

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

Algebrie booleană

Funcții booleană

Particularizare la cazul B_2

4 Circuite logice

Circuite logice, Sisteme digitale

0-DS (Circuite combinaționale, Funcții booleană)

1-DS (Memorii)

2-DS (Automate finite)

Algoritm 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.

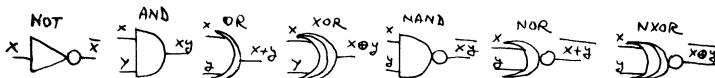
Ele sunt construite pornind de la niște **circuite elementare (porți)**, care implementează operațiile booleane: $+$, \cdot , $\bar{\cdot}$, etc. și efectuând de un număr finit de ori diverse operații de combinare (legare): **legare în serie**, **legare în paralel**, **închiderea prin ciclu**, etc., obținându-se astfel circuite tot mai complexe.

Datele de intrare și ieșire ale acestor circuite sunt valorile de adevăr 0/1 (fals/adevărat).

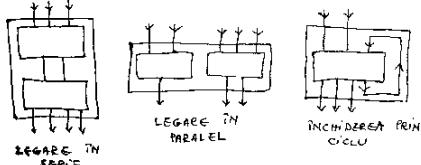


Circuite logice

Porțile sunt:



Operațiile de combinare sunt:



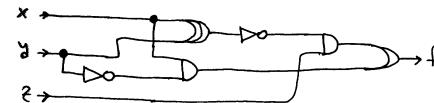
Desenarea săgeților " $>$ " pe liniile de intrare sau ieșire nu este obligatorie, dacă este evident sensul de circulație al datelor; altfel, pe o linie se pot pune oricâte săgeți.



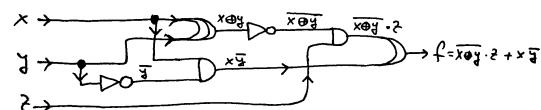
Circuite logice

În general, circuitul care implementează o formulă booleană se poate construi combinând porțile în aceeași ordine în care se compun operațiile booleane implementate de ele pentru a se obține formula.

De exemplu, formula $f(x, y, z) = \overline{x} \oplus yz + x\overline{y}$ poate fi implementată prin circuitul:

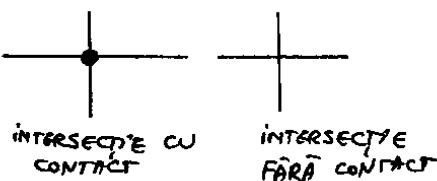


Puteam spori claritatea desenului dacă pe anumite linii (nu neapărat pe toate) adăugam săgeți " $>$ " și/sau notăm în dreptul lor formula de calcul a valorii emise pe acolo, de exemplu:

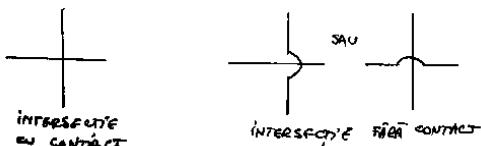


Circuite logice

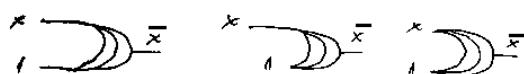
În reprezentările grafice ale circuitelor, avem nevoie să distingem între o intersecție de linii fără contact și una cu contact. În figurile anterioare am folosit următoarea simbolizare, pe care o vom utiliza și în continuare:



O altă variantă de simbolizare, dar pe care nu o vom folosi, este:



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:

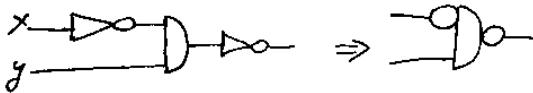


Circuite logice

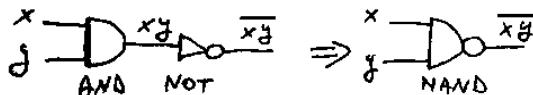


Circuite logice

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:

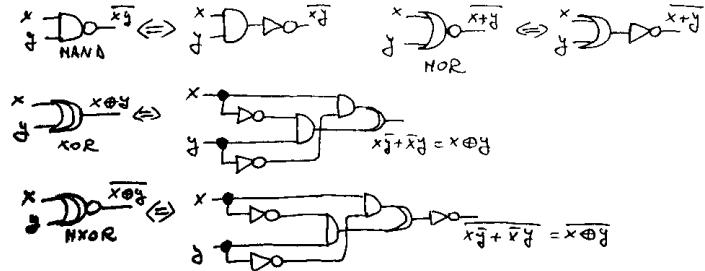


Această simplificare stă la baza simbolurilor folosite pentru porțile "NAND", "NOR", "NXOR"; de exemplu:



Circuite logice

Constatăm că pentru implementarea diverselor formule de calcul logic nu sunt necesare toate portile considerate; de exemplu, sunt suficiente portile "NOT", "AND", "OR", restul portilor putându-se exprima în funcție de acestea:



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.

Situată 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 componentei - 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 placii 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

construcție cu 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 ușor de montat decât unele componente separate.

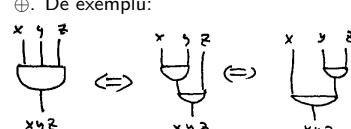
Circuite logice

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 portii.

Printre acestea, sunt circuitele "AND", "OR", "XOR", "NAND", "NOR", "NXOR" cu mai multe intrări:

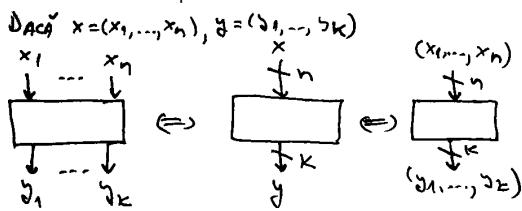


Realizarea lor se bazează pe asociativitatea și comutativitatea operațiilor $,$ $+$, \odot . De exemplu:

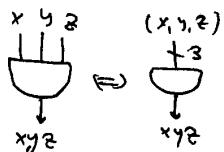


Circuite logice

Uneori, la desenarea circuitelor logice cu multe linii se folosesc următoarea simbolizare mai simplă:



De exemplu:



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, frângăii și scripeți, conducte de apă și robinete, etc.

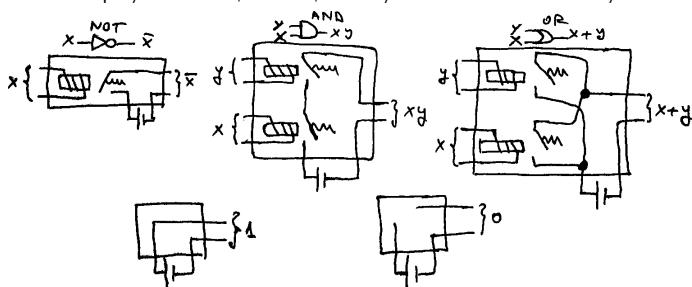
Nu trebuie să confundăm însă circuitul logic cu un anumit mod de realizare tehnică a sa.



Circuite logice

Exemplu: Realizare tehnică cu relee și contacte:

Realizarea portilor "NOT", "AND", "OR" și a intrărilor constante 0 și 1:

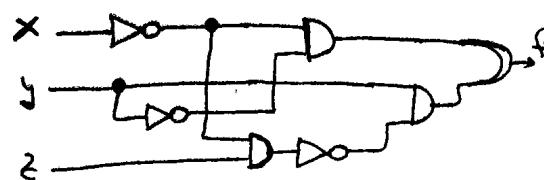


Deci, liniile circuitului sunt perechi de linii electrice, 0 = fără tensiune, 1 = sub tensiune. Fiecare poartă poate avea propria sursă de curent, sau toate portile pot fi conectate la o aceeași sursă și nu contează respectarea unei polarități +/-.



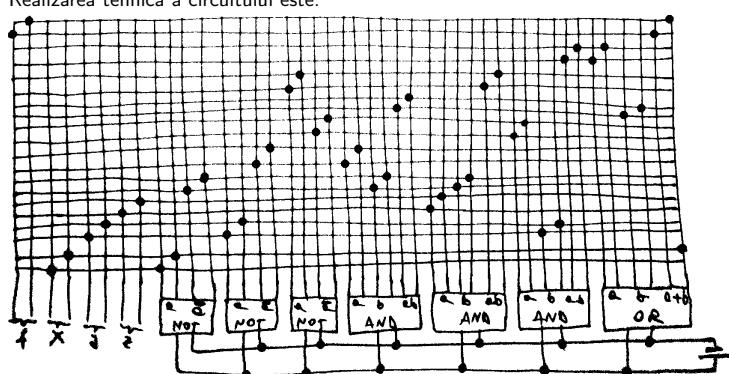
Circuite logice

Circuitul logic care implementează formula $f(x, y, z) = \bar{x} \bar{y} + y \bar{x} \bar{z}$ este:



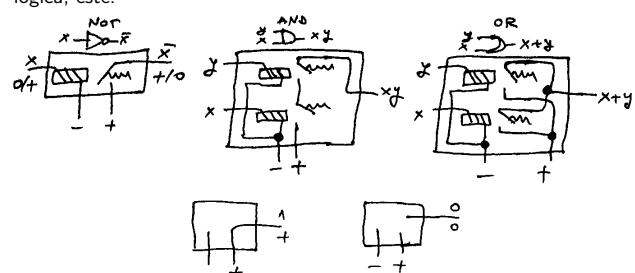
Circuite logice

Realizarea tehnică a circuitului este:



Circuite logice

O variantă de realizare cu relee și contacte a portilor "NOT", "AND", "OR" și a intrărilor constante 0 și 1, care necesită doar o linie electrică pentru o linie logică, este:



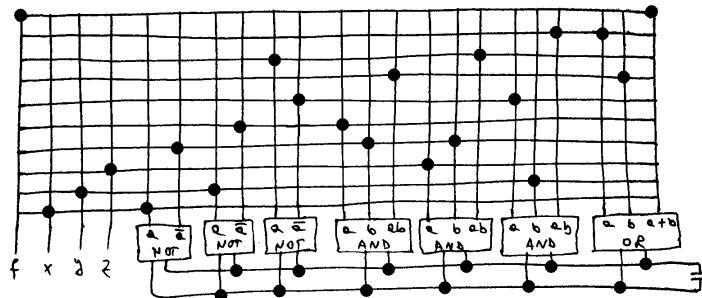
Așadar, 0 = fără tensiune, 1 = tensiune +.

Acum însă toate portile trebuie conectate la o aceeași sursă de curent și trebuie respectată polaritatea +/-.



Circuite logice

Realizarea tehnică a circuitului pentru formula $f(x, y, z) = \overline{x} \cdot \overline{y} + y \cdot \overline{x} \cdot \overline{z}$ este:



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.

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 tranzitiei 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.

În orice moment al funcționării unui circuit logic, orice linie a să transportă ceva: 0, 1 sau un semnal neclar, care poate fi interpretat la destinație în mod eronat ca 0 sau 1.

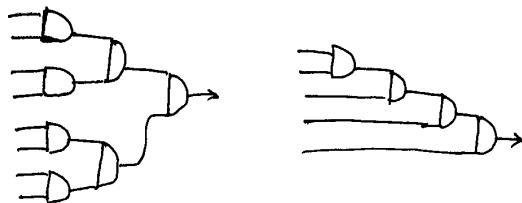
De aceea, uneori când vom analiza circuitul d.p.v. logic vom considera că liniile sale sunt fie în situația "transportă 0", fie în situația "transportă 1", aceste valori putând fi și eronate.

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 stabilă însă imediat că după un anumit interval de timp, necesită 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", prezентate 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:



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 izolare altor porțiuni de circuit unele de altele, împiedicând impudența unui circuit să afecteze impudența altuia.

În electronică, **impudență (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 comutatoare cu tranzistori, sau comandarea unui LED, deoarece poate furniza la ieșire curenti mult mai mari decât necesită ca semnal de intrare.

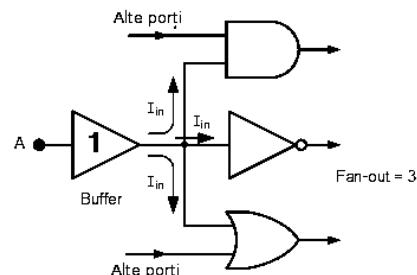
Cu alte cuvinte, bufferul poate fi folosit pentru amplificarea puterii semnalului digital, având o **capacitate fan-out (fan-out capability)** ridicată. Parametrul fan-out al unei porți sau bloc de circuit descrie capacitatea acestuia de a furniza la ieșire curenti mari, oferind o amplificare mai mare 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.i. 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 dirigate 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 porții (propagation delay) se deteriorează rapid ca funcție de "fan-in", astfel că porțile cu "fan-in" mai mare decât 4 trebuie evitate.

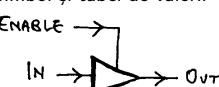
Notăm că proprietăți asemănătoare bufferului, cum ar fi efectul de amplificare a semnalului, îl au și alte porți, ca de exemplu "NOT": - de aceea, poarta "NOT" se mai numește și **buffer inversor**, sau doar **inversor (inverter)**.

◀ □ ▶ ⏪ ⏩ ⏴ ⏵ ⏹ ⏺ ⏻ ⏻

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 impudență 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. **impudență înaltă (Hi-Z)**.

Simbol și tabelă de valori:



ENABLE	IN	OUT
0	0	Hi-Z
0	1	Hi-Z
1	0	0
1	1	1

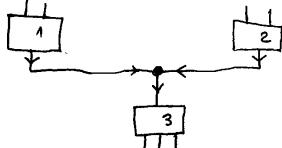
Putem considera Hi-Z ca o a treia valoare de adevăr.

◀ □ ▶ ⏪ ⏩ ⏴ ⏵ ⏹ ⏺ ⏻ ⏻

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

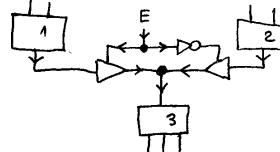
Bufferul cu 3 stări este folosit în multe circuite deoarece permit ca mai multe dispozitive logice să fie conectate la o aceeași linie sau bus fără distrugere fizică sau pierderi de date.

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 emite sau să recepționeze date unul de la altul prin acest unic bus de date în același timp, creând o astfel de **dispută (contention)**.

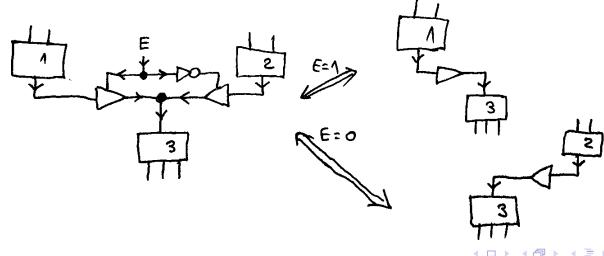
Disputele apar când mai multe dispozitive sunt conectate împreună, deoarece unele intenționează să emită la ieșire o tensiune de nivel înalt, altele una de nivel jos; dacă aceste dispozitive încep să emită sau să recepționeze date în același timp, poate apărea un scurtcircuit atunci când un dispozitiv 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

Putem însă conecta alternativ mai multe ieșiri la o aceeași linie, folosind buffer cu 3 stări:

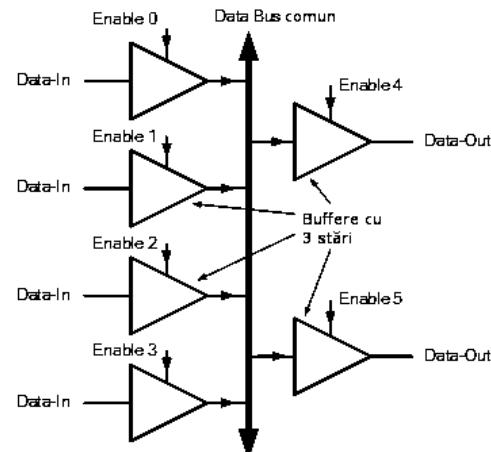


Atunci:



Circuite logice

Controlul bus-ului de date poate fi realizat folosind buffere cu 3 stări:



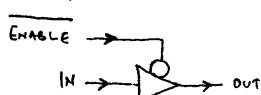
Circuite logice

Există două tipuri de buffer cu 3 stări: unul care 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	H-Z
1	1	H-Z

Circuite logice

Blocurile logice sunt împărțite în două categorii, după cum au sau nu au memorie.

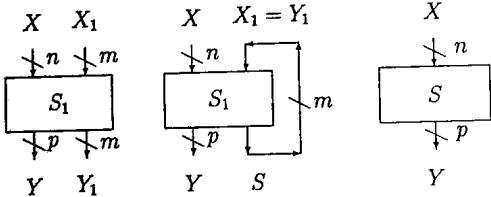
Blocurile fără memorie se zic **combinatoriale**; la un asemenea bloc, ieșirea depinde doar de intrarea curentă și poate fi descrisă printr-un tabel de adevăr; astfel, circuitul implementează de fapt o funcție booleană; blocurile combinatoriale sunt organizate ca circuite logice fără cicluri (0-DS).

În blocurile cu memorie, ieșirea depinde atât de intrarea curentă cât și de valoarea curent păstrată în memorie și care definește **starea** blocului logic; logica care include stări este **logica sequentială**; blocurile cu memorie sunt organizate ca circuite logice cu cel puțin un nivel de cicluri (n -DS, $n \geq 1$).

- Închiderea prin ciclu:

$S = \langle X \times X_1, Y \times Y_1, h \rangle$, unde:
 $X_1 = Y_1$ iar $h = (f, f_1)$, $f : X \times X_1 \rightarrow Y$, $f_1 : X \times X_1 \rightarrow Y_1$

\Rightarrow
 $S' = \langle X, Y, g \rangle$, 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 $f : X \times Q \rightarrow Y$, $f_1 : X \times Q \rightarrow Q$.

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 .

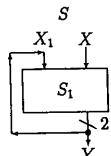
Pentru un $a \in X$ aplicat constant la intrare, ieșirea poate prezenta variații. De aceea, spunem că **autonomia** unui sistem crește ca urmare a introducerii lui într-un ciclu.

Def: Comportarea unui sistem digital este **autonomă** d.d. pentru o intrare constantă ieșirea are un comportament dinamic.

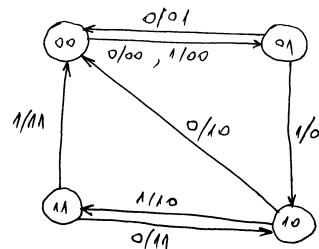
Exemplu: Fie sistemul $S_1 = \langle X \times X_1, Y, f \rangle$, unde $X = \{0, 1\}$, $X_1 = Y = \{0, 1\}^2$, iar f este definită prin tabelul:

$X \times X_1$	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 :



Funcționalitatea sa este ilustrată mai jos (se face abstracție de timpul real cât este aplicat un semnal la intrare); în cerculele este notată starea curentă (elementul lui $X_1 = Y$); pe fiecare săgeată sunt notate intrarea curentă (elementul lui X) și 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.

Observație: Sistemul initial S , fără cicluri, nu avea autonomie - ieșirea să 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 ...

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 i 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 operațiilor 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.

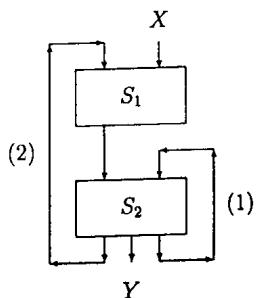
Sisteme digitale

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):



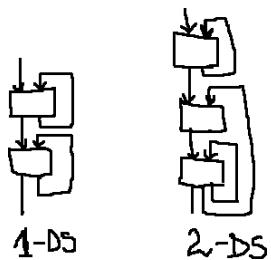
Def: Un **sistem digital de ordin n** (*n-digital system, n-DS*), $n \geq 0$, se definește recursiv astfel:

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

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).

Vom arăta următoarea corespondență ierarhică pentru n -DS, care corespunde arhitecturii unui calculator:

- 0-DS: circuite combinaționale, funcții booleene;
- 1-DS: memori;
- 2-DS: automate finite;
- 3-DS: procesoare;
- 4-DS: calculatoare.

Cuprins

- 1 Performanță 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)
 - Algoritm 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.

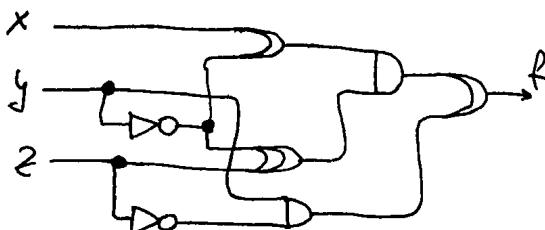


Implementare directă

Implementare directă:

O funcție booleană dată printr-o formulă poate fi implementată combinând portile 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 porti).

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.

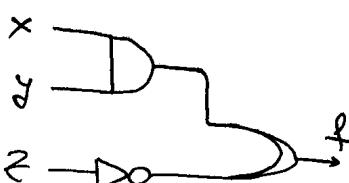


Circuit minimal

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:

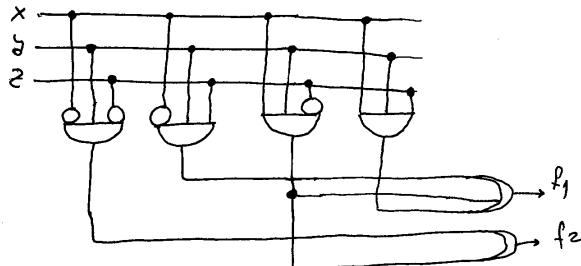
$$f_1(x, y, z) = \bar{x}yz + xy\bar{z} + xyz$$

$$f_2(x, y, z) = \bar{x}y\bar{z} + xy\bar{z}$$



Circuit general

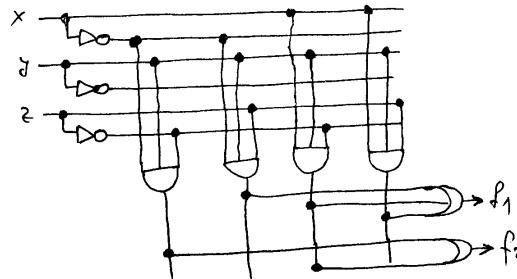
Atunci, funcția f poate fi implementată prin circuitul:



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} .

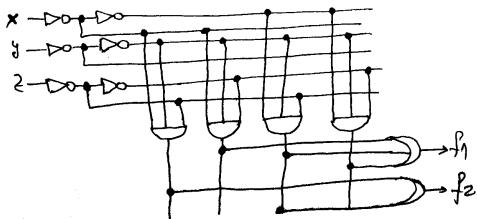
Circuit general

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

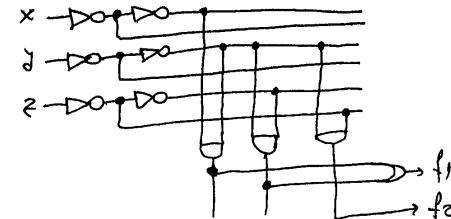
Putem minimiza scrierea funcției ca sumă de produse (să avem cât mai puține operații + și ·) înainte de a o implementa prin circuitul general. Astfel, vom obține un circuit general minimal.

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 placă 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 linioilor 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).

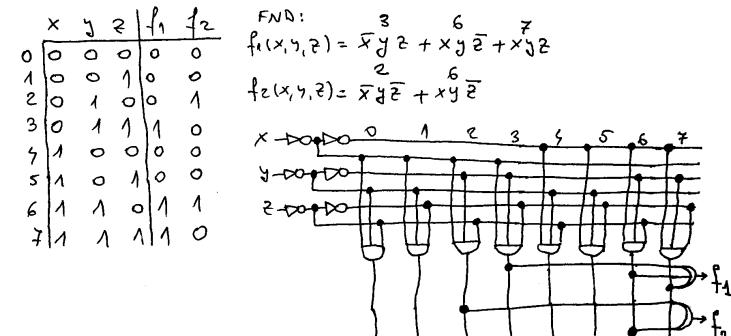
În acest caz, suma de produse implementată de circuit este FND.

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):



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 $\bar{}$ 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 lini îi 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 lini 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 lini ca și suma.

Așadar, liniile desenate au semantici diferite, în funcție de poziția lor pe desen: cele orizontale din blocul de sus sunt lini de circuit, cele verticale din planul "AND" sunt produse, cele orizontale din planul "OR" sunt sume.



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.

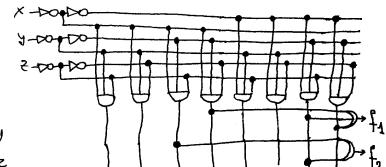
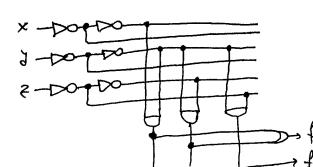
Din punct de vedere tehnic, având în vedere gradul ridicat de standardizare, aceste circuite se pot construi după tehnologii diferite, mai simple, decât circuitele alcătuite din porți și având o organizare oarecare.

Întrucât pot implementa orice funcție booleană, PLA și PROM sunt circuite universale.

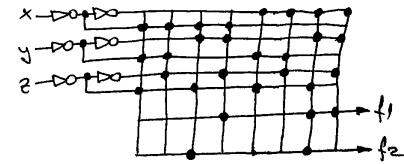
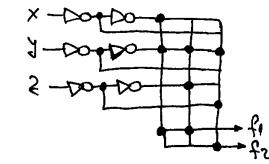


PLA și PROM

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

Din cele de mai sus rezultă că putem construi PROM-ul unei funcții direct din tabelul extins al acesteia, procedând astfel:

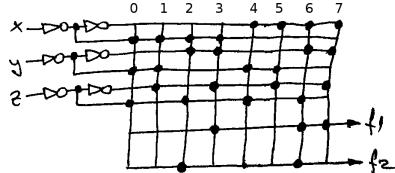
- construim tabelul extins;
- construim blocul care furnizează valorile variabilelor și negațiile lor și planul "AND" al PROM-ului (ele depind doar de numărul de variabile);
- construim planul "OR" al PROM-ului, plasând pe fiecare linie - sumă corespunzătoare unei componente a funcției puncte de contact 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
0	0	1	0	0	1
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



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		f_1	f_2
x	y	0	1		
0	0	0	0	0	0
0	1	0	1	0	\bar{z}
1	0	0	0	0	0
1	1	1	0	1	\bar{z}

Din tabelul compact rezultă următoarele scrieri simplificate ca sume de produse ale componentelor funcției:

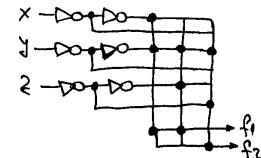
$$f_1(x, y, z) = \bar{x}yz + xy$$

$$f_2(x, y, z) = y\bar{z}$$

Observăm că putem simplifica în continuare formula lui f_1 , folosind absorbția booleană:

$$f_1(x, y, z) = (\bar{x}z + x)y = (z + x)y = xy + yz$$

Pornind de la formulele minimale, obținem următorul PLA:

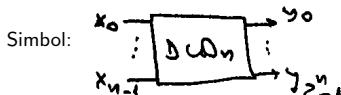


Decodificator

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 trimită 1).

De fapt, acesta este un caz particular de decodificator, numit **decodificator linie (line decoder)**.



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.i. $y_k = 1$ 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, selectabile printr-un cod numeric.



Decodificator

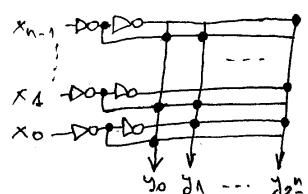
DCD_n implementează o funcție $f : B_2^n \rightarrow B_2^{2^n}$ având următorul tabel de valori:

x_{n-1}	\dots	x_1	x_0	y_0	y_1	\dots	y_{2^n-1}
0	\dots	0	0	1	0	\dots	0
0	\dots	0	1	0	1	\dots	0
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
1	\dots	1	1	0	1	\dots	1

Așadar, coloana fiecărei componente y_k a funcției, $0 \leq k \leq 2^n - 1$, conține o singură valoare 1, anume pe linia k .

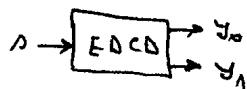
Atunci, dacă implementăm această funcție ca PROM, fiecare produs din planul "AND" va participa la o singură sumă.

În consecință, un decodificator este planul "AND" dintr-un PROM:



Decodificator

Pentru $n = 1$ obținem **decodificatorul elementar (elementary decoder)**, EDCD; simbolizare:

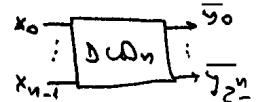


Construcție:



Decodificator

Uneori este utilă o variantă a decodificatorului, în care ieșirile sunt complementate:

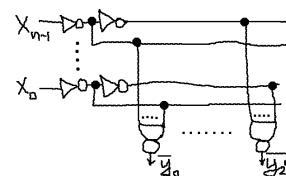


El transformă selectorul $(x_{n-1}, \dots, x_0) = [k]^u_n$ în sistemul de valori y_0, \dots, y_{2^n-1} a.î. $y_k = 0$ și $y_i = 1$ pentru $i \neq k$.

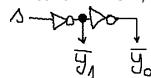


Decodificator

Pentru construcție, în planul "AND" al PROM-ului se înlocuiesc porțile "AND" cu porți "NAND".



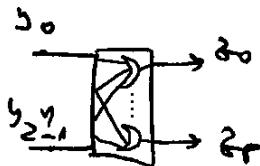
În cazul EDCD, se poate proceda astfel:



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 (însoțit de decodificator) este un circuit universal, i.e. poate implementa orice funcție booleană.

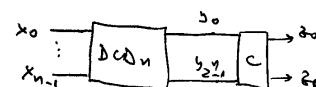
Codificator

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: $(x_{n-1}, \dots, x_0) = [k]^u_n$, $0 \leq k \leq 2^n - 1 \Rightarrow (y_0, \dots, y_{2^n-1}) = (0, \dots, 0, 1, 0, \dots, 0)$ (1 este pe poziția k) $\Rightarrow (z_1, \dots, z_p)$, unde pentru orice $1 \leq i \leq p$, $z_i = z_i(k) = z_i(x_{n-1}, \dots, x_0) : B_2^n \rightarrow B_2$ este o funcție booleană scalară oarecare, deci $z = (z_1, \dots, z_p) : B_2^n \rightarrow B_2^p$ este o funcție booleană vectorială oarecare.



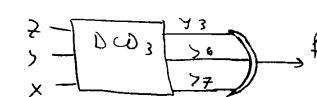
Codificator

Exemplu: Implementați cu ajutorul unui codificator funcția booleană

$f : B_2^3 \rightarrow B_2$	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 DCD₃ 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	1	0
3	1	0	1
4	0	0	0
5	0	1	0
6	1	0	1
7	1	1	1

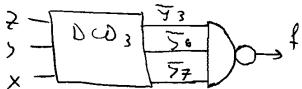


Codificator

Demultiplexor

Dacă folosim un decodificator cu ieșirile negate, în construcția codificatorului se înlocuiesc porțile "OR" cu porți "NAND".

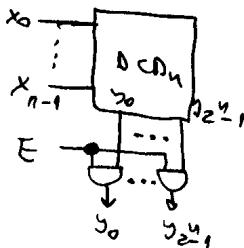
Exemplu: Pentru funcția f din exemplul anterior, obținem implementarea:



Demultiplexor

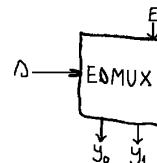
Constatăm că demultiplexorul este o generalizare a decodificatorului; mai exact, un DCD_n este un $DMUX_n$ unde am fixat $E = 1$.

De aceea, demultiplexorul se construiește ușor cu ajutorul unui decodificator - se conjugă toate ieșirile decodificatorului cu E :

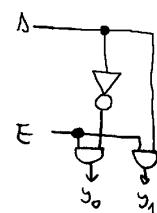


Demultiplexor

Pentru $n = 1$ obținem **demultiplexorul elementar (elementary demultiplexer)**, *EDMUX*; simbolizare:



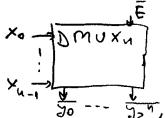
Constructie:



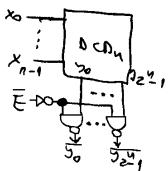
Demultiplexor

Potrivit construcției, putem construi o variantă a demultiplexorului, în care intrarea E și ieșirile y_0, \dots, y_{2^n-1} sunt negate.

Simbolizare:



Constructie:



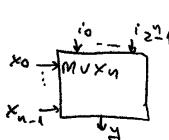
Această variantă are avantajul că și intrarea E va fi protejată iar semnalul va fi amplificat înainte de combinarea cu cele 2^n ieșiri ale decodificatorului, datorită efectului de buffer al portiei "NOT".

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.

Symbol:



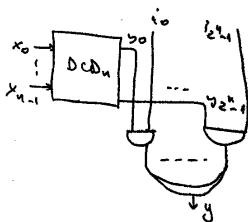
El primește ca intrare:

- 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

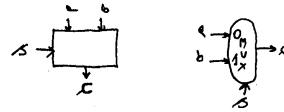
Construcție:



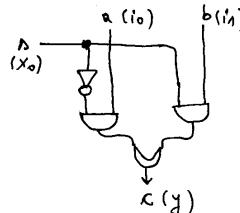
Decodificatorul transformă selectorul $k = (x_{n-1}, \dots, x_0)$ în sistemul de ieșiri $0, \dots, 0, 1, 0, \dots, 0$, unde 1 este pe poziția k ; aceste ieșiri sunt conjugate cu respectiv intrările i_0, \dots, i_{2^n-1} , rezultând sistemul de valori $0, \dots, 0, i_k, 0, \dots, 0$, unde i_k este pe 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:

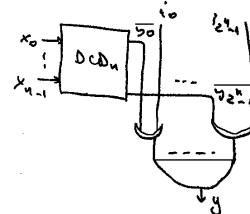
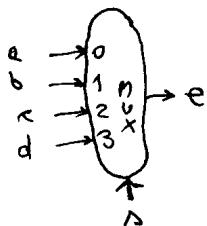


Multiplexor

Multiplexor

Dacă folosim un decodificator cu ieșirile negate, în construcția multiplexorului se interschimbă portile nou adăugate "AND" și "OR":

De asemenea, vom folosi și **MUX_2** , pentru care avem și simbolizarea:



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 disjuns 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

Multiplexor

Multiplexorul este și el un circuit universal, deoarece putem implementa orice funcție booleană scalară cu un multiplexor.

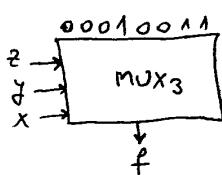
Exemplu: Implementați funcția booleană următoare cu un multiplexor:

$f : B_2^3 \rightarrow B_2$ datează 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:



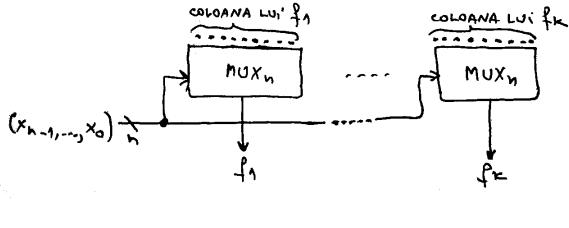
Intuitiv, fiecare sistem de valori ale variabilelor x, y, z , care este reprezentarea unui număr natural $0 \leq k \leq 7$, selectează:

- din tabel, valoarea lui f din linia k ;
- cu multiplexor, intrarea i_k de sus.

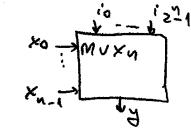
Așadar, multiplexorul implementează pe f dacă plasăm coloana valorilor lui f ca sistem de intrări i_0, \dots, i_7 ; însăci sistemul de valori $0, 0, 0$ ale variabilelor selectează valoarea din linia de sus a tabelului și intrarea i din stânga a multiplexorului, coloana valorilor lui f trebuie plasată 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:



se poate construi ca:



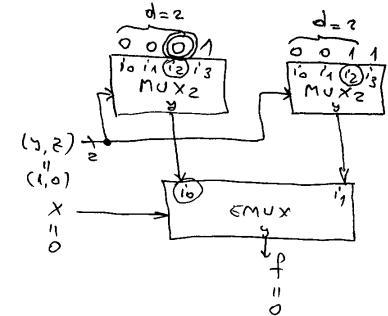
Multiplexor

Multiplexor

Echivalența celor două circuite este ușor de înțeles dacă facem o comparație cu tabelul de valori al funcției implementate; vom ilustra pentru funcția din exemplul anterior, presupunând că $(x, y, z) = (0, 1, 0)$:

$(x, y, z) = (0, 1, 0)$	
$x=0$	$y \quad z \quad f$
0	0 0 0 0
0	0 0 1 0
0	1 0 0 0
0	1 0 1 0
1	1 1 0 1
1	1 1 1 1

$(x, y, z) = (0, 1, 0)$	
$x=1$	$y \quad z \quad f$
0	0 0 0 0
0	0 0 1 0
1	1 0 0 0
1	1 0 1 0
1	1 1 0 1



Multiplexor

Multiplexor

Iesirea 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.

Proprietatea algebraică pe care se bazează construcția recursivă de mai sus este teorema care dă o descriere recursivă a unei funcții booleane scalare cu n variabile prin două funcții booleane scalare cu $n - 1$ variabile:

$$\forall x_{n-1}, \dots, x_0 \in B_2, f(x_{n-1}, \dots, x_0) = \bar{x}_{n-1}f(0, x_{n-2}, \dots, x_0) + x_{n-1}f(1, x_{n-2}, \dots, x_0)$$

MUX_n construit recursiv implementează $f : B_2^n \rightarrow B_2$:

MUX_{n-1} din stânga (a căruia 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 căruia 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

Multiplexor

Observație: Si alte circuite 0 – DS, ca de exemplu DCD_n , sau $DMUX_n$, admit o construcție recursivă (exercițiu).

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 obținem 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.

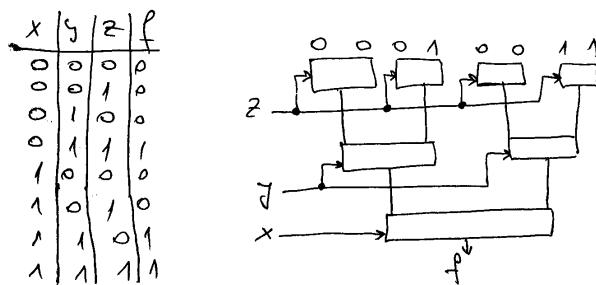
Observație: Alte circuite universale sunt $NAND$ și NOR (putem implementa orice funcție booleană folosind doar $NAND$ -uri sau doar NOR -uri) - exercițiu.



Multiplexor

Multiplexor

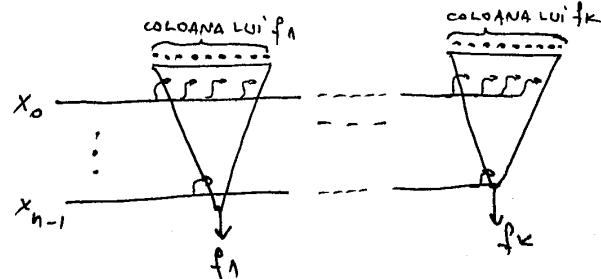
Exemplu: Funcția din exemplul precedent se poate implementa cu $EMUX$ -uri astfel:



Dacă păstrăm convențiile anterioare privind poziția intrărilor și ieșirilor pe desenul $EMUX$ -urilor, nu mai este nevoie să notăm aceste intrări și ieșiri.



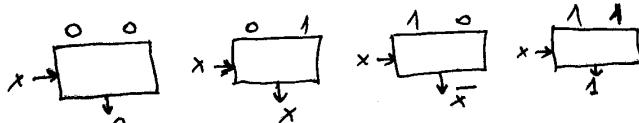
O funcție booleană vectorială $f : B_2^n \rightarrow B_2^k$, $k > 1$, poate fi implementată printr-un sistem de k arbori de $EMUX$ -uri, fiecare calculând câte una din componentele funcției, f_1, \dots, f_k ; fiecare linie corespunzătoare unei variabile a lui f intră ca selector în $EMUX$ -urile de pe un același rând în toți arborii, iar acestia au ca sisteme de intrări i coloane de valori ale componentelor lui f pe care le calculează:



Multiplexor

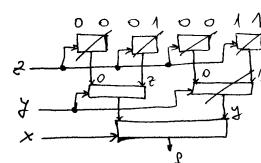
Multiplexor

Numărul $EMUX$ -urilor prin care se implementează o funcție booleană poate fi redus, dacă observăm că:

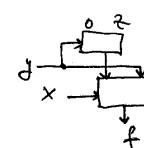


Astfel, dacă acceptăm și prezența porților "NOT", $EMUX$ -urile aflate pe primul rând al arborilor dispar, iar uneori dispar și $EMUX$ -uri de pe rândurile următoare.

Exemplu: Reduceti la maxim numărul de $EMUX$ prin care se implementează funcția f din exemplele anterioare:



Desenul care păstrează doar $EMUX$ -urile rămase este:

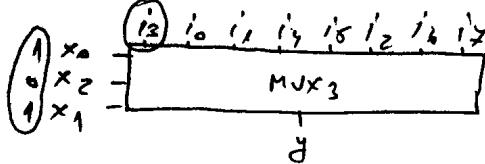


Sfat: Desenul circuitului redus este bine să fie realizat de jos în sus, deoarece avem o perspectivă mai clară care linii trebuie șterse sau redirecționate.

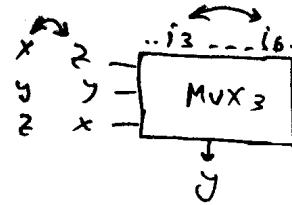


Observație:

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:



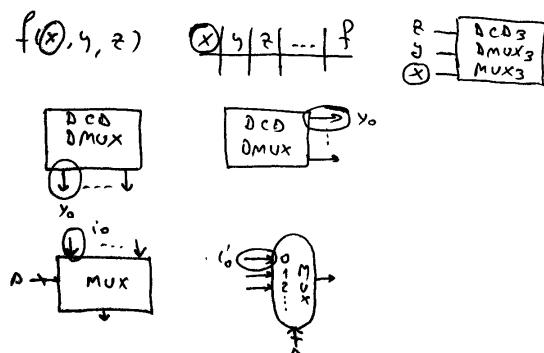
Dacă folosim nume neindexate, contează ordinea semnificațiilor variabilelor și cea de scriere/desenare în funcție de semnificație. De exemplu, dacă inversăm ordinea liniilor selectorului unui MUX_3 , intrarea i_3 se interschimbă cu i_6 :



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 decodificator, codificator, demultiplexor, multiplexor;
- ieșirea y corespunzătoare valorii $0 = (0, \dots, 0)$ a selectorului blocurilor decodificator 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.



Sumator

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 .

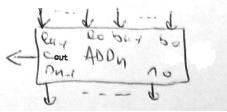
Mai exact, el primește ca intrare două siruri de biți $a_{n-1}, \dots, a_0, b_{n-1}, \dots, b_0$ și eventual un transport de intrare c_{in} (pentru poziția 0) și le aplică algoritmul de adunare cu reprezentările binare ale numerelor naturale folosit în matematică; transportul din poziția $n - 1$ se pierde (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.

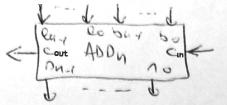
Sumator

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

Sumator

- **Half adder (HA):**

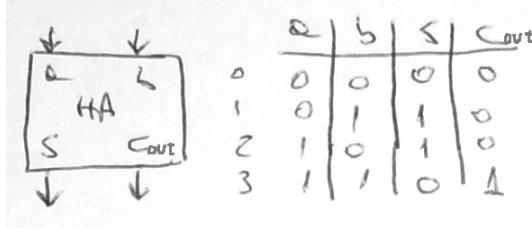
Primeste ca intrare: a, b (doi operanzi de un bit).

Efectueză adunarea: $a + b$.

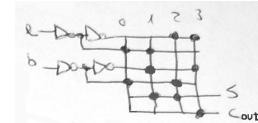
Furnizează ca ieșire: s (bitul sumă, trimis în locația destinație), c_{out} (carry out).

Observăm că: $s = (a + b) \text{ mod } 2$, $c_{out} = (a + b) \text{ div } 2$.

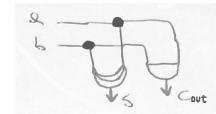
Simbol și tabel de adevăr:



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
 $HA : B_2^2 \rightarrow B_2^2$, $HA(a, b) = (HA_s(a, b), HA_c(a, b))$, unde
 $HA_s(a, b) = s = a \oplus b$, $HA_c(a, b) = c_{out} = a \wedge b$.

Sumator

Sumator

- **Full adder (FA):**

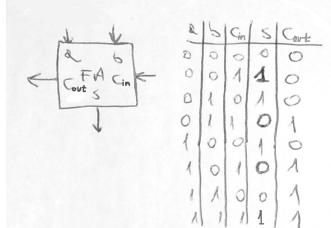
Primeste ca intrare: a, b (doi operanzi de un bit), c_{in} (carry in).

Efectueză 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

$FA : B_2^3 \rightarrow B_2^2$, $FA(a, b, c) = (FA_s(a, b, c), FA_c(a, b, c))$, unde

$FA_s(a, b, c) = s = (a + b + c) \text{ mod } 2$, $FA_c(a, b, c) = c_{out} = (a + b + c) \text{ div } 2$.

Dacă din tabelul anterior extragem FND, rezultă că pentru $a, b, c = c_{in}$ date, avem:

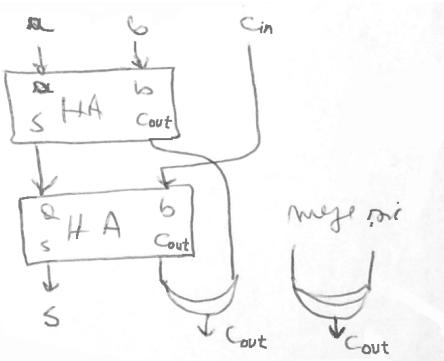
$$\begin{aligned} FA_s(a, b, c) &= (\bar{a} \wedge \bar{b} \wedge c) \vee (\bar{a} \wedge b \wedge \bar{c}) \vee (a \wedge \bar{b} \wedge \bar{c}) \vee (a \wedge b \wedge c) = \\ &= (((\bar{a} \wedge b) \vee (a \wedge \bar{b})) \wedge \bar{c}) \vee (((\bar{a} \wedge \bar{b}) \vee (a \wedge b)) \wedge c) = \\ &= ((a \oplus b) \wedge \bar{c}) \vee (\bar{a} \oplus b \wedge c) = (a \oplus b) \oplus c = HA_s(HA_s(a, b), c_{in}) \end{aligned}$$

$$\begin{aligned} FA_c(a, b, c) &= (\bar{a} \wedge b \wedge c) \vee (a \wedge \bar{b} \wedge c) \vee (a \wedge b \wedge \bar{c}) \vee (a \wedge b \wedge c) = \\ &= (((\bar{a} \wedge b) \vee (a \wedge \bar{b})) \wedge c) \vee (a \wedge b \wedge (\bar{c} \vee c)) = ((a \oplus b) \wedge c) \vee (a \wedge b) = \\ &= HA_c(HA_s(a, b), c_{in}) \vee HA_c(a, b) = HA_c(HA_s(a, b), c_{in}) \oplus HA_c(a, b) \end{aligned}$$

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

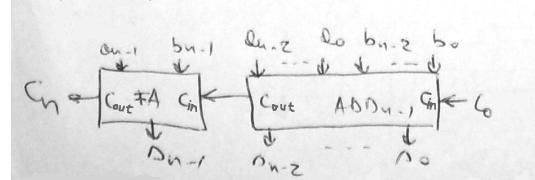
• Sumator serial:

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 fiecarei 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 operanților și, respectiv, sumei, c_i carry 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 formulalele 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:

$c_{i+1} = G_i \vee (P_i \wedge G_{i-1}) \vee (P_i \wedge P_{i-1} \wedge G_{i-2}) \vee \dots \vee (P_i \wedge P_{i-1} \wedge \dots \wedge P_0 \wedge c_0)$ 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 toti G_i, \dots, G_0 sunt 0, atunci $c_{i+1} = P_i \wedge P_{i-1} \wedge \dots \wedge P_0 \wedge c_0$. Altfel, fie $0 \leq k \leq i$ cel mai mare indice a.i. $G_k = 1$; atunci, pentru $j > k$ avem $P_i \wedge P_{i-1} \wedge \dots \wedge P_{j+1} \wedge G_j = 0$, deoarece $G_j = 0$, iar pentru $j < k$ avem $P_i \wedge P_{i-1} \wedge \dots \wedge P_{j+1} \wedge G_j = 0$, deoarece termenul îl conține pe P_k , care este 0 (G_k și P_k nu pot fi simultan 1); deci, $c_{i+1} = P_i \wedge P_{i-1} \wedge \dots \wedge P_{k+1} \wedge G_k$.

Sumator

Formula de calcul a lui c_{i+1} , $0 \leq i \leq n$, de mai sus are asociate următoarele observații intuitive:

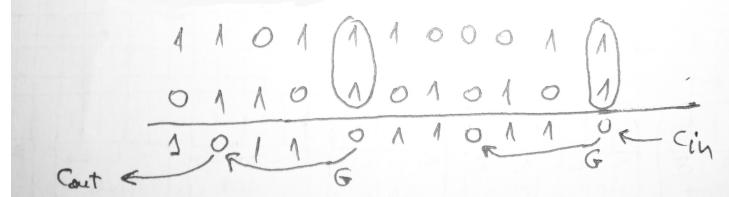
- la o poziție $0 \leq i \leq n$ se generează carry out d.d. $a_i = b_i = 1$, i.e. $G_i = 1$; în acest caz, suma rămasă pentru poziția i este 0, iar $P_i = 0$;
- carry out-ul generat la poziția i se propagă spre poziții j aflate tot mai la stânga (valori j tot mai mari) atât timp cât la pozițiile respective 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.i. $P_k = 0$; 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 așa 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:

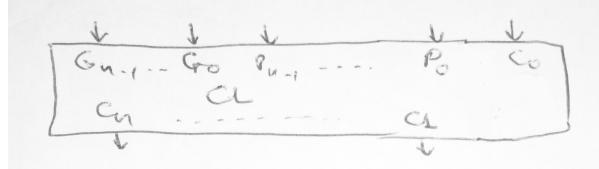
- fie de la un $c_0 = 1$, dacă toate $P_i, 0 \leq i \leq n$, sunt 1 (atunci toate $G_i, 0 \leq i \leq n$, sunt 0, deci nici o poziție nu generează carry out, iar numerele de n biți adunate sunt unul complementul față de unu al celuilalt);
- fie de la cea mai mare poziție $0 \leq i \leq n$ unde s-a generat 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 .



Sumator

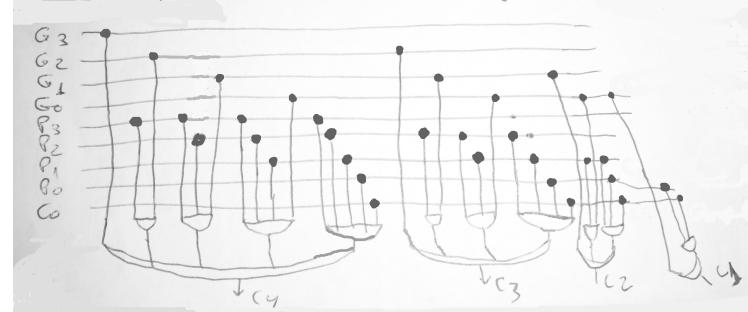
Circuitul CL implementează formula anterioară de calcul pentru toate carry out-urile c_i , $1 \leq i \leq n$, și are două niveluri (un nivel \wedge și un nivel \vee), deci numărul de niveluri și timpul de calcul nu depind de n .

Simbol:



Sumator

Construcție pentru $n = 4$:



Circuitul implementează formulele:

$$c_4 = G_3 \vee (P_3 \wedge G_2) \vee (P_3 \wedge P_2 \wedge G_1) \vee (P_3 \wedge P_2 \wedge P_1 \wedge G_0) \vee (P_3 \wedge P_2 \wedge P_1 \wedge P_0 \wedge c_0)$$

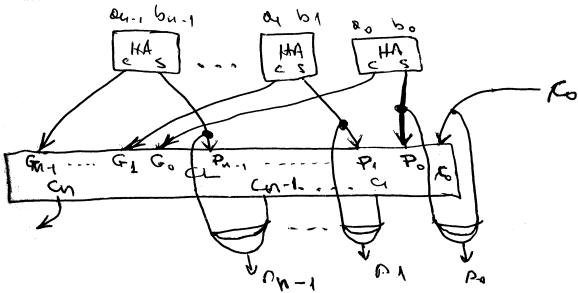
$$c_3 = G_2 \vee (P_2 \wedge G_1) \vee (P_2 \wedge P_1 \wedge G_0) \vee (P_2 \wedge P_1 \wedge P_0 \wedge c_0)$$

$$c_2 = G_1 \vee (P_1 \wedge G_0) \vee (P_1 \wedge P_0 \wedge c_0)$$

$$c_1 = G_0 \vee (P_0 \wedge c_0)$$

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/intreg 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:

$$n = 1 : NOT$$

$$n \rightarrow n + 1 : INC_n \text{ extins serial cu } HA$$

- ADD_n cu intrarea b fixată pe $\underbrace{0 \dots 0}_n 1$;

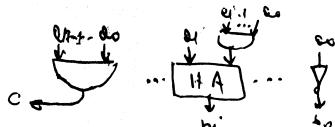
Circuit pentru incrementare

- 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 recalculă 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:

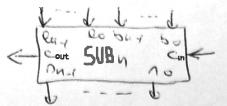
Un circuit pentru scădere pe n biți SUB_n , $n \geq 1$, este un circuit care implementează operația pe care la începutul secțiunii "Reprezentarea numerelor în calculator" am notat-o cu \ominus .

Mai exact, el primește ca intrare două siruri de biți $a_{n-1}, \dots, a_0, b_{n-1}, \dots, b_0$ și un imprumut 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 imprumut 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 subtractor), în aceeași manieră ca FA;
 $n = n + 1$: 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:

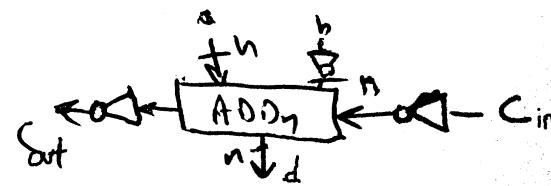
$$b + \bar{b} = 2^n - 1, c + \bar{c} = 1,$$

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

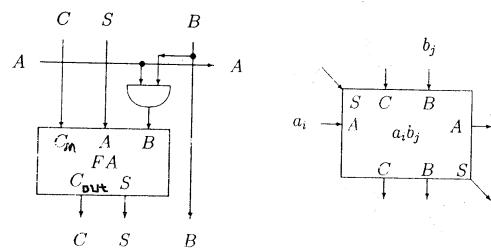
Multiplicator:

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:

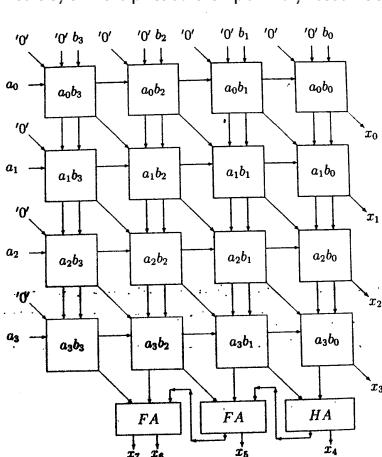
Multiplicator

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 repetitive, pe parcursul mai multor pași rapizi (vom vedea mai târziu).

Comparator

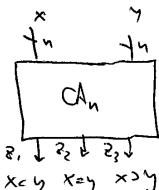
Comparator

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:



Notând că mai înainte \bar{a} , \bar{b} numerele obținute prin negarea bițiilor din reprezentările lui a , respectiv b (complementele față de 1), vom avea:

$$\begin{aligned} a > b &\Leftrightarrow a - b > 0 \Leftrightarrow a + \bar{b} + 1 - 2^n > 0 \Leftrightarrow a + \bar{b} > 2^n - 1 \\ &\Leftrightarrow a + \bar{b} \text{ are carry out} = 1. \end{aligned}$$

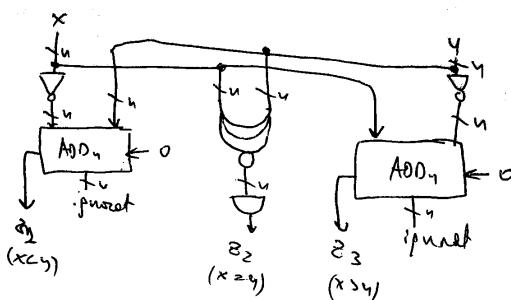
Similar, $a < b \Leftrightarrow \bar{a} + b \text{ are carry out} = 1$.

În fine, $a = b \Leftrightarrow \forall i \in \{0, \dots, n-1\}, a_i = b_i$
 $\Leftrightarrow \forall i \in \{0, \dots, n-1\}, \overline{a_i \oplus b_i} = 1$ (este operația "NXOR")
 $\Leftrightarrow \overline{a_0 \oplus b_0} \wedge \dots \wedge \overline{a_{n-1} \oplus b_{n-1}} = 1$.

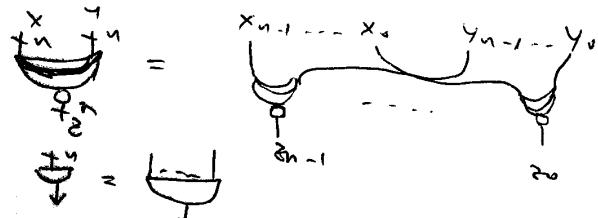
Comparator

Comparator

Atunci, putem construi comparatorul astfel:



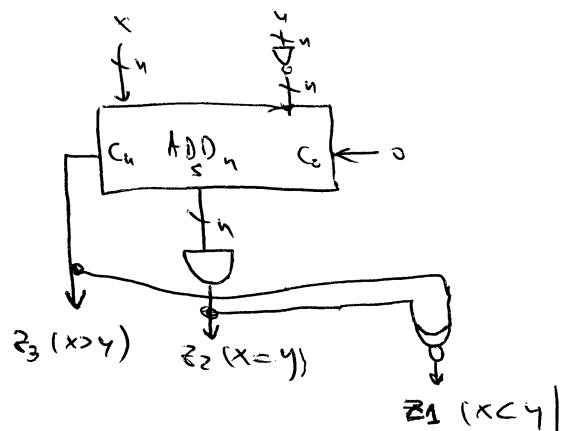
unde am folosit următoarele simboluri:



Comparator

Comparator

Obținem circuitul:



O variantă mai simplă, cu un singur sumator, se poate obține observând că:

$$a > b \Leftrightarrow a + \bar{b} \text{ are carry out} = 1 \text{ (am văzut)}.$$

$$a = b \Leftrightarrow a - b = 0 \Leftrightarrow a + \bar{b} + 1 - 2^n = 0 \Leftrightarrow a + \bar{b} = 2^n - 1$$

$$\Leftrightarrow a + \bar{b} \text{ are toti biții } 1 \Leftrightarrow \text{conjuncția } \wedge \text{ a bițiilor lui } a + \bar{b} \text{ este } 1.$$

$$a < b \Leftrightarrow \text{celălalt caz} \Leftrightarrow \text{"NOR"-ul (adică "nici"-ul) celorlalte două cazuri.}$$

Shifter

Shifter

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 pozitii. 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: $\underbrace{a_{n-k-1}, a_{n-k-2}, \dots, a_0}_k, \underbrace{0, \dots, 0}_{k \text{ biți}}$

- Shiftare logică la dreapta (instrucțiunea "sr1" din limbajul MIPS):

Furnizează sirul: $\underbrace{0, \dots, 0}_k, \underbrace{a_{n-1}, a_{n-2}, \dots, a_k}_k$

- Shiftare aritmetică la dreapta (instrucțiunea "sra" din limbajul MIPS):

Furnizează sirul: $\underbrace{a_{n-1}, \dots, a_{n-1}}_k, \underbrace{a_{n-1}, a_{n-2}, \dots, a_k}_k$

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 multimea 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.

Shifter

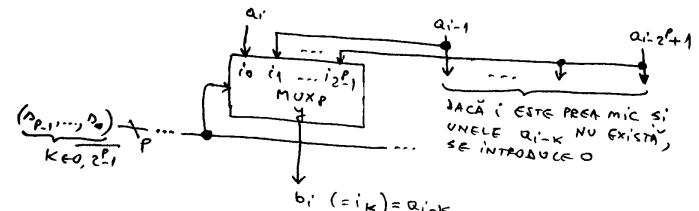
Shifter

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 pozitii k de shiftare (k este valoarea de selecție a multiplexorului), alege dintre $a_{i-1}, \dots, a_{i-2^p+1}$ (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 inexistenti vor primi 0 (în cazul shiftărilor logice) sau a_{n-1} (în cazul shiftării aritmétice la dreapta).

Dacă desenăm generic această asociere evidențind intrările legate la o ieșire b_i , obținem:

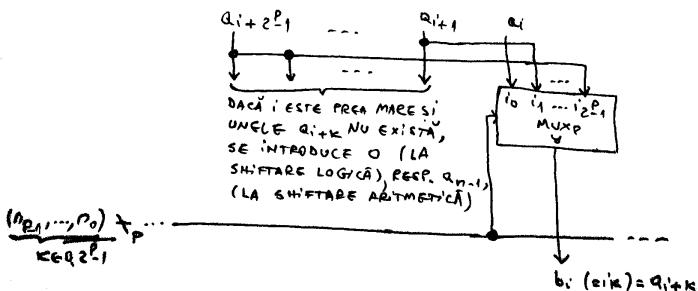
- pentru shiftarea la stânga:



Shifter

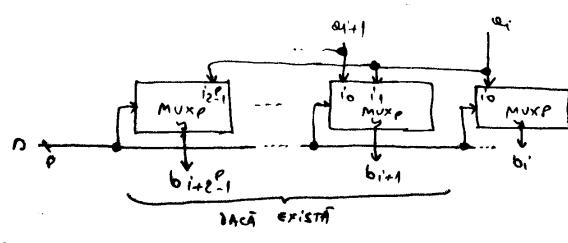
Shifter

- pentru shiftarea la dreapta:



Dacă desenăm generic această asociere evidențind ieșirile la care este legată o intrare a_i , obținem:

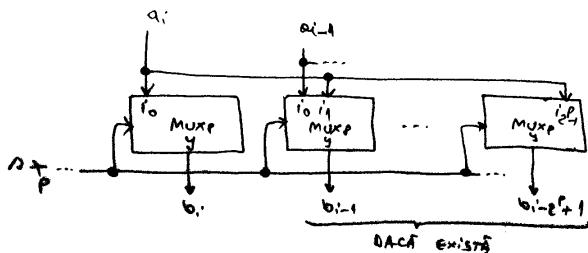
- pentru shiftarea la stânga:



Shifter

Shifter

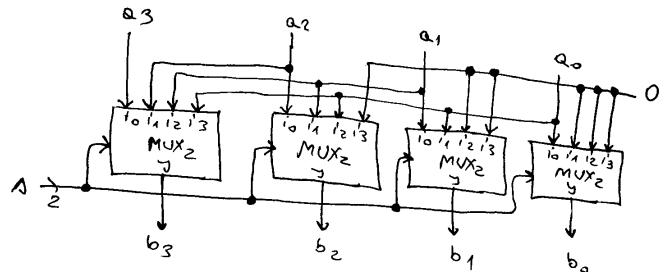
- pentru shiftarea la dreapta:



Regulă de ținut minte: fiecare a_i se ramifică de 2^p ori spre direcția de shiftare.

Exemplu: Explicați circuitele de shiftare pentru $n = 4$, $p = 2$.

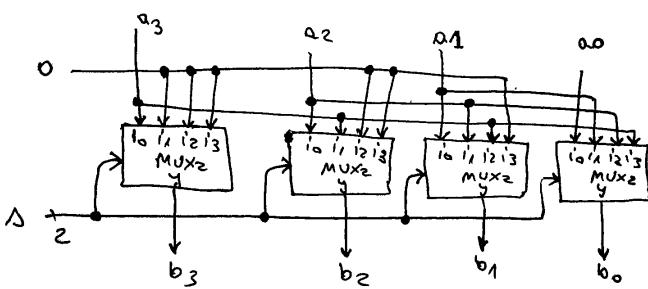
- Shiftarea logică la stânga:



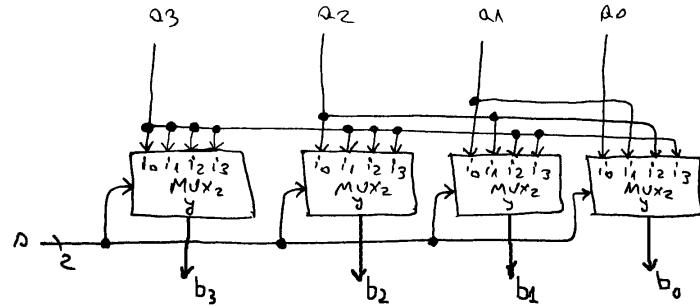
Shifter

Shifter

- Shiftarea logică la dreapta:



- Shiftarea aritmetică la dreapta:

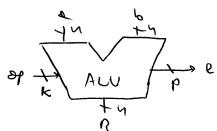


Unitate aritmetică și logică

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 operanze numere întregi pe n biti 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).

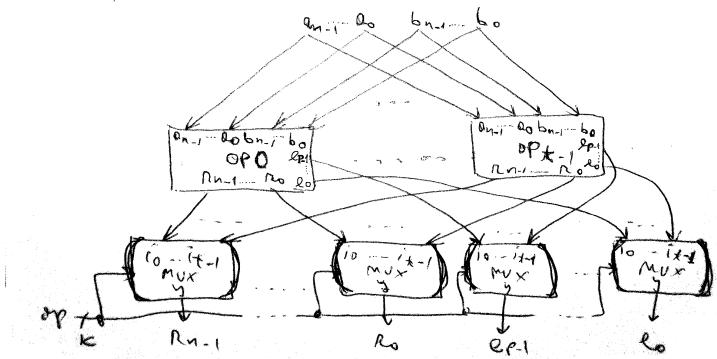
Unitate aritmetică și logică

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} ;
- operanzei 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 multiplexoare).

Unitate aritmetică și logică

Construcția circuitului:

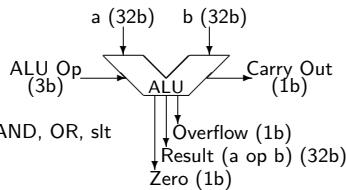


O implementare cu mai puține componente ar presupune să nu construim blocuri separate pentru fiecare operație ci să refolosim cât mai multe componente - de exemplu, scăderea se face cu sumatorul, dar introducând b și c_{in} negate.

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.

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 than) 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 incarcă operandul a din memorie în registrul $t0
lw $t1, b    # se incarcă operandul b din memorie în registrul $t1
slt $t2, $t0, $t1
# registrul $t2 primește valoarea 1/0 a expresiei a < b
lw $t0, c    # se incarcă operandul c din memorie în registrul $t0
lw $t1, d    # se incarcă operandul d din memorie în 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) ("și" pe biți în limbajul C);
# intrucția $t2, $t3 pot conține 1 doar în bitul de rang 0,
#   aceasta este echivalentă cu
#   (a < b) && (c < d) ("și" în limbajul C);
sw $t0, x
# se scrie rezultatul din registrul $t0 în variabila x din memorie
```

Astfel, expresia condițională se evaluatează 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.

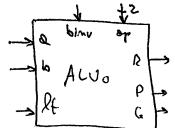
*ALU*₁ se construiește diferit în funcție de rangul bitului: $0, 1 \leq i \leq 30, 31$ (de exemplu, valoarea 1/0 furnizată de *slt* este determinată la poziția 31 și emisă ca rezultat la poziția 0).

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);

It este o informație emisă ca rezultat în cazul "slt"

(rezultatul pe 32 biți va fi $\underbrace{0 \dots 0}_{b_1} \underbrace{0 \dots 0}_{b_0}$ sau $\underbrace{0 \dots 0}_{b_1} \underbrace{1}_{b_0}$);

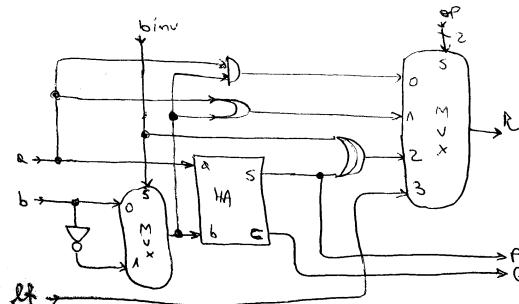
binv este 0 sau 1 după cum se va aduna *a* cu *b* sau *a* cu \bar{b} ;

(la adunare trebuie efectuat $a + b$, la scădere trebuie efectuat $a + \bar{b} + 1$); op este 0, 1, 2, 3 (représentat 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 este rezultatul operației (1 bit);

Unitate aritmetică și logică

Construcția circuitului:



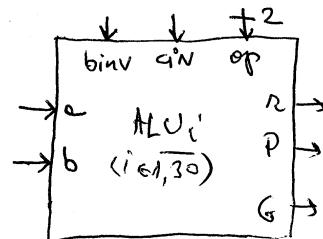
Observații:

- 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);
 - lt este generat de ALU_1 de la poziția 31 și emis prin poziția 0.

Unitate aritmetică și logică

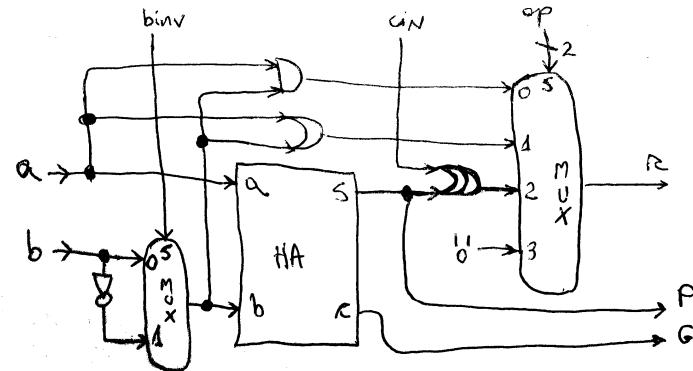
- Cazul $1 < i < 30$:

Simbol:



Unitate aritmetică și logică

Construcția circuitului:

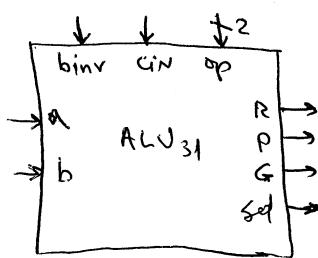


Observație: aici avem c_{in} , care vine din CL , iar $l{t}$ este mereu 0, deoarece numai în poziția 0 se poate emite ceva $\neq 0$ (la operația "l{t}" rezultatul este 0...00 sau 0...01).

Unitate aritmetică și logică

• Cazul 31:

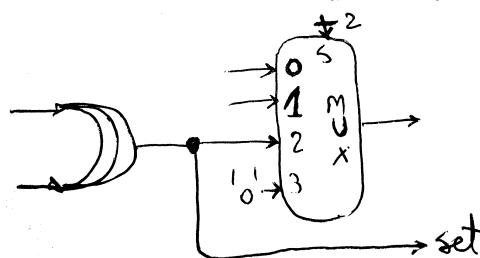
Symbol:



Observație: Apare încă plus ieșirea *Set*, care va intra ca *lt* în *ALU* de la poziția 0.

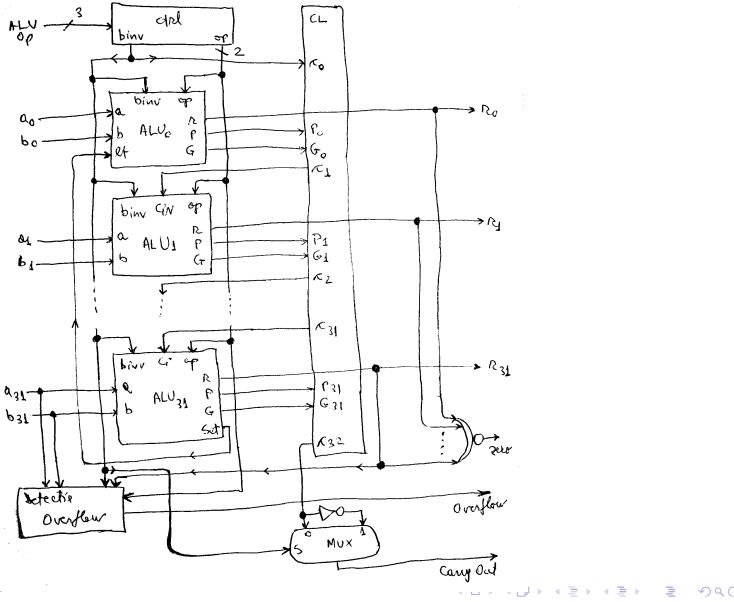
Unitate aritmetică și logică

Construcția circuitului în cazul 31 este asemănătoare celei din cazarile $1 \leq i \leq 30$, cu sigura diferență că ieșirea din "XOR"iese și prin "Set":



Într-adevăr, avem $It = 1$ 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_{1n} 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_{lt} se efectuează scăderea și se testează dacă rezultatul este < 0 , i.e. are bitul de rang 31 egal cu 1; acesta dă intrarea lt din ALU de la poziția 0, dar este scos la poziția 31 prin Set , deoarece la s_{lt} 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 $binv$ și op , pe baza următorului tabel:

	Ace_{Op_2}	Ace_{Op_1}	Ace_{Op_0}	$binv$	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
\sim	1	1	1	1	1	1

Exercițiu: Implementați acest circuit ca PLA , $PROM$ (sau mai simplu, observând că $binv = ALU Op_2$, $op_1 = ALU Op_1$, $op_0 = ALU Op_0$).

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/imprumut 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:

op_1	op_0	$binv$	x_{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

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 Performanța calculatoarelor
Conceptul de performanță
Măsurarea performanței
- 2 Arithmetica 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
Algebrie 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

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, registrii, 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 folosesc 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.

Există două tipuri de cicluri ce închid un CLC:

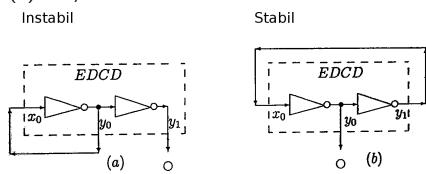
- **Cicluri stabilă**: conțin un număr par de complementări; ele generează o stare stabilă și sunt utile în construcția circuitelor digitale.
- **Cicluri instabile**: conțin un număr impar de complementări; ele generează o stare instabilă la ieșire și pot fi folosite la construcția ceasului.

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

Cicluri

Exemplu: În figura de mai jos, circuitul (a) conține un ciclu instabil, circuitul (b) conține un ciclu stabil:

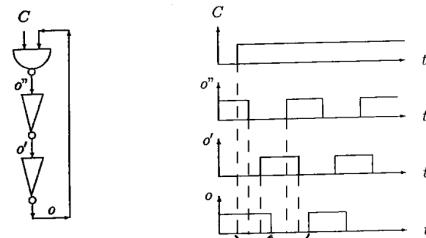


În cazul (a), dacă avem de exemplu starea $y_0 = 0 = x_0$, atunci vom obține la ieșire $y_1 = 0$ și noua stare $y_0 = 1 = x_0$, apoi vom obține la ieșire $y_1 = 1$ și noua stare $y_0 = 0 = x_0$, etc.; astfel, cele două ieșiri ale decodificatorului sunt instabile, comutând de pe 0 pe 1 și invers.

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 stable. Deocamdată însă nu știm cum să comutăm între stări, circuitul neavând o intrare prin care să putem controla schimbarea.

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 diagrama din dreapta. Constatăm că este nevoie de un anumit timp pentru ca semnalul să se propage 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).

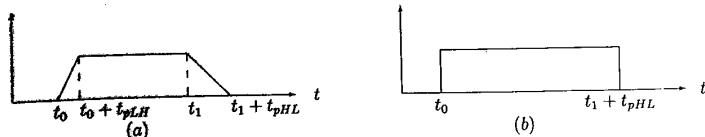
Cicluri

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.

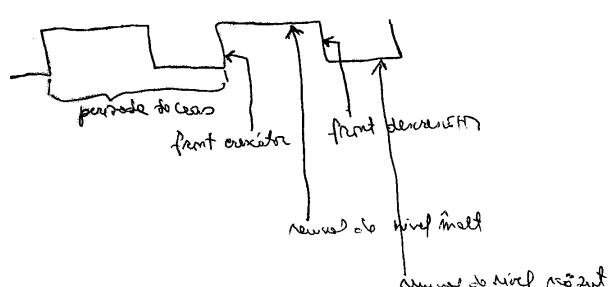
Uneori se consideră o situație ipotetică, de schimbare instantanea a stărilor, iar evoluția se aproximează ca în figura (b).

Ceasul produce un semnal autonom cu perioadă (frecvență) fixă.

Ei se folosesc 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. **actionarea pe frontul ceasului și se notează: respectiv** , respectiv **actionarea pe front este mai bună, deoarece se poate preciza mai exact momentul instalării noii stări.**

În metodologia actionării pe front, frontul crescător / descrescător care determină producerea schimbărilor de stare s.n. **front activ**.

Alegerea lui depinde de tehnologia implementării și nu afectează concepțile 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:

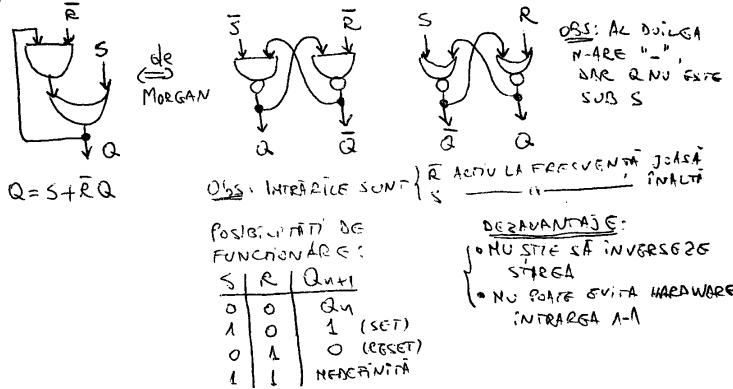
Zăvor elementar (LATCH):



POATE FIS COMANDAT
SĂ COMUTE IN
STARE A SI
PĂRĂSIE LA
ACTIV LA FRECUENȚA
INALTA

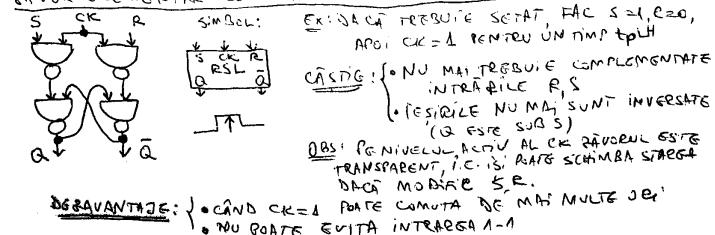
Zăvor elementar eterogen

Zăvor elementar cu ceas (RSL: RESET-SET LATCH):



Zăvor elementar cu ceas

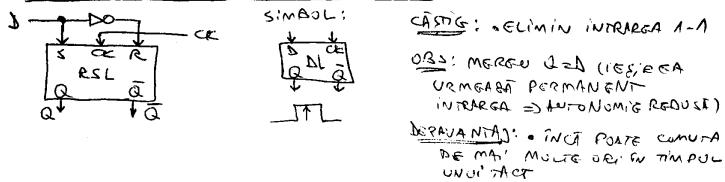
Zăvor elementar cu ceas (RSL: RESET-SET LATCH)



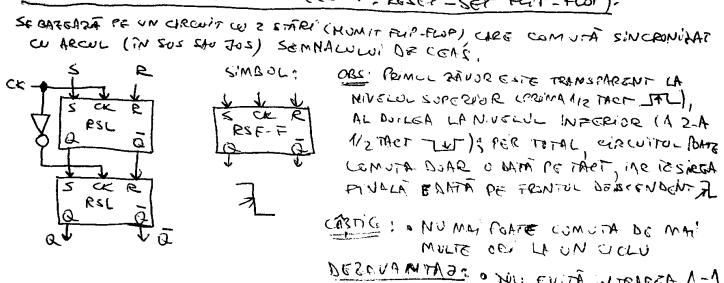
Zăvor de date

Structura master-slave

Zăvor de date (DL: DATA LATCH)

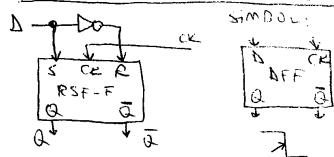


STRUCTURA MASTER-SLAVE (RSF-F: RESET-SET FLIP-FLOP):



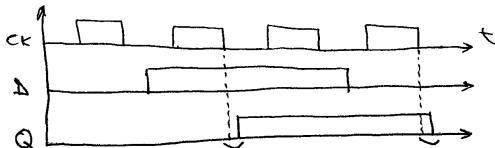
Flip-flop cu întârziere

FLIP-FLOP OU INTEGRATEUR (DFF: DELAY FLIP-FLOP)



CĂSTIG: • SE ELIMINĂ INTRAREA
A-⁺ LA S-R

OBS: IESIREA Q. CONȚINUTUL CU INTEGRAREA DE LA TACT
(MAI EXACT, IESIREA SE COLEGE LA PRIMUL SFÂRȘIT DE
TACT DE DUPĂ MODIFICAREA LUI D);



DIF ESTE UN CIRCUIT IMPORTANT (ARE ARICATTI MULTIE).

CIRCUITE DE MEMORIE SUNT:
 / RAVOARE (LATCH) REACȚIONARE DE CEAS RAVOARE ELEMENTAR
 \ BISTABILĂ (FLIP-FLOP) (SUNT ACȚIONARE DE CEAS) RSF-
 \ RAVOARE (LATCH) ACȚIONARE DE CEAS RAVOARE ELEMENTAR STERGEN
 \ BISTABILĂ (FLIP-FLOP) (SUNT ACȚIONARE DE CEAS) RFF

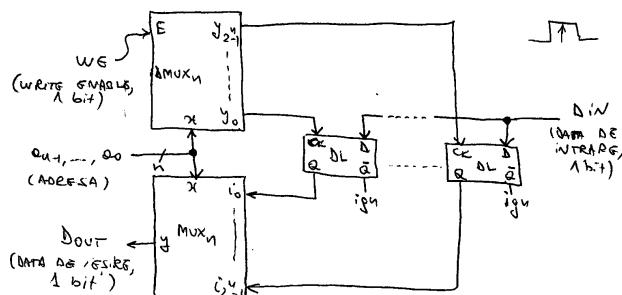
ASTĂZI LA ZÂVORARGLE ACȚIONATE DE CEAȘ CÂT SI LA BISTABILI,
IEȘIREA ESTE EGACĂ CU VALOAREA STĂRII MEMORATE ÎN ICONELE,

DIFERENTA între ele este momentul la care reașul provoacă scăderea stării.

- LA ZÂUARECE ACIONARE DE CEAS STAREA SE SCHIMBĂ DEZIGUR CÂTE OUI ÎNTRĂLILĂ SE SCHIMBĂ SI CEASUL SE ACTIVAT (ABICĂ ALARMĂ UN INTERVAL DE TRANSPARENȚĂ „TL”)
 - LA BISTABILI STAREA SE SCHIMBĂ DOAR PE FRONTUL DE CEAS (LA NOI „Z”)

RAM

MEMÓRIA RAM : ESTA É EXTENSÍVEL PARALELAMENTE - AS



EXEMPLIFICAREA SA FĂCUT PENTRU UN RAM CU LOCATII DE 1 BIT; PENTRU UN RAM CU LOCATII PE K BITI, FAREM O EXTENSIE PARALELĂ DE K RAM-URI CU LOCATII DE 1 BIT; WE SI A VOR FI CUMUNI, DIN S. DOUT VOR FOMA DATELE DE I/O DE K BITI.

RAM

Registrul serial

FUNCTIONARIES:

- LA CITIRE: CU $1, -1$, DO SELECȚEAZĂ PRIN MUX O CELULĂ SÌ, REDA PEIN DOUT INFORMAȚIA ÎNDRĂGĂTĂ ADOLO.
 - LA SCRİERE: CU $1, -1$, DO SELECȚEAZĂ PRIN DMUX O CELULĂ SÌ, TRIMITE ADOLO UE CA SEMNAL DE CEAS; DIN SE DISTRIBUIE LA TOATE CELULELE, DAR UB INTRA DOAR ÎN ACESA.

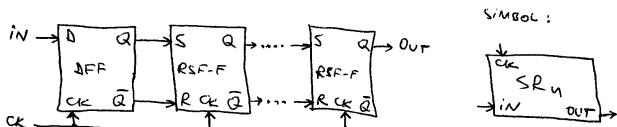
OBS. - RAM ESTE UN CIRCUIT SIMPLU SI SE poate construi UZOR LA DIMENSIUNI MARE SI EXISTA VARIANTE SI OPTIMIZARI - A SE VEDEA CARTEA P&H: SDRAM, DRAM, etc.

- RAM ADMITE SI O DEFINITIE RECURSIVĂ, PLECÂND DE LA DEFINITIA RECURSIVĂ A COMPONENTELOR SALE (EXERCITIU!)
- OBSERVAM CĂ ÎN MOD NATURAL RAM ARE UN NUMAR DE LOCURI ATLASATE A LUI Z.

REGISTRU: SISTEM DE F-F (REF-F SAU AFF) CAPABIL SA STOCHEZE CUVINGE DE N BITI, TOATE F-F AU LINII DE CONTROL (X, CEARA) COMUNE

DUPA MODUL DE EXTRAGERE. REGISTRUL PENTRU:

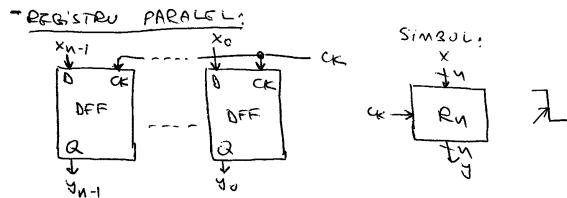
- PEG वर्षा २०१४ -



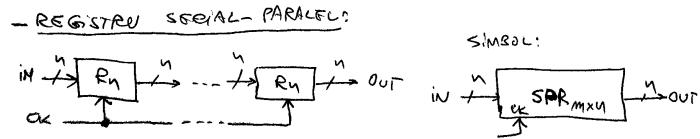
Obiectiv: • REGISTRUL INTRODUCÉ O ÎNTRĂMERE DE ÎMPREJUMLARE ÎNTRU INTRARE
S. ÎNSERIRE
• SE PUNTEAZĂ DEFINI SI RECURSIV (EXTENSIE SERIALĂ) (EXERCITIU)

Registru paralel

Registru serial-paralel

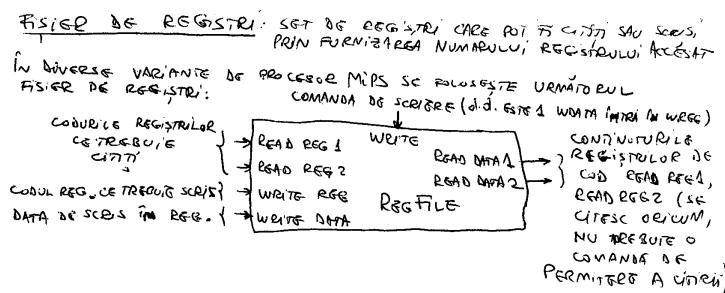


- Obz:
- PRINCIPALUL AVANTAJE ESTE NETRANSPARENȚĂ, CU EXCEPȚIA UNIEI "TRANSPARENȚE NEACCESIBILE" ÎN PRIMUL TAHU MOMEMTIC, DECI, PUTEA FI ÎNCHEAT CU UN NOU CICLU SÌ (PENTRU CÂM ESTE NETRANSPARENT) SE PUTEA ÎNCĂRCA CU O altă VALOARE, INCLUSIV UNA CE DEPINDE DE PROPIUL CONȚINUT,
 - R_n ADMITE SÌ DEFINITIE RECURSIVĂ (EXTENSIE PARALELĂ) (EXPECIUNIU!!)

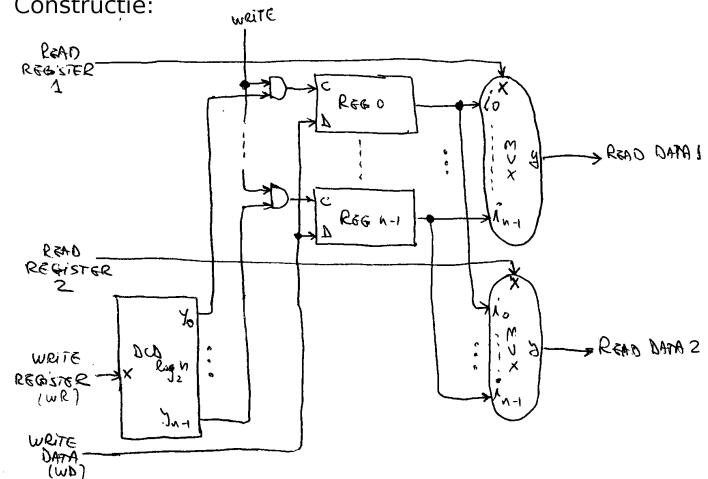


Fișier de registri

Fișier de registri



Construcție:



Fișier de registri

Aplicație la 1-DS

Exercițiu (aplicație la 1-DS):

Fie $\alpha(x) = \sum_{i=0}^n \alpha_i x^i \in \mathbb{Z}_2[x]$ fixat. Construim un circuit care preiaște successiv coecifientii lui, folinom $b_i \in \mathbb{Z}_2[x]$ și produce successiv coecifientii produsului: $\alpha(x)b(x)$.

Reboluare:

În \mathbb{Z}_2 avem $\{ + \text{(adunare)} = \wedge \text{(AND)} \} \wedge \text{ sunt asociative}$ și $\{ \cdot \text{(înmulțirea)} = \oplus \text{(XOR)} \} \oplus \text{ sunt comutative}$ (se poate verifica pe măsură operațiile).

$$\text{Dacă } b(x) = \sum_{i=0}^m b_i x^i \text{ atunci } \alpha(x)b(x) = \sum_{i=0}^{m+n} c_i x^i$$

$$c_i = \alpha_0 b_i + \alpha_1 b_{i-1} + \dots + \alpha_m b_0$$

$$\text{Explicitând calculul: } c_i = \sum_{j=0}^m \alpha_j b_{i-j}$$

$$\text{Pentru că } \alpha_i = \alpha_i b_0 + \alpha_i b_1 + \dots + \alpha_i b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

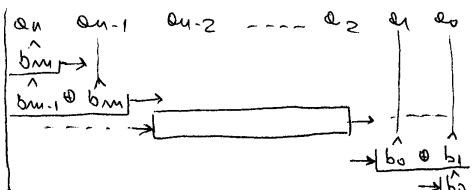
$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots + \alpha_m b_m$$

$$c_i = \alpha_0 b_0 + \alpha_1 b_1 + \dots +$$

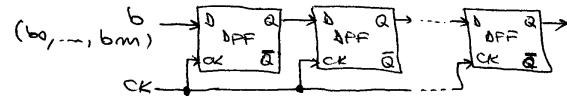
Aplicație la 1-DS

Puteam să presupunem că totalele sumele au doar b_j -urile ($n+1$ termeni), iar acolo unde termenul lipsesc avem b_{j+1} . Ilustrăm mai jos modul de calcul al celor $n+1$ sume de către hui termeni:

Obs. că "termenul" b_0, \dots, b_m AVANSEAZĂ PAS CU PAS spre dreapta și sub b_n, \dots, b_0 și LA fiecare PAS se înmulțesc (\wedge) b_j -urile sub care se află și se emite suma (\oplus) produselor ca sp.

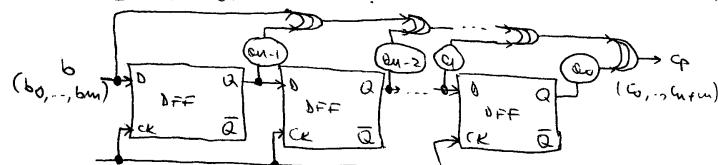


PENTRU IMPLEMENTARE, AVEM NEvoie ÎN PRIMUL RÂND DE UN CIRCUIT CARE SĂ STOCHEZE SEGMENTUL CURENT DE $n+1$ bituri care TREBUIE ÎNMULȚIȚI CU b_j -URII SI SA PERMIȚE AVANSUL PAS CU PAS SPRE DREAPTA, UN ASIMPȚONIC CIRCUIT PUTEA FI:



Aplicație la 1-DS

Acest CIRCUIT TREBUIE ÎMBOGĂȚIT CU UN CLC CARE LA PLECARE PAS SĂ CALCULEZE DIV. b_j -URILOR SĂ b_j -URILE DATE (FIXATE PRIN CIRCUIT) SP-UL CURENT. AVÂND ÎN VEDERE CĂ $b_j \in \{0, 1\}$, AT $b_j = 1$ (DA GRADUL W_i DIN Q_i), CIRCUITUL COMPLET PUTEA FI:



UNDE:

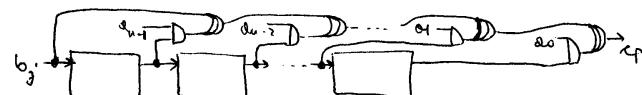


Aplicație la 1-DS

Obs. CONSTATĂM CĂ EXISTENȚILE SEPILE DE BISTABILI (ex. DFF) SE PRESENZĂ LA REZOLVAREA UNOR PROBLEME UNDE SE CEREGE PRELUCRAREA UNOR SÙRURI, ALG. CAROR ELEMENTE SUNT UNITE PE REND LA TACR. SUCEZIVI. EX: RECUNOASTEREA UNOR PATTERNU ÎNTRUL CINT. DE STRUCTURA GENERALĂ A CIRCUITULUI, ESIE O EXTENSIE SERIATĂ DE DFF + UN CLC CE EFECTUAREA CALCULULUI DE LA PLECARE PAS.

Aplicație la 1-DS

DACĂ DREBEM a_i și b_i SÙR FIE DATÈ DE INTRARE (DOAR NR. LOR MÌSÀ FIE FIXAT PRIN CONSTRUCȚIE), PUTEAM AREA URMAȚIURUL CLC:



Coefficienții a_i trebuie însă introduși toți o dată și menținuți la intrare la fiecare tact, în timp ce coeficienții b_i se introduc pe rând, la tacti succesivi.

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
 - Algebrelle booleene
 - Funcțiile 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

ÎN CARISSA P&H SE ARCE: AUT. MODRE AU AVANTAJUL CĂ POT FII MAI RAPIDE, AUT. MEALY AU AVANTAJUL CĂ POT FI MAI MICI (I.E. CU MĂRÎ PUTINĂ STÂRÎ).

CUM SE REZOLVĂ O PROBLEMĂ DE IMPLEMENTARE:

- DIN ENUNȚ EXTRAGEM O DESCRIEREA (SPECIFICARE DUL GRAF), OBȚINANDO TĂRILE PE NIVELUL S, A
- DIN TĂRILE CONSTRUIM CLC CA PLA/PROM ÎN MANIGRĂ ORICĂRUȚĂ S PESTUL CIRCUITULUI E STANDAR

OBS! UNGORI AUTOMATUL PUTEA FI SIMPLIFIAT ÎNINTE DE IMPLEMENTARE, DE EXEMPLU PUTEAM ELIMINA ANUMITE STÂRÎ CARE NU SUNT UTILIZATE (DE EX. SUNT ÎNCOREZIBILE); EXISTĂ ALGORITMI ÎN ACEST SENSI (NEZI LIMBAJE FORMALE).

Automatul DFF

Prezentăm câteva automate importante, ce pot fi folosite ca bistabili (flip-flop) de eficiență sporită:

OPERE DFF SE poate exprima ca un AUT. CU 2 STÂRÎ, UNAC CLC SE REDUCE LA FUNCȚIA IDENTICĂ A INTRĂRII

GRAF:

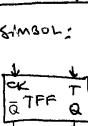
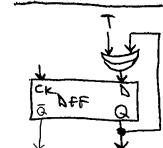


În acest caz, $Q^+ = D$, iar $y = Q$ (sau $y = D$ cu întârziere de un tact).

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

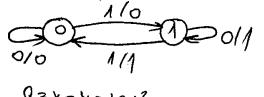
AUTOMATUL T FLIP-FLOP (TFF):



T	Q^+
0	Q
1	\bar{Q}

Defin $T = \begin{cases} r, & \text{pentru } 0 \rightarrow r \\ \bar{r}, & \text{pentru } 1 \rightarrow r \end{cases}$

GRAF:



UTILIZĂRI:

- COUNTER MODULO 2: dacă menținem $T=1$ timp de mai multă vreme, ieșirea va fi: 0, 1, 0, 1, ...
- DIVIZOR DE FRECUENȚĂ:

$$Q = x = y = \{0, 1\}^2$$

Exercițiu rezolvat (sinteză unui automat finit):

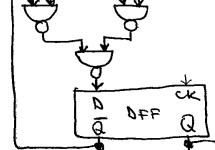
Automatul JKFF

Automatul JKFF

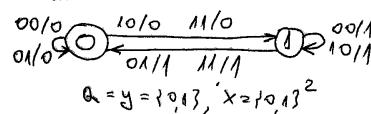
FUNCȚIONARE:

J	K	Q^+
0	0	Q (no op)
0	1	0 (reset)
1	0	1 (set)
1	1	\bar{Q} (switch)

AUTOMATUL JK FLIP-FLOP (JKFF):



GRAF:



Definiție: $JK = \begin{cases} r, & \text{pentru } 0 \rightarrow r \\ -\bar{r}, & \text{pentru } 1 \rightarrow r \end{cases}$
 (ASTfel, PUTEA ALGESE ÎN MAI MULTE TELEURI COMANDA PT. ACELASI EFEKT, S. V. ALGESE A.I. CLC SA IAȘĂ CĂT MAI SIMPLU)

COMENTARIU:

- AUTOMATELE TFF și JKFF, SPRE DEOARECE DE BISTABILI, PUFĂ STĂRÎ SĂ COMUNE ÎN CONTRARA STÂRÎI - LA ASTA ESTE NECESSAR CICLUL SUPLEMENTAR, ÎN PLUS, LA JKFF CAPĂTĂ SENSI INTRAREA I-1.

- JKFF ESTE CEL MAI BUN FLIP-FLOP DG PÂNĂ DE azi; EL LE GENERALIZĂ PE CELELalte.

$$\text{pt. } J = E = \bar{J} \Rightarrow \text{DFF}$$

$$\text{pt. } J = K = T \Rightarrow \text{TFF}$$

- CU JKFF SE PUTE CONSTRUI DIVIZORI DE FRECUENȚĂ

- IMPLEMENTAREA GENERALĂ A UNUI AUTOMAT SE PUTE face PRINCIPAL 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

Construieți 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 formeză secvența 1011).

Exemplu: IN: 0 1 0 1 0 1 1 0 1 0 0 1 1
OUT: 0 0 0 0 0 1 0 0 1 0 0 0 0

Rezolvare:

Problema este una de limbi formale - acolo se dau 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:

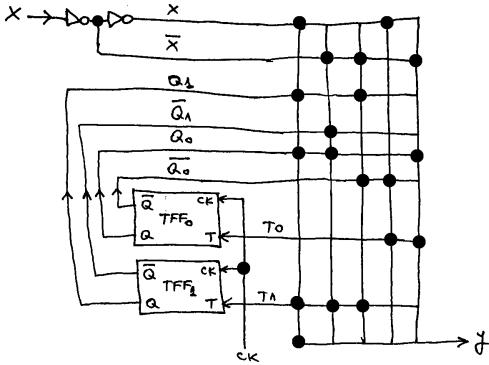
$$00, 01, 10, 11$$



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

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		q_1^+	q_0^+	J_1	K_1	J_0	K_0
q_1	q_0	0	1	0	x	0	-	x	-
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	-	\bar{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		q_1^+	q_0^+	J_1	K_1	J_0	K_0
q_1	q_0	0	1	0	x	0	-	x	\bar{x}
0	0	0 0	0 1	0	x	0	-	x	\bar{x}
0	1	1 0	0 1	\bar{x}	x	\bar{x}	-	x	\bar{x}
1	0	0 0	1 1	x	x	-	\bar{x}	0	\bar{x}
1	1	1 0	0 1	\bar{x}	x	-	\bar{x}	\bar{x}	x

Rezultă următoarele sume de produse minime:

$$\begin{aligned} J_1 &= q_0 \bar{x}, & J_0 &= x, & y &= q_1 q_0 x \quad (\text{a rămas la fel ca mai înainte}). \\ K_1 &= \bar{q}_0 \bar{x} + q_0 x, & K_0 &= \bar{x}, \end{aligned}$$

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 $J_0 = \bar{q}_0 \bar{x}$ (un termen cu două variabile, în loc de una).

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.

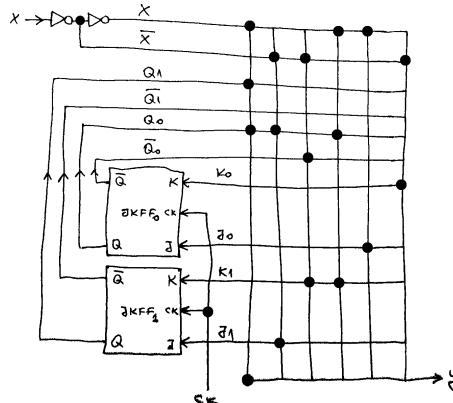
- În mod normal, implementarea cu JKFF ar fi trebuit să conducă la circuitul cel mai simplu (cel mai mic număr de puncte de contact), deoarece avem mai multe variante de valori J, K, dintre care putem alege una cât mai bună.

O explicație de ce nu s-a întâmplat aşa 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

Implementare:



Evaluarea complexității CLC: 17 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.

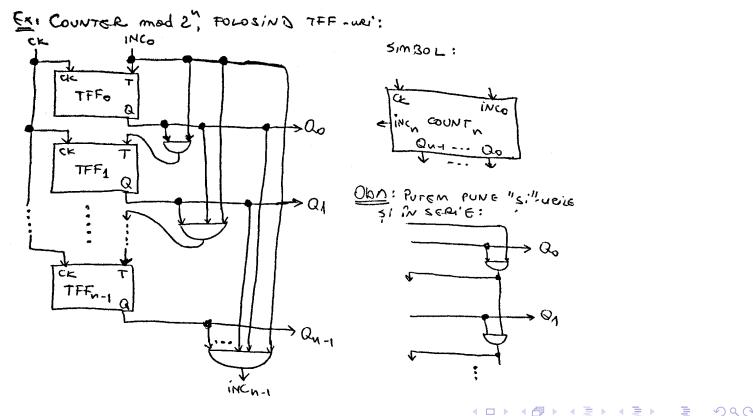
În acest caz este mai eficient să folosim instrumente simple: un editor de fișiere text (ex. "gedit") pentru codul sursă și un compilator în mod linie de comandă (ex. "gcc").

Counter

Counter

În continuare, prezentăm alte câteva automate importante, implementate ca 2-DS:

Numărător (Counter):

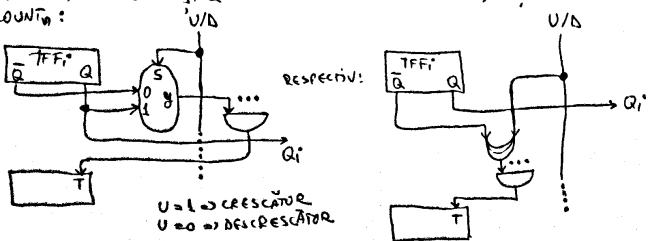


OBS: - IDEEA ESTE CA SE SCHIMBĂ VALOAREA BIȚULUI: URMAȚOR
DOAR DÂTĂ ÎN SPAȚIE ERAU DOAR 1-URĂ!
ACI NE ADUȚĂ FAȚA CĂ TFF SUNT NUMERATORI PÂNĂ LA 1

- COUNTERLE SE POT DEFINI și RECURSIV (COUNT_{n-1} IN SERIE CU UN TFF)
- DACĂ TOATE ÎNTRĂRIILE ÎN POORTILE AND SUNT DIN Q (ÎN LOC DE Q), DĂIN UN COUNTER DESCRISĂTOARE

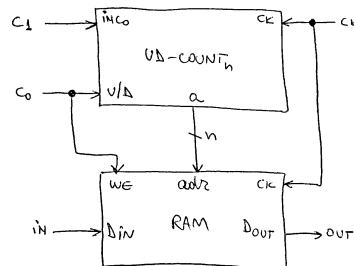
Counter

Se poate adăuga counterului o intrare suplimentară V/D care să indice dacă va număra creșător/descrescător și care va alege cu un SMUX între Q și \bar{Q} sau va face XOR cu Q , obțin circuitul V/D -COUNT:



Stivă

Stivă:

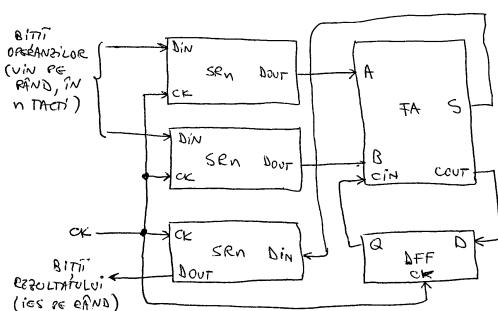


FUNCTIONS:

C0	C1	FCT
0	0	nr op
1	1	push (VA-LOUNTA ESTE INCREMENTAT IAR DATA IN EERE STOCATA (WE=1) LA ADRESA INDICATA DE NOU CONTINUT AL NUMERATORULUI)
0	1	pop (DATA DE LA ADRESA INDICATA DE NUMERATORUL ESTE EMISĂ LA OUT IAR NUMERATORUL ESTE DECREMENTAT LA NOU VÂRF AL STIVII)

Sumator serial

Sumator serial:



OBS: - ADUNĂ SECVENTIA ÎN MULȚIȘI CU DFF SG PĂSTREAZĂ/TRANSMITÈ CARRY-UL DE LA POZIȚIA ALĂTUĂ SE poate OPTIMIZA ÎMPĂRTIEND OPERANZIÎNU ÎN BII, C'SN GRUPURI DE M BII'S IN LOC DE SRU FOLOSIM SPRNXM) ÎN LOC DE FA FOLIOSIM ADDMIS
ERUITĂ: SUMATOR SERIAL PARALLEL

Sumator prefix

PRÉSUPÔUNSM CÂ U

VRZM SA CALCULAM SUCCESIV SUMGLE PREFIXE: $y_1 = x_1$
 $y_2 = x_1$

$$y_2 = x_1 + x_2$$

$\leftarrow x_1 + \dots + x_n$

DACĂ NUMERELE x_i SUNT PE M BI_i,
SUMA AR PUTESA DEPĂSI N BI_i S: HONCI

$$y_0 = x_1 + \dots + x_p$$

SUMA AR OUTRA DEZAS, N BII, 3; HONDA
TRESCAS LOCACI^E AS MULHERES REALIZA

THE BOSTONIAN

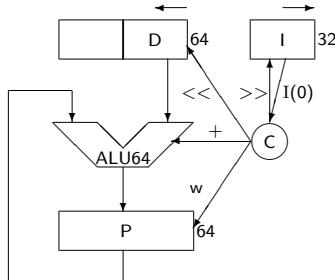
UN M CONVENIABIL - EX: $m+n = \lceil \log_2 (2^n-1) \rceil$

Înmulțire - metoda 1

```

D64, I32, P64 := 0
repetă de 32 x:
  dacă I(0) = 1
    P := P + D
  □
  D := D << 1
  I := I >> 1
  □

```



Aplicație: Calculați $6 \times 5 = 30$, $n = 4$. Completăți tabelul următor, fiecare coloană dublă conține valorile reședințelor LA SFÂRȘITUL unei etape:

	Initial	Iterația 1	Iterația 2	Iterația 3	Iterația 4
>>	I -	0101	-	-	-
+<	D 0000	0110			
P	P 0000	0000			

Rezolvare:

	Initial	Iterația 1	Iterația 2	Iterația 3	Iterația 4	
>>	I D P	0101 0000 0000	- 0010 0000	- 0001 1000	- 0000 0011	- 0000 0110
<+						

Înmulțire - metoda 1

Înmulțire - metoda 2

Analiza algoritmului:

Se efectuează $32\text{ pași} \times 3\text{ operații} \approx 100$ 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.

Obs. că prima jumătate a lui D este nefolosită (e necesară ca să putem aduna pe 64b).

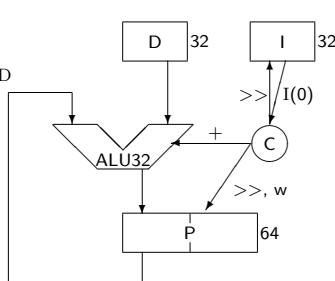
Soluția: facem D de 32b și în loc să deplasăm D deasupra lui P, deplasăm P pe sub D; întotdeauna vom aduna D cu prima jumătate a lui P (adunare pe 32b).

Înmulțire - metoda 2

```

    ┌── D32, I32, P64 := 0
    |   ┌── repetă de 32 x:
    |   |   ┌── dacă I(0) = 1
    |   |   |   ┌── P[63 - 32] := P[
    |   |   |   └──
    |   |   └──
    |   └──
    └──
    ┌── P := P >> 1
    └──
    ┌── I := I >> 1
    └──

```



Rezolvare:

	Initial	Iterația 1	Iterația 2	Iterația 3	Iterația 4
>> I	- 0101	- 0010	- 0001	- 0000	- 0000
+> D	0110	- 0110	- 0110	- 0110	- 0110
+> P	0000 0000 0011 0000	0001 1000 0011 1100	0001 1100 0001 1110	0110 0111	

Aplicație: Calculați $6 \times 5 = 30$, $n = 4$. Completăți tabelul următor, fiecare coloană dublă conține valorile registrilor LA SFÂRSITUL unei etape:

	Initial	Iterația 1	Iterația 2	Iterația 3	Iterația 4
>> I	- 0101	-	-	-	-
+ D	0110 -	-	-	-	-
>> P	0000 0000				

Înmulțire - metoda 2

Înmulțire - metoda 3 (finală)

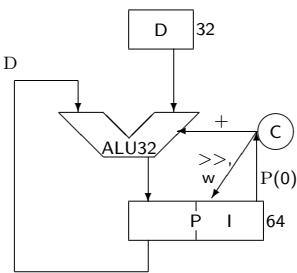
Înmulțire - metoda 3 (finală)

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.

```
D32, P64 := [0, ..., 0, I32]
repetă de 32 x:
  dacă P(0) = 1
    P[63 - 32] := P[63 - 32] + D
  P := P >> 1
  □
```

□



Aplicație: Calculați $6 \times 5 = 30$, $n = 4$. Completăți tabelul următor, fiecare coloană dublă conține valorile regiștrilor LA SFÂRȘITUL unei etape:

	Initial	Iterația 1	Iterația 2	Iterația 3	Iterația 4
D	0110	-	-	-	-
P,I	0000 0101	0011 0010	0001 1001	0011 1100	0001 1110

Înmulțire - metoda 3 (finală)

Înmulțire cu semn

Rezolvare:

	Initial	Iterația 1	Iterația 2	Iterația 3	Iterația 4
D	0110	-	0110	-	0110
P,I	0000 0101	0011 0010	0001 1001	0011 1100	0001 1110

Înmulțirea cu semn este asemănătoare, cu următoarele diferențe:

- se lucrează pe 31 biți, deci se fac 31 iterării, 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 shiftarea trebuie extins semnul produsului (shiftare aritmetică).

Înmulțire - algoritm 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: $\underbrace{1 \dots 1}_k = 2^k - 1 = 1 \underbrace{0 \dots 0}_k - 0 \dots 01$ 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 $0 \dots 01$.

$$\begin{array}{ccc} D & & I \\ \text{De ex. } 0010 * 0110 & = 0010 * 1000 - 0010 * 0010 \\ & & = 1000 - 0010 \end{array}$$

Practic shiftăm I la dreapta cu câte 1 și luăm decizii în funcție de cu încep/continuă/se termină grupurile de 1 sau 0 (algoritm 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 - algoritm Booth

```
D32, P[63, ..., -1] :=  $\overbrace{0, \dots, 0, I_{32}, 0}^{32}$ 
repetă de 32 x:
  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 $5 \times 6 = 30$ ($101 \times 110 = 11110$), $n = 4$. Completăți tabelul următor, fiecare linie conține valorile regiștrilor LA SFÂRȘITUL unei etape:

	I	D	P
Initial	0000 0101	0110 0000	0000 0000
Iterația 1			
Iterația 2			
Iterația 3			
Iterația 4			

Înmulțire - algoritmul Booth

Rezolvare:

Inițial	0000	0110	0
	0101		
Iterația 1	0000	0011	0
	0101		
Iterația 2	1011	0011	0
	1101	1001	1
Iterația 3	1110	1100	1
	0101		
Iterația 4	0011	1100	1
	0001	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:

$$\begin{array}{r} D = 1001010 \\ -1000 \\ \hline 10 \\ 101 \\ 1010 \\ -1000 \\ \hline 10 = R \end{array}$$

D.p.v al mașinii:

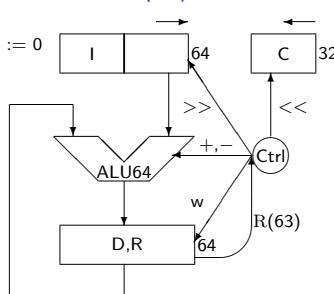
- a vedea dacă I "se cuprinde" revine la a scădea $D - I$ și a compara cu 0 (i.e. a testa bitul cel mai semnificativ);
- dacă $d_3 \geq 0$, adăugăm 1 la C ;
- dacă $d_3 < 0$, adunăm $D + I$ la loc și adăugăm 0 la C ;
- adăugarea unei cifre la C înseamnă $<< 1 +$ cîfră;
- coborârea cifrei următoare înseamnă shiftarea lui I pe sub D la dreapta (metoda 1) sau a lui D pe deasupra lui I la stînga (metodele 2,3), pentru a face altă suprapunere la scădere.



Împărțire - metoda 1

```

R64:=D, I64:=I, 0, ..., 0, C32 := 0
repetă de 33 x:
  R := R - I
  dacă R ≥ 0 (i.e. R(63) = 0)
    C := C << 1 + 1
  altfel
    R := R + I
    C := C << 1 + 0
  I := I >> 1
  
```



Împărțire - metoda 1

Rezolvare:

	Initial	Iterația 1	Iterația 2	Iterația 3	Iterația 4	Iterația 5
<<	C - 0000	- 0000	- 0000	- 0000	- 0000	- 0010
>>	R,D 0000 0111	0000 0111	0000 0111	0000 0111	0000 0111	0000 0001

Aplicație: Calculați 7:3 (adică 111:11), $n = 4$. Completati tabelul următor, fiecare coloană dublă conține valorile regiștrilor LA SFÂRȘITUL unei etape:

	Initial	Iterația 1	Iterația 2	Iterația 3	Iterația 4	Iterația 5
<<	C - 0000	-	-	-	-	-
R,D 0000 0111	0000 0111					
>>	I 0011 0000					



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 stînga.

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 $n+1$ iterării, deci am avea un cât de forma $1\dots$ și nu ar încăpea).

33 cifre

Soluția: se permute operațiile de shiftare și scădere (se face întîi shiftarea și apoi scăderea), eliminându-se o iterărie; la sfîrșit restul este în jumătatea stîngă a lui R . Se obține **metoda 2**, dar nu o prezintăm, ci trecem direct la **metoda 3**, unde în plus se pune C în jumătatea dreaptă alui R - cum R și C se shiftează sincron cu 1 la stînga, nu se pierde nimic din R , C.

Singura problemă este că la sfîrșit jumătatea stîngă a lui R este prea shiftată și se shiftează la dreapta cu 1.

```

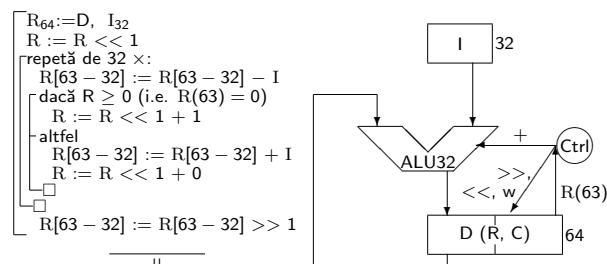
R64:=D, I32
R := R << 1
repetă de 32 x:
  R[63 - 32] := R[63 - 32] - I
  dacă R ≥ 0 (i.e. R(63) = 0)
    R := R << 1 + 1
  altfel
    R[63 - 32] := R[63 - 32] + I
    R := R << 1 + 0
  R[63 - 32] := R[63 - 32] >> 1
  
```

R[63 - 32] = restul, R[31 - 0] = câtul

Aplicație: Calculați 7:3 (adică 111:11), $n = 4$, completând tabelul următor:

Inițial	0000	0111	
R << 1			
Iterația 1			
Iterația 2			
Iterația 3			
Iterația 4			
R[63 - 32] >> 1			

Împărțire - metoda 3



Rezolvare:

Initial	0000 0011	0111
$R << 1$	0000 0011	1110
Iterația 1	1101 0001	1100
Iterația 2	0011 0011	1000
	R[63 - 32] >> 1	

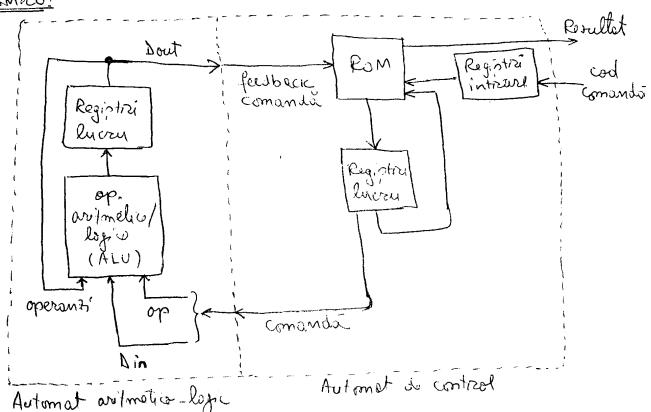
Iterația 3	0000 0001 0011	0001
Iterația 4	1110 0010 0011	0010
	R[63 - 32] >> 1	

Procesor:

3-DS (Procesoare)

Un procesor este un circuit ce leagă într-un ciclu un automat aritmico-logic cu un automat de control.

Exemplu:



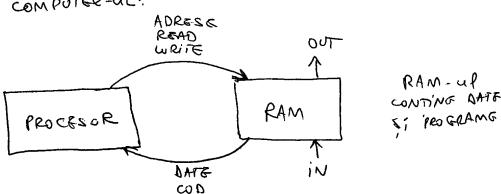
3-DS (Procesoare)

OBS: Organizările a procesorului pot difera mult de la arhitectură la alta; în continuare vom studia aceleași detalii în cazul arhitecturii MiP5.

Calculator:

4-DS (Calculatoare)

Exemplu: COMPUTERUL.



TODO: Mai multe detalii și exemple despre 3-DS și 4-DS.

OBS: PROCESORUL SE poate înținde și în:

- CLC (0-DS)
- STIVĂ (2-DS)
- CO-PROCESOARE (3-DS)

OBS: NESTACIULILE DE ORGANIZARE

OBS: S'ÎN CADUL CALCULATORELOR NESTACIULILE DE ORGANIZARE POT DIFERI MULT DE LA ARHITECTURA LA ALTA.

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)

Algoritm de înmulțire și împărțire hardware

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

n -DS, $n > 4$

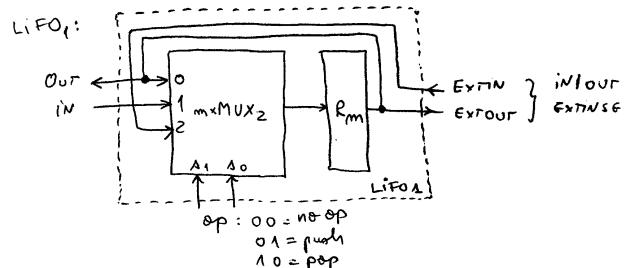


Stivă ca n -DS

Stivă ca n -DS:

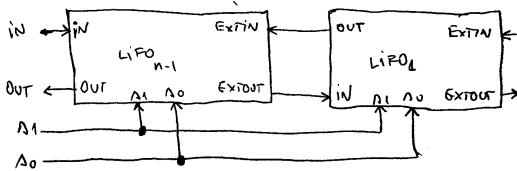
Unele circuite se pot organiza ca n -DS cu $n \geq 4$, căci
două se pot organiza și mai simplu.

Exemplu: Stivă organizată ca n -DS (am văzut că se poate implementa și ca 2-DS):



Stivă ca n -DS

LIFO_n:



Deci, LIFO_n este n -DS.

Acest exemplu ne arată că 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 ușor, funcționarea este deterministă iar comportamentul la un moment dat este unic determinat de conținutul curent din sistem și de intrarea curentă.

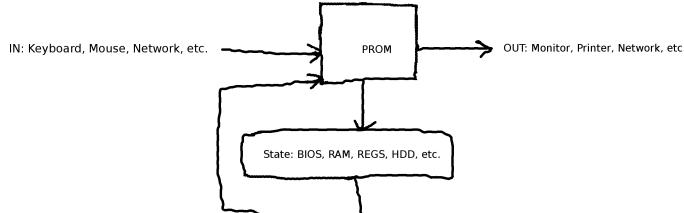
Astfel, am putea construi orice calculator ușor ca un 2-DS (automat finit), în care:

- elementul de stare (de exemplu un registru foarte mare) conține toată informația curent stocată în sistem, inclusiv datele din BIOS, RAM, regiștrii 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 acelaiași 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").