hard necesitând programe noi.

Pentru a avea posibilitatea de a folosi la noile componente hard coduri program deja scrise, ceva trebuia menţinut stabil.

Corpul de cercetători ai firmei IBM (leaderul necontestat în domeniul tehnicii de calcul din acea perioadă) a decis stabilizarea setului de operaţii elementare necesare programatorului.

Definiţia 1.2 *Arhitectura unui calculator este imaginea logică şi funcţională a unui computer cu care lucrează un programator.*

Odată această arhitectură stabilită, cele două componente ale unui computer: hardware (mulţimea pieselor care formează calculatorul) şi software (mulţimea programelor rulate de calculator) se pot dezvolta independent una de alta.

O bună perioadă de timp arhitectura putea fi doar îmbunătăţită, fiind adăugate noi facilităţi, dar nefiind eliminată nici o funcţie.

Deci un program scris pentru o primă versiune a arhitecturii va funcţiona pe orice versiune ulterioară.

Rezumând, architectura calculatorului este o interfaţă între hardul şi softul unui computer şi presupune:

- o maşină capabilă să realizeze o mulţime finită de operaţii elementare,
- o memorie, conţinând o descriere a modului cum pot fi folosite funcţiile elementare pentru a realiza procese de calcul,
- un mecanism de control care permite utilizarea descrierii pentru a pune maşina să lucreze, în două variante:
  - **executînd** descrierea
  - **interpretînd** descrierea.

unde, prin **execuţie** se înţelege o acţiune de un "pas" asociată fiecărui "element" al descrierii, iar prin **interpretare** se înţelege o secvenţă de acţiuni asociate fiecărui "element" al descrierii.

Evident, în ambele variante, acţiunile vor depinde direct de modul de definire al descrierii.

Lucrarea de faţă abordează elementele fundamentale necesare construirii arhitecturii unui calculator. Sunt schiţate noţiunile teoretice şi structurile celor mai simple circuite care formează unitatea logico-aritmetică, unitatea centrală de prelucrare precum şi componentele de memorie.

Celor care vor să aprofundeze domeniul le recomand referinţele bibliografice precum şi numeroase site-uri de Internet dedicate sistemelor de calcul.

Capitolul 2

Reprezentarea numerelor în calculator

2.1 Coduri de reprezentare a datelor

Utilizarea codurilor a apărut pentru a asigura o comunicare între un utilizator şi un sistem de calcul (specific unui calculator); principalul obstacol – care este depăşit printr-un cod de reprezentare a datelor – constă în corelarea dintre modul în care sunt prelucrate datele de către o persoană fizică (care gândeşte în sistemul zecimal) şi un computer (unde sistemul de bază este cel binar).

Definiţia 2.1 *Fiind date două mulţimi finite şi nevide A (alfabet sursă) şi B (alfabet cod), o* **codificare** *este o aplicaţie injectivă*[1] $\phi : A \longrightarrow B^+$.

Observaţia 2.1 *S-a notat cu B^+ mulţimea secvenţelor nevide de caractere formate cu elemente din B. De exemplu, dacă $B = \{0, 1\}$ atunci $B^+ = \{0, 1, 00, 01, 10, 11, 000, \ldots\}$.*

Mulţimea $C = \phi(A)$ se numeşte *cod*, iar elementele sale sunt *cuvinte - cod*.

Dacă B are numai două simboluri, codificarea ϕ se numeşte *binară*, iar $\phi(A)$ este un *cod binar*.

Deşi simple, codurile binare sunt de obicei lungi şi deci greu de manipulat.

Adesea este mai convenabil să grupăm simbolurile binare formând alfabete mai complexe.

Astfel, formând grupuri de câte patru simboluri binare, se obţine *codul hexazecimal*:

[1] O aplicaţie f este injectivă dacă din egalitatea $f(x) = f(y)$ rezultă $x = y$ şi nimic altceva.

0	0000	4	0100	8	1000	C	1100
1	0001	5	0101	9	1001	D	1101
2	0010	6	0110	A	1010	E	1110
3	0011	7	0111	B	1011	F	1111

Reprezentarea în această bază (care foloseşte simbolurile auxiliare A, B, C, D, E, F pentru numerele $10, 11, 12, 13, 14$ şi respectiv 15) se indică adesea prin indicele 16 aşezat la sfârşit. De exemplu,

$$(61)_{16} = (0110\ 0001)_2$$

Dacă nu există nici o confuzie, atunci indicii care semnalează baza se ignoră.

Utilizarea acestui cod este aproape universal acceptată.

Astfel, memoria calculatorului lucrează cu biţi grupaţi în *bytes* (sau octeţi).

Un octet se reprezintă de obicei nu ca o secvenţă de 8 cifre binare, ci ca o pereche de cifre hexazecimale.

Astfel, de exemplu $(00101101)_2$ se scrie $(2D)_{16}$, $(01100000)_2$ se reprezintă prin $(60)_{16}$ etc.

2.1.1 Coduri ponderate şi neponderate

Codurile binare sau hexazecimale nu sunt singurele modalităţi de codificare a numerelor în calculator.

Practic, plecând de la reprezentarea binară, se poate construi o paletă largă de coduri.

Ele sunt clasificate în

- coduri ponderate;
- coduri neponderate.

Codurile ponderate asociază unei cifre zecimale un grup de patru cifre binare, fiecărei cifre binare fiindu-i asociată o "pondere" corespunzătoare puterii lui 2 din cadrul codului.

Codul hexazecimal - binar prezentat mai sus este un exemplu de cod ponderat. El mai este numit adesea şi *codul* 8421.

Codurile neponderate asociază unei cifre zecimale un grup de patru cifre binare, fără a avea semnificaţia unei ponderi.

Cele mai cunoscute astfel de coduri sunt codurile *Exces 3* şi *Gray*.

Codul Exces 3 este rezultat din codul hexazecimal prin adunarea numărului 3 (scris în binar) la fiecare cifră.

În acest fel, prin complementarea cifrelor binare care formează codul cifrei i se obţine codul cifrei $9 - i$.

Pentru codul Gray, caracteristic este faptul că orice codificare a unui număr diferă de codificarea succesorului său prin exact o cifră binară.

Cele două coduri sunt reprezentate în tabelul următor:

$Cifra$	0	1	2	3	4	5	6	7	8	9
$Exces\ 3$	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100
$Gray$	0000	0001	0011	0010	0110	0111	0101	0100	1100	1101

2.1.2 Reprezentarea zecimal codificat binar

Aproape toate calculatoarele folosesc pentru memorarea cifrelor zecimale reprezentarea în cod 8421.

Însă, deoarece sistemele de calcul lucrează cu octeţi, deşi sunt suficiente 4 biţi pentru o cifră zecimală, se folosesc două modalităţi de codificare:

- Forma *condensată* (împachetată), în care, pe cele 8 poziţii ale unui octet sunt reprezentate două cifre zecimale exprimate în binar;
- Forma *dilatată* (despachetată), în care pe fiecare octet se reţine o singură cifră zecimală – pe ultimii patru poziţii binare.

  Primele patru poziţii ale fiecărui octet sunt ocupate de o *marcă* M.

Exemplul 2.1 *Numărul* 342 *este memorat pe doi octeţi în forma împachetată*

0	3	4	2
0 0 0 0	0 0 1 1	0 1 0 0	0 0 1 0

şi pe trei octeţi în forma despachetată:

	3		4		2
M	0 0 1 1	M	0 1 0 0	M	0 0 1 0

2.1.3 Codul ASCII

Char	Cod	Char	Cod	Char	Cod
	$(20)_{16}$	@	$(40)_{16}$	‘	$(60)_{16}$
!	$(21)_{16}$	A	$(41)_{16}$	a	$(61)_{16}$
”	$(22)_{16}$	B	$(42)_{16}$	b	$(62)_{16}$
#	$(23)_{16}$	C	$(43)_{16}$	c	$(63)_{16}$
$	$(24)_{16}$	D	$(44)_{16}$	d	$(64)_{16}$
%	$(25)_{16}$	E	$(45)_{16}$	e	$(65)_{16}$
&	$(26)_{16}$	F	$(46)_{16}$	f	$(66)_{16}$
′	$(27)_{16}$	G	$(47)_{16}$	g	$(67)_{16}$
(	$(28)_{16}$	H	$(48)_{16}$	h	$(68)_{16}$
)	$(29)_{16}$	I	$(49)_{16}$	i	$(69)_{16}$
*	$(2A)_{16}$	J	$(4A)_{16}$	j	$(6A)_{16}$
+	$(2B)_{16}$	K	$(4B)_{16}$	k	$(6B)_{16}$
‘	$(2C)_{16}$	L	$(4C)_{16}$	l	$(6C)_{16}$
−	$(2D)_{16}$	M	$(4D)_{16}$	m	$(6D)_{16}$
.	$(2E)_{16}$	N	$(4E)_{16}$	n	$(6E)_{16}$
/	$(2F)_{16}$	O	$(4F)_{16}$	o	$(6F)_{16}$
0	$(30)_{16}$	P	$(50)_{16}$	p	$(70)_{16}$
1	$(31)_{16}$	Q	$(51)_{16}$	q	$(71)_{16}$
2	$(32)_{16}$	R	$(52)_{16}$	r	$(72)_{16}$
3	$(33)_{16}$	S	$(53)_{16}$	s	$(73)_{16}$
4	$(34)_{16}$	T	$(54)_{16}$	t	$(74)_{16}$
5	$(35)_{16}$	U	$(55)_{16}$	u	$(75)_{16}$
6	$(36)_{16}$	V	$(56)_{16}$	v	$(76)_{16}$
7	$(37)_{16}$	W	$(57)_{16}$	w	$(77)_{16}$
8	$(38)_{16}$	X	$(58)_{16}$	x	$(78)_{16}$
9	$(39)_{16}$	Y	$(59)_{16}$	y	$(79)_{16}$
:	$(3A)_{16}$	Z	$(5A)_{16}$	z	$(7A)_{16}$
;	$(3B)_{16}$	[	$(5B)_{16}$	{	$(7B)_{16}$
<	$(3C)_{16}$	\	$(5C)_{16}$	—	$(7C)_{16}$
=	$(3D)_{16}$	]	$(5D)_{16}$	}	$(7D)_{16}$
>	$(3E)_{16}$	ˆ	$(5E)_{16}$	˜	$(7E)_{16}$
?	$(3F)_{16}$	_	$(5F)_{16}$	DEL	$(7F)_{16}$

Un cod foarte important folosit în reprezentarea standard a simbolurilor alfabetice şi numerice este codul *ASCII (***A***merican* **S***tandard* **C***ode for* **I***nformation* **I***nterchange).*

El are $2^8 = 256$ simboluri sursă, codificate în secvenţe binare de lungime 8; tabelul de mai sus cuprinde simbolurile printabile având codurile ASCII în intervalul $[32, 127]$ (în

domeniul $[0, 31]$ nu sunt simboluri printabile, iar în zona $[128, 255]$ se definesc simboluri auxiliare, care nu sunt pe tastatură).

De exemplu, litera A, va avea codul hexazecimal - binar

$$(41)_{16} = (01000001)_2$$

căruia în zecimal îi corespunde numărul 65.

De remarcat modalitatea diferită de codificare a cifrelor zecimale.

În codul ASCII se codifică *caracterele*; deci – în această codificare, cifrele $0, 1, \ldots, 9$ sunt considerate caractere şi prelucrate ca atare.

De exemplu, nu se pot face operaţii aritmetice obişnuite cu ele, ci doar cu codurile ASCII asociate.

Codul ASCII nu este singurul cod construit pentru caractere.

Cel mai cuprinzător cod folosit – care cuprinde caracterele şi semnele tuturor alfabetelor utilizate pe tastaturi este *Unicode*.

Folosind o notare numerică pe 32 sau 16 biţi, el oferă un cod unic pentru fiecare caracter, indiferent de platforma folosită, de program sau limbaj.

Unicode este solicitat drept standard de XML, Java, ECMAScript (JavaScript), LDAP, CORBA 3.0, WML etc.

Pentru orice detaliu privind versiunile Unicode, se poate folosi site-ul

http://www.unicode.org.

2.2 Reprezentarea numerelor în sistemele de calcul

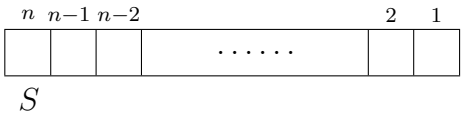
În calculatoare, sistemul de numeraţie utilizat este cel binar.

Reprezentarea numerelor în acest sistem se face în mai multe forme, în funcţie de tipul lor; pentru fiecare tip, operaţiile aritmetice elementare sunt efectuate de către dispozitive speciale înglobate în arhitectura calculatorului (ceea ce asigură o viteză sporită de calcul).

2.2.1 Reprezentarea numerelor întregi

Reprezentarea în memoria calculatorului a numerelor întregi (numită şi *reprezentare algebrică*) se realizează pe un număr fix de n poziţii binare (de regulă $n = 8, 16, 32, 64$ sau 128).

Această reprezentare poate fi schematizată în modul următor:



Prima poziţie din stânga (a n-a poziţie binară) este numită *bit de semn*; ea este ocupată de un bit care are valoarea $'0'$ dacă numărul este nenegativ, $'1'$ dacă numărul este negativ.

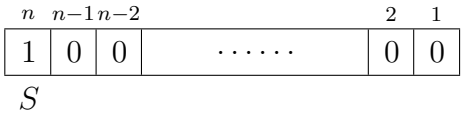
A i-a cifră binară este coeficientul lui 2^{i-1} în reprezentarea numărului în baza 2.

Deci cel mai mare număr întreg care se poate reprezenta pe n biţi este

$$1 \cdot 2^{n-2} + 1 \cdot 2^{n-3} + \ldots + 1 \cdot 2^1 + 1 \cdot 2^0 = 2^{n-1} - 1$$

De exemplu, pentru $n = 16$, acesta este $2^{15} - 1 = 32.767$.

În schimb, cel mai mic număr care poate fi memorat este -2^{n-1}; el are reprezentarea



În cazul $n = 16$, acesta este $-2^{15} = -32.768$.

Deci pe 16 poziţii binare pot fi reprezentate toate numerele întregi din intervalul $[-32.768, 32.767]$.

2.2.2 Operaţii cu numere întregi

Adunarea a două numere întregi se efectuează în binar, poziţie cu poziţie, de la dreapta spre stânga; dispozitivul aritmetic al sistemelor de calcul care efectuează adunarea se numeşte *sumator*.

Exemplul 2.2 *Să efectuăm adunarea* $23+15$ *folosind reprezentarea pe* 8 *cifre binare (un octet). Efectuarea acestei operaţii este redată de figura următoare:*

0	0	0	1	0	1	1	1	23
								+
0	0	0	0	1	1	1	1	15
								=
0	0	1	0	0	1	1	0	38

Pentru scădere se foloseşte acelaşi dispozitiv aritmetic care realizează adunarea.

Motivaţia constă în faptul că orice scădere se poate scrie sub forma unei adunări $x - y = x + (-y)$.

Pentru aceasta este necesară o reprezentare a numerelor negative astfel încât să permită aplicarea relaţiei de mai sus.

Sunt cunoscute trei astfel de reprezentări:

1. *Cod direct*, care coincide cu reprezentarea algebrică (folosind bitul de semn);
2. *Cod invers* (sau *cod complementar la* 1): se obţine prin înlocuirea fiecărei cifre binare a cu $1 - a$;
3. *Cod complementar (la* 2*)*: se obţine din codul invers, la care – în faza a doua – se adună numărul 1.

Exemplul 2.3 *Să reprezentăm în fiecare din cele trei forme, pe* 16 *biţi (doi octeţi), numărul* -320.
(1) *Deoarece* $320 = (101000000)_2$, *reprezentarea lui* -320 *în cod direct este:*

1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0

(2) *Codul invers se obţine din reprezentarea numărului* 320, *prin complementarea tuturor cifrelor binare:*

1	1	1	1	1	1	1	0	1	0	1	1	1	1	1	1

(3) *Pentru codul complementar, se adună* 1 *la codul invers:*

1	1	1	1	1	1	1	0	1	1	0	0	0	0	0	0

Scăderea se transformă în adunare prin reprezentarea scăzătorului în cod invers sau cod complementar.

Deoarece marea majoritate a implementărilor folosesc codul complementar, ne referim la această manieră de operare.

Motivarea matematică este foarte simplă.

Să presupunem că avem un număr întreg, reprezentat în binar (pe n biţi) sub forma $a_n a_{n-1} \dots a_2 a_1$, unde

$$a = \sum_{i=1}^{n-1} a_i 2^{i-1}$$

iar a_n este bitul de semn.

Atunci reprezentarea lui a în cod complementar este

$$\overline{a} = \sum_{i=1}^{n-1} (1 - a_i) 2^{i-1} + 1 = \sum_{i=1}^{n-1} 2^{i-1} - \sum_{i=1}^{n-1} a_i 2^{i-1} + 1 = 2^{n-1} - 1 - a + 1 = 2^{n-1} - a$$

Deci

$$a = 2^{n-1} - \overline{a}$$

şi diferenţa $b - a$ se poate scrie

$$b - a = b + \overline{a} - 2^{n-1}$$

De remarcat că numărul 2^{n-1} nu afectează poziţiile binare ale numărului, ci numai bitul de semn.

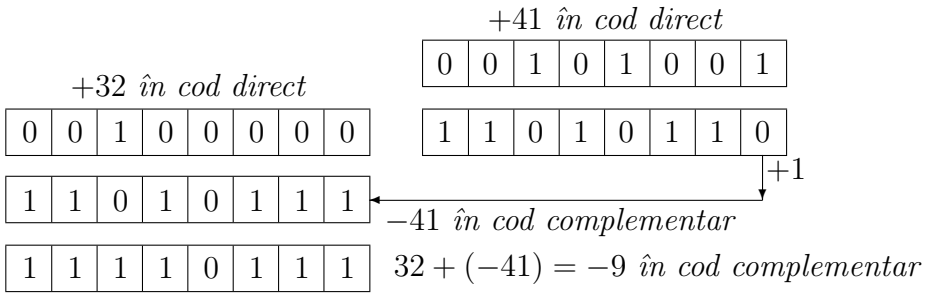
Revenind la algoritm, pentru scăderea $b - a$ se efectuează următorii paşi:

1. Se trece a în cod complementar;
2. Se face adunarea dintre b şi a scris în noua reprezentare;
3. Dacă rezultatul este negativ, acesta este reprezentat tot în cod complementar;
4. Dacă apare transport la poziţia alocată semnului, acesta se ignoră.

Exemplul 2.4 *Să se efectueze* $32 - 41$ *cu reprezentare în cod complementar a scăzătorului, pe* 8 *cifre binare.*

$$32 = (100000)_2, \qquad 41 = (101001)_2.$$

Efectuarea operaţiei de scădere este redată prin schema următoare:



2.2.3 Reprezentarea numerelor reale în virgulă mobilă

Numerele reale se reprezintă în interiorul unui sistem de calcul sub formă fracţionară, într-o codificare numită *virgulă mobilă* (floating point).

În prima fază, orice număr este adus la forma

$$N = \pm 1, f \times 2^{\pm e}$$

unde:

- N este numărul care trebuie reprezentat în virgulă mobilă,
- f - partea fracţionară a lui N,
- $\pm 1, f$ se numeşte *mantisă* şi trebuie să respecte relaţia de normalizare
$$1 \leq |1, f| < 2;$$
- 2 este baza sistemului de numeraţie (sistemul binar),
- $\pm e$ este exponentul bazei sistemului de numeraţie.

Exemplul 2.5 *Numărul* $N = 25$ *se scrie în virgulă mobilă sub forma*

$$25 = (11001)_2 = 1,1001 \times 2^4.$$

Exemplul 2.6 *Numărul* $\dfrac{31}{32}$ *se va scrie sub forma unui număr în virgulă mobilă astfel:*

$$\frac{31}{32} = \frac{2^4 + 2^3 + 2^2 + 2^1 + 2^0}{2^5} = 2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} + 2^{-5} = (0,11111)_2 = 1,1111 \times 2^{-1}.$$

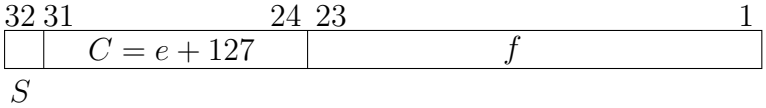
Exemplul 2.7 *Pentru* $N = -\frac{1}{8}$ *reprezntarea în virgulă mobilă este:* $-\frac{1}{8} = -2^{-3} = (-0,001)_2 = -1,0 \times 2^{-3}$.

De observat că – practic – inegalitatea de normalizare coincide cu poziţionarea virgulei în cadrul numărului binar după prima cifră 1, concomitent cu modificarea corespunzătoare a exponentului bazei.

Reprezentarea numerelor în virgulă mobilă are două forme standard:

1. reprezentarea în virgulă mobilă simplă precizie;
2. reprezentarea în virgulă mobilă dublă precizie.

Reprezentarea în simplă precizie se face pe un cuvânt (32 poziţii binare) sub forma:



Prima poziţie (S) reprezintă semnul numărului N; el este codificat cu 0 dacă $N > 0$ şi 1 dacă $N < 0$. Pentru cazul $N = 0$, S este nedeterminat.

Următoarele opt poziţii binare sunt ocupate de o constantă numită *caracteristică* (C), construită în aşa fel încât să codifice exponentul împreună cu semnul său.

Caracteristica este calculată după formula

$$C = e + 127$$

Dacă $C \geq 127$, atunci $e \geq 0$, iar pentru $C < 127$, exponentul va fi negativ.

Valoarea maximă a caracteristicii este $2^8 - 1 = 255$, ceea ce înseamnă că cel mai mare exponent admis de reprezentarea în simplă precizie este $e = 255 - 127 = 128$.

Ultimele 23 poziţii binare sunt alocate reprezentării fracţiei aliniată la stânga şi completată (eventual) cu zerouri (nesemnificative).

Exemplul 2.8 *Pentru reprezentarea în virgulă mobilă simplă precizie a numărului din Exemplul 2.5, deoarece* $25 = 1,1001 \times 2^4$*, vom avea* $S = 0$, $f = 1001$ *şi* $C = e + 127 = 4 + 127 = 131 = (10000011)_2$.

Deci

32	31							24	23															1
0	1	0	0	0	0	0	1	1	1	0	0	1	0	0	0	0	.....	0	0					

Exemplul 2.9 *Pentru* $\frac{31}{32} = 1,1111 \times 2^{-1}$ *(Exemplul 2.6) avem* $S = 0,\ f = 1111,$ $C = -1 + 127 = 126 = (01111110)_2,$ *deci*

32	31							24	23															1
0	0	1	1	1	1	1	1	0	1	1	1	1	0	0	0	0	.....	0	0					

Exemplul 2.10 *Numărul* $-\frac{1}{8} = -1,0 \times 2^{-3}$ *(Exemplul 2.7) are* $S = 1,\ f = 0$ *şi* $C = -3 + 127 = 124 = (01111100)_2,$ *deci*

32	31							24	23															1
1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	.....	0	0					

Reprezentarea în virgulă mobilă dublă precizie se realizează pe 64 poziţii binare (cuvânt dublu), a cărui structură este

64	63											53	52											1
	$C = e + 1023$												f											
S																								

În acest caz, caracteristica este reprezentată pe 11 cifre binare, având posibilitatea de a înmagazina o valoare maximă $2^{11} - 1 = 2047$, ceea ce corespunde unui exponent $e = 2047 - 1023 = 1024$.

Partea fracţionară ocupă 52 poziţii binare.

Exemplul 2.11 *Să reprezentăm în virgulă mobilă dublă precizie numărul* $-124,0625$. *Vom avea* $124,0625 = (1111100,0001)_2 = 1,1111000001 \times 2^6$*, deci* $S = 1,\ f = 1111000001,$ $C = e + 1023 = 6 + 1023 = 1029 = (10000000101)_2.$

Reprezentarea este

64	63											53	52											1
1	1	0	0	0	0	0	0	0	0	1	0	1	1	1	1	0	0	0	0	0	1	0	0	.....	0

La ambele forme de reprezentare în virgulă mobilă, cifra 1 dinaintea virgulei nu se reprezintă; observaţia este utilă atunci când se doreşte obţinerea numărului real din una din formele de reprezentare în virgulă mobilă.

Exemplul 2.12 *Un număr în virgulă mobilă are forma:*

$$11000010\ 11001111\ 00001111\ 00100001$$

Urmărind forma de reprezentare, se constată că:
- Prima cifră $S = 1$ *indică un număr negativ;*
- Următoarele 8 *cifre binare specifică valoarea caracteristicii*
$C = 10000101 = 133$, *deci* $e = 133 - 127 = 6$*;*
- Partea fracţionară ocupă ultimele 23 *cifre binare* $f = 10011110000111100100001$.
Deci numărul este
$N = -1,10011110000111100100001 \times 2^6 = -1100111,10000111100100001 = -103,5296.$

2.2.4 Operaţii în virgulă mobilă

Adunarea şi scăderea a două numere în virgulă mobilă se efectuează astfel:

1. Se compară cei doi exponenţi pentru a-l determina pe cel mai mare;
2. Se aliniază mantisa numărului cu exponent mai mic, prin deplasarea virgulei corespunzătoare exponentului mai mare;
3. Se adună (scad) mantisele aliniate, păstrând exponentul comun;
4. Eventual se normalizează mantisa, concomitent cu modificarea rezultatului.

Exemplul 2.13 *Să se efectueze adunarea în virgulă mobilă* $\frac{3}{4} + 7$.
$x = \frac{3}{4} = \frac{2^1 + 2^0}{2^2} = 2^{-1} + 2^{-2} = (0,11)_2 = 1,1 \times 2^{-1}$,
$y = 7 = (111)_2 = 1,11 \times 2^2$.
Deoarece y *are exponentul mai mare,* x *se va alinia corespunzător:*

$$x = 1,11 \times 2^{-1} = 0,0011 \times 2^2$$

Acum
$x + y = (0,0011 + 1,11) \times 2^2 = 1,1111 \times 2^2$.
Nu este necesară normalizarea mantisei.

Exemplul 2.14 *Să calculăm în virgulă mobilă* $34 - 9$*:*
$x = 34 = (100010)_2 = 1,001 \times 2^5$,
$y = 9 = (1001)_2 = 1,001 \times 2^3 = 0,01001 \times 2^5$.
Deci $x - y = (1,00010 - 0,01001) \times 2^5 = 0,11001 \times 2^5$.
Normalizând, se obţine

$$x - y = 1,1001 \times 2^4 = (11001)_2 = 25$$

Operaţiile de înmulţire şi împărţire presupun:

1. Adunarea (scăderea) exponenţilor;
2. Înmulţirea (împărţirea) mantiselor;
3. Eventuala normalizare a mantisei.

Exemplul 2.15 *Să se efectueze în virgulă mobilă* $5 \cdot 9$*:*
$x = 5 = (101)_2 = 1,01 \times 2^2, \quad y = 9 = (1001)_2 = 1,001 \times 2^3$.
$x \cdot y = (1,01 \cdot 1,001) \times 2^{2+3} = 1,01101 \times 2^5 = (101101)_2 = 45$.

Exemplul 2.16 *Să se efectueze în virgulă mobilă* $5 : 10$.
$x = 5 = (101)_2 = 1,01 \times 2^2, \quad 10 = (1010)_2 = 1,01 \times 2^3$.
$x; y = (1,01 : 1,01) \times 2^{2-3} = 1,0 \times 2^{-1} = (0,1)_2 = 0,5$.

2.3 Exerciţii

1. Să se treacă din baza 10 în baza 2 numerele
$$18926; \quad -7772; \quad 7863,15; \quad -0,7.$$

2. Să se treacă din baza 2 în baza 10 numerele
$$1101100010100110; \qquad 100010101001,0011011001.$$

3. Să se efectueze calculele
$$11+32600, \quad 599-10692, \quad 9862+20043-18552.$$

4. Să se definească scăderea a două numere întregi folosind ca reprezentare codul invers.

5. Să se scrie în virgulă mobilă numerele
$$6489; \quad -0,1; \quad \frac{85}{256}.$$

6. Să se reprezinte numerele scrise în virgulă mobilă în exerciţiul anterior, pe
   (1) 32 poziţii binare (simplă precizie);
   (2) 64 poziţii binare (dublă precizie).

7. Ce număr real are reprezentarea în simplă precizie
$$00001001001101011010001101010101 1.$$

8. Aceeaşi problemă pentru numărul a cărui reprezentare în dublă precizie este:
$$1000100100100100001110110101010000100101011010010000101000011010 1.$$

9. Să se efectueze în virgulă mobilă calculele
$$8+\frac{7}{32}; \quad \frac{25}{36}-799; \quad 361\times 12,495; \quad 38398/73.$$

10. Să se detalieze operaţiile de adunare şi scădere cu numere reprezentate în forma condensată.

Capitolul 3

Algebre şi funcţii boolene

3.1 Definirea algebrelor booleene

Elementul teoretic fundamental necesar pentru analiza şi sinteza circuitelor care formează arhitectura unui calculator este cel de *Algebră Boole*[1].

Definiţia 3.1 *O algebră Boole este un sistem* $\mathcal{B}= (B, \vee, \wedge, 0, 1)$ *cu proprietăţile:*

1. $\forall a, b, c \in B \quad (a \vee b) \vee c = a \vee (b \vee c)$; *(asociativitatea lui* $\vee$*)*
2. $\forall a, b, c \in B \quad (a \wedge b) \wedge c = a \wedge (b \wedge c)$; *(asociativitatea lui* $\wedge$*)*
3. $\forall a, b \in B \quad a \vee b = b \vee a$; *(comutativitatea lui* $\vee$*)*
4. $\forall a, b \in B \quad a \wedge b = b \wedge a$; *(comutativitatea lui* $\wedge$*)*
5. $\exists 0 \in B$ *unic astfel încât* $\forall a \in B \quad a \vee 0 = 0 \vee a = a$; *(element unitate pentru* $\vee$*)*
6. $\exists 1 \in B$ *unic astfel încât* $\forall a \in B \quad a \wedge 1 = 1 \wedge a = a$; *(element unitate pentru* $\wedge$*)*
7. $\forall a, b, c \in B \quad a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$; *(distributivitatea lui* $\vee$ *faţă de* $\wedge$*)*
8. $\forall a, b, c \in B \quad a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$; *(distributivitatea lui* $\wedge$ *faţă de* $\vee$*)*
9. $\forall a \in B, \exists \overline{a} \in B \quad a \wedge \overline{a} = 0, \quad a \vee \overline{a} = 1$. *(complement)*

O primă observaţie este că o algebră booleană are cel puţin două elemente 0 şi 1, iar $0 \neq 1$.

[1] George Boole (1815 - 1864): matematician şi filosof englez, considerat – prin prisma algebrelor care îi poartă numele – unul din fondatorii informaticii teoretice.

Exemplul 3.1 *Fie Q o mulţime. Mulţimea $\mathcal{Q}=\{P|P\subseteq Q\}=2^Q$ formează o algebră Boole cu operaţiile de reuniune şi intersecţie.*

Elementul 0 este mulţimea vidă $\emptyset$, iar elementul 1 este mulţimea Q.

Axiomele din Definiţia 3.1 se verifică imediat.

Exemplul 3.2 *Mulţimea $B=\{0,1\}$ cu operaţiile*

$\vee$	0	1
0	0	1
1	1	1

$\wedge$	0	1
0	0	0
1	0	1

formează de asemenea o algebră booleană $\mathcal{B}=(B,\vee,\wedge,0,1)$.

3.2 Proprietăţi ale algebrelor booleene

În acest paragraf vom face o trecere în revistă a principalelor proprietăţi îndeplinite de o algebră Boole.

Teorema 3.1 *Într-o algebră booleană sunt verificate legile de idempotenţă:*

$$a\vee a=a,\quad a\wedge a=a.$$

Demonstraţie: Vom avea
$a\vee a=(a\vee a)\wedge 1=(a\vee a)\wedge(a\vee\overline{a})=a\vee(a\wedge\overline{a})=a\vee 0=a.$

Pentru a doua relaţie vom proceda similar (prin dualitate):
$a\wedge a=(a\wedge a)\vee 0=(a\wedge a)\vee(a\wedge\overline{a})=a\wedge(a\vee\overline{a})=a\wedge 1=a.$

Teorema 3.2 $a\vee 1=1,\quad a\wedge 0=0.$

Demonstraţie:
$a\vee 1=(a\vee 1)\wedge 1=(a\vee 1)\wedge(a\vee\overline{a})=a\vee(1\wedge\overline{a})=a\vee\overline{a}=1,$
$a\wedge 0=(a\wedge 0)\vee 0=(a\wedge 0)\vee(a\wedge\overline{a})=a\wedge(0\vee\overline{a})=a\wedge\overline{a}=0.$

Teorema 3.3 $a\vee(a\wedge b)=a,\quad a\wedge(a\vee b)=a.$ *(absorbţie)*

Demonstraţie:
$a\vee(a\wedge b)=(a\wedge 1)\vee(a\wedge b)=a\wedge(1\vee b)=a\wedge(b\vee 1)=a\wedge 1=a,$
$a\wedge(a\vee b)=(a\wedge a)\vee(a\wedge b)=a\vee(a\wedge b)=a.$

Teorema 3.4 *Pentru orice $a \in B$, $\overline{a}$ este unic.*

Demonstraţie: Presupunem prin absurd că există două complemente $\overline{a}, \overline{a}_1$ ale lui a.

Conform Definiţiei 3.1, avem

$$a \vee \overline{a} = a \vee \overline{a}_1 = 1, \quad a \wedge \overline{a} = a \wedge \overline{a}_1 = 0.$$

Atunci

$\overline{a}_1 = 1 \wedge \overline{a}_1 = (a \vee \overline{a}) \wedge \overline{a}_1 = \overline{a}_1 \wedge (a \vee \overline{a}) = (\overline{a}_1 \wedge a) \vee (\overline{a}_1 \wedge \overline{a}) = 0 \vee (\overline{a}_1 \wedge \overline{a}) = (a \wedge \overline{a}) \vee (\overline{a}_1 \wedge \overline{a}) = (a \vee \overline{a}_1) \wedge \overline{a} = 1 \wedge \overline{a} = \overline{a}$.

Teorema 3.5 $\overline{\overline{a}} = a$.

Demonstraţie: $\overline{\overline{a}}$ este complementul lui $\overline{a}$. Dar şi a este complementul lui $\overline{a}$.

Deoarece complementul este unic (Teorema 3.4), rezultă $\overline{\overline{a}} = a$.

Deci, putem considera complementara ca o aplicaţie bijectivă

$$- : B \longrightarrow B.$$

Teorema 3.6 $\overline{a \vee b} = \overline{a} \wedge \overline{b}, \quad \overline{a \wedge b} = \overline{a} \vee \overline{b}$. *(regulile De Morgan)*

Demonstraţie: Vom demonstra că $(a \vee b) \vee (\overline{a} \wedge \overline{b}) = 1$ şi $(a \vee b) \wedge (\overline{a} \wedge \overline{b}) = 0$; deci $a \vee b$ şi $\overline{a} \wedge \overline{b}$ sunt complementare.

Din Teorema 3.4 rezultă atunci $\overline{a \vee b} = \overline{a} \wedge \overline{b}$.

Vom avea

$(a \vee b) \vee (\overline{a} \wedge \overline{b}) = [(a \vee b) \vee \overline{a}] \wedge [(a \vee b) \vee \overline{b}] = [(b \vee a) \vee \overline{a}] \wedge [a \vee (b \vee \overline{b})] = [b \vee (a \vee \overline{a})] \wedge (a \vee 1) = (b \vee 1) \wedge 1 = 1$ şi

$(a \vee b) \wedge (\overline{a} \wedge \overline{b}) = [a \wedge (\overline{a} \wedge \overline{b})] \vee [b \wedge (\overline{a} \wedge \overline{b})] = [(a \wedge \overline{a}) \wedge \overline{b}] \vee [b \wedge (\overline{b} \wedge \overline{a})] = 0 \vee [(b \wedge \overline{b}) \wedge \overline{a}] = 0$.

Pentru a doua relaţie se procedează similar.

3.3 Alte operaţii booleene

Înafara celor trei operaţii folosite până acum de o algebră booleană $(\vee, \wedge, -)$, mai sunt cunoscute şi alte operaţii.

Astfel, putem enumera:

1. Diferenţa simetrică: $a \oplus b = (a \wedge \overline{b}) \vee (\overline{a} \wedge b)$;
2. Operatorul Sheffer: $a|b = \overline{a \wedge b}$;
3. Echivalenţa: $a \sim b = (a \vee \overline{b}) \wedge (\overline{a} \vee b)$;
4. Implicaţia: $a \rightarrow b = b \vee \overline{a}$.

Observaţia 3.1 *În arhitectura calculatorului, implementarea operatorului* $\oplus$ *este notată cu* XOR*, iar* $|$ *cu* $NAND$*.*

Cei patru operatori au o serie de proprietăţi specifice ale căror demonstraţii le lăsăm ca exerciţiu.

Propoziţia 3.1

1. $0 \oplus 0 = 1 \oplus 1 = 0, \quad 0 \oplus 1 = 1 \oplus 0 = 1$;
2. $\oplus$ *este asociativă şi comutativă;*
3. $a \oplus 0 = a, \quad a \oplus 1 = \overline{a}$;
4. $a \oplus a = 0, \quad a \oplus \overline{a} = 1$;
5. $a \oplus b = c \implies a \oplus c = b$;
6. $a \oplus b = c \implies a \oplus b \oplus c = 0$;
7. $a \wedge (b \oplus c) = (a \wedge b) \oplus (a \wedge c)$.

Propoziţia 3.2

1. $a \sim b = \overline{a \oplus b}$;
2. $a \sim b = \overline{a} \sim \overline{b}$;
3. $a \sim b = (a \to b) \wedge (b \to a)$;

Propoziţia 3.3 *Afirmaţiile*

1. $a \oplus b = 0$;
2. $a \sim b = 1$;
3. $a = b$;

sunt echivalente.

3.4 Funcţii booleene

În continuare vom considera o algebră booleană particulară (vezi şi Exemplul 3.2) $\mathcal{A}= (\{0,1\}, +, \cdot, 0, 1)$ unde

$+$	0	1
0	0	1
1	1	1

$\cdot$	0	1
0	0	0
1	0	1

Mai avem $\overline{0} = 1$, $\overline{1} = 0$. Axiomele unei algebre booleene se verifică imediat.

În mod uzual, operatorul $\cdot$ se omite (similar cu operatorul de înmulţire din matematică).

Vom nota de asemenea $\{0,1\}^n = \underbrace{\{0,1\} \times \{0,1\} \times \ldots \times \{0,1\}}_{n \text{ ori}}$.

Definiţia 3.2 *O funcţie booleană* $f(x_1, x_2, \ldots, x_n)$ *este o aplicaţie*

$$f : \{0,1\}^n \longrightarrow \{0,1\}.$$

Exemplul 3.3 *Pentru* $n = 2$ *se pot construi* 16 *funcţii booleene de două variabile:*

x_1	x_2	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

În general – pentru n oarecare – vom avea:

Propoziţia 3.4 *Pentru orice* $n \geq 1$ *se pot defini* 2^{2^n} *funcţii booleene de* n *variabile.*

Demonstraţie: Este imediată, deoarece $\{0,1\}^n$ are 2^n elemente, iar $\{0,1\}$ – numai două.

Definiţia 3.3 *Fie* $f, g : \{0,1\}^n \longrightarrow \{0,1\}$. *Definim*

- $f + g = h$ *prin* $h(x_1, x_2, \ldots, x_n) = f(x_1, x_2, \ldots, x_n) + g(x_1, x_2, \ldots, x_n)$;
- $fg = h$ *prin* $h(x_1, x_2, \ldots, x_n) = f(x_1, x_2, \ldots, x_n) \cdot g(x_1, x_2, \ldots, x_n)$;
- $\overline{f} = g$ *prin* $g(x_1, x_2, \ldots, x_n) = \overline{f(x_1, x_2, \ldots, x_n)}$.

Exemplul 3.4 *Să considerăm* $n = 2$ *şi funcţiile booleene* (f_5 *şi* f_1 *din Exemplul 3.3):*

f	0	1
0	0	1
1	0	1

g	0	1
0	0	0
1	0	1

Atunci funcţiile $f + g$, fg, $\overline{f}$ *şi* $\overline{g}$ *sunt definite conform următorului tablou:*

x_1	x_2	f	g	$f+g$	fg	$\overline{f}$	$\overline{g}$
0	0	0	0	0	0	1	1
0	1	1	0	1	0	0	1
1	0	0	0	0	0	1	1
1	1	1	1	1	1	0	0

Deci operaţiile cu funcţii sunt definite *punct cu punct*; reprezentarea lor sub formă tabelară constituie o modalitate convenabilă de calcul deoarece folosesc în mod direct formulele din tabelele de operaţii ale algebrei booleene $\mathcal{A}$.

Teorema 3.7 *Fie F_n mulţimea funcţiilor booleene de n ($n \geq 1$) variabile. Sistemul $\mathcal{F}_n = (F_n, +, \cdot, \mathbf{0}, \mathbf{1})$ formează o algebră booleană (algebra Boole a funcţiilor booleene de n variabile).*

Demonstraţie: Axiomele algebrei boolene se verifică uşor, folosind Definiţia 3.3. Funcţia $\mathbf{0}$ este definită

$$\mathbf{0}(x_1, x_2, \ldots, x_n) = 0, \ \forall x_i \in \{0,1\} \ (1 \leq i \leq n),$$

iar funcţia $\mathbf{1}$ prin

$$\mathbf{1}(x_1, x_2, \ldots, x_n) = 1, \ \forall x_i \in \{0,1\} \ (1 \leq i \leq n).$$

Definiţia 3.4 *Fie $\mathcal{F}_n$ algebra Boole a funcţiilor booleene de n variabile. Se numeşte "pondere" o aplicaţie $w : F_n \longrightarrow \mathcal{N}$ definită $w(f) = card(f^{-1}(1))$ (numărul de elemente din $\{0,1\}^n$ care au imaginea 1 prin f).*

Exemplul 3.5 *În Exemplul 3.3 sunt listate elementele algebrei $\mathcal{F}_2$. Ponderile acestor elemente sunt listate în tabelul următor:*

	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
w	0	1	1	2	1	2	2	3	1	2	2	3	2	3	3	4

Următorul rezultat este imediat.

Teorema 3.8

1. $w(f) + w(\overline{f}) = 2^n$;
2. $w(f+g) + w(fg) = w(f) + w(g)$.

Vom încerca în continuare să definim o reprezentare a funcţiilor booleene, utilă în construirea circuitelor liniare.

Această reprezentare se bazează pe noţiunea de *minterm*.

Definiţia 3.5 *Fie n ($n \geq 1$) un număr întreg şi $i \in [0, 2^n-1]$. Considerăm $(i_1, i_2, \ldots, i_n)_2$ reprezentarea binară a lui i:*

$$i = \sum_{k=1}^{n} i_k 2^{n-k} \qquad i_k \in \{0, 1\}.$$

Atunci funcţia minterm $m_i(x_1, x_2, \ldots, x_n) \in F_n$ este definită prin

$$m_i(x_1, x_2, \ldots, x_n) = \begin{cases} 1 & dacă \quad x_1 = i_1,\ x_2 = i_2, \ldots, x_n = i_n \\ 0 & altfel \end{cases}$$

Vom nota în continuare $i = (i_1, i_2, \ldots, i_n)_2$ reprezentarea binară a numărului întreg $i \in [0, 2^n - 1]$.

Propoziţia 3.5

1. $m_i m_j = 0$ *dacă* $i \neq j$.
2. $m_i m_i = m_i$.

Demonstraţie: Imediat.

Teorema 3.9 *Orice funcţie booleană poate fi reprezentată în mod unic ca sumă de mintermi.*

Demonstraţie: Prin inducţie după $k = w(f)$.

$k = 0$: trivial;

$k = 1$: orice astfel de funcţie este un minterm.

$k \to k+1$: Să presupunem că f este o funcţie de pondere $k+1$. Deci există un întreg $i = (i_1, i_2, \ldots, i_n)_2$ astfel ca $f(i_1, i_2, \ldots, i_n) = 1$.

Atunci – conform Teoremei 3.8 – $f(x_1, x_2, \ldots, x_n) = m_i(x_1, x_2, \ldots, x_n) + g(x_1, x_2, \ldots, x_n)$ unde $w(g) \leq k$.

Unicitatea este şi ea imediată.

Corolarul 3.1 *Orice funcţie booleană $f \in F_n$ este de forma*

$$f(x_1, x_2, \ldots, x_n) = \sum_{i \in I} m_i(x_1, x_2, \ldots, x_n)$$

unde $I \subseteq \{0, 1, \ldots, 2^n - 1\}$.

Corolarul 3.2 *Dacă*

$f(x_1, x_2, \ldots, x_n) = \sum_{i \in I} m_i(x_1, x_2, \ldots, x_n)$, $\quad g(x_1, x_2, \ldots, x_n) = \sum_{i \in J} m_i(x_1, x_2, \ldots, x_n)$

atunci

$f(x_1, x_2, \ldots, x_n) + g(x_1, x_2, \ldots, x_n) = \sum_{i \in I \cup J} m_i(x_1, x_2, \ldots, x_n)$,

$f(x_1, x_2, \ldots, x_n) g(x_1, x_2, \ldots, x_n) = \sum_{i \in I \cap J} m_i(x_1, x_2, \ldots, x_n)$,

$f'(x_1, x_2, \ldots, x_n) = \sum_{i \notin I} m_i(x_1, x_2, \ldots, x_n)$.

Exemplul 3.6 *Fie $n = 2$ şi funcţia booleană:*

$$f(x_1, x_2) = m_0(x_1, x_2) + m_2(x_1, x_2) + m_3(x_1, x_2)$$

Un tabel cu valorile tuturor funcţiilor minterm de două variabile şi cu valorile funcţiei f este:

x_1	x_2	$m_0(x_1, x_2)$	$m_1(x_1, x_2)$	$m_2(x_1, x_2)$	$m_3(x_1, x_2)$	f
0	0	1	0	0	0	1
0	1	0	1	0	0	0
1	0	0	0	1	0	1
1	1	0	0	0	1	1

3.5 Forme canonice

Scopul acestui paragraf este de a asocia fiecărei funcţii booleene f o expresie booleană scrisă sub o formă unică. Aceasta se va numi *formă canonică a lui f* sau *forma normal disjunctivă a lui f*.

O consecinţă importantă a acestei reprezentări va fi aceea că putem verifica echivalenţa a două expresii cercetând dacă acestea pot fi reduse la aceeaşi formă canonică (deci nu vom mai face apel la construcţia tabelelor de valori a funcţiei asociate, destul de costisitoare din punct de vedere al complexităţii).

Definiţia 3.6 *Un "literal" este o variabilă sau complementul ei.*

Definiţia 3.7 *Fie $B_n = \{x_1, x_2, \ldots, x_n\}$. O expresie α este un "minterm" sau un "produs fundamental" în algebra Boole $\mathcal{B}_n = (B_n, +, \cdot, 0, 1)$ dacă şi numai dacă este concatenarea a n literali distincţi.*

Exemplul 3.7 *Cei opt mintermi din $\mathcal{B}_3$ sunt*

$$x_1x_2x_3 \qquad \overline{x}_1x_2x_3 \qquad x_1\overline{x}_2x_3 \qquad x_1x_2\overline{x}_3$$
$$\overline{x}_1\overline{x}_2x_3 \qquad \overline{x}_1x_2\overline{x}_3 \qquad x_1\overline{x}_2\overline{x}_3 \qquad \overline{x}_1\overline{x}_2\overline{x}_3$$

Mintermii vor fi notaţi în felul următor: fie $(i_1, i_2, \ldots, i_n) \in \{0, 1\}^n$.

Atunci un minterm standard este

$$x_1^{i_1} x_2^{i_2} \ldots x_n^{i_n}$$

unde

$$x_j^{i_j} = \begin{cases} \overline{x}_j & \text{pentru } i_j = 0 \\ x_j & \text{pentru } i_j = 1 \end{cases} \qquad (1 \leq j \leq n).$$

Exemplul 3.8 $x_1^0x_2^1x_3^1 = \overline{x}_1x_2x_3, \quad x_1^1x_2^0x_3^0 = x_1\overline{x}_2\overline{x}_3$ *etc.*

Observaţia 3.2 *Fiecare minterm* $x_1^{i_1}x_2^{i_2}\dots x_n^{i_n}$ *corespunde funcţiei booleene* $m_i(x_1, x_2, \dots, x_n)$ *unde* $i = (i_1, i_2, \dots, i_n)_2$.

Teorema 3.10 *Dacă* f *este o funcţie booleană de o variabilă* ($f \in F_1$) *atunci*

$$f(x) = xf(1) + \overline{x}f(0).$$

Demonstraţie: Să considerăm cele două valori posibile pe care le poate lua variabila booleană x.

Pentru $x = 1$ vom avea

$$f(1) = 1 \cdot f(1) + 0 \cdot f(0) = f(1).$$

Similar pentru asignarea $x = 0$.

Deci egalitatea se verifică pentru toate asignările posibile ale variabilei.

Următoarea teoremă dă forma canonică pentru orice funcţie booleană de n variabile.

Teorema 3.11 *Dacă* f *este o funcţie booleană de* n *variabile* ($f \in F_n$) *atunci*

$$f(x_1, x_2, \dots, x_n) = \sum_{i_1=0}^{1}\sum_{i_2=0}^{1}\dots\sum_{i_n=0}^{1} f(i_1, i_2, \dots, i_n)x_1^{i_1}x_2^{i_2}\dots x_n^{i_n}.$$

Aceasta este forma normal disjunctivă a funcţiei f.

Demonstraţie: Prin inducţie după numărul n de variabile.

Pentru $n = 1$ avem rezultatul dat de Teorema 3.10.

Să presupunem Teorema 3.11 adevărată pentru $n - 1$ variabile, adică

$f(x_1, x_2, \dots, x_n) = \sum_{i_2=0}^{1}\dots\sum_{i_n=0}^{1} f(x_1, i_2, \dots, i_n)x_2^{i_2}\dots x_n^{i_n}$.

Pe de altă parte, $f(x_1, i_2, \dots, i_n) = x_1^0 f(0, i_2, \dots, i_n) + x_1^1 f(1, i_2, \dots, i_n)$ şi, după înlocuire

$f(x_1, x_2, \dots, x_n) = \sum_{i_2=0}^{1}\dots\sum_{i_n=0}^{1}(x_1^0 f(0, i_2, \dots, i_n) + x_1^1 f(1, i_2, \dots, i_n))x_2^{i_2}\dots x_n^{i_n} =$

$\sum_{i_1=0}^{1}\dots\sum_{i_n=0}^{1} f(i_1, i_2, \dots, i_n)x_1^{i_1}x_2^{i_2}\dots x_n^{i_n}$.

Corolarul 3.3 *Forma normal disjunctivă a unei funcţii booleene este unică.*

Demonstraţie: Să presupunem că există două forme normale diferite corespunzătoare lui f.

Deci există cel puţin un $(i_1, i_2, \dots, i_n) \in \{0, 1\}^n$ pentru care coeficienţii din cele două expresii sunt diferiţi.

Considerăm asignarea definită prin $x_1 = i_1, \dots, x_n = i_n$.

Evaluând prima expresie obţinem $f(i_1, i_2, \ldots, i_n) = 1(0)$, iar pentru a doua expresie $f(i_1, i_2, \ldots, i_n) = 0(1)$, ceea ce contrazice faptul că ambele expresii reprezintă f.

Fiind dată o funcţie booleană f, să vedem cum se poate scrie forma sa normal disjunctivă.

Cel mai sugestiv este să arătăm aceasta printr-un exemplu.

Exemplul 3.9 *Fie $f \in F_3$ definită de următorul tabel:*

x_1	0	0	0	0	1	1	1	1
x_2	0	0	1	1	0	0	1	1
x_3	0	1	0	1	0	1	0	1
f	0	1	0	1	0	1	0	1

Pe linia lui f, primul element este 0.

Deci termenul corespunzător este $f(0,0,0)x_1^0x_2^0x_3^0 = 0 \cdot \overline{x}_1\overline{x}_2\overline{x}_3 = 0$.

Deoarece $a + 0 = a$, contribuţia oricărei coloane unde valoarea lui f este 0 *poate fi ignorată.*

Dacă valoarea lui f este 1*, termenul respectiv va fi păstrat.*

Deci

$f(x_1, x_2, x_3) = x_1^0x_2^0x_3^1 + x_1^0x_2^1x_3^1 + x_1^1x_2^0x_3^1 + x_1^1x_2^1x_3^1 = \overline{x}_1\overline{x}_2x_3 + \overline{x}_1x_2x_3 + x_1\overline{x}_2x_3 + x_1x_2x_3$.

3.6 Inele booleene

Definiţia 3.8 *Un inel boolean este un inel unitar în care orice element este idempotent.*

Fie $\mathcal{B} = (B, \vee, \wedge, 0, 1)$ o algebră Boole, în care definim operaţiile (pentru orice $x, y \in B$):

$$x \oplus y = (x \wedge \overline{y}) \vee (\overline{x} \wedge y), \qquad x \odot y = x \wedge y.$$

Structura $(B, \oplus, \odot)$ formează o structură de inel boolean.

Regulile de comutativitate şi asociativitate faţă de $\odot$ se deduc din postulatele algebrei booleene $\mathcal{B}$.

Din relaţia $x \odot 1 = 1 \odot x = x$ rezultă că 1 este element neutru.

În plus, orice element este idempotent, pentru că $x \odot x = x$.

Pentru $\oplus$: comutativitatea rezultă în mod banal.

Pentru asociativitate:

$(x \oplus y) \oplus z = ((x \oplus y) \wedge \overline{z}) \vee (\overline{x \oplus y} \wedge z) = [((x \wedge \overline{y}) \vee (\overline{x} \wedge y)) \wedge \overline{z}] \vee [\overline{(x \wedge \overline{y}) \vee (\overline{x} \wedge y)} \wedge z] = [(x \wedge \overline{y} \wedge \overline{z}) \vee (\overline{x} \wedge y \wedge \overline{z})] \vee [(\overline{x} \vee y) \wedge (x \vee \overline{y}) \wedge z] = (x \wedge \overline{y} \wedge \overline{z}) \vee (\overline{x} \wedge y \wedge \overline{z}) \vee (\overline{x} \wedge \overline{y} \wedge z) \vee (x \wedge y \wedge z)$, relaţie simetrică în x, y, z. Deci $(x \oplus y) \oplus z = x \oplus (y \oplus z)$.

Elementul nul este 0 pentru că $x \oplus 0 = (x \wedge \overline{0}) \vee (\overline{x} \wedge 0) = (x \wedge 1) \vee 0 = x$.

Fiecare element este propriul său invers, deoarece

$$x \oplus x = (x \wedge \overline{x}) \vee (\overline{x} \wedge x) = 0 \vee 0 = 0.$$

Această relaţie se poate scrie $2x = 0$, ceea ce înseamnă că avem un inel de caracteristică 2 (proprietate care se poate demonstra pentru orice inel boolean).

Distributivitatea se verifică similar:

$x \odot y \oplus x \odot z = ((x \odot y) \wedge \overline{x \odot z}) \vee (\overline{x \odot y} \wedge (x \odot z)) = (x \wedge y \wedge (\overline{x} \vee \overline{z})) \vee ((\overline{x} \vee \overline{y}) \wedge x \wedge z) = (x \wedge y \wedge \overline{z}) \vee (x \wedge \overline{y} \wedge z) = x \wedge [(y \wedge \overline{z}) \vee (\overline{y} \wedge z)] = x \odot (y \oplus z)$.

Teorema 3.12 *Orice inel boolean $(B, +, \cdot)$ se poate structura ca o algebră Boole, cu operaţiile*

$$x \vee y = x + y + x \cdot y, \qquad x \wedge y = x \cdot y.$$

Demonstraţie: Se verifică axiomele Definiţiei 3.1, unde complementara este definită prin $\overline{x} = 1 + x$. În particular, vom avea:

$x \vee \overline{x} = x + \overline{x} + x \cdot \overline{x} = x + (1 + x) + x \cdot (1 + x) = 1 + 4x = 1$,

$x \wedge \overline{x} = x \cdot \overline{x} = x \cdot (1 + x) = x + x = 0$.

Se poate stabili deci o corespondenţă biunivocă între inelele booleene şi algebrele booleene.

Aceasta conduce la o serie de aplicaţii interesante, în special prin inducerea proprietăţilor structurilor algebrice în domeniul algebrelor Boole.

3.7 Exerciţii

1. Să se arate că într-o algebră booleană au loc relaţiile:
$$a \vee (\overline{a} \wedge b) = a \vee b, \quad a \wedge (\overline{a} \vee b) = a \wedge b.$$

2. Într-o algebră Boole se defineşte o relaţie de ordine prin
$$a \leq b \iff b = a \vee b$$
Să se arate că $a \leq b \iff \overline{b} \leq \overline{a}$.

3. $a \leq b \iff a \wedge \overline{b} = 0 \iff \overline{a} \vee b = 1$.

4. $a \wedge b = 0 \iff b \leq \overline{a}$.

5. Demonstraţi afirmaţiile din Propoziţiile 3.1, 3.2 şi 3.3.

6. Să se arate că pe $\mathcal{F}_n$ se poate defini o distanţă prin relaţia
$$\delta(f, g) = w(f + g) - w(fg).$$
Să se construiască un tabel cu distanţele elementelor lui $\mathcal{F}_2$.

7. Demonstraţi Teorema 3.8, Propoziţia 3.5 şi Corolarul 3.2.

8. Fie A o mulţime finită nevidă. Structuraţi 2^A sub formă de inel boolean.

9. Dacă $f \in \mathcal{F}_n$, atunci
$f(x_1, \ldots, x_n) = f(x_1, \ldots, x_{n-1}, 0) + [f(x_1, \ldots, x_{n-1}, 0) + f(x_1, \ldots, x_{n-1}, 1)]x_n$.

10. Construiţi un algoritm de generare a formei normal conjunctive asociată unei funcţii booleene.

11. Să se arate echivalenţa expresiilor booleene
$[b \wedge (\overline{a \wedge c}) \wedge (a \vee \overline{c})] \vee (\overline{b \vee c})$, $\qquad (\overline{a} \vee \overline{b} \vee \overline{c}) \wedge (\overline{a} \vee b \vee \overline{c}) \wedge (a \vee \overline{b} \vee \overline{c}) \wedge (a \vee b \vee \overline{c})$.

12. Să se arate că expresiile booleene

$$a \vee (\overline{a} \wedge b) \vee (b \wedge c), \quad b \vee (a \wedge \overline{b})$$

sunt echivalente. Să se scrie forma normal disjunctivă şi normal conjunctivă a funcţiei asociate.

Capitolul 4

Sisteme digitale

4.1 Circuite combinaţionale

Odată cu acest capitol vom începe un studiu sistematic al elementelor fundamentale de construcţie a circuitelor.

Definiţia 4.1 *Un "circuit" este un graf orientat cu cel puţin o intrare şi cel puţin o ieşire, care are două tipuri de noduri: "conectori" şi "porţi".*

Intrările unui circuit primesc "semnale", sub forma unor seturi de valori din mulţimea $\{0,1\}^n$ (n fiind numărul de intrări ale circuitului).

Definiţia 4.2 *Un circuit combinaţional (CLC[1]) este o reţea arborescentă în care ieşirea la momentul (tactul) t depinde numai de intrările la momentul (tactul) t.*

Un circuit este *activ* dacă ieşirea sa este $'1'$; în caz contrar, circuitul este *inactiv*.

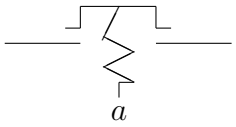
4.1.1 Porţi

Cele mai simple circuite sunt circuitele constante $'0'$ şi $'1'$; de obicei primul este marcat printr-un punct ., iar al doilea printr-o linie ———.

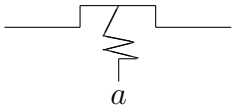
O variabilă booleană (logică) a poate controla un circuit, activându-l (dacă ia valoarea $'1'$) sau dezactivându-l (la valoarea $'0'$).

Indiferent dacă modul de activare/dezactivare este de tip mecanic, electric, electromagnetic, integrat, imprimat, neural, ideea de bază este aceeaşi: pentru înţelegere, vom considera circuitul sub forma unei linii legată la o sursă permanentă de curent, linie întreruptă de o "poartă" al cărei rol este de a închide sau deschide circuitul:

[1]Denumirea completă este *Circuit Logic Combinaţional*.



$a = 0$ (dezactivat)



$a = 1$ (activat)

Poarta poate fi deschisă (dezactivată) prin intermediul unui arc, care în poziţie de repaos ($a = 0$) este destins.

Dacă $a = 1$, acest arc este comprimat şi închide poarta, activând astfel circuitul, prin care trece (de fapt) valoarea lui a.

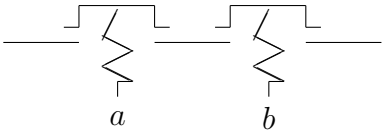
Vom nota un astfel de circuit prin a ————.

Să definim modul de combinare al circuitelor elementare, folosind operaţiile booleene (disjuncţie, conjuncţie, negaţie).

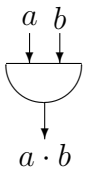
Implementarea acestor operaţii se realizează prin circuite numite *porţi*.

- Poarta AND:

  Fie a, b două variabile booleene. Pentru ab (sau – în funcţie de notaţie – $a \wedge b$) se poate construi circuitul



(1)



(2)

$\cdot$	0	1
0	0	0
1	0	1

(3)

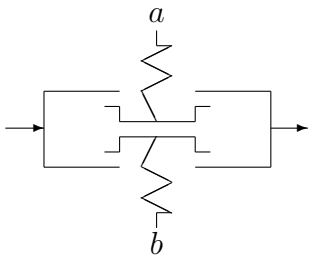
  (1) este structura funcţională a circuitului, (2) este notaţia logică, iar (3) - tabela de operaţie a operatorului AND – operatorul $\wedge$ dintr-o algebră Boole.

  Se observă că valoarea de ieşire este 1 dacă şi numai dacă $a = b = 1$.

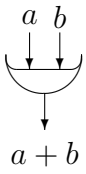
  Simbolul porţii AND (Figura (2)) poate fi generalizat la n intrări $a_1, \ldots, a_n$, obţinându-se o poartă AND_n.

- Poarta OR:

  Pentru poarta care realizează circuitul $a + b$ ($a \vee b$ în notaţie booleană), structura funcţională este reprezentată în Figura (1), simbolul logic – de Figura (2), iar tabela de operaţii - de Figura (3):



(1)



(2)

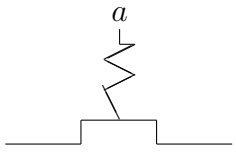
+	0	1
0	0	1
1	1	1

(3)

Evident, şi această poartă poate fi extinsă la n intrări: OR_n.

- Poarta NOT:

  O poartă pentru realizarea operaţiei de complementare (negaţie) a variabilei booleene a este Figura (1):



(1)

(2)

(3)

  Se observă că circuitul este închis prin poziţia relaxată a arcului ($a = 0$); la comprimarea acestuia ($a = 1$) circuitul se dezactivează (deschide).

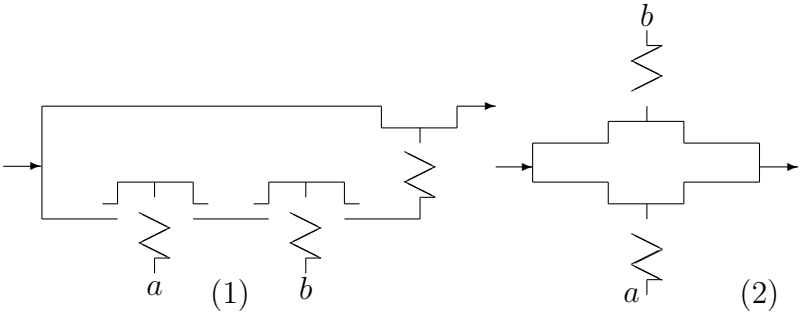
  Simbolul logic este (2), iar uneori chiar (3).

Înafara acestor operaţii, în construcţia circuitelor se mai folosesc trei tipuri suplimentare de porţi: $NAND$, NOR şi XOR.

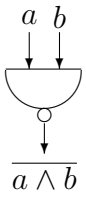
- Poarta $NAND$:

  Este un circuit care implementează expresia booleană $\overline{a \wedge b}$. Figura (2) reprezintă o altă implementare, obţinută în urma relaţiei De Morgan: $\overline{a \wedge b} = \overline{a} \vee \overline{b}$.

  Simbolul logic al porţii $NAND$ este Figura (3).



(1) (2)



(3)

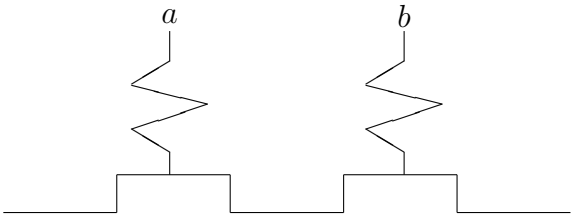
Tabela de operaţii a operatorului $NAND$ (similar operatorului Sheffer din algebra Boole) este

$\|\|$	0	1
0	1	1
1	1	0

- Poarta NOR:

  Este un operator mai puţin folosit în teoria algebrelor Boole (nu corespunde unui operator logic consacrat).

  Reprezintă operaţia $\overline{a \vee b} = \overline{a} \wedge \overline{b}$.



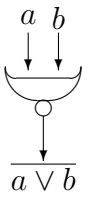


  Tabela de operaţii este:

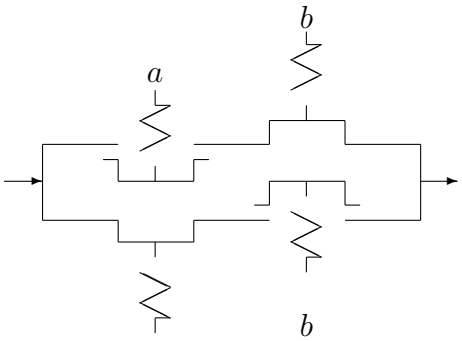
NOR	0	1
0	1	0
1	0	0

  De fapt aceşti operatori ($NAND$ şi NOR) se obţin prin legarea în serie a două porţi: o poartă AND (respectiv OR) şi o poartă NOT.

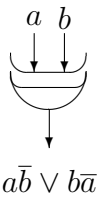
  Teoretic ei se pot generaliza la $NAND_n$ (respectiv NOR_n).

- Poarta XOR:

  Este implementarea operaţiei "sau logic" $a\overline{b} \vee b\overline{a}$ (operatorul $\oplus$ dintr-o algebră Boole):



(1)



(2)

$\oplus$	0	1
0	0	1
1	1	0

(3)

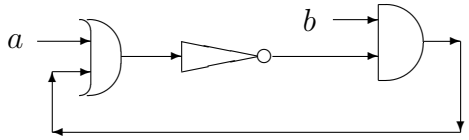
4.1.2 Circuite

În linii mari, arhitectura unui calculator se bazează pe construirea de circuite care formează diverse componente ale unui computer (în special unitatea logico-aritmetică, unităţile de memorie, magistralele de sincronizare, structuri de control).

Cele mai simple circuite sunt cele combinaţionale.

Conform Definiţiei 4.2, un circuit combinaţional este o structură arborescentă construită folosind un număr finit de porţi.

De exemplu, circuitul următor nu este combinaţional:

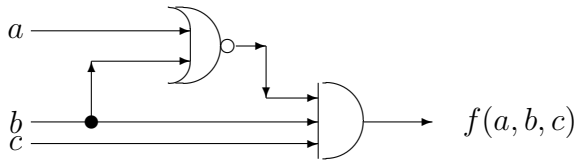


În general vom considera circuitele combinaţionale cu un număr de intrări (noduri de intrare) egal cu numărul de variabile şi având o singură ieşire (nod final).

Definiţia 4.3 *Fie X un circuit combinaţional de intrări $a_1, \ldots, a_n$. Se numeşte funcţie de transmisie a lui X funcţia booleană $f_X(a_1, \ldots, a_n)$ cu proprietatea că $f_X = 1$ dacă şi numai dacă există un drum activ de la nodurile de intrare la nodul final.*

Teorema 4.1 *Două circuite combinaţionale X, Y sunt echivalente dacă $f_X = f_Y$.*

Exemplul 4.1 *Să considerăm funcţia booleană $f(a, b, c) = bc(\overline{a \vee b})$. Un circuit combinaţional care o implementează este*



Cum această expresie este însă egală cu $bc\overline{a}\overline{b} = 0$, pentru ea se poate construi un circuit echivalent redus la un simplu punct.

Evident, acesta este mult mai simplu (nu are nici o poartă, în comparaţie cu circuitul iniţial – cu două porţi).

Definiţia 4.4 *Complexitatea unui circuit combinaţional este dat de numărul de porţi.*

Nu vom face o prezentare a noţiunilor de complexitate (spaţiu sau timp), acestea fiind identice cu cele folosite în algoritmică.

4.2 Extensii

Putem construi întreaga arhitectură a unui calculator plecând de la un element teoretic de bază numit *sistem digital*.

Definiţia 4.5 *Fie V un alfabet finit nevid şi m, n două numere naturale. Se numeşte "sistem digital" structura $S = (X, Y, f)$ unde $X = V^n,\ Y = V^m,\ f : X \longrightarrow Y$.*

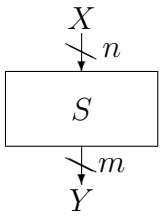
Un exemplu simplu de sistem digital este *circuitul combinaţional*, exprimabil printr-o funcţie booleană.

În această carte vom considera numai *sistemele digitale binare*, unde $V = \{0, 1\}$. De aceea nu va fi nici o ambiguitate dacă vom desemna *sistemele digitale binare* cu termenul general de *sisteme digitale*.

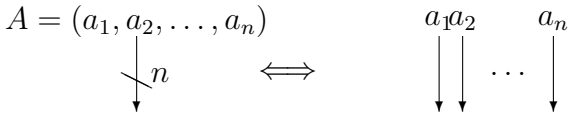
Funcţia f se numeşte *funcţie de transfer*.

De asemenea, cazurile $n = 0$ sau $m = 0$ corespund unor sisteme digitale speciale (de *ieşire* respectiv *intrare*) care vor fi studiate separat. În continuare considerăm numai $m * n \neq 0$.

Sistemele digitale date prin Definiţia 4.5 se pot reprezenta grafic sub forma următoare:



unde – prin convenţie – un flux de n date se reprezentă printr-o singură săgeată marcată cu valoarea lui n:



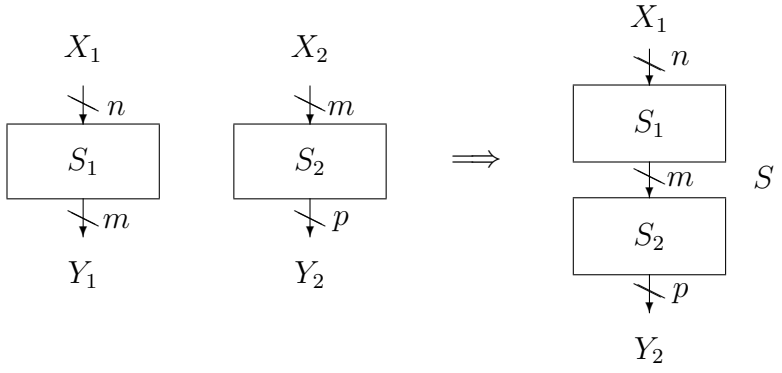
Operaţiile de bază care se pot defini pentru sistemele digitale sunt cele de legare în serie şi în paralel, operaţii numite *extensii*.

Definiţia 4.6 *Fie sistemele digitale* $S_i = (X_i, Y_i, f_i),\ i = 1, 2$ *unde*

$$X_1 = \{0,1\}^n,\ Y_1 = X_2 = \{0,1\}^m,\ Y_2 = \{0,1\}^p$$

Se numeşte "extensie serială" sistemul digital $S = (X, Y, f)$ *unde* $X = X_1,\ Y = Y_2$, *iar funcţia de transfer* $f : X \longrightarrow Y$ *este definită prin* $f = f_2 \circ f_1$.

Grafic, are loc următoarea operaţie:

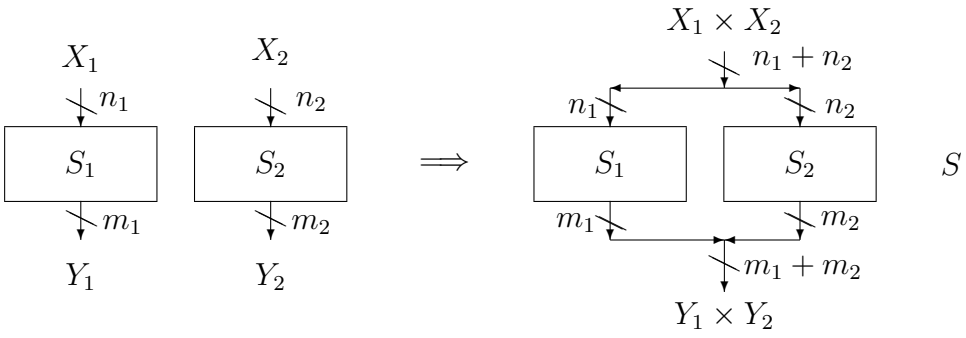


Definiţia 4.7 *Fie* $S_i = (X_i, Y_i, f_i),\ (i = 1, 2)$ *două sisteme digitale. Extensia paralelă* $S_1 \times S_2$ *este sistemul*

$$S = (X_1 \times X_2, Y_1 \times Y_2, f_{12})$$

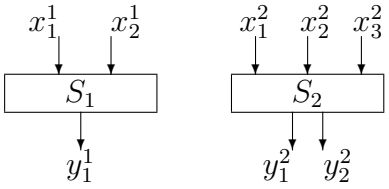
unde $f_{12} : X_1 \times X_2 \longrightarrow Y_1 \times Y_2$ *este definită* $f_{12}(x, y) = (f_1(x), f_2(y))$.

Grafic, are loc operaţia de combinare:

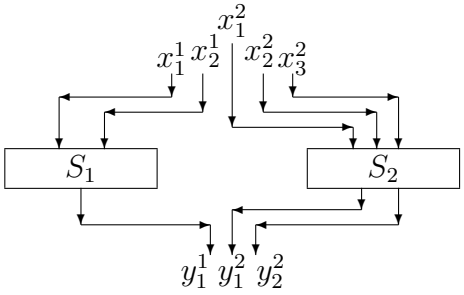


De remarcat că într-o extensie paralelă cele două sisteme digitale nu interacţionează.

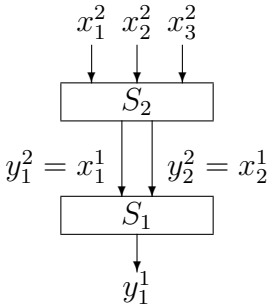
Exemplul 4.2 *Să considerăm sistemele digitale* $S_1 = (\{x_1^1, x_2^1\}, \{y_1^1\}, f_1)$ *şi* $S_2 = (\{x_1^2, x_2^2, x_3^2\}, \{y_1^2, y_2^2\}, f_2)$ *date de figura*



Extensia paralelă $S_1 \times S_2$ *este*



iar extensia serială a lui S_2 *cu* S_1*:*

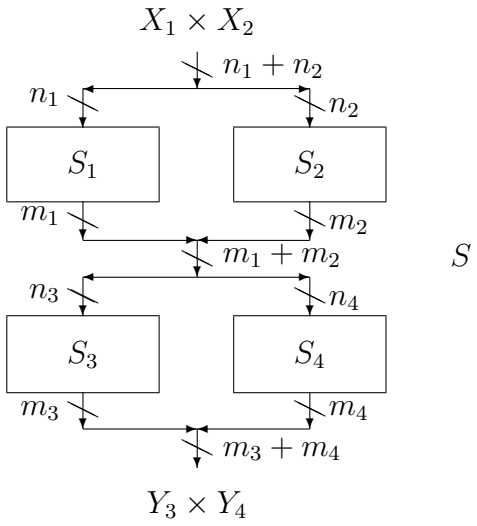


Definiţia 4.8 *Fie sistemele digitale* $S_i = (X_i, Y_i, f_i)$, $(1 \leq i \leq 4)$ *cu* $X_i = \{0,1\}^{n_i}$, $Y_i = \{0,1\}^{m_i}$ *şi* $m_1 + m_2 = n_3 + n_4$. *O extensie "serial-paralelă" este sistemul*

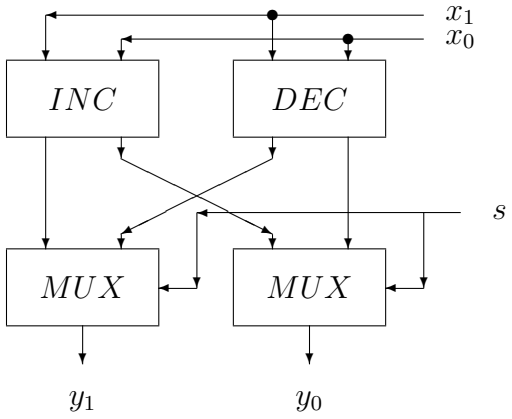
$$S = (X_1 \times X_2, Y_3 \times Y_4, f)$$

unde $f : X_1 \times X_2 \longrightarrow Y_3 \times Y_4$ *este definită* $f = f_{34} \circ f_{12}$ *(*f_{12} *şi* f_{34} *sunt funcţiile de transfer ale extensiilor paralele* $S_1 \times S_2$ *respectiv* $S_3 \times S_4$*).*

Grafic, o extensie serial - paralelă are forma



Exemplul 4.3 *Un circuit combinaţional care incrementează ($s = 0$) sau decrementează ($s = 1$) un număr x_1x_0 format din doi biţi, poate fi definit astfel:*



$INCR$ (incrementare), $DECR$ (decrementare), MUX (selecţie) sunt circuite combinaţionale (care vor fi definite mai târziu), iar circuitul final rezultat prin extensia serial - paralelă este – evident – tot combinaţional.

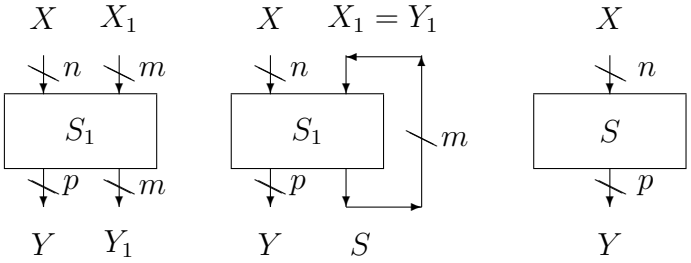
4.3 Cicluri

Pentru creşterea gradului de complexitate al circuitelor digitale se defineşte o nouă operaţie numită *ciclu*.

Definiţia 4.9 *Fie sistemul digital* $S_1 = (X \times X_1, Y \times Y_1, h)$ *unde* $X_1 = Y_1$. *Notăm* $h = (f, f_1)$ *unde* $f : X \times X_1 \longrightarrow Y,\ f_1 : X \times X_1 \longrightarrow Y_1$.

Prin identificarea (conectarea serială) a lui Y_1 *cu* X_1 *se obţine o ciclu şi noul sistem are forma* $S = (X, Y, g)$, *unde* $g : X \longrightarrow Y$ *este definită* $g(x) = f(x, f_1(x, y))$; f_1 *se numeşte "funcţie de tranziţie" şi verifică definiţia recursivă* $y = f_1(x, f_1(x, y))$.

Vom avea deci următoarea situaţie:



După cum se observă în Definiţia 4.5, apare o nouă variabilă "ascunsă" care ia valori în mulţimea $Q = X_1 = Y_1$; atunci $f : X \times Q \longrightarrow Y,\ f_1 : X \times Q \longrightarrow Q$.

Mulţimea Q caracterizează comportarea internă a sistemului digital; această comportare se mai numeşte *stare internă* sau pe scurt *stare*.

Efectul fundamental al comportării interne constă în evoluţia sistemului pe spaţiul de valori Q, fără modificări ale intrării X.

Pentru un $a \in X$ aplicat constant la intrare, ieşirea poate prezenta diverse variaţii. De aceea, spunem că **autonomia unui sistem creşte ca urmare a introducerii acestuia într-o ciclu**.

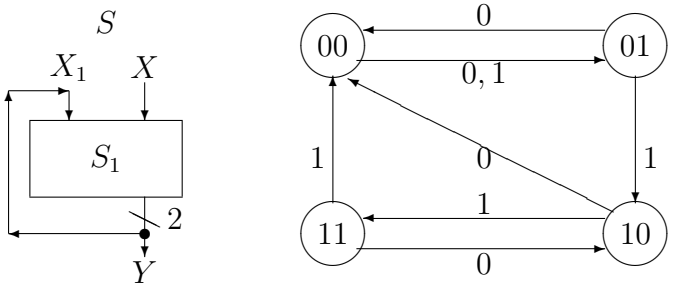
De obicei noile elemente legate de stare sunt introduse în definiţia sistemului; astfel, el se va scrie $S = (X, Y, Q, f_1, g)$.

Exemplul 4.4 *Să considerăm sistemul* $S_1 = (X \times X_1, Y, f)$ *unde* $X = \{0, 1\},\ X_1 = Y = \{0, 1\}^2$, *iar* f *este definit de tabelul*

$X \times X_1$	Y	$X \times X_1$	Y
0 00	01	1 00	01
0 01	00	1 01	10
0 10	00	1 10	11
0 11	10	1 11	00

Prin identificarea (legarea serială a) lui Y cu X_1 se obţine un nou sistem S.

Tranziţiile sale sunt ilustrate mai jos (se face abstracţie de timpul real cât este aplicat un semnal la intrare).



Ieşirea sistemului S_1 este aceeaşi cu cea a sistemului S.

După închiderea ciclului ($X_1 = Y$), sistemul S_1 capătă – în funcţie de tranziţie – un comportament al ieşirii autonom de intrare.

De exemplu, intrarea (constantă) $11111\ldots$ în sistemul S va determine ieşirea ciclică $01\ 10\ 11\ 00\ 01\ldots$

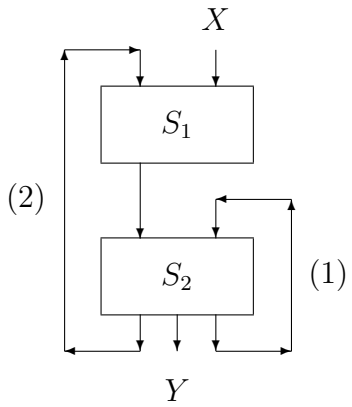
Definiţia 4.10 *Comportarea unui sistem digital S este autonomă dacă şi numai dacă pentru o intrare constantă, ieşirea are un comportament dinamic.*

Să introducem acum noţiunea de *subciclu*.

Definiţia 4.11 *Un ciclu A este inclus în alt ciclu B dacă A aparţine unui sistem care face parte dintr-o extensie serială închisă prin ciclul B.*

Spunem că A este subciclu al lui B.

De exemplu, în figura următoare, (1) este subciclu al lui (2).



Se poate da acum o definiţie a sistemelor digitale de ordin n.

Definiţia 4.12 *Un sistem de ordin n ($n \geq 0$) $n - DS$ se defineşte recursiv astfel:*

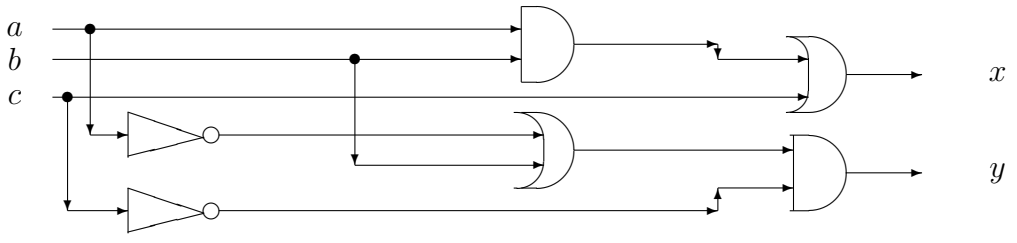
1. *Orice circuit combinaţional (fără cicluri) este un $0 - DS$.*
2. *Un $(n+1) - DS$ sistem se obţine dintr-un $n - DS$ adăugând un ciclu care include toate cele n cicluri anterioare.*
3. *Orice $n - DS$ se obţine prin aplicarea regulilor anterioare.*

De-a lungul acestei cărţi se va arăta următoarea corespondenţă ierarhică pentru $n - DS$, care corespunde arhitecturii unui calculator:

- $0 - DS$: sunt circuite combinaţionale (fără autonomie);
- $1 - DS$: circuite de memorie (cu autonomie pe spaţiul stărilor);
- $2 - DS$: automate finite (cu autonomie pe tipul de comportare);
- $3 - DS$: procesoare (cu autonomie pe interpretarea stărilor interne);
- $4 - DS$: calculatoare

4.4 Exerciţii

1. Construiţi circuite care realizează porţile AND, OR, NOT, NOR şi XOR folosind numai porţi $NAND$.
2. Construiţi circuite care realizează porţile $AND, OR, NOT, NAND$ şi XOR folosind numai porţi NOR.
3. Se dă circuitul:



Construiţi tabela sa de adevăr.

4. Se dă funcţia booleană definită prin tabelul

a	0	0	0	0	1	1	1	1
b	0	0	1	1	0	0	1	1
c	0	1	0	1	0	1	0	1
$f(a,b,c)$	0	1	1	1	0	0	1	1

   Să se construiască un circuit care implementează această funcţie.

5. Construiţi un circuit care implementează funcţia booleană

$$f(a,b,c) = \overline{a}bc + \overline{a}\overline{b}\overline{c} + ab\overline{c}$$

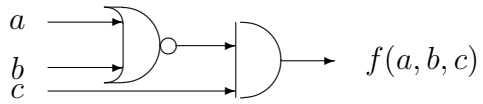
6. Aceeaşi problemă pentru funcţia booleană

$$g(a,b,c,d,e) = a(bc \oplus \overline{b}\overline{c}) + b(cd \oplus e)$$

7. Sunt echivalente următoarele două funcţii booleene ?

$$f(a,b,c) = abc + \overline{a}b\overline{c} \qquad g(a,b,c) = (\overline{a \oplus c})b$$

8. Scrieţi o funcţie booleană descrisă de circuitul următor:



   Aranjaţi răspunsul în forma normal disjunctivă, apoi forma normal conjunctivă.

9. Să se construiască un circuit cu două porţi pentru implementarea funcţiei booleene

$$f(a,b) = ab + \overline{a}\overline{b}$$

Capitolul 5

Sisteme $0 - DS$

Acestea sunt cele mai simple tipuri de sisteme digitale, caracterizate complet prin funcţii booleene.

Astfel, multe funcţii digitale pot fi generate plecând de la circuitele elementare care le realizează, la care se adaugă o regulă recursivă (pentru executarea funcţiei independent de mărimea intrării).

Vom trece în revistă cele mai importante sisteme care pot fi descrise în acest mod.

5.1 Decodificatoare

Un sistem digital are drept mulţime de valori de intrare n-tupluri binare din $\{0,1\}^n$. Semnalul care vine printr-un canal fizic trebuie transferat întâi într-o astfel de valoare, acceptată de intrarea sistemului.

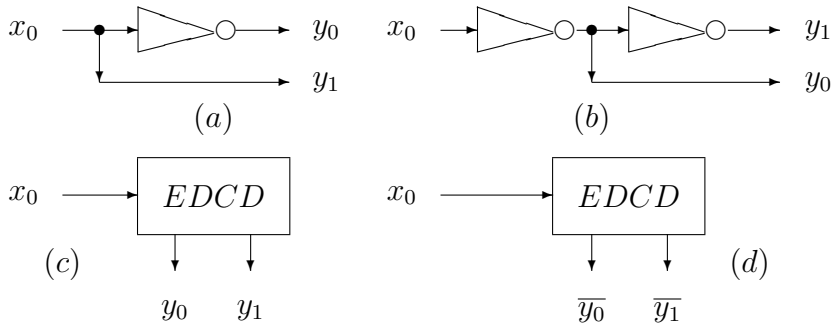
De aceea, prima problemă care trebuie rezolvată este standardizarea şi determinarea exactă a semnalului recepţionat.

Aceasta este una din principalele funcţii ale unui sistem digital. Înainte de a genera un răspuns semnalului primit, trebuie să ştim **ce** semnal s-a primit.

Circuitul care îndeplineşte acest deziderat se numeşte *decodificator*.

Cel mai simplu decodificator este *decodificatorul elementar* ($EDCD$), care ca scop determinarea valorii unui bit.

Schema sa de construcţie este dată de Figura (a) sau (b), iar reprezentarea simbolică – Figura (c) respectiv (d):



Decodificatorul elementar a fost obţinut prin extensia paralelă a circuitelor care realizează cele mai simple funcţii $f(x_0) = y_1 = x_0$ (funcţia identică) şi $g(x_0) = y_0 = \overline{x}_0$ (funcţia NOT).

Se observă că permanent una şi numai una din liniile de ieşire are valoarea $'1'$ (este activă): dacă este activă ieşirea y_0, atunci intrarea a avut valoarea $'0'$, iar dacă este activă ieşirea y_1, intrarea a avut valoarea $'1'$.

Determinarea valorii exacte a bitului de intrare x_0 se face deci aflând ce ieşire a decodificatorului este activă.

Pentru a izola ieşirea din canal (sau din alt sistem digital) de intrarea în decodificator (lucru recomandat în practică), între cele două circuite se mai inserează o poartă logică; cum singurul operator unar utilizat este NOT, el se va adăuga printr-o extensie serială, obţinându-se un $EDCD$ *cu buffer* (figura (b)).

În acest fel, comportamentul unui $EDCD$ este protejat de eventuale probleme ale circuitului care asigură intrarea.

Generalizând, obiectivul acestei secţiuni este aceea de a construi un *decodificator* (DCD) cu n intrări (DCD_n) pentru decodificarea semnalelor formate din n biţi.

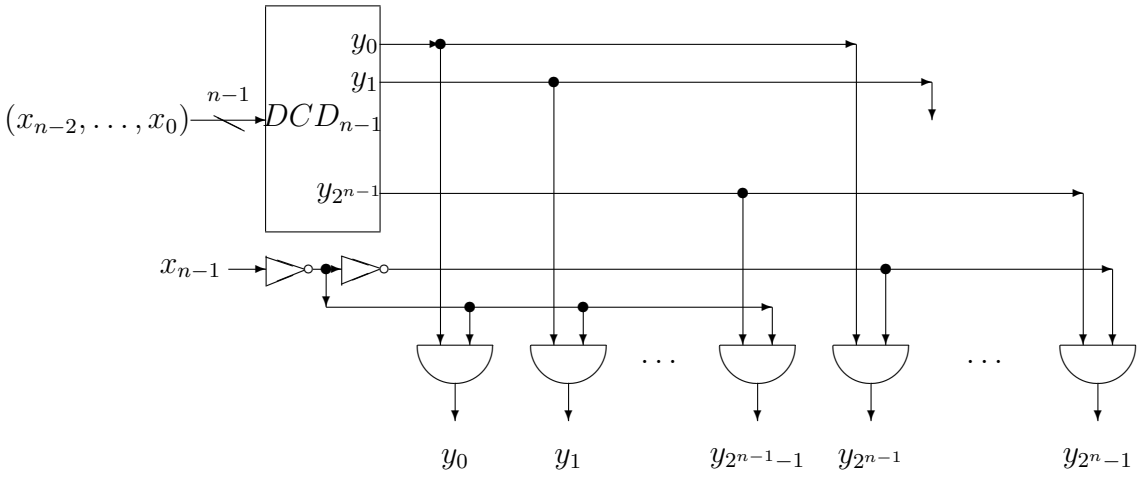
Ulterior, **orice sistem digital va avea intrarea conectată la ieşirea unui decodificator.**

Definiţia 5.1 *Un DCD_n este un circuit combinaţional cu intrarea $X = \{0,1\}^n$ formată din n biţi $(x_{n-1}, \ldots, x_1, x_0)$ şi ieşirea Y reprezentată prin $m = 2^n$ biţi $y_{m-1}, \ldots, y_0$.*

Fiecare ieşire este activă numai pentru o anumită configuraţie binară de intrare, şi – reciproc – fiecare configuraţie de intrare activează un singur bit de ieşire.

Structura unui DCD_n se defineşte recursiv. Astfel:

1. Un DCD_1 este un $EDCD$ reprezentat în figura (b);
2. Un DCD_n $(n \geq 2)$ se obţine combinând un DCD_{n-1} cu un DCD_1 după regula din figura următoare:



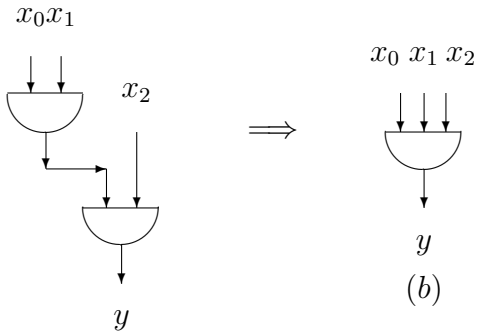
DCD_n

Singura dificultate în această construcţie constă în adâncimea (numărul mare de nivele ale) structurii.

În plus, sistemul conţine pe aceste nivele un total de $2^n + 2^{n-1} + \ldots + 2 = 2^{n+1} - 2$ porţi AND (înafara celor $2n$ porţi NOT).

Deci un DCD_n construit pe baza definiţiei recursive de mai sus, are o complexitate exponenţială.

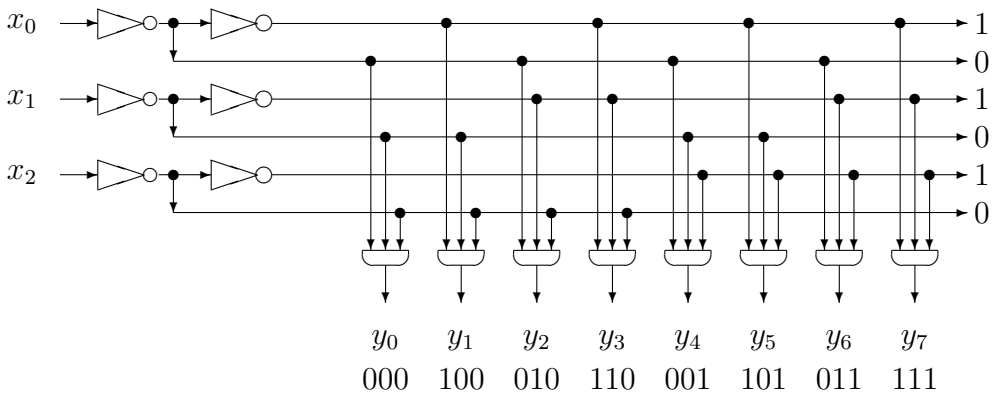
Ea poate fi micşorată substanţial, reducând adâncimea decodificatorului, de la ordinul n până la ordinul 1, pe baza următoarei observaţii: intrarea în fiecare AND este de forma (a); folosind asociativitatea, cele două componente AND pot fi înlocuite cu una singură, având trei intrări, cum este în (b):



Noul DCD va avea numai $n - 1$ nivele.

Procedeul se repetă de încă $n - 1$ ori; în final se obţine un DCD_n cu n intrări, $2n$ invertoare, 2^n ieşiri, dar numai 2^n porţi AND, toate situate pe un singur nivel.

Exemplul 5.1 *Pentru $n = 3$ vom avea decodificatorul DCD_3 cu schema:*



Pentru un mesaj de forma 110*, intrarea va fi* $x_0 := 1$, $x_1 := 1$, $x_0 := 0$. *Urmărind circuitul decodificator DCD_3, se obţine $y_3 = 1$ şi $y_i = 0$ pentru $i \neq 3$.*

Ca o remarcă, ieşirea activă dintr-un DCD_n va fi y_i dacă şi numai dacă reprezentarea în binar a lui i este $x_{n-1}x_{n-2}\ldots x_0$.

5.2 Codificatori

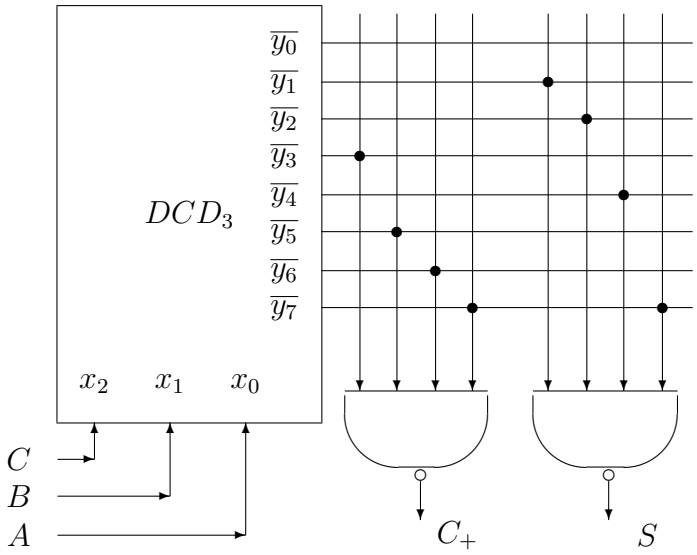
După cum s-a văzut, un decodificator sistematizează intrarea într-un sistem digital.

În particular, el pune o problemă sub o formă standard. Răspunsul la această problemă este generat de un *codificator*.

Definiţia 5.2 *Un $(2^n, p)$ - codificator este un circuit cu $m = 2^n$ intrări dintre care la fiecare moment numai una este activă; pentru fiecare intrare el generează o configuraţie binară de lungime p. Circuitul constă din p porţi OR sau din p porţi $NAND$, fiecare poartă având maxim m intrări.*

Un codificator nu este definit niciodată independent: intrările lui sunt de fapt n variabile care activează un DCD_n, iar cele $m = 2^n$ ieşiri ale acestuia reprezintă intrările în codificator. Vom completa ulterior Definiţia 5.2.

Exemplul 5.2 *Un sumator complet este un circuit combinaţional cu $n = 3$ intrări:*



A, B sunt numerele (de un bit) care se adună, iar C este bitul suplimentar de deplasare (de la însumarea altor doi biţi precedenţi). Sunt două ieşiri: S - valoarea sumei binare dintre A şi B, şi C_+ - valoarea noii deplasări.

Pentru construcţie vom folosi:

- Un decodificator DCD_3 ale cărui porţi de ieşire sunt complementate (în acest exemplu);

- Un codificator compus din două porţi $NAND$, fiecare cu câte 4 intrări.

Deci avem de-a face cu un $(8, 2)$ - codificator. Pentru a vedea cum lucrează, vom ţine cont de următoarea tabelă de decodificare:

		y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7
A	x_0	0	1	0	1	0	1	0	1
B	x_1	0	0	1	1	0	0	1	1
C	x_2	0	0	0	0	1	1	1	1

Să presupunem că avem $A = 1,\ B = 0,\ C = 1$*; aceasta corespunde situaţiei* $y_5 = 1,\ y_i = 0\ (i \neq 5)$*. Deci din* DCD_3 *va ieşi* $\overline{y}_5 = 0,\ \overline{y}_i = 1$*. Rezultatele celor două porţi* $NAND$ *sunt:*

$$S = \overline{\overline{y}_1 \wedge \overline{y}_2 \wedge \overline{y}_4 \wedge \overline{y}_7} = \overline{1 \wedge 1 \wedge 1 \wedge 1} = \overline{1} = 0$$

$$C_+ = \overline{\overline{y}_3 \wedge \overline{y}_5 \wedge \overline{y}_6 \wedge \overline{y}_7} = \overline{0} = 1$$

Într-adevăr, suma $1 + 0 + 1$ *este* 0*, iar transportul* 1*.*

Observaţia 5.1 *În exemplul 5.2, dacă ieşirile decodificatorului nu ar fi complementate, atunci codificatorul ar avea porţi* OR *în loc de* $NAND$*; aceasta rezultă din relaţiile De Morgan:*

$$p \vee q = \overline{\overline{p} \wedge \overline{q}}$$

În general, operatorul OR *corespunde noţiunii de codificare iar* AND *- celei de decodificare (*OR *distruge identitatea componentelor, pe când* AND *prefigurează o anumită configuraţie binară).*

Dacă dorim să reprezentăm toate funcţiile binare cu 3 intrări şi 2 ieşiri, vor fi necesare două porţi $NAND$ cu cel mult opt intrări fiecare.

Cele 2^{16} funcţii logice posibile vor fi implementate folosind circuitul din Exemplul 5.2 în care se pun punctele (conectorii) conform ieşirilor din tabela care defineşte funcţia logică respectivă.

Circuitul rezultat poate fi interpretat şi ca o *memorie fixă*.

Într-adevăr, dacă se consideră intrările drept *adrese*, atunci la fiecare adresă există stocată o anumită valoare binară, conform distribuţiei conectorilor.

Astfel, folosind Exemplul 5.2, la adresa 010 este stocat cuvântul 01, la adresa 110 cuvântul stocat este 10, ş.a.m.d.

Acest tip de memorie este numit **read only memory** (ROM).

Exemplul 5.3 *Să considerăm funcţia booleană cu* 3 *intrări şi* 3 *ieşiri definită prin*

$$f(x_0, x_1, x_2) = (x_0x_1 + \overline{x}_2,\ (x_0 + x_1)x_2,\ \overline{x}_0x_1 + \overline{x}_1x_2 + \overline{x}_2x_0).$$

Tabelul ei de valori va fi

	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7
x_0	0	1	0	1	0	1	0	1
x_1	0	0	1	1	0	0	1	1
x_2	0	0	0	0	1	1	1	1
$f_1(x_0, x_1, x_2)$	1	1	1	1	0	0	0	1
$f_2(x_0, x_1, x_2)$	0	0	0	0	0	1	1	1
$f_3(x_0, x_1, x_2)$	0	1	1	1	1	1	1	0

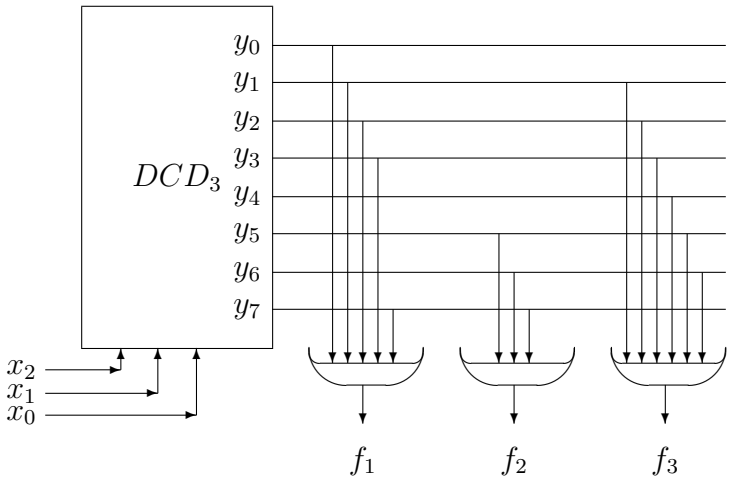
unde s-a notat

$f_1(x_0, x_1, x_2) = x_0x_1 + \overline{x}_2$,

$f_2(x_0, x_1, x_2) = (x_0 + x_1)x_2$,

$f_3(x_0, x_1, x_2) = \overline{x}_0x_1 + \overline{x}_1x_2 + \overline{x}_2x_0$.

Funcţia $f(x_0, x_1, x_2) = (f_1(x_0, x_1, x_2), f_2(x_0, x_1, x_2), f_3(x_0, x_1, x_2))$ *este implementată de circuitul codificator:*



Folosind deci astfel de circuite, orice funcţie booleană $f : V^n \longrightarrow V^p$ poate fi implementată de un $(2^n, p)$ - codificator.

Prin urmare, *codificatorul* este un circuit *universal*, capabil să implementeze orice funcţie logică.

Cum orice funcţie logică se poate defini pe baza unei tabele de adevăr, rezultă că un circuit *ROM* este universal.

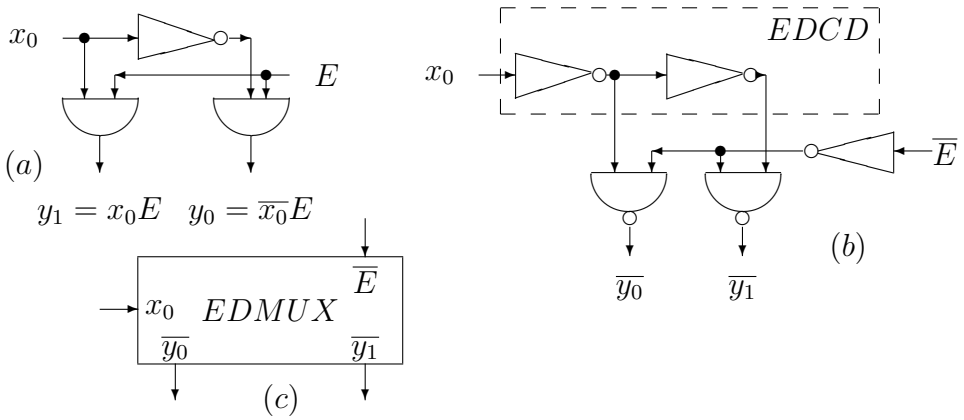
5.3 Demultiplexori

Demultiplexorul elementar ($EDMUX$) permite transferul unui semnal E spre două destinaţii distincte (y_0 sau y_1), conform unei anumite funcţii de selecţie x_0.

Circuitul asociat acestui deziderat este reprezentat în Figura (a).

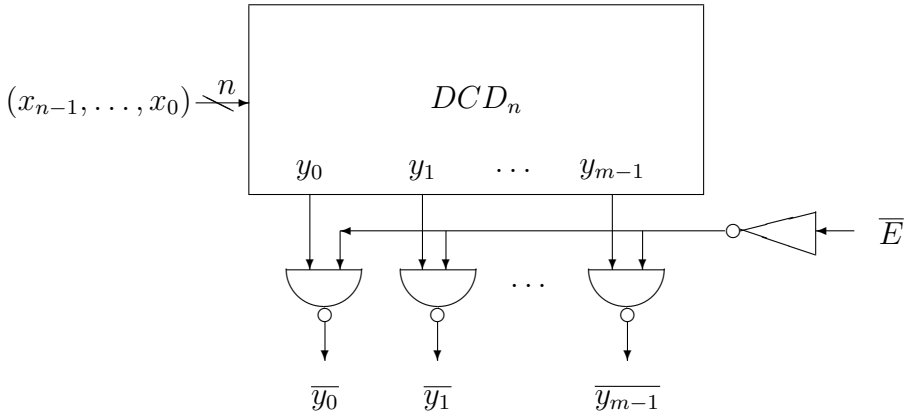
În practică, pentru securizarea intrărilor la circuitele driver-ului, schema sa se complică prin folosirea unui decodificator elementar ($EDCD$) pentru selectarea celor două porţi şi un buffer invertor pentru demultiplexarea intrării; de asemenea, porţile AND se înlocuiesc cu porţi $NAND$.

Se obţine astfel circuitul (b), a cărui reprezentare simbolică este (c):



Definiţia 5.3 *Un demultiplexor cu n intrări ($DMUX_n$) transferă semnalul de intrare E la una din cele $m = 2^n$ ieşiri $y_{m-1}, \ldots, y_0$, în conformitate cu un codul binar de selecţie $x_{n-1}, \ldots, x_0$.*

O primă soluţie pentru implementarea unui demultiplexor constă în generalizarea schemei (b): anume, utilizarea unui decodificator cu n intrări şi $m = 2^n$ porţi $NAND$, conectate ca în figură:



Observaţia 5.2 *A nu se face confuzie de notaţie: variabilele y_i din interiorul DCD_n sunt variabile interne ale acestui circuit. Ele diferă calitativ de variabilele $\overline{y_i}$ care formează ieşirea din $DMUX_n$.*

Deoarece pe nivelul de ieşire al unui decodificator se află porţi AND, un procedeu similar cu cel de la decodificatori permite reconstruirea unui $DMUX$ cu m porţi $NAND$ controlate de n $EDCD$-uri.

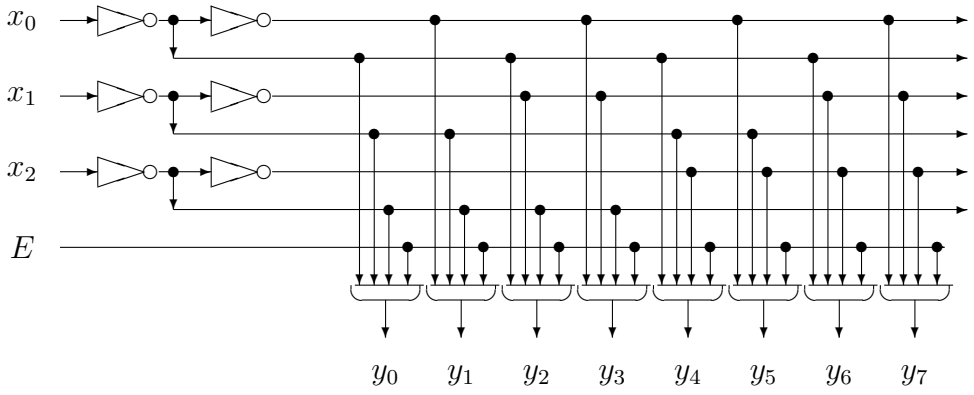
De asemenea, plecând de la definiţia recursivă a decodificatorilor, se poate construi o definiţie recursivă şi pentru demultiplexori.

Exemplul 5.4 *Să construim un circuit $DMUX_3$.*

Pentru simplificare, vom renunţa la complementările de securitate.

De asemenea, vom combina cele opt porţi AND_3 de la ieşirea din DCD_3 (varianta pe un singur nivel) cu porţile AND din $DMUX$, obţinând opt porţi AND_4, toate situate pe un singur nivel.

Cu aceste convenţii, structura circuitului este:



De exemplu, pentru a trimite semnalul E la adresa y_6 se foloseşte codul de selecţie $x_2 = 1,\ x_1 = 1,\ x_0 = 0$ (deoarece reprezentarea lui 6 în binar este 110).

În acest fel, ieşirea y_6 devine activă şi preia semnalul E, iar celelalte ieşiri sunt blocate de valoarea $'0'$ emisă de DCD_3.

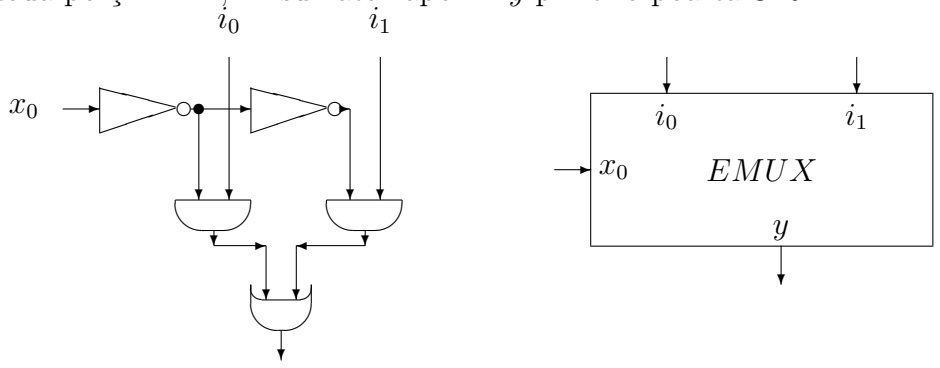
5.4 Multiplexori

Funcţia inversă unui $DMUX$ este *multiplexarea* (MUX): un circuit care adună într-un singur loc informaţia venită din mai multe zone.

Ea este de asemenea şi o funcţie de comunicare, asigurând interconectarea dintre blocuri distincte ale unui sistem digital.

Un multiplexor elementar ($EMUX$) este un *selector* care transmite semnalul i_1 sau i_0 spre y, în funcţie de valoarea codului de selecţie x_0.

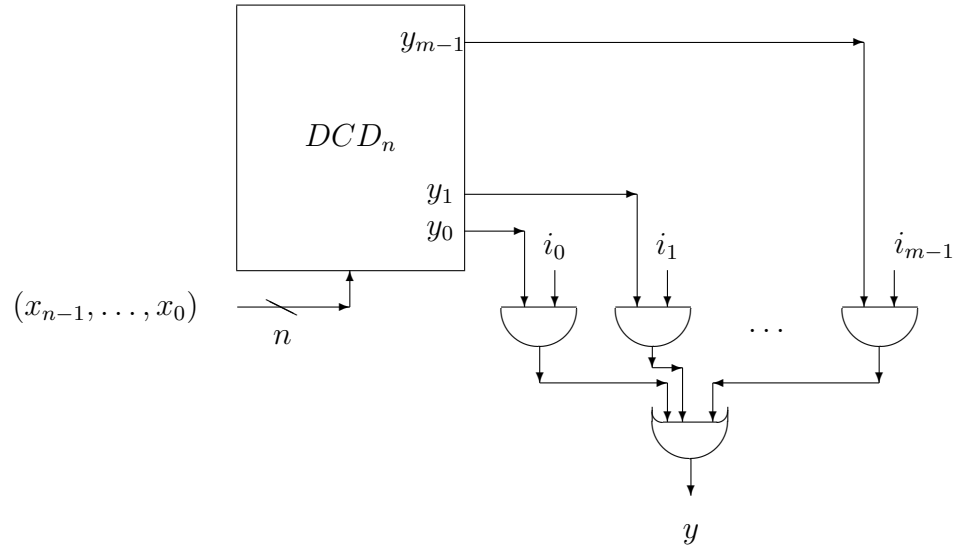
Circuitul este prezentat mai jos, unde un $EDCD$ cu intrarea x_0 deschide numai una din cele două porţi AND, "însumate" apoi în y printr-o poartă OR:



$$y = x_0 i_1 + \overline{x}_0 i_0$$

Definiţia generală a acestui circuit este:

Definiţia 5.4 *Un multiplexor (MUX_n) este un circuit combinaţional cu n biţi de control $(x_{n-1}, \ldots, x_0)$, care selectează la ieşirea y numai o intrare din cele $m = 2^n$ posibilităţi $i_{m-1}, \ldots, i_1, i_0$.*



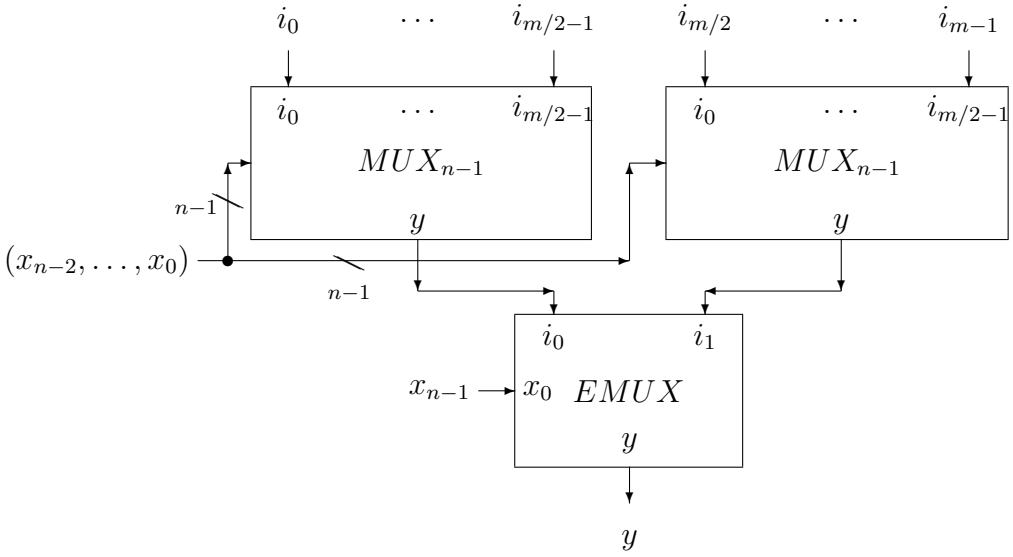
O implementare posibilă foloseşte un DCD_n legat serial cu o structură $AND - OR$, ca în figura anterioară.

Ieşirea din decodificator activează numai o poartă AND, care transferă la ieşire intrarea selectată.

Acestui circuit i se poate aplica proprietatea de asociativitate între porţile AND ale ieşirilor din decodificator şi porţile AND suplimentare (ca la DCD-uri).

Se poate da şi o definiţie recursivă a unui multiplexor, de complexitate mult redusă:

Definiţia 5.5 *MUX_n poate fi construit printr-o conectare serială a unui $EMUX$ cu două MUX_{n-1} conectate paralel; în plus, $MUX_1 = EMUX$.*



Deşi un multiplexor este (prin definiţie) un circuit de selecţie, el poate fi utilizat şi pentru realizarea unor funcţii booleene.

Astfel, să considerăm $x_{n-1} \ldots, x_0$ ca variabile booleene de intrare ale unei funcţii care ia valorile $i_{m-1}, \ldots, i_0$.

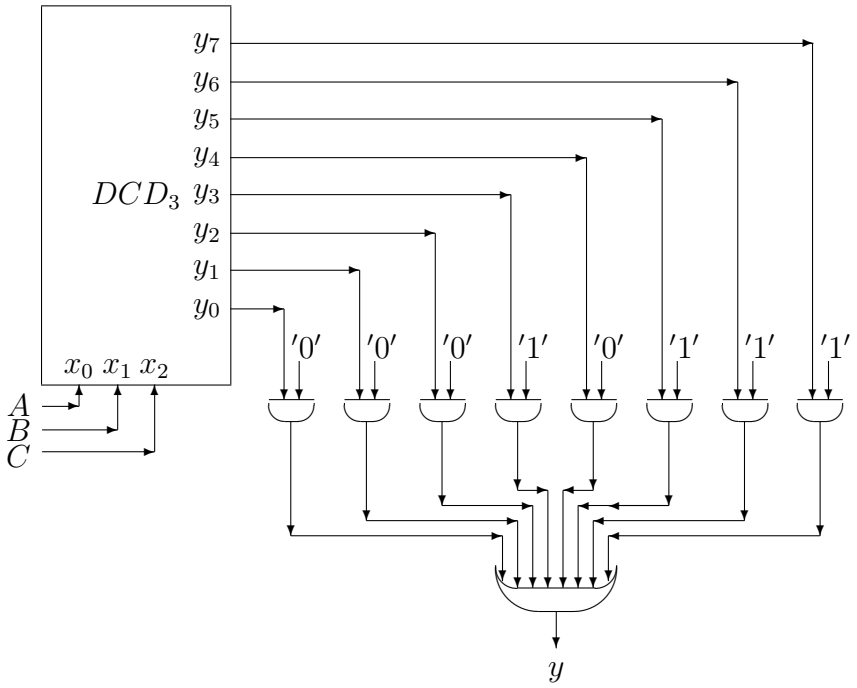
Pentru codul de selecţie (considerat intrare) $00 \ldots 0$ se alege valoarea lui i_0 ş.a.m.d., până la intrarea $11 \ldots 1$ care selectează valoarea i_{m-1}.

Deci MUX_n "execută" tabela de adevăr a unei funcţii booleene $f : V^n \longrightarrow V$.

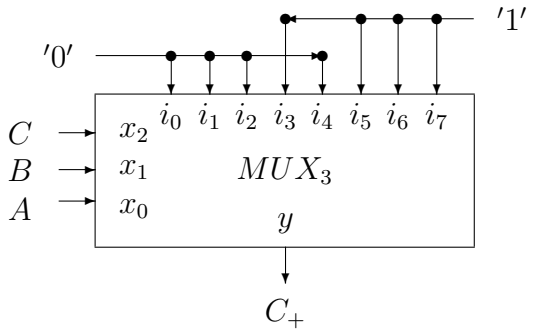
Exemplul 5.5 *Să considerăm funcţia de transport C_+ a sumatorului construit în Exemplul 5.2. Tabela sa de adevăr este dată de secvenţa*

$$y_3 + y_5 + y_6 + y_7 = 00010000 + 00000100 + 00000010 + 0000001 = 00010111$$

Dacă această secvenţă este aplicată ieşirilor din DCD_3 cu variabilele de intrare (selecţie) A, B, C, se obţine $y = C_+$:



sau, mai succint,



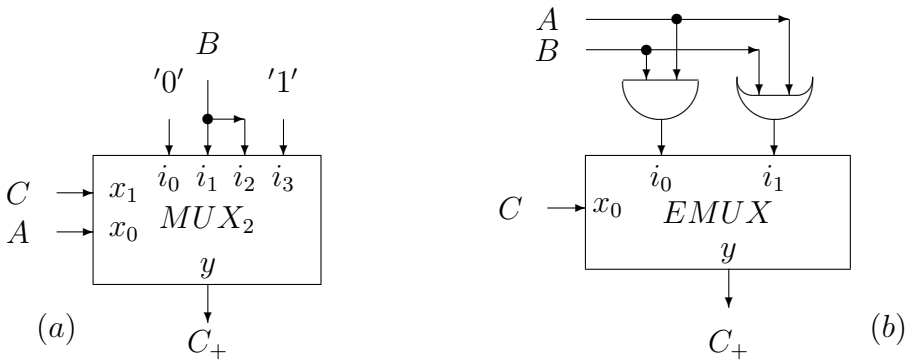
De exemplu, dacă avem intrarea 101*, aceasta corespunde lui* $y_5 = 1$ *şi* $y_i = 0$ *pentru* $i \neq 5$*. Cum* $i_5 = 1$*, pe a şasea poartă* AND *se obţine* $'1'$ *care - prin poarta* OR *- conduce la ieşirea* $y = 1$*.*

La fel, pentru intrarea 001 *avem* $y_4 = 1$ *şi toate ieşirile din porţile* AND *au valoarea* $'0'$*, ceea ce duce în final la* $y = 0$*.*

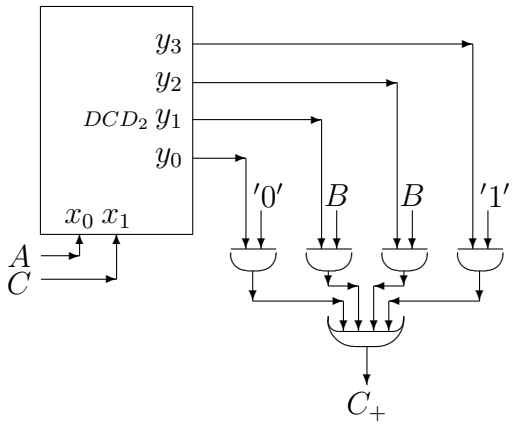
Soluţia prezentată este însă prea generală şi risipitoare de resurse.

Ea poate fi minimizată la un MUX_2 *(Figura* (a) *de pe pagina următoare) folosind una din variabile pe poziţie de selector.*

Ideea poate fi îmbunătăţită în continuare, ajungându-se în final la un $EMUX$ *care selectează cu* C *două funcţii de două variabile* A *şi* B *(Figura* (b)*):*

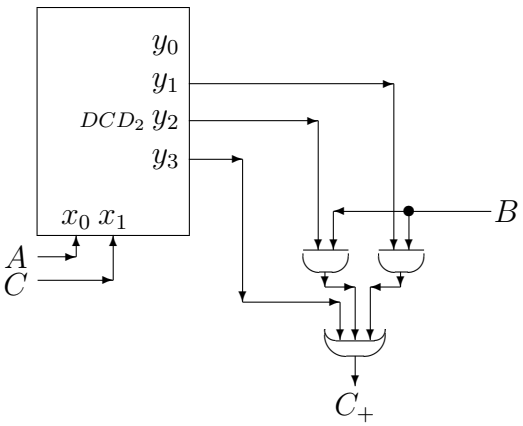


Pentru clarificare, vom detalia cazul (a); acesta este o notaţie simplificată a circuitului:



Construcţia tabelei de codificare conduce la un comportament identic cu circuitul dat iniţial, deşi acesta are numai 4 *porţi* AND *în loc de* 8*.*

Mai mult, deoarece prima şi ultima poartă sunt controlate de constante, acestea pot fi eliminate şi se ajunge la circuitul:



Exemplul 5.5 sugerează ideea că un multiplexor poate fi o soluţie pentru implementarea funcţiilor booleene.

Teorema 5.1 *Orice funcţie booleeană* $f : V^n \longrightarrow V \quad (n \geq 0)$ *poate fi implementată cu un circuit combinaţional logic (CLC).*

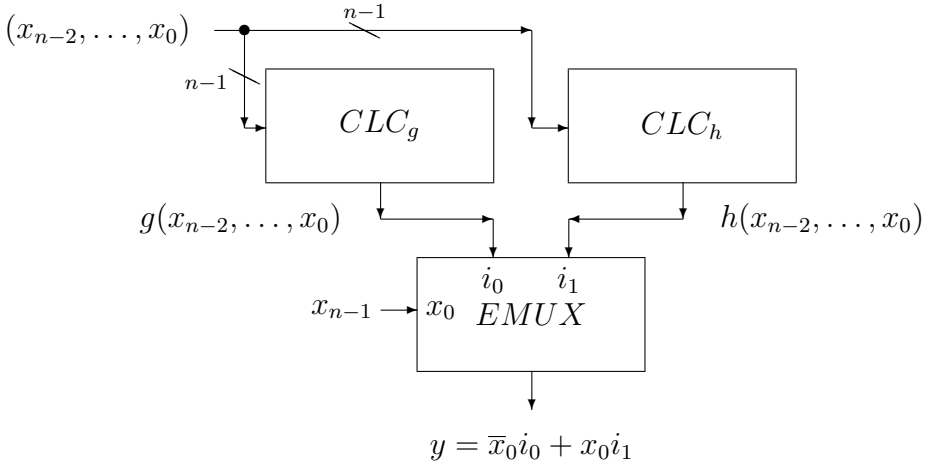
Demonstraţie: Pentru $n = 0$ afirmaţia este banală.

Pentru $n \geq 1$, orice funcţie booleană de n variabile se poate pune sub forma normală

$$f(x_{n-1}, \ldots, x_0) = \overline{x}_{n-1} g(x_{n-2}, \ldots, x_0) + x_{n-1} h(x_{n-2}, \ldots, x_0)$$

Pentru g şi h se poate aplica inductiv acelaşi procedeu.

Din aceste două circuite, legate serial cu un $EMUX$, se obţine circuitul solicitat.



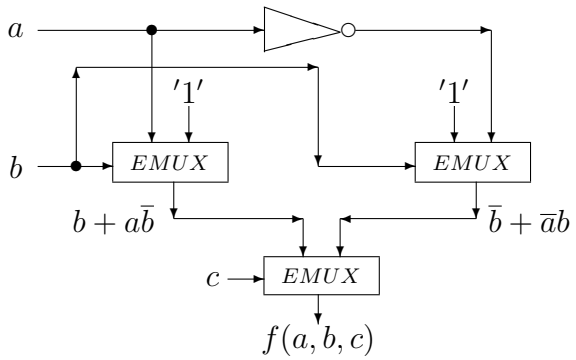
Exemplul 5.6 *Să considerăm funcţia* $f(a, b, c) = a\overline{b} + b\overline{c} + c\overline{a}$. *În prima etapă o reprezentăm sub formă normal disjunctivă:*

$f(a, b, c) = a\overline{b}(c + \overline{c}) + (a + \overline{a})b\overline{c} + \overline{a}(b + \overline{b})c = a\overline{b}c + a\overline{b}\overline{c} + ab\overline{c} + \overline{a}b\overline{c} + \overline{a}bc + \overline{a}\overline{b}c = (a\overline{b} + \overline{a}b + \overline{a}\overline{b})c + (a\overline{b} + ab + \overline{a}b)\overline{c}$.

Fie funcţiile

$$h(a, b) = a\overline{b} + \overline{a}b + \overline{a}\overline{b} = \overline{a}b + (a + \overline{a})\overline{b} = \overline{a}b + \overline{b} \text{ şi}$$
$$g(a, b) = a\overline{b} + ab + \overline{a}b = (a + \overline{a})b + a\overline{b} = b + a\overline{b}.$$

Pe baza Teoremei 5.1 se poate construi circuitul combinaţional logic al lui f*:*



5.5 Codificatori cu prioritate

Este posibil ca unele circuite de codificare să solicite

- Un număr de intrări care să nu fie multiplu de 2;
- Mai multe configuraţii active printre aceste intrări.

Din aceste motive apare necesitatea de a extinde ideea de codificator, astfel ca să se permită un număr arbitrar de intrări şi să se asigure un proces de selecţie care să aleagă bitul activ aflat pe cea mai semnificativă poziţie (bitul de prioritate maximă între biţii activi).

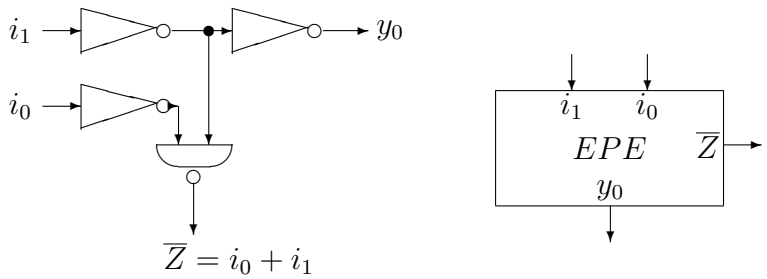
În aceste condiţii, *codificatorul cu prioritate* foloseşte o ordonare a biţilor de intrare (ca o condiţie suplimentară, necesară selecţiei).

Pe baza acesteia, dacă sunt activi mai mulţi biţi, numai cel mai semnificativ dintre ei va fi selectat la codificare.

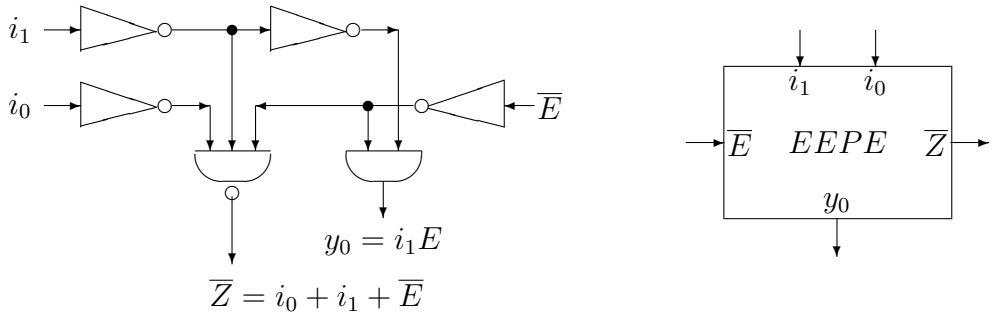
Definiţia 5.6 *Un codificator cu prioritate (PE) este un circuit cu m intrări* $i_{m-1}, \ldots, i_0$ *(în ordinea descrescătoare a priorităţilor) şi două zone de ieşire: o zonă Z de un bit care indică dacă intrarea conţine (*$Z = 0$*) sau nu (*$Z = 1$*) biţi activi, şi o zonă formată din* $n = \lceil log_2 m \rceil$ *biţi* $y_{n-1}, \ldots, y_0$ *care indică poziţia celui mai semnificativ bit activ din intrare (dacă există) (altfel generează eventual* $00 \ldots 0$*).*

Interpretarea numerică a funcţiei PE este: circuitul calculează partea întreagă a logaritmului (în baza 2) dintr-un număr întreg, indicând eventual o eroare (funcţia nu este definită) pe bitul Z.

Cel mai simplu astfel de circuit - *codificatorul cu prioritate elementar* (EPE) $(m = 2)$ este definit prin ecuaţiile booleene $Z = \bar{i}_1 \bar{i}_0,\ y_0 = i_1$ pentru circuitul de bază



sau $Z = \overline{i}_1\overline{i}_0E$, $y_0 = i_1E$ în cazul versiunii cu acces ($EEPE$); pentru facilitarea implementării, intrarea E şi ieşirea Z sunt complementate.



Rolul lui E este de a activa/dezactiva circuitul $EEPE$. Când $E = 1$, cele două structuri sunt echivalente.

Pentru $E = 0$, $EEPE$ scoate valoarea $\overline{Z} = 1$ indiferent de intrare (circuitul este decuplat).

Interpretarea comportamentului circuitului EPE (respectiv $EEPE$) este următoarea:

- $\overline{Z} = 0$: nu sunt biţi activi, iar y_0 este ignorat (sau se consideră $y_0 = 0$).
- $\overline{Z} = 1$: există biţi activi: cel mai semnificativ se află pe poziţia y_0.

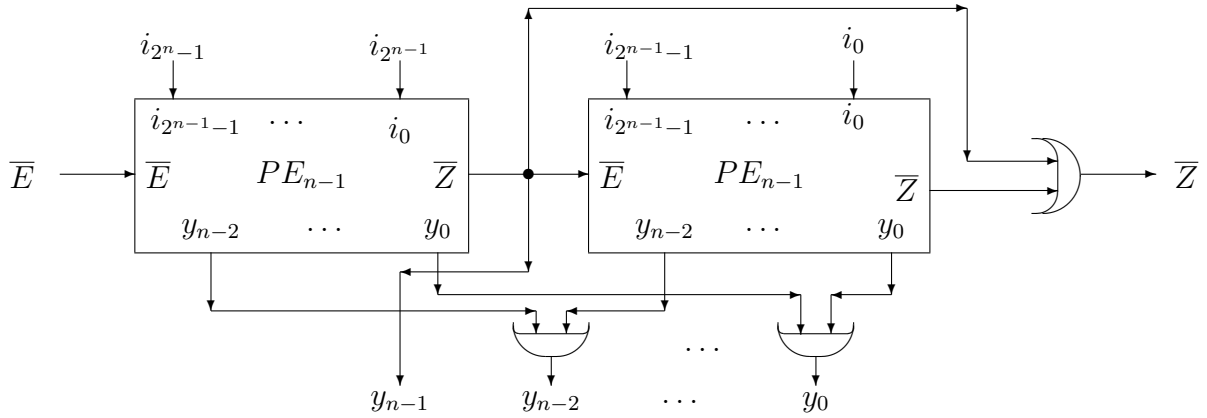
Comportamentul unui EPE (sau $EEPE$ în cazul $E = 1$) poate fi sumarizat în tabelul:

i_1	i_0	$\overline{Z}$	y_0	Comentarii
0	0	0	0	Nu sunt biţi activi
0	1	1	0	i_0 este bit activ
1	0	1	1	i_1 este bit activ
1	1	1	1	i_1 este cel mai semnificativ bit activ

Observaţia 5.3 *Dacă există cel puţin un bit activ, atunci* $y_0 = [log_2 n]$ *unde* $(n)_2 = i_1i_0$.

Extinzând sunt două variante recursive de construcţie a codificatorilor cu prioritate cu n ieşiri PE_n, bazate pe extensii seriale respectiv extensii paralele.

În prima variantă, se alege $PE_1 = EEPE$; în continuare, un PE_n ($n > 1$) cu acces este definit prin conectarea serială a două PE_{n-1} cu acces, ca în figură:



Observaţia 5.4 *Dacă $2^{n-1} < m < 2^n$, atunci intrărilor $i_m, i_{m+1}, \ldots, i_{2^n-1}$ li se asignează valoarea constantă $'0'$.*

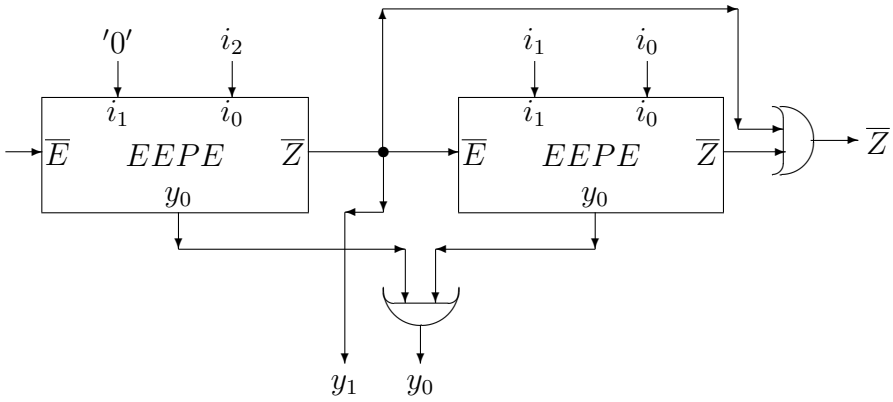
Cea mai semnificativă ieşire este $y_{n-1} = \overline{Z}$; celelalte ieşiri se obţin ca rezultante prin porţi OR ale ieşirilor corespunzătoare pentru cele două PE_{n-1}.

Într-adevăr, dacă toţi biţii din prima jumătate a intrării sunt inactivi, atunci cel mai semnificativ bit de ieşire este 0, ieşirile din PE_{n-1} corespunzătoare sunt 0, iar restul biţilor sunt daţi de al doilea PE_{n-1} (cu biţi mai puţin semnificativi).

Altfel, $y_{n-1} = \overline{Z} = 1$, ieşirea celui de-al doilea PE_{n-1} este dezafectată (deoarece $E = Z = 0$) şi tot restul ieşirii este dat de primul (cel mai semnificativ) PE_{n-1}.

Observaţia 5.5 *Accesul la un EP_n se realizează totdeauna cu $E = 1$ (deci $\overline{E} = 0$).*

Exemplul 5.7 *Să construim un codificator cu prioritate pentru $m = 3$. Atunci $n = \lceil log_2 3 \rceil = 2$ şi – conform schemei de mai sus – vom avea:*



Deoarece $m = 3$, se asignează la intrarea de prioritate maximă (corespunzătoare lui i_m) valoarea $'0'$.

Să presupunem că se solicită acest PE_2 (activat prin $E \leftarrow 1$) cu intrările i_2 şi i_0; deci $i_2 i_1 i_0 = 101$.

Atunci, din primul $EEPE = PE_1$ se obţine (atenţie: $i_1 \leftarrow '0'$, $i_0 \leftarrow i_2$)

$$Z = \overline{i}_1 \overline{i}_0 E = 1 \wedge 0 \wedge 1 = 0 \quad şi \quad y_0 = i_1 E = 0 \wedge 1 = 0$$

Cum $\overline{Z} = 1$, această valoare merge la ieşire în y_1 ($y_1 = 1$) şi constituie intrarea în al doilea $EEPE = PE_1$. Cum $\overline{E} = 1$ (deci $E = 0$), acesta este dezafectat.

Ieşirea va fi deci formată din $y_1 y_0 = 10$, care reprezintă – în binar – poziţia bitului 2: acesta (i_2) este bitul activ care este selectat ca prioritar.

Dacă intrarea în PE_2 este $i_2 i_1 i_0 = 001$, primul $EEPE$ scoate $y_0 = 0$ şi $Z = 1$.

Cum $\overline{Z} = 0$, aceasta va fi – simultan – valoarea finală a lui y_1 ($y_1 = 0$) cât şi filtrul care permite accesul la al doilea $EEPE$ (cu $E = Z = 1$).

Acesta are asignările $i_1 \leftarrow 0$, $i_0 \leftarrow 1$ şi va scoate $Z = \overline{i}_1 \overline{i}_0 E = 1 \wedge 0 \wedge 1 = 0$ – semnal că există biţi activi.

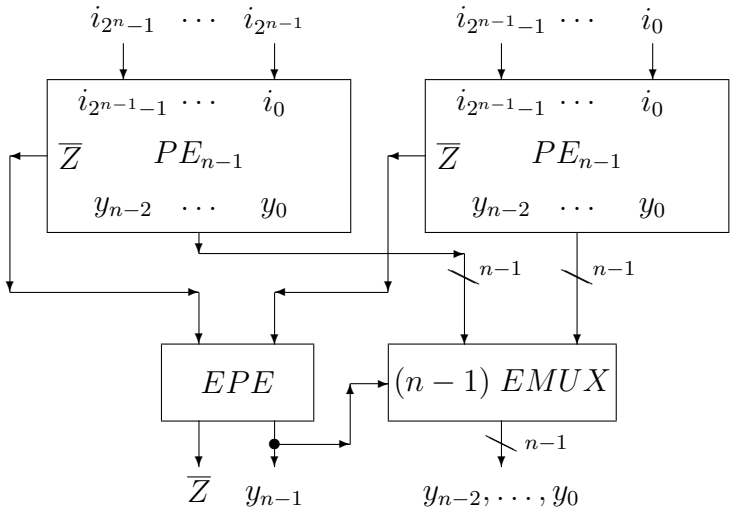
Se calculează $y_0 = i_1 E = 0$ care – prin poarta OR – merge în valoarea finală a lui y_0.

Deci $y_1 y_0 = 00$, adică primul bit i_0 este cel selectat ca activ.

Implementarea acestui circuit este însă prea lentă pentru aplicaţii (numărul de nivele creşte exponenţial); de aceea se foloseşte o altă construcţie, bazată pe extensia paralelă.

Aici PE_n este construit prin extensia paralelă a două PE_{n-1}; ieşirea $\overline{Z}$ şi cel mai semnificativ bit y_{n-1} sunt calculate cu un EPE din cele două $\overline{Z}$; restul biţilor de ieşire sunt selectaţi cu $n-1$ multiplexori elementari din ieşirile celor două PE_{n-1}.

EPE indică unde este cel mai semnficativ bit care are valoarea 1 (la intrarea primului sau celui de-al doilea PE_{n-1}), iar y_{n-1} selectează restul codului.



Adâncimea acestei variante este comparabilă cu n, deci mult mai simplă şi rapidă.

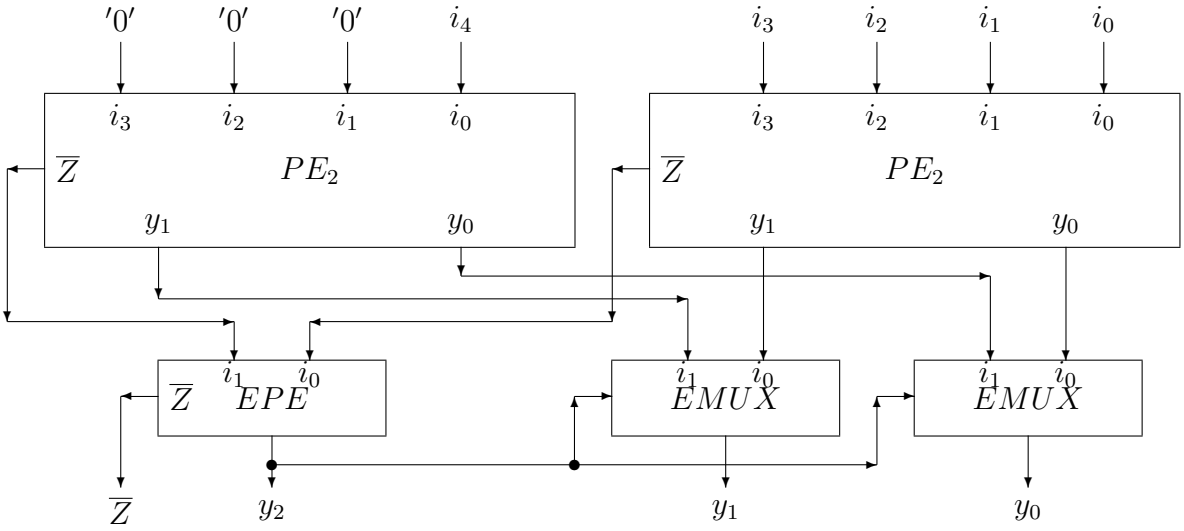
De reţinut că – în general – o extensie paralelă generează circuite rapide.

Dezavantajul constă în mărimea lor, care creşte exponenţial.

Exemplul 5.8 *Să considerăm $m = 5$. Atunci $n = \lceil log_2 5 \rceil = 3$, deci vom avea un PE_3. De remarcat că din cele opt intrări, primele 3 (cele mai semnificative) sunt setate pe $'0'$. Sunt posibile două situaţii:*

1. *$i_4 = 1$. Atunci $\overline{Z}_1 = 1$, deci $\overline{Z} = 1$ şi $y_2 = 1$, indiferent de valorile biţilor $i_3, \ldots, i_0$. Din primul PE_2 iese $y_1 y_0 = 00$, combinaţie selectată de cele două $EMUX$-uri (datorită valorii $y_2 = 1$ luate de funcţia de selecţie).*

   Deci bitul cel mai activ este pe poziţia $(y_2 y_1 y_0)_2 = (100)_2 = 4$.



2. *$i_4 = 0$. Atunci $\overline{Z}_1 = 0$ şi – indiferent de valoarea lui Z_2 – din EPE iese $y_2 = 0$.*

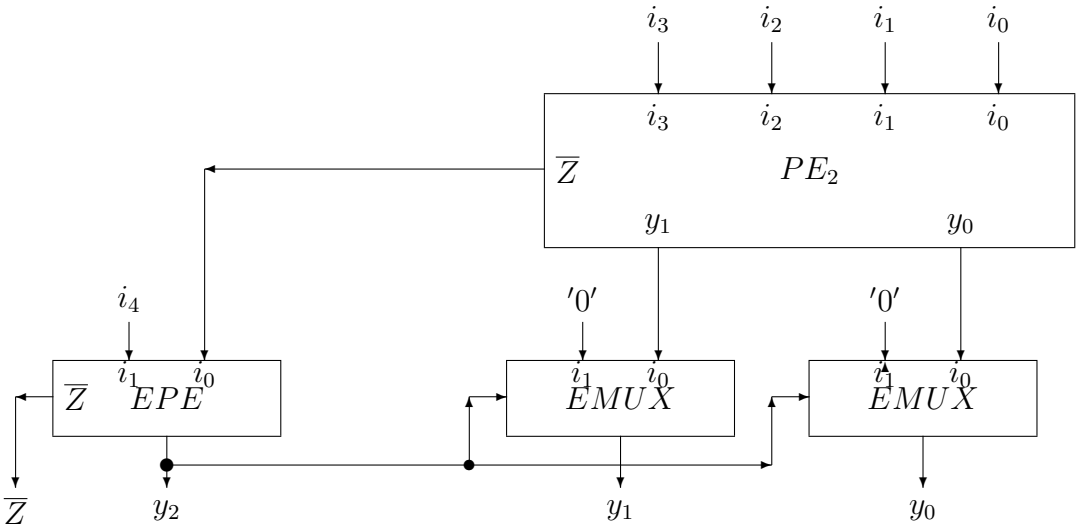
   Dacă $i_3 i_2 i_1 i_0 = 0000$, se obţine $\overline{Z}_2 = 0$, deci $\overline{Z} = 0$ şi nu avem biţi activi.

   Dacă printre aceşti patru biţi există cel puţin unul activ, atunci $\overline{Z}_2 = 1$.

   EPE anunţă existenţa unui bit activ ($\overline{Z} = 1$) pe poziţia $y_2 = 0$.

   Cum y_2 joacă rol de selector pentru cele două $EMUX$-uri, acestea vor transfera la ieşire valorile y_1, y_0 obţinute de al doilea PE_2, deci adresa celui mai semnificativ bit activ din secvenţa $i_3 i_2 i_1 i_0$.

Cum primele trei intrări sunt totdeauna 0, primul PE_2 poate fi eliminat, obţinându-se circuitul



De fapt întotdeauna va fi posibil să eliminăm intrările $i_m, \ldots, i_{2^n-1}$, simplificând corespunzător codificatorii cu prioritate.

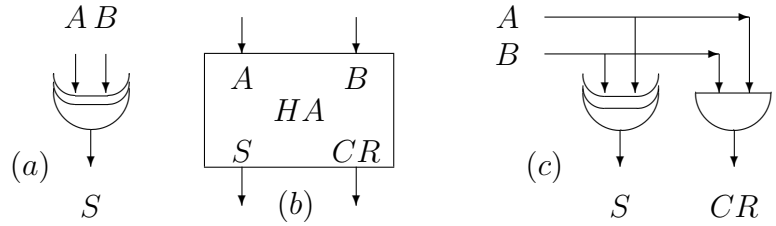
5.6 Sumatori

Una din cele mai importante funcţii digitale este operaţia aritmetică de *adunare*.

Această funcţie se defineşte de obicei plecând de la circuite simple (suma a două numere de 1 bit), capabile să se extindă recursiv pentru a asigura sumatori pe n biţi, pentru n arbitrar de mare.

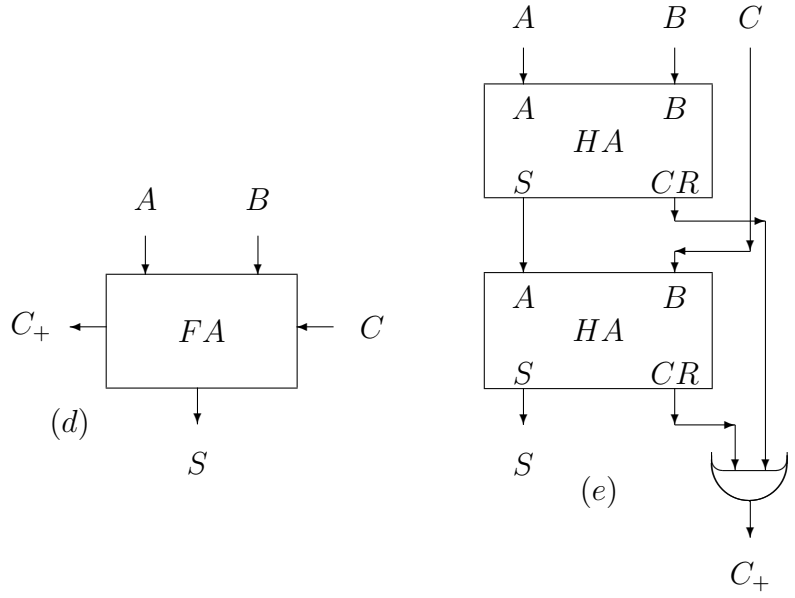
După cum se ştie, interpetarea numerică a funcţiei XOR (figura (a)) este *suma modulo* 2. Pentru extensia ei la suma modulo 2^n este necesar să calculăm şi funcţia de *transport* CR.

Atunci, un *semi-sumator* HA va fi format dintr-un XOR şi un AND (care ia valoarea 1 numai când ambii operanzi sunt 1); Figura (b) dă reprezentarea simbolică a unui semi-sumator, iar Figura (c) - structura sa internă.



După cum am văzut în Capitolul 2, două numere întregi de n biţi ($n > 1$) se adună bit-cu-bit, începând de pe cea mai puţin semnificativă poziţie.

Fiecare sumă de doi biţi este influenţată – prin funcţia de transport – de suma binară anterioară; de aceea, pentru a ţine cont şi de acest transport, se va adăuga o intrare suplimentară C de transport, obţinându-se un *sumator complet* FA (full adder), a cărui reprezentare simbolică este dată de Figura (d) şi conţinut - de Figura (e):



Primul HA însumează cei doi biţi (corespunzători operanzilor A şi B), iar al doilea adună la suma astfel calculată valoarea deplasării; transportul de ieşire este dat de primul sau al doilea HA.

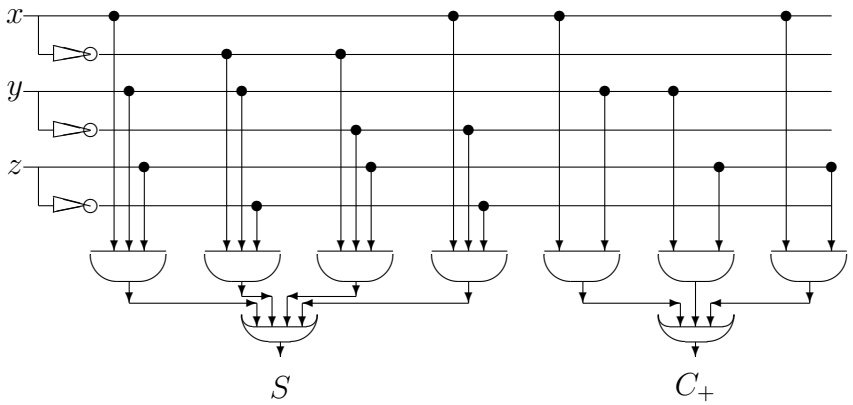
Observaţia 5.6 *Sumatorul complet poate fi construit şi pe baza unui codificator (acesta fiind un circuit universal); într-adevăr, funcţia sum* : $\{0,1\}^3 \longrightarrow \{0,1\}^2$ *de adunare a trei cifre binare poate fi definită prin tabela de valori*

A	0	0	0	0	1	1	1	1
B	0	0	1	1	0	0	1	1
C	0	1	0	1	0	1	0	1
S	0	0	0	1	0	1	1	1
C_+	0	1	1	0	1	0	0	1

De aici putem genera forma canonică celor două componente care formează funcţia sum:

$S = xyz + \overline{x}y\overline{z} + \overline{x}\,\overline{y}z + x\overline{y}\,\overline{z}$
$C_+ = \overline{x}yz + x\overline{y}z + xy\overline{z} + xyz = xy(z+\overline{z}) + xz(y+\overline{y}) + yz(x+\overline{x}) = xy + yz + xz$

Deci, un circuit combinaţional echivalent cu un FA poate fi construit imediat:



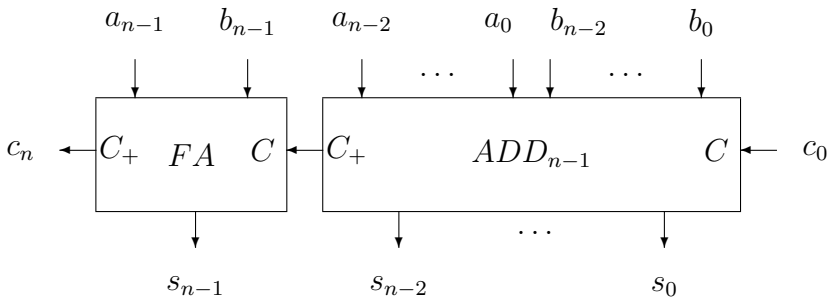
Dacă definim **adâncimea** *unui circuit ca fiind numărul maxim de porţi situate pe un drum, circuitul de mai sus este un circuit combinaţional de adâncime* 3.

Dacă toate porţile permit aceeaşi viteză de traversare, atunci acesta este mai rapid decât definiţia unui FA.

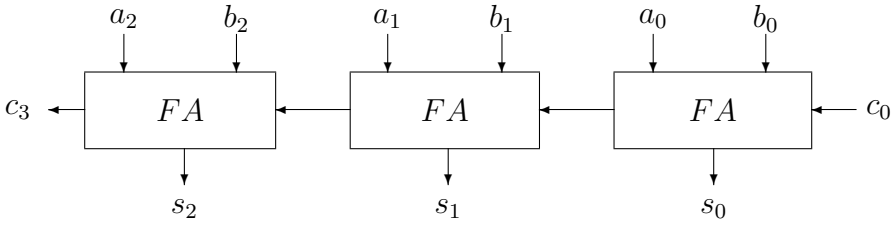
Dezavantajul său major este numărul mare de porţi, deci un cost hard mai ridicat.

O problemă de bază în arhitectura calculatorului constă în realizarea de circuite care să ofere avantaje atât din punctul de vedere al vitezei de lucru, cât şi al costului echipamentului hard.

Un sumator de n biţi ADD_n *în cascadă*, se poate defini recursiv prin extensia serială a unui ADD_{n-1} cu un FA. Iniţial, ADD_1 este un FA.



Exemplul 5.9 *Un sumator ADD_3 construit în cascadă are forma*



Din păcate, în această construcţie serială a circuitului, calculul transportului la fiecare însumare de doi biţi conduce la un proces de calcul lent.

Pentru a obţine o viteză de calcul mare, se foloseşte altă construcţie a sumatorului ADD_n, prin însumarea separată a transportului fiecărei sume binare.

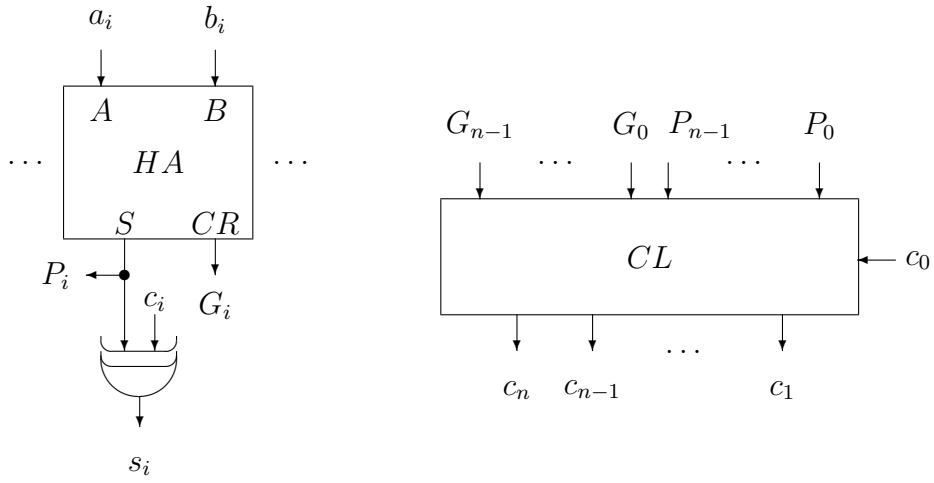
Pentru calculul transportului fiecărei sume binare se introduce un circuit suplimentar de *control al transportului* CL (carry lookahead). Ecuaţiile care descriu CL se bazează pe relaţia

$$c_{i+1} = a_i b_i \oplus (a_i \oplus b_i)c_i = G_i \oplus P_i c_i.$$

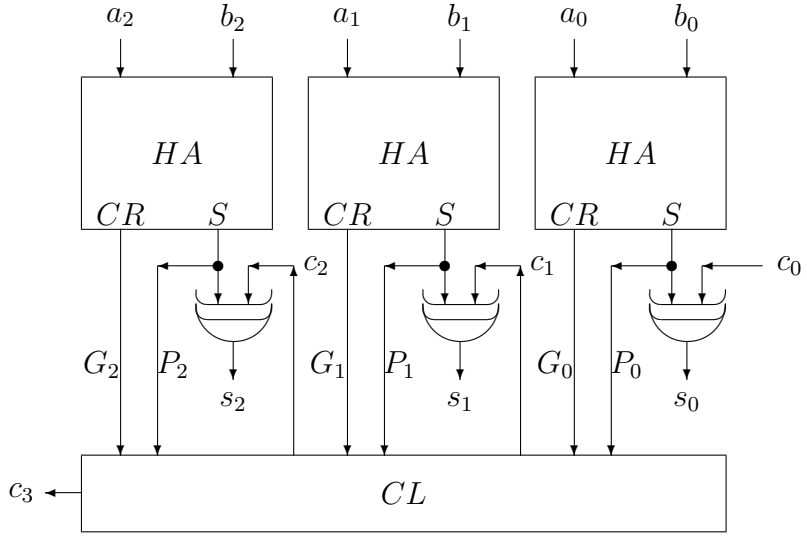
(reamintim, $\oplus$ este notaţia matematică pentru operaţia XOR).

Ieşirile fiecărui HA generează – prin porţile AND şi XOR – ieşirile G_i respectiv P_i. Aplicând această regulă de i ($0 \leq i \leq n$) ori, se obţine:

$$c_{i+1} = G_i \oplus P_i G_{i-1} \oplus P_i P_{i-1} G_{i-2} \oplus \ldots \oplus P_i P_{i-1} \ldots P_0 c_0$$



Exemplul 5.10 *Reluând exemplul unui sumator pe* 3 *biţi, vom avea circuitul*



Un tabel cu valorile scoase de CL *(a cărui structură o lăsăm ca exerciţiu) este:*

a_i	0	0	0	0	1	1	1	1
b_i	0	0	1	1	0	0	1	1
c_i	0	1	0	1	0	1	0	1
c_{i+1}	0	0	0	1	0	1	1	1

5.6.1 Circuit digital pentru incrementare

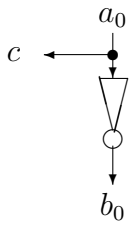
Un caz particular de sumator este cel de incrementare, când al doilea operand este egal cu $'1'$.

Se poate folosi un circuit de adunare obişnuit, din cele prezentate anterior, unde valoarea $'1'$ se extinde pe un număr de biţi egal cu lungimea primului operand; operaţia devine însă costisitoare (ca timp şi spaţiu) în condiţiile în care incrementarea este folosită mult mai frecvent decât adunarea obişnuită.

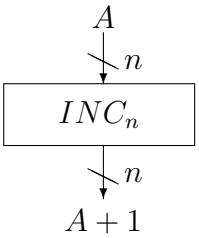
De aceea, ţinându-se cont de valoarea fixă a celui de-al doilea operand, se pot face reduceri substanţiale.

Observaţia 5.7

- *Un FA în care $B = 0$, se reduce la un HA care face suma dintre A şi C.*
- *Un FA în care $B = 1$, $C = 0$, se reduce la un HA care face suma dintre A şi $'1'$. Mai mult, deoarece $s = A \oplus 1 = \overline{A}$, $c = A \wedge 1 = A$, acesta poate fi construit folosind numai un invertor:*



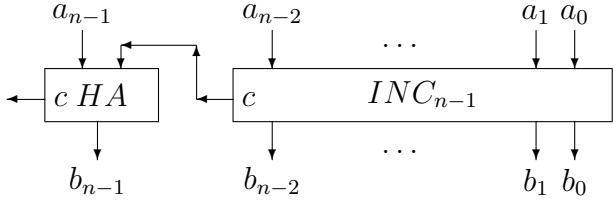
Simbolul pentru circuitul de incrementare al unui număr A – reprezntat pe n biţi – este:



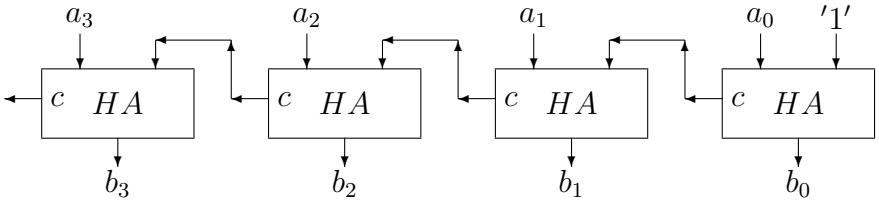
Similar sumatoarelor, şi aici sunt două tipuri de construire a lui INC_n.

O prima variantă – în cascadă – se defineşte recursiv astfel:

- INC_1 este circuitul din Observaţia 5.7.
- INC_n se obţine prin extensia serială a unui HA cu INC_{n-1} sub forma



Exemplul 5.11 *Un circuit de incrementare pe 4 biţi (INC_4) este*



Cel mai din dreapta HA poate fi inlocuit cu invertorul din Observaţia 5.7.

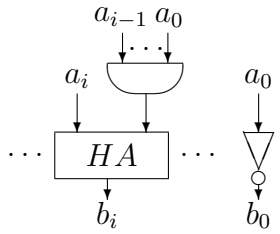
Dezavantajul acestei construcţii este un consum mai mare de timp.

De aceea se poate construi şi o extensie paralelă, bazată pe următoarele observaţii:

Dacă în definiţia ecuaţiilor de transport din construirea CL se iau $c_0 = 1$ şi $b_i = 0$, atunci $G_i = 0,\ P_i = a_i$ $(0 \leq i \leq n-1)$ şi deci

$$c_i = a_{i-1}a_{i-2}\ldots a_0, \quad (i > 0)$$

Pe baza acestei relaţii, INC_n poate fi generat astfel:



unde cele $n-1$ semi-sumatoare (pentru $i = 1, 2, \ldots, n-1$) lucrează în paralel.

Circuitul este extrem de rapid; dezavantajul lui sunt porţile AND_i, a căror complexitate creşte odată cu valoarea lui i.

5.6.2 Circuit digital pentru scădere

După cum ştim (Capitolul 2), operaţia de scădere a două numere întregi poate fi redusă la operaţia de adunare.

Dacă B este un număr pe n biţi, iar $\overline{B}$ se obţine din B prin complementarea tuturor biţilor, atunci $B + \overline{B} = 2^n - 1$ şi deci

$$A - B = A + \overline{B} + 1 - 2^n$$

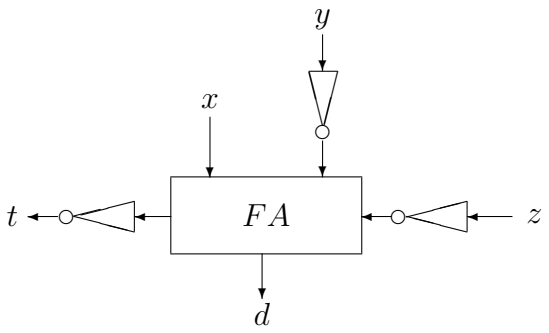
Numărul $\overline{B} + 1$ este reprezentarea lui B în cod complementar, iar operarea cu 2^n inversează bitul final de transport.

Fie funcţia $sub : \{0,1\}^3 \longrightarrow \{0,1\}^2$ definită $sub(x,y,z) = x - y - z$. Tabela sa de valori este

x	0	0	0	0	1	1	1	1
y	0	0	1	1	0	0	1	1
z	0	1	0	1	0	1	0	1
d	0	1	1	0	1	0	0	1
t	0	1	1	1	0	0	0	1

Deoarece $x - y - z = x + (\overline{y} + 1 - 2^1) + (\overline{z} + 1 - 2^1) = x + \overline{y} + \overline{z} + 2^1 - 2^2$, rezultă că diferenţa $d(x,y,z)$ este rezultatul adunării $x + \overline{y} + \overline{z}$, iar $t(x,y,z)$ este valoarea de transport complementată (datorită adunării cu 2^1) a acestei sume.

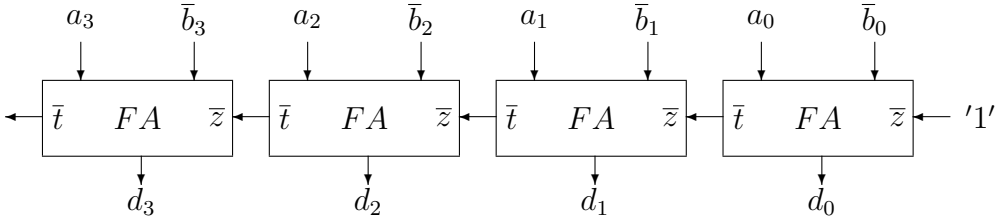
Deci un circuit digital – numit FS (full substract) – pentru scădere este:



Pentru un circuit care să realizeze scăderea a două numere de câte n biţi se va face o extensie serială (în cascadă) formată din n astfel de circuite, similar construcţiei de la ADD_n sau INC_n.

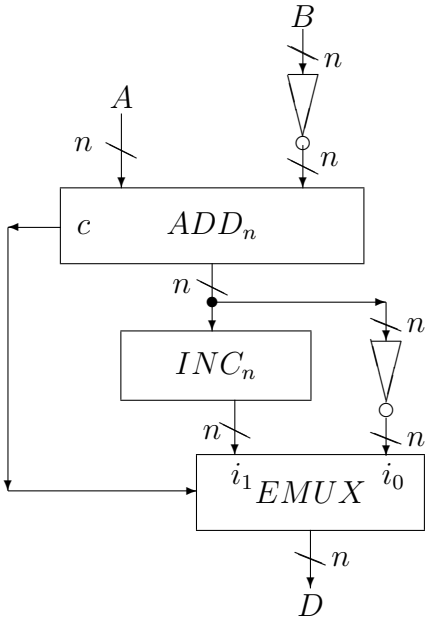
La identificarea transportului t de la un FS cu variabila z de la FS-ul anterior se pot elimina cele două complementări (care se anulează reciproc).

Exemplul 5.12 *Să construim un circuit pentru scăderea a două numere de 4 biţi (FS_4). Schema sa este (pentru simplificare, porţile NOT au fost subînţelese):*

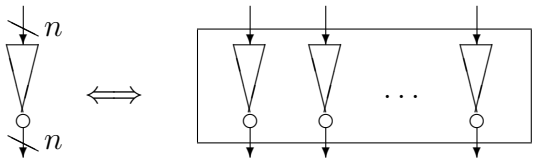


Fie de exemplu numerele $A = 12$, $B = 9$; deci $a_3a_2a_1a_0 = 1100$, $b_3b_2b_1b_0 = 1001$. Introducând în circuitul FS_4 aceste valori (unde $\overline{b}_3\overline{b}_2\overline{b}_1\overline{b}_0 = 0110$) se obţine imediat $d_3d_2d_1d_0 = 0011$, ceea ce verifică operaţia $12 - 9 = 3$.

Un circuit mai compact pentru scădere – care să folosească elementele deja construite – este



unde am reprezentat printr-o singură complementare grupul de n invertori care inversează biţii scăzătorului:



Avantajul acestui circuit este acela că se poate opera şi cu rezultate negative.

Semnul diferenţei este dat de $\overline{c}$ (deci $c = 0$ semnifică un rezultat negativ, iar $c = 1$ – unul pozitiv).

Exemplul 5.13 *Să presupunem că circuitul de scădere este construit pentru* $n = 4$ *şi avem din nou* $A = 12,\ B = 9$.

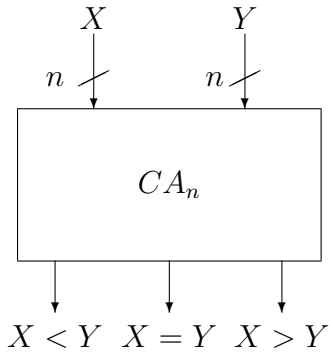
- *Pentru* $A - B$, *sumatorul* ADD_4 *va determina* $A + \overline{B} = 1100 + 0110$ *care dă suma* 0110 *şi transportul* $c = 1$. *Aceasta semnifică un rezultat pozitiv, şi variabila* c – *pe post de selector – va alege din* $EMUX$ *valoarea* 0010 *incrementată, adică* 0011. *Deci rezultatul este* $12 - 9 = 3$.

- *Pentru* $B - A$, *sumatorul va calcula* $B + \overline{A} = 1001 + 0011$, *cu rezultatul* 1100 *şi* $c = 0$. *deci rezultatul este negativ, iar selectorul* $c = 0$ *va alege* $\overline{1100} = 0011$*; în final se obţine* $9 - 12 = -3$.

5.7 Circuite de comparare

Să construim un bloc de comparare între două numere pe n biţi.

O reprezentare simbolică este uşor de dat:



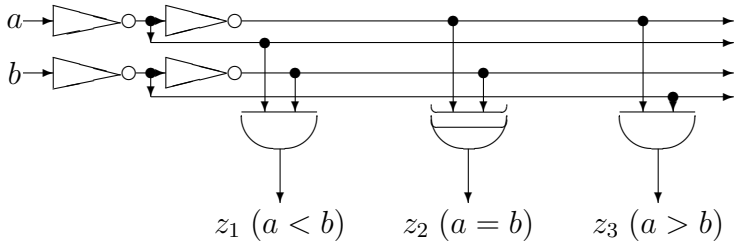
Dar, ce se ascunde în spatele ei ?

Pentru $n = 1$, problema este simplă: este suficient să construim un codificator (de exemplu) bazat pe funcţia $f : \{0,1\}^2 \longrightarrow \{0,1\}^3$ definită prin tabela de valori

a	0	0	1	1
b	0	1	0	1
z_1	0	1	0	0
z_2	1	0	0	1
z_3	0	0	1	0

Dacă $f(a,b) = (z_1, z_2, z_3)$, atunci $z_1 = \overline{a}b$ anunţă dacă $a < b$, $z_2 = \overline{a}\overline{b} + ab = \overline{a \oplus b}$ ia valoarea 1 dacă şi numai dacă $a = b$, iar $z_3 = a\overline{b}$ semnalizează dacă $a > b$.

Un circuit se poate construi imediat:



Problema se complică pentru cazul general ($n > 1$).

Atunci un comparator CA_n va avea $2n$ intrări binare, deci o tabelă de adevăr ar conţine 2^{2n} linii, lucru extrem de dificil de realizat la nivel de codificator (din cauza numărului mare de porţi necesare).

Pentru construcţia unui comparator eficient pe n biţi, să observăm că $X > Y$ este echivalent cu $X - Y > 0$. Cum $Y = (2^n - 1) - \overline{Y}$, relaţia se rescrie

$$X - Y = X + \overline{Y} > 2^n - 1 = 11 \ldots 1$$

unde secvenţa binară de 1 are lungime n.

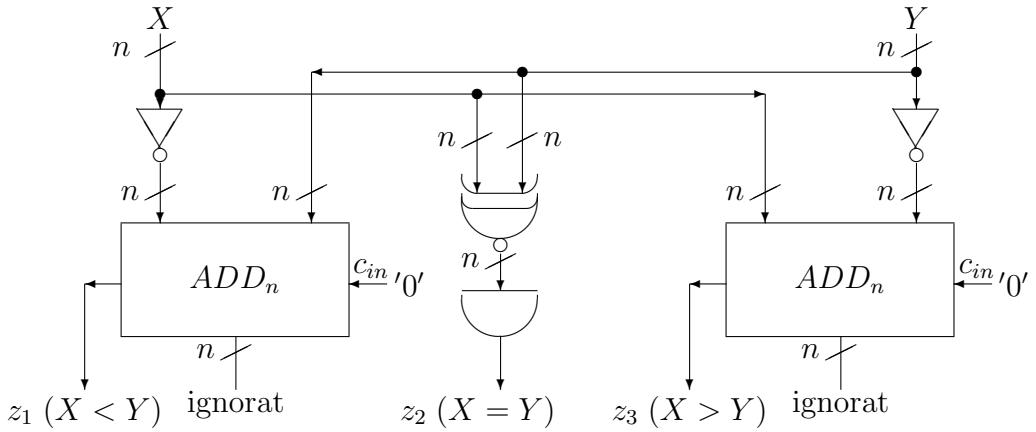
Dacă această inegalitate este satisfăcută, atunci bitul de transport al sumei $c_{out} = 1$, deoarece suma $X + \overline{Y}$ depăşeşte n biţi.

Exemplul 5.14 *Pentru $n = 4$, dacă $X = 1100$, $Y = 1001$, vom avea $\overline{Y} = 0110$ şi deci $X + \overline{Y} = 1100 + 0110 = 10010$, pentru care bitul de transport este* 1.
Concluzia este $X > Y$.

Testarea comparaţiei $X > Y$ poate fi realizată deci în doi paşi:

1. Se află $\overline{Y}$ prin aplicarea unui invertor pe n biţi lui Y;
2. Se efectuează $X + \overline{Y}$ cu un n - sumator şi se foloseşte ca ieşire doar bitul de transport c_{out}. Dacă $c_{out} = 1$ atunci $X > Y$; altfel, $X \leq Y$.

Implementarea acestei scheme arată astfel:



Algoritmul de sus este implementat pentru z_3 $(X > Y)$ pe n biţi.
Dacă se permută X cu Y, se generează z_1 $(X < Y)$.

Suma rezultată nu este utilizată, deci ieşirile corespunzătoare se suspendă.

Mai trebuie determinată ieşirea z_2, necesară pentru cazul $X = Y$.

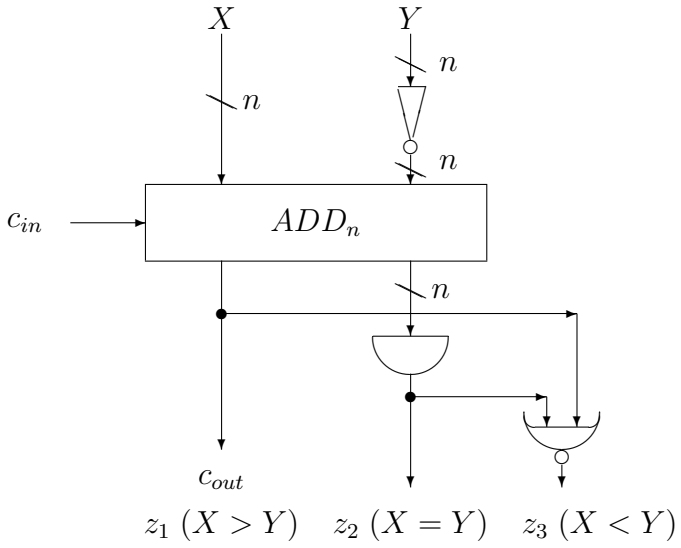
Pentru aceasta trebuie făcută compararea tuturor biţilor x_i din X cu biţii y_i corespunzători din Y.

Dezideratul se atinge cu n porţi $XOR - NOT$ care produc $\overline{x_i \oplus y_i}$.

Vom avea $z_2 = 1$ dacă $\overline{x_i \oplus y_i} = 1,\ (1 \leq i \leq n)$; adică

$$z_2 = (\overline{x_{n-1} \oplus y_{n-1}}) \wedge (\overline{x_{n-2} \oplus y_{n-2}}) \wedge \ldots \wedge (\overline{x_0 \oplus y_0})$$

O altă variantă mai simplă pentru CA_n – cu un singur sumator – este următoarea:



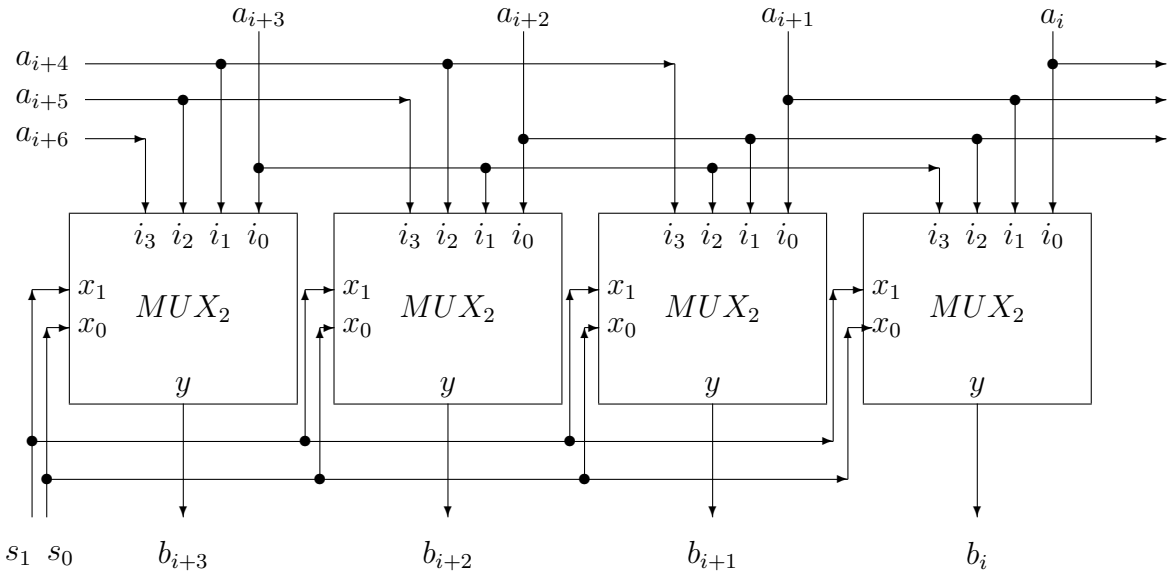
5.8 Circuite de deplasare

O altă funcţie aritmetică de bază este înmulţirea/împărţirea cu o putere a lui 2.

Circuitul care realizează această funcţie se numeşte *circuit de deplasare* (shift).

El constă din mai mulţi multiplexori conectaţi în paralel la aceeaşi intrare şi aceeaşi funcţie de selecţie.

Principiul este descris de figura următoare (pentru cazul deplasării spre dreapta cu cel mult trei poziţii).



O valoare de selecţie $(s_1s_0)_2 = x$ implică $b_j = a_{j+x}$ (la deplasarea spre dreapta) sau $b_j = a_{j-x}$ (la deplasarea spre stânga) pentru $x = 0, 1, 2, 3$ reprezentat în binar.

Dacă $j - x < 0$, atunci $b_j = 0$ (prin deplasarea spre stânga, biţii cei mai puţin semnificativi se completează cu $'0'$)

Exemplul 5.15 *Dacă în figura anterioară valoarea de selecţie este* $(s_1s_0)_2 = (10)_2 = 2$, *atunci fiecare multiplexor conectează ieşirea la intrarea* i_2.

Rezultatul va fi

$$b_{i+3}b_{i+2}b_{i+1}b_i \longleftarrow a_{i+5}a_{i+4}a_{i+3}a_{i+2}$$

În general, pentru operaţia de deplasare cu 0 până la $m - 1$ poziţii sunt necesare mai multe MUX_n unde $n = \lceil log_2 m \rceil$, legate în mod similar cu cele din figură.

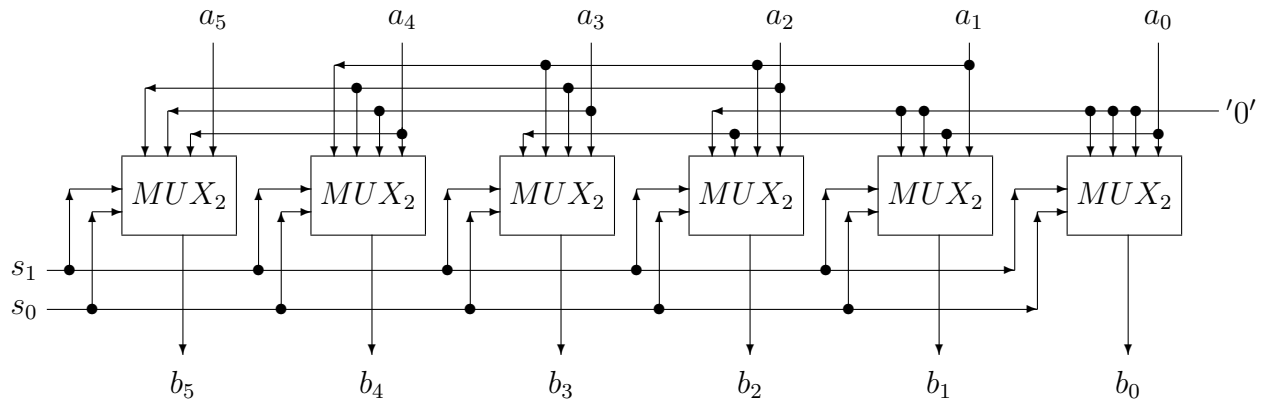
Numărul multiplexoarelor este egal cu numărul de biţi ai numărului asupra căruia se aplică operaţia de deplasare.

Observaţia 5.8 *Deplasarea spre stânga cu x biţi corespunde înmulţirii unui număr cu 2^x.*

În acest caz, cei mai semnificativi x biţi din reprezentarea binară a numărului se pierd, iar cei mai puţin semnificativi x biţi primesc valoarea 0.

În mod similar, deplasarea spre dreapta cu x biţi corespunde împărţirii cu 2^x. Aici se pierd cei mai puţin semnificativi x biţi.

Exemplul 5.16 *Să construim un circuit de deplasare spre stânga cu cel mult 3 poziţii, pentru numere formate din 6 biţi. Se va obţine:*



De exemplu, pentru numărul 9, cu reprezentarea 001001 pe şase biţi, vom avea:

Codul $s_1s_0 = 00$ conduce la ieşirea 001001 (adică tot numărul 9);
Codul $s_1s_0 = 01$ conduce la ieşirea 010010 (adică numărul $18 = 9 \times 2$);
Codul $s_1s_0 = 10$ conduce la ieşirea 100100 (adică numărul $26 = 9 \times 2^2$);
Codul $s_1s_0 = 11$ conduce la ieşirea 001000 (depăşire, datorată pierderii unui bit semnificativ de valoare $'1'$).

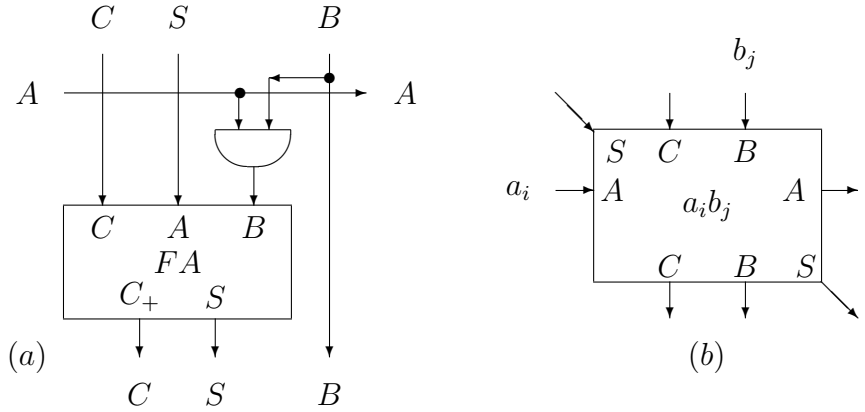
5.9 Multiplicatori

Altă operaţie aritmetică importantă este înmulţirea.

O variantă combinaţională poate fi construită pe baza unui circuit elementar, bazat pe cele două funcţii simple implicate de înmulţirea binară:

- înmulţirea pe un bit (cu ajutorul unei porţi AND);
- un FA pentru completarea operaţiei;

Circuitul de înmulţire completă a biţilor A şi B este descris de Figura (a); Figura (b) defineşte reprezentarea simbolică a înmulţirii biţilor a_i cu b_j.

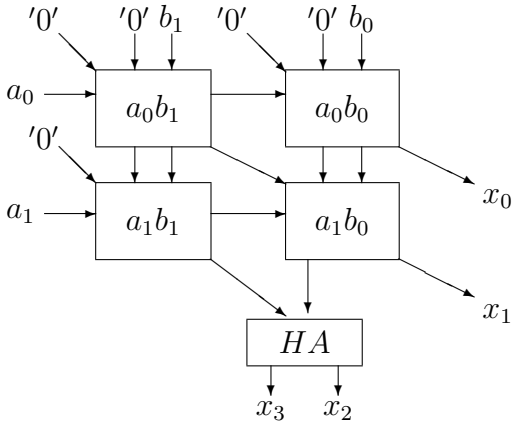


Exemplul 5.17 *Să construim un multiplicator pentru a înmulţi două numere de doi biţi:* $a_1a_0 \times b_1b_0$. *Rezultatul are patru biţi* $x_3x_2x_1x_0$, *care verifică relaţiile:*

$$x_0 = a_0 \cdot b_0; \qquad x_1 = a_1 \cdot b_0 + a_0 \cdot b_1 + c_{x_0};$$
$$x_2 = a_1 \cdot b_1 + c_{x_1}; \qquad x_3 = c_{x_2}.$$

(s-a notat c_{x_i} *transportul obţinut la suma din care a rezultat* x_i*).*

Circuitul este:



Comentarii:

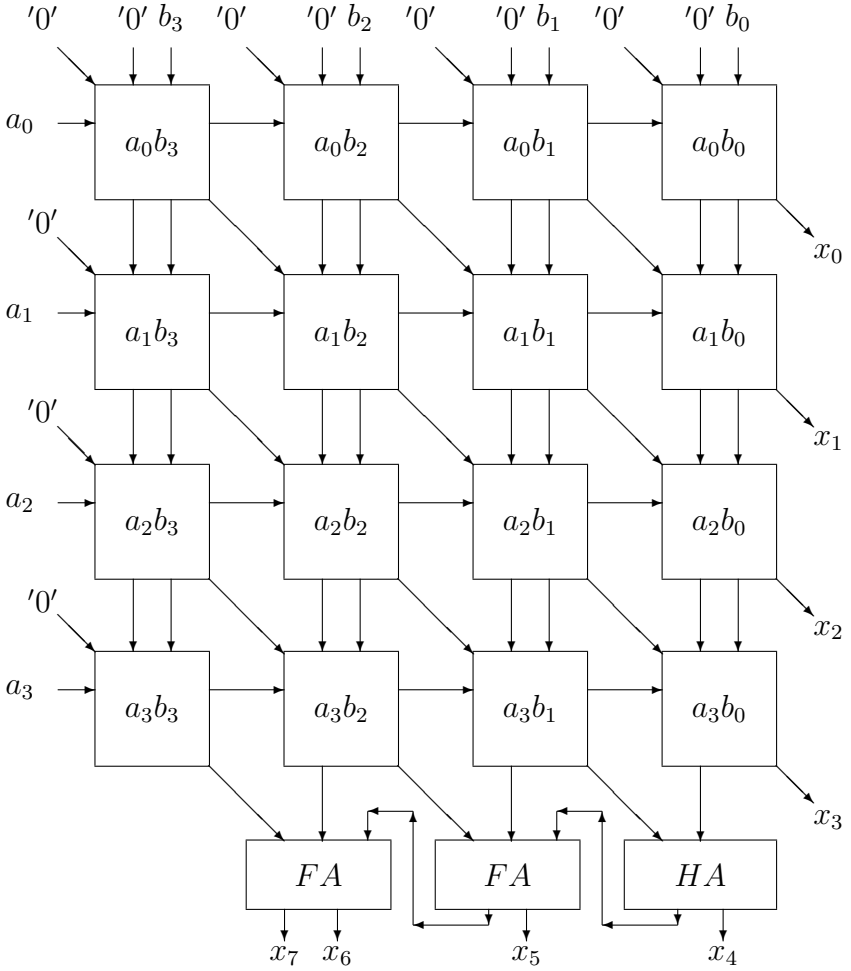
i) x_0 *se obţine din înmulţirea biţilor* a_0 *cu* b_0.

ii) Pentru x_1 *se adună* $a_1 \cdot b_0$ *cu* $a_0 \cdot b_1$ *şi cu transportul de la calculul lui* x_0 *(transport formal, deoarece el este totdeauna egal cu* 0*).*

iii) x_2 *se obţine din adunarea lui* $a_1 \cdot b_1$ *cu transportul de la calculul lui* x_2.

Transportul rezultat din această adunare va fi x_3.

Această structură poate fi generalizată uşor la multiplicatori pentru numere de n *biţi. Prezentăm – pentru claritate – schema unui multiplicator complet pentru cuvinte de* 4 *biţi:*



După cum se vede, complexitatea spaţiu creşte rapid odată cu numărul de biţi ai unui număr.

De aceea, practic se folosesc alţi algoritmi de multiplicare, bazaţi pe adunări şi deplasări (înmulţiri cu puteri ale lui 2).

Problemele de sincronizare care apar fac însă ca aceste circuite să nu fie de tipul $0-DS$.

5.10 Circuit logic programabil

Am văzut că pentru orice funcţie booleană de n variabile se poate construi cel puţin un circuit. În general circuitele generate erau clădite pe două tipuri de structuri:

- recursivă, bazată pe noţiunea de multiplexor MUX_n;
- structura simbolică a unui şir de $m = 2^n$ biţi, ale cărui elemente sunt selectate la ieşirea dintr-un decodificator.

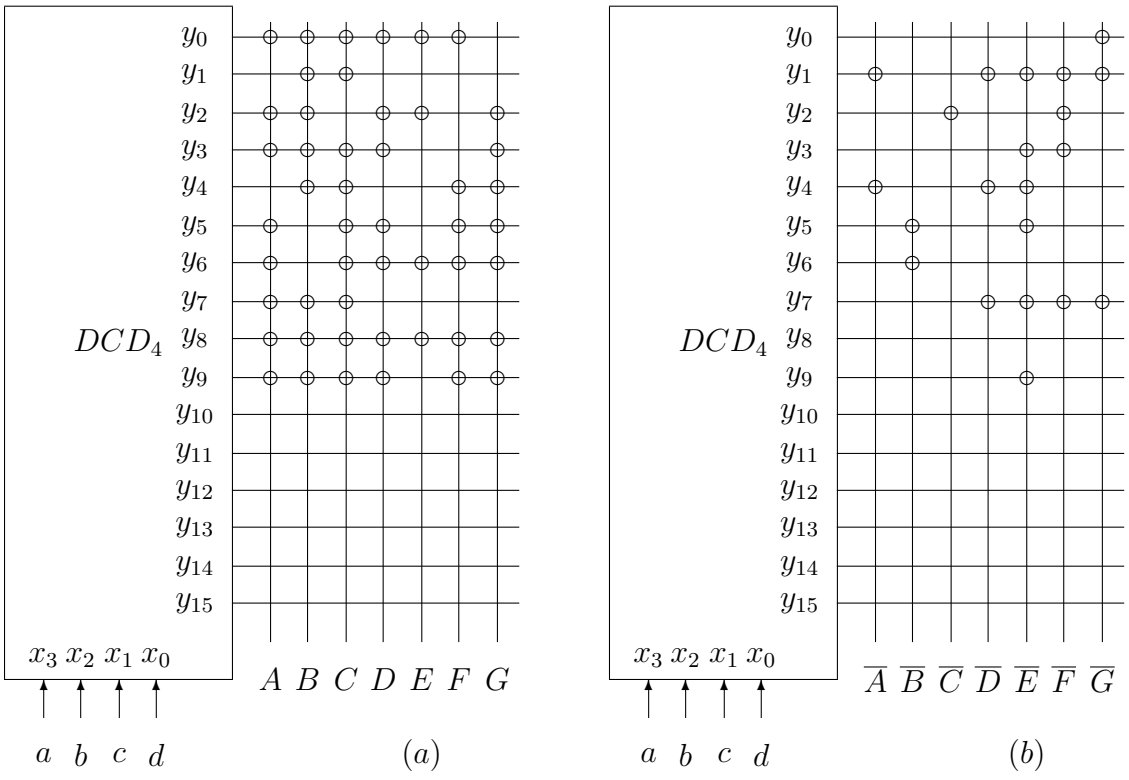
O astfel de soluţie este recomandată numai dacă şirul binar – selectat la ieşirea din decodificator – este aleator (nu are o structură predefinită).

Dacă şirul binar prezintă anumite forme fixe, atunci pot fi efectuate unele simplificări la implementarea circuitelor corespunzătoare, ceea ce duce la scăderea complexităţii lor.

Astfel, pot fi construite circuite cu porţile fixate în prealabil pe o anumită poziţie (închis sau deschis), faza de implementare constând doar în imbinarea lor sub forma unor tablouri.

O astfel de construcţie se numeşte *tablou logic programabil*.

Exemplul 5.18 *Unul din cele mai uzuale circuite combinaţionale este transcoderul care transformă cifrele zecimale scrise în binar, în reprezentări pe* 7 *biţi (codul ASCII).*



Fie a, b, c, d cei patru biţi care codifică cifrele zecimale (de la 0000 pentru 0 până la 1001 pentru 9) şi A, B, C, D, E, F, G biţii din reprezentarea finală (de la $0110000 = 0(60)_8$ - codul ASCII pentru 0, până la $0111001 = 0(71)_8$ - codul ASCII pentru 9).

Trebuie construite deci 7 funcţii de 4 variabile.

Cea mai bună soluţie pare să fie un DCD_4 de tipul celui din paragraful 5.1; va rezulta Figura (a) (numită şi ROM) folosită la implementarea circuitelor combinaţionale.

Pentru simplificarea desenului, porţile OR au fost reprezentate prin linii verticale, iar intrările, prin $\circ$.

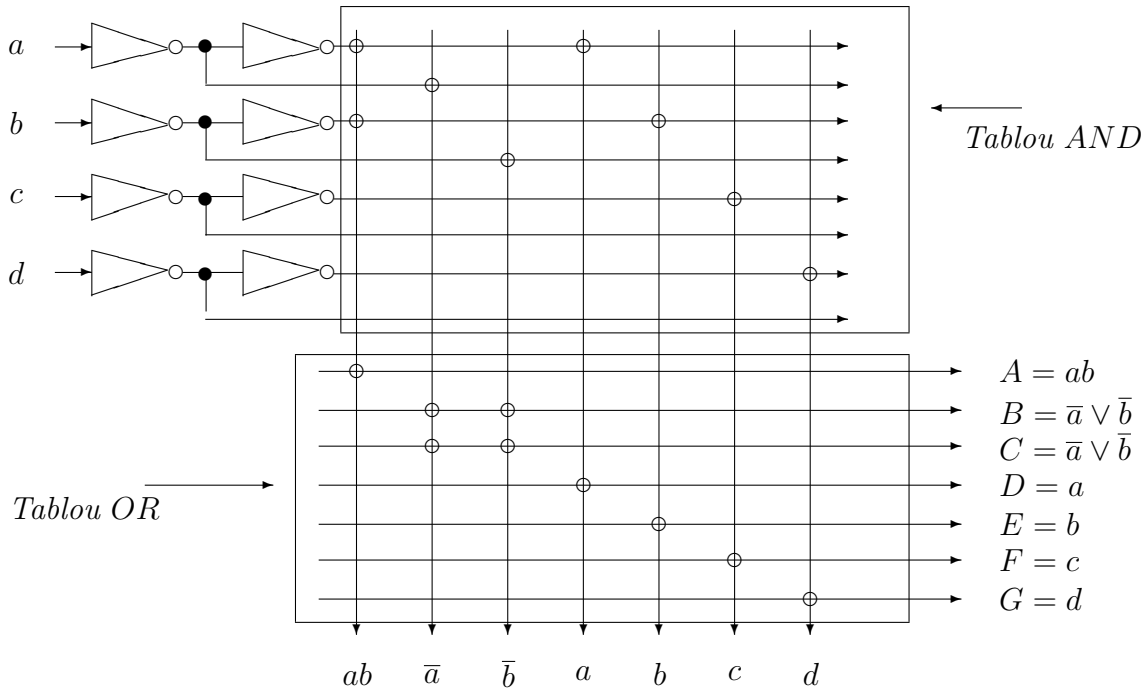
Deoarece circuitul DC_4 conţine $2^4 + 2^3 + 2^2 + 8 = 36$ porţi, iar reţeaua are 49 puncte, rezultă că mărimea acestei soluţii este 85.

O posibilitate de a obţine un circuit mai simplu poate fi prin complementarea funcţiilor; ea conduce (Figura (b)) la în acest caz la un circuit având $36 + 21 = 57$ porţi.

Circuitul este foarte mare – mai ales din cauza ieşirilor nefolosite $y_{10} - y_{15}$.

Decodificatorul tratează toate configuraţiile binare de 4 biţi, deşi numai o parte din ele sunt folosite de cele şapte funtii.

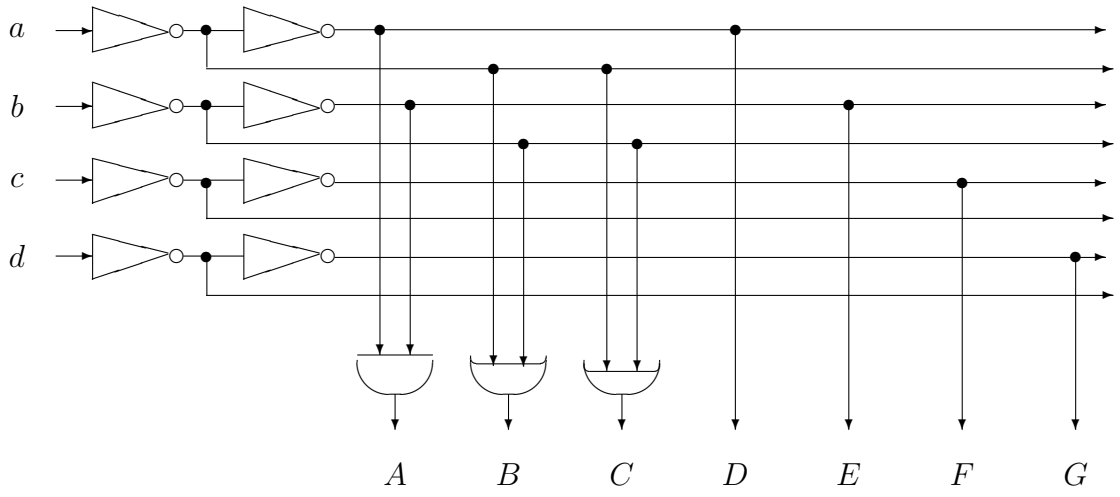
Ţinând cont de această observaţie, se poate construi alt circuit, cunoscut sub numele de "tablou logic programabil" PLA (programmable logic array); soluţia aceasta are numai 25 porţi (şi eventual mai poate fi optimizată).



Valorile funcţiilor $A, \ldots, G$ s-au dedus uşor din tabelul de valori:

a	0	0	0	0	0	0	0	0	1	1
b	0	0	0	0	1	1	1	1	0	0
c	0	0	1	1	0	0	1	1	0	0
d	0	1	0	1	0	1	0	1	0	1
A	0	0	0	0	0	0	0	0	0	0
B	1	1	1	1	1	1	1	1	1	1
C	1	1	1	1	1	1	1	1	1	1
D	0	0	0	0	0	0	0	0	1	1
E	0	0	0	0	1	1	1	1	0	0
F	0	0	1	1	0	0	1	1	0	0
G	0	1	0	1	0	1	0	1	0	1

Evident, PLA-ul de sus nu este cel mai simplu circuit de implementare. Mult mai simplu este decodificatorul (cu numai 11 *porţi)*



Definiţia 5.7 *Un PLA standard $PLA_{n,p,m}$ este un circuit digital format din 4 nivele:*

1. *Nivelul de intrare, compus din n decodificatori elementari ($EDCD$);*
2. *Un decodificator programabil format din $p \ll 2^n$ porţi AND (sau $NAND$, dacă se folosesc regulile De Morgan); fiecare intrare poate fi conectată sau nu la ieşirea corespunzătoare a primului nivel; deci maxim p clase pot fi decodificate la intrarea $X = x_{n-1} \dots x_0$.*
3. *Un codificator programabil format din $m \leq p$ porţi OR (sau $NAND$, dacă se folosesc regulile De Morgan); fiecare intrare poate fi conectată sau nu la ieşirea corespunzătoare de la al doilea nivel, fiecare funcţie $y_{m-1}, \dots, y_0$ putând fi implementată folosind unul sau mai mulţi factori decodificaţi la al doilea nivel;*
4. *Nivelul de ieşire, format din m circuite $EMUX$ (sau XOR - uri cu o intrare conectată la 0 sau 1), cu scopul de a da forma funcţiei – complementată sau nu.*

Deci, PLA este un circuit care începe şi se termină cu cele mai simple circuite funcţionale ($EDCD$ respectiv $EMUX$) şi conţine două tablouri programabile (numite "tablou AND" respectiv "tablou OR").

Un PLA nu este un circuit optim din punct de vedere al complexităţii.

Avantajul său major constă în posibilitatea de a lucra cu tablouri "prefabricate", pe care apoi le poate îmbina cu $EDCD$-uri pentru intrări, $EMUX$-uri pentru ieşiri.

Dacă tabloul OR este inclus într-un ROM (componentă a unui codificator standard), atunci PLA-ul se numeşte de obicei $PROM$ (Programmed ROM).

5.11 Unitatea aritmetică şi logică

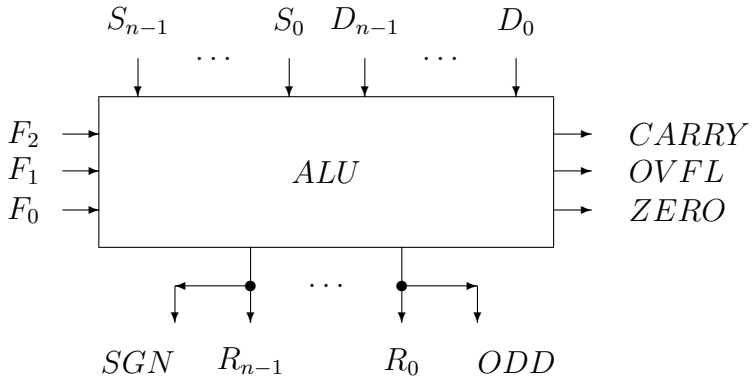
Toate circuitele prezentate până acum aveau asociate câte o singură funcţie (aritmetică, de selecţie sau de comparaţie).

Vom încheia prezentarea sistemelor $0 - DS$ construind un circuit care poate realiza mai multe funcţii, selectate pe baza unui *cod de selecţie*.

Numele sub care este cunoscut acest circuit este *unitatea aritmetică şi logică* (ALU).

Vom considera o unitate relativ simplă cu numai 8 funcţii. În cadrul ei vor exista mai multe tipuri de conexiuni:

- Două intrări de n biţi fiecare, pentru operanzii S (stâng) şi D (drept), codificaţi $S_{n-1}, \ldots, S_0$ respectiv $D_{n-1}, \ldots, D_0$;
- O intrare F de trei biţi (F_2, F_1, F_0) pentru specificarea funcţiei printr-un cod;
- O ieşire $R = (R_{n-1}, \ldots, R_0)$ pentru rezultat;
- O ieşire pentru indicatori (*flags*) - biţi cu diverse semnificaţii: $CARRY, OVFL, ZERO, SGN, ODD$.



Mulţimea funcţiilor poate fi de exemplu:

AND: realizează funcţia logică AND bit cu bit ($R_i = S_i \wedge D_i$);

OR: funcţia logică OR;

XOR: funcţia XOR;

ADD: face suma aritmetică modulo 2^n dintre numerele de n biţi reprezentate binar prin S (operand stâng) respectiv D (operand drept);

SUB: face diferenţa aritmetică dintre S şi D;

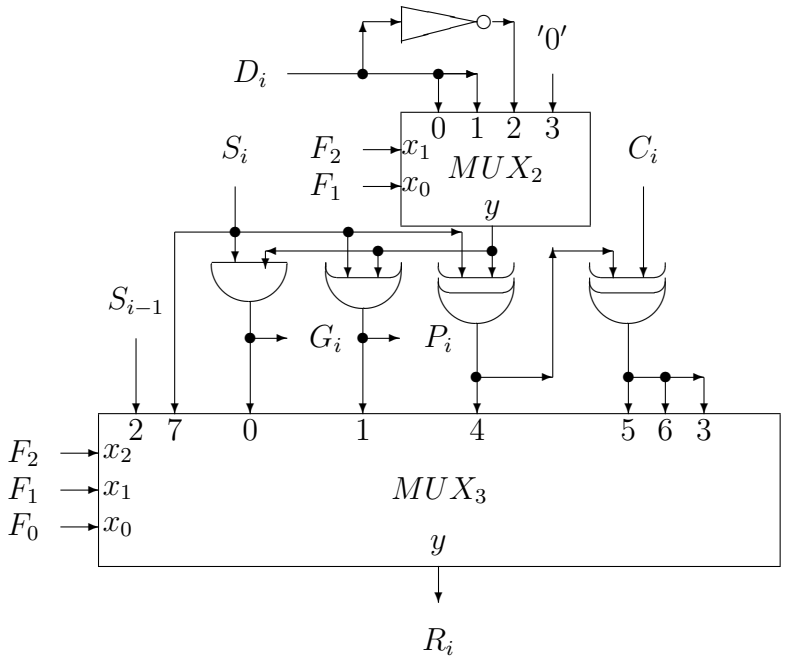
INC: măreşte (incrementează) valoarea lui S cu $'1'$;

LEFT: transferă la ieşire valoarea lui S;

SHL: deplasează cu o poziţie spre stânga valoarea lui S, ultimul bit (cel mai puţin semnificativ) fiind pierdut;

Structura internă a ALU constă în acest caz din n componente ("felii"), câte una pentru fiecare intrare S_i, D_i, un circuit CLC şi câteva porţi suplimentare.

Forma unei astfel de componente este dată de Figura:



- MUX_2 selectează drept al doilea operand valoarea din D_i, complementul ei la 1 (pentru scădere), sau valoarea $'0'$ (pentru incrementare) (C_0 de la intrarea lui CL şi o poartă pentru generarea diferiţilor flag);
- Porţile AND, OR şi primul XOR realizează funcţiile logice şi generează semnale pentru funcţiile aritmetice (G_i, P_i pentru CL şi suma dintre S_i şi D_i);
- Al doilea XOR realizează suma semnalelor de transport generate de CL, completând operaţiile aritmetice;
- MUX_3 selectează – conform codului F – funcţia realizată de ALU;
- Intrarea S_{i-1} este primită de la componenta ("felia") din dreapta, pentru realizarea deplasării;

Descrierea comportamentului acestui ALU este dat de tabelul:

F_2	F_1	F_0	Intrarea	R_i	Operaţia
0	0	0	0	$S_i \wedge D_i$	AND
0	0	1	1	$S_i \vee D_i$	OR
0	1	0	2	S_{i-1}	SHL
0	1	1	3	$S_i \oplus D_i \oplus C_i$	ADD
1	0	0	4	$S_i \oplus D_i$	XOR
1	0	1	5	$(S_i \oplus \overline{D_i}) \oplus C_i$	SUB
1	1	0	6	$S_i \oplus C_i$	INC
1	1	1	7	S_i	$LEFT$

Este evident că cele mai lente operaţii realizate de ALU sunt operaţiile aritmetice, deoarece semnalele sunt întârziate suplimentar de trecerea prin CL pentru generarea semnalelor C_i.

5.12 Exerciţii

1. Se dă funcţia booleană $f : \{0,1\}^3 \longrightarrow \{0,1\}^3$ definită

$$f(x, y, z) = (x + \overline{y}z,\ y + \overline{z}x,\ z + \overline{x}y)$$

   Construiţi un codificator pentru implementarea ei.

2. Construiţi un codificator pentru funcţia $f : \{0,1\}^3 \longrightarrow \{0,1\}$ definită prin octetul 10010100.

3. Construiţi o memorie ROM pe 3 biţi, în care la adresa x se află valoarea $x+3$ $(mod\ 8)$ (codul Excess 3 pe trei biţi).

4. Se dă la intrare o secvenţă de 5 biţi. Să se construiască un circuit care scoate bitul care apare majoritar in acea secvenţă.

5. Se dă la intrare o secvenţă x de 6 biţi. Să se construiască un circuit codificator care scoate valoarea 1 dacă şi numai dacă numărul a cărui reprezentare binară este x, este divizibil cu 4.

6. Construiţi un circuit pentru $DMUX_2$.

7. Daţi o construcţie directă şi una recursivă pentru MUX_3.

8. Să se construiască funcţia *sum* a trei biţi folosind MUX_3.

9. Folosind numai $EMUX$ construiţi un circuit pentru funcţia booleană

$$f(a,b,c,d) = a(b+\overline{c})d + \overline{a}(b+d)(b+c)(c+d) + \overline{b}\,\overline{c}\,\overline{d}.$$

10. Aceeaşi problemă, folosind codificatori.

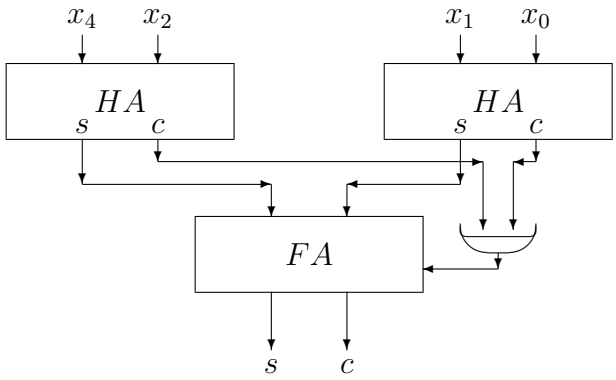
11. Fie funcţia $f(x,y,z) = x + \overline{y} + z$. Să se construiască un circuit combinaţional logic folosind:

    (a) Codificatori;

    (b) Multiplexori elementari.

12. Aceeaşi problemă pentru $f(x,y,z) = (x + \overline{y}z,\ \overline{x}y + \overline{x}\,\overline{y}z,\ y + xz)$.

13. Se dă structura din figură (formată din două HA, un FA şi o poartă OR):



    Construiţi tabela de valori.

14. Construiţi un codificator cu prioritate pentru $m = 7$ biţi.

15. Construiţi un circuit de deplasare dreapta/stânga cu 0,1 poziţii pentru un vector de 4 biţi.

16. Detaliaţi construirea CL (Carry lookahead) pentru un sumator pe 4 biţi.

17. Să se construiască un comparator pe pe 4 biţi şi apoi – pe baza lui – un comparator pe 16 biţi.

18. Construiţi un comparator pe 4 biţi folosind ca bază un singur $4 - FA$.

19. Se dă funcţia booleană $f : \{0,1\}^3 \longrightarrow \{0,1\}^3$ definită

$$f(x,y,z) = (\overline{x} + yz,\ x + \overline{y}z,\ x + y\overline{z})$$

(a) Construiţi un PLA pentru implementarea ei.

(b) Construiţi un $PROM$ pentru implementarea ei.

Capitolul 6

Sisteme $1 - DS$ (Memorii)

6.1 Caracteristicile sistemelor cu un ciclu

Conform ideilor prezentate în Capitolul 4, trecerea spre circuitele de nivel superior constă în introducerea unui ciclu care să închidă din exterior circuitele deja existente.

Apare aici un prim grad de autonomie al circuitului – ceea ce am definit prin *stare*.

Starea unui circuit depinde numai parţial de semnalul de intrare, fapt care va conduce la o independenţă parţială a ieşirii faţă de configuraţia de intrare.

Evoluţia ieşirilor circuitului rămâne bineînţeles sub controlul intrărilor, dar stările oferă o autonomie parţială.

Altă caracteristică a sistemelor din $1 - DS$ constă în posibilitatea de a putea păstra informaţia de intrare o perioadă determinată de timp.

Această proprietate este specifică memoriilor.

Informaţia memorată este pusă la dispoziţia diverselor circuite în anumite faze de lucru. Pentru aceasta apare necesitatea unei sincronizări a operaţiilor în desfăşurare, pentru ca acestea să poată conlucra corect şi eficient.

Motiv pentru care apare un dispozitiv general de control al circuitelor: *ceasul* (CK).

Acest circuit bistabil asigură o discretizare a timpului de calcul, făcând posibilă definirea unor noţiuni temporale cum ar fi *moment actual, tact, istoric, dezvoltare ulterioară* etc.

Principalele circuite cu un ciclu intern sunt:

- *zăvor elementar*: circuit format din două cicluri cuplate la două porţi de intrare;
- *flip - flop* **master - slave**: extensie serială formată din două zăvoare elementare;
- *random access memory (RAM)*: extensie paralelă formată din n zăvoare elementare accesate printr-un $DMUX_n$ şi un MUX_n;

- *registrul*: extensie serial - paralelă formată prin legarea în paralel de flip - flopuri master - slave.

6.2 Cicluri stabile şi instabile

Există două tipuri de cicluri care închid un circuit combinaţional logic; numai unul din ele este util în construcţia circuitelor digitale, anume cel care generează o *stare stabilă*; celălalt circuit generează o stare instabilă la ieşire şi poate fi folosit la construirea ceasului.

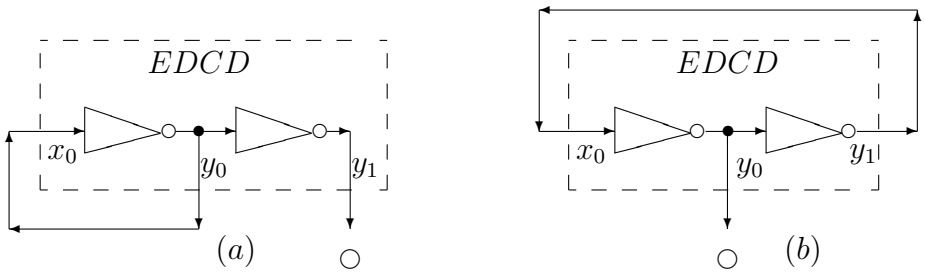
Diferenţa dintre ele poate fi reliefată prin următorul exemplu:

Exemplul 6.1 *Ciclul din Figura* (a) *este închis conectând ieşirea* y_0 *a unui* $EDCD$ *la intrarea* x_0.

Să presupunem că $y_0 = 0 = x_0$.

La ieşire semnalul devine 1, *apoi* 0 *etc; astfel, cele două ieşiri ale decodificatorului sunt instabile, comutând de pe* 0 *pe* 1 *şi invers.*

Momentul de schimbare a valorii de ieşire (din 0 *în* 1 *şi invers) defineşte frecvenţa circuitului şi se numeşte "tact".*



Al doilea tip de ciclu se obţine conectând ieşirea y_1 *la intrarea* x_0 *(Figura* (b)*).*

Dacă $y_1 = 0 = x_0$, *atunci* $y_0 = 1$ *fixând ieşirea* y_1 *la valoarea* 0.

Similar dacă $y_1 = 1 = x_0$.

Deci, acest circuit are două stări stabile.

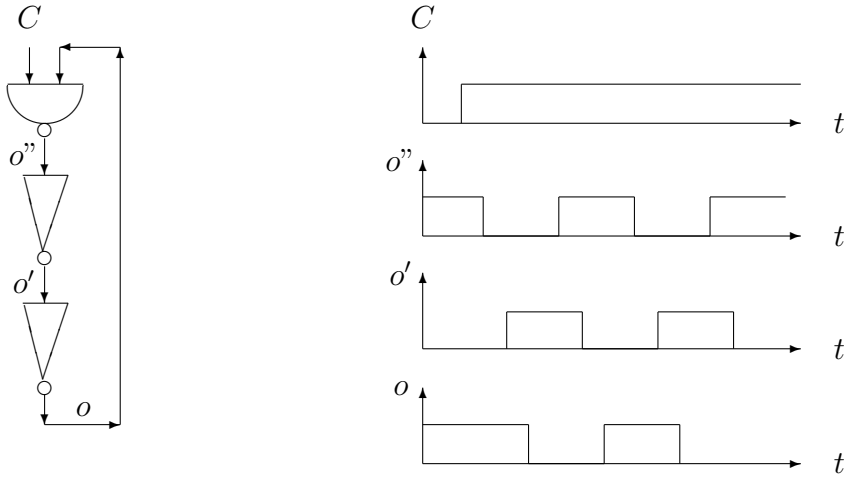
Pentru moment nu ştim cum să comutăm de la o stare la alta, circuitul neavând nici o intrare prin care să putem controla schimbarea.

Principala diferenţă de construcţie dintre cele două structuri este:

- Ciclul instabil are un *număr impar* de complementări (deci semnalul revine la ieşire cu o valoare complementată);

- Ciclul stabil conţine un *număr par* de complementări;

Exemplul 6.2 *Să considerăm un circuit cu* 3 *nivele de complementare:*



Dacă la intrare apare comanda $C = 0$*, atunci ciclul este "deschis", adică fluxul semnalului este întrerupt; dacă valoarea lui* C *devine* 1*, atunci comportarea circuitului este descrisă de diagramele din dreapta.*

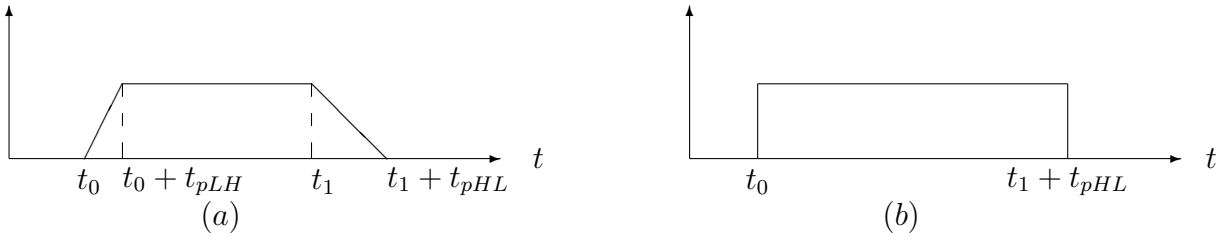
Circuitul va genera un semnal periodic.

Pentru a putea fi utilizat practic, un circuit trebuie să treacă printr-un număr par de complementări *pentru toate combinaţiile binare aplicate la intrare.* Altfel, pot exista combinaţii pentru care circuitul se destabilizează.

Schimbarea stării unui circuit nu este instantanee, ea depinzând de diverse caracteristici fizice ale circuitului.

Vom nota cu t_{pHL} intervalul de timp în care un circuit comută de la starea 1 la starea 0, şi cu t_{pLH} intervalul de timp de trecere de la starea 0 la starea 1.

Ambele valori sunt numere nenegative şi considerate constante pentru un circuit.



Situaţia reală este reprezentată în Figura (a).

Modalitatea de reprezentare din Figura (b) este utilizată atunci când se consideră o situaţie ipotetică – de schimbare instantanee a stărilor.

6.3 Zăvoare elementare

Cel mai simplu circuit stabil – fără nici o inversiune şi cu o singură intrare – este cel din Figurile (a) şi (b):

Primul circuit poate fi comandat să comute numai în starea 0, iar al doilea – numai în starea 1.

Într-adevăr, poarta AND cu valoarea $'1'$ la intrare are ieşire stabilă atât în starea 0 cât şi în starea 1.

Starea poate fi schimbată din 1 în 0 aplicând $'0'$ la intrare.

Timpul minim de trecere în starea 0 este t_{pHL} (timpul de propagare al semnalului prin ciclu).

După acest interval de timp, indiferent de valoarea semnalului de intrare ($'1'$ sau $'0'$), circuitul (a) rămâne stabil în starea 0 – aceasta fiind şi valoarea de ieşire.

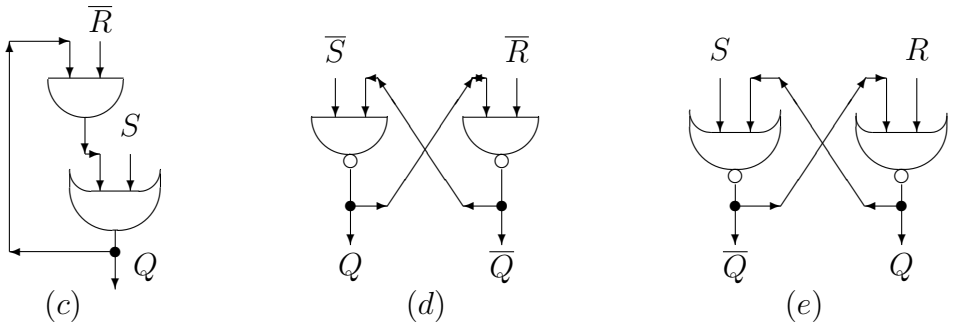
Spunem că *circuitul este activ la frecvenţă joasă.*

La circuitul (b), semnalul inactiv de intrare este $'0'$; circuitul poate trece din starea 0 în starea 1 prin aplicarea la intrare a valorii $'1'$ într-un interval de timp cel puţin egal cu t_{pLH}. După aceea, starea rămâne 1 indiferent de intrare.

Spunem că (b) *este activ la frecvenţă înaltă.*

Ambele circuite *zăvorăsc* în ele semnalul aplicat la intrare: $'0'$ pentru circuitul (a), $'1'$ pentru circuitul (b).

Cele două circuite sunt combinate în circuitul (c).



Acest *zăvor elementar eterogen* are două intrări: una activă la frecvenţă joasă ($\overline{R}$) pentru a reseta circuitul în starea 0, şi alta activă la frecvenţă înaltă (S) pentru a seta circuitul în starea 1.

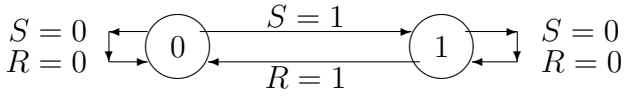
Valoarea $'0'$ trebuie să rămână la intrarea $\overline{R}$ cel puţin $2t_{pHL}$ pentru a fi asigurată comutarea; din acelaşi motiv, valoarea $'1'$ va rămâne la intrarea S cel puţin $2t_{pLH}$ unităţi de timp.

Există următoarele posibilităţi de funcţionare (notăm cu Q_n starea/ieşirea la tactul n):

- $S = 0,\ R = 0$: atunci circuitul rămâne în starea în care a fost ($Q_{n+1} = Q_n$);
- $S = 1,\ R = 0$: are loc o setare a circuitului – $Q_{n+1} = 1$ (indiferent cât a fost Q_n).
- $S = 0,\ R = 1$: circuitul este resetat – se revine la starea $Q_{n+1} = 0$.

De remarcat că varianta $R = 1,\ S = 1$ nu este acceptată, starea Q_{n+1} neputând fi nedeterminată.

O reprezentare a funcţionării circuitului este dată de graful:



Fiind un circuit de tipul $1 - DS$, ecuaţia sa va fi exprimată sub o formă recursivă:

$$Q = S + \overline{R}Q$$

Aplicând regulile lui De Morgan acestei ecuaţii se vor obţine circuite cu structuri simetrice:
- transformarea porţii OR conduce la circuitul (d), construit numai cu porţi $NAND$;
- transformarea porţii AND conduce la circuitul (e), construit numai cu porţi NOR.

În aplicaţii se găsesc de obicei numai aceste ultime două variante.

Punctele slabe ale zăvorului elementar sunt legate de necesitatea de a complementa intrări (la variantele (c) şi (d)) sau de inversare a ieşirilor (la varianta (e)).

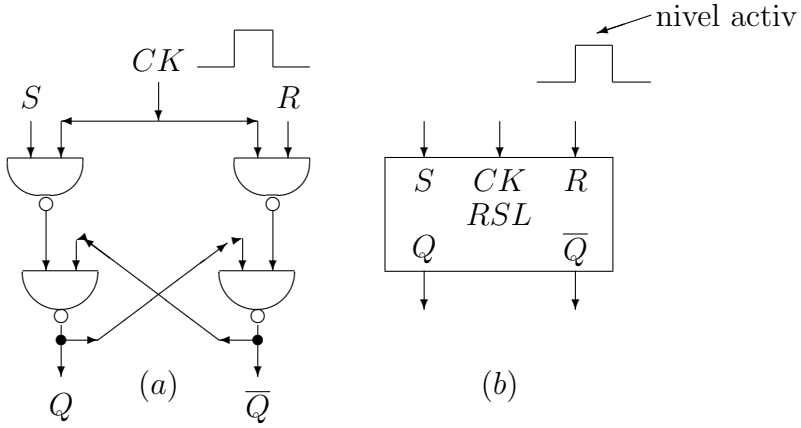
De asemenea, atunci când se solicită schimbarea stării (fără a se preciza valoarea ei), nu ştim ce intrare să acţionăm: setare sau resetare.

6.3.1 Zăvoare elementare cu ceas

O rezolvare parţială a punctelor slabe menţionate anterior constă în extensia serială a două zăvoare (pentru a elimina complementarea variabilelor de intrare din varianta (d)) şi introducerea unui ceas pentru sincronizare.

Se obţine un circuit numit "zăvor elementar cu ceas" (notat RSL).

Structura acestuia este prezentată de Figura (a), iar simbolul logic – de Figura (b).



Cele două intrări sunt pentru setare (S) şi resetare (R), iar ceasul este acţionat acţionând temporar CK pe 1.

Dacă zăvorul trebuie setat, atunci $S = 1,\ R = 0$, după care se face $CK = 1$ pentru un interval de timp egal cu t_{pLH}.

Spunem în acest caz că *nivelul activ* al ceasului este la frecvenţă înaltă (high level).

Pentru resetare, procedura este analogă.

De remarcat că pentru nivelul activ al lui CK zăvorul este *transparent*, adică orice modificare a intrărilor S, R modifică starea circuitului.

Într-adevăr, dacă $CK = 1$, atunci la activarea lui S (R) zăvorul este setat (respectiv resetat).

Această separare dintre cele două semnale conduce la diferenţierea dintre *cum* se realizează schimbarea stării circuitului şi *când* are loc această schimbare.

Zăvorul elementar cu ceas rezolvă o parte din problemele ridicate anterior şi permite un control mai strict asupra acţiunii de memorare a informaţiei.

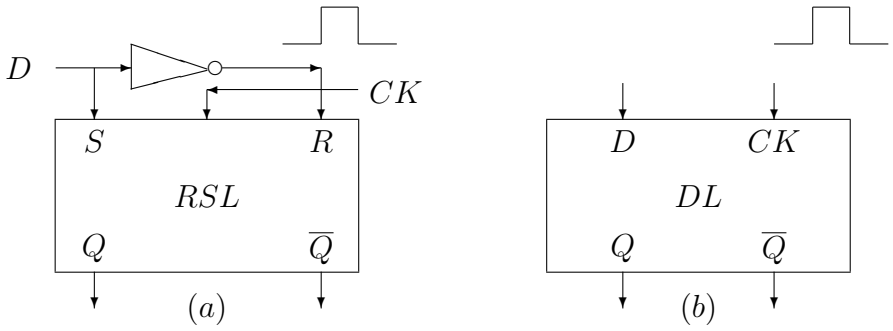
Totuşi o problemă importantă este aceea că un RSL poate comuta de mai multe ori în perioada când ceasul este setat pe $'1'$.

În general aplicaţiile cer schimbarea stării la un anumit moment, pe care îl semnalează prin CK. Ori un RSL permite doar un interval de transparenţă, nu şi o menţinere a stării pe toată durata acestui interval.

6.3.2 Zăvorul de date

În circuitul precedent apare o situaţie specială dacă $R = S = 1$; atunci CK comută zăvorul într-o stare care nu poate fi prevăzută apriori.

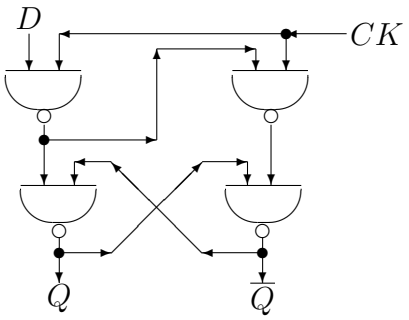
Această variantă poate fi evitată introducând o complementare între intrările unui RSL. Vom nota cu D (date) această unică intrare.



Ieşirea unui DL urmează permanent intrarea D (deci circuitul beneficiază de o autonomie redusă).

Ieşirea este sincronizată cu ceasul numai dacă pe nivelul activ al lui CK intrarea D este stabilă.

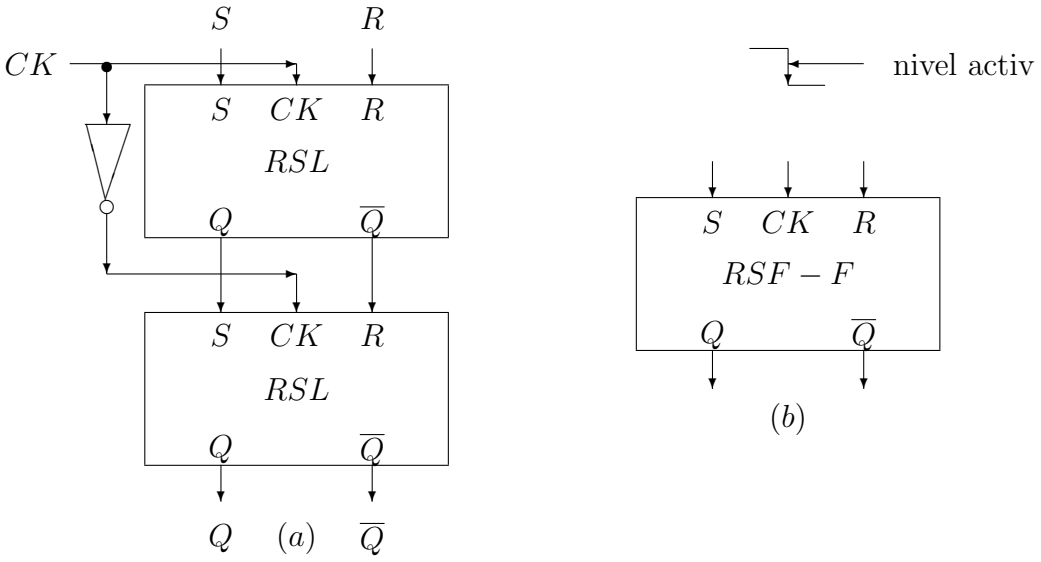
O variantă de construcţie a zăvorului de date – fără nici un invertor – este:



6.4 Structura master - slave

O structură de zăvor, care elimină multe probleme ale circuitelor anterioare, este sistemul numit *master - slave*.

Acesta se bazează pe un circuit cu două stări – numit *flip - flop*, care comută sincronizat cu arcul (în sus sau în jos) semnalului dat de ceas.



Ideea constă în conectarea serială a două zăvoare RSL şi punând (prin intermediul complementării) în antifază semnalul de ceas.

Primul zăvor este transparent la nivelul superior al ceasului (prima jumătate a unui tact), iar al doilea zăvor este transparent la nivelul inferior (a doua jumtătate a tactului).

Deci nu există nici un interval de timp în care toată structura să fie transparentă.

În prima fază (nivelul superior al lui CK), primul zăvor (*master*) comută în conformitate cu semnalele S şi R; în faza a doua (nivelul inferior al lui CK), al doilea zăvor (*slave*) comută copiind starea masterului.

Ieşirea din toată structura este modificată numai după ce a "căzut" tranziţia intrării CK.

Vom spune că circuitul RS *master - slave* comută *flip - flop* conform cu arcul în jos al lui CK.

Pe o structură *master - slave* vom putea deci controla totdeauna momentele când poate fi schimbată starea sistemului.

Există două intervale de timp care trebuie avute în vedere:

- **Timpul de setare** t_S - timpul anterior arcului activ al lui CK, în care intrările S şi R trebuie menţinute stabile pentru a avea siguranţa existenţei unei comutări conforme cu valorile lor;
- **Timpul de susţinere** t_H - timpul posterior arcului activ al lui CK, în care R şi S trebuie menţinute stabile.

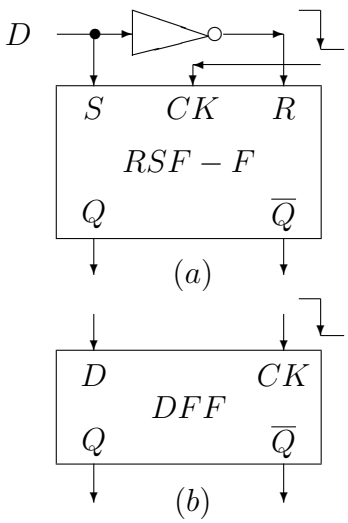
În timpul comutării – adică aproximativ intervalul $t_S + t_H$ – intrările trebuie să fie evident stabile, deoarece altfel flip - flopul nu "ştie" starea în care trebuie să comute.

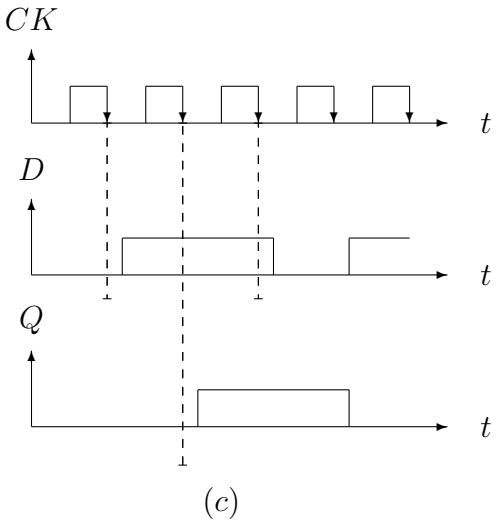
6.4.1 Flip - flop cu întârziere

O altă modalitate de eliminare a situaţiei de nedeterminare $R = S = 1$, deci de a avea $R = \overline{S}$ o constituie următorul circuit, numit *flip-flop cu întârziere* (DFF).

De remarcat că intrarea este numită tot D (ca la "date"), dar acum ea are semnificaţia de *delay*.

În afară de restricţia din construcţie ($R = \overline{S}$), acest flip-flop mai are un avantaj: ieşirea lui D copiază intrarea, întârziată cu un tact.





DFF este unul din cele mai importante circuite construite până acum, în special datorită numeroaselor aplicaţii.

Exemplul 6.3 *Să construim un sumator serial, care adună două numere întregi binare fără semn X, Y ambele de aceeaşi lungime (arbitrară), producând rezultatul $Z = X + Y$.*

Numerele sunt introduse şi prelucrate serial (bit cu bit), iar rezultatul este obţinut tot serial.

La tactul i sumatorul serial primeşte doi biţi x_i, y_i şi calculează un bit z_i.

De asemenea el mai produce şi un bit de transport c_i, care participă la operaţia de adunare de la următorul tact.

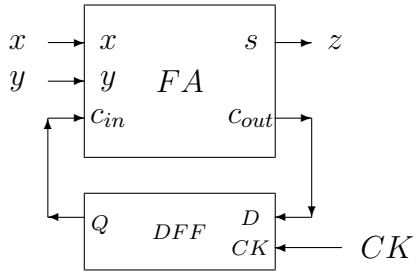
Ieşirea la tactul i este deci

$$c_i z_i = x_i + y_i + c_{i-1}$$

unde valoarea lui c_{i-1} va fi determinată de starea s_i a sumatorului.

Sunt posibile doar două stări interne: s_0 – când $c_{i-1} = 0$, şi s_1 – când $c_{i-1} = 1$.

O memorie cu două stări interne constă dintr-un DFF – care stochează variabila de stare s. O construcţie imediată este



Pentru două numere X, Y de n biţi fiecare, sumatorul serial va lucra $n + 1$ tacţi.

La ultimul tact, intrările x şi y vor primi valoarea $'0'$, iar c_{in} va primi valoarea stocată în DFF.

Ieşirea va fi ultimul bit pentru suma Z, şi valoarea de transport $'0'$ care resetează DFF.

6.5 Memoria *RAM*

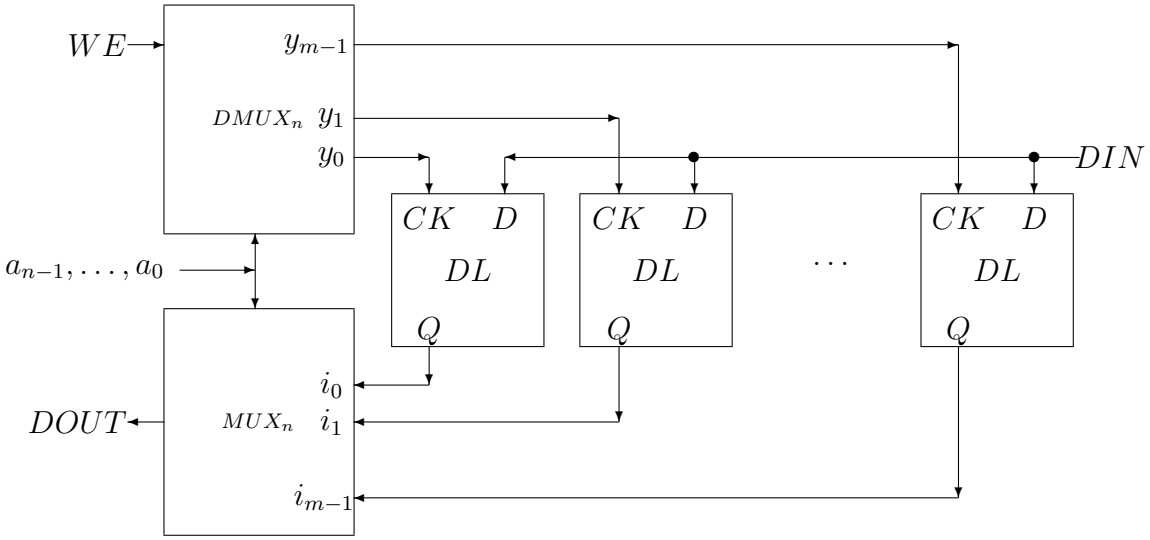
Extensia paralelă a unui $1 - DS$ generează una din cele mai importante circuite din sistemele digitale: **random access memory** (sau pe scurt – memoria RAM).

Fiind unul din cele mai simple circuite, el poate fi construit uşor pe dimensiuni mari.

Definiţia 6.1 *O memorie RAM_m de m biţi este o colecţie liniară de $m = 2^n$ DL-uri cu date de intrare comune, fiecare primind un semnal de ceas distribuit de un $DMUX_n$ şi fiind citite de un MUX_n. Codul de selecţie $a_{n-1}, \ldots, a_0$ este comun pentru $DMUX$ şi MUX.*

Structura asociată acestei definiţii este reprezentată mai jos, unde WE semnifică "*capabil de scriere*" şi este un semnal activ inferior care permite operaţia de scriere în celula de memorie selectată de adresa $a_{n-1}, \ldots, a_0$.

Am notat de asemenea cu DIN magistrala datelor de intrare, şi cu $DOUT$ ieşirea din circuit.



Se poate da şi o definiţie recursivă a memoriei RAM, plecând de la definiţiile recursive ale componentelor sale. Nu vom detalia această variantă.

6.6 Regiştri

O altă componentă de bază a unui calculator o constituie *registrul*.

El este şi principalul suport pentru trecerea la următorul ordin de complexitate al circuitelor – închiderea celui de-al doilea ciclu.

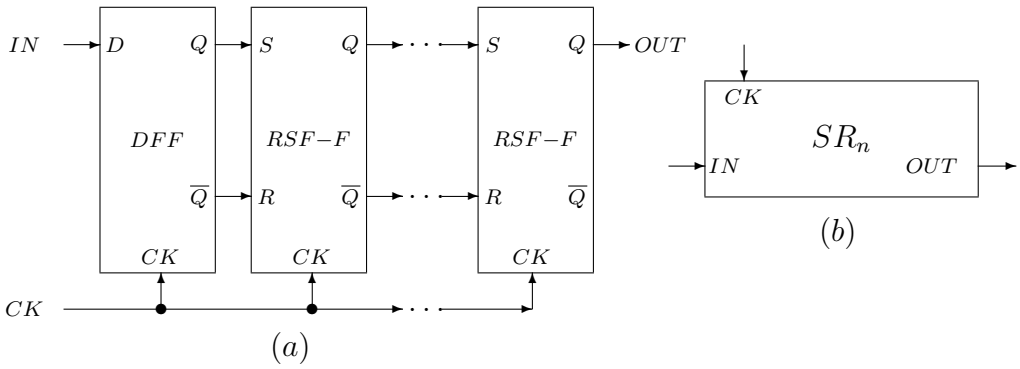
Un registru de n biţi este o secvenţă ordonată de flip-flopuri capabile să stocheze un cuvânt de n biţi.

Fiecare bit este stocat într-un $RSF - F$ sau DFF.

Toate unităţile de memorie au linii de control (ceas, resetare) comune. După modul de extensie folosit, vom avea regiştri seriali, paraleli sau serial-paraleli.

6.6.1 Registrul serial

Figura următoare prezintă un SR_n (structura şi simbolul logic):



Definiţia 6.2 *Un registru serial SR_n de n biţi se defineşte recursiv astfel:*

(i) *SR_1 este un DFF;*

(ii) *SR_n se obţine prin extensia serială a unui SR_{n-1} cu un $SRF - F$.*

Este evident că SR_n introduce o întârziere de n tacţi între intrare şi ieşire.

Un bit care intră într-un SR_n la tactul i, va ieşi din registru la tactul $n + i$, fiecare tact permiţând trecerea acestei informaţii de la un flip-flop la următorul.

Din acest motiv, regiştrii seriali sunt folosiţi cu precădere în construcţia liniilor controlate "de întârziere".

Rolul ceasului este de a sincroniza componentele registrului.

6.6.2 Registrul paralel şi registrul serial - paralel

Definiţia 6.3 *Un registru (paralel) R_n de n biţi se defineşte recursiv astfel:*

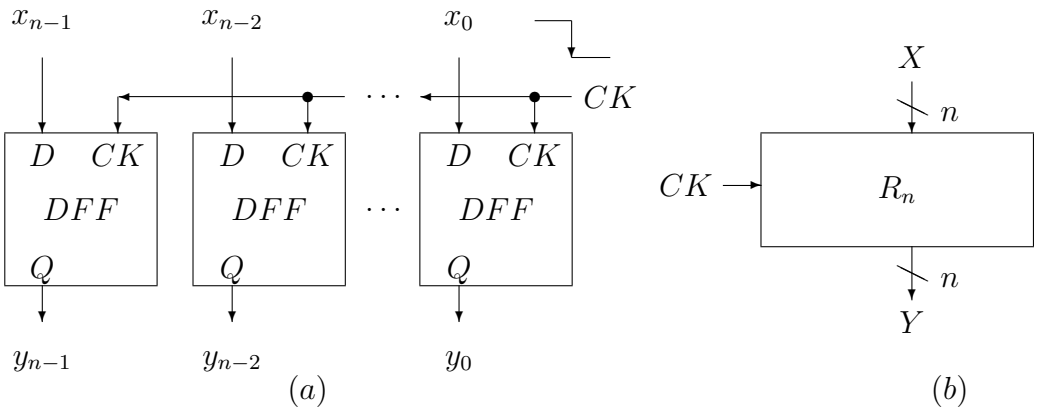
(i) R_1 *este un* DFF*;*

(ii) R_n *se obţine prin extensia paralelă a lui* R_{n-1} *cu un* DFF*.*

Principalul avantaj al acestei componente este netransparenţa sa, cu excepţia unei "transparenţe nedecidabile" în primele $t_S + t_H$ momente.

Deci el poate fi închis eventual cu un nou ciclu.

Mai mult, netransparenţa asigură posibilitatea de a încărca registrul cu orice valoare, inclusiv cu o valoare care depinde de propriul său conţinut.



Mai mulţi regiştri paraleli pot fi legaţi în serie, obţinându-se un *registru serial - paralel* $R_{n\times m}$.

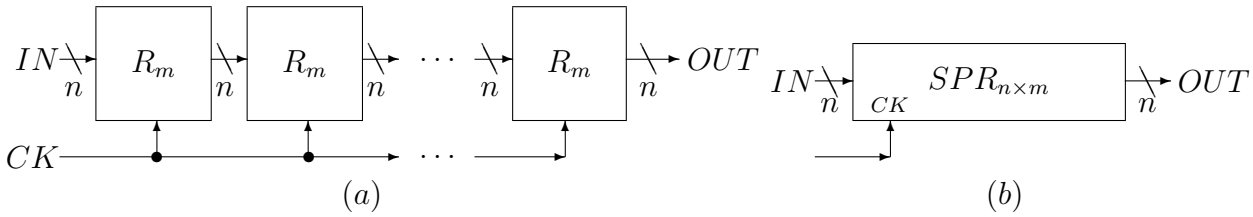
Definiţia sa recursivă este:

Definiţia 6.4 *Un registru serial - paralel* $SPR_{n\times m}$ *se obţine recursiv astfel:*

(i) $SPR_{1\times m}$ *este registrul* R_m*;*

(ii) $SPR_{n\times m}$ *se obţine prin extensia serială a unui* $SPR_{(n-1)\times m}$ *cu un* R_m*.*

Structura şi simbolul logic al unui astfel de registru sunt:



6.7 Exerciţii

1. Fiind date două numere pe n biţi (cu primul bit drept bit de semn) să se construiască un circuit care să selecteze pe cel mai mare dintre ele.

2. Fie inelul boolean $\mathcal{Z}= (Z_2, \oplus, \cdot)$. Fiind fixat polinomul

$$a(X) = a_0 \oplus a_1 X \oplus a_2 X^2 \oplus \ldots \oplus a_n X^n \in Z_2[X]$$

să se construiască un circuit de înmulţire cu $a(X)$ în $\mathcal{Z}$.

Detaliind, să se construiască un circuit care să primească la intrare coeficienţii unui polinom $b(X) \in Z_2[X]$ şi să scoată coeficienţii polinomului $a(X) \cdot b(X)$.

3. Să se construiască un circuit de împărţire la un polinom dat

$$a(X) = a_0 \oplus a_1 X \oplus a_2 X^2 \oplus \ldots \oplus a_n X^n \in Z_2[X]$$

Caz particular: $a(X) = X \oplus 1$.

4. Se dă un DFF. Să se construiască un circuit a cărui ieşire este 1 dacă şi numai dacă DFF schimbă starea (din 0 în 1 sau din 1 în 0). De exemplul pentru intrarea 00110 ieşirea va fi 00101.

5. Să se construiască un circuit care are ieşirea 1 dacă cel puţin doi din ultimii trei biţi de intrare sunt 1. De exemplu, pentru intrarea 10011010011 ieşirea va fi 00001110001.

Capitolul 7

Sisteme $2 - DS$ (Automate)

Următorul pas în construcţia sistemelor digitale constă în adăugarea unui nou ciclu la sistemele de ordin 1. Acest al doilea ciclu creşte autonomia comportării sistemului inclus.

Acum sistemele au o evoluţie a spaţiului stărilor parţial independentă de dinamica intrării.

Vom lua în considerare numai *circuitele sincrone*: circuite în care orice ciclu conţine cel puţin un registru.

7.1 Definiţii de bază

Structura fundamentală a unui sistem $2 - DS$ este *automatul*[1].

Definiţia 7.1 *Un automat A este un 5 - tuplu $A = (Q, X, Y, \delta, \lambda)$ unde*

- *Q este o mulţime nevidă de elemente numite "stări";*
- *X este mulţimea (finită, nevidă) a variabilelor "de intrare";*
- *Y este mulţimea (finită, nevidă) a variabilelor "de ieşire";*
- *$\delta : Q \times X \longrightarrow Q$ este funcţia "de tranziţie" a stărilor;*
- *λ este funcţia de ieşire, având forma*

  $\lambda : Q \times X \longrightarrow Y$ *pentru automatul tip Mealy,*

  $\lambda : Q \longrightarrow Y$ *pentru automatul tip Moore.*

[1] În teoria limbajelor formale, termenul corespunde celui de *translator finit*

La fiecare tact, starea automatului comută şi ieşirea ia valoarea conform cu noua stare (şi valoarea curentă de intrare – în structura Mealy).

Definiţia 7.2 *Un automat cu întârziere (Mealy sau Moore) este un automat având la ieşire valorile generate de un registru (cu întârzieri); deci valoarea curentă de la ieşire corespunde stării interne anterioare a automatului.*

Teorema 7.1 *Relaţia de timp dintre valorile de intrare şi ieşire poate cuprinde următoarele forme:*

1. *Pentru automatul Mealy,* $y_t = \lambda(q_t, x_t)$*;*
2. *Pentru automatul Mealy cu întârziere,* $y_t = \lambda(q_{t-1}, x_{t-1})$*;*
3. *Pentru automatul Moore,* $y_t = \lambda(q_t) = \lambda(\delta(q_{t-1}, x_{t-1}))$*;*
4. *Pentru automatul Moore cu întârziere,* $y_t = \lambda(q_{t-1}) = \lambda(\delta(q_{t-2}, x_{t-2}))$*.*

Demonstraţia este evidentă şi pleacă de la definiţie.

Din această teoremă rezultă că avem la dispoziţie automate cu diverse faze de reacţie la variaţiile intrării.

Deoarece în toate implementările cunoscute mulţimea Q a stărilor este finită, vom folosi următoarea variantă de automat:

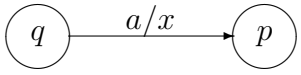
Definiţia 7.3 *Un automat finit este un automat în care Q este finită iar δ acceptă o definiţie nerecursivă.*

Este interesant, dar această definiţie conduce la o structură mai complexă pentru automatele finite, decât structura generală a automatelor.

În continuare vom folosi numai automate *Mealy* (se poate demonstra că – abstracţie făcând eventual de primul caracter de ieşire – cele două tipuri de automate sunt echivalente).

Automatele pot fi descrise şi sub forma unui graf orientat, în care nodurile sunt marcate cu stările din Q, iar arcele se marchează cu perechi de elemente din $X \times Y$ în felul următor:

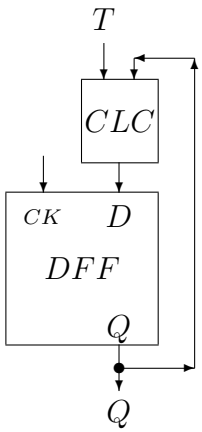
$$\text{Există un arc de la } q \text{ la } p \text{ marcat cu } (a, x) \iff d(q, a) = p,\ \lambda(q, a) = x$$



Mărimea şi complexitatea unui automat sunt direct proporţionale cu dimensiunile mulţimilor care le compun.

Deci, din acest punct de vedere, cel mai simplu automat are numai două stări $Q = \{0, 1\}$ (reprezentate pe un bit), un bit de intrare $T \in \{0, 1\}$ şi $Y = Q$.

Structura asociată este



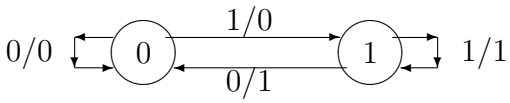
unde există un bit de intrare T, un registru de un bit (de obicei DFF) pentru stocarea unei stări de un bit, şi un circuit combinaţional logic CLC pentru calculul funcţiei δ.

Exemplul 7.1 *Un DFF poate fi exprimat ca un automat cu două stări (CLC se reduce la funcţia identică). Formal $Q = X = Y = \{0, 1\}$, iar funcţiile $\delta, \lambda : \{0, 1\}^2 \longrightarrow \{0, 1\}$ sunt definite prin tabelele de valori*

δ	0	1
0	0	1
1	0	1

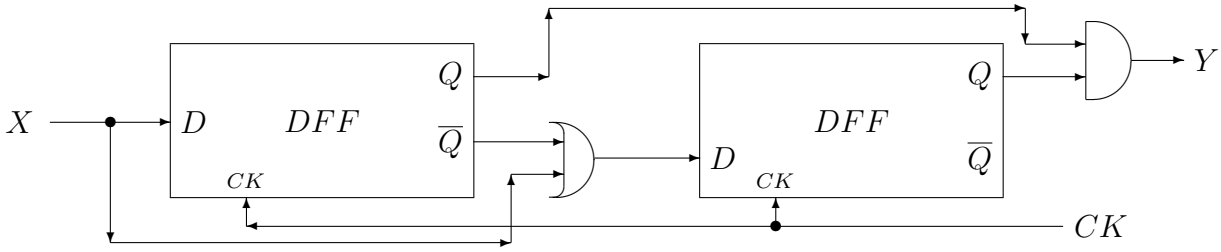
λ	0	1
0	0	0
1	1	1

Graful de funcţionare al unui DFF este



Observaţia 7.1 *Pentru reprezentarea funcţionării unui DFF, automatul Moore este mai adecvat, ieşirea copiind permanent starea în care se afla circuitul.*

Exemplul 7.2 *Să considerăm circuitul compus din două DFF, o poartă AND şi o poartă OR:*



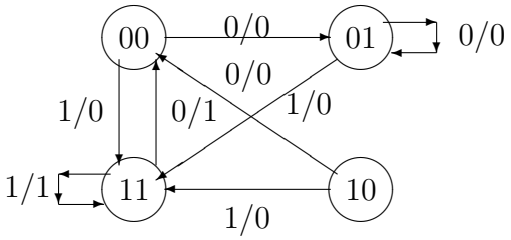
Circuitul are o singură intrare, patru stări date de combinaţiile $Q_1Q_2 \in \{00, 01, 10, 11\}$, *intrările în* DFF*-uri* $D_1 = X,\ D_2 = X \vee \overline{Q}_1$ *şi o ieşire* $Y = Q_1 \wedge Q_2$.

Deci $X = Y = \{0, 1\},\ Q = \{00, 01, 10, 11\},\ \delta(Q_1Q_2, x)$ *şi* $\lambda(Q_1Q_2, x)$ *definite prin tabelele*

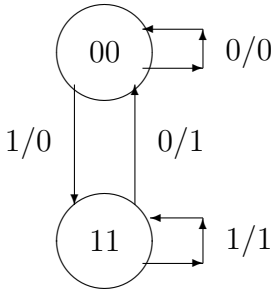
δ	0	1
00	01	11
01	01	11
10	00	11
11	00	11

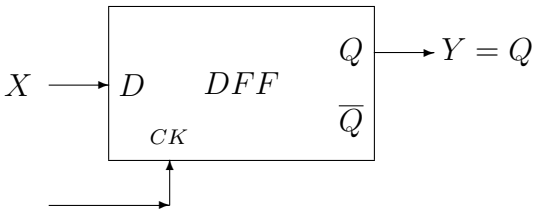
λ	0	1
00	0	0
01	0	0
10	0	0
11	1	1

Graful automatului este



În acest automat, starea 10 *este inaccesibilă din starea de început* 00, *iar stările* 00 *şi* 01 *sunt echivalente (funcţionează identic)). Dacă le elimină*[2] *obţinem un automat cu numai două stări (graful din stânga)*





Dacă se renotează starea 00 *cu* 0 *şi* 11 *cu* 1, *se obţine un* DFF *(figura din dreapta).*

Deci circuitul din acest exemplu poate fi înlocuit cu un flip-flop de date.

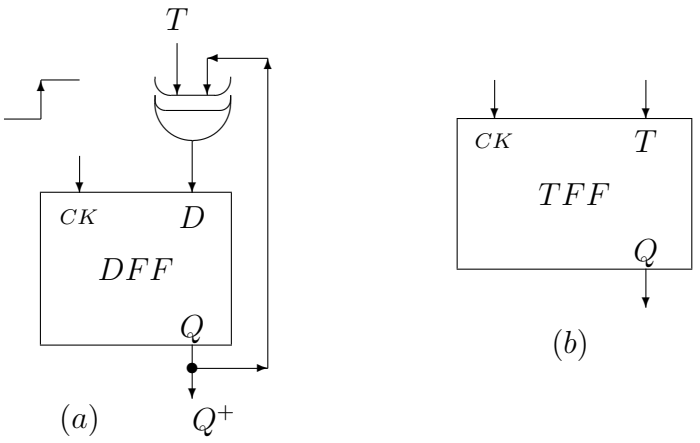
[2]Algoritmii sunt identici cu cei de la Limbaje formale.

7.2 Flip - Flopuri

Folosind structura bazată pe DFF-uri definită în secţiunea anterioară, se pot construi alte două flip-flopuri de eficienţă sporita: T flip-flopuri şi JK flip-flopuri.

7.2.1 Automatul $T\ Flip-Flop$

Dacă drept circuit logic combinaţional folosim o poartă XOR se obţine circuitul (a) – numit T *flip-flop* (TFF pe scurt) – având notaţia (b):



Ecuaţia de funcţionare este $Q^+ = \delta(Q,T) = T \oplus Q$ iar ieşirea $Y = \lambda(Q,T) = Q$.

Ce ar putea semnifica "mesajul" adus de un bit T pentru acest automat cu două stări ?

$T = 0 \implies$ starea automatului rămâne aceeaşi ($Q = 0$ sau $Q = 1$).

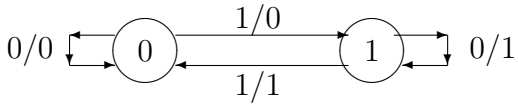
$T = 1 \implies$ starea automatului comută.

Structura de automat este: $Q = X = Y = \{0, 1\}$,

δ	0	1
0	0	1
1	1	0

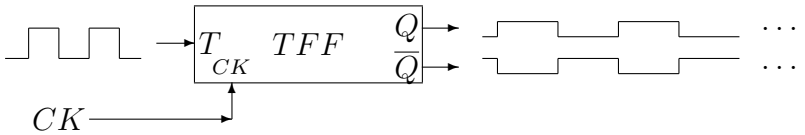
λ	0	1
0	0	0
1	1	1

iar graful de funcţionare



Acest circuit simplu poate fi folosit cu mai multe interpretări:

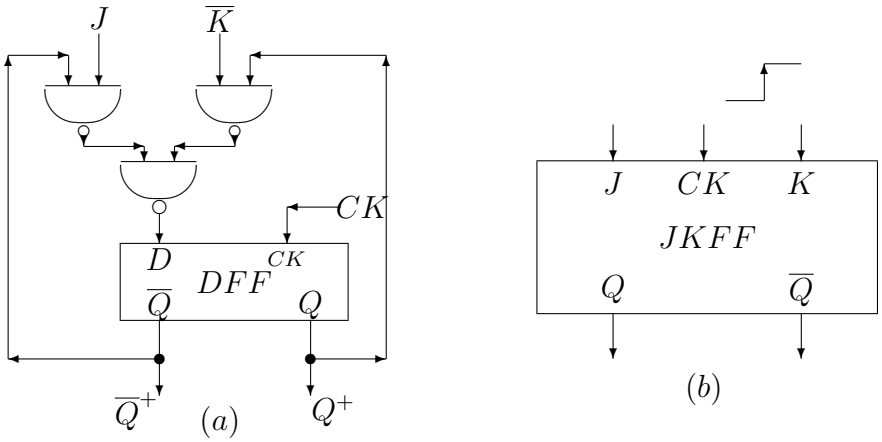
- Ca numărător modulo 2, deoarece pentru $T = 1$ timp de mai mulţi tacţi, ieşirea va fi $01010101\ldots$;
- Ca divizor de frecvenţă. Dacă frecvenţa ceasului este f_{CK}, atunci frecvenţa semnalului primit la ieşirea din automat este $f_{CK}/2$ (după fiecare ciclu, circuitul se întoarce în aceeaşi stare).



Observaţia 7.2 *Câteva remarci referitoare la semnalele de ieşire Q şi $\overline{Q}$:*
- Ele sunt exact inverse unul altuia;
- Frecvenţa lor este exact jumătate din frecvenţa semnalului de intrare.

7.2.2 Automatul $JK\ Flip-Flop$

Să considerăm alt automat, cu două intrări notate J şi K, numit *JK flip flop* ($JKFF$):

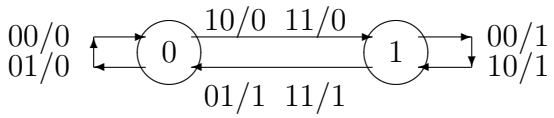


Un $JKFF$ are $Q = Y = \{0,1\}$, $X = \{0,1\}^2$ şi

δ	00	01	10	11
0	0	0	1	1
1	1	0	1	0

λ	00	01	10	11
0	0	0	0	0
1	1	1	1	1

Graful de funcţionare va fi deci



Din structură se deduc ecuaţiile de tranziţie

$$\delta(Q, JK) = (Q \wedge \overline{K}) \vee (\overline{Q} \wedge J), \qquad \lambda(Q, JK) = Q$$

Deci structura de funcţionare poate fi exprimată mai simplu prin tabelul

J	0	0	1	1
K	0	1	0	1
D	Q	0	1	$\overline{Q}$

Cele patru mesaje de intrare posibile sunt:

1. **no op**: $J = K = 0$ – ieşirea rămâne nemodificată (ca la T flip flop cu $T = 0$);
2. **reset**: $J = 0,\ K = 1$ – ieşirea ia valoarea 0 (ca la $DF - F$);
3. **set**: $J = 1,\ K = 0$ – ieşirea ia valoarea 1 (ca la $DF - F$);
4. **switch**: $J = K = 1$ – ieşirea comută pe starea complementară (similar cu $T = 1$ la T flip flop).

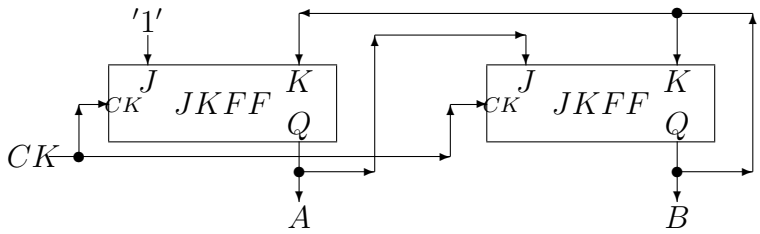
Ciclul este obligatoriu numai pentru ultima funcţie; aici flip - flopul trebuie să "ştie" care era starea anterioară a automatului, pentru a o putea schimba.

Prin această comandă ciclul îşi arată autonomia.

Am obţinut o maşină cu două stări şi două intrări, cu proprietatea că pentru orice configuraţie de intrare posibilă, circuitul are o comportare previzibilă.

JK flip - flopul este cel mai bun flip - flop construit până acum; toate celelalte pot fi considerate cazuri particulare ale sale (pentru $J = K = T$ se obţine TFF, iar pentru $\overline{K} = J = D$ avem DFF).

Pe baza acestui flip - flop se pot construi divizori impari de frecvenţă. Astfel, circuitul următor (cu o singură intrare şi două $JKFF$) asigură o diviziune prin 3 a unui tact.



Pentru intrarea constantă $J = 1$, ieşirea va fi 00 10 11 00 10 .... Ieşirea A va avea valoarea 1 la două momente din trei, iar ieşirea B – la un moment din trei.

Deci o extensie serială a unei magistrale de ceas cu acest circuit face posibilă segmentarea unui tact în cicluri de 1/3 (33, 333%) şi 2/3 (66, 667%).

7.3 Numărători (counteri)

Multe circuite digitale folosite în extrem de numeroase aplicaţii, necesită numărări (înainte şi înapoi).

De aceea s-au dezvoltat diverse modalităţi de generare a numărătorilor.

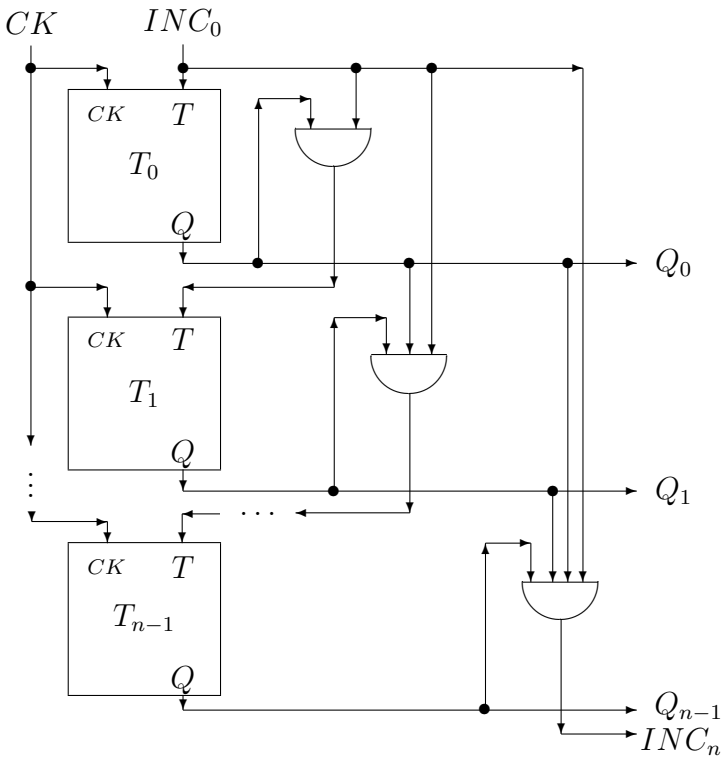
Deşi există multe variante, principiile de bază sunt aceleaşi.

Astfel, o primă caracteristică comună a lor este generarea recursivă.

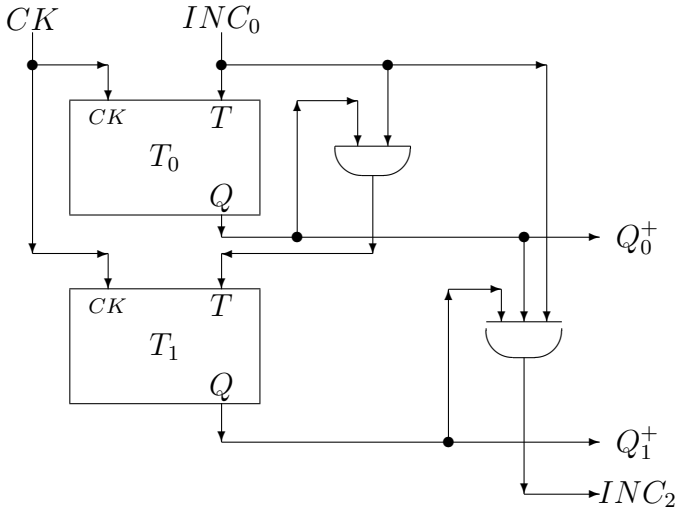
O variantă de construcţie pleacă de la circuitul TFF: ştim că un astfel de circuit poate număra modulo 2, deci cum am putea să legăm n T flip - flopuri $T_{n-1}, \ldots, T_0$ pentru a număra modulo 2^n ?

O idee este următoarea: circuitele vor fi aşezate astfel ca T_i $(1 \leq i \leq n-1)$ să comute doar dacă $T_{i-1}, \ldots, T_0$ sunt simultan în starea 1; în plus, T_0 comută la fiecare pas.

Pentru a detecta aceste condiţii, pentru fiecare $i = 0, 1, \ldots, n-1$ se va asocia un TFF T_i cu o poartă AND_{i+1}, conform figurii următoare:



Exemplul 7.3 *Un counter pentru $n = 2$ are forma*



Dacă notăm cu X intrarea în circuit, cu T_0, T_1 intrările în cele două TFF şi cu Q_0^+, Q_1^+ ieşirile din cele două TFF, ecuaţiile de funcţionare sunt

$$T_0 = X, \qquad T_1 = Q_0^+ \wedge X$$

iar ieşirile

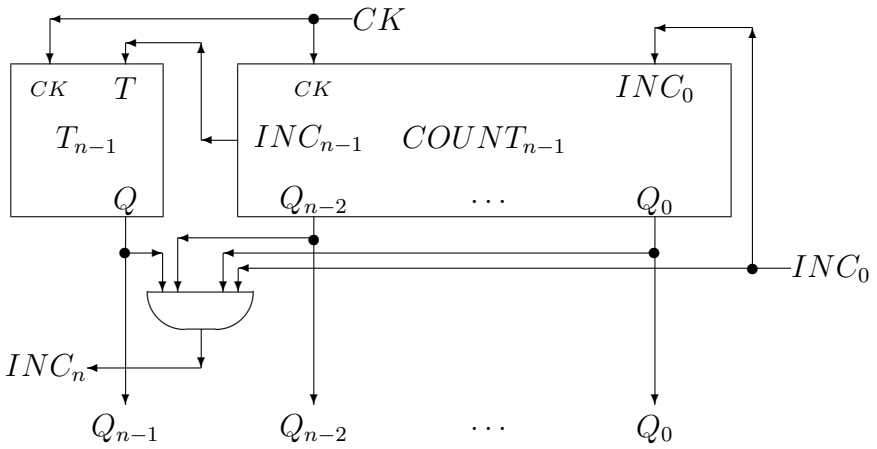
$$Q_0^+ = Q_0 \oplus X, \qquad Q_1^+ = Q_1 \oplus (Q_0^+ \wedge X), \qquad INC_2 = X \wedge Q_0^+ \wedge Q_1^+$$

Comportamentul circuitului poate fi sumarizat în tabela

X	Q_0	Q_1	T_0	Q_0^+	T_1	Q_1^+	*Ieşire*$(Q_1^+Q_0^+)$	INC_2
–	0	0	0	0	0	0	00	0
1	0	0	1	1	0	0	01	0
1	1	0	1	0	1	1	10	0
1	0	1	1	1	0	1	11	1
1	1	1	1	0	1	0	00	0
1	0	0	1	1	0	0	01	0
		...		...			...	...

Extinzând această idee, într-o definiţie recursivă un numărător sincron $COUNT_n$ de n biţi este construit prin extensia serială dintre $COUNT_{n-1}$ şi un TFF, folosind o poartă AND_{n+2} care determină condiţia de comutare pentru următorul TFF; intrarea o constituie INC_0 şi $Q_{n-1}, \ldots, Q_0$ (figura de jos este parţială) iar ieşirea INC_n.

$COUNT_1$ este un TFF şi un AND_2, cu intrarea Q_0, INC_0 şi ieşirea INC_1.

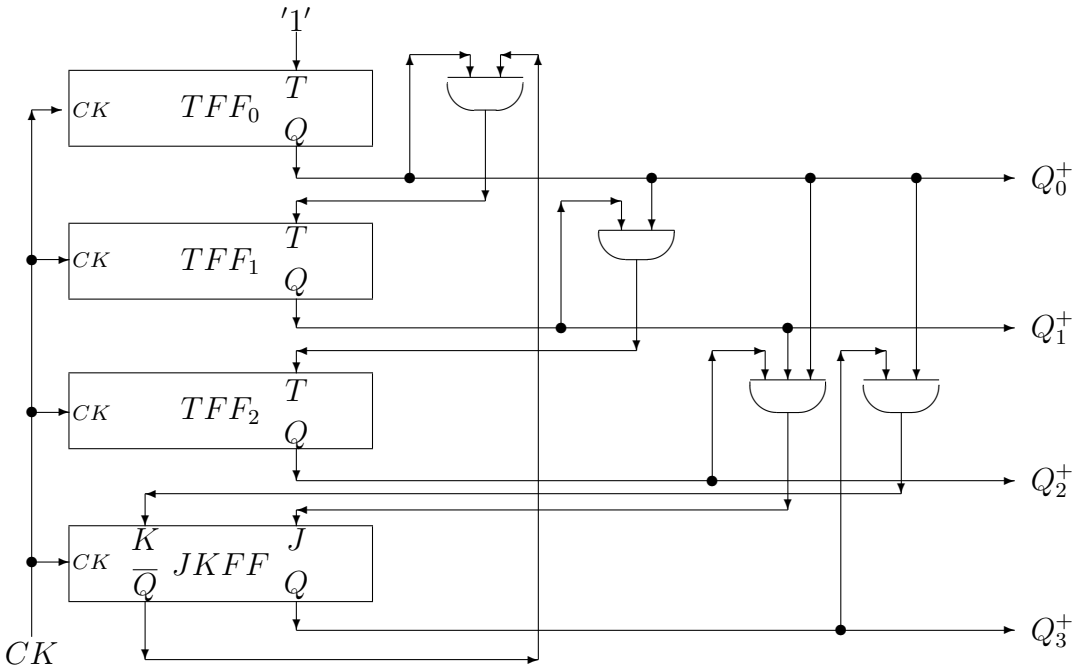


Acest circuit este rapid, dar dimensiunile sale cresc mult odată cu n: la fiecare pas se adaugă un T flip - flop şi un sumator al cărui număr de intrări depinde de n.

Se pot da şi alte soluţii sincrone care să păstreze o creştere rezonabilă a dimensiunilor, dar viteza de lucru va fi mai mică.

Dacă toate intrările în porţile AND sunt din $\overline{Q}$ (în loc de Q) se obţine un numărător binar descrescător.

Exemplul 7.4 *Să construim un numărător zecimal.*



Cum reprezentarea cifrelor $0, 1, \ldots, 9$ *se poate realiza codificat pe patru biţi (de la* 0000 *la* 1001*), aceasta va necesita doar patru circuite de tip* TFF *sau* $JKFF$*.*

Pentru uşurinţă, în structura reprezentată anterior s-au folosit trei circuite TFF *şi un* $JKFF$*.*

Bineînţeles, este posibilă o construcţie în care toate cele patru module să fie $JKFF$*; în acest caz, la primele trei module* $JKFF$ *se va folosi aceeaşi valoare de intrare pentru* J *şi* K*).*

Ecuaţiile counterului zecimal sunt:

$$T_0 = '1', \qquad T_1 = Q_0^+ \wedge (\overline{Q_3})^+, \qquad T_2 = Q_1^+ \wedge Q_0^+, \qquad J = Q_0^+ \wedge Q_1^+ \wedge Q_2^+, \qquad K = Q_0^+ \wedge Q_3^+$$

$$Q_0^+ = \overline{Q_0}, \qquad Q_1^+ = Q_1 \oplus T_0, \qquad Q_2^+ = Q_2 \oplus T_2, \qquad Q_3^+ = (Q_3 \wedge \overline{K}) \vee (\overline{Q_3} \wedge J)$$

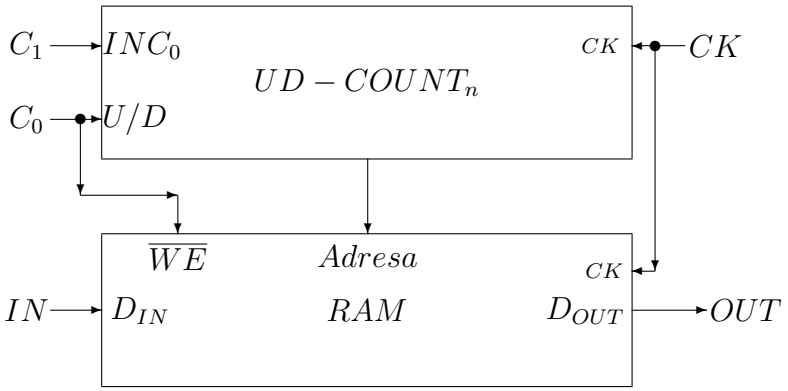
Lăsăm ca exerciţiu verificarea că acesta este un counter pe patru biţi în care, după valoarea 1001 *(corespunzătoare cifrei* 9*) urmează* 0000*.*

7.4 Stive

Înafara memoriilor prezentate până acum (ROM, RAM, regiştri), o memorie frecvent folosită este cea de tip stivă (sau $LIFO$).

Există multe soluţii practice pentru implementarea acestui gen de memorie.

Una din cele mai simple idei foloseşte un numărător *up-down* $UD - COUNT_n$ şi o memorie RAM.



Un $UD - COUNT_n$ este un numărător $COUNT_n$ cu o intrare suplimentară U/D care specifică dacă numărarea se efectuează crescător (*up*): $C_0 = 1$, sau descrescător (*down*) – $C_0 = 0$.

Numărarea descrescătoare se implementează la fel de simplu ca cea crescătoare: singura diferenţă constă la intrarea în porţ ile AND unde vor ajunge ieşirile $\overline{Q}$ ale flip - flopurilor (la crescător se folosea numai ieşirea Q).

Introducerea acestei noi restricţii se poate realiza folosind un $EMUX$ (pentru selectarea lui Q sau Q') sau un XOR (pentru complementarea lui Q la comanda semnalului U/D).

CK asigură sincronizarea celor două circuite.

Structura din figură poate simula trei operaţii (funcţii):

1. **no op**: $C_0 = 0,\ C_1 = 0$ – nici o operaţie;

2. **push**: $C_0 = 1,\ C_1 = 1$ – $UD - COUNT_n$ este incrementat şi data aflată la intrare (IN) este stocată ($WE = 0$) în RAM la adresa indicată de noul conţinut al numărătorului.

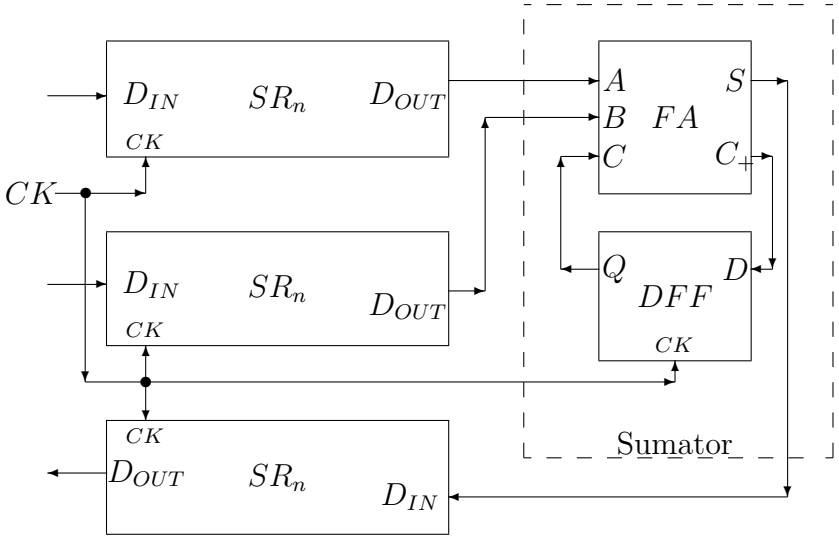
3. **pop**: $C_0 = 0,\ C_1 = 1$ – configuraţia binară stocată la adresa indicată de conţinutul lui $UD - COUNT_n$ este citită la ieşirea OUT şi $UD - COUNT_n$ este decrementat la noul punct de *vârf al stivei*.

7.5 Circuite aritmetice

7.5.1 Sumator serial

Pentru sumatorii de n biţi se poate construi în acest moment o altă versiune, care uneori poate fi mai avantajoasă din punct de vedere al dimensiunii.

Astfel, folosind un automat, putem stoca cei doi operanzi în regiştri seriali (şi nu în regiştri paraleli, cum s-au folosit în construcţia din Capitolul 6).



Acest sumator serial conţine 3 regiştri (doi pentru operanzi şi unul pentru rezultat) şi un *automat de însumare*.

Acesta este un automat cu două stări, bazat pe un DFF legat printr-un ciclu cu un sumator FA.

Starea curentă reţine transportul sumei ciclului precedent. Intrările A şi B primesc sincronizat – la acelaşi tact – biţii de pe aceeaşi poziţie din cei doi regiştri seriali care conţin operanzii.

Automatul este iniţial în starea 0, ceea ce semnifică $C_+ = 0$.

Ieşirea S este stocată bit cu bit – timp de n tacţi – de al treilea registru.

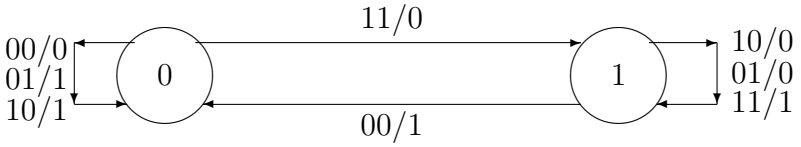
Rezultatul final este dat de al treilea registru serial şi de starea rămasă în DFF.

Formal, $Q = \{0, 1\},\ X = \{0, 1\}^2,\ Y = \{0, 1\}$ şi

δ	00	01	10	11
0	0	0	0	1
1	0	1	1	1

λ	00	01	10	11
0	0	1	1	0
1	1	0	0	1

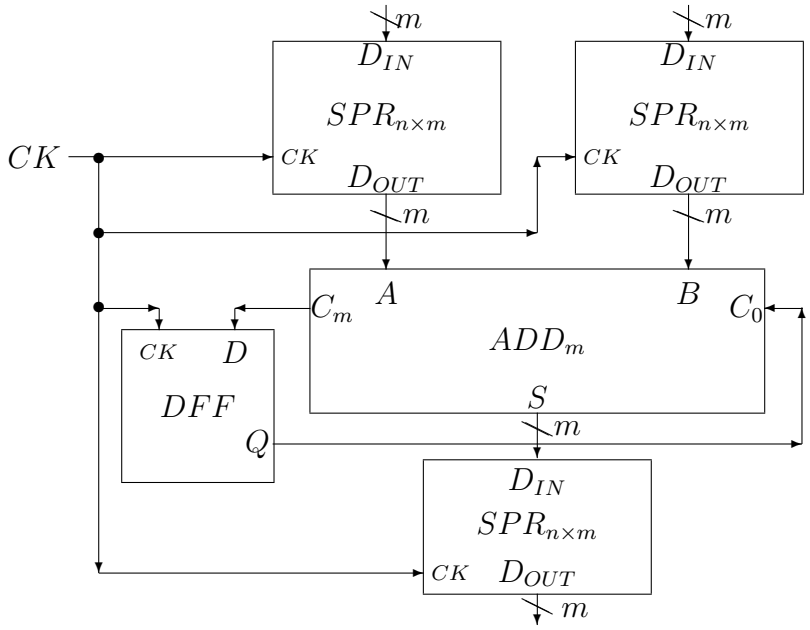
Graful de funcţionare al automatului concentrează toată această informaţie în:



7.5.2 Circuit aritmetic serial - paralel

Este posibil ca timpul de execuţie al circuitului anterior să fie considerat prea mare (datorită extensiei seriale); atunci se poate face o combinaţie între extensia serială şi cea paralelă.

Un astfel de circuit este de exemplu:



Aici se folosesc trei regiştri serial - paraleli $SPR_{n\times m}$, un sumator de m biţi ADD_m şi un registru cu două stări (de exemplu un DFF).

Intrările şi ieşirile circuitului sunt secvenţe de m biţi.

Circuitul realizează suma a două numere de $n \cdot m$ biţi în n tacţi.

7.5.3 Sumatoare - prefix

Automatele construite până acum au fost foarte simple, toate având numai două stări.

Să mărim puterea acestor automate, conservând simplitatea funcţiei de tranziţie, dar mărind numărul stărilor.

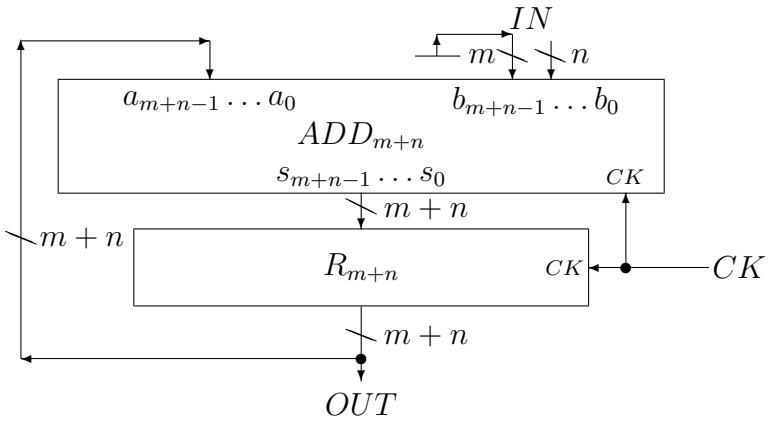
Multe aplicaţii solicită circuite cu funcţie şi structură de acumulator, adică sisteme capabile să adune mai multe numere şi să returneze în final suma lor – sau toate sumele lor parţiale (*prefixe*).

Astfel, să considerăm p numere $x_1, \ldots, x_p$. Sumele prefix sunt

$$
\begin{aligned}
&y_1 = x_1 \\
&y_2 = x_1 + x_2 = y_1 + x_2 \\
&y_3 = x_1 + x_2 + x_3 = y_2 + x_3 \\
&\ldots \\
&y_p = x_1 + x_2 + \ldots + x_p = y_{p-1} + x_p
\end{aligned}
$$

(în mod similar se poate defini orice $\circ - prefix$, înlocuind adunarea "+" cu un operator arbitrar $\circ$).

Vom prezenta aici un exemplu de automat aritmetic care să genereze la fiecare tact câte un prefix (începând cu y_1).



Valoarea iniţială din registrul paralel R_{m+n} este 0.

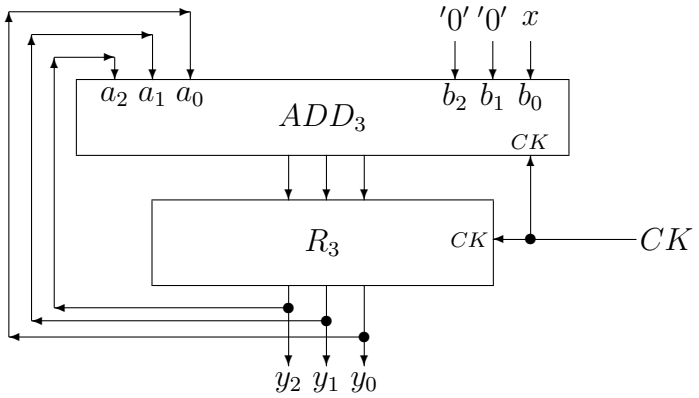
Automatul are 2^{m+n} stări şi calculează sumele parţiale pentru $p \leq 2^{m+1} - 1$ numere de câte n biţi.

La fiecare tact se introduce un număr format din n biţi, iar pe cele m poziţii suplimentare din stânga vor intra permanent $'0'$.

Aceşti m biţi suplimentari sunt necesari pentru a obţine un rezultat corect şi în situaţiile cele mai nefavorabile, când sumele depăsesc n biţi.

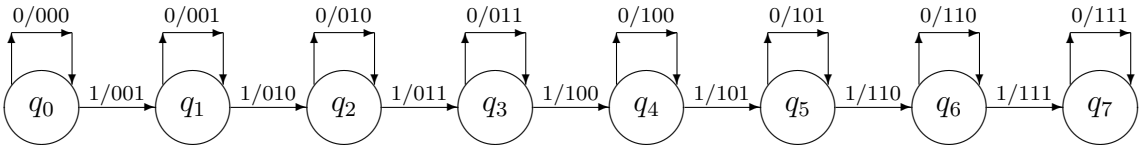
Automatul este evident foarte simplu. Cele două componente sunt circuite simple care pot fi definite recursiv.

Exemplul 7.5 *Să particularizăm sumatorul prefix definit anterior, pentru* $m = 2,\ n = 1$. *El va avea forma*



Acest sumator lucrează pe trei biţi, adunând permanent o valoare binară ('0' sau '1'); el poate fi folosit şi ca un counter pe trei biţi – $COUNT_3$.

Definit ca un automat, vom avea $Q = \{0,1\}^3,\ X = \{0,1\},\ Y = \{0,1\}^3$.
Funcţiile de tranziţie şi de ieşire rezultă imediat din graful de funcţionare:



7.5.4 Multiplicator

Să construim un circuit secvenţial pentru înmulţirea a două numere binare.

Spre deosebire de circuitul combinaţional de înmulţire, acesta foloseşte circuitul de deplasare, bazându-se pe faptul că înmulţirea este de fapt o adunare repetată.

Pentru uşurinţa expunerii, vom dezvolta o construcţie pentru înmulţirea a două numere întregi cu semn reprezentate pe 8 biţi (generalizarea fiind imediată).
Fie numerele binare $X = x_0 \dots x_7$ şi $Y = y_0 \dots y_7$, din care formăm produsul $P = X \times Y$.

Numerele au reprezentarea standard: biţii marcaţi cu 0 sunt biţi de semn, iar restul (notat X_M respectiv Y_M) semnifică mărimea numărului (considerat subunitar).

Valoarea lui este dată de numărul

$$N = \sum_{i=1}^{7} x_i 2^{-i}$$

Când $x_0 = 1$, valoarea lui X este $-N$.

Algoritmul de înmulţire va implementa întâi $P_M := X_M \times Y_M$, unde $P_M = p_1 \dots p_{14}$ este valoarea absolută a produsului.

Semnul lui P este dat de $p_0 := x_0 \oplus y_0$.

Rezultatul final este numărul de 15 biţi $P = p_0 p_1 \dots p_{14}$.

Valoarea P_M este calculată iterativ prin şapte adunări/deplasări, definite de relaţiile

$$P_i := P_i + x_{7-i} Y_M; \quad P_{i+1} := 2^{-1} P_i \quad (1 \le i \le 6);$$

unde $P_0 = 0,\ P_7 = P_M$.

Când $x_{7-i} = 1$, avem $P_i := P_i + Y_M$, iar pentru $x_{7-i} = 0$ va fi $P_i := P_i + 0$.

Deci, la fiecare pas, la produsul parţial P_i se adună înmulţitorul Y_M sau 0.

Înmulţirea cu 2^{-1} este o deplasare a lui P_i spre dreapta cu o poziţie (după adunare); aceasta este echivalentă cu o împărţire la 2.

Fiecare astfel de pas (adunare + deplasare) adaugă un bit la produsul parţial, care creşte astfel de la 7 la 14 biţi (la care se va adăuga în final şi bitul de semn).

Putem specifica acum principalele componente necesare în construcţia multiplicatorului pe 8 biţi.

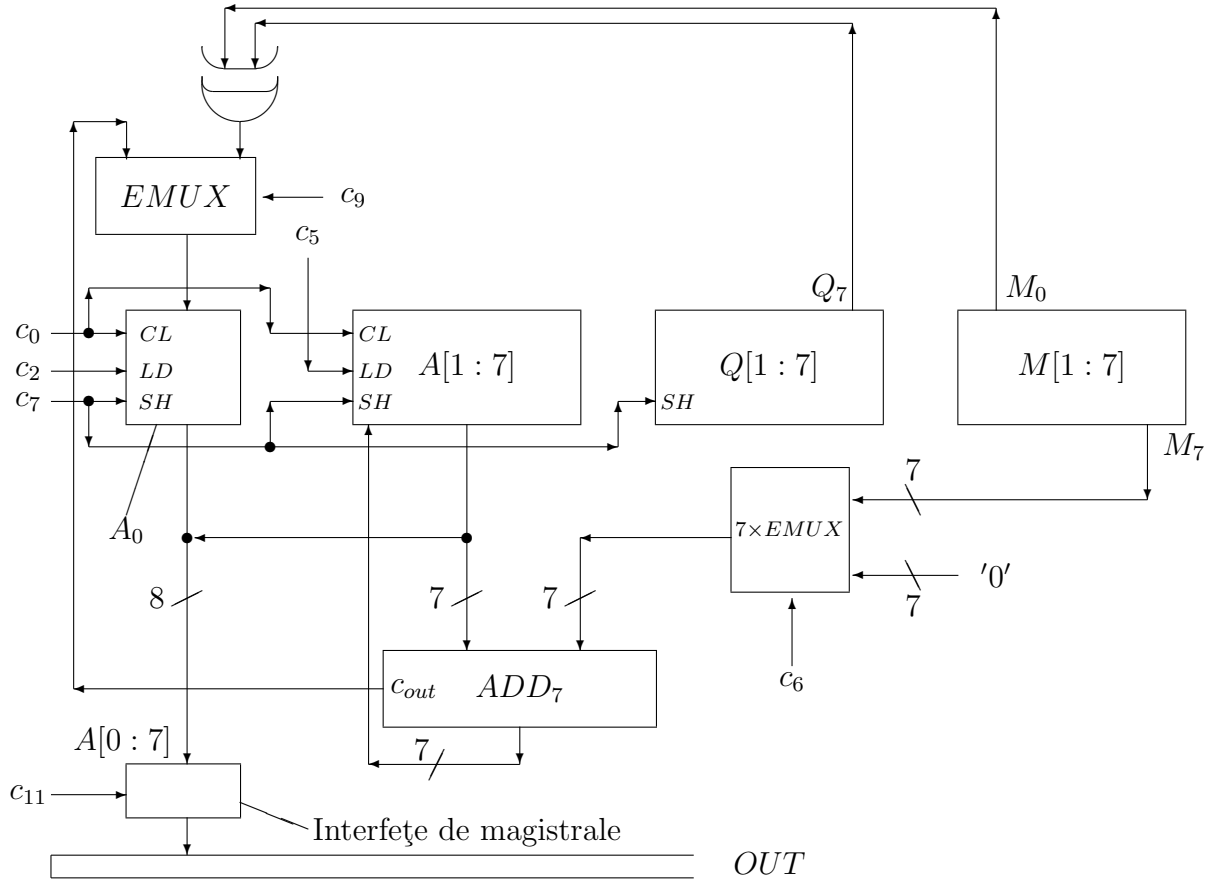
- Doi regiştri: Q (pentru înmulţitor) şi M (deînmulţit) vor conţine X respectiv Y.
- Un registru dublu (de 16 biţi) $A.Q$ va stoca produsele parţiale P_i. De remarcat că acesta conţine în a doua jumătate registrul înmulţitor.

- Adunarea este realizată cu un sumator combinaţional (paralel) pe 7 biţi (se poate folosi un sumator serial, dar va fi de şapte ori mai lent).

  Sumatorul va avea ieşirea şi o intrare conectate la A, iar cealaltă intrare va trebui să comute între M şi 0.

- Funcţia de deplasare la dreapta cu un bit se poate obţine folosind pentru A un registru de deplasare dreapta cu intrare/ieşire paralelă.

Circuitul are structura următoare:



Conform definiţiei, suma este controlată de bitul x_{7-i}, stocat în registrul Q.

Unitatea de control a înmulţirii va trebui să scaneze la fiecare pas conţinuturile lui Q de la dreapta spre stânga.

Dacă Q este un registru de deplasare, atunci x_{7-i} se va obţine totdeauna din cel mai la dreapta flip-flop al lui Q ($Q[7]$) deplasând Q spre dreapta înainte de a extrage următorul x_{7-i}.

Deci, X_M este redus treptat de la 7 biţi la 1 bit, în timp ce P_i este expandat de la 7 la 14 biţi, tot prin deplasare la dreapta.

Aceasta conduce la ideea de a combina A şi Q într-un singur registru dublu de deplasare, în care jumătatea dreaptă este Q, iar cea stângă – A.

Multiplicatorul este completat prin includerea magistralelor de date externe IN şi OUT şi o unitate de control cu un numărător pe 3 biţi $COUNT_3$.

După cum se observă, în centrul construcţiei se află sumatorul paralel pe 7 biţi şi registrul dublu $A.Q$ care implementează formulele de calcul.

Semnalul c_{out} de ieşire al transportului din sumator este al 8-lea bit (cel mai semnificativ) al sumei şi este conectat la intrarea A_0.

Contorul $COUNT_3$ (nereprezentat aici) este incrementat şi testat după fiecare pas adunare/deplasare, pentru a vedea dacă prima etapă de calcul s-a terminat.

Când $COUNT_3$ a ajuns la $'111'$ (adică 7), P_M ocupă biţii $1:14$ ai registrului $A.Q$, adică biţii $A[1:7].Q[0:6]$.

Bitul de semn p_0 este apoi calculat din x_0 (stocat în Q_7) şi y_0 (stocat în M_0), după care este depus în A_0.

Simultan, un $'0'$ este pus în Q_7 pentru a extinde rezultatul final de la 15 la 16 biţi.

Unitatea de control a multiplicatorului este formată din toate semnalele şi punctele de control necesare în implementarea operaţiilor specificate.

Listăm mai jos câteva semnale de control necesare în construirea unui multiplicator.

Semnal control	Operaţia controlată	Semnal control	Operaţia controlată
c_0	Resetează A	c_7	Deplasare dreapta $A.Q$
c_1	Resetează $COUNT_3$	c_8	Incrementare $COUNT$
c_2	Încarcă A_0	c_9	Selecţie c_{out} sau $M_0 \oplus Q_7$ pentru A_0
c_3	Încarcă M din IN	c_{10}	Anulează Q_7
c_4	Încarcă Q din IN	c_{11}	$OUT \longleftarrow A$
c_5	Încarcă suma în $A[1:7]$	c_{12}	$OUT \longleftarrow Q$
c_6	Selecţie M sau $'0000000'$		

În unele cazuri sunt necesare mai multe semnale de control pentru implementarea unei operaţii.

De exemplu, adunarea foloseşte c_6 pentru a selecta operandul drept al adunării, c_9 pentru a selecta c_{out} care se va încărca în A_0, c_2 şi c_5 pentru pentru a încărca suma curentă de 8 biţi în $A[0:7]$.

Numărul semnalelor de control variază în funcţie de logica utilizată pentru implementarea unităţii de control.

Tabelul următor descrie un proces complet de înmulţire între numerele $X = 10110011$ şi $Y = 01010101$.

Bitul de semn $x_0 = 1$ al lui X (care arată că este negativ) este subliniat.

Datele din $A.Q$ aflate la stânga lui x_0 reprezintă produsul parţial curent P_i.

Pas	Acţiune	Acumulator A	Registru Q
0	Iniţializare regişţri	00000000	$\underline{1}$0110011
1		01010101	
	Add M **to** A	01010101	$\underline{1}$0110011
	Shift $A.Q$	00101010	1$\underline{1}$011001
2		01010101	
	Add M **to** A	01111111	1$\underline{1}$011001
	Shift $A.Q$	00111111	11$\underline{1}$01100
3		00000000	
	Add 0 **to** A	00111111	11$\underline{1}$01100
	Shift $A.Q$	00011111	111$\underline{1}$0110
4		00000000	
	Add 0 **to** A	00011111	111$\underline{1}$0110
	Shift $A.Q$	00001111	1111$\underline{1}$011
5		01010101	
	Add M **to** A	01100100	1111$\underline{1}$011
	Shift $A.Q$	00110010	01111$\underline{1}$01
6		01010101	
	Add M **to** A	10000111	01111$\underline{1}$01
	Shift $A.Q$	01000011	10111$\underline{1}$0
7		00000000	
	Add 0 **to** A	01000011	10111$\underline{1}$0
	Shift $A.Q$	00100001	11011$\underline{1}$1
8	Pune semnul în A_0	10100001	11011110 - Produsul
	$Q_7 \longleftarrow 0$		final

7.5.5 Circuit pentru produsul scalar (MAC)

Una din cele mai importante funcţii aritmetice care pot fi realizate în această etapă este produsul scalar $a_1b_1 + a_2b_2 + \ldots + a_nb_n$ dintre doi vectori $A = (a_1, \ldots, a_n)$ şi $B = (b_1, \ldots, b_n)$.

Circuitul – numit MAC ("*Multiple Accumulate Circuit*") – poate fi realizat în mai multe moduri posibile.

Construcţia prezentată aici foloseşte o extensie serială a două automate, numite automat "*acumulator*" şi automat *"bits eater" (mâncător de biţi).*

Folosim şi aici faptul că multiplicarea este în esenţă tot o adunare.

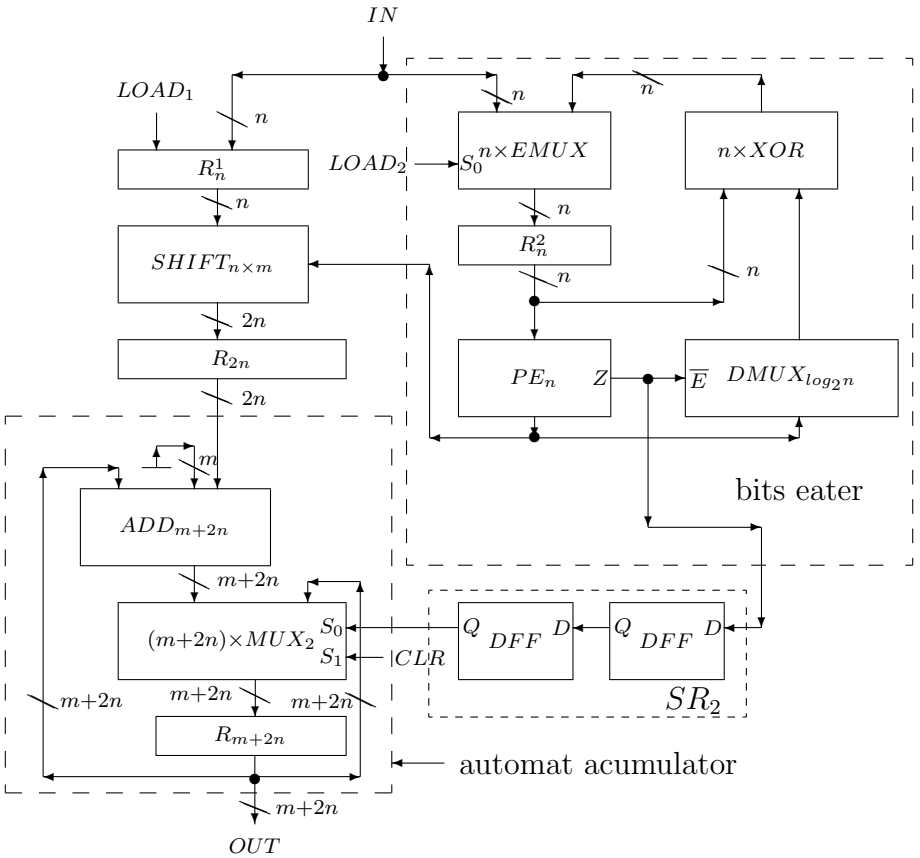
Deci se va construi un automat acumulator pentru a implementa în paralel ambele operaţii: înmulţirea şi suma produselor; în acest fel timpul de execuţie rămâne acelaşi.

Un astfel de circuit MAC este compus din:

- **Automat "bits eater"**, care indică succesiv poziţiile biţilor din operandul b, care au valoarea 1; de asemenea el precizează sfârşitul operaţiei de înmulţire; circuitul conţine:
  - R_n^2 - registru de stare, conţinând operandul b_i;
  - $n \times EMUX$ - n multiplexori elementari care lucrează în paralel; ei fac iniţializarea conţinutului registrului de stare (pentru $LOAD_2 = 1$) şi închide ciclul automatului (pentru $LOAD_2 = 0$).
  - PE_n - codificator cu prioritate care indică poziţia celui mai semnificativ bit $'1'$ din registrul de stare, precum şi sfârşitul înmulţirii, prin ieşirea Z;
  - $DMUX_{log_2 n}$ - ieşirea activă a automatului; $Z = 0$ corespunde celui mai semnificativ bit $'1'$ din registrul de stare;
  - $n \times XOR$ - porţi folosite pentru complementarea – la comanda demultiplexorului – a celui mai semnificativ $'1'$ din registrul de stare.
- **Circuit de deplasare combinaţional**: rolul lui este de a deplasa primul operand a_i, cu un număr de poziţii indicat de automatul anterior; el este format din
  - R_n^1 - registrul care conţine operandul a_i, încărcat la comanda $LOAD_1$;
  - $SHIFT_{n \times m}$ - circuit combinaţional de deplasare spre stânga cu maxim n poziţii (înmulţire cu 2^i, $i = 0, 1, \ldots, n$) pentru cuvinte de n biţi.
  - R_{2n} - registru de regularizare a trecerii (*pipeline register*).

- **Automat acumulator**: care realizează sumele parţiale cu produsele obţinute la fiecare pas; structura lui este:
  - ADD_{m+2n} - adună la conţinutul registrului de ieşire valoarea primită de la circuitul de deplasare prin registrul de regularizare R_{2n};
  - $(m+2n)\times MUX_2$ - ansamblu de $m+2n$ multiplexori, folosit pentru iniţializarea la zero a registrului de ieşire ($S_1 = 1$), pentru închiderea ciclului prin sumator ($S_1S_0 = 00$) şi pentru păstrarea valorii finale când calculul se încheie $S_1S_0 = 01$);
  - R_{m+2n} - este registrul de stare al automatului şi totodată registrul de ieşire al sistemului; poate memora $m + 2n$ biţi, permiţând 2^m adunări posibile de numere rezultate din înmulţirea a două numere de n biţi fiecare.
- SR_2: este o linie de întârziere pentru sincronizarea semnalului Z.

Grafic, structura se reprezintă astfel pentru simplificare, nu s-au mai trecut şi circuitele de sincronizare prin ceas):



Circuitul funcţionează în felul următor:

1. Se introduce operandul a în R_n^1 (la comanda din $LOAD_1$);
2. Se introduce operandul b în R_n^2 (la comanda din $LOAD_2$);
3. PE_n selectează un vector $00\ldots10\ldots0$ cu $'1'$ pe cea mai semnificativă poziţie a lui b. Acesta:
   (a) Comandă o deplasare ($SHIFT$) a lui a cu un număr de poziţii egal cu valoarea dată;
   (b) Face (pe circuitul $XOR - EMUX$ $'0'$ pe această poziţie a lui b;
   (c) Când toţi termenii sunt $'0'$, anunţă încheierea înmulţirii şi începe adunarea în automatul acumulator.

Operaţia este realizată într-un număr de tacţi egal cu numărul de valori $'1'$ din al doilea operand. Deci timpul mediu de execuţie este egal cu $n/2$.

Observaţia 7.3 *Dacă cei doi vectori se reduc la câte un singur număr ($n = 1$), circuitul MAC construit lucrează ca un multiplicator, calculând produsul celor două numere.*

7.6 Reprezentarea automatelor finite

Comportarea unui automat finit poate fi definită în mai multe moduri (grafuri, tabele de tranziţie, scheme logice, diagrame etc).

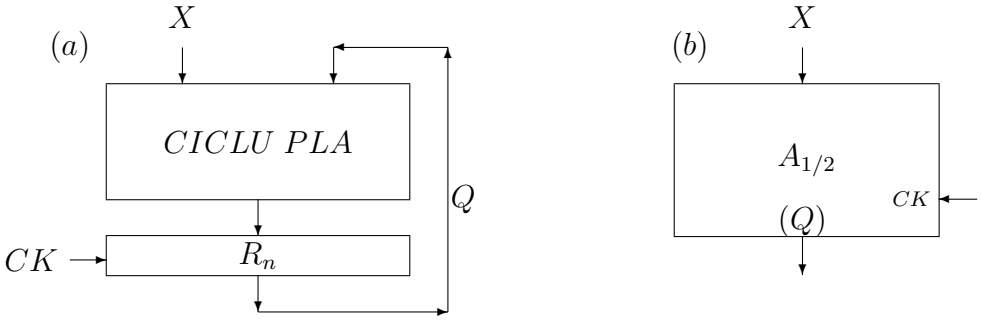
Toate acestea necesită însă o definire nerecursivă a funcţiilor δ şi λ.

În această secţiune vom construi o reprezentare a automatelor finite bazată pe tablouri logic programabile (PLA).

Astfel va fi posibil să reducem complexitatea funcţiilor de transfer şi de ieşire ale automatelor, efectuând simplificări ale tablourilor PLA corespunzătoare.

Definiţia 7.4 *Un semi-automat este un triplet $A_{1/2} = (Q, X, \delta)$ unde Q, X, δ au semnificaţia dată la definiţia automatelor.*

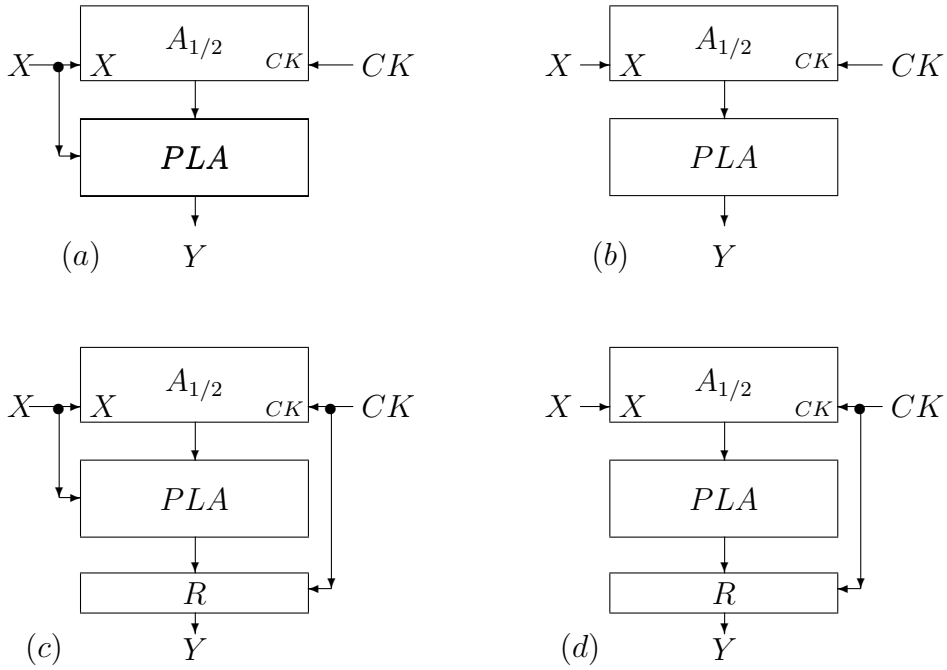
Un semi-automat este deci un automat fără funcţia de ieşire λ.



Utilitatea acestui concept (mai apropiat de noţiunea formală de *automat finit*) rezidă din faptul că multe tehnici de optimizare sunt legate doar de cicluri, nu de ieşiri.

Cele patru tipuri de automate pot fi definite atunci (prin semi-automate) astfel:

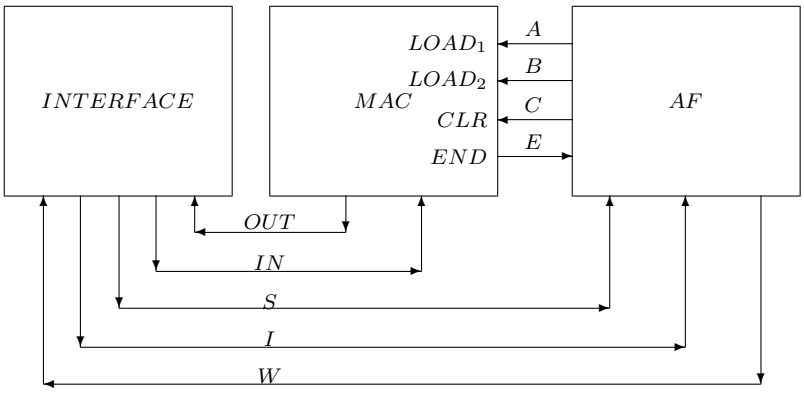
- **automatul Mealy**: rezultă prin conectarea unui semi-automat cu un PLA care determină funcţia λ din intrarea X şi "ieşirea" semi-automatului (Figura (a) de pe pagina următoare);
- **automatul Moore**: se obţine conectând la "ieşirea" semi-automatului un PLA (Figura (b));
- **automate cu întârzieri**): se obţin conectând încă un registru R la construcţiile anterioare ((c) pentru Mealy, (d) pentru Moore).



Vom ilustra prin două exemple utilizarea semi-automatelor PLA de transfer şi de ieşire.

7.6.1 Automat pentru MAC

Să construim un automat pentru funcţionarea circuitului multiplicator MAC (multiple-accumulate circuit) definit în paragraful anterior[3].



[3]Evident, în loc de MAC se poate folosi un automat pentru realizarea oricărui grup de operaţii aritmetice.

Aici MAC este conectat la o interfaţă sincronă ($INTERFACE$), întregul sistem fiind controlat de un automat finit FA.

Acest automat are trei intrări:

1. S: este generat de $INTERFACE$ indicând că circuitul MAC a fost selectat pentru transferul de date;
2. I: este generat de $INTERFACE$, indicând sensul de transfer al datelor: dacă $I = 1$ atunci transferul se face de la $INTERFACE$ spre MAC; altfel, în sens contrar;
3. E: este generat de MAC semnalizând sfârşitul operaţiei de înmulţire.

şi patru ieşiri:

1. $LOAD_1$: comandă încărcarea primului operand primit sincron la intrarea IN de la $INTERFACE$;
2. $LOAD_2$: comandă încărcarea în mod similar a celui de-al doilea operand, şi porneşte operaţia;
3. CLR: comandă încărcarea valorii $'0'$ în registrul de ieşire din MAC (resetare);
4. $WAIT$: MAC este în curs de a realiza operaţia, dar registrul de ieşire nu conţine încă rezultatul final.

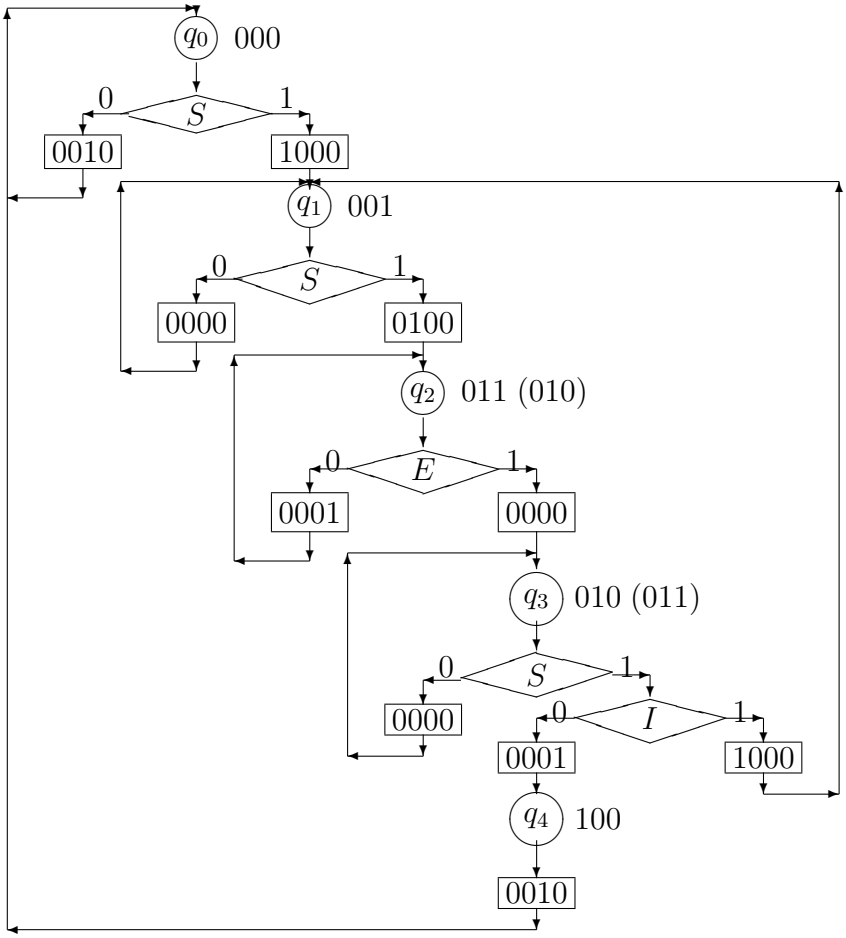
Ieşirea din circuit este organizată într-o secvenţă de 4 biţi, în ordinea $LOAD_1, LOAD_2$, $CLR, WAIT$.

În schema logică de pe pagina următoare este descrisă comportarea automatului ilustrată de funcţiile δ şi λ.

Automatul FA este de tip Mealy şi are 5 stări cu următoarele semnificaţii:

- q_0 – starea iniţială, în care automatul aşteaptă să fie pornit, iar – pentru $S = 1$ – încarcă primul operand;
- q_1 – automatul a primit primul operand (într-un tact în care semnalul S a fost activat; vezi dreptunghiul precedent, în care $LOAD_1 = 1$) şi aşteaptă al doilea operand, care se încarcă în primul tact în care S devine activ;
- q_2 – este starea în care se aşteaptă sfârşitul operaţiei de înmulţire (se generează spre interfaţă semnalul de aşteptare $WAIT = 1$);
- q_3 – se aşteaptă o nouă selecţie; dacă este selectat MAC, testează sensul de transfer solicitat; dacă trebuie încărcată în MAC o nouă pereche de operanzi, atunci îl încarcă pe primul şi sare la starea q_1; altfel trece în q_4;

- q_4 – procesul de înmulţire este terminat, rezultatul este transferat la interfaţă şi registrul de ieşire este curăţat.



Deci $Q = \{0,1\}^3$ (sunt necesari minim trei biţi pentru a codifica cinci stări), $X = \{0,1\}^3$ (o intrare este reprezentată de secvenţa EIS), $Y = \{0,1\}^4$ (secvenţă de patru biţi care codifică în ordine ieşirile $LOAD_1, LOAD_2, CLR, WAIT$).

O stare este reprezentată printr-o secvenţă de trei biţi $Q_2Q_1Q_0$; am codificat

$$q_0 = 000,\ q_1 = 001,\ q_2 = 011,\ q_3 = 010,\ q_4 = 100$$

(este posibilă şi o inversiune de notaţie între q_2 şi q_3).

Funcţia de transfer $\delta(Q_2Q_1Q_0, EIS) = Q_2^+Q_1^+Q_0^+$ este definită

δ	000	001	010	011	100	101	110	111	Q_2^+	Q_1^+	Q_0^+
000	000	001	000	001	000	001	000	001	0	0	S
001	001	011	001	011	001	011	001	011	0	S	1
011	011	011	011	011	010	010	010	010	0	1	$\overline{E}$
010	010	100	010	001	010	100	010	001	$S\overline{I}$	$\overline{S}$	SI
100	000	000	000	000	000	000	000	000	0	0	0

unde pe ultimele trei coloane s-a descris formal starea rezultată (in funcţie de intrare).

Putem defini acum ecuaţiile care descriu semi-automatul PLA de transfer:

$Q_2^+ = Q_1\overline{Q}_0S\overline{I}$ (Q_2 se poate reduce prin minimizarea funcţiei booleene)
$Q_1^+ = \overline{Q}_1Q_0S \vee Q_1Q_0 \vee Q_1\overline{S}$
$Q_0^+ = \overline{Q}_2\overline{Q}_1S \vee \overline{Q}_1Q_0 \vee Q_0\overline{E} \vee Q_1\overline{Q}_0SI$

Deci

$$\delta(Q_2Q_1Q_0, EIS) = (Q_1\overline{Q}_0S\overline{I},\ \overline{Q}_1Q_0S \vee Q_1Q_0 \vee Q_1\overline{S},\ \overline{Q}_2\overline{Q}_1S \vee \overline{Q}_1Q_0 \vee Q_0\overline{E} \vee Q_1\overline{Q}_0SI)$$

Similar, funcţia de ieşire $\lambda(Q_2Q_1Q_0, EIS) = (LOAD_1, LOAD_2, CLR, WAIT)$ este definită prin tabelul

λ	000	001	010	011	100	101	110	111				
000	0010	1000	0010	1000	0010	1000	0010	1000	S	0	$\overline{S}$	0
001	000	0100	0000	0100	0000	0100	000	0100	0	S	0	0
011	0001	0001	0001	0001	0001	0001	0001	0001	0	0	0	1
010	0000	0001	0000	1000	0000	0001	0000	1000	SI	0	0	$S\overline{I}$
100	0010	0010	0010	0010	0010	0010	0010	0010	0	0	1	0

Pe baza lui se pot scrie ecuaţiile care descriu PLA-ul de ieşire:

$$LOAD_1 = \overline{Q}_2\overline{Q}_1\overline{Q}_0S \vee Q_1\overline{Q}_0SI, \qquad LOAD_2 = \overline{Q}_1Q_0S$$
$$CLR = Q_2 \vee \overline{Q}_1\overline{Q}_0\overline{S}, \qquad WAIT = Q_1Q_0 \vee Q_1S\overline{I}$$

Deci

$$\lambda(Q_2Q_1Q_0, EIS) = (\overline{Q}_2\overline{Q}_1\overline{Q}_0S \vee Q_1\overline{Q}_0SI,\ \overline{Q}_1Q_0S,\ Q_2 \vee \overline{Q}_1\overline{Q}_0\overline{S},\ Q_1Q_0 \vee Q_1S\overline{I})$$

Graful funcţional (unde $x, y, z \in \{0, 1\}$) este

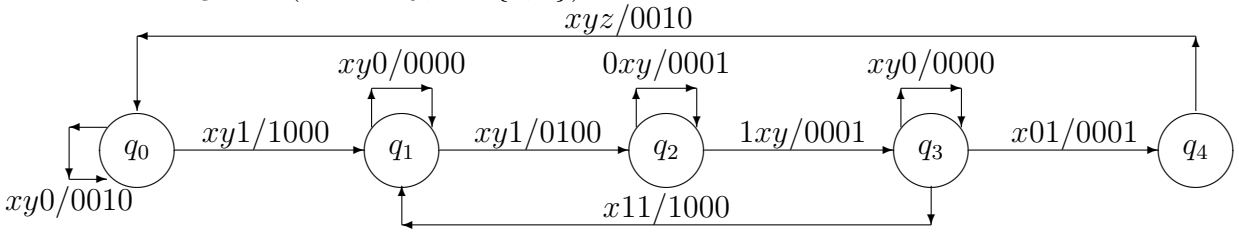
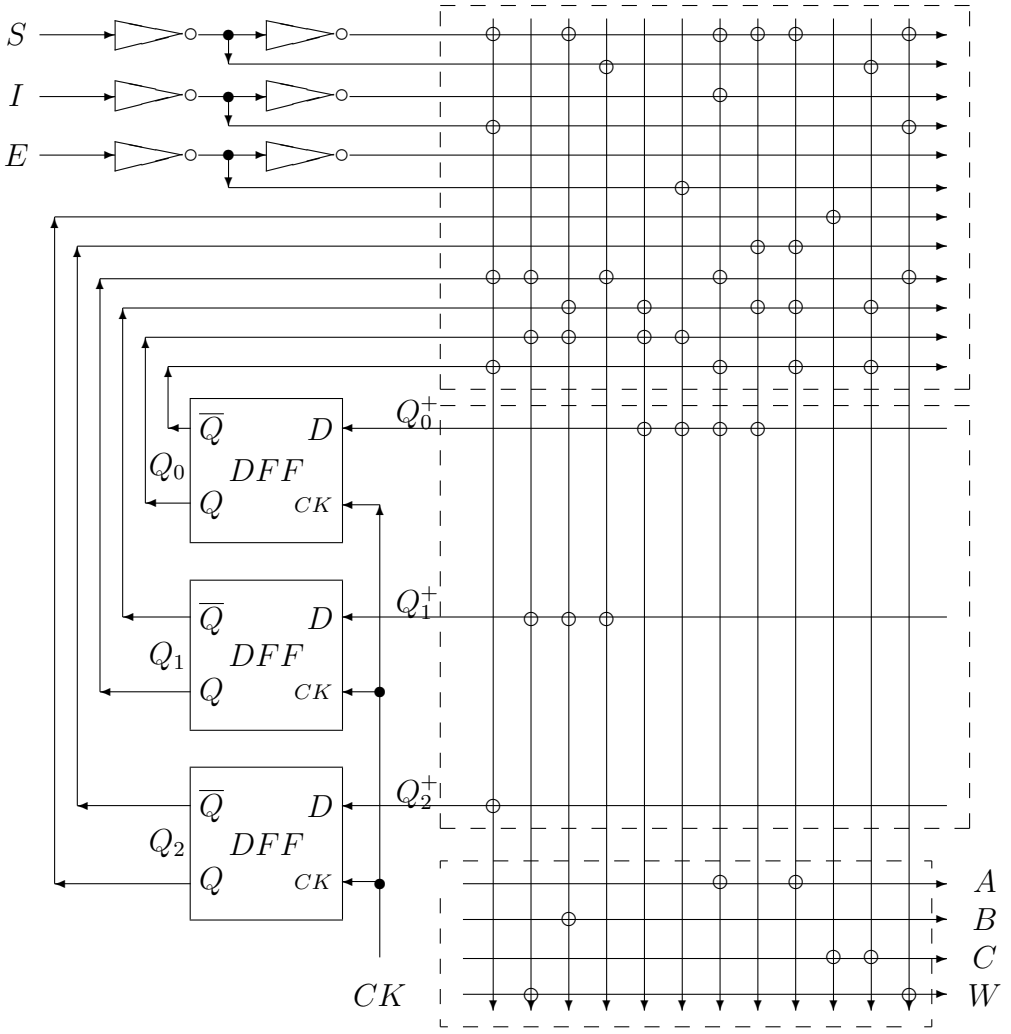


Figura de mai jos descrie circuitul rezultat; cele două PLA-uri de conjuncţie (corespunzătoare funcţiilor de transfer şi de ieşire) sunt unite în unul singur (cel superior), deci termenii comuni sunt utilizaţi împreună.

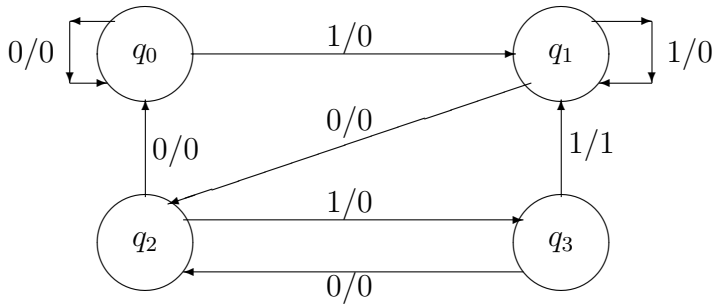


7.6.2 Automat pentru recunoaşterea unei secvenţe binare

O problemă des întâlnită este aceea de a recunoaşte apariţia unei anumite subsecvenţe în cadrul unei secvenţe binare.

Să construim de exemplu un automat care să semnaleze apariţia subsecvenţei 1011.

Dintr-un studiu bazat pe limbaje formale, un graf al acestui automat se poate construi uşor:



De exemplu, pentru intrarea 010101101110011 ieşirea va fi 000000100100000.

Automatul având patru stări, acestea pot fi codificate folosind doi biţi:

$$q_0 \equiv 00,\ q_1 \equiv 01,\ q_2 \equiv 10,\ q_3 \equiv 11$$

Deci $Q = \{0,1\}^2,\ X = Y = \{0,1\}$. Funcţiile de transfer şi de ieşire sunt

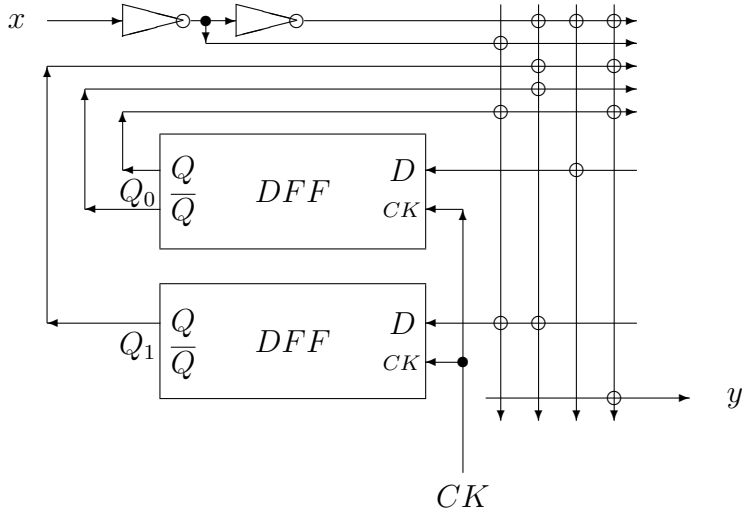
δ	0	1	Q_1^+	Q_0^+
00	00	01	0	x
01	10	01	$\overline{x}$	x
10	00	11	x	x
11	10	01	$\overline{x}$	x

λ	0	1	y
00	0	0	0
01	0	0	0
10	0	0	0
11	0	1	x

De aici rezultă imediat forma celor două PLA (de transfer şi de ieşire):

$$\delta(Q_1Q_0, x) = (Q_0\overline{x} \vee Q_1\overline{Q}_0x,\ x), \qquad \lambda(Q_1Q_0, x) = Q_1Q_0x$$

Pentru implementare sunt necesare două DFF (necesare codificării celor patru stări), un codificator elementar (pentru variabila de intrare x) şi două PLA (care au componenta AND comună).



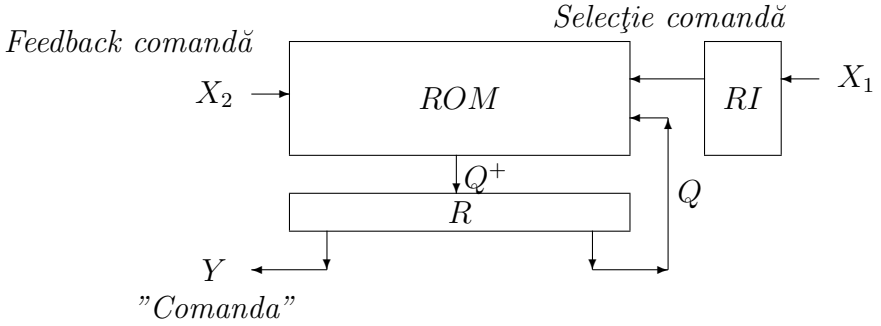
7.7 Automate de control

Din exemplele precedente se întrevede apariţia unei clase importante de automate utilizate în arhitectura unui calculator: clasa *automatelor de control*.

Un astfel de automat este inclus într-un sistem de calcul în două poziţii principale:

- În prima poziţie (X_1) automatul de control primeşte informaţie despre comenzile sau testele pe care trebuie să le facă, printr-un cod care selectează secvenţa de control ce va fi executată;
- În a doua poziţie (X_2), automatul de control primeşte informaţii prin mai mulţi biţi, despre efectele (feedbackuri) pe care le-a avut ieşirea sa Y asupra subsistemelor gestionate.

Mărimea şi complexitatea secvenţei de control necesită înlocuirea PLA cu un ROM, cel puţin în procesul de reprezentare şi testare.



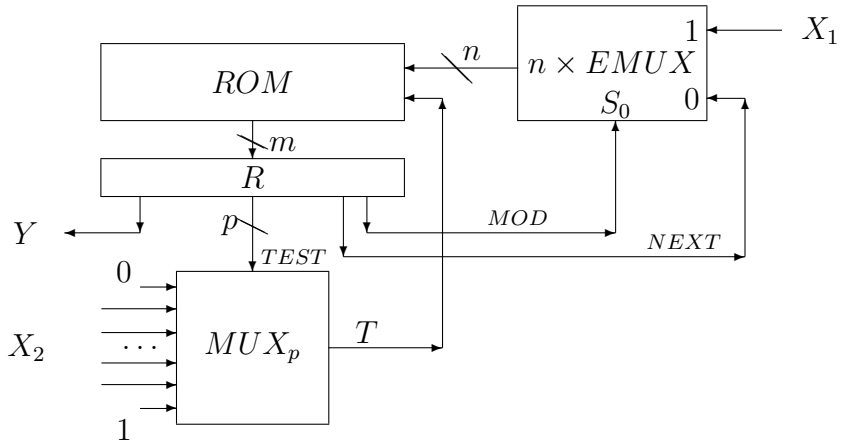
Structura automatului de control foloseşte doi regiştri:

- Un registru de intrare (RI) în care se face o selecţie a comenzii (sau testului).
- Un registru de lucru, unde se formează comanda ce va fi trimisă către un anumit subsistem.

Aplicaţiile accentuează două lucruri importante:

1. Automatul poate stoca în spaţiul stărilor informaţii despre definirea unei acţiuni (fiecare stare curentă se reprezintă printr-o linie care codifică o aplicaţie de la spaţiul iniţial al stărilor, la o singură stare);
2. În majoritatea stărilor, automatul testează numai un bit aflat pe poziţia de adresă X_2; dacă nu sunt acoperite toate situaţiile, problema este rezolvată prin adăugarea unor stări suplimentare, activate automat (stări de eroare, stări finale etc).

Pe baza acestor observaţii, structura automatului de control poate fi modificată, obţinându-se o formă redusă prezentată în figura următoare.



Deoarece secvenţa de comenzi poate fi iniţializată numai prin intermediul codului X_1, acest cod se poate folosi doar pentru adresarea la prima linie de comandă din ROM, cu ajutorul unei stări prin care componenta $MOD = 1$.

Această facilitate se realizează adăugând n $EMUX$-uri şi o nouă ieşire spre ROM pentru a genera semnalul MOD.

De remarcat că pentru aceste n $EMUX$-uri, comanda $S_0 = 1$ înseamnă "iniţializare", iar $S_0 = 0$ înseamnă următoarea linie de comandă.

Această modificare permite simplificarea liniilor din ROM, o parte din informaţie fiind transferată spre preselecţia comandată de multiplexori.

A doua modificare se referă la intrarea X_2.

Deoarece biţii asociaţi acestei intrări sunt testaţi în diverse stări, MUX_p alege în fiecare stare bitul corespunzător, folosind selectorul $TEST$.

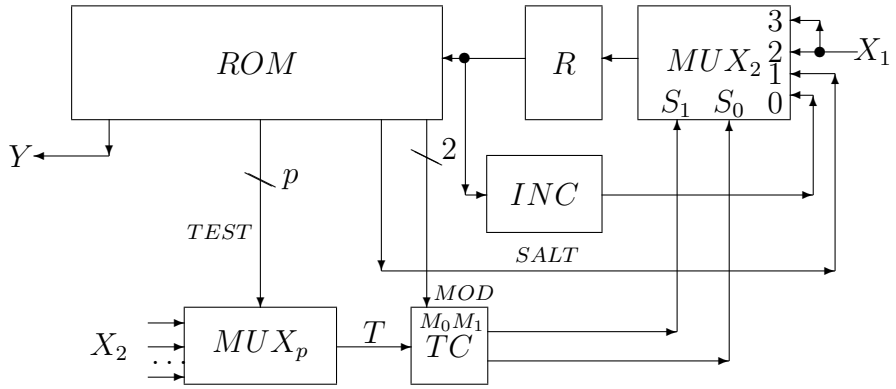
Deci biţii asociaţi intrării X_2 sunt separaţi de intrarea în ROM, adăugându-le câteva condiţii.

Astfel în loc de circa 2^p biţi, la intrarea în ROM se va conecta unul singur – T.

Această structură lucrează la fel ca cea iniţială, dar mărimea ROM-ului este foarte mult simplificată.

Lucrând cu această variantă a automatului de control, se pot realiza şi alte modificări. Astfel, cea mai mare parte a secvenţelor generate sunt organizate ca o secvenţe liniare. Deci, comenzile asociate pot fi stocate în ROM la adrese succesive (fiecare adresă din ROM se poate obţine prin incrementarea adresei curente aflate în registrul R).

Rezultă structura reprezentată în figura următoare:



Noutatea constă în circuitul de incrementare INC conectat la registrul de stare, precum şi un mic circuit combinaţional (TC – Circuit de testare) care codifică semnalele M_1, M_0, T într-un cod de selecţie pentru MUX_2.

Ieşirea din ROM poate fi privită ca o *micro - instrucţiune* definită astfel:

$$\begin{aligned} <\textit{Micro-instrucţiune}> &::= <\textit{Comandă}><\textit{Mod}><\textit{Test}><\textit{Salt}> \\ <\textit{Comandă}> &::= <\textit{definită de structura controlată}> \\ <\textit{Mod}> &::= JMP|CJMP|INIT \\ <\textit{Test}> &::= <\textit{definit de structura controlată}> \\ <\textit{Salt}> &::= <\textit{adresă nouă de maxim 6 simboluri}> \end{aligned}$$

Circuitul de testare TC construieşte funcţiile de selecţie:

$$S_1 = M_1 \wedge \overline{M}_0,\ S_0 = M_1 \vee (M_0 \wedge T)$$

O tabelă a valorilor celor două funcţii, precum şi a comenzilor selectate de MUX_2 pe baza acestor valori este:

M_1	M_2	T	S_1	S_0	Comandă
0	0	0	0	0	Următoarea comandă
0	0	1	0	0	Următoarea comandă
0	1	0	0	0	Următoarea comadă
0	1	1	0	1	Salt
1	0	0	1	1	Iniţializare
1	0	1	1	1	Iniţializare
1	1	0	0	1	Salt
1	1	1	0	1	Salt

Această variantă de circuit va fi numită $CROM$ (controller cu ROM) şi este utilizată în majoritatea structurilor circuitelor complexe de calculator.

Exemplul 7.6 *Să presupunem că în memoria ROM asociată unei unităţi logico-aritmetice (ALU) şi gestionată de un automat de control, se află instrucţiuni aritmetice.*

Să presupunem de asemenea că o instrucţiune standard are 16 *biţi (un semicuvânt) şi este de forma*

$$< operatie >< op_1 >< op_2 >< mod >< test >< adresa >$$

unde

1. $<$ **operatie** $>$ *este o operaţie aritmetică sau logică (adunare, scădere, comparare, deplasare etc) codificată pe* 3 *biţi, a cărei execuţie este asigurată de ALU.*

2. $<$ **op$_1$**, $<$ **op$_2$** $>$ *sunt adresele (pe câte* 2 *biţi) a două registre de memorie unde se află operanzii. Rezultatul se depune în* $< op_1 >$.

   Dacă operaţia necesită un singur operand, a doua adresă este 00.

3. $<$ **mod** $>$ *indică (pe un bit) modul de continuare: salt la instrucţiunea următoare sau salt condiţionat de valoarea unui anumit bit din* $< op_1 >$.

4. $<$ **test** $>$ *dă – pe* 3 *biţi – adresa bitului de test (dacă* $< mod >= 1$*) sau este ignorat (dacă* $< mod >= 0$*).*

5. $<$ **adresa** $>$ *conţine (pe* 5 *biţi) adresa instrucţiunii din memoria ROM care va fi accesată în continuare.*

Atunci memoria ROM care va conţine setul de instrucţiuni este un $(5, 16)$*-codificator.*

Astfel, să considerăm că la adresa 10110 *se află*

01101101 00001001

Interpretarea va fi următoarea:

- Se execută adunarea (cod 011 *executat de ALU) numerelor aflate în regiştrii R*1 *(cod* 01*) şi R*2 *(cod* 10*).*

- Dacă bitul aflat pe poziţia 0 *(cod* 000*) din rezultat este* 1 *(situaţie întâlnită atunci când suma depăşeşte lungimea standard alocată numerelelor), atunci instrucţiunea care se execută în continuare este la adresa* 01001*; alfel se execută instrucţiunea aflată la adresa următoare (*10111*).*

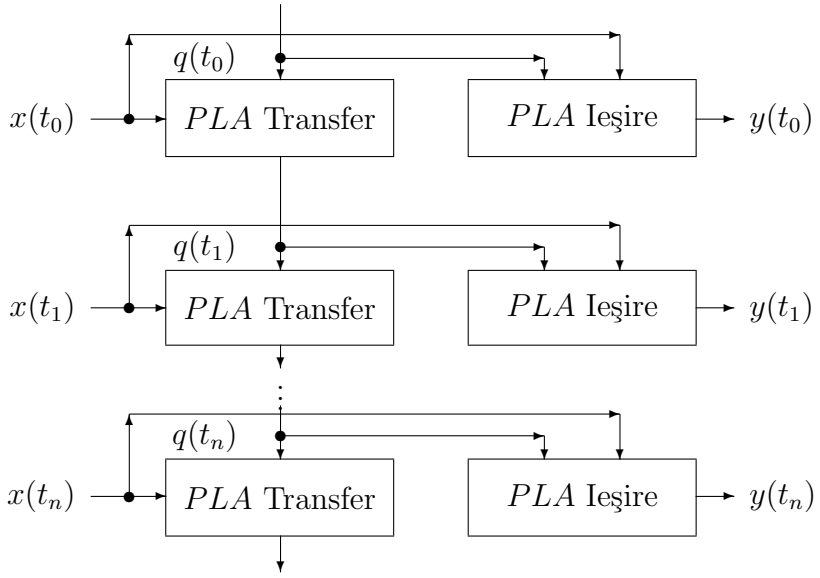
7.8 Transformarea automatelor în circuite combinaţionale

Atât circuitele combinaţionale $(0-DS)$ cât şi automatele $(2-DS)$ reprezintă modalităţi de implementare a funcţiilor booleene.

Ne putem pune următoarele întrebări:

1. Care este legătura dintre porţile unei reţele şi un automat care execută aceeaşi funcţie?
2. În ce condiţii putem transforma un circuit combinaţional într-un automat şi invers?

Răspunsul este simplu de dat şi se bazează pe tablouri logic programabile.



Fie un automat Mealy cu cele două PLA-uri ale sale (de transfer şi de ieşire), starea iniţială $q(t_0)$ şi secvenţa de intrare pentru primul ciclu de n tacţi: $x(t_0), x(t_1), \ldots, x(t_n)$.

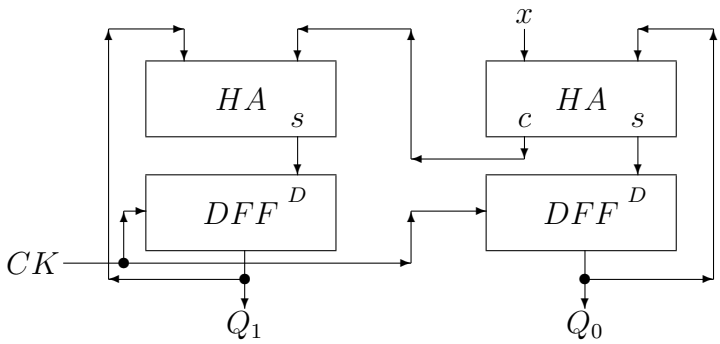
Circuitul combinaţional de sus generează secvenţa de ieşire corespunzătoare

$$y(t_0), y(t_1), \ldots, y(t_n)$$

Într-adevăr, prima pereche de două PLA-uri ("Transfer" şi "Ieşire") determină prima ieşire $y(t_0)$ şi starea următoare $q(t_1)$, care va fi utilizată de a doua pereche de PLA-uri pentru calculul celei de-a doua ieşiri şi stări etc.

7.9 Exerciţii

1. Se dă automatul



   Să se arate că poate fi folosit drept un counter crescător pe 2 biţi ($COUNT_2$).

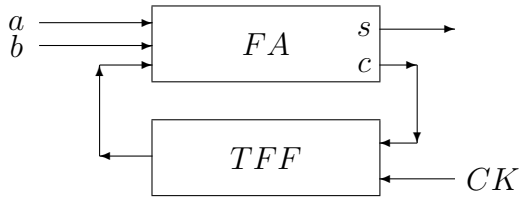
   Pe baza lui să se construiască un counter descrescător pe 2 biţi

2. Să se construiască un counter pe trei biţi ($UD - COUNT_3$).

   Se va folosi un bit de selecţie s cu semnificaţia: când $s = 0$ counterul va fi crescător, iar când $s = 1$, counterul va fi descrescător.

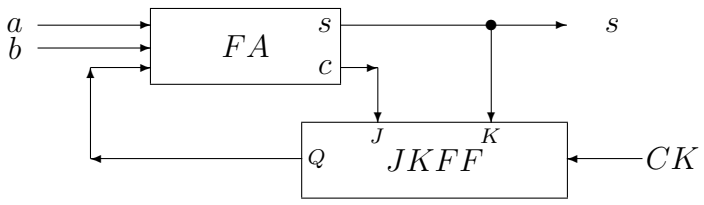
3. Să se construiască un sumator prefix pentru cazul $m = 1,\ n = 2$.

4. Se dă automatul

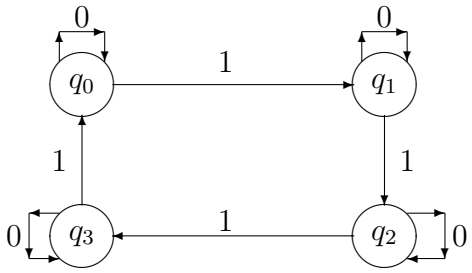


   Să se construiască funcţia de tranziţie, funcţia de ieşire şi graful de funcţionare.

5. Aceeaşi problemă pentru automatul



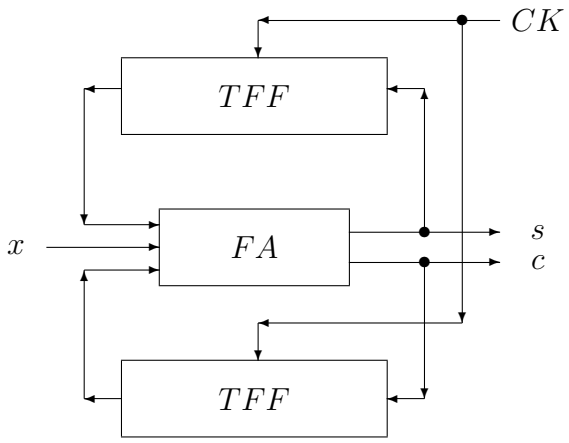
6. Se dă automatul finit reprezentat de graful



Să se construiască un circuit care să implementeze acest automat.

Cum trebuie definită funcţia de ieşire λ pentru ca acest automat să funcţioneze ca un counter descrescător pe 2 biţi ?

7. Se dă automatul:



Să se listeze comportarea sa (tabela funcţiei de tranziţie, graful de tranziţie).

8. Să se implementeze automatul anterior folosind numai DFF.

9. Să se implementeze automatul anterior folosind numai JK regiştri.

10. Să se verifice circuitul din Exemplul 7.4, arătând că este un counter zecimal pe 4 biţi

11. Să se construiască un circuit care primeşte o secvenţă binară şi numără de câte ori apare subsecvenţa 00. Numărul de apariţii se va da pe 3 biţi.

    De exemplu, pentru intrarea 0101110010000110 răspunsul este 100 (apar patru subsecvenţe 00).

12. Să se implementeze circuitul anterior folosind numai DFF.

13. Să se implementeze circuitul anterior folosind numai JK regiştri.

Capitolul 8

Sisteme $3 - DS$ sisteme (procesori)

Trecând la nivelul următor de complexitate al circuitelor, al treilea ciclu poate fi închis în trei variante, în funcţie de cele trei tipuri de circuite prezentate anterior.

1. Cel mai simplu tip de sistem $3-DS$ se obţine închizând al treilea ciclu cu un circuit combinaţional (de exemplu, peste un automat sau o reţea de automate, ciclul este închis folosind un $0-DS$).

2. Al doilea tip închide al treilea ciclu folosind un tip de memorie $(1-DS)$.

3. A treia variantă utilizată conectează printr-un ciclu două automate.

Acest ultim tip este caracteristic pentru un $3-DS$, definind ca principală aplicaţie *procesorul*.

8.1 Automate JK - regiştri

Un automat este specific unui circuit de ordin doi, dar funcţia sa poate fi implementată mai eficient folosind facilităţile oferite de sisteme cu un nivel mai înalt.

În această secţiune vom înlocui registrul de stări al automatului, cu un mecanism mai autonom: un "registru" de tip JK flip - flop[1].

Să vedem cum se poate înlocui un D flip - flop (DFF) – circuit cu o singură intrare – cu un JK flip - flop (circuit cu două intrări) şi cum această substituţie conduce la o minimizare a complexităţii circuitului.

[1] JK - registrul astfel construit nu este propriu-zis un registru; el este o reţea de automate simplificate, conectate în paralel.

Deoarece un $JKFF$ are o autonomie crescută faţă de DFF, la trecerea dintr-o stare în alta semnalele de intrare pot conţine un grad mai mare de ambiguitate.

Funcţia de tranziţie a unui JK flip - flop poate fi comandată astfel:

- pentru tranziţia $0 \longrightarrow 0$, JK poate fi 00 sau 01; vom scrie $JK = 0\_$ ("_" semnifică "orice valoare");
- pentru $0 \longrightarrow 1$, JK poate fi 11 sau 10; deci $JK = 1\_$;
- pentru $1 \longrightarrow 0$, JK poate fi 11 sau 01; deci $JK = \_1$;
- pentru $1 \longrightarrow 1$, JK poate fi 00 sau 10; deci $JK = \_0$.

Putem restrânge aceste reguli sub o formă unitară:

$$JK = \begin{cases} A\_ & \text{pentru } 0 \longrightarrow A \\ \_\,\overline{A} & \text{pentru } 1 \longrightarrow A \end{cases}$$

De aici rezultă că fiecare tabelă de tranziţie care defineşte o nouă valoare a stării Q_i^+ poaate fi translatată în două diagrame de tranziţie: una pentru J_i, a doua pentru K_i.

Exemplul 8.1 *Să reluăm de la diagrama funcţiei de tranziţie care descrie funcţia δ a unui MAC (a se vedea secţiunea* 7.6.1*).*

δ	000	001	010	011	100	101	110	111
000	000	001	000	001	000	001	000	001
001	001	011	001	011	001	011	001	011
011	011	011	011	011	010	010	010	010
010	010	100	010	001	010	100	010	001
100	000	000	000	000	000	000	000	000

Ea era implementată prin trei DFF care descriau comportamentele stărilor Q_2, Q_1, Q_0.

Noile diagrame de tranziţie pentru J_i şi K_i pot fi definite pe baza regulii anterioare, rezultând următoarele două tabele pentru cele 6 *intrări ale JK - registrului ($J_2J_1J_0$ respectiv $K_2K_1K_0$).*

δ_J	000	001	010	011	100	101	110	111	J_2^+	J_1^+	J_0^+
000	000	001	000	001	000	001	000	001	0	0	S
001	00_	01_	00_	01_	00_	01_	00_	01_	0	S	_
011	0__	0__	0__	0__	0__	0__	0__	0__	0	_	_
010	0_0	1_0	0_0	0_1	0_0	1_0	0_0	0_1	$\overline{I} \wedge S$	_	$I \wedge S$
100	_00	_00	_00	_00	_00	_00	_00	_00	_	0	0

δ_K	000	001	010	011	100	101	110	111	K_2^+	K_1^+	K_0^+
000	___	___	___	___	___	___	___	___	_	_	_
001	__0	__0	__0	__0	__0	__0	__0	__0	_	_	0
011	_00	_00	_00	_00	_01	_01	_01	_01	_	0	E
010	_0_	_1_	_0_	_1_	_0_	_1_	_0_	_1_	_	S	_
100	1__	1__	1__	1__	1__	1__	1__	1__	1	_	_

Deci tabelele PLA de tranziţie sunt descrise de aplicaţiile

$\delta_J(Q_2Q_1Q_0, EIS) = (\overline{Q}_2 \wedge Q_1 \wedge \overline{Q}_0 \wedge \overline{I} \wedge S,\ \overline{Q}_2 \wedge \overline{Q}_1 \wedge Q_0 \wedge S,\ (\overline{Q}_2 \wedge \overline{Q}_1 \wedge \overline{Q}_0 \wedge S) \vee (\overline{Q}_2 \wedge Q_1 \wedge \overline{Q}_0 \wedge I \wedge S))$

$\delta_K(Q_2Q_1Q_0, EIS) = (\prime 1\prime,\ \overline{Q}_2 \wedge Q_1 \wedge \overline{Q}_0 \wedge S,\ \overline{Q}_2 \wedge Q_1 \wedge Q_0 \wedge E)$

Deoarece cu trei biţi am codificat numai cinci stări (maximul fiind opt): $000, 001, 011, 010, 100$*, expresiile pot fi simplificate*[2]*:*

Astfel, din

$Q_2Q_1Q_0$	$\overline{Q}_2 \wedge Q_1 \wedge \overline{Q}_0$	$Q_1 \wedge \overline{Q}_0$	$\overline{Q}_0$
000	0	0	_
001	0	0	_
011	0	0	_
010	1	1	1
100	0	0	_

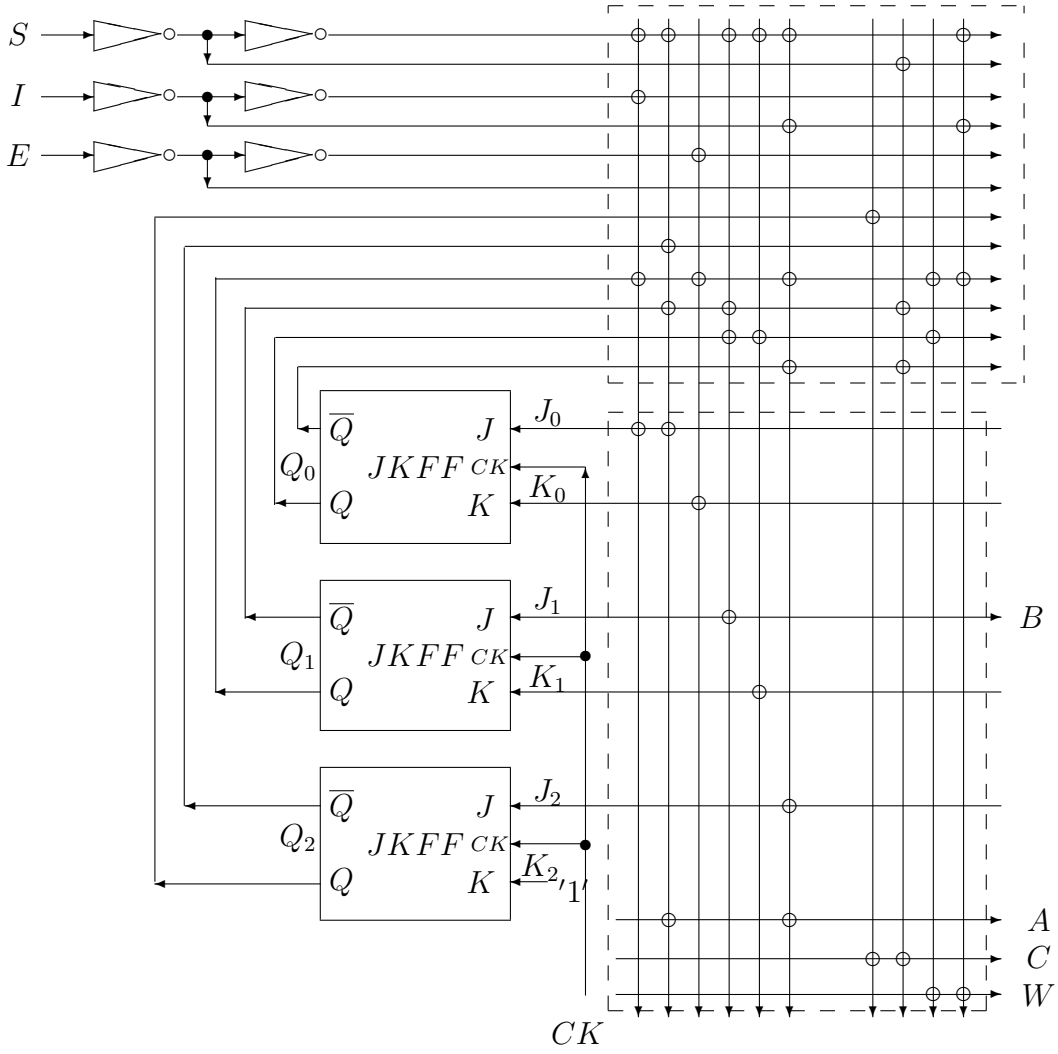
rezultă $\overline{Q}_2 \wedge Q_1 \wedge \overline{Q}_0 \wedge S \wedge \overline{I} = Q_1 \wedge \overline{Q}_0 \wedge S \wedge \overline{I}$ *şi* $\overline{Q}_2 \wedge Q_1 \wedge \overline{Q}_0 \wedge S = \overline{Q}_0 \wedge S$

Similar,
$\overline{Q}_2 \wedge Q_1 \wedge Q_0 \wedge E = Q_1 \wedge E$, $\overline{Q}_2 \wedge Q_1 \wedge \overline{Q}_0 \wedge S \wedge I = Q_1 \wedge S \wedge I$,
$\overline{Q}_2 \wedge \overline{Q}_1 \wedge \overline{Q}_0 \wedge S = \overline{Q}_2 \wedge \overline{Q}_1 \wedge S$, $\overline{Q}_2 \wedge \overline{Q}_1 \wedge Q_0 \wedge S = \overline{Q}_1 \wedge Q_0 \wedge S$.

Deci
$\delta_J(Q_2Q_1Q_0, EIS) = (Q_1 \wedge \overline{Q}_0 \wedge \overline{I} \wedge S,\ \overline{Q}_1 \wedge Q_0 \wedge S,\ (\overline{Q}_2 \wedge \overline{Q}_1 \wedge S) \vee (Q_1 \wedge I \wedge S))$
$\delta_K(Q_2Q_1Q_0, EIS) = (\prime 1\prime,\ \overline{Q}_0 \wedge S,\ Q_1 \wedge E)$

Pe baza acestor ecuaţii se poate construi circuitul:

[2]Reamintim, pe poziţia marcată "_" poate apare orice valoare.



Funcţia de ieşire λ descrisă este cea definită în secţiunea 7.6.1.

*Este evident că distribuţia punctelor în cele două PLA-uri este mai mică decât cea din soluţia dată anterior (*38 *puncte faţă de* 48 *în construcţia precedentă).*

Prin această înlocuire a fiecărui DFF cu un $JKFF$, o parte din PLA-ul de transfer a fost *segregat* în structura JK - registrului.

Această metodă acţionează ca un mecanism care subliniază regularităţile unui proces şi permite construcţia de circuite mai mici şi mai simple pentru aceeaşi funcţie.

Eficienţa acestei metode este direct proporţională cu complexitatea sistemului.

Exemplul 8.2 *Pentru automatul construit în Capitolul 7, secţiunea* 7.6.2 *înlocuirea celor două DFF cu $JKFF$ va creşte mărimea tablourilor logic programabile.*

Într-adevăr, din funcţia de transfer

δ	0	1	Q_1^+	Q_0^+
00	00	01	0	x
01	10	01	$\overline{x}$	x
10	00	11	x	x
11	10	01	$\overline{x}$	x

rezultă imediat tabelele funcţiilor δ_J şi δ_K:

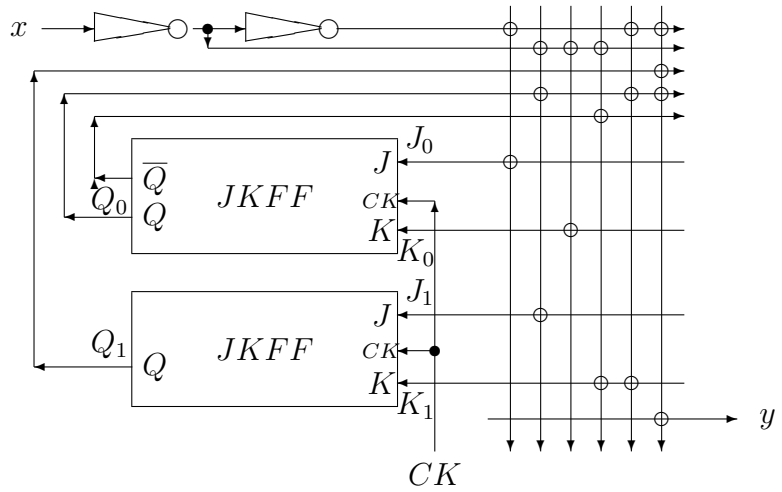
δ_J	0	1	J_1	J_0
00	00	01	0	x
01	1 _	0 _	$\overline{x}$	_
10	_ 0	_ 1	_	x
11	_ _	_ _	_	_

δ_K	0	1	K_1	K_0
00	_ _	_ _	_	_
01	_ 1	_ 0	_	$\overline{x}$
10	1 _	0 _	$\overline{x}$	_
11	01	10	x	$\overline{x}$

De aici pot fi scrise funcţiile de transfer (după reduceri):

$$\delta_J(Q_1Q_0, x) = (Q_0 \wedge \overline{x},\ x), \qquad \delta_K(Q_1Q_0, x) = ((\overline{Q}_0 \wedge \overline{x}) \vee (Q_0 \wedge x),\ \overline{x})$$

Implementarea întregului automat va fi



Deci folosirea automatelor JK regiştri este eficientă numai pentru tablouri aleatoare de dimensiune mare.

Posibilitatea de minimizare a PLA-urilor este permisă de autonomia crescută a $JKFF$ faţă de DFF. Un DFF are doar o autonomie parţială de a sta într-o anumită stare, pe când un $JKFF$ poate evolua în spaţiul stărilor.

Un DFF trebuie să "raporteze" permanent la intrare care va fi următoarea stare (0 sau 1), pe când un $JKFF$ permite comenzi mult mai generale, $J = K = 1$ – care spune "treci în altă stare" (sau $J = K = 0$ – "păstrează starea"), fără a o indica exact care este aceasta (de fapt, al doilea ciclu ajută un $JKFF$ să-şi cunoască starea curentă).

8.2 Regiştri numărători

Există diverse moduri de a simplifica un PLA, prin personalizarea unui automat.

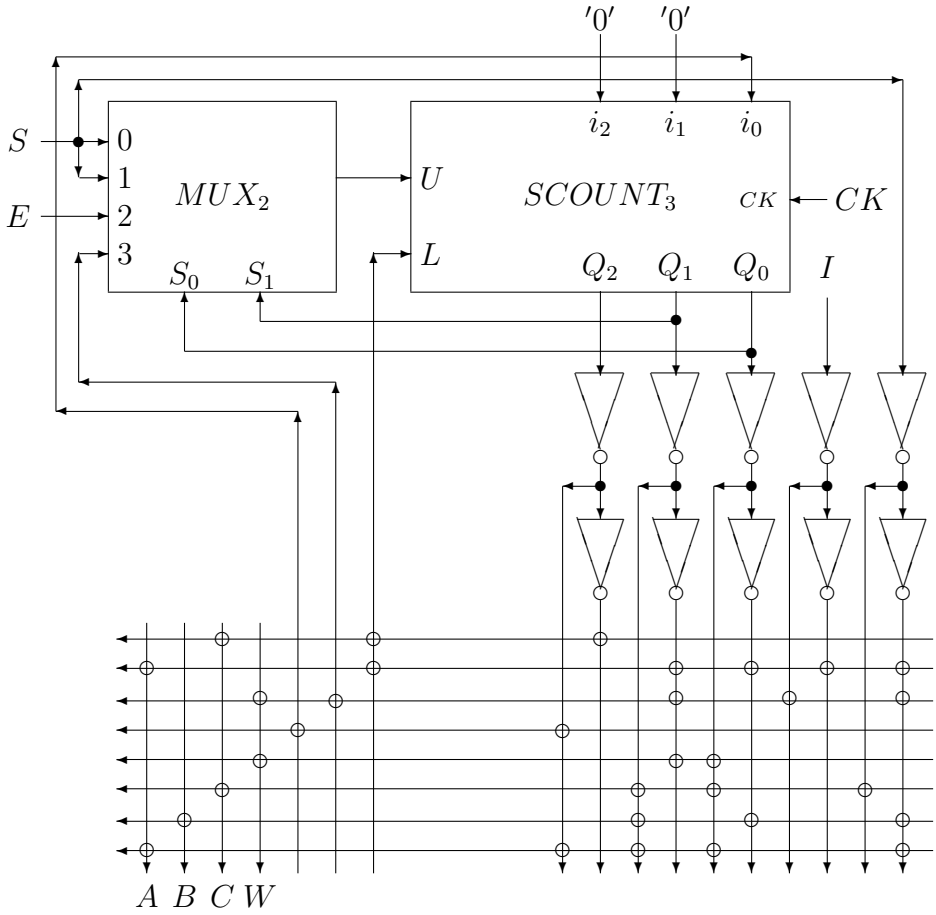
O soluţie des întâlnită constă în folosirea unui numărător sincron setabil $SCOUNT_n$.

Acesta este un circuit care combină două funcţii: cea de registru (încărcat la comanda L) şi cea de numărător (crescător, la comanda U).

Din cele două funcţii, cea de încărcare are prioritate.

Exemplul 8.3 *Să reluăm circuitul MAC definit în* 7.5.5. *Pentru a adapta asignarea stărilor utilizând un numărător setabil ca registru de stări, va trebui să utilizăm codurile stărilor (detaliate în schema logică din* 7.6.1*).*

Ideea este de a avea în acea schemă logică cât mai multe tranziţii obţinute prin incrementare.



Construcţia dată în figura următoare pleacă de la un $SCOUNT_3$ şi un MUX_2.

Multiplexorul selectează funcţia de numărare în fiecare stare, conform cu descrierea din schema logică.

Astfel de exemplu, din starea 000 *tranziţia este dată de contor dacă* $S = 1$*; altfel, automatul rămâne în aceeaşi stare.*

Deci, multiplexorul selectează – pentru intrarea U *a contorului – valoarea lui* S*.*

Funcţia de încărcare a contorului este selectată cu funcţia $L = Q_2 \vee (Q_1 \wedge Q_0 \wedge S \wedge I)$ *(extrasă direct din schema logică).*

Totul se face pentru ca intrările i_0, i_1, i_2 *în* $SCOUNT_3$ *să aibă valori corecte numai dacă în starea curentă tranziţia se poate realiza încărcând o anumită valoare în circuitul de control.*

Deci, în definiţia funcţiilor logice asociate cu aceste intrări putem avea destul de multă libertate.

Va rezulta $i_2 = i_1 = 0,\ i_0 = \overline{Q}_2$*.*

Intrarea 3 *a multiplexorului este* $Q_1 \wedge S \wedge \overline{I}$ *(pentru a maximiza termenii folosiţi în comun de* PLA*).*

Structura rezultată în circuitul descris anterior are o componentă aleatoare minimă.

Procesul de segregare este mai adânc dacă implementăm circuite simple, dar definite recursiv.

8.3 Automat aritmetic şi logic

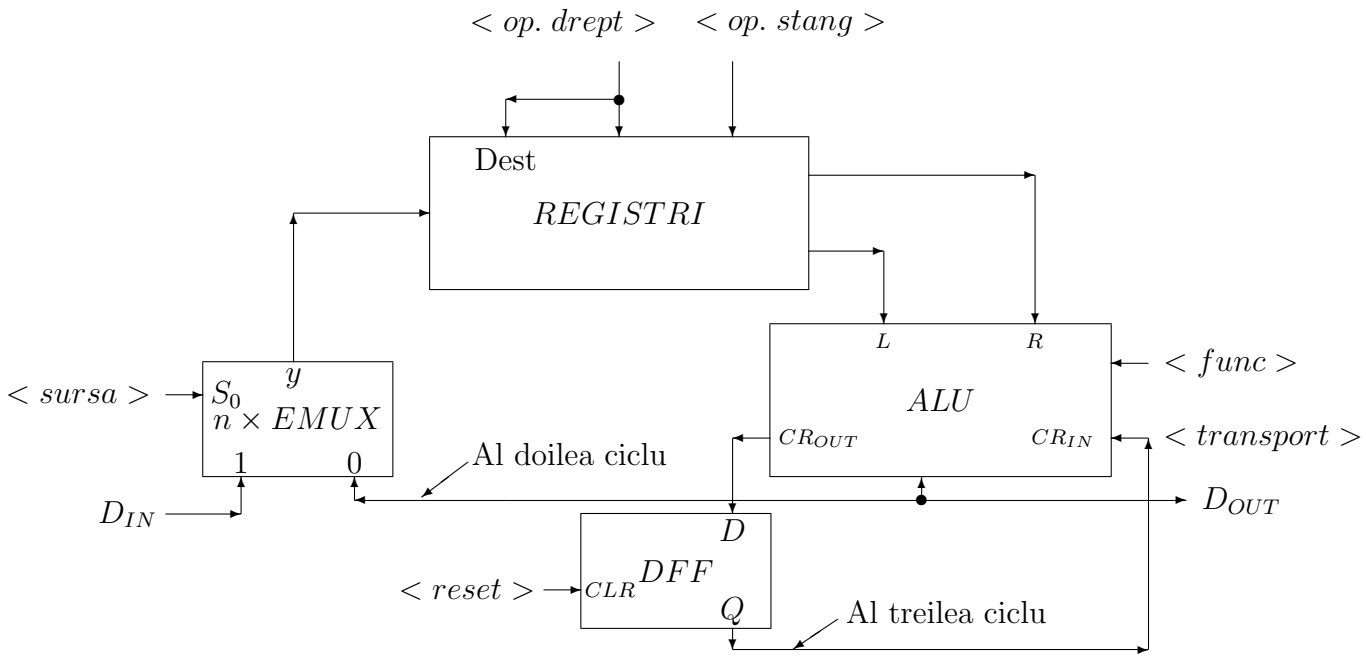
Cele două secţiuni anterioare s-au referit la exemple de circuite în care al treilea ciclu se închide într-un circuit combinaţional.

Să tratăm acum un automat având un ciclu închis printr-o memorie.

Scopul este de a sublinia efectul acestui de-al treilea ciclu în sistemele care conţin astfel de circuite.

Structura prezentată mai jos este folosită pentru realizarea de funcţii aritmetice şi logice (ALU) cu date stocate în opt regiştri de câte n biţi fiecare; pentru simplificarea circuitului, nu s-au mai descris ceasurile de sincronizare; de asemenea, toate magistralele sunt de n biţi.

Vom analiza două variante de construcţie, fără şi cu DFF conectat între CR_{OUT}-ul şi CR_{IN}-ul circuitului ALU.



Circuitul conţine:

- **REGISTRI**: o grupare de 8 regiştri de câte n biţi fiecare; la fiecare tact sunt selectaţi doi din ei pentru a transmite codurile operanzilor drept respectiv stâng (la intrarea în ALU); un al treilea registru este desemnat la fiecare tact pentru ieşire, cu codul $<$ *op drept* $>$ (pentru selecţie se pot folosi două $n \times MUX_3$, iar la destinaţie ceasul este distribuit printr-un $DMUX_3$).

- **ALU**: are structura şi funcţiile prezentate în *Unitatea aritmetică şi logică* (5.11).
- **n** $\times$ **EMUX**: închide ciclul prin aducerea rezultatului de la ieşirea din ALU la regiştri (pentru $S_0 = 0$) sau permite încărcarea regiştrilor cu date primite de la intrarea D_{IN} (pentru $S_0 = 1$).
- **D**: este un DFF care generează la intrarea CR_{IN} în ALU valoarea de transport generată la tactul precedent de ieşirea CR_{OUT} din ALU; poate fi resetat activând bitul "$reset$". Prima variantă pe care o vom prezenta nu va conţine acest flip - flop.

Automatul de control este un $CROM$ (secţiunea 7.7).

Varianta I

Linia de comandă primită de la un automat de control are definiţia:

$$\begin{array}{rcl} < comanda > & ::= & < op\ stang >< op\ drept >< func >< transport >< sursa > \\ & ::= & < reset > \\ < op.\ stang > & ::= & L0 \mid L1 \mid \ldots \mid L7 \\ < op\ drept > & ::= & R0 \mid R1 \mid \ldots \mid R7 \\ < func > & ::= & AND \mid OR \mid XOR \mid ADD \mid SUB \mid INC \mid SHL \\ < transport > & ::= & CR \mid \_ \\ < sursa > & ::= & D_{IN} \mid \_ \end{array}$$

Să presupunem că în două perechi de regiştri sunt două numere de câte $2n$ biţi care trebuie adunate.

Primul număr este stocat în perechea de regiştri $(L1, L2)$ ($L2$ conţine cei mai semnificativi n biţi); al doilea număr este stocat similar în $(R1, R2)$.

Secvenţa de comenzi care controlează această adunare este:

$$\begin{array}{cccccc} & L1 & R1 & ADD & CJMP & CR_{OUT} \quad UNU \\ & L2 & R2 & ADD & JMP & DOI \\ UNU & L2 & R2 & ADD & CR & \\ DOI & \ldots & & & & \end{array}$$

Varianta II

Această variantă este un $3 - DS$ în care ciclul se închide printr-un DFF.

Rolul ultimului ciclu este de a reduce, simplifica şi a mări viteza rutinei care realizează aceeaşi operaţie sau alte proceduri dependente.

Diferenţa din linia de comandă constă în eliminarea câmpului $< transport >$ şi adăugarea la $< reset >$ a definiţiei

$< reset >::= RST \mid \_$

Diferenţa funcţională constă în modul în care conţinutul DFF - ului comandă bitul de transport din intrarea în sumator (CR_{IN}).

Astfel, secvenţa de comenzi care realizează suma anterioară devine

$$\begin{array}{lll} \ldots & RST & \ldots \\ L1 & R1 & ADD \\ L2 & R2 & ADD \end{array}$$

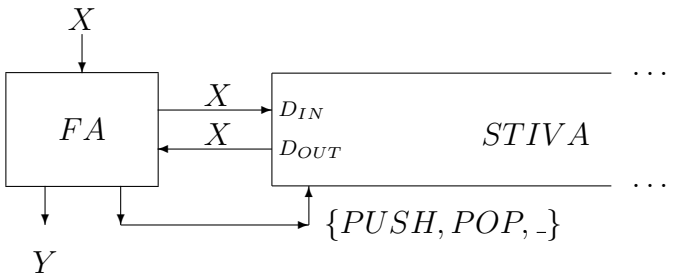
8.4 Automate stivă

Să considerăm acum (în construcţia $3 - DS$ sistemelor) varianta în care al treilea ciclu se închide printr-un automat.

Acest ciclu are posibilitatea de a minimiza mărimea componentei aleatoare.

Mai mult, această micşorare va afecta ordinul de mărime al părţii aleatoare (până acum, această reducere se realiza cu o constantă).

Structura propusă – cu deosebit de multe aplicaţii – este *automatul stivă*.



Definiţia 8.1 *Un automat stivă PDA (PushDown Automaton) este construit prin legarea într-un ciclu a unui automat finit cu o stivă (memorie $LIFO$) (a se vedea Figura).*

Formal, el este un 7-tuplu $PDA = (Q, X, Y, Z, \delta, \lambda, z_0)$, unde

- **Q***: mulţimea (finită a) stărilor automatului;*
- **X***: alfabet finit nevid; secvenţa de intrare, cea stocată în stivă şi cea primită din stivă sunt elemente din X^*.*
- **Y***: alfabetul de ieşire;*
- **Z** *: mulţimea de comenzi către stivă: $\{PUSH, POP, NOP\}$;*

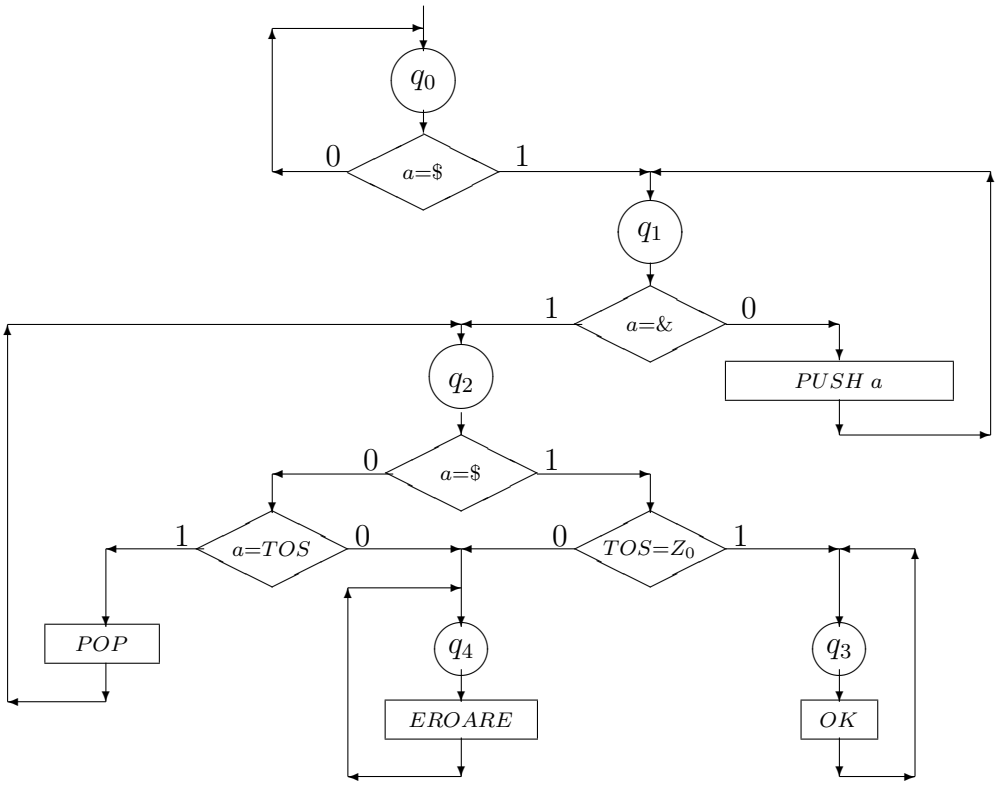
- δ: *funcţia de tranziţie;* $\delta : Q \times X \times X \longrightarrow Q \times X^* \times Z$; *plecând dintr-o stare, simbolul primit şi simbolul din topul stivei* (TOS), *automatul comută într-o nouă stare, o nouă valoare (eventual) în stivă, bazată pe comandă de tip* $PUSH, POP$ *sau* NOP. *Uneori pot apare şi comenzi de început sau sfârşit de program* $(BEGIN, END)$.

- λ: *funcţia de ieşire a automatului;* $\lambda : Q \longrightarrow Y$;

- $\mathbf{z_0}$: *valoarea iniţială din stivă (iniţializarea lui* TOS*)*

Este uşor să înţelegem rolul acestui tip de maşină folosind un exemplu:

Exemplul 8.4 *Să construim un automat care să recunoască şirurile de forma* $\$\alpha\&\tilde{\alpha}\$$, *unde* $\$, \& \in X$, $\alpha \in X^* \backslash \{\$, \&\}$. *Soluţia este un* PDA *descris de schema logică următoare:*



Semnificaţia fiecăreia din cele 5 *stări este:*

- q_0: *este starea iniţială, când se aşteaptă primul caracter* $\$$.

- q_1: *în această stare, simbolurile primite sunt puse în stivă* $(PUSH)$; *la recepţionarea lui* & *automatul comută în starea următoare.*

- *q_2: fiecare simbol primit este comparat cu vârful stivei (TOS); dacă cele două caractere sunt egale, stiva elimină caracterul comparat (POP); dacă simbolul primit este \$ iar TOS conţine z_0, automatul trece în starea q_3; în orice altă situaţie, automatul trece în starea q_4.*

- *q_3: automatul recunoaşte secvenţa primită ca fiind corectă; este o stare finală de acceptare.*

- *q_4: automatul respinge secvenţa primită ca fiind incorectă; este o stare finală de eroare.*

Formal, automatul stivă este $M = (Q, X, Y, Z, \delta, \lambda, z_0)$ unde:
$Q = \{q_0, q_1, q_2, q_3, q_4\}$,
$X = \{z_0, \$, \%\} \cup V$ *unde* V *este un alfabet arbitrar, definit separat,*
$Y = \{OK, EROARE, \_\}$,
$Z = \{PUSH, POP, NOP, BEGIN, END\}$,
funcţia de tranziţie

$$\begin{array}{ll}
\delta(q_0, a, z_0) = (q_0, z_0, NOP), & \delta(q_1, a, b) = (q_1, ab, PUSH\ a),\ \forall a \neq \&,\ b \in V \cup \{z_0\} \\
\delta(q_0, \$, z_0) = (q_1, z_0, BEGIN) & \delta(q_1, \&, b) = (q_2, b, NOP) \\
\delta(q_2, a, a) = (q_2, \epsilon, POP\ a),\ \forall a \neq \$ & \delta(q_2, a, b) = (q_4, b, NOP), \quad \forall a \neq b, \quad a \neq \$ \\
\delta(q_2, \$, z_0) = (q_3, \epsilon, END), & \delta(q_2, \$, a) = (q_4, a, NOP), \quad \forall a \neq z_0
\end{array}$$

şi funcţia de ieşire
$\lambda(q_0) = \lambda(q_1) = \lambda(q_2) = \_\,, \quad \lambda(q_3) = OK, \quad \lambda(q_4) = EROARE.$

Secvenţa de comenzi către stivă poate fi interpetată (şi implementată) ca un microprogram care operează cu şiruri de caractere.

De remarcat că rezolvarea acestei probleme folosind un automat finit este posibilă numai pentru un alfabet X mic şi şiruri de intrare mici.

Mărimea şi complexitatea automatului va depinde direct de aceşti doi parametri. Orice modificare a lor va schimba complet circuitul.

În cazul utilizării un automat stivă însă, mărimea şirului de intrare nu are nici o legătură cu dimensiunea sau forma automatului.

Aici, partea simplă (stiva infinită) este separată de cea complexă (automatul finit, cu 5 stări).

Observaţia 8.1 *Definiţia automatului stivă este foarte asemănătoare cu definiţia translatorului stivă de la limbaje formale. Diferenţele sunt minore: acolo se lucrează cu un automat de tip Mealy, alfabetul stivă este distinct de alfabetul de intrare, se pot folosi definiţii nedeterministe (care – atenţie – nu sunt echivalente cu cele deterministe).*

8.5 Procesorul elementar

Cel mai reprezentativ circuit din $3 - DS$ este procesorul; el se remarcă prin flexibilitate şi posibilităţi funcţionale mult dezvoltate.

Definiţia 8.2 *Un procesor P este un circuit care leagă printr-un ciclu un automat aritmetico - logic cu un automat de control.*

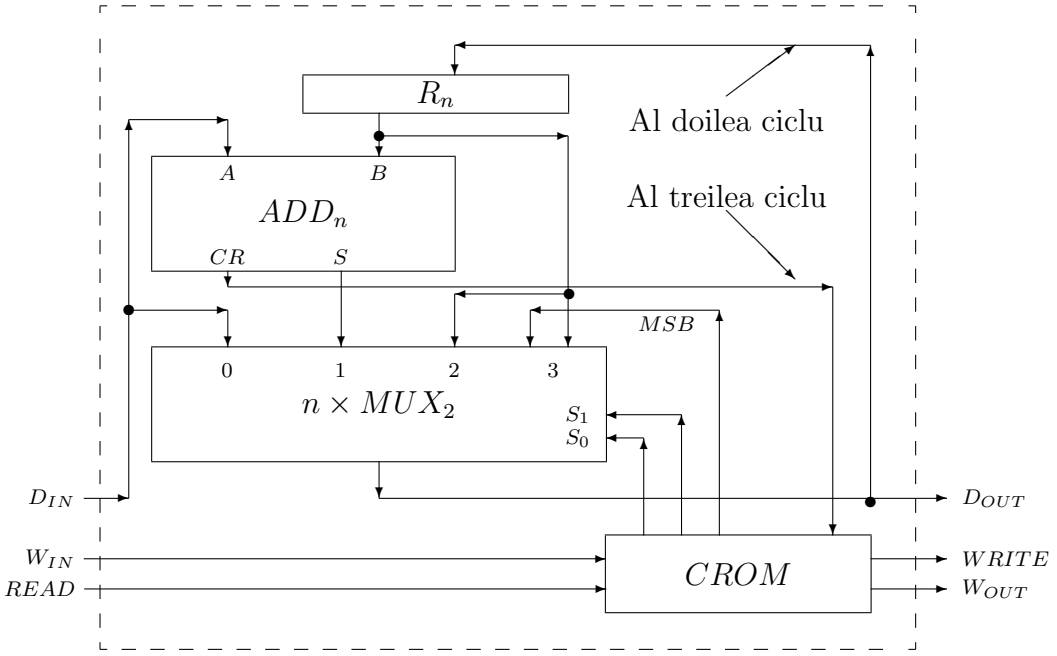
Funcţia unui procesor P este specificată prin secvenţa de comenzi stocată în ROM.

Fiecare secvenţă reprezintă un *microprogram* şi constă dintr-un şir de *micro - instrucţiuni.*

O micro - instrucţiune este compusă din *comenzi* executate de automatul aritmetico - logic (ALU), şi *câmpuri* care permit selectarea următoarei micro - instrucţiuni.

Pentru a înţelege mecanismul principal folosit de acest al treilea ciclu, ne vom referi în principal la *procesorul elementar.*

Un procesor elementar EP este un procesor care execută numai o secvenţă de control, adică automatul de control ($CROM$) asociat este un automat de iniţializare, fără intrarea de iniţializare X_1 (a se vedea circuitul $CROM$, secţiunea 7.7).



Un sistem simplu de PE este dat în figura de sus, unde s-a detaliat numai automatul aritmetico - logic.

Pachetul de n multiplexori $n \times MUX_2$ selectează simultan ieşirea secţiunii aritmetico - logice şi intrarea în registrul R_n (nu s-au mai precizat mărimile magistralelor de date).

Selecţiile efectuate de el sunt:

- **IN**: intrarea de date D_{IN}, accesibilă pentru comanda $S_1S_0 = 00$;
- **ADD$_\mathbf{n}$**: suma dintre numărul (de n biţi) oferit de D_{IN} şi conţinutul registrului R_n, pentru comanda $S_1S_0 = 01$;
- **REG**: conţinutul registrului paralel R_n, pentru comanda $S_1S_0 = 10$;
- **RSH0**: conţinutul registrului R_n deplasat spre dreapta cu o poziţie (având 0 pe cea mai semnificativă poziţie), pentru comanda $S_1S_0 = 11$ completată de $MSB = 0$;
- **RSH1**: conţinutul registrului R_n deplasat spre dreapta cu o poziţie (având 1 pe cea mai semnificativă poziţie) pentru comanda $S_1S_0 = 11$ completată cu $MSB = 1$.

Automatul este un $CROM$ specific unei *funcţii*, care comandă automatul aritmetico - logic prin intermediul grupului de trei biţi (S_0, S_1, MSB); el primeşte comanda de la ieşirea CR a sumatorului.

De asemenea, acest $CROM$ controlează legăturile procesorului. EP este conectat cu un sistem din care primeşte date prin $READ$ şi W_{IN} (intrare de aşteptare) şi căruia îi trimite ieşirile $WRITE$ şi W_{OUT} (ieşirea de aşteptare).

Descrierea funcţională este completată prin *limbajul de microprogramare* al acestui procesor, definit sintactic astfel:

$$
\begin{aligned}
< micro-instructiune > &::= < comanda >< mod >< test >< salt > \\
< comanda > &::= < functie >< iesire > \\
< mod > &::= JMP \mid CJMP \mid INIT \mid \_ \\
< test > &:: = CR \mid WIN \mid WOUT \mid \_ \\
< salt > &::= < \text{orice marcă de maxim 6 caractere} > \\
< functie > &::= IN \mid ADD \mid REG \mid RSH0 \mid RSH1 \\
< iesire > &::= READ \mid WRITE \mid \_
\end{aligned}
$$

Exemplul 8.5 *Singurul micro-program executat de procesorul elementar descris mai sus primeşte un şir de numere şi generează alt şir de numere reprezentând mediile aritmetice ale fiecărei perechi succesive.*

Utilizând limbajul de micro-programare astfel definit, rezultă următorul microprogram:

```
UNU   IN     READ   CJMP   WIN    UNU
DOI   READ   CJMP   WIN    DOI
      ADD    CJMP   CR     TREI
      RSH0
OUT   REG    WRITE  CJMP   WOUT   ZERO
      JMP    UNU
TREI  RSH1   JMP    OUT
ZERO  ...
```

Pe prima linie, controlerul aşteaptă primirea primului număr, încărcând în R_n orice configuraţie binară; când W_{IN} devine inactiv, ultima încărcare pune în registru valoarea corectă.

A doua micro - instrucţiune aşteaptă al doilea număr; la sosirea acestuia, microprogramul merge la linia următoare.

A treia linie adună conţinutul registrului cu numărul sosit în D_{IN}, iar ieşirea CR este testată pentru a alege deplasarea corespunzătoare.

Dacă CR este activ, următoarea instrucţiune este la adresa $TREI$; altfel se trece la micro - instrucţiunea următoare.

A patra şi următoarele micro - instrucţiuni efectuează deplasările spre dreapta corespunzătoare (adică împărţirea), finalizând calculul mediei dintre numerele primite.

Linia OUT trimite afară rezultatul, iar receiverul confirmă primirea.

Saltul (JMP) la UNU reporneşte (la nesfârşit) procedura pentru următoarea pereche de numere.

Structura de ansamblu a unui EP nu "spune" nimic despre funcţia realizată.

Această funcţie este definită numai prin conţinutul ROM-ului din $CROM$. Structura simbolică stocată în ROM combină facilităţi funcţionale simple ale automatului efectiv, adică este un automat aritmetico - logic.

Componenta aleatoare este concentrată în ROM, al cărui conţinut formează singura structură complexă a sistemului – structură de tip simbolic (micro - programare).

Capitolul 9

Sisteme $4 - DS$

Ultimul exemplu din secţiunea 8.5 subliniază destul de clar procesul specific unui $3 - DS$: funcţia sistemului devine tot mai puţin dependentă de structura fizică şi tot mai strâns legată de o structură simbolică (de exemplu, micro-programe).

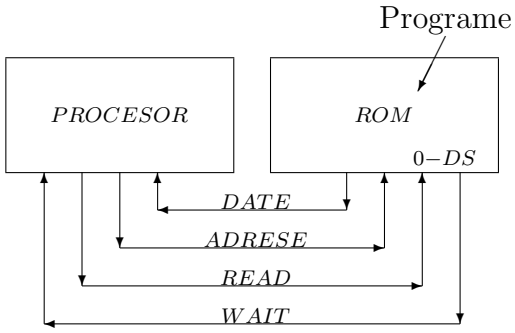
Structura fizică (bazată pe circuite) rămâne simplă, pe când cea simbolică – "stocată" în ROM (eventual convertit în PLA) – presupune o anumită complexitate funcţională.

Al patrulea ciclu crează condiţiile pentru dependenţa funcţională totală faţă de structura simbolică.

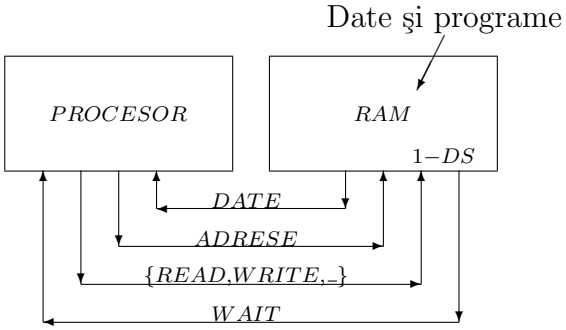
9.1 Tipuri de sisteme de ordin 4

Există patru tipuri principale de astfel de sisteme, în funcţie de ordinul sistemului prin care este închis al patrulea ciclu; ele pot reprezentate schematic astfel:

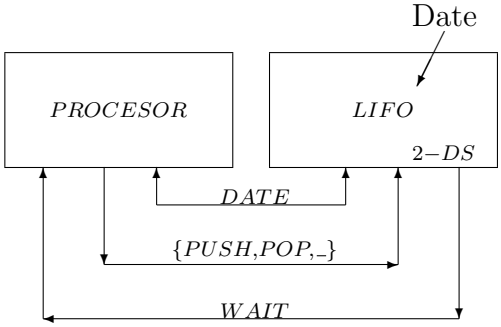
1. $4 - DS$ având ultimul ciclu închis printr-un $0 - DS$; circuitul combinaţional de închidere este un ROM sau PLA, conţinând programe interpretate de procesor.



2. $4-DS$ cu ultimul ciclu închis printr-un $1-DS$; acesta este **computerul**, cea mai reprezentativă structură de acest ordin; a doua componentă este un RAM care stochează date şi programe.

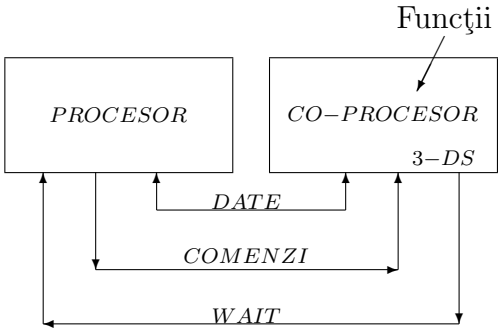


3. $4-DS$ având ultimul ciclu închis printr-un $2-DS$; automatul de închidere este reprezentat printr-o stivă care conţine datele de lucru (sau secvenţe în care distincţia dintre date şi programe nu are loc – cum este de exemplu programarea în limbajul $LISP$);



4. $4-DS$ cu ciclul final închis printr-un $3-DS$; acesta este *co-procesorul* (procesor specializat în executarea performantă a anumitor funcţii critice ale sistemului).

   În majoritatea cazurilor, co-procesorul este un procesor aritmetic.



Ultimele două tipuri de extensii sunt maşini în care procesul de segregare este subliniat de structura fizică (stivă sau co-procesor); în ambele cazuri structura segregată este de asemenea simplă.

Consecinţa este aceea că întregul sistem este de asemenea simplu.

Pe de altă parte, primele două sistem sunt foarte complexe, cu o componentă aleatoare apreciabilă.

Aceasta are ca suport structura ROM/PLA, respectiv memoria RAM.

Conţinutul ROM-ului este iniţial definit simbolic; ulterior el este convertit în structura circuitului respectiv (ROM sau PLA).

În schimb conţinutul RAM- ului rămâne permanent în forma simbolică, ceea ce îi conferă o mai mare flexibilitate.

Aceasta este principala raţiune prin care se consideră structura (b) (Computer) ca fiind cea mai reprezentativă în $4 - OS$.

Computerul *nu este un circuit.*

El este o entitate nouă, cu o definiţie funcţională specifică, numită uzual *arhitectura calculatorului.*

În principal arhitectura calculatorului este dată de limbajul maşină interpretat de procesorul care execută programele, stocat în memoria RAM.

Fiecare arhitectură poate avea mai multe structuri de calculator asociate.

De la nivelul sistemului de ordin 4, comportarea întregului sistem poate fi controlată în principal prin structura simbolică a programelor.

Se pune întrebarea ce se întâmplă dacă se adaugă noi cicluri.

Acestea se mai introduc de obicei în interiorul structurii procesorului, pentru a facilita capacitatea sa de a interpreta mai rapid (cu circuite mai simple) limbajul maşină.

O altă motivaţie este de a vedea perechea ($Procesor$, $Co - procesor$) sau ($Procesor$, $Lifo$) într-un ansamblu integrat de procesor cu performanţe superioare extins printr-un ciclu folosind un circuit RAM.

Dar, în principal aceste noi maşini rămân structurate pe funcţiile unui computer.

Funcţia unui calculator necesită cel puţin 4 cicluri, dar poate fi implementată pe orice sistem cu mai multe cicluri.

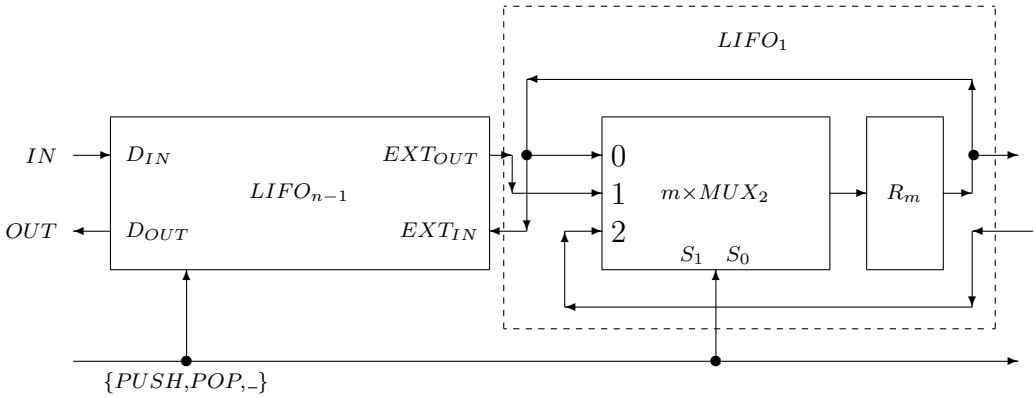
9.2 Stive organizate ca $n-DS$

În structura unui computer pot fi întâlnite anumite structuri hibride implementate ca sisteme digitale cu mai multe cicluri.

O astfel de structură este cea de stivă (care poate fi realizată prin cel puţin două cicluri).

Există diverse soluţii pentru implementarea funcţiei de stivă ca memorie $LIFO$ (last-in-first-out).

O variantă de constructie se bazează pe o definiţie recursivă simplă:



Definiţia 9.1 *O stivă $LIFO_n$ pe n nivele poate fi construită prin extensie serială a unui $LIFO_{n-1}$ cu un $LIFO_1$ (vezi Capitolul 7). Stiva de nivel 1 este un registru R_m conectat în ciclu cu un $m \times MUX_2$, ai cărui selectori asigură secvenţa de control definită:*

- **NO OP***: când $S_1S_0 = 00$ – conţinutul registrului nu se modifică;*
- **PUSH***: $S_1S_0 = 01$ – registrul este încărcat cu valoarea de intrare IN;*
- **POP***: $S_1S_0 = 10$ – registrul este încărcat cu valoarea de intrare extinsă EXT_{IN}.*

Evident, $LIFO_n$ este un registru de deplasare serial - paralel.

Deoarece conţinutul unui astfel de registru poate fi deplasat în ambele direcţii, fiecare R_m este inclus în două feluri de cicluri:

- Prin propriul său multiplexor (pentru funcţia **NO OP**;
- Prin două $LIFO$ succesive.

Deci, $LIFO_1$ este un $2-DS$, $LIFO_2$ este un $3-DS$, ..., $LIFO_n$ este un $(n+1)-DS$.

Acesta este un exemplu de circuit implementat folosind un număr arbitrar de cicluri incluse unul în altul. Utilizarea lui este însă strict subsumată unui circuit de tip $4-DS$, adică unui computer.

Capitolul 10

Structura unui computer la nivel de performanţă

În arhitectura unul calculator, construcţiile componentelor la nivel de registru sau procesor se pretează mai puţin la o analiză formală (comparativ cu nivelul circuitelor, bazate pe porţi logice).

Motivul rezultă din dificultatea de a descrie precis comportarea sistemului dorit.

A spune că trebuie construit un calculator care să execute eficient toate programele este o afirmaţie mult prea vagă.

Drumul parcurs a fost preluarea unei structuri standard cu performanţe cunoscute şi modificarea ei pentru a o adapta noilor cerinţe.

Criteriile de performanţă ale unui calculator pot fi rezumate în următoarele cerinţe:

- Calculatorul să fie capabil să execute a instrucţiuni de tip b pe secundă.
- Calculatorul să fie capabil de a accesa c memorie sau dispozitive periferice de tip d.
- Calculatorul trebuie să fie compatibil cu calculatoare de tip e.
- Costul final al sistemului să nu depăşească f.

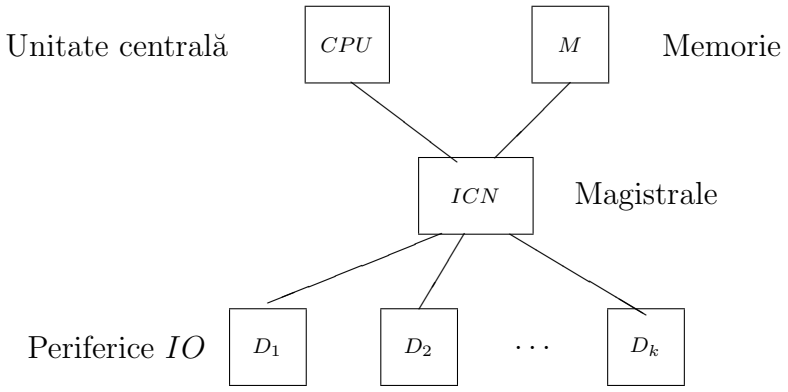
10.1 Structuri standard

Sistemele de calcul cele mai cunoscute[1] sunt calculatoarele dedicate aplicaţiilor generale, care diferă între ele mai ales prin numărul componentelor folosite şi prin gradul de autonomie.

Varietatea interconexiunilor sau structurilor de comunicaţie folosite este foarte restrânsă.

În linii mari, sistemele de calcul utilizate sunt structurate după una din schemele următoare:

1. Structura folosită de prima generaţie de computere şi de multe sisteme mici de microprocesoare actuale este:



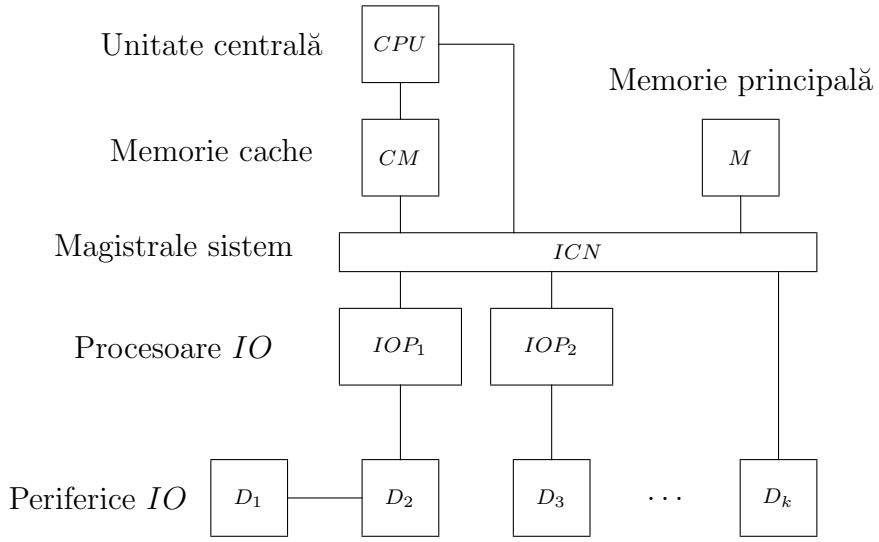
2. Adăugarea de procesoare specializate pentru structurile de intrare/ieşire (IO) este un proces tipic pentru calculatoare începând cu doua generaţie.

   O structură generală care subliniază gradul de complexitate al unor astfel de sisteme de calcul este prezentată în pagina următoare.

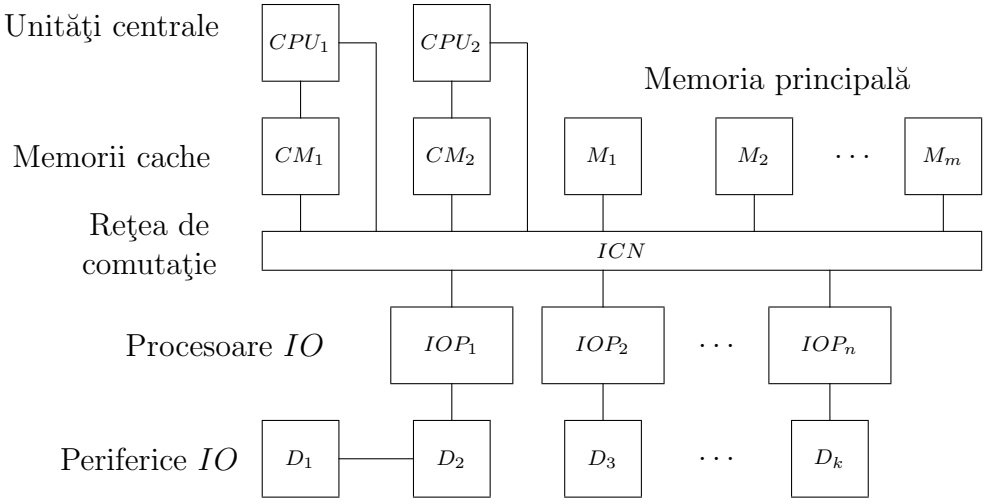
   Componenta de circuit notată ICN reprezintă o reţea de interconexiune care controlează comunicaţiile procesor - memorie.

   Aceste două variante de sisteme prezentate sunt de tip mono-procesor.

[1] Reamintim – cele mai utilizate sisteme de calcul sunt micro-controllerele (Capitolul 1).



3. O structură standard pentru un multi-procesor este descrisă mai jos (pentru două procesoare). Variantele anterioare sunt cazuri particulare ale acestuia.



4. Structuri mai complexe (de exemplu reţele de calculatoare) se pot obţine prin conectarea mai multor astfel de structuri.

10.2 Măsurarea performanţelor unui sistem

Majoritatea performanţelor unui calculator rezultă din caracteristicile unităţii sale centrale de prelucrare (CPU).

Viteza sa de lucru este măsurată uşor – dar grosier – prin frecvenţa f de ceas, dată în megaherţi. Pot fi daţi însă şi alţi indicatori de performanţă, cum ar fi:

1. *Viteza medie de execuţie a instrucţiunilor* ($MIPS$).

   Este numărul mediu de tacţi folosiţi de CPU pentru executarea unei instrucţiuni. Ea este legată de timpul mediu T (în microsecunde – μs) necesari pentru execuţia a N instrucţiuni, conform formulei

   $$T = \frac{N \times CPI}{f} \quad \mu s$$

   Deci timpul mediu t_E de execuţie al unei instrucţiuni este

   $$t_E = \frac{T}{N} = \frac{CPI}{f} \quad \mu s$$

   Dacă f depinde în principal de tehnologia circuitului integrat, folosită pentru implementarea CPU, CPI (numărul mediu de cicluri per instrucţiune) depinde în primul rând de arhitectura sistemului.

2. Pentru t_E se poate da şi o altă formulă, considerând distribuţia instrucţiunilor de diverse tipuri şi viteze în prelucrarea unor programe standard.

   Fie $I_1, I_2, \ldots, I_n$ un set de tipuri de instrucţiuni reprezentative.

   Notăm cu t_s timpul mediu de execuţie (în μs) al unei instrucţiuni de tipul I_s şi p_s probabilitatea de apariţie a instrucţiunilor de tipul I_s într-un cod obiect standard.

   Atunci timpul mediu de executare al unei instrucţiuni este dat de formula

   $$t_E = \sum_{i=1}^{n} p_i t_i \quad \mu s$$

   Valori pentru t_i se pot obţine uşor din specificaţiile CPU, dar valori cât mai corecte pentru p_i pot fi obţinute numai pe cale experimentală.

3. *Performanţe legate de memorie.*

   Mărimea memoriei principale şi a memoriei cache pot oferi indicaţii (destul de vagi) despre capacitatea sistemului.

Un parametru de memorie legat de viteza de calcul este *lăţimea de bandă*: rata maximă – în milioane de biţi per secundă (Mb/sec) la care informaţia poate fi transferată spre şi de la unitatea de memorie.

Lăţimea de bandă afectează performanţele CPU deoarece viteza de procesare este limitată în ultima instanţă de rata la care instrucţiunile şi datele pot fi accesate sau memorate.

4. O măsură a performanţei – des solicitată – este costul executării de către sistem a unui set de programe reprezentative.

   Acest cost poate fi timpul total de execuţie T (incluzând acţiunile legate de memorie, memorie cache, CPU, şi alte sisteme componente).

   Un set de programe reprezentative pentru un mediu de calcul particular poate fi utilizat pentru evaluarea performanţei întregului sistem.

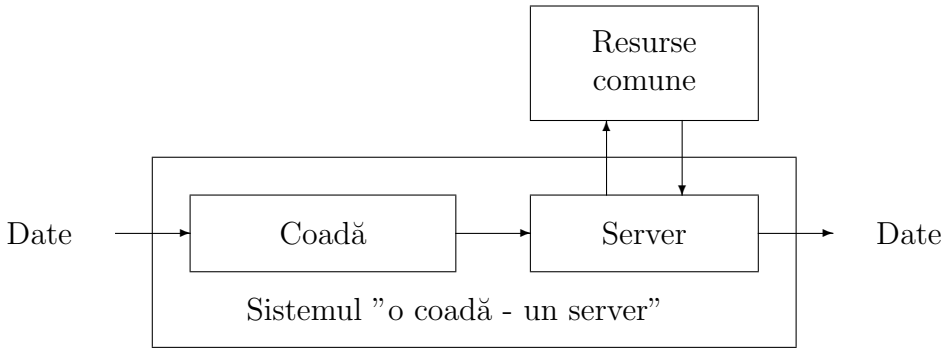
   Adesea setul de programe este un standard construit special pentru evaluare.

10.3 Modele bazate pe cozi

Pentru a sublinia importanţa modelării analizei performanţelor unui sistem de calcul, vom detalia o aproximare bazată pe teoria cozilor. Prezentarea este pur informală, deoarece necesită cunoştinţe destul de avansate de probabilităţi.

Modelul coadă – spre deosebire de stivă – este un model de tip FIFO[2].

Reprezentarea sa generală este dată de figura:



Este reprezentat un *server* (de exemplu un CPU sau un computer) cu un set de programe care trebuie executate.

[2]sau $M/M/1$ (din raţiuni istorice)

Sarcinile sunt activate sau sosesc la momente arbitrare şi sunt stocate în coadă, până ce sunt prelucrate de CPU.

Parmetrii cheie ai modelului sunt rata în care vin sarcinile şi rata de prelucrare; ambele sunt măsurate în *sarcini/sec*.

Mediile venirii sarcinilor şi media prelucării lor sunt notate cu λ respectiv μ.

Sosirea şi procesarea variază aleator şi aceste valori medii sunt determinate prin calcule statistice. Ultima va caracteriza comportamentul sistemului modelat.

Vom nota $\rho = \lambda/\mu$; acesta este *utilizarea medie* a serverului, adică fracţiunea de timp cât este ocupat (în medie).

De exemplu, dacă vin în medie două probleme pe secundă ($\lambda = 2$) şi serverul le poate procesa cu o rată medie de 8 probleme pe secundă ($\mu = 8$), atunci $\rho = 2/8 = 0.25$.

Sosirea problemelor în sistem este un proces aleator caracterizat de *distribuţia timpului dintre sosiri* $\rho_I(t)$: probabilitatea ca cel puţin o problemă să sosească într-o perioadă de timp de lungime t.

Cazul $M/M/1$ presupune o sosire de tip Poisson, unde probabilitatea de distribuţie este

$$\rho_I(t) = 1 - e^{-\lambda t}$$

Fie $p_S(t)$ probabilitatea ca serviciul solicitat de o problemă să fie realizat de CPU într-un timp maxim t după eliminarea ei din coadă.

Procesul de servire este modelat de o formulă Poisson similară

$$p_S(t) = 1 - e^{-\mu t}$$

Pentru caracterizarea performanţelor unui sistem cu o singură coadă pot fi folosiţi diverşi parametri.

De exemplu:

- Utilizarea lui ρ, care dă fracţiunea medie de timp cât serverul este ocupat.
- Numărul mediu de probleme aflate în coadă (atât cele care aşteaptă prelucrarea cât şi cele care sunt în faza de procesare).

  Parametrul este numit *lungime medie a cozii* şi este dat de formula (Robertazzi - 1994)

  $$l_Q = \frac{\rho}{1 - \rho}$$

- Timpul mediu în care problemele aşteaptă prelucrarea precum şi timpul de prelucrare; este numit *timpul mediu de aşteptare* t_Q.

Variabilele l_Q şi t_Q sunt legate direct prin formula $l_Q = \lambda t_Q$ sau

$$t_Q = \frac{l_Q}{\lambda}$$

Această ecuaţie se numeşte *ecuaţia lui Little* şi este valabilă pentru toate tipurile de sisteme cu coadă, nu numai modelul $M/M/1$.

Din combinarea ultimelor două relaţii se obţine

$$t_Q = \frac{1}{\mu - \lambda}$$

Valorile L_Q şi t_Q se referă atât la probleme aflate în stadiul de aşteptare cât şi la cele care sunt în proces de prelucrare.

Numărul mediu de probleme care aşteaptă în coadă dar nu sunt încă procesate se notează cu l_w, iar t_w este timpul mediu de aşteptare în coadă (exclusiv timpul de prelucrare).

Utilizarea medie a unui server în sistemul $M/M/1$ este numărul mediu de probleme aflate în stadiu de prelucrare, adică λ/μ.

Deci săzând acest număr din l_Q se obţine l_w:

$$l_w = l_Q - \rho = \frac{\lambda^2}{\mu(\mu - \lambda)}$$

Similar

$$t_w = t_Q - \frac{1}{\mu} = \frac{\lambda}{\mu(\mu - \lambda)}$$

unde $1/\mu$ este timpul mediu luat de prelucrarea unei probleme.

Comparând aceste ultime două relaţii se obţine $t_w = l_w/\lambda$; deci ecuaţia Little are loc şi pentru indicele w de aşteptare.

Exemplul 10.1 *O mică societate are un calculator cu un singur terminal folosit de salariaţi.*

Cam 10 *persoane folosesc terminalul în timpul celor* 8 *ore de lucru, iar fiecăruia îi vine rândul circa* 30 *minute, adesea pentru probleme simple de rutină.*

Managerul societăţii consideră că acest calculator este sub-utilizat, deoarece este defect în medie 3 *ore pe zi. Nu vrea să cumpere mai multe terminale, deoarece este convins că salariaţii le vor folosi pentru jocuri sau alte probleme personale.*

Aceştia – la rândul lor – consideră că terminalul este supra-utilizat deoarece sunt nevoiţi să aştepte cel puţin o oră fiecare pentru a avea acces la el.

Ei cer conducerii să mai cumpere terminale şi să construiască o reţea.

Să studiem această problemă folosind teoria cozilor, printr-un model $M/M/1$.

Pentru că în medie sunt 10 *utilizatori în cele* 8 *ore de lucru, vom nota* $\lambda = 10/8$ *utilizatori/oră* $= 0.0208$ *utilizatori/minut.*

Calculatorul este ocupat în medie 5 *din cele* 8 *ore; deci utilizarea* $\rho = 5/8$ *implică* $\mu = 1/30 = 0.0333$.

Înlocuind aceste valori în ultima ecuaţie, se obţine $t_w = 50$ *minute, ceea ce confirmă estimarea salariaţilor privind timpul lor mediu de acces la computer.*

Directorul societăţii este astfel convins că societatea are nevoie de terminale noi şi acceptă să cumpere destule astfel ca timpul de aşteptare t_w *să fie redus de la* 50 *la* 10 *minute.*

Apare astfel o întrebare: Câte terminale noi va trebui să conţină reţeaua ?

Pentru a studia acest aspect, vom modela fiecare terminal împreună cu utilizatorii săi printr-un sistem $M/M/1$ *independent.*

Fie m *numărul minim de posturi de lucru necesare pentru a reduce* $t_w < 10$, *sau – echivalent –* $t_Q < 40$.

Putem presupune că venirea utilizatorilor la terminale poate fi separată în m *cozi.*

Rata de sosire λ^* *per terminal este* $\lambda/m = 0.0208/m$ *utilizatori/minut.*

Dacă, aşa cum s-a presupus, lucrările sunt de rutină, unitatea CPU *nu este afectată de noile componente* IO, *aşa că timpul de răspuns nu se modifică substanţial.*

De aceea vom lua acelaşi timp mediu pentru rata de prelucrare $\mu^* = \mu = 0.0333$ *utilizatori/minut.*

Solicitarea performanţei cerute duce la relaţia

$$t_Q^* = \frac{1}{\mu^* - \lambda^*} = \frac{1}{\mu - \frac{\lambda}{m}} < 40$$

de unde rezultă $m > 2.5$.

Deci sunt necesare trei posturi de lucru, asă că se face comandă pentru două terminale noi.

De remarcat că modelul este unul pesimist, în care salariaţii se împart în trei cozi distincte, fără legătură între ele, deci fără posibilitatea de a trece de la o coadă la alta când de exemplu unul din periferice a devenit liber.

Un studiu mai complex duce la un rezultat apropiat pentru m; *una din valorile* $m = 2$ *sau* $m = 3$.

Soluţii şi indicaţii la problemele propuse

Capitolul 2:

1.
$18926_{10} = 100100111101110_2$
$-7772_{10} = -1111001011100_2$
$7863,15_{10} = 1111010110111,001(0011)_2$
$0,7_{10} = 0,1(0110)_2$

2. Cele două numere sunt: 55462 şi respectiv

$$2217 + \frac{1}{8} + \frac{1}{16} + \frac{1}{64} + \frac{1}{128} + \frac{1}{1024} = 2217 + \frac{219}{1024} = 2217,211914\ldots$$

3. Vom lucra cu reprezentarea binară pe 16 biţi (primul bit fiind bit de semn).

   (a) $11_{10} = 0000000000001011_2$, $\quad 32600 = 0111111101011000_2$.
   Suma bit cu bit a celor doua numere dă 0111111101100011.
   Pentru verificare, acest rezultat este reprezentarea binară a lui $32611 = 11 + 32600$.

   (b) $599_{10} = 0000001001010111_2$.
   Numărul $10692 = 0010100111000100_2$ trebuie trecut în cod complementar. Complementând biţii şi adunând apoi valoarea 1 se obţine 1101011000111100.
   Adunăm cele două numere:

$$0000001001010111 + 1101011000111100 = 1101100010010011$$

   Pentru verificare: rezultatul fiind negativ (bitul de semn este 1), valoarea sa absolută este complementul la 2; după ce scădem 1 şi complementăm biţii, se obţine $0010011101101101 = 10093_{10}$.

(c) $9862_{10} = 0010011010000110_2$, $20043_{10} = 0100111001001011$,
$-18552_{10} = 1011011110001000$ (cod complementar).

0010011010000110+0100111001001011+1011011110001000 = 0010110001011001

(bitul final de transport s-a ignorat), valoare binară ce corespunde numărului 11353.

4. Pentru $x - y$ se efectuează următorii paşi:
   (a) Se transformă y în cod invers;
   (b) Se adună x cu această valoare şi se ignoră bitul final de transport;
   (c) Dacă rezultatul este negativ, se trece în cod invers;
   (d) Dacă rezultatul este pozitiv, se adună 1.

5. $6489_{10} = 1100101011001_2 = 1,100101011001 \cdot 2^{12}$
   $-0,1_{10} = -0,00011(0011)_2 = -1,1(0011) \cdot 2^{-4}$
   $\frac{85}{256} = 2^{-2} + 2^{-4} + 2^{-6} + 2^{-8} = 0,01010101_2 = 1,010101 \cdot 2^{-2}$

6. Pentru primul număr: $C = 12 + 127 = 139_{10} = 10001011_2$, $f = 100101011001$. Deci reprezentarea lui 6489 ca număr real pe 32 biţi este

   0100 0101 1100 1010 1100 1000 0000 0000

   Pentru uşurinţă, aceste reprezentări se scriu în hexazecimal. Sub această formă, numărul este 45 CA $C8$ 00.

   Pentru al doilea număr: $C = -4 + 127 = 123_{10} = 01111011_2$,
   $f = 10011001100110011001100$. Deci reprezentarea lui $-0,1$ ca număr real pe 32 biţi este

   1011 1101 1100 1100 1100 1100 1100 1100

   sau – în hexazecimal: BD CC CC CC.

   În mod similar se rezolvă şi restul exerciţiului.

7. $2^{-109}(2+2^{-2}+2^{-3}+2^{-5}+2^{-7}+2^{-8}+2^{-10}+2^{-14}+2^{-15}+2^{-17}+2^{-19}+2^{-21}+2^{-23}+2^{-24})$.

Capitolul 3:

1. Aplicăm pe rând axiomele 7, 9 şi 6 din Definiţia algebrelor booleene şi obţinem

$$a \vee (\overline{a} \wedge b) = (a \vee \overline{a}) \wedge (a \vee b) = 1 \wedge (a \vee b) = a \vee b$$

A doua relaţie se demonstrează analog.

2. Cele trei axiome ale relaţiei de ordine (reflexivitate, antisimetrie, tranzitivitate) se verifică imediat. Se pot scrie echivalenţele

$\overline{b} \leq \overline{a} \iff \overline{a} = \overline{a} \vee \overline{b} \iff a = \overline{\overline{a}} = \overline{\overline{a} \vee \overline{b}} = a \wedge b$

Atunci $a \vee b = (a \wedge b) \vee b = b$ deci $a \leq b$.

3. $a \leq b \iff a \vee b = b \iff a \wedge \overline{b} = a \wedge (\overline{a \vee b}) = a \wedge (\overline{a} \wedge \overline{b}) = (a \wedge \overline{a}) \wedge \overline{b} = 0$

$a \wedge \overline{b} = 0 \iff \overline{a \wedge \overline{b}} = 1 \iff \overline{a} \vee b = 1$

4. $a \wedge b = 0 \iff \overline{\overline{a}} \wedge b = 0 \iff b \wedge \overline{\overline{a}} = 0 \iff b \leq \overline{a}$

5. Primele patru proprietăţi din Propoziţia 3.1 se verifică prin calcul simplu.

Proprietatea 5: $a \oplus b = c \implies a \oplus a \oplus b = a \oplus c \implies 0 \oplus b = a \oplus c \implies b = a \oplus c$

Proprietatea 6: $a \oplus b \oplus c = (a \oplus b) \oplus c = c \oplus c = 0$

Proprietatea 7: $(a \wedge b) \oplus (a \wedge c) = [(a \wedge b) \wedge (\overline{a \wedge c})] \vee [(\overline{a \wedge b}) \wedge (a \wedge c)] = [(\overline{a} \vee \overline{b}) \wedge (a \wedge c)] \vee [(a \wedge b) \wedge (\overline{a} \vee \overline{c})] = [(\overline{a} \wedge a \wedge c) \vee (\overline{b} \wedge a \wedge c)] \vee [(\overline{a} \wedge a \wedge b) \vee (\overline{c} \wedge a \wedge b)] = (a \wedge \overline{b} \wedge c) \vee (a \wedge b \wedge \overline{c}) = a \wedge [(\overline{b} \wedge c) \vee (b \wedge \overline{c})] = a \wedge (b \oplus c)$

Propoziţia 3.2:

$(1): \quad \overline{a \oplus b} = \overline{(a \wedge \overline{b}) \vee (\overline{a} \wedge b)} = (\overline{a \wedge \overline{b}}) \wedge (\overline{\overline{a} \wedge b}) = (\overline{a} \vee b) \wedge (a \vee \overline{b}) = a \sim b$

$(2): \quad \overline{a} \sim \overline{b} = (\overline{a} \vee \overline{\overline{b}}) \wedge (\overline{\overline{a}} \vee \overline{b}) = (\overline{a} \vee b) \wedge (a \vee \overline{b}) = a \sim b$

Propoziţia 3.3: Implicaţiile $(3) \implies (1)$, $(3) \implies (2)$ sunt banale.

$(1) \implies (2): \quad a \sim b = \overline{a \oplus b} = \overline{0} = 1$

$(2) \implies (3):$ Presupunem $(a \vee \overline{b}) \wedge (\overline{a} \vee b) = 1$. De aici se obţine (exerciţiu) $\overline{a} \vee b = 1$ şi $a \vee \overline{b} = 1$ deci (Exerciţiul 3) $a \leq b$ şi $b \leq a \implies a = b$.

6. Tabela este construită pe pagina următoare.

7. -

	f_{15}	f_{14}	f_{13}	f_{12}	f_{11}	f_{10}	f_9	f_8	f_7	f_6	f_5	f_4	f_3	f_2	f_1	f_0
f_0	4	3	3	2	3	2	2	1	3	2	2	1	2	1	1	0
f_1	3	4	2	3	2	3	1	2	2	3	1	2	1	2	0	
f_2	3	2	4	3	2	1	3	2	2	1	3	2	1	0		
f_3	2	3	3	4	1	2	2	3	1	2	2	3	0			
f_4	3	2	2	1	4	3	3	2	2	1	1	0				
f_5	2	3	1	2	3	4	2	3	1	2	0					
f_6	2	1	3	2	3	2	4	3	1	0						
f_7	1	2	2	3	2	3	3	4	0							
f_8	3	2	2	1	2	1	1	0								
f_9	2	3	1	2	1	2	0									
f_{10}	2	1	3	2	1	0										
f_{11}	1	2	2	3	0											
f_{12}	2	1	1	0												
f_{13}	1	2	0													
f_{14}	1	0														
f_{15}	0															

8. Cele două operaţii sunt Δ (diferenţa simetrică) şi intersecţia.

9. Similar demonstraţiei din Teorema 3.10, se consideră $x_n = 0$ şi $x_n = 1$.

10. -

11. Ambele expresii booleene se reduc la $\overline{c}$.

12. Ambele expresii se reduc la $a \vee b$.

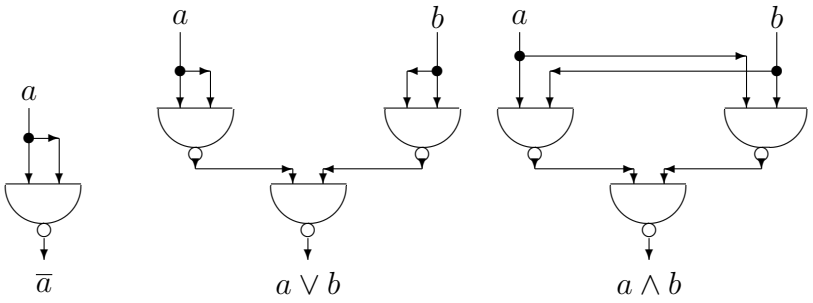
    Cele două forme normale sunt: $(a + b + c)(a + b + \overline{c})$ (normal conjunctivă) şi $abc + \overline{a}bc + a\overline{b}c + ab\overline{c} + \overline{a}b\overline{c} + a\overline{b}\,\overline{c}$ (normal disjunctivă).

Capitolul 4:

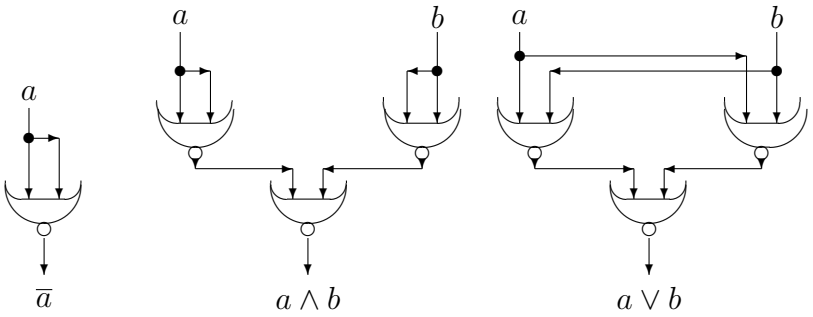
1. Notând cu $||$ operatorul Sheffer ($NAND$), principalele operaţii booleene se definesc astfel:

$$\overline{a} = a||a, \qquad a \vee b = (a||a)||(a||a), \qquad a \wedge b = (a||b)||(a||b)$$

   Circuitele sunt:



2. Similar cu exerciţiul precedent:



3.

a	0	0	0	0	1	1	1	1
b	0	0	1	1	0	0	1	1
c	0	1	0	1	0	1	0	1
x	0	1	0	1	0	1	1	1
y	1	0	1	0	0	0	1	0

4. Din tabelul de valori se determină imediat forma normal disjunctivă:

$$f(a,b,c) = \overline{a}\ \overline{b}c + \overline{a}b\overline{c} + \overline{a}bc + ab\overline{c} + abc$$

Se poate construi un circuit direct pentru această funcţie.

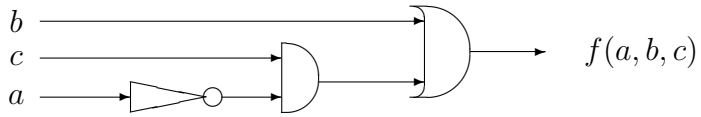
Altă variantă este de a simplifica întâi funcţia, după care se construieşte un circuit corespunzător acestei noi reprezentări.

Nu există un algoritm determinist pentru simplificarea unei funcţii booleene. În consecinţă, forma găsită prin simplificare nu este unică.

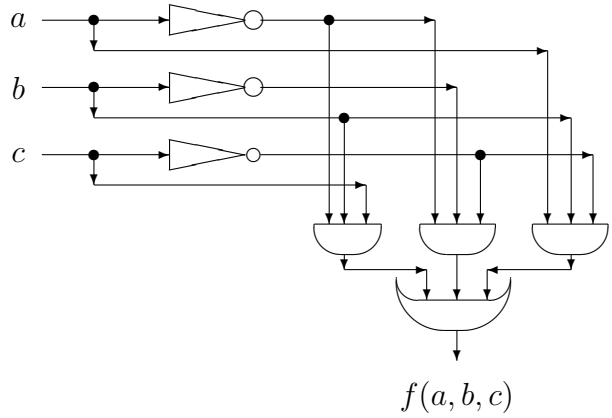
Pentru funcţia din acest exerciţiu, o modalitate de reducere este:

$$f(a,b,c) = \overline{a}c(\overline{b}+b) + (\overline{a}+a)b\overline{c} + (\overline{a}+a)bc = \overline{a}c + b\overline{c} + bc = \overline{a}c + b(\overline{c}+c) = \overline{a}c + b$$

Circuitul:



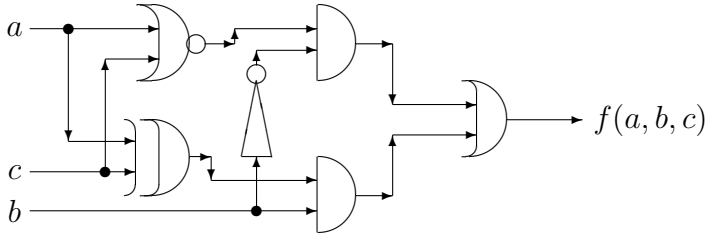
5. Un circuit bazat direct pe forma normală a funcţiei este:



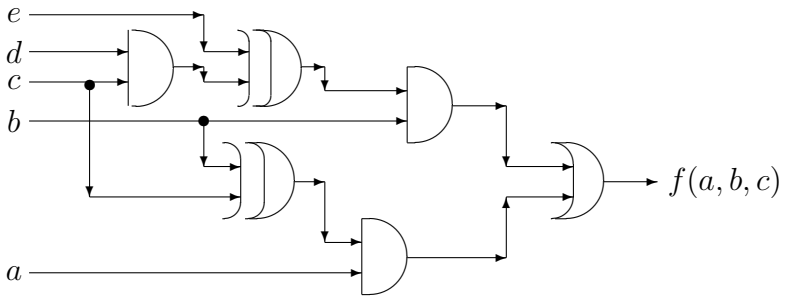
O variantă de simplificare este:

$$f(a,b,c) = b(\overline{a}c + a\overline{c}) + \overline{b}(\overline{a \vee c}) = b(a \oplus c) + \overline{b}(a\ NOR\ c)$$

care conduce la circuitul



6. Circuitul următor rezultă direct din forma funcţiei:



7. $f(a,b,c) = b(ac + \overline{a}\,\overline{c}) = b(\overline{a \oplus c}) = g(a,b,c)$

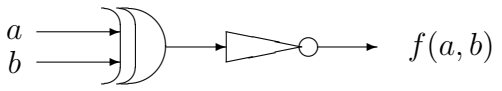
   Altă demonstraţie posibilă este prin construirea tabelelor de valori pentru cele două funcţii şi observaţia că acestea coincid.

8. $f(a,b,c) = (\overline{a \vee b})c = \overline{a}\,\overline{b}\,\overline{c}$

   Această este forma normal disjunctivă. Forma normal conjunctivă este

   $$f(a,b,c) = (a+b+c)(a+\overline{b}+c)(\overline{a}+b+c)(\overline{a}+\overline{b}+c)(a+\overline{b}+\overline{c})(\overline{a}+b+\overline{c})(\overline{a}+\overline{b}+\overline{c})$$

9. Funcţia se poate scrie $f(a,b) = \overline{a \oplus b}$.



Capitolul 5:

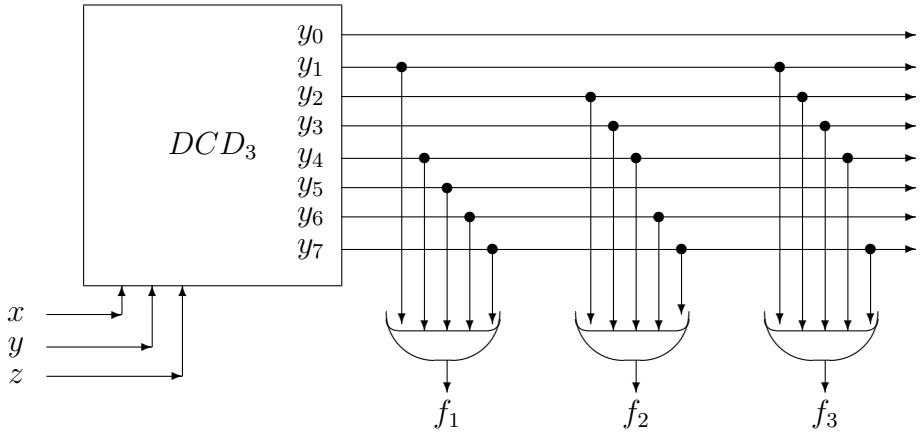
1. Se aduc cele trei expresii ale funcţiei $f(x,y,z) = (f_1(x,y,z),\ f_2(x,y,z),\ f_3(x,y,z))$ la forma normal disjunctivă:

   $f_1(x,y,z) = xyz + x\overline{y}z + xy\overline{z} + x\overline{y}\,\overline{z} + \overline{x}\,\overline{y}z,$

   $f_2(x,y,z) = xyz + \overline{x}yz + xy\overline{z} + \overline{x}y\overline{z} + x\overline{y}\,\overline{z},$

   $f_3(x,y,z) = xyz + \overline{x}yz + x\overline{y}z + \overline{x}y\,\overline{z} + \overline{x}\,\overline{y}z.$

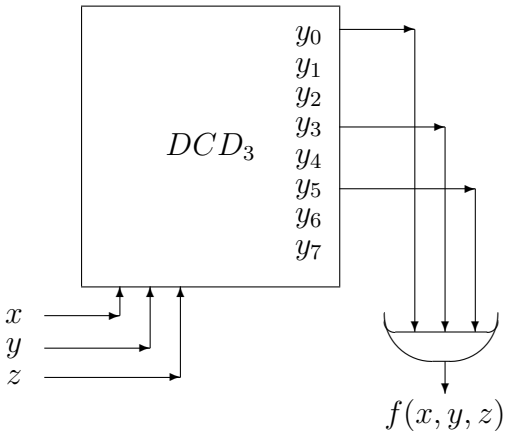
   Circuitul codificator este:



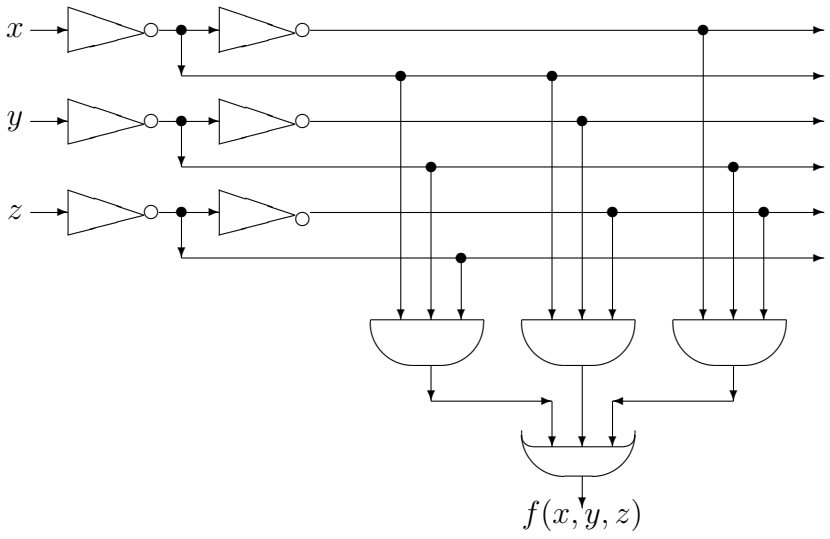
2. Tabelul de valori al funcţiei este:

x	0	0	0	0	1	1	1	1
y	0	0	1	1	0	0	1	1
z	0	1	0	1	0	1	0	1
$f(x,y,z)$	1	0	0	1	0	1	0	0

De aici rezultă forma normal disjunctivă $f(x,y,z) = \overline{x}\ \overline{y}\ \overline{z} + \overline{x}yz + x\overline{y}z$. Circuitul codificator este:



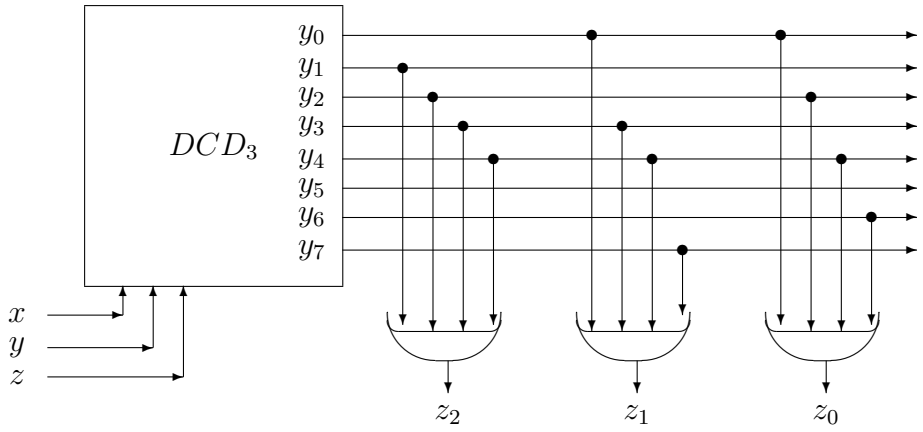
sau – dacă detaliem şi decodificatorul:



3. Funcţia asociată este $f : \{0,1\}^3 \longrightarrow \{0,1\}^3$ cu tabelul

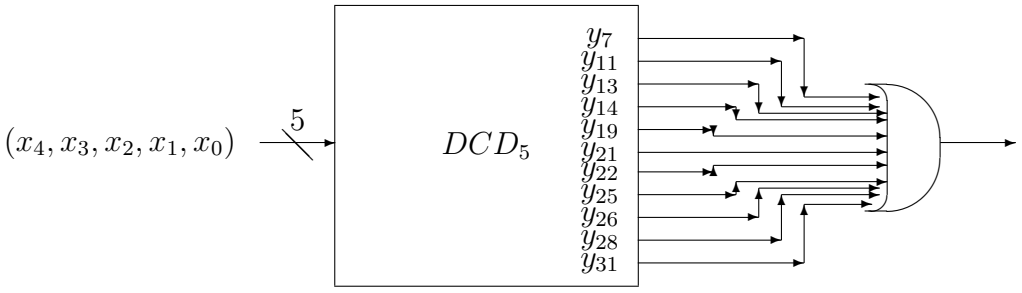
x_2	0	0	0	0	1	1	1	1
x_1	0	0	1	1	0	0	1	1
x_0	0	1	0	1	0	1	0	1
z_2	0	1	1	1	1	0	0	0
z_1	1	0	0	1	1	0	0	1
z_0	1	0	1	0	1	0	1	0

Codificatorul:



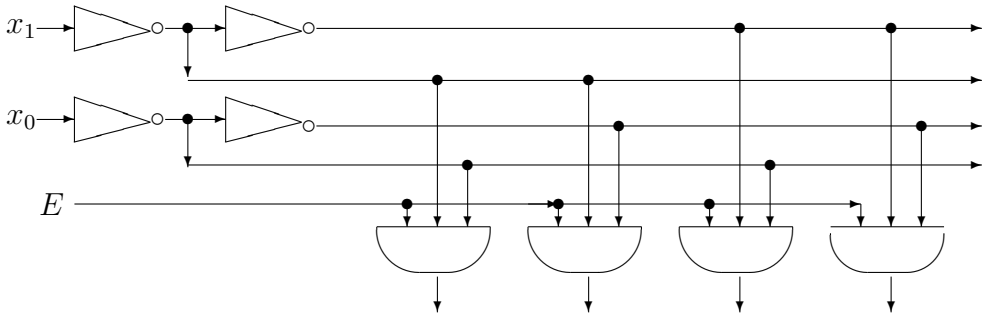
4. O funcţie booleană de cinci variabile care să implementeze condiţia trebuie să aibă $f(1,1,1,1,1) = f(1,1,1,0,0) = f(1,1,0,1,0) = f(1,1,0,0,1) = f(1,0,1,1,0) = f(1,0,1,0,1) = f(1,0,0,1,1) = f(0,1,1,1,0) = f(0,1,1,0,1) = f(0,1,0,1,1) = f(0,0,1,1,1) = 1$ şi 0 în rest.

O astfel de funcţie în forma normală se scrie imediat, din care reiese codificatorul (am trasat numai ieşirile utile din DCD_5, care intră în poarta OR_{11}:

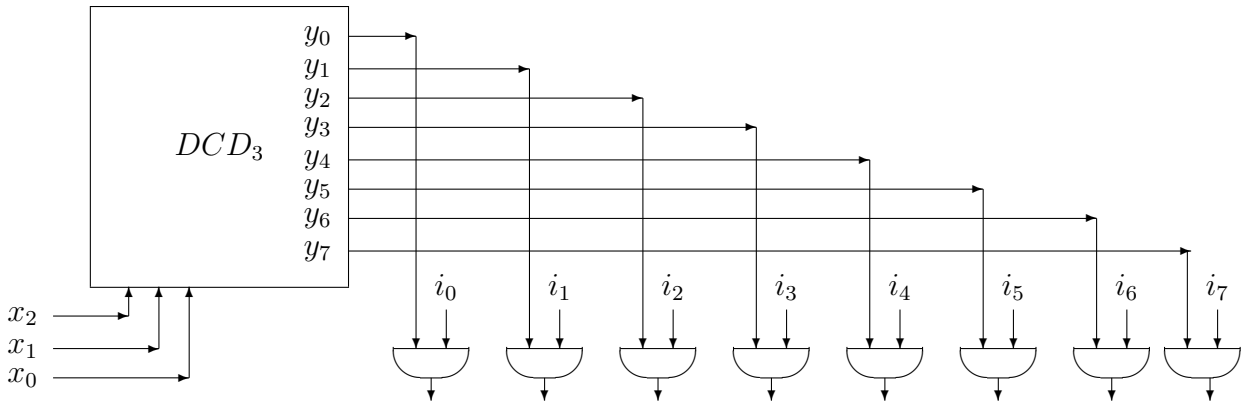


5. Mulţimea numerelor din intervalul $[0, 2^6-1]$ divizibile cu 4 este $A = \{0, 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60\}$ (în total 16). Se construieşte un DCD_6 în care ieşirile y_i cu $i \in A$ sunt conectate la o poartă OR_{16}, din care iese rezultatul final.

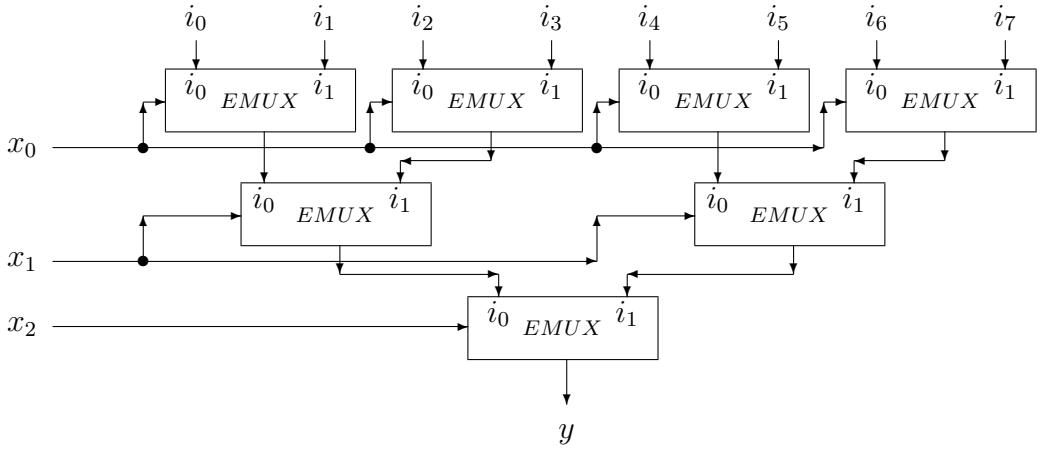
6. O simplă particularizare:



7. O construcţie directă este:



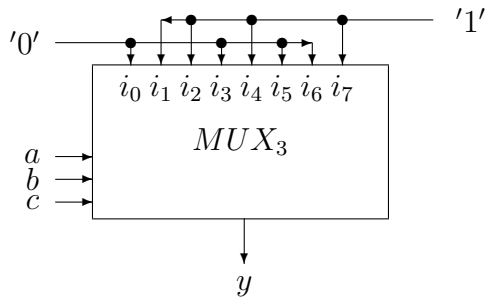
iar una recursivă:



8. Tabela de valori a funcţiei *sum* este

a	0	0	0	0	1	1	1	1
b	0	0	1	1	0	0	1	1
c	0	1	0	1	0	1	0	1
sum	0	1	1	0	1	0	0	1

Circuitul MUX_3 este imediat (eventual se poate detalia):



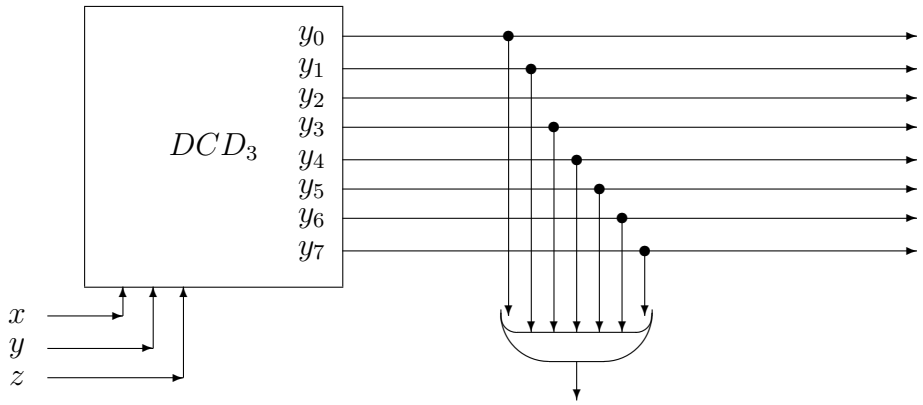
9. -

10. -

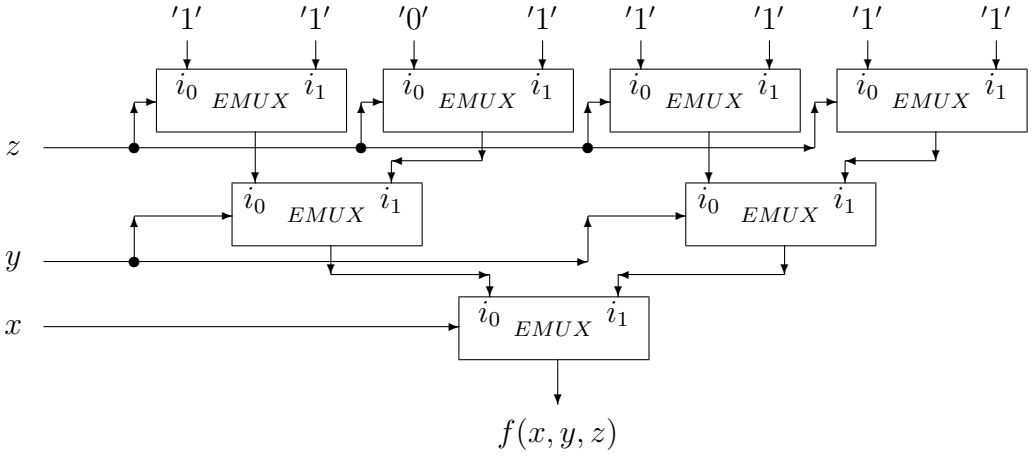
11. Tabela de valori a funcţiei f este

x	0	0	0	0	1	1	1	1
y	0	0	1	1	0	0	1	1
z	0	1	0	1	0	1	0	1
$f(x,y,z)$	1	1	0	1	1	1	1	1

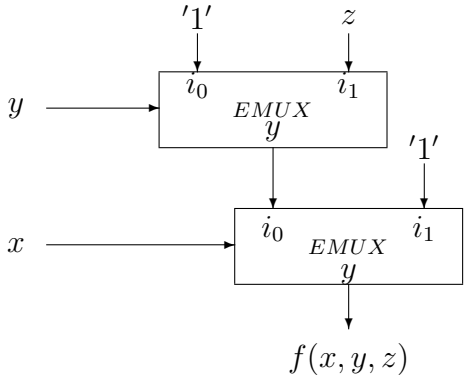
Un circuit codificator pentru implementare este:



Circuitul bazat pe multiplexori elementari este:



El se poate reduce spectaculos la:

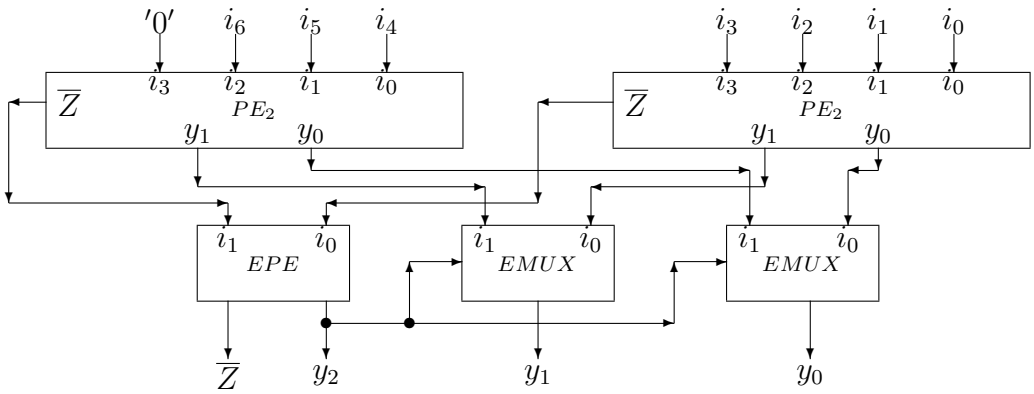


12. -

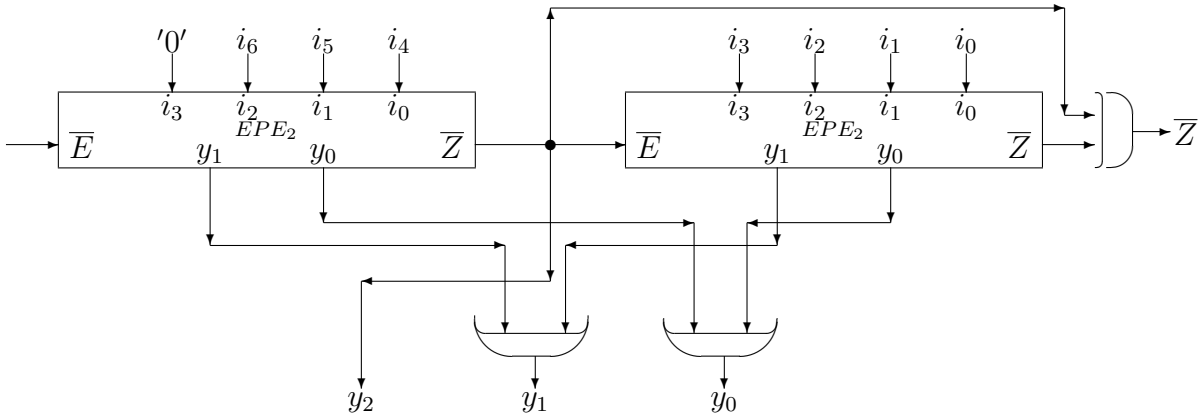
13. Tabela de valori obţinută este (am introdus şi rezultatele parţiale):

x_0	x_1	x_2	x_3	s_1	s_2	c_1	c_2	s	c
0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	1	0	0	1	0
0	0	1	0	0	1	0	0	1	0
0	0	1	1	0	0	0	1	1	0
0	1	0	0	1	0	0	0	1	0
0	1	0	1	1	1	0	0	0	1
0	1	1	0	1	1	0	0	0	1
0	1	1	1	1	0	0	1	0	1
1	0	0	0	1	0	0	0	1	0
1	0	0	1	1	1	0	0	0	1
1	0	1	0	1	1	0	0	0	1
1	0	1	1	1	0	0	1	0	1
1	1	0	0	0	0	1	0	0	1
1	1	0	1	0	1	1	0	0	1
1	1	1	0	0	1	1	0	0	1
1	1	1	1	0	0	1	1	1	0

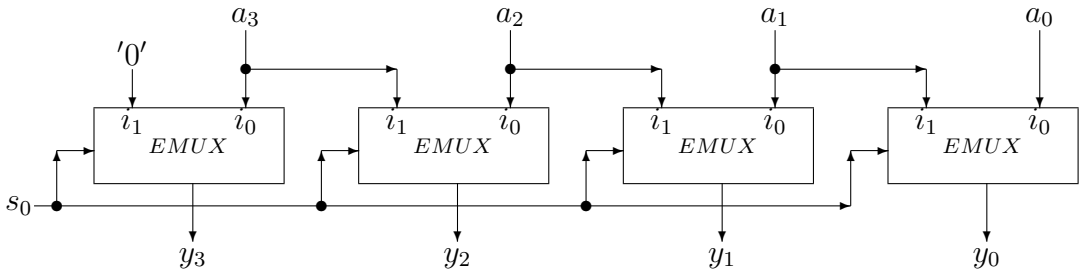
14. Un circuit bazat pe codificatori simpli (fără acces):



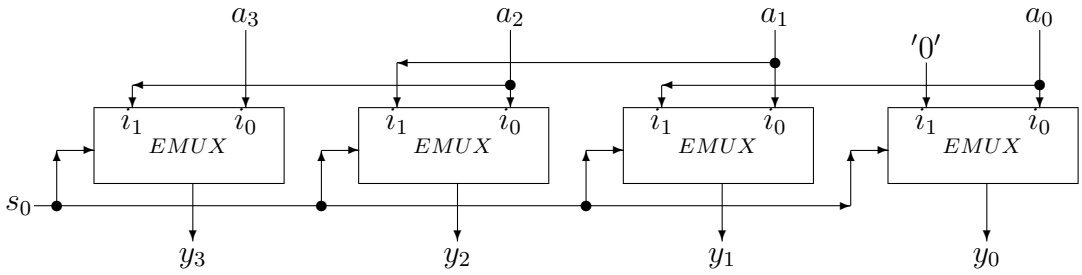
Dacă folosim construcţia recursivă bazată pe codificatori cu acces, avem:



15. Circuitul de deplasare spre dreapta cu 0 sau 1 poziţii este:



Similar se construieşte şi circuitul de deplasare spre stânga cu $0, 1$ poziţii:



16. Detaliem ecuaţia de transport $c_{i+1} = G_i \oplus P_i c_i$ pentru $i = 0, 1, 2, 3$, eliminând şi recursivitatea:

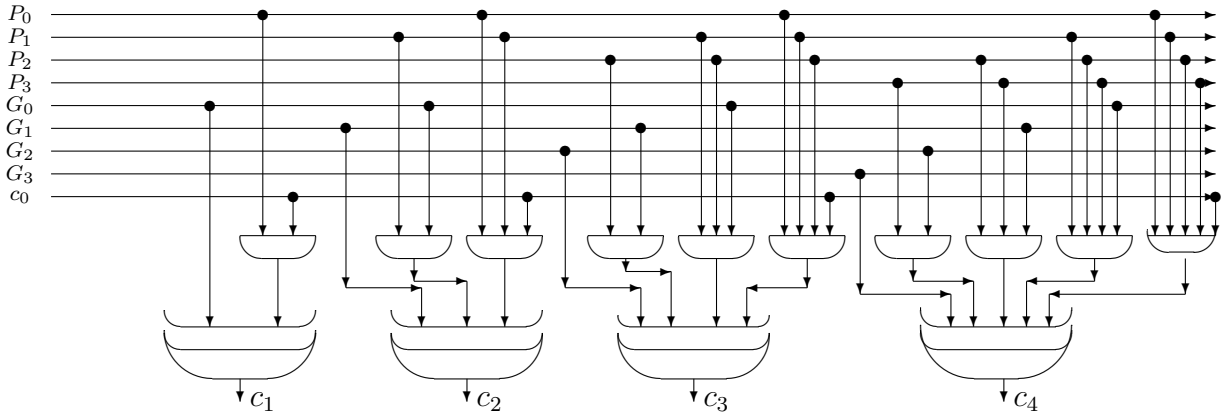
$c_1 = G_0 \oplus P_0 c_0$

$c_2 = G_1 \oplus P_1 c_1 = G_1 \oplus P_1 G_0 \oplus P_1 P_0 c_0$

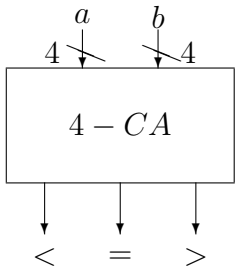
$c_3 = G_2 \oplus P_2 c_0 = G_2 \oplus P_2 G_1 \oplus P_2 P_1 G_0 \oplus P_2 P_1 P_0 c_0$

$c_4 = G_3 \oplus P_3 c_3 = G_3 \oplus P_3 G_2 \oplus P_3 P_2 G_1 \oplus P_3 P_2 P_1 G_0 \oplus P_3 P_2 P_1 P_0 c_0$

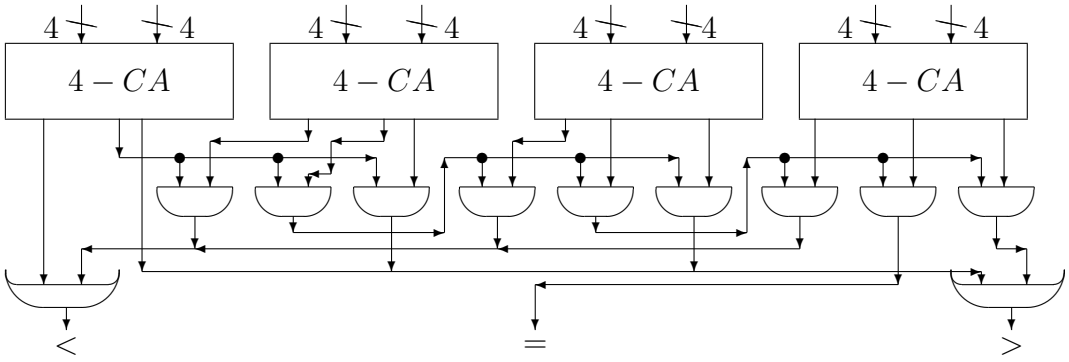
Un circuit posibil se construieşte imediat (nu este singura variantă).



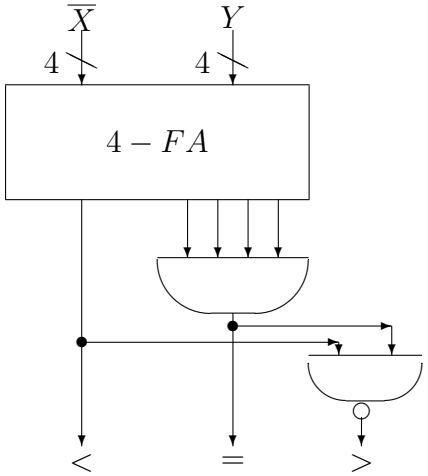
17. Construcţia unui comparator pe 4 biţi se realizează imediat particularizând structura din secţiunea 5.7. Dacă notăm



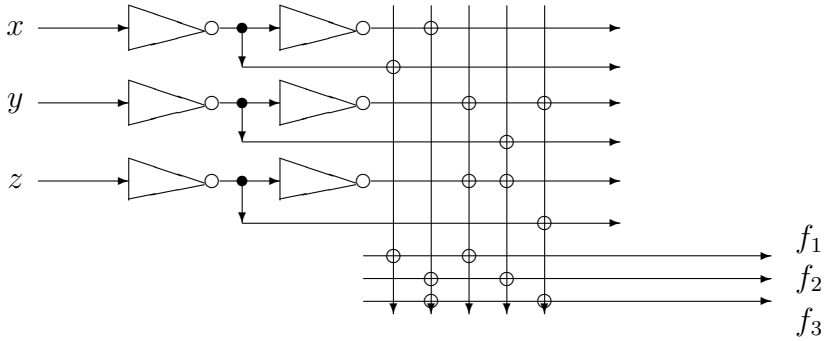
un circuit de comparare pe 4 biţi, el poate fi folosit ca bază pentru construirea unui circuit pe 16 biţi astfel:



18. Circuitul:

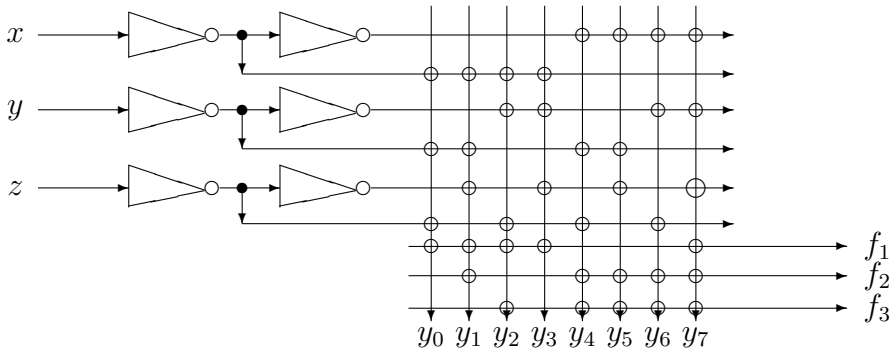


19. Un PLA pentru funcţie:



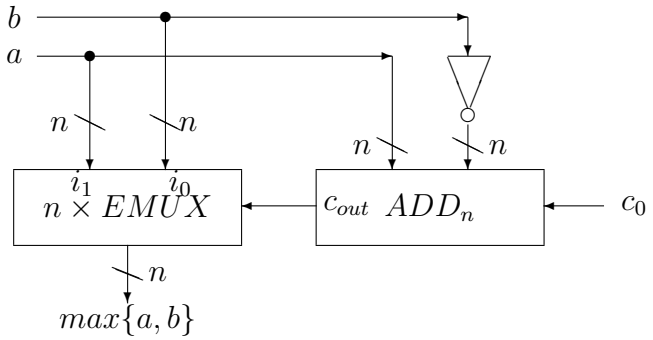
Pentru circuitul $PROM$ este nevoie de tabela de valori (sau – echivalent – de forma normal disjunctivă):

x	0	0	0	0	1	1	1	1
y	0	0	1	1	0	0	1	1
z	0	1	0	1	0	1	0	1
f_1	1	1	1	1	0	0	0	1
f_2	0	1	0	0	1	1	1	1
f_3	0	0	1	0	1	1	1	1



Capitolul 6:

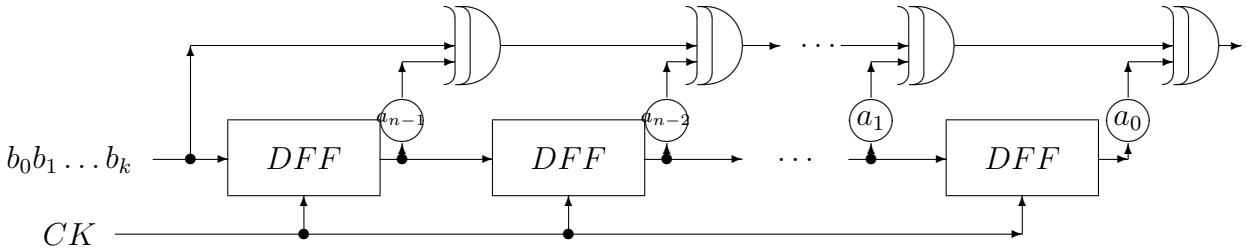
1. Se foloseşte faptul că $a > b$ este echivalent cu $a + \overline{b} > 11\ldots1$. Deci se testează bitul de semn al sumei $a + \overline{b}$. Acesta va fi selectorul a n multiplexori elementari, care triază între biţii numărului a şi cei ai numărului b.



2. Fără a restrânge generalitatea, putem considera $a_n = 1$.

   Circuitul este format dintr-un registru serial SR_n (format din n DFF-uri) şi cel mult $n-1$ porţi XOR.

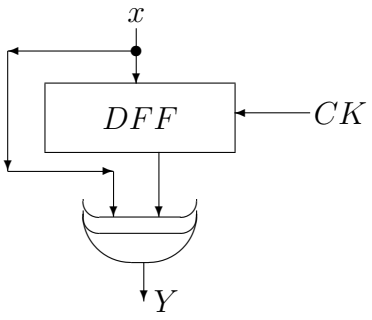
   O reprezentare a circuitului este



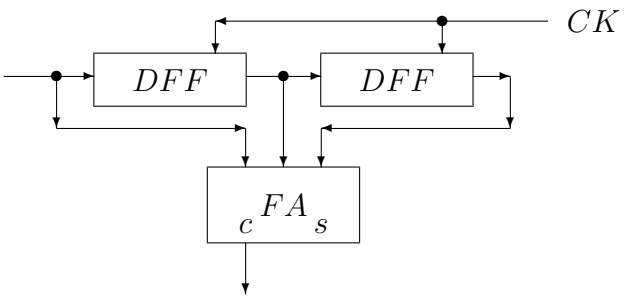
   Un arc marcat cu a_i este suspendat dacă $a_i = 0$ (iar poarta XOR in care intră acesta poate fi eliminată). Dacă $a_i = 1$, atunci acesta este un arc normal.

3. -

4. O soluţie posibilă este:



5. O soluţie posibilă este:



(din FA se reţine numai transportul, valoarea sumei nefiind necesară).

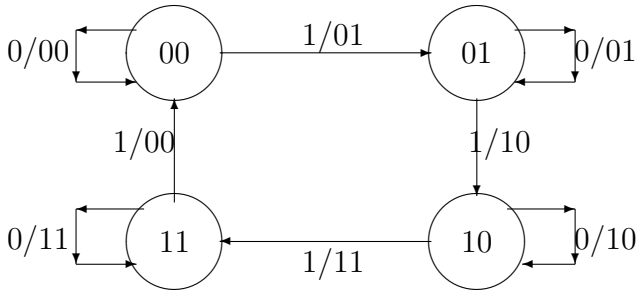
Capitolul 7:

1. Componentele din definiţie se construiesc uşor: $Q = \{0,1\}^2,\ X = \{0,1\},\ Y = \{0,1\}$

δ	0	1
00	00	01
01	01	10
10	10	11
11	11	00

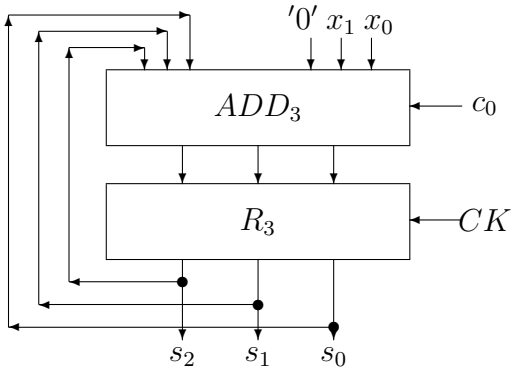
λ	0	1
00	00	01
01	01	10
10	10	11
11	11	00

(funcţiile de tranziţie şi de ieşire coincid). Graful de funcţionare este:



2. Se particularizează counterul din curs pentru $n = 3$, sau – eventual – se foloseşte exemplul de la sumatorul prefix.

3. Particularizând sumatorul prefix din 7.5.3, se obţine



Acest automat are $Q = \{0,1\}^3$, $X = \{0,1\}^2$, $Y = \{0,1\}^3$, iar funcţiile sunt $\lambda(abc, xyz) = abc$ pentru ieşire şi

δ	00	01	10	11
000	000	001	010	011
001	001	010	011	100
010	010	011	100	101
011	011	100	101	110
100	100	101	110	111
101	101	110	111	000
110	110	111	000	001
111	111	000	001	010

pentru transfer.

Graful de funcţionare se construieşte imediat.

4. Componentele automatului sunt: $Q = \{0,1\}$, $X = \{0,1\}^2$, $Y = \{0,1\}$,

δ	00	01	10	11
0	0	0	0	1
1	1	0	0	0

λ	00	01	10	11
0	0	1	1	0
1	1	0	0	1

Graful de funcţionare se construieşte imediat.

5. Componentele automatului sunt: $Q = \{0,1\}$, $X = \{0,1\}^2$, $Y = \{0,1\}$,

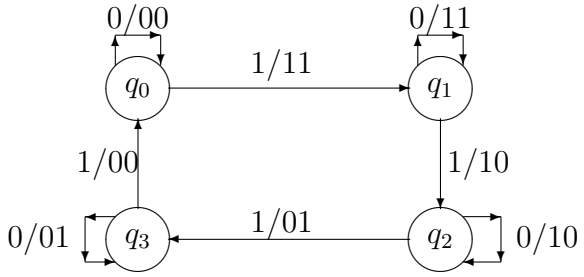
δ	00	01	10	11
0	0	0	0	1
1	1	1	1	0

λ	00	01	10	11
0	0	1	1	0
1	1	0	0	1

De remarcat că automatul diferă diferă de cel anterior numai prin funcţia de transfer.

Graful de funcţionare se construieşte imediat.

6. Completăm marcajele arcelor cu ieşirile:



Notăm stările q_i cu reprezentarea în baza 2 a lui i. Pentru cele patru stări sunt suficienţi doi biţi.

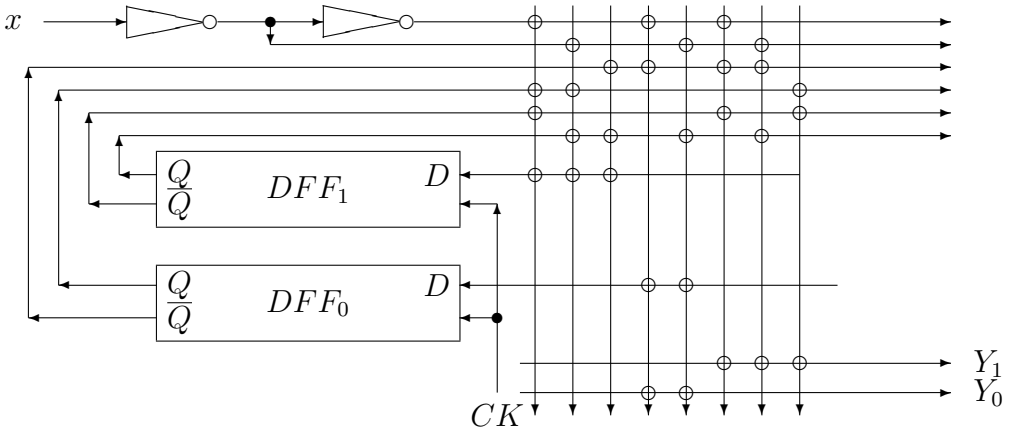
Deci: $X = \{0,1\},\ Q = Y = \{0,1\}^2$ şi (am notat cu x bitul de intrare curent)

δ	0	1	Q_1^+	Q_0^+
00	00	01	0	x
01	01	10	x	$\overline{x}$
10	10	11	1	x
11	11	00	$\overline{x}$	$\overline{x}$

λ	0	1	Y_1	Y_0
00	00	11	x	x
01	11	10	1	$\overline{x}$
10	10	01	$\overline{x}$	x
11	01	00	0	$\overline{x}$

De aici rezultă forma funcţiilor de tranziţie şi de ieşire:

$$\delta(Q_1Q_0, x) = (\overline{Q}_1Q_0x + Q_1Q_0\overline{x} + Q_1\overline{Q}_0,\ \overline{Q}_0x + Q_0\overline{x})$$
$$\lambda(Q_1Q_0, x) = (\overline{Q}_1\overline{Q}_0x + Q_1\overline{Q}_0\overline{x} + \overline{Q}_1Q_0,\ \overline{Q}_0x + Q_0\overline{x})$$

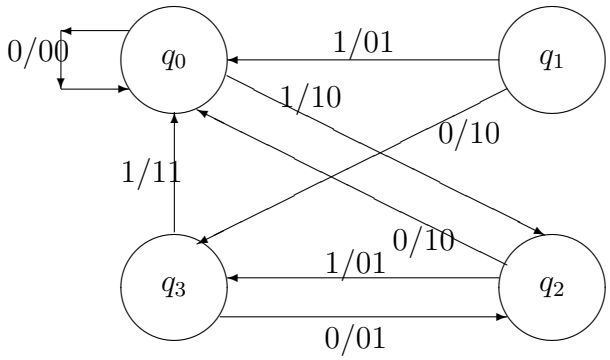


7. Din reprezentarea grafică rezultă $X = \{0,1\},\ Q = Y = \{0,1\}^2$ şi

δ	0	1
00	00	10
01	11	00
10	00	11
11	10	00

λ	0	1
00	00	10
01	10	01
10	10	01
11	01	11

Graful de tranziţie este



De aici (sau – eventual folosind un algoritm de eliminare a stărilor inaccesibile) – se observă că starea 01 poate fi eliminată, ea neputând fi accesibilă plecând din starea iniţială 00.

8. Reluăm tabelele celor două funcţii din exerciţiul anterior, completând cu coloanele care dau informaţii despre forma lor analitică:

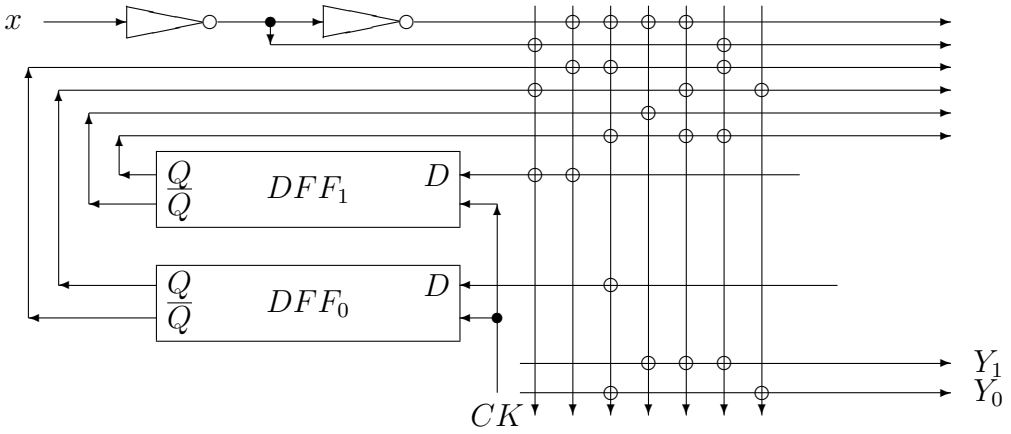
δ	0	1	Q_1^+	Q_0^+
00	00	10	x	0
01	$--$	$--$	$-$	$-$
10	00	11	x	x
11	10	00	$\overline{x}$	0

λ	0	1	Y_1	Y_0
00	00	10	x	0
01	$--$	$--$	$-$	$-$
10	10	01	$\overline{x}$	x
11	01	11	x	1

Deci putem da o formă mai simplă funcţiilor de tranziţie şi de ieşire:

$$\delta(Q_1Q_0, x) = (\overline{Q}_0 x + Q_0\overline{x},\ Q_1\overline{Q}_0 x),$$
$$\lambda(Q_1Q_0, x) = (\overline{Q}_1 x + Q_1Q_0 x + Q_1\overline{Q}_0\overline{x},\ Q_1\overline{Q}_0 x + Q_0)$$

O implementare bazată pe tablouri logic programabile este:



9. Separăm tabloul funcţiei de tranziţie pe cele două tablouri pentru δ_J respectiv δ_K:

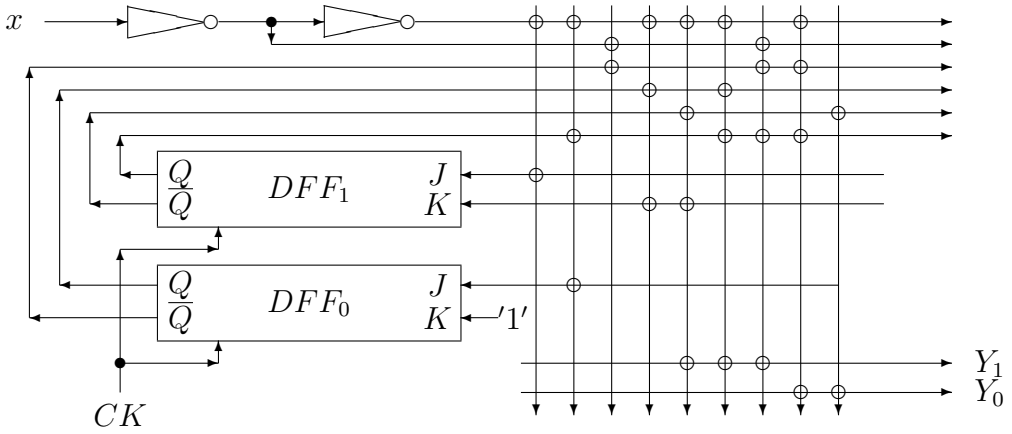
δ_J	0	1	J_1^+	J_0^+
00	00	10	x	0
01	$--$	$--$	$-$	$-$
10	-0	-1	$-$	x
11	$--$	$--$		$-$

δ_K	0	1	K_1^+	K_0^+
00	$--$	$--$	$-$	$-$
01	$--$	$--$	$-$	$-$
10	$1-$	$0-$	$\overline{x}$	$-$
11	01	11	x	1

O formă pentru cele două funcţii poate fi:

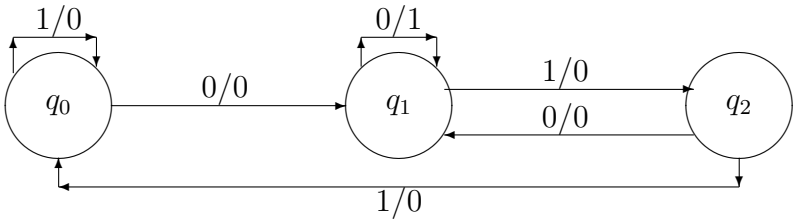
$$\delta_J(Q_1Q_2, x) = (x,\ Q_1x), \qquad \delta_K(Q_1Q_0, x) = (\overline{Q}_0\overline{x}, +Q_0x,\ '1')$$

Funcţia de ieşire rămâne neschimbată.



10. -

11. Folosind cunoştinţe de limbaje formale, se poate construi automatul care acceptă secvenţe binare marcând orice apariţie de două zerouri consecutive:



Ieşirea este legată apoi de un counter crescător pe 3 biţi (eventual cel construit la exerciţiul 1).

Din graful de tranziţie rezultă tabelele de valori ale celor două funcţii:

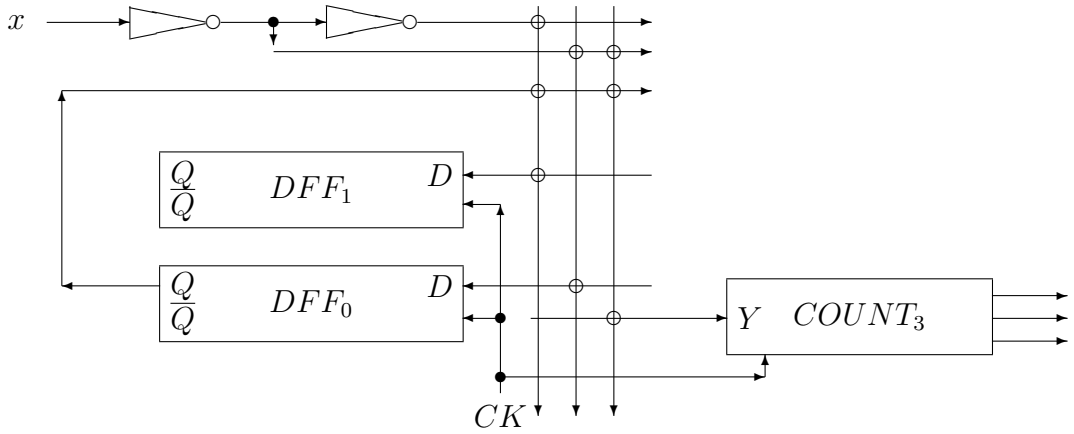
δ	0	1	Q_1^+	Q_0^+
00	01	00	0	$\overline{x}$
01	01	10	x	$\overline{x}$
10	01	00	0	$\overline{x}$

λ	0	1	Y^+
00	0	0	0
01	1	0	$\overline{x}$
10	0	0	0

Deci

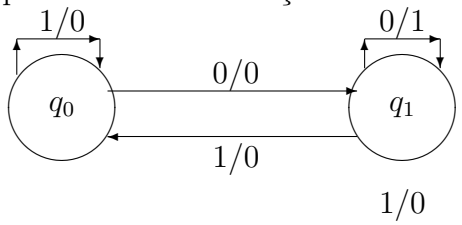
$$\delta(Q_1Q_0, x) = (Q_0x,\ \overline{x}), \qquad \lambda(Q_1Q_0, x) = Q_0\overline{x}$$

Un circuit care rezolvă problema este:



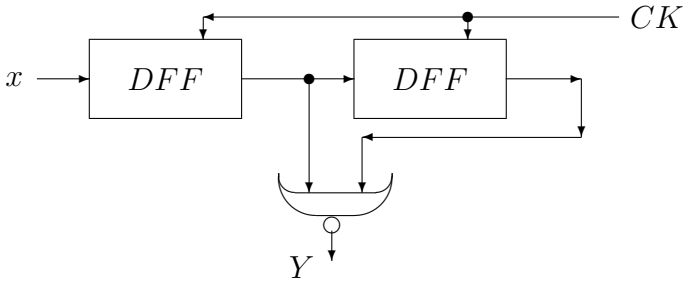
Observaţii:

(a) O analiză a automatului arată că unul din cele două DFF nu este utilizat. Deci el se poate reduce, fapt care revine formal la ideea că stările q_0 şi q_2 sunt echivalente – deci ele se pot identifica. Se obţine automatul



a cărui implementare utilizează un singur DFF.

(b) O soluţie interesantă – primită de la mulţi studenţi – este:



Eroarea constă în faptul că în fază iniţială automatul este resetat şi ambele DFF conţin valoarea 0. Deci atunci când va începe să funcţioneze, automatul va scoate 1 ca primă valoare (deşi nu a primit încă nici un bit), ceea ce va afecta valoarea counterului.

12. A se vedea soluţia anterioară, în care se construieşte un $COUNT_3$ folosind numai DFF-uri.

13. -

Bibliografie

1. Hayes J. P. - *Computer Architecture and Organisation*, McGraw Hill Series in Computer Engineering, Third edition, 2000

2. Hayes. J.P. - *Introduction to Digital Logic Design*, Reading MA, Addison Wesley, 1993

3. Hwang K - *Advanced Computer Architecture*, New York, Mc. Graw-Hill, 1993

4. Murdocca M., Heuring V. - *Principles of Computer Architecture*, Prentice Hall, 2001

5. Ştefan Gh. - *Circuit Complexity, Recursion, Grammars and Information*, Ed. Univ. Transilvania, Braşov, 1997

6. *http://plato.stanford.edu/entries/church-turing/*

7. *http://www.unicode.org*

8. *http://computer.howstuffworks.com/boolean.htm*

9. *http://www.play-hookey.com/digital/electronics/*