

Arhitectura sistemelor de calcul

Curs

Drăgulici Dumitru Daniel

Facultatea de matematică și informatică,
Universitatea București

2014

Cuprins

1 Performanța calculatoarelor

- Conceptul de performanță
- Măsurarea performanței

2 Aritmetică sistemelor de calcul

- Reprezentarea numerelor în matematică
- Reprezentarea numerelor în calculator
- Reprezentarea numerelor naturale ca întregi fără semn
- Reprezentarea numerelor întregi în complement față de 2
- Reprezentarea numerelor reale în virgulă mobilă

3 Logica pentru calculatoare

- Algebre booleene
- Funcții booleene
- Particularizare la cazul B_2

4 Circuite logice

- Circuite logice, Sisteme digitale
- 0-DS (Circuite combinaționale, Funcții booleene)
- 1-DS (Memorii)
- 2-DS (Automate finite)
- Algoritmi de înmulțire și împărțire hardware
- 3-DS (Procesoare) și 4-DS (Calculatoare)
- n -DS, $n > 4$

Bibliografie:

- John L. Hennessy, David A. Patterson:
"Organizarea și proiectarea calculatoarelor - interfață hardware/software",
Ed. All, 2002
- Adrian Atanasiu:
"Arhitectura calculatorului",
Ed. InfoData, 2006
- Materialele de laborator (pentru limbajul MIPS)

Cuprins

1 Performanța calculatoarelor

Conceptul de performanță

Măsurarea performanței

2 Aritmetică sistemelor de calcul

Reprezentarea numerelor în matematică

Reprezentarea numerelor în calculator

Reprezentarea numerelor naturale ca întregi fără semn

Reprezentarea numerelor întregi în complement față de 2

Reprezentarea numerelor reale în virgulă mobilă

3 Logica pentru calculatoare

Algebre booleene

Funcții booleene

Particularizare la cazul B_2

4 Circuite logice

Circuite logice, Sisteme digitale

0-DS (Circuite combinaționale, Funcții booleene)

1-DS (Memorii)

2-DS (Automate finite)

Algoritmi de înmulțire și împărțire hardware

3-DS (Procesoare) și 4-DS (Calculatoare)

n -DS, $n > 4$

Cuprins

1 Performanța calculatoarelor

Conceptul de performanță

Măsurarea performanței

2 Aritmetică sistemelor de calcul

Reprezentarea numerelor în matematică

Reprezentarea numerelor în calculator

Reprezentarea numerelor naturale ca întregi fără semn

Reprezentarea numerelor întregi în complement față de 2

Reprezentarea numerelor reale în virgulă mobilă

3 Logica pentru calculatoare

Algebre booleene

Funcții booleene

Particularizare la cazul B_2

4 Circuite logice

Circuite logice, Sisteme digitale

0-DS (Circuite combinaționale, Funcții booleene)

1-DS (Memorii)

2-DS (Automate finite)

Algoritmi de înmulțire și împărțire hardware

3-DS (Procesoare) și 4-DS (Calculatoare)

n -DS, $n > 4$

Conceptul de performanță

Performanța poate avea mai multe accepțiuni:

- (1) la un PC, suntem interesați de **timpul de răspuns** al unui program: este timpul fizic scurs între momentul când lansăm cererea și momentul când primim rezultatul; el trebuie să fie cât mai mic;
- (2) la un centru de calcul, suntem interesați de **productivitate (throughput)**: numărul de sarcini executate în unitatea de timp; el trebuie să fie cât mai mare.

Obs: (1) \Rightarrow (2) dar (2) $\not\Rightarrow$ (1).

De exemplu, centrul de calcul poate avea calculatoare lente dar multe; atunci pe fiecare calculator vom avea timpi de răspuns mari, dar numărul de sarcini executate în unitatea de timp în centrul de calcul va fi tot mare.

Ambele accepțiuni ale performanței necesită abordări complexe, astăzi pentru simplitate trebuie să ne limităm la una din ele.

În cele ce urmează, **performanța** se va referi la **timpul de răspuns**.

El privește rularea unui program cu niște date și depinde de mai mulți factori: codul programului, datele cu care se rulează, promptitudinea cu care utilizatorul introduce datele când sunt cerute, sistemul de operare, încărcarea sistemului (activitatea concomitentă a altor utilizatori), arhitectura hardware, etc.

Conceptul de performanță

Pentru a da o dimensiune numerică performanței (P) și a descrie dependența ei descrescătoare în raport cu timpul de răspuns (T), adoptăm formula:

$$P = \frac{1}{T} \quad (1)$$

Deja putem să comparăm performanțele sau să calculăm un raport al performanțelor unui program cu niște date fixate, pe două mașini X și Y :

$$P_X > P_Y \stackrel{\text{def}}{\iff} T_X < T_Y, \quad \frac{P_X}{P_Y} = n \stackrel{\text{def}}{\iff} \frac{T_Y}{T_X} = n \quad (2)$$

(am notat cu P_X , P_Y , T_X , T_Y , performanțele respectiv timpii de răspuns, pe mașinile X și Y).

Conceptul de performanță

Timpul de răspuns (al unui program cu niște date) are următoarele componente:

- (1) Timpul consumat cu executarea unor operații din program;
- (2) Timpul consumat cu executarea unor operații din sistemul de operare, dar legate de programul nostru (ex: apeluri sistem cerute de program);
- (3) Timpul consumat de program în așteptarea unor evenimente/resurse (ex: programul este adormit în "scanf()" așteptând ca utilizatorul să introducă date de la consolă);
- (4) Timpul în care programul este suspendat de sistemul de operare, pentru a executa operații administrative și sarcini legate de alte programe (care rulează în paralel cu al nostru).

Notăm (1) + (2) = **Timpul CPU**

Deci timpul CPU este timpul consumat de program cu executarea unor operații (calcule), fie ele din program sau din sistemul de operare dar legate de program. El este inclus în timpul de răspuns, iar ceea ce rămâne este timpul consumat de program în așteptare, fie că așteaptă un eveniment/resursă, fie că este suspendat de sistemul de operare pentru a executa operații administrative și sarcini legate de alte programe.

Conceptul de performanță

Pe anumite sisteme există comenzi cu ajutorul cărora se pot măsura și afișa diferite componente ale timpului de răspuns pentru o anumită cerere. De exemplu, în Linux comanda:

time cmd

afișază diferite componente ale timpului de răspuns obținute la executarea comenții *cmd*. De exemplu, poate afișa:

<i>real</i>	<i>2m39.000s</i>	(timpul de răspuns)
<i>user</i>	<i>1m30.700s</i>	(timpul utilizator, (1))
<i>sys</i>	<i>0m12.900s</i>	(timpul sistem, (2))

Astfel, putem calcula procentajul timpului CPU din timpul de răspuns:

$$\frac{90.7s + 12.9s}{159s} = 65\%$$

Deci $> 1/3$ din timpul de răspuns a fost consumat în așteptare, ceea ce ne spune ceva despre mediul în care a fost rulat programul - resurse greu disponibile, grad mare de încărcare a sistemului, etc.

Conceptul de performanță

Distingem între:

performanța sistemului = timpul de răspuns pe un sistem neîncărcat;
performanța CPU = timpul CPU.

În cele ce urmează restrângem și mai mult domeniul abordării și vom considera că **performanță înseamnă performanță CPU** (deci în formula (1) este vorba de **timpul CPU**).

Cuprins

1 Performanța calculatoarelor

Conceptul de performanță

Măsurarea performanței

2 Aritmetică sistemelor de calcul

Reprezentarea numerelor în matematică

Reprezentarea numerelor în calculator

Reprezentarea numerelor naturale ca întregi fără semn

Reprezentarea numerelor întregi în complement față de 2

Reprezentarea numerelor reale în virgulă mobilă

3 Logica pentru calculatoare

Algebre booleene

Funcții booleene

Particularizare la cazul B_2

4 Circuite logice

Circuite logice, Sisteme digitale

0-DS (Circuite combinaționale, Funcții booleene)

1-DS (Memorii)

2-DS (Automate finite)

Algoritmi de înmulțire și împărțire hardware

3-DS (Procesoare) și 4-DS (Calculatoare)

n -DS, $n > 4$

Măsurarea performanței

Timpul (CPU) se poate măsura în:

- **secunde (s)**
- **cicli de ceas (c)**

Legătura dintre cele două este dată de perioada de ceas (durata ciclului) D (măsurată în secunde (s), nanosecunde (ns), etc.) sau frecvența F (măsurată în herzi (Hz), megaherzi (MHz), etc.), după formula:

$$F = \frac{1}{D}$$

Când măsurăm în cicli nu mai contează fizica circuitelor (durata ciclului) ci doar împărțirea programului în operații mașina (logica programului) și a operațiilor mașină în ciclii (logica circuitelor) - avem astfel posibilitatea să comparăm d.p.v. logic o gamă mai largă de mașini.

Măsurarea performanței

O mărime care ne dă indicații asupra performanței este **numărul total de cicli execuți** C .

El depinde de program, de date și de împărțirea instrucțiunilor în cicli (am presupus programul scris în limbaj mașină); nu depinde de durata ciclului.

Performanța este cu atât mai mare cu cât C este mai mic.

Dacă presupunem programul scris într-un limbaj de nivel înalt, contează și împărțirea codului sursă în instrucțiuni mașină - deci intervine și performanța compilatorului.

Atunci putem calcula timpul CPU T după formula:

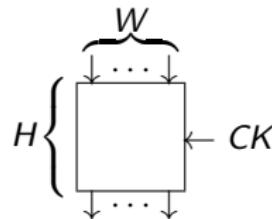
$$T = C \times D = \frac{C}{F} \quad (3)$$

Această formulă arată că putem reduce timpul T (deci mări performanța) reducând numărul de cicli execuți C sau/și durata ciclului D .

Este dificil însă de redus simultan ambii factori - de obicei reducerea uneia atrage creșterea celuilalt.

Măsurarea performanței

Pentru exemplificare, să considerăm următorul model simplificat de circuit, care execută câte o operație la fiecare ciclu:



El este construit prin conectarea altor circuite mai simple, în serie, în paralel, etc. (a se vedea capitolul de circuite logice), are o anumită lățime W , dată de numărul de intrări și de conectări în paralel, și o anumită înalțime H , dată de numărul de niveluri de conectare în serie; W este cu atât mai mare cu cât datele de prelucrat au o dimensiune mai mare (mai mulți biți), iar H este cu atât mai mare cu cât operația este mai complexă.

La fiecare tact de ceas (CK), datele de intrare sunt prelucrate prin circuit și transformate în niște date de ieșire, care la următorul tact vor fi preluate ca date intrare de un circuit conectat în continuare. Această prelucrare necesită un anumit timp, iar D trebuie să fie \geq acest timp - altfel, dacă următorul tact vine prea devreme (înainte ca ieșirile să fie stabilite), vor fi culese (și trimise mai departe) niște date eronate.

Măsurarea performanței

Creșterea D nu este influențată de W , deoarece intrările sunt procesate în paralel, ci de H (lungimea traseelor ce trebuie parcuse); în capitolul de circuite logice vom vedea, de exemplu, că o adunare pe 32 biți se face la fel de repede ca o adunare pe 64 biți (deoarece biții operanzilor sunt procesați în paralel), dar o adunare se face mai repede decât o înmulțire.

Să considerăm acum că circuitul desenat mai sus execută o operație pe parcursul mai multor cicli - deci sunt necesare mai multe treceri prin el pentru a se efectua toate prelucrările necesare.

Întrucât efectuarea unei anumite operații necesită o anumită cantitate de prelucrări (care depinde de natura matematică a operației), proiectantul circuitului va avea de ales între două variante:

- va împărți prelucrarea în etape puține, realizate câte una la fiecare ciclu, dar atunci la fiecare etapă trebuie făcute calcule multe, deci H va fi mare și astfel D va fi mare; atunci, la executarea programului vom avea C mic și D mare;
- va împărți prelucrarea în etape multe și atunci la fiecare etapă trebuie făcute calcule puține, deci H și D vor fi mici; atunci, la executarea programului vom avea C mare și D mic.

Exercițiu rezolvat

Exercițiu rezolvat:

Un program (cu niște date fixate) durează pe mașina A , având frecvența 400 MHz , 10 secunde. Dorim o mașină B pe care rularea să dureze 6 secunde. Proiectantul spune că poate crește frecvența, dar programul va necesita de 1.2 ori mai mulți cicli. Care va fi frecvența mașinii B ?

Explicație: proiectantul dorește reducerea duratei ciclului prin micșorarea înalțimii circuitelor (H din desenul anterior); atunci instrucțiunile vor necesita mai multe cicluri pentru a fi executate, factorul mediu de multiplicare fiind 1.2 .

Exercițiu rezolvat

Rezolvare:

Din enunț ni se dau:

$F_A = 400 \text{ MHz}$ (frecvența mașinii A , în MHz);

$T_A = 10 \text{ s}$ (timpul pe mașina A , în secunde s);

$T_B = 6 \text{ s}$ (timpul pe mașina B , în secunde s);

$C_B/C_A = 1.2$ (raportul numerelor de cicluri pe mașinile A, B);

Se cere:

$F_B = ?$ (frecvența mașinii B , în MHz).

Vom avea:

$$F_B \stackrel{(3)}{=} \frac{C_B}{T_B} \stackrel{ip}{=} \frac{1.2 \times C_A}{T_B} \stackrel{(3)}{=} \frac{1.2 \times F_A \times T_A}{T_B} \stackrel{ip}{=} \frac{1.2 \times 400 \text{ MHz} \times 10 \text{ s}}{6 \text{ s}}$$
$$= 800 \text{ MHz}$$

Comentariu: deși s-a dublat frecvența, performanța (raportul timpilor) nu s-a dublat; deci, performanța se obține greu.

Exercițiu rezolvat

Sfat:

În redactarea rezolvărilor, la efectuarea calculelor, menționați la fiecare pas și unitățile de măsură; utilitatea este următoarea:

- unele unități de măsură pot conține multipli care trebuie luați în considerare (de exemplu, 1 Mega = 10^6);
- avem posibilitatea să ne verificăm corectitudinea aplicării formulelor (de exemplu, dacă ne așteptăm să obținem secunde și rezultă cicli, nu este bine).

Convenție:

În cele ce urmează, mărimea fizică va fi notată cu literă mare iar unitatea de măsură cu literă mică, pe cât posibil același literă.

De exemplu, $C = 10 c$ înseamnă că numărul total de cicli execuți este 10 (cicli).

Măsurarea performanței

Alte două mărimi care ne dă indicații asupra performanței sunt **numărul total de instrucțiuni executate** I și **numărul mediu ce cicli per instrucțiune executată** (engl: **cycles per instruction**) CPI , calculat după formula:

$$CPI = \frac{C}{I} \quad (4)$$

I depinde de program și de date; nu depinde de împărțirea instrucțiunilor în cicli și nici de durata ciclului.

Performanța este cu atât mai mare cu cât I este mai mic.

CPI depinde de program, de date și de împărțirea instrucțiunilor în cicli; nu depinde de durata ciclului.

Performanța este cu atât mai mare cu cât CPI este mai mic.

Măsurarea performanței

CPI ne permite să comparăm două implementări hardware diferite ale unui același set de instrucțiuni.

De exemplu, să presupunem că avem de implementat o listă ce conține instrucțiunile + (adunare) și * (înmulțire).

Presupunem că avem două implementări, în care numărul de cicli necesari fiecărei instrucțiuni este dat de tabelul:

	Nr. cicli +	Nr. cicli *
Implementare 1	10 c	100 c
Implementare 2	12 c	70 c

Se pune problema care implementare este mai bună. Răspunsul depinde de procentajul cu care sunt executate fiecare dintre cele două instrucțiuni.

Dacă pentru categoria de programe și de date avute în vedere, la rulare se execută în medie $> 93.75\%$ + și restul *, este preferabilă prima implementare; dacă se execută în medie $< 93.75\%$ + și restul *, este preferabilă a doua implementare (exercițiu!).

Numărul mediu de cicli per instrucțiune calculat ținând cont de ponderea apariției fiecărui tip de instrucțiuni este tocmai *CPI*. Așa că, cu cât *CPI* este mai mic, performanța este mai mare (pentru categoria de programe și de date vizată).

Măsurarea performanței

Din formulele (3) și (4) obținem **ecuația elementară a performanței**:

$$T = I \times CPI \times D = \frac{I \times CPI}{F} \quad (5)$$

Ea leagă cei trei **factori cheie** ai performanței: numărul de instrucțiuni execuțate I , numărul mediu de cicli pe instrucțiune execuții CPI și durata ciclului D (sau frecvența F).

Există interdependențe între factori; de exemplu, scăderea CPI poate atrage creșterea D (deoarece la fiecare ciclu se vor executa operații mai multe și/sau mai complexe).

Acești factori trebuie considerați simultan atunci când analizăm un sistem, altfel putem trage concluzii eronate privind performanța - a se vedea următorul exercițiu rezolvat.

Măsurarea performanței

Ne punem problema cum putem măsura T folosind formulele (3) sau (5).

C este greu de măsurat - el depinde de program, de date și de arhitectură (împărțirea instrucțiunilor în cicli).

I este mai ușor de măsurat - depinde doar de program și de date; unele procesoare au chiar contoare hardware care numără instrucțiunile executate - ele se folosesc la simulări.

CPI este însă greu de măsurat - el depinde de program, de date și de arhitectură (împărțirea instrucțiunilor în cicli).

Măsurarea performanței

Uneori poate fi suficientă o aproximare a lui C , obținută urmărind execuția programului pe clase de instrucțiuni. Mai exact:

Considerăm n clase de instrucțiuni.

De exemplu: clasa instrucțiunilor de salt (din care face parte 'goto'), clasa instrucțiunilor aritmetice aditive (din care fac parte adunarea, scăderea), clasa instrucțiunilor aritmetice multiplicative (din care fac parte înmulțirea, împărțirea).

Pentru fiecare clasa $k \in \{1, \dots, n\}$ notăm cu CPI_k media numerelor de cicli necesitați de instrucțiunile din clasa k .

De exemplu, $CPI_{aditive}$ este media dintre numărul de cicli necesitați de o adunare și numărul de cicli necesitați de o scădere.

Notăm deci că la calcularea lui CPI_k se ia câte un singur exemplar din fiecare instrucțiune a clasei k , fără ponderi/procentaje.

Pentru un program și niște date considerate, notăm cu I_k numărul de instrucțiuni executate din clasa k , $k \in \{1, \dots, n\}$.

Măsurarea performanței

Atunci pentru programul, datele și arhitectura considerate, vom avea:

$$C \simeq \sum_{k=1}^n (CPI_k \times I_k) \quad (6)$$

Valorile n și CPI_k , $k \in \{1, \dots, n\}$, depind doar de arhitectură - pentru o mașină dată ele sunt fixate și pot fi comunicate prin documentația mașinii.

Valorile I_k , $k \in \{1, \dots, n\}$, depind de program și de date, nu și de arhitectură.

Astfel, pe o mașină dată, dacă vrem să aflăm (cu aproximare) numărul de cicli C consumați la rularea unui anumit program cu anumite date, este suficient să aflăm valorile I_k , $k \in \{1, \dots, n\}$, iar acestea sunt mai ușor de aflat, deoarece instrucțiunile se numără mai ușor decât ciclii.

Măsurarea performanței

Valoarea C calculată cu formula (6) este însă doar una aproximativă.

De exemplu, presupunem că pe o mașină dată avem:

Clasa	Instrucțiuni	Nr. de cicli necesitați
aditive	adunare	1
	scădere	3
multiplicative	înmulțire	10
	împărțire	30

Rezultă $CPI_{aditive} = 2$, $CPI_{multiplicative} = 20$.

Considerăm un program și niște date a.î. la rulare se execută o adunare și o înmulțire.

Valoarea exactă a lui C este: $1 + 10 = 11$.

Valoarea lui C calculată cu formula (6) este: $2 + 20 = 22$.

Considerăm acum un program și niște date a.î. la rulare se execută o scădere și o împărțire.

Valoarea exactă a lui C este: $3 + 30 = 33$.

Valoarea lui C calculată cu formula (6) este tot: $2 + 20 = 22$.

Măsurarea performanței

Într-adevăr, cu formula (6) am considerat de fiecare dată că se execută o instrucțiune aditivă și o instrucțiune multiplicativă, nesenzând că la primul program se execută cele mai rapide instrucțiuni din cele două clase, iar la al doilea program cele mai lente.

Valoarea lui C calculată cu formula (6) se apropie de valoarea exactă cu atât mai mult cu cât din fiecare clasă instrucțiunile se execută cu ponderi mai apropriate (cam tot atâtea adunări câte scăderi, cam tot atâtea înmulțiri câte împărțiri).

Exercițiu rezolvat

Exercițiu rezolvat:

Pe o mașină dată, avem următoarele clase de instrucțiuni și CPI asociate:

clasa	CPI_{clasa}
A	1
B	2
C	3

Considerăm două programe și niște date, pentru care s-au măsurat:

Program	Nr. de instrucțiuni executate din fiecare clasă		
	A	B	C
programul 1	2	1	2
programul 2	4	1	1

- Se cer:
- Care program execută mai multe instrucțiuni ?
 - Care program este mai rapid ?
 - Cât este CPI pentru fiecare program ?

Exercițiu rezolvat

Rezolvare:

a) Notăm:

I_k^p = numărul de instrucțiuni executate de programul $p \in \{1, 2\}$ din clasa $k \in \{A, B, C\}$;

I^p = numărul total de instrucțiuni executate de programul $p \in \{1, 2\}$.

Avem:

$$I^1 = I_A^1 + I_B^1 + I_C^1 = 2i + 1i + 2i = 5i$$

$$I^2 = I_A^2 + I_B^2 + I_C^2 = 4i + 1i + 1i = 6i$$

Comentariu: am obținut $I^1 < I^2$, ceea ce pare să arate că programul 1 este mai performant.

Exercițiu rezolvat

b) Notăm:

C^p = numărul total de cicli execuții de programul $p \in \{1, 2\}$.

Folosind formula (6), avem:

$$\begin{aligned}C^1 &= CPI_A \times I_A^1 + CPI_B \times I_B^1 + CPI_C \times I_C^1 \\&= 1 \frac{c}{i} \times 2i + 2 \frac{c}{i} \times 1i + 3 \frac{c}{i} \times 2i = 10c\end{aligned}$$

$$\begin{aligned}C^2 &= CPI_A \times I_A^2 + CPI_B \times I_B^2 + CPI_C \times I_C^2 \\&= 1 \frac{c}{i} \times 4i + 2 \frac{c}{i} \times 1i + 3 \frac{c}{i} \times 1i = 9c\end{aligned}$$

Comentariu:

Am obținut $C^1 > C^2$, ceea ce pare să arate că programul 2 este mai performant, în contradicție cu a).

Dintre mărimile I și C , cea care măsoară realist durata execuției este C , deoarece folosește aceeași unitate de măsură, ciclul (instrucțiunile nu durează toate la fel de mult). Deci răspunsul corect este cel dat de b).

Explicația erorii de la a): programul 1 execută instrucțiuni mai puține, dar mai lungi (execută două din clasa C , care durează în medie câte 3 cicli).

Exercițiu rezolvat

c) Notăm:

CPI^p = numărul mediu de cicli pe instrucțiune execuția de programul $p \in \{1, 2\}$.

Folosind formula (4), avem:

$$CPI^1 = \frac{C^1}{I^1} = \frac{10c}{5i} = 2\frac{c}{i} \text{ (sau } 2cpi\text{)}.$$

$$CPI^2 = \frac{C^2}{I^2} = \frac{9c}{6i} = 1.5\frac{c}{i} \text{ (sau } 1.5cpi\text{)}.$$

Comentariu:

Am obținut $CPI^1 > CPI^2$, ceea ce pare să arate că programul 2 este mai performant, ca la b) (și în contradicție cu a)).

Întrucât am văzut că b) oferă răspunsul corect, rezultă că, cel puțin în acest caz, mărimea CPI descrie performanța mai bine decât mărimea I .

Răspunsul de la c) era de așteptat, deoarece programul 2 este mai rapid deși execută mai multe instrucțiuni și de aceea trebuie ca instrucțiunile executate de el să fie mai rapide.

Oricum, acest exercițiu ilustrează ce am spus mai devreme, că factorii care influențează performanța trebuie considerați simultan atunci când analizăm un sistem, altfel putem trage concluzii eronate privind performanța.

Măsurarea performanței

O altă mărime care ne dă indicații asupra performanței este **frecvența de executare a instrucțiunilor**:

$$FI = \frac{I}{T} \quad (7)$$

Întrucât de obicei valorile lui FI (măsurate în instrucțiuni pe secundă) sunt mari, se obișnuiește introducerea încă unui factor 10^6 la numitor, obținându-se mărimea:

$$MIPS = \frac{I}{T \times 10^6} \quad (8)$$

măsurată în milioane de instrucțiuni pe secundă (**millions instructions per second**).

FI și $MIPS$ depind de program, de date, de împărțirea instrucțiunilor în cicli și de durata ciclului.

Performanța este cu atât mai mare cu cât FI și $MIPS$ sunt mai mari.

Măsurarea performanței

Ca și în cazul altor mărimi discutate mai devreme, *FI* și *MIPS* trebuie considerate împreună cu ceilalți factori atunci când analizăm un sistem, altfel putem trage concluzii eronate privind performanța - a se vedea următorul exercițiu rezolvat.

Exercițiu rezolvat

Exercițiu rezolvat:

Pe o mașină dată, având frecvența de **500 MHz**, avem următoarele clase de instrucțiuni și **CPI** asociate:

clasa	CPI_{clasa}
A	1
B	2
C	3

Considerăm două programe și niște date, pentru care s-au măsurat:

Program	Nr. de instrucțiuni executate din fiecare clasă		
	A	B	C
programul 1	5	1	1
programul 2	10	1	1

- Se cer:
- Care sunt timpii CPU pentru cele două programe ?
 - Care sunt valorile MIPS pentru cele două programe ?

Exercițiu rezolvat

Rezolvare:

Situația seamănă cu cea din exercițiul precedent și atunci la fel ca acolo putem afla numărul total de cicli execuții. Acum știm în plus și frecvența, iar aceasta ne va permite ca din numărul de cicli să aflăm timpii CPU.

Notăm:

I_k^p = numărul de instrucțiuni executate de programul $p \in \{1, 2\}$ din clasa $k \in \{A, B, C\}$;

I^p = numărul total de instrucțiuni executate de programul $p \in \{1, 2\}$.

C^p = numărul total de cicli execuții de programul $p \in \{1, 2\}$.

T^p = timpul CPU consumat de programul $p \in \{1, 2\}$.

$MIPS^p$ = valoarea MIPS pentru programul $p \in \{1, 2\}$.

Exercițiu rezolvat

a) Folosind formula (6), avem:

$$C^1 = CPI_A \times I_A^1 + CPI_B \times I_B^1 + CPI_C \times I_C^1 \\ = 1\frac{c}{i} \times 5i + 2\frac{c}{i} \times 1i + 3\frac{c}{i} \times 1i = 10c$$

$$C^2 = CPI_A \times I_A^2 + CPI_B \times I_B^2 + CPI_C \times I_C^2 \\ = 1\frac{c}{i} \times 10i + 2\frac{c}{i} \times 1i + 3\frac{c}{i} \times 1i = 15c$$

Atunci, folosind formula (3), avem:

$$T^1 = \frac{C^1}{F} = \frac{10c}{500 \text{ MHz}} = \frac{10c}{500 \times 10^6 \text{ Hz}} = \frac{10c}{500 \times 10^6 \frac{c}{s}} \\ = 2 \times 10^{-8} s$$

$$T^2 = \frac{C^2}{F} = \frac{15c}{500 \text{ MHz}} = \frac{15c}{500 \times 10^6 \frac{c}{s}} = 3 \times 10^{-8} s$$

Comentariu: am obținut $T^1 < T^2$, ceea ce pare să arate că programul 1 este mai performant. Mărimea T măsoară realist performanța, deoarece folosește același etalon pentru ambele programe (toate secundele sunt la fel de lungi).

Exercițiu rezolvat

b) Avem:

$$I^1 = I_A^1 + I_B^1 + I_C^1 = 5i + 1i + 1i = 7i$$

$$I^2 = I_A^2 + I_B^2 + I_C^2 = 10i + 1i + 1i = 12i$$

Atunci, folosind formula (8), avem:

$$\text{MIPS}^1 = \frac{I^1}{T^1 \times 10^6} \text{ mips} = \frac{7i}{2 \times 10^{-8} \times 10^6} \text{ mips} = 350 \text{ mips}$$

$$\text{MIPS}^2 = \frac{I^2}{T^2 \times 10^6} \text{ mips} = \frac{12i}{3 \times 10^{-8} \times 10^6} \text{ mips} = 400 \text{ mips}$$

Comentariu: am obținut $\text{MIPS}^1 < \text{MIPS}^2$, ceea ce pare să arate că programul 2 este mai performant, în contradicție cu a).

Am văzut însă că răspunsul corect este cel dat de a). Explicația erorii de la b) constă în faptul că programul 2 execută instrucțiuni rapide (deci execută multe instrucțiuni pe secundă), dar foarte multe (și astfel necesită multe secunde, mai multe decât programul 1).

Așadar, un MIPS mare nu înseamnă neapărat un timp mai scurt.

Deci, parametrii performanței trebuie analizați împreună, nu izolat.

Cuprins

1 Performația calculatoarelor

Conceptul de performanță
Măsurarea performanței

2 Aritmetică sistemelor de calcul

Reprezentarea numerelor în matematică
Reprezentarea numerelor în calculator
Reprezentarea numerelor naturale ca întregi fără semn
Reprezentarea numerelor întregi în complement față de 2
Reprezentarea numerelor reale în virgulă mobilă

3 Logica pentru calculatoare

Algebre booleene
Funcții booleene
Particularizare la cazul B_2

4 Circuite logice

Circuite logice, Sisteme digitale
0-DS (Circuite combinaționale, Funcții booleene)
1-DS (Memorii)
2-DS (Automate finite)
Algoritmi de înmulțire și împărțire hardware
3-DS (Procesoare) și 4-DS (Calculatoare)
 n -DS, $n > 4$

Numerele și operațiile cu ele sunt concepte matematice.

Pentru a comunica numerele și a efectua ușor operațiile cu ele, transcriem numerele prin niște reprezentări simbolice și operațiile cu numere prin algoritmi de calcul cu aceste reprezentări.

Modul uzual de reprezentare a numerelor în matematică este prin scriere pozițională, ca sir de cifre într-o anumită bază de enumerație, de regulă baza 10. Algoritmii de calcul cu aceste reprezentări operează cifră cu cifră, propagând un transport sau luând un împrumut.

Modul de reprezentare a numerelor în calculator și algoritmii de calcul cu aceste reprezentări seamănă cu cele folosite în matematică, dar nu sunt identice. De exemplu, în matematică putem folosi ușor oricâte simboluri ajutătoare: virgula, semnul '-', etc. În calculator, pentru a stoca informații se folosesc biți, iar un bit stochează o informație bivalentă; odată ce am fixat semnificația ei, de exemplu cifra 0 și respectiv cifra 1, nu mai avem la dispoziție o altă valoare de bit care să însemne virgulă sau '-' - totul trebuie să fie doar sir de cifre (binare).

În prima secțiune vom descrie pe scurt modul de reprezentare și algoritmii de calcul cu reprezentări folosite în matematică, iar în următoarele secțiuni le vom detalia pe cele folosite în calculator.

Cuprins

1 Performația calculatoarelor

Conceptul de performanță
Măsurarea performanței

2 Aritmetică sistemelor de calcul

Reprezentarea numerelor în matematică

Reprezentarea numerelor în calculator

Reprezentarea numerelor naturale ca întregi fără semn

Reprezentarea numerelor întregi în complement față de 2

Reprezentarea numerelor reale în virgulă mobilă

3 Logica pentru calculatoare

Algebre booleene

Funcții booleene

Particularizare la cazul B_2

4 Circuite logice

Circuite logice, Sisteme digitale

0-DS (Circuite combinaționale, Funcții booleene)

1-DS (Memorii)

2-DS (Automate finite)

Algoritmi de înmulțire și împărțire hardware

3-DS (Procesoare) și 4-DS (Calculatoare)

n -DS, $n > 4$

Reprezentarea numerelor într-o bază

Modul de reprezentare a numerelor într-o anumită bază și algoritmii de calcul cu aceste reprezentări sunt cunoscute din școală/liceu, nu le vom descrie cu toate detaliile ci doar vom reaminti câteva noțiuni și algoritmi, evidențиind anumite idei.

La baza scrierii pozitionale într-o bază se află scrierea polinomială în puterile succesive ale bazei.

De exemplu, reprezentarea 1100 în baza 2 înseamnă numărul care se calculează $1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$, adică 12.

Reprezentarea numerelor într-o bază

Când definim modul de reprezentare și de calcul cu numerele într-o bază b , interfeșă două limbaje:

- limbajul matematic combinat cu cel natural, în care vorbim (este limbajul gazdă a discursului);
- limbajul reprezentărilor în baza b , despre care vorbim (este limbajul obiect al discursului).

Mai spunem că primul este metalimbaj pentru al doilea.

Când ne aflăm în metalimbaj și vrem să comunicăm numere sau operații cu ele, trebuie să folosim tot reprezentări și algoritmi cu aceste reprezentări, deoarece nu putem comunica direct niște idei ci doar prin niște formalizări ale lor. În acest scop, folosim reprezentările și algoritmii din baza 10 (se presupun cunoșcuți).

Deci, în final, metalimbajul și limbajul sunt limbaje de același tip: cel al reprezentărilor poziționale într-o bază, doar că metalimbajul este în baza 10 iar limbajul în baza b .

În metalimbaj vom scrie reprezentări și algoritmi cu reprezentări dar ne vom gândi la numere și operații cu numere, în limbaj vom scrie reprezentări și algoritmi cu reprezentări și ne vom gândi chiar la ele (reprezentarea pozițională în baza b este un sir finit de simboluri, numite cifre ale bazei b).

Reprezentarea numerelor într-o bază

Anumite sintagme pot apărea la fel și în metalimbaj și în limbaj, iar dacă nu se deduce din context unde ne aflăm, pot apărea confuzii. De exemplu, scrierea

1 1 0 0 poate fi considerată în metalimbaj (baza 10) și înseamnă numărul 'o mie o sută', sau în limbajul bazei 2 și însemnă sirul ' 1 1 0 0', prin care se reprezintă numărul 'doisprezece'.

De aceea, când este pericol de confuzie, vom folosi scrieri diferite. Mai exact:

- o scriere de forma 12 este în metalimbaj și înseamnă numărul 'doisprezece';
- o scriere de forma $(12)_2$ este tot în metalimbaj și înseamnă că numărului 'doisprezece' ii aplicăm procedeul de conversie în baza 'doi' ($(.)_2$ este de fapt o funcție bijectivă de la numere/reprezentări zecimale la reprezentări binare);
- o scriere de forma $\overline{1100}_2$ este în limbaj și înseamnă reprezentarea (sirul de cifre binare) '1 1 0 0', prin care se reprezintă numărul 'doisprezece'.

Dacă nu este pericol de confuzie, putem folosi notații simplificate. De exemplu, dacă din context este clară baza, putem scrie doar $\overline{1100}$, iar dacă este clar și nivelul (metalimbaj/limbaj), putem scrie doar 1100.

În restul acestei secțiuni vom descrie (reaminti) regulile de conversie a numerelor dintr-o bază în alta și algoritmii de calcul cu numerele reprezentate într-o anumită bază.

baza 10 → baza b ; numere întregi

Trecerea din baza 10 într-o bază b :

- Cazul numerelor întregi:

Fie $x \in \mathbb{Z}$ și $b \in \mathbb{N}$, $b \geq 2$, baza.

Considerăm x și b scrise în baza 10; facem calculele în baza 10.

Regulă:

- împărțim cu rest pe $|x|$ la b , apoi câtul la b , etc., până obținem câtul 0;
- luăm resturile în ordine inversă și le înlocuim cu cifre ale bazei b ;
- dacă $x < 0$, punem în față “ $-$ ”.

Observație: dacă am continua procedeul după ce am obținut câtul 0, am obține noi câturi și resturi 0, care ar genera cifre 0 în stânga reprezentării, iar acestea nu schimbă semantica reprezentării (nu afectează valoarea reprezentată).

baza 10 → baza b; numere întregi

Calculele se pot redacta astfel (c_0, c_1, \dots, c_n sunt cifre ale bazei b):

```

x      |   b
      |-----
.      | cat0 |   b
.      .   |-----
rest0   .   |   cat1
C0      .           Obtinem reprezentarea: Cn ... C1 C0
      rest1
      .
      C1
      .
      cat(n-1) |   b
      .           |-----
      .           |   0
      .
      restn
      Cn

<===== citim

```

baza 10 \rightarrow baza b; numere întregi

Exemplu: $(4235)_{16} = ?$

$$\begin{array}{r|l} 4235 & 16 \\ 32 & \hline \end{array}$$
$$\begin{array}{r|l} --- & 264 | 16 \\ 103 & 16 | \hline \end{array}$$
$$\begin{array}{r|l} 96 & -- | 16 | 16 \\ -- & 104 | 16 | \hline \end{array}$$
$$\begin{array}{r|l} ==75 & 96 -- | 1 | 16 \\ 64 & -- 0 0 | \hline \end{array}$$
$$\begin{array}{r|l} -- & ==8 0 - | 0 \\ 11 & 8 1 \\ B & 1 \end{array}$$

<=====

Deci: $(4235)_{16} = \overline{108B}$

baza 10 → baza b; numere întregi

Dacă b are (în baza 10) doar o cifră, atunci cat_k și rest_k se pot calcula mental, iar calculele se pot redacta mai simplu:

x		rest0	C0	/\	
cat0		rest1	C1		
cat1					
.	.				
.	.				-----
cat(n-1)		restn	Cn		Obtinem reprezentarea: Cn ... C1 C0
0					citim

baza 10 \rightarrow baza b; numere întregi

Exemplu: $(105)_2 = ?$

105		1	/\
52		0	
26		0	
13		1	
6		0	
3		1	
1		1	
0			

(pentru $b < 10$ cifrele bazei b sunt aceleași ca în baza 10 și nu este nevoie să le mai scriem o dată în dreapta).

Deci: $(105)_2 = \overline{1101001}$

baza 10 → baza b ; numere reale

- Cazul numerelor reale (fracționare):

Fie $x \in \mathbb{R}$ și $b \in \mathbb{N}$, $b \geq 2$, baza.

Considerăm x și b scrise în baza 10; facem calculele în baza 10.

Regulă:

- partea întreagă a lui $|x|$ se reprezintă ca la numere întregi;
- partea fracționară a lui $|x|$ se înmulțește cu b și se ia partea întreagă, apoi partea fracționară rămasă se înmulțește cu b și se ia partea întreagă, etc., până obținem o parte fracționară 0 sau care a mai fost întâlnită (dacă $x \in \mathbb{R} \setminus \mathbb{Q}$ procedeul continuă la infinit, nu este algoritm);
- luăm părțile întregi obținute în ordine directă și le înlocuim cu cifre ale bazei b ;
- dacă $x < 0$, punem în față “-”.

baza 10 → baza b ; numere reale

Observații:

- regula nu o contrazice pe cea pentru numere întregi ci o completează (afirmă ceva și pentru partea fracționară, care la numere întregi lipsește); cifrele produse de cele două părți ale regulii (pentru partea întregă, respectiv fracționară) nu se suprapun, deoarece se află în părți diferite ale virgulei;
- cu partea întreagă facem împărțiri succesive la b și obținem cifre de la virgulă spre stânga; cu partea fracționară facem înmulțiri succesive cu b și obținem cifre de la virgulă spre dreapta; per total, cifrele se obțin de la virgulă spre extremități;
- fiecare din părțile fracționare obținute reprezintă un număr real din intervalul $[0, 1)$; când o înmulțim cu b , obținem un număr real din intervalul $[0, b)$; luând apoi partea întreagă, obținem un număr întreg din mulțimea $\{0, \dots, b - 1\}$, care corespunde unei cifre în baza b - aceasta este următoarea zecimală (în baza b); cu partea fracționară rămasă continuăm procedeul;

baza 10 → baza b ; numere reale

- până când se aplică procedeul:

- dacă la un moment dat întâlnim o parte fracționară 0, înmulțind cu b obținem o nouă parte întreagă (i.e. o nouă zecimală) 0 și o nouă parte fracționară 0, etc., deci continuând vom obține noi zecimale 0 nesemnificate; de aceea ne putem opri și conchidem că am obținut o fracție zecimală finită;
- dacă la un moment dat obținem o parte fracționară care a mai fost întâlnită, continuând procedeul vom obține aceeași succesiune de zecimale ca după prima întâlnire; de aceea ne putem opri și conchidem că am obținut o fracție zecimală periodică (simplă sau mixtă); perioada este succesiunea de zecimale generate după prima întâlnire;

baza 10 → baza b; numere reale

- Întrucât partile fractionare generate de procedeu reprezintă numere reale din intervalul $[0, 1)$ și există o infinitate de asemenea numere distințe, este posibil ca prin continuarea procedeului să nu obținem nici o dată o parte fractionară 0 sau care a mai fost întâlnită; atunci procedeul nu se termină (nu este algoritm); acest lucru se întâmplă însă doar în cazul numerelor iraționale;

mai exact: dacă $x \in \mathbb{Q}$ atunci în orice bază b reprezentarea lui x va fi o fracție zecimală finită, periodică simplă sau periodică mixtă; dacă $x \in \mathbb{R} \setminus \mathbb{Q}$ atunci în orice baza b reprezentarea lui x va fi o fracție zecimală infinită neperiodică;

deci, dacă într-o aplicație se va cere "reprezentați 7.8 în baza 2 ", vom ști că procedeul se termină (și obținem o fracție zecimală finită, periodică simplă sau periodică mixtă), deoarece în baza sursă (adică 10) numărul s-a reprezentat printr-o fracție zecimală finită, deci este rațional; nu vom cere "reprezentați $\sqrt{2}$ în baza 2 ";

atenție însă că un același număr rațional poate avea într-o bază o reprezentare prin fracție zecimală finită, în alta prin fracție zecimală periodică simplă, în alta prin fracție zecimală periodică mixtă.

baza 10 → baza b; numere reale

Exemplu: $(7.8)_2 = ?$

7		1	/\	2 * 0.8 = 1.6		citim
3		1		-		
1		1		2 * 0.6 = 1.2		
0				-		
				2 * 0.2 = 0.4		
				-		
				2 * 0.4 = 0.8	\/	
				-		
				2 * 0.8	stop, am reintalnit 0.8	

Deci: $(7.8)_2 = \overline{111.(1100)}$

baza 10 → baza b; numere reale

Dacă b are (în baza 10) doar o cifră, atunci înmulțirea unei părți fracționare cu b se poate face mintal, iar calculele se pot redacta mai simplu:

parte_fractionara_0		zecimala_1		citim
parte_fractionara_1		zecimala_2		
.				
.				
.			\	

În cazul exemplului de mai sus vom avea:

0.8		1		
0.6		1		
0.2		0		
0.4		0	\	
0.8				

baza 10 → baza b; numere reale

Exemplu: $(7.8)_5 = ?$

$$\begin{array}{r|l} 7 & 2 \\ 1 & 1 \\ 0 & \end{array}$$

0.8 | 4
0 | stop, am intalnit partea fractionara 0

Deci: $(7.8)_5 = \overline{12.4}$

Observăm că 7.8 se reprezintă în baza 2 prin fracție zecimală periodică, iar în baza 5 (și în baza 10) prin fracție zecimală finită.

baza 10 → baza b; numere reale

Dacă numărul x este dat (în baza sursă, 10) printr-o fracție zecimală periodică (simplă sau mixtă), nu putem efectua direct calculele regulii de conversie, deoarece nu știm să înmulțim fracții zecimale infinite.

Regula de conversie se bazează însă pe proprietăți matematice, care nu depind de modul de reprezentare a numerelor; de aceea, o putem aplica, dar vom alege alte reprezentări, convenabile - anume prin fracții ordinare.

baza 10 \rightarrow baza b; numere reale

Exemplu: $(4.(3))_2 = ?$

$$\begin{array}{r|l} 4 & 0 \\ 2 & | 0 \\ 1 & | 1 \\ 0 & | \end{array} \quad \begin{aligned} 2 * 0.(3) &= 2 * (3/9) = 2 * (1/3) = (2/3) = 0 + (2/3) \\ 2 * (2/3) &= 4/3 = 1 + (1/3) \\ 2 * (1/3) &\text{ stop, am reintalnit } 1/3 \end{aligned}$$

Deci: $(4.(3))_2 = \overline{100.(01)}$

baza $b \rightarrow$ baza 10

Trecerea dintr-o bază b în baza 10 ($b \in \mathbb{N}$, $b \geq 2$):

Regulă:

- înlocuim cifrele cu numerele pe care le reprezintă (scrise în baza 10) și scrierea pozitională cu scriere polinomială în puterile succesive ale lui b (scris tot în baza 10);
- facem calculele în baza 10.

Exemple:

$$(\overline{1101})_2^{-1} = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 13$$

$$(\overline{A2C})_{16}^{-1} = 10 \times 16^2 + 2 \times 16^1 + 12 \times 16^0 = 2604$$

$$(\overline{12.4})_5^{-1} = 1 \times 5^1 + 2 \times 5^0 + 4 \times 5^{-1} = 7 + \frac{4}{5} = \frac{39}{5} = 7.8$$

Observații:

- în primul exemplu, în stânga primului '=' , 0 și 1 înseamnă cifre (în baza 2), iar în dreapta numere (în baza 10); în al doilea exemplu confuzia între cifre și numere este mai mică;
- în ultimul exemplu vedem că pentru cifrele părții fracționare exponenții scad în continuare, la valori negative.

baza $b \rightarrow$ baza 10

Dacă reprezentarea în baza b de la care pornim este o fracție zecimală periodică (simplă sau mixtă), scrierea polinomială rezultată este infinită. O asemenea sumă se poate calcula folosind instrumentul matematic al seriilor.

Putem evita folosirea seriilor dacă înlocuim fracția zecimală sursă cu o fracție ordinară (în baza b).

Regula de trecere de la fracție zecimală la fracție ordinară în baza b este la fel ca în baza 10, cu diferența că 9 se înlocuiește cu $b - 1$.

Exemplu:

$$\overline{(10.1(011))}_2^{-1} = (\overline{10} + \frac{\overline{1011} - \overline{1}}{\overline{1110}})_2^{-1} = (\overline{10})_2^{-1} + \frac{(\overline{1011})_2^{-1} - (\overline{1})_2^{-1}}{(\overline{1110})_2^{-1}} = 2 + \frac{11 - 1}{14} = 2 + \frac{10}{14} = \frac{38}{14} = \frac{19}{7} = 2.(714285)$$

Observație: am folosit și faptul că funcțiile de forma $(.)_b$ și $(.)_b^{-1}$ au proprietăți de morfism (comută cu operațiile aritmetice).

baza $b_1 \rightarrow$ baza b_2

Trecerea dintr-o bază b_1 într-o bază b_2 ($b_1, b_2 \in \mathbb{N}$, $b_1, b_2 \geq 2$):

Problema este de aceeași natură ca în cazul trecerilor baza $10 \rightarrow$ baza b și baza $b \rightarrow$ baza 10 de mai înainte, deci am putea să aplicăm regulile de acolo. Dar, dacă procedăm ca la trecerea baza $10 \rightarrow$ baza b , va trebui să facem calculele în baza b_1 , iar dacă procedăm ca la trecerea baza $b \rightarrow$ baza 10 , va trebui să facem calculele în baza b_2 .

Dacă vrem să facem calculele doar în baza 10 , vom trece prin baza 10 compunând cele două reguli.

Regulă:

baza $b_1 \rightarrow$ baza $10 \rightarrow$ baza b_2

baza $b_1 \rightarrow$ baza b_2

Trecerea baza $b_1 \rightarrow$ baza b_2 se poate face mai simplu, fără a mai trece prin baza 10, dacă una din baze este putere a celeilalte:

- Trecerea $b^k \rightarrow b$ ($b, k \in \mathbb{N}$ $b \geq 2, k \geq 1$)

Regulă:

- înlocuim fiecare cifră a bazei b^k cu câte un grup de k cifre ale bazei b ;

- eliminăm 0-le extreme (din stânga părții întregi și din dreapta părții fracționare);

- Trecerea $b \rightarrow b^k$ ($b, k \in \mathbb{N}$ $b \geq 2, k \geq 1$)

Regulă:

- grupăm câte k cifre, de la virgulă spre stânga și spre dreapta, completând eventual cu 0-uri grupurile extreme (pentru a avea grupuri de k cifre); aceste 0-uri se pun la stânga în grupul aflat cel mai la stânga al părții întregi și la dreapta în grupul aflat cel mai la dreapta al părții fracționare;

- înlocuim fiecare grup de k cifre ale bazei b cu câte o cifră a bazei b^k .

baza $b_1 \rightarrow$ baza b_2

Observații:

- pentru a aplica regulile de mai înainte, este necesar să știm cele 2^k corespondențe posibile între o cifră a bazei b^k și un grup de k cifre ale bazei b ; acestea se pot determina aplicând trecerea prin baza 10 (baza $b^k \rightarrow$ baza 10 \rightarrow baza b) și se pot reține într-un tabel (cu 2^k linii);
- explicația intuitivă a acestor reguli este următoarea: dacă în scrierea polinomială în puterile succesive ale lui b , cu coeficienți $\in \overline{0, b - 1}$, grupăm câte k termeni consecutivi și dăm factor comun puterea lui b cea mai mică, obținem o scriere polinomială în puterile succesive ale lui b^k , cu coeficienți $\in \overline{0, b^k - 1}$; din unicitatea acestor scrieri rezultă că ele dau reprezentările în bazele b și b^k .

baza $b_1 \rightarrow$ baza b_2

Exemplu: Să se convertească $\overline{1A.C}_{16}$ în baza 2 și $\overline{11011.101}_2$ în baza 16.

Avem $16 = 2^4$, iar grupurile posibile de 4 cifre binare și cifrele hexa corespunzătoare reprezintă numerele $\in [0, 15]$ și sunt cuprinse în tabelul:

Zec	Bin	Hex									
0	0000	0	4	0100	4	8	1000	8	12	1100	C
1	0001	1	5	0101	5	9	1001	9	13	1101	D
2	0010	2	6	0110	6	10	1010	A	14	1110	E
3	0011	3	7	0111	7	11	1011	B	15	1111	F

Atunci avem:

$$\overline{1A.C}_{16} \Rightarrow \underbrace{1}_{\text{Zec}} \underbrace{A}_{\text{Hex}} \cdot \underbrace{C}_{\text{Hex}} \Rightarrow \underbrace{0001}_{\text{Bin}} \underbrace{1010}_{\text{Bin}} \cdot \underbrace{1100}_{\text{Bin}} \Rightarrow 00011010.1100 \Rightarrow \overline{11010.11}_2$$

$$\overline{11011.101}_2 \Rightarrow \underbrace{1}_{\text{Zec}} \underbrace{1011}_{\text{Bin}} \cdot \underbrace{101}_{\text{Bin}} \Rightarrow \underbrace{0001}_{\text{Bin}} \underbrace{1011}_{\text{Bin}} \cdot \underbrace{1010}_{\text{Bin}} \Rightarrow \underbrace{1}_{\text{Zec}} \underbrace{B}_{\text{Hex}} \cdot \underbrace{A}_{\text{Hex}} \Rightarrow \overline{1B.A}_{16}$$

baza $b_1 \rightarrow$ baza b_2

Observație: metodele de mai sus nu se pot aplica pentru a trece direct din baza 8 în baza 16 sau invers, deoarece 16 este multiplu al lui 8, dar nu putere a lui 8. Putem însă proceda în doi pași, trecând prin baza 2, de exemplu:

baza 8 \rightarrow baza 2 \rightarrow baza 16

Operații aritmetice într-o bază b

Operațiile aritmetice cu numerele scrise într-o bază $b \geq 2$ oarecare se fac după reguli asemănătoare ca în baza 10, dar transportul și împrumutul trebuie să se facă la b , nu la 10.

Pentru calculele de o cifră în baza b putem trece numerele în baza 10, facem calculele acolo, apoi trecem rezultatele în baza b ; întrucât toate calculele de o cifră în baza b posibile sunt în număr de b^2 , este suficient să le facem pe toate o singură dată și să reținem rezultatele într-o tablă a operației respective în baza b .

Operații aritmetice într-o bază b

Exemplu: În baza 2, să se adune: 1011 + 110

Tabla adunării în baza 2 este:

+	0	1
0	0	1
1	1	10

De exemplu: $\bar{1} + \bar{1} = 1 + 1 = 2 = \overline{10}$.

Atunci, avem:

$$\begin{array}{r} 1\ 0\ 1\ 1\ + \\ 1\ 1\ 0 \\ \hline 1\ 0\ 0\ 0\ 1 \end{array}$$

Pe poziția unităților am avut $\bar{1} + \bar{0} = \bar{1}$, fără transport; pe următoarea poziție am avut $\bar{1} + \bar{1} = \overline{10}$ (adică numărul 2), s-a păstrat $\bar{0}$ și s-a propagat $\bar{1}$; pe următoarea poziție am avut $\bar{0} + \bar{1} + \bar{1}$ (ultimul $\bar{1}$ provenit din transport) = $\overline{10}$, s-a păstrat $\bar{0}$ și s-a propagat $\bar{1}$; etc.

Operații aritmetice într-o bază b

Exemplu: În baza 16, să se scadă: BA - 9B

Avem:

$$\begin{array}{r} B \ A \ - \\ 9 \ B \\ \hline 1 \ F \end{array}$$

Pe poziția unităților avem $\bar{A} - \bar{B} = 10 - 11 < 0$; de aceea, împrumutăm $\bar{1}$ de pe poziția următoare și atunci calculul este $1 \times 16 + 10 - 11 = 15 = \bar{F}$. Pe poziția următoare avem $\bar{B} - \bar{1} - \bar{9}$ (acel $\bar{1}$ a fost cedat la împrumut) $= 11 - 1 - 9 = 1 = \bar{1}$.

Operații aritmetice într-o bază b

Exemplu: În baza 2, să se înmulțească: 110.11 x 1.001

Avem:

$$\begin{array}{r} 1 \ 1 \ 0 \cdot 1 \ 1 \ x \\ 1 \cdot 0 \ 0 \ 1 \\ \hline 1 \ 1 \ 0 \ 1 \ 1 \\ 1 \ 1 \ 0 \ 1 \ 1 \\ \hline 1 \ 1 \ 1.1 \ 0 \ 0 \ 1 \ 1 \end{array}$$

Înmulțirea într-o bază oarecare se face tot după reguli asemănătoare ca în baza 10, dar pentru baza 2 aceste reguli se pot simplifica, deoarece singurele cifre sunt $\bar{0}$ și $\bar{1}$, care desemnează respectiv numerele 0 și 1, care sunt factor anulator, respectiv element neutru, la înmulțire; înmulțirea cu un $\bar{0}$ presupune scrierea unui rând de $\bar{0}$ -uri, care nu contează la adunare și se pot omite, iar înmulțirea cu un $\bar{1}$ revine la a scrie o copie a deînmulțitului; aşadar, pentru a face înmulțirea, este suficient să parcurgem înmulțitorul de la dreapta spre stânga și pentru fiecare $\bar{1}$ întâlnit să mai scriem o copie a deînmulțitului, cu cifra unităților aliniată la acel $\bar{1}$, iar în final să adunăm rândurile scrise - ceea ce am făcut mai sus; în final, numărul de zecimale ale produsului este suma numerelor de zecimale ale factorilor, la fel ca în cazul bazei 10.

Operații aritmetice într-o bază b

Exemplu: În baza 2, să se împartă: 10000.01 : 11

Avem:

$$\begin{array}{r} 1 \ 0 \ 0 \ 0 \ 0.0 \ 1 \mid 1 \ 1 \\ 1 \ 1 \qquad \qquad \qquad | \text{-----} \\ \text{-----} \qquad \qquad \qquad | \ 1 \ 0 \ 1.0 \ 1(1 \ 0) \\ = \ 1 \ 0 \ 0 \\ \qquad \qquad \qquad 1 \ 1 \\ \text{-----} \\ = \ 1 \ 0 \ 1 \\ \qquad \qquad \qquad 1 \ 1 \\ \text{-----} \\ = \ 1 \ 0 \ 0 \\ \qquad \qquad \qquad 1 \ 1 \\ \text{-----} \\ = \ 1 \ 0 \end{array}$$

Operații aritmetice într-o bază b

Când facem o împărțire, trebuie să urmărim dacă câtul rezultă ca o fracție zecimală finită sau se formează o perioadă. Pentru aceasta, după ce la deîmpărțit am trecut de virgulă și am coborât ultima cifră nenulă, reținem resturile succesive obținute într-o listă; dacă la un moment dat un rest este $\bar{0}$, ne oprim, și rezultat o fracție zecimală finită; dacă la un moment dat un rest se repetă, ne oprim, și rezultat o fracție zecimală periodică, iar perioada începe cu cifra generată la prima apariție a restului care s-a repetat.

În exemplul nostru, după ce la deîmpărțit am trecut de virgulă și am coborât ultima cifră nenulă, am obținut succesiv următoarele resturi: $\bar{10}, \bar{1}, \bar{10}$ (repetiție).

Cuprins

1 Performația calculatoarelor

Conceptul de performanță
Măsurarea performanței

2 Aritmetică sistemelor de calcul

Reprezentarea numerelor în matematică

Reprezentarea numerelor în calculator

Reprezentarea numerelor naturale ca întregi fără semn

Reprezentarea numerelor întregi în complement față de 2

Reprezentarea numerelor reale în virgulă mobilă

3 Logica pentru calculatoare

Algebre booleene

Funcții booleene

Particularizare la cazul B_2

4 Circuite logice

Circuite logice, Sisteme digitale

0-DS (Circuite combinaționale, Funcții booleene)

1-DS (Memorii)

2-DS (Automate finite)

Algoritmi de înmulțire și împărțire hardware

3-DS (Procesoare) și 4-DS (Calculatoare)

n -DS, $n > 4$

Reprezentarea în calculator

Fie $M \subseteq \mathbb{N}, \mathbb{Z}, \mathbb{R}$ o mulțime de numere pe care dorim să le reprezentăm în calculator.

Considerăm o locație de n biți: $b_{n-1} \dots b_0$ (de obicei n este multiplu de 8).

Notăm $C_n = \{c_{n-1} \dots c_0 | c_i \in \{0, 1\}\}$, mulțimea configurațiilor de 0, 1 pe n poziții.

O **reprezentare** este o funcție $[.]_n : M \longrightarrow C_n$ bijectivă.

Observații:

- M , n și $[.]_n$ trebuie alese a.î. $[.]_n$ să fie bijectivă, deoarece noi reprezentăm numerele în calculator, operăm asupra lor cu calculatorul, dar la sfârșit dorim să putem interpreta rezultatul (să aflăm ce număr are reprezentarea respectivă), ori aceasta înseamnă să aplicăm $[.]_n^{-1}$;
- $[.]_n$ nu poate fi $(.)_2$ (reprezentarea binară folosită în matematică), deoarece avem valori de biți pentru a desemna doar 0 și 1, nu și $-$, \cdot , etc.; deci, trebuie facută o altă codificare;
- în plus, regula de reprezentare prin care este definită $[.]_n$ trebuie aleasă a.î. să permită o implementare ușoară și eficientă a unor operații hardware pe C_n prin care să traducem operațiile aritmetice pe M .

Reprezentarea în calculator

Considerăm următoarele operații hardware la nivel de bit:

a	b	$a b$ (or)	$a\&b$ (and)	$a \wedge b$ (xor)	a	$\sim a$ (not)
0	0	0	0	0	0	1
0	1	1	0	1	1	0
1	0	1	0	1		
1	1	1	1	0		

Reprezentarea în calculator

Pe C_n putem considera mai multe operații hardware (lor le corespund instrucțiuni în limbajul mașină sau de asamblare), de exemplu:

- Operații logice binare pe biți: \oplus (or), \otimes (and), \ominus (xor), etc.; ele se efectuează aplicând bit cu bit operațiile logice respective, conform tabelului anterior.

Exemplu ($n = 8$):
$$\begin{array}{r} 11000101 \\ \otimes \\ 01000110 \\ \hline 01000100 \end{array}$$

- Operații aritmetice binare: \oplus (adunare), \ominus (scădere), etc.; ele se efectuează aplicând algoritmii de calcul cu reprezentările binare ale numerelor naturale folosiți în matematică (ilustrați în secțiunea precedentă); un transport din poziția b_{n-1} se pierde, iar un împrumut cerut la această poziție se primește automat (din "afară").

Exemplu ($n = 8$):
$$\begin{array}{r} 11000101 \\ \oplus \\ 01000110 \\ \hline 00001011 \end{array} \quad \begin{array}{r} 00001011 \\ \ominus \\ 01000110 \\ \hline 11000101 \end{array}$$

(în primul caz am avut un transport din poziția 7, care s-a pierdut, iar în al doilea caz s-a cerut un împrumut la poziția 6, cererea s-a transmis la poziția 7, unde împrumutul s-a primit automat din "afară", iar când acesta s-a transmis la poziția 6, unde a fost generată cererea, a lasat în urmă un 1 pe poziția 7).

Reprezentarea în calculator

- operații logice unare: \neg (complement față de 1), \boxminus (complement față de 2);
 - $\neg c_{n-1} \dots c_0$ se efectuează negând fiecare bit c_i , conform tabelului anterior;
 - $\boxminus c_{n-1} \dots c_0$ se efectuează negând fiecare bit c_i și apoi adunând 1 aritmetic (cu eventuala propagare a transportului);

Exemplu ($n = 8$): $\neg \frac{01001100}{10110011}$ $\boxminus \frac{01001100}{10110100}$

Observație: $\boxminus c_{n-1} \dots c_0$ se poate obține mai simplu astfel:

- dacă $c_{n-1} = \dots = c_0 = 0$, nu se schimbă nimic;
- altfel, dacă $k = \min\{i \in \{0, \dots, n-1\} | c_i = 1\}$, atunci biții c_{n-1}, \dots, c_{k+1} se neagă iar biții c_k, \dots, c_0 se lasă neschimbați.

Reprezentarea în calculator

Se constată că operațiile hardware pe C_n considerate nu sunt independente logic (se pot exprima unele prin altele):

Teoremă: Pentru orice $\alpha, \beta \in C_n$ avem:

$$\neg \alpha = (\neg \alpha) \oplus 1$$

$$\alpha \ominus \beta = \alpha \oplus (\neg \beta) = \alpha \oplus ((\neg \beta) \oplus 1)$$

(am notat prin 1 configurația 0...01 $\in C_n$).

Exemplu: Într-un exemplu anterior am calculat că $00001011 \ominus 01000110 = 11000101$; să verificăm că obținem același rezultat și cu formula pentru \ominus din teorema:

Dacă $\alpha = 00001011$ și $\beta = 01000110$, atunci:

$$\neg \beta = 10111001$$

$$(\neg \beta) \oplus 1 = 10111001 \oplus 00000001 = 10111010$$

$$\alpha \oplus ((\neg \beta) \oplus 1) = 00001011 \oplus 10111010 = 11000101$$

Reprezentarea în calculator

De aceea, \oplus și \ominus , deși există ca instrucțiuni diferite în limbajul mașină sau de asamblare, intern sunt realizate cu un același circuit, numit **sumator**.

La fel, există instrucțiuni mașină pentru aplicarea algoritmilor de înmulțire și împărțire, care intern sunt realizate prin combinații de aceleași câteva circuite simple (de exemplu, înmulțirea = adunare repetată, împărțirea = scădere repetată).

Operațiile din limbajele de nivel înalt sunt traduse de compilator prin combinații de operații implementate hardware. Cu cât limbajul este de nivel mai jos (i.e. mai apropiat de hardware), cu atât combinația este mai simplă.

De exemplu, în limbajul C există operatorii `|`, `&`, `^`, ..., iar ei pot fi trăduși prin câte o singură operație `⊕`, `⊗`, `⊖`, ... (de aceea am notat așa).

Reprezentarea în calculator

În continuare, vom studia algoritmi de reprezentare $[.]_n$ pentru diverse tipuri de numere.

Pentru un număr oarecare x , vom nota $(x)_2^n$ reprezentarea lui x în baza 2 folosită în matematică (a se vedea secțiunea anterioară), completată cu cifre semnificative 0 până la n poziții.

Cuprins

1 Performația calculatoarelor

Conceptul de performanță
Măsurarea performanței

2 Aritmetică sistemelor de calcul

Reprezentarea numerelor în matematică
Reprezentarea numerelor în calculator
Reprezentarea numerelor naturale ca întregi fără semn
Reprezentarea numerelor întregi în complement față de 2
Reprezentarea numerelor reale în virgulă mobilă

3 Logica pentru calculatoare

Algebre booleene
Funcții booleene
Particularizare la cazul B_2

4 Circuite logice

Circuite logice, Sisteme digitale
0-DS (Circuite combinaționale, Funcții booleene)
1-DS (Memorii)
2-DS (Automate finite)
Algoritmi de înmulțire și împărțire hardware
3-DS (Procesoare) și 4-DS (Calculatoare)
 n -DS, $n > 4$

Reprezentarea ca întreg fără semn

În limbajul C, reprezentarea ca întreg fără semn se folosește în cazul tipurilor întregi însotite de 'unsigned':

`unsigned char, unsigned short, unsigned int,
unsigned long, unsigned long long`

Reprezentarea ca întreg fără semn pe n poziții se definește astfel:

$$M := \{0, \dots, 2^n - 1\} \subseteq \mathbb{N}$$

$$[.]_n := [.]_n^u : \{0, \dots, 2^n - 1\} \longrightarrow C_n, \quad [x]_n^u = (x)_2^n.$$

Exemplu: În limbajul C (versiunea compilatorului TC++ 1.00) avem:

```
unsigned short x, y, z;  
/* n = 16, M = {0, ..., 65535} */  
scanf("%hu%hu", &x, &y);  
/* aplică [.]16u */  
z = x + y;  
/* compilatorul traduce + prin ⊕ */  
printf("%hu", z);  
/* aplică [.]16u-1 */
```

Reprezentarea ca întreg fără semn

Din modul cum se fac operațiile implementate hardware discutat mai devreme, rezultă că operațiile cu numerele naturale reprezentate astfel se fac $\text{mod } 2^n$, mai exact:

Teoremă:

Dacă $op = +, -, \dots$ atunci: $[[x]_n^u \circledcirc [y]_n^u]_n^{u-1} = (x op y) \text{ mod } 2^n$

(" $\text{mod } 2^n$ " se calculează a.î. să producă un rezultat $\in \{0, \dots, 2^n - 1\}$).

În particular, dacă $x op y \in M$, atunci: $[[x]_n^u \circledcirc [y]_n^u]_n^{u-1} = x op y$.

Așadar, dacă reprezentăm x și y și aplicăm \circledcirc din calculator, obținem reprezentarea lui $x op y$ din matematică modulo 2^n ; dacă nu avem depășire, este chiar $x op y$ din matematică.

Reprezentarea ca întreg fără semn

În particular, vom avea:

$$\begin{aligned} \text{Dacă } x, y, x - y \in M, \text{ atunci: } & [[x]_n^u \ominus [y]_n^u]_n^{u-1} = x - y, \\ \text{adică: } & [[x]_n^u \oplus (\neg [y]_n^u)]_n^{u-1} = x - y \\ \text{sau: } & [[x]_n^u \oplus ((\neg [y]_n^u) \oplus 1)]_n^{-1} = x - y \end{aligned}$$

De asemenea, vom avea:

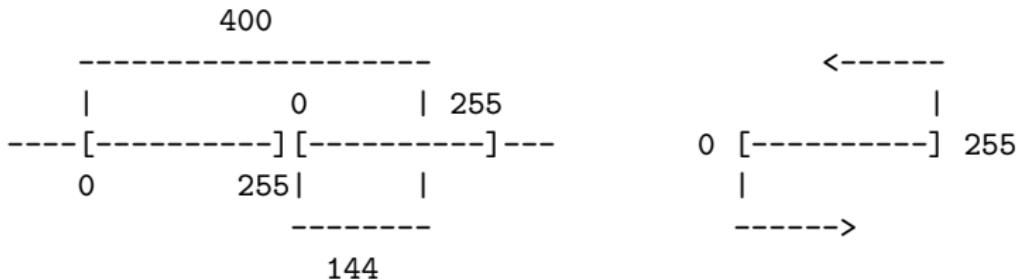
$$\begin{aligned} \text{Dacă } x \in M, \text{ atunci: } & [\neg [x]_n^u]_n^{u-1} = -x + 2^n, \\ \text{adică: } & [(\neg [x]_n^u) \oplus 1]_n^{u-1} = -x + 2^n. \end{aligned}$$

Reprezentarea ca întreg fără semn

Exemplu: În limbajul C considerăm declarațiile:

```
unsigned char x,y,z; /* n = 8, M = {0, ..., 255} */
```

- 1) Secvența de cod: `x = 200; y = 200; z = x + y; printf("%d",z);`
afișază: 144



Intuitiv, ne imaginăm că intervalul $[0, \dots, 255]$ se multiplică de-a lungul axei reale, determinăm punctul aflat la distanța 400 de origine, apoi măsurăm deplasamentul acestui punct față de începutul intervalului în care se află, obținând rezultatul 144 (desenul din stânga).

Sau, ne imaginăm că valoarea 400 se "înfășoară" parcurgând circular intervalul $[0, \dots, 255]$ și ce rămâne în interval este rezultatul 144 (desenul din dreapta).

Reprezentarea ca întreg fără semn

- 2) Secvența de cod: `x = 255; ++x;` face ca x să devină 0
Secvența de cod: `x = 0; --x;` face ca x să devină 255

- 3) Secvența de cod: `for(x = 0; x < 256; ++x);` este un ciclu infinit.
Într-adevăr, x primește succesiv valorile 0, 1, 2, ..., 255, 0, 1, ..., deci vom avea mereu $x < 256$.
De notat că la efectuarea testului $x < 256$ operanții sunt convertiți la tipul int, iar comparația se face în cadrul tipului int.

Cuprins

1 Performația calculatoarelor

Conceptul de performanță
Măsurarea performanței

2 Aritmetică sistemelor de calcul

Reprezentarea numerelor în matematică
Reprezentarea numerelor în calculator
Reprezentarea numerelor naturale ca întregi fără semn
Reprezentarea numerelor întregi în complement față de 2
Reprezentarea numerelor reale în virgulă mobilă

3 Logica pentru calculatoare

Algebre booleene
Funcții booleene
Particularizare la cazul B_2

4 Circuite logice

Circuite logice, Sisteme digitale
0-DS (Circuite combinaționale, Funcții booleene)
1-DS (Memorii)
2-DS (Automate finite)
Algoritmi de înmulțire și împărțire hardware
3-DS (Procesoare) și 4-DS (Calculatoare)
 n -DS, $n > 4$

Reprezentarea în complement față de 2

În limbajul C, reprezentarea în complement față de 2 se folosește în cazul tipurilor întregi ne-însoțite de 'unsigned':

char, signed char, short, signed short, int, signed int,
long, signed long, long long, signed long long

Reprezentarea în complement față de 2 pe n poziții se definește astfel:

$$M := \{-2^{n-1}, \dots, 2^{n-1} - 1\} \subseteq \mathbb{Z}$$

$$[.]_n := [.]_n^s : \{-2^{n-1}, \dots, 2^{n-1} - 1\} \longrightarrow C_n, \quad [x]_n^s = \begin{cases} (x)_2^n, & x \geq 0 \\ (2^n + x)_2^n, & x < 0 \end{cases}$$

(această funcție s.n.**codul complementar față de 2**).

Observație: compact, putem scrie: $\forall x \in M, [x]_n^s = ((2^n + x) \bmod 2^n)_2^n$.

Reprezentarea în complement față de 2

Exemplu: În limbajul C, pentru tipul `char` avem: $n = 8$,
 $M = \{-128, \dots, 127\}$.

Ilustrăm modul de reprezentare a câtorva valori:

$$127: 01111111 = (127)_2^n$$

$$126: 01111110 = (126)_2^n$$

...

$$1: 00000001 = (1)_2^n$$

$$0: 00000000 = (0)_2^n$$

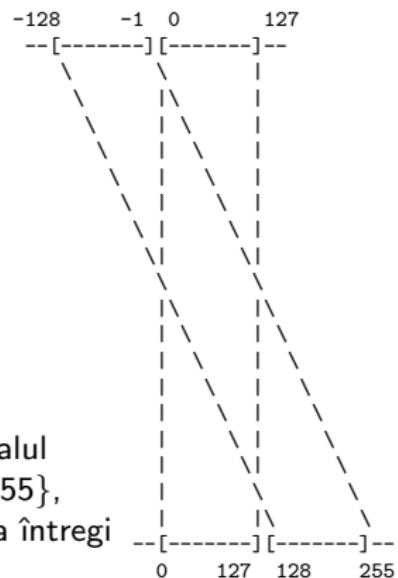
$$-1: 11111111 = (255)_2^n = (256 - 1)_2^n$$

$$-2: 11111110 = (254)_2^n = (256 - 2)_2^n$$

...

$$-127: 10000001 = (129)_2^n = (256 - 127)_2^n$$

$$-128: 10000000 = (128)_2^n = (256 - 128)_2^n$$



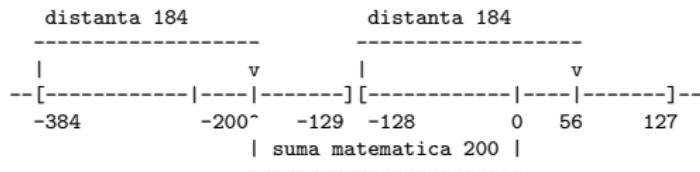
Deci, intervalul $\{0, \dots, 127\}$ este lăsat pe loc, intervalul $\{-128, \dots, -1\}$ este mapat în intervalul $\{128, \dots, 255\}$, apoi numerele naturale rezultate sunt reprezentate ca întregi fără semn.

Reprezentarea în complement față de 2

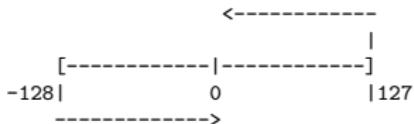
Operațiile aritmetice se fac tot mod 2^n , dar translatat (deplasamentul se măsoară față de începutul intervalului $\{-2^{n-1}, \dots, 2^{n-1} - 1\}$).

Exemplu: În limbajul C, avem:

```
char x, y, z;          /* n = 8, M = {-128, ..., 127} */
x = -100; y = -100; z = x + y;
printf("%d", (int)z); /* afisaza: 56 */
```



Ne putem imagina că intervalul $\{-128, \dots, 127\}$ este parcurs circular:



În particular: $-128 - 1$ produce 127, $127 + 1$ produce -128 .

Reprezentarea în complement față de 2

Teoremă:

- (1) Dacă $[x]_n^s = c_{n-1} \dots c_0$, atunci: $x \geq 0 \Leftrightarrow c_{n-1} = 0$
(c_{n-1} s.n. **bitul de semn**)
- (2) Dacă $x, -x \in M$, atunci $[-x]_n^s = \neg[x]_n^s = (\neg[x]_n^s) \oplus 1$
- (3) Dacă $x, y, x - y \in M$, atunci $[x - y]_n^s = [x]_n^s \oplus (\neg[y]_n^s) = [x]_n^s \oplus ((\neg[y]_n^s) \oplus 1)$
- (4) Dacă $x \in \{-2^{p-1}, 2^{p-1} - 1\}$, $[x]_p^s = c_{p-1} \dots c_0$ și $p < q$, atunci:
 $[x]_q^s = \underbrace{c_{p-1} \dots c_{p-1}}_{q-p \text{ ori}} c_{p-1} \dots c_0$ (se extinde bitul de semn).

Reprezentarea în complement față de 2

Exemplu: Fie $x = 32$, $y = 41$. Calculați $z = x - y$ folosind reprezentarea în complement față de 2 pe 8 biți.

Avem $n = 8$, deci $M = \{-128, \dots, 127\}$.

Deoarece $x, y \geq 0$, avem $[x]_8^s = (x)_2^8$, $[y]_8^s = (y)_2^8$.

Pentru $(x)_2^8$ observăm că $32 = 2^5$, iar pentru $(y)_2^8$ aplicăm regula de conversie baza 10 → baza 2 din prima secțiune:

$$\begin{array}{r} 41 \\ 20 \\ 10 \\ 5 \\ 2 \\ 1 \\ 0 \end{array}$$

Atunci:

$$[y]_8^s = 00101001$$

$$\neg[y]_8^s = 11010110$$

$$1 \oplus (\neg[y]_8^s) = 11010111$$

$$[x]_8^s = 00100000$$

$$x \oplus (1 \oplus (\neg[y]_8^s)) = 11110111$$

Avem $11110111 = (247)_2^8$ (aplicând regulile de conversie baza 2 → baza 10 din prima secțiune) $= [z]_8^s$; deoarece bitul 7 (de semn) din $[z]_8^s$ este 1, înseamnă că $z < 0$, deci $[z]_8^s$ s-a obținut cu al doilea caz al definiției; aşadar

$[z]_8^s = (256 + z)_2^8 = (247)_2^8$, de unde (aplicând inversa funcției $(.)_2^8$) rezultă că $256 + z = 247$, adică $z = -9$.

Putem verifica, efectuând scăderea în baza 10, că $32 - 41 = -9$.

Reprezentarea în complement față de 2

Observație: Dacă efectuăm $\square[x]_n^s$, adică $(\square[x]_n^s) \oplus 1$, pentru $x = -2^{n-1}$, vom obține tot $[x]_n^s$, ceea ce, cu punctul (2) din teorema anterioară, pare să însemne că $-(-2^{n-1}) = -2^{n-1}$ (deoarece au aceeași reprezentare).

De exemplu, pentru $n = 8$ vom obține că $-(-128) = -128$.

Într-adevăr, avem succesiv:

$$[-128]_8^s = 10000000 \xrightarrow{\square} 01111111 \xrightarrow{\oplus 1} 10000000 = [-128]_8^s.$$

Totuși, nu există nici o contradicție aritmetică sau cu bijectivitatea lui $[.]_n^s$, deoarece formula de la punctul (2) din teorema anterioară se aplică doar dacă $x, -x \in M$, iar în acest caz $-x \notin M$.

Reprezentarea în complement față de 2

Exemplu: În limbajul C (TC++ 1.00) considerăm secvența de cod:

```
char x; /* p = 8 */
short y; /* q = 16 */
x = -127; /* se reprezinta -127 ca intreg cu semn pe 8 biti */
y = x; /* se extinde reprezentarea din x de la 8 biti la 16 biti,
prin propagarea bitului de semn */
```

Verificați că valoarea reprezentată în y este tot -127 (punctul (4) din teorema anterioară).

Avem: $[-127]_8^s = (256 - 127)_2^8$ (deoarece $-127 < 0$) $= (129)_2^8 = 10000001$.

Prin propagarea bitului de semn la 16 biți, obținem: $111111110000001 = ((2^{16} - 1) - (2^7 - 1) + 1)_2^{16} = (65535 - 127 + 1)_2^{16} = (65409)_2^{16}$

Aceasta este reprezentarea ca întreg cu semn a unui număr $z < 0$, deoarece bitul 15 (de semn) este 1.

Atunci $[z]_{16}^s = (65536 + z)_2^{16} = (65409)_2^{16}$, deci $65536 + z = 65409$, deci $z = -127$.

Cuprins

1 Performația calculatoarelor

Conceptul de performanță
Măsurarea performanței

2 Aritmetică sistemelor de calcul

Reprezentarea numerelor în matematică
Reprezentarea numerelor în calculator
Reprezentarea numerelor naturale ca întregi fără semn
Reprezentarea numerelor întregi în complement față de 2
Reprezentarea numerelor reale în virgulă mobilă

3 Logica pentru calculatoare

Algebre booleene
Funcții booleene
Particularizare la cazul B_2

4 Circuite logice

Circuite logice, Sisteme digitale
0-DS (Circuite combinaționale, Funcții booleene)
1-DS (Memorii)
2-DS (Automate finite)
Algoritmi de înmulțire și împărțire hardware
3-DS (Procesoare) și 4-DS (Calculatoare)
 n -DS, $n > 4$

Notația științifică

Notația științifică și aritmetica sistemelor de calcul în virgula mobilă folosesc un mod de reprezentare a unei aproximății a unui număr real care să suporte un domeniu larg de valori. Numerele sunt, în general, reprezentate aproximativ printr-o valoare cu un număr fixat de cifre semnificative care este scalată cu ajutorul unui exponent:

$$\pm a \times b^e$$

a este **mantisa** (engl: **mantissa, significand, coefficient**), **b** este **baza** de enumerație în care sunt scrise numerele și baza scalării (de obicei 2, 10 sau 16), **e** este **exponentul**; semnul '+' se poate omite.

Notația științifică

Scalarea cu ajutorul exponentului permite acoperirea unui domeniu larg de valori, atât foarte mici cât și foarte mari, cu aceeași precizie relativă. De exemplu cu 9 cifre semnificative putem descrie, în funcție de exponenții folosiți, atât distanțe în astronomie cu precizie de 1 an lumină, cât și distanțe în fizica atomică cu precizie de 1 femtometru (10^{-15} metri) - ambele precizii fiind acceptabile în domeniul respectiv.

Faptul că se reține un număr fix de cifre pentru mantisă și exponent face ca mulțimea numerelor reale reprezentabile să fie de fapt o mulțime finită de numere raționale, dintr-un interval fixat.

Mai mult, folosirea scalării face ca aceste numere să nu fie uniform spațiate în intervalul respectiv: valorile mici, apropiate de 0, sunt reprezentate cu pași mici (ex. 1 femtometru), în timp ce valorile mari, apropiate de extremele intervalului, sunt reprezentate cu pași mari (ex. 1 an lumină).

Denumirea de **virgulă mobilă** se referă la faptul că **virgula** (engl: **radix point**), numită în funcție de bază și **virgulă zecimală** (engl: **decimal point**), **virgulă binară** (engl: **binary point**), etc., poate fi mutată oriunde în raport cu cifrele semnificative, modificând corespunzător exponentul:

$$0.123 \times 10^2 = 1.230 \times 10^1 = 12.30 \times 10^0 = 123.0 \times 10^{-1}$$

Notația științifică

În informatică, termenul de **virgulă mobilă** (engl: **floating point**) denumește aritmetica sistemelor de calcul ce folosește reprezentări ale numerelor pentru care virgula binară nu este fixată.

O reprezentare în virgulă mobilă este **notație științifică** dacă are o singură cifră în stânga virgulei și este **notație științifică normalizată** dacă în plus acea cifră este nenulă (i.e nu are 0-uri la început).

De exemplu 1.230×10^1 este în notație științifică normalizată, dar 0.123×10^2 , 12.30×10^0 , 123.0×10^{-1} nu sunt.

Constatăm că notația științifică normalizată pentru un număr real nenul este unică.

Formatul intern în virgulă mobilă

Reprezentarea în calculator a numerelor reale în virgulă mobilă este reglementată prin standardul **IEEE 754**, adoptat inițial în 1985 (**IEEE 754-1985**) și reactualizat în 2008 (**IEEE 754-2008**). Mai nou, este folosit standardul **ISO/IEC/IEEE 60559:2011**.

Pentru reprezentare (formatul intern) se aleg două dimensiuni n și k , $2 \leq k \leq n - 2$ și se folosesc locații de n biți $b_{n-1} \dots b_0$ compuse din următoarele câmpuri:

- b_{n-1} (1 bit, cel mai semnificativ): semnul;
- $b_{n-2} \dots b_{n-k-1}$ (următorii k biți): caracteristica;
- $b_{n-k-2} \dots b_0$ (ultimii $n-k-1$ biți, cei mai puțin semnificativi): fracție;

$$\underbrace{b_{n-1}}_{\text{semn (1 bit)}} \underbrace{b_{n-2} \dots b_{n-k-1}}_{\text{caracteristică (k biți)}} \underbrace{b_{n-k-2} \dots b_0}_{\text{fracție (n - k - 1 biți)}}$$

Formatul intern în virgulă mobilă

Mulțimea valorilor reprezentabile în acest format este specificată prin intermediul a trei parametri întregi (baza de enumerație și de scalare este întotdeauna 2):

- $p = n - k$: numărul de cifre ale mantisei (precizia);
- $E_{\min} = -2^{k-1} + 2$: exponentul minim;
- $E_{\max} = 2^{k-1} - 1$: exponentul maxim.

Pentru numerele reale nenule reprezentabile, exponentul E trebuie să se afle în intervalul de numere întregi $E_{\min} \leq E \leq E_{\max}$, dar se dorește reprezentarea lui ca întreg fără semn, nu prin complement față de doi; de aceea, se mai consideră parametrul:

- $bias = 2^{k-1} - 1$: bias (sau polarizare);

iar în câmpul de k biți se reprezintă, ca întreg fără semn, valoarea $c = E + bias$ (avem $1 \leq c \leq 2^k - 2$); aceasta s.n. caracteristică (sau exponent biasat sau exponent polarizat).

Formatul intern în virgulă mobilă

Constatăm că în câmpul caracteristică al formatului pot fi stocate și valorile $c = 0$ (reprezentată cu toți biții 0) și $c = 2^k - 1$ (reprezentată cu toți biții 1), care nu pot fi produse de expresiile $E + bias$; ele sunt folosite pentru a codifica valori speciale, ca ± 0 , $\pm \infty$, numere denormalizate mici, valori invalide **NaN** (Not A Number).

În ceea ce privește mantisa, numerele vor fi mai întâi scalate a.î. mantisa să aibă în stânga virgulei doar o cifră (1 în cazul numerelor normalize și 0 în cazul numerelor denormalizate), iar în ultimii $n - k - 1$ biți ai formatului va fi stocată **fracția**, care este partea mantisei din dreapta virgulei (cifra din stânga putându-se deduce din context). În unele texte, prin 'mantisă' este denumită de fapt 'fracția'.

Se mai spune că această reprezentare este **semn și modul** (engl: **sign and magnitude**) deoarece semnul are un bit separat de restul numărului.

Formatul intern în virgulă mobilă

Valorile reprezentabile în formatul specificat de n și k (sau de p , E_{min} , E_{max}) și modul lor de reprezentare (codare) sunt următoarele:

- Numerele de forma $(-1)^s \times 2^E \times \overline{1.c_1 \dots c_{p-1}}_2$, unde
 $s \in \{0, 1\}$,
 $E_{min} \leq E \leq E_{max}$ este un număr întreg,
 $c_1, \dots, c_{p-1} \in \{0, 1\}$
(numere reale nenule normalize).

Ele sunt reprezentate astfel:

$$b_{n-1} = s, \text{ semnul};$$

$b_{n-2} \dots b_{n-k-1} = E + bias$, exponentul biasat,
reprezentat ca întreg fără semn; sirul de biți prin care se reprezintă variază
de la $0 \dots 01$ (valoarea 1) la $1 \dots 10$ (valoarea $2^k - 2$);

$$b_{n-k-2} \dots b_0 = c_1 \dots c_{p-1}, \text{fracția (se omite deci } c_0 = 1).$$

Formatul intern în virgulă mobilă

- Numerele de forma $(-1)^s \times 2^{E_{\min}} \times \overline{0.c_1 \dots c_{p-1}}_2$, unde $s \in \{0, 1\}$,
 $c_1, \dots, c_{p-1} \in \{0, 1\}$ și cel puțin unul dintre c_i este 1
(numere reale nenule denormalizate mici).

Ele sunt reprezentate astfel:

$b_{n-1} = s$, semnul;

$b_{n-2} \dots b_{n-k-1} = 0$, reprezentat prin sirul de biți 0 ... 0;

$b_{n-k-2} \dots b_0 = c_1 \dots c_{p-1}$, frația (se omite deci $c_0 = 0$).

Formatul intern în virgulă mobilă

- Numerele de forma $\pm 0 = (-1)^s \times 2^{E_{\min}} \times \overline{0.0\dots 0}_2$, unde $s \in \{0, 1\}$.

Ele sunt reprezentate astfel:

$$b_{n-1} = s, \text{ semnul};$$

$$b_{n-2} \dots b_{n-k-1} = 0, \text{ reprezentat prin sirul de biti } 0 \dots 0;$$

$$b_{n-k-2} \dots b_0 = 0 \dots 0, \text{ fractia}.$$

Notăm că numerele de forma ± 0 se reprezintă cu toți bitii 0, mai puțin eventual bitul de semn b_{n-1} .

Deși reprezentările lui $+0$ și -0 sunt diferite (prin bitul de semn), semnul are relevanță în anumite circumstanțe, cum ar fi împărțirea la 0 (împărțirea $1.0 / +0$ produce $+\infty$, împărțirea $1.0 / -0$ produce $-\infty$), iar în altele nu.

Formatul intern în virgulă mobilă

- Valorile $\pm\infty$.

Ele sunt reprezentate astfel:

$$b_{n-1} = s, \text{ semnul } (0 = +, 1 = -);$$

$$b_{n-2} \dots b_{n-k-1} = 2^k - 1, \text{ reprezentat prin sirul de biti } 1 \dots 1;$$
$$b_{n-k-2} \dots b_0 = 0 \dots 0, \text{ fractia nula.}$$

- Valori ***NaN*** (Not A Number) - sunt entități simbolice codificate în format de virgulă mobilă ce semnifică ideea de valoare invalidă.

Ele sunt reprezentate astfel:

$$b_{n-1} \text{ este oarecare;}$$

$$b_{n-2} \dots b_{n-k-1} = 2^k - 1, \text{ reprezentat prin sirul de biti } 1 \dots 1;$$
$$b_{n-k-2} \dots b_0 \text{ este o fractie nenula (cel putin unul dintre } b_{n-k-2}, \dots, b_0 \text{ este 1).}$$

Formatul intern în virgulă mobilă

Există două feluri de NaN :

Signaling NaN (sNaN): semnalează (declanșază) excepția de operație invalidă de fiecare dată când apare ca operand într-o operație; manifestarea excepției poate difera de la o implementare la alta și poate fi de ignorare sau poate consta în setarea unor flag-uri și/sau executarea unui trap (întrerupere) care lansează o rutină ce primește NaN -ul ca parametru;

Quiet NaN (qNaN): se propagă prin aproape toate operațiile aritmetice fără a semnaliza (declanșa) excepții, iar rezultatul operațiilor va fi tot un $qNaN$, anume unul dintre $qNaN$ -urile date ca operand; ele sunt utile atunci când nu vrem să detectăm/tratăm o eroare chiar la momentul apariției ei ci la un moment ulterior.

Standardul IEEE 754 cere a fi implementat cel puțin un $sNaN$ și cel puțin un $qNaN$; bitul de semn și biții de fracție pot difera de la o implementare la alta - de exemplu în ei se poate codifica motivul erorii (în cazul $sNaN$ -urilor el va fi transmis astfel rutinei de tratare a excepției, iar în cazul $qNaN$ -urilor el se va propaga prin operațiile aritmetice până la locul tratării).

Formatul intern în virgulă mobilă

Aplicând invers regulile de mai sus, putem determina valoarea V reprezentată printr-un sir de biți $b_{n-1} \dots b_0$ în formatul specificat de n și k . Notăm:

$s =$ numărul 0 sau 1 stocat în bitul b_{n-1} (semnul);

$c =$ numărul natural reprezentat ca întreg fără semn în biții $b_{n-2} \dots b_{n-k-1}$ (caracteristica);

$f =$ sirul de $p - 1$ cifre binare stocat în biții $b_{n-k-2} \dots b_0$ (fracția).

- Dacă $c = 2^k - 1$ (reprezentat ca 1...1) și $f \neq 0 \dots 0$, atunci V este un NaN (indiferent de s).

- Dacă $c = 2^k - 1$ (reprezentat ca 1...1) și $f = 0 \dots 0$, atunci $v = (-1)^s \times \infty$.

- Dacă $0 < c < 2^k - 1$, atunci

$v = (-1)^s \times 2^{c-2^{k-1}+1} \times \overline{1.f}_2$ (număr nenul normalizat).

- Dacă $c = 0$ (reprezentat ca 0...0) și $f \neq 0 \dots 0$, atunci

$v = (-1)^s \times 2^{-2^{k-1}+2} \times \overline{0.f}_2$ (număr nenul denormalizat).

- Dacă $c = 0$ (reprezentat ca 0...0) și $f = 0 \dots 0$, atunci

$v = (-1)^s \times 0$ (zero).

Formatul intern în virgulă mobilă

Din aceste reguli deducem că mulțimea numerelor reprezentabile este inclusă în intervalul:

$$[-2^{2^{k-1}-n+k} \times (2^{n-k} - 1), +2^{2^{k-1}-n+k} \times (2^{n-k} - 1)]$$

Nu toate numerele reale din acest interval pot fi însă reprezentate, deoarece se rețin doar un număr finit de zecimale în baza 2. Numerele sunt reprezentate cu un anumit pas, care crește cu cât ne apropiem de capetele intervalului.

Operațiile aritmetice în virgulă mobilă rotunjesc (și deci altereză) rezultatul a.î. să fie una dintre valorile reprezentabile. De aceea, pasul reprezentării influențează precizia calculelor.

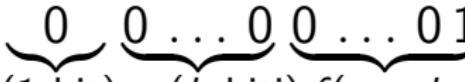
Întrucât mulțimea valorilor reprezentabile este simetrică față de 0 (dacă un număr x este reprezentabil, atunci și $-x$ este reprezentabil, reprezentarea diferind doar în bitul de semn), vom analiza doar intervalul:

$$[0, 2^{2^{k-1}-n+k} \times (2^{n-k} - 1)]$$

Formatul intern în virgulă mobilă

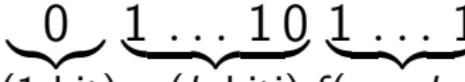
Cel mai mic număr nenul reprezentabil este:

$$2^{-2^{k-1}+2} \times \underbrace{0.0\dots01}_\text{$n-k$ cifre ale mantisei}_2 = 2^{-2^{k-1}-n+k+3},$$

reprezentat: 
s (1 bit) c (k biți) f($n - k - 1$ biți)

Cel mai mare număr nenul reprezentabil este:

$$2^{2^{k-1}-1} \times \underbrace{1.1\dots1}_\text{$n-k$ cifre ale mantisei}_2 = 2^{2^{k-1}-n+k} \times (2^{n-k} - 1),$$

reprezentat: 
s (1 bit) c (k biți) f($n - k - 1$ biți)

Formatul intern în virgulă mobilă

Numerele din intervalul $(0, 2^{-2^{k-1}-n+k+3})$ sunt prea mici pentru a fi reprezentate.

Numerele din intervalul $[2^{-2^{k-1}-n+k+3}, 2^{-2^{k-1}+3}]$ sunt reprezentate cu pasul $2^{-2^{k-1}-n+k+3}$.

Pentru orice $i \in \overline{-2^{k-1} + 3, 2^{k-1} - 2}$, numerele din intervalul $[2^i, 2^{i+1}]$ sunt reprezentate cu pasul $2^{i-n+k+1}$.

Numerele din intervalul $[2^{2^{k-1}-1}, 2^{2^{k-1}-n+k} \times (2^{n-k} - 1)]$ sunt reprezentate cu pasul $2^{2^{k-1}-n+k}$.

Formatul intern în virgulă mobilă

Ilustrare grafică:

$$\underbrace{0}_{s(1)} \underbrace{0 \dots 0}_{c(k)} \underbrace{0 \dots 0}_{f(n-k-1)} = 0$$

∴ (numere nereprezentabile)

$$\underbrace{0}_{s(1)} \underbrace{0 \dots 0}_{c(k)} \underbrace{0 \dots 01}_{f(n-k-1)} = 2^{-2^{k-1}+2} \times \overline{0.0 \dots 01}_2 = 2^{-2^{k-1}+2} \times 2^{-n+k+1}$$

(cel mai mic număr nenul reprezentabil)

∴ pas $2^{-2^{k-1}-n+k+3}$

$$\begin{aligned} \underbrace{0}_{s(1)} \underbrace{0 \dots 0}_{c(k)} \underbrace{1 \dots 1}_{f(n-k-1)} &= 2^{-2^{k-1}+2} \times \overline{0.1 \dots 1}_2 \\ &= 2^{-2^{k-1}+2} \times (2^{n-k-1} - 1) \times 2^{-n+k+1} \\ &= 2^{-2^{k-1}-n+k+3} \times (2^{n-k-1} - 1) \end{aligned}$$

$$\begin{aligned} \text{pas } 2^{-2^{k-1}-n+k+3} \\ \underbrace{0}_{s(1)} \underbrace{0 \dots 01}_{c(k)} \underbrace{0 \dots 0}_{f(n-k-1)} &= 2^{-2^{k-1}+2} \times \overline{1.0 \dots 0}_2 = 2^{-2^{k-1}+2} \end{aligned}$$

∴ pas $2^{-2^{k-1}-n+k+3}$

Formatul intern în virgulă mobilă

$$\begin{array}{ccccccc} \underbrace{0}_{s(1)} & \underbrace{0 \dots 0}_{c(k)} & 1 & \underbrace{1 \dots 1}_{f(n-k-1)} & = & 2^{-2^{k-1}+2} \times \overline{1.1 \dots 1}_2 \\ & & & & & = 2^{-2^{k-1}+2} \times (2^{n-k}-1) \times 2^{-n+k+1} \\ & & & & & = 2^{-2^{k-1}-n+k+3} \times (2^{n-k}-1) \\ & & \text{pas } 2^{-2^{k-1}-n+k+3} & & & & \\ \underbrace{0}_{s(1)} & \underbrace{0 \dots 0}_{c(k)} & 10 & \underbrace{0 \dots 0}_{f(n-k-1)} & = & 2^{-2^{k-1}+2} \times \overline{10.0 \dots 0}_2 = 2^{-2^{k-1}+3} \times \overline{1.0 \dots 0}_2 \\ & & & & & = 2^{-2^{k-1}+3} \\ \vdots & \text{pas } 2^{-2^{k-1}-n+k+4} & & & & & \\ \underbrace{0}_{s(1)} & \underbrace{0 \dots 0}_{c(k)} & 10 & \underbrace{1 \dots 1}_{f(n-k-1)} & = & 2^{-2^{k-1}+3} \times \overline{1.1 \dots 1}_2 \\ & & & & & = 2^{-2^{k-1}+3} \times (2^{n-k}-1) \times 2^{-n+k+1} \\ & & & & & = 2^{-2^{k-1}-n+k+4} \times (2^{n-k}-1) \\ & & \text{pas } 2^{-2^{k-1}-n+k+4} & & & & \\ \underbrace{0}_{s(1)} & \underbrace{0 \dots 0}_{c(k)} & 11 & \underbrace{0 \dots 0}_{f(n-k-1)} & = & 2^{-2^{k-1}+3} \times \overline{10.0 \dots 0}_2 = 2^{-2^{k-1}+4} \times \overline{1.0 \dots 0}_2 \\ & & & & & = 2^{-2^{k-1}+4} \end{array}$$

Formatul intern în virgulă mobilă

⋮

$$\underbrace{0}_{s(1)} \underbrace{1 \dots 1}_{c(k)} \underbrace{0 \dots 0}_{f(n-k-1)} = 2^{2^{k-1}-2} \times \overline{1.0 \dots 0}_2 = 2^{2^{k-1}-2}$$

⋮ pas $2^{2^{k-1}-n+k-1}$

$$\underbrace{0}_{s(1)} \underbrace{1 \dots 1}_{c(k)} \underbrace{0 \dots 0}_{f(n-k-1)} = 2^{2^{k-1}-2} \times \overline{1.1 \dots 1}_2 = 2^{2^{k-1}-2} \times (2^{n-k} - 1) \times 2^{-n+k+1}$$

$$\text{pas } 2^{2^{k-1}-n+k-1}$$

$$\underbrace{0}_{s(1)} \underbrace{1 \dots 1}_{c(k)} \underbrace{0 \dots 0}_{f(n-k-1)} = 2^{2^{k-1}-2} \times \overline{10.0 \dots 0}_2 = 2^{2^{k-1}-1} \times \overline{1.0 \dots 0}_2 = 2^{2^{k-1}-1}$$

⋮ pas $2^{2^{k-1}-n+k}$

$$\underbrace{0}_{s(1)} \underbrace{1 \dots 1}_{c(k)} \underbrace{0 \dots 0}_{f(n-k-1)} = 2^{2^{k-1}-1} \times \overline{1.1 \dots 1}_2 = 2^{2^{k-1}-1} \times (2^{n-k} - 1) \times 2^{-n+k+1}$$

$= 2^{2^{k-1}-n+k} \times (2^{n-k} - 1)$
(cel mai mare număr nenul reprezentabil)

Formatul intern în virgulă mobilă

Observație:

Proiectanții standardului IEEE 754 au dorit o reprezentare a virgulei mobile care să poată fi prelucrată ușor de comparațiile pentru întregi, în special pentru sortare.

Plasarea semnului în bitul cel mai semnificativ permite efectuarea rapidă a testelor ' < 0 ', ' > 0 ', ' $= 0$ '.

Plasarea caracteristicii înaintea fracției și faptul că ea este un număr natural (indiferent dacă exponentul este pozitiv sau negativ) ce crește odată cu exponentul permite sortarea numerelor în virgulă mobilă cu ajutorul circuitelor de comparare pentru întregi, deoarece numerele cu exponenți mai mari par mai mari decât numerele cu exponenți mai mici.

Așadar, reprezentarea IEEE 754 poate fi prelucrată cu ajutorul circuitelor de comparare pentru întregi, accelerând sortarea numerelor în virgulă mobilă. De altfel, pe ilustrarea grafică de mai înainte se poate observa că sirurile de biți cresc în ordine lexicografică odată cu creșterea numerelor reale reprezentate, ceea ce corespunde creșterii interpretărilor acestor siruri ca întregi fără semn.

Formatul intern în virgulă mobilă

Pe aceeași ilustrare grafică se mai poate constata că succesorul lexicografic al șirului de biți prin care s-a reprezentat cel mai mare număr nenul reprezentabil:

$$\underbrace{0}_{s(1)} \underbrace{1 \dots 1}_{c(k)} \underbrace{0}_{f(n-k-1)}$$

este șirul de biți prin care se reprezintă $+\infty$:

$$\underbrace{0}_{s(1)} \underbrace{1 \dots 1}_{c(k)} \underbrace{0 \dots 0}_{f(n-k-1)}$$

ceea ce exprimă ideea că $+\infty >$ orice număr.

Formatele single și double

Proiectarea unui sistem de reprezentare în virgulă mobilă presupune o bună alegere a valorilor n (dimensiunea formatului), k (dimensiunea câmpului caracteristică) și $p - 1$ (dimensiunea câmpului fracție). Reamintim că p este numărul de cifre semnificative considerate (prima este implicită, restul sunt stocate în câmpul fracție). Avem $n = 1 + k + (p - 1) = k + p$.

În general n este un multiplu al dimensiunii cuvântului de memorie (**word**) (ex: 32 biți, 64 biți) și atunci pentru k și p trebuie găsit un compromis, deoarece un bit adăugat la unul dintre câmpuri trebuie luat de la celălalt.

Creșterea lui p determină creșterea preciziei numerelor, în timp ce creșterea lui k determină lărgirea domeniului de numere care pot fi reprezentate.

Principiile de proiectare ale seturilor de instrucțiuni hardware ne învață că o proiectare bună are nevoie de compromisuri bune.

Două formate în virgulă mobilă descrise în standardul IEEE 754, care se găsește practic în orice calculator conceput după anul 1980, sunt **single** și **double** (lor le corespund în limbajul C tipurile float și respectiv double).

Formatele single și double

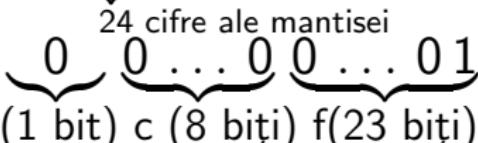
Formatul single (precizie simplă): $n = 32$, $k = 8$, $p = 24$

Rezultă: $E_{\min} = -126$, $E_{\max} = 127$, $bias = 127$

Intervalul de numere ≥ 0 reprezentabile este: $[0, 2^{104} \times (2^{24} - 1)]$

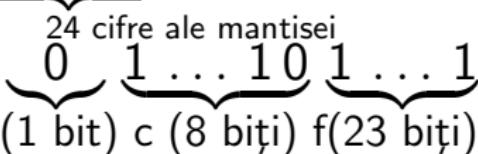
Cel mai mic număr nenul reprezentabil este:

$$2^{-126} \times \underbrace{0.0\dots01}_\text{24 cifre ale mantisei}_2 = 2^{-149},$$

reprezentat: 
s (1 bit) c (8 biți) f(23 biți)

Cel mai mare număr nenul reprezentabil este:

$$2^{127} \times \underbrace{1.1\dots1}_\text{24 cifre ale mantisei}_2 = 2^{104} \times (2^{24} - 1),$$

reprezentat: 
s (1 bit) c (8 biți) f(23 biți)

Formatele single și double

Numerele din intervalul $(0, 2^{-149})$ sunt prea mici pentru a fi reprezentate.

Numerele din intervalul $[2^{-149}, 2^{-125}]$ sunt reprezentate cu pasul 2^{-149} .

Pentru orice $i \in \overline{-125, 126}$, numerele din intervalul $[2^i, 2^{i+1}]$ sunt reprezentate cu pasul 2^{i-23} .

Numerele din intervalul $[2^{127}, 2^{104} \times (2^{24} - 1)]$ sunt reprezentate cu pasul 2^{104} .

Formatele single și double

Ilustrare grafică:

$$\underbrace{0}_{s(1)} \underbrace{0 \dots 0}_{c(8)} \underbrace{0 \dots 0}_{f(23)} = 0$$

⋮ (numere nereprezentabile)

$$\underbrace{0}_{s(1)} \underbrace{0 \dots 0}_{c(8)} \underbrace{0 \dots 01}_{f(23)} = 2^{-126} \times \overline{0.0 \dots 01}_2 = 2^{-126} \times 2^{-23} = 2^{-149}$$

(cel mai mic număr nenul reprezentabil)

⋮ pas 2^{-149}

$$\underbrace{0}_{s(1)} \underbrace{0 \dots 0}_{c(8)} \underbrace{1 \dots 1}_{f(23)} = 2^{-126} \times \overline{0.1 \dots 1}_2 = 2^{-126} \times (2^{23} - 1) \times 2^{-23} \\ = 2^{-149} \times (2^{23} - 1)$$

pas 2^{-149}

$$\underbrace{0}_{s(1)} \underbrace{0 \dots 0}_{c(8)} \underbrace{1}_{f(23)} \underbrace{0 \dots 0} = 2^{-126} \times \overline{1.0 \dots 0}_2 = 2^{-126}$$

⋮ pas 2^{-149}

Formatele single și double

$$\underbrace{0}_{s(1)} \underbrace{0 \dots 0 1}_{c(8)} \underbrace{1 \dots 1}_{f(23)} = 2^{-126} \times \overline{1.1 \dots 1}_2 = 2^{-126} \times (2^{24} - 1) \times 2^{-23}$$
$$= 2^{-149} \times (2^{24} - 1)$$

pas 2^{-149}

$$\underbrace{0}_{s(1)} \underbrace{0 \dots 0 1 0}_{c(8)} \underbrace{0 \dots 0}_{f(23)} = 2^{-126} \times \overline{10.0 \dots 0}_2 = 2^{-125} \times \overline{1.0 \dots 0}_2 = 2^{-125}$$

: pas 2^{-148}

$$\underbrace{0}_{s(1)} \underbrace{0 \dots 0 1 0}_{c(8)} \underbrace{1 \dots 1}_{f(23)} = 2^{-125} \times \overline{1.1 \dots 1}_2 = 2^{-125} \times (2^{24} - 1) \times 2^{-23}$$
$$= 2^{-148} \times (2^{24} - 1)$$

pas 2^{-148}

$$\underbrace{0}_{s(1)} \underbrace{0 \dots 0 1 1}_{c(8)} \underbrace{0 \dots 0}_{f(23)} = 2^{-125} \times \overline{10.0 \dots 0}_2 = 2^{-124} \times \overline{1.0 \dots 0}_2 = 2^{-124}$$

Formatele single și double

⋮

$$\underbrace{0}_{s(1)} \underbrace{1 \dots 1 0 1}_{c(8)} \underbrace{0 \dots 0}_{f(23)} = 2^{126} \times \overline{1.0 \dots 0}_2 = 2^{126}$$

⋮ pas 2^{103}

$$\underbrace{0}_{s(1)} \underbrace{1 \dots 1 0 1}_{c(8)} \underbrace{1 \dots 1}_{f(23)} = 2^{126} \times \overline{1.1 \dots 1}_2 = 2^{126} \times (2^{24} - 1) \times 2^{-23}$$
$$= 2^{103} \times (2^{24} - 1)$$

pas 2^{103}

$$\underbrace{0}_{s(1)} \underbrace{1 \dots 1 0}_{c(8)} \underbrace{0 \dots 0}_{f(23)} = 2^{126} \times \overline{10.0 \dots 0}_2 = 2^{127} \times \overline{1.0 \dots 0}_2 = 2^{127}$$

⋮ pas 2^{104}

$$\underbrace{0}_{s(1)} \underbrace{1 \dots 1 0}_{c(8)} \underbrace{1 \dots 1}_{f(23)} = 2^{127} \times \overline{1.1 \dots 1}_2 = 2^{127} \times (2^{24} - 1) \times 2^{-23}$$
$$= 2^{104} \times (2^{24} - 1)$$

(cel mai mare număr nenul reprezentabil)

Formatele single și double

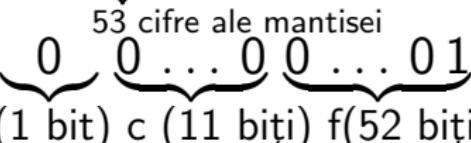
Formatul double (precizie dublă): $n = 64, k = 11, p = 53$

Rezultă: $E_{\min} = -1022, E_{\max} = 1023, bias = 1023$

Intervalul de numere ≥ 0 reprezentabile este: $[0, 2^{971} \times (2^{53} - 1)]$

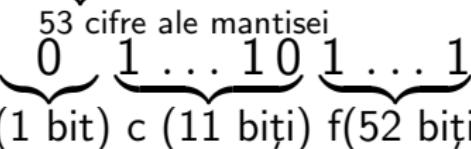
Cel mai mic număr nenul reprezentabil este

$$2^{-1022} \times \underbrace{0.0\dots01}_2 = 2^{-1074},$$

reprezentat: 
s (1 bit) c (11 biți) f(52 biți)

Cel mai mare număr nenul reprezentabil este

$$2^{1023} \times \underbrace{1.1\dots1}_2 = 2^{971} \times (2^{53} - 1),$$

reprezentat: 
s (1 bit) c (11 biți) f(52 biți)

Formatele single și double

Numerele din intervalul $(0, 2^{-1074})$ sunt prea mici pentru a fi reprezentate.

Numerele din intervalul $[2^{-1074}, 2^{-1021}]$ sunt reprezentate cu pasul 2^{-1074} .

Pentru orice $i \in \overline{-1021, 1022}$, numerele din intervalul $[2^i, 2^{i+1}]$ sunt reprezentate cu pasul 2^{i-52} .

Numerele din intervalul $[2^{1023}, 2^{971} \times (2^{53} - 1)]$ sunt reprezentate cu pasul 2^{971} .

Formatele single și double

Ilustrare grafică:

$$\underbrace{0}_{s(1)} \underbrace{0 \dots 0}_{c(11)} \underbrace{0 \dots 0}_{f(52)} = 0$$

⋮ (numere nereprezentabile)

$$\underbrace{0}_{s(1)} \underbrace{0 \dots 0}_{c(11)} \underbrace{0 \dots 01}_{f(52)} = 2^{-1022} \times \overline{0.0 \dots 01}_2 = 2^{-1022} \times 2^{-52} = 2^{-1074}$$

(cel mai mic număr nenul reprezentabil)

⋮ pas 2^{-1074}

$$\underbrace{0}_{s(1)} \underbrace{0 \dots 0}_{c(11)} \underbrace{1 \dots 1}_{f(52)} = 2^{-1022} \times \overline{0.1 \dots 1}_2 = 2^{-1022} \times (2^{52} - 1) \times 2^{-52} \\ = 2^{-1074} \times (2^{52} - 1)$$

pas 2^{-1074}

$$\underbrace{0}_{s(1)} \underbrace{0 \dots 0}_{c(11)} \underbrace{1 \dots 0}_{f(52)} = 2^{-1022} \times \overline{1.0 \dots 0}_2 = 2^{-1022}$$

⋮ pas 2^{-1074}

Formatele single și double

$$\underbrace{0}_{s(1)} \underbrace{0 \dots 0 1}_{c(11)} \underbrace{1 \dots 1}_{f(52)} = 2^{-1022} \times \overline{1.1 \dots 1}_2 = 2^{-1022} \times (2^{53} - 1) \times 2^{-52}$$
$$= 2^{-1074} \times (2^{53} - 1)$$

pas 2^{-1074}

$$\underbrace{0}_{s(1)} \underbrace{0 \dots 0 1 0}_{c(11)} \underbrace{0 \dots 0}_{f(52)} = 2^{-1022} \times \overline{10.0 \dots 0}_2 = 2^{-1021} \times \overline{1.0 \dots 0}_2 = 2^{-1021}$$

: pas 2^{-1073}

$$\underbrace{0}_{s(1)} \underbrace{0 \dots 0 1 0}_{c(11)} \underbrace{1 \dots 1}_{f(52)} = 2^{-1021} \times \overline{1.1 \dots 1}_2 = 2^{-1021} \times (2^{53} - 1) \times 2^{-52}$$
$$= 2^{-1073} \times (2^{53} - 1)$$

pas 2^{-1073}

$$\underbrace{0}_{s(1)} \underbrace{0 \dots 0 1 1}_{c(11)} \underbrace{0 \dots 0}_{f(52)} = 2^{-1021} \times \overline{10.0 \dots 0}_2 = 2^{-1020} \times \overline{1.0 \dots 0}_2 = 2^{-1020}$$

Formatele single și double

⋮

$$\begin{array}{ccccccc} \underbrace{0}_{s(1)} & \underbrace{1 \dots 101}_{c(11)} & \underbrace{0 \dots 0}_{f(52)} & = & 2^{1022} \times \overline{1.0 \dots 0}_2 & = & 2^{1022} \\ \text{pas } 2^{970} \\ \hline \end{array}$$

$$\begin{array}{ccccccc} \underbrace{0}_{s(1)} & \underbrace{1 \dots 101}_{c(11)} & \underbrace{1 \dots 1}_{f(52)} & = & 2^{1022} \times \overline{1.1 \dots 1}_2 & = & 2^{1022} \times (2^{53} - 1) \times 2^{-52} \\ \text{pas } 2^{970} \\ \hline \end{array}$$

$$\begin{array}{ccccccc} \underbrace{0}_{s(1)} & \underbrace{1 \dots 10}_{c(11)} & \underbrace{0 \dots 0}_{f(52)} & = & 2^{1022} \times \overline{10.0 \dots 0}_2 & = & 2^{1023} \times \overline{1.0 \dots 0}_2 & = & 2^{1023} \\ \text{pas } 2^{970} \\ \hline \end{array}$$

$$\begin{array}{ccccccc} \underbrace{0}_{s(1)} & \underbrace{1 \dots 10}_{c(11)} & \underbrace{1 \dots 1}_{f(52)} & = & 2^{1023} \times \overline{1.1 \dots 1}_2 & = & 2^{1023} \times (2^{53} - 1) \times 2^{-52} \\ \text{pas } 2^{971} \\ \hline \end{array}$$

(cel mai mare număr nenul reprezentabil)

Formatul double are ca avantaj față de formatul single atât domeniul mai mare al exponentului cât, mai ales, precizia mai mare, datorată mantisei mai mari.

Formatele single și double

Exemplu: Să se reprezinte în format single și double numărul real
 $x = -4.75$.

Reprezentăm x în baza 2:

$$4 : 2 = 2 \text{ rest } 0$$

$$2 : 2 = 1 \text{ rest } 0$$

$$1 : 2 = 0 \text{ rest } 1$$

$$\text{deci } (4)_2 = \overline{100}$$

$$0.75 \times 2 = 1.5$$

$$0.5 \times 2 = 1.0$$

$$\text{deci } (0.75)_2 = \overline{0.11}$$

$$\text{Rezultă } (x)_2 = -\overline{100.11}$$

În notația științifică normalizată (în baza 2) avem: $x = -\overline{1.0011} \times 2^2$

Deci:

$$\text{semnul} = -$$

$$\text{exponentul} = 2$$

$$\text{mantisa} = \overline{1.0011}$$

În formatul single, deoarece $-126 \leq 2 \leq 127$, vom avea:

$$s = 1$$

$$c = (2 + 127)_2 = (129)_2 = 10000001$$

$$f = 00110000000000000000000000000000$$

Formatele single și double

Deci reprezentarea binară a lui x ca single este:

11000000100110000000000000000000000000000

Determinăm și reprezentarea hexa a lui x ca single:

1100 0000 1001 1000 0000 0000 0000 0000
C 0 9 8 0 0 0 0

deci reprezentarea hexa este:

C0980000

Formatele single și double

În formatul double, deoarece $-1022 \leq 2 \leq 1023$, vom avea:

$$s = 1$$

$$c = (2 + 1023)_2 = (1025)_2 = 10000000001$$

Deci reprezentarea binară a lui x ca double este:

1100000000010011000000000000000
0000000000000000000000000000000

Determinăm și reprezentarea hexa a lui x ca double:

1100 0000 0001 0011 0000 0000 0000 0000
C 0 1 3 0 0 0 0
0000 0000 0000 0000 0000 0000 0000 0000
0 1 1 1 0 1 1 1

deci reprezentarea hexa este:

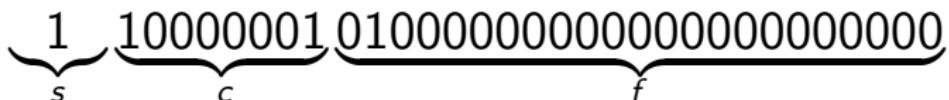
C013000000000000

Formatele single și double

Exemplu: Să se determine valoarea x reprezentată de următorul format single:

11000000101000000000000000000000000000000

Identificăm valorile câmpurilor componente:

The binary number 11000000101000000000000000000000000000000 is shown with three groups of bits underlined: 's' under the first bit (1), 'c' under the next 8 bits (10000001), and 'f' under the remaining 23 bits (0100000000000000000000000).

Deci:

$$s = 1$$

$$c = 10000001 = (2^7 + 1)_2 = (129)_2$$

$$f = 0100000000000000000000000$$

Întrucât $0 < c < 2^8 - 1 = 255$, rezultă (pentru single $bias = 127$):

$$\begin{aligned}x &= (-1)^1 \times 2^{129-127} \times \overline{1.0100000000000000000000000}_2 = \\&= -2^2 \times (2^0 + 2^{-2}) = -4 \times (1 + 0.25) = -4 \times 1.25 = -5\end{aligned}$$

Formatele single și double

Observație:

Reprezentările single și double sunt date multiocet și se supun acelorași convenții de ordonare a octetilor în memorie ca și celelalte date.

Reamintim: convenția de ordonare este fixată hardware pentru fiecare arhitectură și poate fi **little endian** (datele sunt reprezentate cu partea low la adresa mai mică) sau **big endian** (datele sunt reprezentate cu partea low la adresa mai mare); arhitectura Intel folosește convenția little endian.

Când ilustrăm grafic conținutul memoriei obișnuim să desenăm octetii cu adresele crescând spre dreapta și atunci, într-o arhitectură care folosește convenția little endian, locațiile ce conțin cele două reprezentări din primul exemplu de mai devreme vor fi desenate astfel (un octet = două cifre hexa):

Locația ce conține reprezentarea single C0980000 va fi desenată 00 00 98 C0

Locația ce conține reprezentarea double C0130000000000000 va fi desenată 00 00 00 00 00 00 13 C0

În cazul regiștrilor se obișnuiește să se deseneze biții și octetii cu semnificația crescând spre stânga (așa cum se scriu numerele în matematică). Astfel, un registru de 32 biți ce conține reprezentarea single C0980000 va fi desenat tot C0980000

Reglementări IEEE 754

Standardul IEEE 754 conține reglementări referitoare la:

- (1) Operațiile cu numere în virgulă mobilă.**
- (2) Rotunjire și Controlul preciziei rotunjirii.**
- (3) Infinit-uri, Nan-uri, zero-uri cu semn.**
- (4) Excepții.**
- (5) Trap-uri.**

Prezentăm în continuare câteva reglementări importante:

Reglementări IEEE 754

(1) Operațiile implementate cu numere în virgulă mobilă sunt:

- operații aritmetice: adunare, scădere, înmulțire, împărțire, aflare rest (reminder);
- extragere rădăcină pătrată;
- rotunjirea la un întreag reprezentat în virgulă mobilă;
- conversie între diferite formate de virgulă mobilă;
- conversie între formate de virgulă mobilă și formate de întregi;
- conversie binar \leftrightarrow zecimal;
- comparații;
- în unele implementări se poate considera operație și copierea fără schimbarea formatului.

Cu excepția conversiei binar \leftrightarrow zecimal, care este supusă unor constrângeri mai slabe, toate operațiile sunt efectuate ca și când ar produce mai întâi un rezultat intermediar exact (cu precizie infinită și domeniu nelimitat), iar apoi l-ar rotunji.

Există totodată particularități legate de ± 0 , $\pm \infty$ și Nan-uri și excepții cauzate de operanzi și rezultate exceptate.

Reglementări IEEE 754

- **Operații aritmetice: adunare, scădere, înmulțire, împărțire, aflare rest (reminder):**

Se pot aplica unor operanzi de aceeași dimensiune a formatului, iar în unele implementări și unor operanzi de dimensiuni diferite; formatul destinației trebuie să fie de dimensiune \geq cu cel al operanzilor (indiferent de controlul preciziei rotunjirii); rezultatele se rotunjesc.

Dacă $y \neq 0$, restul (reminder) $r = x \text{REM } y$ este definit (indiferent de modul de rotunjire) prin relația matematică $r = x - y \times n$, unde n este întregul cel mai apropiat de valoarea exactă x/y ; dacă există doi întregi n a.î. $|n - x/y| = \frac{1}{2}$, atunci se alege n par. Astfel, restul este întotdeauna exact. Dacă $r = 0$, semnul său va fi cel al lui x .

Controlul preciziei rotunjirii nu se aplică operației reminder.

- **Extragere rădăcină pătrată:**

Este implementată pentru toate formatele de virgulă mobilă suportate. Dimensiunea formatului destinație trebuie să fie \geq cea a operandului. Rezultatul este definit și are semn pozitiv pentru orice operand ≥ 0 , cu excepția faptului că $\sqrt{-0}$ va fi -0 ; rezultatul se rotunjește.

Reglementări IEEE 754

- **Conversie între diferite formate de virgulă mobilă:**

Sunt implementate conversii între toate formatele de virgulă mobilă suportate. Dacă conversia este spre un format cu precizie mai mică, rezultatul se rotunjește; dacă este spre un format cu precizie mai mare, rezultatul este exact.

- **Conversie între formate de virgulă mobilă și formate de întregi:**

Sunt implementate conversii între toate formatele de virgulă mobilă și toate formatele de întregi suportate.

Conversia către întreg se efectuează prin rotunjire.

Conversia între întregii reprezentați în virgulă mobilă și formatele de întregi va fi exactă, în afara cazurilor când se semnalează o excepție (ex. overflow).

- **Rotunjirea unui număr în virgulă mobilă la o valoare întreagă**
reprezentată în virgulă mobilă, în același format.

Se folosesc regulile uzuale de rotunjire, cu mențiunea că atunci când se aplică 'rotunjirea la cel mai apropiat' (Round to Nearest), dacă operandul nerotunjit se află exact la jumătatea distanței între doi întregi consecutivi, rezultatul rotunjit este par.

Reglementări IEEE 754

- Conversie binar \leftrightarrow zecimal:

Este implementată între cel puțin unul dintre formatele de stringuri zecimale și toate formatele de bază de virgulă mobilă suportate, pentru numere din intervalele următoare (valorile stringurilor zecimale sunt $\pm M \times 10^{\pm N}$):

Format	Zecimal \rightarrow Binar		Binar \rightarrow Zecimal	
	MaxM	MaxN	MaxM	MaxN
Single	$10^9 - 1$	99	$10^9 - 1$	53
Double	$10^{17} - 1$	999	$10^{17} - 1$	340

La intrare sunt adăugate/eliminate zerouri la M pentru a minimiza N ; dacă $M > \text{MaxM}$, anumite implementări pot modifica cifrele semnificative dincolo de a 9-a pentru single, respectiv a 17-a pentru double, de obicei prin înlocuire cu 0. Conversiile vor fi corect rotunjite pentru operanzi în intervalele următoare:

Format	Zecimal \rightarrow Binar		Binar \rightarrow Zecimal	
	MaxM	MaxN	MaxM	MaxN
Single	$10^9 - 1$	13	$10^9 - 1$	13
Double	$10^{17} - 1$	27	$10^{17} - 1$	27

Conversiile sunt monotone (creșterea valorii sursă nu va produce descreșterea valorii destinație), iar când se aplică 'rotunjirea la cel mai apropiat' (Round to Nearest) și se respectă precizia stringurilor zecimale de 9, respectiv 17 cifre, conversia binar \rightarrow zecimal \rightarrow binar este identitatea.

Reglementări IEEE 754

- **Comparații:**

Sunt implementate comparații între toate formatele de virgulă mobilă suportate (chiar și dacă formatele operanzilor diferă).

Ele sunt întotdeauna exacte și nu produc overflow/underflow.

Două valori se pot afla în 4 relații mutual exclusive: *mai mic*, *egal*, *mai mare*, *neordonat*; ultima apare când măcar un operand este NaN; un NaN este *neordonat* cu orice, inclusiv cu sine.

Comparațiile vor ignora semnul lui 0 (deci $+0 = -0$).

Rezultatul unei comparații poate fi furnizat, în funcție de implementare:

- printr-un cod condițional ce identifică una din cele 4 relații;
- ca un răspuns true/false la un predicat ce numește o anumită comparație; în acest caz, dacă operanții sunt neordonați iar predicatul nu testează și neordonarea, se semnalează excepția 'operație invalidă'.

Când sunt implementate predicate, trebuie implementate cel puțin următoarele (notația este convențională, ? însemană *neordonat*): $=$ (*egal*), $? <>$ (*neordonat sau diferit*), $>$ (*mai mare*), $>=$ (*mai mare sau egal*), $<$ (*mai mic*), $<=$ (*mai mic sau egal*); de asemenea, trebuie implementat un mod de negare a predicatelor, de exemplu: *NOT(>)*. Se mai pot implementa: $??$ (*neordonat*), $? <>$ (*diferit*), $? >=$ (*neordonat sau mai mare sau egal*), etc.

Reglementări IEEE 754

De exemplu (cu α am notat un NaN):

$1 < 2 \Rightarrow \text{true}$

$2 < 1 \Rightarrow \text{false}$

$1 < 1 \Rightarrow \text{false}$

$1 < \alpha \Rightarrow \text{false} + \text{excepția 'operație invalidă'}$

$\alpha = \alpha \Rightarrow \text{false}$

$\alpha? <> \alpha \Rightarrow \text{true}$

$\alpha <> \alpha \Rightarrow \text{false} + \text{excepția 'operație invalidă'}$

$\alpha?\alpha \Rightarrow \text{true}$

Observație: Negația lui $x = y$ este $NOT(x = y)$ și este echivalentă cu $x? <> y$; pentru alte predicate, negația nu este echivalentă cu predicatul inversat; de exemplu, negația lui $x < y$ este $NOT(x < y)$ și nu este echivalentă cu $x? \geq y$, deoarece ultima nu semnalează excepția 'operație invalidă' dacă x și y sunt neordonatați. Aplicând negația NOT unui predicat, se inversează răspunsurile sale true/false, dar se semnalează în aceleași cazuri excepția 'operație invalidă'.

Reglementări IEEE 754

(2) Rotunjirea și Controlul preciziei rotunjirii:

Rotunjirea ia un număr considerat ca având precizie infinită și, dacă este necesar, îl modifică pentru a se potrivi formatului destinație, semnalând totodată excepția 'inexact'.

Cu excepția conversiei binar \leftrightarrow zecimal, care este supusă unor constrângeri mai slabe, toate operațiile sunt efectuate ca și când ar produce mai întâi un rezultat intermediar exact (cu precizie infinită și domeniu nelimitat), iar apoi l-ar rotunji într-unul din modurile descrise mai jos.

Modurile de rotunjire afectează toate operațiile aritmetice, cu excepția comparațiilor și restului (reminder). Ele pot afecta semnele sumelor nule și afectează limitele dincolo de care se poate semnala overflow/underflow (a se vedea mai jos).

Reglementări IEEE 754

Moduri de rotunjire:

Modul de rotunjire implicit:

- **Rotunjirea la cel mai apropiat (Round to Nearest)**: Rezultatul furnizat este valoarea reprezentabilă cea mai apropiată de valoarea exactă.
Dacă există două valori reprezentabile între care se află valoarea exactă și ele sunt egal depărtate de aceasta, va fi furnizată aceea dintre ele care are bitul cel mai puțin semnificativ (i.e. ultima zecimală a câmpului fracție) egal cu 0.
Dacă valoarea exactă are modulul $\geq 2^{E_{\max}}(2 - 2^{-P})$, ea se rotunjește la ∞ cu același semn.

Moduri de rotunjire dirijate (Directed Roundings) selectable de către utilizator:

- **Rotunjirea către $+\infty$ (rounding toward $+\infty$)**: Rezultatul furnizat este cea mai apropiată valoarea reprezentabilă (inclusiv $+\infty$) \geq valoarea exactă.
- **Rotunjirea către $-\infty$ (rounding toward $-\infty$)**: Rezultatul furnizat este cea mai apropiată valoarea reprezentabilă (inclusiv $-\infty$) \leq valoarea exactă.
- **Rotunjirea către 0 (rounding toward 0)**: Rezultatul furnizat este cea mai apropiată valoarea reprezentabilă \leq în modul decât valoarea exactă.

Reglementări IEEE 754

Precizia rotunjirii:

În mod normal, rezultatul furnizat este rotunjit conform preciziei locației destinație.

Unele sisteme însă sunt construite a.î. să funizeze rezultate doar în locații de anumite formate, mari (de exemplu doar double). În asemenea sisteme se permite utilizatorului (care poate fi un compilator al unui limbaj de nivel înalt) să poată specifica ca rezultatul să fie rotunjit conform unui format mai mic (de exemplu single), chiar dacă el va fi stocat într-o locație de format mare (double). Operanții trebuie să fie însă tot de formatul mic (de exemplu nu se permit operații care combină operanzi double pentru a produce un rezultat single, cu o singură rotunjire).

Controlul preciziei rotunjirii are ca scop să permită sistemelor care au doar destinații de format mare să emuleze, în absența overflow/underflow, precizia sistemelor cu destinații de format mic.

Reglementări IEEE 754

(3) Infinit-uri, Nan-uri, zero-uri cu semn.

Infiniturile au proprietatea: $-\infty < \text{orice număr finit} < +\infty$.

Aritmetica lui ∞ este mereu exactă și nu semnalează excepții, în afară de 'operație invalidă' pentru ∞ .

Excepțiile legate de ∞ sunt semnalate doar când:

- ∞ este obținut din operanzi finiți prin overflow sau împărțire la 0, cu trapul respectiv dezactivat;
- ∞ este un operand invalid.

Reglementări IEEE 754

Toate operațiile vor suporta sNaN-uri și qNaN-uri.

În toate operațiile prezentate mai devreme, prezența unui operand sNaN semnalează excepția 'operație invalidă'.

Orice operație care implică un sNaN sau 'operație invalidă', dacă nu apare un trap și trebuie furnizat un rezultat în virgulă mobilă, va furniza un rezultat qNaN.

Orice operație care implică unul sau doi operanzi NaN, nici unul nefiind sNaN, nu va semnaliza excepții, iar dacă trebuie furnizat un rezultat în virgulă mobilă, va furniza ca rezultat unul dintre qNaN-urile de intrare (conversiile de format ar putea împiedica însă furnizarea aceluiasi NaN).

În operațiile ce nu furnizează un rezultat în virgulă mobilă (anume comparațiile și conversiile spre formate ce nu au NaN-uri), qNaN-urile au efecte similare cu sNaN-urile.

Reglementări IEEE 754

Nu se impune o anumită semnificație semnului unui NaN.

Altfel, semnul unui produs sau cât este xor-ul semnelor operanzilor.

Semnul unei sume, sau diferențe $x - y$ privite ca $x + (-y)$, diferă de cel mult unul din semnele operanzilor.

Semnul rezultatului operației de rotunjire a unui număr în virgulă mobilă la o valoare întreagă este semnul operandului.

Aceste reguli se aplică chiar și dacă operanzii sau rezultatele sunt zero sau infiniți.

Când suma a doi operanzi de semne contrare sau diferența a doi operanzi de același semn este zero, semnul sumei/diferenței va fi + în toate modurile de rotunjire, în afară de rotunjirea către $-\infty$, când semnul va fi -.

Rezultatul operației $x + x = x - (-x)$ va avea același semn ca x , chiar și atunci când x este zero.

Orice rădăcină pătrată validă va avea semnul +, mai puțin $\sqrt{-0}$, care va fi -0.

Reglementări IEEE 754

(4) Excepții:

Semnalarea unei excepții constă în setarea unui **flag de stare** și/sau executarea unui **trap**. Se recomandă ca fiecărei excepții să îi fie asociat un trap sub controlul utilizatorului. Răspunsul implicit la o excepție trebuie să fie însă continuarea fără trap.

Pentru fiecare tip de excepție trebuie să existe un flag de stare care trebuie setat de fiecare dată când apare excepția respectivă și nu se execută nici un trap. Flag-ul poate fi resetat doar la cererea utilizatorului. Utilizatorul trebuie să poată testa sau modifica flagurile de stare individual și eventual să poată salva/restaura toate flagurile odată.

Sigurele excepții care pot coincide sunt 'inexact' cu 'overflow' și 'inexact' cu 'underflow'.

Reglementări IEEE 754

Există cinci tipuri de excepții:

- **Excepția operație invalidă:**

Este semnalată dacă un operand este invalid pentru operația ce trebuie efectuată.

Când excepția apare fără trap iar destinația are un format de virgulă mobilă, rezultatul va fi un qNaN.

Operații invalide sunt:

- orice operație asupra unui sNaN;
- adunarea/scăderea/scăderea în modul a infiniturilor, de exemplu:
 $(+\infty) + (-\infty)$;
- înmulțirea $0 \times \infty$;
- împărțirea $0/0$ sau ∞/∞ ;
- aflarea restului (reminder) $x \text{ REM } y$, când y este zero sau x este infinit;
- rădăcina pătrată a unui operand $< zero$;
- conversia unui număr în virgulă mobilă binar la un format întreg sau zecimal când un overflow, infinit sau NaN preîntâmpină o reprezentare fidelă în acel format iar aceasta nu poate fi semnalată altfel;
- comparația prin intermediul unor predicate ce implică $<$ sau $>$ fără ?, când operanții sunt *neordonatați*.

Reglementări IEEE 754

- Excepția **împărțire la zero**:

Este semnalată dacă deîmpărțitul (divident) este un număr finit nenul, iar împărțitorul (divizor) este zero.

Dacă nu se execută trap, rezultatul este un ∞ cu semn adecvat (a se vedea operațiile aritmetice, mai sus).

Reglementări IEEE 754

- Excepția **overflow** (depășire superioară):

Este semnalată dacă cel mai mare număr finit reprezentabil conform formatului destinației este excedat în modul de ceea ce ar fi trebuit să fie rezultatul în virgulă mobilă rotunjit, cu domeniul exponentului nemărginit.

Dacă nu se execută trap, rezultatul este determinat de modul de rotunjire și de semnul rezultatului intermediar, după cum urmează:

- rotunjirea la cel mai apropiat (Round to Nearest) duce toate situațiile de overflow către ∞ cu semnul rezultatului intermediar;
- rotunjirea către 0 (rounding toward 0) duce toate situațiile de overflow către cel mai mare număr finit reprezentabil conform formatului, cu semnul rezultatului intermediar;
- rotunjirea către $-\infty$ (rounding toward $-\infty$) duce toate situațiile de overflow pozitiv către cel mai mare număr finit reprezentabil conform formatului, și duce toate situațiile de overflow negativ către $-\infty$;
- rotunjirea către $+\infty$ (rounding toward $+\infty$) duce toate situațiile de overflow negativ către cel mai negativ număr finit reprezentabil conform formatului, și duce toate situațiile de overflow pozitiv către $+\infty$.

Reglementări IEEE 754

Overflow-urile cu trap, în toate operațiile în afara conversiilor, vor furniza handler-ului asociat trap-ului rezultatul obținut prin împărțirea rezultatului exact la 2^α și apoi rotunjire, unde α (bias adjust) este 192 pentru single și 1536 pentru double (α este ales a.î. valorile cu overflow/underflow să fie translatătate cât mai aproape de mijlocul domeniului exponentului și astfel, dacă se dorește, să poată fi folosite în operații scalate ulterioare, cu risc mai mic de a cauza alte excepții).

Overflow-urile cu trap în conversii dinspre un format în virgulă mobilă vor furniza handler-ului asociat trap-ului un rezultat în același format sau într-unul mai larg, eventual cu bias-ul exponentului ajustat, dar rotunjit conform preciziei destinației.

Overflow-urile cu trap în conversii de la zecimal la virgulă mobilă vor furniza handler-ului asociat trap-ului un rezultat în cel mai larg format suportat, eventual cu bias-ul exponentului ajustat, dar rotunjit conform preciziei destinației; dacă rezultatul este situat mult în afara domeniului pentru a putea fi ajustat bias-ul, va fi furnizat în loc un qNaN.

Reglementări IEEE 754

- Excepția **underflow** (depășire inferioară):

La underflow contribuie două evenimente corelate:

- producerea unui rezultat nenul foarte mic, între $\pm 2^{E_{\min}}$, care, fiind atât de mic, poate cauza ulterior alte excepții, ca overflow la împărțire;
- pierdea accentuată a acurateței în timpul aproximării unor numere atât de mici prin numere denormalizate.

Modul de detectare a acestor evenimente este dependent de implementare, dar trebuie să se facă la fel pentru toate operațiile.

Micimea valorii (tininess) poate fi detectată:

- după rotunjire: când un rezultat nenul calculat ca și când domeniul exponentului ar fi nemărginit s-ar situa strict între $\pm 2^{E_{\min}}$;
- înaintea rotunjirii: când un rezultat nenul calculat ca și când atât domeniul exponentului cât și precizia ar fi nemărginire s-ar situa strict între $\pm 2^{E_{\min}}$.

Pierderea acurateței (loss of accuracy) poate fi detectată:

- ca o pierdere la denormalizare: când rezultatul furnizat diferă de ceea ce s-ar fi calculat dacă domeniul exponentului ar fi nemărginit;
- ca un rezultat inexact: când rezultatul furnizat diferă de ceea ce s-ar fi calculat dacă atât domeniul exponentului cât și precizia ar fi nemărginire (aceasta este condiția numită 'inexact' în cazul excepției 'inexact', a se vedea mai jos).

Reglementări IEEE 754

Dacă un trap pentru undeflow nu este implementat sau nu este activat (cazul implicit), excepția underflow va fi semnalată (prin intermediul flag-ului de underflow) doar dacă au fost detectate atât micimea cât și pierderea acurateței. Metoda de detectare nu afectează rezultatul furnizat, care poate fi zero, număr denormalizat, sau $\pm 2^{E_{\min}}$.

Dacă este implementat și activat un trap pentru underflow, excepția underflow va fi semnalată când a fost detectată micimea, indiferent de pierderea acurateței.

Underflow-urile cu trap în cazul tuturor operațiilor mai puțin conversia vor furniza handler-ului asociat trap-ului rezultatul obținut prin înmulțirea rezultatului exact cu 2^α și apoi rotunjire (α este ca mai devreme).

Underflow-urile cu trap în cazul conversiilor vor fi tratate analog ca în cazul overflow-urilor.

Reglementări IEEE 754

Observație: din cele de mai sus rezultă ca factorul cel mai important care determină apariția overflow/underflow este situaarea exponentului sub E_{\min} (underflow), respectiv deasupra lui E_{\max} (overflow).

- Excepția **inexact**:

Este semnalată dacă rezultatul rotunjit al unei operații este inexact sau efectuează overflow fără trap.

Rezultatul rotunjit sau având overflow este furnizat destinației sau, dacă se efectuează trap la inexact, handler-ului asociat trap-ului.

Reglementări IEEE 754

(5) Trap-uri:

Utilizatorul poate solicita un **trap** (întrerupere software) în cazul oricărei dintre cele cinci excepții de mai sus, prin specificarea unei subroutines **handler** asociată lui.

De asemenea, poate solicita ca un asemenea handler să fie dezactivat, salvat, restaurat, și poate testa dacă handler-ul este activat sau nu.

Dacă este semnalată o excepție cu trap-ul dezactivat, ea este tratată aşa cum am prezentat în secțiunea 'excepții'.

Dacă este semnalată o excepție cu trap-ul activat, execuția programului în care a apărut excepția este suspendată iar handler-ul este apelat, furnizându-i-se (de exemplu ca parametru) un rezultat aşa cum am prezentat în secțiunea 'excepții'.

Reglementări IEEE 754

Un **handler de trap** trebuie să aibă capabilitățile unei subrutine ce poate returna o valoare spre a fi folosită în locul rezultatului operației exceptat; acest rezultat este nedefinit, în afara cazului când este furnizat de handler-ul de trap. De asemenea, flag(-urile) corespunzătoare excepțiilor semnalate cu trap activat pot avea valori nedefinite, în afara cazului când ele sunt setate/resetate de handler.

Se recomandă ca un handler de trap să poată determina:

- ce excepție / excepții au apărut în această operație;
- tipul operației efectuate;
- formatul destinației;
- în cazul excepțiilor 'overflow', 'underflow' și 'inexact', rezultatul corect rotunjit, inclusiv informația că s-ar putea să nu se potrivească în formatul destinației;
- în cazul excepțiilor 'operație invalidă' și 'împărțire la zero', valorile operandului.

Dacă sunt activate, trap-urile asociate excepțiilor 'overflow' și 'underflow' au **prioritate (precedență)** asupra unui trap separat asociat excepției 'inexact'.

Implementarea operațiilor în virgulă mobilă

În continuare prezentăm detalii referitoare la modul de implementare a unor operații în virgulă mobilă, mai exact adunarea și înmulțirea, pentru a îndeplini reglementările din standardul IEEE 754.

Adunarea în virgulă mobilă

Algoritmul de adunare în virgulă mobilă:

- (1) Se compară exponenții celor două numere.

Se deplasează spre dreapta cifrele mantisei numărului cu exponent mai mic (deci se deplasează spre stânga virgula sa) până când exponenții celor două numere devin egali.

- (2) Se adună mantisele. Adunarea mantiselor determină și semnul sumei.

- (3) Se normalizează suma, fie deplasând dreapta și incrementând exponentul, fie deplasând stânga și decrementând exponentul.

Se testează dacă în urma normalizării sumei s-a produs overflow/underflow (exponentul sumei este în afara domeniului corespunzător formatului).

Dacă da, se semnalează **excepție**.

Dacă nu, se trece la (4).

- (4) Se rotunjește suma, conform formatului.

- (5) Se testează dacă suma rotunjită este normalizată (de exemplu, dacă la rotunjire se adună un 1 la un sir de biți 1, rezultatul poate să nu fie normalizat).

Dacă da, **stop**.

Dacă nu, se reia pasul (3).

Adunarea în virgulă mobilă

Circuit logic pentru adunarea în virgulă mobilă:

TODO (H&P, Fig. 4.45, p.262)

Adunarea în virgulă mobilă

Exemplu: Fie $x = 0.5$, $y = -0.4375$. Calculați $x + y$ aplicând algoritmul de adunare în virgulă mobilă pentru formatul single.

Nu este nevoie să determinăm formatul intern (ca single) al celor două numere. Putem opera pe reprezentarea lor matematică în baza 2 în notație științifică, dar vom ține cont de parametrii formatului single: $n = 32$, $k = 8$, $p = 24$, $E_{\min} = -126$, $E_{\max} = bias = 127$.

Reprezentăm x , y în baza 2:

Părțile lor întregi sunt $\bar{0}$; pentru părțile lor fracționare, avem:

$$0.5 \times 2 = 1.0$$

$$0.4375 \times 2 = 0.875$$

$$0.875 \times 2 = 1.75$$

$$0.75 \times 2 = 1.5$$

$$0.5 \times 2 = 1.0$$

Deci $(x)_2 = \overline{0.1}$, $(y)_2 = \overline{-0.0111}$.

În notație științifică normalizată, avem: $x = \overline{1.0} \times 2^{-1}$, $y = \overline{-1.11} \times 2^{-2}$.

Deoarece exponenții -1 și -2 sunt în intervalul $\{-126, \dots, 127\}$ iar fracțiile 0 și 11 încap pe 23 biți, numerele x și y se vor reprezenta ca single în formă normalizată, cu s , c , f deduse din notația științifică normalizată de mai sus (nu mai scriem reprezentările respective).

Adunarea în virgulă mobilă

Aplicăm pașii algoritmului de adunare în virgulă mobilă descris mai devreme:

(1) Avem $-2 < -1$, deci se deplasează spre dreapta cifrele mantisei lui y până ce exponentul său îl egalează pe cel al lui x : $y = -\overline{0.111} \times 2^{-1}$.

(2) Se adună mantisele: $1.0 - 0.111 = 0.001$, deci suma $x + y = \overline{0.001} \times 2^{-1}$.

(3) Se normalizează suma, verificând dacă se produce overflow/underflow:

Avem $\overline{0.001} \times 2^{-1} = \overline{1.0} \times 2^{-4}$ și cum $-126 \leq -4 \leq 127$, nu avem overflow/underflow.

(4) Se rotunjește suma, conform formatului single (23 zecimale binare):
Întrucât fracția 0 începe pe 23 biți, rotunjirea nu produce nici o modificare a biților.

(5) Suma rezultată este deja normalizată, deci nu se impune încă o normalizare.

În final, am obținut suma $\overline{1.0} \times 2^{-4} = 1 \times 2^0 \times 2^{-4} = 2^{-4} = \frac{1}{16} = 0.0625$.

Putem verifica, efectuând adunarea (scăderea) în baza 10, că $0.5 - 0.4375 = 0.0625$.

$\hat{\text{I}}$ nmulțirea în virgulă mobilă

Algoritmul de înmulțire în virgulă mobilă:

- (1) Se adună exponenții celor două numere, obținându-se exponentul produsului.

Alternativ, se adună exponenții biasați, scăzând o dată biasul, obținându-se exponentul biasat al produsului.

- (2) Se înmulțesc mantisele.

- (3) Se normalizează produsul, dacă este necesar, prin deplasare dreapta și incrementând exponentul.

Se testează dacă în urma normalizării produsului s-a produs overflow/underflow (exponentul produsului este în afara domeniului corespunzător formatului).

Dacă da, se semnalează **excepție**.

Dacă nu, se trece la (4).

- (4) Se rotunjește produsul, conform formatului.

Se testează dacă produsul rotunjit este normalizat.

Dacă da, se trece la (5).

Dacă nu, se reia pasul (3).

- (5) Se stabilește semnul produsului ca fiind pozitiv, dacă operanții au același semn și negativ, dacă operanții au semn contrar (se face 'xor' între biții de semn ai reprezentările operanților).

Stop.

Înmulțirea în virgulă mobilă

Exemplu: Fie $x = 0.5$, $y = -0.4375$. Calculați $x \times y$ aplicând algoritmul de înmulțire în virgulă mobilă pentru formatul single.

Ca în exemplul precedent, nu este nevoie să determinăm formatul intern (ca single) al celor două numere, ci putem opera pe reprezentarea lor matematică în baza 2 în notație științifică, ținând cont de parametrii formatului single: $n = 32$, $k = 8$, $p = 24$, $E_{\min} = -126$, $E_{\max} = bias = 127$.

Am văzut în exemplul precedent că, în notație științifică normalizată, avem: $x = \overline{1.0} \times 2^{-1}$, $y = \overline{-1.11} \times 2^{-2}$ și că ambele numere se vor reprezenta ca single în formă normalizată, cu s , c , f deduse din această notație (deoarece exponenții -1 și -2 sunt în intervalul $\{-126, \dots, 127\}$ iar fracțiile 0 și 11 încap pe 23 biți).

Înmulțirea în virgulă mobilă

Aplicăm pașii algoritmului de înmulțire în virgulă mobilă descris mai devreme:

(1) Se adună exponenții, obținându-se exponentul produsului:

$$-1 + (-2) = -3.$$

Alternativ, se adună exponenții biasati (caracteristicile), scăzând o dată biasul, obținându-se exponentul biasat (caracteristica) al produsului:

$$(-1 + 127) + (-2 + 127) - 127 = 124.$$

(2) Se înmulțesc mantisele:

$$\begin{array}{r} 1.0\ 0\ x \\ 1.1\ 1 \\ \hline 1\ 0\ 0 \\ 1\ 0\ 0 \\ \hline 1.1\ 1\ 0\ 0 \end{array}$$

Deci produsul fără semn este (deocamdată): $\overline{1.11} \times 2^{-3}$

(3) Se normalizează produsul, verificând dacă se produce overflow/underflow:

Produsul este deja normalizat și, deoarece exponentul său se încadrează în domeniul $-126 \leq -3 \leq 127$ (sau exponentul său biasat se încadrează în domeniul $1 \leq 124 \leq 254$), nu avem overflow/underflow.

Înmulțirea în virgulă mobilă

(4) Se rotunjește produsul, conform formatului single (23 zecimale binare) și se verifică dacă după rotunjire este normalizat:

Întrucât fracția 11 începe pe 23 biți, rotunjirea nu produce modificări, iar produsul este, în continuare, normalizat.

(5) Se stabilește semnul produsului ca fiind negativ, deoarece semnele operanzilor diferă.

În final, se obține produsul

$$-\overline{1.11} \times 2^{-3} = -(1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2}) \times 2^{-3} = -(1 + \frac{1}{2} + \frac{1}{4}) \times \frac{1}{8} = -\frac{7}{4} \times \frac{1}{8} = -\frac{7}{32} = -0.21875.$$

Putem verifica, efectuând înmulțirea în baza 10, că $0.5 \times (-0.4375) = -0.21875$.

Alte operații în virgulă mobilă

Am văzut că algoritmul de adunare poate efectua (în funcție de semnul celui de-al doilea operand) și scăderea.

Înmulțirea se face în general mai lent decât adunarea și scăderea, dar există multe procedee de a adăuga hardware pentru a o face mai rapidă.

Este însă greu de conceput o împărțire în virgulă mobilă rapidă și exactă.

O tehnică se bazează pe iterația lui Newton, unde împărțirea este transformată în aflarea zeroului unei funcții, pentru a determina valoarea inversă $1/x$, care apoi este înmulțită cu celălalt operand.

Tehnicile iterative însă nu pot fi rotunjite corect dacă nu se calculează mai mulți biți suplimentari.

Pe de altă parte, tehnica împărțirii SRT (Sweeney, Robertson, Tocher) încearcă să estimeze în fiecare pas mai mulți biți ai câtului, utilizând un tabel de căutare pe baza celor mai semnificativi biți ai deîmpărțitului și restului; pașilor următori le revine sarcina corectării estimărilor greșite.

Aritmetică precisă

Așa cum am mai spus, standardul IEEE 754 cere ca toate operațiile, cu excepția conversiei binar \leftrightarrow zecimal, să fie efectuate ca și când ar produce mai întâi un rezultat intermediar exact (cu precizie infinită și domeniu nelimitat), iar apoi l-ar rotunji.

De aceea, pentru o rotunjire corectă, hardware-ul trebuie să includă în calcul biți suplimentari. În algoritmii de adunare și înmulțire descriși mai devreme nu am precizat numărul de biți pe care poate să-l ocupe o reprezentare intermediară, dar în mod evident, dacă fiecare rezultat intermediar ar fi trunchiat la numărul fixat de biți pe care trebuie să-l ocupe reprezentarea finală, rotunjirea nu ar mai avea sens, iar rezultatul final ar fi alterat prea tare.

Din acest motiv, hardware-ul folosește în timpul calculelor intermediare trei biți suplimentari, adăugați în partea cea mai puțin semnificativă a fracției:

- **bitul de gardă**, îl vom nota b_{-1} ;
- **bitul de rotunjire**, îl vom nota b_{-2} ;
- **bitul de colectare (sticky bit)**, îl vom nota b_{-3} .

Reprezentările rezultatelor intermediare vor avea deci biții $b_n \dots b_0 b_{-1} b_{-2} b_{-3}$. Când se deplasează la dreapta mantisele, biții de gardă și rotunjire vor păstra primele zecimale care au ieșit din câmpul fracție al formatului, iar bitul de colectare va fi setat cu 1 d.d. există zecimale nenule la dreapta bitului de rotunjire.

Aritmetică precisă

Exemplu: Fie $x = \overline{1.1001} \times 2^{-1} = 1.5625$, $y = \overline{1.0001} \times 2^3 = 8.5$. Calculați $x + y$ aplicând algoritmul de adunare în virgulă mobilă pentru formatul de virgulă mobilă descris de parametrii: $n = 8$, $k = 3$, $p = 5$ (deci fracția are 4 zecimale binare), $E_{\min} = -2^2 + 2 = -2$, $E_{\max} = bias = 2^2 - 1 = 3$.

Se va efectua calculul atât în cazul când sunt prezenți cei trei biți suplimentari (deci reprezentările intermediare au fracția de 7 biți) cât și în cazul când aceștia nu sunt prezenți (deci reprezentările intermediare au fracția de 4 biți).

Deoarece exponenții -1 și 3 sunt în intervalul $\{-2, \dots, 3\}$ iar fracțiile 1001 și 0001 încap pe 4 biți, numerele x și y se vor reprezenta în formă normalizată, cu s , c , f deduse din notația științifică normalizată dată în enunț.

Avem $-1 < 3$, deci se deplasează spre dreapta mantisa lui x .

În cazul când folosim biții suplimentari, mantisa lui x va deveni 0.0001101 (zecimalele pierdute în dreapta bitului de rotunjire sunt 01 , de aceea bitul de colectare a devenit 1), deci vom avea de adunat $0.0001101 + 1.0001000 = 1.0010101$, deci rezultatul intermediar este 1.0010101×2^3 .

El este normalizat și, deoarece $-2 \leq 3 \leq 3$, nu se produce overflow/underflow.

Aplicând rotunjirea 'round to nearest' conform formatului dat (4 zecimale), obținem rezultatul intermediar 1.0011×2^3 . Aceasta este normalizat.

Deci, rezultatul final este: $x + y = 1.0011 \times 2^3 = 9.5$

Aritmetică precisă

În cazul când nu folosim biții suplimentari, prin deplasarea la dreapta mantisa lui x va deveni 0.0001, deci vom avea de adunat $0.0001 + 1.0001 = 1.0010$, deci rezultatul intermediar este 1.0010×2^3 .

El este normalizat și, deoarece $-2 \leq 3 \leq 3$, nu se produce overflow/underflow. De asemenea, nu avem ce rotunji.

Astfel, rezultatul final este: $x + y = 1.0010 \times 2^3 = 9$.

Rezultatul exact, obținut efectuând adunarea în baza 10, este:

$$x + y = 1.5625 + 8.5 = 10.0625.$$

Aritmetică precisă

Observații:

- Adunarea are nevoie doar de un bit suplimentar (bitul de gardă), dar înmulțirea poate avea nevoie de doi (bitul de gardă și cel de rotunjire). Un produs binar poate avea în față un bit 0, prin urmare pasul de normalizare poate deplasa produsul spre stânga cu un bit. Aceasta deplasează bitul de gardă în poziția celui mai puțin semnificativ bit al produsului (ultimul bit al fracției), lăsând bitul de rotunjire să ajute corecta rotunjire a produsului.
- Bitul de colectare permite calculatorului să facă diferență, atunci când face rotunjirea, între $0.10\dots00$ (aflat la jumătatea distanței între 0 și 1 și care astfel, cu regula 'round to nearest', s-ar rotunji la întregul par 0) și $0.10\dots01$ (care, cu regula 'round to nearest' s-ar rotunji la întregul 1). Bitul de colectare poate fi setat, de exemplu, în timpul adunării, când numărul cu exponent mai mic este delasat spre dreapta (a se vedea exemplul anterior).

Aritmetică precisă

Precizia virgulei mobile este măsurată în mod obișnuit prin numărul de biți eronati, din ultimii biți semnificativi ai mantisei (biții cei mai puțin semnificativi ai fracției); măsura s.n. numărul de **unități din ultimele poziții** sau **ulp (units in the last place)**. Dacă un număr este mai mic cu 2 în biții cei mai puțin semnificativi, se va spune că este mai puțin cu 2 ulpi.

Dacă nu se semnalează excepții (operație invalidă, overflow, underflow), standardul IEEE 754 garantează că numărul folosit de calculator este cu o precizie de până la o jumătate de ulp.

Inadvertențe și capcane

Diferența dintre precizia limitată a aritmeticii calculatoarelor și precizia nelimitată a aritmeticii din matematică conduce la mai multe inadvertențe și capcane aritmetice, la care trebuie să fim atenți atunci când scriem programe.

Una din inadvertențe este că adunarea numerelor reale din matematică este asociativă, în timp ce adunarea în virgulă mobilă din calculator nu este.

De exemplu, considerăm următoarea secvență de cod în limbajul C:

```
float x,y,z,r1,r2; /* float se implementeaza prin single */
x = -1.5e38; y = 1.5e38; z = 1.0;
r1 = x + (y + z);
r2 = (x + y) + z;
printf("%f    %f", r1, r2); /* afisaza: 0.000000    1.000000 */
```

Într-adevăr, la adunarea $y + z$, adică $1.5 \times 10^{38} + 1.0$, diferența între operanzi este atât de mare încât la deplasarea spre dreapta a operandului mai mic 1, pentru aducere la același exponent cu operandul mai mare 1.5×10^{38} , toți biții nenuli ai mantisei au ieșit din câmpul fracție al reprezentării single, a.î. s-a adunat de fapt $1.5 \times 10^{38} + 0$.

Numerele pozitive < cel mai mic număr pozitiv denormalizat reprezentabil sunt prea mici pentru a putea fi reprezentate iar când se obțin dintr-un calcul și nu apare underflow se asimilează cu 0 (asemenea numere care în matematică sunt nenule dar în calculator se asimilează cu 0 s.n. **zerouri ale mașinii**).

Inadvertențe și capcane

Exemplu: Ilustrăm alte câteva fenomene legate de apariția, propagarea, amplificarea erorilor la efectuarea operațiilor în virgulă mobilă, considerând următoarea secvență de cod în limbajul C:

```
float x,y,z;
x = 1 / -3e38; printf("%f\n", x);
/* afisaza: -0.000000 */
x = 3e38; y = 3e38; z = x + y; printf("%f\n", z);
/* afisaza: inf (+infinit) */
x = (1 / -3e38) * (3e38 + 3e38); printf("%f\n", x);
/* afisaza: -2.000000 */
/* adica: (-0) * (+infinity) == -2; semnul vine de la -0 */
x = 3e38; y = 1e-30; z = x + y; printf("%g\n", z);
/* afisaza: 3e+38 (== x) */
x = 1e-38 * (1 / 1e-38) - 1; printf("%g\n", x);
/* afisaza: -1.11022e-16 */
/* deci !=0 dar foarte mic */
```

Atenție deci la erorile de calcul și zerourile mașinii.

Inadvertențe și capcane

Din cauza erorilor de calcul care apar prin pierderea ultimelor zecimale, se recomandă ca în programele cu multe operații în virgulă mobilă să se înlocuiască testele de forma $x == 0$ cu teste de forma $\text{fabs}(x) < \text{eps}$, unde eps este un număr real strict pozitiv foarte mic, convenabil ales, de exemplu 0.00001.

De exemplu, dacă programăm metoda lui Gauss de rezolvare a unui sistem de ecuații liniare (algoritmul presupune multiple înmulțiri și împărțiri la elemente nenule din matricea sistemului alese ca pivot, în urma căror matricea capătă 0 sub diagonală), elemente care matematic ar trebui să fie 0 în calculator sunt numere nenule cu modul foarte mic - deci dacă am căuta pivotii nenuli după condiția $a[i][j] != 0$, am risca să alegem drept pivot un număr care matematic (dacă am face calculele exact) ar trebui să fie 0.

În general, efectele negative ale erorilor de calcul în virgulă mobilă sunt foarte mari dacă pe baza rezultatelor calculelor aritmetice se iau decizii logice.

Inadvertențe și capcane

Concluzie: operațiile aritmetice din calculator au alte proprietăți decât cele din matematică, astfel încât algoritmii din matematică trebuie modificați pentru a fi corecți și cu operațiile din calculator.

Cuprins

1 Performația calculatoarelor

Conceptul de performanță
Măsurarea performanței

2 Aritmetică sistemelor de calcul

Reprezentarea numerelor în matematică
Reprezentarea numerelor în calculator
Reprezentarea numerelor naturale ca întregi fără semn
Reprezentarea numerelor întregi în complement față de 2
Reprezentarea numerelor reale în virgulă mobilă

3 Logica pentru calculatoare

Algebre booleene
Funcții booleene
Particularizare la cazul B_2

4 Circuite logice

Circuite logice, Sisteme digitale
0-DS (Circuite combinaționale, Funcții booleene)
1-DS (Memorii)
2-DS (Automate finite)
Algoritmi de înmulțire și împărțire hardware
3-DS (Procesoare) și 4-DS (Calculatoare)
 n -DS, $n > 4$

Cuprins

1 Performanța calculatoarelor

Conceptul de performanță
Măsurarea performanței

2 Aritmetică sistemelor de calcul

Reprezentarea numerelor în matematică
Reprezentarea numerelor în calculator
Reprezentarea numerelor naturale ca întregi fără semn
Reprezentarea numerelor întregi în complement față de 2
Reprezentarea numerelor reale în virgulă mobilă

3 Logica pentru calculatoare

Algebre booleene

Funcții booleene
Particularizare la cazul B_2

4 Circuite logice

Circuite logice, Sisteme digitale
0-DS (Circuite combinaționale, Funcții booleene)
1-DS (Memorii)
2-DS (Automate finite)
Algoritmi de înmulțire și împărțire hardware
3-DS (Procesoare) și 4-DS (Calculatoare)
 n -DS, $n > 4$

Sistemele de calcul în general se bazează pe logica clasică (Aristotel).

Corespondentul algebric al acestei logici este constituit din algebrelle și funcțiile booleene. O deosebită importanță o vor avea algebra cu două elemente B_2 și funcțiile booleene corespunzătoare acestui caz.

Noțiunea de algebră booleană are mai multe definiții echivalente. O vom introduce progresiv.

Inversa unei relații binare

Def: Fie R o relație binară pe o mulțime A . **Relația inversă** lui R este relația binară R^{-1} pe A definită astfel:

$$xR^{-1}y \stackrel{d}{\Leftrightarrow} yRx$$

Th: $R^{-1-1} = R$

Dacă A este finită, putem reprezenta grafic o relație binară R pe A printr-o diagramă în care: elementele lui A sunt reprezentate prin puncte, iar faptul că xRy este reprezentat printr-o săgeată de la punctul x la punctul y .

În acest caz, reprezentarea lui R^{-1} se obține inversând sensul săgeților.

TODO: desen pentru R și R^{-1} .

Mulțimi ordonate

Def: Fie A o mulțime. O relație binară \leq pe A s.n. **relație de ordine (parțială)**, dacă satisfac axiolele:

reflexivitate: $x \leq x$

antisimetrie: dacă $x \leq y$ și $y \leq x$, atunci $x = y$

tranzitivitate: dacă $x \leq y$ și $y \leq z$, atunci $x \leq z$

În acest caz, perechea $\langle A, \leq \rangle$ s.n. **mulțime (parțial) ordonată**.

Inversa relației \leq se notează \geq .

Două elemente $x, y \in A$ sunt **comparabile** dacă $x \leq y$ sau $y \leq x$; altfel, sunt **incomparabile**.

Vom nota cu $x < y$ (sau $y > x$) faptul că $x \leq y$ și $x \neq y$.

Th: \leq este relație de ordine $\Leftrightarrow \geq$ este relație de ordine

(mulțimile ordonate $\langle A, \leq \rangle$ și $\langle A, \geq \rangle$ se zic **duale** un celeilalte).

Mulțimi ordonate

Dacă E este o noțiune sau proprietate formulată pentru o anumită mulțime ordonată, vom nota cu E^{op} noțiunea/proprietatea analoagă formulată pentru duala acesteia (practic, se înlocuiește în E relația de ordine cu inversa ei). E^{op} s.n. **duala** lui E .

Evident, avem $E^{op\,op} = E$, iar E și E^{op} sunt duale una alteia.

Principul dualității pentru mulțimi ordonate afirmează că dacă P este o proprietate formulată pentru o anumită mulțime ordonată și considerăm duala ei P^{op} , atunci:

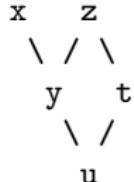
$$P \text{ este adevărată} \Leftrightarrow P^{op} \text{ este adevărată}$$

În general, noțiunile legate de ordine se definesc în perechi de noțiuni duale. Pentru ele sunt formulate proprietăți duale.

Mulțimi ordonate

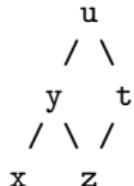
O relație de ordine \leq pe o mulțime finită A se poate reprezenta grafic printr-o **diagramă Hasse**, astfel: elementele mulțimii A sunt reprezentate prin puncte, iar faptul că $x < y$ și nu există $z \in A$ a.î. $x < z < y$ se reprezintă printr-o linie ce unește punctele x și y , y fiind situat mai sus ca x .

Exemplu:



Această diagramă arată că elementele aflate în relație sunt: $u \leq u, y, t, x, z$; $y \leq y, x, z$; $t \leq t, z$; $x \leq x$; $z \leq z$; observăm că nu este nevoie să evidențiem prin săgeți toate perechile de elemente aflate în relație (în general, avem $a \leq b$ d.d. $a = b$ sau există un lanț ascendent de la a la b).

Diagrama Hasse asociată relației inverse \geq este desenul "răsturnat":



Mulțimi ordonate

Def: Relația de ordine R pe mulțimea A este **totală** dacă satisface în plus axioma:

pentru orice $x, y \in A$, avem $x \leq y$ sau $y \leq x$
(i.e. oricare două elemente sunt comparabile).

În acest caz, perechea $\langle A, \leq \rangle$ s.n. **mulțime total ordonată** (sau **liniar ordonată**).

Th: \leq este relație de ordine totală $\Leftrightarrow \geq$ este relație de ordine totală.

Diagrama Hasse a unei relații de ordine totale pe o mulțime finită este un lanț descendenter:

x
|
y
|
z

Mulțimi ordonate

Def: Fie $\langle A, \leq \rangle$ o mulțime ordonată.

- Un element $z \in A$ este **majorant/minorant** pentru o submulțime $M \subseteq A$ dacă $\forall x \in M (x \leq z)$ / $\forall x \in M (z \leq x)$.
- Un element $z \in A$ este **element maximal/element minimal** pentru o submulțime $M \subseteq A$ dacă $z \in M$ și $\forall x \in M (z, x \text{ comparabile} \rightarrow x \leq z)$ / $z \in M$ și $\forall x \in M (z, x \text{ comparabile} \rightarrow z \leq x)$.
- Un element $z \in A$ este **maxim/minim** pentru o submulțime $M \subseteq A$ dacă $z \in M$ și $\forall x \in M (x \leq z)$ / $z \in M$ și $\forall x \in M (z \leq x)$.
- Un element $z \in A$ este **supremum/infimum** pentru o submulțime $M \subseteq A$ dacă z este majorant pentru M și $\forall x \in A (x \text{ majorant pentru } M \rightarrow z \leq x)$ / z este minorant pentru M și $\forall x \in A (x \text{ minorant pentru } M \rightarrow x \leq z)$.

Mulțimi ordonate

Th: Fie $\langle A, \leq \rangle$ o mulțime ordonată și $M \subseteq A$.

- Dacă $x, y \in A$ sunt maxime pentru M , atunci $x = y$; cu alte cuvinte, maximul lui M , dacă există, este unic; el se notează $\max M$ și se mai numește **cel mai mare element al lui M** , sau **ultimul element al lui M** .

Dacă $x, y \in A$ sunt minime pentru M , atunci $x = y$; cu alte cuvinte, minimul lui M , dacă există, este unic; el se notează $\min M$ și se mai numește **cel mai mic element al lui M** , sau **primul element al lui M** .

- $\max M$, dacă există, este și element maximal pentru M , iar M nu mai are alte elemente maximale.

$\min M$, dacă există, este și element minimal pentru M , iar M nu mai are alte elemente minimale.

- Dacă $x, y \in A$ sunt supremumuri pentru M , atunci $x = y$; cu alte cuvinte, supremumul lui M , dacă există, este unic; el se notează $\sup M$.

Dacă $x, y \in A$ sunt infimumuri pentru M , atunci $x = y$; cu alte cuvinte, infimumul lui M , dacă există, este unic; el se notează $\inf M$.

- Dacă există $\max M$, atunci există și $\sup M$ și $\sup M = \max M$.

Dacă există $\min M$, atunci există și $\inf M$ și $\inf M = \min M$.

- Dacă $x, y \in A$, $x \leq y$, atunci x este singurul element minimal, minimul și infimumul mulțimii $\{x, y\}$, iar y este singurul element maximal, maximul și supremumul mulțimii $\{x, y\}$.

Mulțimi ordonate

Observații:

- Un majorant/minorant al lui M nu este neapărat element al lui M ; M poate avea 0 sau mai mulți majoranți / minoranți.

Dacă $z \in A$ este majorant/minorant al lui M și $z \in M$, atunci $z = \max M / \min M$ și M nu are alți majoranți / minoranți.

În particular, dacă există $z = \sup M / \inf M$ și $z \in M$, atunci $z = \max M / \min M$ (și M nu are alți majoranți / minoranți).

- Un element maximal/minimal al lui M este un element al lui M care este mai mare / mai mic decât toate elementele lui M cu care este comparabil (dar nu este neapărat comparabil cu toate); M poate avea 0 sau mai multe elemente maximale / minimale.
- $\max M / \min M$ este un element al lui M comparabil și mai mare / comparabil și mai mic decât toate elementele lui M ; M poate avea 0 sau cel mult un max / min.
- $\sup M / \inf M$ este cel mai mic majorant / cel mai mare minorant al lui M ; M poate avea 0 sau cel mult un sup / inf; max / min sunt noțiuni mai tari decât sup / inf.
- Noțiunile minorant/majorant, element minimal/element maximal, minim/maxim, infimum/supremum, sunt respectiv duale.

Mulțimi ordonate

Exemplu: Fie mulțimea ordonată următoare:



Submulțimea $\{x, y\}$ sunt majoranții z, t , minoranții a, b, u , elementele maximale și minimale x, y , nu sunt min sau max, sunt sup $= z$ și nu sunt inf.

Submulțimea $\{z, x, y, c\}$ sunt majoranții z, t , nu sunt minoranți, sunt sup $= \max = z$ (și singurul element maximal), elementele minimale x, c , nu sunt min sau inf.

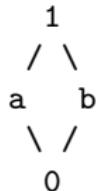
Exemplu: Fie mulțimea ordonată $\langle \mathbb{R}, \leq \rangle$.

- Pentru $M = (0, 1) \subseteq \mathbb{R}$, avem: $1 = \sup M$ (deoarece majoranții lui M sunt elementele mulțimii $[1, \infty)$ iar $1 = \min[1, \infty)$), dar $\not\exists \max M$.
- Pentru $M = (0, 1] \subseteq \mathbb{R}$, avem: $1 = \sup M = \max M$ (majoranții lui M sunt tot elementele mulțimii $[1, \infty)$).

Latici

Def: S.n. **latice Ore** o mulțime ordonată $\langle A, \leq \rangle$ a.î. $\forall x, y \in A, \exists \begin{cases} \inf\{x, y\} \\ \sup\{x, y\} \end{cases}$

Exemplu: Mulțimea ordonată următoare este o latice Ore:



Observăm că $0 = \inf\{a, b\}$, dar $\nexists \min\{a, b\}$, $1 = \sup\{a, b\}$, dar $\nexists \max\{a, b\}$.

Def: S.n. **latice Dedekind** o algebră $\langle A, \vee, \wedge \rangle$ de tip $(2, 2)$, care satisfac
axiomele:

asociativitate: $(x \vee y) \vee z = x \vee (y \vee z)$, $(x \wedge y) \wedge z = x \wedge (y \wedge z)$

comutativitate: $x \vee y = y \vee x$, $x \wedge y = y \wedge x$

absorbție: $x \vee (x \wedge y) = x$, $x \wedge (x \vee y) = x$

Operațiile \vee și \wedge s.n. **disjuncție**, respectiv **conjuncție**.

Th: Într-o latice Dedekind, pentru orice două elemente x, y , avem:

$$x \vee y = y \Leftrightarrow x \wedge y = x.$$

Cele două noțiuni de latice definite mai devreme sunt echivalente, mai exact:

- Daca $\mathcal{A} = \langle A, \leq \rangle$ este o latice Ore, atunci $\Phi(\mathcal{A}) \stackrel{d}{=} \langle A, \vee, \wedge \rangle$, unde:
 $\forall x, y \in A, x \vee y \stackrel{d}{=} \sup\{x, y\}, x \wedge y \stackrel{d}{=} \inf\{x, y\}$
este o latice Dedekind.
- Daca $\mathcal{A} = \langle A, \vee, \wedge \rangle$ este o latice Dedekind, atunci $\Psi(\mathcal{A}) := \langle A, \leq \rangle$, unde:
 $\forall x, y \in A, x \leq y \Leftrightarrow x \vee y = y \quad (\Leftrightarrow x \wedge y = x)$
este o latice Ore.
- Transformările Φ și Ψ sunt inverse una alteia:
 $\forall \mathcal{A}$ latice Ore, $\Psi(\Phi(\mathcal{A})) = \mathcal{A}; \quad \forall \mathcal{A}$ latice Dedekind, $\Phi(\Psi(\mathcal{A})) = \mathcal{A}$.

În cele ce urmează, prin **latice** vom înțelege o latice Dedekind, dar având în vedere echivalența de mai sus, pe o latice vom considera automat atât operațiile \vee, \wedge , cât și relația de ordine \leq . Am putea evidenția toate aceste componente în notația ei scriind: $\langle A, \leq, \vee, \wedge \rangle$. Când operațiile și relația de ordine se subînțeleg din contex, vom desemna însă laticea doar prin A (multimea subiacentă).

Th: $\langle A, \vee, \wedge \rangle$ este o latice $\Leftrightarrow \langle A, \wedge, \vee \rangle$ este o latice
(cele două latice se zic **duale** un celeilalte).

Dacă E este o noțiune sau proprietate formulată pentru o anumită latice, duala ei E^{op} se obține interschimbând peste tot \vee și \wedge (aceste operații sunt duale una celeilalte).

Principul dualității pentru latice rezultă din principul dualității pentru mulțimi ordonate și afirmă că dacă P este o proprietate formulată pentru o anumită latice și considerăm duala ei P^{op} , atunci:

$$P \text{ este adevărată} \Leftrightarrow P^{op} \text{ este adevărată}$$

Așa cum am văzut deja, în general noțiunile / proprietățile legate de latice se dau în perechi de noțiuni / proprietăți duale.

De exemplu, avem noțiunile de majorant / minorant, max / min, etc.; de asemenea, avem câte două legi de asociativitate, comutativitate, absorbție.

Th: Orice latice satisface:

idempotență: $x \vee x = x, x \wedge x = x$

Def: O latice A se zice **completă**, dacă $\forall M \subseteq A, \exists \sup M, \inf M \in A$.

Th: Orice latice finită este completă.

Latici distributive, mărginite, complementate

Th: În orice latice A , următoarele proprietăți sunt echivalente:

- $\forall x, y, z \in A, x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$
- $\forall x, y, z \in A, x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$

Def: O latice se zice **distributivă**, dacă satisfac axioma:

distributivitate: $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$, $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$
(conform th. anterioare, e suficient să cerem doar una dintre cele două relații).

Def: S.n. **latice mărginită** (sau **latice cu prim și ultim element**) o algebră $\langle A, \vee, \wedge, 0, 1 \rangle$ de tip $(2, 2, 0, 0)$ (așadar 0, 1 sunt elemente distinse din A), a.î.:

- $\langle A, \vee, \wedge \rangle$ este latice.
- $0 = \min A$, $1 = \max A$.

Observații:

- Condiția $0 = \min A$ se poate înlocui cu una dintre identitățile $\forall x \in A (0 \vee x = x)$, $\forall x \in A (0 \wedge x = 0)$; condiția $1 = \max A$ se poate înlocui cu una dintre identitățile $\forall x \in A (1 \vee x = 1)$, $\forall x \in A (1 \wedge x = x)$.
- O latice mărginită este în mod necesar nevidă (deoarece trebuie să conțină cel puțin pe 0 și 1), dar nu trebuie neapărat ca $0 \neq 1$; dacă $0 = 1$, atunci A se reduce la un singur element (deoarece $\forall x \in A (0 \leq x \leq 1 = 0)$).
- O noțiune/proprietate referitoare la latici mărginite se dualizează interschimbând \vee cu \wedge și 0 cu 1; principiul dualității se păstrează.

Latici distributive, mărginită, complementate

Def: Fie $\langle A, \vee, \wedge, 0, 1 \rangle$ o latice mărginită.

- Dacă $x \in A$, s.n. **complement** al lui x orice $y \in A$ a.î. $x \vee y = 1$ și $x \wedge y = 0$.
- Un element $x \in A$, se zice **complementat**, dacă are cel puțin un complement.
- Laticea mărginită A se zice **complementată**, dacă orice element al său este complementat.

Th: Într-o latice mărginită și distributivă orice element poate avea cel mult un complement.

Observații:

- Numai o latice mărginită poate fi complementată.
- În orice latice mărginită, 0 are ca unic complement pe 1, 1 are ca unic complement pe 0, iar 0 și 1 sunt singurele elemente comparabile cu un complement al lor.
- Dacă o latice complementată nu este și distributivă, un element poate avea mai multe complemente.
- Dacă o latice complementată este și distributivă, adică este o algebră booleană, orice element x are un complement unic, notat \bar{x} ; în acest caz, corespondența $x \mapsto \bar{x}$ se poate adăuga signaturii algebrei booleene, ca o operație unară (a se vedea mai jos).

Algebrelle booleene

Def: S.n. **algebră booleană** o algebră $\langle A, \vee, \wedge, \neg, 0, 1 \rangle$ de tip $(2, 2, 1, 0, 0)$ care este latice distributivă, mărginită și complementată (cuvântul "mărginită" este redundant). Cu alte cuvinte, ea satisface axiomele:

asociativitate: $(x \vee y) \vee z = x \vee (y \vee z)$, $(x \wedge y) \wedge z = x \wedge (y \wedge z)$

comutativitate: $x \vee y = y \vee x$, $x \wedge y = y \wedge x$

absorbție: $x \vee (x \wedge y) = x$, $x \wedge (x \vee y) = x$

distributivitate: $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$, $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$

mărginire: $0 = \min A$, $1 = \max A$

(sau, echivalent, $0 \vee x = x$, $0 \wedge x = 0$, $1 \vee x = 1$, $1 \wedge x = x$)

complementare: $x \vee \bar{x} = 1$, $x \wedge \bar{x} = 0$

Observații:

- O algebră booleană este nevidă, deoarece conține 0, 1, dar nu trebuie neapărat ca $0 \neq 1$; dacă $0 = 1$, atunci algebra booleană se reduce la un singur element (algebra singleton B_1); dacă algebra are ≥ 2 elemente, atunci în mod necesar $0 \neq 1$; există algebra booleană cu numai două elemente, 0 și 1, notată B_2 .
- Singurele algebrelle booleene total ordonate sunt B_1 și B_2 ; ambele sunt complete.
- O noțiune/proprietate referitoare la algebrelle booleene se dualizează interschimbând \vee cu \wedge , 0 cu 1 și lăsând \neg intact. Astfel, unele noțiuni/proprietăți, de exemplu cele formulate doar în termenii lui \neg , vor fi propriile lor duale (vor fi **autoduale**). Prințipiuul dualității se păstrează.

Algebrelor booleene

Def: În orice algebră booleană se definesc operațiile derivate:

implicația: $x \rightarrow y \stackrel{d}{=} \bar{x} \vee y$

diferența: $x - y \stackrel{d}{=} x \wedge \bar{y}$

echivalența: $x \leftrightarrow y \stackrel{d}{=} (x \rightarrow y) \wedge (y \rightarrow x)$

disjuncția exclusivă (xor): $x \triangle y \stackrel{d}{=} (x \wedge \bar{y}) \vee (\bar{x} \wedge y) = (x - y) \vee (y - x)$

Observație: Operațiile \leftrightarrow și \triangle sunt duale una alteia; fiecare dintre operațiile \rightarrow și $-$ este duala celeilalte compuse cu permutarea operanzilor ($x \rightarrow y$ se interschimbă cu $y - x$).

Algebrelor booleene

Th: În orice algebră booleană au loc următoarele proprietăți:

$$x \rightarrow y = \bar{y} \rightarrow \bar{x}, x - y = \bar{y} - \bar{x}$$

$$(x \leftrightarrow y) \leftrightarrow z = x \leftrightarrow (y \leftrightarrow z), (x \Delta y) \Delta z = x \Delta (y \Delta z)$$

(\leftrightarrow și Δ sunt asociative)

$$x \leftrightarrow y = y \leftrightarrow x, x \Delta y = y \Delta x$$

(\leftrightarrow și Δ sunt comutative)

$$0 \Delta x = x, 1 \leftrightarrow x = x$$

$$1 \Delta x = \bar{x}, 0 \leftrightarrow x = \bar{x}$$

$$x \rightarrow y = \overline{x - y}, x - y = \overline{x \rightarrow y}$$

$$x \Delta y = \overline{x \leftrightarrow y}, x \leftrightarrow y = \overline{x \Delta y}$$

$\bar{\bar{x}} = x$ (legea dublei negații)

$$x \vee (\bar{x} \wedge y) = x \vee y, x \wedge (\bar{x} \vee y) = x \wedge y \text{ (absorbția booleană)}$$

$$\overline{x \vee y} = \bar{x} \wedge \bar{y}, \overline{x \wedge y} = \bar{x} \vee \bar{y} \text{ (legile lui de Morgan)}$$

(în prezența legii dublei negații, acestea se pot formula echivalent:

$$\overline{\bar{x} \vee \bar{y}} = x \wedge y, \overline{\bar{x} \wedge \bar{y}} = x \vee y)$$

$$x \vee y = 1 \text{ și } x \wedge y = 0 \text{ implică } y = \bar{x} \text{ (unicitatea complementului)}$$

$$x \vee y = 0 \text{ d.d. } x = y = 0, \quad x \wedge y = 1 \text{ d.d. } x = y = 1$$

Observație: Constatăm că, în baza principiului dualității, proprietățile apar în perechi de proprietăți duale. Legea dublei negații este autoduală, de aceea apare singură.

Algebrelor booleene

Exemplu: Dacă M este o mulțime, mulțimea $\mathcal{P}(M)$ a părților lui M este o algebră booleană completă, cu operațiile:

$\vee = \cup$ (reuniunea)

$\wedge = \cap$ (intersecția)

$\neg = C$ (complementara unei submulțimi)

$0 = \emptyset$ (mulțimea vidă)

$1 = M$ (mulțimea totală)

Rezultă că avem relația de ordine:

$\leq = \subseteq$ (incluziunea)

și următoarele operații derive:ate:

$- = \setminus$ (diferența de mulțimi)

$\Delta = \Delta$ (diferența simetrică de mulțimi).

Exemplu: Algebra booleană cu două elemente $B_2 = \{0, 1\}$ are operațiile date de următoarele tabele:

x	y	$x \vee y$	$x \wedge y$	x	\bar{x}
0	0	0	0	0	1
0	1	1	0	1	0
1	0	1	0		
1	1	1	1		

Relația de ordine este: $0 \leq 1$.

Această algebră este importantă în cele ce urmează și vom reveni asupra ei mai târziu.

Inele booleene

Def: S.n. **inel unitar** o algebră $\langle A, +, \cdot, -, 0, 1 \rangle$ de tip $(2, 2, 1, 0, 0,)$ a.î.:

- $\langle A, +, -, 0 \rangle$ este grup abelian, adică satisfacă:

$$(x + y) + z = x + (y + z)$$

$$x + 0 = 0 + x = x$$

$$x + (-x) = (-x) + x = 0$$

$$x + y = y + x$$

- $\langle A, \cdot, 1 \rangle$ este semigrup unitar (monoid), adică satisfacă:

$$(x \cdot y) \cdot z = x \cdot (y \cdot z)$$

$$x \cdot 1 = 1 \cdot x = x$$

- operația \cdot este distributivă față de operația $+$, adică satisfacă:

$$x \cdot (y + z) = x \cdot y + x \cdot z, (y + z) \cdot x = y \cdot x + z \cdot x$$

(în notație, am presupus că \cdot are precedență mai mare decât $+$ și am omis parantezele inutile).

Inelul unitar este **comutativ**, dacă \cdot este comutativă:

$$x \cdot y = y \cdot x$$

Def: S.n. **inel boolean** un inel unitar $\langle A, +, \cdot, -, 0, 1 \rangle$ a.î. operația \cdot este idempotentă, adică satisfacă:

$$x \cdot x = x$$

Inele booleene

Th: Orice inel boolean satisfacă:

$$x + x = 0$$

$$x \cdot y = y \cdot x \text{ (adică inelul boolean este comutativ)}$$

Exemplu: Inelul claselor de resturi modulo 2, $\langle \mathbb{Z}_2, +, \cdot, -, \hat{0}, \hat{1} \rangle$, este inel boolean.

Inele booleene

Noțiunile de algebră booleană și inel boolean sunt echivalente, mai exact:

- Dacă $\mathcal{A} = \langle A, \vee, \wedge, \bar{}, 0, 1 \rangle$ este o algebru a booleană, atunci $\rho(\mathcal{A}) \stackrel{d}{=} \langle A, +, \cdot, -, 0, 1 \rangle$, unde:
 $\forall x, y \in A, x + y \stackrel{d}{=} x \Delta y = (x \wedge \bar{y}) \vee (\bar{x} \wedge y), x \cdot y \stackrel{d}{=} x \wedge y$ și $\forall x \in A, -x = x$ este un inel boolean.
- Dacă $\mathcal{A} = \langle A, +, \cdot, -, 0, 1 \rangle$ este un inel boolean, atunci $\beta(\mathcal{A}) \stackrel{d}{=} \langle A, \vee, \wedge, \bar{}, 0, 1 \rangle$, unde:
 $\forall x, y \in A, x \vee y \stackrel{d}{=} x + y + x \cdot y, x \wedge y \stackrel{d}{=} x \cdot y$ și $\forall x, y \in A, \bar{x} = x + 1$ este o algebră booleană.
- Transformările ρ și β sunt inverse una alteia:
 $\forall \mathcal{A}$ algebră booleană, $\beta(\rho(\mathcal{A})) = \mathcal{A}$; $\forall \mathcal{A}$ inel boolean, $\rho(\beta(\mathcal{A})) = \mathcal{A}$.

Cuprins

1 Performanța calculatoarelor

Conceptul de performanță
Măsurarea performanței

2 Aritmetică sistemelor de calcul

Reprezentarea numerelor în matematică
Reprezentarea numerelor în calculator
Reprezentarea numerelor naturale ca întregi fără semn
Reprezentarea numerelor întregi în complement față de 2
Reprezentarea numerelor reale în virgulă mobilă

3 Logica pentru calculatoare

Algebre booleene
Funcții booleene
Particularizare la cazul B_2

4 Circuite logice

Circuite logice, Sisteme digitale
0-DS (Circuite combinaționale, Funcții booleene)
1-DS (Memorii)
2-DS (Automate finite)
Algoritmi de înmulțire și împărțire hardware
3-DS (Procesoare) și 4-DS (Calculatoare)
 n -DS, $n > 4$

Funcții booleene

Fie $\langle B, \vee, \wedge, \neg, 0, 1 \rangle$ o algebră booleană și $n \in \mathbb{N}$, $n \geq 1$.

Fie B^{B^n} mulțimea funcțiilor definite pe B^n cu valori în B .

Printre elementele lui B^{B^n} se află:

- **funcțiile constante:** $f_a : B^n \longrightarrow B$, $f_a(x_1, \dots, x_n) = a$, $a \in B$;
- **funcțiile proiecție:** $\pi_i : B^n \rightarrow B$, $\pi_i(x_1, \dots, x_n) = x_i$, $1 \leq i \leq n$.

Th: B^{B^n} devine algebră booleană cu operațiile definite astfel:

$$(f \vee g)(x_1, \dots, x_n) \stackrel{d}{=} f(x_1, \dots, x_n) \vee g(x_1, \dots, x_n)$$

$$(f \wedge g)(x_1, \dots, x_n) \stackrel{d}{=} f(x_1, \dots, x_n) \wedge g(x_1, \dots, x_n)$$

$$\bar{f}(x_1, \dots, x_n) \stackrel{d}{=} \overline{f(x_1, \dots, x_n)}$$

$$0(x_1, \dots, x_n) \stackrel{d}{=} 0$$

$$1(x_1, \dots, x_n) \stackrel{d}{=} 1$$

Funcții booleene

Def: Funcțiile booleene din B^{B^n} se definesc recursiv astfel:

- (i) Dacă $f \in B^{B^n}$ este o funcție constantă sau funcție proiecție, atunci f este funcție booleană.
- (ii) Dacă $f, g \in B^{B^n}$ sunt funcții booleene, atunci $f \vee g$, $f \wedge g$ sunt funcții booleene; dacă $f \in B^{B^n}$ este funcție booleană, atunci \bar{f} este funcție booleană.
- (iii) Toate funcțiile booleene din B^{B^n} se obțin aplicând de un număr finit de ori (i) și (ii).

Observații:

- Mulțimea funcțiilor booleene din B^{B^n} este cea mai mică submulțime $F \subseteq B^{B^n}$ a.î.:
 - (1) F conține funcțiile constante (în particular funcția constantă 0 și funcția constantă 1) și funcțiile proiecție.
 - (2) F este închisă la operațiile \vee , \wedge , $\bar{}$: dacă $f, g \in F$, atunci $f \vee g, f \wedge g \in F$, dacă $f \in F$, atunci $\bar{f} \in F$.

O submulțime a lui B^{B^n} ce satisfacă (1), (2) se zice **închisă boolean**; deci, mulțimea funcțiilor booleene din B^{B^n} este cea mai mică submulțime a lui B^{B^n} închisă boolean.

- Într-o altă terminologie, mulțimea funcțiilor booleene din B^{B^n} este subalgebra booleană a lui B^{B^n} generată de funcțiile constante și funcțiile proiecție.
- O funcție din B^{B^n} este funcție booleană d.d. se poate scrie ca o expresie algebrică booleană (termen boolean) de funcții constante și funcții proiecție.

Funcții booleene

Exemplu: Dacă $a, b, c \in B$ și $n \geq 4$, atunci

$$(\overline{f_a \vee f_b} \wedge \pi_4) \vee f_c$$

este o funcție booleană.

Aplicând proprietățile de calcul într-o algebră booleană, putem exprima o funcție booleană prin mai multe expresii echivalente. Am dori ca printre ele să existe și o formă (expresie) standard, în care să putem exprima în mod unic orice funcție booleană.

Teorema următoare afirmă că există chiar două (duale una celeilalte): forma normală disjunctivă și forma normală conjunctivă.

Funcții booleene

Th: Pentru $f \in B^{B^n}$ sunt echivalente:

(1) f este funcție booleană.

(2) $\forall x_1, \dots, x_n \in B$,

$$f(x_1, \dots, x_n) = \bigvee_{\alpha_1, \dots, \alpha_n \in \{0,1\}} (f(\alpha_1, \dots, \alpha_n) \wedge x_1^{\alpha_1} \wedge \dots \wedge x_n^{\alpha_n})$$

(forma normală disjunctivă, FND)

$$(3) \forall x_1, \dots, x_n \in B, f(x_1, \dots, x_n) = \bigwedge_{\alpha_1, \dots, \alpha_n \in \{0,1\}} (f(\overline{\alpha_1}, \dots, \overline{\alpha_n}) \vee x_1^{\alpha_1} \vee \dots \vee x_n^{\alpha_n})$$

(forma normală conjunctivă, FNC)

unde am notat: $x_i^0 = \overline{x_i}$, $x_i^1 = x_i$, $1 \leq i \leq n$.

Observații:

- Dacă interschimbăm α_i cu $\overline{\alpha_i}$, $1 \leq i \leq n$ și ținem cont că funcția $\{0,1\}^n \rightarrow \{0,1\}^n$, $(\alpha_1, \dots, \alpha_n) \mapsto (\overline{\alpha_1}, \dots, \overline{\alpha_n})$ este o bijectie, (3) poate fi înlocuit cu:

$$(3') \forall x_1, \dots, x_n \in B, f(x_1, \dots, x_n) = \bigwedge_{\alpha_1, \dots, \alpha_n \in \{0,1\}} (f(\alpha_1, \dots, \alpha_n) \vee x_1^{\overline{\alpha_1}} \vee \dots \vee x_n^{\overline{\alpha_n}})$$

- Scrierea în FND, FNC, este unică, abstracție făcând de o permutare a factorilor/termenilor pe baza asociativității și comutativității lui \wedge/\vee .

Consecință: Orice funcție booleană : $B^n \rightarrow B$ este unic determinată de valorile ei pe $\{0,1\}^n$ ($\subseteq B^n$).

Funcții booleene

Th: Dacă $f : B^n \rightarrow B$, $n \geq 1$, este o funcție booleană și $1 \leq i \leq n$, atunci:
 $\forall x_1, \dots, x_n \in B, f(x_1, \dots, x_n) =$
 $(x_i \wedge f(x_1, \dots, x_{i-1}, 1, x_{i+1}, \dots, x_n)) \vee (\overline{x_i} \wedge f(x_1, \dots, x_{i-1}, 0, x_{i+1}, \dots, x_n))$

Cuprins

1 Performația calculatoarelor

Conceptul de performanță
Măsurarea performanței

2 Aritmetică sistemelor de calcul

Reprezentarea numerelor în matematică
Reprezentarea numerelor în calculator
Reprezentarea numerelor naturale ca întregi fără semn
Reprezentarea numerelor întregi în complement față de 2
Reprezentarea numerelor reale în virgulă mobilă

3 Logica pentru calculatoare

Algebre booleene
Funcții booleene
Particularizare la cazul B_2

4 Circuite logice

Circuite logice, Sisteme digitale
0-DS (Circuite combinaționale, Funcții booleene)
1-DS (Memorii)
2-DS (Automate finite)
Algoritmi de înmulțire și împărțire hardware
3-DS (Procesoare) și 4-DS (Calculatoare)
 n -DS, $n > 4$

Algebra booleană B_2

Algebra booleană cu două elemente B_2 are o deosebită importanță în informatică.

Multe dintre noțiunile și proprietățile referitoare la algebrelor și funcțiilor booleene în general, pe care le-am prezentat mai devreme, în cazul B_2 capătă o formă mai simplă, pe care o vom prezenta în continuare.

Vom folosi notații din aritmetică pentru unele dintre operațiile booleene, care să ne permită operarea mai ușoară cu formulele de calcul:

- disjuncția va fi notată $+$ (în loc de \vee);
- conjuncția va fi notată \cdot sau juxtapunere (în loc de \wedge);
- disjuncția exclusivă (xor) va fi notată \oplus (în loc de Δ).

De asemenea, vom folosi uneori terminologia din aritmetică, spunând: "sumă", "produs", etc., în loc de: "disjuncție", "conjuncție", etc..

Vom considera următoarea precedență între operații, în baza căreia vom putea omite o parte din paranteze: $+, \oplus < \cdot < -$

Algebra booleană B_2

Reamintim câteva dintre formulele anterioare, scrise cu noile notații:

Axiomele prin care se definește o algebră booleană $\langle A, +, \cdot, \bar{}, 0, 1 \rangle$ sunt:

asociativitate: $(x + y) + z = x + (y + z)$, $(xy)z = x(yz)$

comutativitate: $x + y = y + x$, $xy = yx$

absorbție: $x + xy = x$, $x(x + y) = x$

distributivitate: $x(y + z) = xy + xz$, $x + yz = (x + y)(x + z)$

mărginire: $0 + x = x$, $0x = 0$, $1 + x = 1$, $1x = x$

complementare: $x + \bar{x} = 1$, $x\bar{x} = 0$

Operațiile derivate sunt:

implicația: $x \rightarrow y \stackrel{d}{=} \bar{x} + y$

diferența: $x - y \stackrel{d}{=} x\bar{y}$

echivalența: $x \leftrightarrow y \stackrel{d}{=} (x \rightarrow y)(y \rightarrow x)$

disjuncția exclusivă (xor): $x \oplus y \stackrel{d}{=} x\bar{y} + \bar{x}y = (x - y) + (y - x)$

Algebra booleană B_2

Alte proprietăți valabile în algebrelor booleene:

$$x + x = x, \quad xx = x \quad (\text{idempotență, valabilă în orice latice})$$

$$x \rightarrow y = \bar{y} \rightarrow \bar{x}, \quad x - y = \bar{y} - \bar{x}$$

$$(x \leftrightarrow y) \leftrightarrow z = x \leftrightarrow (y \leftrightarrow z), \quad (x \oplus y) \oplus z = x \oplus (y \oplus z)$$

(\leftrightarrow și \oplus sunt asociative)

$$x \leftrightarrow y = y \leftrightarrow x, \quad x \oplus y = y \oplus x \quad (\leftrightarrow și \oplus sunt comutative)$$

$$0 \oplus x = x, \quad 1 \leftrightarrow x = x, \quad 1 \oplus x = \bar{x}, \quad 0 \leftrightarrow x = \bar{x}$$

$$x \oplus y = \overline{x \leftrightarrow y}, \quad x \leftrightarrow y = \overline{x \oplus y}, \quad x \rightarrow y = \overline{x - y}, \quad x - y = \overline{x \rightarrow y}$$

$$\bar{\bar{x}} = x \quad (\text{legea dublei negații})$$

$$x + \bar{x}y = x + y, \quad x(\bar{x} + y) = xy \quad (\text{absorbția booleană})$$

$$\overline{x + y} = \bar{x} \bar{y}, \quad \overline{xy} = \bar{x} + \bar{y} \text{ sau } \bar{\bar{x}} + \bar{\bar{y}} = xy, \quad \overline{\bar{x} \bar{y}} = x + y \quad (\text{legile lui Morgan})$$

$$x + y = 1 \text{ și } xy = 0 \text{ implică } y = \bar{x} \quad (\text{unicitatea complementului})$$

$$x + y = 0 \text{ d.d. } x = y = 0, \quad xy = 1 \text{ d.d. } x = y = 1$$

Algebra booleană B_2

Algebra booleană $\langle B_2, +, \cdot, \bar{}, 0, 1 \rangle$ este caracterizată prin următoarele:

- Mulțimea subiacentă se reduce la cele două elemente distinse, minim și maxim, iar acestea sunt diferite între ele: $B_2 = \{0, 1\}$, $0 \neq 1$. De obicei, elementele 0/1 au semnificația valorilor de adevăr din logica clasică *fals/adevărat* (*false/true*); de aceea, terminologia legată de această algebră este preluată din logică: 0 și 1 sunt valori de adevăr, operațiile $+, \cdot, \bar{}$ s.n. respectiv "sau", "și", "not" ("or", "and", "not"), tabelele prin care sunt definite ele s.n. tabele de adevăr, etc.. Alteori, elementele 0/1 pot semnifica valorile unor biți, numerele 0/1, cifrele binare, etc.
- Operațiile și relația de ordine pe B_2 sunt definite astfel:

x	y	$x + y$	xy	$x \rightarrow y$	$x - y$	$x \leftrightarrow y$	$x \oplus y$	x	\bar{x}
0	0	0	0	1	0	1	0	0	1
0	1	1	0	1	0	0	1	1	0
1	0	1	0	0	1	0	1	0	1
1	1	1	1	1	0	1	0	1	0

Operația 0-ară 0 este elementul 0,

Operația 0-ară 1 este elementul 1,

Relația de ordine este: $0 \leq 1$.

Algebra booleană B_2

Sigurele funcții constante : $B_2^n \rightarrow B_2$, $n \geq 1$, sunt:

funcția constantă 0: $0(x_1, \dots, x_n) \stackrel{d}{=} 0$

funcția constantă 1: $1(x_1, \dots, x_n) \stackrel{d}{=} 1$

Acestea sunt elementele 0, respectiv 1, ale algebrei booleene $B_2^{B_2^n}$ și astfel orice expresie algebraică booleană de funcții din $B_2^{B_2^n}$ se poate transforma într-o echivalentă fără aceste elemente.

Th: Orice funcție $f : B_2^n \rightarrow B_2$, $n \geq 1$, este funcție booleană (i.e. se poate scrie ca o expresie algebraică booleană de funcții constante și funcții proiecție).

Astfel, în cazul B_2 putem elimina (i), (ii), (iii) din definiția funcțiilor booleene și să numim funcție booleană orice funcție : $B_2^n \rightarrow B_2$, $n \geq 1$.

Algebra booleană B_2

În cele ce urmează, vom folosi însă o definiție mai generală:

Def: S.n. **funcție booleană** orice funcție $f : B_2^n \rightarrow B_2^k$, $n, k \geq 1$.

Dacă $k = 1$, f se mai numește **funcție booleană scalară**.

Dacă $k > 1$, f se mai numește **funcție booleană vectorială**.

Componentele unei funcții booleene vectoriale $f : B_2^n \rightarrow B_2^k$ sunt funcțiile $f_i \stackrel{def}{=} \pi_i \circ f : B_2^n \rightarrow B_2$, unde $\pi_i : B_2^k \rightarrow B_2$ este funcția proiecție, $1 \leq i \leq k$.

Observație: O funcție booleană vectorială $f : B_2^n \rightarrow B_2^k$ se poate asimila cu sistemul componentelor sale: $f = (f_1, \dots, f_k)$ (pe elemente: $f(x_1, \dots, x_n) = (f_1(x_1, \dots, x_n), \dots, f_k(x_1, \dots, x_n))$), iar componentele $f_i : B_2^n \rightarrow B_2$ sunt funcții booleene scalare.

Algebra booleană B_2

O funcție booleană poate fi dată în mai multe feluri:

- Informal (dar suficient de precis):

Fie funcția booleană care asociază unui sistem de trei biți bitul majoritar.

- Printr-o formulă, de exemplu o expresie algebraică booleană:

Fie $f : B_2^3 \rightarrow B_2$, $f(x, y, z) = (z(\overline{y} \oplus \overline{z}) + x\overline{y})$, $x + z$, $z \oplus \overline{x}\overline{y}$, y)

- Printr-un tabel (daca funcția este definită pe B_2^n , tabelul are 2^n linii):

Fie $f : B_2^3 \rightarrow B_2^2$, dată prin tabelul:

x	y	z	$f_1(x, y, z)$	$f_2(x, y, z)$
0	0	0	1	1
0	0	1	0	1
0	1	0	0	0
0	1	1	0	0
1	0	0	1	1
1	0	1	0	0
1	1	0	1	1
1	1	1	0	0

Algebra booleană B_2

Când descriem o funcție booleană printr-un tabel, trebuie să fixăm o ordine a semnificațiilor variabilelor și o ordine de amplasare a coloanelor acestora, în concordanță cu semnificația; în exemplul anterior, ordinea semnificațiilor variabilelor este: $x > y > z$ (x este mai semnificativ decât y , care este mai semnificativ decât z), iar coloanele variabilelor au fost amplasate în ordinea descrescătoare a semnificațiilor, de la stânga la dreapta:

x	y	z	$f_1(x, y, z)$	$f_2(x, y, z)$
-----	-----	-----	----------------	----------------

De asemenea, este bine să generăm sistemele de valori ale variabilelor pe linii după o anumită regulă; în exemplul anterior, aceste sisteme de valori au fost generate în ordine lexicografică, de sus în jos.

Algebra booleană B_2

Când procedăm astfel, tabelul are următoarele proprietăți:

- Pe prima coloană, cea a variabilei celei mai semnificative, valorile alternează cu o perioadă mare: avem jumătate 0, apoi jumătate 1; pe coloana a doua, cea a variabilei următoare ca semnificație, perioada se înjumătășește: un sfert 0, apoi un sfert 1, apoi un sfert 0, apoi un sfert 1; pe coloana variabilei următoare ca semnificație, perioada se înjumătășește iar: o optime 0, apoi o optime 1, etc..
- Pe fiecare linie, sistemul valorilor variabilelor constituie transcrierea în baza 2 a numărului de ordine al liniei, numărând de la 0, de sus în jos; de exemplu, sistemul valorilor variabilelor de pe linia 3 este 011 (transcrierea binară a lui 3):

	x	y	z	$f_1(x, y, z)$	$f_2(x, y, z)$
0	0	0	0	1	1
1	0	0	1	0	1
2	0	1	0	0	0
3	0	1	1	0	0
4	1	0	0	1	1
5	1	0	1	0	0
6	1	1	0	1	1
7	1	1	1	0	0

Ambele proprietăți vor avea semnificație și se vor dovedi utile la realizarea circuitelor logice.

Algebra booleană B_2

Observație:

Dacă am fixat ordinea semnificațiilor variabilelor, cea de amplasare a coloanelor acestora și regula după care generăm sistemele de valori ale variabilelor pe linii, pentru a descrie o funcție booleană scalară $f : B_2^n \rightarrow B_2$, $n \geq 1$, este suficient să dăm sistemul de valori din coloana funcției (deoarece putem deduce pentru ce valori ale variabilelor corespunde o anumită valoare a funcției).

Un asemenea sistem de valori este transcrierea în baza 2 a unui număr întreg de la 0 la $2^n - 1$, iar acest număr este unic determinat de f .

În exemplul de mai sus, f_1 poate fi descrisă de sistemul de 8 valori (octetul) 10001010 (de exemplu, bitul 3 (numărând de la stânga și de la 0) este 0 și este valoarea $f_1(x, y, z)$ corespunzătoare liniei 3 din tabel, care corespunde valorilor variabilelor ce transcriu în baza 2 numărul 3: $x = 0, y = 1, z = 1$). Acest octet este transcrierea în baza 2 a numărului 81 (ținând cont că bitul scris în stânga este cel de rang 0, care corespunde unităților).

Astfel, funcțiile booleene scalare de n variabile, $n \geq 1$, sunt în corespondență bijectivă cu numerele $0, \dots, 2^n - 1$, ceea ce ne permite să mutăm studiul acestor funcții într-un cadru aritmetic.

În particular, numărul funcțiilor booleene scalare de n variabile, $n \geq 1$, este 2^{2^n} .

Numărul funcțiilor booleene $f : B_2^n \rightarrow B_2^k$, $n, k \geq 1$, este $(2^{2^n})^k$.

Algebra booleană B_2

Pentru a trece de la un mod de descriere a unei funcții booleene la altul, putem proceda astfel:

- Trecerea de la tabel la formulă (expresie):

Pentru fiecare componentă a funcției, din tabel se poate obține FND sau FNC (vom vedea mai târziu).

- Trecerea de la formulă (expresie) la tabel:

De la stânga la dreapta, asociem coloane variabilelor în ordinea descrescătoare a semnificației, apoi subexpresiilor din formulă, în ordinea crescătoare a agregării lor din alte subexpresii, până obținem coloane corespunzătoare componentelor funcției.

Procedând astfel, putem completa tabelul pe coloane de la stânga la dreapta, fiecare nouă coloană obținându-se din niște coloane deja complete, folosind o singură operație booleană.

Algebra booleană B_2

Astfel, pentru funcția din exemplul anterior:

$$f : B_2^3 \longrightarrow B_2^4, f(x, y, z) = (\underbrace{z(y \oplus \bar{z}) + x\bar{y}}, \underbrace{x+z}, \underbrace{z \oplus \bar{x}\bar{y}}, \underbrace{y})$$

avem:

x	y = f_4	z	\bar{y}	\bar{z}	$x\bar{y}$	$y \oplus \bar{z}$	$\bar{y} \oplus \bar{z}$	$\bar{y} \oplus \bar{z} + x\bar{y}$	f_1	f_2	f_3	f_4	xy	$\bar{x}\bar{y}$	f_3
0	0	0	1	1	0	1	0	0	0	0	0	0	0	1	1
0	0	1	1	0	0	0	1	1	1	1	0	0	0	1	0
0	1	0	0	1	0	0	1	1	0	0	0	0	1	1	1
0	1	1	0	0	0	1	0	0	0	1	0	1	0	1	0
1	0	0	1	1	1	1	0	1	0	1	0	1	0	1	1
1	0	1	1	0	1	0	1	1	1	1	1	0	1	1	0
1	1	0	0	1	0	0	1	1	0	0	1	1	0	0	0
1	1	1	0	0	0	1	0	0	0	1	1	1	0	0	1

De exemplu, coloana a 9-a ($\bar{y} \oplus \bar{z} + x\bar{y}$) s-a calculat din coloanele a 8-a ($\bar{y} \oplus \bar{z}$) și a 6-a ($x\bar{y}$) folosind operația +.

Algebra booleană B_2

Sfat:

La completarea unei coloane, în loc să urmărim pe fiecare linie valorile sursă pentru a calcula valoarea rezultată, se poate dovedi mai util să deducem o regulă de aranjare a valorilor din coloana rezultată din regulile de aranjare a valorilor în coloanele sursă și să completăm noua coloană direct.

De exemplu, observăm că în coloana y alternează doi de 0, doi de 1, ...; atunci deducem că în coloana \bar{y} alternează doi de 1, doi de 0, ... și putem completa coloana \bar{y} direct, fără a mai privi la fiecare linie în coloana y .

În cazul coloanei $x\bar{y}$, observăm că x începe cu 4 linii 0, care este element anulator la conjuncție, deci și coloana $x\bar{y}$ începe tot cu 4 linii 0; apoi x are 4 linii 1, care este element neutru la conjuncție, deci următoarele 4 linii din coloana $x\bar{y}$ se copiază din coloana \bar{y} , unde deja am văzut că ultimile 4 linii alternează doi de 1, doi de 0.

Algebra booleană B_2

În cazul B_2 , teorema și observațiile referitoare la FND și FNC se simplifică, deoarece orice funcție $f : B_2^n \rightarrow B_2$, $n \geq 1$, este funcție booleană și pentru orice $\alpha_1, \dots, \alpha_n \in \{0, 1\} = B_2$, $f(\alpha_1, \dots, \alpha_n)$ poate fi doar 0 sau 1:

Th: Pentru orice funcție $f : B_2^n \rightarrow B_2$, $n \geq 1$, (i.e. orice funcție booleană scalară) avem:

$$(2) \forall x_1, \dots, x_n \in B_2, f(x_1, \dots, x_n) = \sum_{\substack{\alpha_1, \dots, \alpha_n \in \{0, 1\} \\ f(\alpha_1, \dots, \alpha_n)=1}} x_1^{\alpha_1} \cdots x_n^{\alpha_n}$$

(forma normală disjunctivă, FND)

$$(3) \forall x_1, \dots, x_n \in B_2, f(x_1, \dots, x_n) = \prod_{\substack{\alpha_1, \dots, \alpha_n \in \{0, 1\} \\ f(\overline{\alpha_1}, \dots, \overline{\alpha_n})=0}} (x_1^{\alpha_1} + \cdots + x_n^{\alpha_n})$$

(forma normală conjunctivă, FNC)

$$(3') \forall x_1, \dots, x_n \in B_2, f(x_1, \dots, x_n) = \prod_{\substack{\alpha_1, \dots, \alpha_n \in \{0, 1\} \\ f(\alpha_1, \dots, \alpha_n)=0}} (x_1^{\overline{\alpha_1}} + \cdots + x_n^{\overline{\alpha_n}})$$

(forma normală conjunctivă, FNC)

unde am notat: $x_i^0 = \overline{x_i}$, $x_i^1 = x_i$, $1 \leq i \leq n$.

Scrierea în FND, FNC, este unică, abstracție făcând de o permutare a factorilor/termenilor pe baza asociativității și comutativității lui $\cdot / +$.

Pentru realizarea circuitelor logice ne va fi mai utilă FNC dată de (3'), de aceea, în cele ce urmează, pe aceasta o vom numi FNC.

Algebra booleană B_2

Pentru a determina FND și FNC pentru o funcție booleană scalară $f : B_2^n \rightarrow B_2$, $n \geq 1$, putem proceda astfel:

- Dacă funcția este dată printr-un tabel:

Determinarea FND:

- Considerăm liniile tabelului corespunzătoare valorii $f(\alpha_1, \dots, \alpha_n) = 1$;
- Pentru fiecare linie de forma $\alpha_1, \dots, \alpha_n, 1 (= f)$, considerăm produsul (conjuncția) $x_1^{\alpha_1} \cdots x_n^{\alpha_n}$ (deci, pentru orice $1 \leq i \leq n$, x_i apare negat dacă $\alpha_i = 0$ și nenegat dacă $\alpha_i = 1$);
- Considerăm suma (disjuncția) produselor.

Determinarea FNC:

- Considerăm liniile tabelului corespunzătoare valorii $f(\alpha_1, \dots, \alpha_n) = 0$;
- Pentru fiecare linie de forma $\alpha_1, \dots, \alpha_n, 0 (= f)$, considerăm suma (disjuncția) $x_1^{\overline{\alpha_1}} + \cdots + x_n^{\overline{\alpha_n}}$ (deci, pentru orice $1 \leq i \leq n$, x_i apare negat dacă $\alpha_i = 1$ și nenegat dacă $\alpha_i = 0$);
- Considerăm produsul (conjuncția) sumelor.

Algebra booleană B_2

Exemplu: Determinați FND, FNC, pentru $f : B_2^3 \rightarrow B_2$, dată prin tabelul:

x	y	z	$f(x, y, z)$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Numerotând liniile tabelului de sus în jos de la 0 la 7, liniile corespunzătoare valorii $f = 1$ sunt: 1, 2, 4, 6, 7, iar cele corespunzătoare valorii $f = 0$ sunt: 0, 3, 5. Rezultă:

$$\text{FND: } f(x, y, z) = \bar{x} \bar{y} z + \bar{x} y \bar{z} + x \bar{y} \bar{z} + x y \bar{z} + x y z$$

$$\text{FNC: } f(x, y, z) = (x + y + z)(x + \bar{y} + \bar{z})(\bar{x} + y + \bar{z})$$

Algebra booleană B_2

Constatăm că sistemul de negări/nenegări de deasupra unui produs/sume transcrie sistemul de valori 0 și 1 ale variabilelor din linia corespunzătoare produsului/sumei respective, care la rândul lui transcrie în baza 2 numărul de ordine al liniei; diferența este că la FND se pune $\bar{}$ la 0, iar la FNC se pune $\bar{}$ la 1:

	x	y	z	$f(x,y,z)$		1	2	4	6	7
0	0	0	0	0	FND:	$\bar{x}\bar{y}z + \bar{x}y\bar{z} + x\bar{y}\bar{z} + xy\bar{z} + xy\bar{y}z$				
1	0	0	1	1		0	3	5		
2	0	1	0	1	FNC:	$(x+y+z)(x+\bar{y}+\bar{z})(\bar{x}+y+\bar{z})$				
3	0	1	1	0						
4	1	0	0	1						
5	1	0	1	0						
6	1	1	0	1						
7	1	1	1	1						

Astfel, în cazul FND, sistemul $\bar{}\bar{}\bar{}$ de deasupra produsului $x\bar{y}\bar{z}$ transcrie sistemul de valori ale variabilelor din linia corespunzătoare, 100 ($\bar{} = 0$), care transcrie în baza 2 numărul liniei, 4.

Similar, în cazul FNC, sistemul $\bar{}\bar{}\bar{}$ de deasupra sumei $x+\bar{y}+\bar{z}$ transcrie sistemul de valori ale variabilelor din linia corespunzătoare, 011 ($\bar{} = 1$), care transcrie în baza 2 numărul liniei, 3.

Algebra booleană B_2

Astfel, dacă o funcție booleană scalară $f : B_2^n \rightarrow B_2$, $n \geq 1$, este descrisă printr-un tabel construit cu respectarea convențiilor anterioare privind ordinea semnificațiilor variabilelor, cea de amplasare a coloanelor acestora și regula după care generăm sistemele de valori ale variabilelor pe linii, putem determina FND și FNC ale funcției mai repede, fără să urmărim pe fiecare linie valorile variabilelor, astfel:

- FND: scriem atâtea produse $x_1 \cdots x_n$, cu $+$ între ele, câte linii din tabel conțin $f = 1$, apoi transcriem numerele de ordine ale acestor linii prin sisteme de \cup și \cap deaspre lor ($\neg = 0$).
- FNC: scriem atâtea sume $(x_1 + \cdots + x_n)$, cu \cdot între ele, câte linii din tabel conțin $f = 0$, apoi transcriem numerele de ordine ale acestor linii prin sisteme de \cup și \cap deaspre lor ($\neg = 1$).

Algebra booleană B_2

Observație:

Pentru o funcție booleană scalară, numărul de produse ale FND nu trebuie să coincidă neapărat cu numărul de sume ale FNC (în exemplul precedent, FND avea 5 produse, FNC avea 3 sume); astfel, FND și FNC nu sunt neapărat la fel de complexe.

În practică, putem prefera pe cea mai simplă dintre ele (urmărim coloana valorilor funcției și determinăm care valoare apare mai rar: 0 sau 1).

Pentru realizarea circuitelor logice însă, va avea o deosebită importanță FND.

Algebra booleană B_2

- Dacă funcția este dată printr-o formulă (expresie):

Determinarea FND:

- Aplicând proprietățile de calcul într-o algebră booleană (de exemplu legile de distributivitate), transformăm expresia prin care este dată funcția a.î. să fie o sumă (disjuncție) de produse (conjuncții) de variabile cu/fără $\bar{}$.
- La fiecare produs, înmulțim (conjugăm) câte un factor de forma $(t + \bar{t})$, pentru fiecare variabilă t care lipsește din produsul respectiv (în fapt, am înmulțit niște 1, care nu afectează valoarea produsului).
- Desfacem parantezele, folosind distributivitatea lui \cdot față de $+$ (și alte proprietăți de calcul într-o algebră booleană), a.î. expresia să devină din nou o sumă de produse; acum toate produsele vor conține toate variabilele, dar unele produse pot să se repete.
- Eliminăm produsele dupicate, folosind idempotența lui $+$ (și alte proprietăți de calcul într-o algebră booleană).

Algebra booleană B_2

Determinarea FNC:

- Aplicând proprietățile de calcul într-o algebră booleană (de exemplu legile de distributivitate), transformăm expresia prin care este dată funcția a.î. să fie un produs (conjuncție) de sume (disjuncții) de variabile cu/fără \neg .
- La fiecare sumă, adunăm (disjungem) câte un termen de forma $t\bar{t}$, pentru fiecare variabilă t care lipsește din suma respectivă (în fapt, am adunat niște 0, care nu afectează valoarea sumei).
- Folosind distributivitatea lui $+$ față de \cdot (și alte proprietăți de calcul într-o algebră booleană), transformăm expresia a.î. să devină din nou un produs de sume; acum toate sumele vor conține toate variabilele, dar unele sume pot să se repete.
- Eliminăm sumele dupicate, folosind idempotența lui \cdot (și alte proprietăți de calcul într-o algebră booleană).

Algebra booleană B_2

Observație:

Dacă funcția este dată printr-o formulă (expresie), o altă metodă de a determina FND, FNC, este ca din formulă să obținem tabelul de valori și apoi să procedăm ca atunci când funcția este dată prin tabel.

Algebra booleană B_2

Exemplu: Determinați FND, FNC pentru $f : B_2^3 \rightarrow B_2$,
 $f(x, y, z) = \overline{xy} \oplus x + z$:

FND:

$$f(x, y, z) =$$

(aducem f la forma unei sume de produse)

$$\overline{\overline{xy}} \overline{x} + \overline{\overline{xy}} x + z = (\overline{x} + \overline{y})\overline{x} + xyx + z = \overline{\overline{x} + xy} + z = \overline{\overline{x} + y} + z = \\ \overline{\overline{x}} \overline{y} + z = x \overline{y} + z =$$

(înmulțim cu $(t + \bar{t})$ pentru fiecare variabilă t lipsă)

$$x\bar{y}(z + \bar{z}) + (x + \bar{x})(y + \bar{y})z =$$

(desfacem parantezele și rescriem ca sumă de produse)

$$x\bar{y}z + x\bar{y}\bar{z} + xyz + x\bar{y}z + \bar{x}yz + \bar{x}\bar{y}z =$$

(eliminăm produsele dupicate)

$$x\bar{y}z + x\bar{y}\bar{z} + xyz + \bar{x}yz + \bar{x}\bar{y}z$$

Algebra booleană B_2

FNC:

$$f(x, y, z) = x \bar{y} + z = \text{(am calculat deja)}$$

(aducem f la forma unui produs de sume)

$$(x + z)(\bar{y} + z) =$$

(adunăm cu $t \bar{t}$ pentru fiecare variabilă t lipsă)

$$(x + y \bar{y} + z)(x \bar{x} + \bar{y} + z) =$$

(rescriem ca produs de sume)

$$(x + y + z)(x + \bar{y} + z)(x + \bar{y} + z)(\bar{x} + \bar{y} + z) =$$

(eliminăm sumele dupăcat)

$$(x + y + z)(x + \bar{y} + z)(\bar{x} + \bar{y} + z)$$

Exercițiu: obțineți FND, FNC, pentru această funcție construind tabelul și apoi folosind acest tabel (trebuie să rezulte aceeași expresii pentru FND, FNC, abstracție făcând de o permutare a factorilor/termenilor pe baza asociativității și comutativității lui $\cdot / +$).

Algebra booleană B_2

Teorema precedentă și exemplele anterioare ne oferă și o regulă după care recunoaștem dacă o formulă (expresie) este o FND sau FNC:

- O expresie algebraică booleană în variabilele x_1, \dots, x_n este o FND, dacă:
 - este o sumă de produse unice (care nu se repetă) de variabile cu/fără $\bar{}$;
 - fiecare produs conține toate variabilele x_1, \dots, x_n .

Exemple:

Expresia $x + \bar{y}z$ în variabilele x, y, z nu este o FND

(deși este o sumă de produse, nu toate produsele conțin toate variabilele x, y, z).

Expresia $x\bar{y}z + \bar{x}\bar{y}z$ în variabilele x, y, z este o FND

(produsele conțin toate variabilele x, y, z , chiar dacă nu sunt toate cele $2^3 = 8$ produse posibile în aceste variabile).

Algebra booleană B_2

- O expresie algebrică booleană în variabilele x_1, \dots, x_n este o FNC, dacă:
 - este un produs de sume unice (care nu se repetă) de variabile cu/fără \neg ;
 - fiecare sumă conține toate variabilele x_1, \dots, x_n .

Exemple:

Expresia $x(\bar{y} + z)$ în variabilele x, y, z nu este o FNC

(deși este un produs de sume, nu toate sumele conțin toate variabilele x, y, z).

Expresia $(x + \bar{y} + z)(\bar{x} + \bar{y} + z)$ în variabilele x, y, z este o FNC

(sumele conțin toate variabilele x, y, z , chiar dacă nu sunt toate cele $2^3 = 8$ sume posibile în aceste variabile).

Algebra booleană B_2

Observație:

O expresie algebrică booleană în variabilele x_1, \dots, x_n care este o sumă de produse de variabile cu/fără $\bar{}$ și care conține toate cele 2^n produse posibile este FND a funcției constante 1 (corespunde unui tabel care are 1 în toata coloana valorilor funcției).

Astfel, dacă ea apare ca factor într-un produs, se poate elimina.

De exemplu, avem:

$$xz + x\bar{z} + zxy + zx\bar{y} + z\bar{x}\bar{y} + z\bar{x}\bar{y} = x(z + \bar{z}) + z(xy + x\bar{y} + \bar{x}y + \bar{x}\bar{y}) = x \cdot 1 + z \cdot 1 = x + z$$

Similar, o expresie algebrică booleană în variabilele x_1, \dots, x_n care este un produs de sume de variabile cu/fără $\bar{}$ și care conține toate cele 2^n sume posibile este FNC a funcției constante 0 (corespunde unui tabel care are 0 în toata coloana valorilor funcției).

Astfel, dacă ea apare ca termen într-o sumă, se poate elimina.

Algebra booleană B_2

Reamintim, cu noile notații, și teorema care dă o descriere recursivă a funcțiilor booleene scalare, prin funcții care au cu o variabilă mai puțin:

Th: Dacă $f : B_2^n \rightarrow B_2$, $n \geq 1$, este o funcție booleană scalară și $1 \leq i \leq n$, atunci:

$$\forall x_1, \dots, x_n \in B_2,$$

$$f(x_1, \dots, x_n) = x_i f(x_1, \dots, x_{i-1}, 1, x_{i+1}, \dots, x_n) + \bar{x}_i f(x_1, \dots, x_{i-1}, 0, x_{i+1}, \dots, x_n)$$

Algebra booleană B_2

Dacă notăm variabilele unei funcții booleene în ordinea descrescătoare a semnificației cu x_{n-1}, \dots, x_0 , $n \geq 1$, și folosim teorema anterioară pentru $i = 0$:

$$f(x_{n-1}, \dots, x_0) = f(x_{n-1}, \dots, x_1, 0)\bar{x}_0 + f(x_{n-1}, \dots, x_1, 1)x_0$$

putem descrie o funcție booleană $f = (f_1, \dots, f_p) : B_2^n \longrightarrow B_2^p$, $n, p \geq 1$, printr-un tabel mai compact, având doar 2^{n-1} linii, însă valorile din tabel nu vor mai fi doar 0 și 1, ci și \bar{x}_0 sau x_0 .

În acest scop, scriem valorile componentelor funcției pentru valori succesive ale variabilei celei mai puțin semnificative x_0 pe aceeași linie (conform regulilor de organizare a tabelului de valori descrise mai devreme, ele apar pe linii succesive în tabelul extins) iar în niște coloane separate (care vor fi coloanele rezultat) vom exprima unitar aceste valori ca funcții de x_0 , folosind formula de recurență de mai sus.

Algebra booleană B_2

Mai exact, fiecare pereche de linii succesive:

x_{n-1}	\dots	x_1	x_0	f_1	\dots	f_p
a_{n-1}	\dots	a_1	0	b_1	\dots	b_p
a_{n-1}	\dots	a_1	1	c_1	\dots	c_p
\vdots				\vdots		

se înlocuiește cu:

x_{n-1}	\dots	x_1	$\overbrace{0 \quad 1}^{x_0}$	f_1	\dots	f_p
a_{n-1}	\dots	a_1	$b_1 \dots b_p$	$c_1 \dots c_p$	d_1	\dots
\vdots			\vdots	\vdots	\vdots	

unde pentru orice $1 \leq i \leq p$, avem $d_i = b_i \bar{x}_0 + c_i x_0$, adică avem corespondența:

b_i	c_i	d_i
0	0	0
0	1	x_0
1	0	\bar{x}_0
1	1	1

Algebra booleană B_2

Exemplu: Pentru funcția booleană $f : B_2^3 \rightarrow B_2^2$ dată prin tabelul extins:

x	y	z	$f1$	$f2$
0	0	0	0	0
0	0	1	0	0
0	1	0	1	1
0	1	1	1	0
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	0

vom avea tabelul compact:

x	y	z		$f1$	$f2$
x	y	0	1	$f1$	$f2$
0	0	0 0	0 0	0	0
0	1	1 1	1 0	1	\bar{z}
1	0	1 0	0 1	\bar{z}	z
1	1	0 1	1 0	z	\bar{z}

Acest tabel ne arată, de exemplu, că pentru orice $z \in \{0, 1\}$ avem:

$$f_1(0, 0, z) = 0, \quad f_2(0, 1, z) = \bar{z}$$

Algebra booleană B_2

Din teorema referitoare la FND și formula de recurență pentru funcții booleane scalare în cazul general, deducem o regulă după care putem obține o scriere a funcției ca sumă de produse (disjuncție de conjuncții), pornind de la tabelul compact:

Dacă $f : B_2^n \rightarrow B_2$ este o funcție booleană scalară, $n \geq 1$, atunci pentru orice $x_{n-1}, \dots, x_0 \in B_2$ avem:

$$\begin{aligned} f(x_{n-1}, \dots, x_0) &= \\ \sum_{\alpha_{n-1}, \dots, \alpha_0 \in B_2} f(\alpha_{n-1}, \dots, \alpha_0) x_{n-1}^{\alpha_{n-1}} \cdots x_0^{\alpha_0} &= \\ \sum_{\alpha_{n-1}, \dots, \alpha_1 \in B_2} x_{n-1}^{\alpha_{n-1}} \cdots x_1^{\alpha_1} (f(\alpha_{n-1}, \dots, \alpha_1, 0) \bar{x}_0 + f(\alpha_{n-1}, \dots, \alpha_1, 1) x_0) &= \\ \sum_{\alpha_{n-1}, \dots, \alpha_1 \in B_2} x_{n-1}^{\alpha_{n-1}} \cdots x_1^{\alpha_1} E(\alpha_{n-1}, \dots, \alpha_1) \end{aligned}$$

unde $E(\alpha_{n-1}, \dots, \alpha_1) = f(\alpha_{n-1}, \dots, \alpha_1, 0) \bar{x}_0 + f(\alpha_{n-1}, \dots, \alpha_1, 1) x_0$ este valoarea care apare în coloana rezultat din tabelul compact al lui f , pe linia corespunzătoare valorilor $\alpha_{n-1}, \dots, \alpha_1$ ale variabilelor x_{n-1}, \dots, x_1 ; întrucât f ia valori în B_2 , $E(\alpha_{n-1}, \dots, \alpha_1)$ poate fi doar 0, 1, x_0 sau \bar{x}_0 .

Algebra booleană B_2

Rezultă următoarea regulă de a obține scrierea funcției ca sumă de produse pornind de la tabelul compact (dacă funcția este vectorială, regula se aplică pentru fiecare componentă îndeapta):

- Considerăm liniile tabelului corespunzătoare valorii $E(\alpha_{n-1}, \dots, \alpha_1) \neq 0$.
 - Pentru fiecare linie de forma $\alpha_{n-1}, \dots, \alpha_1, E$ (unde E poate fi 1, x_0 sau \bar{x}_0), considerăm produsul (conjuncția) $x_{n-1}^{\alpha_{n-1}} \cdots x_1^{\alpha_1} E$.
- Deci, pentru orice $n - 1 \geq i \geq 1$, x_i apare negat dacă $\alpha_i = 0$ și nenegat dacă $\alpha_i = 1$, ca la FND, iar E se copiază ca atare din tabel; dacă $E = 1$, se poate omite.
- Considerăm suma (disjuncția) produselor.

Exemplu: Pentru funcția $f : B_2^3 \longrightarrow B_2^2$ din exemplul precedent, avem:

$$f_1(x, y, z) = \bar{x}y + x\bar{y}\bar{z} + xyz$$

$$f_2(x, y, z) = \bar{x}y\bar{z} + x\bar{y}z + xy\bar{z}$$

Aceste scrieri nu sunt neapărat FND, de exemplu termenul $\bar{x}y$ nu conține toate variabilele.

Algebra booleană B_2

Scrierea ca sumă de produse se poate simplifica în continuare, folosind proprietatea menționată mai devreme, că suma tuturor produselor cu/fără formate cu niște variabile date este egală cu 1.

Mai exact, dacă în notațiile precedente $E(\alpha_{n-1}, \dots, \alpha_1)$ are o aceeași valoare y pentru niște $\alpha_{i_k}, \dots, \alpha_{i_1}$ fixate ($n - 1 \geq i_k > \dots > i_1 \geq 1, k \geq 1$) și toate valorile posibile 0 sau 1 ale celorlalte α_i ($n - 1 \geq i \geq 1$), atunci în scrierea:

$$f(x_{n-1}, \dots, x_0) = \sum_{\alpha_{n-1}, \dots, \alpha_1 \in B_2} x_{n-1}^{\alpha_{n-1}} \cdots x_1^{\alpha_1} E(\alpha_{n-1}, \dots, \alpha_1)$$

putem da factor comun și înlocui:

$$\sum_{\substack{\alpha_i \in B_2 \\ i \neq i_k, \dots, i_1}} x_{n-1}^{\alpha_{n-1}} \cdots x_{i_k+1}^{\alpha_{i_k+1}} x_{i_k}^{\alpha_{i_k}} x_{i_k-1}^{\alpha_{i_k-1}} \cdots x_{i_1+1}^{\alpha_{i_1+1}} x_{i_1}^{\alpha_{i_1}} x_{i_1-1}^{\alpha_{i_1-1}} \cdots x_1^{\alpha_1} y =$$

$$\left(\sum_{\substack{\alpha_i \in B_2 \\ i \neq i_k, \dots, i_1}} x_{n-1}^{\alpha_{n-1}} \cdots x_{i_k+1}^{\alpha_{i_k+1}} x_{i_k-1}^{\alpha_{i_k-1}} \cdots x_{i_1+1}^{\alpha_{i_1+1}} x_{i_1-1}^{\alpha_{i_1-1}} \cdots x_1^{\alpha_1} \right) x_{i_k}^{\alpha_{i_k}} \cdots x_{i_1}^{\alpha_{i_1}} y =$$

$$1 \cdot x_{i_k}^{\alpha_{i_k}} \cdots x_{i_1}^{\alpha_{i_1}} y = x_{i_k}^{\alpha_{i_k}} \cdots x_{i_1}^{\alpha_{i_1}} y$$

Dacă $E(\alpha_{n-1}, \dots, \alpha_1)$ are o aceeași valoare y pentru toate valorile posibile 0 sau 1 ale tuturor α_i ($n - 1 \geq i \geq 1$), atunci:

$$f(x_{n-1}, \dots, x_0) = \sum_{\alpha_{n-1}, \dots, \alpha_1 \in B_2} x_{n-1}^{\alpha_{n-1}} \cdots x_1^{\alpha_1} y = y$$

Algebra booleană B_2

Exemplu: Pentru funcția $f : B_2^3 \longrightarrow B_2^2$ din exemplele precedente, avem:

$$f_2(x, y, z) = y\bar{z} + x\bar{y}z$$

deoarece avem aceeași valoare $f_2(= E) = \bar{z}$ pentru $y = 1$ și toate valorile posibile 0 sau 1 ale lui x .

Aceasta corespunde următorului calcul prin care simplificăm formula obținută anterior:

$$f_2(x, y, z) = \bar{x}y\bar{z} + x\bar{y}z + xy\bar{z} = (\bar{x} + x)y\bar{z} + x\bar{y}z = 1 \cdot y\bar{z} + x\bar{y}z = y\bar{z} + x\bar{y}z$$

Algebra booleană B_2

Așadar, dacă în tabelul compact un grup de linii conține aceleași valori într-o coloană rezultat (a unei componente a funcției) și în niște coloane ale variabilelor, iar în celelalte coloane ale variabilelor sunt parcuse toate combinațiile posibile de 0 sau 1, acel grup de linii furnizează un singur termen în scrierea ca sumă de produse a componentei respective a funcției, iar acest termen conține doar variabilele cu valoare comună (cu / fără $\bar{}$) și valoarea comună rezultat:

x_{n-1}	\dots	x_{i_k}	\dots	x_{i_1}	\dots	x_1	\dots	f_j	\dots
				\vdots				\vdots	
(*) {	\dots	α_{i_k}	\dots	α_{i_1}	\dots			y	
		\vdots						\vdots	
	\dots	α_{i_k}	\dots	α_{i_1}	\dots			y	
				\vdots				\vdots	

Grupul de 2^{n-k} linii (*) de mai sus nu sunt neapărat consecutive în tabel și conțin aceleași valori în coloanele variabilelor x_{i_k}, \dots, x_{i_1} și în coloana rezultat f_j , iar în coloanele celorlalte $n - k$ variabile sunt parcuse toate cele 2^{n-k} combinații posibile de 0 sau 1; acest grup de linii furnizează un singur termen, $x_{i_k}^{\alpha_{i_k}} \cdots x_{i_1}^{\alpha_{i_1}} y$, în scrierea ca sumă de produse a componentei f_j .

Algebra booleană B_2

Exemplu: Pentru funcția booleană $f : B_2^3 \rightarrow B_2^4$, în variabilele x, y, z , dată prin tabelul compact (am numerotat liniile):

	x	y	f_1	f_2	f_3	f_4
0	0	0	\bar{z}	0	1	\bar{z}
1	0	1	\bar{z}	z	z	\bar{z}
2	1	0	0	z	\bar{z}	\bar{z}
3	1	1	z	1	z	\bar{z}

avem:

$$f_1(x, y, z) = \bar{x} \bar{z} + xyz$$

(liniile 0, 1 au furnizat un singur termen, $\bar{x} \bar{z}$, deoarece în coloana rezultat f_1 și în coloana variabilei x avem aceleași valori, iar în coloana variabilei rămase y sunt parcuse toate cele 2 combinații posibile de 0 sau 1)

$$f_2(x, y, z) = \bar{x}yz + x\bar{y}z + xy$$

(liniile 1, 2 nu pot furniza un singur termen, căci deși în coloana rezultat f_2 avem aceeași valoare z , în coloanele variabilelor rămase x, y nu sunt parcuse toate cele 4 combinații posibile de 0 sau 1)

Algebra booleană B_2

$$f_3(x, y, z) = \bar{x} \bar{y} + yz + x\bar{y} \bar{z}$$

(liniile 1, 3 au furnizat un singur termen)

$$f_4(x, y, z) = \bar{z}$$

(toate cele 4 linii au furnizat un singur termen, \bar{z} , deoarece în coloana rezultat f_4 avem aceeași valoare \bar{z} , iar în coloanele variabilelor rămase x, y sunt parcuse toate cele 4 combinații posibile de 0 sau 1).

Observăm că în cazul unui tabel compact cu 4 linii numerotate ca mai sus, formula unei componente se simplifică dacă valoarea ei se repetă în liniile:

0 și 1 (dispare variabila a doua),

2 și 3 (dispare variabila a doua),

0 și 2 (dispare prima variabilă),

1 și 3 (dispare prima variabilă),

0, 1, 2, 3 (dispar ambele variabile);

nu se simplifică dacă valoarea se repetă doar în liniile 0 și 3 sau 1 și 2.

Cuprins

1 Performanța calculatoarelor

Conceptul de performanță
Măsurarea performanței

2 Aritmetică sistemelor de calcul

Reprezentarea numerelor în matematică
Reprezentarea numerelor în calculator
Reprezentarea numerelor naturale ca întregi fără semn
Reprezentarea numerelor întregi în complement față de 2
Reprezentarea numerelor reale în virgulă mobilă

3 Logica pentru calculatoare

Algebre booleene
Funcții booleene
Particularizare la cazul B_2

4 Circuite logice

Circuite logice, Sisteme digitale
0-DS (Circuite combinaționale, Funcții booleene)
1-DS (Memorii)
2-DS (Automate finite)
Algoritmi de înmulțire și împărțire hardware
3-DS (Procesoare) și 4-DS (Calculatoare)
 n -DS, $n > 4$

Cuprins

1 Performanța calculatoarelor

Conceptul de performanță
Măsurarea performanței

2 Aritmetică sistemelor de calcul

Reprezentarea numerelor în matematică
Reprezentarea numerelor în calculator
Reprezentarea numerelor naturale ca întregi fără semn
Reprezentarea numerelor întregi în complement față de 2
Reprezentarea numerelor reale în virgulă mobilă

3 Logica pentru calculatoare

Algebre booleene
Funcții booleene
Particularizare la cazul B_2

4 Circuite logice

Circuite logice, Sisteme digitale

0-DS (Circuite combinaționale, Funcții booleene)
1-DS (Memorii)
2-DS (Automate finite)
Algoritmi de înmulțire și împărțire hardware
3-DS (Procesoare) și 4-DS (Calculatoare)
 n -DS, $n > 4$

Circuite logice

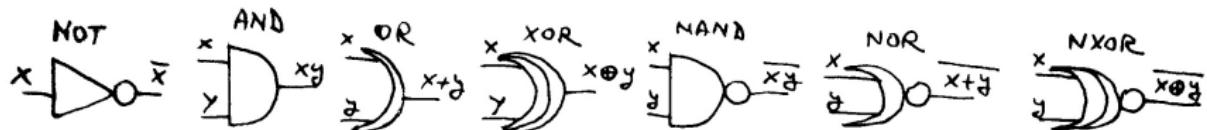
Circuitele logice sunt dispozitive tehnice care permit efectuarea automată a calculelor logice.

Ele sunt construite pornind de la niște **circuite elementare (porți)**, care implementează operațiile booleene: $+$, \cdot , \neg , etc. și efectuând de un număr finit de ori diverse operații de combinare (legare): **legare în serie**, **legare în paralel**, **închiderea prin ciclu**, etc., obținându-se astfel circuite tot mai complexe.

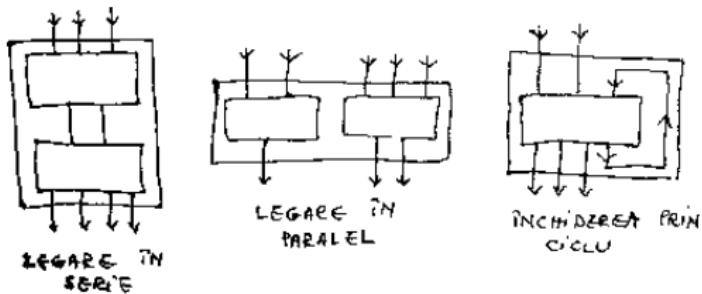
Datele de intrare și ieșire ale acestor circuite sunt valorile de adevăr 0/1 (fals/adevărat).

Circuite logice

Porțile sunt:



Operațiile de combinare sunt:

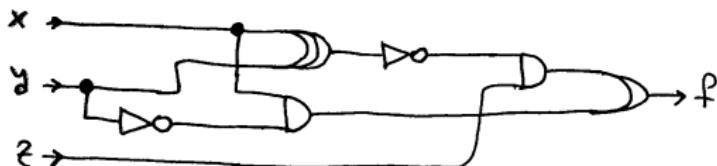


Desenarea săgeților " $>$ " pe liniile de intrare sau ieșire nu este obligatorie, dacă este evident sensul de circulație al datelor; altfel, pe o linie se pot pune oricâte săgeți.

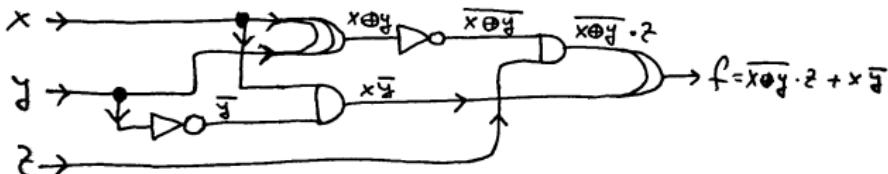
Circuite logice

În general, circuitul care implementează o formulă booleană se poate construi combinând porțile în aceeași ordine în care se compun operațiile booleene implementate de ele pentru a se obține formula.

De exemplu, formula $f(x, y, z) = \overline{x \oplus y}z + x\bar{y}$ poate fi implementată prin circuitul:

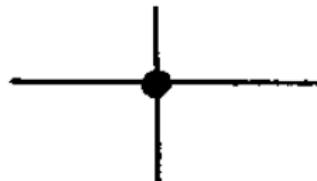


Putem spori claritatea desenului dacă pe anumite linii (nu neapărat pe toate) adăugam săgeți ">" și/sau notăm în dreptul lor formula de calcul a valorii emise pe acolo, de exemplu:



Circuite logice

În reprezentările grafice ale circuitelor, avem nevoie să distingem între o intersecție de linii fără contact și una cu contact. În figurile anterioare am folosit următoarea simbolizare, pe care o vom utiliza și în continuare:



INTERSECȚIE CU
CONTACT

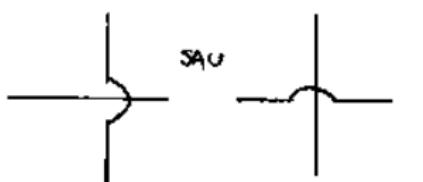


INTERSECȚIE
FĂRĂ CONTACT

O altă variantă de simbolizare, dar pe care nu o vom folosi, este:



INTERSECȚIE
CU CONTACT



INTERSECȚIE FĂRĂ CONTACT

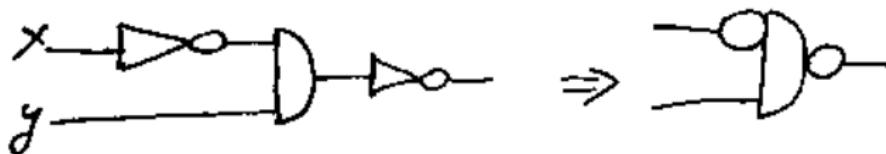
Circuite logice

Intrările se pot nota în dreptul unor scurte linii de intrare sau direct lângă poarta în care intră; unele intrări sunt fixate pe 0 sau 1, iar acestea se vor nota ca atare; de exemplu:

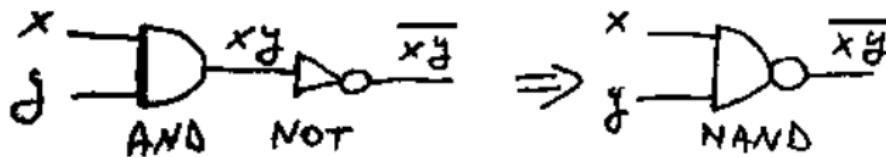


Circuite logice

O simplificare folosită uneori în reprezentarea grafică a circuitelor logice este următoarea: dacă o poartă "NOT" este legată în serie cu o poartă "OR", "AND", "XOR", nu se mai desenează triunghiul iar cerculețul este lipit de poarta respectivă; de exemplu:



Această simplificare stă la baza simbolurilor folosite pentru porțile "NAND", "NOR", "NXOR"; de exemplu:



Circuite logice

Constatăm că pentru implementarea diverselor formule de calcul logic nu sunt necesare toate porțile considerate; de exemplu, sunt suficiente porțile "NOT", "AND", "OR", restul porților putându-se exprima în funcție de acestea:

$$\text{NAND} \Leftrightarrow \text{NOT} \quad \text{NOR} \Leftrightarrow \text{NOT}$$

$$\text{XOR} \Leftrightarrow \text{NOT}(x \oplus y) = x\bar{y} + \bar{x}y$$

$$\text{XNOR} \Leftrightarrow \text{NOT}(x \oplus y) = x\bar{y} + \bar{x}y$$

Circuite logice

Două justificări pentru luarea în considerare a mai multor porți sunt următoarele:

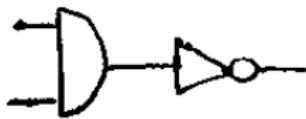
- Anumite porțiuni de circuit (**blocuri**) apar foarte frecvent în structura diverselor circuite; atunci, în loc să le desenăm detaliat de fiecare dată, le asociem un simbol și desenăm simbolul. Astfel, circuitele sunt mai ușor de desenat și de înțeles.

Situația seamănă cu cea din programare când un anumit fragment de cod se repetă de multe ori, eventual cu alte variabile/valori, și în loc să-l rescriem de fiecare dată, preferăm să-l încorporăm într-o procedură, eventual cu parametri, și să apelăm procedura.

Circuite logice

- Realzarea unui circuit logic prin combinări de porți poate fi asemănătă cu realizarea unui circuit electronic prin montarea unor componente electronice (tranzistori, rezistori, condensatori, diode, etc.) pe o placă cu contacte (cablaj imprimat).

Procesarea efectuată de o componentă electronică se bazează pe fenomene fizice foarte rapide care au loc în interiorul componente - de exemplu interacțiunea dintre mai multe straturi de semiconductori - le vom numi fenomene de tip (1). Comunicarea între componente se bazează pe fenomene de circulație a curentului electric prin liniile plăcii de contacte - le vom numi fenomene de tip (2). De exemplu, în circuitul:



procesarea presupune un fenomen de tip (1) desfășurat în interiorul lui "AND", apoi un fenomen de tip (2) pentru comunicarea cu "NOT", apoi un fenomen de tip (1) în interiorul lui "NOT".

Circuite logice

Dacă acest circuit apare frecvent ca bloc în alte circuite, preferăm să încorporăm toată procesarea sa, printr-un fenomen de tip (1), în interiorul unei singure componente (poartă) de un tip nou, simbolizate:



Astfel, toate circuitele care vor conține noua poartă în locul blocului anterior vor funcționa mai repede.

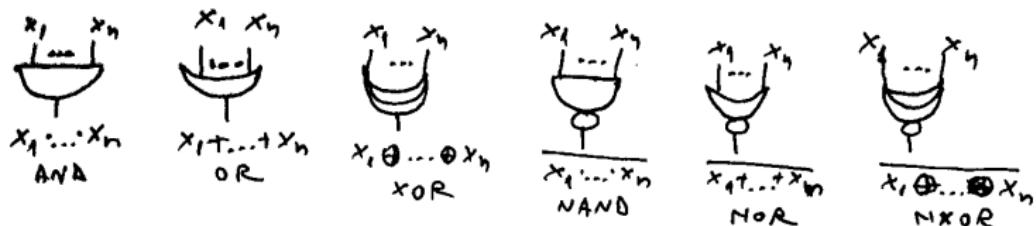
Această abordare este întâlnită în cazul multor circuite electronice care apar frecvent ca parte a altor circuite: în loc să se construiască circuitul de fiecare dată din mai multe componente simple montate pe placă cu contacte, se construiește ca un circuit integrat (chip) care se montează ca o singură componentă pe placă respectivă. Utilizarea frecventă a acestuia justifica fabricarea sa în serie ca un nou tip de componentă electronică.

Circuitele care conțin asemenea chip-uri sunt mai mici ca gabarit, mai ieftine și mai rapide decât cele realizate din componente simple și multe.

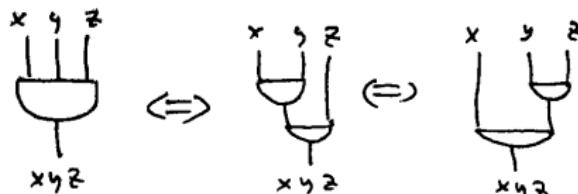
Circuite logice

În cele ce urmează vor exista și alte circuite, mai complexe, care sunt des întâlnite ca blocuri în alte circuite și de aceea vor avea un simbol propriu, care va fi folosit în locul circuitului detaliat: decodificator, multiplexor, sumator, etc.. Din punct de vedere tehnic, ele se pot realiza ca tipuri distincte de componenete electronice, alături de porti.

Printre acestea, sunt circuitele "AND", "OR", "XOR", "NAND", "NOR", "NXOR" cu mai multe intrări:



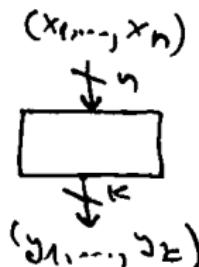
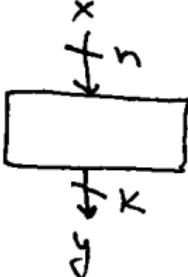
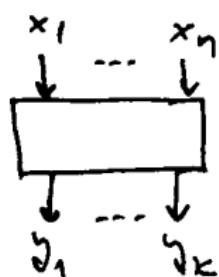
Realizarea lor se bazează pe asociativitatea și comutativitatea operațiilor \cdot , $+$, \oplus . De exemplu:



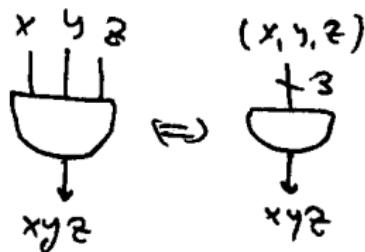
Circuite logice

Uneori, la desenarea circuitelor logice cu multe linii se folosește următoarea simbolizare mai simplă:

Dacă $x = (x_1, \dots, x_n)$, $y = (y_1, \dots, y_k)$



De exemplu:



Circuite logice

Subliniem că termenul de "circuit logic" se referă la o anumită logică de organizare și funcționare și un anumit scop la care este folosit circuitul, nu și la mijloacele tehnice prin care este el construit.

Cel mai bine, ne imaginăm circuitele logice ca fiind niște niște circuite conceptuale, prin care circulă valori de adevăr; ele descriu în mod abstract un calcul logic.

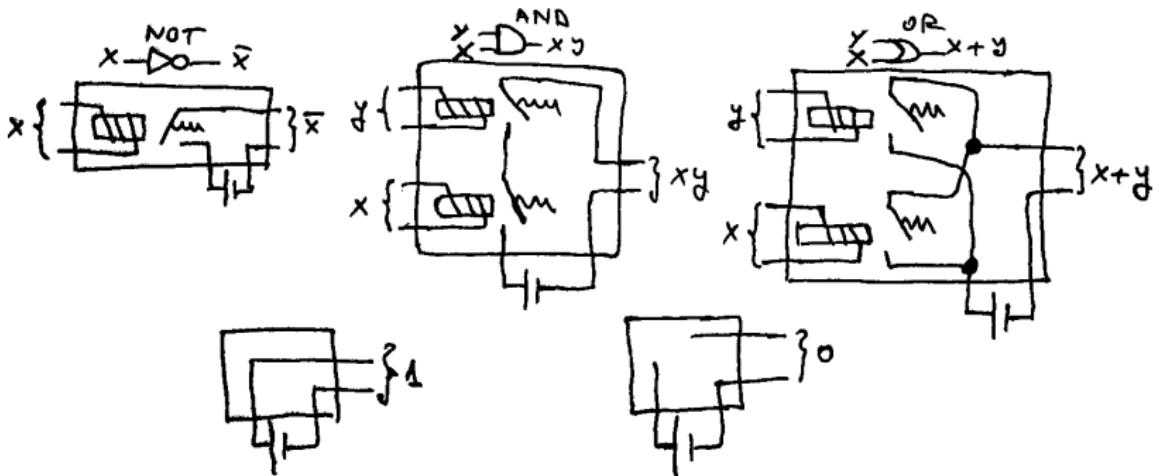
Circuitele logice se pot realiza prin diverse mijloace tehnice: circuite electronice, relee și contacte, angrenaje cu roți dințate, sisteme de pârghii, frânghii și scripeți, conducte de apă și robinete, etc.

Nu trebuie să confundăm însă circuitul logic cu un anumit mod de realizare tehnică a sa.

Circuite logice

Exemplu: Realizare tehnică cu relee și contacte:

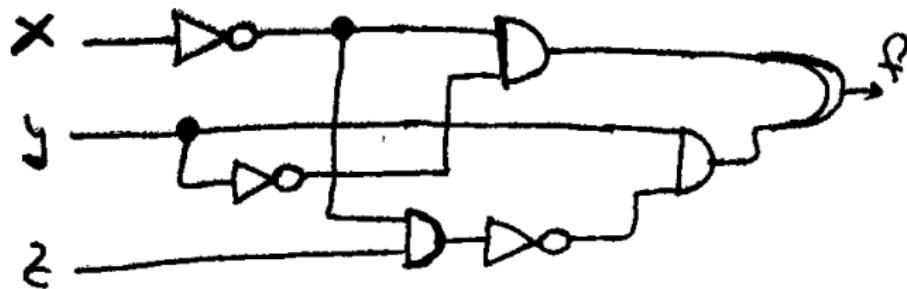
Realizarea porților "NOT", "AND", "OR" și a intrărilelor constante 0 și 1:



Deci, liniile circuitului sunt perechi de linii electrice, 0 = fără tensiune, 1 = sub tensiune. Fiecare poartă poate avea propria sursă de curent, sau toate porțile pot fi conectate la o aceeași sursă și nu contează respectarea unei polarități $+/-$.

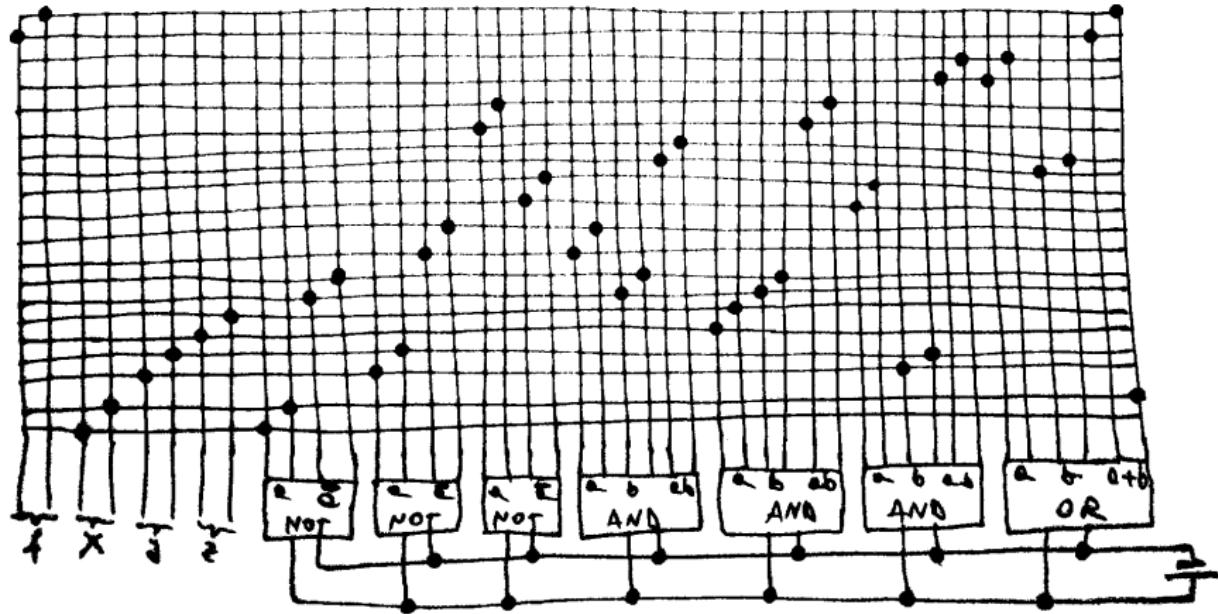
Circuite logice

Circuitul logic care implementează formula $f(x, y, z) = \bar{x} \bar{y} + y \bar{x} z$ este:



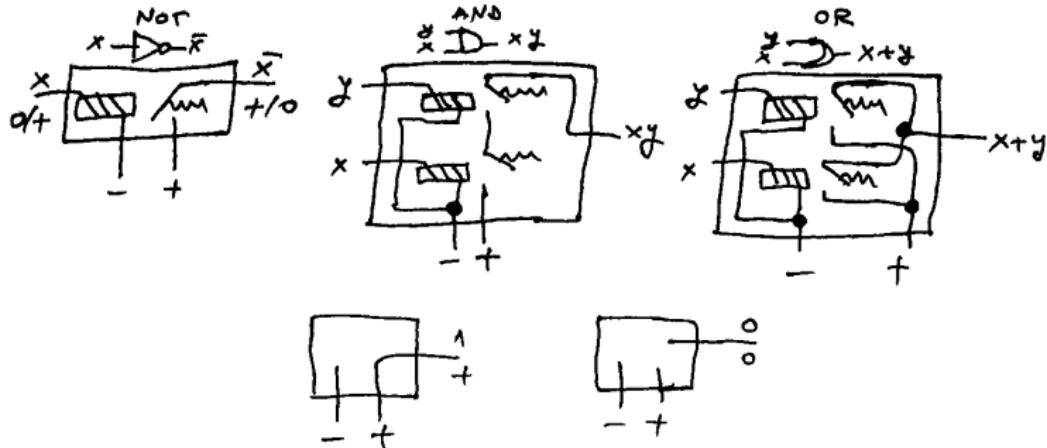
Circuite logice

Realizarea tehnică a circuitului este:



Circuite logice

O variantă de realizare cu relee și contacte a porților "NOT", "AND", "OR" și a intrărilor constante 0 și 1, care necesită doar o linie electrică pentru o linie logică, este:

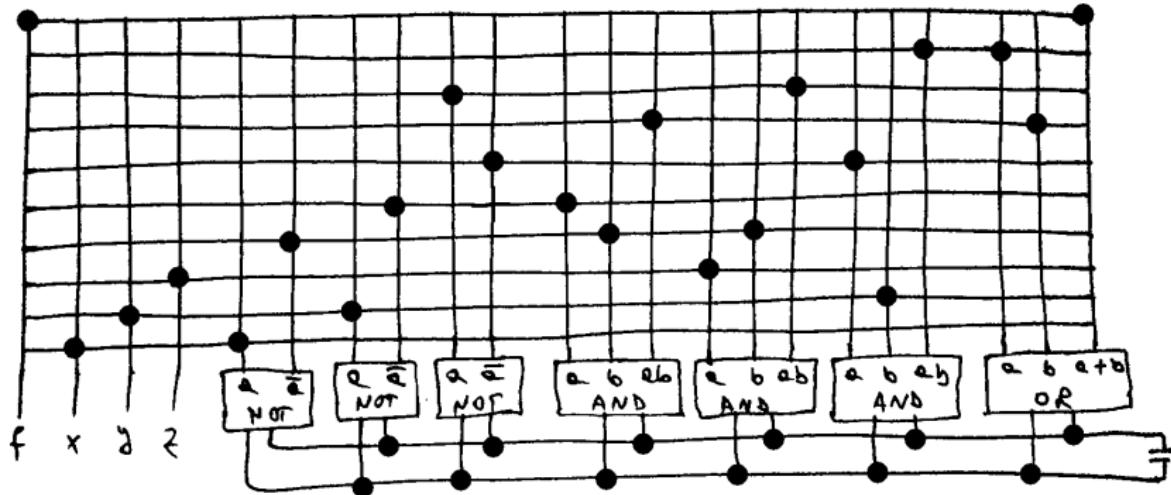


Așadar, 0 = fără tensiune, 1 = tensiune +.

Acum însă toate porțile trebuie conectate la o aceeași sursă de curent și trebuie respectată polaritatea +/-.

Circuite logice

Realizarea tehnică a circuitului pentru formula $f(x, y, z) = \bar{x} \bar{y} + y \bar{x}z$ este:



Circuite logice

TODO: Realizarea tehnică a porților prin mijloace mecanice (roți dințate, pârghii, etc.)

TODO: Realizarea tehnică a porților prin mijloace electronice (componente electronice simple: tranzistori, etc., pe o placă cu contacte).

A se vedea:

<http://www.electronics-tutorials.ws/category/logic>

Circuite logice

De obicei, circuitele logice sunt realizate tehnic prin mijloace electronice (circuite electronice cu chip-uri), deoarece oferă gabarit și cost redus și viteză de funcționare mare; de aceea simbolistica și terminologia sunt preluate din electronică.

Când realizăm circuitele logice prin mijloace electronice putem modela în diverse moduri valorile 0/1; de exemplu:

- 0 = nu trece curentul, 1 = trece curentul;
- 0 = trece curentul, 1 = nu trece curentul;
- și la 0 și la 1 trece curentul, dar are altă tensiune, modulară (transportă un alt tip de semnal).

De obicei este folosită ultima variantă.

Circuite logice

Mai exact:

- Circuitele electronice din interiorul calculatorului modern sunt **circuite digitale**.

Electronica digitală operează cu doar două niveluri de tensiune electrică importante: o **tensiune înaltă (nivel înalt)** și o **tensiune joasă (nivel jos)**.

Celelalte valori de tensiune sunt temporare și apar în timpul tranziției între cele două valori (o deficiență în proiectarea digitală poate fi prelevarea unui semnal care nu este în mod clar nici înalt nici jos).

- În diverse categorii de dispozitive logice, valorile și relațiile dintre cele două valori de tensiune diferă.

Astfel, în loc să se facă referire la valorile de tensiune, se discută despre semnale care sunt (logic) **adevărate**, sau 1, sau **activate (asserted)** și despre semnale care sunt (logic) **false**, sau 0, sau **dezactivate (deasserted)**.

Valorile 0 și 1 sunt **complemente** sau **inverse** una celeilalte.

Circuite logice

Un circuit (bloc) este **activ**, dacă ieșirea sa este 1, și **inactiv**, dacă ieșirea sa este 0.

O variabilă logică, asociată unei intrări, poate controla un circuit (bloc) **activându-l**, dacă ia valoarea 1, sau **dezactivându-l**, dacă ia valoarea 0.

Mai general, se vorbește despre **activare** sau circuite (blocuri) **active la nivel înalt**, respectiv **la nivel jos**, după cum valoarea de intrare relevantă (care declanșază funcționalitatea cea mai importantă a circuitului) este 1, respectiv 0.

Circuite logice

În orice moment al funcționării unui circuit logic, orice linie a sa transportă ceva: 0, 1 sau un semnal neclar, care poate fi interpretat la destinație în mod eronat ca 0 sau 1.

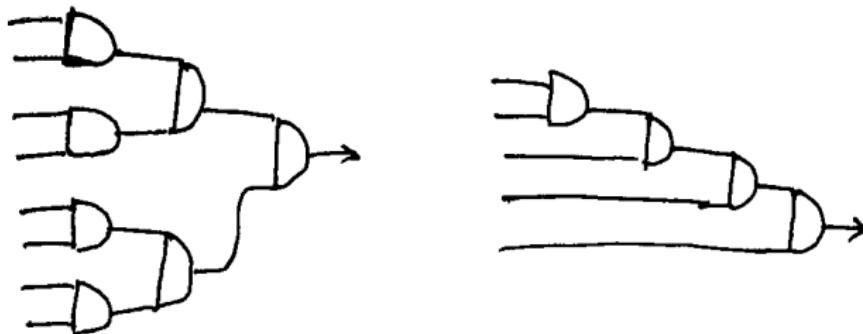
De aceea, uneori când vom analiza circuitul d.p.v. logic vom considera că liniile sale sunt fie în situația "transportă 0", fie în situația "transportă 1", aceste valori putând fi și eronate.

Circuite logice

Dacă schimbăm valorile pe liniile de intrare, vom avea alte valori pe liniile de ieșire, rezultate în urma procesării (calculului) efectuate de circuit. Valorile rezultat nu apar și nu rămân stabile însă imediat ci după un anumit interval de timp, necesitat de structura circuitului și natura fenomenelor fizice folosite; până atunci, valorile pe liniile de ieșire pot fluctua și, în orice caz, nu sunt relevante.

Acest interval de timp este cu atât mai lung cu cât circuitul este mai dezvoltat pe verticală (are mai multe niveluri de legare în serie).

De exemplu, în figura de mai jos, circuitul din stânga este mai rapid decât cel din dreapta, deși are mai multe porți:



Circuite logice

Pe lângă porțile "NOT", "AND", "OR", "XOR", "NAND", "NOR", "NXOR", prezentate mai devreme, uneori mai sunt considerate și alte porți, a căror utilitate este legată de mijloacele tehnice prin care sunt construite circuitele (fenomenele fizice folosite); de exemplu, în electronica digitală se mai folosesc:

- **Bufferul:**

Simbol și tabelă de valori:



IN	OUT
0	0
1	1

Așadar, bufferul transmite exact valoarea de la intrare la ieșire, cu o mică întârziere cauzată de procesarea sa internă.

Deși nu efectuează un calcul semnificativ (implementează formula $f(x) = x$), bufferul are mai multe utilități:

- Permite izolarea altor porțiuni de circuit unele de altele, împiedicând impedanța unui circuit să afecteze impedanța altuia.

În electronică, **impedanța (impedance)** este o mărime fizică care generează rezistență electrică (distincția între cele două se manifestă în cazul curentului alternativ), se notează cu Z și se măsoară în **ohmi (Ω)**.

Circuite logice

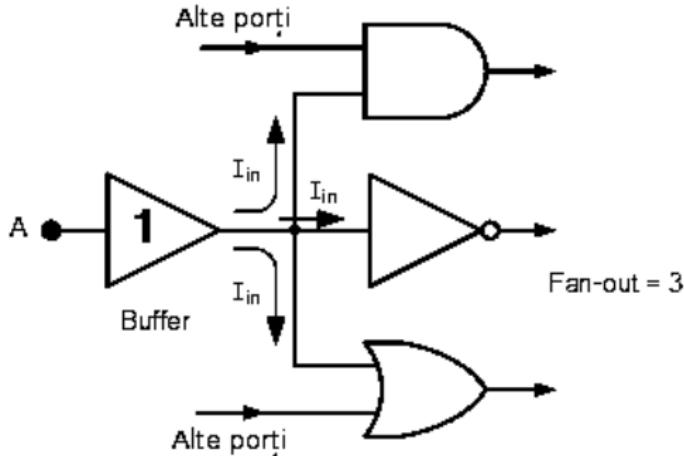
- Permite dirijarea unor încărcări mari de curent, cum ar fi cele folosite de comutatoarele cu tranzistori, sau comandarea unui LED, deoarece poate furniza la ieșire curenți mult mai mari decât necesită ca semnal de intrare.

Cu alte cuvinte, bufferul poate fi folosit pentru amplificarea puterii semnalului digital, având o **capacitate fan-out (fan-out capability)** ridicată. Parametrul fan-out al unei porți sau bloc de circuit descrie capacitatea acestuia de a furniza la ieșire curenți mari, oferind o amplificare mai mare semnalului de intrare.

Această proprietate ne permite și să legăm ieșirea unui bloc la intrarea mai multor alte blocuri.

Circuite logice

Într-adevăr, pentru a funcționa corect, fiecare intrare consumă o anumită cantitate de curent din ieșirea respectivă, a.î. distribuirea ieșirii la mai multe blocuri crește încărcarea blocului sursă. Atunci, inserarea unui buffer între blocul sursă și blocurile destinație poate rezolva problema:



Parametrul "fan-out" este numărul de încărcări paralele care pot fi dirijate simultan de către o singură poartă. Acționând ca o sursă de curent, un buffer poate avea un rating "fan-out" înalt de până la 20 porți din aceeași familie logică.

Circuite logice

Dacă o poartă are un rating "fan-out" înalt (sursă de curent), ea trebuie să aibă de asemenea un rating fan-in înalt (consumator de curent). Totuși, întârzierea cauzată de procesarea internă a portii (propagation delay) se deteriorează rapid ca funcție de "fan-in", astfel că portile cu "fan-in" mai mare decât 4 trebuie evitate.

Notăm că proprietăți asemănătoare bufferului, cum ar fi efectul de amplificare a semnalului, îl au și alte porți, ca de exemplu "NOT":  - de aceea, poarta "NOT" se mai numește și **buffer inversor**, sau doar **inversor (inverter)**.

Circuite logice

- **Bufferul cu 3 stări (Tri-state Buffer):** Este un tip de buffer a cărui ieșire poate fi, la cerere, deconectată "electronic" de la circuitele la care este legată. Mai exact, el poate genera la ieșire un semnal care nu este logic nici 0, nici 1, iar care d.p.v. funcțional se comportă ca și când linia de ieșire ar fi deconectată de la intrare (se produce o condiție de circuit deschis); d.p.v. tehnic, poarta se va comporta ca o componentă electronică cu impedanță foarte mare, sau ca un contact electric întrerupt (ca rezultat, nu se consumă curent de la sursă); de aceea, această valoare de ieșire s.n. **impedanță înaltă (Hi-Z)**.

Simbol și tabel de valori:



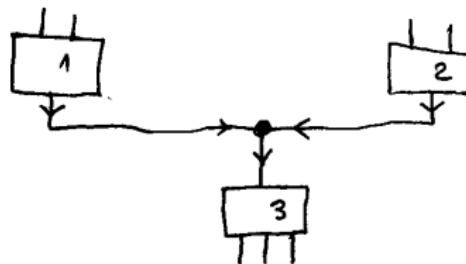
ENABLE	IN	OUT
0	0	Hi-Z
0	1	Hi-Z
1	0	0
1	1	1

Putem considera Hi-Z ca o a treia valoare de adevăr.

Circuite logice

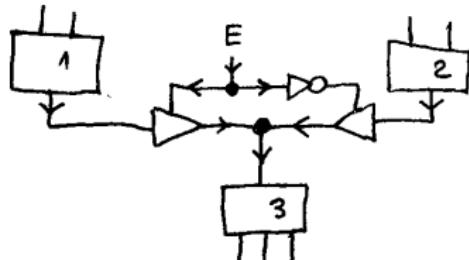
Bufferul cu 3 stări este folosit atunci când, din motive tehnice, dorim să decuplăm funcțional un bloc de la circuit, fără să-l eliminăm fizic (circuitul să se comporte însă ca și când acel bloc n-ar exista).

De exemplu, într-un circuit logic nu se permite / nu se dă sens contactului între linii care aduc valori (în funcție de modul tehnic de realizare, întâlnirea între cele două semnale poate avea efecte imprevizibile (de exemplu, un scurtcircuit):

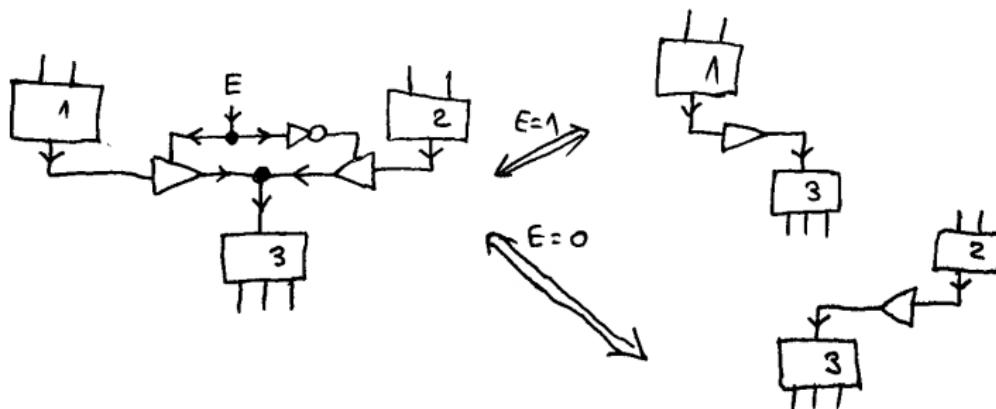


Circuite logice

Putem însă conecta alternativ mai multe ieșiri la o aceeași linie, folosind buffere cu 3 stări:



Atunci:



Circuite logice

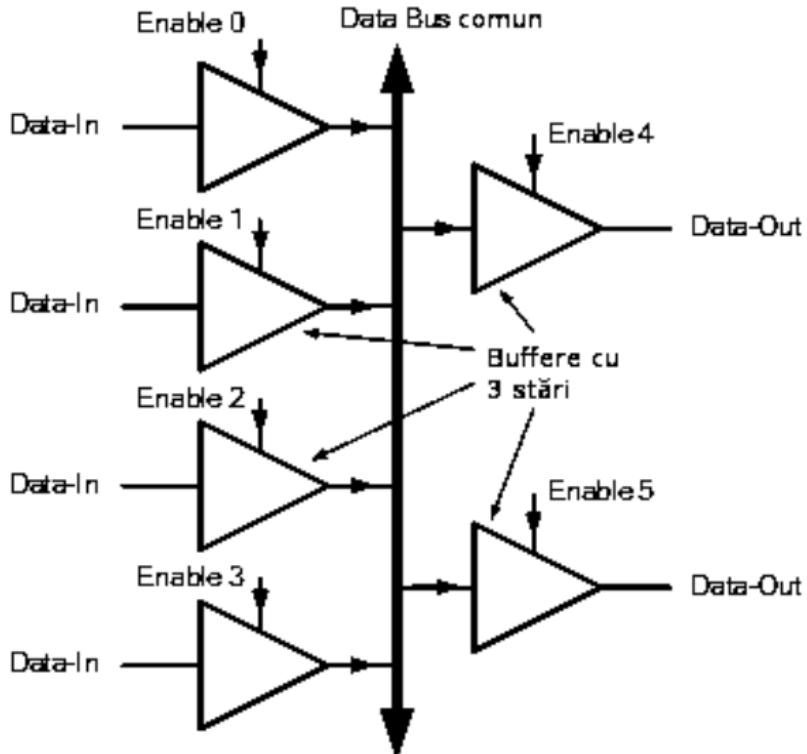
Bufferul cu 3 stări este folosit în multe circuite deoarece permit ca mai multe dispozitive logice să fie conectate la o aceeași linie sau bus fără distrugere fizică sau pierderi de date.

De exemplu, putem avea o linie de date sau bus de date la care sunt conectate memorii, dispozitive de I/O, alte dispozitive periferice sau procesor. Fiecare dintre aceste dispozitive este capabil să emită sau să recepționeze date unul de la altul prin acest unic bus de date în același timp, creând o aşa zisă **dispută (contention)**.

Disputele apar când mai multe dispozitive sunt conectate împreună, deoarece unele intenționează să emită la ieșire o tensiune de nivel înalt, altele una de nivel jos; dacă aceste dispozitive încep să emită sau să recepționeze date în același timp, poate apărea un scurtcircuit atunci când un dispozitiv emite spre bus o valoare 1 (care poate însemna tensiunea sursei de curent), în timp ce altul este pe valoarea 0 (care poate însemna legătura cu pământul, ground); acesta poate cauza distrugerea fizică a unor dispozitive sau pierderi de date.

Circuite logice

Controlul bus-ului de date poate fi realizat folosind buffere cu 3 stări:



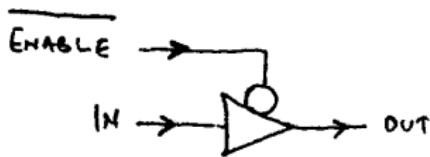
Circuite logice

Există două tipuri de buffer cu 3 stări: unul a cărui ieșire este controlată de un semnal de control activ la nivel înalt (Active-HIGH) și unul care este controlat de un semnal de control activ la nivel jos (Active-LOW).

Varianta descrisă până acum este bufferul cu 3 stări activ la nivel înalt (Active HIGH Tri-state Buffer): el copiază intrarea la ieșire atunci când semnalul *Enable* are valoarea 1 (altfel furnizează la ieșire HI-Z).

Bufferul cu 3 stări activ la nivel jos (Active LOW Tri-state Buffer) copiază intrarea la ieșire atunci când semnalul *Enable* are valoarea 0 (altfel furnizează la ieșire HI-Z).

Simbol și tabel de valori:



ENABLE	IN	OUT
0	0	0
0	1	1
1	0	Hi-Z
1	1	Hi-Z

Circuite logice

Blocurile logice sunt împărțite în două categorii, după cum au sau nu au memorie.

Blocurile fără memorie se zic **combinatoriale**; la un asemenea bloc, ieșirea depinde doar de intrarea curentă și poate fi descrisă printr-un tabel de adevăr; astfel, circuitul implementează de fapt o funcție booleană; blocurile combinatoriale sunt organizate ca circuite logice fără cicluri (0-DS).

În blocurile cu memorie, ieșirea depinde atât de intrarea curentă cât și de valoarea curentă păstrată în memorie și care definește **starea** blocului logic; logica care include stări este **logica secvențială**; blocurile cu memorie sunt organizate ca circuite logice cu cel puțin un nivel de cicluri (n -DS, $n \geq 1$).

Circuite logice

O modelare teoretică a circuitelor logice care evidențiază organizarea (structura) acestora se face cu ajutorul grafurilor orientate.

Def: Un **circuit** este un graf orientat cu cel puțin o intrare și cel puțin o ieșire, care are două tipuri de noduri: **conectori și porți**.

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

Ieșirile unui circuit vor furniza, de asemenea, semnale.

TODO: Descrierea structurii de graf a unui circuit logic printr-o formulă de Network Algebra.

Sisteme digitale

O modelare teoretică a circuitelor logice care evidențiază funcționalitatea acestora o constituie sistemele digitale.

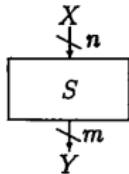
Def: Un **sistem digital (digital system, DS)** este o structură $\langle X, Y, f \rangle$, unde $X = V^n$, $Y = V^m$, $f : X \rightarrow Y$, $n, m \in \mathbb{N}$, iar V este un **alfabet finit, nevid**; f s.n. **funcție de transfer**.

Considerăm doar sisteme digitale **binare**, i.e. cu $V = \{0, 1\} = B_2$.

Cazurile $n = 0$ sau $m = 0$ corespund unor sisteme digitale speciale, de **ieșire**, respectiv **intrare**, care vor fi studiate separat.

Deocamdată presupunem $n, m \neq 0$.

Simbol:



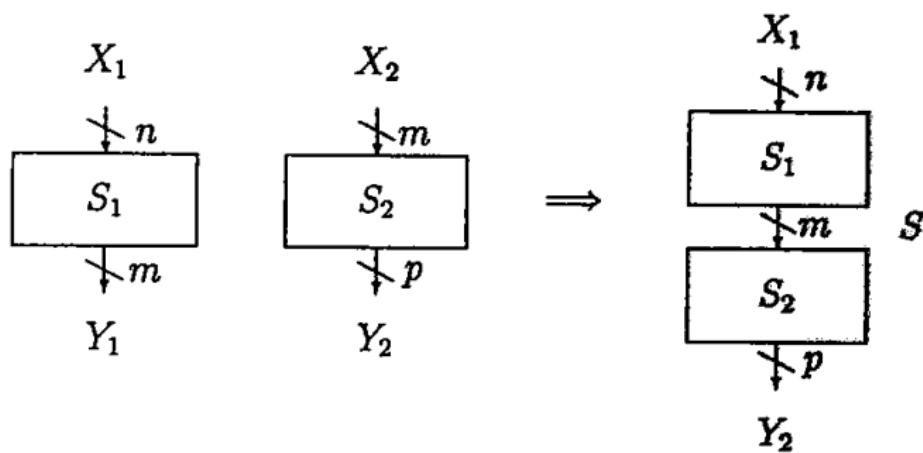
unde $A = (a_1, a_2, \dots, a_n)$ $\updownarrow n \iff \begin{matrix} a_1 & a_2 & \dots & a_n \\ \downarrow & \downarrow & \dots & \downarrow \end{matrix}$ (flux de n date).

Sisteme digitale

Operații cu sisteme digitale:

- Extensia serială:

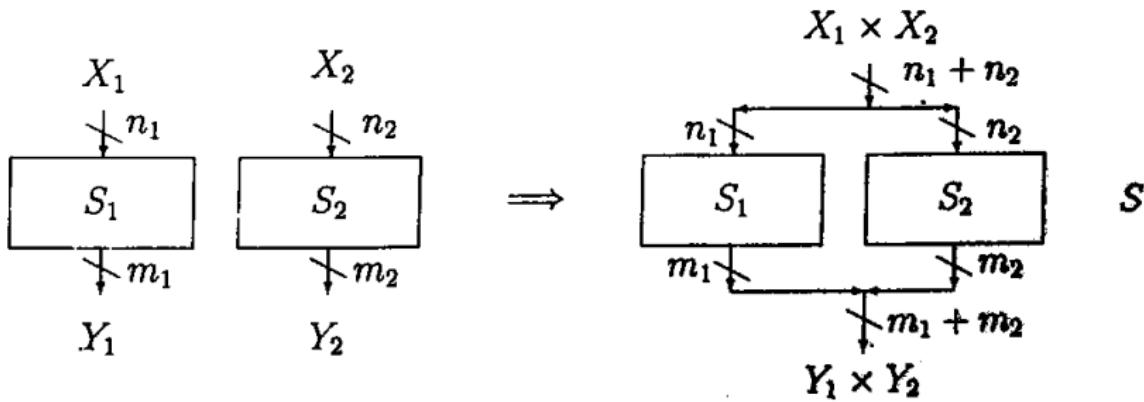
$$\left. \begin{array}{l} S_1 = \langle X_1, Y_1, f \rangle \\ S_2 = \langle X_2, Y_2, g \rangle \\ Y_1 = X_2 \end{array} \right\} \Rightarrow S = \langle X_1, Y_2, g \circ f \rangle$$



Sisteme digitale

- Extensia paralelă:

$$\left. \begin{array}{l} S_1 = \langle X_1, Y_1, f_1 \rangle \\ S_2 = \langle X_2, Y_2, f_2 \rangle \end{array} \right\} \Rightarrow S = S_1 \times S_2 = \langle X_1 \times X_2, Y_1 \times Y_2, f_{12} \rangle, \\ \text{unde } f_{12}(x_1, x_2) = (f_1(x_1), f_2(x_2))$$



Observăm că într-o extensie paralelă cele două sisteme nu interacționează.

Sisteme digitale

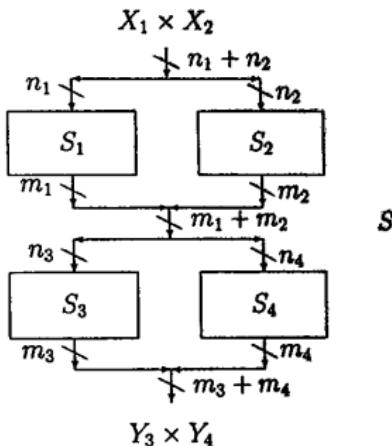
- Extensia serial-paralelă:

$S_i = \langle X_i, Y_i, f_i \rangle$, $X_i = \{0, 1\}^{n_i}$, $Y_i = \{0, 1\}^{m_i}$, $1 \leq i \leq 4$ și $m_1 + m_2 = n_3 + n_4$



$S = \langle X_1 \times X_2, Y_3 \times Y_4, f \rangle$, unde $f = f_{34} \circ f_{12} : X_1 \times X_2 \longrightarrow Y_3 \times Y_4$

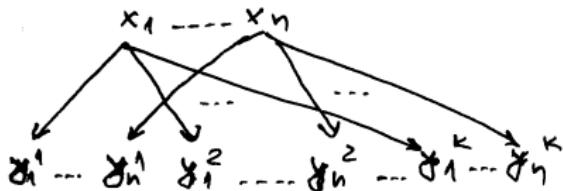
(f_{12} , f_{34} sunt funcțiile de transfer ale extensiilor paralele $S_1 \times S_2$, resp. $S_3 \times S_4$)



Notăm că ieșirile lui S_1 pot fi distribuite atât unor intrări ale lui S_3 cât și unor intrări ale lui S_4 ; la fel și ieșirile lui S_2 .

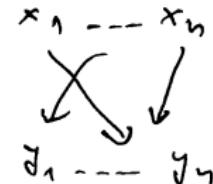
Sisteme digitale

Pentru a modela diversele posibilități de ramificare sau permutare a unor linii de circuit, se pot folosi niște sisteme speciale, care sunt angajate și ele în extensiile pe care le facem:



$$S = (x^n, x^{n+k}, f)$$

$$f(x_1, \dots, x_n) = (x_1, \dots, x_n, \dots, x_1, \dots, x_n)$$



$$S = (x^n, x^n, f)$$

$$f(x_1, \dots, x_n) = (x_{f(1)}, \dots, x_{f(n)})$$

f permutarea de ordin y

Sisteme digitale

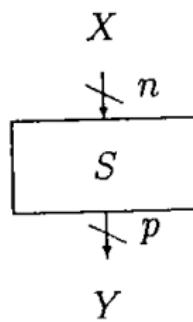
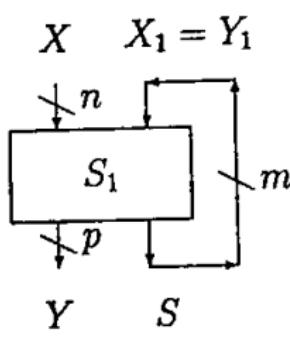
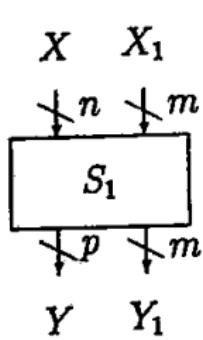
- Închiderea prin ciclu:

$S = \langle X \times X_1, Y \times Y_1, h \rangle$, unde:

$X_1 = Y_1$ iar $h = (f, f_1)$, $f : X \times X_1 \rightarrow Y$, $f_1 : X \times X_1 \rightarrow Y_1$

\implies

$S' = \langle X, Y, g \rangle$, unde $g : X \rightarrow Y$ este definită prin $g(x) = f(x, f_1(x, y))$;
 f_1 s.n. **funcție de tranziție** și verifică definiția recursivă $y = f_1(x, f_1(x, y))$



Observăm că apare o variabilă nouă "ascunsă" care ia valori în mulțimea $Q = X_1 = Y_1$; atunci $f : X \times Q \rightarrow Y$, $f_1 : X \times Q \rightarrow Q$.

Sisteme digitale

Mulțimea Q caracterizează comportarea internă a sistemului, care se mai numește **stare**.

Uneori, elementele legate de stare sunt introduse în definiția sistemului respectiv, a.î. el se va scrie: $S' = \langle X, Y, Q, f_1, g \rangle$.

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

Pentru un $a \in X$ aplicat constant la intrare, ieșirea poate prezenta variații. De aceea, spunem că **autonomia** unui sistem crește ca urmare a introducerii lui într-un ciclu.

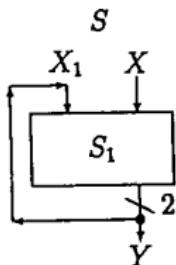
Def: Comportarea unui sistem digital este **autonomă** d.d. pentru o intrare constantă ieșirea are un comportament dinamic.

Sisteme digitale

Exemplu: Fie sistemul $S_1 = \langle X \times X_1, Y, f \rangle$, unde $X = \{0, 1\}$, $X_1 = Y = \{0, 1\}^2$, iar f este definită prin tabelul:

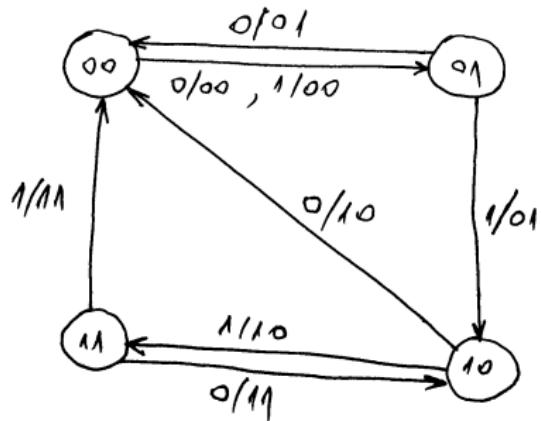
$X \times X_1$	Y	$X \times X_1$	Y
0 00	01	1 00	01
0 01	00	1 01	10
0 10	00	1 10	11
0 11	10	1 11	00

Prin ramificarea în două a ieșirilor Y și apoi identificarea (legarea serială) a uneia dintre ramificări cu X_1 se obține un nou sistem S :



Sisteme digitale

Funcționalitatea sa este ilustrată mai jos (se face abstracție de timpul real cât este aplicat un semnal la intrare); în cerculețe este notată starea curentă (elementul lui $X_1 = Y$); pe fiecare săgeată sunt notate intrarea curentă (elementul lui X) "/" ieșirea curentă (elementul lui Y):



De exemplu, din starea "01", cu intrarea "0", se trece în starea "00" și se emite ieșirea "01".

Dependența ieșirii de intrare și stare este dată de tabelul lui f de mai devreme.

Sisteme digitale

Observație: Sistemul inițial S , fără cicluri, nu avea autonomie - ieșirea sa curentă depindea doar de intrare. După închiderea acestuia printr-un ciclu, sistemul obținut S' capătă, în funcție de tranziție, un comportament al ieșirii autonom de intrare.

De exemplu, intrarea (constantă) 11...1 în sistemul S' va determina ieșirea ciclică: 01 10 11 00 01

Sisteme digitale

Mai notăm că funcționalitatea circuitelor logice (sistemelelor digitale) fără cicluri nu implementează ideea de etapizare - d.p.v. logic, este o funcționare atemporală, la niște intrări se asociază niște ieșiri, pe baza unui tabel; ieșirea curentă depinde doar de intrarea curentă.

Evident, utilizatorul poate folosi circuitul de mai multe ori, dar această etapizare este a utilizatorului, nu a circuitului - el nu reține informații de la o etapă la alta. De exemplu, un calculator de buzunar simplu (neprogramabil și fără memorie) efectuează fiecare operație care îi se comandă ca și cum ar fi prima.

Sisteme digitale

La circuitele logice (sistemele digitale) cu cicluri, ieșirea curentă depinde atât de intrarea curentă cât și de o informație existentă în circuit (ca stare), introdusă acolo la o operare anterioară; de asemenea, în urma operației efectuate se actualizează și informația (starea) internă.

Astfel, funcționalitatea circuitelor cu cicluri implementează ideea de etapizare. Ele sunt folosite în sisteme unde funcționarea este împărțită în etape ce se succed logic; controlul este asigurat cu ajutorul unui circuit special numit **ceas**, care emite un semnal pulsatoriu în timp, iar aceste pulsații declanșază etapele; la fiecare etapă, blocurile logice componente preiau o intrare, efectuează o operație și produc un rezultat care depinde atât de intrarea preluată cât și de starea lor curentă; din acest rezultat este produsă o ieșire și o nouă stare.

În proiectarea unor asemenea sisteme trebuie rezolvate probleme suplimentare legate de durata fizică a ciclului de ceas (pentru a permite blocurilor logice să efectueze operațiile implementate de ele), sincronizarea blocurilor logice (rezultatul furnizat la ieșirea unui bloc să fie disponibil atunci când îl solicită ca intrare blocul următor), etc.

Sisteme digitale

În teoria circuitelor logice:

- **Logica combinațională** se referă la un tip de circuit a cărui ieșire depinde doar de intrarea curentă.
- **Logica secvențială** se referă la un tip de circuit a cărui ieșire depinde nu numai de intrarea curentă ci și de istoricul intrărilor sale anterioare.

Aceasta înseamnă că logica secvențială are **stare (memorie)**, în timp ce logica combinațională nu.

Cu alte cuvinte, logica secvențială este logică combinațională cu memorie.

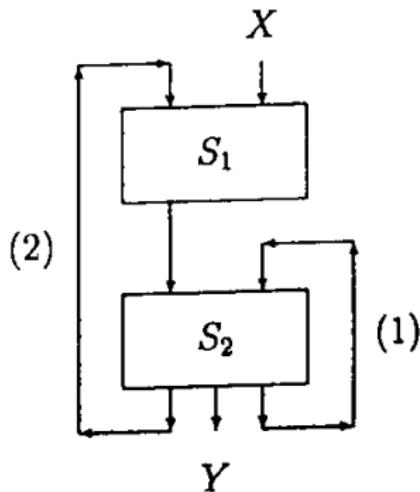
Din cele de mai sus, rezultă că circuitele logice (sistemele digitale) fără cicluri sunt prezente în logica combinațională, iar cele cu cicluri în logica secvențială.

Sisteme digitale

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

Spunem că A este subciclu al lui B .

Exemplu: În figura de mai jos, (1) este subciclu al lui (2):



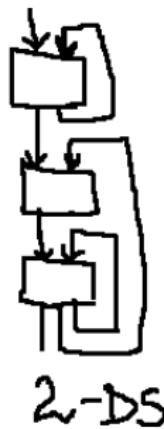
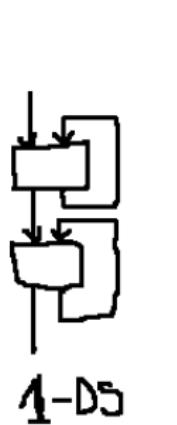
Sisteme digitale

Def: Un sistem digital de ordin n (**n -digital system, n -DS**), $n \geq 0$, se definește recursiv astfel:

1. Orice circuit combinațional (fără cicluri) este un 0-DS.
2. Un $(n + 1)$ -DS se obține dintr-un n -DS adăugând un ciclu care include toate ciclurile anterioare.
3. Orice n -DS se obține aplicând de un număr finit de ori regulile anterioare.

Sisteme digitale

Obs: Un n -DS are n niveluri de cicluri incluse unele în altele, nu n cicluri. De exemplu:



(am presupus că blocurile simbolizate nu conțin cicluri).

Sisteme digitale

Vom arăta următoarea corespondență ierarhică pentru n -DS, care corespunde arhitecturii unui calculator:

- 0-DS: circuite combinaționale, funcții booleene;
- 1-DS: memorii;
- 2-DS: automate finite;
- 3-DS: procesoare;
- 4-DS: calculatoare.

Cuprins

1 Performația calculatoarelor

Conceptul de performanță
Măsurarea performanței

2 Aritmetică sistemelor de calcul

Reprezentarea numerelor în matematică
Reprezentarea numerelor în calculator
Reprezentarea numerelor naturale ca întregi fără semn
Reprezentarea numerelor întregi în complement față de 2
Reprezentarea numerelor reale în virgulă mobilă

3 Logica pentru calculatoare

Algebre booleene
Funcții booleene
Particularizare la cazul B_2

4 Circuite logice

Circuite logice, Sisteme digitale
0-DS (Circuite combinaționale, Funcții booleene)
1-DS (Memorii)
2-DS (Automate finite)
Algoritmi de înmulțire și împărțire hardware
3-DS (Procesoare) și 4-DS (Calculatoare)
 n -DS, $n > 4$

Circuite combinaționale

Sistemele 0-DS sunt circuite logice fără cicluri.

La aceste circuite, ieșirea curentă depinde doar de intrarea curentă, deci ele nu au memorie și astfel sunt prezente în logica combinațională; de aceea se mai numesc și **circuite combinaționale (combinational logic circuit, CLC)**.

Dependența ieșirii curente de intrarea curentă a unui 0-DS poate fi descrisă printr-un tabel de valori, astfel că un 0-DS implementează o funcție booleană. Așa cum vom vedea mai târziu, orice funcție booleană poate fi implementată printr-un 0-DS, deci avem o echivalență între 0-DS și funcții booleene.

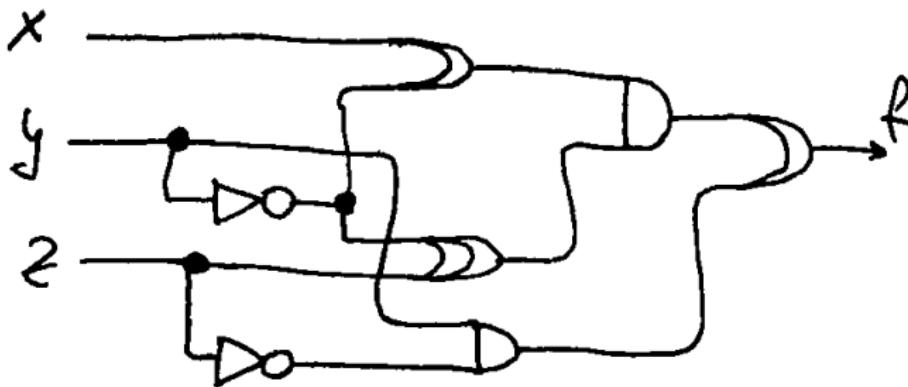
Există mai multe modalități de a implementa o funcție booleană ca un 0-DS: implementare directă a unei formule date, minimizarea numărului de operații și implementarea printr-un circuit minimal, implementarea printr-un circuit general construit după un anumit tipar, implementarea cu ajutorul unor blocuri CLC speciale: codificator, multiplexor, etc.

Implementare directă

Implementare directă:

O funcție booleană dată printr-o formulă poate fi implementată combinând porțile în aceeași ordine în care se compun operațiile booleene implementate de ele pentru a se obține formula.

Exemplu: Funcția $f : B_2^3 \rightarrow B_2$, $f(x, y, z) = (x + \bar{y})(z \oplus \bar{y}) + y\bar{z}$ poate fi implementată prin circuitul:



Circuit minimal

Circuit minimal:

O funcție booleană poate fi implementată descriind-o mai întâi printr-o formulă cu număr minim de operații și implementând apoi direct această formulă (circuitul va avea un număr minim de porți).

Problema generală a minimizării circuitului este considerată **intractabilă (intractable)**, i.e. poate fi rezolvată în teorie (de exemplu, fiind dat un timp lung, dar finit), dar în practică durează prea mult pentru ca soluția ei să fie utilă.

Există însă metode euristice eficiente, ca **hărțile Karnaugh (Karnaugh maps)** și **algoritmul QuineMcCluskey**.

În multe cazuri poate fi util să încercăm să transformăm formula, aplicând proprietăți de algebră booleană, până când aceasta nu mai conține 0 sau 1 iar variabilele nu se mai repetă.

Circuit minimal

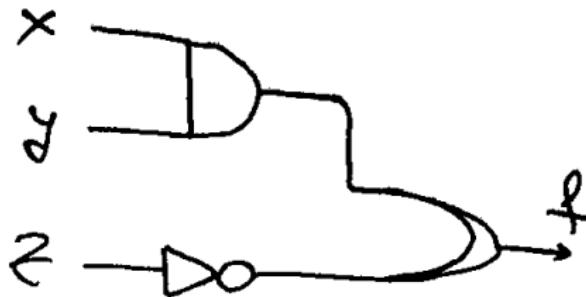
Exemplu: Pentru funcția din exemplul anterior, avem:

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

$$(x + \bar{y})(yz + \bar{y}\bar{z}) + y\bar{z} = xyz + x\bar{y}\bar{z} + \bar{y}yz + \bar{y}\bar{y}\bar{z} + y\bar{z} =$$

$$xyz + x\bar{y}\bar{z} + \bar{y}\bar{z} + y\bar{z} = xyz + \bar{y}\bar{z} + y\bar{z} = xyz + (\bar{y} + y)\bar{z} = xyz + \bar{z} = xy + \bar{z}$$

Astfel, ea poate fi implementată prin circuitul:



Circuit general

Circuit general:

O funcție booleană poate fi implementată printr-un circuit general, construit după un anumit tipar, pornind de la o formulă standard a funcției - de exemplu, scrierea ei ca sumă de produse sau ca FND.

Exemplu: Fie funcția $f = (f_1, f_2) : B_2^3 \rightarrow B_2^2$ ($f_1, f_2 : B_2^3 \rightarrow B_2$) dată prin tabelul:

x	y	z	f_1	f_2
0	0	0	0	0
0	0	1	0	0
0	1	0	0	1
0	1	1	1	0
1	0	0	0	0
1	0	1	0	0
1	1	0	1	1
1	1	1	1	0

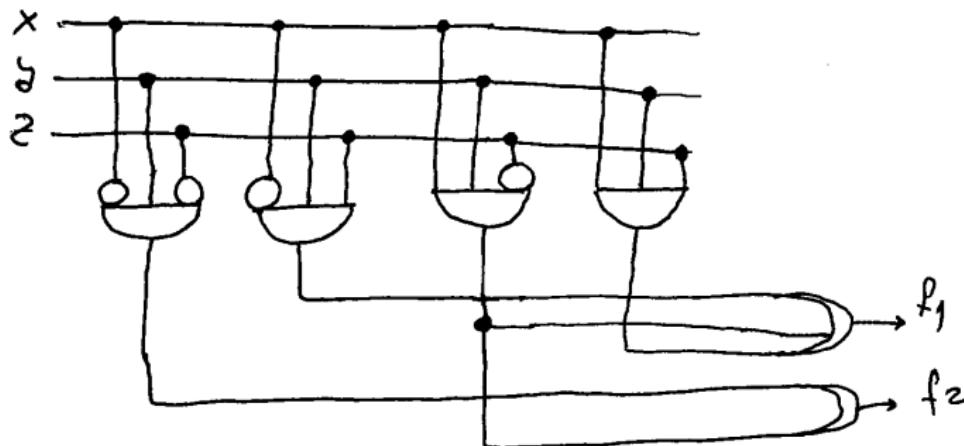
Din tabel rezultă că f_1, f_2 au respectiv următoarele FND:

$$f_1(x, y, z) = \bar{x}yz + xy\bar{z} + xyz$$

$$f_2(x, y, z) = \bar{x}y\bar{z} + xy\bar{z}$$

Circuit general

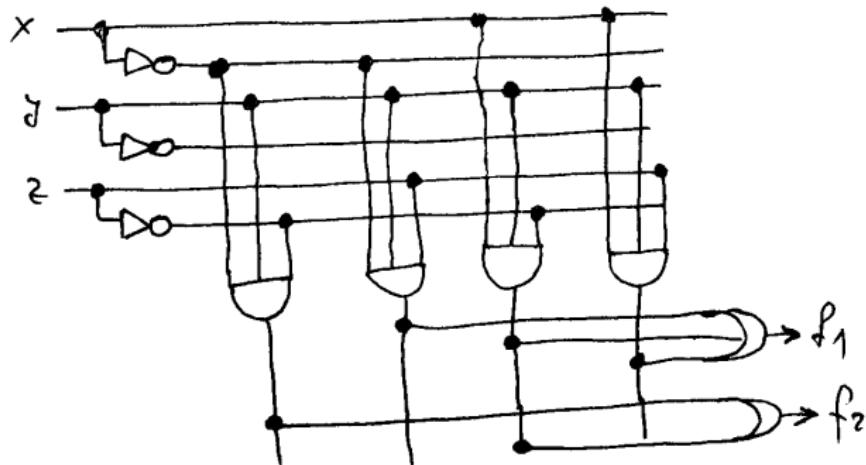
Atunci, funcția f poate fi implementată prin circuitul:



Observăm că am folosit două porți "NOT" pentru a calcula aceeași valoare \bar{x} și două porți "NOT" pentru a calcula aceeași valoare \bar{z} .

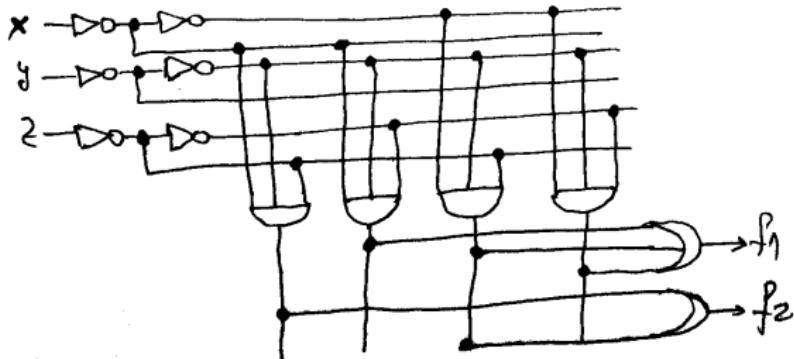
Circuit general

Putem evita folosirea mai multor porți "NOT" pentru negarea aceleiași variabile construind circuitul astfel:



Circuit general

O variantă a circuitului de mai sus, pe care o vom prefera în cele ce urmează, este următoarea:



Plasarea porților "NOT" imediat la intrările x, y, z are următoarele avantaje (datorate efectului de buffer al porților "NOT"):

- se împiedică pătrunderea semnalelor perturbate adânc în circuit (din porțile "NOT" respective va ieși un semnal clar 0 sau 1);
- se izolează circuitul de alte circuite aflate la intrare, evitând influența negativă a impedanței acestora;
- semnalul este amplificat și astfel poate fi distribuit mai multor porți "AND" (un circuit cu n variabile de intrare poate avea până la 2^n porți "AND").

Circuit general

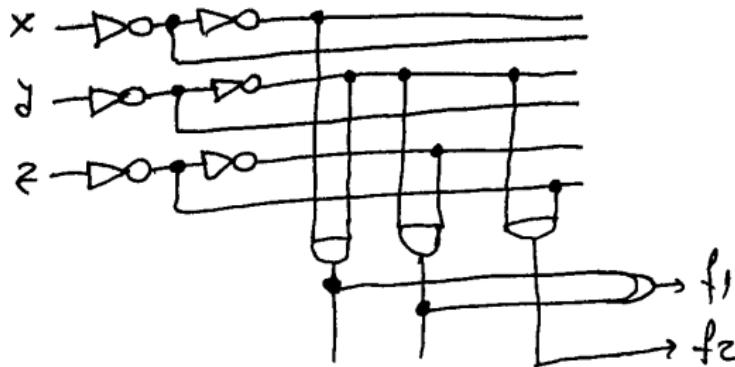
Putem minimiza scrierea funcției ca sumă de produse (să avem cât mai puține operații + și ·) înainte de a o implementa prin circuitul general. Astfel, vom obține un circuit general minimal.

Exemplu: Pentru funcția f din exemplul anterior, avem:

$$f_1(x, y, z) = \bar{x}yz + xy\bar{z} + xyz = \bar{x}yz + xy(\bar{z} + z) = \bar{x}yz + xy = (\bar{x}z + x)y = (z + x)y = xy + yz$$

$$f_2(x, y, z) = \bar{x}y\bar{z} + xy\bar{z} = (\bar{x} + x)y\bar{z} = y\bar{z}$$

Atunci, f poate fi implementată prin circuitul:



Circuit general

Toate variantele de circuit general prezentate până acum, cu excepția primeia, au trei părți:

- un bloc care calculează variabilele și negațiile lor;
- un bloc care calculează produse "AND";
- un bloc care calculează sume "OR"

(prima variantă constructivă calculează negațiile în blocul al doilea).

Structura primului bloc depinde doar de numărul de variabile, în timp ce structura blocurilor al doilea și al treilea depinde de funcție.

Dacă ne imaginăm că implementăm funcțiile cu un număr fixat de variabile prin plăci de extensie inserate într-un soclu pe o placă de bază, primul bloc ar putea fi integrat în soclu (pe placa de bază), iar blocurile al doilea și al treilea pe placă de extensie.

Putem standardiza și mai mult circuitul general și să integrăm mai mult în soclu dacă în blocul al doilea am calcula toate cele 2^n produse posibile cu cele n variabile date (ele corespund liniilor din tabelul de valori al funcției) - atunci structura primelor două blocuri va depinde doar de numărul de variabile (iar ele vor putea fi integrate în soclu) și doar structura celui de-al treilea bloc va depinde de funcție (iar el va fi integrat în placă de extensie).

În acest caz, suma de produse implementată de circuit este FND.

Circuit general

Exemplu: Pentru funcția f din exemplele anterioare, ale cărei componente au, reamintim, respectiv următoarele FND:

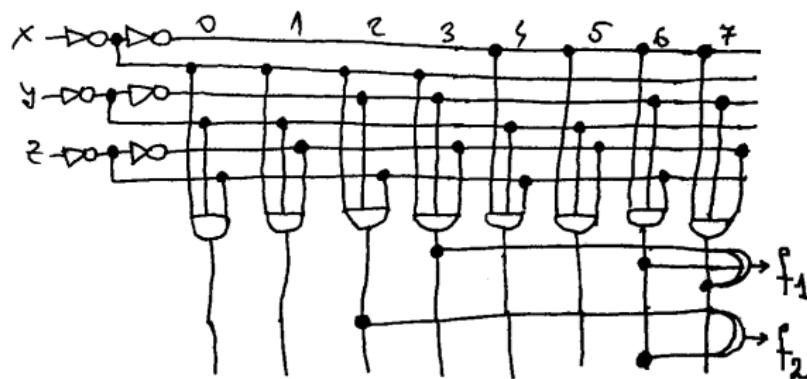
$$f_1(x, y, z) = \bar{x}yz + xy\bar{z} + xyz, \quad f_2(x, y, z) = \bar{x}y\bar{z} + xy\bar{z}$$

obținem circuitul următor (am numerotat liniile din tabel, termenii din FND și produsele din blocul "AND" pentru a observa mai ușor corespondențele):

	x	y	z	f_1	f_2
0	0	0	0	0	0
1	0	0	1	0	0
2	0	1	0	0	1
3	0	1	1	1	0
4	1	0	0	0	0
5	1	0	1	0	0
6	1	1	0	1	1
7	1	1	1	1	0

$$FND: f_1(x,y,z) = \bar{x}^3y^6z^7$$

$$f_2(x,y,z) = \bar{x}^2\bar{y}\bar{z} + xy\bar{z}^6$$



Circuit general

Putem desena mai rapid circuitul, observând următoarele proprietăți:

- Punctele de contact ale produselor din blocul "AND" sunt în concordanță cu aparițiile cu/fără \neg ale variabilelor în termenii corespunzători din FND, care sunt în concordanță cu valorile 0/1 ale acestor variabile în tabel, care traduc în binar numerele de ordine ale liniilor tabelului; de aceea, ne putem gândi că transcriem în binar aceste numere de ordine direct prin puncte de contact (așa cum s-au transcris prin sistemele de \cup și \cap deasupra termenilor din FND): 0 înseamnă contact pe linia de jos, 1 înseamnă contact pe linia de sus (în perechea de linii care furnizează valoarea unei variabile și negația ei);
- Modul în care alternează valorile în tabel în coloanele variabilelor, cu perioade care se înjumătătesc odată cu scăderea semnificației variabilelor (în coloana variabilei celei mai semnificative, jumătate 0, jumătate 1, în coloana variabilei următoare ca semnificație, un sfert 0, un sfert 1, etc.) și care ne permite să completăm tabelul pe coloane, ne permite să plasăm punctele de contact în blocul "AND" pe liniile: în fiecare produs, variabila cea mai semnificativă are jumătate din contacte pe linia de jos, jumătate pe linia de sus, variabila următoare ca semnificație are un sfert din contacte pe linia de jos, un sfert pe linia de sus, apoi iar un sfert pe linia de jos, un sfert pe linia de sus, etc..
- Sumele din blocul "OR" au punctele de contact la produsele care corespund liniilor din tabel unde componenta corespunzătoare a funcției are valoarea 1.

Circuit general

Constatăm că orice funcție booleană se poate implementa printr-un circuit general, în oricare din variantele constructive prezentate mai sus, de aceea circuitele de acest tip au statut de **circuite universale**.

Totodată, vedem cum orice funcție booleană poate fi implementată printr-un 0-DS, ceea ce încheie justificarea echivalenței între 0-DS și funcții booleene, afirmată mai devreme.

PLA și PROM

PLA și PROM:

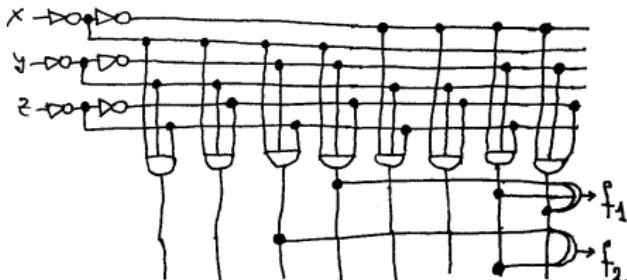
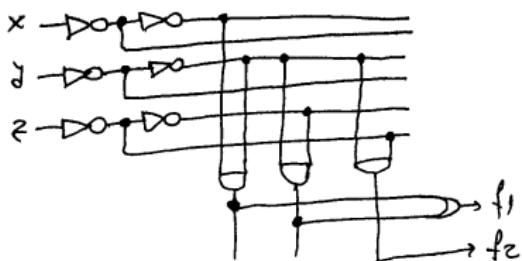
Circuitele generale prezentate mai devreme se pot desena folosind următoarele convenții grafice mai simple:

- Blocul care furnizează valorile variabilelor și negațiile lor se desenează la fel.
- Fiecare produs din blocul "AND" (care acum se mai numește și **planul "AND"**) se desenează printr-o linie verticală care are contacte pe aceleași linii ca și produsul.
- Fiecare sumă din blocul "OR" (care acum se mai numește și **planul "OR"**) se desenează printr-o linie orizontală care are contacte pe aceleași linii ca și suma.

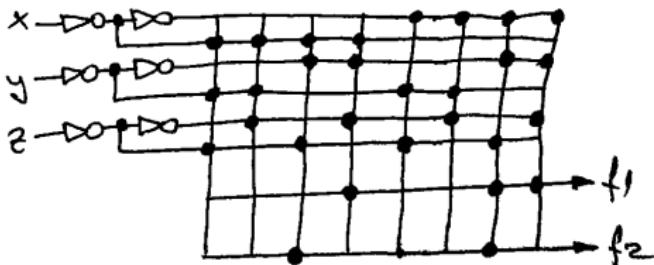
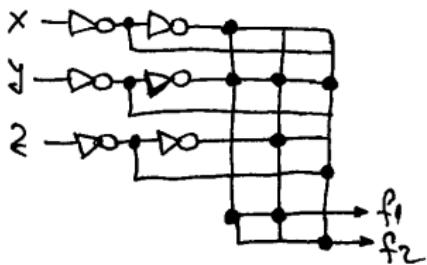
Așadar, liniile desenate au semantici diferite, în funcție de poziția lor pe desen: cele orizontale din blocul de sus sunt linii de circuit, cele verticale din planul "AND" sunt produse, cele orizontale din planul "OR" sunt sume.

PLA și PROM

Exemplu: Următoarele circuite care implementează funcția f din exemplele anterioare (cel cu număr minim de produse/sume și cel cu toate produsele posibile):



se vor desena astfel:



PLA și PROM

Circuitele generale desenate după convențiile mai simple de mai sus s.n. **PLA (Programmable Logic Array)**; cel care are în planul "AND" toate produsele posibile cu variabilele date se mai numește și **PROM (Programmable Read-only Memory)** (deci un PROM este un caz particular de PLA). De obicei însă, denumirea de PLA este folosită doar pentru circuitul cu număr minim de produse/sume.

Din punct de vedere tehnic, având în vedere gradul ridicat de standardizare, aceste circuite se pot construi după tehnologii diferite, mai simple, decât circuitele alcătuite din porti și având o organizare oarecare.

Întrucât pot implementa orice funcție booleană, PLA și PROM sunt circuite universale.

PLA și PROM

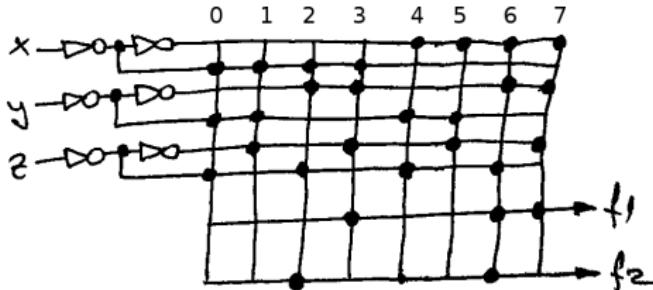
Din cele de mai sus rezultă că putem construi PROM-ul unei funcții direct din tabelul extins al acesteia, procedând astfel:

- construim tabelul extins;
- construim blocul care furnizează valorile variabilelor și negațiile lor și planul "AND" al PROM-ului (ele depind doar de numărul de variabile);
- construim planul "OR" al PROM-ului, plasând pe fiecare linie - sumă corespunzătoare unei componente a funcției puncte de contact cu liniile - produs corespunzătoare liniilor din tabel în care componenta respectivă are valoarea 1.

PLA și PROM

Exemplu: Pentru funcția $f = (f_1, f_2) : B_2^3 \rightarrow B_2^2$ ($f_1, f_2 : B_2^3 \rightarrow B_2$) din exemplele anterioare, avem (pentru claritate, am numerotat liniile din tabel și liniile - produs corespunzătoare din PROM și am menționat separat liniile în care componentele lui f au valoarea 1):

x	y	z	f_1	f_2	
0	0	0	0	0	0 $f_1 : 3, 6, 7$
0	0	1	0	0	1 $f_2 : 2, 6$
0	1	0	0	1	2
0	1	1	1	0	3
1	0	0	0	0	4
1	0	1	0	0	5
1	1	0	1	1	6
1	1	1	1	0	7



PLA și PROM

Pentru a construi PLA-ul unei funcții, având numărul minim de produse/sume, putem proceda astfel:

- construim tabelul extins;
- din tabelul extins extragem FND-urile componentelor funcției;
- minimizăm scrierea componentelor funcției ca sume de produse, transformând FND-urile cu proprietăți de algebră booleană;
- construim PLA-ul pornind de la formulele minimale, într-o manieră asemănătoare celei în care am construit circuitul general minimal.

Un alt mod de a construi PLA-ul unei funcții folosește tabelul compact (a se vedea sfârșitul secțiunii referitoare la algebra booleană B_2):

- construim tabelul compact;
- din tabelul compact extragem direct scrieri simplificate ca sume de produse ale componentelor funcției; dacă aceste scrieri nu sunt minimale, ele se pot transforma în continuare cu proprietăți de algebră booleană;
- construim PLA-ul pornind de la formulele minimale, într-o manieră asemănătoare celei în care am construit circuitul general minimal.

PLA și PROM

Exemplu: Ilustrăm ultima modalitate de construcție a PLA, pentru aceeași funcție ca mai devreme. Tabelul său compact este:

		z			
x	y	0	1	f1	f2
0	0	0 0	0 0	0	0
0	1	0 1	1 0	z	\bar{z}
1	0	0 0	0 0	0	0
1	1	1 1	1 0	1	\bar{z}

Din tabelul compact rezultă următoarele scrierii simplificate ca sume de produse ale componentelor funcției:

$$f_1(x, y, z) = \bar{x}yz + xy$$

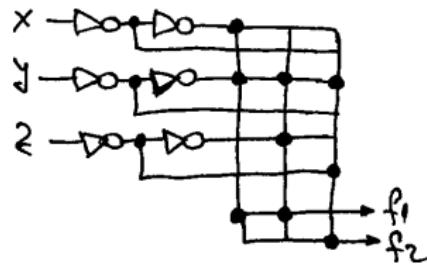
$$f_2(x, y, z) = y\bar{z}$$

Observăm că putem simplifica încă mai mult formula lui f_1 , folosind proprietatea booleană:

$$f_1(x, y, z) = (\bar{x}z + x)y = (z + x)y = xy + yz$$

PLA și PROM

Pornind de la formulele minimale, obținem următorul PLA:



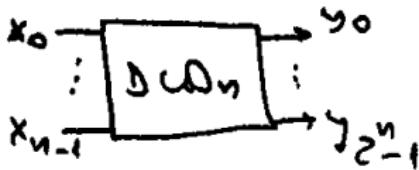
Decodificator

Decodificator:

Un **decodificator (decoder)** cu selector pe n biți DCD_n , $n \geq 1$, este un circuit care transformă un cod numeric k pe n biți într-o alegere fizică, a liniei de ieșire cu numărul k (prin care trimite 1).

De fapt, acesta este un caz particular de decodificator, numit **decodificator linie (line decoder)**.

Simbol:



El primește ca intrare un sistem de valori x_{n-1}, \dots, x_0 , numit **selector**, care este reprezentarea în calculator a unui număr natural $k \in \{0, 2^n - 1\}$ ca întreg fără semn și furnizează ca ieșire un sistem de valori y_0, \dots, y_{2^n-1} a.î. $y_k = 1$ și $y_i = 0$ pentru $i \neq k$.

Liniile de ieșire y_0, \dots, y_{2^n-1} pot fi conectate la diverse echipamente (circuite) și astfel va fi activat echipamentul cu numărul k .

Deci, cu ajutorul unui decodificator, putem activa diverse echipamente, selectable printr-un cod numeric.

Decodificator

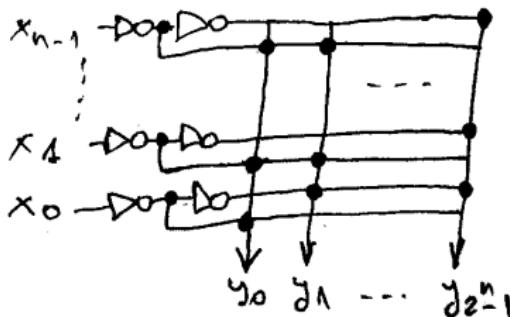
DCD_n implementează o funcție $f : B_2^n \rightarrow B_{2^n}$ având următorul tabel de valori:

$x_{n-1} \dots x_1 x_0$	$y_0 \quad y_1 \quad \dots \quad y_{2^n-1}$
0 0 0	1 0 0 0 0 0 0 0
0 0 1	0 1 0 0 0 0 0 0
⋮	⋮
1 1 1	0 0 1 0 0 0 0 0

Așadar, coloana fiecărei componente y_k a funcției, $0 \leq k \leq 2^n - 1$, conține o singură valoare 1, anume pe linia k .

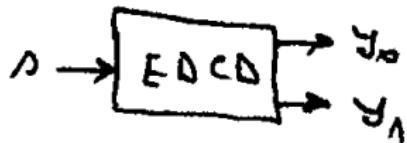
Atunci, dacă implementăm această funcție ca PROM, fiecare produs din planul "AND" va participa la o singură sumă.

În consecință, un decodificator este planul "AND" dintr-un PROM:

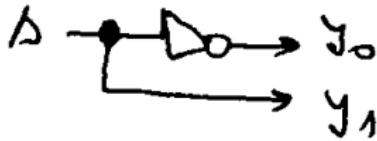


Decodificator

Pentru $n = 1$ obținem **decodificatorul elementar (elementary decoder)**, $EDCD$; simbolizare:

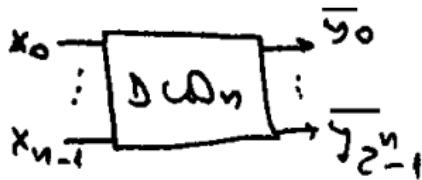


Construcție:



Decodicator

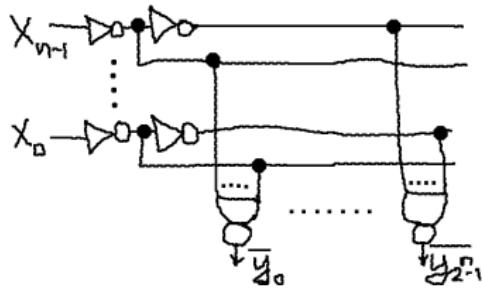
Uneori este utilă o variantă a decodicatorului, în care ieșirile sunt complementate:



El transformă selectorul $(x_{n-1}, \dots, x_0) = [k]^u_n$ în sistemul de valori y_0, \dots, y_{2^n-1} a.î. $y_k = 0$ și $y_i = 1$ pentru $i \neq k$.

Decodicator

Pentru construcție, în planul "AND" al PROM-ului se înlocuiesc porțile "AND" cu porți "NAND":



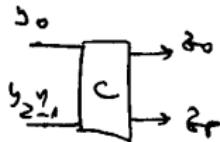
În cazul EDCD, se poate proceda astfel:



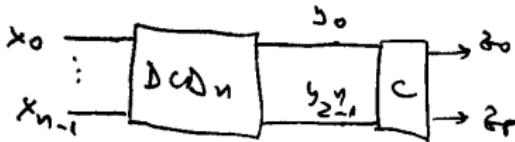
Codificator

Codificator:

Un $(2^n, p)$ - codificator (encoder) este un circuit cu 2^n intrări dintre care la fiecare moment doar una este activă (i.e. are valoarea 1) și care generează la ieșire o configurație binară oarecare de lungime p . Simbol:



Pentru a se garanta că dintre cele 2^n intrări la fiecare moment exact una este activă, codificatorul este însoțit întotdeauna de un decodificator:

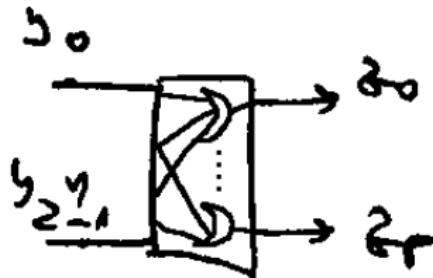


Circuitul rezultat transformă: $(x_{n-1}, \dots, x_0) = [k]_n^u, 0 \leq k \leq 2^n - 1 \Rightarrow (y_0, \dots, y_{2^n-1}) = (0, \dots, 0, 1, 0, \dots, 0)$ (1 este pe poziția k) $\Rightarrow (z_1, \dots, z_p)$, unde pentru orice $1 \leq i \leq p$, $z_i = z_i(k) = z_i(x_{n-1}, \dots, x_0) : B_2^{n-1} \rightarrow B_2$ este o funcție booleană scalară oarecare, deci $z = (z_1, \dots, z_p) : B_2^{n-1} \rightarrow B_2^p$ este o funcție booleană vectorială oarecare.

Codificator

Întrucât am văzut că orice funcție booleană se poate implementa într-un mod standard ca PROM iar decodificatorul este planul "AND" al acestuia, rezultă că codificatorul poate fi construit ca planul său "OR".

Mai exact, codificatorul se construiește ca un sistem de porți "OR", fiecare furnizând ca ieșire câte un z_i , $1 \leq i \leq p$, și primind ca intrare acei y_k pentru care, dacă valoarea este 1, atunci și $z_i = 1$:



Așadar, codificatorul este planul "OR" dintr-un PROM.

Întrucât am văzut că PROM-ul este un circuit universal, rezultă că și codificatorul (însorit de decodificator) este un circuit universal, i.e. poate implementa orice funcție booleană.

Codificator

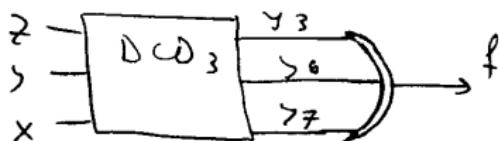
Exemplu: Implementați cu ajutorul unui codificator funcția booleană

$f : B_2^3 \rightarrow B_2$ dată prin tabelul:

x	y	z	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Codificatorul va conține un "OR" ce furnizează ca ieșire f și are ca intrări ieșirile y_k al DCD_3 ce corespund liniilor cu numerele k din tabel pentru care $f = 1$:

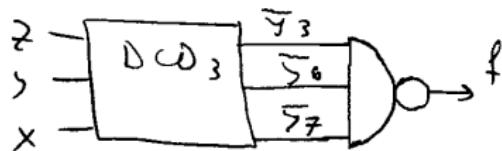
x	y	z	f
0	0	0	0
1	0	0	0
2	0	0	0
3	1	0	1
4	0	0	0
5	0	0	0
6	1	0	1
7	1	1	1



Codicator

Dacă folosim un decodificator cu ieșirile negate, în construcția codificatorului se înlocuiesc porțile "OR" cu porți "NAND".

Exemplu: Pentru funcția f din exemplul anterior, obținem implementarea:

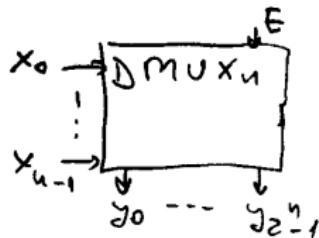


Demultiplexor

Demultiplexor:

Un **demultiplexor (demultiplexer)** cu selector pe n biți $DMUX_n$, $n \geq 1$, este un comutator de tip "one into many", care poate conecta o intrare unică la o ieșire selectabilă printr-un cod numeric.

Simbol:



El primește ca intrare:

- un sistem de valori x_{n-1}, \dots, x_0 , numit **selector**, care este reprezentarea în calculator a unui număr natural $k \in \{0, 2^n - 1\}$ ca întreg fără semn;
- o valoare $E \in \{0, 1\}$;

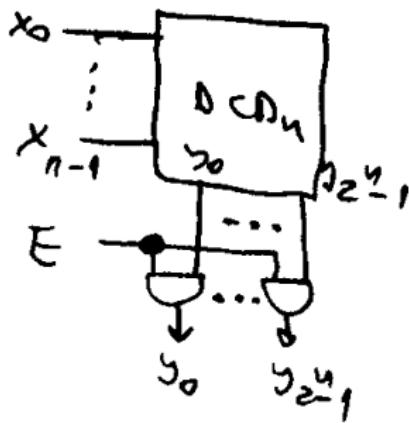
și furnizează ca ieșire:

- un sistem de valori y_0, \dots, y_{2^n-1} a.î. $y_k = E$ și $y_i = 0$ pentru $i \neq k$.

Demultiplexor

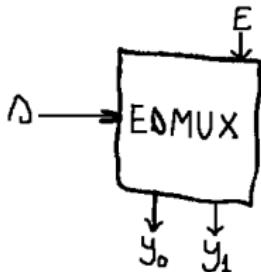
Constatăm că demultiplexorul este o generalizare a decodificatorului; mai exact, un DCD_n este un $DMUX_n$ unde am fixat $E = 1$.

De aceea, demultiplexorul se construiește ușor cu ajutorul unui decodificator - se conjugă toate ieșirile decodificatorului cu E :

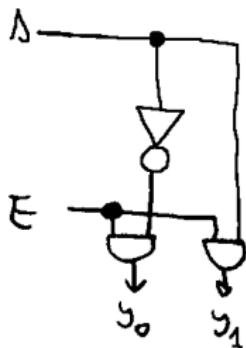


Demultiplexor

Pentru $n = 1$ obținem **demultiplexorul elementar (elementary demultiplexer)**, **EDMUX**; simbolizare:



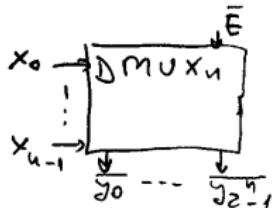
Construcție:



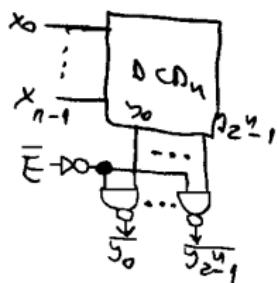
Demultiplexor

Putem construi o variantă a demultiplexorului, în care intrarea E și ieșirile y_0, \dots, y_{2^n-1} sunt negate.

Simbolizare:



Construcție:



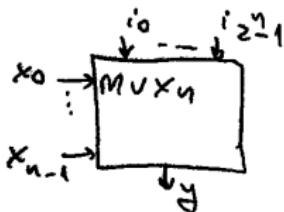
Această variantă are avantajul că și intrarea E va fi protejată iar semnalul va fi amplificat înainte de combinarea cu cele 2^n ieșiri ale decodificatorului, datorită efectului de buffer al porții "NOT".

Multiplexor

Multiplexor:

Un **multiplexor (multiplexer)** cu selector pe n biți MUX_n , $n \geq 1$, este un comutator de tip "many into one", care poate conecta o intrare selectabilă printr-un cod numeric la o ieșire unică.

Simbol:

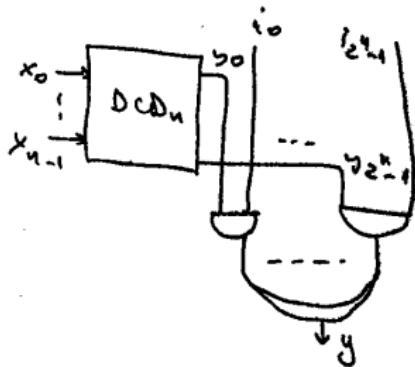


El primește ca intrare:

- un sistem de valori x_{n-1}, \dots, x_0 , numit **selector**, care este reprezentarea în calculator a unui număr natural $k \in \{0, 2^n - 1\}$ ca întreg fără semn;
 - un sistem de valori oarecare i_0, \dots, i_{2^n-1} ;
- și furnizează ca ieșire: i_k .

Multiplexor

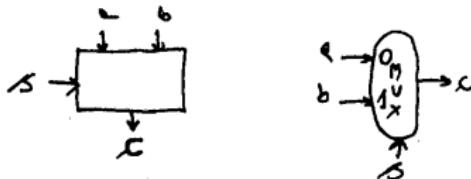
Construcție:



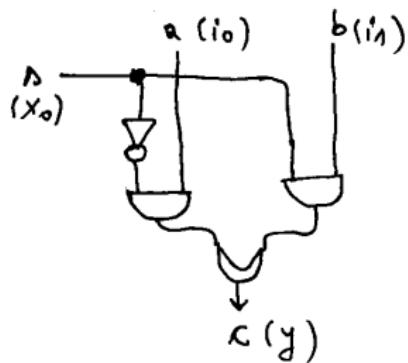
Decodificatorul transformă selectorul $k = (x_{n-1}, \dots, x_0)$ în sistemul de ieșiri $0, \dots, 0, 1, 0, \dots, 0$, unde 1 este pe poziția k ; aceste ieșiri sunt conjugate cu respectiv intrările i_0, \dots, i_{2^n-1} , rezultând sistemul de valori $0, \dots, 0, i_k, 0, \dots, 0$, unde i_k este pe poziția k ; aceste valori intră într-o poartă "OR", care efectuează $0 + \dots + i_k + \dots + 0$, rezultând în final i_k .

Multiplexor

Pentru $n = 1$ obținem **multiplexorul elementar (elementary multiplexer)**, EMUX; variante de simbolizare (în primul caz subînțelegem că intrarea corespunzătoare selectorului $s = 0$ este în stânga):

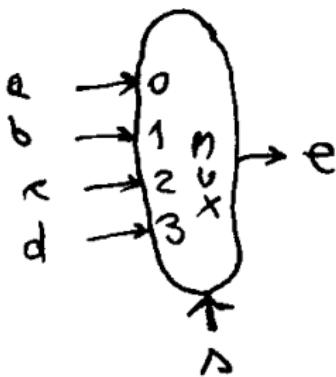


Construcție:



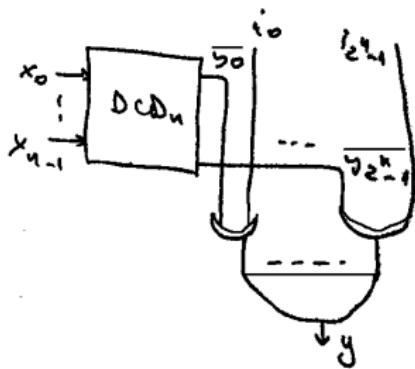
Multiplexor

De asemenea, vom folosi și MUX_2 , pentru care avem și simbolizarea:



Multiplexor

Dacă folosim un decodificator cu ieșirile negate, în construcția multiplexorului se interschimbă porțile nou adăugate "AND" și "OR":



Acum, decodificatorul transformă selectorul $k = (x_{n-1}, \dots, x_0)$ în sistemul de ieșiri $1, \dots, 1, 0, 1, \dots, 1$, unde 0 este pe poziția k ; aceste ieșiri sunt disjunse cu respectiv intrările i_0, \dots, i_{2^n-1} , rezultând sistemul de valori $1, \dots, 1, i_k, 1, \dots, 1$, unde i_k este pe poziția k ; aceste valori intră într-o poartă "AND", care efectuează $1 \cdot \dots \cdot i_k \cdot \dots \cdot 1$, rezultând în final i_k .

Multiplexor

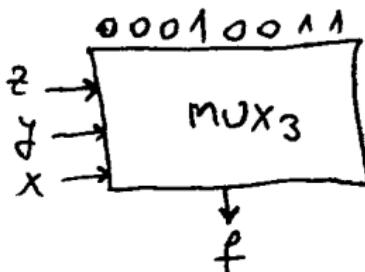
Multiplexorul este și el un circuit universal, deoarece putem implementa orice funcție booleană scalară cu un multiplexor.

Exemplu: Implementați funcția booleană următoare cu un multiplexor:

$f: B_2^3 \rightarrow B_2$ date prim
tabelul:

x	y	z	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Implementare:



Multiplexor

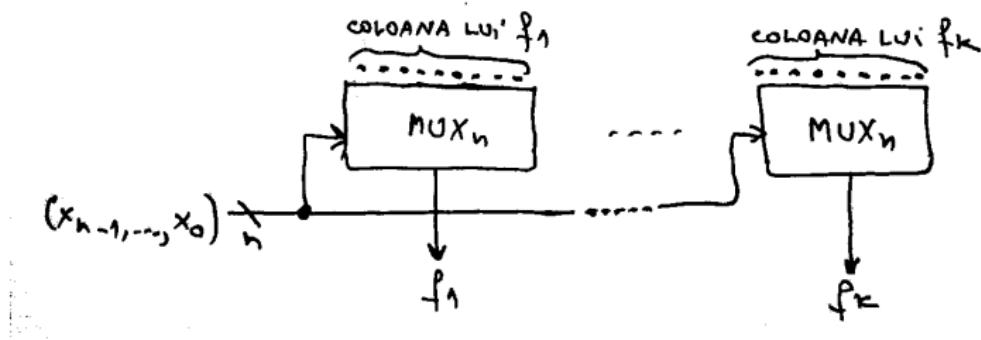
Intuitiv, fiecare sistem de valori ale variabilelor x, y, z , care este reprezentarea unui număr natural $0 \leq k \leq 7$, selectează:

- din tabel, valoarea lui f din linia k ;
- cu multiplexorul, intrarea i_k de sus.

Așadar, multiplexorul implementează pe f dacă plasăm coloana valorilor lui f ca sistem de intrări i_0, \dots, i_7 ; încănd sistemu de valori $0, 0, 0$ ale variabilelor selectează valoarea din linia de sus a tabelului și intrarea i din stânga a multiplexorului, coloana valorilor lui f trebuie plasată orizontal, cu partea de sus spre stânga; notăm că variabila cea mai semnificativă, x , a fost desenată, conform convențiilor adoptate, în tabel în stânga și la multiplexor jos.

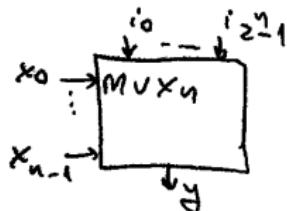
Multiplexor

O funcție booleană vectorială $f : B_2^n \rightarrow B_2^k$, $k > 1$, poate fi implementată printr-un sistem de k multiplexoare MUX_n , fiecare calculând câte una din componentele funcției, f_1, \dots, f_k ; liniile corespunzătoare variabilelor lui f intră simultan ca selector în toate multiplexoarele, iar acestea au ca sisteme de intrări i coloanele de valori ale componentelor lui f pe care le calculează:

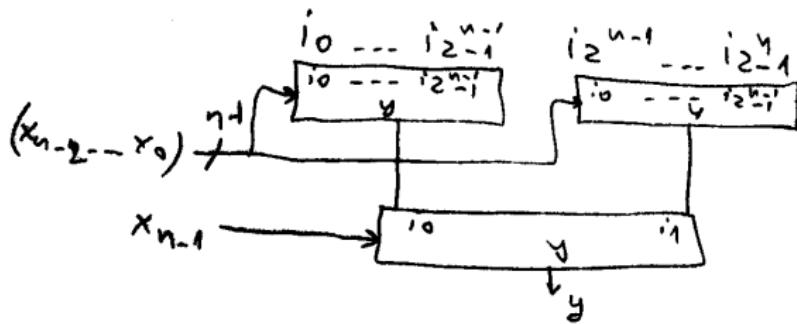


Multiplexor

MUX_n admite și o construcție recursivă; mai exact:



se poate construi ca:



Multiplexor

În interiorul dreptunghiurilor am notat intrările/ieșirile proprii ale multiplexoarelor (parametrii formali), iar lângă dreptunghiuri am notat liniile conectate la ele (parametri actuali).

De exemplu, intrarea i_{2^n-1} a lui MUX_n este conectată la intrarea i_0 a lui MUX_{n-1} din dreapta.

Multiplexor

Echivalența celor două circuite este ușor de înțeles dacă facem o comparație cu tablul de valori al funcției implementate; vom ilustra pentru funcția din exemplul anterior, presupunând că $(x, y, z) = (0, 1, 0)$:

$$(x, y, z) = (0, 1, 0)$$

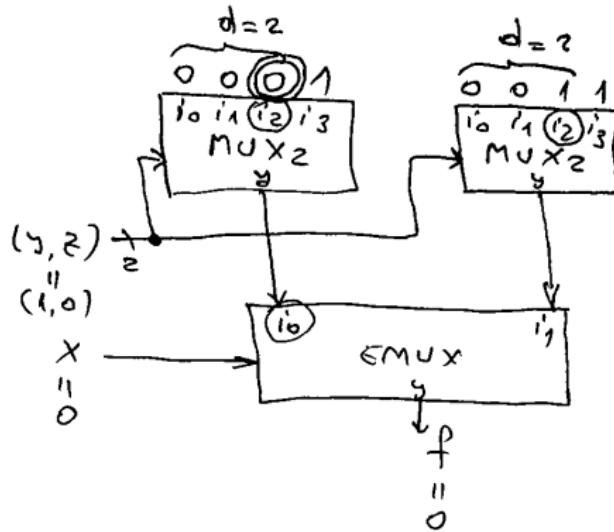
x	y	z	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

$x=0$

$d=2$

$x=1$

$d=2$



Multiplexor

Ieșirea y a lui MUX_3 este valoarea finală a lui f ; după valoarea variabilei celei mai semnificative, x , ea este aleasă din prima sau a doua jumătate a tabelului, respectiv din ceea ce furnizează primul sau al doilea MUX_2 ; întrucât $x = 0$, se alege din prima jumătate a tabelului, respectiv ce furnizează MUX_2 din stânga.

Sistemul valorilor celor mai puțin semnificative variabile, (y, z) , apare de două ori în tabel, o dată în prima jumătate, o dată în a doua jumătate, și în fiecare caz selectează valoarea lui f de pe o linie aflată la aceeași distanță d față de începutul jumătății; corepunzător, sistemul (y, z) intră ca selector în ambele MUX_2 și în fiecare caz selectează intrarea i cu același indice d ; întrucât $(y, z) = (1, 0)$, se alege din fiecare jumătate a tabelului valoarea lui f din linia 2 (numărând de la 0) și de la fiecare MUX_2 intrarea i_2 ($d = 2 = \overline{10}_2$).

Întrucât variabila cea mai semnificativă este $x = 0$, prima dintre cele două valori, adică 0, este emisă ca valoare finală.

Multiplexor

Proprietatea algebrică pe care se bazează construcția recursivă de mai sus este teorema care dă o descriere recursivă a unei funcții booleene scalare cu n variabile prin două funcții booleene scalare cu $n - 1$ variabile:

$$\forall x_{n-1}, \dots, x_0 \in B_2,$$

$$f(x_{n-1}, \dots, x_0) = \overline{x_{n-1}}f(0, x_{n-2}, \dots, x_0) + x_{n-1}f(1, x_{n-2}, \dots, x_0)$$

MUX_n construit recursiv implementează $f : B_2^n \rightarrow B_2$;

MUX_{n-1} din stânga (a cărui ieșire este selectată de $EMUX$ pentru $x_{n-1} = 0$) implementează $f_0 : B_2^{n-1} \rightarrow B_2$, $f_0(x_{n-2}, \dots, x_0) = f(0, x_{n-2}, \dots, x_0)$;

MUX_{n-1} din dreapta (a cărui ieșire este selectată de $EMUX$ pentru $x_{n-1} = 1$) implementează $f_1 : B_2^{n-1} \rightarrow B_2$, $f_1(x_{n-2}, \dots, x_0) = f(1, x_{n-2}, \dots, x_0)$.

Formula de recursie din teoremă este funcția implementată de un $EMUX$:

$$EMUX(s, a, b) = \bar{s}a + sb.$$

Multiplexor

Observație: și alte circuite 0 – DS, ca de exemplu DCD_n sau $DMUX_n$, admit o construcție recursivă (exercițiu).

Multiplexor

Putem continua construcția recursivă a lui MUX_n de mai devreme, exprimând fiecare MUX_{n-1} prin câte două MUX_{n-2} și un $EMUX$, ș.a.m.d., până obijnem un circuit, asemănător unui arbore, alcătuit doar din $EMUX$ -uri.

Astfel, întrucât orice funcție booleană scalară se poate implementa cu un multiplexor iar orice multiplexor se poate înlocui cu un arbore de $EMUX$ -uri, rezultă că orice funcție booleană scalară se poate implementa cu un circuit alcătuit doar din $EMUX$ -uri.

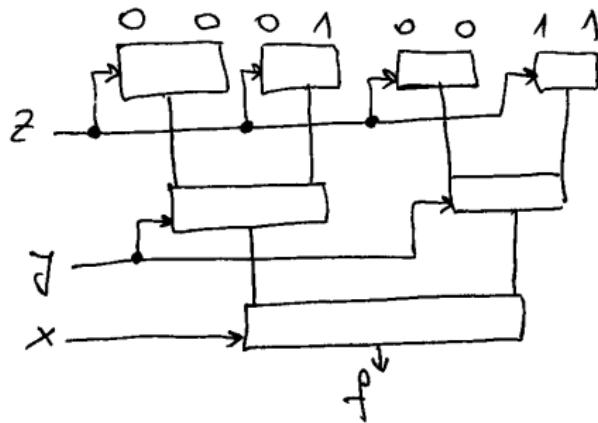
Deci, și $EMUX$ este circuit universal.

Observație: Alte circuite universale sunt $NAND$ și NOR (putem implementa orice funcție booleană folosind doar $NAND$ -uri sau doar NOR -uri) - exercițiu.

Multiplexor

Exemplu: Funcția din exemplul precedent se poate implementa cu *EMUX*-uri astfel:

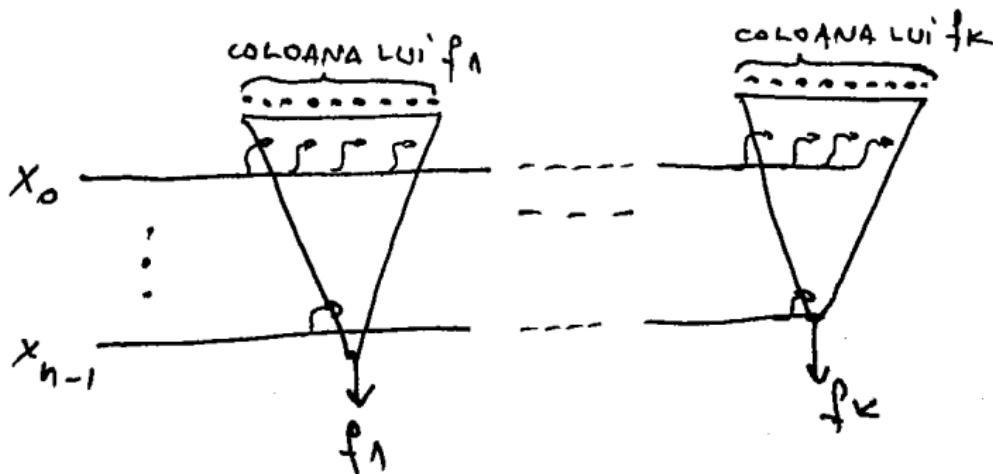
x	y	z	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



Dacă păstrăm convențiile anterioare privind poziția intrărilor și ieșirilor pe desenul *EMUX*-urilor, nu mai este nevoie să notăm aceste intrări și ieșiri.

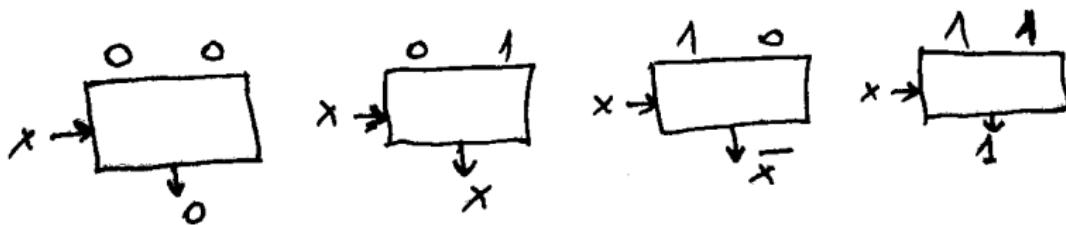
Multiplexor

O funcție booleană vectorială $f : B_2^n \rightarrow B_2^k$, $k > 1$, poate fi implementată printr-un sistem de k arbori de EMUX-uri, fiecare calculând câte una din componentele funcției, f_1, \dots, f_k ; fiecare linie corespunzătoare unei variabile a lui f intră ca selector în EMUX-urile de pe un același rând în toți arborii, iar aceștia au ca sisteme de intrări i coloanele de valori ale componentelor lui f pe care le calculează:



Multiplexor

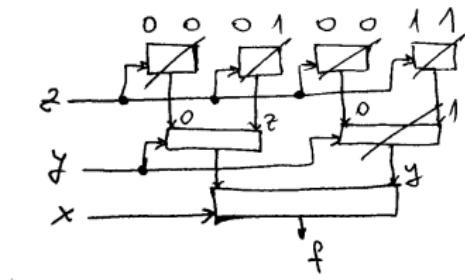
Numărul *EMUX*-urilor prin care se implementează o funcție booleană poate fi redus, dacă observăm că:



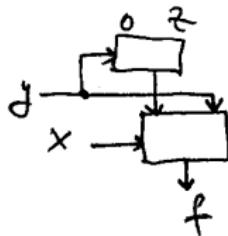
Astfel, dacă acceptăm și prezența porțiilor "NOT", *EMUX*-urile aflate pe primul rând al arborilor dispar, iar uneori dispar și *EMUX*-uri de pe rândurile următoare.

Multiplexor

Exemplu: Reduceți la maxim numărul de EMUX prin care se implementează funcția f din exemplele anterioare:



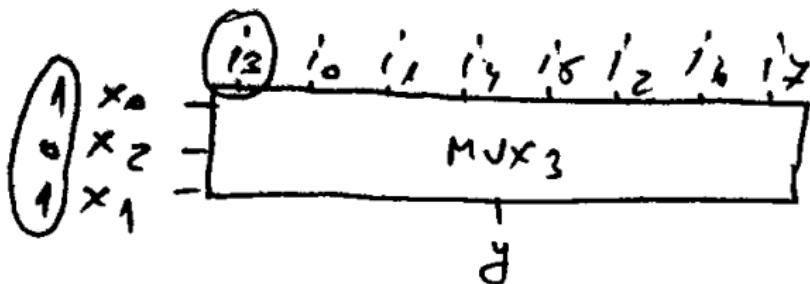
Desenul care păstrează doar EMUX-urile rămase este:



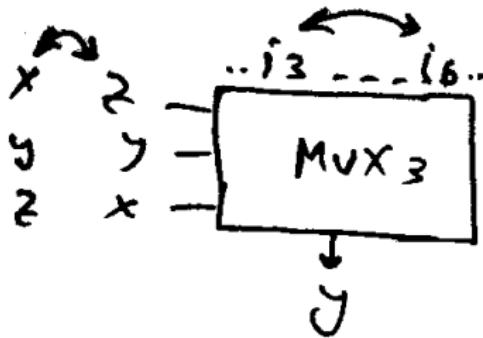
Sfat: Desenul circuitului redus este bine să fie realizat de jos în sus, deoarece avem o perspectivă mai clară care linii trebuie sterse sau redirecționate.

Observație:

În notațiile și simbolizările grafice folosite până acum, dacă dăm nume indexate variabilelor, intrărilor, ieșirilor, nu mai este nevoie să respectăm o anumită ordine de scriere/desenare a acestora. De exemplu, în cazul unui MUX_3 , vom ști că intrarea i_3 corespunde valorii selectorului $x_2 = 0$, $x_1 = 1$, $x_0 = 1$, indiferent unde sunt scrise/desenate aceste linii:



Dacă folosim nume neindexate, contează ordinea semnificațiilor variabilelor și cea de scriere/desenare în funcție de semnificație. De exemplu, dacă inversăm ordinea liniilor selectorului unui MUX_3 , intrarea i_3 se interschimbă cu i_6 :

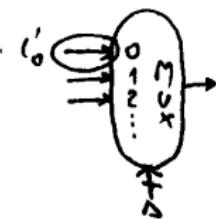
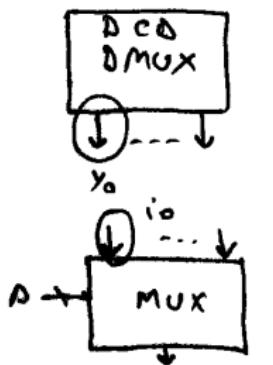
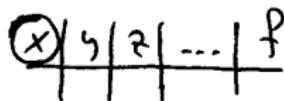


Oricare convenție de ordonare este bună, dar trebuie aleasă una și păstrată cu consecvență.

Convenția de ordonare folosită în acest curs este:

- variabila cea mai semnificativă este scrisă în stânga notației cu paranteze a funcției, în coloana din stânga a tabelului de valori și în partea de jos a selectorului blocurilor decodicator, codicator, demultiplexor, multiplexor;
- ieșirea y corespunzătoare valorii $0 = (0, \dots, 0)$ a selectorului blocurilor decodicator și demultiplexor este scrisă în stânga sau sus;
- intrarea i corespunzătoare valorii $0 = (0, \dots, 0)$ a selectorului blocului multiplexor este scrisă în stânga sau sus.

$$f(x, y, z)$$



În cele ce urmează, până la sfârșitul secțiunii referitoare la $0 - DS$, vom nota:
 $\vee, \wedge, \neg, \oplus, \setminus$, operațiile de algebră booleană "OR", "AND", "NOT",
"XOR", diferență

și vom nota:

$+, -, \cdot$ sau juxtapunere, div , mod , operațiile aritmetice pe numere întregi
de adunare, scădere, înmulțire, aflarea câtului întreg, aflarea restului întreg.

Sumator:

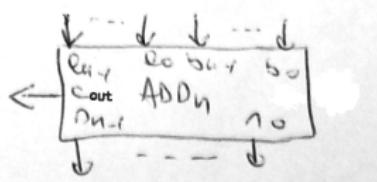
Un **sumator** pe n biți ADD_n , $n \geq 1$, este un circuit care implementează operația pe care la începutul secțiunii "Reprezentarea numerelor în calculator" am notat-o cu \oplus .

Mai exact, el primește ca intrare două siruri de biți a_{n-1}, \dots, a_0 , b_{n-1}, \dots, b_0 și eventual un transport de intrare c_{in} (pentru poziția 0) și le aplică algoritmul de adunare cu reprezentările binare ale numerelor naturale folosit în matematică; transportul din poziția $n - 1$ se pierde (de fapt, el este emis pe o linie separată).

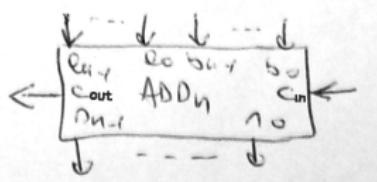
Am văzut că rezultatul acestei operații are semnificația de sumă atât în cazul când sirurile de biți reprezintă numere naturale ca întregi fără semn, cât și în cazul când ele reprezintă numere întregi în complement față de 2.

Sumator

Simbol:



sau, dacă considerăm și un transport de intrare:



Vom defini mai întâi sumatorii pe un bit, apoi, pe baza acestora, sumatorii pe mai mulți biți.

- **Sumator pe un bit:**

Sumatorii pe un bit pot fi folosiți:

- pentru adunarea biților cei mai puțin semnificativi (biții de rang 0, ai unităților) ai unor operanzi mai lungi și atunci nu avem transport de intrare (carry in), dar avem transport de ieșire (carry out); circuitul respectiv s.n. **half adder**;
- pentru adunarea biților mai semnificativi (de rang ≥ 1) ai unor operanzi mai lungi și atunci avem și transport de intrare (carry in) și transport de ieșire (carry out); circuitul respectiv s.n. **full adder**.

Sumator

- **Half adder (HA):**

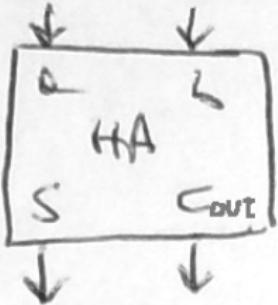
Primește ca intrare: a, b (doi operanzi de un bit).

Efectuează adunarea: $a + b$.

Furnizează ca ieșire: s (bitul sumă, trimis în locația destinație), c_{out} (carry out).

Observăm că: $s = (a + b) \text{ mod } 2$, $c_{out} = (a + b) \text{ div } 2$.

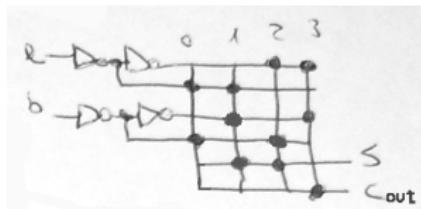
Simbol și tabel de adevăr:



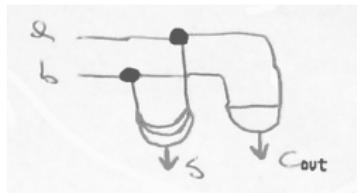
	a	b	s	c_{out}
0	0	0	0	0
1	0	1	1	0
2	1	0	1	0
3	1	1	0	1

Sumator

Construcția ca PROM:



O construcție cu mai puține porți se poate obține observând din tabel că:
 $s = a \oplus b$, $c_{out} = a \wedge b$:



Half adder-ul implementează funcția booleană
 $HA : B_2^2 \longrightarrow B_2^2$, $HA(a, b) = (HA_s(a, b), HA_c(a, b))$, unde
 $HA_s(a, b) = s = a \oplus b$, $HA_c(a, b) = c_{out} = a \wedge b$.

Sumator

- Full adder (FA):

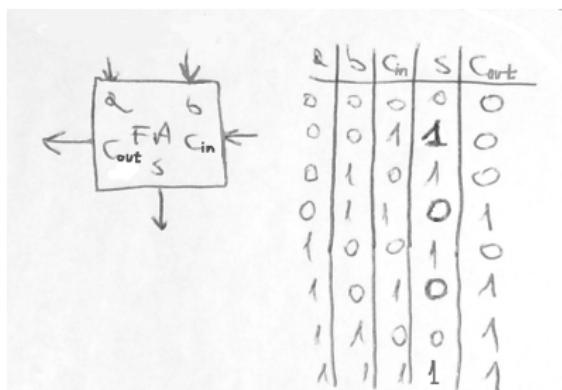
Primește ca intrare: a, b (doi operanzi de un bit), c_{in} (carry in).

Efectuează adunarea: $a + b + c_{in}$.

Furnizează ca ieșire: s (bitul sumă, trimis în locația destinație), c_{out} (carry out).

Observăm că: $s = (a + b + c_{in}) \text{ mod } 2$, $c_{out} = (a + b + c_{in}) \text{ div } 2$.

Simbol și tabel de adevăr:



Full adder-ul implementează funcția booleană

$$FA : B_2^3 \longrightarrow B_2^2, FA(a, b, c) = (FA_s(a, b, c), FA_c(a, b, c)), \text{ unde}$$

$$FA_s(a, b, c) = s = (a + b + c) \text{ mod } 2, FA_c(a, b, c) = c_{out} = (a + b + c) \text{ div } 2.$$

Sumator

Dacă din tabelul anterior extragem FND, rezultă că pentru $a, b, c = c_{in}$ date, avem:

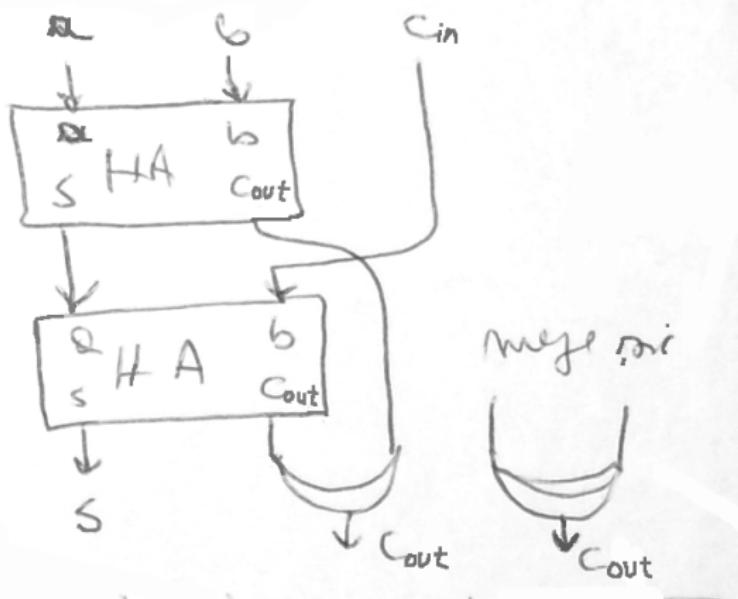
$$\begin{aligned} FA_s(a, b, c) &= (\bar{a} \wedge \bar{b} \wedge c) \vee (\bar{a} \wedge b \wedge \bar{c}) \vee (a \wedge \bar{b} \wedge \bar{c}) \vee (a \wedge b \wedge c) = \\ &= (((\bar{a} \wedge b) \vee (a \wedge \bar{b})) \wedge \bar{c}) \vee (((\bar{a} \wedge \bar{b}) \vee (a \wedge b)) \wedge c) = \\ &= ((a \oplus b) \wedge \bar{c}) \vee (\overline{a \oplus b} \wedge c) = (a \oplus b) \oplus c = HA_s(HA_s(a, b), c_{in}) \end{aligned}$$

$$\begin{aligned} FA_c(a, b, c) &= (\bar{a} \wedge b \wedge c) \vee (a \wedge \bar{b} \wedge c) \vee (a \wedge b \wedge \bar{c}) \vee (a \wedge b \wedge c) = \\ &= (((\bar{a} \wedge b) \vee (a \wedge \bar{b})) \wedge c) \vee (a \wedge b \wedge (\bar{c} \vee c)) = ((a \oplus b) \wedge c) \vee (a \wedge b) = \\ &= HA_c(HA_s(a, b), c_{in}) \vee HA_c(a, b) = HA_c(HA_s(a, b), c_{in}) \oplus HA_c(a, b) \end{aligned}$$

Ultima egalitate de jos se bazează pe faptul că $HA_c(HA_s(a, b), c_{in})$ și $HA_c(a, b)$ nu pot fi simultan 1 (de aceea, în loc de \vee putem folosi \oplus); într-adevăr, conform tabelului lui half adder, dacă $HA_c(a, b) = 1$, atunci $HA_s(a, b) = 0$, deci $HA_c(HA_s(a, b), c_{in}) = HA_c(0, c_{in}) = 0$.

Sumator

Atunci putem construi un circuit full adder folosind două circuite half adder și un "OR" sau "XOR", astfel:



Sumator

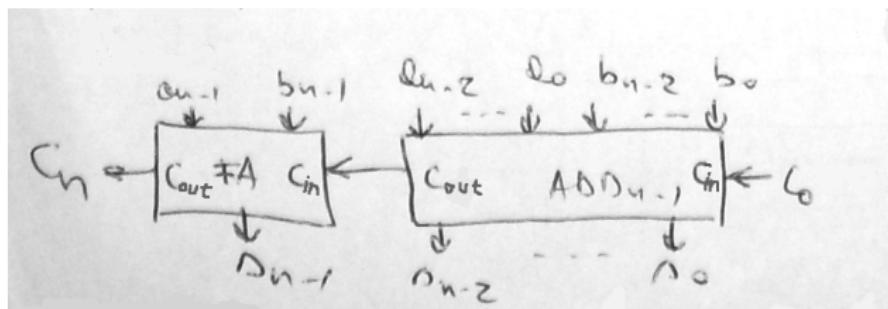
- **Sumator serial:**

Sumatorul serial pe n biți ADD_n , calculează biții sumei succesiv, de la cel de rang minim la cel de rang maxim, folosind la calculul fiecărui nou bit transportul (carry out) obținut la bitul anterior.

Ei se poate defini (construi) recursiv, astfel:

ADD_1 este un FA

ADD_n , $n > 1$, este extensia serială a unui ADD_{n-1} cu un FA:



Dezavantaj: dezvoltarea pe verticală a circuitului (numărul de niveluri de legare în serie) este mare și depinde de n ; astfel, circuitul este lent.

Sumator

- **Sumator paralel:**

Sumatorul paralel pe n biți ADD_n , este o variantă mai rapidă, care efectuează însumarea separată a transportului fiecărei sume binare, folosind un circuit CL (**carry lookahead**).

Dacă pentru orice $0 \leq i \leq n$ notăm:

a_i, b_i, s_i biții de pe poziția i ai operanzilor și, respectiv, sumei,

c_i carry in la poziția i (și carry out la poziția $i - 1$, dacă $i \geq 1$),

iar c_n este carry out final (de pe poziția $n - 1$),

atunci, din formulele obținute pentru FA_s , FA_c mai devreme, rezultă că pentru orice $0 \leq i \leq n$, avem:

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

unde am notat:

$G_i = a_i \wedge b_i$, deplasarea generată la poziția i ,

$P_i = a_i \oplus b_i$, deplasarea propagată la poziția i .

Având în vedere că $a_i \wedge b_i$ și $a_i \oplus b_i$ nu pot fi simultan 1 (se poate vedea, de exemplu, folosind tabelele de valori), în formula lui c_{i+1} de mai sus în loc de \vee se poate folosi \oplus .

Sumator

Iterând formula anterioară, rezultă că pentru orice $0 \leq i \leq n$ avem:

$$c_{i+1} = G_i \vee (P_i \wedge G_{i-1}) \vee (P_i \wedge P_{i-1} \wedge G_{i-2}) \vee \cdots \vee (P_i \wedge P_{i-1} \wedge \cdots \wedge P_0 \wedge c_0)$$

și, ca mai înainte, în această formulă în loc de \vee se poate folosi \oplus .

Întrucât fiecare G_i , P_i depinde de a_i , b_i printr-o formulă cu un nivel (o operație \wedge , respectiv \oplus), iar fiecare c_{i+1} depinde de sistemul de G_i -uri, P_i -uri și c_0 printr-o formulă cu două niveluri (o disjuncție \vee de conjuncții \wedge), rezultă că fiecare c_{i+1} depinde de sistemul de a_i -uri, b_i -uri și c_0 printr-o formulă cu trei niveluri, deci se poate calcula din sistemul de a_i -uri, b_i -uri și c_0 în timp constant (care nu mai depinde de n).

Notăm că pentru orice $0 \leq i \leq n$, cel mult unul din termenii disjuncției prin care se calculează c_{i+1} poate fi 1.

Într-adevăr, dacă toți G_i, \dots, G_0 sunt 0, atunci $c_{i+1} = P_i \wedge P_{i-1} \wedge \cdots \wedge P_0 \wedge c_0$. Altfel, fie $0 \leq k \leq i$ cel mai mare indice a.î. $G_k = 1$; atunci, pentru $j > k$ avem $P_i \wedge P_{i-1} \wedge \cdots \wedge P_{j+1} \wedge G_j = 0$, deoarece $G_j = 0$, iar pentru $j < k$ avem $P_i \wedge P_{i-1} \wedge \cdots \wedge P_{j+1} \wedge G_j = 0$, deoarece termenul îl conține pe P_k , care este 0 (G_k și P_k nu pot fi simultan 1); deci, $c_{i+1} = P_i \wedge P_{i-1} \wedge \cdots \wedge P_{k+1} \wedge G_k$.

Sumator

Formula de calcul a lui c_{i+1} , $0 \leq i \leq n$, de mai sus are asociate următoarele observații intuitive:

- la o poziție $0 \leq i \leq n$ se generează carry out d.d. $a_i = b_i = 1$, i.e. $G_i = 1$; în acest caz, suma rămasă pentru poziția i este 0, iar $P_i = 0$;
- carry out-ul generat la poziția i se propagă spre pozițiile j aflate tot mai la stânga (valori j tot mai mari) atât timp cât la pozițiile respective s-au adunat $a_j = 0$ și $b_j = 1$ sau $a_j = 1$ și $b_j = 0$ (pentru ca suma rămasă pentru poziția j să fie 1), adică atât timp cât $P_j = 1$; notăm că pentru aceste poziții avem $G_j = 0$;
- dacă la stânga lui i avem doar valori $P_j = 1$ (va rezulta $G_j = 0$), carry-ul generat la poziția i va fi carry out final c_n ;
- altfel, considerăm cea mai mică poziție $k > i$ a.î. $P_k = 0$; atunci la poziția k s-au adunat $a_k = 0$ și $b_k = 0$ sau $a_k = 1$ și $b_k = 1$, deci suma rămasă pentru poziția k este 0 și atunci carry-ul generat la poziția i se oprește în acest 0 (nu se propagă mai departe); nu este obligatoriu ca $G_k = 1$ (avem aşa doar dacă la poziția k s-au adunat $a_k = 1$ și $b_k = 1$), deci k este \leq următoarea poziție $i' > i$ unde se generează carry out;

Sumator

- astfel, cantitatea totală transportată printr-o poziție $0 \leq i \leq n$ nu va fi niciodată > 1 (carry out-urile nu ajung să se cumuleze);
- un carry out final $c_{n+1} = 1$ poate proveni:
 - fie de la un $c_0 = 1$, dacă toate P_i , $0 \leq i \leq n$, sunt 1 (atunci toate G_i , $0 \leq i \leq n$, sunt 0, deci nici o poziție nu generează carry out, iar numerele de n biți adunate sunt unul complementul față de unu al celuilalt);
 - fie de la cea mai mare poziție $0 \leq i \leq n$ unde s-a generat carry out, adică $G_i = 1$, dacă pentru pozițiile $j > i$ avem $P_j = 1$ (adică se permite propagarea); eventualele carry out venite din spate se vor opri cel mult în poziția k .

A handwritten diagram illustrating a binary addition process. Two binary numbers are shown above a horizontal line:

1	1	0	1	(1)	1	0	0	0	1	(1)
0	1	1	0	(1)	0	1	0	1	0	(1)

The sum below the line is:

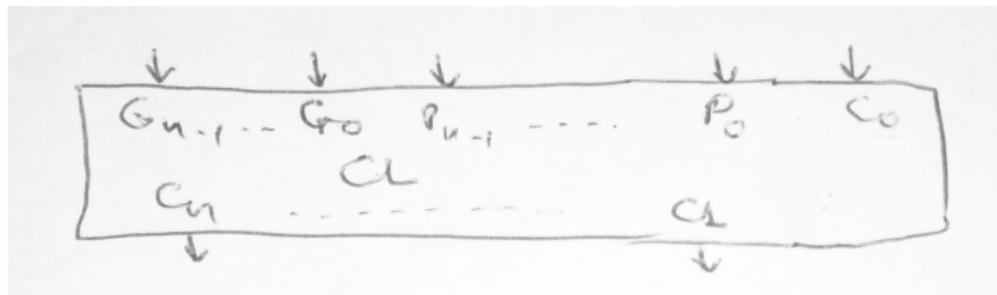
1	0	1	1	0	1	1	0	1	1	0
---	---	---	---	---	---	---	---	---	---	---

Annotations include circled '1's at positions 4 and 10, circled '0's at positions 5 and 9, and circled 'G' at position 9. Arrows labeled 'G' point from the circled 'G' to the circled '0' at position 9 and to the circled '1' at position 4. An arrow labeled 'cin' points to the circled '0' at position 9. An arrow labeled 'Cout' points to the circled '1' at position 4.

Sumator

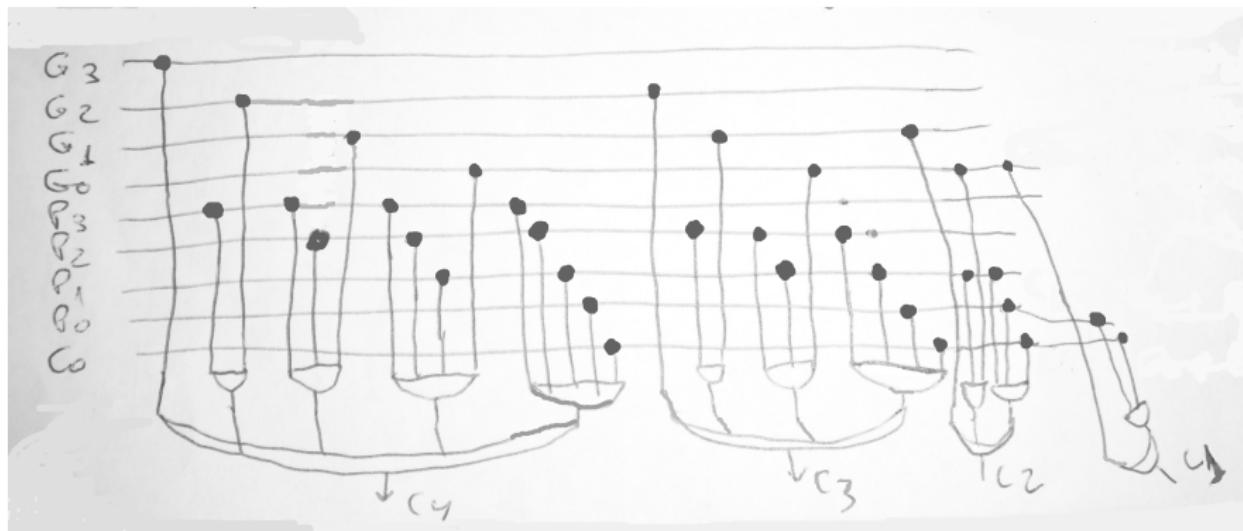
Circuitul CL implementează formula anterioară de calcul pentru toate carry out-urile c_i , $1 \leq i \leq n$, și are două niveluri (un nivel \wedge și un nivel \vee), deci numărul de niveluri și timpul de calcul nu depind de n .

Simbol:



Sumator

Construcție pentru $n = 4$:



Circuitul implementează formulele:

$$c_4 = G_3 \vee (P_3 \wedge G_2) \vee (P_3 \wedge P_2 \wedge G_1) \vee (P_3 \wedge P_2 \wedge P_1 \wedge G_0) \vee (P_3 \wedge P_2 \wedge P_1 \wedge P_0 \wedge c_0)$$

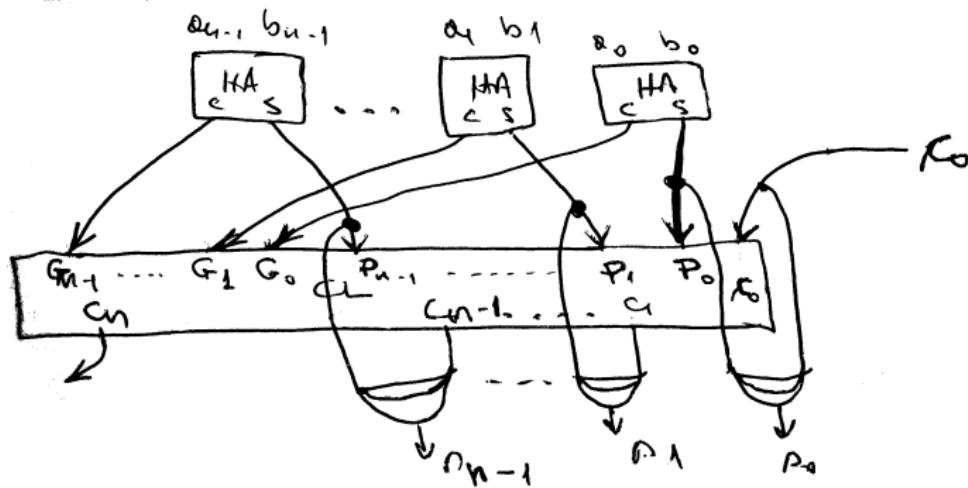
$$c_3 = G_2 \vee (P_2 \wedge G_1) \vee (P_2 \wedge P_1 \wedge G_0) \vee (P_2 \wedge P_1 \wedge P_0 \wedge c_0)$$

$$c_2 = G_1 \vee (P_1 \wedge G_0) \vee (P_1 \wedge P_0 \wedge c_0)$$

$$c_1 = G_0 \vee (P_0 \wedge c_0)$$

Sumator

Construcția sumatorului:



Într-adevăr, formulele anterioare ne arată că pentru orice $0 \leq i \leq n - 1$ avem:

$$s_i = FA_s(a_i, b_i, c_i) = HA_s(HA_s(a_i, b_i), c_i) = HA_s(a_i, b_i) \oplus c_i$$

$$G_i = a_i \wedge b_i = HA_c(a_i, b_i)$$

$$P_i = a_i \oplus b_i = HA_s(a_i, b_i)$$

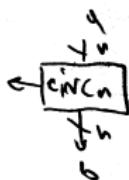
Circuitul are 4 niveluri de legare în serie (deoarece HA are un nivel iar CL două niveluri), deci numărul de niveluri și timpul de calcul nu depind de n .

Circuit pentru incrementare

Circuit pentru incrementare:

Un **circuit pentru incrementare** pe n biți INC_n , $n \geq 1$ este un circuit care adună aritmetic 1 la un număr natural/întreg reprezentat binar pe n biți, transportul din bitul cel mai semnificativ fiind emis pe o linie separată; cu alte cuvinte, efectuează $\oplus 1$, în sensul operației \oplus definită la începutul secțiunii "Reprezentarea numerelor în calculator".

Simbol:



Variante de construcție:

- Construcție serială - exercițiu:

$$n = 1 : NOT$$

$n \rightarrow n + 1 : INC_n$ extins serial cu HA

- ADD_n cu intrarea b fixată pe $\underbrace{0 \dots 0}_n 1$;

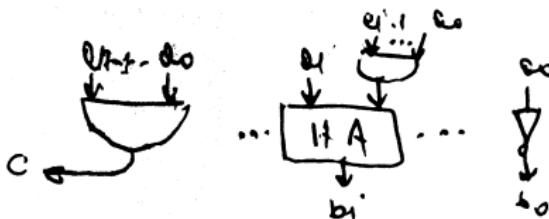
Circuit pentru incrementare

- Construcție paralelă (cea mai performantă):

Se bazează pe următoarele observații:

- la ADD_n , dacă fixăm $b_i = 0$, $0 \leq i \leq n - 1$, și $c_0 = 1$, obținem INC_n ;
- la CL , dacă fixăm $b_i = 0$, $0 \leq i \leq n - 1$, și $c_0 = 1$, atunci pentru orice $0 \leq i \leq n - 1$ avem $G_i = a_i \wedge 0 = 0$, $P_i = a_i \oplus 0 = a_i$, deci $c_{i+1} = G_i \vee (P_i \wedge c_i) = a_i \wedge c_i = \dots = a_i \wedge \dots \wedge a_0 \wedge c_0 = a_i \wedge \dots \wedge a_0$.

Atunci obținem circuitul:



Intuitiv: la fiecare poziție i se adună a_i cu carry venit din urmă $a_{i-1} \wedge \dots \wedge a_0$ și se obține un b_i și un carry out ce nu mai trebuie calculat aici, deoarece se recalculează la poziția $i + 1$.

Circuitul are doar două niveluri de legare în serie, deci este rapid (iar timpul de calcul nu depinde de n); această fapt este important, deoarece programele efectuează în general multe incrementări (ex: " $++i$ " într-o instrucțiune "for") și este important ca ele să se execute repede.

Circuit pentru scădere

Circuit pentru scădere:

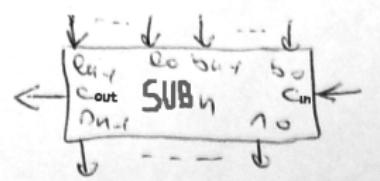
Un **circuit pentru scădere** pe n biți SUB_n , $n \geq 1$, este un circuit care implementează operația pe care la începutul secțiunii "Reprezentarea numerelor în calculator" am notat-o cu \ominus .

Mai exact, el primește ca intrare două siruri de biți a_{n-1}, \dots, a_0 , b_{n-1}, \dots, b_0 și un împrumut de intrare c_{in} și le aplică algoritmul de scădere cu reprezentările binare ale numerelor naturale folosit în matematică, obținând un sir de biți ai rezultatului (diferenței) d_{n-1}, \dots, d_0 și un împrumut de ieșire c_{out} , care este emis pe o linie separată.

Am văzut că rezultatul acestei operații are semnificația de diferență atât în cazul când sirurile de biți reprezintă numere naturale ca întregi fără semn, cât și în cazul când ele reprezintă numere întregi în complement față de 2.

Circuit pentru scădere

Simbol:



Variante de construcție:

- Construcție serială - exercițiu:

$n = 1$: se defiește un circuit FS (**full subtract**), în aceeași manieră ca FA;
 $n \rightarrow n + 1$: se extinde serial SUB_n cu un FS
(și se mai pot face niște simplificări).

- Construcția bazată pe ADD_n :

Se bazează pe observația că dacă a, b sunt numere naturale/întregi pe n biți, c este numărul 0 sau 1 pe un bit, iar $\bar{a}, \bar{b}, \bar{c}$ sunt numerele obținute negând biții din reprezentările lui a, b , respectiv c (complementele față de 1), atunci:

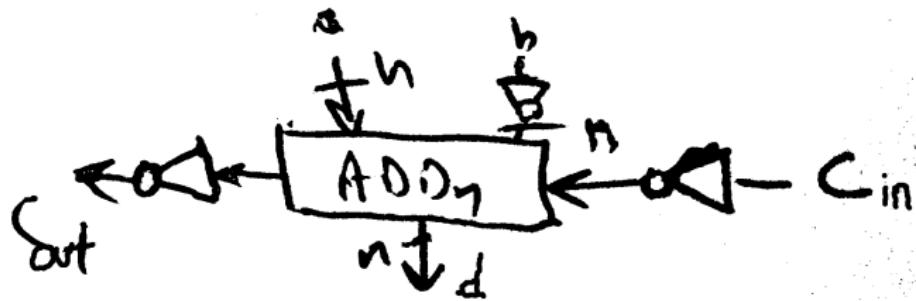
$$b + \bar{b} = 2^n - 1, c + \bar{c} = 1,$$

deci

$$a - b - c = a + \bar{b} - 2^n + 1 + \bar{c} - 1 = a + \bar{b} + \bar{c} - 2^n.$$

Circuit pentru scădere

Rezultă circuitul:



Așadar, scăderea se reduce la adunare (se poate efectua cu circuitul pentru adunare).

Multiplicator

Multiplicator:

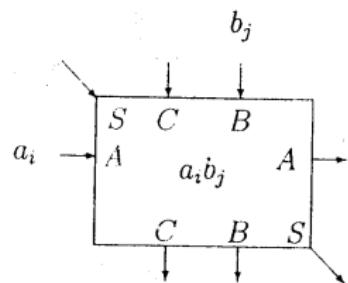
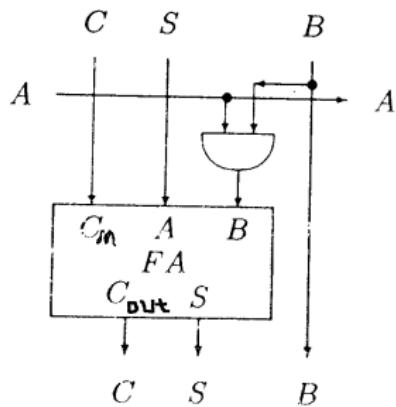
Un **multiplicator** pe n biți MUL_n , $n \geq 1$, este un circuit care primește ca intrare două siruri de biți a_{n-1}, \dots, a_0 , b_{n-1}, \dots, b_0 și le aplică algoritmul de înmulțire cu reprezentările binare ale numerelor naturale folosit în matematică, obținând un sir de biți ai rezultatului (produsului) x_{2n-1}, \dots, x_0 .

Cu un asemenea circuit se poate efectua înmulțirea unor numere naturale/întregi.

Prezentăm mai jos varianta de construcție descrisă în cartea:
Adrian Atanasiu: "Arhitectura calculatorului", Ed. InfoData, 2006:

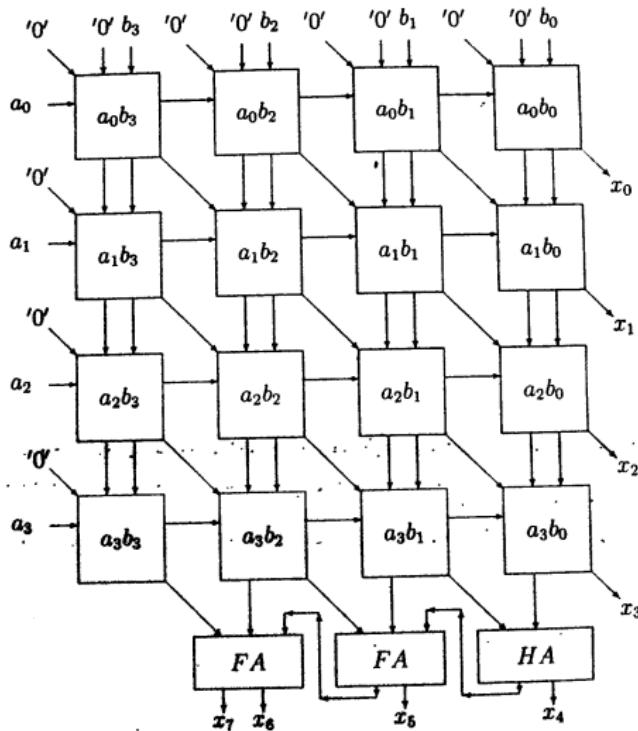
Multiplicator

Se folosește un bloc pentru înmulțirea pe un bit (cu ajutorul unei porți "AND") și adunarea produselor și carry-urilor vecine (cu ajutorul unui *FA*); construcția blocului (stânga) și simbolul său (dreapta) sunt următoarele:



Multiplicator

Construcția multiplicatorului pe n biți este ilustrată mai jos, pentru $n = 4$:



Multiplicator

Dezavantaj: dezvoltarea pe verticală a circuitului (numărul de niveluri de legare în serie) este mare și depinde de n ; astfel, circuitul este lent.

De aceea, pentru înmulțirea hardware nu se folosește un $0 - DS$, care efectuează calculul într-un singur pas foarte lent, ci un $2 - DS$, care efectuează calculul prin adunări și shiftări repetate, pe parcursul mai multor pași rapizi (vom vedea mai târziu).

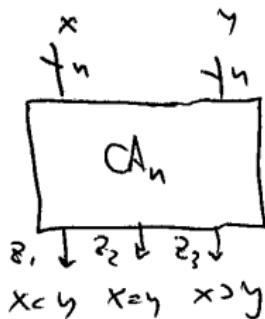
Comparator

Comparator:

Un **comparator** pe n biți CA_n , $n \geq 1$, este un circuit care efectuează comparația aritmetică a două numere întregi.

El primește ca intrare reprezentările celor două numere a și b în complement față de 2 pe n biți, a_{n-1}, \dots, a_0 și respectiv b_{n-1}, \dots, b_0 , și furnizează trei ieșiri de un bit, corespunzătoare celor trei relații posibile în care se pot afla a și b : z_1 ($a < b$), z_2 ($a = b$), z_3 ($a > b$) (fiecare ieșire este 1 sau 0 după cum relația respectivă are sau nu loc).

Simbol:



Comparator

Notând ca mai înainte \bar{a} , \bar{b} numerele obținute prin negarea bițiilor din reprezentările lui a , respectiv b (complementele față de 1), vom avea:

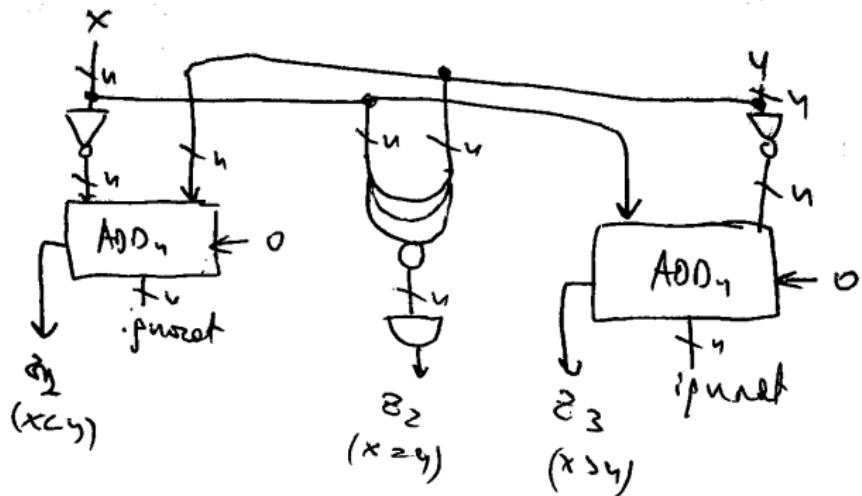
$$\begin{aligned} a > b &\Leftrightarrow a - b > 0 \Leftrightarrow a + \bar{b} + 1 - 2^n > 0 \Leftrightarrow a + \bar{b} > 2^n - 1 \\ &\Leftrightarrow a + \bar{b} \text{ are carry out} = 1. \end{aligned}$$

Similar, $a < b \Leftrightarrow \bar{a} + b \text{ are carry out} = 1$.

$$\begin{aligned} \text{În fine, } a = b &\Leftrightarrow \forall i \in \{0, \dots, n-1\}, a_i = b_i \\ &\Leftrightarrow \forall i \in \{0, \dots, n-1\}, \overline{a_i \oplus b_i} = 1 \quad (\overline{\cdot \oplus \cdot} \text{ este operația "NXOR"}) \\ &\Leftrightarrow \overline{a_0 \oplus b_0} \wedge \dots \wedge \overline{a_{n-1} \oplus b_{n-1}} = 1. \end{aligned}$$

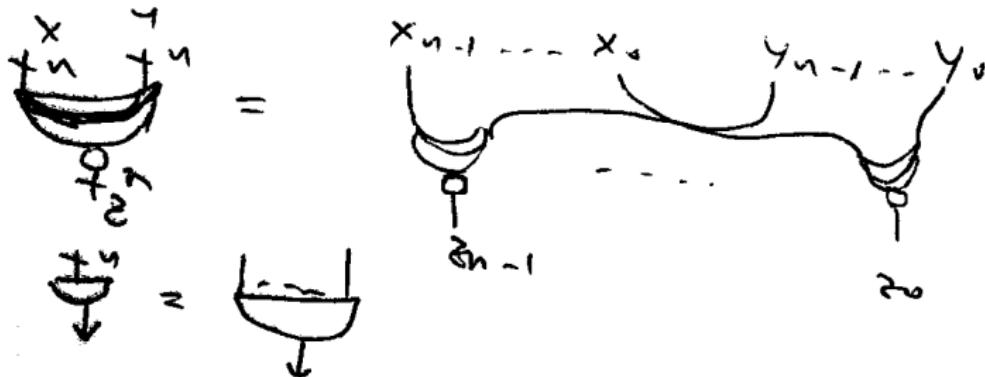
Comparator

Atunci, putem construi comparatorul astfel:



Comparator

unde am folosit următoarele simboluri:



Comparator

O variantă mai simplă, cu un singur sumator, se poate obține observând că:

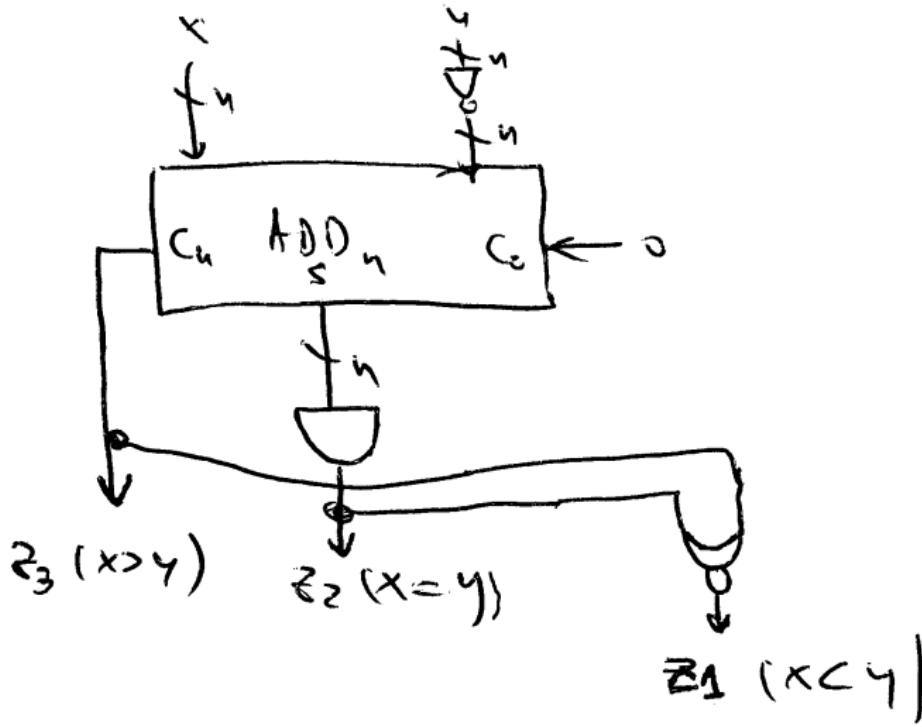
$a > b \Leftrightarrow a + \bar{b}$ are carry out = 1 (am văzut).

$a = b \Leftrightarrow a - b = 0 \Leftrightarrow a + \bar{b} + 1 - 2^n = 0 \Leftrightarrow a + \bar{b} = 2^n - 1$
 $\Leftrightarrow a + \bar{b}$ are toți biții 1 \Leftrightarrow conjuncția \wedge a biților lui $a + \bar{b}$ este 1.

$a < b \Leftrightarrow$ celălalt caz \Leftrightarrow "NOR"-ul (adică "nici"-ul) celorlalte două cazuri.

Comparator

Obținem circuitul:



Circuit pentru deplasare (shifter):

Un **circuit pentru deplasare (shifter)** pe n biți, $n \geq 1$, este un circuit care primește ca intrare un sir de n biți a_{n-1}, \dots, a_0 și un număr natural k prin reprezentarea sa în calculator ca întreg fără semn și furnizează ca ieșire sirul deplasat la stânga sau la dreapta cu k poziții. Biții care ies din sistemul de n se pierd (eventual sunt recuperati pe alte linii), iar locurile goale apărute în partea opusă se completează după o anumită regulă, în funcție de tipul de shiftare.

Există trei tipuri de shiftare:

- **Shiftare logică la stânga** (instrucțiunea "sll" din limbajul MIPS):

Furnizează sirul: $a_{n-k-1}, a_{n-k-2}, \dots, a_0, \underbrace{0, \dots, 0}_{k \text{ biți}}$

- **Shiftare logică la dreapta** (instrucțiunea "srl" din limbajul MIPS):

Furnizează sirul: $\underbrace{0, \dots, 0}_{k \text{ biți}}, a_{n-1}, a_{n-2}, \dots, a_k$

- **Shiftare aritmetică la dreapta** (instrucțiunea "sra" din limbajul MIPS):

Furnizează sirul: $\underbrace{a_{n-1}, \dots, a_{n-1}}_{k \text{ biți}}, a_{n-1}, a_{n-2}, \dots, a_k$

Dacă a_{n-1}, \dots, a_0 este reprezentarea în calculator a unui număr natural a ca întreg fără semn sau a unui număr întreg a în complement față de 2, atunci:

- shiftarea logică la stânga cu k furnizează reprezentarea lui $a \times 2^k$;
- shiftarea logică la dreapta cu k furnizează reprezentarea lui $\lfloor a/2^k \rfloor$, doar pentru numere $a \geq 0$;
- shiftarea aritmetică la dreapta cu k furnizează reprezentarea lui $\lfloor a/2^k \rfloor$, pentru orice număr întreg a reprezentat în complement față de 2 (ea completează locurile goale apărute în stânga cu copii ale bitului de semn).

În toate cazurile, proprietatea are loc cu condiția ca rezultatul să se afle în mulțimea valorilor reprezentabile (să nu apară overflow).

Deci, shiftările pot fi folosite pentru a efectua rapid înmulțiri și împărțiri ale unor numere întregi cu puteri ale lui 2.

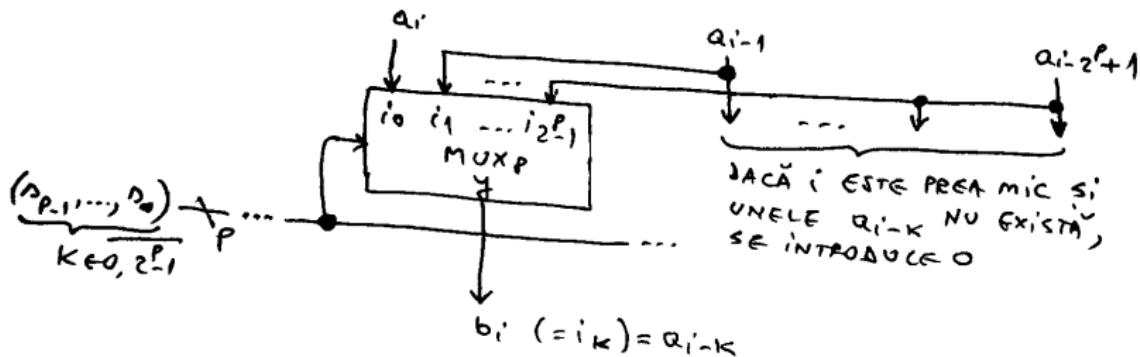
Pentru construcția circuitului, presupunem $0 \leq k \leq 2^p - 1$; ca să reprezentăm valorile k ca întregi fără semn, avem nevoie de p biți.

Atunci, pentru fiecare bit destinație b_i , $0 \leq i \leq n - 1$, construim un MUX_p care, în funcție de numărul de poziții k de shiftare (k este valoarea de selecție a multiplexorului), alege dintre a_i, \dots, a_{i-2^p+1} (în cazul shiftării la stânga) sau dintre a_{i+2^p-1}, \dots, a_i (în cazul shiftării la dreapta), anume îl alege pe $a_{i \pm k}$. Intrările multiplexoarelor care corespund unor a_j inexistenți vor primi 0 (în cazul shiftărilor logice) sau a_{n-1} (în cazul shiftării aritmetice la dreapta).

Shifter

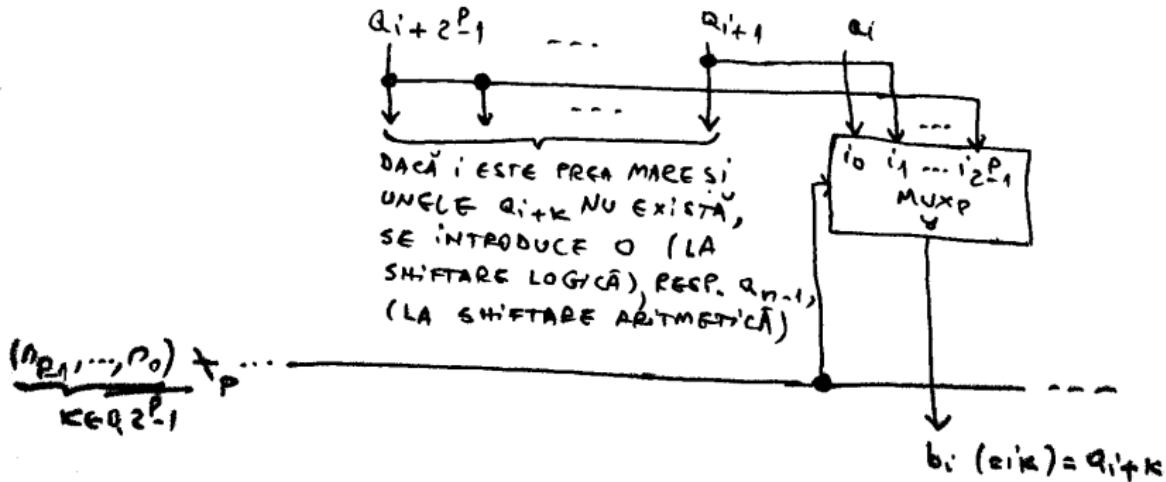
Dacă desenăm generic această asociere evidențiind intrările legate la o ieșire b_i , obținem:

- pentru shiftarea la stânga:



Shifter

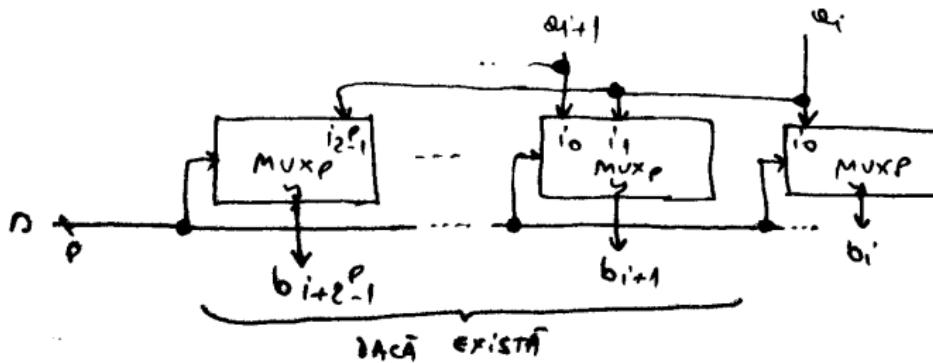
- pentru shiftarea la dreapta:



Shifter

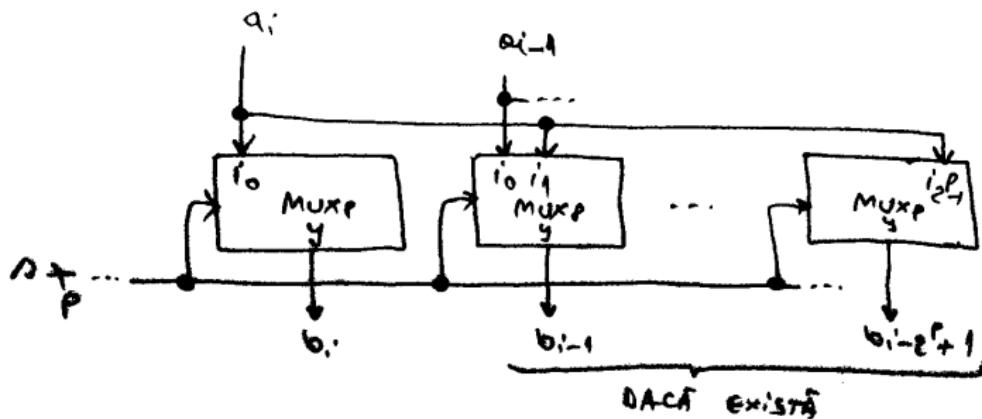
Dacă desenăm generic această asociere evidențiind ieșirile la care este legată o intrare a_i , obținem:

- pentru shiftarea la stânga:



Shifter

- pentru shiftarea la dreapta:

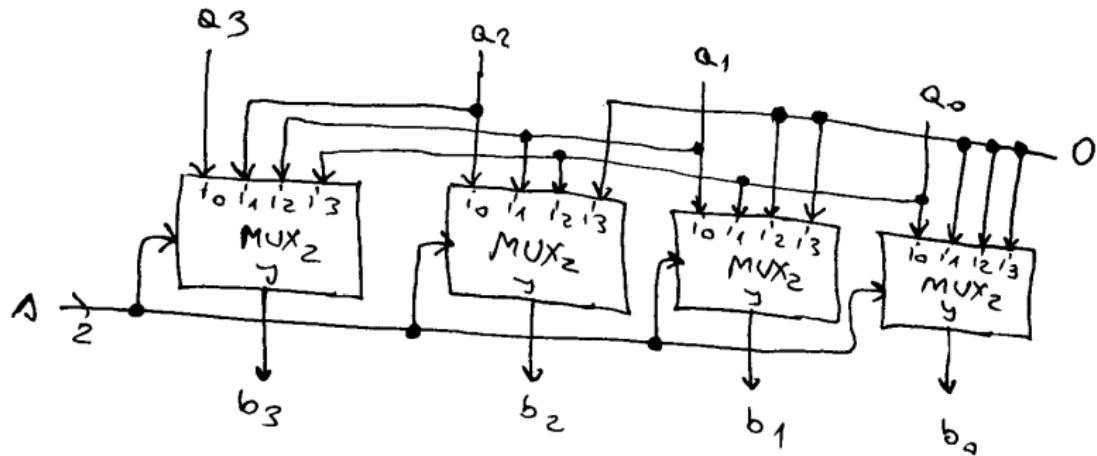


Regulă de ținut minte: fiecare a_i se ramifică de 2^P ori spre direcția de shiftare.

Shifter

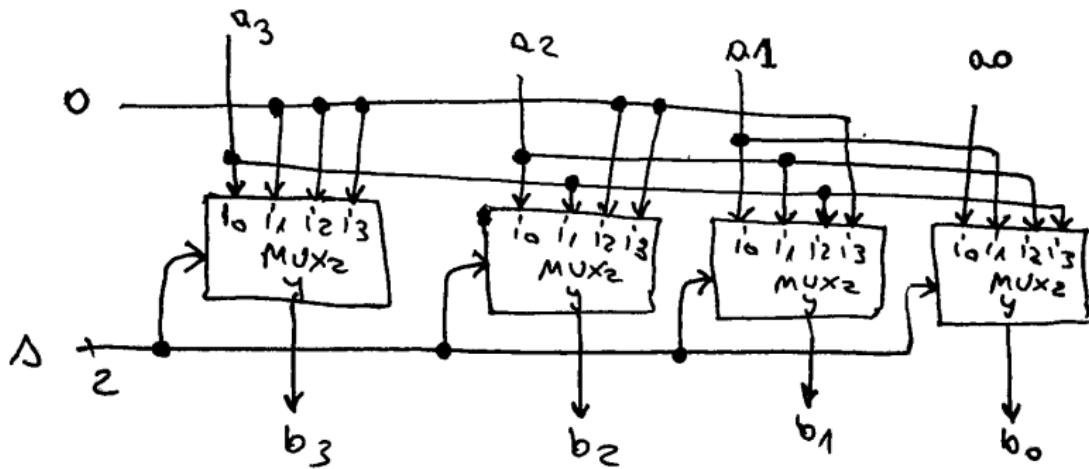
Exemplu: Explicați circuitele de shiftare pentru $n = 4$, $p = 2$.

- Shiftarea logică la stânga:



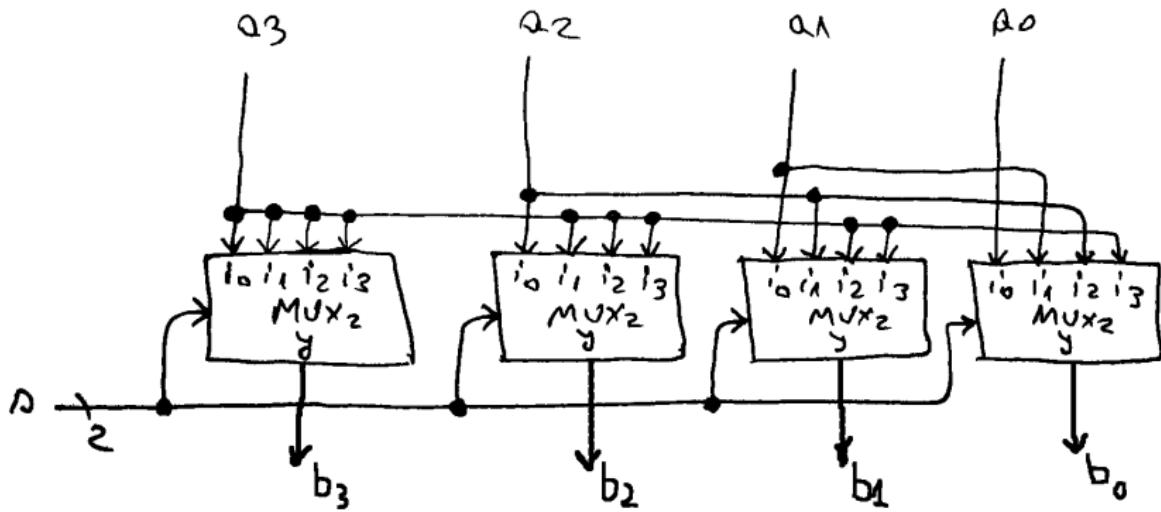
Shifter

- Shiftarea logică la dreapta:



Shifter

- Shiftarea aritmetică la dreapta:

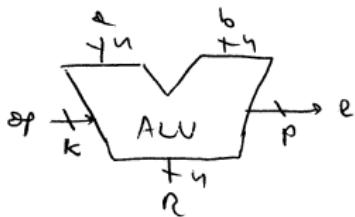


Unitate aritmetică și logică

Unitate aritmetică și logică (ALU):

O **unitate aritmetică și logică (arithmetic logic unit, ALU)** pe n biți ALU_n , $n \geq 1$, este un circuit care aplică unor operanzi numere întregi pe n biți o operație aritmetică sau logică selectată printr-un cod numeric; operanze și codul operației sunt date prin reprezentarea lor în calculator ca sir de biți.

Simbol:



unde: a, b sunt operanze (pe n biți), op este codul operației (pe k biți), r este rezultatul (pe n biți), e este un sistem de p alte ieșiri: carry out, overflow, etc. (uneori, o parte din aceste ieșiri se desenează în partea de jos a simbolului).

ALU implementează un număr $t \leq 2^k$ de operații, a.î. valorile valide ale lui op sunt doar t dintre numerele de la 0 la $2^k - 1$ (nu neapărat succesive).

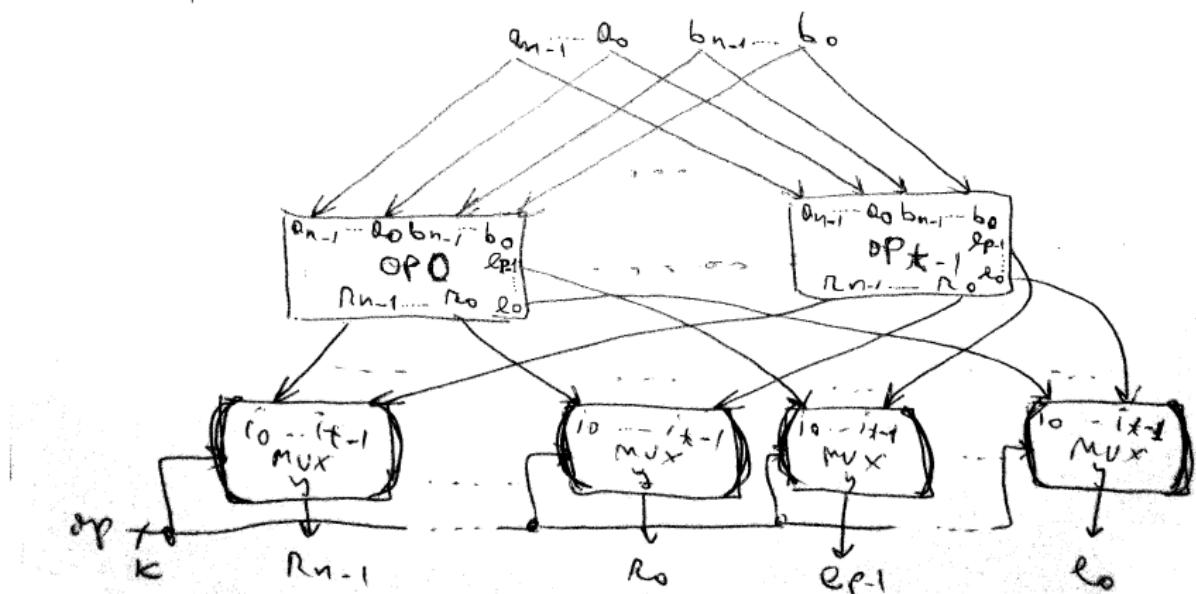
Unitate aritmetică și logică

O implementare directă se poate face astfel:

- construim t blocuri $0 - DS$, care implementează operațiile considerate; presupunem că valorile valide ale lui op sunt numerele succesive de la 0 la $t - 1$; notăm blocurile operațiilor corespunzătoare acestor valori OP_0, \dots, OP_{t-1} ;
- operanții a și b intră simultan în blocurile OP_0, \dots, OP_{t-1} ;
- fiecare ieșire finală r_i sau e_i , este aleasă dintre ieșirile corespunzătoare cu același i ale blocurilor OP_0, \dots, OP_{t-1} folosind un MUX_k ce are ca selector pe op (op intră simultan ca selector în toate aceste multiplexoare).

Unitate aritmetică și logică

Construcția circuitului:



Unitate aritmetică și logică

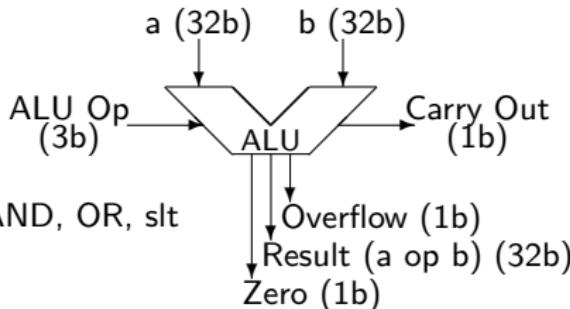
O implementare cu mai puține componente ar presupune să nu construim blocuri separate pentru fiecare operație ci să refolosim cât mai multe componente - de exemplu, scăderea se face cu sumatorul, dar introducând b și c_{in} negate.

O altă idee este să construim câte un ALU optimizat ca mai sus pentru fiecare bit, iar ALU pe n biți să se obțină prin legarea în paralel a n ALU pe un bit.

Vom ilustra această ultima variantă pe un caz particular: ALU_{32} folosit în diversele variante de procesor MIPS care vor fi prezentate mai târziu.

Unitate aritmetică și logică

Simbol:



efectuează: +, -, AND, OR, slt

a , b , $Result$, sunt operanții, respectiv rezultatul, operației (32 biți), $ALU\ Op$ este codul operației (3 biți), $Carry\ out$, $Overflow$, $Zero$ sunt ieșiri de 1 bit prin care se emite 1 d.d. la efectuarea operației a existat transport sau împrumut în bitul cel mai semnificativ, respectiv a avut loc depășire, respectiv rezultatul a avut toți biții 0.

Unitate aritmetică și logică

Acest *ALU* implementează operațiile: + (adunare), – (scădere), *AND* ("și" pe biți), *OR* ("sau" pe biți), *slt*, deci numai 5 dintre cele 8 numere naturale care se pot scrie pe 3 biți: 0, ..., 7, sunt valori valide ale lui *op*.

"*slt*" (set if less then) este operația MIPS care se aplică la trei regiștri:

slt reg1, reg2, reg3

și efectuează: $reg1 := \begin{cases} 1, & \text{dacă } reg2 < reg3 \\ 0, & \text{altfel} \end{cases}$

Unitate aritmetică și logică

Operația "slt" este utilă la implementarea calculelor booleene. De exemplu, secvența de cod în limbajul C:

x = (a < b) && (c < d)
poate fi tradusă de compilator în :

```
lw $t0, a    # se incarca operandul a din memorie in registrul $t0
lw $t1, b    # se incarca operandul b din memorie in registrul $t1
slt $t2, $t0, $t1
    # registrul $t2 primește valoarea 1/0 a expresiei a < b
lw $t0, c    # se incarca operandul c din memorie in registrul $t0
lw $t1, d    # se incarca operandul d din memorie in registrul $t1
slt $t3, $t0, $t1
    # registrul $t3 primește valoarea 1/0 a expresiei c < d
and $t0, $t2, $t3
    # registrul $t0 primește valoarea expresiei
    #      (a < b) & (c < d) ("si" pe biti in limbajul C);
    # intrucat $t2, $t3 pot contine 1 doar in bitul de rang 0,
    #      aceasta este echivalenta cu
    #      (a < b) && (c < d) ("si" in limbajul C);
sw $t0, x
    # se scrie rezultatul din registrul $t0 in variabila x din memorie
```

Astfel, expresia condițională se evaluează cu un cod ce se execută liniar, în locul unor blocuri condiționale îmbricate.

Unitate aritmetică și logică

Pentru construcția circuitului ALU_{32} , vom construi blocuri ALU_1 (ALU pe 1 bit), apoi vom lega în paralel 32 asemenea blocuri.

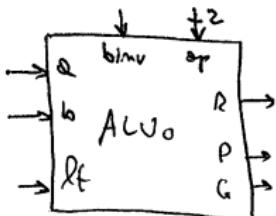
ALU_1 se construiește diferit în funcție de rangul bitului: $0, 1 \leq i \leq 30, 31$ (de exemplu, valoarea 1/0 furnizată de s/t este determinată la poziția 31 și emisă ca rezultat la poziția 0).

Vom nota aceste ALU_1 cu ALU_0 , ALU_i ($1 \leq i \leq 30$), respectiv ALU_{31} (a nu se confunda notația indexată ALU_i , care înseamnă ALU pe i biți, cu ALU_i , care înseamnă ALU_1 de la poziția i).

Unitate aritmetică și logică

- **Cazul 0:**

Simbol:



a, b sunt operanții (1 bit);

lft este o informație emisă ca rezultat în cazul "slt"

(rezultatul pe 32 biți va fi $\underbrace{0 \dots 0}_{b_{31}} \underbrace{0}_{b_0}$ sau $\underbrace{0 \dots 0}_{b_{31}} \underbrace{1}_{b_0}$);

$binv$ este 0 sau 1 după cum se va aduna a cu b sau a cu \bar{b} ;

(la adunare trebuie efectuat $a + b$, la scădere trebuie efectuat $a + \bar{b} + 1$);

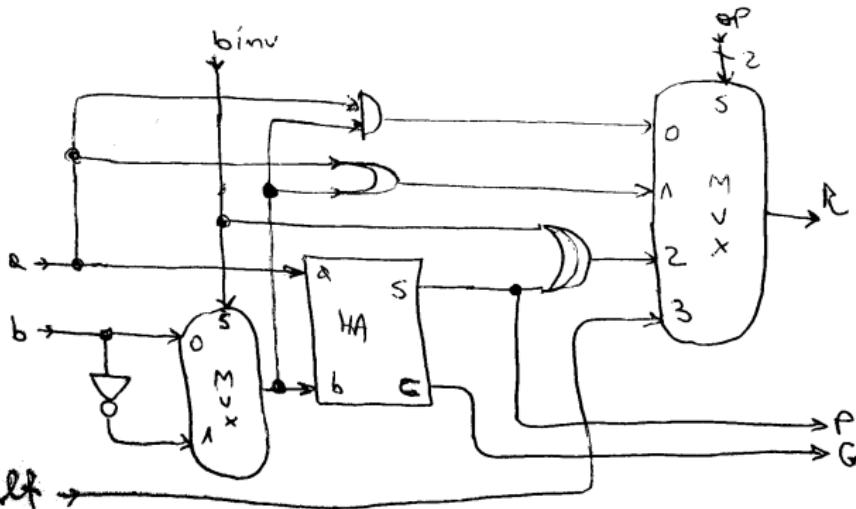
op este 0, 1, 2, 3 (reprezentat pe 2 biți) pentru a desemna respectiv operațiile $\wedge, \vee, +/-$ (distincția va fi făcută de $binv$), $<$; nu este intrarea $ALU\ Op$ a lui ALU_{32} , dar se calculează din aceasta și $binv$;

r este rezultatul operației (1 bit);

P, G vor fi intrări pentru circuitul CL .

Unitate aritmetică și logică

Construcția circuitului:



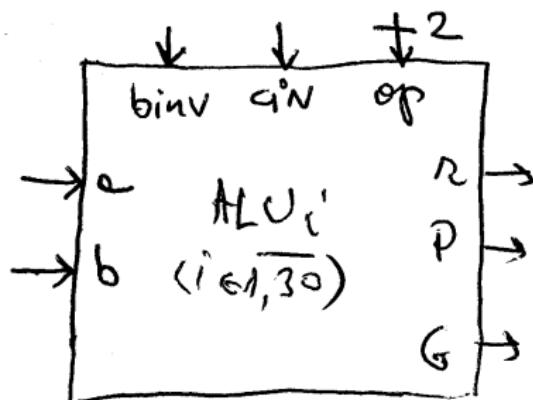
Observații:

- folosim HA, nu FA, deoarece aşa este construit circuitul ADD cu CL;
- pentru poziția 0, avem $c_{in} = \begin{cases} 0, & \text{în cazul } + \\ 1, & \text{în cazul } - \end{cases} = b_{inv}$; de aceea b_{inv} se compune "XOR" cu s pentru a da r (a se vedea circuitul ADD cu CL);
- It este generat de ALU₁ de la poziția 31 și emis prin poziția 0.

Unitate aritmetică și logică

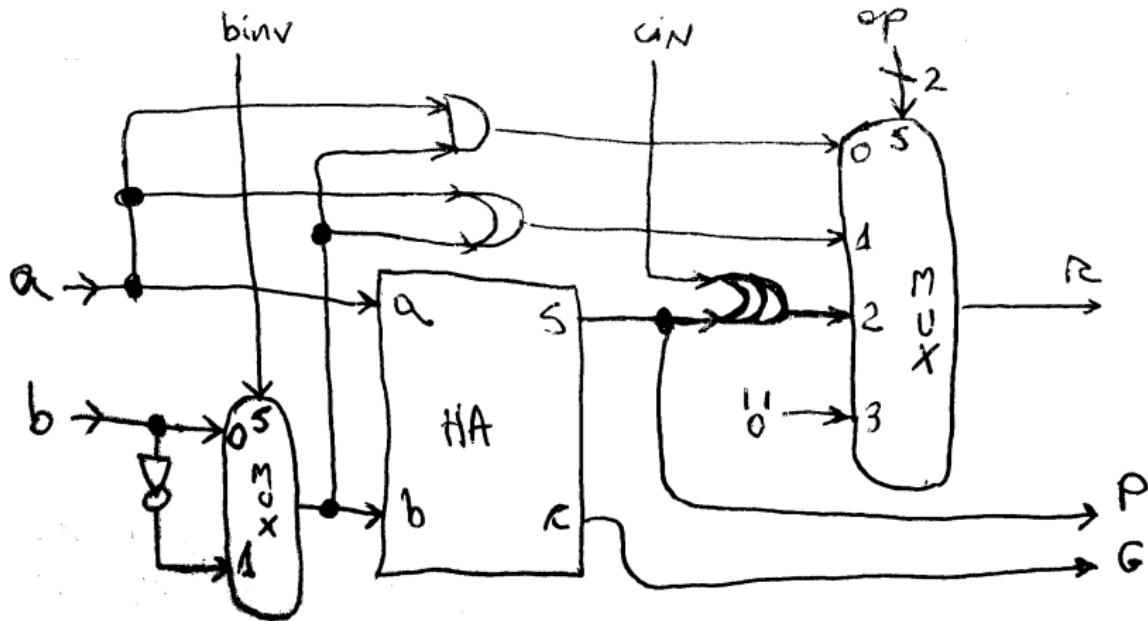
- Cazul $1 \leq i \leq 30$:

Simbol:



Unitate aritmetică și logică

Construcția circuitului:

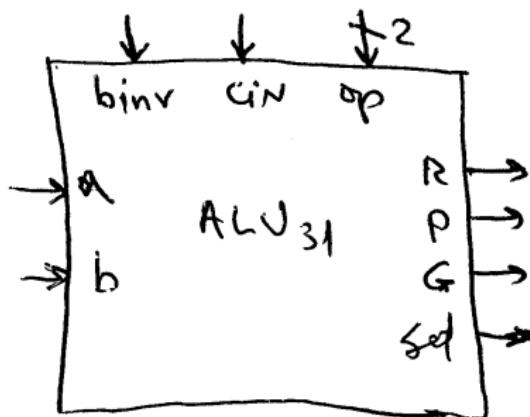


Observație: aici avem c_{in} , care vine din CL , iar l/t este mereu 0, deoarece numai în poziția 0 se poate emite ceva $\neq 0$ (la operația "lt" rezultatul este 0...00 sau 0...01).

Unitate aritmetică și logică

- **Cazul 31:**

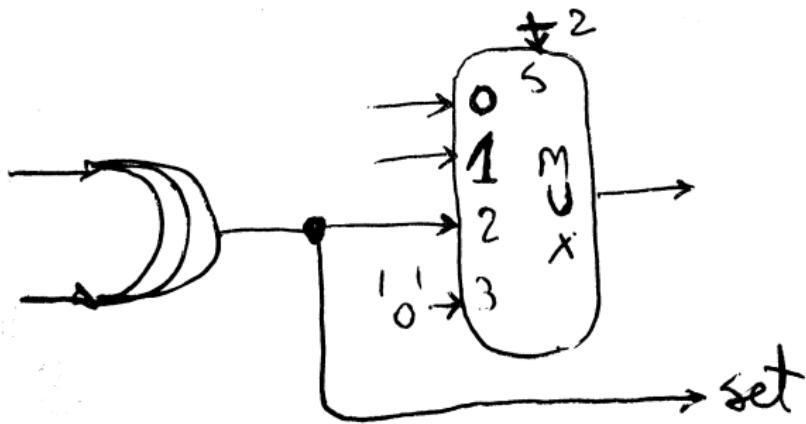
Simbol:



Observație: Apare în plus ieșirea *Set*, care va intra ca *It* în *ALU* de la poziția 0.

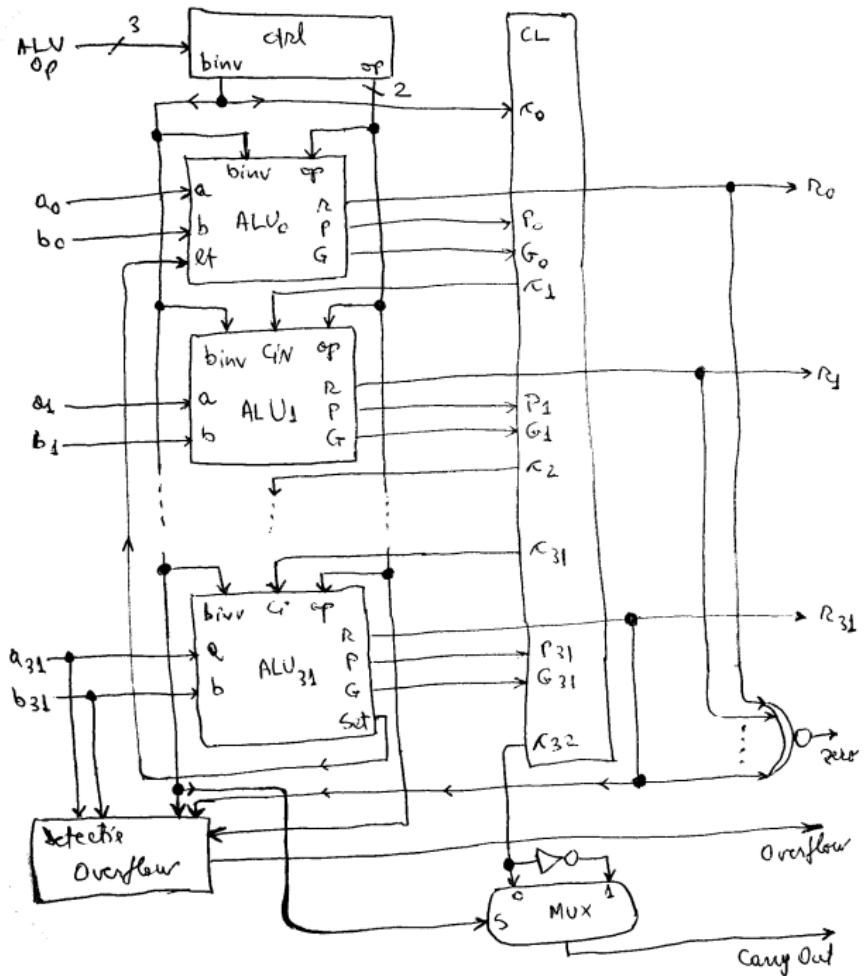
Unitate aritmetică și logică

Construcția circuitului în cazul 31 este asemănătoare celei din cazurile $1 \leq i \leq 30$, cu sigura diferență că ieșirea din "XOR" ieșe și prin "Set":



Într-adevăr, avem $t = 1$ d.d. pentru operanții pe 32 biți a și b avem $a < b$, d.d. $a - b < 0$, d.d. bitul de rang maxim al lui $a - b$ (i.e. s "XOR" c_{in} în cazul 31) este 1; în cazul 31 el nu trebuie emis prin r (care trebuie să fie tot 0) ci printr-o ieșire separată Set.

Construcția lui ALU pe 32 biți este pe slide-ul următor.



Unitate aritmetică și logică

Observație: pentru s/t se efectuează scăderea și se testează dacă rezultatul este < 0 , i.e. are bitul de rang 31 egal cu 1; acesta dă intrarea l_t din ALU de la poziția 0, dar este scos la poziția 31 prin Set , deoarece la s/t bitul de rang maxim r_{31} al rezultatului trebuie să fie tot 0.

Unitate aritmetică și logică

Ctrl este un circuit 0 – DS ce sintetizează din *ALU Op* pe b_{inv} și op , pe baza următorului tabel:

	$ALU Op_2$	$ALU op_1$	$ALU op_0$	b_{inv}	Op_1	Op_0
AND	0	0	0	0	0	0
OR	0	0	1	0	0	1
+	0	1	0	0	1	0
-	1	1	0	1	1	0
sel	1	1	1	1	1	1

Exercițiu: Implementați acest circuit ca *PLA*, *PROM* (sau mai simplu, observând că $b_{inv} = ALU Op_2$, $op_1 = ALU Op_1$, $op_0 = ALU Op_0$).

Unitate aritmetică și logică

Detectie Overflow este un circuit $0 - DS$ care detectează depășirea la $+$, $-$, după următoarea regulă:

op	a	b	rez
$a + b$	≥ 0	≥ 0	< 0
$a + b$	< 0	< 0	≥ 0
$a - b$	≥ 0	< 0	< 0
$a - b$	< 0	≥ 0	≥ 0

Observație: A avea depășire nu este totușa cu a avea transport/împrumut în bitul cel mai semnificativ (deși poate exista o legătură între ele). De exemplu, în limbajul C pe 32 biți, calculul $0 - 1 = -1$ efectuat în cadrul tipului int (numere întregi pe 32 biți) are împrumut în bitul cel mai semnificativ (deoarece pe biți se efectuează $0 \dots 00 - 0 \dots 01 = 1 \dots 1$), dar nu are depășire, deoarece rezultatul -1 începe în mulțimea de valori $\{-2^{31}, 2^{31} - 1\}$ a tipului int.

Unitate aritmetică și logică

Conform regulilor de reprezentare a numerelor întregi în complement față de 2, avem $x \leq 0$ d.d. $x_{31} = 1$. Atunci tabelul de valori implementat de circuitul *Detectie Overflow* este următorul:

a_{31}	a_{0_0}	b_{1NU}	a_{31}	b_{31}	r_{31}	Overflow
1	0	0	0	0	1	1
1	0	0	1	1	0	1
1	0	1	0	1	1	1
1	0	1	1	0	0	1
1	0	0	0	0	0	0

Observație: Overflow are sens doar la $+$, $-$ și de aceea, în rest, spunem că valoarea "Overflow" este 0.

Exercițiu: Implementați acest circuit ca *PLA*, *PROM*.

Cuprins

1 Performația calculatoarelor

Conceptul de performanță
Măsurarea performanței

2 Aritmetică sistemelor de calcul

Reprezentarea numerelor în matematică
Reprezentarea numerelor în calculator
Reprezentarea numerelor naturale ca întregi fără semn
Reprezentarea numerelor întregi în complement față de 2
Reprezentarea numerelor reale în virgulă mobilă

3 Logica pentru calculatoare

Algebre booleene
Funcții booleene
Particularizare la cazul B_2

4 Circuite logice

Circuite logice, Sisteme digitale
0-DS (Circuite combinaționale, Funcții booleene)
1-DS (Memorii)
2-DS (Automate finite)
Algoritmi de înmulțire și împărțire hardware
3-DS (Procesoare) și 4-DS (Calculatoare)
 n -DS, $n > 4$

Cicluri

Sistemele 1-DS sunt sisteme 0-DS închise prin exterior printr-un ciclu (mai general, pot avea mai multe cicluri, dar un singur nivel de cicluri).

Apare un prim grad de autonomie a circuitului, prin **stare** - ea depinde doar parțial de intrare, ceea ce conduce la o independență parțială a ieșirii de intrare. Evoluția ieșirilor rămâne sub controlul intrărilor, dar stările dău o autonomie parțială.

O altă caracteristică a 1-DS este că **pot păstra informația** de intrare pentru o perioadă determinată de timp, proprietate specifică **memoriilor** - de aceea, sistemele 1-DS sunt folosite pentru a construi diverse circuite de memorie (RAM, regiștri, etc.).

Informația memorată este pusă la dispoziția diverselor circuite în anumite faze de lucru, de aceea trebuie să existe o sincronizare a operațiilor în desfășurare, a.î. ele să poată conlucra corect și eficient.

Pentru aceasta se folosește un dispozitiv general de control al circuitelor, **ceasul (CK)** - el este un circuit bistabil ce asigură o discretizare a timpului de calcul, făcând posibilă definirea unor noțiuni temporale, ca: moment actual, tact, istoric, dezvoltare ulterioară, etc.

Cicluri

Există două tipuri de cicluri ce închid un CLC:

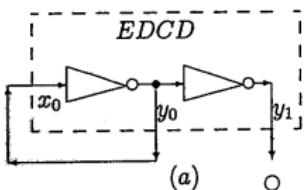
- **Cicluri stabile:** conțin un număr par de complementări; ele generează o stare stabilă și sunt utile în construcția circuitelor digitale.
- **Cicluri instabile:** conțin un număr impar de complementări; ele generează o stare instabilă la ieșire și pot fi folosite la construcția ceasului.

Pentru a fi stabil, un circuit trebuie să treacă printr-un număr par de complementări pentru toate combinațiile binare aplicate la intrare (altfel, pentru anumite combinații, se destabilizează).

Cicluri

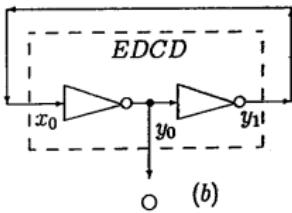
Exemplu: În figura de mai jos, circuitul (a) conține un ciclu instabil, circuitul (b) conține un ciclu stabil:

Instabil



(a)

Stabil



(b)

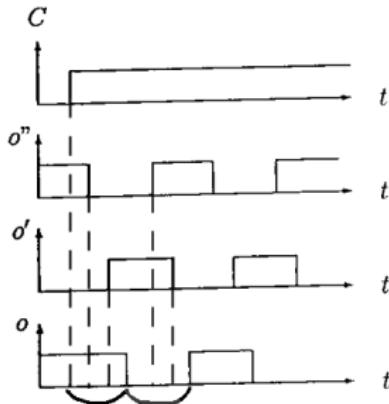
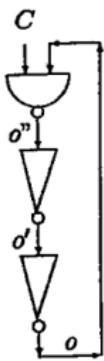
În cazul (a), dacă avem de exemplu starea $y_0 = 0 = x_0$, atunci vom obține la ieșire $y_1 = 0$ și noua stare $y_0 = 1 = x_0$, apoi vom obține la ieșire $y_1 = 1$ și noua stare $y_0 = 0 = x_0$, etc.; astfel, cele două ieșiri ale decodificatorului sunt instabile, comutând de pe 0 pe 1 și invers.

Momentul de schimbare a valorii de ieșire (din 0 în 1 și invers) definește **frecvența** circuitului și s.n. **tact**.

În cazul (b), dacă avem de exemplu $y_1 = 0 = x_0$, atunci starea y_1 va fi fixată la valoarea 0 iar la ieșire vom avea constant $y_0 = 1$; similar, dacă $y_1 = 1 = x_0$ (la ieșire vom avea constant $y_0 = 0$); deci, acest circuit are două stări stabile. Deocamdată însă nu știm cum să comutăm între stări, circuitul neavând o intrare prin care să putem controla schimbarea.

Cicluri

Exemplu: Considerăm următorul circuit cu 3 niveluri de complementare (ciclu instabil):



Dacă la intrare aplicăm comanda $C = 0$, pe fiecare linie valoarea semnalului rămâne constantă, circuitul își conservă starea.

Dacă aplicăm comanda $C = 1$, circuitul generează un semnal periodic.

Comportarea circuitului este descrisă de diagramea din dreapta.

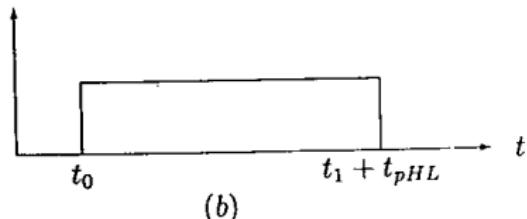
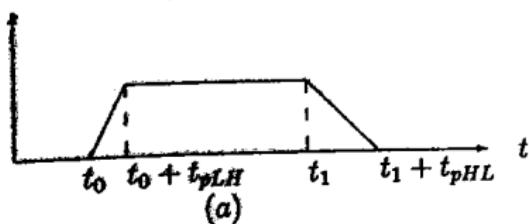
Constatăm că este nevoie de un anumit timp pentru ca semnalul să se propagă prin circuit, timp care depinde de structura circuitului și natura fenomenelor fizice folosite (am marcat cu arce intervalele de timp necesare propagării semnalului la două ciclări succesive).

Cicluri

Așa cum am văzut și în exemplul precedent, schimbarea stării unui circuit nu este instantanee și depinde de anumite caracteristici fizice și structurale ale circuitului.

Vom nota cu t_{PLH} intervalul de timp în care un circuit comută de la starea 0 la starea 1 și cu t_{PHL} intervalul de timp de trecere de la starea 1 la starea 0.

Ambele valori sunt numere ≥ 0 și considerate constante pentru un circuit; ele nu sunt neapărat egale între ele.



Situația reală este cea din figura (a) de mai sus.

Uneori se consideră o situație ipotetică, de schimbare instantanee a stărilor, iar evoluția se aproximează ca în figura (b).

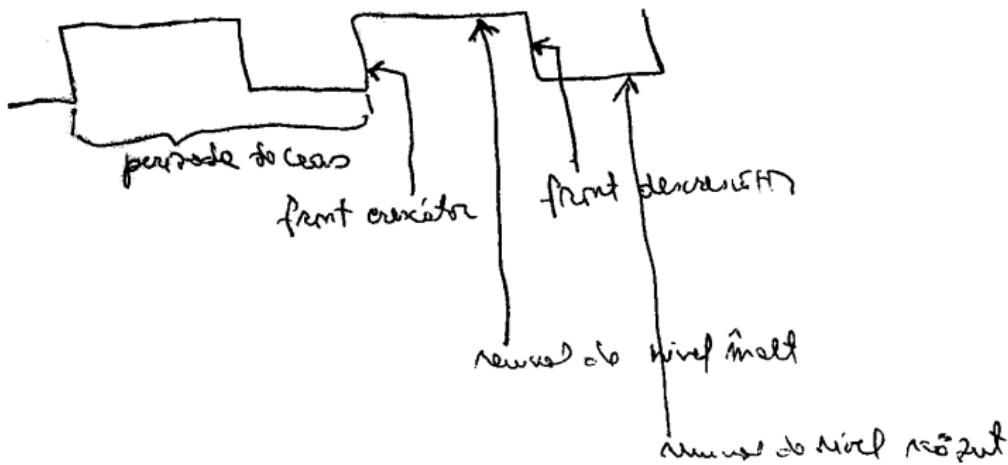
Cicluri

Ceasul produce un semnal autonom cu perioadă (frecvență) fixă.

El se folosește în logica secvențială pentru a decide momentul în care trebuie actualizat un element ce conține stare.

Un sistem acționat cu ceas se mai numește și **sistem sincronizat**.

Semnalul de ceas are următoarele componente:



Cicluri

La circuitele controlate de ceas, schimbările de stare se pot produce:

- fie în intervalul când semnalul este de nivel înalt (în acest interval modificarea intrărilor determină modificarea stării ieșirii);
notăm acest lucru prin: 
- fie pe un front de ceas, crescător (ascendent) sau descrescător (descendent); aceasta s.n. **acționarea pe frontul ceasului** și se notează:  , respectiv  acționarea pe front este mai bună, deoarece se poate preciza mai exact momentul instalării noii stări.

În metodologia acționării pe front, frontul crescător / descrescător care determină producerea schimbărilor de stare s.n. **front activ**.

Alegerea lui depinde de tehnologia implementării și nu afectează conceptele implicate în proiectarea logicii.

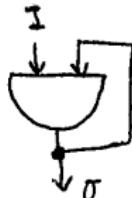
Constrângerea principală într-un sistem sincronizat este că semnalul ce trebuie scris în elementele de stare trebuie să fie valid (în particular stabil, să nu se mai modifice până nu se modifică intrările) la apariția frontului de ceas activ.

De aceea, perioada ceasului trebuie să fie suficient de lungă a.î. semnalele respective să se stabilizeze (există o limită inferioară a perioadei).

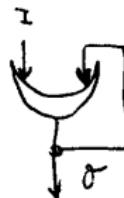
Zăvor elementar

În continuare, prezentăm principalele circuite cu un ciclu intern:

ZĂVOR ELEMENTAR (LATCH):



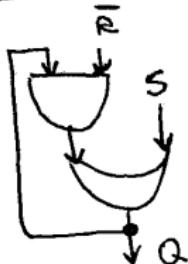
$I = 1 \Rightarrow$ CONSERVĂ STAREA
 $I = 0 \Rightarrow$ TRECE ÎN STAREA 0
Deci, poate fi comandat
SĂ COMURE ÎN STAREA
0 și PĂVORÂSTE-o
ESIG ACTIV LA FRECUENȚA
JOASK



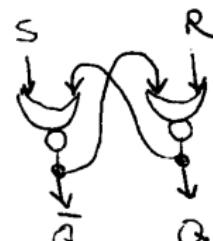
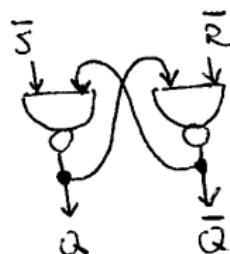
POTRĘ FI COMANDAT
SĂ COMURE ÎN
STAREA 1 și
PĂVORÂSTE 1
ACTIV LA FRECUENȚA
INALTA

Zăvor elementar eterogen

ZĂVOR ELEMENTAR ETEROGEN:



de
MORGAN



OBS: AC DULGA
N-ARE "-",
DAR Q NU ESTE
SUB S

$$Q = S + \bar{R}Q$$

OBS: INTRĂRIE SUNT $\left\{ \begin{array}{l} \bar{R} \text{ ACTUA FRACVENTA JOSĂ} \\ S \text{ ÎNALTĂ} \end{array} \right.$

POSSIBILITĂȚI DE
FUNCȚIONARE:

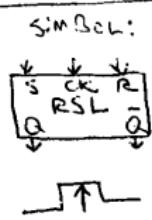
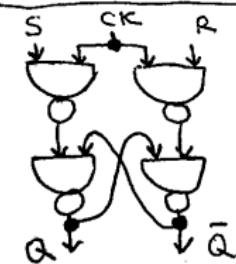
S	R	Qn+1
0	0	Qn
1	0	1 (SET)
0	1	0 (RESET)
1	1	NEACȚINȚĂ

DEZAVANTAJE:

- NU STIE SA INVERSEZĂ STAREA
- NU poate evita hardware INTRARGA A-A

Zăvor elementar cu ceas

ZĂVOR ELEMENTAR CU CEAS (RSL: RESET-SET LATCH)



EX: DACĂ TREBUIE SETAT FĂC $S=1, R=0$,
APOI $CK=1$ PENTRU UN TIPOS t_{PLH}

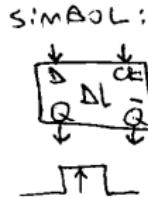
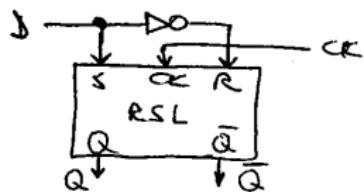
CĂSTIG: { • NU MAI TREBUIE COMPLEMENTATE
INTRĂRIILE R, S
• ÎESIRILE NU MAI SUNT INVERSATE
(Q ESTE SUB S)

OBS: PE NIVELUL ACTIV AL CK ZĂVORUL ESTE
TRANSPARENT, I.C. ÎȘI AGAFIE SCHIMBA STARE
DACĂ MODIFICE S, R.

DEZAVANTAJE: { • CÂND $CK=1$ POATE COMUTA DE MAI MULT DE
• NU POATE EVITA INTRAREA 1-1

Zăvor de date

ZĂVOR DE DATE (DL: DATA LATENȚĂ)



CĂȘTIG: • ELIMIN INTRAGA 1-1

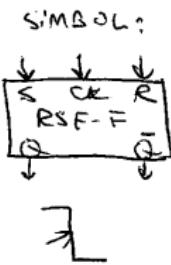
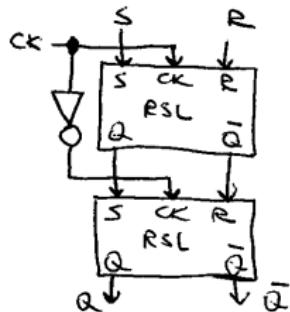
OBS: MERG J=1 (IEGEREA URMEASĂ PERMANENT INTRAGA \Rightarrow AUTONOMIE REDUSĂ)

DSPRAVANTAJ: • ÎNCĂ POATE COMUNĂ DE MAI MULTE ORI ÎN TIMPUL UNUI ZACĂ

Structura master-slave

STRUCTURA MASTER-SLAVE (RSF-F: RESET-SET FLIP-FLOP):

SE BAZA ÎN PE UN CIRCUIT CU 2 STĂRÎ (NUMIT FLIP-FLOP) CARE COMUTĂ SINCRONIZAT CU ARFUL (ÎN SUS SAU JOS) SEMNALULUI DE COMANDĂ.



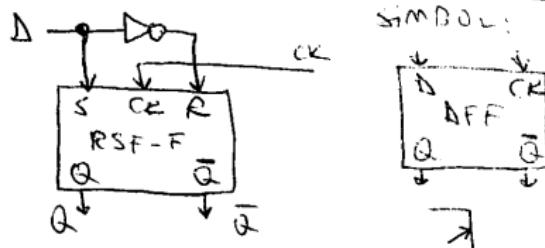
OBS: PRIMUL RĂVOR ESTE TRANSPARENT LA NIVELUL SUPERIOR (PRIMA 1/2 TAET T_{FL}), AL DOILEA LA NIVELUL INFERIOR (A 2-A 1/2 TAET T_{FL}). ÎN TOTAL, CIRCUITUL PUTEA COMUTA DOAR O DATĂ PE TAET, îNEIESIREA PIVALĂ BANTĂ PE FRONTUL DESCENDENT AL

CĂSTIG: • NU MAI PUTEA COMUTA DE MULTĂ MULTE ori LA UN CICLU

DEZAVANTAJE: • NU EVITĂ ÎNTREAGĂ 1-1

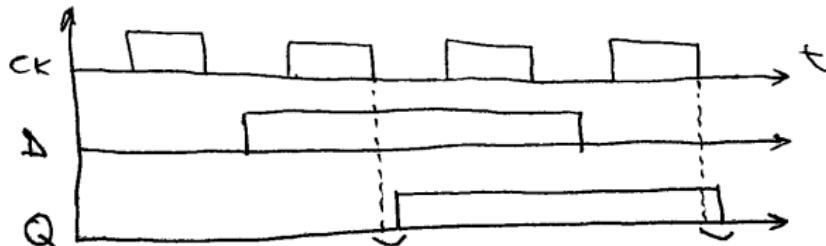
Flip-flop cu întârziere

FLIP-FLOP CU ÎNTÂRZIRE (DFF: DELAY FLIP-FLOP)

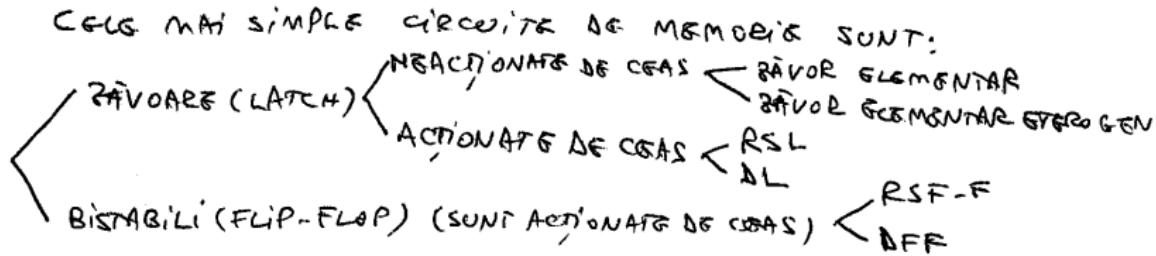


CĂSTIG: \rightarrow SE ELIMINĂ ÎNTRAREA
A-1 LA S-R

OBS. IESIREA Q CONȚINE D CU ÎNTRÂRIGELE DE 1 TACT
(MAI EXACT, IESIREA SE COLOGE LA PRIMUL SFÂRȘIT DE
TACT DE DURĂ MODIFICAREA LUI D):



DFF ESTE UN CIRCUIT IMPORTANT (ARE APlicațII MULTe).

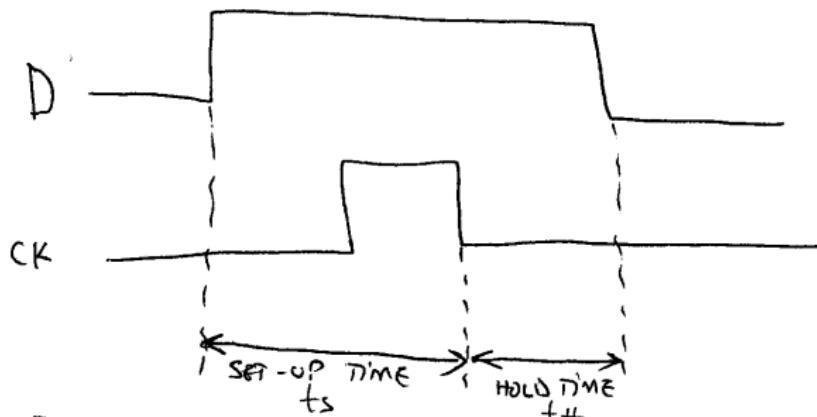


AFAȚ LA ZĂVORALE ACIONATE DE CEAS CÂT SI LA BISTABILI, ÎNSIREA ESTE EGALĂ CU VÂLUAREA STĂRII MEMORATE ÎN ELEMENT. DIFFERENȚA ÎNTRU ELE ESTE MOMENTUL LA CARE CEASUL PROVOACĂ SCHIMBAREA STĂRII.

- LA ZĂVORALE ACIONATE DE CEAS STAREA SE SCHIMBĂ DÉCĂ CÂTE OI ÎNTRĂRILE SE SCHIMBĂ SI CEASUL ESTE ACTIVAT (ABICĂ ALĂM UN INTERVAL DE TRANSPARENȚĂ TTL)
- LA BISTABILI STAREA SE SCHIMBĂ DOAR PE FRONTUL DE CEAS (LA NOI \overline{z})

INTRAREA TREBUIE SĂ FIE TOTUȘI VALIDĂ (STABILĂ) PENTRU UN INTERVAL DE TIME MAȚINE (TIIMPUL DE STARARE - SET-UP TIME) și DUPĂ (TIIMPUL DE MENȚINERE - HOLD TIME) FRONTULUI.

ILLUSTRARE ÎN CAZUL DFF:

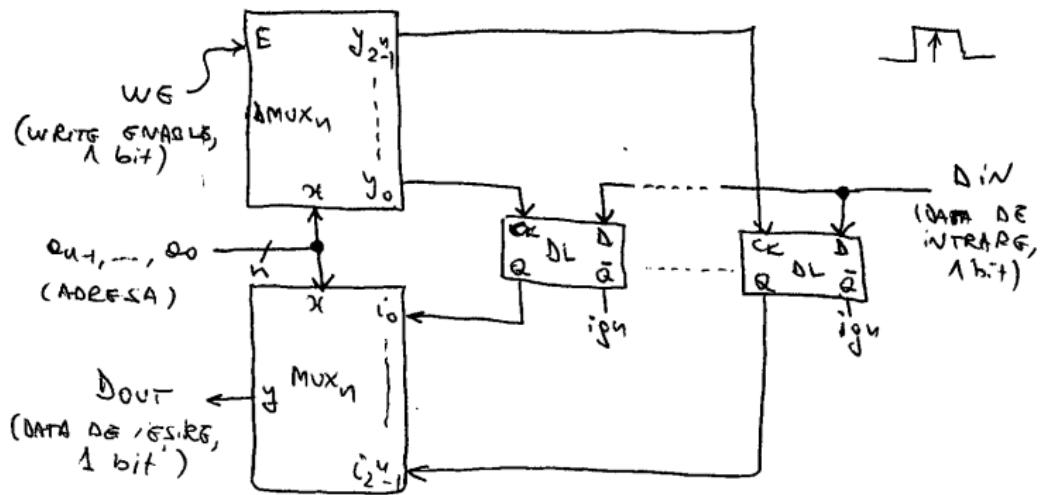


DACĂ NU SE RESPECTĂ ACESTE INTERVALE DE STABILITATE, ȘI SEGA BISTABILULUI PUTEA SA NU FIE PREVIZIBIL.

HOLD TIME DE OBICEI ESTE O SUA ROTARE MICĂ, DECİ NU REPREZINTĂ UN MOTIV DE INGEDIȚARE.

RAM

MEMORIA RAM : ESTE O EXTENSIIE PARALELA DE 1-DS



EXEMPLIFICAREA SA FACUT PENTRU UN RAM CU LOCATII DE 1 BIT;
PENTRU UN RAM CU LOCATII PE K BITI, FACEM O EXTENSIIE PARALELA
DE K RAM-URI CU LOCATII DE 1 BIT; WE SI AL VOR FI COMUNI,
DIN SI DOUT VOR FORMA DATELE DE I/O DE K BIT.

RAM

FUNCȚIONARE:

- LA CITIRE: $Q_{u-1,..,0}$ SELECȚEază PRIN MUX O CELULĂ și
READ PEIN DOUT INFORMAȚIA STOCATĂ Aceea
- LA SCRITURă: $Q_{u-1,..,0}$ SELECȚEază PRIN DMUX O CELULĂ și
TRIMITE ACEAȘI CA SEMNAL DE CEAȘ; DIN SE DISTRIBUIE
LA TOATE CELULELE, DAR VA INTRA DOAR ÎN ACEEA.

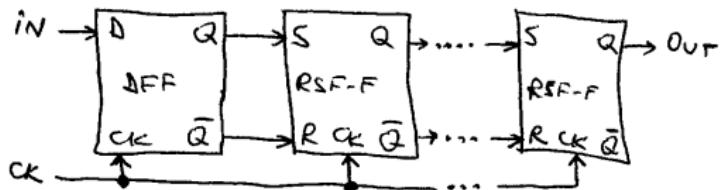
- OBS.:
- RAM ESTE UN CIRCUIT SIMPLU și SE PUTE CONSTRUI UJOR LA DIMENSIUNI MARI și EXISTĂ VARIANTE și OPTIMIZĂRI - A SE VEDEA CARTEA P&H : SRAM, DRAM, etc.
 - RAM ADMITE și o DEFINIȚIE RECURSIVĂ, PLECÂND DE LA DEFINIȚIA RECURSIVĂ A COMPOUNENTELOR SALE (exclusiv!)
 - OBSERVAM că în mod NATURAL RAM ARE UN NUMAR DE LOCII PUTEREA A LUI 2.

Registru serial

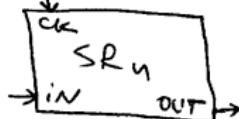
REGISTRU: SISTEM DE F-F (RSF-F SAU DFF) CAPABIL SA STOCKEZE COVINTE DE N BITI'S TOATE F-F AU LINII DE CONTROL (EX.CEAS) COMUNE.

DUPA MODUL DE EXTENSIE, REGISTRUL PUTEA FI:

- REGISTRU SERIAL:



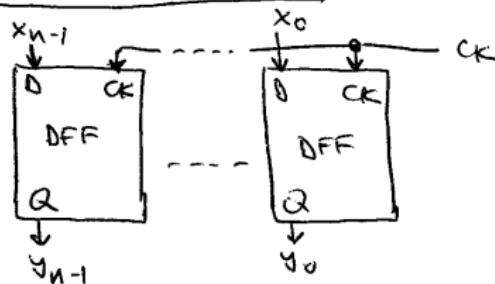
SIMBOL:



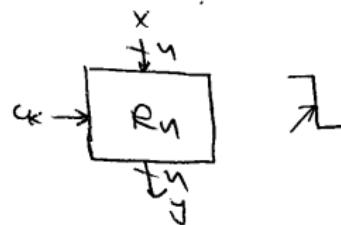
- Obn:
- REGISTRUL INTRODUCÉ O ÎNTRARE DE ~~N~~ MAGI ÎNTRÉ ÎNTRAPÉ SI IEȘIRE
 - SE PUTEA DEFINI SI RECURSIV (EXTENSIE SEQUALĂ) (EXERCITIU!)

Registru paralel

- REGISTRU PARALEL:



SIMBOL:

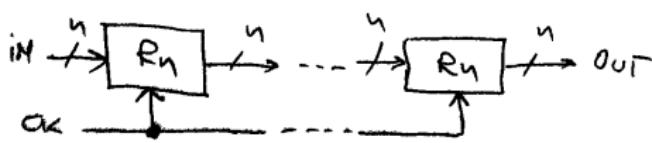


- Obn:
- PRINCIPALUL AVANTAJ ESTE NETRANSPARENȚA, CU EXCEPȚIA UNUI "TRANSPARENȚE NEACORDABIL" ÎN PRIMELE $t_S + t_H$ MOMENȚI; DECI, PUTEA SĂ ÎNCHEIE CU UN NOU CICLU SÌ (PENTRU CÂZUL NETRANSPARENT) SE PUTEA ÎNCĂRCA CU O altă VALOARE, INCLUSIV UNA CE DERINDE DE PROPRIUL CONȚINUT,
 - R_n ADMITE SÌ o DEFINIȚIE RECURSIVĂ (EXTENSIILE PARALELĂ).

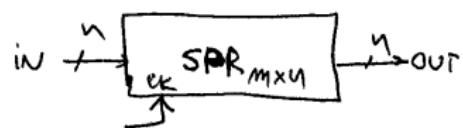
(excepție!!)

Registrul serial-paralel

- REGISTRU SERIAL-PARALEL:



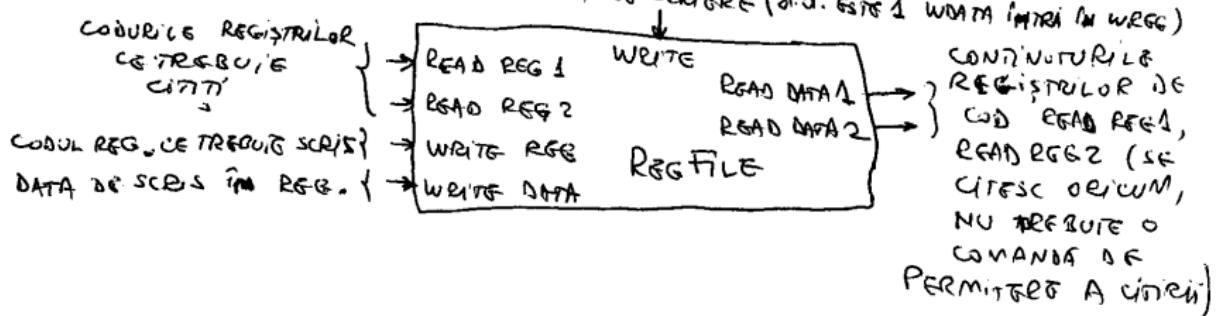
SIMBOL:



Fisier de registri

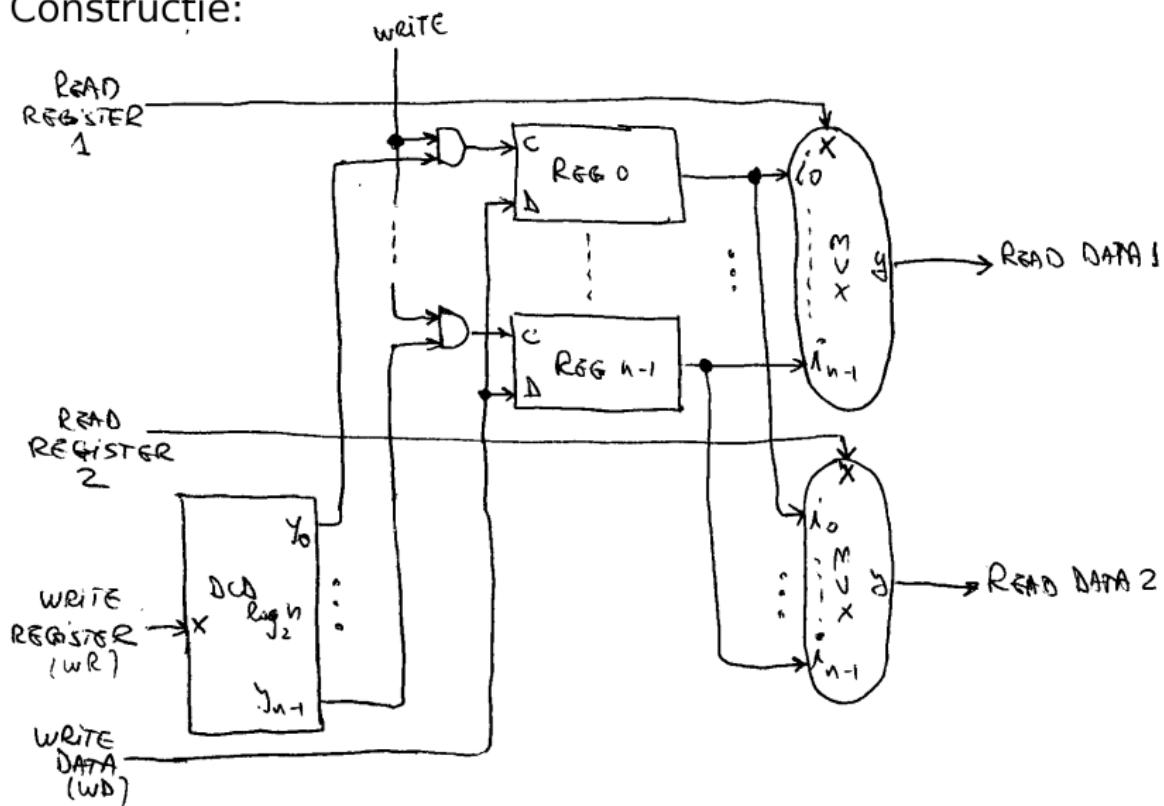
FISIER DE REGISTRI: SET DE REGISTRI CARE POT FI CITI SAU SCRI, PRIN FURNIZAREA NUMARULUI REGISTRULUI ACCESAT

ÎN DIVERSE VARIANTE DE PROCESE DE MIPS SE REALIZEAZĂ URMĂTOAREL
FISIER PE REGISTRI:



Fișier de regiștri

Construcție:



Fisier de registri

- Obl: • WR, WD, W AU CONSTRAÎNGERI PRIVIND TS, TH, PENTRU A ASIGURA CĂ ÎN FISIERUL DE REGISTRI ESTE CORECT SCRISĂ DATA
- CE SE PENTÂPLĂ DACĂ UN ACELAȘI REGISTRU ESTE CITIT SI SCRIS ÎN ACELAȘI CICLU DE CEAS? Deoarece scrierea se produce pe frontierul ceasului, REGISTRUL ESTE VALID PE PERIOADA DE TIIMP CÂND ESTE CITIT; VALOAREA ÎNTOARSĂ LA CITIRE ESTE VALOAREA SCRISĂ AColo ÎNTR'UN CICLU PRECEDENT; DACĂ AVORIM CĂ CITIREA SĂ ÎNTOARCA VALOAREA SCRISĂ ÎN ACELAȘI CICLU, ESTE NEvoie DE LOGICĂ SUPLIMENTARĂ ÎN FISIERUL DE REGISTRI SAU ÎN AFARA LUI.

Aplicație la 1-DS

Exercițiu (aplicație la 1-DS):

Fie $a(x) = \sum_{i=0}^n a_i x^i \in \mathbb{Z}_2[x]$ fixat. Construim un circuit care primește succesiiv coeficienții unui polinom $b(x) \in \mathbb{Z}_2[x]$ și produce succesiiv coeficienții produsului $a(x) \cdot b(x)$.

REZOLVARE:

În \mathbb{Z}_2 au loc $\left\{ \begin{array}{l} \text{(înmulțirea)} \xrightarrow{\quad} \wedge \text{(AND)} \\ \text{(adunarea)} \xrightarrow{\quad} \oplus \text{(XOR)} \end{array} \right\}$ și sunt asociative (se poate verifica pe tabelele operațiilor).

$$\text{Dacă } b(x) = \sum_{i=0}^m b_i x^i, \text{ atunci } a(x) \cdot b(x) = \sum_{p=0}^{m+n} c_p x^p,$$

$$\text{și } \forall p \in \overline{0, m+n} \quad c_p = \sum_{\substack{i+j=p \\ i \leq m \\ j \leq n}} a_i b_j = \bigoplus_{\substack{i+j=p \\ i \leq m \\ j \leq n}} a_i \wedge b_j.$$

Explicităm calculul c_p pentru cătreva p :

$$c_{n+m} = a_n b_m$$

$$c_{n+m-1} = a_n b_{m-1} \oplus a_{n-1} b_m$$

$$c_{n+m-2} = a_n b_{m-2} \oplus a_{n-1} b_{m-1} \oplus a_{n-2} b_m$$

$$\dots$$

$$c_2 = a_2 b_0 \oplus a_1 b_1 \oplus a_0 b_2$$

$$c_1 = a_1 b_0 \oplus a_0 b_1$$

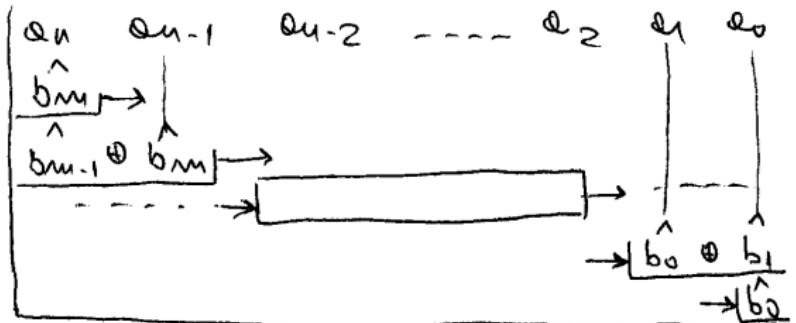
$$c_0 = a_0 b_0$$

} Obs. că oarecare sume și cresc că nr. de termeni, apoi (când $p < m, m$) scad

Aplicație la 1-DS

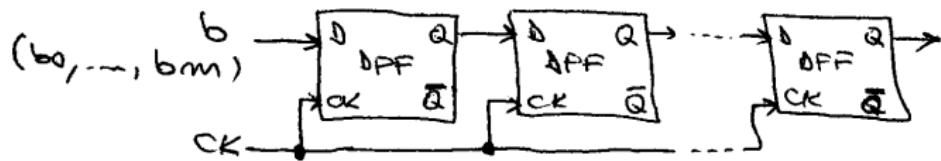
PUTEAM SĂ PRESUPUNEM CĂ TOȚIE SUMELE AU BOAȚE a_i -URILE ($m+1$ TERMENI) IAR ACOLU UNDE TERMENUL LIPSEȘTE AVEM $b_j \geq 0$. ILUSTREAM MAI DESS MODUL DE CALCUL AL CELEOR ~~M+M~~ SUME DE CARE $n+1$ TERMENI:

Obs. că "TERMENUL"
 b_0, \dots, b_m AVANSGAZĂ
PAS CU PAS SPRE
DREAPTA și SUB
 a_n, \dots, a_0 și LA
FICARE PAS SE
INMULTEȘTE (\wedge) b_j -URILE CU a_i -URILE SUB CARE SE AFLĂ și SE
EMIȚE SUMA (\oplus) PRODUSELOR CA SP.



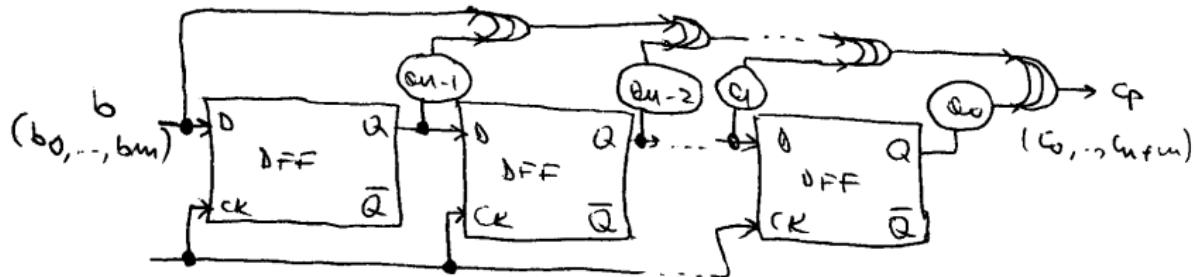
Aplicație la 1-DS

PENTRU IMPLEMENTAREA AUEM NEvoie ÎN PRIMUL RÂND DE UN CIRCUIT CARE SĂ STOCHEZE SEGMENTUL CURENT AF_{n+1} DE LA URMIATORUL TREBUIE ÎNMULȚIT CU BIJURII SI SA PERMIȚE AVANSUL PAS CU PAS SPRE DEGADĂ, UN ASCENDENȚĂ CIRCUIT PENTRU:

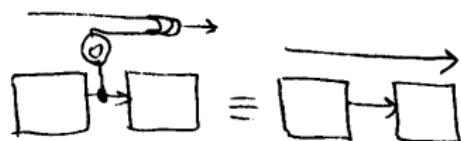


Aplicație la 1-DS

Acest circuit trebuie imbogățit cu un CLC care să fie legat la ieșirea lui b_n . Deoarece curentul rezistență este preluat din D/Q DFF-urilor și nu este fixat prin circuit, se va folosi un curent de arănd în vedere că $Q_i \in \{0, 1\}$ și $Q_n = 1$ (datorită gradului lui Q_n), circuitul complet poate fi:

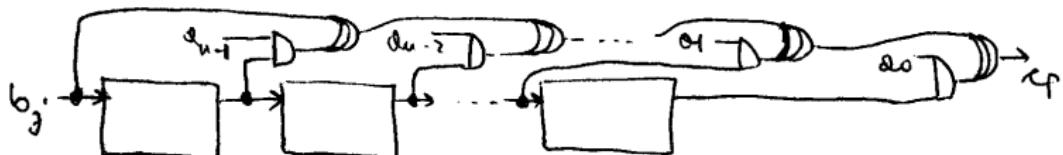


UNDE:



Aplicație la 1-DS

DACĂ DOREAM CA SĂ OI SĂ FIE DATE DE ÎNTRARE (DOAR NR. LOR UTMĂÎNTE FIXE PRIN CONSTRUCȚIE), PUȚEM AREA URMAȚIUL PLC:



Coeficienții a_i trebuie însă introduși toti o dată și menținuți la intrare la fiecare tact, în timp ce coeficienții b_j se introduc pe rând, la tacți succesivi.

Aplicație la 1-DS

Obs. CONSTATĂM CĂ EXȚENSIILE SERIALE DE BISTABILI (EX. DFF) SE PREZĂ LA REZOLVAREA UNOR PROBLEME UNDE SE CEREGE PRELUCRAREA UNOR SIRURI, ALE CĂROR ELEMENTE SUNT UNELE PE REFERINȚA LA TACRUI SUCCESIVI; EX: RECONOASTEREA UNOR PATTERN ÎN SPUL CINTĂ. STRUCTURA GENERALĂ A CRÎCITULUI ESTE O EXȚENSIUNE SERIALĂ DE BPF + UN CRC CE EFECTUARE CALCULUL DE LA REFERIRE PAS.

Cuprins

1 Performația calculatoarelor

Conceptul de performanță
Măsurarea performanței

2 Aritmetică sistemelor de calcul

Reprezentarea numerelor în matematică
Reprezentarea numerelor în calculator
Reprezentarea numerelor naturale ca întregi fără semn
Reprezentarea numerelor întregi în complement față de 2
Reprezentarea numerelor reale în virgulă mobilă

3 Logica pentru calculatoare

Algebre booleene
Funcții booleene
Particularizare la cazul B_2

4 Circuite logice

Circuite logice, Sisteme digitale
0-DS (Circuite combinaționale, Funcții booleene)
1-DS (Memorii)
2-DS (Automate finite)
Algoritmi de înmulțire și împărțire hardware
3-DS (Procesoare) și 4-DS (Calculatoare)
 n -DS, $n > 4$

Automate finite

AUTOMATELE SUNT MASINI ABSTRACTE (OBIECTE MATEMATICI), NU TEHNICE. EXISTA MAI MULTE FELURI DE ASEMANA MASINI: AUTOMATE FINITE (DETERMINISTE, NEDETERMINISTE, NEDETERMINISTE CU λ -TRANSITII), AUTOMATE STIVI, TRANSLATORI (DE DIVERSE FELURI), MASINI TURING, ETC.

ELE SUNT FOLOSITE IN DIVERSE DOMENII ALG MATEMATICII SI INFORMATICI TEORETICE PENTRU A DESCRIE O TRANSFORMARE PRIN PEGLUCRAREA SPECIATĂ PE O ASEMANA MASINA: LIMBAJ FORMALE, TEORIA COMPLĂRUI, CALCULABILITATE, ETC.

ÎN ADOREM CĂ ÎMPLIMENTAREA FUNCIONALITATEA UNUI AUTOMAT PE ÎNTRUN CIRCUIT. VOM VEDEA CĂ ORICE AUTOMAT FINIT SE poate împlementa într-un mod standard ca 2-DS. DE LA CÂZ LA CÂZ, ÎN FUNCȚIE DE UTILITATEA DE MOMENT, SE poate construi și un 1-DS 3-DS, etc., DAR ÎN TOATE CAPURILE VA EXISTA SI UN 2-DS ECHIVALENT.

ÎN ORICE CAP, NU TREBUIE CONFUNDAT AUTOMATUL (OBIECT MATEMATIC) CU 2-DS-UL CARE IL IMPLEMENTAREA (ASA CUM NU TREBUIE CONFUNDATA O FUNCȚIE BOOLEANĂ CU PLA-UL/ROM-UL CARE O IMPLIMENTEAZĂ).

Automate finite

Def: AUTOMAT: SISTEM $A = (Q, X, Y, \delta, \lambda)$, unde:

Q MULTIME NEVIDĂ (stări)

X MULTIME FINITĂ NEVIDĂ (valoare de intrare)

Y MULTIME FINITĂ NEVIDĂ (valoare de ieșire)

$\delta: Q \times X \rightarrow Q$ (funcția de tranziție a stărilor)

λ este funcția de ieșire, definită

$\lambda: Q \times X \rightarrow Y$ (pt. AUT. MEALY)

$\lambda: Q \rightarrow Y$ (pt. AUT. MOORE)

OBS: De fapt am definit noțiunea
de TRANSLATOR, din teoria limbajelor formale.

Def: AUTOMATUL ESTE FINIT, dacă Q FINITĂ

și acceptă o definiție nerecurzivă

OBS: SE poate demonstra că AUTOMATELE MEALY/MOORE SUNT GENEVALENTE
(I.E. SE POT DESCRIVE UNUL PRIN CEALALT) ABSTRACTE FĂCÂND EVENTUAL
DE PRIMA VALOARE DE IEȘIRE.

ÎN CELE CE URMEAZĂ, DACĂ NU SPECIFICAȚI ALTECAU, VOM CONSIDERA
AUTOMATE FINITE MEALY (ÎN CARESA PĂRȚILE SUNT FOLOSITE
MULȚ AUTOMATE MOORE).

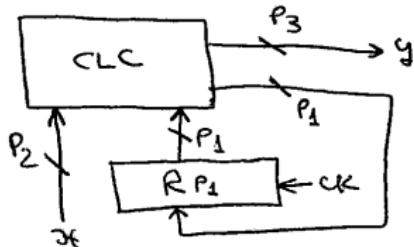
Automate finite

- UN AUTOMAT poate fi descris:
- SPECIFICAND $\{Q, X, Y\}$ PRIN SEMNIFEREG
 $\{f, \lambda\}$ PRIN TABEL
 - PRINTRUN GRAF DE TRANSITIE, AVÂND:
 - { NOURI = STARI: $\{p\}$
 - { ARCI: { LA AUT. MEALY: $\{q\} \xrightarrow{x/y} \{t\}$, INSEMNAND } $\delta(p, x) = t$
LA AUT. MOORE: $\{q\} \xrightarrow{x} \{t\}$, INSEMNAND } $\delta(q, x) = t$
 $\lambda(q) = y$

Automate finite

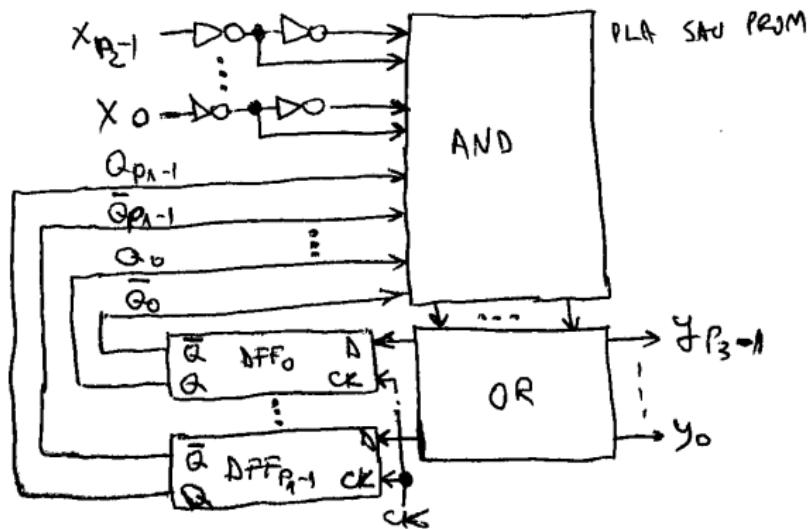
IMPLEMENTARE:

- NUMEROTAM ELEMENTELE LUI Q, X, Y (EX: Q_0, \dots, Q_{n-1}) SI ANLOCUIM FIECARA ELEMENT CU CODUL SAU, REPREZENTAT PE UN NUMAR DE BITI (EX: $Q_6 \rightarrow 6 \rightarrow 110$). AECI $Q = \{0, 1\}^3, X = \{0, 1\}^2, Y = \{0, 1\}^3$.
- FOLOSIM UN REGISTRU PENTRU A REGINE STAREA CURENTA SI IL INCADRIM PRINTR-UN CICLU CE CONDUC UN CLC CE IMPLEMENTEaza si λ .
VARIANTA: IN LOC DE UN REGISTRU FOLOSIM UN SISTEM DE DFF



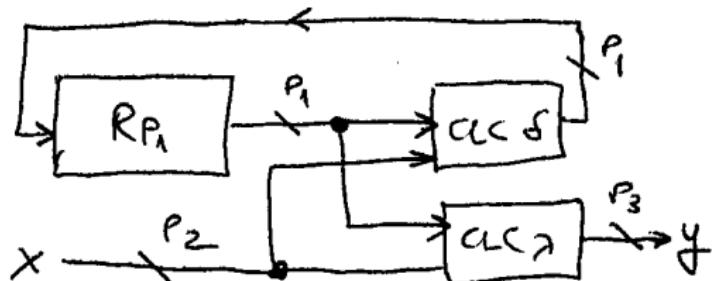
Automate finite

VARIANTA IN CARE FOLOSIM DFF + PLA/PRoM:

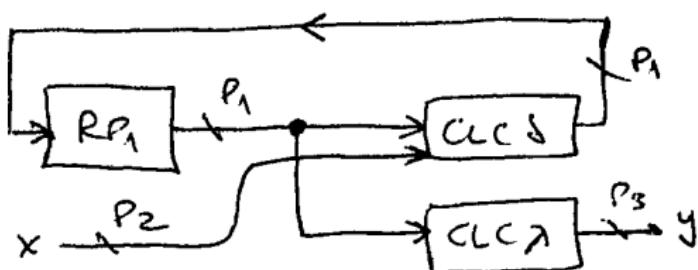


Automate finite

SE poate opti pentru CLC separate pentru f_1, f_2
AVANTAJ: CLC mai mici



AUTOMAT
MEALY



AUTOMAT
MOORE

Automate finite

ÎN CĂRȚEA P&H SE ZICE: AUT. MOORE AU AVANTAJUL CĂ POT FI MAI RAPIDE, AUT. MEALY AU AVANTAJUL CĂ POT FI MAI MICI (I.E. CU MAI PUTINE STĂRI).

CUM SE REZOLVĂ O PROBLEMA DE IMPLEMENTARE:

- DIN ENUNȚ EXTRAGEM O DESCRIEREA (SPECIFICARE SAU GRAF), OBȚINENDO TABELLE PENTRU δ, λ
- DIN TABELE CONSTRUIM CLC CA PLA/PROH ÎN MANIGRĂ ORICĂRUI; RESTUL CIRCUITULUI E STANDAR

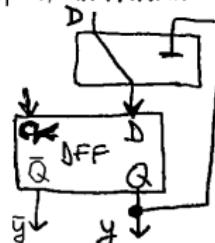
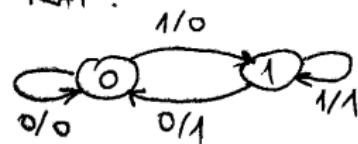
Obs: UNORI AUTOMATUL PUTEA S' SIMPLIFICA ÎNAINTE DE IMPLEMENTARE, DE EXEMPLU PUTEAM ELIMINA ANUMITE STĂRI CARE NU SUNT UTILIZATE (DE EX. SUNT INACCESEABILE); EXISTĂ ALGORITMI ÎN ACEST SENZ (VEZI LUMBAZ FORMALE).

Automatul DFF

Prezentăm câteva automate importante, ce pot fi folosite ca bistabili (flip-flop) de eficiență sporită:

ORICE DFF SE poate exprima ca un AUT. CU 2 STĂRÌ, UNDE
CLIC SE REDUCE LA FUNCȚIA IDENTICĂ A INTRĂRII
IMPLEMENTARE;

GRAF:

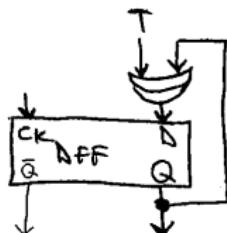


În acest caz, $Q^+ = D$, iar $y = Q$ (sau $y = D$ cu întârziere de un tact).

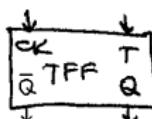
Unui automat DFF trebuie să-i dăm comanda $D = r$, pentru a-l determina să treacă din starea curentă $Q = s$ în starea nouă $Q^+ = r$ (pe scurt: $D = r$, pentru $s \rightarrow r$); la ieșire va furniza starea curentă s .

Automatul TFF

AUTOMATUL T FLIP-FLOP (TFF):



SIMBOL:

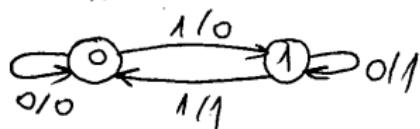


FUNCȚIONARE:

T	Q^+
0	Q
1	\bar{Q}

Dacă $T = \pi$, pe ntru $0 \rightarrow \pi$
 π , pe ntru $1 \rightarrow \pi$

GRAF:



$$Q=x=y=\{0,1\}$$

UTILIZĂRI:

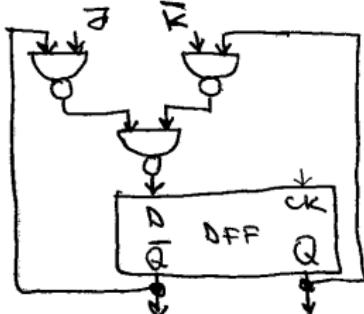
- COUNTER MODULO 2: DACĂ MENTIN $T=1$ TEMP DE MULți MULTIPLEXERI, IEȘIREA VA FI: 0, 1, 0, 1, ...

- DIVIDER DE FRECUENȚĂ

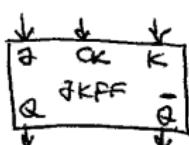


Automatul JKFF

AUTOMATUL J K FLIP-FLOP (JKFF):



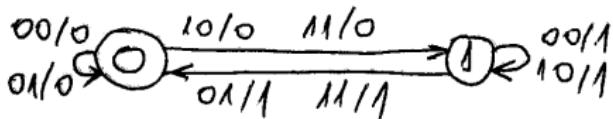
SIMBOL:



FUNCTIONARE:

J	K	Q ⁺
0	0	Q (no op)
0	1	0 (Reset)
1	0	1 (Set)
1	1	Q (switch)

GRAF:



$$Q = y = \{0, 1\}, X = \{0, 1\}^2$$

Deei
 $JK = \begin{cases} \pi, & \text{PENTRU } 0 \rightarrow r \\ -\pi, & \text{PENTRU } 1 \rightarrow r \end{cases}$

(ASTfel, pot alege în mai multe feluri comanda pt. a călăsi efect, și voi alege A.i, că să iasa cât mai simplu)

COMENTARII:

- AUTOMATELE TFF, JKFF, SPRE OFERIREA DE BISTABILII RSFF DFF STIV SĂ COMUTE ÎN CONTRARA STĂRII - LA ASTA ESTE NECESSAR CICLUL SUPLEMENTAR; ÎN PLUS, LA JKFF CAPADĂ SENZ INTRAREA 1-1.
- JKFF ESTE CEL MAI BUN FILT-PROP DG DÂNA ACUM; EL LE GENERALIZAREA PE CELFLAȚU:
 - PT. $J = \bar{E} = 1 \Rightarrow DFF$
 - PT. $J = K = T \Rightarrow TFF$
- CU JKFF SE POT CONSTRUI DIVIZORI IMPARI DE FRECVENȚĂ
- IMPLEMENTAREA GENERALĂ A UNUI AUTOMAT SE OLTRE PASA, VIZ
SISTEME DE TFF SAU JKFF + CLC (ÎN LOC DE DFF + CLC).

Observație: Dacă la implementarea generală a unui automat, în locul bistabililor DFF (care sunt 1-DS) folosim automate DFF, TFF sau JKFF (care sunt 2-DS), circuitul rezultat nu va mai fi un 2-DS, ci un 3-DS.

Aplicație la 2-DS

Exercițiu rezolvat (sinteză unui automat finit):

Construji un automat finit care recunoaște apariția secvenței 1011 în cadrul unei secvențe binare citite succesiv (i.e. furnizează la ieșire 1 d.d. ultimii 4 biți citiți formează secvența 1011).

Exemplu: IN: 0 1 0 1 0 1 1 0 1 1 1 0 0 1 1
OUT: 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0

Rezolvare:

Problema este una de limbaje formale - acolo se dău metode cu care putem construi automatul din enunț în mod algoritmic; aici îl construim intuitiv, plecând de la următoarele observații:

- Automatul trebuie să aibă 4 stări, corespunzătoare etapelor în care este recunoscut 1011: 1, 10, 101, 1011; aceste stări se pot numera q_0, q_1, q_2, q_3 și se pot asimila cu numerele 0, 1, 2, 3, pe care le putem scrie pe doi biți:

00, 01, 10, 11:



În fiecare stare se poate primi la intrare 1 sau 0, deci avem câte două arce; în fiecare caz, se poate emite la ieșire 1 = recunoscut, 0 = nerecunoscut (încă). Astfel, $Q = \{0,1\}^2$, $X = Y = \{0,1\}$.

Aplicație la 2-DS

- Drumul pe care este recunoscută o apariție a lui 1011 este:

$$q_0 \xrightarrow{1/0} q_1 \xrightarrow{0/0} q_2 \xrightarrow{1/0} q_3 \xrightarrow{1/1}$$

Restul arcelor care vor fi adăugate corespund celorlalte cazuri.

Notăm că tranziția din starea q_3 pentru intrarea $x = 1$ este singura tranziție a automatului în care se emite la ieșire $y = 1$ (deoarece doar acum se recunoaște o apariție a lui 1011); deci, arcul corespunzător este singurul din desen care va avea notat "/1" (restul vor avea notat "/0").

Aplicație la 2-DS

- Dorim ca automatul să recunoască aparițiile lui 1011 chiar dacă se suprapun parțial, de exemplu:

1 0 1 1 0 1 1

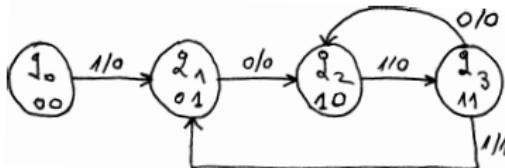
De asemenea, chiar dacă ultimii 4 biți introduși nu formează 1011, este posibil ca o parte din ei, aflați la sfârșit, să formeze 1011 împreună cu biții care vor fi citiți în continuare.

De aceea, la procesarea unei intrări $x = 1$ sau 0 , indiferent dacă a recunoscut sau nu o apariție a lui 1011, automatul trebuie să treacă într-o stare care să permită păstrarea celui mai lung sufix al secvenței introduse care ar putea fi prefix al unei (noi) apariții a lui 1011.

Aplicație la 2-DS

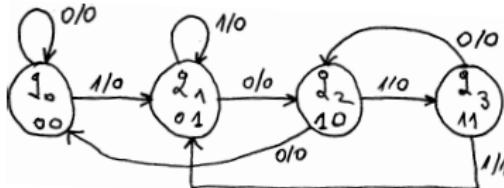
Astfel, tranziția din starea $q_3 (= 11)$ pentru intrarea $x = 1$ trebuie să ducă în starea $q_1 (= 01)$, deoarece în acest moment ultimii 4 biți introduși sunt 1011 și dintre ei putem păstra doar ultimul 1 (el ar putea fi primul 1 într-o nouă apariție a lui 1011 și ar corespunde drumului $q_0 \xrightarrow{1/0} q_1$).

De asemenea, tranziția din starea $q_3 (= 11)$ pentru intrarea $x = 0$ trebuie să ducă în starea $q_2 (= 10)$, deoarece în acest moment ultimii 4 biți introduși sunt 1010 și dintre ei putem păstra doar sufixul 10 (el ar putea prefixul 10 într-o viitoare apariție a lui 1011 și ar corespunde drumului $q_0 \xrightarrow{1/0} q_1 \xrightarrow{0/0} q_2$):



Aplicație la 2-DS

În final, graful de tranziție al automatului este:



Specificația automatului folosind tabele de valori compacte (a se vedea sfârșitul secțiunii referitoare la algebra booleană B_2) este:

$$Q = \{0, 1\}^2, X = Y = \{0, 1\}.$$

		x		$q_1^+(D_1)$	$q_0^+(D_0)$
δ :		0	1		
q_1	q_0	0	1		
0	0	0 0	0 1	0	x
0	1	1 0	0 1	\bar{x}	x
1	0	0 0	1 1	x	x
1	1	1 0	0 1	\bar{x}	x

		x		y
λ :		0	1	y
q_1	q_0	0	1	y
0	0	0	0	0
0	1	0	0	0
1	0	0	0	0
1	1	0	1	x

În cazul implementării prin DFF + PLA, biții de stare rezultați q_1^+, q_0^+ , sunt chiar comenziile D_1, D_0 ce trebuie date DFF-urilor pentru a trece în stările respective (conform regulii: " $D = r$, pentru $s \rightarrow r$ "); în tabel, am notat acest fapt între paranteze.

Aplicație la 2-DS

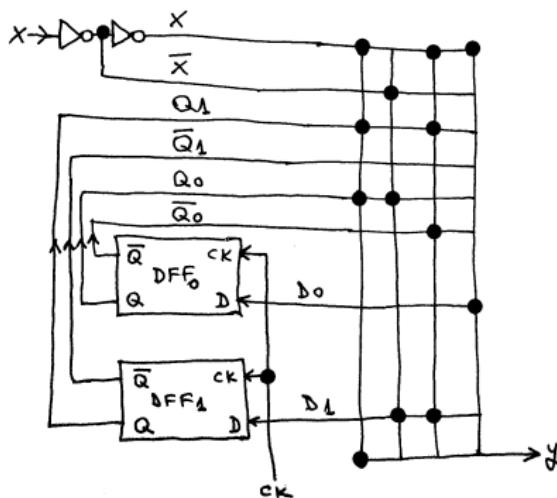
Din tabelele precedente, rezultă următoarele sume de produse minime:

$$D_1 = q_0 \bar{x} + q_1 \bar{q}_0 x$$

$$D_0 = x$$

$$y = q_1 q_0 x$$

Implementare:



Evaluarea complexității CLC: 13 puncte de contact.

Obs: \bar{Q}_1 nu are contacte și putea fi omis din circuit.

Aplicație la 2-DS

În cazul implementării prin TFF + PLA, trebuie să completăm tabelul lui δ cu coloane în care să calculăm comenziile T_1 , T_0 ce trebuie date TFF-urilor pentru a trece din stările q_1 , q_0 în respectiv stările q_1^+ , q_0^+ .

Pentru fiecare $i = 1, 2$, T_i se calculează din q_i și q_i^+ , pe baza regulii:

$$T = \begin{cases} r, & \text{pentru } 0 \rightarrow r \\ \bar{r}, & \text{pentru } 1 \rightarrow r \end{cases}$$

Obținem tabelul:

		x					
$\delta:$		$\overbrace{\quad\quad}$		q_1^+	q_0^+	T_1	T_0
q_1	q_0	0	1				
0	0	0 0	0 1	0	x	0	x
0	1	1 0	0 1	\bar{x}	x	\bar{x}	\bar{x}
1	0	0 0	1 1	x	x	\bar{x}	x
1	1	1 0	0 1	\bar{x}	x	x	\bar{x}

Rezultă următoarele sume de produse minimale:

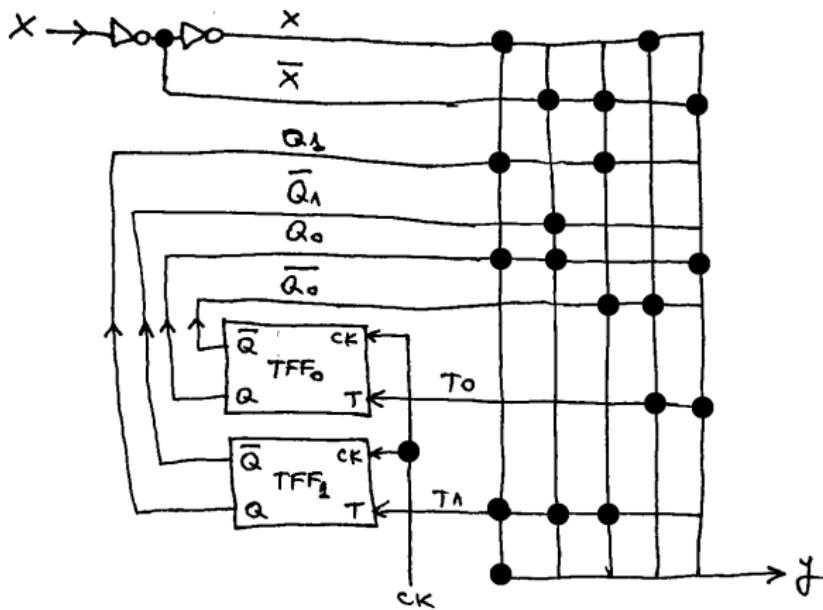
$$T_1 = \overline{q_1} q_0 \bar{x} + q_1 \overline{q_0} \bar{x} + q_1 q_0 x$$

$$T_0 = \overline{q_0} x + q_0 \bar{x}$$

$y = q_1 q_0 x$ (a rămas la fel ca mai înainte).

Aplicație la 2-DS

Implementare:



Evaluarea complexității CLC: 19 puncte de contact.

Aplicație la 2-DS

În cazul implementării prin JKFF + PLA, trebuie să completăm tabelul lui δ cu coloane în care să calculăm comenziile J_1, K_1, J_0, K_0 ce trebuie date JKFF-urilor pentru a trece din stările q_1, q_0 în respectiv stările q_1^+, q_0^+ . Pentru fiecare $i = 1, 2$, J_i, K_i , se calculează din q_i și q_i^+ , pe baza regulii:

$$JK = \begin{cases} r_-, & \text{pentru } 0 \rightarrow r \\ -\bar{r}, & \text{pentru } 1 \rightarrow r \end{cases}$$

Obținem tabelul (am păstrat doar coloanele relevante):

		x							
$\delta:$		$\overbrace{\quad\quad}$		q_1^+	q_0^+	J_1	K_1	J_0	K_0
q_1	q_0	0	1						
0	0	0 0	0 1	0	x	0	-	x	-
0	1	1 0	0 1	\bar{x}	x	\bar{x}	-	-	\bar{x}
1	0	0 0	1 1	x	x	-	\bar{x}	x	-
1	1	1 0	0 1	\bar{x}	x	-	x	-	\bar{x}

Au rezultat niște coloane ce conțin valori indiferente " $-$ ", pe care trebuie să le înlocuim cu ceva concret pentru a putea obține din aceste coloane o scriere a lui J_i, K_i , $i = 1, 2$, ca sumă de produse.

Aplicație la 2-DS

Înlocuirea se va face a.î. sumele de produse rezultate să fie cât mai simple (cât mai puțini termeni, cât mai puține variabile), conform criteriilor prezentate la sfârșitul secțiunii referitoare la algebra booleană B_2 .

Am adăugat coloanele obținute la sfârșitul tabelului:

		x																											
$\delta:$		q_1		q_0		0		1		q_1^+		q_0^+		J_1		K_1		J_0		K_0		J_1		K_1		J_0		K_0	
q_1	q_0	0	1	0	1	q_1^+	q_0^+	0	x	0	-	x	-	0	\bar{x}	x	\bar{x}	0	\bar{x}	x	\bar{x}	0	\bar{x}	x	\bar{x}				
0	0	0	0	0	1	0	x	0	-	x	-	-	\bar{x}	0	\bar{x}	x	\bar{x}	0	\bar{x}	x	\bar{x}	0	\bar{x}	x	\bar{x}				
0	1	1	0	0	1	\bar{x}	x	\bar{x}	-	-	-	-	\bar{x}	0	\bar{x}	x	\bar{x}	0	\bar{x}	x	\bar{x}	0	\bar{x}	x	\bar{x}				
1	0	0	0	1	1	x	x	-	\bar{x}	x	-	x	-	0	\bar{x}	x	\bar{x}	0	\bar{x}	x	\bar{x}	0	\bar{x}	x	\bar{x}				
1	1	1	0	0	1	\bar{x}	x	-	x	-	x	-	\bar{x}	0	\bar{x}	x	\bar{x}	0	\bar{x}	x	\bar{x}	0	\bar{x}	x	\bar{x}				

Rezultă următoarele sume de produse minimale:

$$J_1 = q_0 \bar{x}, \quad J_0 = x, \quad y = q_1 q_0 x \text{ (a rămas la fel ca mai înainte).}$$

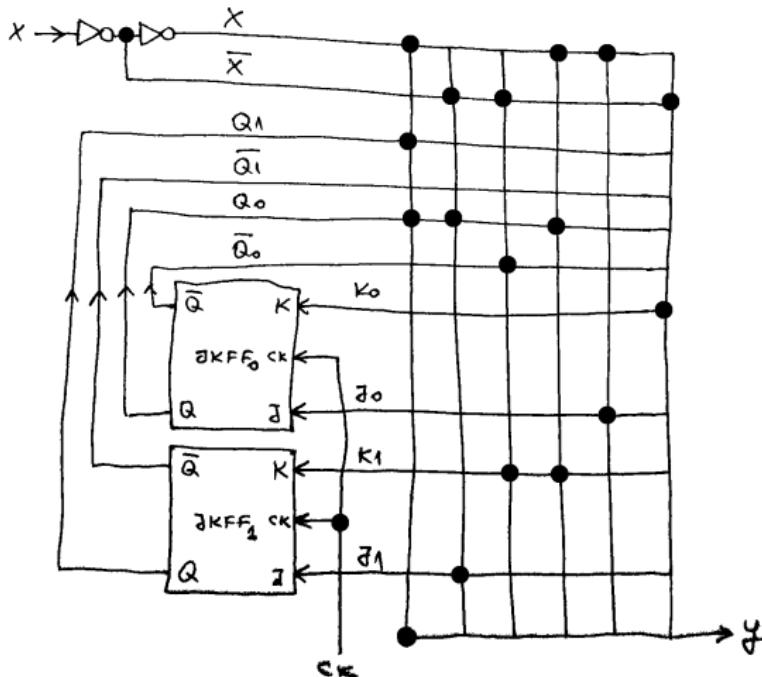
$$K_1 = \bar{q}_0 \bar{x} + q_0 x, \quad K_0 = \bar{x},$$

Obs: Dacă în coloana J_1 am fi înlocuit cele două " - " cu 0, ar fi rezultat

$J_1 = \bar{q}_1 q_0 \bar{x}$ (un termen cu trei variabile, în loc de două); de asemenea, dacă în coloana J_0 am fi înlocuit cele două " - " cu 0, ar fi rezultat $J_0 = \bar{q}_0 x$ (un termen cu două variabile, în loc de una).

Aplicație la 2-DS

Implementare:



Evaluarea complexității CLC: 17 puncte de contact.

Aplicație la 2-DS

Observații:

- Mai sus, am încercat să minimizăm CLC minimizând fiecare sumă de produse în parte.

Un alt criteriu de minimizare ar fi să înlocuim valorile indiferente " _ " a.î. sumele de produse rezultate să aibă cât mai mulți termeni comuni.

- În mod normal, implementarea cu JKFF ar fi trebuit să conducă la circuitul cel mai simplu (cel mai mic număr de puncte de contact), deoarece avem mai multe variante de valori J, K, dintre care putem alege una cât mai bună.

O explicație de ce nu s-a întâmplat așa poate fi complexitatea mare a instrumentului folosit - JKFF are două comenzi, spre deosebire de DFF și TFF, care au doar una. Aceasta adaugă în mod artificial complexitatea circuitului.

În general, instrumentele eficiente dar complexe își evidențiază avantajele în cazul sarcinilor complexe (circuite complexe, cu multe puncte de contact). În cazul sarcinilor simple (cum este circuitul cerut în această problemă) sunt mai bune instrumentele simple (în cazul de față DFF - pentru el am obținut numărul minim de puncte de contact).

Aplicație la 2-DS

Și în dezvoltarea de software, dacă avem de scris un program complex, cu mii de linii de cod, o ierarhie complexă de clase, multe fișiere sursă, este de preferat să folosim un IDE complex, care să ne ofere multiple instrumente de a gestiona proiecte de mare anvergură.

Dacă însă avem de scris un program de 10 - 20 linii, un IDE complex se poate dovedi incomod - trebuie să creăm un proiect, să facem mai multe setări, pot apărea probleme de incompatibilitate dacă dorim să recompilăm proiectul cu o altă versiune a IDE-ului, etc.

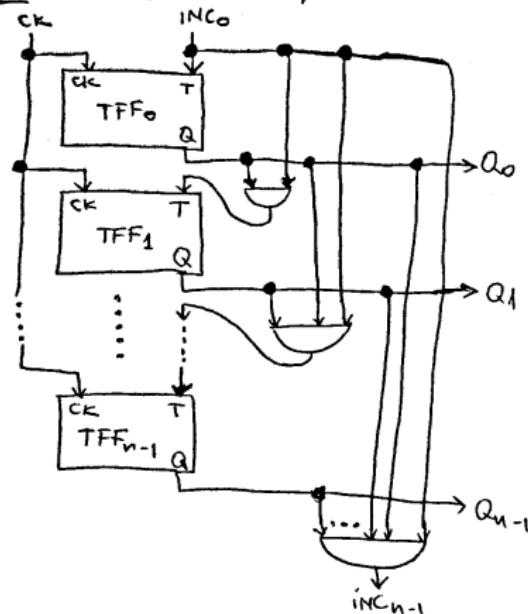
În acest caz este mai eficient să folosim instrumente simple: un editor de fișiere text (ex. "gedit") pentru codul sursă și un compilator în mod linie de comandă (ex. "gcc").

Counter

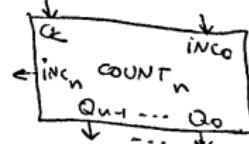
În continuare, prezentăm alte câteva automate importante, implementate ca 2-DS:

Numărător (Counter):

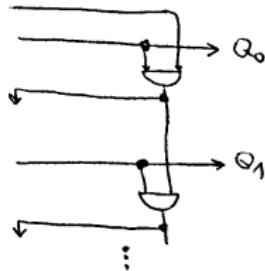
Ex: Counter mod 2^n , folosind TFF-uri:



SIMBOL:



Obț: Purtem punct "si" urmări
și în serie:

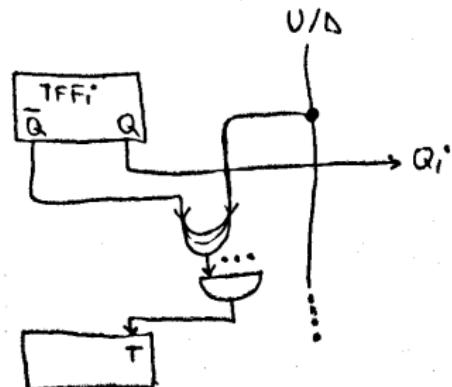
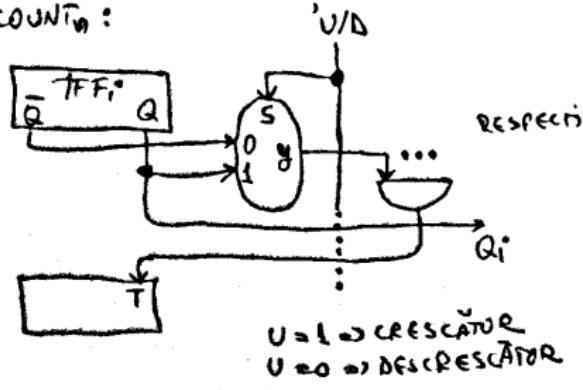


Counter

- OBS:
- Iatăcă este că se schimbă valoarea bitului: următor
doar dacă în spate erau doar 1-uri
Aici ne ajută faptul că TFF sunt numărători până la 1
 - COUNTER-LE SE POT DEFINI și RECURSIV ($COUNT_{n-1}$, în serie
cu un TFF)
 - Dacă toate intrările în poartile AND sunt din \bar{Q} (în loc
de Q), obținem un counter descrescător

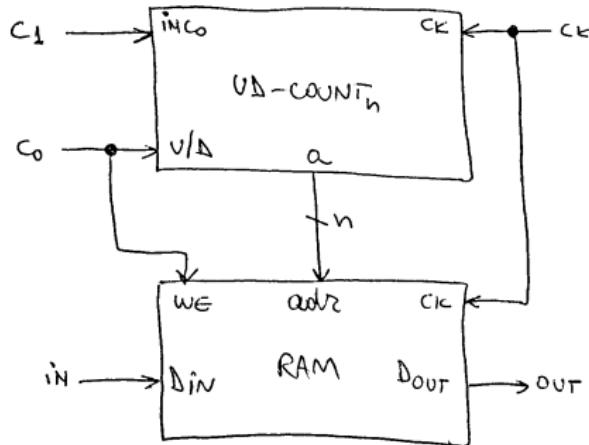
Counter

Se poate adăuga counterului o intrare suplimentară U/D care să indice dacă va număra crescător/descrescător și care va alege cu unMUX între Q și \bar{Q} sau va face XOR cu Q ; obțin circuitul UD -counter:



Stivă

Stivă:



FUNCTIONARE:

C_0	C_1	F_{C1}
0	0	nop
1	1	push (VA-COUNT _n este incrementat iar DATA IN este scriuta (WE=1) la adresa indicata de nouu continut al numaratorului)
0	1	pop (DATA de la adresa indicata de numarator este emisa la OUT iar numaratorul este decrementat la nouu valoare al sauui)

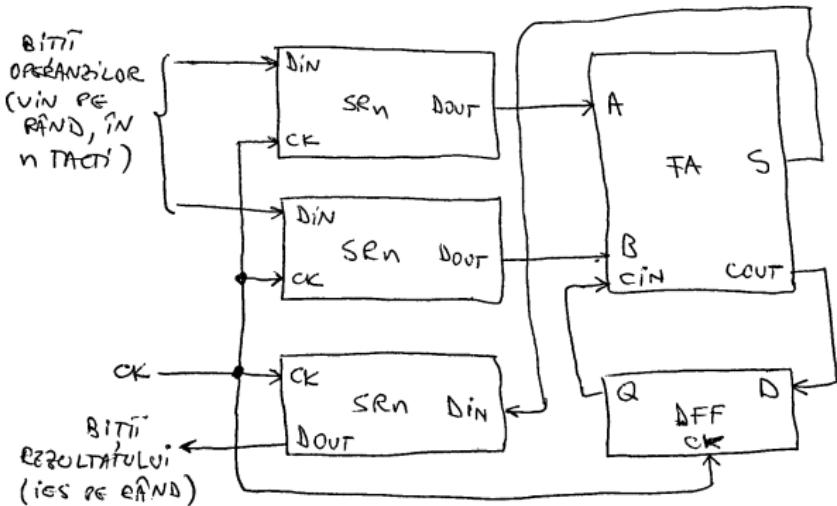
0 0 nō op

1 1 push (VA-COUNT_n este incrementat iar DATA IN este scriuta (WE=1) la adresa indicata de nouu continut al numaratorului)

0 1 pop (DATA de la adresa indicata de numarator este emisa la OUT iar numaratorul este decrementat la nouu valoare al sauui)

Sumator serial

Sumator serial:



- OBS.:
- ADUNĂ SECVENTIAL ÎN N TACI'S CU DFF SE PĂSTREAZĂ/TRANSMITE CARRY-UL DE LA O POZIȚIE LA ALTA ÎN FAZĂ CU SRN
 - SE PĂSTREAZĂ ÎMPĂRȚIND OPERANZIÎNU ÎN BIJ' C'SN GRUPURI DE M BIJ'S ÎN LOC DE SRN FOLOSIM SPRNxM
 - ÎN LOC DE FA FOLOSIM ADDmj
 - rezultat: SUMATOR SERIAL-PARALEL

Sumator prefix

Sumator prefix:

PREZUMUDEM CĂ VINDE O SECVENTĂ DE NUMERE: x_1, \dots, x_p

VIZAM SĂ CALCULAM SUCCESIV SUMELE PREFIXE: $y_1 = x_1$

$$y_2 = x_1 + x_2$$

⋮

$$y_p = x_1 + \dots + x_p$$

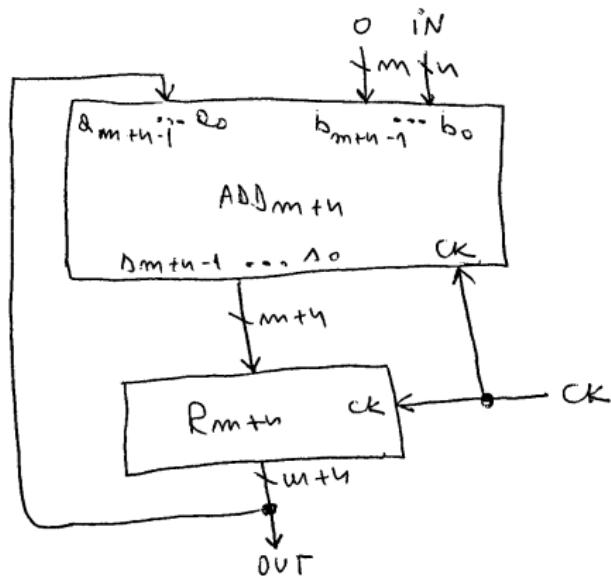
DACĂ NUMERELE x_i SUNT PE M BIȚI,
SUMA AR DUREA DEPĂSI n BIȚI, ȘI ÎNȚELE

FOLOSIM LOCATII DE $m+n$ BIȚI, PENTRU

UN m CONVENABIL - EX: $m+n = \lceil \log_2 p \cdot (2^n - 1) \rceil$

Sumator prefix

Circuitul este:



OBS: AUTOMATUL ARE 2^{M+N} STĂRI

TODO: Alte automate importante, implementate ca 2-DS.

Cuprins

1 Performația calculatoarelor

Conceptul de performanță
Măsurarea performanței

2 Aritmetică sistemelor de calcul

Reprezentarea numerelor în matematică
Reprezentarea numerelor în calculator
Reprezentarea numerelor naturale ca întregi fără semn
Reprezentarea numerelor întregi în complement față de 2
Reprezentarea numerelor reale în virgulă mobilă

3 Logica pentru calculatoare

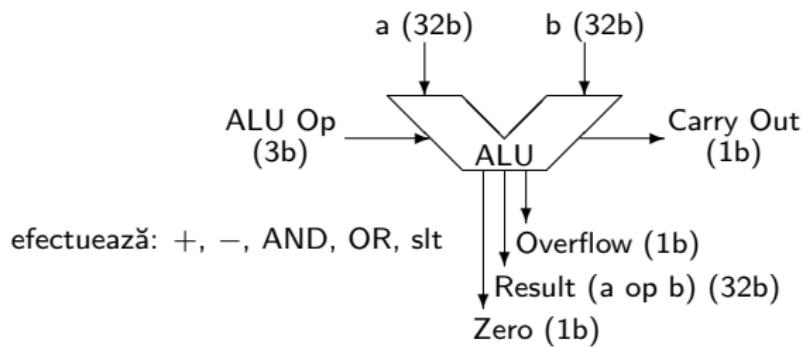
Algebre booleene
Funcții booleene
Particularizare la cazul B_2

4 Circuite logice

Circuite logice, Sisteme digitale
0-DS (Circuite combinaționale, Funcții booleene)
1-DS (Memorii)
2-DS (Automate finite)
Algoritmi de înmulțire și împărțire hardware
3-DS (Procesoare) și 4-DS (Calculatoare)
 n -DS, $n > 4$

Unitatea aritmetico-logică (ALU)

Reamintim unitatea aritmetico-logică (ALU), construită mai devreme (ca 0-DS):



Unitatea aritmetico-logică (ALU)

ALU efectuează operații într-un ciclu.

În ALU sunt implementate câteva operații simple (+, -, AND, OR, slt), ceea ce permite efectuarea de cicluri scurte.

Înmulțirea și împărțirea sunt operații complexe.

Ele ar putea fi implementate într-un ciclu, în ALU, dar atunci structura acestuia s-ar dezvolta foarte mult pe verticală, ceea ce ar duce la lungirea ciclului.

De aceea, înmulțirea și împărțirea sunt implementate într-o unitate separată, folosind mai multe cicluri ALU, conform unor anumiți algoritmi.

Prezentăm în continuare câțiva algoritmi de înmulțire și împărțire.

Înmulțire - metoda 1

Reproduce metoda clasică, aplicată manual:

De ex. pt. a calcula $\overbrace{100}^D \times \overbrace{101}^I = \overbrace{11110}^P$ având dim. word-ului $n = 4$, efectuăm:

$$\begin{array}{r} 110 \times \\ 101 \\ \hline 110 \leftarrow \text{Parcurem I dr.} \rightarrow \text{stg. și pentru fiecare 1 copiez D shiftat;} \\ 110 \qquad \text{apoi adun.} \\ \hline 11110 \leftarrow P \text{ are dim. max. } 2n. \end{array}$$

Reformulare cu acțiuni mai apropiate de operațiile mașină:

Shiftez de n ori I la dreapta și D la stânga;

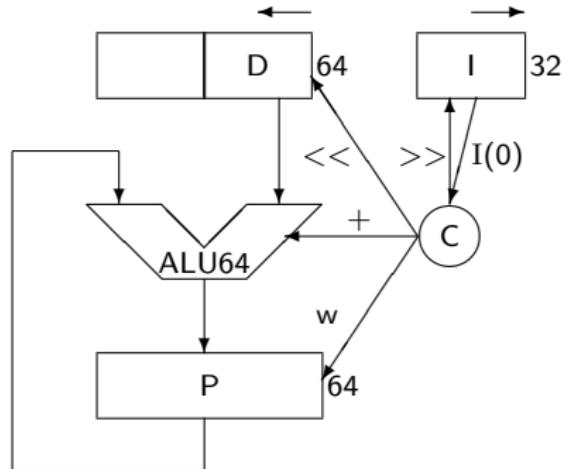
de fiecare dată când ultimul bit din I (adică $I(0)$) este 1, adun D la P
(în poziția curentă).

Înmulțire - metoda 1

```

D64, I32, P64 := 0
repetă de 32 ×:
  dacă I(0) = 1
    P := P + D
  □
  D := D << 1
  I := I >> 1
  □
}

```



Aplicație: Calculați $6 \times 5 = 30$, $n = 4$. Completați tabelul următor, fiecare coloană dublă conține valorile regiștrilor LA SFÂRȘITUL unei etape:

	Initial		Iterația 1		Iterația 2		Iterația 3		Iterația 4	
>>	I	-	0101	-		-		-		
<<	D	0000	0110							
+	P	0000	0000							

Înmulțire - metoda 1

Rezolvare:

		Initial	Iterația 1	Iterația 2	Iterația 3	Iterația 4	
>>	I	-	<u>0101</u>	-	<u>0010</u>	-	<u>0001</u>
<<+	D	0000	0110	0000	1100	0001	1000
	P	0000	0000	0000	0110	0000	0110

Înmulțire - metoda 1

Analiza algoritmului:

Se efectuează $32 \text{ pași} \times 3 \text{ operații} \approx 100 \text{ cicluri pe instrucțiune.}$

Analizele statistice arată că $+$, $-$ sunt de $5 - 100 \times$ mai frecvente decât $*$.

Atunci, cu regula "execută rapid operațiile frecvente", rezultă că această implementare pentru înmulțire este acceptabilă.

Totuși, dorim și putem obține implementări mai eficiente.

Înmulțire - metoda 2

Obs. că prima jumătate a lui D este nefolosită (e necesară ca să putem aduna pe 64b).

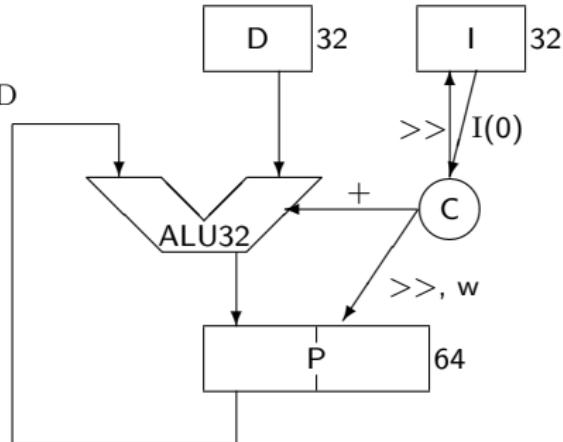
Soluția: facem D de 32b și în loc să deplasăm D deasupra lui P, deplasăm P pe sub D; întotdeauna vom aduna D cu prima jumătate a lui P (adunare pe 32b).

Înmulțire - metoda 2

```

D32, I32, P64 := 0
repetă de 32 ×:
  dacă I(0) = 1
    P[63 – 32] := P[63 – 32] + D
    □ P := P >> 1
    □ I := I >> 1
  □

```



Aplicație: Calculați $6 \times 5 = 30$, $n = 4$. Completați tabelul următor,
fiecare coloană dublă conține valorile regiștrilor LA SFÂRȘITUL unei etape:

	Initial	Iterația 1	Iterația 2	Iterația 3	Iterația 4
>> I	-	0101	-	-	-
D	0110	-	-	-	-
>> P	0000	0000			

Înmulțire - metoda 2

Rezolvare:

		Initial	Iterația 1	Iterația 2	Iterația 3	Iterația 4	
>>	I	-	<u>0101</u>	-	<u>0010</u>	-	<u>0001</u>
	D	0110	-	0110	-	0110	-
+>>	P	0000	0000	0011	0000	0001	1000
		0110		0111			

Înmulțire - metoda 3 (finală)

Obs. că jumătatea inferioară (LO) a lui P se consumă spre dreapta în același ritm ca I.

Atunci putem pune I în jumătatea inferioară a lui P; în rest este la fel.

Înmulțire - metoda 3 (finală)

D₃₂, P₆₄ := [0, ..., 0, I₃₂]

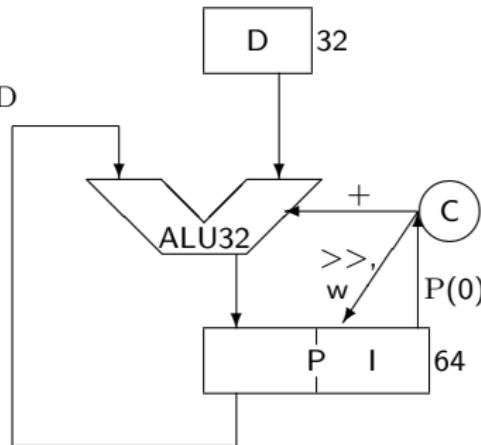
repetă de 32 ×:

dacă P(0) = 1

P[63 – 32] := P[63 – 32] + D

P := P >> 1

□



Aplicație: Calculați $6 \times 5 = 30$, $n = 4$. Completați tabelul următor,
fiecare coloană dublă conține valorile regiștrilor LA SFÂRȘITUL unei etape:

	Ințial	Iterația 1	Iterația 2	Iterația 3	Iterația 4
D	0110	-	-	-	-
P,I	0000	0101			

Înmulțire - metoda 3 (finală)

Rezolvare:

	Initial	Iterația 1	Iterația 2	Iterația 3	Iterația 4
D	0110	-	0110	-	0110
P,I	0000	<u>0101</u>	<u>0011</u>	<u>0010</u>	<u>0001</u>
	0110		0111		

Înmulțirea cu semn

Înmulțirea cu semn este asemănătoare, cu următoarele diferențe:

- se lucrează pe 31 biți, deci se fac 31 iterații, neglijând bitul de semn;
- în final, semnul produsului este XOR între biții de semn ai factorilor (deci este 1 dacă au semne diferite ("-") și 0 dacă au același semn ("+"));
- algoritmul 3 (final) funcționează corect și pentru numere cu semn, dar la shiftare trebuie extins semnul produsului (shiftare aritmetică).

Înmulțire - algoritmul Booth

Face câteva îmbunătățiri plecând de la următoarele observații:

- Înmulțirea cu un grup de 0 din I se reduce la shiftări;
- Înmulțirea cu un grup de 1 din I: $\underbrace{1\dots 1}_k = 2^k - 1 = \underbrace{10\dots 0}_k - 0\dots 01$ se reduce la

adunarea lui D shiftat cu k pentru bitul 1 din grupul $10\dots 0$ și o scădere a lui D pentru bitul 1 din grupul $0\dots 01$.

D I

$$\text{De ex. } 0010 * \underbrace{0110}_I = 0010 * 1000 - 0010 * 0010 \\ = 1000 - 0010$$

Practic shiftăm I la dreapta cu câte 1 și luăm decizii în funcție de cu încep/continuă/se termină grupurile de 1 sau 0 (algoritmul și circuitul seamănă cu cel de la metoda 3):

01100 → se adaugă un bit fictiv 0, pentru a avea un context;

01100 → se shiftează P;

00110 → se scade D (începe un grup de 1 și se face întâi scăderea lui D);

00011 → se shiftează P (continuă grupul de 1);

00001 → se adună D (se termină grupul de 1 și se face adunarea lui D);

00000 → se shiftează P.

Înmulțire - algoritmul Booth

D₃₂, P[63, ..., -1] := $\overbrace{[0, \dots, 0]}^{32}, I_{32}, 0$
repetă de 32 ×:

testează (P(0), P(-1))

cazul 01: P[63 - 32] := P[63 - 32] + D

cazul 10: P[63 - 32] := P[63 - 32] - D

□

P := P >> 1 (shift aritmetic)

□

Algoritmul și circuitul le adaptează pe cele de la metoda 3.

Algoritmul funcționează corect și pentru numere negative și e performant (se poate folosi pe grupuri de biți pentru a construi înmulțitoare rapide.)

Aplicație: Calculați $5 \times 6 = 30$ ($101_2 \times 110_2 = 11110_2$), $n = 4$.

Completați tabelul următor, fiecare linie conține valorile regiștrilor LA SFÂRȘITUL unei etape:

	D	I	P
Inițial	0000 0101	0110	0
Iterația 1			
Iterația 2			
Iterația 3			
Iterația 4			

Înmulțire - algoritmul Booth

Rezolvare:

Inicital	0000 0101	0110 0	
Iterația 1	0000 0101	0011 0	
Iterația 2	1011 1101 0101	0011 1001 0	1
Iterația 3	1110 0101	1100 0	1
Iterația 4	0011 0001	1100 1110	0

Împărțire - metoda 1

Trebuie efectuat $D:I \rightarrow C,R$ ($D = I \times C + R$, $R < I$).

Metoda 1 reproduce metoda clasică, aplicată manual, de exemplu:

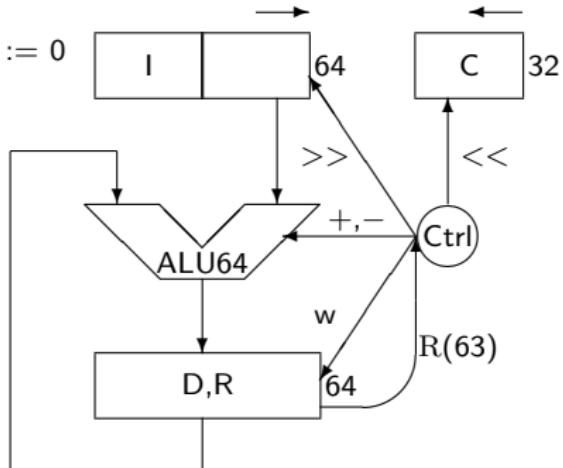
$$\begin{array}{r} D = 1001010 \\ \underline{-1000} \quad | \begin{array}{l} 1000 \\ 1001 \end{array} = I \\ \begin{array}{c} 10 \\ 101 \\ 1010 \\ \underline{-1000} \\ 10 \end{array} = R \end{array}$$

D.p.v al mașinii:

- a vedea dacă I "se cuprinde" revine la a scădea $D - I$ și a compara cu 0 (i.e. a testa bitul cel mai semnificativ);
 - dacă $d_0 \geq 0$, adăugăm 1 la C ;
 - dacă $d_0 < 0$, adunăm $D + I$ la loc și adăugăm 0 la C ;
- adăugarea unei cifre la C înseamnă $<< 1 + \text{cifră}$;
- coborârea cifrei următoare înseamnă shiftarea lui I pe sub D la dreapta (metoda 1) sau a lui D pe deasupra lui I la stânga (metodele 2,3), pentru a face altă suprapunere la scădere.

Împărțire - metoda 1

$R_{64} := D$, $I_{64} := [I, \underbrace{0, \dots, 0}_{32}]$, $C_{32} := 0$
 repetă de 33 x:
 R := R - I
 dacă $R \geq 0$ (i.e. $R(63) = 0$)
 C := C << 1 + 1
 - altfel
 R := R + I
 C := C << 1 + 0
 □
 I := I >> 1
 □



Aplicație: Calculați 7:3 (adică 111:11), $n = 4$. Completați tabelul următor, fiecare coloană dublă conține valorile regiștrilor LA SFÂRSITUL unei etape:

	Initial	Iterația 1	Iterația 2	Iterația 3	Iterația 4	Iterația 5
<<	C - 0000	-	-	-	-	-
R,D	0000 0111					
>>	I 0011 0000					

Împărțire - metoda 1

Rezolvare:

	Initial	Iterația 1	Iterația 2	Iterația 3	Iterația 4	Iterația 5
<< C	- 0000	- 0000	- 0000	- 0000	- 0001	- 0010
R,D	0000 0111	0000 0111	0000 0111	0000 0111	0000 0001	0000 0001
>> I	0011 0000	0001 1000	0000 1100	0000 0110	0000 0011	0000 0001

1101 0111 1110 1111 1111 1011 0000
0001 1111 1110 0001 1111 1110 0001

Împărțire - metoda 1

Obs. că doar jumătate din I conține informație utilă; atunci, putem folosi un ALU32 și să shiftăm R pe deasupra lui I la stânga.

Apoi obs. că algoritmul nu poate produce un 1 în prima fază, căci rezultatul ar fi prea lung pentru C și n-ar încăpea într-un registru de 32b (avem $n + 1$ iterații, deci am avea un cât de forma $\underbrace{1\dots}_{33\text{ cifre}}$ și nu ar încăpea).

Soluția: se permute operațiile de shiftare și scădere (se face întâi shiftarea și apoi scăderea), eliminându-se o iterație; la sfârșit restul este în jumătatea stângă a lui R. Se obține **metoda 2**, dar nu o prezentăm, ci trecem direct la **metoda 3**, unde în plus se pune C în jumătatea dreaptă a lui R - cum R și C se shiftează sincron cu 1 la stânga, nu se pierde nimic din R, C.

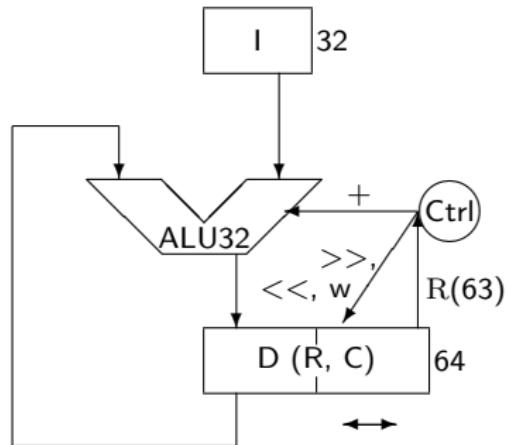
Singura problemă este că la sfârșit jumătatea stângă a lui R este prea shiftată și se shiftează la dreapta cu 1.

Împărțire - metoda 3

```

R64 := D, I32
R := R << 1
repetă de 32 ×:
  R[63 - 32] := R[63 - 32] - I
  dacă R ≥ 0 (i.e. R(63) = 0)
    R := R << 1 + 1
  altfel
    R[63 - 32] := R[63 - 32] + I
    R := R << 1 + 0
  ┌───┐
  └───┘
  R[63 - 32] := R[63 - 32] >> 1

```



$R[63 - 32] = \text{restul}, R[31 - 0] = \underline{\text{câtul}}$

Aplicație: Calculați 7:3 (adică 111:11), $n = 4$, completând tabelul următor:

Initial	0000 0011	0111
R << 1		
Iterația 1		
Iterația 2		

Iterația 3		
Iterația 4		
R[63 - 32] >> 1		

Împărțire - metoda 3

Rezolvare:

	Inițial	0000	0111
	0011		
R << 1	0000	1110	
	0011		
Iterată 1	<u>1101</u>		
	0001	1100	
	0011		
Iterată 2	<u>1110</u>		
	0011	1000	
	0011		

	0000	0001
Iterată 3	0001	0001
	0011	
Iterată 4	<u>1110</u>	
	0010	0010
	0011	
R[63 – 32] >> 1	0001	0010

Cuprins

1 Performația calculatoarelor

Conceptul de performanță
Măsurarea performanței

2 Aritmetică sistemelor de calcul

Reprezentarea numerelor în matematică
Reprezentarea numerelor în calculator
Reprezentarea numerelor naturale ca întregi fără semn
Reprezentarea numerelor întregi în complement față de 2
Reprezentarea numerelor reale în virgulă mobilă

3 Logica pentru calculatoare

Algebre booleene
Funcții booleene
Particularizare la cazul B_2

4 Circuite logice

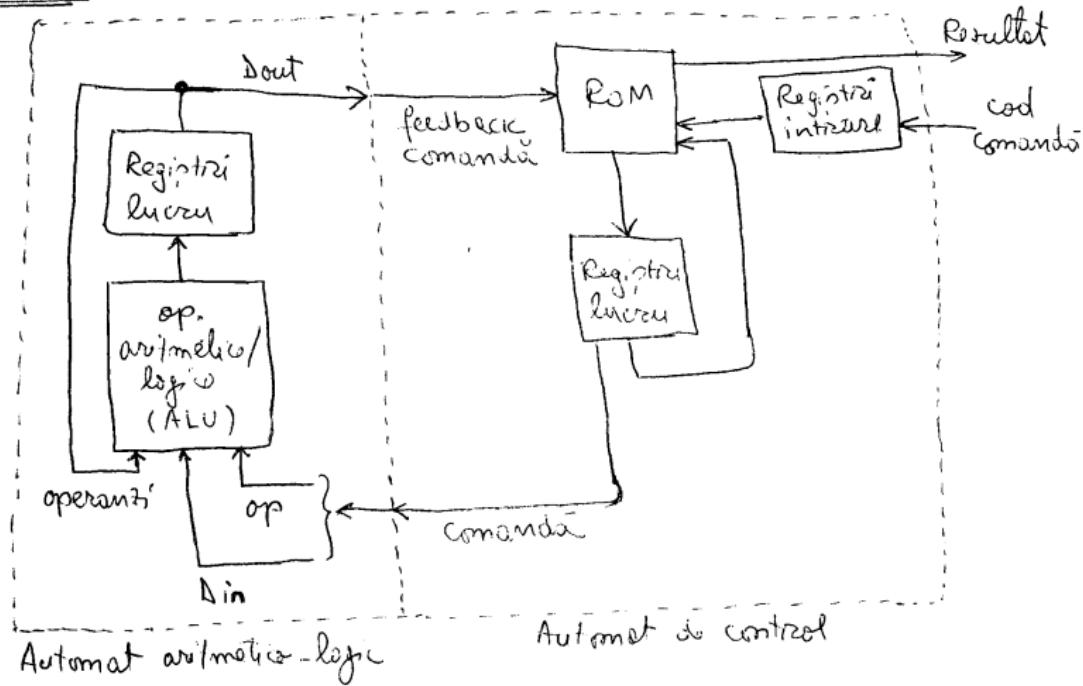
Circuite logice, Sisteme digitale
0-DS (Circuite combinaționale, Funcții booleene)
1-DS (Memorii)
2-DS (Automate finite)
Algoritmi de înmulțire și împărțire hardware
3-DS (Procesoare) și 4-DS (Calculatoare)
 n -DS, $n > 4$

Procesor:

3-DS (Procesoare)

UN PROCESOR ESTE UN CIRCUIT CE LUMA ÎNTR-UN CICLU UN AUTOMAT ARITMETICO-LOGIC CU UN AUTOMAT DE CONTROL

EXEMPLU:



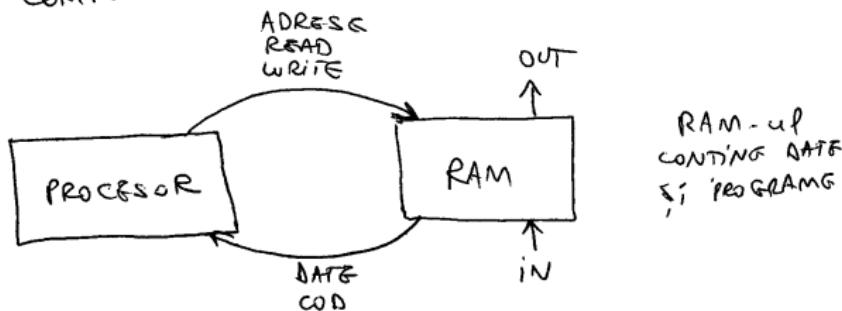
3-DS (Procesoare)

OBS: Detaliile de organizare a procesorului pot差别 mult
de la arhitectură la alta; în continuare vom studia
asemenea detaliu în cazul arhitecturii MIPS.

4-DS (Calculatoare)

Calculator:

EXEMPLU: COMPUTER-UL:



OBS: PROCESSORUL SE poate închipui și cu:

- CLC (0-DS)
- STIVĂ (2-DS)
- CO-PROCESSOR (3-DS)

OBS: ÎN CASUL CALCULATORELOR DE TAILORE DE ORGANIZARE
POTE DIFERI MULT DE LA O ARHITECTURĂ LA ALTRĂ.

TODO: Mai multe detalii și exemple despre 3-DS și 4-DS.

Cuprins

1 Performanța calculatoarelor

Conceptul de performanță
Măsurarea performanței

2 Aritmetică sistemelor de calcul

Reprezentarea numerelor în matematică
Reprezentarea numerelor în calculator
Reprezentarea numerelor naturale ca întregi fără semn
Reprezentarea numerelor întregi în complement față de 2
Reprezentarea numerelor reale în virgulă mobilă

3 Logica pentru calculatoare

Algebre booleene
Funcții booleene
Particularizare la cazul B_2

4 Circuite logice

Circuite logice, Sisteme digitale
0-DS (Circuite combinaționale, Funcții booleene)
1-DS (Memorii)
2-DS (Automate finite)
Algoritmi de înmulțire și împărțire hardware
3-DS (Procesoare) și 4-DS (Calculatoare)
 n -DS, $n > 4$

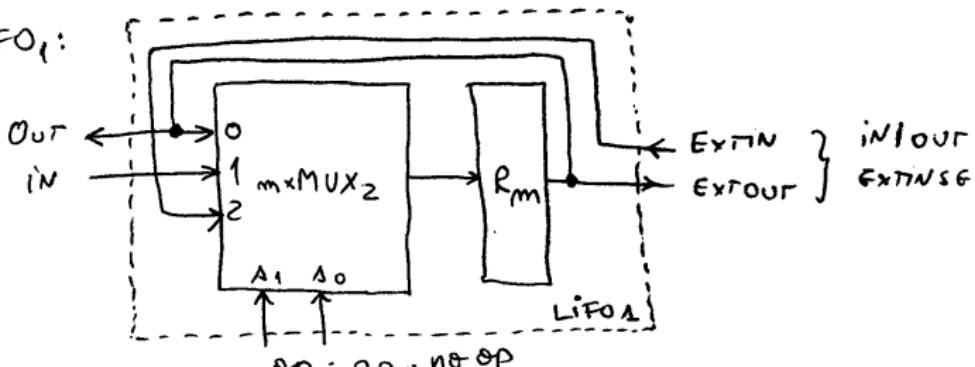
Stivă ca n-DS

Stivă ca n-DS:

UNELTE CIRCUITE SE POT ORGANIZA CA n-DS CU $n > 4$, OMAR
DACĂ SE POT ORGANIZA SÌ MAI SIMPLU.

Exemplu: STIVĂ ORGANIZATĂ CA n-DS (AM VĂZUT CĂ SE PODEA
IMPLEMENTA SÌ CA 2-DS):

LIFO₁:



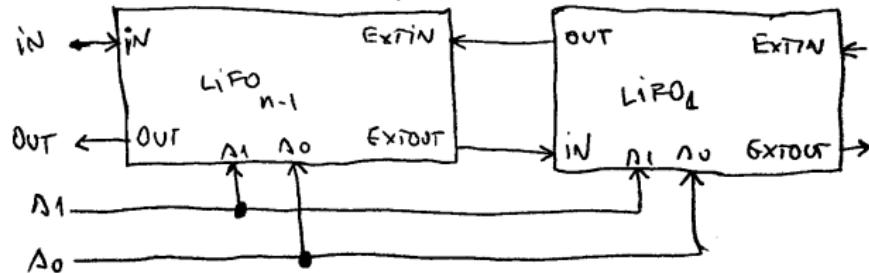
Op : 00 = no op

01 = push

10 = pop

Stivă ca n-DS

LIFO_n:



Deci, LIFO_n este n-DS.

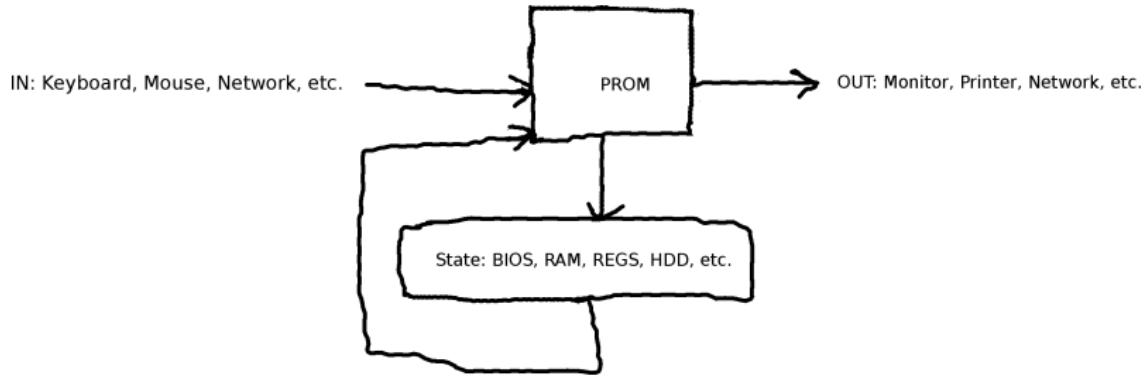
Acest exemplu ne arată ca prin creșterea numărului de niveluri de cicluri nu obținem neapărat circuite mai "deștepte" (o stivă este un circuit simplu, am văzut mai devreme că se poate construi și ca 2-DS).

Un avantaj al creșterii numărului de niveluri de cicluri este reducerea complexității structurale a circuitului (numărul de componente).

De exemplu, la un calculator ușual, funcționarea este deterministă iar comportamentul la un moment dat este unic determinat de conținutul curent din sistem și de intrarea curentă.

Astfel, am putea construi orice calculator ușual ca un 2-DS (automat finit), în care:

- elementul de stare (de exemplu un registru foarte mare) conține toată informația curent stocată în sistem, inclusiv datele din BIOS, RAM, regisitrii procesorului, hard disk, etc.
- liniile de intrare furnizează toată informația venită la un moment dat de la tastatură, mouse, rețea, etc.
- liniile de ieșire furnizează toată informația emisă la un moment dat spre monitor, imprimantă, rețea, etc.
- circuitul combinațional (de exemplu un PROM foarte mare) calculează, în fiecare moment, din informația stocată în sistem și cea venită la intrare, noua informație care va fi stocată în sistem și informația care va fi emisă la ieșire.



Un asemenea circuit ar fi însă inacceptabil de mare (ca număr de componente) - de exemplu, elementul de stare ar putea avea ≥ 1 TB (pentru a putea stoca conținutul hard disk-urilor) și atunci circuitul combinațional, dacă ar fi un PROM, ar avea $\geq 2^1$ TB "AND"-uri.

Prin creșterea numărului de niveluri de cicluri, la o aceeași procesare se fac mai multe treceri prin circuit iar unele componente se refolosesc.

Aceasta permite construcția același tip de echipament (de exemplu calculator), care să facă același tip de procesare, cu mai puține componente.

TODO (au fost predate la curs sau laborator și au fost furnizate în fișiere separate, le-am indicat mai jos):

- Arhitectura MIPS
(fișierele "_mips1b.txt", "_mips2b.txt", "_mips3c.txt").
- Procesorul MIPS cu 1 ciclu per instrucțiune (fișierul "opi.pdf").
- Procesorul MIPS cu cicluri multiple (fișierul "iccm_mic.pdf").