UNIVERSITATEA TEHNICĂ DIN TIMIŞOARA
Facultatea de Automatizări şi Calculatoare

Autor:
Ing. Nikolaos S. PETRAKIS

CREŞTEREA FIABILITĂŢII ŞI DISPONIBILITĂŢII SISTEMELOR DE CALCUL PRIN EFICIENTIZAREA AUTOCONTROLULUI

~ Teză de doctorat ~

Conducător ştiinţific:
Prof. dr. ing. Mircea VLĂDUŢIU

Timişoara
1995

UNIVERSITATEA TEHNICĂ DIN TIMIŞOARA
Facultatea de Automatizări şi Calculatoare

Autor:
Ing. Nikolaos S. PETRAKIS

CREŞTEREA FIABILITĂŢII ŞI DISPONIBILITĂŢII SISTEMELOR DE CALCUL PRIN EFICIENTIZAREA AUTOCONTROLULUI

~ Teză de doctorat ~

Conducător ştiinţific:
Prof. dr. ing. Mircea VLĂDUŢIU

Timişoara
1995

Cuprins

Capitolul 1.

Capitolul 2.

Capitolul 3.

Capitolul 4.

1. Introducere

Fiabilitatea unui sistem a fost o preocupare majoră încă din momentul apariției calculatoarelor electronice numerice. Există preocupări legate de analiza de fiabilitate, constând în evaluarea de indicatori numerici, atât previzional cât şi experimental, pentru dezideratul siguranței în funcționare. Pe lângă această arie de preocupări, fiabilitatea implică sinteza unor sisteme de calcul care prin proiectare să îndeplinească anumiți indicatori de fiabilitate impuşi. În acest sens se apelează la metode de creştere a fiabilității, care, în accepțiunea cea mai largă, sunt divizate în două clase: evitarea defectelor (fault avoidance) sinonim cu netolerarea defectelor (fault intolerance) şi tolerarea defectelor (fault tolerance). Obiectivul esențial al unui sistem fiabil este reprezentat de către defect, fie în sensul evitării acestuia, fie în sensul tolerării lui, context în care se impun unele precizări legate de terminologie. Asupra acestora se insistă în extenso în capitolul doi al lucrării, dar anticipând, vom înțelege prin defect orice deviere de la funcționarea normală a unui sistem, aceasta putând fi cauzată fie prin erori de proiectare, fie prin anomalii ale echipamentelor, fie prin erori ale operatorilor sau ale personalului de întreținere, acestea din urmă denumite în lucrare erori de interacțiune. Prima clasă de metode foloseşte componente de înaltă fiabilitate, tehnici de proiectare conservative şi recenzii vaste în faza de proiectare, pentru eliminarea defectelor de proiectare. Obiectivul acestor metode este să reducă posibilitatea de defectare. Totuşi, eşecuri apar chiar şi în cazul celei mai atente evitări de defecte, de aici termenul de netolerare a defectelor (fault intolerance). În mod firesc, se justifică existența celei de a doua clase de metode care, în scopul de această dată de a tolera defectele, foloseşte componente adiționale numite redundante. Pentru claritate, amintim că indicele aproape unanim recunoscut ca obiectiv de proiectare este raportul performanță/cost. În vederea optimizării acestuia, majoritatea tehnicilor de proiectare apelează la eliminarea redundanței. În mod contrar, față de tehnicile anterioare, noile metode introduc redundanța pentru a conferi sistemelor acest aparte deziderat al proiectării care îl reprezintă fiabilitatea prin tolerare la defecte. Redundanța poate fi introdusă prin soluții eminamente hardware, eminamente software sau soluții hibride (hardware/software), dar în toate cazurile se urmăreşte evitarea efectelor defectelor. Amândouă tehnicile sunt uzual folosite în sistemele de înaltă fiabilitate.

În trecut, sistemele de calcul tolerante la defecțiuni ocupau doar o mică parte din piața comercială din cauza costului excesiv de mare. Multe sisteme au fost proiectate ca să dea soluții unor aplicații specializate, dar în ultimul timp, calculul tolerant la defecțiuni începe să joace un rol important în proiectarea calculatoarelor comerciale din mai multe motive, cum sunt: progresele deosebite în tehnologia VLSI, posibilitatea de integrare pe tot wafer-ul (WSI), costul mare de întreținere al echipamentelor sofisticate, câmpul vast de aplicații care pot fi abordate de un sistem şi pretențiile utilizatorilor pentru fiabilitate superioară. Deoarece calculatoarele continuă să preia din ce în ce mai multe servicii critice pentru funcționarea cu succes a spitalelor, uzinelor producătoare, combinatelor energetice, băncilor, oficiilor, etc., au crescut substanțial pretențiile utilizatorilor şi este imperativ necesară construirea şi utilizarea sistemelor tolerante la defecțiuni.

Termenul de calculare tolerantă la defecțiuni (fault-tolerant computing) este definit ca abilitatea de a calcula în prezența erorilor. Obiectivul calculării tolerante la defecțiuni este să dezvolte sistemele de calcul şi să asigure funcționarea lor într-un mod satisfăcător în prezența defecțiunilor. Datorită progreselor tehnologiei

semiconductoarelor s-a produs o creştere a fiabilităţii hardware-ului, în timp ce fiabilitatea software-ului scade cu o rată neaşteptat de mare odată cu creşterea complexităţii software-ului. În prezent au fost făcute câteva progrese în software-ul tolerant la defecţiuni, dar încă rămâne un teren de cercetare vast.

În ultimul deceniu complexitatea circuitelor, referitor la numărul de porţi, a mărit costul testării. Această creştere nu apare numai din cauza mărimii circuitului, ci şi din cauza modalităţii în care a fost proiectat. Acest lucru înseamnă că în faza de proiectare s-a pus accentul pe implementarea funcţiilor circuitului fără să se ţină cont de modul în care se va testa ulterior acest circuit.

Independent de mărimea circuitului, diferiţi factori fac dificilă testarea. Primul este imposibilitatea iniţializării elementelor de memorare din circuit într-o stare cunoscută; în trecut multă muncă de cercetare a fost direcţionată spre problema generării secvenţelor de iniţializare (homing sequences) ale căror obiectiv a fost să conducă circuitul într-o anumită stare cunoscută indiferent de starea lui prezentă. Al doilea factor este imposibilitatea controlării şi observării semnalelor care se află în nodurile interioare din circuit. Ultimul factor este realizarea funcţiilor logice utilizând tranzistoare de trecere (pass transistors) care nu pot fi modelate în mod convenţional în porţi echivalente. Pe de altă parte, patul de cuie sau în general metodele de testare în circuit (in-circuit), larg utilizate până acum, nu se mai pot folosi la testarea plăcilor realizate în noile tehnologii din cauza geometriei micşorate şi sensibilităţii ridicate ale acestor plăci.

Recent, au evoluat câteva metodologii de proiectare ale căror obiectiv este să reducă nu numai costul de generare a configuraţiilor binare de test (patterns), dar şi al testării în general, prin:

(1) realizarea nodurilor interne mai accesibile;
(2) transformarea, pe motive de testare, a circuitelor secvenţiale în unele combinaţionale şi/sau descompunerea circuitelor complexe în unele mai puţin complexe;
(3) realizarea de circuite autotestabile (self-testing);
(4) reducerea cantităţii de date de test necesare pentru testarea circuitelor.

Chiar dacă metodologiile de proiectare care îmbunătăţesc testabilitatea unui circuit sunt foarte dorite, ele afectează totuşi negativ libertatea proiectantului, precum şi performanţele circuitului. Pentru a se accepta un anumit stil de proiectare în vederea uşurării problemei de testare, el trebuie să fie uşor de aplicat şi cu cât mai puţine restricţii, ca să nu inhibe ingeniozitatea proiectantului; de asemenea, trebuie să aibă suportul software adecvat, adică programe care vor verifica la un proiect respectarea regulilor de proiectare impuse de un anumit stil. În plus, din moment ce majoritatea metodelor apelează la hardware adiţional, creşte mărimea fizică a circuitului având un efect negativ asupra producţiei lui; hardware-ul adiţional introduce întârzieri suplimentare semnalelor, lucru care afectează performanţele circuitului. Apare, de asemenea, necesitatea pinilor adiţionali de intrare/ieşire, care măresc costul de împachetare. Având în vedere efectele negative relevate, pentru justificarea unui anumit stil particular de proiectare se impune ca acesta să determine o reducere considerabilă a costurilor de generare a experimentului de testare, în condiţiile respectării performanţelor circuitului şi investiţia în efort de proiectare.

În general, proiectarea pentru testabilitate (Design For Testability) implică modificarea circuitului pentru îmbunătăţirea procesului generării pattern-urilor de test precum şi a celui de aplicare ale acestora. Tehnicile de îmbunătăţire a testabilităţii sunt

grupate în trei categorii principale: metode de circumstanță (ad hoc), metode structurate şi metode de incorporare a mecanismelor de implementare a testării (built-in test). Metodele de circumstanță, cum sunt: inserarea punctelor de test, utilizarea logicii de blocare şi a registrelor de stare a testării, etc., nu au evoluat pentru a rezolva testarea unui circuit complex ci datorită nevoii de a rezolva o anumită problemă de testare. În contrast, metodele structurate sunt mai formale şi nu se introduc ulterior ci sunt incorporate în proiect de la începutul lui. Metodele de testare incorporată încearcă să reducă cantitatea datelor de test, care trebuie transferate şi procesate, fiind deosebit de utile mai ales în cazul testării circuitelor complexe.

Un sistem digital este constituit din componente logice care se pot deteriora în timpul operării normale. După ce s-a defectat sistemul, inginerii trebuie să identifice orice componente defecte şi să le înlocuiască. Identificarea, sau diagnosticarea erorii poate lua mult timp şi poate întrerupe servirea (funcționarea sistemului). În multe aplicații, cum sunt sistemele de comutare pentru telecomunicații (telephone-switching systems) şi sistemele de ghidare ale sateliților (satellite-guiding systems), aceste întreruperi sunt inacceptabile sau în cel mai bun caz foarte costisitoare. Pe de altă parte, dacă defecțiunea nu este localizată rapid, ea poate afecta multe date valoroase.

O soluție ideală pentru a trata sistemele defecte constă în abilitatea identificării rapide a unei defecțiuni şi în existența unui alt sistem, de rezervă (backup system), pentru a prelua şi a continua servirea. Sistemul de rezervă inițiază câteva mecanisme care activează operația de întoarcere înapoi (rollback) pentru a restabili (recovery) sistemul şi apoi instruieşte sistemul pentru a-şi relua operarea din acest punct. Unitatea defectă se semnalizează singură ca o sursă cu probleme. Pentru a implementa această soluție la nivel macro, ar trebui să se utilizeze circuite combinaționale cu autoverificare (self-checking). Aceasta înseamnă că fiecare circuit realizează funcția lui şi semnalizează orice defecțiune din interiorul lui într-o manieră observabilă.

O presupunere tradițională în testarea sistemelor defecte este aceea că o unitate nu are mai mult decât o defecțiune. Această presupunere se bazează pe faptul că un sistem este testat chiar înainte de intrarea lui în operare, şi în acel moment nu are defecțiuni. De asemenea, dacă apar defecțiuni, ele apar mai probabil una câte una decât de odată. Presupunem că o a doua defecțiune apare într-un circuit deja defect. Dacă ea apare înainte de detectarea primei defecțiuni, presupunerea tradițională nu mai este valabilă. Astfel, trebuie făcută o altă ipoteză: timpul dintre apariția unei defecțiuni şi detecția ei să fie destul de mic pentru a fi detectată înainte de apariția unei alte defecțiuni. Această perioadă dintre apariție şi detecție este numită perioada de latență a erorii (error latency period), deoarece o defecțiune este prezentă dar nu este încă detectată.

În majoritatea circuitelor cu autoverificare (self-checking circuits) disponibile azi, ieşirile circuitului sunt codificate printr-un cod detector de erori. Dacă într-un circuit apare o defecțiune, atunci sunt afectate ieşirile acestuia, şi se poate observa această schimbare. În unele situații, o defecțiune poate afecta mai multe ieşiri, iar codul va apărea corupt. În situația cea mai defavorabilă, o defecțiune poate afecta aproape toate ieşirile iar construcția codului va deveni foarte dificilă. Pentru a evita această problemă, proiectanții se limitează la circuite cu autoverificare (self-checking) în care o defecțiune poate afecta mai multe ieşiri, dar toate schimbările de ieşiri sunt în aceeaşi direcție. Aceasta înseamnă că toate sunt fie 0-uri care se schimbă în 1, fie 1-uri care se schimbă în 0. Aceeaşi defecțiune nu poate cauza ambele tipuri de schimbări.

Codurile numite coduri detectoare de erori unidirecționale detectează astfel de erori. Un exemplu este codul m din n, în care un circuit produce n ieşiri, din care exact m dintre ele sunt pe 1, iar restul (n-m) sunt pe 0.

Aşa cum s-a spus înainte, o defecţiune care cauzează la ieşire o schimbare unidirecţională poate modifica fie 1-urile, fie 0-urile, dar nu ambele. Astfel, acest tip de eroare se poate detecta relativ simplu. Clasa circuitelor ale căror defecţiuni produc erori unidirecţionale este totuşi restrânsă. Dacă se doreşte ca aceste circuite să aibă funcţii arbitrare trebuie, de asemenea, să se codifice intrările lor. Dacă se codifică intrările unor astfel de circuite într-un cod detector de erori unidirecţionale, se pot proiecta aceste circuite pentru a realiza funcţii arbitrare.

Pe de altă parte, circuitele VLSI sunt critice pentru progresul industriei electronice. Din moment ce piaţa a devenit mai competitivă, proiectanţilor li s-a cerut să construiască circuite cu performanţe mult mai bune, fără defecte şi cu producţie mare, în timp ce timpii de proiectare şi de fabricare se menţin la un minim. Tehnicile de sinteză asistată de calculator (computer-aided synthesis) pentru circuite VLSI au fost utilizate pentru realizarea acestor ţinte. Implementările comerciale ale sistemelor de sinteză au devenit practice pentru proiectarea circuitelor digitale la nivel de producţie. În special, au fost utilizate cu eficacitate tehnicile de sinteză a circuitelor şi de optimizare pentru a sintetiza sisteme sau componente de la specificaţiile (caracteristicile) de nivel înalt, cu intervenţie umană limitată sau chiar fără intervenţie umană.

Prezenta teză de doctorat abordează aspectul creşterii fiabilităţii şi disponibilităţii sistemelor numerice de calcul valorificând avantajele oferite prin utilizarea autocontrolului. Lucrarea se extinde pe 6 capitole plus două anexe şi o lista bibliografică.

După o scurtă introducere care constituie conţinutul primului capitol, capitolul 2 cuprinde tatonarea domeniului, unele precizări cu privire la terminologia uzitată, precum şi o trecere în revistă a metodelor care sporesc tolerarea defecţiunilor în sisteme de calcul, insistând asupra celor care se bazează pe autocontrol implementat atât software, cât şi hardware. Sinteza originală rezultată constituie punctul de plecare a prezentei lucrări.

Capitolul 3 abordează problematica specifică autocontrolului bazat pe hardware, prezentându-se în prima parte a capitolului o clasificare a metodelor de autocontrol într-o manieră originală. Ca şi criteriu de clasificare, autorul utilizează nivelul de structură dintr-un sistem la care se adresează proiectarea în scopul implementării autocontrolului. Astfel, apar metode de proiectare pornind de la nivel de bistabil, la nivel de registru, la nivel de bloc ajungând chiar şi până la nivel de procesor. În a doua parte a acestui capitol se tratează problematica proiectării circuitelor de verificare cu proprietăţi speciale cum ar fi testabilitatea, autotestabilitatea şi altele, precum şi a circuitelor cu autoverificare totală, restrângându-se astfel domeniul de investigaţii.

În cadrul capitolului 4, urmărind valorificarea eficientizării autocontrolului, am focalizat cercetările asupra părţii de aritmetică a unui sistem de calcul, ţintind operaţia fundamentală care prezintă cea mai dificilă gamă de probleme şi anume împărţirea binară. Se prezintă un studiu asupra operaţiei de împărţire, evidenţiindu-se principalele avantaje şi dezavantaje ale metodelor existente. Plecând de la algoritmul de împărţire fără refacerea restului, autorul stabileşte un nou algoritm de împărţire având capacitatea să opereze numere cu semn, în virgulă fixă. Autorul fructifică cunoştinţele de specialitate prezentate sintetic în capitolele precedente (cumulate în timpul doctoratului), grefând controlul de paritate pe o schemă originală de împărţire paralelă bazată pe algoritmul propus.

Capitolul 5 prezintă rezultatele experimentale obținute prin simularea schemei propuse, precum şi detalii legate de simulatorul utilizat, simulator realizat de autorul tezei în scopul susținerii practice a metodei propuse.

Capitolul 6 conține concluziile şi sinteza contribuțiilor originale aduse de autor pe parcursul lucrării.

Autorul tezei ține să mulțumească în mod deosebit conducătorului ştiințific, domnului prof. dr. ing. Vlăduțiu Mircea, pentru îndrumarea competentă, extrem de eficientă şi plină de înțelegere acordată de a lungul perioadei de pregătire a doctoratului.

Autorul mulțumeşte de asemenea cadrelor didactice din catedra de calculatoare, precum şi foştilor colegi de facultate cu care a colaborat, pentru discuțiile purtate, pentru sugestiile şi recomandările, precum şi pentru încurajările primite.

În final, autorul doreşte să dedice această lucrare părinților săi ca o modestă manifestare de recunoştință pentru sprijinul acordat atât material, cât şi moral, în toți acești ani de formare şi pregătire profesională.

2. Metode de creştere a fiabilităţii şi disponibilităţii

2.1. Indicatori pentru caracterizarea fiabilităţii şi disponibilităţii

2.1.1. Clasificarea defecţiunilor

Într-un sistem există trei categorii majore de surse de erori: greşeli de proiectare, defecţiuni fizice, şi erorile de interacţiune ale operatorului uman (procedural errors). Aceste surse de erori contribuie la deteriorările sistemului. Chiar dacă greşelile de proiectare apar atât în hardware cât şi în software, cele din urmă sunt predominante şi se elimină greu din sistem [TOY88]. Pe de altă parte, defecţiunile sunt rezultatul îmbătrânirii componentelor sau a influenţei mediului care cauzează devierea unor caracteristici ale echipamentului peste limitele câmpurilor de toleranţă specificate. Se produc în acest mod erori cauzate de aşa numitele defecţiuni parametrice hardware. Acţiunile improprii ale operatorului la panourile de control şi de întreţinere pot determina deteriorarea (malfuncţionarea) sistemului. Ceea ce introduce operatorul este de prioritate înaltă, şi multe sisteme sunt fatal vulnerabile la comenzi improprii şi la erori de operare.

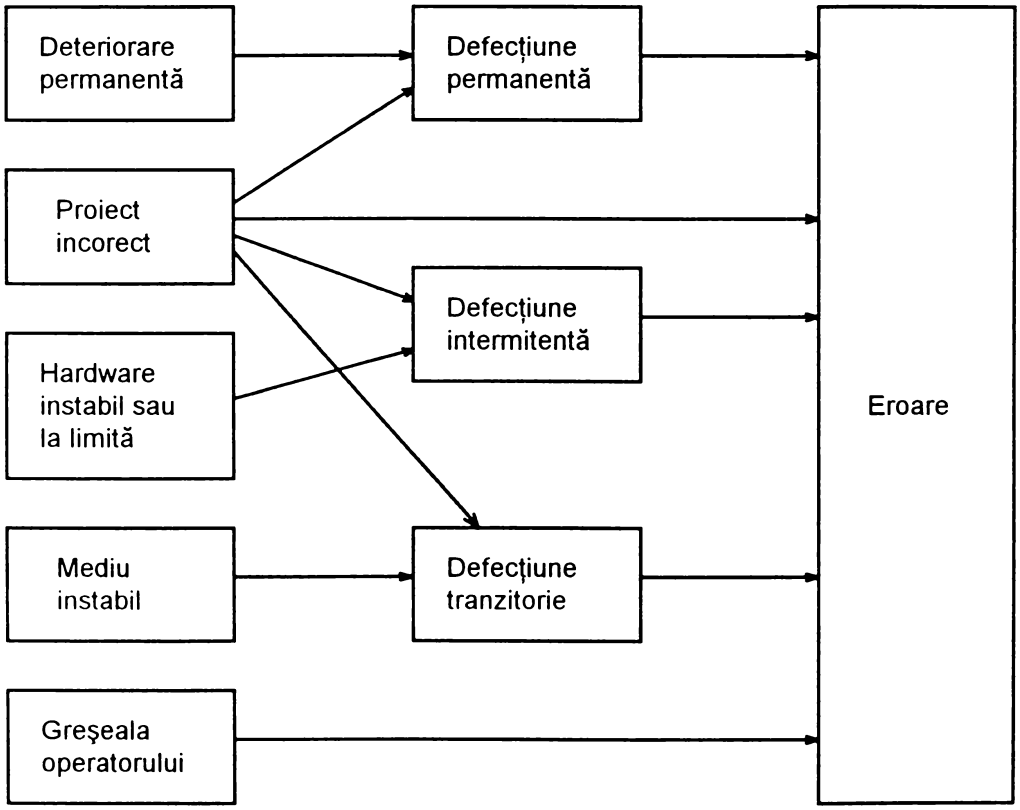


Figura 2.1 Surse de erori.

Proiectarea unui sistem tolerant la defecţiuni, necesită găsirea unei metode care să împiedice defecţiunile logice, provenite de la deteriorări fizice, să cauzeze o eroare. Figura 2.1 prezintă sursele posibile ale unei erori [SIEV91].

De-a lungul istoriei calculării tolerante la defecţiuni, au fost dezacorduri importante legate de definiţii şi concepte fundamentale. De exemplu, termeni ca defecţiune, deteriorare şi eroare se foloseau interschimbabil. Unii consideră că o deteriorare s-a petrecut atunci când calculatorul cu care lucrează nu mai răspunde la

comenzile lor. Alții susțin că o deteriorare este o defecțiune fizică mai specifică componentelor electronice. Datorită diversității părerilor în privința înțelegerii acestor noțiuni şi pentru că se consideră vitală lămurirea lor autorul consideră binevenită prezentarea următoarelor definiții sintetizate din [SIEV91],[LALA85] şi [JOHN89]:

- Deteriorare sau malfuncționare (failure): Apare atunci când deservirea livrată deviază față de deservirea specificată. Deservirea poate fi privită din diferite niveluri de abstractizare: deservirea livrată de un circuit integrat privită de un alt circuit integrat , sau deservirea livrată de un sistem privită de utilizatorul lui.
- Defecțiune sau defect (fault): Starea eronată a hardware-ului sau a software-ului cauzată de: deteriorări ale componentelor, interferențe fizice ale mediului înconjurător, erori de interacțiune ale operatorului uman, sau proiectare incorectă.
- Eroare (error): Manifestarea unei defecțiuni într-un program sau structură de date. Eroarea poate să apară la o anumită îndepărtare de la locul defecțiunii.
- Permanent: Descrie o deteriorare, o defecțiune sau o eroare care este continuă şi stabilă. În ceea ce priveşte hardware-ul, o deteriorare permanentă reflectă o schimbare fizică ireversibilă.
- Intermitent (intermittent): Descrie o defecțiune sau o eroare repetitivă care este prezentă numai ocazional din cauza hardware-ului instabil sau din cauză că variază starea hardware-ului sau software-ului (de exemplu, funcție de încărcare sau de activitate). Defecțiuni intermitente cauzate de îmbătrânirea componentelor pot deveni permanente în timp.
- Tranzitoriu (transient): Descrie o defecțiune sau o eroare nerepetitivă care rezultă din condiții temporare. Defecțiunile tranzitorii sunt cauzate de obicei de radiația particulelor α, sau de fluctuația tensiunii de alimentare, şi sunt nereparabile pentru că nu există defect fizic în hardware. Se pot evita astfel de situații prin proiectarea mai atentă şi mai riguroasă atât a hardware-ului cât şi software-ului, şi bineînțeles, prin respectarea condițiilor de funcționare prescrise de fabricantul sistemului.

Notă: În literatura de specialitate [VLAD89], [SIEV91] se utilizează cuvintele hard şi respectiv soft interschimbabile cu cuvântul permanent şi respectiv intermitent sau tranzitoriu. Aceste denumiri nu au altă legătură decât eventual etimologică cu componentele constituante, hardware şi software, ale unui sistem de calcul, ci sunt folosite în sens de dur, ferm şi respectiv moale sau mai puțin ferm.

Deteriorările de sistem sunt clasificate din punct de vedere funcțional în defecțiuni hardware, erori software şi erori de procedură. Există relații strânse între cele trei tipuri de deteriorări de sistem şi categoriile surselor de defecțiuni. Defecțiunile fizice apar numai în hardware. Erorile de proiectare, chiar dacă apar în hardware, majoritatea produc deteriorări de sistem de tip software. Erorile de interacțiune sunt de obicei cauzate de greşelile făcute de operatorul sistemului. Procentajele de deteriorare ale unui sistem sunt atribuite fiecărei categorii de erori, dependent de configurația hardware a sistemului, de structura redundanței, de complexitatea software şi de interfața om-maşină. De exemplu: figura 2.2 arată rezultatele experimentate pe procesorul 1ESS (Electronic Switching System) realizat de AT&T, dedicat telecomunicațiilor. Procentajele din această figură reprezintă fracțiuni ale timpului total în care sistemul rămâne neoperațional, atribuite diferitelor cauze (Staehler and Watters, 1976). Defecțiunile software justifică procentajul de 50% din timpul neoperațional, comparativ cu procentajul de 20% atribuit defecțiunilor hardware şi cu procentajul de 30% cauzat de erorile cauzate de personalul de operare.

Aceste procentaje sunt motivate de faptul că procesorul 1ESS este dublat. Dacă una din unități se defectează, ea se decuplează, rămânând numai dublura ei, păstrându-se astfel continuitatea funcționării sistemului. Între timp, unitatea defectă se repară. Se întâmplă rar să apară o defecțiune hardware în unitatea duplicată, în timpul reparării primei unități, dar atunci când apare, determină deteriorarea întregului sistem. Structura redundantă tolerează o singură defecțiune hardware, astfel, defecțiunile hardware simultane (câte una în fiecare procesor în timpul reparării) determină deteriorarea sistemului şi contribuie la procentajul deteriorării sistemului atribuit defecțiunilor hardware.

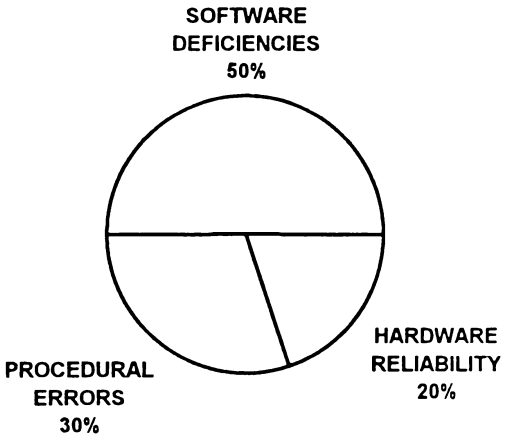


Figura 2.2 Cauzele deteriorărilor de sistem pentru procesorul 1ESS. Bazat pe datele provenite de la Staehler şi Watters (1976).

Erorile de interacțiune justifică 30% din deteriorările sistemului. Această cifră este comparabilă cu cele experimentate în sistemele de calcul comerciale. În sistemele de calcul bazate pe tranzacții, aşa cum este Tandem, raportul dintre defecțiunile hardware şi software, indicat în figura 2.2, este diferit de cel al sistemelor ESS. Acest lucru se datorează raportului diferit dintre hardware şi software pentru două aplicații diferite. În sistemele ESS, cantitatea mare de software (câteva milioane de linii de cod), reactualizarea continuă, şi adăugarea frecventă de facilități noi, contribuie la complexitatea şi volatilitatea componentelor software în sistem. În maşini bazate pe tranzacții, aplicația pretinde o bază de date largă, deci un număr mare de unități de disc. Hardware-ul domină în acest tip de aplicație. Din acest motiv multe din deteriorările sistemului sunt atribuite defecțiunilor hardware. Erorile de interacțiune sunt aproape la fel pentru amândouă sistemele.

În [TOY78] se prezintă acelaşi tip de date statistice tot pentru sistemele ESS. Singura diferență este că procentajul de 50% alocat deficiențelor software este limitat la 15%, restul de 35% fiind alocat deficiențelor de restabilire. Aceste deficiențe pot fi cauzate de defecte nedetectate sau de izolarea incorectă a defectelor.

2.1.1.1. Defecțiuni hardware

Fiabilitatea este văzută adesea ca fiind măsura calitativă a performanței unui dispozitiv sau a unui sistem. Totuşi, fiabilitatea componentelor hardware poate fi măsurată cantitativ, pentru că componentele se defectează cu rate care pot fi apreciate statistic. Definiția fiabilității (ca probabilitatea unei componente de a realiza funcția

cerută sub condiții controlate pentru o anumită perioadă de timp) devine potrivită pentru analizarea defecțiunilor componentei. Analizele statistice presupun în general că hardware-ul testat, este inițial fără defecțiuni şi că se deteriorează cu timpul.

În afară de deteriorările normale ale dispozitivelor, la defecțiunile hardware mai contribuie condițiile mediului înconjurător (radiațiile), erorile de proiectare (greşelile), şi limitările neadecvate (timing problems). Greşelile de proiectare, ca şi în cazul erorilor software, sunt foarte greu de apreciat cantitativ. Calculele de fiabilitate se bazează mai mult pe defecțiunile dispozitivelor.

2.1.1.2. Erori software

Spre deosebire de defecțiunile hardware, erorile software nu sunt cauzate de îmbătrânire. Deteriorările de tip software sunt cauzate de erori de specificații şi de implementarea lor. Există multe cazuri speciale pe care programatorul s-ar putea să le scape din vedere sau să le trateze impropriu.

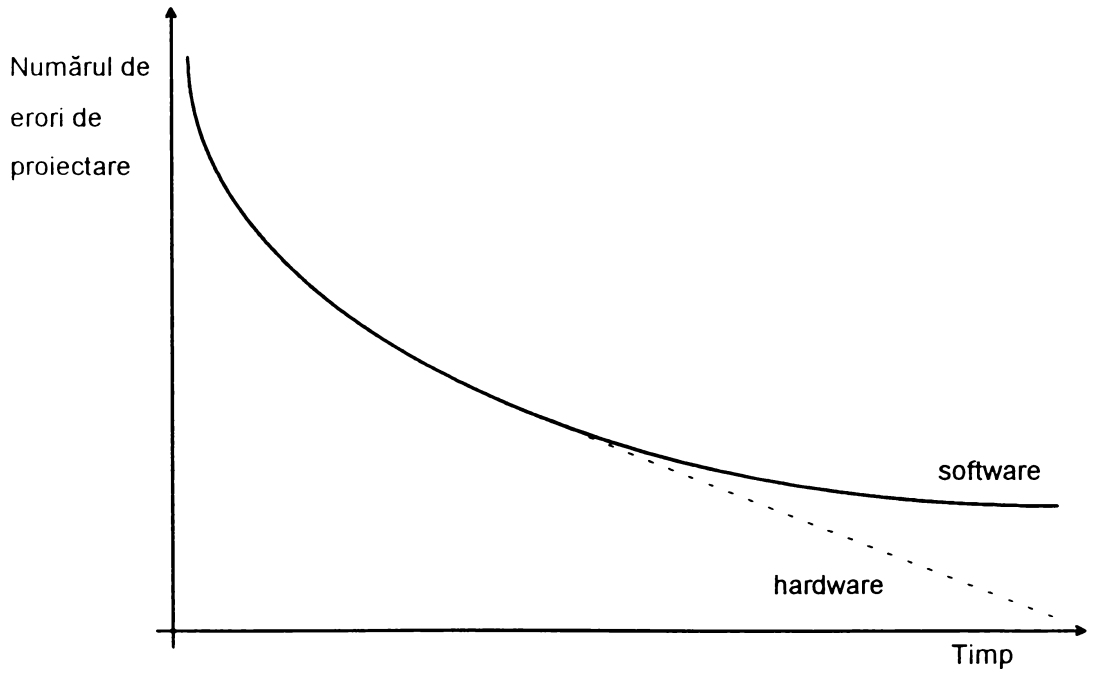


Figura 2.3 Erorile software într-un sistem larg.

Elementele de bază ale software-ului sunt structuri ale căror comportament nu se schimbă cu timpul. Erorile software-ului rezultă din greşelile de proiectare prin combinarea incorectă a instrucțiunilor. Interacțiunile între instrucțiuni sunt mult mai complicate decât interconectările componentelor hardware. O maşină fizică are un număr relativ mic de stări interne distincte, comparativ cu un sistem software. Programatorii, de obicei, presupun că proiectele hardware sunt corecte. Proiectarea software-ului are un număr enorm de mare de stări diferite care trebuie luate în considerare; astfel, chiar după eforturi mari de validare şi corectare, nu se poate garanta corectitudinea unui proiect de sistem cu mult software. Nu există tehnici disponibile pentru măsurarea numărului de erori software dintr-un program, iar fiecare schimbare dintr-un sistem software creează un sistem nou care are proprietăți diferite față de cel inițial. Corectarea unei erori software poate avea efecte secundare în alte

părți ale sistemului care ar putea mări în loc de a micşora numărul de erori. Evident, corectarea unei erori software nu este tot atât de simplă ca şi corectarea unei defecțiuni hardware, prin înlocuirea unei componente defecte cu una bună, dar odată ce erorile software sunt corectate, ele nu se mai repetă aşa cum se întâmplă cu cele cauzate de hardware.

Erorile software nu se pot prevedea, cu excepția celor care există la fiecare sistem. Sunt folosite câteva definiții pentru fiabilitatea software. Una dintre ele afirmă că fiabilitatea software este probabilitatea că un sistem software va realiza funcțiile cerute pentru un număr specificat de cazuri de intrare sub condiții de intrare stabilite. Din moment ce secvența codului executat depinde de valorile parametrilor de intrare, probabilitatea obținerii rezultatului corect depinde de asemenea de datele de intrare. Această definiție este probabilistică, din cauza nesiguranței în selecția parametrilor de intrare atunci când sistemul este în funcționare. De obicei nu este posibil ca testarea să epuizeze toate cazurile de intrare posibile. Astfel unele erori se manifestă numai când apar cazuri de intrare netestate în funcționarea actuală a sistemului.

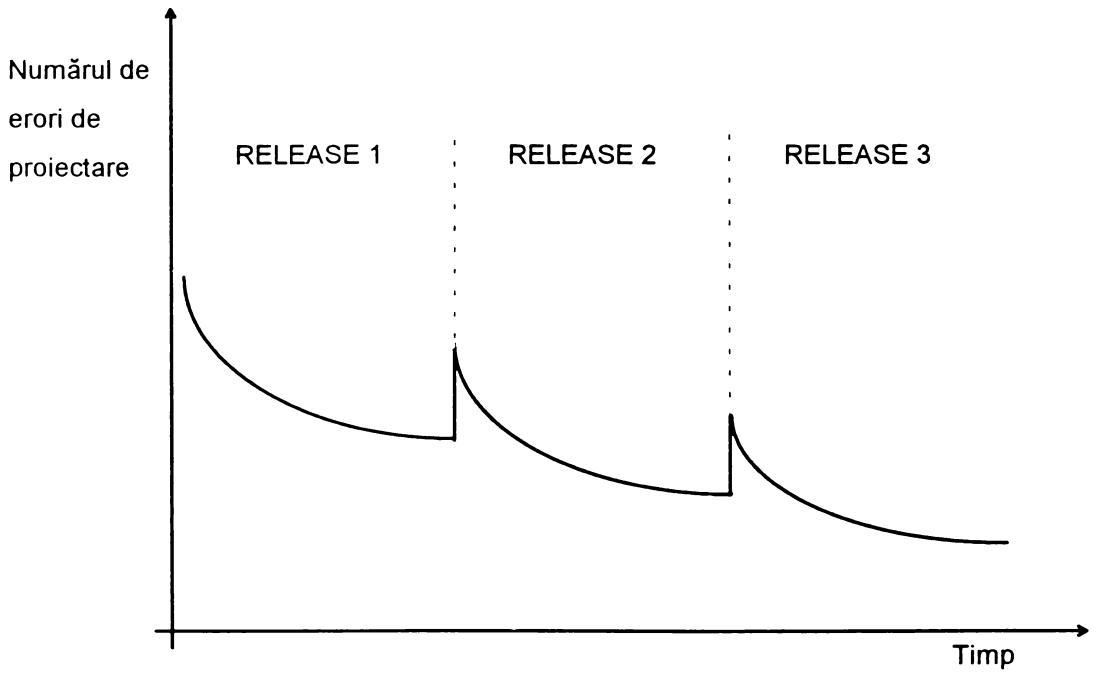


Figura 2.4 Erorile software într-un program revizuit de mai multe ori.

Pentru un sistem cu mult software, testarea totală (exaustivă) este imposibilă. Câteva programe conțin întotdeauna erori rezidente care au supraviețuit proiectării, dezvoltării şi stadiilor de testare. Apariția erorilor software în dezvoltarea unui program urmăreşte o curbă descrescătoare (figura 2.3). Inițial, sistemul conține un număr mare de erori software. În timp ce sistemul este folosit foarte frecvent şi la capacitate maximă, erorile majore sunt detectate şi corectate, reducându-se astfel numărul total de erori aflate în el. Numărul continuă să descrească asimptotic odată cu timpul, spre o limită de un număr pozitiv fix. Ar fi de aşteptat ca numărul erorilor software să descrească monoton spre zero, din moment ce erorile sunt detectate şi îndepărtate continuu din program. Aceasta este situația pentru erorile de proiectare hardware, aşa cum apare cu linie punctată în figura 2.3, pentru că proiectarea hardware devine stabilă cu timpul.

Software-ul se schimbă atât de uşor încât rar devine stabil. Fiecare schimbare software introduce erori noi, astfel că valoarea limitei inferioare este în funcție de rata cu care se adaugă noi facilități în software-ul respectiv. Procesul de corectare a unei erori detectate poate să introducă neintenționat câteva erori subtile în alte părți ale programului. Un alt motiv pentru existența continuă a erorilor software este acela că o porțiune mare din program nu este testată sau exersată. Erorile din această porțiune de cod rămân latente pentru un timp nedeterminat.

Modificarea numărului de erori ale unui sistem software care a apărut în mai multe versiuni (releases) succesive ar putea urma o curbă similară celei din figura 2.4. Fiecare versiune reprezintă o revizie majoră şi o actualizare a parametrilor programului şi eventual modificarea software-ului pentru a include facilități noi. Cu cât sunt detectate şi corectate mai multe erori software de tip rezidual, înainte ca programul să fie dat spre utilizare, cu atât vârfurile curbei se apropie de axa orizontală şi produsul devine mai fiabil. Acesta este obiectivul procesului de evaluare şi validare.

2.1.1.3. Erori de interacțiune

Deteriorările de sistem mai sunt cauzate de greşelile umane ale personalului de întreținere şi ale operatorilor. Erorile de interacțiune (interaction faults) sunt cauzate de intrările la sistem prin interfețe operator-maşină în timpul funcționării sau întreținerii, care nu sunt corespunzătoare stării curente a sistemului. Aceste erori sunt neprevizibile statistic şi apar cu o rată la fel de neprevizibilă. Ele pot fi cauzate de o documentare neadecvată sau incorectă (de exemplu: manualele utilizatorului), sau pur şi simplu de nerespectarea instrucțiunilor din manualele de operare corespunzătoare. De exemplu, un operator poate decupla din greşeală legătura dintre unitatea centrală de procesare şi unitatea de bază pentru stocarea informației, divizând întregul sistem. Este imposibil să se anticipeze şi să se ia precauții împotriva tuturor erorilor de interacțiune posibile.

Problema erorilor de procedură a rămas în continuare o grijă mare pentru utilizatorii sistemelor de prelucrare a informației. Câteva modalități ajută la evitarea erorilor de interacțiune. Multe companii țin cursuri intensive pentru operatori şi au manuale de întreținere şi de operare complete şi explicite. O altă cale care reduce numărul erorilor pune operatorul să lucreze pe sisteme mai sofisticate şi mai apropiate de utilizator (user-friendly systems); sofisticarea sistemelor include înlocuirea panourilor de control cu console mai inteligente bazate pe microprocesoare. Includerea unor protocoluri bine stabilite în proiectarea interfețelor de comunicare operator-maşină poate cerne intrările operatorului pentru multe tipuri de erori, dar erorile procedurale apar încă în sistem cu rate mari.

Este de aşteptat ca apariția erorilor de interacțiune să aibă o curbă descrescătoare similară cu cea de la erorile software (figura 2.3). Într-o primă fază, există un număr considerabil de erori de interacțiune rezultate de la documente inadecvate şi incorecte şi de la operatorii fără experiență. Deoarece aceste neajunsuri se corectează, erorile de interacțiune descresc asimptotic odată cu timpul. De asemenea se corectează neajunsurile din software-ul care are legătură cu procedurile operatorului pentru a se îmbunătăți robustețea sistemului. Erorile de interacțiune continuă să descrească, dar, pentru că operatorii umani nu sunt perfecți, ele ajung până la un număr fix. Există, de asemenea, o schimbare continuă de operatori, care tinde să perpetueze erorile de interacțiune.

2.1.2. Estimarea fiabilității

Estimarea fiabilității este un proces de apreciere a fiabilității realizabile de un articol (element, subsistem sau sistem), având disponibile datele ratei de defectare, adică, prin estimarea fiabilității se apreciază probabilitatea îndeplinirii obiectivelor articolului respectiv pentru o aplicație specifică. Aceste calcule sunt utile în stadiile de început ale unui proiect. Utilizarea valorilor numerice pure pentru fiabilitatea diferitelor componente vor da în general un mic avantaj în evaluarea proiectării inițiale. Aceste cifre sunt extrem de valoroase pentru selectarea de către proiectant a unui proiect în cazul în care sunt mai multe alternative. Aceasta dă posibilitatea proiectantului să facă o selecție bazată pe un tip particular de redundanță de sistem şi pe fiabilitatea asociată lui.

2.1.2.1. Rata de defectare

Când un articol nu mai lucrează aşa cum era proiectat nu mai poate să îndeplinească funcția cerută. Un articol poate fi orice element, subsistem, sistem sau echipament care poate fi evaluat individual sau testat separat. Defectările bine definite care sunt atât bruşte cât şi complete sunt referite ca şi defectări catastrofale. Acestea sunt neprevăzute şi s-ar putea să nu fie evidențiate în timpul procedurilor normale de test. Defectările care au loc treptat în echipament, dar totuşi echipamentul mai funcționează, sunt clasificate ca defecțiuni degradante şi sunt de obicei parțiale (echipamentul va funcționa corect o parte din timp). Defecțiunile degradante sunt rezultatul îmbătrânirii, care cauzează devierea anumitor caracteristici ale echipamentului în afara limitelor de toleranță specificate. În câteva situații defecțiunile de acest tip determină condiții intermitente sau de limită care sunt extrem de dificil de izolat. Din această cauză, se folosesc tehnici de "stresare" (stressing techniques) care forțează defecțiunile parțiale să devină defecțiuni complete, acționând asupra condițiilor de operare, pentru a identifica componentele slabe înainte ca ele să devină supărătoare pentru sistemul în lucru.

Graficul defectării unui echipament pus în lucru poate fi împărțit în trei perioade diferite de funcționare. La început, uzual, se defectează destul de curând orice părți slabe inerent care sunt rezultatul unei proiectări improprii, a unei fabricații necorespunzătoare sau a unei întrebuințări greşite. Rata timpurie de defectare (the early failure rate) deşi relativ mare, descreşte progresiv şi se nivelează în timp ce componentele slabe sunt înlocuite. Această situație este ilustrată în figura 2.5 şi se numeşte perioada timpurie a vieții unui sistem (early life period). Diagrama din figura 2.5 este referită prin numele de curba "cada de baie" (bathtub curve) sau curba "în şa" şi este împărțită în 3 perioade. Multe defectări timpurii pot fi evidențiate prin "burn-in test" sau inspecție 100%. O astfel de tehnică obişnuieşte să elimine componentele slabe supuse la probe în condiții accelerate. Astfel, sistemele sunt operate pentru o perioadă de timp sub condiții care variază, pentru a se asigura detectarea defectărilor timpurii sau a celor posibile să apară. Această tehnică este cu siguranță o necesitate imperativă pentru echipamentele de bord ale sateliților care sunt nereparabile în timpul misiunilor. Acest tip de inspecție este de asemenea de dorit pentru echipamente reparabile cum sunt amplificatoarele transatlantice sub mare, a căror reparație este o sarcină majoră şi foarte costisitoare. Pentru calculatoarele mici cu cost redus, o tehnică de inspecție 100% a componentelor nu este convenabilă din punct de vedere economic. Oricum, o anumită cantitate de "stressing" a componentelor (ex. variând

tensiunea de alimentare sau crescând frecvența de tact) ar putea identifica componentele unui sistem care funcționează la limită.

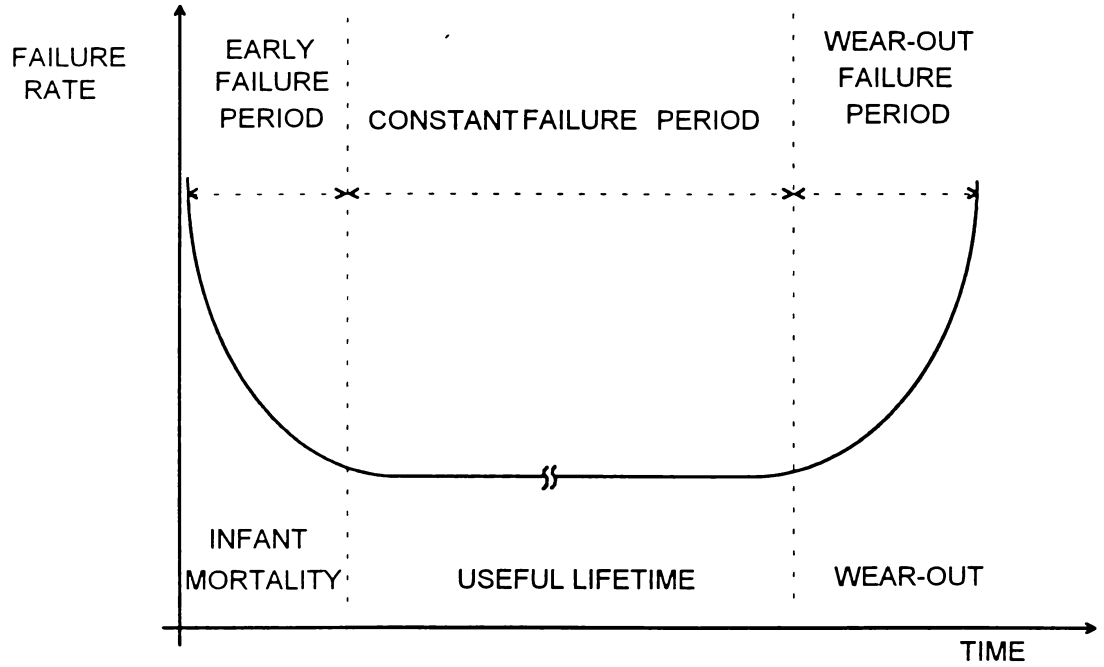


Figura 2.5 Curba "cada de baie" a ratei de deteriorare în funcție de timp.

După îndepărtarea defecțiunilor timpurii, componentele se mențin pe o perioadă lungă la o rată de defectare aproximativ constantă. În timpul acestei perioade, rata de defectare este de obicei scăzută şi este puțin probabil ca defecțiunile să provină de la o singură cauză. Aceasta înseamnă că defectările provin de la o mare varietate de cauze, apar aleator şi cu o rată uniformă fără o regulă evidentă. Viața normală de lucru a unui sistem se află în timpul acestui interval, şi este numită perioada de viață utilă a unui sistem (useful life period of a system).

În perioada de uzură (wear-out period) componentele se deteriorează rapid, odată cu fiecare eventuală uzare. Rata de defectare, aşa cum o indică figura 2.5, creşte din nou. Defectările de uzură (wear-out failures) pot fi evitate prin înlocuirea componentelor înainte ca ele să ajungă în această perioadă.

2.1.2.2. Calcule de fiabilitate cu rata de defectare constantă

Rata de defectare constantă, reprezentată în porțiunea de viață utilă din figura 2.5, scoate în evidență faptul că probabilitatea de defectare este independentă de vârstă. Aceasta înseamnă că un echipament vechi încă aflat în funcționare este tot atât de bun ca şi un echipament nou care a fost recent instalat. Pentru orice rată de defectare constantă, valoarea fiabilității depinde numai de timp. Funcția fiabilității $R(t)$ (reliability) care este caracterizată de o rată constantă de defectare este o distribuție negativă exponențială şi are forma:

$$R(t) = e^{-\lambda t}$$

unde λ = rata de defectare şi t = timp.

Se presupune că atunci când un sistem intră în funcționare (începe misiunea lui la timpul t=0) toate componentele lui sunt operaționale. Conform acestei presupuneri

avem R(0)=1. După un timp infinit toate componentele sunt defecte, R(∞)=0. Importanța distribuției negative exponențiale este aceea că fiabilitatea este independentă de definirea momentului t=0. Dacă un element al echipamentului are o rată de defectare λ, fiabilitatea lui pentru perioada de timp t este $e^{-\lambda t}$. Dacă la sfârşitul acestui timp, elementul este încă în aceeaşi condiție de funcționare, fiabilitatea lui pentru următoarea perioadă de timp, de aceeaşi durată, este tot $e^{-\lambda t}$. În timpul intervalului când rata de defectare a echipamentului este relativ constantă, funcția exponențială negativă este o reprezentare bună a fiabilității echipamentului.

Un sistem fizic se compune, în mod normal, din mai multe tipuri diferite de componente (ex: circuite integrate, conectoare, comutatoare). De obicei, fiecare tip de componente are o rată de defectare instantanee diferită. Un alt mijloc de a caracteriza un sistem fizic este de a considera că fiecare componentă este cuplată în serie cu celelalte componente din sistem. În consecință, când se defectează o singură componentă, se defectează întregul echipament. Funcția fiabilității întregului sistem este reprezentată de produsul funcțiilor de fiabilitate ale tuturor componentelor. În această relație am presupus că fiabilitatea fiecărei componente este independentă de toate celelalte componente. Funcția fiabilității întregului sistem poate fi reprezentată cu expresia:

$$R_T = R_1 R_2 R_3 \cdots R_n$$

Dacă funcția fiabilității pentru fiecare componentă este dată de o distribuție exponențială, relația de mai sus devine:

$$R_T = e^{-\lambda_1 t} e^{-\lambda_2 t} e^{-\lambda_3 t} \cdots e^{-\lambda_n t}$$

Restrângând termenii şi simplificând relația de mai sus obținem următoarea expresie:

$$R_T = e^{-(\lambda_1+\lambda_2+\lambda_3+\cdots+\lambda_n)t}$$

Pentru o conexiune în serie a componentelor a căror funcții de fiabilitate sunt exponențiale, rata de defectare pentru întregul sistem este suma ratelor de defectare individuale ale tuturor componentelor.

2.1.2.3. Timpul mediu dintre defectări

Timpul mediu dintre defectări (Mean Time Between Failures MTBF) este timpul mediu, de obicei exprimat în ore, în care un articol ar putea să funcționeze corect înainte de a se deteriora [LALA85], [TOY88]. Nu există certitudinea că articolul nu se va defecta înainte de sfârşitul acestei perioade sau, că nu va funcționa pentru o perioadă mai lungă. Totuşi, intervalul dintre defectări pentru o componentă a echipamentului este dat de MTBF-ul ei.

În general, MTBF-ul unui sistem ar putea fi tratat ca şi integrala funcției de fiabilitate a întregului sistem:

$$MTBF = \int_0^\infty R(t)dt$$

MTBF-ul poate fi, de asemenea, prezentat intuitiv presupunând că o anumită unitate are rata de defectare de 10^{-6} defectări pe oră. Dacă această unitate se testează şi se înlocuieşte cu una identică de fiecare dată când se defectează, unitatea se va defecta, în medie, o dată la fiecare 10^6 ore. De aceea, MTBF-ul este egal cu 10^6 ore.

Adică, este inversul ratei constante de defectare pentru funcția exponențială, unde $\lambda = 10^{-6}$ defectări pe oră.

MTBF-ul este măsura cantitativă a fiabilității. El dă intervalul mediu de timp în care echipamentul este aşteptat să opereze fără defectare. În câteva cărți de literatură intervalul este, de asemenea, referit ca timpul mediu de defectare (Mean Time To Failure) MTTF. Tehnic vorbind, MTTF şi MTBF nu sunt identici, MTBF-ul poate fi definit, într-o formă alternativă, ca:

$$MTBF = MTTF + MTTR,$$

unde MTTR este timpul mediu de reparație (Mean Time To Repair). Dacă o componentă sau un sistem se repară instantaneu când apare o defecțiune, atunci MTTF şi MTBF au aceeaşi valoare. În general, MTTF-ul poate fi folosit în locul MTBF-ului, întrucât MTTR-ul este foarte mic în comparație cu MTTF-ul.

2.1.2.4. Rata de reparație

Rata de reparație μ (repair rate) sau inversul timpului mediu de reparație ($\mu = 1/MTTR$), este un alt factor care afectează substanțial fiabilitatea şi mentenabilitatea unui sistem. Când o unitate dintr-un sistem duplicat este defectă, sistemul depinde de unitatea a doua ca să continue operația. Dacă unitatea defectă este reparată repede, riscul să se deterioreze întregul sistem devine mic, pentru că unitatea a doua va opera în aşa mod încât să păstreze integritatea funcționării sistemului. Întrucât sistemul este vulnerabil numai până la reparația unității defecte, un timp scurt de reparație poate creşte foarte mult fiabilitatea sistemului.

Timpul de reparație al sistemului ar putea fi divizat în două intervale separate, numite timpul pasiv de reparație (passive repair time) şi timpul activ de reparație (activ repair time). Timpul pasiv de reparație este intervalul de timp măsurat din momentul în care o defecțiune este recunoscută în sistem până în momentul în care personalul de întreținere soseşte şi începe acțiunea de reparație. Acest interval este determinat în întregime de suportul administrativ furnizat de utilizatorul sistemului.

Partea activă a timpului de reparație este timpul efectiv solicitat de personalul de întreținere pentru a izola, a diagnostica, a repara şi a verifica dacă condiția de eroare s-a corectat. Acest timp este direct afectat de proiectarea echipamentului. Timpul activ de reparație poate fi redus prin îmbunătățirea atât a proiectării hardware-ului cât şi a proiectării software-ului, având în vedere minimizarea calificărilor de întreținere necesare ca să susțină sistemul.

2.1.2.5. Sistem reparabil cu redundanță dublă

Structura redundantă dublă este una din mult folositele arhitecturi care furnizează control continuu în timp real (real-time continuous control). Această tehnică a fost utilizată cu succes în ultimii 25 de ani, în sistemele de comutare electronică pentru telecomunicații. Paşi importanți ca să se realizeze înaltă fiabilitate într-un sistem dual sunt: detectarea defecțiunii, restabilirea şi reparația. Amândouă unitățile sunt urmărite continuu, astfel ca defecțiunile din unitatea a doua (backup unit) să fie găsite tot atât de rapid ca şi cele din unitatea principală (on-line unit). Acest lucru este realizat prin rularea celor două unități într-un mod de operare sincron (synchronous match mode of operation). Fiecare operație din cele două unități este

executată pas cu pas, comparându-se rezultatele cheie pentru detectarea erorilor. Dacă există o nepotrivire, fiecare unitate rulează un program de recunoaştere a erorii pentru a se determina care din cele două unități ale sistemului este defectată. Unitatea suspectă este scoasă din funcțiune şi sistemul continuă să funcționeze.

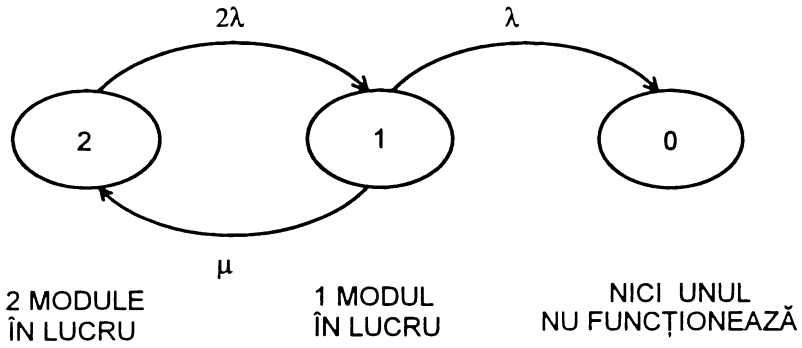


Figura 2.6 Modelul proces Markov; presupune că ratele de deteriorare sunt constante.

Despre un sistem dual redundant se poate spune că este vulnerabil la o deteriorare totală de sistem, numai când unitatea activă originală s-a defectat şi trece prin procesul de reparație. Fiabilitatea este deci, legată nu numai de rata de defectare, dar şi de rata de reparație, sau rata la care defecțiunea este corectată.

MTBF-ul provenit din modelul lui Markov pentru un sistem dual reparabil ca şi în figura 2.6 este dat de $MTBF = (3/2\lambda) + (\mu / 2\lambda^2)$,

unde μ este rata de reparație sau inversul timpului mediu de reparație (MTTR) şi λ este rata de defectare pentru unul din module [TOY88], [SIEV82].

2.1.2.6. Sistem TMR reparabil

Sistemul standard TMR (Triple-Modular Redundant system) redundant triplu modular, operează în mod sincron ca şi sistemul dual, dar arhitectura TMR-ului are capacitatea inerentă de a masca defecțiunile. Detectarea şi corectarea erorii este făcută într-un singur pas fără a perturba funcționarea sistemului. Această proprietate unică este puterea structurii TMR-ului. Restabilirea sistemului după defectarea unuia din module este automată. Erorile tranzitorii sunt corectate automat. Aceste atribute sunt realizate folosind trei module în loc de două.

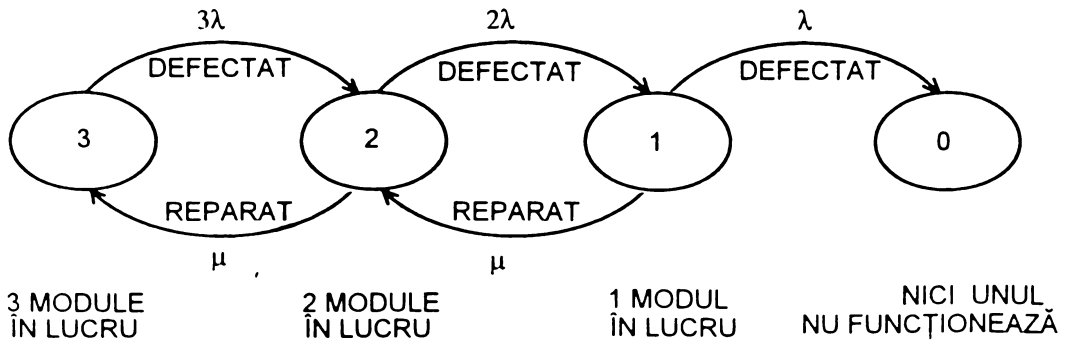


Figura 2.7 Modelul Markov pentru TMR 3-2-1.

Când unul din module se defectează, cele două module rămase continuă să funcționeze ca un sistem dual numai cu capacitatea de detectare a erorii. Dacă un al doilea modul se defectează înaintę ca primul să fie complet reparat, al treilea modul continuă operația sistemului. Sistemul se deteriorează numai în cazul în care al doilea şi al treilea modul se defectează în timpul reparației primului. MTBF-ul este mult mai mare decât cel al sistemului dual. Modelul Markov pentru sistemul TMR este reprezentat în figura 2.7. În starea a treia (starea normală), toate trei modulele sunt fără defecțiuni şi sunt în stare operațională. Atâta timp cât modulele sunt fără defecțiuni, sistemul rămâne în această stare. Din moment ce rata de defectare a unui modul este λ şi sunt trei module în sistem, atunci probabilitatea de trecere din starea a treia în starea a doua este 3λ. Când un modul pică, sistemul intră în starea a doua. După reparație, cu probabilitatea ratei de reparație μ, sistemul revine în starea a treia. Dacă un alt modul se defectează înainte de a se termina reparația primului modul, sistemul trece din starea a doua în starea întâia. Tranziția înapoi în starea inițială se face reparând modulele defecte, unul câte unul. Starea zero numită şi stare capcană (trapping state), apare atunci când se defectează şi ultimul modul, aducând sistemul în stare nefuncțională. De notat că coeficientul lui λ scade cu unu odată cu fiecare tranziție de la o stare superioară la una inferioară. Acest coeficient corespunde cu numărul de module funcționale. Calculul MTBF-ului este dat de timpul care i-ar trebui sistemului să ajungă în starea zero. Dacă timpul de reparație este scurt, posibilitatea de a trece din starea a doua în starea întâia este foarte mică, iar posibilitatea de a trece din starea întâia în starea zero este extrem de mică. Calculul MTBF-ului unui sistem TMR se face cu o procedură similară. Rezultatul este:

$$MTBF = (11/6\lambda) + (2\mu/3\lambda^2) + (\mu^2/6\lambda^3)$$

Notă: În cazul în care sistemul se află în starea unu, cu două module defecte şi doar unul funcțional, probabilitatea de trecere din această stare la starea doi se poate dubla (din μ la 2μ) dacă există posibilitatea reparării a celor două module defecte în paralel apelând de exemplu la un al doilea reparator.

2.1.2.7. Comparații între MTBF-uri

În tabelul 2.1 este prezentată estimarea MTBF-ului pentru configurațiile: singulară, dual şi TMR. λ este rata de defectare pentru un singur modul şi presupunem că are valoarea 10^{-4}, o defecțiune la fiecare 10^4 ore. MTBF-ul pentru un sistem neredundant este $(1/10^{-4})$, 10^4 ore, sau aproximativ un an (un an are $365 \times 24 = 8760$ ore, dar se obişnuieşte în calcule să se considere egal cu 10000 de ore). Pentru un sistem dual MTBF-ul este dominat de termenul al doilea, care variază invers proporțional cu λ^2. Presupunem că rata de reparație μ este 0,125 (timpul de reparație este de 8 ore). MTBF-ul a fost calculat ca fiind 625 ani, ceea ce arată o creştere substanțială a fiabilității în comparație cu sistemul singular neredundant. Această valoare este limita superioară şi oricum scade substanțial pentru o acoperire (coverage) imperfectă, atunci când apare o eroare în sistem. În cazul TMR-ului, MTBF-ul ajunge la valoarea de 260000 ani. Din nou, ca şi în cazul sistemului dual, MTBF-ul se va reduce considerabil din cauza unei acoperiri imperfecte.

Sistem	MTBF	MTBF în ani
simplex	$1/\lambda$	1
dual 2-1	$(3/2\lambda)+(\mu/2\lambda^2)$	625
TMR 3-2	$(5/6\lambda)+(\mu/6\lambda^2)$	208
TMR 3-2-1	$(11/6\lambda)+(2\mu/3\lambda^2)+(\mu^2/6\lambda^3)$	260400

Tabelul 2.1 MTBF-ul configurațiilor simplex, dual şi TMR

2.1.2.8. Efectul acoperirii

Tehnica redundanței dinamice necesită în general doi paşi esențiali: detecția şi corecția. O defecțiune este întâi detectată, şi apoi acțiunea de restabilire (recovery action) reconfigurează o stare funcțională fără defecțiuni (fault-free operational state). Un factor important în redundanța dinamică este conceptul de acoperire (coverage), adică, abilitatea de a reveni cu succes dintr-o defecțiune. În sistemul dual abilitatea de a izola unitatea defectă depinde mult de amândouă suporturile, hardware şi software, ca să diagnosticheze şi să localizeze modulul defect. Incapacitatea programului de restabilire de a configura sistemul în lucru, în jurul unității defecte reduce factorul de acoperire.

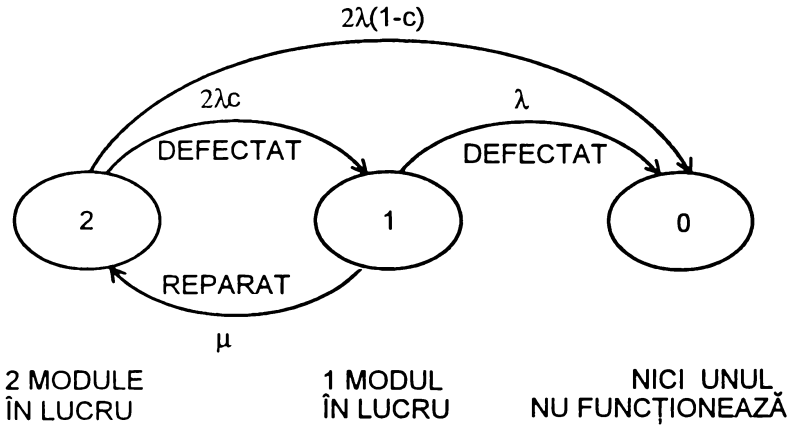


Figura 2.8 Efectul acoperirii imperfecte în cazul unui sistem dual.

Factorul de acoperire poate fi inclus în modelarea fiabilității aşa cum se arată în figura 2.8 pentru o configurație dualã. Se presupune cã sistemul nu se poate restabili după apariția oricărei defecțiuni şi cã c este factorul de acoperire care indicã probabilitatea revenirii sistemului, fiind dat cã o defecțiune a avut loc. Probabilitatea de tranziție de la starea a doua la starea întâia este $2\lambda c$, iar cea de la starea a doua la starea zero este $2\lambda(1-c)$. În cazul unui sistem care se aflã în stare normalã de funcționare, depinde de factorul de acoperire dacă apariția unei defecțiuni duce la deteriorarea întregului sistem. Pornind de la modelul lui Markov, aşa cum se aratã în figura 2.8, rezultă următoarea formulă:

$$MTBF = \frac{\lambda(1+2c)+\mu}{2\lambda[\lambda+\mu(1-c)]}$$

Se observă cã factorul (1-c) de la numitorul fracției are o influență mare asupra valorii MTBF-ului. În tabelul 2.2 se observă cã pentru o valoare a lui c de 0,99

sau 99%, MTBF-ul se reduce de la 625 de ani (pt. c=1) la 50 de ani. Aceasta reprezintă o reducere substanțială a fiabilității. Dacă c=0,95, MTBF-ul este redus în continuare la valoarea de 10 ani. Factorul de acoperire este foarte important la calcularea fiabilității.

Pentru structura TMR-ului, care include şi factorul de acoperire, calculul fiabilității devine complicat, implicând mulți factori. Programele de modelare ale fiabilității au fost vast utilizate ca să faciliteze aceste calcule [BALA93]. Un astfel de program este ARIES (Automated Reliability Interactive Estimation System).

MTBF în ani				
Sistem	**c=1**	**c=0,99**	**c=0,95**	**c=0**
dual 2-1	625	50	10	0,5
TMR 3-2	208	—	—	—
TMR 3-2-1	260400	20833	4167	208

Tabelul 2.2 MTBF-ul configurațiilor cu acoperire dual şi TMR (μ=0,125, $\lambda = 10^{-4}$)

2.1.3. Disponibilitate

Disponibilitatea $A(t)$ (availability) a unui echipament este o funcție de timp, şi poate fi definită ca fiind probabilitatea ca echipamentul să fie operațional la momentul t, atâta timp cât este folosit sub condițiile stabilite. Conceptul de disponibilitate este utilizat în măsurarea eficacității sistemului. Atât fiabilitatea cât şi mentenabilitatea îşi aduc aportul la conceptul de disponibilitate. Această conexiune poate fi exprimată în felul următor [LALA85], [TOY88]:

$$\text{disponibilitatea} = \frac{\text{timpul total}}{\text{timpul total} + \text{timpul total de defectare}} =$$

$$= \frac{\text{timpul total}}{\text{timpul total} + (\text{nr. defecțiunilor} \times \text{MTTR})} =$$

$$= \frac{\text{timpul total}}{\text{timpul total} + (\lambda \times \text{timpul total} \times \text{MTTR})} =$$

$$\left.\begin{aligned} &= \frac{1}{1 + \lambda \times \text{MTTR}} \\ &\text{şi pentru că: } \lambda = \frac{1}{\text{MTTF}} \end{aligned}\right\} \Rightarrow \text{disponibilitatea} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$$

unde funcția fiabilității se presupune că este o distribuție exponențială $R = e^{-\lambda t}$, şi timpul total reprezintă timpul total de funcționare corectă a sistemului.

Pentru că MTTF este o măsură a fiabilității şi MTTR este o măsură a mentenabilității, poate fi aranjată o negociere între ele astfel încât să se obțină o disponibilitate dată. Astfel, dacă MTTF-ul (fiabilitatea) creşte, poate să crească de

asemenea şi MTTR-ul. Aceasta înseamnă că un sistem mai fiabil poate tolera un timp de reparaţie mai mare. O negociere ar putea fi cea în care întreţinerea şi/sau reparaţia se fac mai relaxat bazându-se pe o fiabilitate mai înaltă, menţinându-se aceeaşi disponibilitate.

O altă situaţie apare atunci când MTTR-ul poate fi redus. Astfel va creşte disponibilitatea iar sistemul va deveni mai economic. De multe ori, este preferabil un sistem la care defecţiunile se pot diagnostica rapid, faţă de unul cu rata de defectare mai mică, dar la care ia mult timp localizarea cauzei de malfuncţionare [LALA85].

2.1.4. Dependabilitate

Din moment ce sistemele multiprocesor preiau din ce în ce mai multe şi mai importante sarcini, devine esenţială evaluarea exactă şi rapidă atât a performanţei cât şi a dependabilităţii acestor arhitecturi paralele. Din păcate, în timp ce eforturi considerabile de cercetare au fost îndreptate spre analiza performanţei sistemelor multiprocesor, predicţia dependabilităţii a receptat mai puţină atenţie.

În general, creşterea dependabilităţii unui sistem necesită mutarea resurselor din alte obiective de proiectare, ca de exemplu înalta performanţă. În cazul în care se cere atât putere mare de calcul cât şi înaltă dependabilitate, proiectarea sistemului devine foarte complexă. Totuşi, prin modelare, un proiectant de sistem poate stabili dacă un proiect îndeplineşte cerinţele de dependabilitate, identifică gâtuirile proiectului, şi selectează arhitectura optimă.

În 1982, Laprie şi Coste au definit formal dependabilitatea ca un cadru de specificaţii de nivel înalt care include mărimi ca şi fiabilitatea şi disponibilitatea ca două faţete distincte ale specificaţiilor sistemului. Pe baza caracteristicilor unui sistem, dependabilitatea lui se descrie fie prin fiabilitate fie prin disponibilitate. De exemplu, putem caracteriza cât mai bine un sistem nereparabil prin fiabilitatea lui, iar un sistem reparabil prin disponibilitatea lui.

În literatura mai recentă [JOHN89] se specifică că termenul de dependabilitate include pe lângă conceptele de fiabilitate şi disponibilitate menţionate anterior, şi cele de siguranţă în funcţionare, mentenabilitate, performabilitate, şi testabilitate. Dependabilitatea reprezintă calitatea deservirii furnizate de un anumit sistem. Fiabilitatea, disponibilitatea, siguranţa, mentenabilitatea, performabilitatea, şi în fine testabilitatea sunt mărimi utilizate pentru estimarea cantitativă a dependabilităţii unui sistem.

Mărimile de dependabilitate nu reflectă performanţa unui sistem multiprocesor, ci numai starea lui de operare (operaţională). Noţiunea de stare operaţională utilizată anterior apare datorită faptului că un sistem de procesare paralelă poate oferi servicii utile chiar şi în situaţia în care unele dintre componentele lui nu sunt funcţionale, adică se poate considera operaţional chiar şi cu unele componente deteriorate. Cercetătorii au sugerat patru moduri sau modele operaţionale (terminal dependability, multiterminal dependability, task-based dependability, network dependability [DAS90]) pentru analiza dependabilităţii. Fiecare model defineşte un minim de cerinţe necesare pentru a considera un sistem ca fiind operaţional.

Ţinta proiectării tolerante la defecţiuni este îmbunătăţirea dependabilităţii prin înarmarea unui sistem pentru realizarea funcţiei cerute în prezenţa unui număr dat de defecte. Totuşi, este de notat că un sistem tolerant la defecţiuni nu este neapărat de înaltă dependabilitate, şi nici înalta dependabilitate nu necesită neapărat toleranţa la defectare [NELS90].

2.1.5. Mentenabilitate

Aşa cum s-a precizat mai sus, sistemele de calcul, ca şi orice alt produs de altfel, se pot clasifica în două categorii [GEBE84]: cele cu funcţie unică (simplă), la care prima defectare constituie şi finalul lor de viaţă şi cele cu funcţie repetată sau sisteme cu reînoire (restabilire), la care elementele defecte pot fi înlocuite cu altele noi, bune, caz în care aceste produse au caracter reparabil.

Se înţelege prin mentenanţă [GEBE84] ansamblul tuturor acţiunilor tehnico-organizatorice necesare, efectuate în scopul menţinerii sau restabilirii unui produs în starea necesară îndeplinirii funcţiei cerute. Astfel de acţiuni pot fi, fie cu caracter corectiv (depistarea cauzei unei defecţiunii, repararea defectului prin înlocuirea elementelor defecte, verificarea corectitudinii operaţiilor de mentenanţă întreprinse), fie cu caracter preventiv (revizii, reglaje, verificări şi reparaţii planificate executate în vederea evitării unor viitoare defecţiuni inerente). Personalul şi baza materială, necesare acestor acţiuni, se numesc suportul mentenanţei.

În aceste condiţii, este necesar a exprima cantitativ aptitudinea unui produs ca acesta să fie repus în funcţiune în urma unui defect, ceea ce se exprimă, ca şi în cazul fiabilităţii, printr-o probabilitate în funcţie de timp. Mentenabilitatea este, aşa dar, aptitudinea unui produs ca, în condiţii date de utilizare, să fie menţinut sau restabilit în starea de a-şi îndeplini funcţia specificată (cerută), atunci când acţiunile de mentenanţă se efectuează în condiţii precizate şi într-un timp dat, cu procedee şi remedii prescrise. Corespunzător acestei definiţii, legătura dintre aspectul probabilistic şi cel funcţional se exprimă astfel:

$$M(t_r) = \text{Prob}(t_r \le T_r),$$

unde: t_r este timpul de restabilire, T_r este o limită impusă duratei de restabilire, iar $M(t_r)$ este funcţia de mentenabilitate.

Relaţia între mentenabilitatea M şi rata de reparaţie μ sau timpul mediu de reparaţie MTTR este următoarea [LALA85]:

$$M(t_r) = 1 - e^{-(\mu \cdot t_r)} = 1 - e^{-\left(\frac{t_r}{\text{MTTR}}\right)}.$$

La fel ca fiabilitatea, testabilitatea, dependabilitatea etc., mentenabilitatea trebuie planificată şi estimată sau evaluată încă din fazele cele mai timpurii ale concepţiei unui produs.

2.1.6. Cerinţele fiabilităţii aplicate

Cerinţele fiabilităţii variază considerabil de la o aplicaţie la alta. De exemplu, obiectivele fiabilităţii unui oficiu telefonic cu comutare electronică proiectat pentru utilizarea în aplicaţii telefonice sunt: (I) timpul total în care sistemul rămâne neoperativ să nu depăşească trei minute pe an cu un timp mediu de reparaţie (MTTR) de patru ore; şi (II) să nu se piardă sau să nu se trateze incorect în timpul funcţionării sistemului mai mult de 0,01% din apeluri. Funcţionarea satisfăcătoare, în acest caz, nu este 100% fiabilitate; câteva cuplări incomplete sau greşite sunt permise, din moment ce clientul va forma din nou numărul şi va obţine corecţia. Pe de altă parte, o defecţiune într-un echipament critic, cum sunt amplificatoarele subacvatice în sistemul de cabluri

transatlantice, poate face ca un sistem întreg să devină neoperativ. În acest caz, funcționarea satisfăcătoare pretinde ca toate amplificatoarele să funcționeze. Această situație poate fi comparată cu un lanț de beculețe pentru Crăciun conectate în serie; unde, dacă un beculeț este defect nici unul dintre celelalte nu se poate alimenta.

Durata timpului de funcționare pentru un sistem de comutare telefonic este pur şi simplu durata vieții echipamentului care a fost investit în sistem. Sistemul de comutare trebuie să funcționeze continuu, fără întreruperi, până când echipamentul este înlocuit la sfârşitul vieții lui, sau este îndepărtat pentru orice alt motiv. Din moment ce servirea trebuie să fie asigurată 24 de ore pe zi, nu poate să existe timp programat pentru reparații sau întreținere, în care sistemul să fie neoperațional. Obiectivul fiabilității de trei minute de neoperare pe an (2 ore de nefuncționare în 40 de ani [PRAD86]) şi MTTR-ul de patru ore pot fi traduşi într-o disponibilitate de sistem de 99,9995%.

În contrast, aplicații critice pentru controlul avioanelor, ca şi în cazul lui SIFT (Software-Implemented Fault Tolerance) şi FTMP (Fault-Tolerant MultiProcessor) pretind fiabilitate ultra înaltă. Parametrii proiectării cer ca sistemul să treacă prin mai puțin de 10^{-9} deteriorări în timpul unei misiuni de 10 ore cu un MTTR de 10 ore. Factorul de disponibilitate este egal, în această situație, cu

$$\frac{10^{10}}{10^{10}+10}\cdot 100\% = 99,9999999\%.$$

Cerințele fiabilității determină într-un grad mare arhitectura sistemului, tipul şi cantitatea de redundanță precum şi costul sistemului. Redundanța dual se dovedeşte a fi adecvată pentru aplicații telefonice. Pe de altă parte, pentru controlul critic al avioanelor, triplarea cu rezervă este necesară pentru a achita un grad mare de fiabilitate.

2.1.7. Efectul solicitării sistemului

Toate ratele de defectare sau curbele fiabilității pentru hardware, software şi interfețele de interacțiune operator-maşină, sunt mari în perioada timpurie a vieții unui sistem. Aceste defecte descresc şi se stabilizează la un nivel constant atunci când sistemul devine matur. Componentele hardware slabe sunt înlocuite, erorile de proiectare sunt corectate, iar interacțiunea cu operatorul uman este mai cernută. Faza de uzură apare numai la curbele componentelor hardware, unde fiecare componentă eventual se uzează şi este înlocuită.

Fiabilitatea software trebuie să țină cont şi de mediul de operare. Un program mare conține atât de multe căi posibile de rulare încât este imposibilă testarea completă. Multe din erorile lui nu sunt detectate decât atunci când apare o anumită combinație de variabile de intrare. Probabilitatea apariției acestor erori latente creşte odată cu folosirea sistemului. Există o relație distinctă între rata de deteriorare şi folosirea sistemului pentru amândouă tipurile de defecțiuni hardware şi software. Studiul statistic efectuat la Stanford Linear Accelerator Center (SLAC) pe complexul lor de calcul care conține două calculatoare IBM 370/168s şi unul IBM 360/91 într-un model triplex, arată o creştere a ratei de deteriorare la solicitări mai mari. Cele mai scăzute rate de deteriorare sunt în timpul orelor nelucrătoare. Ele cresc rapid la începutul zilei lucrătoare (8:00 A.M.) şi au maxime înainte şi după ora prânzului.

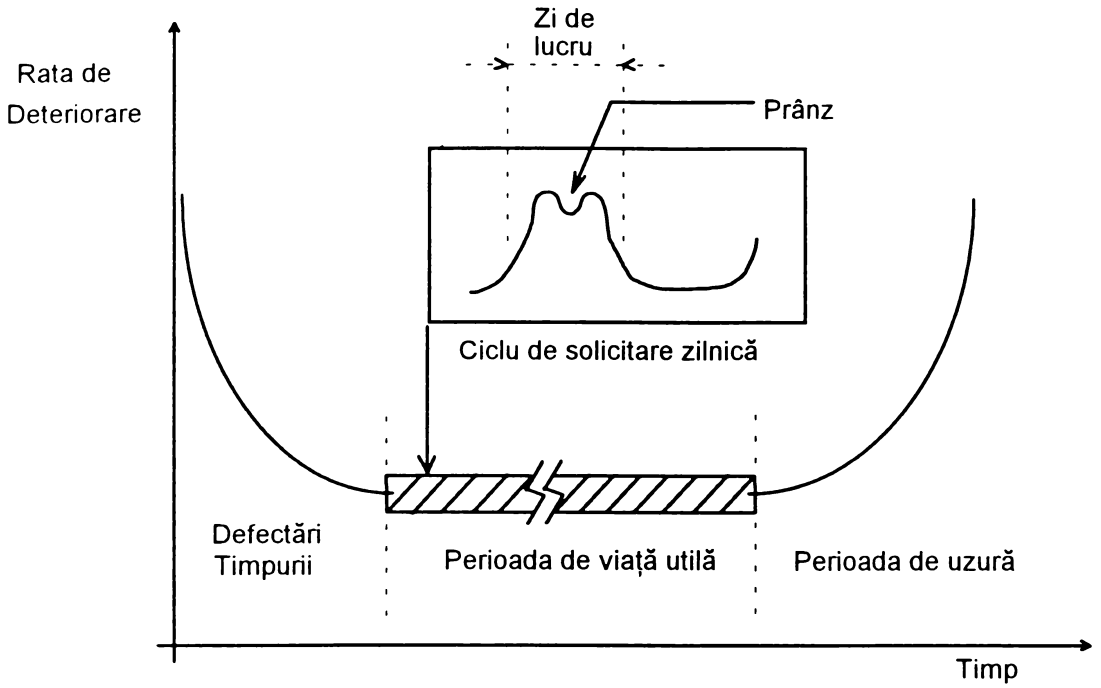


Figura 2.9 Detalierea curbei "cada de baie".

Sfârşitul zilei lucrătoare se face aparent cu o scădere semnificativă a ratelor de defectare. Acest studiu arată relația directă dintre ratele de defectare şi gradul de solicitare aplicat unui anumit modul sau subsistem. Aceste observații arată că rata de deteriorare totală este compusă din două cantități separate. Prima este rata de deteriorare inerentă aşa cum s-a determinat direct din modelele clasice de fiabilitate. A doua este rata de deteriorare indusă, care depinde de utilizarea sistemului şi este periodică, repetându-se zilnic. Curba "bathtub" clasică poate fi îmbunătățită incluzând acest efect periodic (figura 2.9).

2.1.8. Tehnici de redundanță

Dacă un calculator ar fi fost nedefectabil, hardware-ul şi software-ul s-ar fi comportat întotdeauna aşa cum ar fi fost proiectate. Calculatorul perfect însă nu s-a construit până acum, şi de aceea apar erori atât în hardware cât şi în software. Defecțiunile hardware pot afecta secvențele de control sau cuvintele de date care se află în interiorul maşinii. Acest lucru duce la două tipuri de erori.

1. Secvența programului este neschimbată dar defecțiunea a afectat rezultatele finale.
2. Secvența programului este schimbată şi programul nu mai execută algoritmul specificat.

Erorile software sunt rezultatul unor traduceri (translatări) greşite sau implementări improprii ale algoritmilor originali. Cursul execuției instrucțiunilor deviază de la secvența corectă de control. În multe situații, din păcate, defecțiunile hardware şi software nu se pot distinge. Deci, sistemele trebuie să fie tolerante la defecțiuni şi capabile să calculeze corect în prezența lor, indiferent de sursele acestor defecțiuni.

O modalitate bună de a produce calculatoare tolerante la defecțiuni este aceea de a introduce redundanța prin multiplicarea părților lor. Redundanța permite calculatoarelor să ocolească erorile şi în felul acesta rezultatele finale sunt corecte.

Aceasta este cunoscută ca redundanță de protecție şi este compusă din combinații de redundanțe hardware, software şi temporală (de timp). Redundanța hardware este constituită din componente adiționale care detectează şi corectează erorile. Redundanța software conține programe adiționale care restabilesc sub condiții problematice, un sistem funcțional fără erori. Ea poate să mai conțină programe de detecție a erorilor, de diagnosticare şi de autocontrol prin care se testează periodic toate circuitele logice ale calculatorului pentru defecțiuni hardware. Redundanța temporală se realizează printr-o rejudecare (retrial) a unei operații eronate. Ea include repetarea unui program sau a unui segment de program imediat după detectarea unei erori. Rejudecarea este făcută adesea prin hardware. De exemplu: logica hardware poate iniția recitirea automată a unei locații de memorie în care s-a detectat o eroare de paritate.

Chiar dacă redundanța de protecție este clasificată funcțional în trei tipuri diferite, un tip poate include un alt tip sau chiar amândouă celelalte tipuri. În cazul redundanței software, programul de control are nevoie atât de spațiu de memorie (hardware) cât şi de execuție (timp). Fiecare din aceste tipuri de redundanță şi diferitele lor combinații au fost folosite în proiectarea calculatoarelor tolerante la defecțiuni; alegerea accentuării asupra unui anumit tip de redundanță depinde de aplicația utilizatorului (beneficiarului) şi de cerințele asociate fiabilității.

2.1.8.1. Redundanța hardware

Există două tipuri de redundanță hardware, statică şi dinamică. Redundanța statică foloseşte copii masive de componente, circuite şi subsisteme. Corectarea erorilor se face automat. Redundanța dinamică are nevoie de părți adiționale sau subsisteme pe posturi de rezerve. Amândouă tehnicile de redundanță sunt folosite în instalații de calculatoare complexe.

2.1.8.1.1. Redundanța hardware statică

În sistemele în care cel mai frecvent tip de defecțiune întâlnit la o componentă este cauzat de întreruperea circuitului, punerea în paralel a două componente identice introduce redundanța. În calitate de exemplu presupunem că componenta aceasta este un tranzistor de tip npn. Figura 2.10(a) arată conectarea în paralel a tranzistoarelor. Defectarea numai a unuia din tranzistoare (printr-o defecțiune echivalentă cu întreruperea joncțiunii emitor-colector) nu influențează circuitul. Dacă cele mai multe defectări apar din cauza scurtcircuitelor (printr-o defecțiune echivalentă cu scurtcircuitarea joncțiunii emitor-colector), o configurație serie este necesară pentru mascarea unei astfel de defecțiuni (figura 2.10(b)).

Prin combinarea celor două tipuri de conexiuni prezentate mai sus putem obține conexiunile serie-paralel sau paralel-serie. Amândouă pot masca o întrerupere şi/sau un scurtcircuit. În plus, conexiunea serie-paralel (figura 2.10(c)) poate masca chiar două scurtcircuite dacă apar paralel, iar conexiunea paralel-serie (figura 2.10(d)) poate masca chiar două întreruperi dacă apar în serie. Deci, dacă probabilitatea de defectare este comparabilă pentru ambele tipuri de defecțiuni, şi se doreşte sporirea capacității de mascare, conexiunea serie-paralel este de preferat atunci când scurtcircuitele sunt mai probabile, iar conexiunea paralel-serie este de preferat atunci când întreruperile sunt totuşi mai probabile

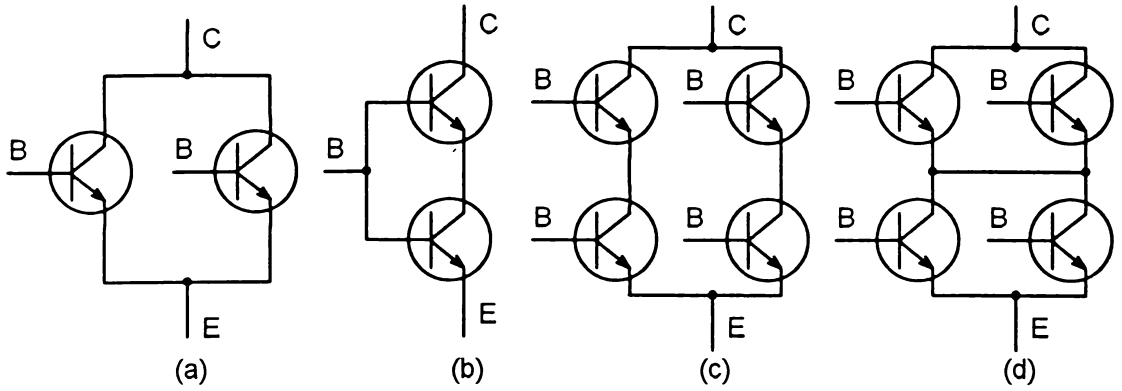


Figura 2.10 Conexiuni folosite pentru redundanța statică:
(a) conexiune redundantă paralel; (b) conexiune redundantă serie;
(c) conexiune redundantă serie-paralel; (d) conexiune redundantă paralel-serie.

Detectarea defecțiunilor este extrem de dificilă când redundanța statică este aplicată la nivel de componentă, pentru că componenta defectă este mascată de hardware-ul redundant. Dacă o defecțiune nu este pasibilă la mascare şi cauzează o eroare, aceasta va continua să rămână nedetectată şi nu va fi corectată. Pentru un proiect de sistem reparabil această metodă de corectare a erorii este nedorită din punctul de vedere al izolării defecțiunii; de exemplu: o defecțiune poate că nu este necesar să fie identificată la nivelul sistemului şi nici izolată într-o unitate specifică.

Conceptul original al lui von Neumann de logică suplimentară la nivel de circuit şi de subsistem a fost studiat pe larg de mulți specialişti în fiabilitate, şi a fost extins în mod special pentru aplicații militare. Această tehnică implică triplarea blocurilor funcționale şi folosirea circuitelor de votare (figura 2.11). Circuitele de votare mențin ieşirea potrivită atunci când o defecțiune este prezentă în unul din blocurile funcționale. Logica suplimentară este aplicată şi la circuitele de votare ca precauție împotriva defectării lor.

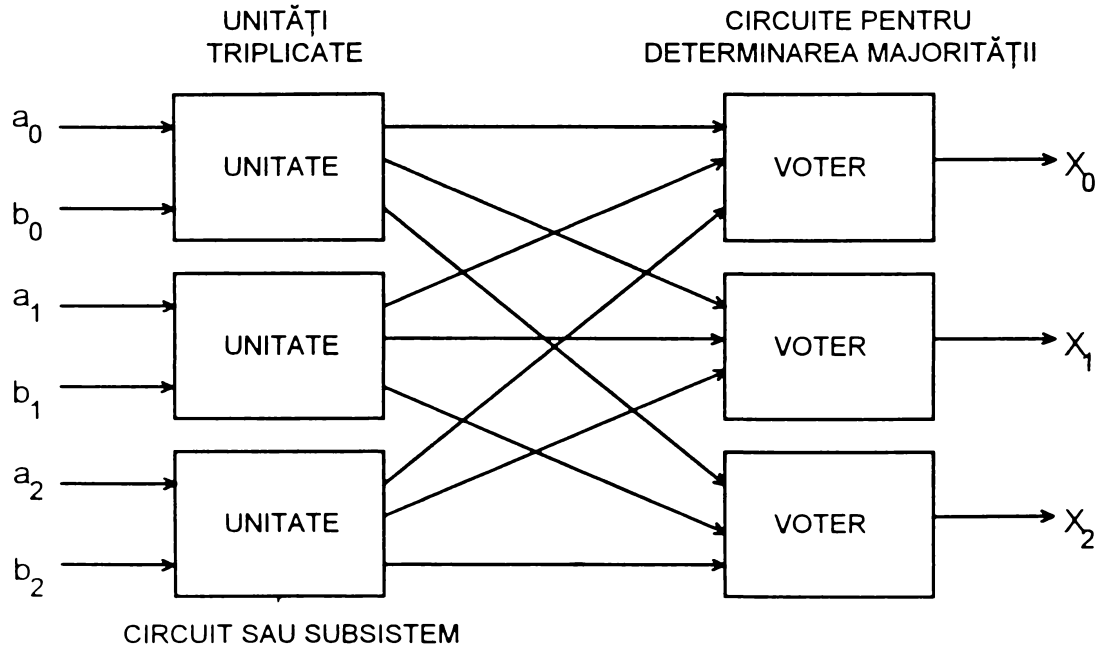


Figura 2.11 Redundanța triplă cu determinarea majorității.

Un alt tip de redundanță statică sunt codurile corectoare de erorii. Tehnicile bazate pe astfel de coduri utilizează hardware adițional şi informații (date) suplimentare, de aceea i-se mai spune şi redundanță informațională. Codurile de corecție Hamming sunt foarte des utilizate în acest sens. Figura 2.12 arată funcțional hardware-ul redundant şi informațiile redundante pentru corectarea erorii într-un sistem de memorie.

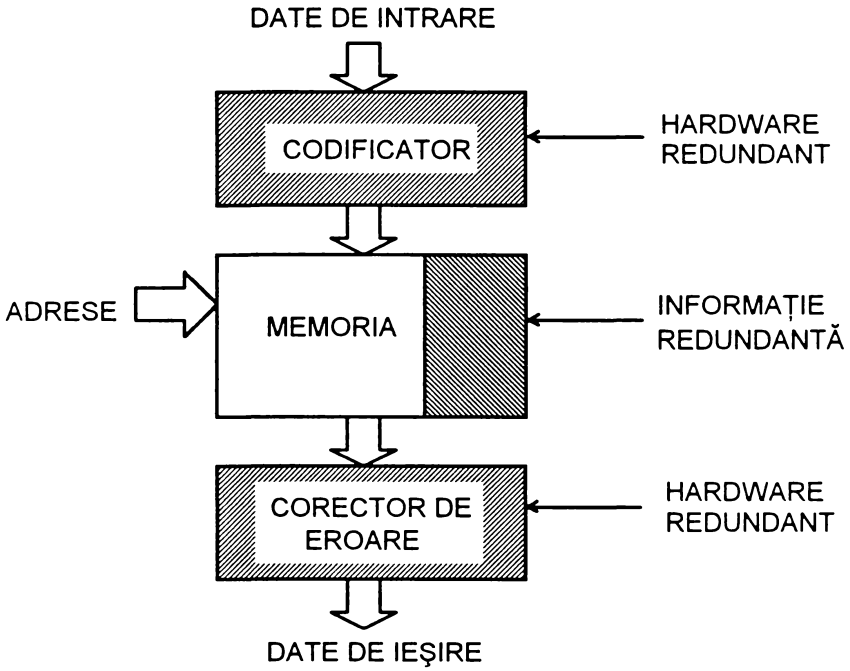


Figura 2.12 Redundanța într-o schemă de corectare a erorii.

2.1.8.1.2. Redundanța hardware dinamică

Redundanța dinamică, referită şi sub numele de redundanță selectivă, pretinde alegeri judicioase pentru a se obține o protecție cât mai eficace împotriva defecțiunilor. Ca răspuns la o eroare, unitatea defectă este înlocuită automat (sau manual) cu una bună pentru a se rezolva problema ivită. Unitatea selectată înainte de intrarea ei în funcțiune, putea fi activă (alimentată) sau pasivă (nealimentată). Există trei paşi necesari în procedura de redundanță dinamică: detecția erorii, diagnosticarea şi revenirea din eroare. Principalul pas în procedură este detectarea rapidă a erorii. Dacă logica de detectare a erorii indică o eroare într-o singură unitate înlocuibilă, atunci al doilea pas (diagnosticarea) nu este necesar, dar dacă logica de detectare a erorilor include un număr de unități înlocuibile, diagnosticarea trebuie inițiată pentru a localiza unitatea defectă. În timpul diagnosticării, eroarea este analizată de un alt circuit hardware specializat sau de software-ul de diagnosticare; rezultatul analizei atribuie defecțiunea la un anumit dispozitiv sau la o anumită unitate. Al treilea pas şi ultimul este acțiunea de revenire prin eliminarea erorii înlocuind unitatea defectă cu una operațională. În plus, pentru un sistem care lucrează în timp real, dacă eroarea apare în mijlocul unei operații atunci este necesară reluarea programului de la un anumit punct (rollback) pentru a se elimina datele eronate şi a se recupera cât mai multe date posibile.

2.1.8.2. Redundanța software

Redundanța software protejează sistemul de erorile hardware şi software prin programe adiționale sau instrucțiuni atât la nivel macro cât şi la nivel micro. Avižienis consideră că chiar şi în cazul în care partea hardware a unui calculator ar fi fără defecțiuni, tot ar fi necesar un anumit grad de redundanță software ca să se asigure fiabilitatea.

Ca şi la redundanța hardware, la fel şi la cea software se întâlnesc două tipuri de redundanțe. Redundanța software statică presupune copierea extinsă. Programele copii sunt scrise şi executate concurent pe suporturi hardware separate. Detectarea erorii se realizează prin comparare. Când există mai mult de două programe, se ia decizie pe baza majorității cu ajutorul software-ului realizându-se un mediu de detecție şi de corecție imediată a erorii. Acesta este echivalentul software al sistemului cu redundanță triplă-modulară (TMR).

Redundanța software dinamică este frecvent utilizată în combinație cu redundanța hardware dinamică. Când sistemul detectează o defecțiune, operația de revenire din defecțiune este solicitată să configureze sistemul în jurul componentei defecte. În timpul funcționării normale a sistemului sunt făcute din când în când copii software ale stării fiecărei componente a sistemului. Ca să se corecteze stările eronate ale maşinii cauzate de o defecțiune, sistemul se întoarce înapoi la ultima copie software a stărilor.

O operație de întoarcere înapoi (rollback operation) face uz de conceptul checkpoint (punctelor de control). Un checkpoint este un punct de întâlnire preprogramată (scheduled) în secvența de execuție unde sistemul salvează stările. Rollback-ul forțează ca execuția să reînceapă din ultimul checkpoint şi după aceea începe prelucrarea datelor salvate; se presupune că datele au rămas neafectate. Deci, revenirea din defecțiune trebuie să implice atât hardware-ul cât şi software-ul ca să se garanteze continuitatea funcționării sau cel puțin ca să se minimizeze perturbarea sistemului.

Un exemplu de redundanță software statică este programarea în N-versiuni (N-version programming), [AVIZ85]. Diferite versiuni ale unui program sunt scrise şi rulate concurent pe suporturi hardware diferite. Înainte de a avea loc orice acțiune, ieşirile lor sunt comparate. Dacă una din versiunile programului nu este de acord cu celelalte, sunt folosite rezultatele majorității, aşa cum se face şi la structura redundantă triplă-modulară. Dacă o versiune a programului este eronată la un moment dat, nu este necesar să o destituim, din moment ce este probabil că ea s-ar putea să fie mai târziu de acord cu alte porțiuni din versiunile programului. Singura revenire din defecțiune care ar putea fi necesar să se facă este ca datele utilizate în programul eronat să se pună în concordanță cu cele folosite în celelalte versiuni. Acest lucru poate fi făcut prin copierea datelor folosite de la una din celelalte versiuni. În cazul în care defecțiunea este cauzată de hardware, este folosit rezultatul corect al majorității. Programarea în N-versiuni asigură protecția împotriva atât a defecțiunilor hardware cât şi a celor software, dar efortul mare de a scrie N versiuni şi cantitatea puterii de prelucrare necesară pentru rularea celor N versiuni sunt aproape de N ori mai mari decât efortul şi puterea de prelucrare necesare pentru un program simplex.

Decât consumarea unui efort de N ori mai mare în dezvoltarea programului, mai bine se foloseşte un mod mai eficace care să dezvolte un singur program folosind modele înalt structurate şi modele formale care să dovedească corectitudinea software-ului. Prin dezvoltarea tehnicilor formale care asigură că programele lucrează corect s-au evitat problemele dificile cum este scrierea versiunilor multiple ale unui

program, testarea lor, administrarea şi întreţinerea pachetului de programe, precum şi reactualizarea modificărilor. Această cale este urmată de sistemul Software-Implemented Fault Tolerance (SIFT) proiectat de Institutul Stanford Research pentru sistemele de control ale avioanelor în timp real. Aşa cum se vede după nume, fiabilitatea se bazează în principal pe mecanisme software. Sistemul software este un sistem dovedit matematic a fi corect şi rulează independent pe un număr de elemente de calcul. Ieşirea corectă este aleasă prin votarea implementată software, spre deosebire de voterii hardware de la structurile TMR.

2.1.8.3. Redundanţa temporală

Redundanţa temporală sau rejudecarea (retrial), este o altă formă de redundanţă. Rejudecarea este folosită pentru corectarea erorilor cauzate de defecţiuni tranziente prin repetarea programelor sau a porţiunilor de programe. Arhitectura sistemului trebuie să ia anumite decizii despre rejudecare:

1. de unde trebuie să înceapă rejudecarea ca să se asigure corectarea erorii;
2. probabilitatea (şi necesitatea) corectării erorii cauzate de defecţiune, prin acţiunea de rollback;
3. ce raport cost/beneficiu este pentru rollback, în termeni de timp real, întrebuinţare hardware, constrângeri software;
4. când este permis rollback-ul de către sistemul de operare
5. consecinţele rollback-ului în lumea reală.

Maşina trebuie să fie restabilită la starea punctului de rollback; toate acţiunile şi schimbările de date făcute după punctul de rollback ca rezultat al execuţiei programului trebuie refăcute. Evenimente singulare în sistemele de timp real care reprezintă comenzi de ieşire pentru iniţierea acţiunilor ireversibile nu pot fi repetate; de exemplu: o operaţie de I/O care controlează o parte mecanică a unui proces de perforare sau de tăiere, nu poate fi făcută de două ori. Dacă un astfel de eveniment este repetat din cauza rollback-ului, atunci se pot provoca consecinţe serioase în lumea reală. În procedura rollback trebuie să fie incorporate măsuri pentru mânuirea evenimentelor singulare.

Folosirea redundanţei temporale în transfer de date sau în comunicaţie de I/O este mult mai uşoară şi mult mai eficace în corectarea erorilor tranziente decât folosirea redundanţelor hardware sau software. De exemplu: dacă datele recepţionate conţin o eroare, circuitul de detecţie a erorilor poate iniţia o reîncercare. Sursa sau emiţătorul retransmite aceleaşi date. Dacă nu este detectată nici o eroare în rejudecare, atunci sistemul procedează ca şi cum nimic nu s-ar fi întâmplat. Evenimentul este înregistrat pentru analize ulterioare. Acest lucru este complet transparent pentru software.

2.2. Metode de autocontrol implementate hardware

Atât structură redundanţei dinamice cât şi cea a redundanţei statice descrise mai sus sunt metode care folosesc părţi de rezervă pentru a face sistemul capabil să tolereze defecţiunile. Atunci când se foloseşte redundanţa statică, unităţile de rezervă (circuite, componente sau subsisteme) sunt părţi permanent active ale sistemului. Ele

corectează erorile sau le maschează împiedicând astfel propagarea lor în sistem. Funcția mascării are loc automat; acțiunea de corectare este imediată şi incorporată (wired in). Tipuri statice de redundanță au fost folosite mai întâi în aplicațiile militare care pretindeau înaltă fiabilitate pentru o durată scurtă, iar mai recent în aplicațiile comerciale.

Redundanța dinamică, la care subsistemele adiționale servesc ca rezerve în interiorul sistemului, s-a folosit în aplicațiile comerciale. Componentele majore ale tolerării defecțiunilor sunt detectarea erorii, diagnosticarea erorii (izolarea), restabilirea şi repararea. Iar în cazul redundanței dinamice cea mai importantă componentă este detectarea erorii (figura 2.13). Dacă toate erorile s-ar fi detectat şi s-ar fi aplicat tehnici de restabilire corespunzătoare, atunci nici o defecțiune n-ar fi condus sistemul la deteriorare (malfuncționare). Acest tip de acoperire nu se poate realiza în practică.

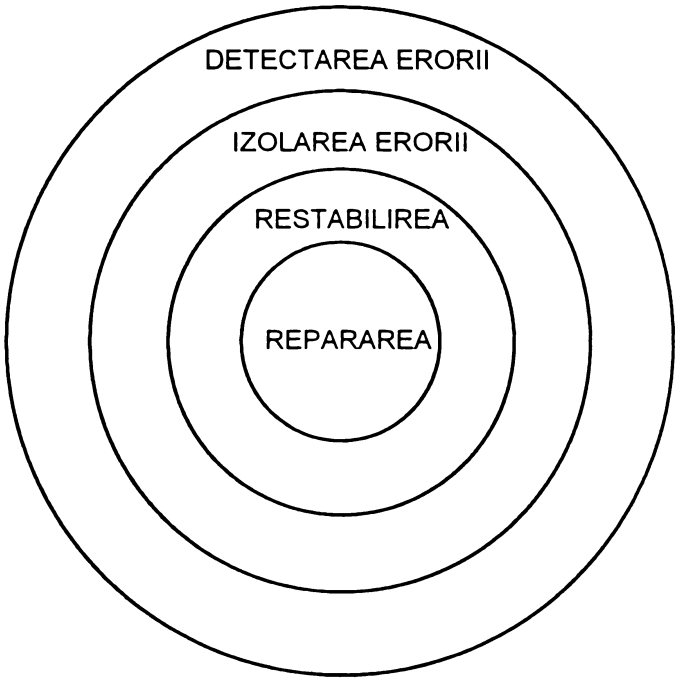


Figura 2.13 Componentele toleranței la defecțiuni.

Viteza detectării erorii uşurează procesul de localizare a defecțiunii şi stăpânirea ei. De asemenea, este important să se realizeze cât mai repede posibil următorul pas, mânuirea sau izolarea defecțiunii, astfel încât defecțiunea să nu se propage şi să rămână izolată într-o anumită unitate. Detectarea întârziată poate deforma date importante în tot sistemul. Viteza detectării este de asemenea importantă pentru localizarea sursei unei erori: cu cât întârzierea este mai mare cu atât este mai greu să se găsească sursa. Diagnosticarea incorectă împiedică rezervele funcționale de la înlocuirea corectă a unităților defecte. Mai mult decât atât, viteza de detectare afectează direct revenirea sistemului.

În general, detecția erorii este realizată prin utilizarea hardware-ului, firmware-ului şi software-ului. ,Tipul schemelor de control (checking circuitry) utilizate depinde atât de structura logică a maşinii cât şi de utilizarea operațională şi funcțională a datelor şi a semnalelor de control.

Schemele hardware de detectare a erorii incorporate într-un sistem de calcul pot avea mai multe forme. Majoritatea acestor tehnici se încadrează în clasificarea următoare:

- controale prin copii (replication checks)
- controale de codificare (coding checks)
- controale de temporizare (timing checks)
- controale de excepții (exception checks)

Detectarea erorii poate fi amplasată strategic în interiorul unei unități funcționale sau la interfață (figura 2.14). Este mai avantajos dacă detecția erorii se face intern, la stadiul ei timpuriu, în timpul funcționării sistemului care generează rezultatele.

Controalele interne sau timpurii minimizează cantitatea activității sistemului şi tranzițiile eronate cauzate de o defecțiune. Astfel, nu este timp suficient pentru a se propaga defecțiunea în interiorul sistemului, iar acțiunile necesare pentru izolarea ei şi pentru revenire vor fi probabil mai simple. Pe de altă parte, controalele de interfață sau cele de ultimul moment sunt activate înainte de a se transmite orice fel de rezultate de la o unitate funcțională la alta. Acest lucru împiedică propagarea în exterior a erorilor spre o altă unitate funcțională şi simplifică cea mai dificilă problemă, şi anume revenirea globală. Eroarea este conținută în interiorul nivelului în care a fost detectată.

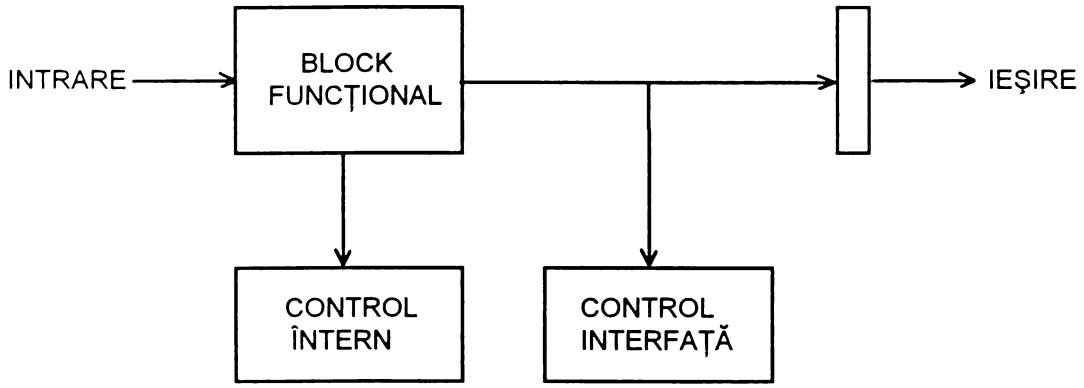


Figura 2.14 Alocarea mecanismului de detecție a erorii.

2.2.1. Controlul prin copii

Controlul prin copii este una din cele mai complete metode de detectare a erorilor dintr-un sistem de calcul. Din cauza hardware-ului necesar, această metodă este de asemenea şi cea mai costisitoare tehnică de redundanță. Cu toate acestea, dezvoltarea rapidă a tehnologiei VLSI şi a tehnologiei microprocesoarelor a început să facă acest tip de redundanță convenabil din punct de vedere al costului, pentru multe aplicații care necesită înaltă fiabilitate.

Controalele prin copii detectează defecțiunile hardware. Ele sunt bazate pe afirmația că proiectarea sistemului este corectă şi că deteriorările apar independent. Un sistem pentru controlul prin copii are o copie identică a circuitului (sau a subsistemului), care prelucrează semnalele de intrare în paralel cu circuitul original. Cele două seturi de ieşiri sunt comparate cu ajutorul unui circuit comparator. Dacă sistemul este proiectat corect şi componentele se defectează independent, atunci erorile nu pot rămâne nedetectate în nici una din versiunile circuitului.

Controalele prin copii sunt făcute la nivel de circuit sau la nivel de subsistem. Alegerea nivelului este influențată într-un grad mare de proiectarea generală a tolerării defecțiunilor; proiectantul trebuie să țină cont de fiabilitate, de cost şi de performanță. Aranjarea unităților aritmetice şi logice (UAL) exemplifică controlarea copiilor la nivel de circuit (figura 2.15). UAL-ul I este duplicat şi ieşirile de la amândouă UAL-urile sunt comparate după fiecare operație UAL. Ieşirile UAL-ului I sunt utilizate ca şi sursă propriu-zisă de rezultate din UAL, pentru a fi transmise mai departe la restul logicii procesorului. Atunci când ieşirile UAL-ului I sunt conduse spre magistrala de ieşire, datele sunt trimise în comparator şi sunt comparate cu ieşirile UAL-ului II. În felul acesta se controlează complet rezultatele operației UAL-ului. Paritatea rezultatului este generată de la ieşirile UAL-ului II.

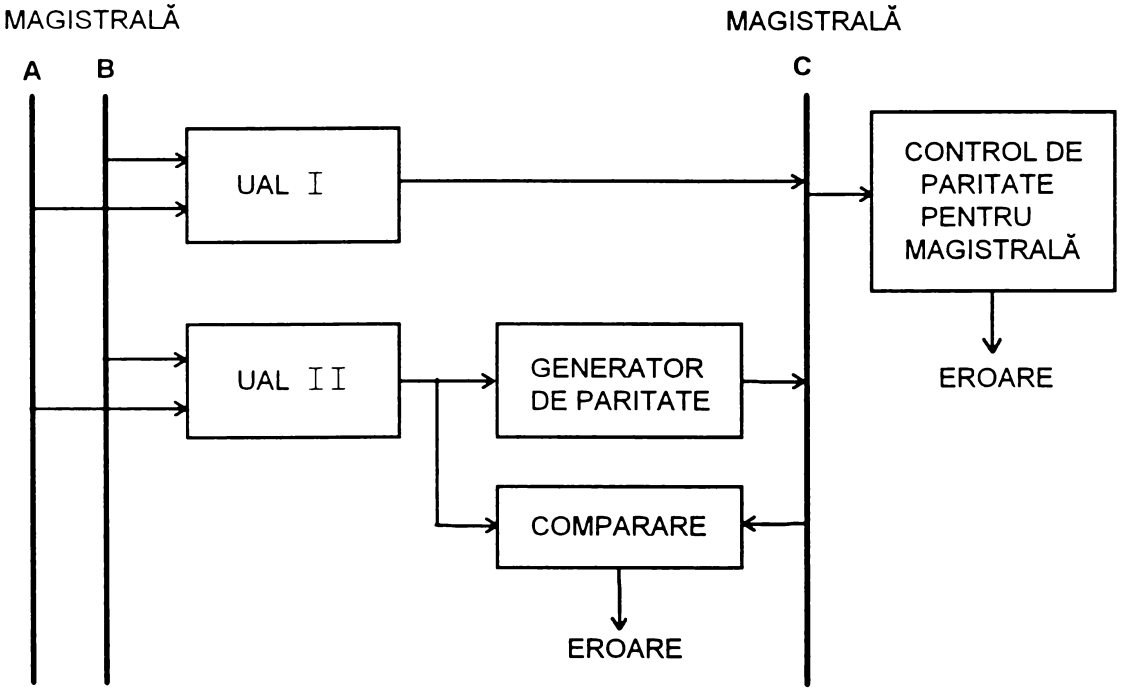


Figura 2.15 Copie la nivel de circuit.

În sistemele AT&T de comutare controalele prin copii sunt făcute la nivel de subsistem. Modalitatea adoptată de proiectanții sistemelor de comutare 1ESS, 1AESS, 2ESS şi 4ESS se bazează pe ideea dublării şi comparării [TOY78]. În această modalitate, amândouă unitățile centrale de procesare prelucrează aceeaşi informație de intrare şi rulează sincron una față de cealaltă. Datele critice de ieşire ale unei maşini sunt comparate cu cele ale celeilalte maşini după completarea fiecărei operații interne de control a maşinii. În realitate, numai un CPU, numit maşina on-line, tratează apelurile de procesare. Al doilea CPU funcționează ca un "standby" alături de CPU-ul on-line putându-se astfel realiza compararea. Echipamentele periferice sunt controlate de CPU-ul on-line. Când ieşirile CPU-urilor nu sunt egale, un program de detecție a defecțiunii este apelat pentru a determina care CPU este defect. Figura 2.16 arată configurația sistemului 2ESS, care are o structură mult mai simplă decât cea a succesorului său 1AESS. În sistemul de comutare 2ESS există doar un singur comparator, acesta este amplasat în centrul de întreținere (maintenance center) şi este neduplicat. Comparatorul compară întotdeauna registrele de intrare ale stocării apelului CS (call store input registers) în CPU-uri atunci când operațiile de stocare ale apelurilor se fac sincron.

O defecțiune aproape în orice parte a oricărui CPU provoacă rapid o nepotrivire în registrele de intrare de stocare ale apelului. Acest lucru se întâmplă pentru că aproape toată tratarea de date realizată atât în controlul programului cât şi în controlul de intrare-ieşire implică returnarea datelor procesate în aceste registre CS. Intrarea stocării apelului este un punct important pe unde trec datele direct spre CS. Prin compararea intrărilor CS, se realizează un control eficace al echipamentelor sistemului.

Spre deosebire cu compararea mai complexă de la CPU-ul 1AESS, detecția erorii în CPU-ul 2ESS nu este tot atât de rapidă din moment ce se compară numai un nod crucial al procesului. În cazul procesorului 1AESS sunt comparate patru noduri ale acestuia. În procesorul 2ESS anumite defecțiuni rămân nedetectate până când erorile se propagă în registrul CS. Acest interval, de obicei, nu este mai mare de câteva zeci sau sute de microsecunde. În timpul unui aşa scurt interval, defecțiunea afectează doar un singur apel telefonic.

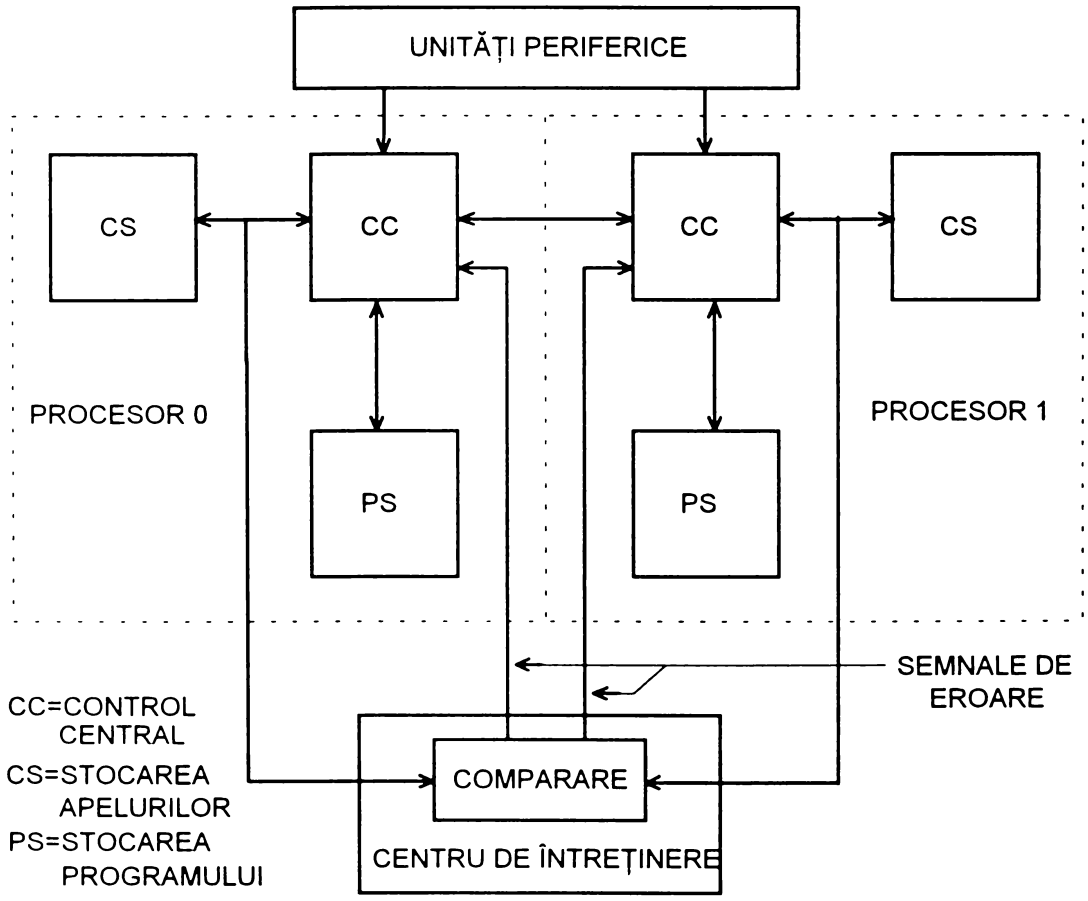


Figura 2.16 Acces pentru comparare la sistemul 2ESS.

Modalitatea de dublare şi comparare este eficientă din punct de vedere economic, din moment ce unitatea de rezervă este necesară numai pentru servirea neîntreruptă.

Numărul copiilor dintr-un sistem nu trebuie să fie limitat la doi. Se pot folosi mai multe copii ale unui sistem, aşa cum este şi în cazul sistemelor de redundanță triplă modulară. Atunci când trei sau mai multe copii ale unui sistem sunt implicate, controlul comparării este referit ca votare (voting). Votarea asigură ca o ieşire eronată să fie suprimată, prin mascare în favoarea majorității ieşirilor. Comenzi mai puternice

de redundanță au posibilitatea de a face corectarea erorii identificând care copie de modul este eronată. Capacitatea de detectare a erorilor este aceeași, atât în cazul în care unitatea este dublată cât și în cazul în care este triplată. Este de notat faptul că controlul de eroare care se bazează numai pe dublare și comparare nu identifică care copie a unității sistemului conține defecțiunea.

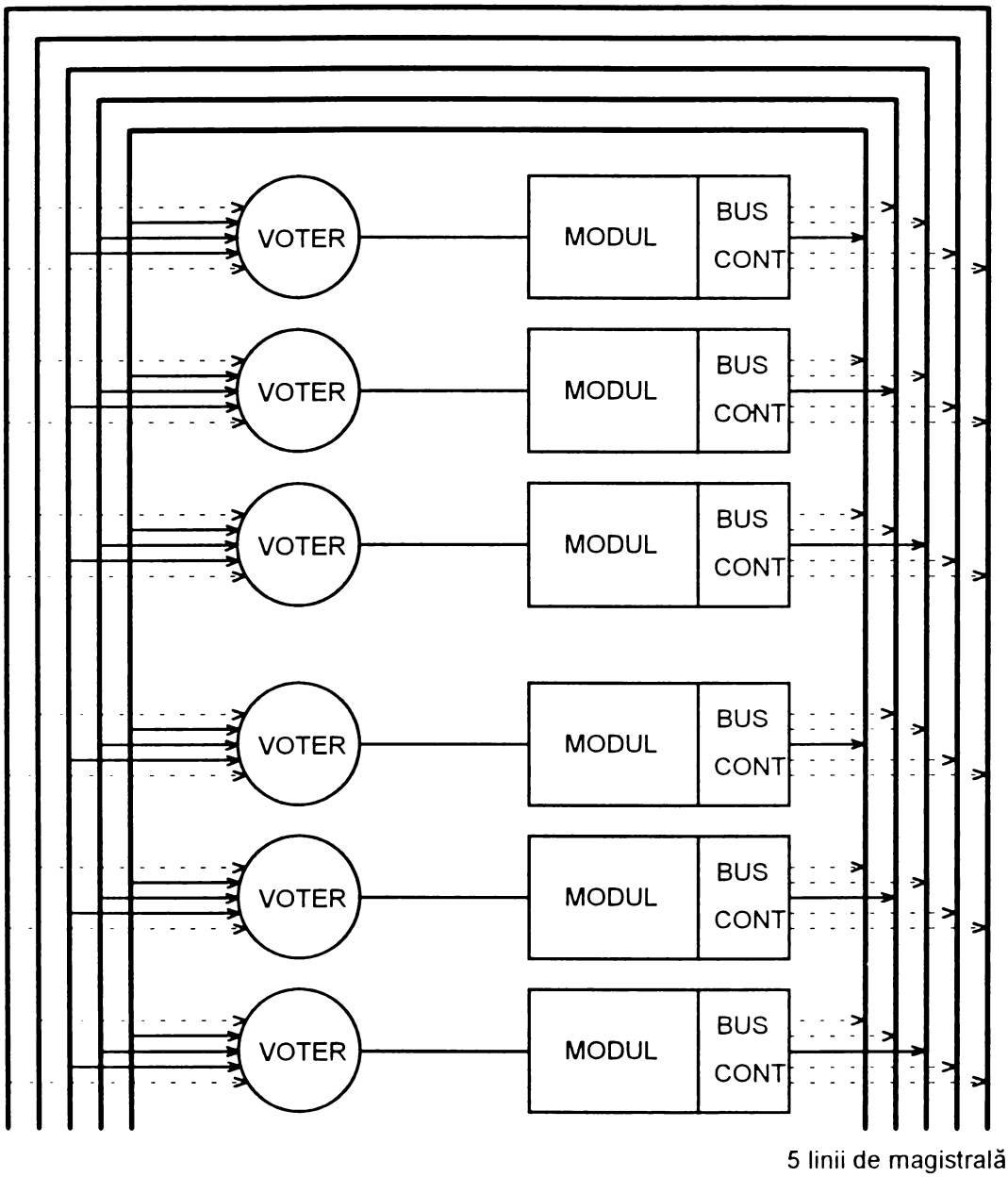


Figura 2.17 Vedere simplificată a triadelor sistemelor FTMP.

Un exemplu de controlare prin copii într-un sistem de redundanță triplă modulară este Fault Tolerant Multiple Processor (FTMP) dezvoltat pentru calculatoarele avioanelor [HOPK78]. FTMP este un sistem multiprocesor care conține un număr mai mare de module de procesare, module de memorie globală, precum și module de intrare-ieșire conectate împreună prin magistrale multiple. Hardware-ul specializat activează trei module de același tip care operează într-o configurație numită triadă (triad). Figura 2.17 reprezintă o schemă mai simplificată a unei triade de module din FTMP care este conectată logic și funcționează ca și unitățile TMR. Toată activitatea este condusă de triadele modulelor și de magistrale. O triadă de module

este formată prin asocierea a oricăror trei module de acelaşi tip. Aceasta înseamnă că orice modul poate fi folosit ca o rezervă în orice triadă. O triadă de linii de magistrală este întotdeauna activă pentru fiecare din magistralele sistemului. Fiecare modul are acces la toate liniile magistralei şi conţine un element de decizie care selectează varianta corectă a liniilor de magistrală. Există cinci linii de magistrală dintre care trei sunt active (desenate cu linie continuă în figura 2.17). Ele sunt conectate la un voter pentru fiecare modul, constituind astfel un element TMR. Celelalte două linii de magistrală (desenate punctat) nu sunt active şi servesc ca rezerve. Cele trei linii de magistrală active transferă trei versiuni de date generate independent, fiecare provenind de la un membru diferit al triadei, care transmite datele pe o linie de magistrală specifică (linie continuă).

Controalele prin copii bazate pe copii identice ale unui sistem nu detectează consecinţele greşelilor de proiectare. Greşelile de proiectare afectează toate copiile unităţii; deci nu sunt detectabile prin multiplicarea unităţilor. Tehnicile formale de verificare a proiectării trebuie să fie făcute astfel încât să se asigure că nu există greşeli de proiectare în structură, care ar putea ocoli protecţia realizată de copiere.

În [MAHM88], Mahmood şi McCluskey prezintă câteva tehnici de detecţie a erorilor bazate pe utilizarea unui procesor "watchdog". Un astfel de procesor, care se poate numi coprocesor de control, este mai puţin complex decât procesorul principal şi detectează erorile prin monitorizarea comportamentului sistemului. Informaţia furnizată către coprocesorul de control poate fi legată de: comportamentul acceselor la memorie, cursul comenzilor, semnalele de comandă sau chiar de justificarea rezultatelor. Asemenea duplicării, pentru detecţia erorilor această tehnică nu depinde de modelul de defecţiuni adoptat, în schimb are avantajul că necesită mai puţin hardware.

2.2.2. Controale de codificare

Codurile detectoare de erori se formează prin adăugarea unor biţi de control la un cuvânt de date. În acest caz, capacitatea de detecţie a erorilor este direct proporţională cu numărul de biţi de control incluşi într-un cuvânt. Există două tipuri de controale de codificare: separabile şi neseparabile (în literatura de specialitate se întâlnesc şi cu denumirile sistematice şi nesistematice). Controalele separabile (separable checks), cum sunt controlul de paritate şi codurile aritmetice, sunt caracterizate prin adăugarea biţilor de control la cuvintele de date. Controalele neseparabile (nonseparable checks), cum este codul m-din-n, sunt codificate în formate specializate.

Controlul de paritate este tipul codului de control cel mai răspândit, pentru că el are nevoie de un număr minim de biţi de control asigurând o capacitate bună de detecţie a erorii. Controlul de paritate este un tip de cod detector de erori separabil, şi este realizat prin alocarea unui bit de paritate la fiecare cuvânt de date. Un control de paritate impar (odd-parity check) are nevoie ca numărul total de biţi aflaţi în starea logică "1" din cuvântul creat în urma adăugării bitului de paritate să fie un număr impar. Un control de paritate par (even-parity check) are nevoie ca numărul total de biţi aflaţi în starea logică "1" din cuvântul creat în urma adăugării bitului de paritate să fie un număr par. De exemplu: cuvintele de date 10110100 şi 10101000 conţin patru biţi pe "1" (număr par) şi respectiv trei biţi pe "1" (număr impar). Pentru un control de paritate impar, bitul de control trebuie să fie poziţionat pe "1" pentru primul cuvânt şi pe "0" pentru al doilea cuvânt; adică 101101001 şi respectiv 101010000. În acest

exemplu bitul de control ocupă poziția celui mai puțin semnificativ bit din fiecare cuvânt de date. Pentru un control de paritate pară bitul de control trebuie să fie poziționat pe "0" pentru primul cuvânt şi respectiv pe "1" pentru al doilea cuvânt. Orice eroare pe un singur bit sau erori multiple impare sunt detectate printr-un singur bit al controlului parității. Erorile cauzate de un număr par de biți eronați nu pot fi detectate prin control de paritate pentru că un număr par de erori este transparent pentru controlul de paritate. Controlul de paritate este utilizat în căi de date (data paths) şi în subsisteme de memorie pentru sisteme de comutare electronică [TOY78]. În subcapitolul 3.2. sunt tratate mai pe larg circuitele de verificare (checkers), de diferite tipuri inclusiv cele de verificare a parității, cu proprietăți speciale cum ar fi autotestarea.

Un alt tip de cod separabil utilizat pentru detectarea erorilor este cel al codurilor aritmetice [IONE81], [NIKO88], [RAO89]. Acesta se bazează pe teoremele restului din aritmetica reziduurilor. Diferența cea mai mare dintre cele două categorii de coduri este că aceste coduri reziduale sunt păstrate sub operații aritmetice. Pentru codurile reziduale operanzii (x, y) şi simbolurile lor de control (x', y') sunt tratate separat; (x, y) generează rezultatul z, în timp ce (x', y') generează rezultatul de control z'. Acest algoritm calculează restul rezultatului z şi îl compară cu rezultatul de control z'. Dacă cele două valori se potrivesc, atunci nici o eroare nu este detectată. O discordanță între cele două valori este considerată ca o eroare fie în unitatea aritmetică de bază fie în circuitul de control. Codurile aritmetice sunt folosite rar în proiectarea sistemelor tolerante la defecțiuni cu circuite LSI şi VLSI, pentru că tendința prezentă este îndreptată spre hardware mai puțin scump şi CPU-uri pe un singur circuit (single chip CPUs). Toy susține că pentru detecția erorilor este mai economic să dublezi şi să compari rezultatele.

Codul cu ponderea fixă (fixed-weight) sau m-din-n este cel mai important tip de cod de control neseparabil. În teoria codurilor ponderea unui cuvânt este definită ca fiind numărul componentelor lui diferite de zero. Deci un cod binar cu ponderea fixă are un număr fix de unități. Unica proprietate a controlului prin codificare (codare) cu pondere fixă este abilitatea lui de a detecta toate erorile multiple unidirecționale. Acest tip de eroare apare atunci când o defecțiune cauzează schimbarea de la starea de "0" logic la cea de "1" logic sau invers, a tuturor biților de date (afectați) din cuvântul codificat. Erorile unidirecționale cauzează modificarea ponderii cuvântului codificat făcând erorile detectabile prin logica controlului. Erorile care cauzează modificarea biților de date în amândouă direcțiile simultan nu sunt întotdeauna detectate prin utilizarea controlului de acest tip.

Codurile neordonate (unordered codes) sunt de asemenea orientate spre detecția erorilor unidirecționale. Un cod se numeşte neordonat dacă şi numai dacă nici un cuvânt al codului nu este inclus în alt cuvânt din acelaşi cod, adică pozițiile de unu dintr-un cuvânt niciodată nu sunt o submulțime a pozițiilor de unu dintr-un alt cuvânt [BOSE91] În anumite situații aceste coduri sunt identice cu codurile cu ponderea fixă. O subclasă a codurilor neordonate sunt codurile echilibrate (balanced codes) unde fiecare cuvânt are acelaşi număr de unu-uri cât şi de zero-uri .

Semnalele binare codificate folosite într-un sistem de calcul reprezintă două tipuri de informație: date şi comenzi. Semnalele de date reprezintă la rândul lor o varietate de entități, ,incluzând numere, caractere, adrese, şi etichete. Aceste entități pot fi utilizate direct, fără modificări, sau pot fi operate cu orice funcții aritmetice şi logice disponibile în procesor, ca să se calculeze valorile specifice necesare execuției curente a programului. Codurile separabile detectoare de erori ca şi paritatea permit datelor originale să fie determinate fără nici o decodificare adițională. Această

caracteristică este foarte convenabilă pentru procesarea şi mânuirea semnalelor de date în interiorul sistemului de calcul. Semnalele binare codificate pentru funcţii de comandă pot folosi similar coduri separabile detectoare de erori, dar aceste semnale sunt uzual decodificate la destinaţia lor (de exemplu: porţile logice care generează fiecare din semnalele de comandă). Cu toate acestea codurile neseparabile detectoare de erori cu caracteristici speciale sunt mai eficace pentru detectarea erorilor pentru semnale de comandă individuale. Codurile m-din-n au fost folosite intens în procesorul 3ESS.

Codul ciclic este un tip de cod separabil care formează un cuvânt codificat din orice deplasare ciclică a codului. Codurile ciclice sunt uşor de codificat şi de decodificat folosind registre de deplasare cu bucle de reacţie (LFSRs), aşa că ele sunt adesea folosite pentru controlul şirurilor de date seriale. Aceste coduri au fost dezvoltate pentru a se realiza detectarea eficientă a erorii pentru blocuri de date. În momentul de faţă, o controlare cu coduri ciclice redundante este folosită la discurile de stocare a informaţiei de la sistemul de comutare 1AESS şi la sistemul de disc 3B2OD.

2.2.3. Controale de temporizare

Procedurile de control hardware cum sunt tehnicile de copiere şi cele de codificare, sunt proiectate să detecteze defecţiunile fizice ale circuitelor în unităţile aritmetice şi logice, în căi de date, în secţiunile de control, şi la unul sau mai multe subsisteme specifice. În general, controalele erorilor prin hardware nu sunt capabile să detecteze erorile software. Chiar şi atunci când două procesoare rulează sincronizate şi sunt comparate pentru detectarea erorii, s-ar putea să apară aceeaşi eroare şi în programul care rulează pe maşina on-line şi în copia programului care rulează pe maşina standby. Astfel, eroarea trece nedetectată în acest caz.

Controlul de temporizare este o formă eficace de a controla software-ul pentru detecţia erorilor în programele duplicate, atunci când specificaţia unei componente include constrângeri de timp. Sistemul de operare normal supraveghează şi coordonează activităţile din sistem. Pentru unele aplicaţii, ca cea pentru controlul unui oficiu de telefoane, sistemul de operare este de natură ciclică; ceea ce înseamnă că programul lui întotdeauna se întoarce în punctul lui de bază de unde a plecat ca să completeze sarcinile programate. Programul principal este, în mod normal, exersat şi depanat amănunţit înainte de integrarea lui într-un sistem. Totuşi foarte rar se procedează ca fiecare parte concepută din program să fie controlată sub toate condiţiile posibile. În consecinţă, unele din cele mai puţin traversate ramificaţii ale programului de bază s-ar putea să conţină erori logice subtile care pot devia rularea secvenţei de execuţie de bază în aşa fel încât să nu se mai întoarcă niciodată la programul de bază. Această situaţie, bineînţeles, ar mai putea rezulta din defecţiuni hardware nedetectate şi nu numai din erori de programare (program bugs).

Un temporizator (timer) hardware, numit şi "watchdog timer", este folosit în sisteme de comutare telefonică pentru apărare împotriva erorilor de programare din cauza cărora sistemul nu-şi mai poate reveni. Conceptul este relativ simplu. Un ceas hardware merge continuu în procesor şi este resetat periodic de către programul principal dacă nimic neobişnuit nu se întâmplă ca să devieze acest program din secvenţa lui normală de execuţie. Dacă dintr-un anumit motiv (cum este o eroare software sau o defecţiune hardware) secvenţa execuţiei nu se mai întoarce la programul principal, ceasul nu se resetează. Atunci el pune în circulaţie o cerere de

întrerupere de înaltă prioritate şi face acţiunile necesare ca să reiniţializeze sistemul. În cazul în care există în sistem încă un procesor standby, întreruperea ceasului poate produce comutarea automată a controlului pe maşină standby, într-o tentativă de revenire din eroarea de sistem.

Operaţiile circuitelor, mai ales cele care implică comunicarea între două unităţi funcţionale, folosesc de asemenea controale de temporizare. De exemplu, în accesarea memoriei, transferul de date între CPU şi memoria principală (main store) se face de obicei asincron prin "handshaking". Atunci când este iniţiată o operaţie de citire a memoriei, CPU-ul aşteaptă un semnal de răspuns din partea unităţii de control a memoriei, care arată, că cuvântul de date adresat este pregătit pentru a fi transferat. Dacă, pentru anumite motive, ca de exemplu o defecţiune hardware în logica de control, nu se generează semnalul de răspuns, CPU-ul va aştepta nelimitat. Din moment ce operaţia normală de citire a memoriei ia un anumit timp, un ceas hardware poate fi folosit pentru detecţia erorilor.

În calculatorul Tandem care este proiectat să funcţioneze nonstop, controalele de temporizare sunt integrate în sistemul de operare ca una din măsurile majore pentru detectarea erorilor. Fiecare procesor trimite un mesaj special din secundă în secundă la toate celelalte procesoare din sistem. În afară de acest lucru fiecare procesor controlează din două în două secunde dacă au ajuns semnalele de la celelalte procesoare. Dacă un mesaj nu este recepţionat se presupune ca procesorul respectiv s-a blocat (failed) şi se acţionează corespunzător. De asemenea, control de temporizare se utilizează pentru toate operaţiile de intrare-ieşire.

Cu toate că semnalele de "timeout" de la un ceas hardware indică existenţa unei probleme în sistem, absenţa lor nu înseamnă neapărat că sistemul lucrează satisfăcător. Controalele de temporizare asupra funcţionării unui circuit nu sunt controale complete pentru sistem. Ele relevă prezenţa defecţiunilor (erorilor), dar nu şi absenţa lor. Ca urmare, arhitecţii calculatoarelor folosesc controalele de temporizare ca să suplimenteze alte controale şi ca să acopere un procent mai mare de defecţiuni dintr-un sistem.

2.2.4. Controale de excepţie

Programele rulează în medii protejate folosind seturi de constrângeri prestabilite. Dacă programele sunt fără erori, ele ţin cont de constrângeri şi realizează corespunzător funcţiile specificate, dar greşelile de proiectare din software adesea violează aceste constrângeri şi în felul acesta se poate influenţa nefavorabil întregul sistem. Unele circuite hardware de detecţie sunt adesea proiectate în sistem pentru a recunoaşte erorile de proiectare şi a le trata ca excepţii. Deci, tratarea excepţiilor se referă la detectarea şi "mânuirea" evenimentelor nenormale sau nedorite.

Aceste constrângeri pot fi atribuite hardware-ului sau software-ului. Câteva exemple hardware sunt:

- alinierea de adrese improprie
- locaţii de memorii neechipate
- opcod neutilizat
- depăşirea stivei

Aceste constrângeri de obicei rezultă de la faptul că hardware-ul nu este capabil să furnizeze serviciile necesare pentru software, de exemplu cazul alinierii improprii a

adreselor. În multe calculatoare moderne, tipurile de date întregi de mărime 8, 16, 32, şi 64 de biţi (byte, half-word, word, double word) sunt susţinute de setul de instrucţiuni. Este convenabil şi mai simplu din punct de vedere hardware ca aceste tipuri de date să fie aliniate pe cuvinte, aceasta fiind limita care se potriveşte cu aranjarea (organizarea) memoriei (Figura 2.18). Prin alinierea tipurilor de date pe cuvinte, deviaţiile de la format sunt detectate ca excepţii ale unei alinieri de adresă improprie.

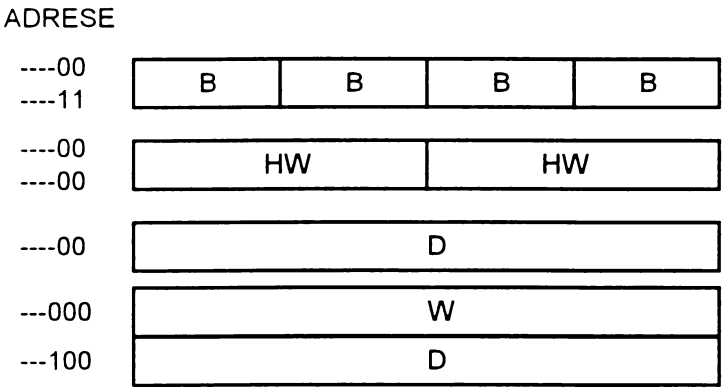


Figura 2.18 Aşezarea tipurilor de date în memorie.

O constrângere hardware, bineînţeles că îngreunează suplimentar programatorul de la observarea constrângerilor predefinite. Deoarece costul hardware-ului continuă să scadă, este normal să se reducă sau chiar să se elimine unele constrângeri hardware. De exemplu, calculatorul VAX 11/780 are tipuri de date întregi de 8, 16, 32, şi 64 de biţi. Sistemul este "generalizat" şi astfel aceste tipuri de date ocupă 1, 2, 4, şi 8 octeţi succesivi, începând de la orice octet limită. Astfel de considerente încep să fie aplicate din ce în ce mai mult în proiectarea calculatoarelor moderne.

Structura software pune de asemenea câteva constrângeri sistemului pentru a îmbunătăţi robusteţea lui şi pentru a furniza un mediu protejat pentru programele de aplicaţie. Câteva controale de excepţie software sunt:

- execuţie ilegală a instrucţiunilor privilegiate
- depăşire de adresă (out-of-range address)
- violarea memoriei (memory acces violation)
- operanzi ilegali
- operaţii aritmetice necorespunzătoare cum sunt: depăşirea, şi împărţirea cu zero.

Detectarea unei erori iniţiază o excepţie, care este urmată de invocarea automată a subrutinei de mânuire a excepţiei respective. În multe cazuri excepţia este atribuită unei greşeli de proiectare a programului.

2.2.5. Restabilirea erorii

Restabilirea este cea mai complexă şi cea mai dificilă funcţie pentru toate sistemele [TOY88]. Neajunsurile proiectării hardware sau software de a detecta erorile atunci când ele apar au un efect direct asupra abilităţii sistemului de a se restabili. Când erorile continuă să fie nedetectate, sistemul rămâne defect până când

este recunoscută problema. Un alt tip de problemă de restabilire apare dacă sistemul nu este capabil să izoleze corespunzător un subsistem defect şi să reconfigureze în jurul lui un sistem operaţional. Mulţimea mare de stări posibile ale sistemului care pot să apară sub condiţii problematice face restabilirea un proces complicat.

Cu cât este detectată mai rapid o eroare, cu atât este mai uşor să se determine care componentă este defectă ca să se realizeze restabilirea. Deci, detectarea rapidă a erorii şi stăpânirea (izolarea) ei corespunzătoare sunt condiţii fundamentale pentru ca restabilirea să se facă cu succes.

2.2.5.1. Clasificarea procedurilor de restabilire

Există trei clase de proceduri de restabilire: restabilirea totală (full recovery), restabilirea degradată (degraded recovery) şi închiderea pentru salvare (safe shutdown). Procedurile de restabilire pot fi invocate automat (fără nici o interacţiune între operatorul de întreţinere şi sistem) sau manual. Restabilirile iniţiate manual fac uz de secvenţe vaste controlate prin program. Cele trei clase de proceduri de restabilire sunt descrise mai jos.

- Restabilirea totală. Într-o facilitate de timp real, sistemul furnizând continuu servicii trebuie să rămână operaţional chiar şi într-un mediu defect. Aceasta înseamnă că simptomele trebuie să fie recunoscute repede şi unităţile defecte să fie reparate cu puţină sau chiar fără intervenţie din partea utilizatorului. Pentru a dispune de sistem, tot timpul la întreaga sa capacitate, subsistemele corespunzătoare de schimb trebuie să fie disponibile ca unităţi de înlocuire pentru cele defecte. O procedură de restabilire totală, uzual, are nevoie de toate cinci aspectele calculării tolerante la defecţiune: detectarea erorii, izolarea erorii, restabilirea sistemului, diagnosticarea erorii, şi repararea. Secvenţa evenimentelor este descrisă în figura 2.19. Înaintea oricărei acţiuni a sistemului, defecţiunea trebuie să fie detectată. Obiectivul izolării defecţiunii este să identifice subsistemul (de exemplu, un modul de memorie, o unitate de procesare, un controlor al unui canal de I/O, etc.) unde defecţiunea a apărut. Prin utilizarea comutării automate, controlate prin program, sistemul este reconfigurat prin interschimbarea subsistemului defect cu unul de rezervă corespunzător. Procedura de restabilire este atunci iniţiată să restaureze sistemul la capacitatea lui de calculare originală, fără să piardă nici o caracteristică hardware sau software. Perioada de întrerupere a servirii, în timpul procesului de restabilire, este minimizată şi de obicei nu este observabilă de utilizator. Diagnosticarea şi repararea, care sunt sarcinile cele mai consumatoare de timp ale procedurii de restabilire, pot fi amânate şi intercalate cu operaţia normală a sistemului (figura 2.19).
- Restabilirea degradată. Aceasta este de asemenea referită şi ca o operaţie de degradare graţioasă (graceful degradation operation). Ca şi în cazul restabilirii totale, toţi paşii implicaţi la tolerarea defecţiunii (detectare, izolare, restabilirea sistemului, diagnosticare, şi reparare) trebuie să fie incluşi în procedură. Secvenţa de evenimente observată în figura 2.19 se aplică de asemenea la procedura degradată, exceptând faptul că în acest caz nu se cuplează alt subsistem în locul celui defect. În schimb, componenta defectă este scoasă din funcţie şi sistemul se întoarce la o stare operaţională fără defecţiuni. Funcţiile de calculare selectate sunt lăsate să opereze în sisteme care utilizează proceduri de restabilire degradată, iar

din cauza aceasta caracteristicile performanței lor în timp real scad sub un standard normal acceptat până când se realizează reparațiile corespunzătoare.

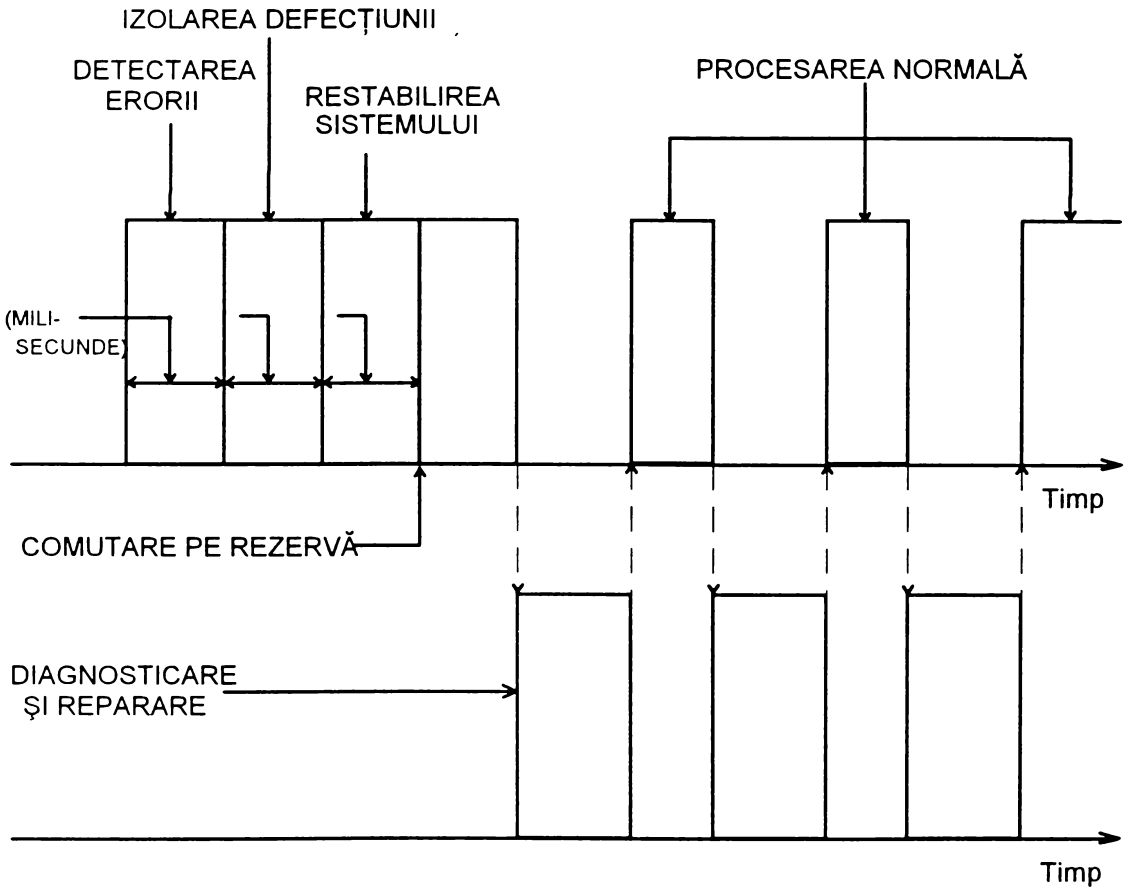


Figura 2.19 Secvența de restabilire după eroare (cu piesă de schimb).

- Închidere pentru salvare. Aceasta este adesea denumită şi operație de salvare din malfuncționare (fail-safe operation). Ea apare ca un caz limită al restabilirii degradate, când capacitatea de calcul a sistemului a scăzut sub limita minimă acceptabilă a operației. Scopurile închiderii pentru salvare sunt:
  1. Să se evite distrugerea oricăror elemente din sistem sau a modulelor software stocate (programe sau date) care pot fi încă compromise după ce operația de restabilire degradată a fost lăsată să aibă loc;
  2. Să se termine interacțiunea cu orice sistem asociat (utilizator) într-un mod ordonat;
  3. Să se distribuie diagnostic şi mesaje de închidere la utilizatori, la anumite sisteme şi la personalul de întreținere.

Categoria procedurii de închidere pentru salvare este similară cu un sistem fără nici o redundanță (figura 2.20). Acțiunea de izolare a defecțiunii trebuie să fie inițiată pentru a se determina identitatea ei şi locul unde se află. Operația normală a sistemului, care a fost momentan întreruptă la momentul detectării defecțiunii, trebuie acum să fie suspendată datorită diagnosticării şi reparării. Sistemul, atunci, trebuie să fie restabilit la o astfel de stare hardware şi la un punct din program de unde să se poată relua procesarea normală.

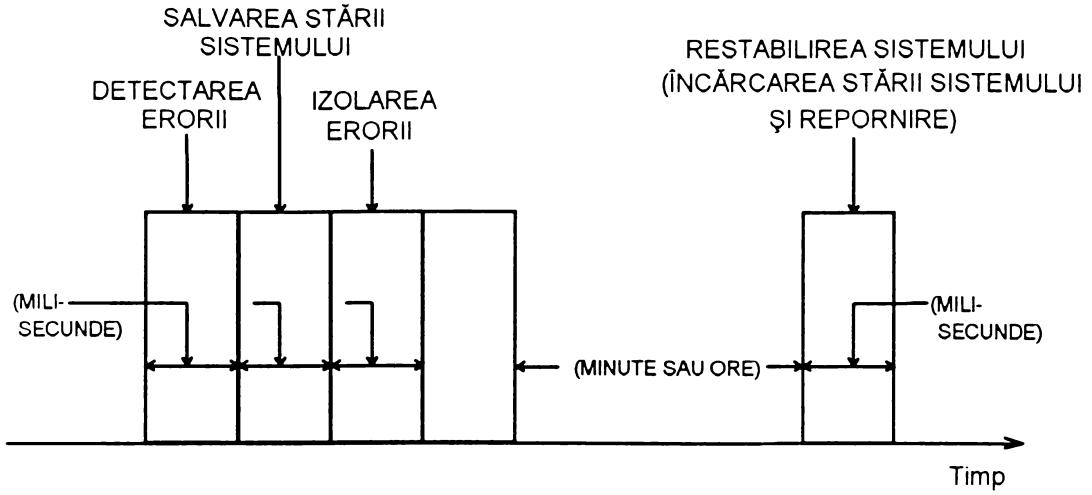


Figura 2.20 Secvența de restabilire după eroare (fără piesă de rezervă).

La toate trei tipurile de restabilire după defecțiune, este necesar să se facă anumite operații după ce defecțiunea este detectată. O comparație între figurile 2.19 şi 2.20 scoate în evidență câteva avantaje de întreținere atunci când există redundanță hardware. Primul avantaj constă în faptul că sarcinile care consumă mult timp (diagnosticare şi reparare) pot fi amânate şi intercalate cu procesarea normală a sistemului pe baza împărțirii timpului (time-shared) după ce sistemul a fost restabilit la starea operațională. Al doilea avantaj se bazează pe faptul că disponibilitatea piesei de schimb permite ca maşina fără defecțiune să interogheze şi să diagnosticheze maşina defectă. Acest tip de testare este uşor de făcut automat sub controlul unor programe de autocontrol. Altfel, este necesar a avea un operator pentru a forța manual maşina prin paşii de diagnosticare şi restabilire.

2.2.5.2. Reconfigurarea

În sistemele redundante un ansamblu de piese de schimb sau de unități multifuncționale asigură continuitatea funcționării sistemului. Structura cea mai simplă este configurația duplex în care fiecare unitate funcțională este duplicată. Dacă una din unități se defectează, unitatea duplicată se cuplează şi se păstrează continuitatea funcționării în timp ce unitatea defectă se repară. Dacă apare o defecțiune în unitatea duplicată în timpul intervalului de reparație, bineînțeles că întregul sistem se va deteriora, dar dacă intervalul de reparație este relativ scurt, atunci, probabilitatea de apariție simultană a defecțiunilor în două unități identice este foarte mică.

Capacitatea sistemului de a reconfigura dinamic modulele lui, într-un sistem în lucru, furnizează operația continuă necesară pentru multe aplicații critice în timp real. În general, redundanța hardware la nivel de subsistem este esențială la tolerarea defecțiunii şi la uşurarea muncii de reparație. Au fost utilizate cu succes câteva structuri ca să se realizeze înaltă fiabilitate bazându-se pe reconfigurarea dinamică. Figura 2.21(a) arată cea mai simplă structură duplex. CPU-ul şi magistrala asociată lui sunt duplicate. O singură maşină este activă şi controlează sistemul; cea de rezervă este utilizată ca unitate "standby". În acest aranjament, maşinile standby nu produc sau nu contribuie la nici o muncă utilă în afară de situațiile de nevoie. Figura 2.21(b) arată un aranjament cu împărțire a încărcării (load-sharing arrangement) în care amândouă CPU-urile sunt active realizând concurent diferite operații. Atunci când o eroare hardware care reduce performanța capacităților sistemului este detectată şi localizată

la una din unități, unitatea de procesare defectă este scoasă din funcție. Dacă aplicația poate tolera acest fel de deteriorare, configurația multiplă activă furnizează per totalul sistemului performanță mai bună decât un sistem duplex simplu.

Metoda procesoarelor multiple combinate cu piese de rezervă standby este arătată în figura 2.21(c). Pretențiile de performanță şi fiabilitate sunt satisfăcute în mod ideal prin această structură. Structura este modulară, în care sistemul poate creşte grațios prin adăugarea modulelor de procesare. Fiabilitatea înaltă este realizată prin numărul de module de schimb care se află în sistem. Dacă unul din modulele active se deteriorează, îl înlocuieşte una din rezerve şi sistemul îşi revine la capacitatea lui maximă. Performanța maximă este întreținută până când se epuizează toate piesele de schimb.

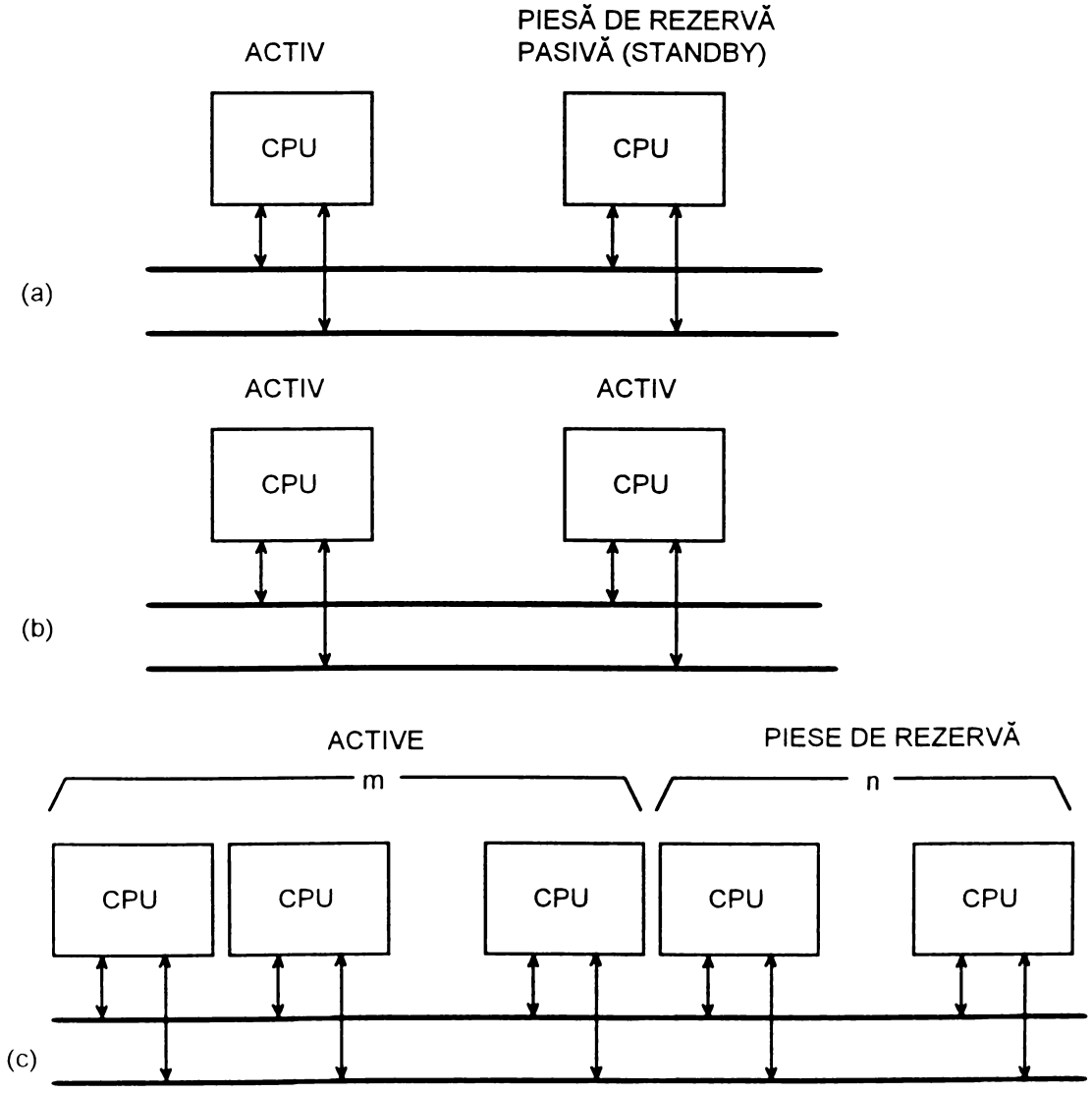


Figura 2.21 Structuri cu redundanță dinamică.
(a) Duplex cu rezervă pasivă.
(b) Duplex cu rezervă activă.
(c) Multiprocesor cu piese de rezervă.

Structura duplex la care şi unitatea de rezervă este activă (activ-standby), aşa cum se observă în figura 2.21(b) este foarte simplă. Pentru simplitatea ei, această

structură redundantă a fost utilizată peste tot în procesoarele ESS [TOY78]. Pentru procesoarele ESS medii şi mici, figura 2.22 arată o structură de sistem conținând câteva unități funcționale care sunt tratate ca şi o singură entitate. Structura se compune din două unități de stocare, program store (PS) şi call store (CS). PS-ul este în memoria read-only şi conține programe de procesare ale apelului, programe de întreținere şi de administrare. De asemenea conține parametrii de sistem. În această aranjare întregul procesor este tratat ca şi un singur bloc funcțional care este dublat. Acest tip de sistem duplex cu o singură unitate are două configurații posibile: fie procesorul 0 fie procesorul 1 poate fi asignat ca şi sistem de lucru on-line, în timp ce cealaltă unitate serveşte ca şi o unitate de rezervă "backup standby" (redundanță activă). Configurația duplex cu o singură unitate are avantajul de a fi foarte simplă în ceea ce priveşte numărul de blocuri de comutare în sistem. Această configurație nu simplifică numai programul de restabilire, dar şi interconexiunile hardware, pentru că sunt eliminate accesele adiționale necesare ca să facă fiecare bloc duplicat capabil să se cupleze independent în configurația sistemului on-line.

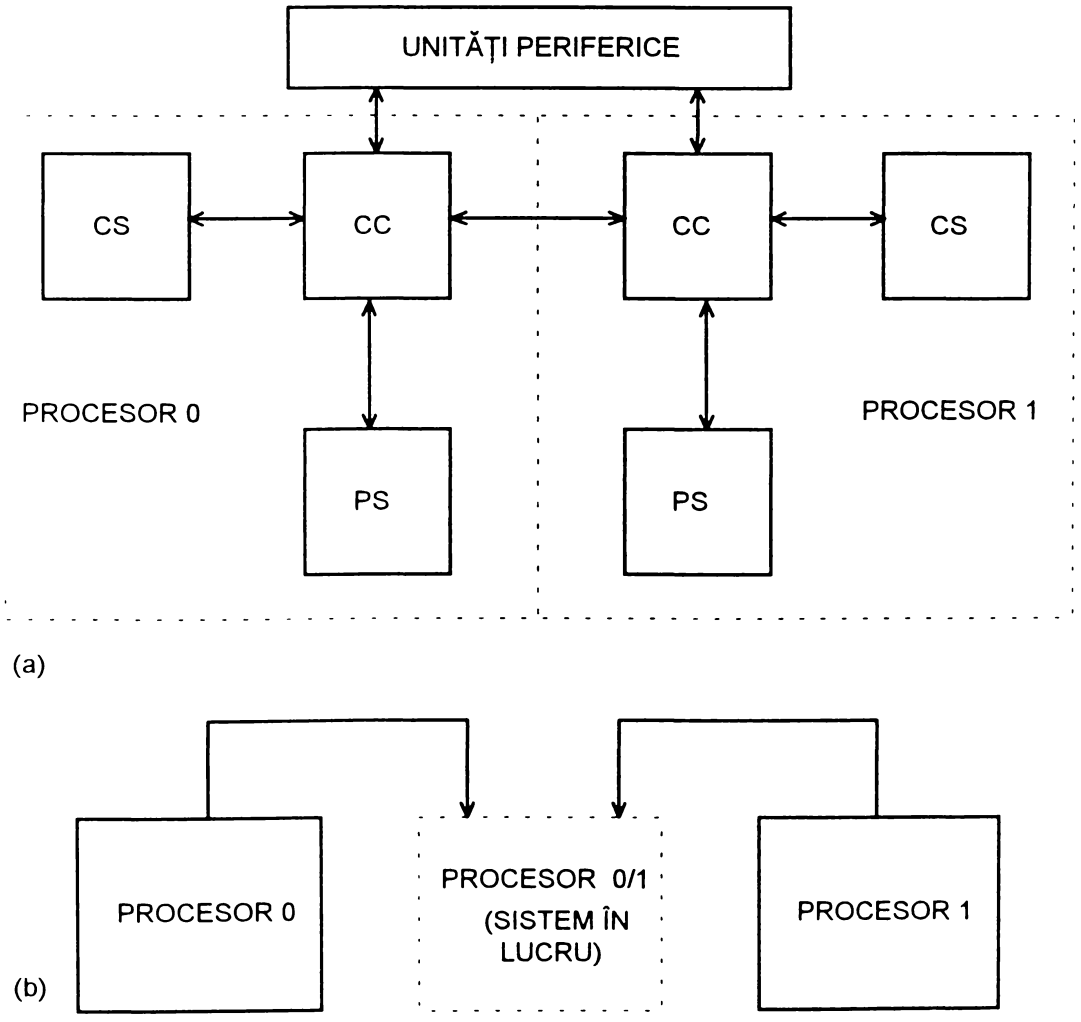


Figura 2.22 Configurație duplex de unitate-singulară.
(a) Structura procesorului.
(b) Două configurații posibile.

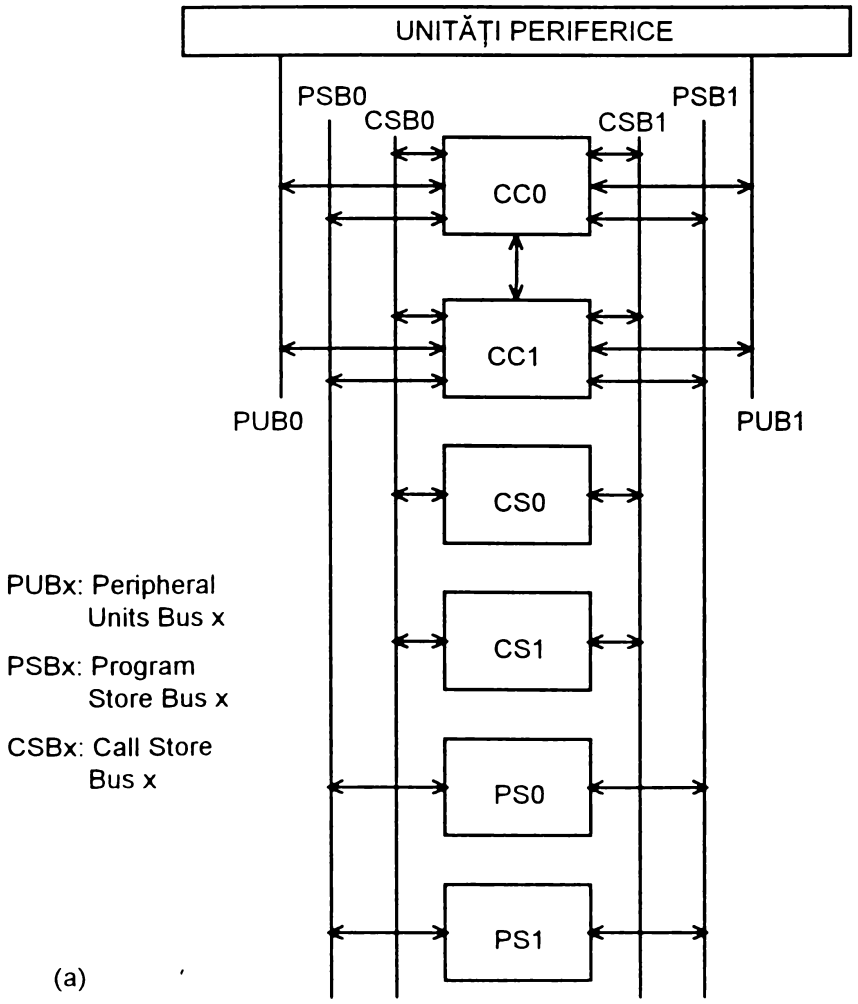
În sistemele mari de comutare 1ESS care conțin multe componente, timpul mediu de defectare (MTTF) devine prea mic pentru a putea satisface cerințele

standard de fiabilitate. Ca să crească MTTF-ului, trebuie să scadă sau numărul de componente neredundante (rata de defectare) sau timpul de reparație.

Configurația duplex cu o singură unitate poate fi partiționată într-o configurație duplex multiunitate (figura 2.23). În acest aranjament, fiecare subunitate conține un număr mic de componente care pot fi comutate într-un sistem în lucru. Sistemul va pica numai dacă o eroare apare în subunitatea redundantă, în timp ce subunitatea originală este supusă reparației. Din moment ce fiecare subunitate conține câteva componente, probabilitatea de apariție a defecțiunilor simultane într-o pereche dublată de subunități este redusă. Un sistem în lucru este configurat cu un aranjament fără defecte

$$CCx - CSx - CSBx - PSx - PSBx - PUBx,$$

unde x este sau subunitatea 0 sau subunitatea 1. Acest lucru înseamnă că există 2^6 adică 64 de combinații posibile de configurații ale sistemului. Reconfigurarea sub condiții problematice într-un sistem în lucru poate fi o sarcină complexă, depinzând de gravitatea defecțiunii. De exemplu, procesorul poate să piardă abilitatea de a lua deciziile corespunzătoare. Această problemă în procesorul 1AESS este adresată printr-un circuit hardware autonom la fiecare CC ca să asiste la asamblarea unui sistem funcțional.



(b)

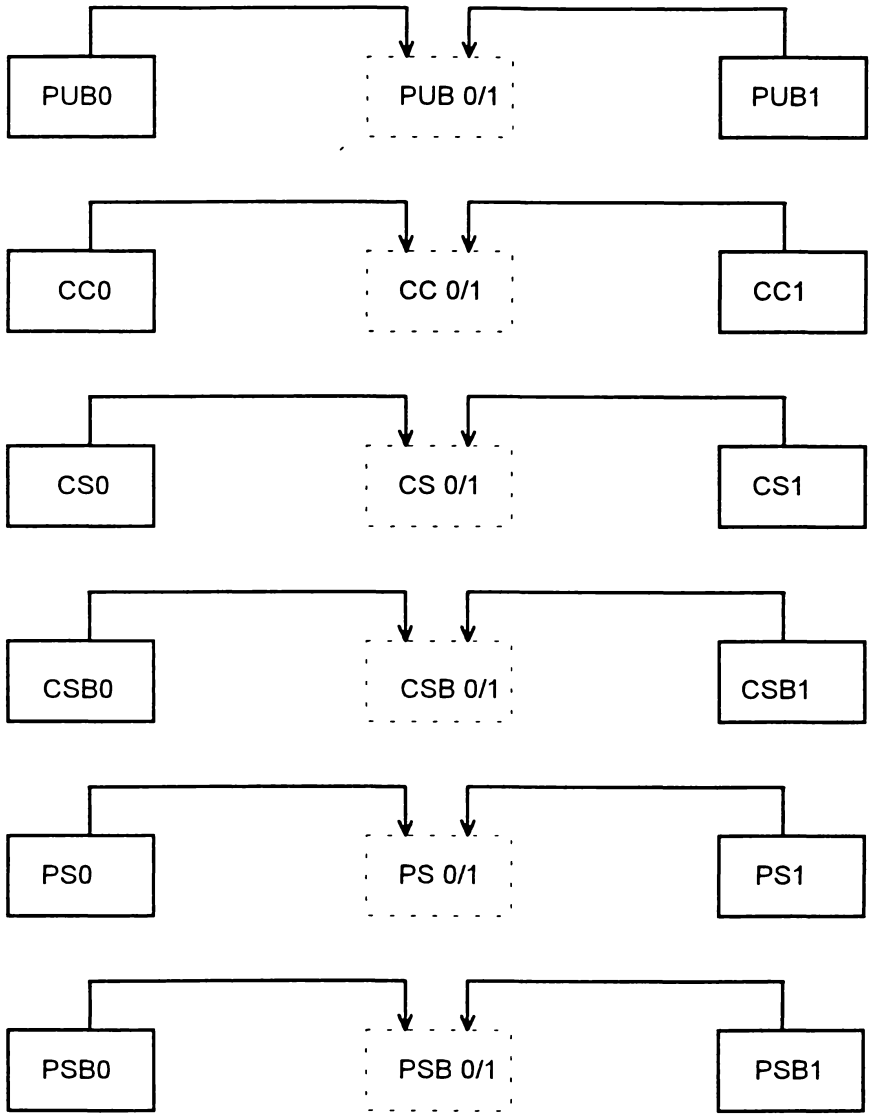


Figura 2.23 Configurație duplex multiunitate.
a) Structura procesorului;
b) Şaizeci şi patru de posibilități.

2.3. Metode de autocontrol implementate software

Necesitatea dezvoltării mecanismelor software pentru detectarea erorilor software este foarte acută atunci când se pune problema construirii unor sisteme fiabile. Mecanismele hardware, chiar dacă sunt foarte avansate tehnologic, pot detecta şi izola defecțiunile hardware dar nu sunt specializate în analizarea erorilor software. Primele scheme software de detecție se bazau pe mecanismele de protecție hardware cum sunt controalele de excepție pentru indicarea apariției unei erori de program. Această sprijinire pe mecanismele hardware nu este eficientă, pentru că de multe ori în timpul în care o eroare software cauzează o excepție hardware, ea se propagă şi afectează fără rost o bună parte din sistem.

Eroarea inițială se poate "aranja" în aşa mod încât să treacă nedetectată şi necorectată. În sistemele mari modalitatea detectării erorilor software bazată pe principiul că eventual, ele cauzează excepții hardware este irealizabilă. Sunt necesare

facilități de detecție mai bune, dar proiectarea lor nu este deloc simplă, din cauză că erorile software nu se pot distinge așa cum se disting deteriorările componentelor hardware. Ele trebuie să se detecteze rapid în timpul execuției programului mai ales în situațiile sistemelor care lucrează în timp real. În cele mai multe cazuri procesul se oprește, rezultând o operație eronată, și multe date se distrug irecuperabil.

Spre deosebire de ratele de deteriorare ale componentelor hardware, despre comportamentul și distribuția erorilor software nu există statistici fiabile (de încredere). Este foarte dificil să se aprecieze greșelile de proiectare sub formă de "deteriorări de așteptat". Detecția erorilor software se poate realiza numai prin recunoașterea comportamentului anormal al sistemului. Așadar, este nevoie de un set de standarde pentru descrierea comportamentului normal, ca să se poată sesiza eventualele devieri de la el. Prin controlarea motivațiilor comportării programului la anumite stadii ale calculării, se detectează erorile destul de devreme pentru a se limita propagarea datelor eronate.

Există mai multe tehnici generale pentru urmărirea comportamentului unui sistem de calcul ca să se detecteze devierile de la comportamentul normal.

- Funcția unui proces. Motivarea ieșirilor pentru un set dat de intrări este controlată la nivelul blocului funcțional. Anumite măsuri de performanță pot fi folosite pentru a se indica dacă sistemul funcționează corespunzător. Când încărcarea (workload) aplicată este normală dar măsurătorile care caracterizează performanța sistemului (ca și timpul de răspuns, debitul (throughput), și timpul necesar pentru a se realiza o funcție standard) sunt în afara valorilor limită, probabil că sistemul are una sau mai multe erori.
- Secvența de comandă a unui proces. Secvența de calcule făcută de un proces care se află în execuție este referită cu numele de secvență de comandă (control sequence). Fiecare calcul schimbă multe stări ale sistemului și neregularitățile sistemului pot fi detectate.
- Date de proces. Integritatea datelor unui sistem și structura lui pot fi observate în timp ce sistemul execută secvența programului. Deteriorarea datelor poate fi cauzată atât de o defecțiune hardware cât și de o eroare software.

Un proces este definit aici ca fiind o porțiune de calcule de sine stătătoare, care, odată inițiată, se realizează complet fără a mai avea nevoie de intrări adiționale.

2.3.1. Controlul funcțional

Aspectele funcționale ale unui proces pot fi controlate prin verificarea motivării (justificării) ieșirilor pentru un set dat de intrări (functional checking). Când relația dintre intrări și ieșiri este unu la unu, pot fi utilizate controale inverse (reversal checks) pentru verificarea corectitudinii procesului. Un control invers ia ieșirile sistemului și calculează ce intrări ar fi trebuit să fie. Intrările calculate sunt apoi comparate cu cele actuale pentru a controla condițiile de eroare. În sistemele 1AESS de comutare AT&T, un simplu control invers este aplicat la o operație de scriere pe banda magnetică prin citirea datelor scrise pe banda magnetică și compararea lor cu datele originale. Bine definitele funcții matematice apelează adesea la controalele inverse. De exemplu, dacă ieșirea este soluția unui set de ecuații matematice, corectitudinea ieșirii poate fi verificată prin substituția ieșirii într-o ecuație și controlând consistența. În unele cazuri, poate fi o simplă relație între variabilele de

ieşire. Ieşirea poate fi verificată prin controlul acestei relaţii. De exemplu, ieşirea unui program de sortare poate fi testată pentru a se asigura că este într-adevăr sortată (ordonată) şi conţine numărul corect de elemente.

În multe alte cazuri, corectitudinea ieşirii poate fi verificată numai printr-un algoritm care este tot atât de complicat ca şi algoritmul original, riscând astfel, prin utilizarea lui, introducerea altor erori. Această situaţie complică mai mult fiabilitatea sistemului. Un control riguros al corectitudinii ieşirii nu este practic, de aceea este controlată doar motivaţia ieşirii. O ieşire nerezonabilă indică de obicei prezenţa erorilor, dar nu şi invers (adică o ieşire rezonabilă nu exclude existenţa erorilor).

Controalele de temporizare cum sunt cele menţionate în secţiunea 2.2.3. sunt de asemenea utilizate ca să controleze procesele date. Un control de temporizare este o cale clasică de observare uşoară a performanţei interne a sistemului în timpul detectării strangulărilor "bottlenecks" şi buclelor infinite. Procedura presupune setarea unui temporizator "timer" ca să activeze o alarmă după trecerea unui timp adecvat sistemului pentru a îndeplini funcţia respectivă în cazul în care totul merge bine. Dacă întreruperea de la timer a apărut înainte ca sistemul să-l reseteze, tratarea întreruperii trebuie să presupună că procesul nu s-a realizat corespunzător. Din nou este un control pentru motivaţie. Lipsa timeout-ului, adică, neactivarea cererii de întrerupere nu implică în mod cert absenţa erorilor.

Controlul funcţional bazat pe software este analog cu o metodă "black-box" în testarea sistemelor hardware, examinând relaţia dintre intrările şi ieşirile corespunzătoare. Controlul funcţional software nu este nici eficient, nici eficace. O cale mai bună pentru detectarea erorii este realizată când un acces direct la o structură internă monitorizează comportarea ei internă incluzând şi secvenţa de comandă şi comportarea dinamică a variabilelor critice sau globale.

2.3.2. Controlul secvenţei de comandă

Atunci când fiecare sarcină reprezintă un set bine definit de operaţii, secvenţa corectă în care aceste operaţii sunt executate determină ieşirea potrivită a procesului. Orice deviaţie de la secvenţa de execuţie specificată, produce o ieşire eronată. O eroare software care cauzează o secvenţă de execuţie incorectă este numită eroare de comandă (control fault). O eroare de comandă poate avea următoarele consecinţe:

- execuţia unei bucle infinite
- execuţia unei bucle de program de un număr de ori incorect
- traversarea unui salt greşit sau ilegal

Aceste tipuri de condiţii de eroare pot fi detectate prin diferite metode proiectate pentru a controla secvenţa de comandă (control sequence checking). Au fost implementate [TOY88] trei tipuri de secvenţe de comandă: control de salt-permis (branch-allowed checking), metoda de rulare în ture (relay-runner scheme), şi combinarea metodei de rulare în ture cu cea a temporizatorului "watchdog"

Un control de salt-permis este un mijloc de detectare a execuţiei unei operaţii improprii de salt. Acest tip de control este implementat în câteva sisteme de comutare electronică AT&T. Aşa cum se observă în figura 2.24, un bit de control numit bit BA, (Branch-Allowed), este asignat la fiecare cuvânt din memoria principală. Dacă bitul BA este pe "0" atunci conţinutul locaţiei din memoria principală nu poate fi referit de nici o instrucţiune de salt. Totuşi locaţia poate fi referită prin numărătorul de program

(PC) în modul lui normal de adresare secvențială (acesta este modul în care PC este incrementat cu unu ca să indice locația instrucțiunii următoare). Dacă bitul BA este pe "1", conținutul acestei locații poate fi referit de oricare instrucțiune de salt aflată oriunde în memoria principală. Dacă o instrucțiune de salt se execută într-o secvență de procesare normală, bitul BA al locației țintă este controlat pentru a se vedea dacă o operație de salt poate să apeleze conținutul acelei locații. Dacă bitul BA este în starea zero, înseamnă că a fost executat un salt impropriu şi logica controlului BA indică faptul că a avut loc o eroare.

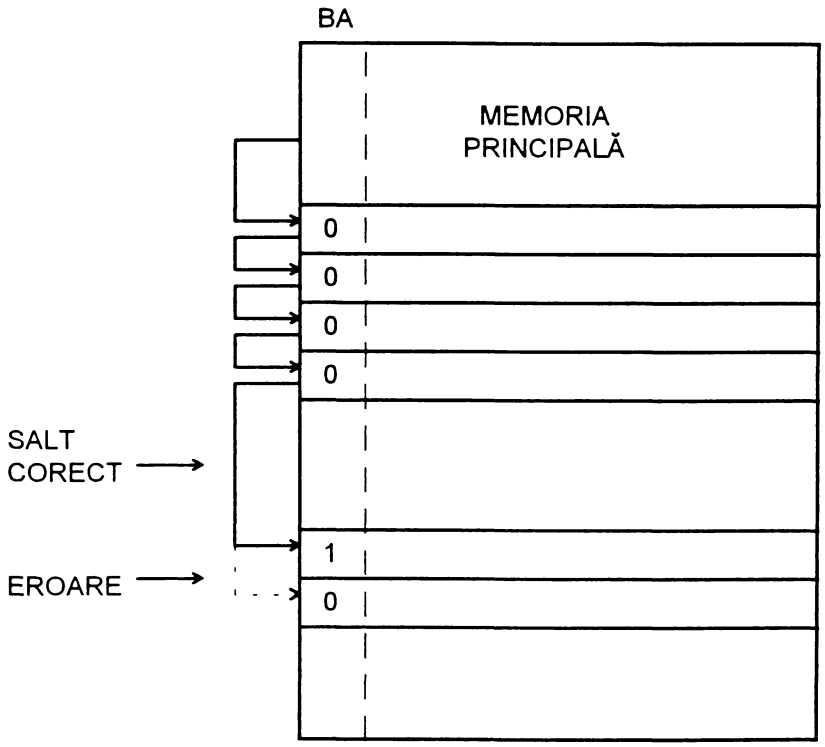


Figura 2.24 Control de salt-permis.

De asemenea, metoda de rulare în ture oferă protecție împotriva salturilor ilegale care pot fi cauzate de erorile software sau de defecțiunile hardware. În această metodă o ştafetă (baton) care este similară cu o parolă (password), este purtată împreună cu transferul de comandă şi este controlată în punctele corespunzătoare.

Când se face un salt ilegal, comanda nu are valoarea validă a ştafetei. Eroarea este detectată la următorul punct de control (checkpoint). Figura 2.25 arată o diagramă a acestei metode. O bucată din program este partiționată în blocuri funcționale separate prin puncte de control de tură (relay). Aceste puncte de control sunt instrucțiuni condiționate care testează dacă cursul programului este corect, controlând dacă codul curent al ştafetei purtate cu el este valid. Programul de aplicație înscrie primul cod din seria de coduri ale ştafetei într-o adresă specificată, de exemplu CODE1.

Când execuția programului ajunge în primul punct de control de tură, instrucțiunea compară conținutul de la CODE1 cu un număr de cod prestabilit. Dacă sunt egale, conținutul de la CODE1 este pus pe zero şi un nou cod de ştafetă este stocat în locația CODE2. Dacă codurile nu sunt egale, este apelată o rutină numită capcană de eroare (error-trap) şi execuția normală este oprită. Procesul de control şi de reactualizare a codului ştafetei este realizat în diferite puncte strategice din întregul

program. Metoda este analoagă cu alergarea cu ştafete, în care un alergător nu porneşte alergarea decât după ce recepţionează ştafeta de la alergătorul anterior.

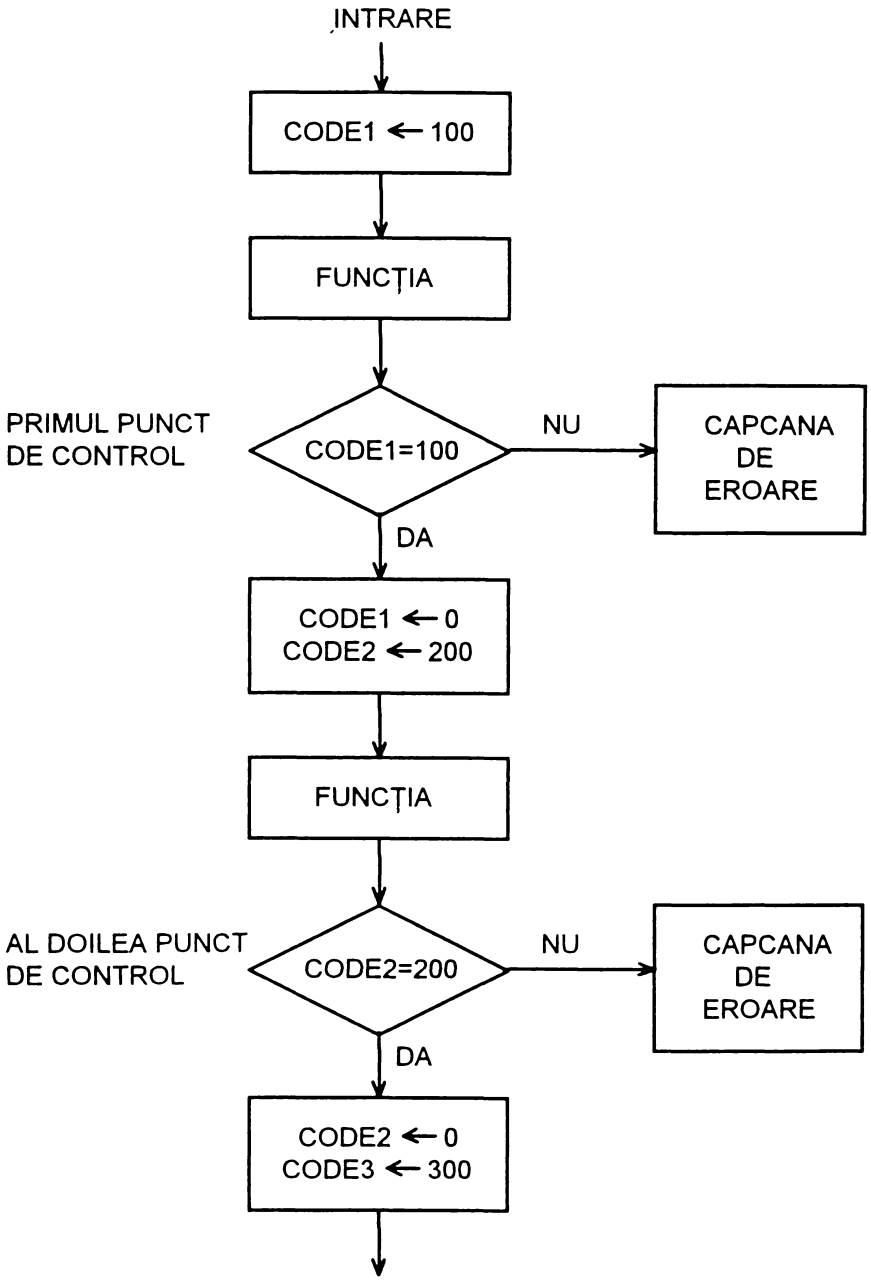


Figura 2.25 Schema de rulare în ture (relay-runner)

Combinaţia dintre temporizatorul "watchdog" şi metoda de rulare în ture combină eficacitatea operaţiei de temporizare watchdog la detectarea buclelor infinite de program cu abilitatea metodei de rulare în ture de a detecta erorile din structura de comandă a secvenţei unui program.

Deşi operaţia unui temporizator hardware este simplu de implementat pentru majoritatea procesoarelor, totuşi precauţiile adiţionale în secvenţa de program şi în generarea semnalelor de reset ale temporizatorului sunt furnizate de metoda de rulare în ture. Temporizatorul hardware trebuie să fie resetat numai atunci când sistemul lucrează corespunzător; dacă nu, temporizatorul trebuie lăsat să ceară o acţiune de revenire. De exemplu, programul poate fi segmentat, cu controale de tură inserate în punctele corespunzătoare (figura 2.26). În plus, un marcator păstrează numărul de

puncte de control care au fost traversate de program. Temporizatorul este activat la primul segment de program. Înainte ca resetul temporizatorului să fie generat, programul trebuie să treacă cu succes prin fiecare punct de control şi să aibă parcurs un număr corect de puncte de control. Al doilea control, în care metoda de rulare în ture asigură că toate segmentele sunt succedate corect, este redundant. Dacă programul sare în faţă, peste unul sau mai multe segmente de program, codul ştafetei nu va fi egal cu numărul de cod al punctului de control următor. Dacă programul sare înapoi cu unul sau mai multe segmente, valoarea ştafetei vechi este deja resetată la zero, astfel controlul punctului de tură va detecta această condiţie.

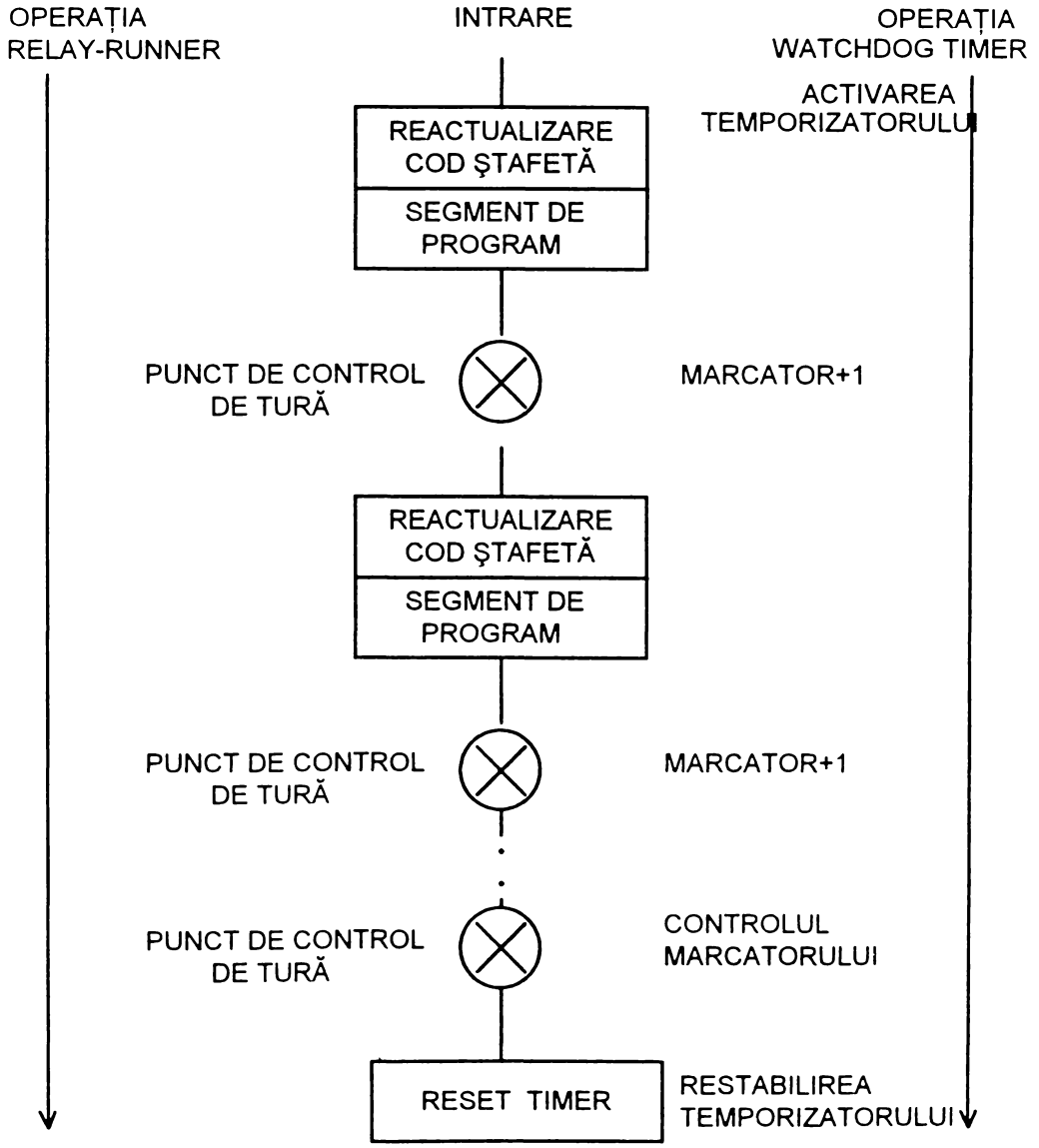


Figura 2.26 Combinaţie între schema de rulare în ture şi cea a temporizatorului watchdog.

În [MAHM88] s-au clasificat metodele cunoscute pentru controlarea secvenţei de comandă în două categorii, cele cu verificarea semnăturilor asignate (assigned-signature checking) şi cele cu verificarea semnăturilor derivate (derived-signature checking). Cele din urmă sunt mai avantajoase atât prin prisma latenţei erorilor detectate cât şi prin cea a acoperirii lor. Metodele prezentate mai sus, adică controlul

de salt permis şi tehnica de rulare în ture, fac parte din prima categorie. În [SAXE90], Saxena şi McCluskey prezintă o tehnică bazată pe sume de verificare de precizie extinsă, care se încadrează în a doua categorie.

2.3.3. Controlul datelor

Erorile software, aşa cum s-a arătat anterior, pot fi cauzate atât de erori reziduale de proiectare software cât şi de defecţiuni hardware. Fiecare poate să ducă la distrugerea informaţiei memoriei. Deşi, uzual prin "date" se înţelege articolele de informaţie care sunt procesate de program, în general referirea este făcută la toate informaţiile stocate în memoria principală. Aceasta înseamnă atât instrucţiuni de program cât şi date. Textul care conţine instrucţiunile programului poate fi controlat printr-un control funcţional sau printr-un control al secvenţei de comandă, aşa cum a fost descris în secţiunile anterioare. Integritatea valorii datelor de instrucţiuni poate fi protejată uşor prin tehnici de codificare aşa cum sunt codurile corectoare Hamming şi evaluarea unei sume de control (checksum) a conţinutului unei porţiuni de cod. Codul de corecţie Hamming este foarte eficace în asigurarea integrităţii valorii datelor atât pentru instrucţiuni cât şi pentru date sub forma în care sunt ele utilizate şi procesate.

În afară de instrucţiuni, controalele de date pot fi realizate prin controale care se află în program (in-line) sau prin programe independente de detectare a erorilor software numite "audits". Controalele in-line includ în sistem un cod pentru a controla valabilitatea structurilor de date de fiecare dată când sunt procesate prin rutinele sistemului. Dacă structurile de date sunt controlate înainte ca ele să fie utilizate, erorile introduse anterior de componentele sistemului sunt identificate imediat după ce ele sunt modificate, iar rutina care a cauzat eroarea este de obicei identificată. De exemplu, dacă un articol este inserat într-o listă înlănţuită, un control poate fi făcut pe înlănţuire în direcţia opusă ca să se verifice integritatea ei după inserare. Testele construite corespunzător pot detecta multe erori cu o mică cantitate de procesare. Pe de altă parte, controlul extins introduce adesea o supraîncărcare neacceptabilă şi o scădere substanţială a performanţei.

Un alt control in-line eficace acoperă parametrii trecuţi la rutina sistemului de la programe utilizator. Este foarte posibil ca o eroare dintr-una din rutine să cauzeze picarea altei rutine din cauza unui parametru eronat. Acest lucru ascunde sursa originală a erorii. Deoarece controlul tuturor parametrilor transferaţi între rutinele sistemului ar produce, fără îndoială, foarte multă supraîncărcare, poate fi aplicat un control mai structurat cu capacitate adecvată pentru detectarea erorii. Această modalitate profită de avantajele din mediul programului structurat pe niveluri: parametrii sunt controlaţi în timp ce trec de la un nivel superior la unul inferior. De exemplu, programele utilizator se află la un nivel superior faţă de programele supervizor de sistem. Controalele nu sunt făcute în direcţie inversă, şi nici parametri nu sunt controlaţi între rutinele de pe acelaşi nivel. Prin această procedură simplă se realizează un echilibru între controlul erorii şi supraîncărcarea controlului in-line.

O alternativă la controlul in-line este audit-ul. Audit-urile sunt integrate în sistemul software, şi astfel în mod normal consumă o mică porţiune din capacitatea totală de procesare a sistemului. Sistemul invocă periodic audit-urile să ruleze rutinele de control. Audit-urile pot fi invocate manual când o problemă este suspectă. Programele audit controlează neconsecvenţele din structurile de date, care reflectă în mod normal operaţiile eronate ale sistemului. Ele nu anticipează problemele sau nu

determină cauzele lor. Astfel, ele sunt mecanisme complete şi rapide de detectare a erorii.

Câteva tehnici utilizate de programele audit sunt similare cu cele pentru controlul in-line. Aceste tehnici, includ controale de consecvenţă, controale de legătură, controale pentru integritate şi controale time-out. Controalele de consecvenţă (consistency checks) sunt bazate pe informaţia redundantă stocată cum sunt înregistrările backup. De exemplu, dacă programul şi datele critice sunt stocate în memoria principală, o copie backup de revenire este făcută pe un sistem de disc cu cost redus. Când sunt făcute schimbări, înregistrările sunt păstrate atât în versiunea anterioară cât şi în cea schimbată. Comparaţia backup-ului actualizat cu memoria principală furnizează un mijloc de detectare a erorii din memoria principală. Corecţia este făcută prin simpla scriere, de la stocarea secundară, peste datele deteriorate. În plus, dacă este necesar, versiunea anterioară a programului poate fi reîncărcată pentru a se permite o iniţializare mai drastică.

Controalele de legătură (linkage checks) verifică dacă registrele asociate cu facilităţi sunt corect legate împreună, prin utilizarea redundanţei inerente din structura de legătură. De exemplu, când o listă dublu înlănţuită este controlată (audited), redundanţa permite ultimei intrări în listă să fie identificată prin intrarea anterioară şi prin cea următoare. Similar, controlul unei bucle (loop-around) poate fi făcut de registre legate într-o listă circulară.

Controalele pentru integritate (integrity checks) sunt făcute pe date de stări asociate cu facilităţi. Aceste controale sunt făcute prin verificarea consecvenţei datelor cu stările actuale ale resurselor. Codurile detectoare de erori pot fi folosite ca să codifice starea facilităţilor.

Controalele time-out localizează facilităţile care pot fi puse fals în stare activă (busy). Dacă se precizează o limită absolută de timp pentru desfăşurarea unei facilităţi, ea poate fi examinată periodic cu o perioadă egală cu timpul maxim de desfăşurare. Dacă controlul (audit) găseşte o facilitate în stare activă (nonidle) după expirarea timpului maxim admis, controlul presupune că facilitatea s-a pierdut şi poate fi dezactivată şi deci o restaurează în stare operaţională.

Programele audit nu realizează observarea continuă a comportării sistemului ci realizează observarea eşantionat; deci, acest fel de programe generează mai puţină supraîncărcare decât controalele in-line. Ele totuşi, nu furnizează la fel de repede o detectare de eroare cum fac controalele in-line. Multe rutine pot accesa date invalide înainte ca un program audit să determine că o eroare a avut loc. Operaţiile anterioare cu date eronate pot cauza deteriorarea şi a altor date ale sistemului, făcând revenirea mult mai dificilă şi mai drastică. Pe de altă parte, supraîncărcarea relativ mică a programelor audit le permit să controleze sistemul mai în mare decât controalele in-line.

2.3.4. Restabilirea software

Obiectivul restabilirii după o eroare este revenirea sistemului la o stare consecventă de la una eronată, permiţând astfel sistemului să funcţioneze corespunzător. Există trei strategii de restabilire: restabilirea cu întoarcere (backward error recovery), restabilirea cu salt înainte (forward error recovery), şi reset. Tehnica de restabilire cu întoarcere, care se mai numeşte rulare înapoi şi reîncercare (rollback-and-retry), implică salvarea stării sistemului (de exemplu, salvarea conţinuturilor memoriei principale şi registrelor procesorului într-un depozit fiabil de backup) la

etape diferite în timpul execuției programului. Aşa cum se observă din figura 2.27, punctele din timpul execuției la care stările sunt salvate sunt numite puncte de restabilire (recovery points). Dacă o eroare este detectată, tehnica de restabilire cu întoarcere restabileşte sistemul la o stare înregistrată anterior şi reporneşte execuția programului. După fiecare restabilire cu întoarcere, rezultatele calculate existente după punctul de restabilire se îndepărtează, şi se repetă calcularea lor în speranța că de această dată nu va apărea eroare.

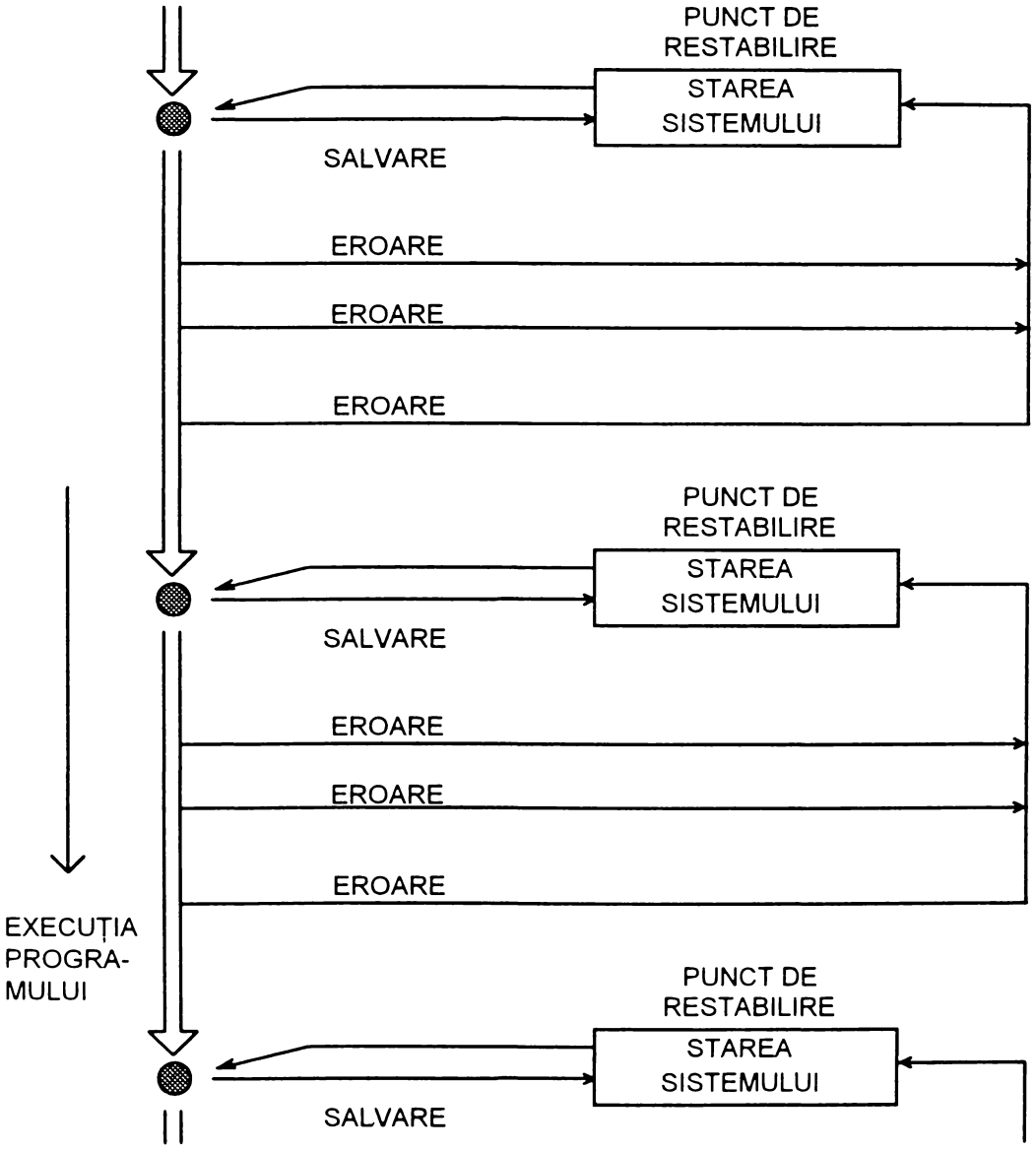


Figura 2.27 Restabilirea cu întoarcere.

Tehnica de restabilire rollback-and-retry nu este potrivită pentru defecțiuni hardware deoarece aceeaşi eroare condiționează rezultatul. Pentru ca restabilirea cu întoarcere să lucreze în această situație, rollback-ul trebuie făcut în hardware fără defecțiuni. De exemplu, în sistemul de calcul Tandem, programul este rulat din nou de perechi de procese. Unul din procese este considerat primar, cu toată execuția programului făcută în el. Celălalt proces este procesul backup, compus din punctul de restabilire şi reactualizări periodice ale procesului backup. Punctul de control (checkpoint) are datele alocate în hardware diferit de procesare. Punctele de control asigură ca

procesul backup să aibă toată informația necesară ca să preia controlul în cazul unei defecțiuni la procesul primar.

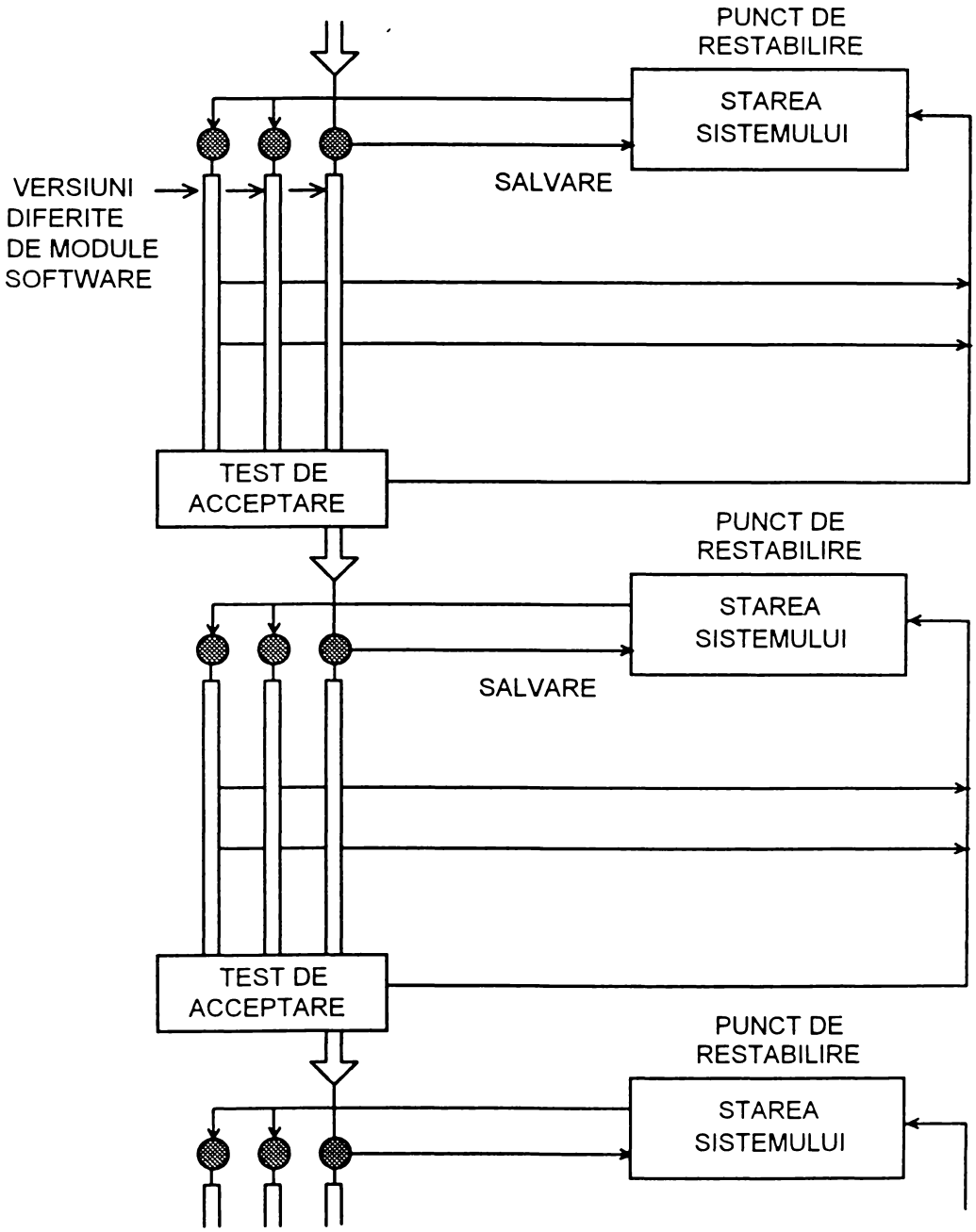


Figura 2.28 Reconfigurarea software (metoda de restabilire în bloc).

În cazul unei erori software este nevoie de o strategie diferită. Reconfigurarea software poate înfrânge problema repetării erorilor de același tip prin înlocuirea unui modul software suspect cu o versiune alternativă. Figura 2.28 arată metoda de restabilire în bloc, care este o metodă structurată de combinare a trei tehnici: utilizarea rutinelor de detecție a erorii, restabilirea înapoi și utilizarea de versiuni multiple ale modulelor software. Versiunile multiple ale modulelor software produc rezultate de calcule similare sau chiar identice. Se utilizează un test de acceptare sau validare ca și un criteriu pentru determinarea acceptabilității rezultatelor execuției modulelor software sau blocurilor obiect. Dacă testul de acceptare rezultă negativ, procesul se

întoarce la punctul de restabilire şi face o altă încercare alternativă, cu un alt bloc obiect. O proiectare imperfectă a unui bloc obiect este astfel ocolită.

În contrast, tehnicile de restabilire cu salt înainte au sistemul însuşi făcut să utilizeze mai departe starea prezentă eronată ca să obţină o altă stare (figura 2.29). Corectarea erorii înainte este dependentă de obicei, de aplicaţie. Se bazează foarte mult pe cunoaşterea naturii erorii şi pe consecinţele ei. De exemplu, un sistem de control în timp real în care un răspuns nenimerit ocazional la o intrare senzorială este tolerabil, se poate restabili prin omiterea răspunsului lui şi procesând imediat următoarele mostre de intrare. Deşi restabilirea cu salt înainte trebuie să fie proiectată special pentru fiecare sistem, ea restabileşte cu eficacitate defecţiunile, atunci când ele sunt cunoscute şi consecinţele lor sunt complet anticipate.

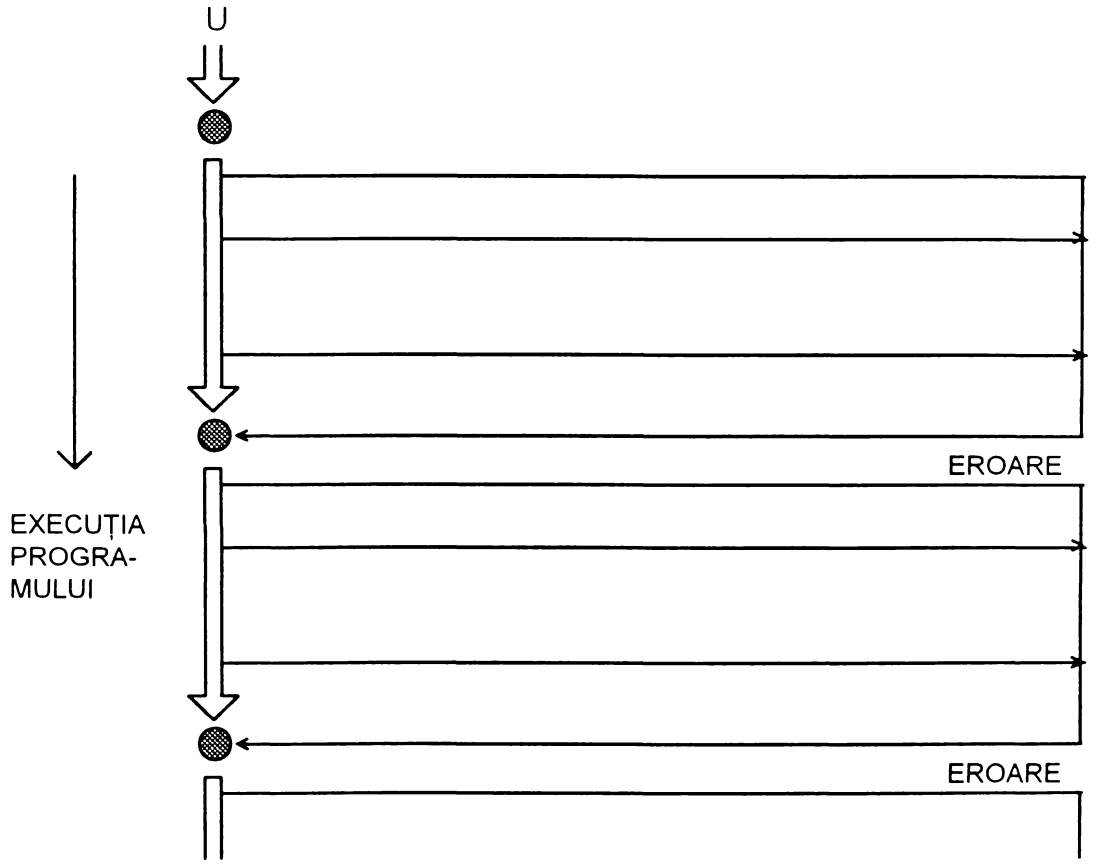


Figura 2.29 Restabilirea cu salt înainte.

Modalitatea cea mai drastică de restabilire de la o avarie neanticipată cauzată de defecţiunile sistemului este un reset fixat. Un reset este o modalitate bună de restabilire pentru că poziţionează sistemul într-o stare predefinită (fixă). Reset-urile sunt proiectate mai mult pentru a reduce efectele defecţiunilor, decât pentru a preveni apariţia lor. Un exemplu pentru aceasta este sistemul telefonic de comutare, în care câteva apeluri telefonice se pierd, iar după aceea sistemul continuă să furnizeze servicii. Figura 2.30 arată un aranjament de reset format dintr-un set de stări de iniţializare la care sistemul poate fi resetat. Nivelul actual de reset utilizat pentru o condiţie particulară eronată este selectat prin starea sistemului la apariţia erorii. Dacă eroarea este cauzată de exemplu de o defecţiune hardware, selecţia include o nouă configuraţie hardware ca să se asigure o restabilire cu succes.

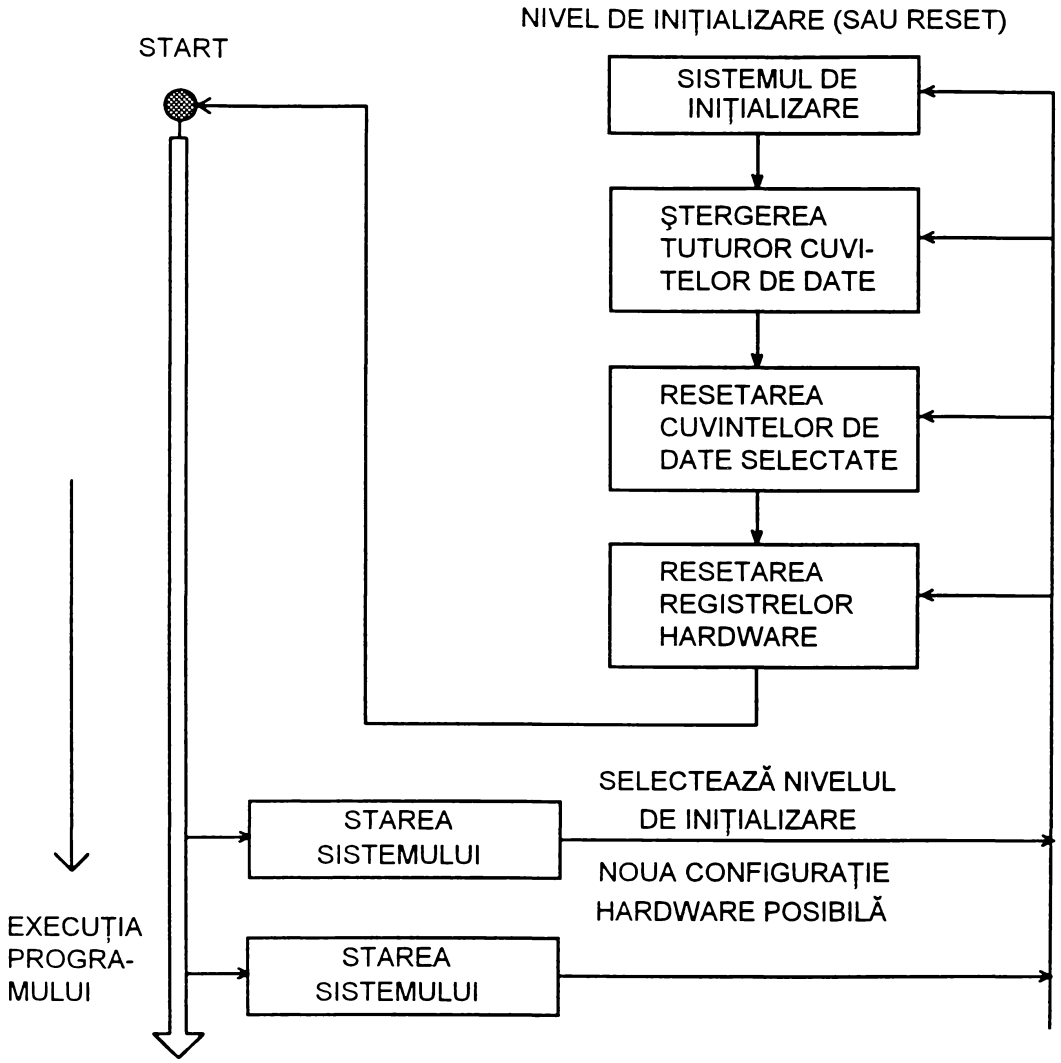


Figura 2.30 Restabilirea prin reset.

În sistemul de comutare 2ESS există şase niveluri de programe de iniţializare pentru restabilirea sistemului. Nivelurile sunt similare cu cele arătate în figura 2.30. La nivelul 1 registrele hardware sunt resetate. Nivelurile de la 2 până la 5 şterg datele de apel de la sistem şi resetează structurile de date corespunzătoare. Toate datele temporare din registrele de stocare ale apelurilor sunt şterse la nivelul 6. Se fac iniţializări succesive la nivelurile mai mari, în ordinea crescătoare, în funcţie de gravitatea erorii, până când nu mai sunt detectate erori.

2.3.5. Diagnosticarea erorii

Ţinând cont că detectarea erorii determină dacă un circuit funcţionează corect, diagnosticarea erorii localizează defecţiunea la o unitate înlocuibilă. Unitatea înlocuibilă poate fi o componentă, un circuit, sau un subsistem. Rutina de diagnosticare a erorii utilizează hardware pentru detectarea erorii şi secvenţe de test ca să ajute la localizarea unităţii defecte. Dacă detectarea erorii identifică o singură entitate ca şi sursă de eroare, diagnosticarea nu mai este necesară, din moment ce defecţiunea poate fi corectată simplu, prin înlocuirea entităţii. Dacă circuitul de

detecție este mai puțin specializat, atunci rutina de diagnosticare a erorii poate fi solicitată la izolarea mai departe a unității afectate.

Tehnica modernă de împachetare utilizând circuitele LSI şi VLSI la implementarea atât a memoriilor cât şi a procesoarelor, a încurajat utilizarea plăcilor de circuite de complexitate mare. Localizarea erorii nu este necesar să fie aşa de precisă, din moment ce procedura standard de reparare oricum înlocuieşte întreaga placă în care se află eroarea. Un procesor este tipic implementat într-unul sau două plăci de circuite. Dacă o eroare apare şi este izolată la unul din cele două plăci de procesor, repararea prin înlocuire este o cale rapidă, simplă şi convenabilă din punct de vedere economic pentru deservirea unității defecte. Costul de înlocuire a unei plăci de circuite este justificat dacă se ține cont de avantajele care reies din micşorarea timpilor de reparare şi din economisirea realizată prin faptul că nu trebuie să se dezvolte hardware şi software adițional pentru diagnosticarea erorilor.

O problemă de bază continuă să fie, atunci când este diminuată analizarea erorii din cauza utilizării plăcilor largi în implementarea unui procesor. şi anume, după ce unitatea defectă a fost înlocuită, trebuie verificată integritatea unității înlocuitoare ca să se asigure că nu este defectă şi ea. Sistemul întreg, incluzând şi unitatea înlocuitoare, trebuie să fie verificat complet ca să se asigure că eroarea a fost corectată. Aceasta de obicei înseamnă aplicarea secvenței complete de teste şi diagnosticare la sistemul care se află în reparație prin rularea programelor de autocontrol. Chiar dacă hardware-ul procesorului are incorporată logica adițională ca să acționeze ca şi un ajutor în detectarea defecțiunilor şi izolarea lor la o anumită unitate în procesor, este încă necesară o evaluare completă a hardware-ului prin programe de diagnosticare procesor.

Generarea testului pentru circuite logice digitale a fost studiat intens încă de la mijlocul anilor '60. Obiectivul acestor studii a fost de a dezvolta o procedură sistematică care permite unui proiectant să obțină un set de teste care expun toate defecțiunile posibile într-un circuit logic. Ideea de bază observată de proiectanții unor astfel de secvențe de test este de a aplica o secvență de intrare (vectori de test) la un anumit circuit logic care face ca ieşirile să difere față de cele obținute sub condiții fără erori. Identificarea secvențelor corespunzătoare de test pentru un circuit logic complex LSI poate fi o procedură foarte laborioasă, chiar şi cu utilizarea programelor de calcul sofisticate. În principiu există patru tehnici care au o vastă utilizare: tehnica activării unei căi (path sensitizing), tehnica algoritmului D, tehnica diferenței booleane, şi tehnica lui Poage.

Unul din cele mai importante aspecte ale proiectării diagnosticării în primele etape de proiectare este specificarea caracteristicilor care să fie incorporate în hardware (ca şi punctele de test, punctele de observare, punctele de control particulare, analiza semnăturii, level-sensitive scan design, etc.) pe care diagnosticarea le utilizează ca să evalueze hardware-ul. Aceste caracteristici trebuie să fie specificate şi aprobate atât de proiectantul logic cât şi de programatorul de diagnosticare, în timpul fazei de proiectare a hardware-ului.

Conceptul de întreținere centralizată şi de la distanță apare atractiv pentru multe instalații chiar dacă este mai laborios. În această modalitate, aparatura necesară pentru a realiza întreținerea sistemului este concentrată la o locație centrală. În loc de tehnicieni de expediere la locul unei maşini defecte, erorile sunt diagnosticate de la distanță. Liniile standard de telefoane sunt utilizate ca şi legături de date la facilitatea de întreținere centralizată. Sistemele de calcul ca şi VAX11/780, IBM 4300, IBM System/38, şi HP 3000 seria 33 au toate capacitatea de diagnosticare de la distanță.

2.3.6. Validarea fiabilității

Cele mai dificile sarcini ale proiectării întreținerii sunt: restabilirea sistemului şi diagnosticarea. Eficacitatea lor, în restabilirea de la o eroare şi rezolvarea diagnosticării, poate fi determinată prin simularea comportării sistemului în prezența unei anumite defecțiuni. Prin intermediul simulării, deficiențele proiectării pot fi identificate şi corectate înainte de a se da sistemul spre utilizare. Este necesar să se evalueze abilitatea sistemului la detectarea erorilor, la revenirea automată la un sistem funcțional, şi la furnizarea informației de diagnosticare (de exemplu: locația defecțiunii). Simularea defecțiunii este deci, un aspect important al proiectării întreținerii.

Există două tehnici utilizate pentru simularea defecțiunilor sistemelor digitale: simularea fizică şi simularea digitală. Simularea fizică este un proces de inserare a defecțiunilor într-un model fizic aflat în lucru. Când se compară cu simularea digitală, această metodă are o comportare mai realistică sub condiții defecte. În plus, o clasă mai mare de defecțiuni poate fi aplicată sistemului. Dar, simularea defecțiunii nu poate începe decât atunci când proiectarea a fost completă şi echipamentul este complet operațional. În afară de aceasta, nu este posibil să se introducă defecțiuni la punctele interioare ale logicii (în interiorul circuitelor integrate).

Simularea digitală a defecțiunii este un mijloc de prezicere a comportării sub deteriorare, a unui procesor modelat într-un program. Calculatorul utilizat să execute programul, numit gazdă este în general diferit de procesorul care este simulat, numit obiect. Simularea digitală a defecțiunii furnizează un grad mare de automatizare şi acces excelent la punctele interioare ale logicii, permițând proiectanților să monitorizeze curgerea semnalului urmărit. Un alt avantaj al acestei metode este faptul că permite ca evaluarea şi dezvoltarea testului de diagnosticare să se facă în avans față de fabricarea unității. Costul simulării pe calculator poate fi foarte mare pentru un sistem larg şi complex.

Simularea fizică a fost folosită prima dată de sistemul Bell pentru a genera date de diagnosticare pentru testul funcțional al sistemului ESS. Peste 50000 de defecțiuni cunoscute au fost introduse dinadins într-o unitate de control central (CC) ca să fie diagnosticate prin programul lui de diagnosticare. Rezultatele testului asociate cu fiecare defecțiune au fost înregistrate, sortate şi apoi tipărite în ordine alfabetică ca să formuleze un dicționar de diagnosticare (trouble-locating manual TLM). Prin consultarea TLM-ului a fost posibil ca personalul de întreținere să determine, sub condiții problematice, care capsule de circuite ar putea conține componenta defectă. Utilizând această tehnică de dicționar timpul mediu de reparații a fost menținut scăzut şi întreținerea a devenit mai uşoară.

Un simulator digital logic numit LAMP (Logic Analyzer for Maintenance Planning) a fost proiectat pentru dezvoltarea sistemului 1AESS. Acesta a jucat un rol important în dezvoltarea hardware a procesorului 1A şi a diagnosticărilor. Simulatorul este capabil să simuleze un subsistem având până la 65000 de porți logice. Toate defecțiunile clasice pentru porțile logice standard pot fi simulate ca şi erori logice de blocare la 0 sau la 1. Înainte ca unitățile fizice să fie disponibile, simularea digitală poate fi extrem de eficace în verificarea proiectării, în evaluarea accesului de diagnosticare, şi în dezvoltarea testelor de diagnosticare. Amândouă tehnicile de simulare a erorilor au fost integrate pentru dezvoltarea procesorului 1A ca să se profite simultan de avantajele fiecărei metode. Utilizarea simulării complementare permite defecțiunilor să fie simulate fizic (în sistemul respectiv) şi logic (pe un calculator). Multe din deficiențele fiecărei tehnici de simulare sunt compensate

corespunzător prin cealaltă tehnică. Ele se complementează una pe alta, de aici termenul de simulare complementară. Metoda complementară furnizează atât o metodă convenabilă pentru validarea rezultatelor cât şi date de simulare ale erorii mai vaste decât cele disponibile, dacă fiecare tehnică ar fi fost utilizată separat.

3. Autocontrolul ca mijloc de creştere a fiabilităţii şi disponibilităţii sistemelor de calcul

3.1. Metode de autocontrol aplicate la diferite niveluri de structură

3.1.1. Caracteristicile metodelor de proiectare structurată la nivel de bistabil

Progresele recente din tehnologia VLSI au avut o influenţă mare asupra testării sistemelor digitale (numerice). Sistemele digitale de azi sunt constituite din 100.000 până la un milion de porţi de logică aleatoare şi celule de memorie, făcând generarea testului şi simularea defecţiunilor extrem de dificilă. Chiar dacă se pot folosi maşini hardware dedicate sau supercomputers pentru a genera configuraţii de test eficace, costul face aceste soluţii inacceptabile. Pentru a se îmbunătăţi această situaţie, cercetarea s-a îndreptat spre a găsi alte metode de testare a sistemelor digitale incluzând proiectarea pentru testabilitate şi generarea de test la nivel funcţional/simulare de defecţiuni.

Dintre toate tehnicile propuse până în prezent, proiectarea pentru testabilitate (DFT, Design For Testability) este cea mai renumită, iar dintre toate tehnicile DFT cea mai obişnuită este tehnica scanării (scan design). În tehnica scanării (în [MARC93] i-se spune tehnică SCAN sau metodă SCAN), bistabilele sunt conectate într-un circuit serial şi sunt folosite ca şi terminale de I/O. Prin aplicarea tehnicii de scanare putem transforma un circuit secvenţial într-unul combinaţional, făcând astfel mai simplă generarea pattern-urilor de test pentru circuit. De asemenea, se poate partiţiona logic circuitul în câteva subcircuite şi se pot genera, independent, pattern-uri de test pentru fiecare.

Firma NEC a aplicat cu succes tehnica proiectării de scanare la sistemele de calcul comerciale încă din anii '68. Tehnica, numită cale de scanare (scan-path), a fost implementată de la nivel de capsulă la nivel de sistem şi a devenit o ustensilă puternică pentru testarea şi diagnosticarea sistemelor de calcul VLSI. Conceptul scan in/scan out, care stă la baza tehnicii de scanare (scan design), a fost introdus prima dată în anul 1964 de Carter s.a. pentru dezvoltarea testelor de localizare a defecţiunilor la sisteme IBM 360.

NEC a introdus metoda căii de scanare pentru testare şi diagnosticare atât la nivel de componente cât şi la nivel de sistem. Ei au dezvoltat sistemul de testare funcţională a plăcilor de circuit imprimat folosind calea de scanare în anul 1970, iar în anul 1971 au dezvoltat un sistem automat pentru localizarea defecţiunilor la sistemele comerciale mari de calcul .

NEC a dezvoltat calea de scanare pentru a descurca problemele din anii '70, cum ar fi utilizarea intensivă a MSI/LSI din sistemele comerciale de calcul , precum şi numărul mare de porţi aflate pe plăcile acestor sisteme. Pentru sistemele mari, localizarea defecţiunii a fost o problemă deoarece multe registre nu erau observabile. De asemenea, a fost foarte dificilă generarea automată a pattern-urilor de test.

Metoda căii de scanare pare să fie o soluţie promiţătoare pentru rezolvarea acestor probleme, deoarece permite mai multă controlabilitate şi observabilitate asupra circuitului de testat. În plus, se pot testa atât circuitele combinaţionale cât şi cele secvenţiale. Circuitul poate fi partiţionat logic în câteva subcircuite, iar programele de test pot fi generate pentru fiecare subcircuit independent. Costul total al întregului circuit scade la α^2/n, unde n este numărul de subcircuite, iar α este rata de suprapunere [FUNA89] Tabelul 3.1 arată o comparaţie mărime-cost.

Conceptele de controlabilitate şi observabilitate menționate anterior se definesc în felul următor [LALA85], [PRAD86], [JOHN89]:

- Controlabilitatea arată cât de uşor se poate produce un semnal arbitrar valid la intrările unei componente (subcircuit) prin excitarea intrărilor primare ale circuitului. De fapt, reprezintă abilitatea de a comanda (conceptul putea fi numit în limba română "comandabilitate") un semnal doar prin intermediul intrărilor primare. O linie care poate fi poziționată pe "1" logic se numeşte 1-controlabilă, una care poate fi poziționată pe "0" logic se numeşte 0-controlabilă, iar una care poate fi poziționată pe ambele stări logice se numeşte complet controlabilă.
- Observabilitate arată cât de uşor se poate determina la ieşirile primare ale circuitului ce se întâmplă la ieşirile unei componente (subcircuit). Cu alte cuvinte, este abilitatea activării unei căi de la semnalul cercetat până la un punct măsurabil.

Prin componente se înțeleg fie circuite integrate standard (SSI şi MSI) pentru circuite la nivel de placă, fie celule standard de module de bibliotecă pentru circuite LSI şi VLSI. De asemenea se presupune că componentele sunt interconectate cu legături unidirecționale.

	Mărime	Cost
Circuit original	1	1
Subcircuit	α / n	$(\alpha / n)^2$
Circuit expandat	α	α^2 / n

Tabelul 3.1. Comparația mărime-cost dintre un circuit nepartiționat şi circuitele partiționate utilizând metoda căi de scanare

Evident că α depinde de n, de aceea proiectanții trebuie să decidă cu atenție câte subcircuite sunt necesare. Pentru un circuit bine proiectat, care este potrivit pentru partiționare, valoarea lui α este sub 1,7. Calea de scanare poate facilita localizarea defecțiunii pentru că procesorul de diagnosticare poate să urmărească uşor starea internă a sistemului. Hardware-ul adițional este foarte puțin (de la 4% la 10%), şi este potrivit pentru un calculator VLSI deoarece există relativ puțini pini de I/O adiționali.

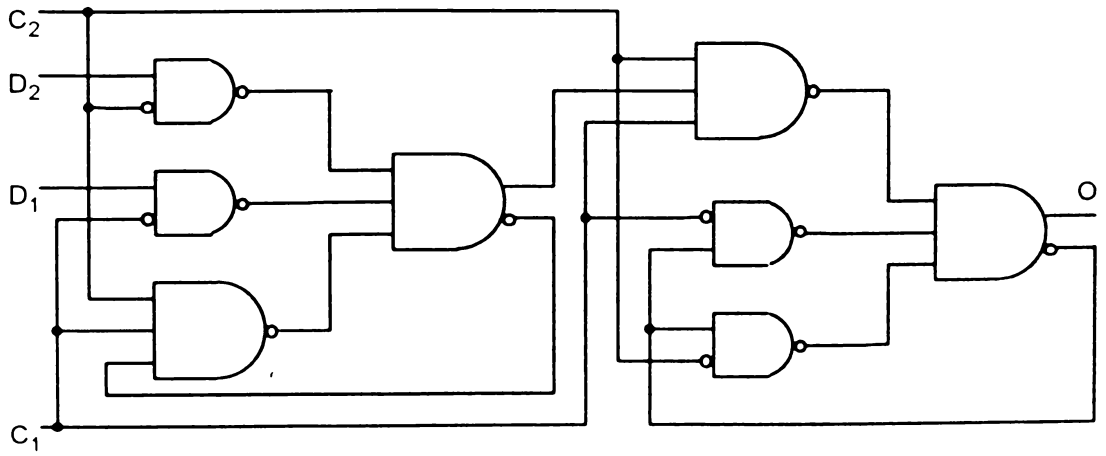


Figura 3.1 Bistabilul transformat este elementul de bază al căii de scanare

Figurile 3.1 şi 3.2 arată prima implementare a căii de scanare la sistemele de calcul comerciale. Figura 3.1 arată un bistabil transformat cu cale de scanare utilizat ca şi circuitul de bază. Acest bistabil include două tipuri de semnale de clock, C_1 pentru operaţia normală şi C_2 pentru operaţia de deplasare. Figura 3.2 arată configuraţia căii de scanare a unei plăci logice, în care bistabilele sunt conectate în serie printr-o cale de scanare. Aceste bistabile operează ca un registru de deplasare serial utilizând Clock II, intrarea de test (scan in), şi ieşirea de test (scan out). Utilizând o adresă de selecţie a plăcii logice, putem selecta o anumită placă dintr-o unitate logică care are multe tipuri de plăci logice.

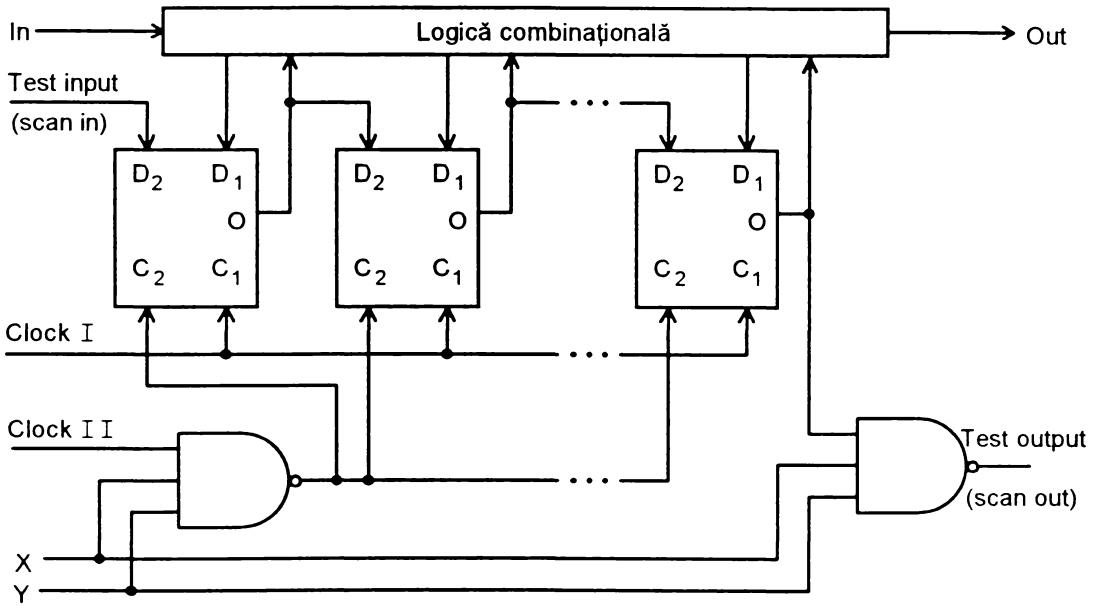


Figura 3.2 Configuraţia calea de scanare pe o placă logică.

În anul 1971, NEC a utilizat calea de scanare pentru a localiza defecţiuni la nivel de sistem pentru sistemele de calcul seriile 2200 modelul 700. Această aplicaţie, care include generarea automată a configuraţiilor de test, a fost prima utilizare intensivă a tehnicii de scanare în testarea sistemelor de la nivel de componentă la nivel de sistem.

Figura 3.3 arată o altă implementare a căii de scanare. Acest circuit utilizează un singur clock şi intrări multiplexate la bistabil. Circuitul are două moduri de operare (deplasare). Atunci când intrarea "Shift mode" (mod de deplasare) este în starea high, circuitul se află în modul normal. Iar atunci când intrarea modului de deplasare este în starea low, bistabilele din circuit operează ca un registru de deplasare serial

Chiar dacă tehnica scanării (scan design) este o tehnică DFT puternică, pentru testarea logicii aleatoare, aplicarea ei poate pune probleme în diferite tipuri de circuite. Un astfel de tip sunt şirurile de memorie incorporate în circuite de logică aleatoare. Aplicarea directă a tehnicii de scanare la fiecare celulă de memorie din şir costă foarte mult. O soluţie posibilă este aplicarea tehnicii de scanare la un anumit cuvânt din şirul de memorie şi accesarea cuvântului fixat în timpul testării circuitului. Oricum, din moment ce, în timpul testării memoria funcţionează ca un registru, ea trebuie testată utilizând altă procedură.

Tehnica scanării s-a dovedit foarte eficace pentru proiectarea circuitelor bipolare, dar este greu de aplicat la un circuit MOS, mai ales la un circuit MOS

dinamic. Dificultatea apare în primul rând din cauza hardware-ului adițional. Un bistabil MOS dinamic este limitat la câteva tranzistoare, astfel, supraîncărcarea de hardware adițional este mare pentru un circuit VLSI MOS.

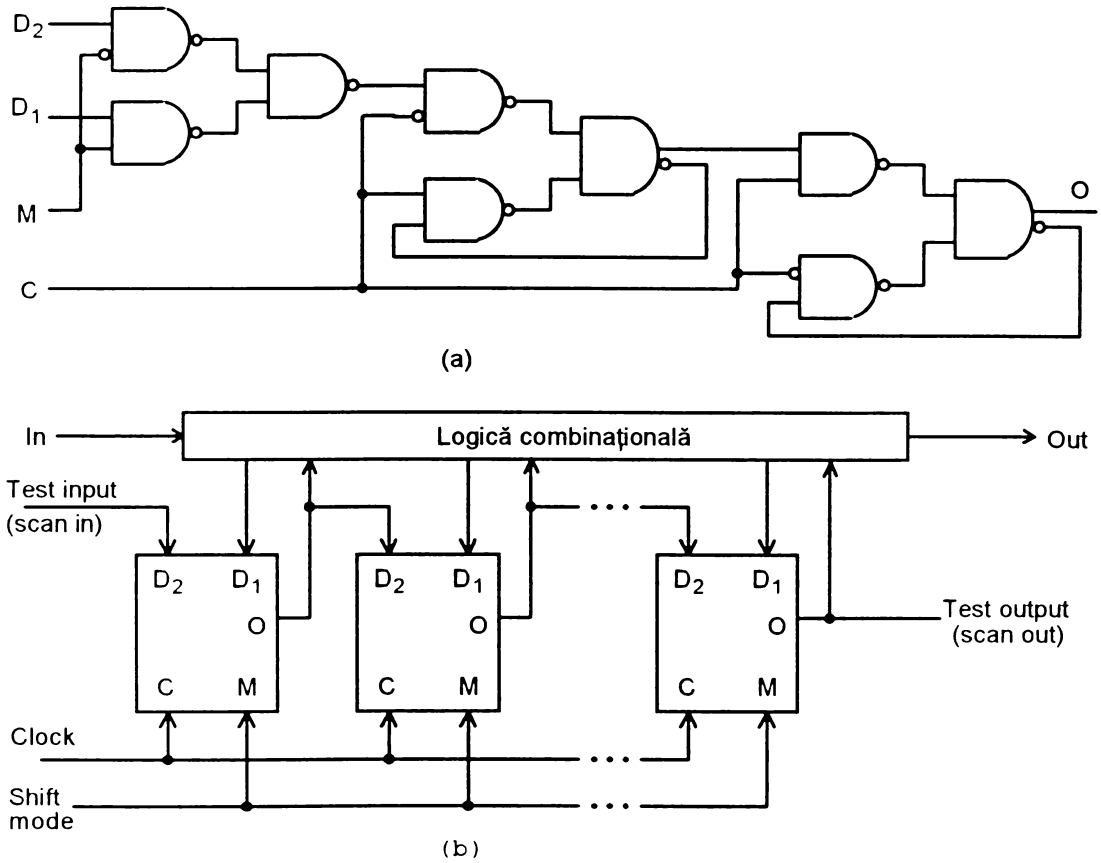


Figura 3.3 Configurația căii de scanare implementată cu un singur Clock.

O altă problemă este testarea rapidă. Operația de scan in/scan out necesară în timpul testării circuitului face ca viteza per total să fie mult mai scăzută decât cea a operațiilor normale de circuit. Incorporarea unor scheme de autotest (built-in self-test) este o cale posibilă pentru execuția testării în timp real [FUNA89].

În 1977, IBM introduce tehnica LSSD (Level Sensitive Scan Design) în cadrul căreia se utilizează tehnica scanării, dar în plus se impune ca toate schimbările de stare sa fie controlate de nivelul semnalului de ceas şi nu de front (Level Sensitive Design) [RUSS85], [LALA85], [YARM90]. Această abordare reduce dependența funcționării de timpii de propagare, eliminând cursele sau hazardurile. Celula de bază este elementul de memorare al informației, dependent de nivel, care asigură şi tratarea semnalului de scan în modul test, element numit SRL (Shift Register Latch).

3.1.2. Caracteristicile metodelor de proiectare structurată la nivel de registru

Deşi s-au scris foarte multe despre standardul de test "boundary scan" (IEEE 1149.1 Standard Test Acces Port and Boundary-Scan Architecture), doar câteva dispozitive au implementată (incorporată) această arhitectură. Acordarea completă cu acest standard, ceea ce înseamnă că proiectanții ar putea folosi oricare dispozitiv, cu

asigurarea că este compatibil cu boundary-scan, va apărea numai după câțiva ani [DONN91]. Până atunci plăcile cu cablaj imprimat vor fi hibride conținând circuite (dispozitive) scanabile (după standardul IEEE) şi dispozitive nescanabile. Asemenea plăci necesită metodologii de testare care se bazează pe ambele tehnici, pat de cuie (bed of nails) şi boundary-scan.

Testarea cu boundary-scan poate micşora drastic timpul de dezvoltare al sistemelor complexe. Ken Parker estimează că dezvoltarea vectorilor de test pentru testarea adecvată a funcțiilor de I/O ale unui dispozitiv complex de talia lui Motorola 68040, ar consuma 6 luni de zile din timpul unui inginer. În schimb, vectorii de test pentru acelaşi dispozitiv se pot genera în două minute, în ipoteza că are capabilitate boundary-scan şi că se dispune de ustensile software corespunzătoare.

De asemenea boundary-scan promite posibilitatea testării sistemelor care utilizează componente cu montare la suprafață de mare densitate (high-density surface-mounted components) şi plăci multistrat complexe (complex multilayer pc boards). Astăzi, aceste sisteme se pot testa cu tehnica patului de cuie, dar cu o mulțime de probleme din cauza geometriei micşorate a pinilor care se impune prin tehnologia de montare la suprafață (surface-mounting). Şi în acest caz, având ustensile potrivite şi dispozitive care să susțină boundary-scan, toată placa poate fi testată complet folosind numai calea boundary-scan [DONN91], [BLEE91].

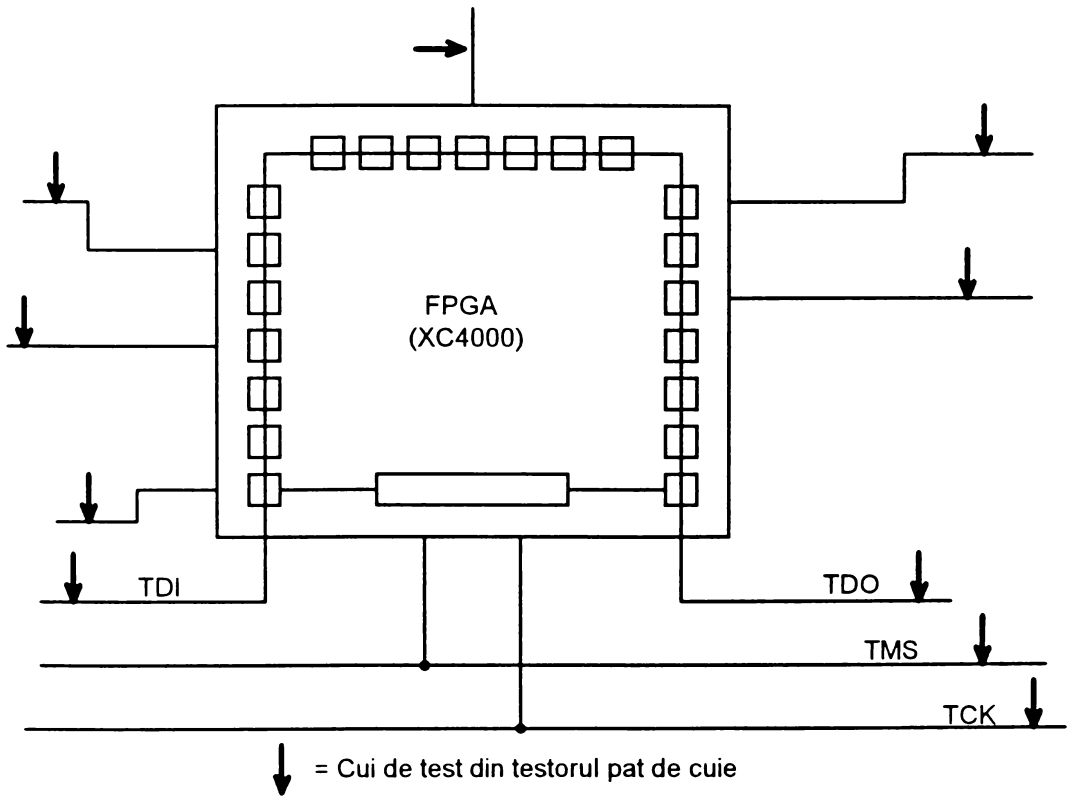


Figura 3.4 Dispozitivul scanabil Field-Programmable Gate Array (FPGA).

Pe de altă parte, patul de cuie sau în general metodele de test în circuit (in-circuit) oferă câteva avantaje importante care nu se întâlnesc în boundary-scan. Un astfel de avantaj cheie al testării în circuit, este abilitatea diagnosticării deteriorărilor multiple la o singură trecere prin test.

În plus, detectarea scurtcircuitelor poate fi realizată înainte de punerea sub tensiune a plăcii, reducând posibilitatea deteriorării catastrofale din cauza

scurtcircuitelor între pini şi tensiunea de alimentare. De asemenea, testele în circuit (in-circuit testing) permit realizarea testării parametrice simultan cu cea digitală [PARK89].

Boundary-scan, în general, nu se referă la testarea dispozitivelor analogice sau pasive. Până când producătorii de semiconductoare nu oferă o gamă largă de dispozitive cu boundary-scan, plăcile o să se testeze utilizând testarea tradiţională în circuit, aplicarea tehnicilor boundary-scan făcându-se numai unde este necesar. Contrar impresiei răspândite larg, boundary-scan şi testarea în circuit nu sunt incompatibile. Dimpotrivă, folosind o combinaţie de boundary-scan şi testare în circuit se poate simplifica semnificativ testarea plăcilor (Figura 3.4).

Mai întâi să presupunem un sistem numai cu un singur circuit scanabil. Presupunând că sistemul este destul de complicat şi că are un număr mare de pini, este uşoară aplicarea unei tehnici hibride de testare prin boundary scan şi testare în circuit. Luând, de exemplu, testarea dispozitivelor cu grad de integrare mare, ca şi Xilinx XC4005 (field-programmable gate array FPGA).

Aceste dispozitive deseori susţin o bună parte din logica sistemului. Testarea rapidă a unui astfel de dispozitiv este dificilă în comparaţie cu testarea individuală a componentelor SSI şi MSI care realizau această funcţie iniţial. În dispozitivele cu grad mare de integrare, nu se mai pot accesa nodurile interne ale dispozitivului. Astfel, se pune problema determinării stimulilor care aplicându-se la intrările dispozitivului vor forţa ieşirile la o stare cunoscută. Dacă dispozitivul are incorporat boundary-scan, datele se pot scana în registrul boundary-scan şi se poziţionează astfel fiecare pin de I/O la o stare cunoscută, fără a se ţine cont de funcţionalitatea internă a circuitului de testat. Bineînţeles că acest lucru simplifică problema testării.

Dispozitivul XC4005 FPGA este primul de la o familie de circuite compatibile cu standardul boundary-scan. Fiecare pin de I/O al acestui dispozitiv poate fi complet controlat şi urmărit (observat) prin utilizarea unor pattern-uri de date introduse serial în registrul de boundary-scan al lui XC4005 prin pinul TDI (Test Data Input) şi controlarea prin pinul TMS (Test Mode Select) şi TCK (Test Clock) provenit din canalele ATE (Automatic Test Equipment). Acest lucru este echivalent cu stimularea complexă a intrărilor pentru a obţine aceeaşi acoperire de test fără boundary scan.

3.1.2.1. Conceptul cuiului de siliciu

A doua generaţie a plăcilor cu cablaj imprimat va fi hibridă conţinând dispozitive scanabile şi nescanabile. Opţiunile de testare cresc odată cu apariţia mai multor dispozitive scanabile pentru care capabilitatea de boundary-scan permite testarea căii dintre două dispozitive scanabile fără utilizarea unui cui fizic aşa cum se arată în figura 3.5.

Pentru căile dintre două dispozitive scanabile, proiectanţii pot folosi strategii de test care utilizează conceptul cuiului de siliciu, cu care proiectanţii verifică semnalele prin captarea lor în registrele de boundary-scan, urmată de deplasarea acestor date în afară, prin pinul TDO (Test Data Output).

Pentru majoritatea plăcilor de tehnologie hibridă, existenţa accesului fizic în câteva din reţelele plăcii este cel mai critic factor în determinarea economiei metodelor de testare. Atunci când se hotărăşte care reţele vor avea acces fizic, proiectantul, în timpul amplasării pe placă, trebuie să aibă în vedere o anumită strategie de test. Din acest motiv, trebuie ca proiectanţii de circuite să comunice celor care realizează

amplasarea pe placă, informații legate de accesele virtuale realizate de cuiele de siliciu pe placă şi necesitățile de acces impuse de implementarea strategiei de test aplicate.

În prezent, poate nu există nici o placă care să conțină numai circuite scanabile. Teoretic, o astfel de placă ar putea fi testată complet prin intermediul căii de scanare, utilizând numai acele semnale prezente la cuplă (edge connector) şi la cuiele de siliciu.

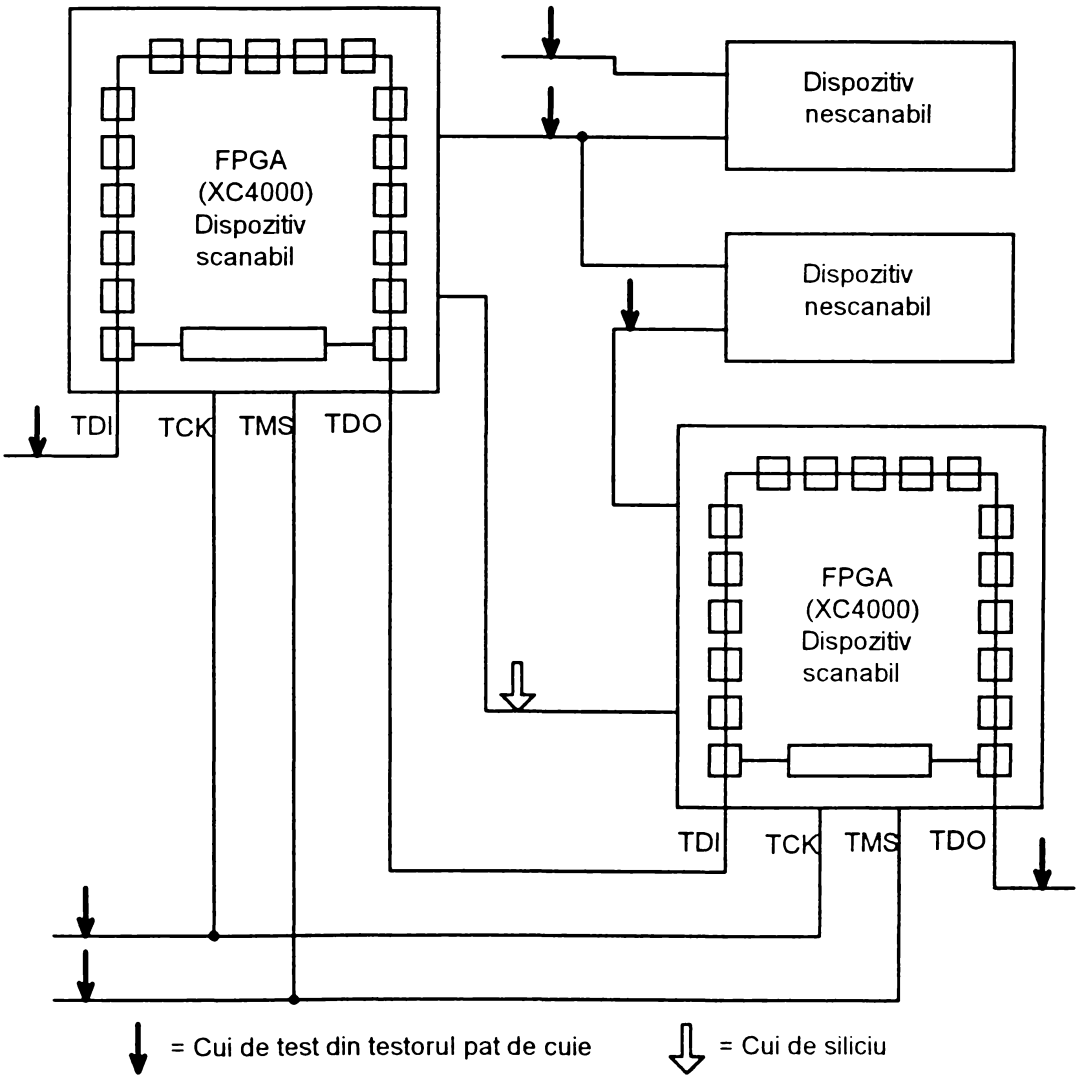


Figura 3.5 Cuiele de siliciu pot înlocui cuiele fizice pentru căile dintre două dispozitive scanabile.

Prin calea de scanare, un testor poate accesa traseele individuale ale dispozitivelor care servesc ca şi cuie de siliciu, oferind abilitatea controlării şi observării acestor noduri ale circuitului. Aceste noduri sunt cele care altfel trebuia să fie accesate prin intermediul cuielor fizice conectate pe canalele ATE-ului.

Prin utilizarea cuielor de siliciu nu mai este necesară coordonarea (backdriving) funcțiilor de I/O ale dispozitivelor; numărul canalelor de test este redus, eliminându-se complexitatea şi costul testării efectuate cu ajutorul cuielor fizice. De asemenea se simplifică generarea vectorilor de test. Atunci când se utilizează boundary-scan, programele de test pot efectiv să ignore complexitatea funcțională a unui dispozitiv şi să genereze vectori de scanare care să testeze atât interconexiunile în

întregime, cât şi integritatea lipiturilor şi structura de I/O a dispozitivului. În plus, o dată ce s-a definit amplasarea pe placă, dezvoltarea testelor poate să înceapă independent de modificările minore de logică care vor apărea inevitabil în faza de verificare a proiectului.

Dar nici boundary-scan nu este fără limitări. La o placă fără accese fizice pentru canalele ATE, foarte puține lucruri se pot determina fără să se alimenteze placa. Odată ce s-a alimentat placa ajung câteva secunde ca să se determine dacă calea de scanare este funcțională sau nu.

Pe durata acestui interval de timp, s-ar putea să apară condiții de curenți excesivi, din cauza scurtcircuitelor între pini şi masă sau tensiune de alimentare. Presupunând că nu se ajunge la deteriorarea acestor părți, inginerul specialist la testare trebuie să decidă dacă aceste părți se pot folosi în continuare în sistem sau dacă au fost stresate în exces şi trebuie înlocuite.

Dacă nu există deteriorări catastrofale după alimentarea plăcii, primul pas este controlul stării portului. Cea mai simplă formă a acestui control constă în inserarea datelor în calea boundary-scan şi verificarea acelor date din pinul Test Data Output (TDO) la nivelul plăcii. Din cauză că registrul de scanare al fiecărui dispozitiv vine în stare de reset, se aşteaptă un şir de zerouri de lungime cunoscută la ieşirea TDO.

Dacă nu se întâmplă acest lucru, înseamnă că există probleme cu calea de scanare şi placa trebuie să se trimită înapoi la analiză şi reparare. Nu se pot face alte teste în placă până când nu se înlătură toate defectele care influențează integritatea căii boundary-scan. Dacă portul are multe probleme, atunci sunt necesare mai multe treceri prin faza de testare, ceea ce bineînțeles este nedorit şi este cunoscut în literatura de specialitate sub numele de looping.

3.1.2.2. Evitarea hazard-urilor posibile în timpul testării plăcilor hibride

În general, componentele analogice sunt nepotrivite pentru a se testa cu boundary-scan. Astfel, plăcile echipate cu astfel de componente necesită accese fizice dacă se doreşte aplicarea metodei de testare în circuit pentru circuite analogice. Cu toate că hibridizarea dintre dispozitivele scanabile şi cele nescanabile nu este o problemă, trebuie acordată atenție specială la evitarea hazardurilor provocate de anumite caracteristici ale tehnicii boundary-scan. Un exemplu este în cazul în care o bancă de memorie se selectează printr-un decodificator standard 3 la 8 (figura 3.6). În timpul funcționării normale a sistemului, numai o singură ieşire a decodificatorului poate fi activă la un moment dat. Astfel, numai o singură bancă este selectată şi conduce magistrala. Dacă presupunem că decodificatorul este un dispozitiv scanabil, atunci, oricare sau chiar toate ieşirile lui pot fi poziționate pe starea activă simultan. Acest lucru creează direct conflictul ieşirilor memoriilor pe magistrală.

Una dintre soluții este utilizarea dispozitivelor de memorie scanabile. Această soluție, totuşi, pare să fie insuficientă.

O altă soluție se bazează pe faptul că majoritatea memoriilor au pe lângă semnalul Chip Select (CS) şi semnalul Output Enable (OE). În multe situații sunt cuplate împreună în cadrul sistemului. Separând aceste două semnale şi folosind un cui fizic pentru a controla unul din semnalele de selecție al memoriei s-ar putea asigura că nu se vor activa două bănci de memorie în acelaşi timp.

O altă problemă posibilă în plăcile hibride este apariția unui scurtcircuit la un dispozitiv nescanabil în calea dintre două dispozitive scanabile (figura 3.7). Dacă ieşirea unui dispozitiv nescanabil este blocată la "0" logic sau la "1" logic sau, şi mai

rău, oscilează, datele recepționate la pinul TDO vor fi incorecte sau nedeterminate. Acest lucru ar putea deruta testorul, făcându-l să indice că problema ar fi provenit de la dispozitivele scanabile. O soluție ar fi poziționarea unui cui fizic pe acest semnal forțându-l într-o stare cunoscută. În această situație vor fi cel puțin determinate datele recepționate din TDO. Desigur, mai există multe alte capcane în hibridizarea dispozitivelor scanabile cu cele nescanabile, precum şi multe alte metodologii de test.

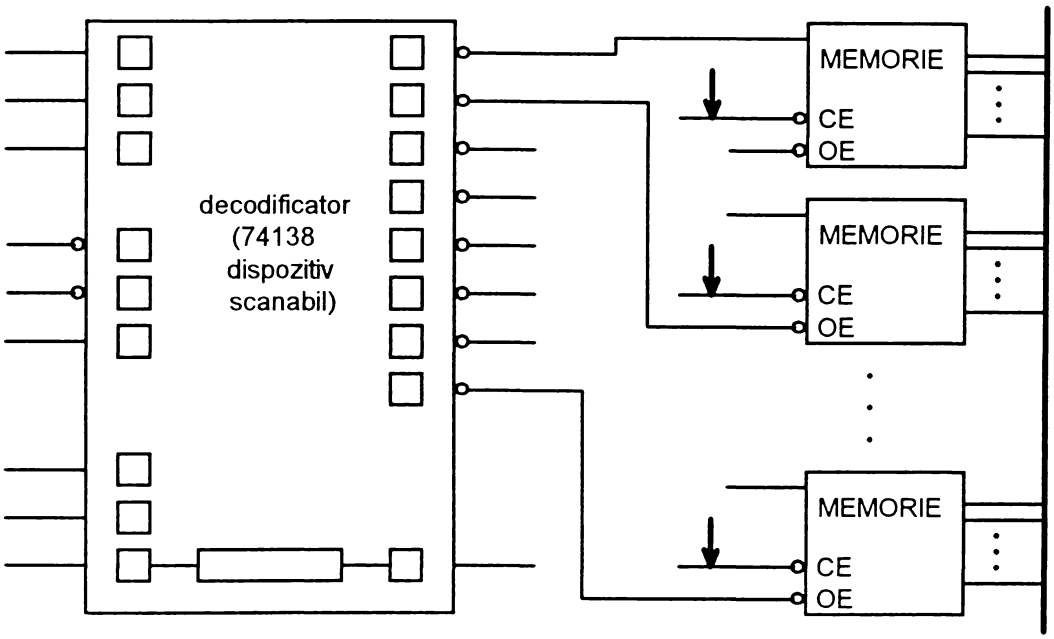


Figura 3.6 Un exemplu de hazard în cazul plăcilor hibride (care conțin dispozitive scanabile şi nescanabile)

O altă limitare este lipsa de pachete software, aplicabile universal, pentru generarea automată a vectorilor de test pentru dispozitive scanabile. Diferiți producători au definit cu succes descrierile dispozitivelor lor particulare şi au scris software-ul pentru generarea vectorilor de test ai acestor dispozitive. Pentru a se evita dezavantajele provenite din faptul că fiecare producător îşi furniza descrierea şi pachetele de programe proprii, IEEE-ul a examinat propunerea întocmită de Hewlett Packard numită BSDL (Boundary-Scan Description Language).

Standardul IEEE 1149.1 prezintă trei variante de interconectări la nivel de placă. Acestea includ conectarea în serie a dispozitivelor scanabile folosind un semnal TMS (Test Mode Select), conectarea în două lanțuri paralele a dispozitivelor scanabile, sau în căi multiple independente la care sunt comune semnalele TMS şi TCK (Test Clock). Alegerea arhitecturii de interconectare trebuie să se bazeze mai întâi pe software-ul suportat de ATE. Elaborarea schemelor de interconectare care nu sunt suportate de software, pur şi simplu mai adaugă probleme de test.

Din cauza naturii seriale a testării cu boundary-scan, cineva poate că se gândeşte că împărțind un lanț lung care conține 10000 de vectori în două căi, câte 5000 de vectori fiecare, ar putea micşora timpul de test. Dimpotrivă, o analiză mai atentă arată că factorul de limitare a micşorării timpilor de test nu este aplicarea vectorilor ci timpul de acces necesar al discului din testor.

Să luăm ca exemplu inserarea a 10000 de vectori într-un lanț de scan cu modesta rată de introducere de 1 MHz. Acest lucru are nevoie de numai 10 ms, timp incomparabil mai mic decât secundele necesare pentru încărcarea programului de test.

Ca rezultat, împărțirea lanțului de scanare în două jumătăți ar putea efectiv să dubleze timpul de test în loc de a-l micşora.

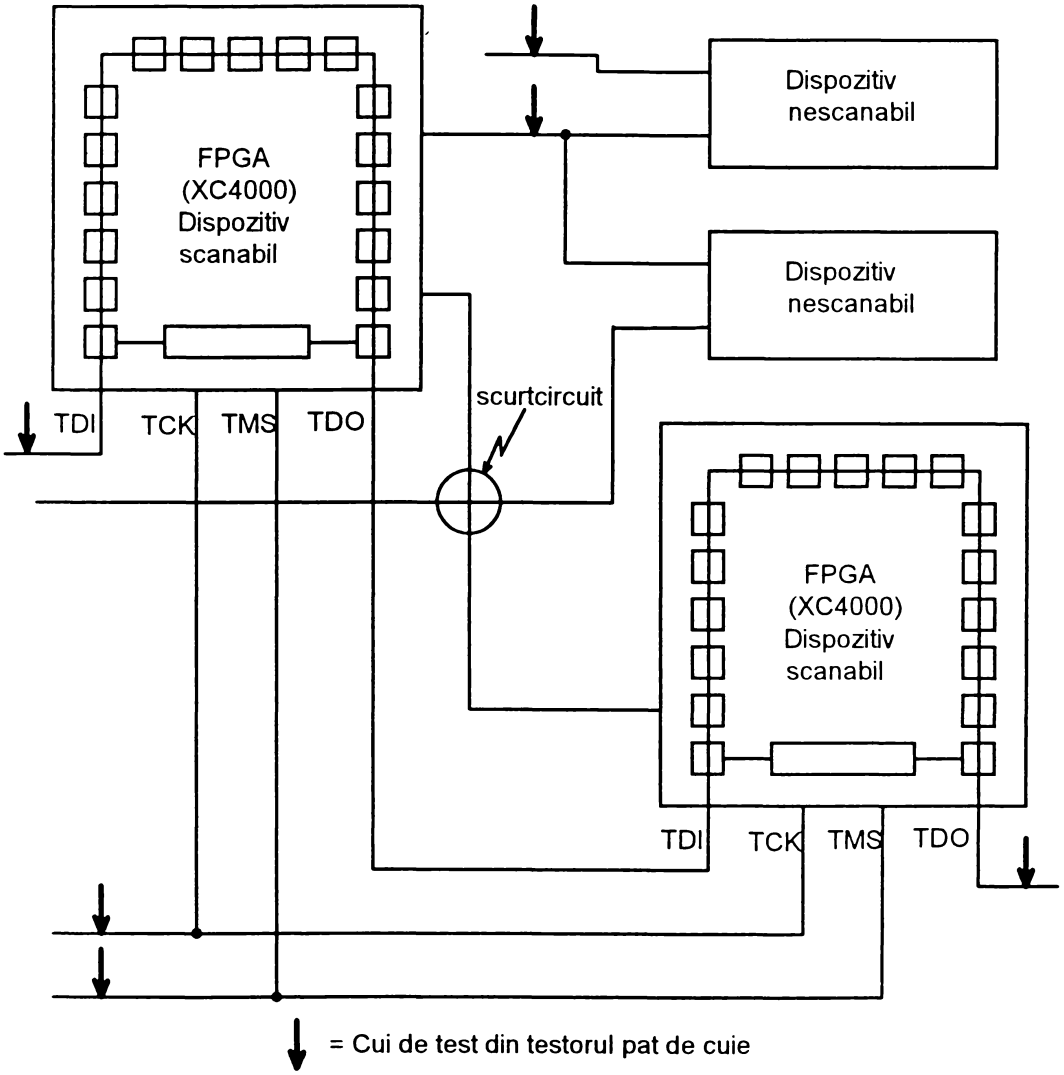


Figura 3.7 Scurtcircuitul între o ieşire a unui dispozitiv nescanabil şi calea dintre două dispozitive scanabile este o situație nefavorabilă dar posibilă în cazul plăcilor hibride.

3.1.3. Proiectarea structurată la nivel de bloc

Incorporarea facilităților de test (BIT, Built-In Test) poate micşora considerabil timpii de întreținere şi costurile pentru sistemele electronice complexe, prin accelerarea detecției şi izolării defecțiunilor. Din acest motiv, sistemul este mult mai disponibil iar reparațiile sunt mai ieftine. La nivel de placă, testul incorporat (BIT) se realizează de obicei prin aplicarea unui set de pattern-uri pseudoaleatoare, pentru a se verifica existența defecțiunilor, şi prin comprimarea informației răspunsului de test pentru a se obține o semnătură. Aceste tehnici sunt eficace pentru plăcile utilizate pentru scopuri generale atâta timp cât seturile de chip-uri care se testează sunt testabile la testarea pseudoaleatoare şi la comprimarea informațiilor răspunsurilor de test, dând rar semnături identice pentru plăci bune şi defecte.

Notă: În [D&'T89], McCluskey precizează că nu este recomandată utilizarea expresiei "test-response compression" ca un sinonim al expresiei "test-response compaction", pentru că prin "compaction" (compactare) se pierde din informație, în timp ce prin "compression" (comprimare) se elimină doar redundanța fără pierderea de informație. În majoritatea literaturii consultate [LALA85], [JOHN89], [YARM90], [FUJI90a], nu este respectată această opinie. Din această cauză, precum şi pe motiv că în limba română când este vorba de reducerea volumului secvențelor binare răspuns este consacrat termenul de comprimare [VLAD82b], [GEBE84], [VLAD89], în continuare se va folosi cuvântul "comprimare".

3.1.3.1. Autocontrolul bazat pe principiul BILBO

Tehnica de determinare a stării unui chip sau a unei plăci utilizând informația comprimată a răspunsului de test se numeşte analiza semnăturii. În general, metodele de analiză a semnăturii pentru testarea plăcilor utilizează circuite cu observatori logici de bloc incorporați (BILBO, Built-In-Logic Block Observers) pentru a genera pattern-uri pseudoaleatoare şi pentru a comprima răspunsurile. Deci, BILBO este o schema de generare a testului incorporată care utilizează în conjuncție scan design-ul cu analiza de semnături. Componenta de bază a acestei tehnici de autotest este un registru de deplasare numit registru BILBO care are 4 moduri de funcționare determinate de valoarea liniilor de control B_1 şi B_2 (figura 3.8).

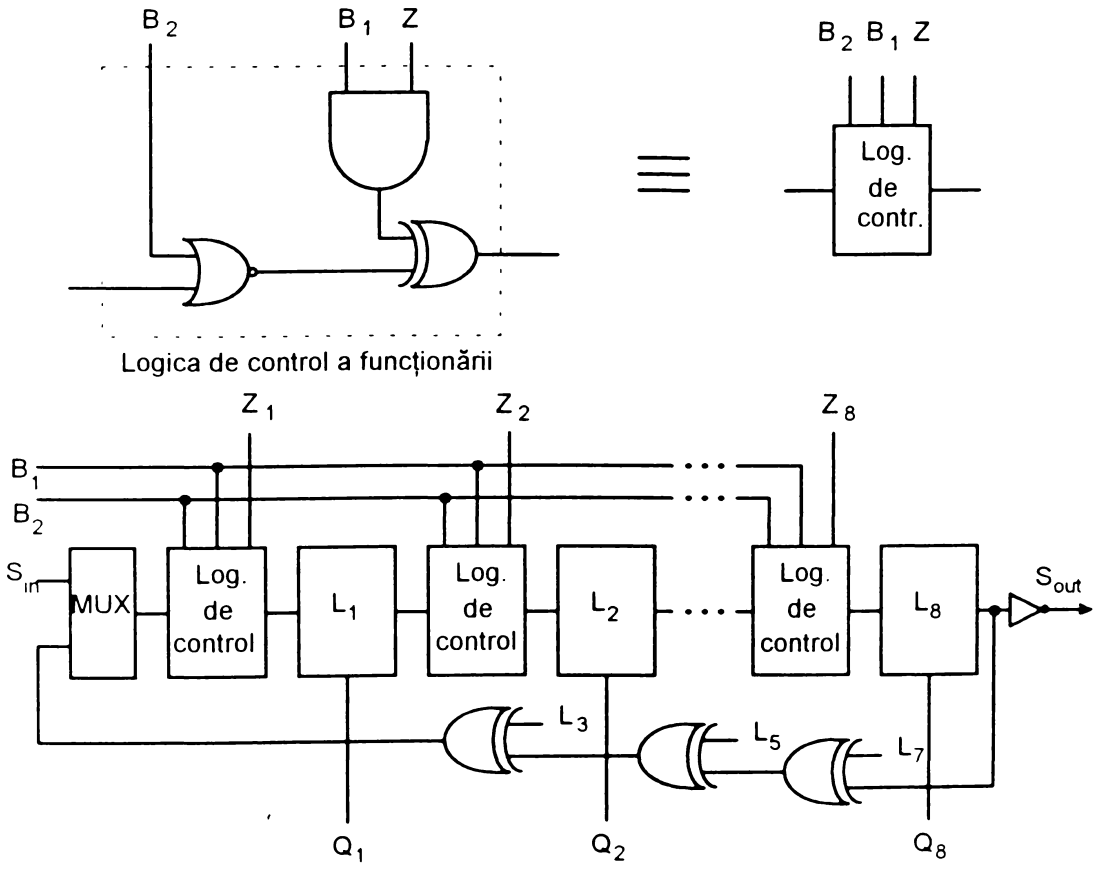


Figura 3.8 Registrul BILBO.

În practică, bistabilele utilizate la BILBO sunt bistabile normale ale sistemului, unde intrările Z sunt ieşirile de la blocul combinaţional predecesor iar ieşirile Q sunt intrări la blocul combinaţional succesor. Blocurile BILBO partiţionează sistemul pe motivul uşurării testării.

Liniile de control B_1 şi B_2 controlează modul de funcţionare al registrului BILBO în felul următor:

(1) $B_1 = B_2 = 0$: BILBO este configurat într-un registru de deplasare lung formând astfel calea de scanare (scan-path).

(2) $B_1 = B_2 = 1$: bistabilele din BILBO funcţionează ca un set de bistabile al sistemului, adică intrările Z apar la ieşirile Q pentru operaţia normală a circuitului.

(3) $B_1 = 1$, $B_2 = 0$: registrul BILBO este configurat într-un LFSR (Linear Feedback Shift-Register) formând analizorul de semnături, captând datele în paralel.

(4) $B_1 = 0$, $B_2 = 1$: În această situaţie bistabilele registrului BILBO sunt resetate.

Procedeul de testare utilizând BILBO este următorul: un registru BILBO funcţionează ca generator de vectori stimuli test pseudoaleatoare care le aplică blocului combinaţional de testat; iar un alt BILBO este utilizat ca registru analizor de semnături, care captează răspunsurile provenite de la circuitul testat. În felul acesta conţine după N tacte o semnătură în funcţie de starea circuitului (faulty sau fault-free). Registrul BILBO care conţine semnătura se reconfigurează ca să formeze un registru de deplasare şi să trimită spre exterior (scan out) semnătura spre analiză. După aceasta se inversează rolurile astfel încât să se testeze următorul block de circuit combinaţional. Figura 3.9 ilustrează acest procedeu.

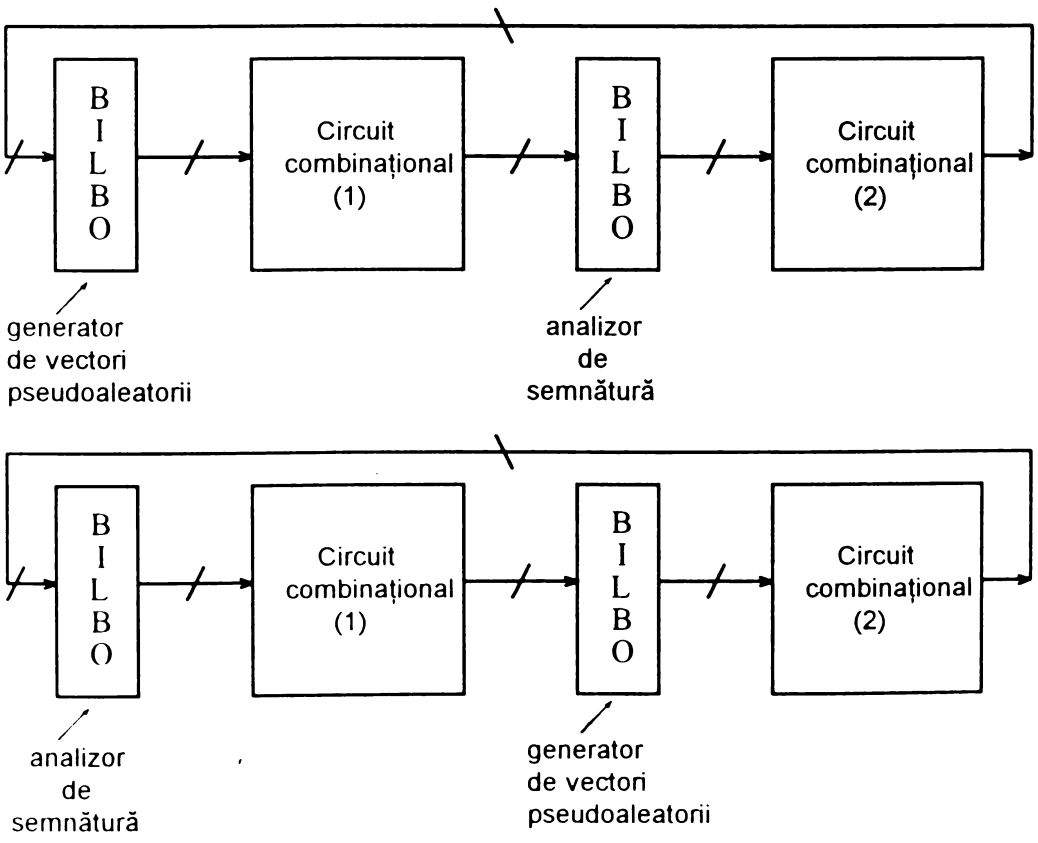


Figura 3.9 Testarea utilizând registrele BILBO

Această tehnică elimină nevoia generării şi aplicării pattern-urilor de test din exterior. Totuşi este nevoie de simularea defecţiunilor pentru a se determina acoperirea de defecţiuni, realizată de secvenţe de vectori pseudoaleatorii; Mai mult decât atât circuitul trebuie să fie simulat ca să se obţină valorile semnăturilor în cazul funcţionării corecte. Cantitatea datelor de test care trebuie stocată şi analizată este redusă din moment ce răspunsul circuitului la N pattern-uri de test este comprimat la un singur cuvânt.

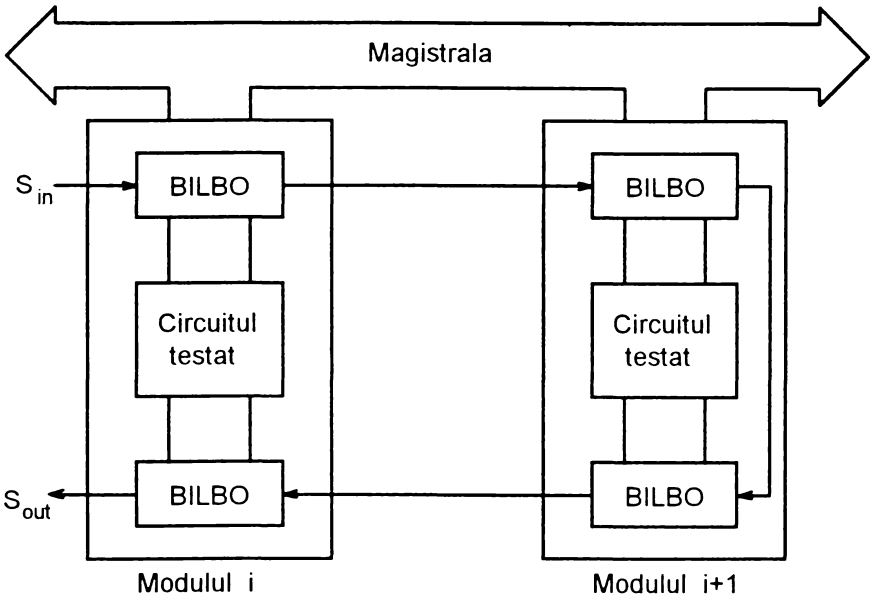


Figura 3.10 Structura BILBO orientată pe magistrală.

Schema bloc din figura 3.10 arată structura BILBO orientată pe magistrală. Registrele BILBO din partea superioară lucrează ca generatoare de vectori pseudoaleatorii în timp ce cele din partea inferioară lucrează ca registre de semnături. Testarea se face introducând (scanând) în registrele BILBO configuraţia binară de iniţializare (sămânţa) a registrelor şi după aceea se comprimă în analizorul de semnături rezultatele obţinute de circuitul testat în urma stimulării lui cu vectori pseudoaleatorii generaţi de registre BILBO cu rol de generatoare. După aceea se extrage rezultatul conţinut în registre BILBO prin "scan out" în timp ce se introduce altă configuraţie binară. Semnăturile obţinute se verifică comparându-se cu cele corecte, obţinute prin simulare.

3.1.3.2. Modulul comutator de testare incorporat pentru izolarea defectelor

În unele aplicaţii, în timpul operaţiei normale, circuitele BILBO supraîncarcă schema, nejustificat, din punct de vedere al consumului de energie iar izolarea defecţiunilor adesea durează mult. În locul registrului BILBO s-a propus [KANO89] o altă variantă de analiză a semnăturii care utilizează un modul numit comutator de testare SW (testing-switch). Modulele SW pot îmbunătăţi la nivel de placă şi timpul de test şi supraîncărcarea de putere a schemelor de analiză a semnăturii implementate utilizând circuitele BILBO

În multe probleme de izolare a defecțiunii, este mai convenabilă izolarea defecțiunii la un grup de chip-uri decât la un singur circuit. Aceste grupuri se numesc grupuri de ambiguitate (AG). Modulul SW generează pattern-uri de test pseudoaleatoare pentru a fi aplicate la intrările unui set AG, în timp ce simultan comprimă datele răspunsului de test de la ieşirile unui alt set AG. Această caracteristică poate minimiza timpul total al testului de izolare a defecțiunii.

Incorporarea autotestării şi a boundary-scan în fiecare IC complex ar fi cea mai elegantă soluție la nivel de placă, dar poate că producătorii consideră că este prea costisitoare incorporarea acestor tehnici la toate produsele IC standard. Proiectanților s-ar putea să li se pară mai eficient din punct de vedere economic, pentru unele aplicații, reproiectarea minoră a plăcilor utilizând IC-uri achiziționate de înainte fără boundary-scan/self-test şi câteva chip-uri cu module BIT, decât reachiziția masivă de IC-uri cu autotest şi boundary-scan incorporat.

3.1.3.2.1. Arhitectura comutatorului de testare

În figura 3.11 este prezentată schema bloc a modulului SW care conține trei părți principale:

- un generator de pattern-uri de test, care furnizează pattern-uri pseudoaleatoare de test unității care se testează (UT);
- un analizor de semnătură, care recepționează răspunsul UT-ului şi produce semnătura UT-ului;
- o rețea de interschimbare a datelor, care transferă date de la intrare la ieşire fără a realiza asupra lor nici o operație logică.

Intrările de control ale modului determină cum să se configureze căile de date de la rețeaua de interschimbare. Intrările de date (data_in) sau ieşirile generatorului de pattern-uri de test (tpg_out) pot fi trimise la ieşirile de date (data_out). Similar, oricare intrare poate fi trimisă la analizorul de semnături pentru a comprima răspunsul.

Generatorul de pattern-uri de test este un registru de deplasare cu bucle de reacție (feedback) care poate fi cuplat în cascadă. Acest generator este cu "user-defined seed", adică se poate determina pattern-ul inițial (semința) al secvenței de test, iar feedback-ul lui este programabil. Generatorul este implementat într-o formă care minimizează întârzierile porților în calea de feedback şi controlează corelarea dintre vectori de test generați. Datorită faptului că generatorul are feedback-ul programabil, se oferă o flexibilitate considerabilă pentru controlarea repetării vectorilor de test şi în ce secvență să apară aceştia.

Analizorul de semnături este un registru de deplasare cu feedback-ul fixat (bazat pe un polinom primitiv), care nu se poate cupla în cascadă, şi are o structură pe 16 biți pentru a limita necesitățile de I/O ale modulului SW. Această structură este general acceptabilă pentru că probabilitatea scăpării detecției erorilor este aproape 2^{-16} atunci când se utilizează un registru feedback bazat pe un polinom primitiv. Analizorul de semnături, pe lângă comprimarea răspunsului intrărilor sau ieşirilor de date provenit de la generatorul pattern-urilor de test, monitorizează, atunci când este nevoie, pinii de ieşire a datelor.

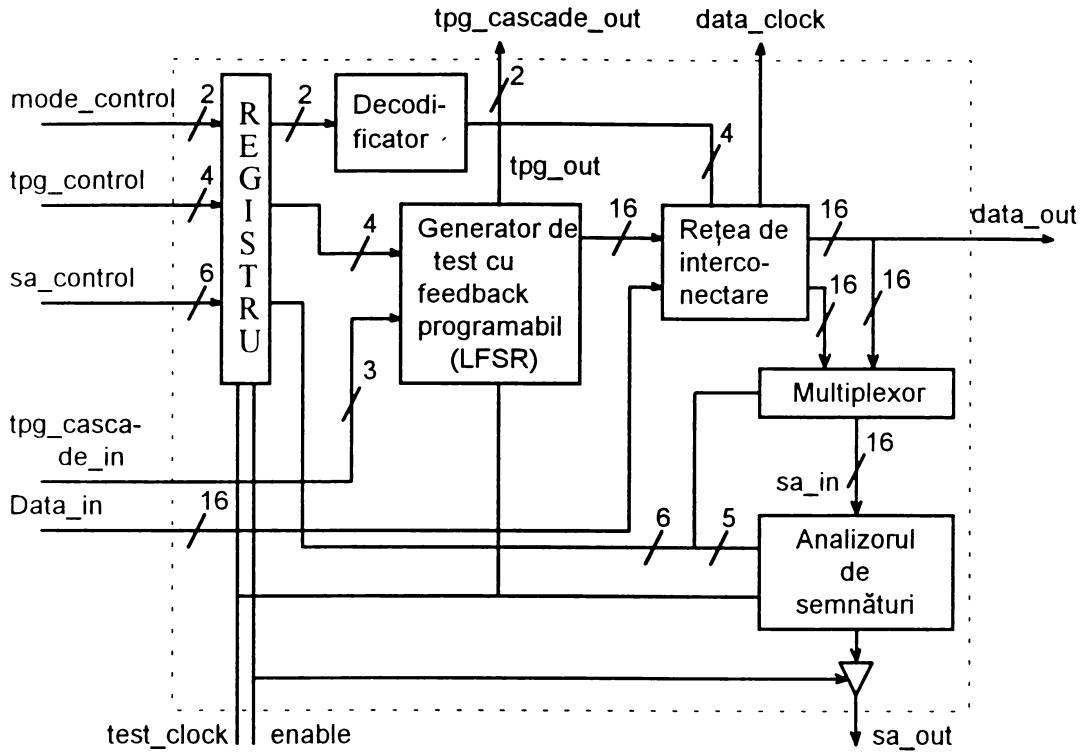


Figura 3.11 Arhitectura unui modul comutator de testare (SW)

Rețeaua de interschimbare a datelor este proiectată în aşa fel încât de fiecare dată când ieşirile generatorului pattern-urilor de test sunt trecute la ieşirile de date, intrările de date să fie trecute la analizorul de semnături pentru comprimare. Acest mod de operare, numit mod "TPG & SA", permite testarea simultană a mai multor AG-uri, fapt care reduce timpul necesar pentru a izola erorile. Rețeaua de interschimbare a datelor are, de asemenea, un mod "pass-through", în care intrările de date sunt trecute la ieşirile de date; un mod "clocked-transfer", în care intrările de date sunt transferate la ieşirile de date sub controlul unui clock derivat din clock-ul plăcii; şi un mod de "autotest", în care sunt direcționate spre analizorul de semnături, fie datele din generatorul pattern-urilor de test, fie datele pinilor de ieşire. Modul de autotest se utilizează pentru a verifica funcționalitatea generatorului pattern-urilor de test şi analizorului de semnături şi pentru a controla pinii de ieşire ai chip-ului.

Toate aceste moduri de operare au fost verificate [KANO89] prin implementarea unui chip prototip de comutator de test în tehnologia CMOS.

3.1.3.2.2. Utilizarea modulelor comutatoare de testare

Primul pas pentru incorporarea modulelor SW la proiectarea unei plăci este acela de a se partiționa placa în AG-uri. Se presupune că AG-urile includ chip-uri care utilizează logica sincronă. După identificarea AG-urilor, se introduc module SW astfel încât toate intrările AG-urilor să treacă prin aceste module. Figura 3.12 arată un exemplu de AG-uri interconectate, în timp ce figura 3.13 ilustrează cum sunt incorporate modulele SW în proiectul respectiv. Caracteristicile tipice de interconectare cum sunt feedback-ul şi fanout-ul între AG-uri sunt incluse în figura 3.12. Chiar dacă nu se iau în considerare în mod explicit intrările de control ale plăcii se presupune că toate aceste intrări trec de asemenea prin modulele SW pentru a

furniza controlarea corespunzătoare a funcțiilor AG în timpul testului. În anumite situații o unitate externă de test poate controla direct aceste intrări.

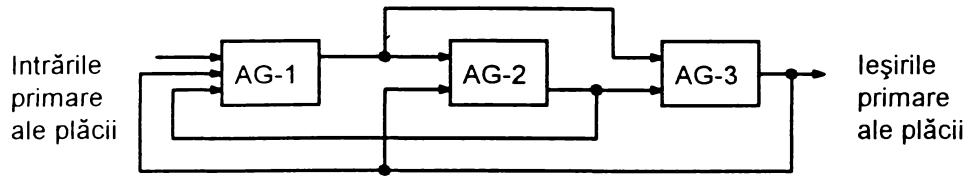


Figura 3.12 O interconectare tipică a grupurilor ambigue.

În fine, dacă registrele de la intrările AG-urilor sunt implementate ca chip-uri separate, atunci chip-urile SW înlocuiesc acolo unde este posibil registrele existente, şi utilizează modul clock-transfer în timpul operării normale. Dacă registrele de la intrările AG-urilor sunt părți ale chip-urilor din AG, atunci modulele SW sunt adăugate la intrările AG-urilor, şi se foloseşte modul pass-through în timpul operării normale.

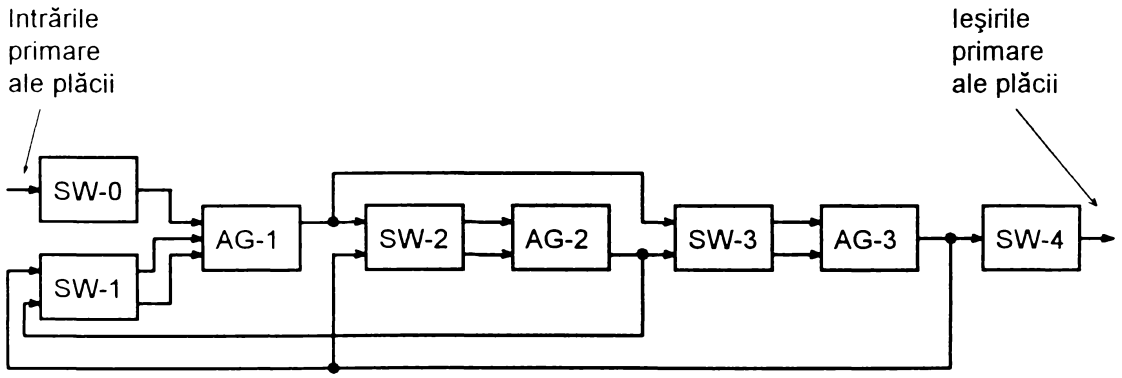


Figura 3.13 Schema din fig. 3.12 după inserarea modulelor SW.

3.1.3.2.3. Izolarea defectelor cu ajutorul modulelor comutatoare de testare

Unitatea care conduce testarea (TCU, Test-Control Unit) realizează în principiu două teste preliminare pentru a verifica operarea corespunzătoare a modulelor SW din placă:

Test 1. În acest test, TCU-ul resetează toate modulele SW ale plăcii aplicând un semnal de reset global. După aceasta, selectează fiecare comutator (switch) prin intermediul liniei lui de activare şi poziționează semnalele lui de control astfel încât să activeze calea dintre generatorul pattern-urilor de test şi analizorul de semnături. TCU-ul, de asemenea, inițializează generatorul pattern-urilor de test şi analizorul de semnături la stările dorite. După un număr predeterminat de tacte, o semnătură se deplasează prin nodul sa_out şi TCU-ul o compară cu semnătura aşteptată.

Acest test ajută la verificarea integrității generatorului pattern-urilor de test pentru structura feedback aleasă. De asemenea, verifică analizorul de semnături şi o parte din rețeaua de interschimbare a datelor. Testul se repetă cu câteva polinoame feedback judicios alese pentru a verifica structura feedback programabilă a generatorului.

Test 2. În acest test, TCU-ul resetează (sau poziționează la o stare cunoscută) global toate AG-urile şi activează calea între intrarea de date şi analizorul de semnături (modul TPG & SA) din fiecare SW. După un ciclu de tact, deplasează conținuturile analizorului de semnături pentru a verifica dacă ieşirile AG-urilor sunt într-adevăr inițializate. De asemenea în timpul acestui test, TCU-ul inițializează ieşirile generatorului pattern-urilor de test la 0 logic şi la 1 logic. TCU-ul încarcă în paralel datele de la pinii de ieşire în analizorul de semnături şi le transferă serial spre exterior pentru a verifica integritatea portului de ieşire. În continuare, el activează modurile pass-through şi clocked transfer şi verifică încă o dată valorile aşteptate la pinii de ieşire.

După aceste teste preliminare, TCU-ul testează AG-urile. Testarea AG-urilor constă din cinci paşi.

Pas 1. TCU-ul identifică AG-urile care pot fi testate simultan utilizând informația despre interconexiunile lor. Figura 3.14 arată un exemplu de AG-uri care pot fi testate simultan. În figura 3.13, AG-urile nu se pot testa simultan din cauza că trebuie să se păstreze unele ieşiri ale AG-urilor la stări predeterminate în timp ce alte AG-uri se testează. Pentru a ilustra aceasta, se presupune că se testează AG-1 din figura 3.13 prin utilizarea generatoarelor din SW-0 şi SW-1 şi analizoarele de semnături din SW-2 şi SW-3. Presupunând că se comprimă în timpul testului toate datele de ieşire AG-1, trebuie să se păstreze atât ieşirile AG-2 cât şi ieşirile AG-3 la stări predeterminate prin aplicarea inițializării AG-urilor în timpul testului. Acest lucru permite verificarea funcționalității lui AG-1 în exclusivitate prin urmărirea semnăturilor finale din SW-2 şi SW-3.

Dacă nu există AG-uri testabile simultan sau au fost deja testate, TCU-ul selectează un AG netestat până acum (dacă există unul) utilizând o ordine predefinită dictată de proiectarea plăcii.

Pas 2. TCU-ul inițializează toate AG-urile utilizând semnale de control cum sunt reset şi preset. El selectează unul câte unul modulele SW asociate cu AG-urile din pasul 1 pe care le inițializează programând feedback-ul generatorului, introducând valoarea inițială şi poziționând corespunzător semnalele de control.

Pas 3. TCU-ul activează clock-urile plăcii. De asemenea, asigură ca clock-ul de test să aibă frecvența dorită şi relația de fază potrivită cu clock-urile plăcii. Modulele SW aplică pattern-urile pseudoaleatoare de test la AG-urile corespunzătoare şi comprimă răspunsurile.

Pas 4. După un număr predeterminat de tacte, TCU-ul opreşte deplasarea din toate analizoarele de semnături. Atunci, el selectează fiecare analizor de semnături utilizând intrarea lui de activare şi transferă spre exterior semnătura care o conține. Compară această semnătură cu cea aşteptată, obținută din simulare şi determină starea AG-ului.

Pas 5. TCU-ul repetă paşii 2-4 pentru toate modurile de control necesare ale AG-urilor.

TCU-ul repetă procedura pasului 5 până când toate AG-urile sunt testate. Pasul 4 izolează o eroare în interiorul unui AG şi în modulele SW asociate lui. Din moment ce sunt deja verificate modulele SW în testele preliminare, defecțiunea este de fapt izolată în modulul AG.

Modulul SW, este de asemenea eficace pentru testarea deterministică a AG-urilor. Pentru acest tip de testare TCU-ul trebuie să insereze vectori de test în modulele SW. Testele deterministice nu sunt în general teste în timp real din cauza nevoii mai multor tacte pentru introducerea în serial a unui vector de test şi aplicarea lui în paralel la AG. Avantajul acestei capabilități este că o hibridizare între testele

deterministice şi cele pseudoaleatoare poate minimiza mărimea supraîncărcării modulelor AG şi BIT.

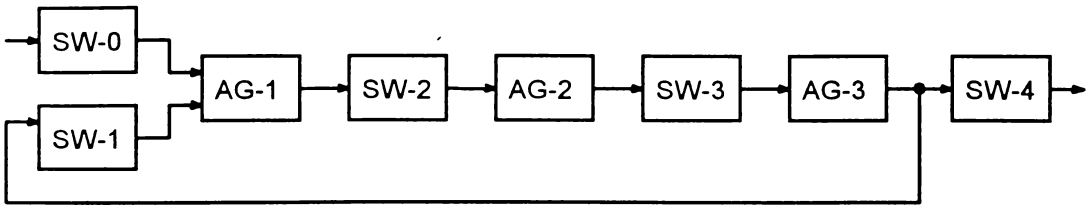


Figura 3.14 Un exemplu de grupuri ambigue testabile simultan.

Se poate de asemenea realiza un test de acceptabilitate a plăcii (go/no-go) pentru a determina starea ei operațională înainte de testul pentru izolarea defecțiunii. În figura 3.13, testul acceptabilității utilizează generatorul de vectori pseudoaleatoare din SW-0 pentru a aplica date de intrare la AG-1, în timp ce analizorul de semnături din SW-4 comprimă ieşirile. Restul de comutatoare de testare, SW-1, SW-2 şi SW-3 sunt puse în modul normal de operare, fie pass-through fie clocked-transfer. Acest test de acceptabilitate necesită teste adiționale care trebuie să verifice căile de la intrările primare la intrările AG-1 prin SW-0, şi de la ieşirile AG-3 la ieşirile primare prin SW-4. Testele asupra AG-urilor individuale ar putea fi de asemenea necesare în unele cazuri pentru a întări convingerea unei decizii afirmative de acceptare.

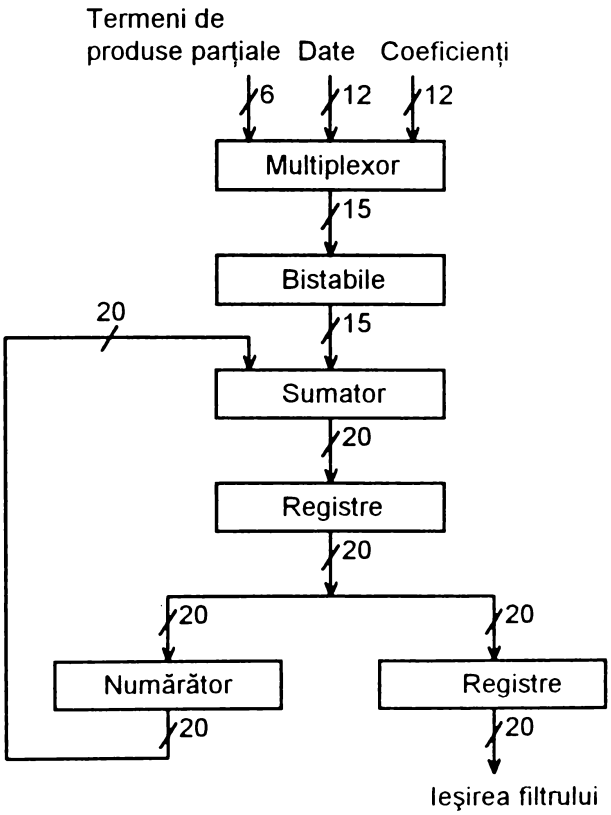


Figura 3.15 O implementare a unui filtru digital.

Figura 3.15 reprezintă o schemă bloc a unui filtru digital de impulsuri finite. Implementarea acestei funcții se face pe o placă cu aproximativ 35 de chip-uri de complexitate MSI. Lățimea maximă a căilor de date este 20.

Figura 3.16 arată schema după incorporarea modulelor SW. Se presupune că s-au utilizat chip-uri SW de lățime 20 de biți. Supraîncărcarea aproximativă în chip-uri din cauza incorporării facilităților de test (BIT) este 5 la 35 de chip-uri, adică aproape 14%. Nu s-a numărat chip-ul SW-3 care a înlocuit chip-ul de bistabile din schema inițială. În afară de SW-3 toate celelalte chip-uri SW operează în modul pass-through în timpul operării normale. SW-3 funcționează în modul clocked-transfer în timpul operării normale.

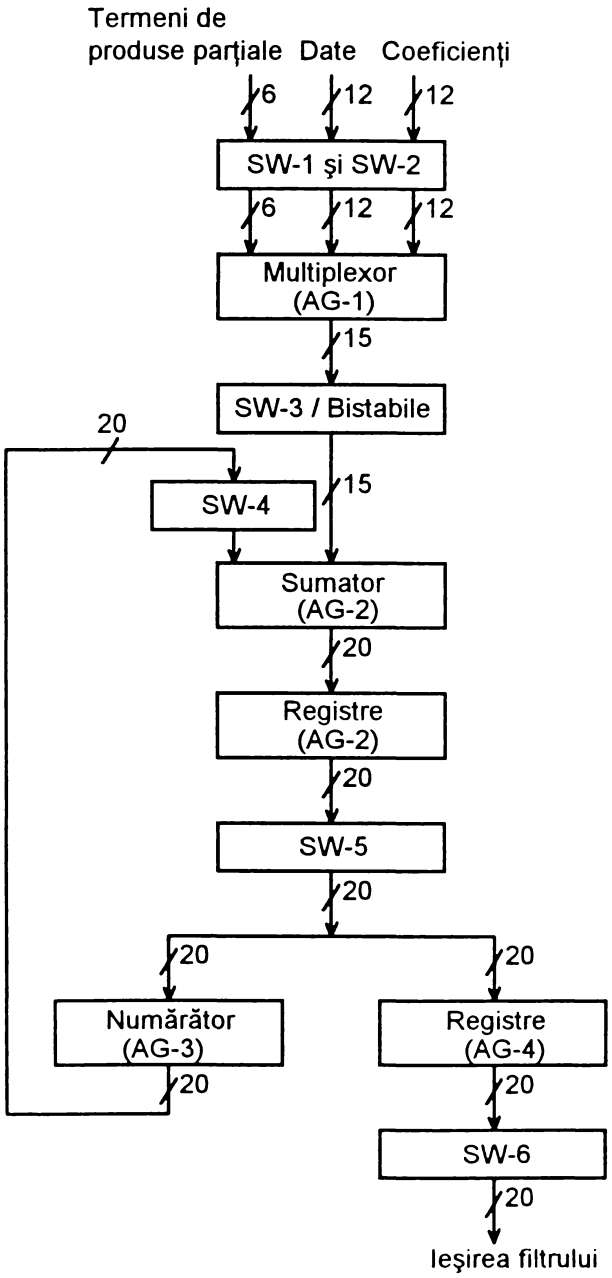


Figura 3.16 Incorporarea modulelor comutatoarelor de testare în schema filtrului digital din figura 3.15.

Pentru a se verifica eficacitatea incorporării modulelor SW în proiectarea acestei plăci de filtru digital, s-au dezvoltat modele la nivel de poartă pentru toate chip-urile din placă utilizând CADAT logic şi simulatorul de defecțiuni de la HHB Systems. CADAT-ul s-a utilizat pentru verificarea funcționalității plăcii după

incorporarea BIT-ului şi pentru simularea defecţiunilor, luând în considerare numai defecţiunile singulare de tip stuck-at.

În timpul simulării s-au controlat în mod deterministic intrările semnalelor de control cum sunt read/write şi chip-select, în timp ce se aplicau vectori pseudoaleatorii generaţi de modulele SW la intrările căilor de date. În acest exemplu, controlul este relativ simplu. Se schimbă intrările de control numai de două ori în timpul întregii simulări. Pentru un test de acceptabilitate, modulele SW-1 şi SW-2 din figura 3.16 au lucrat ca generator de pattern-uri de test, în timp ce modulul SW-6 a fost analizor de semnături.

Nu s-au obţinut toate semnăturile corespunzătoare tuturor defectelor circuitului din cauza timpului şi costului exagerat de mare. În schimb, s-a utilizat capabilitatea CADAT-ului pentru simularea concurentă a defecţiunilor. Utilizând această caracteristică, s-a determinat acoperirea de defecţiuni a acestor seturi de vectori test pseudoaleatorii prin compararea ciclu de ciclu a răspunsurilor de la ieşirile filtrului cu cele aşteptate. În consecinţă, nu s-a ţinut cont cât de eficace a fost analizorul de semnături în comprimarea datelor de ieşire. Din acest motiv, rezultatele s-ar putea să fie un pic mai optimiste.

Pentru testarea defecţiunilor la nivel de pini, s-au folosit 5 seturi de teste pseudoaleatoare cu 200 de vectori per set. Acoperirea defecţiunilor pentru totalul de 1244 de defecţiuni de pini în placa de filtru a fost de 100%. Aceste seturi distincte de vectori de test au fost generate prin utilizarea diferiţilor vectori de iniţializare (initial seeds) pentru generatorul pattern-urilor de test (configurat cu ajutorul modulelor SW-1 şi SW-2).

Pentru testarea defecţiunilor la nivel de poartă, s-au folosit 5 seturi de teste pseudoaleatoare cu 500 de vectori per set. Acoperirea defecţiunilor pentru totalul de 12292 de defecţiuni la nivel de poartă de tip stuck-at a fost de 99.75%. S-a putut obţine o acoperire atât de mare din cauza că placa a fost testabilă la teste pseudoaleatoare. O intrare de control eronată dădea întotdeauna o ieşire eronată, în aşa fel încât astfel de erori erau uşor de detectat.

3.1.3.2.4. Performanţa şi costul comutatoarelor de testare

S-a măsurat eficacitatea izolării defecţiunilor cu module SW prin introducerea arbitrară a defecţiunilor alese (singulare sau multiple) în fiecare AG singular sau/şi în modulele SW asociate lui. Singurul lucru relevant este obţinerea unei semnături diferită de semnătura aşteptată corectă. Deci, se pare că modulele de comutatoare pentru testare (SW) sunt o modalitate viabilă pentru a obţine acoperiri de defecţiuni mari şi acurateţe în izolarea defecţiunilor pentru plăci care sunt testabile cu ajutorul pattern-urilor pseudoaleatoare.

O problemă care apare în utilizarea modulelor SW pentru izolarea defecţiunilor este aceea că ele introduc întârzieri adiţionale între AG-uri în timpul modului normal de operare. Totuşi, dacă proiectanţii implementează chip-ul cu comutatorul de testare utilizând tehnici de împachetare şi tehnologii avansate, ei pot minimiza întârzierile adiţionale. Se pare că, în cazul multor aplicaţii, această întârziere va avea un efect neglijabil asupra performanţelor sistemului. Cei care au propus această soluţie estimează că adăugarea modulelor de comutatoare pentru testare la o placă de filtru digital ar mări perioada clock-ului cel mult cu 5%. Proiectanţii pot de asemenea să menţină creşterea spaţiului ocupat de chip-urile SW la un minim prin partiţionarea corespunzătoare în AG-uri a plăcii.

Din moment ce acest modul BIT suportă o tehnică de testare off-line, supraîncărcarea de putere disipată a BIT-ului trebuie să fie minimă în timpul operării normale. Presupunând că tehnologia CMOS a fost utilizată pentru implementarea comutatorului de testare, supraîncărcarea de putere este minimă în timpul modului pass-through. Motivul este acela că, în timpul acestui mod, se poate dezactiva clock-ul comutatorului de testare şi orice disipare de putere a comutatorului este de 10%. S-a estimat că supraîncărcarea de putere produsă în cazul plăcii de filtru va fi mai mică decât 2% în timpul operării normale.

3.1.4. Controlul erorilor la nivel de procesor

În 1972, Pradhan şi Reddy au arătat că codul de paritate, utilizând redundanța de ordinul duplicării, poate realiza cu eficacitate controlarea erorilor. Această afirmație, că numai duplicarea poate realiza corecția erorilor în procesoare, a fost realizată parțial în calculatorul (4,2). Figura 3.17 ilustrează acest concept [KROL86], care este deja utilizat ca un produs comercial prin sistemul de comutare Philips S2500.

S2500 utilizează patru perechi de procesoare-memorie. Fiecare procesor fiind pe 16 biți, iar memoria doar pe 8 biți. De aceea, există redundanța de ordinul patru pentru procesoare şi de ordinul doi pentru memorie. Ieşirea de 16 biți a fiecărui procesor este codificată într-o versiune (4,2) $GF(2^8)$ de cod R-S, producând un cod de 32 de biți. Acest cod poate corecta orice bloc singular de 8 biți. Ieşirea fiecărui procesor este codificată de codificatoare separate. Fiecare codificator produce doar 8 biți din cei 32 de biți ai cuvântului de cod. Memoria asociată celui de al *i*-lea procesor stochează al *i*-lea octet; de aceea, codificatorul pentru procesorul *i* produce 8 biți care corespund celui de al *i*-lea octet. Altfel spus, memoria primului procesor stochează primul octet, memoria celui de al doilea stochează cel de al doilea octet, etc.

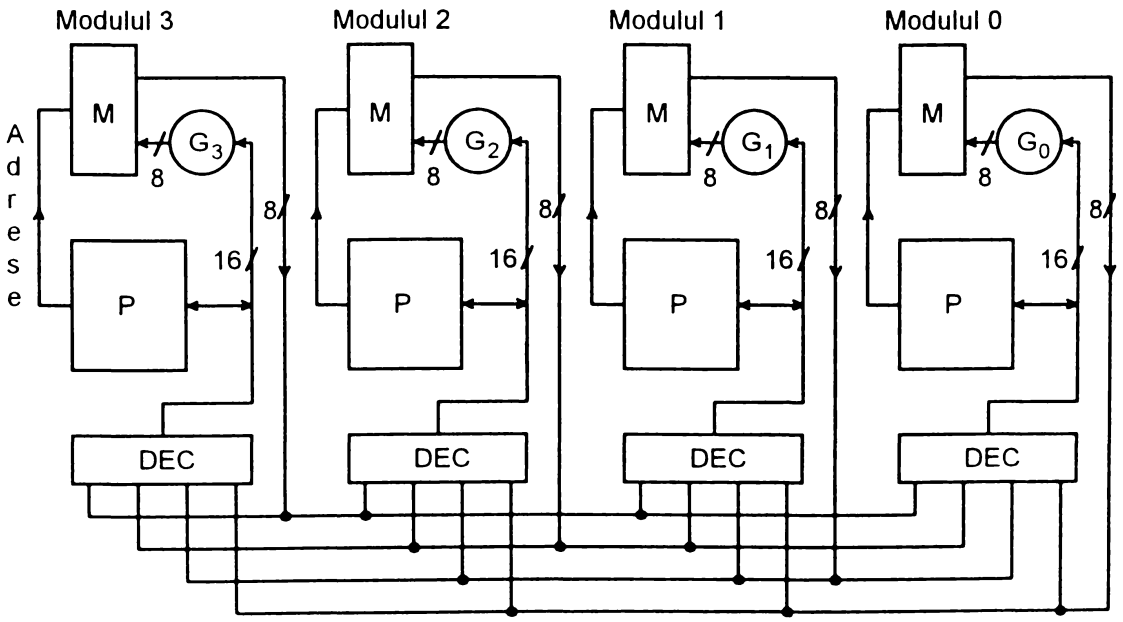


Figura 3.17 Calculatorul (4,2)

La operația de citire, toți cei patru octeți sunt aduşi din memorie şi decodificați de decodificator, care poate corecta orice octet singular eronat sau orice eroare de doi

biți în doi octeți diferiți. Este de notat faptul că deteriorarea unui singur procesor sau a unei singure memorii afectează doar un singur octet din cuvântul de cod; astfel, sistemul poate tolera deteriorarea oricărei perechi singulare procesor-memorie. În plus, codul poate corecta ştergerea unui octet singular precum şi erorile singulare de bit. Astfel, cineva poate demonta o pereche procesor-memorie pentru reparație iar în acelaşi timp sistemul continuă să opereze utilizând decodificarea de erori-ştergeri (error-erasure decoding). La nivel de bit poate fi corectată orice subsecvență de eroare singulară.

Această utilizare a codificării pentru controlarea erorii de procesor este atractivă. Singurul dezavantaj real este acela că, deoarece liniile de adrese la memorie nu sunt codificate, nu are loc corecție de eroare la operația de scriere. În general, prin utilizarea căilor netradiționale ca cele descrise mai sus, codul controlului de paritate poate furniza un control eficace al erorilor procesorului. Utilizarea lor va elimina nevoia de conversie a codului şi întârzierea asociată în transfer procesor-memorie, furnizând astfel controlul uniform al erorii. [KROL86] [FUJI90b] [RAO89]

3.2. Problematica construcției checker-elor de erori

Multe sisteme digitale includ detectoare de erori, adică circuite care detectează şi semnalizează apariția erorilor. Printr-o eroare se înțelege prezența unui semnal a cărui valoare este diferită față de valoarea lui în cazul unui sistem care operează corect. Există două motive principale pentru includerea detectoarelor de erori într-un sistem, unul pentru a preveni ajungerea erorii până la ieşirea sitemului, iar altul pentru a localiza poziția erorii. O cale pentru a preveni propagarea erorii este controlarea (verificarea) datelor recepționate după transferul de date şi dacă o eroare este detectată, fie se cere retransmisia, fie se corectează datele transmise utilizând un cod corector de erori (ca în memoriile RAM). O altă cale este controlarea (verificarea) rezultatului unei operații aritmetice şi repetarea operației dacă rezultatul conține o eroare detectată. Sistemele mari includ de obicei detectoare de erori împreună cu circuite pentru înregistrarea locației (poziției) şi frecvenței evenimentelor de eroare [MCCL90]. Această informație este utilizată pentru a facilita întreținerea rapidă şi exactă.

3.2.1. Tehnici de proiectare pentru checker-e de erori incorporate testabile

Cele mai familiare detectoare de erori sunt checker-ele de paritate. Ele detectează orice număr impar de biți incorecți dintr-un cuvânt cu n biți. Funcționarea unui detector de erori depinde de prezența informației codificate. Informația din sistem este codificată în cuvinte de cod. Aceste cuvinte de cod conțin mai mulți biți decât numărul minim de biți necesar pentru a reprezenta informația, din moment ce numai un subset din toate cuvintele posibile, numit cuvinte de cod (code words), este prezent într-un sistem fără erori. Când într-un cuvânt apare o eroare detectabilă, ea schimbă cuvântul dintr-un cuvânt de cod într-un cuvânt din afară codului (noncode word), aşa zis cuvânt ilegal. De exemplu, într-un sistem care codifică fiecare octet într-un cod de paritate, sunt utilizați 9 biți pentru fiecare octet. Dacă este utilizat un cod de paritate impară, un număr impar de biți din fiecare cuvânt este egal cu 1. Orice număr impar de biți schimbați într-un cuvânt face ca cuvântul să aibă paritate pară.

Detectoarele de erori conțin checker-e, adică circuite de verificare, care primesc la intrări informația codificată şi determină dacă cuvintele prezente sunt cuvinte de cod sau nu. Astfel, checker-ul pentru octeții codificați după paritate este un circuit cu 9 intrări care calculează paritatea intrărilor lui.

Din moment ce funcția unui circuit de verificare (checker) este aceea de a furniza un semnal de ieşire corespunzător cuvintelor de intrare care nu fac parte din cod, testarea checker-ului necesită aplicarea şi a cuvintelor din afară codului ca intrări. În cadrul unui sistem fără erori, checker-ele recepționează la intrări numai cuvinte de cod. Astfel, nu este posibilă testarea unui checker incorporat în sistem decât dacă această abilitate este proiectată special în cadrul sistemului. Dacă sistemul include o facilitate de tip scan-path ca o caracteristică de proiectare pentru testabilitate (Design-For-Testability) atunci orice checker ale cărui intrări provin numai de la bistabile (latches sau flip-flops) poate fi testat complet prin scanarea (serializarea) în bistabile a unui set corespunzător de configurații (patterns) de test. Pentru checker-e ale căror intrări nu provin toate de la bistabilele scan-path, este necesară modificarea proiectării checker-ului pentru a permite testarea completă a defecțiunilor singulare de tip "punere pe" (stuck-at). În continuare se prezintă tehnici de proiectare pentru asigurarea testabilității checker-elor incorporate care nu pot fi testate prin bistabilele scan-path.

Structura unui detector de erori este determinată în mod special de codul detector de erori utilizat în cuvintele care vor fi controlate. Caracteristica cea mai importantă a unui cod este aceea dacă el este separabil sau nu. Fiecare cuvânt al unui cod separabil (numit şi cod sistematic) poate fi împărțit în două părți: partea de date care reprezintă conținutul informației utile şi partea de control care este determinată de partea de date. Un cod controlor de paritate pară este un cod separabil cu un singur bit de paritate egal cu paritatea părții controlate din cuvântul de cod. Un alt exemplu este codul rezidual în care partea de control este egală cu restul modulo M a părții de date. Un cod separabil foarte utilizat este duplicarea în care partea de control este identică cu partea de date. La un cod neseparabil nu este posibil să se determine care cuvânt de informație este reprezentat doar de un subset de biți din cuvântul de cod. Codul cu pondere fixă (M din N) este un cod neseparabil.

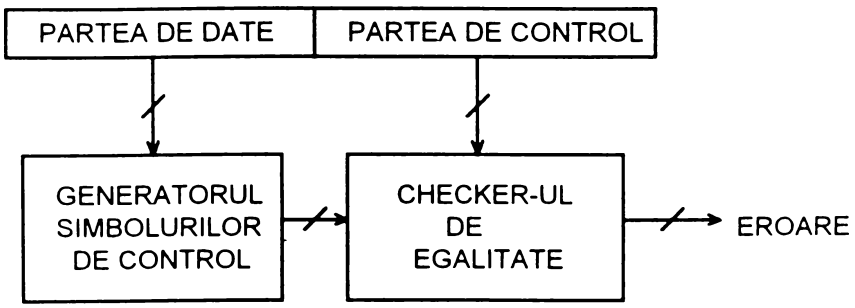


Figura 3.18 Structura generală a unui checker pentru un cod separabil

Oricare cod separabil poate fi controlat printr-o structură ca şi cea din figura 3.18. Dacă, cum este în mod obişnuit, toate combinațiile de biți pot să apară în partea de date, atunci nu există nici o problemă în testarea generatorului simbolurilor de control. Pentru checker-ul de egalitate situația nu este la fel de simplă, din moment ce combinațiile de intrare corespunzătoare situațiilor de erori nu pot să apară atunci când nu sunt prezente erori. Tehnicile de testare pentru checker-ele de egalitate vor fi descrise mai târziu.

3.2.1.1. Checker-e de paritate testabile

Codul de paritate este un cod separabil cu o caracteristică specială aceea că partea de control este un singur bit. Din acest motiv checker-ele de paritate utilizează o structură mai simplă decât structura generală a checker-elor de coduri separabile din figura 3.18.

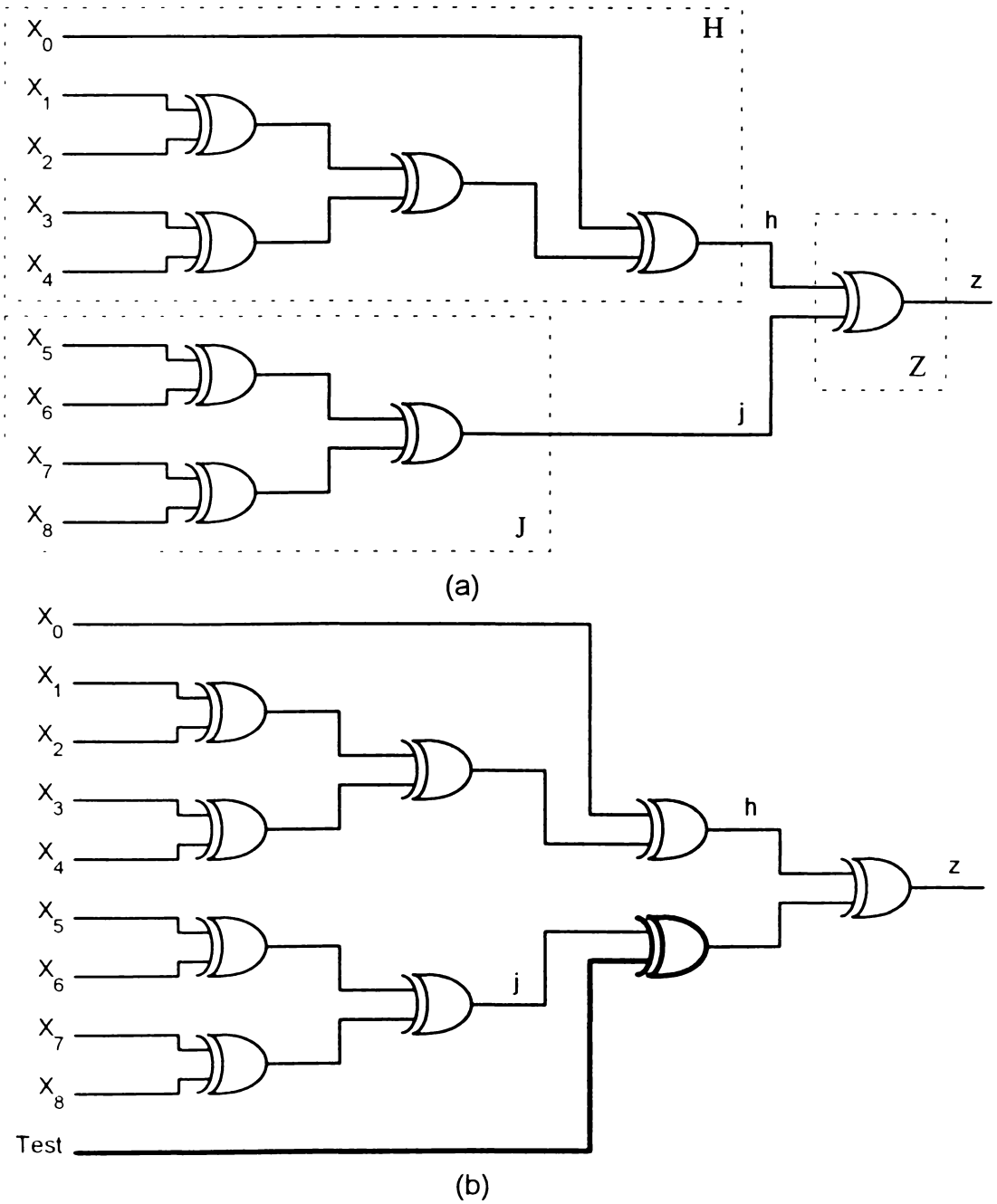


Figura 3.19 Checker-e de paritate pentru un cod de paritate impară pe 9 biți (a) arborele SAU-EXCLUSIV şi (b) arborele SAU-EXCLUSIV testabil

De fapt, ceea ce se cere este o schemă pentru calcularea sumei modulo 2 sau SAU-EXCLUSIV a biților din cuvântul de cod. Figura 3.19(a) arată un arbore de porți SAU-EXCLUSIV cu două intrări, care controlează paritatea unui octet codificat după

un cod de paritate pe 9 biți. Dacă este utilizat un cod de paritate impară, ieşirea z este egală cu 1 pentru toate cuvintele de cod valide.

Schema din figura 3.19(a) poate fi considerată ca trei subscheme: subschema H, intrările şi porțile conectate pentru a conduce semnalul h; subschema J, intrările şi porțile conectate pentru a conduce semnalul j; şi subschema Z fiind constituită dintr-o singură poartă conectată la ieşirea z. Orice eroare singulară de tip "stuck-at" (o eroare care cauzează punerea pe 0 logic sau pe 1 logic a unei linii de semnal din circuit în loc de a se determina valoarea ei de o ieşire a unei porți sau de o intrare primară) în schema H sau J poate fi testată din moment ce aceste subscheme recepționează toate combinațiile de intrare posibile. De fapt, aceste subscheme au proprietatea de autotestare (self-testing). O subschemă a unui checker se numeşte "self-testing" dacă şi numai dacă orice defecțiune singulară de tip stuck-at din subschemă cauzează o indicare de eroare la ieşirea checker-ului pentru cel puțin un cuvânt de cod valid prezent la intrarea lui. (Uneori sunt utilizate alte modele de defecțiuni în locul modelului de defecțiune singulară de tip stuck-at chiar dacă acesta este cel mai uzual.) Checker-ele sunt proiectate des să aibă proprietatea de autotestare (self-testing) pentru a garanta indicarea erorii în cazul apariției unei defecțiuni în schema lor în timpul operării normale.

3.2.1.2. Checker-e cu autotestare

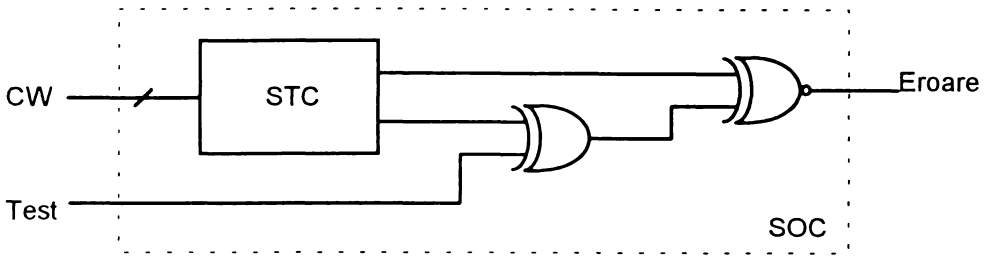
Orice circuit de verificare cu autotestare completă (completely self-testing checker), adică unul pentru care întregul checker este cu autotestare (self-testing), trebuie să aibă cel puțin două ieşiri, din moment ce o defecțiune de tip "stuck-at" care indică "NO-ERROR" pe o singură ieşire nu poate fi detectată aplicând un cuvânt de cod la intrare [MCCL90]. Practica obişnuită este proiectarea unui checker cu autotestare cu două ieşiri ale căror semnale 01 şi 10 indică funcționarea fără erori iar semnalele 00 şi 11 apar ca răspuns la o defecțiune singulară de tip stuck-at din checker sau la un cuvânt de intrare care nu aparține codului. Acestea sunt semnalele care apar la linii h şi j din figura 3.19(a), astfel acest circuit este cu autotestare dacă este acceptabilă utilizarea a două semnale în loc de unul pentru indicarea erorii. Orice circuit pentru controlul parității, care are două ieşiri, fiecare egal cu paritatea unuia dintre cele două subseturi de intrări, este cu autotestare completă.

Proprietatea de autotestare furnizează o tehnică generală pentru asigurarea testabilității checker-elor incorporate. Există proiectări pentru toate checker-ele importante cu autotestare completă. Majoritatea sistemelor cu checker-e incorporate necesită câteva indicații, atunci când o eroare a fost detectată de un checker de erori. (Operarea sistemului poate fi oprită şi se verifică condițiile checker-elor individuale pentru diagnosticare.) Trebuie să fie furnizată o facilitate pentru a combina ieşirile checker-elor individuale într-un singur indicator. Structurile pentru combinarea ieşirilor dublă cale (two-rail outputs) ale checker-elor cu autotestare vor fi discutate ulterior.

3.2.1.3. Checker-e cu o singură ieşire

Dacă se cere un checker de paritate cu o singură ieşire, atunci este posibil să se facă circuitul din figura 3.19(a) complet testabil prin adăugarea circuitului adițional desenat cu linii groase în figura 3.19(b). În timpul operării normale, intrarea "Test"

adăugată este ținută pe 0 logic, şi operarea circuitului nu este afectată de poarta adăugată. Pentru testarea porții de ieşire, intrarea "Test" se poziționează pe 1 logic pentru a plasa combinațiile de 11 şi 00 la intrările acestei porți. De notat că poarta SAU-EXCLUSIV adăugată nu este cu autotestare (self-testing) din moment ce intrarea de test este tot timpul pe 0 logic în cazul operării normale, dar poate fi testată complet prin poziționarea semnalului Test = 1.



STC = Self-Testing Checker (circuit verificator cu autotestare)
SOC = Single-Output Checker (circuit verificator cu o singură ieşire)
CW = Code Word (cuvânt din code, legal)

Figura 3.20 Structura de convertire a unui checker cu autotestare într-unul testabil cu o singură ieşire.

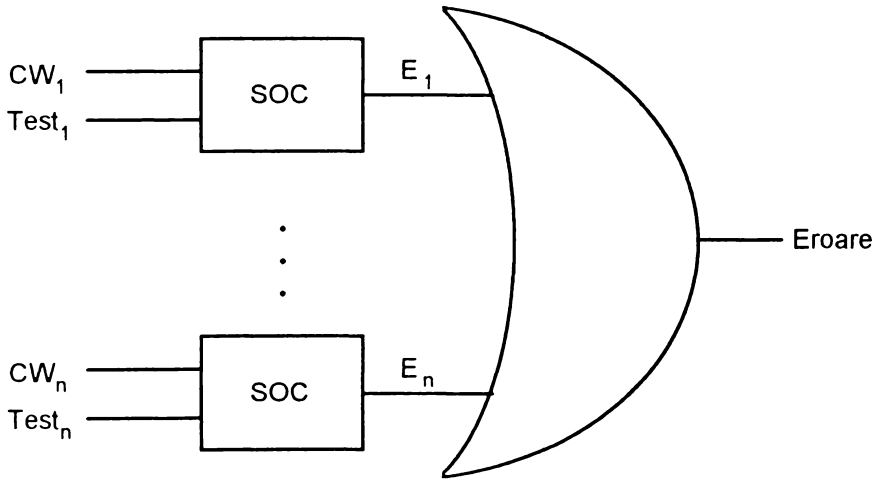


Figura 3.21 Circuitul care combină ieşirile a n checker-e cu o singură ieşire într-un singur semnal de eroare.

Tehnica ilustrată în figura 3.19(b) este generală în sensul că poate fi folosită pentru a converti oricare circuit STC (Self-Testing Checker) cu două ieşiri într-unul testabil cu o singură ieşire, aşa cum se arată în figura 3.20. O poartă SAU-NU-EXCLUSIV prezentată ca şi poartă de ieşire furnizează un semnal de eroare activ la nivel high.

Figura 3.21 prezintă o structură pentru combinarea ieşirilor a n checker-e cu o singură ieşire într-un singur semnal de eroare. Acest circuit necesită n semnale de test ($Test_1 ... Test_n$), unul pentru fiecare checker cu o singură ieşire. Poarta SAU de la ieşire nu este cu autotestare din moment ce ea are la toate intrările ei 0 logic în timpul operării normale. Ea este testabilă deoarece semnalele de test individuale pot fi utilizate pentru a aplica toate configurațiile cu un singur 1 necesare pentru a se testa complet poarta la toate defecțiunile singulare de tip stuck-at

3.2.1.4. Circuite de verificare dublă cale

Un dezavantaj al structurii din figura 3.21 este faptul că ea necesită n semnale adiționale de test, câte unu pentru fiecare checker individual. Figura 3.22 arată o structură de combinare a ieşirilor a n checker-e cu autotestare (STC) utilizând numai un semnal de test. Un circuit de verificare dublă cale (two-rail checker), adică un circuit care controlează dacă fiecare pereche de intrări are valori complementare, este utilizat pentru a converti cele n perechi de semnale într-o singură pereche de semnale complementare dacă şi numai dacă toate cele n perechi de intrare au semnale complementare. Cele două ieşiri sunt apoi convertite într-un singur semnal de ieşire testabil utilizând circuitul din figura 3.20.

Figura 3.25 arată o schemă cu un circuit de verificare dublă cale care converteşte două perechi de semnale de intrare într-o singură pereche de semnale de ieşire. Acest circuit este cu autotestare, dacă şi numai dacă, în timpul operării normale toate cele patru cuvinte de cod valide apar la intrările lui. Un arbore de circuite ca şi cel din figura 3.25 poate fi utilizat pentru a converti n perechi de intrări într-o singură pereche de ieşire. Dacă acest arbore este cu autotestare sau nu, depinde de intrările pe care le recepționează în timpul operării corecte (dacă sunt suficiente pentru a îl testa).

3.2.1.5. Checker-e de egalitate

Un tip foarte important de checker-e este cel care verifică egalitatea (comparator), adică un checker care compară două cuvinte de intrare pentru a determina dacă biții corespunzători din ambele cuvinte au aceeaşi valoare. Checker-ul de egalitate este componenta cheie în checker-ele pentru coduri separabile (figura 3.18) şi pentru compararea ieşirilor circuitelor duplicate. Cea mai simplă schemă de checker de egalitate este ilustrată în figura 3.23 printr-un circuit de comparare a două cuvinte X şi Y a câte 4 biți fiecare. Fiecare pereche de biți este conectată la o poartă SAU-EXCLUSIV a cărei ieşire trebuie tot timpul să fie pe 0 dacă sistemul operează corect. Toate ieşirile porților SAU-EXCLUSIV sunt conectate la o poartă SAU a cărei ieşire va fi 0 atâta timp cât toate ieşirile porților SAU-EXCLUSIV sunt 0.

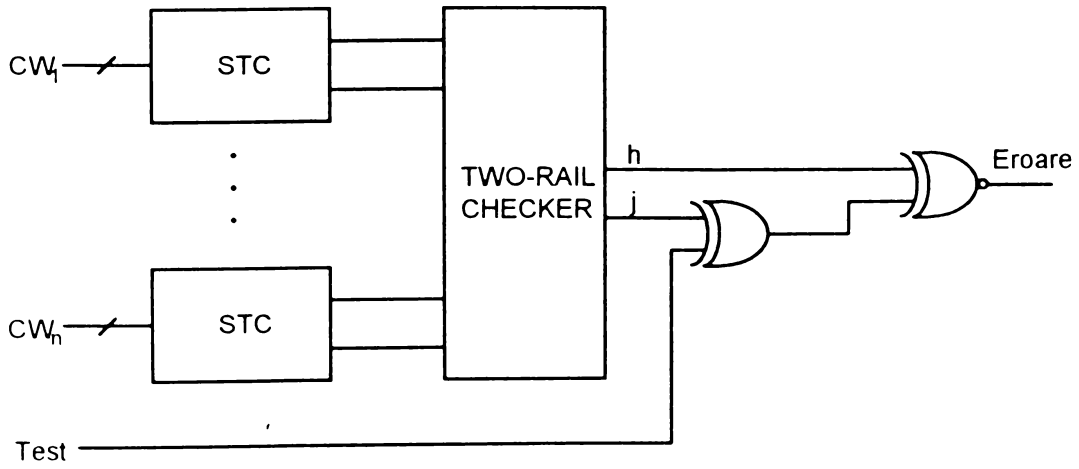


Figura 3.22 Circuitul de combinare a n perechi de ieşiri ale checker-elor cu autotestare într-un singur semnal de eroare

Deoarece acest circuit este foarte simplu, nu este cu autotestare. De fapt, este foarte greu să se şi testeze. Orice defecţiune stuck-at-0 la o ieşire a unei porţi nu poate fi detectată prin nici o intrare normală din circuit. Circuitul poate fi făcut testabil prin adăugarea a câte unei porţi SAU-EXCLUSIV şi a câte unui semnal de test la fiecare din porţile SAU-EXCLUSIV existente. Aceasta face posibilă complementarea selectivă a câte uneia din intrări de la porţile SAU-EXCLUSIV originale aşa cum se observă în figura 3.19 pentru intrarea j. Această modificare mai mult decât dublează complexitatea circuitului care tot nu capătă abilitatea de autotestare.

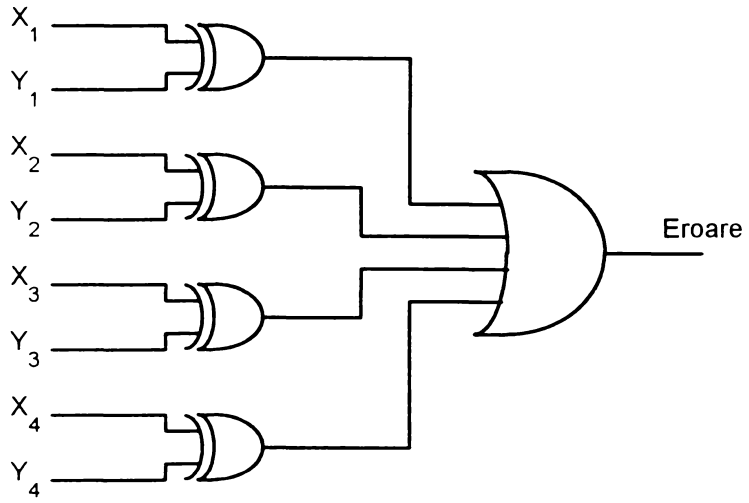


Figura 3.23 Un exemplu de structură de checker de egalitate care nu este cu autotestare şi nici testabil.

Un checker de egalitate cu autotestare poate fi obţinut prin complementarea tuturor biţilor din unul dintre cuvintele care trebuie să fie comparate pentru a forma un cod dublă cale (two-rail code). Checker-ele dublă cale discutate mai sus pot fi utilizate pentru a realiza un checker de egalitate testabil. O altă posibilitate este aceea de a concatena cuvântul complementat şi cel necomplementat pentru a forma un cuvânt de cod care are exact jumătate din biţii lui egali cu 1. Un checker k din 2k poate fi utilizat pentru a forma un checker de egalitate testabil. Nu este sigur că această cale are vreun avantaj faţă de schemele cu checker-e dublă cale.

3.2.1.6. Checker-ele M din N

Rămâne de discutat o clasă de checker-e incorporate: checker-ele M din N. Deşi sunt mai puţin importante decât celelalte, totuşi sunt utilizate. Checker-ul 1 din n este folosit pentru a monitoriza operarea corectă a reţelelor complete de decodificare. O implementare cu autotestare a acestui checker este formată prin conectarea tuturor ieşirilor decodificate care corespund la intrările cu paritate pară la o poartă SAU şi tuturor ieşirilor care corespund la intrările cu paritate impară la o altă poartă SAU. Ieşirile celor două porţi SAU au fie 10 fie 01 atunci când decodificatorul operează corect. Această schemă este cu autotestare din moment ce porţile SAU recepţionează toate configuraţiile de intrare (00, 10, 01) necesare pentru testarea lor pentru defecţiuni singulare de tip "punere pe" (stuck-at) în timpul operării normale. Această

schemă poate fi convertită într-un circuit testabil cu o singură ieşire cu ajutorul structurii de convertire din figura 3.20.

Codurile cu ponderea fixă M din N se utilizează atunci când se doreşte nu numai detectarea tuturor defectelor singulare de tip stuck-at, ci şi detectarea tuturor defectelor unidirecţionale. Un defect unidirecţional este orice defect care cauzează semnale eronate care toate au aceeaşi valoare incorectă. Toate erorile sunt cauzate fie prin trecerea 0-urilor corecte în 1-uri incorecte, fie prin trecerea 1-urilor corecte în 0-uri incorecte; nu este posibil să fie un 0 incorect şi un 1 incorect prezente într-un cuvânt de date eronat. Defecţiunile singulare din scheme fără inversare (de exemplu, partea internă a unui PLA) sunt modelate cu exactitate ca defecţiuni unidirecţionale.

Codul zecimal 2 din 5 este cel mai cunoscut exemplu al unui cod M din N. Codurile k din 2k sau k din 2k+1 sunt implementările uzuale din moment ce ele conţin numărul maxim de cuvinte de cod pentru o lungime de cuvânt dată. Scheme cu autotestare pentru checker-ele de cod M din N au fost studiate de teoreticienii specialişti la calcularea tolerantă la defecţiuni. Din moment ce aceste implementări de checker-e sunt cu autotestare (self-testing) ele nu prezintă o problemă de testare. Metodele descrise mai înainte pot fi aplicate pentru a obţine scheme testabile cu o singură ieşire sau pentru a combina ieşirile de la câteva checker-e într-un singur indicator de eroare.

Prin utilizarea tehnicii corespunzătoare de proiectare pentru testabilitate este posibilă garantarea testabilităţii pentru erori singulare de tip stuck-at pentru toate schemele importante de checker-e incorporate.

3.2.2. Circuite de verificare incorporate cu autoverificare totală

În 1968, Carter şi Schneider au ridicat ţintele autoverificării (self-checking) şi au dezvoltat circuitul TSC (Totally Self-Checking). Un circuit TSC are mai multe intrări şi mai multe ieşiri. În timpul operării normale (fără defecţiuni) un circuit TSC nu recepţionează toate combinaţiile posibile la intrări. Recepţionează un subset al tuturor combinaţiilor posibile numit spaţiul codului de intrare (input-code space). De asemenea, nu produce toate combinaţiile posibile de ieşire ci numai un subset numit spaţiul codului de ieşire (output-code space).

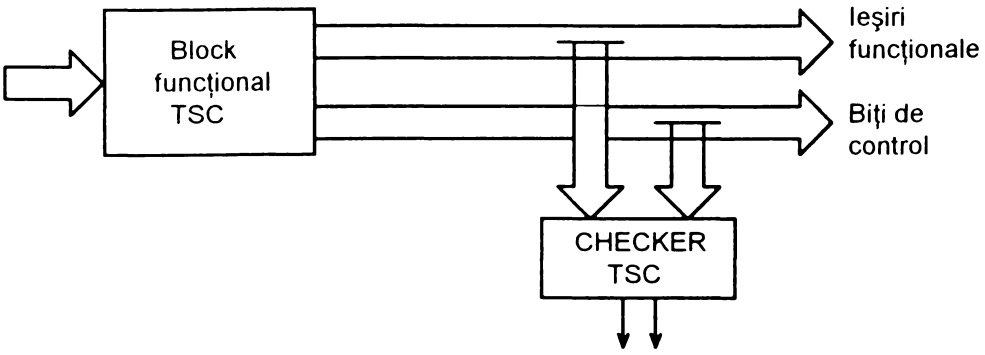


Figura 3.24 Un sistem cu autoverificare totală (totally self-checking system)

Figura 3.24 arată un sistem TSC tipic. Checker-ul TSC este un circuit combinaţional care simplifică sarcina unui observator. Dacă unitatea funcţională TSC produce o ieşire care aparţine spaţiului codului de ieşire, checker-ul TSC indică faptul

că nu există nici o eroare. O ieşire care nu aparţine spaţiului codului de ieşire se numeşte ieşire ilegală (noncode). Dacă un checker TSC găseşte o ieşire ilegală la unitatea funcţională, indică faptul că există o eroare.

O situaţie interesantă apare atunci când checker-ul are o defecţiune în sine. Din moment ce se presupune că unitatea are numai o singură defecţiune la un moment dat, se poate considera că dacă checker-ul are o defecţiune, unitatea funcţională nu are nici o defecţiune. Cu alte cuvinte, unitatea funcţională nu va produce nici o ieşire ilegală. Astfel avem nevoie de un mecanism prin care fiecare defect din interiorul checker-ului să se releve ca o eroare la ieşirea lui.

Trebuie de asemenea să se codifice ieşirile checker-ului printr-un cod detector de erori unidirecţionale. Checker-ul trebuie să producă cât mai puţine ieşiri deoarece trebuie să fie monitorizate din exterior. Cel mai mic cod nesistematic cunoscut, numit şi "cod dublă cale" (dual-rail code), este {01, 10}. O eroare unidirecţională transformă acest cod în {00 sau 11} şi astfel ieşirile checker-ului vor indica o eroare. Deci se poate detecta o defecţiune/eroare prin observarea unei ieşiri ilegale la un checker. Iar pentru a detecta o defecţiune în interior este nevoie de aplicarea unor intrări specifice (configuraţii de biţi, vectori), numite intrări de test. Astfel, se presupune în mod implicit că toate intrările de test ale unui sistem TSC sunt aplicate în modul operării normale.

Făcând o paranteză, se prezintă succint următoarele definiţii ce se întâlnesc în literatura de specialitate [GAIT85], [LALA85], [NANY88], [RAO89], [JOHN89], [CATU89], [KUND90], [BURN94]:

Un circuit se numeşte **fault-secure** (FS) pentru un set Φ de defecţiuni, dacă pentru orice defecţiune singulară $\tau \in \Phi$, circuitul produce fie o ieşire de cod aşteptată pentru intrarea respectivă, fie o ieşire ilegală (noncode), dar nu produce un cuvânt de cod neaşteptat pentru aceea intrare.

Un circuit este numit **self-testing** (ST) (cu autotestare) pentru un set Φ de defecţiuni dacă şi numai dacă pentru orice defecţiune singulară $\tau \in \Phi$, circuitul produce o ieşire ilegală (noncode) pentru cel puţin o intrare din cod.

Un circuit este numit **totally-self-checking** (TSC) dacă el este atât self-testing cât şi fault-secure pentru orice defecţiune singulară $\tau \in \Phi$.

Un circuit se defineşte drept "**code-disjoint**" (CD) dacă în absenţa defectelor, aplicând intrărilor primare un vector binar valid al codului, la ieşire se obţine de asemenea un vector binar valid al respectivului cod, aşa cum dacă aplicând intrărilor primare un vector binar ilegal pentru codul dat, la ieşire se obţine de asemenea un vector binar ilegal al respectivului cod.

Un circuit TSC este numit **TSC checker** dacă el este de asemenea code-disjoint.

Iniţial definiţia circuitelor TSC a fost dată pentru circuitele combinaţionale iar ulterior s-a extins acoperind atât cele secvenţiale [GAIT85], [NANY87a], cât şi în general sisteme întregi (care de fapt se pot considera că se comportă ca şi maşinile secvenţiale) [NANY87b], [NANY88]

Revenind din nou la checker-ul TSC, se punctează faptul că ieşirea lui devine 00 sau 11, dacă şi numai dacă, o eroare este detectată la ieşirea unei unităţi funcţionale sau dacă checker-ul în sine are o defecţiune hardware. Deci, un checker TSC trebuie să fie code-disjoint. Adesea există situaţii în care chiar dacă un checker este proiectat pentru code-disjointness, nu sunt disponibile de la unitatea funcţională TSC toate intrările de test necesare pentru testarea lui.

Să luăm, de exemplu, un sistem bazat pe un cod dublă cale, care este des utilizat în sistemele cu autoverificare (self-checking). Într-un sistem dublă cale, două linii (doi biţi) reprezintă o singură variabilă şi au valori complementare. Variabila a_1 poate fi indicată prin două linii care au semnalele corespunzătoare de a_1 şi $b_1(\overline{a}_1)$. Semnalele $a_1b_1 = 11$ şi 00 sunt semnale invalide (ilegale) şi reprezintă o eroare. Pentru a determina dacă sunt valide semnalele de la două variabile, reprezentate prin liniile a_1b_1 şi a_2b_2, se utilizează circuitul code-disjoint din figura 3.25 descris de următoarele ecuaţii:

$$f = a_1b_2 + a_2b_1, \quad g = a_1a_2 + b_1b_2$$

Tabelul 3.2 arată testele pentru circuitul din figura 3.25. Ieşirile normale sunt fg = 01 sau 10. O intrare eronată cauzează o ieşire eronată. De exemplu, dacă $a_1b_1 = 11$, atunci fg = 11 şi dacă $a_1b_1 = 00$, atunci fg = 00.

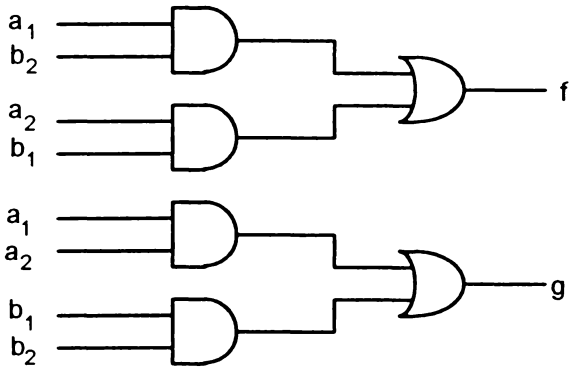


Figura 3.25 Circuit de verificare "comparator dublă cale" cu două perechi de intrări (two-rail checker).

a_1	b_1	a_2	b_2
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0

Tabelul 3.2. Testele pentru circuitul din fig. 3.25

Pentru a fi cu autotestare acest circuit de verificare dublă cale, are nevoie de intrările din tabelul 3.3, dar să presupunem că recepţionează numai trei din cele patru teste de care are nevoie aşa cum arată tabelul 3.4. În această situaţie pentru f din figura 3.25 nu se poate detecta defectul b_2 stuck-at-0 pentru că $a_1 = 1$ şi $a_2 = 0$ nu sunt disponibile. Similar, pentru g nu se pot detecta defectele a_2 stuck-at-1 sau b_1

stuck-at-1. Astfel, chiar dacă acest checker este code-disjoint el nu este cu autotestare fără cele patru teste care sunt date în tabelul 3.3.

a_1	b_1	a_2	b_2
1	0	1	0
1	0	0	1
0	1	1	0
0	1	0	1

Tabelul 3.3. Intrările necesare unui checker dublă cale pentru a fi cu autotestare.

a_1	b_1	a_2	b_2
1	0	1	0
0	1	0	1
0	1	1	0

Tabelul 3.4. Intrările posibile ale unui checker dublă cale care nu-i permit să fie cu autotestare.

Dacă circuitele care generează f şi g se fac self-testing, ecuaţiile noi pentru f şi g sunt: $f = a_2 b_1$ $g = a_1 + b_2$

Sub aceste ecuaţii, dacă $a_2 = 1$ şi $b_2 = 1$ când $a_1 = 1$ şi $b_1 = 0$, atunci $f = 0$ şi $g = 1$. Astfel, acest circuit nu este code-disjoint. Deci ne aflăm la o situaţie fără soluţie. Dacă circuitul se face code-disjoint, atunci el nu este self-testing. Dacă circuitul este făcut self-testing, atunci nu este code-disjoint. Astfel, în această situaţie nu se va obţine niciodată un checker TSC, adică un circuit care să fie atât self-testing cât şi code-disjoint.

Pentru a face un circuit atât self-testing cât şi code-disjoint ar trebui să se suspende operaţia normală din când în când pentru a alimenta (to flush) checker-ul cu intrări de test care în mod normal nu sunt disponibile. Dar în această situaţie sistemul nu mai este on-line şi există dificultăţi practice în alimentarea (flushing) checker-ului. O cale mai bună este proiectarea checker-elor care să fie atât self-testing cât şi code-disjoint, care nu au nevoie de acces din exterior şi sunt uşor de incorporat.

3.2.2.1. Un sistem TSC

Figura 3.24 arată o schemă bloc a unui sistem TSC. Biţii de control sunt aleşi de obicei în aşa fel încât ieşirile blocului funcţional să formeze un cod detector de erori unidirecţionale. În această proiectare, propusă în [KUND90] de Kundu şi Reddy, este necesar ca biţii de control să fie selectaţi pentru a permite proiectarea checker-elor TSC. Această necesitate poate implica partiţionarea setului de ieşiri funcţionale în aşa fel încât să se codifice biţii din fiecare bloc din partiţie într-un cod nesistematic. Se poate astfel proiecta un checker TSC cu două ieşiri. Checker-ul TSC descris utilizează o schemă de arbore de comparatoare dublă cale cu două intrări

Intrările unui comparator dublă cale se realizează cu două perechi de intrări (a_1, b_1) şi (a_2, b_2)

Cele două ieşiri f şi g ale comparatorului cu două intrări sunt:

$$f = a_1 b_2 + a_2 b_1, \qquad g = a_1 a_2 + b_1 b_2$$

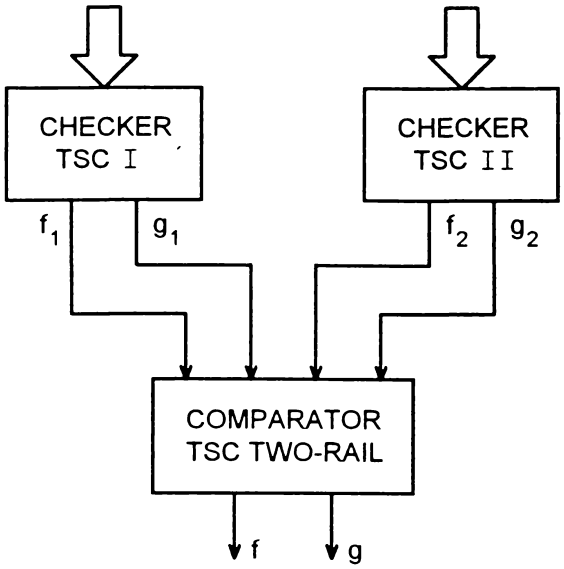


Figura 3.26 Un circuit pentru a monitoriza ieşirile a două checker-e TSC.

f_1	g_1	f_2	g_2
0	1	0	1
1	0	1	0

Tabelul 3.5. Un set posibil de ieşiri al checker-elor de la primul nivel din fig. 3.26.

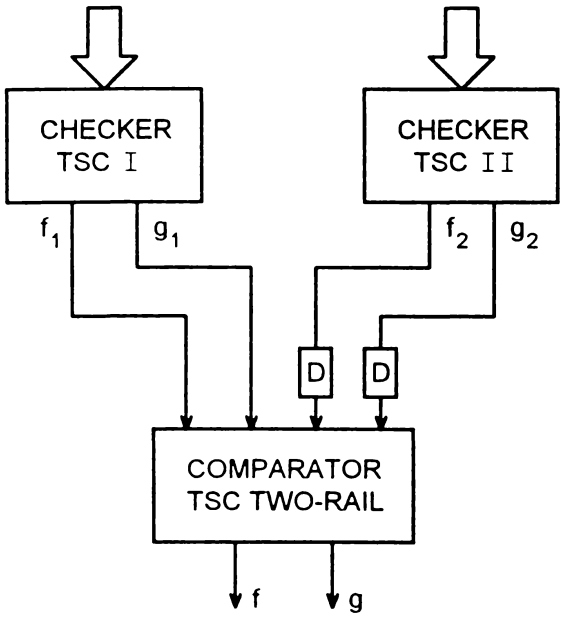


Figura 3.27 Un checker TSC pentru ieşirile din tabelul 3.5.

Majoritatea checker-elor TSC pentru coduri detectoare de erori au două ieşiri care produc în timpul funcționării normale fie 01 fie 10. Deci, se pot monitoriza ieşirile a două checker-e TSC prin utilizarea checker-ului dublă cale cu două intrări din figura 3.26. Întregul circuit din figură poate fi TSC, dacă şi numai dacă, f_1g_1 şi f_2g_2 furnizează toate cele patru teste din tabelul 3.2. De exemplu, dacă f_1g_1, f_2g_2 iau numai cele două valori din tabelul 3.5, este clar că checker-ul din figura 3.26 nu este TSC. În acest caz, se pot genera toate testele pentru comparatorul dublă cale prin inserarea unei întârzieri de tact (clock-delay) sau a bistabilelor D la ieşirile unuia dintre

checker-ele TSC aşa cum arată figura 3.27, la ieşirea oricărui checker de la primul nivel.

Se poate verifica uşor că, astfel, sunt disponibile toate testele pentru comparatorul dublă cale. Necesitatea de a avea bistabile D sincrone în checker-ul din figura 3.27 nu pune nici o problemă din moment ce bistabilele sincrone sunt părţi obişnuite din majoritatea sistemelor digitale.

3.2.2.2. Proiectarea unui sistem TSC

Fie $y_1, y_2, y_3, \cdots, y_k$ ieşirile funcţionale (vezi fig. 3.24). Astfel procedura următoare conduce la o unitate funcţională TSC şi la un checker TSC.

1. Partiţionarea ieşirilor funcţionale $y_1, y_2, y_3, \cdots, y_k$ în blocuri astfel încât biţii din fiecare bloc să poată fi codificaţi printr-un cod nesistematic pentru care există un checker normal TSC cu două ieşiri.
2. Proiectarea checker-elor TSC cu două ieşiri pentru codurile de la punctul 1.
3. Monitorizarea ieşirilor de la checker-ele TSC printr-un arbore de comparatoare dublă cale cu două intrări.

Un exemplu ilustrează această procedură de proiectare. Fie spaţiul de ieşire aşteptat al unităţii funcţionale (partea de informaţie a ieşirilor codificate) ca cel dat în tabelul 3.6. De subliniat că numai 8 din cele 32 (2^5) combinaţii posibile apar la ieşirile unităţii funcţionale.

Apoi, se aplică algoritmul dat de Kundu şi Reddy pentru a arăta cum poate fi proiectat un checker TSC pentru acest tabel de ieşire. Se partiţionează ieşirile astfel: (y_1, y_2, y_3) şi (y_4, y_5). Se codifică fiecare partiţie printr-un cod nesistematic. De notat că (y_4, y_5) sunt deja neordonate şi astfel nu necesită codificare. Se poate codifica (y_1, y_2, y_3) printr-un cod Berger prin utilizarea a doi biţi de control d_1 şi d_2. Astfel, spaţiul de ieşire codificat rezultant este cel din tabelul 3.7.

Se poate construi un checker TSC pentru partea de cod Berger urmărind rezultatele studiului făcut de M.Ashjaee şi S.Reddy. Referitor la figura 3.28, se poate observa că ieşirea checker-ului TSC pentru cod Berger cât şi y_4 şi y_5 au valorile din tabelul 3.8 sau permutaţiile de coloane O_1 şi O_2 ale acestui tabel.

Deci, pentru a monitoriza aceste ieşiri, un comparator dublă cale necesită întârzierea uneia din perechile de intrare (aşa cum se observă în figura 3.28).

Intrare	y_1	y_2	y_3	y_4	y_5
1	0	0	0	1	0
2	0	1	0	0	1
3	0	0	1	1	0
4	1	1	0	0	1
5	1	0	1	0	1
6	0	1	1	0	1
7	1	1	1	1	0
8	1	0	0	0	1

Tabelul 3.6. O mostră de spaţiu de ieşire aşteptat pentru o unitate funcţională.

Intrare	y_1	y_2	y_3	d_1	d_2	y_4	y_5
1	0	0	0	1	1	1	0
2	0	1	0	1	0	0	1
3	0	0	1	1	0	1	0
4	1	1	0	0	1	0	1
5	1	0	1	0	1	0	1
6	0	1	1	0	1	0	1
7	1	1	1	0	0	1	0
8	1	0	0	1	0	0	1

Tabelul 3.7. Spațiul de ieşire codificat pentru partea de cod Berger dintr-un checker TSC.

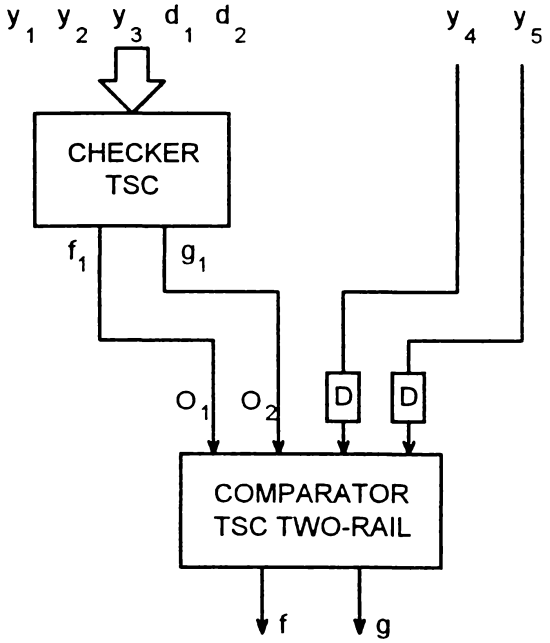


Figura 3.28 Un checker TSC care utilizează întârzieri şi verifică pentru concordanța între m biți, cei mai semnificativi dintr-un cuvânt de cod şi cei generați.

Intrare	O_1	O_2	y_4	y_5
1,7	1	0	1	0
2,4,5,6,8	0	1	0	1
3	0	1	1	0

Tabelul 3.8. Valorile de ieşire ale unui checker TSC pentru codul Berger împreună cu y_4 şi y_5.

Pentru a dovedi aplicabilitatea acestei metode la toate sistemele este nevoie să se satisfacă două condiții. Prima este să se arate că fiind date orice ieşiri funcționale $y_1, y_2, y_3, \cdots, y_k$, se pot aplica întotdeauna punctele 1 şi 2 din procedura de proiectare descrisă anterior. A doua condiție este aceea de a arăta că fiind dată orice pereche de doi biți de intrare (a_1, b_1), (a_2, b_2) astfel încât $a_i = \overline{b}_i$, unde i = 1,2, se poate construi întotdeauna un comparator TSC dublă cale cu două intrări pentru a monitoriza (a_1, b_1), (a_2, b_2). Prima condiție se poate satisface simplu prin codificarea fiecărui y_i într-un cod dublă cale prin adăugarea unui bit de control, $\overline{y}_i$ pentru fiecare y_i. Se

poate arăta că a doua condiție este satisfăcută dacă nici o ieşire funcțională nu este o constantă. Aceasta înseamnă că nici un y_i nu este tot timpul 0 sau 1.

Rezultatele procedurii descrise au multe aplicații. Una dintre acestea implică proiectarea checker-elor TSC pentru codurile A× N aritmetice. O altă aplicație implică un şir logic iterativ TSC, sau ILA (Iterative Logic Array).

3.2.2.3. Checker-e TSC pentru coduri AxN

Codurile A× N sunt coduri aritmetice de detecție/corecție a erorilor şi sunt utilizate în unitățile aritmetice şi logice ale CPU-urilor sistemelor de calcul. Ideea de bază a codurilor aritmetice este foarte simplă. În loc de a utiliza un număr i, unde $0 \le i \le N$, se utilizează numărul i× A, unde A este o constantă aritmetică. De exemplu, dacă intervalul numerelor este de la 0 la 255 (8 biți) şi A = 3, atunci 0 este notat cu 0, 1 cu 3, 2 cu 6 ş.a.m.d., 255 cu 3 x 255 = 765. Astfel, cei 8 biți de informație sunt codificați ca o configurație binară pe 10 biți, din moment ce avem nevoie de 10 biți pentru a reprezenta orice număr mai mare decât 511 şi mai mic decât 1024. De fapt, lungimea binară (numărul de biți) a unui cod A× N este lungimea lui N plus lungimea lui A. Pentru a reprezenta lungimea lui N se utilizează n, iar pentru lungimea lui A se utilizează m.

La proiectarea unui checker TSC pentru coduri aritmetice A× N, apare următoarea problemă: fiind dat un vector binar de lungime m+n, cum se poate determina dacă îl putem divide cu A şi cum să se informeze mediul exterior. Soluția există pentru valori ale lui $A = 2^m - 1$ dar nu este cunoscută pentru alte valori ale lui A diferite de $2^m - 1$ [NIKO88], [KUND90]. În [GAIT83b], se prezintă un checker TSC pentru coduri aritmetice A× N în cazul special când A=3. Prin aplicarea conceptelor descrise mai înainte se poate proiecta un checker pentru coduri aritmetice A× N pentru majoritatea valorilor lui A.

Când se înmulțeşte un număr N de n biți cu un număr A de m biți se obține un cuvânt de m+n biți. În proiectarea propusă de Kundu şi Reddy se selectează n biți, cei mai puțin semnificativi, de la un astfel de vector binar pentru a genera complementele a m biți, cei mai semnificativi. Un checker dublă cale TSC care utilizează întârzieri trebuie să verifice concordanța dintre cei m biți, cei mai semnificativi, ai cuvântului de cod şi biții cei mai semnificativi generați. Figura 3.29 ilustrează acest lucru.

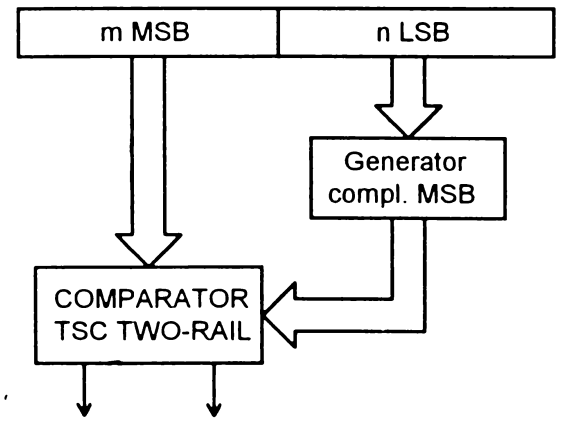


Figura 3.29 Un checker TSC pentru coduri AxN.

Pentru a se asigura că blocul pentru generarea biților cei mai semnificativi (MSB) este self-testing trebuie să se verifice că sunt disponibile toate combinațiile binare pentru ultimii n biți. În această situație se garantează că logica combinațională care generează complementele biților MSB este testabilă.

Pentru a se garanta această proprietate de self-testing mai întâi se arată că dacă N_1 şi N_2 sunt două numere diferite, atunci $AN_1 \bmod 2^n \neq AN_2 \bmod 2^n$. Astfel, dacă AN are 2^n cuvinte, atunci AN mod 2^n are 2^n cuvinte, care dovedeşte că toate cele 2^n combinații sunt posibile pentru cei n biți mai puțin semnificativi (LSB). Pentru a arăta că $AN_1 \bmod 2^n \neq AN_2 \bmod 2^n$ se presupune prin absurd că $AN_1 \bmod 2^n = AN_2 \bmod 2^n$. Atunci, $(AN_1 - AN_2) \bmod 2^n = 0$, ceea ce înseamnă că 2^n divide fie pe A fie pe $(N_1 - N_2)$. Nu se iau în considerare codurile AxN unde A este divizibil la 2^n pentru că dacă A ar fi fost divizibil la 2^n, atunci s-ar fi putut scrie A ca $A = r \cdot 2^n$. Aceasta este la fel ca şi înmulțirea unui număr N_1 cu r şi adăugarea a n 0-uri la dreapta reprezentării binare a acestui produs. Adăugarea 0-urilor la dreapta acestor vectori binari nu îmbunătățeşte capacitățile lor de detecție/corecție a erorilor, astfel se poate considera că 0-urile nu se adaugă niciodată în această formă.

Dacă 2^n nu divide pe A, atunci el trebuie să dividă pe $(N_1 - N_2)$. Dar diferența este mai mică decât 2^n. Contradicție. Astfel, se ajunge la concluzia că fiecare număr N_1 pe n biți are o imagine unică de n biți prin $AN_1 \bmod 2^n$.

Pentru că checker-ul dublă cale să fie self-testing, codul nu poate avea cei mai semnificativi biți ca şi constante. Această necesitate se realizează automat, din moment ce dacă orice bit din cuvântul de cod este o constantă cunoscută, el se poate şterge fără a reduce capacitatea de detecție/corecție a erorilor.

Proiectarea checker-elor TSC pentru coduri A× N arată că prin utilizarea procedurilor descrise se pot obține checker-e cu autoverificare totală (totally self-checking) fără utilizarea biților de control adiționali care ar fi necesari pentru a obține gradul dorit de detecție a erorii.

3.2.2.4. Proiectarea TSC ILA

Un circuit de logică combinațională furnizează configurația prezentă la ieşirile lui bazându-se numai pe starea prezentă la intrările lui. Pe de altă parte, un circuit secvențial furnizează configurația ieşirilor lui bazându-se atât pe intrările lui prezente cât şi pe starea lui prezentă. Starea prezentă a unui circuit este determinată de intrările lui trecute şi sunt adăugate în memoria circuitului. Un circuit cu logică iterativă se poate înțelege bine, dacă îl considerăm că mai multe copii ale unui circuit secvențial fără elementele lui de memorare.

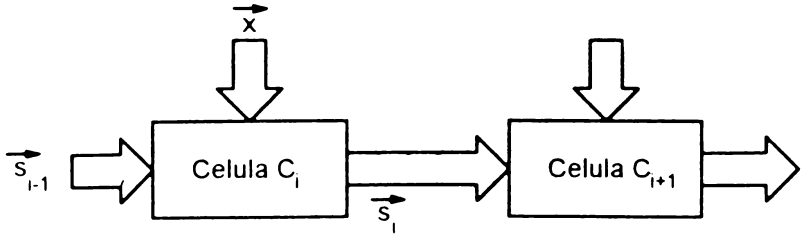


Figura 3.30 Un şir logic iterativ unidimensional (ILA) (Iterative Logic Array)

Informația stării prezente a circuitului vine de la circuitul vecin ca şi informația stării următoare pentru acel circuit. Astfel, intrările eşantionate de timp ale unui circuit secvențial sunt adăugate în eşantionul curent dar sunt comandate de o secvență spațială. Figura 3.30 ilustrează această ordine. Dacă se codifică informația stării următoare (informația celulei următoare) atunci se poate proiecta un TSC ILA.

Kolupaev a fost primul care s-a ocupat de această problemă şi a definit ceea ce este cunoscut sub forma generalizată ca circuite cu autoverificare (self-checking circuits) care includ circuitele TSC ca şi un caz particular. Dhawan şi De Vris au găsit câteva neajunsuri în proiectarea lui Kolupaev şi au propus o proiectare nouă pentru circuite ILA self-checking. S.Kundu şi S.Reddy consideră că următoarele presupuneri ale lui Dhawan şi De Vris sunt greu de justificat.

1. O singură defecțiune nu conduce la erori multiple;
2. Pentru informația stării următoare (celulei următoare), toate combinațiile sunt posibile. Ceea ce înseamnă că dacă se utilizează trei biți pentru informația celulei următoare, atunci sunt posibile $2^3 = 8$ stări.

Prima presupunere, nu este în general corectă din moment ce o defecțiune la un nivel de intrare primar se poate manifesta într-un număr mai mare de erori. A doua presupunere ignoră cazurile în care numărul de stări nu este de forma 2^k. Pentru a ilustra acest lucru considerăm tabelul de tranziții de stări cu trei stări ca în tabelul 3.9. Pentru a codifica trei stări este nevoie de cel puțin doi biți. Din moment ce doi biți pot reprezenta patru stări, iar în cazul nostru există numai trei, se pot atribui valori multiple la o singura stare şi se asigură apariția tuturor combinațiilor binare posibile. O atribuire posibilă este:

$S_1 \rightarrow 00,\ 01$

$S_2 \rightarrow 10$

$S_3 \rightarrow 11$

Tabelul 3.10 reprezintă tabelul tranzițiilor.

Dhawan şi De Vris sugerează că trebuie combinate celulele consecutive ale lui ILA pentru a creşte numărul de stări pentru a satura ieşirile de informație a stării următoare. S.Kundu şi S.Reddy afirmă că nu este nici un avantaj pentru a face aşa ceva. Chiar dacă s-ar atinge acest scop, procesul ar putea schimba limitele funcțiilor care nu sunt tot timpul cele dorite.

	0	**1**
S_1	S_1	S_2
S_2	S_3	S_1
S_3	S_2	S_3

Tabelul 3.9. Tabelul de tranziții cu trei stări.

	0	**1**
00	01	10
01	00	10
10	11	00
11	10	11

Tabelul 3.10. Tabelul de tranziții atunci când $S_1 = 00,\ 01$; $S_2 = 10$, şi $S_3 = 11$.

Pentru aceste motive S. Kundu şi S. Reddy propun metoda lor de proiectare a unui TSC ILA. Figura 3.30 arată un ILA unidimensional. O celulă oarecare C_i este alimentată de intrările externe $x_1, x_2, \cdots, x_n$ reprezentate prin $\bar{x}$ şi intrările de informaţii legate de stare (numite şi intrări interne) $s_{(i-1)1}, s_{(i-1)2}, \cdots, s_{(i-1)k}$ reprezentate prin $\bar{s}_{(i-1)}$. Această celulă produce ieşirile de stare $s_{i1}, s_{i2}, \cdots, s_{ik}$ reprezentate prin $\bar{s}_i$. Tehnica de proiectare propusă se bazează pe următoarele presupuneri legate de proiectarea TSC ILA:

1. $\bar{x}$ este fără erori, şi poate fi controlat din exterior;
2. Sunt permise numai defectele singulare de tip stuck-at;
3. Pentru orice celulă C_i $\bar{s}_{(i-1)}$ şi $\bar{s}_i$ sunt coduri detectoare de erori unidirecţionale.
4. Numai erorile unidirecţionale pot apărea din moment ce circuitele pot fi proiectate fără inversare.
5. Nu sunt necesare toate combinaţiile de intrări.
6. Ieşirea unui checker TSC aparţine setului {01, 10} în cadrul operării fără defecte. 00 şi 11 sunt ieşiri care indică defecţiuni.

Fiecare C_i din proiect are un checker care verifică integritatea lui $\bar{s}_{(i-1)}$ şi produce ieşiri care indică eroarea. Se codifică $\bar{s}_{(i-1)}$ utilizând tehnicile descrise mai sus. Prin combinarea ieşirilor indicatoare de erori din toate C_is, utilizând metoda descrisă mai sus, se obţin ieşirile finale indicatoare de erori.

Într-adevăr on-line checking este o problemă foarte dificilă, precum şi proiectarea checker-elor TSC fără a pune condiţii în plus în sistem.

3.2.2.5. Obiectivul proiectării TSC

Conform celor discutate anterior un circuit TSC, ale cărui ieşiri sunt codificate într-un cod detector de erori, produce întotdeauna un cuvânt ilegal ca primă ieşire eronată din cauza unei defecţiuni. Din moment ce acest comportament este foarte avantajos şi de dorit este cunoscut sub denumirea de obiectivul proiectării TSC (TSC goal). Există totuşi şi alte clase de circuite logice care îndeplinesc acest obiectiv.

Proprietatea de strongly fault-secure caracterizează o clasă mai largă de circuite logice care realizează acest obiectiv sub aceleaşi premize privitoare la defecţiunile care se iau în considerare. Această proprietate se defineşte în felul următor:

Un circuit (sistem) este **strongly fault-secure** (SFS) pentru un set Φ de defecţiuni, dacă pentru orice defecţiune singulară $\tau \in \Phi$, circuitul (sistemul) este fie

a) TSC pentru $\{\tau\}$, fie

b) FS pentru $\{\tau\}$, şi dacă defectul τ apare, atunci circuitul (sistemul) rezultant este tot SFS pentru $\Phi-\{\tau\}$[NANY88], [RAO89].

Din punct de vedere al proiectantului este mai avantajoasă proiectarea circuitelor SFS care îndeplinesc obiectivul TSC faţă de cea a circuitelor TSC pentru

că cele din urmă impun proiectării mai multe restricții. De asemenea, chiar dacă proiectarea sistemelor fail-safe a fost studiată înainte şi independent de cea a circuitelor (sistemelor) TSC şi SFS, circuitele fail-safe sunt echivalente cu circuitele fault-secure cu o anumită codificare. Figura 3.31 prezintă relația între clasele de circuite TSC, SFS, şi Fail-safe.

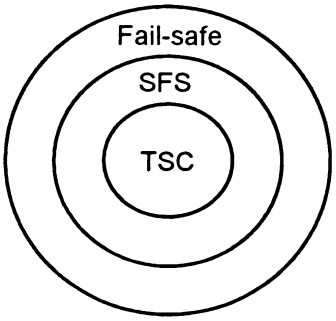


Figura 3.31 Relația între circuite TSC, SFS, şi Fail-safe.

O altă proprietate interesantă se defineşte după cum urmează:

Un circuit este "**strongly code-disjoint**" (SCD) pentru un set Φ de defecțiuni, dacă pentru orice defecțiune singulară $\tau \in \Phi$, circuitul este fie

a) CD, şi este ST pentru $\{\tau\}$, fie

b) CD, şi dacă defectul τ apare, atunci circuitul rezultant este tot SCD pentru $\Phi-\{\tau\}$[NANY88], [RAO89].

Combinația dintre un circuit TSC (care realizează o anumită funcție) şi un checker TSC aparține clasei rețelelor logice cu componente TSC. Aceste rețele îndeplinesc obiectivul TSC sub presupunerea defecțiunilor singulare, ceea ce înseamnă că defecțiunile apar una câte una, iar între două defecțiuni este suficient timp astfel încât toate intrările din cod să fie aplicate rețelei. Sub presupunerea defecțiunilor duble, prin care se înțelege că între apariția a două defecțiuni care afectează circuitul care realizează funcția dorită, sau între apariția a două defecțiuni care afectează checker-ul, există suficient timp astfel încât toate intrările din cod să fie aplicate rețelei, rețelele de mai sus nu îndeplinesc obiectivul TSC. În [GAIT88c], Gaitanis prezintă o nouă clasă de circuite de verificare TSC cu proprietăți mai puternice de autoverificare (self-checking) pentru a îndeplini obiectivul TSC în cazul rețelelor de checker-e când se iau în considerare defecțiunile duble.

Un TSC checker este **strongly self-checking** (SSC) pentru un set Φ de defecțiuni, dacă Fc, Fe, şi E sunt submulțimi disjuncte ale mulțimii de ieşiri ilegale [GAIT88c], unde:

Fc este mulțimea ieşirilor neaşteptate cauzate de defecțiuni interne $\tau \in \Phi$ şi intrări din cod ale checker-ului TSC,

Fe este mulțimea ieşirilor neaşteptate cauzate de defecțiuni interne $\tau \in \Phi$ şi intrări din afara codului (ilegale) ale checker-ului TSC,

E este mulțimea ieşirilor produse de checker-ul TSC fără defecțiuni cauzate de intrările ilegale.

4. Autocontrolul operației de împărțire binară

În cadrul acestui capitol, autorul urmăreşte să scoată în relief problematica aplicării autocontrolului asupra operației de împărțire binară. Față de alte operații aritmetice fundamentale sau operații logice s-a preferat operația de împărțire, pentru că ea este operația care necesită cel mai mult timp de execuție şi în plus este foarte greu de controlat. Metodele de control ale operațiilor aritmetice şi logice bazate pe coduri aritmetice sau coduri ciclice (semnături) [CRIŞ93] sunt inaplicabile sau aplicarea lor implică costuri prohibitive în cazul operației de împărțire din cauza naturii operației. Un alt fel de control mai sofisticat sau mai pretențios axat pe execuția operației inverse ar prelungi şi mai mult timpul de execuție al operației de împărțire. Din aceste considerente s-a ajuns la concluzia aplicării controlului de paritate asupra operației de împărțire binară. De asemenea, încă un motiv în favoarea acestei alegerii este faptul că prin paritate se pot controla relativ uşor toate operațiile aritmetice şi logice precum şi cele de transfer, asigurându-se unicitatea controlului într-un sistem de calcul numeric.

În continuare, se prezintă sintetic problematica operației de împărțire pentru acomodarea cititorului în acest domeniu, pe urmă se stabileşte o metodă şi se defineşte la nivel de cel mai mic detaliu o schemă pentru grefarea şi exemplificarea aplicării parității.

4.1. Teoria de bază pentru operația de împărțire

Metodologiile pentru calcularea unei împărțiri se pot clasifica în mare, în iterative sau recursive (ca metoda lui Newton cu tangente şi algoritmul lui Goldschmidt prezentate în subcapitolele 4.1.2.1.1. şi 4.1.2.1.2.) şi în algoritmii cifră cu cifră (digit-by-digit) care sunt foarte asemănători cu cunoscuta şi uzuala tehnică creion-hârtie. Ne vom concentra atenția în aria metodelor cifră cu cifră (cunoscute şi sub numele metode directe) datorită faptului că ele sunt mai potrivite pentru implementări în VLSI .

Există în literatura de specialitate multe articole atât pentru împărțirea convențională cât şi pentru on-line, precum şi împărțirea combinată cu extracția de rădăcină. Metodele directe calculează rezultatul final prin efectuarea unor iterații, unde o cifră nouă a rezultatului (care depinde de valoarea curentă a restului parțial şi de împărțitor) este calculată în concordanță cu (în funcție de) un set de reguli de selecție dintr-un tabel pentru selecția cifrei. După aceea se actualizează restul parțial conform cifrei selectate ca să fie pregătit pentru următoarea iterație. În proiectarea algoritmilor de împărțire punctele cheie sunt tehnicile pentru selecția cifrei şi pentru actualizarea restului parțial. În general pentru actualizarea restului parțial se utilizează [carry-free addition] şi reprezentarea redundantă a restului. Se pot obține îmbunătățiri ale performanțelor prin suprapunerea execuției unor părți din fiecare iterație utilizând baze (radices) mari sau foarte mari [MONT94]. O altă tehnică este prescalarea (prescaling) introdusă în [9] chiar dacă şi alte lucrări au fost prezentate utilizând transformarea rangului operanzilor (operand range transformation). Ideea este să se efectueze manipulări simple cu ambii operanzi astfel încât să nu se altereze valoarea câtului, iar împărțitorul să fie limitat în cadrul unui interval îngust apropiat de 1; în felul acesta, partea cea mai semnificativă a restului parțial furnizează noul bit al câtului.

Numărul mare de valori posibile de cifre poate fi folositor pentru obținerea unor reguli mai simple de selecție a cifrei, pentru că mai multe posibilități sunt permise pentru ajustarea estimării câtului în pașii următori, fiind dat span-ul sporit al fiecărei

cifre. Pe de altă parte cu cât setul cifrei este mai larg cu atât actualizarea restului parțial este mai complicată, pentru că un număr mai mare de multipli ai împărțitorului trebuie generat pentru a acoperi toate selecțiile posibile ale unei valori de cifră. Astfel trebuie făcută o selecție atentă a valorilor de cifră; o soluție specifică pentru împărțirea în baza patru (radix-4) este discutată în detaliu în [MONT94], pentru a arăta un exemplu practic de împărțire cu reprezentarea supraredundantă a câtului (over-redundant).

O metodologie care combină utilizarea seturilor de cifre supraredundante cu prescalarea împărțitorului este de asemenea studiată în [MONT94], pentru a obține unități de împărțire radix-B cu funcții de selecție a cifrei triviale.

Un alt aspect al tehnicii de împărțire influențate de alegerea reprezentării supraredundante este conversia în "zbor" (on-the-fly) a rezultatului redundant la forma neredundantă. De fapt o cifră cu semn care vine, poate influența rezultatul calculat anterior într-o manieră mai extinsă, din moment ce valoarea maximă absolută posibilă este mai mare decât la celelalt algoritm de conversie. O modificare în metoda pentru reprezentare simplu redundantă este prezentată ca să permită de asemenea on-the-fly conversion pentru numere supraredundante.

De fapt, algoritmul pentru calcularea împărțirii X/Y în baza $B=2^b$ se bazează pe următoarea formulă:

$$w_i = B(w_{i-1} - q_i Y) = B^{i+1}(X - Q_i Y) \quad \text{şi} \quad Q_i = Q_{i-1} + B^{-i} q_i \qquad (4.1)$$

unde: i este pasul de iterație; w_{i-1} este valoarea restului parțial (deplasat) la pasul i-1 cu $w_0 = BX$; q_i este cifra rezultatului parțial dezvoltat care a fost calculat la iterația i a relației de mai sus; Δ_s este setul de cifre, adică setul de valori posibile pentru q_i; $\Delta_s = \{-s, -s+1, \ldots, 0, +1, \ldots, +s\}$ (s este întreg şi $q_i \in \Delta_s$); Q_i este valoarea câtului după iterația i, cu $Q_0 = 0$. Pentru acest algoritm s-a definit factorul de redundanță:

$$\eta = \frac{s}{B-1}. \qquad (4.2)$$

Pentru împărțire, X şi Y sunt de obicei normalizate la $1/2 \le X < Y < 1$. Acest lucru face ca rezultatul Q=X/Y sa fie normalizat la $1/2 < Q < 1$. Se presupune că X şi Y sunt disponibile în forma lor de precizie completă [şi in carry assimilated form] la începutul iterațiilor, astfel excluzând din această analiză algoritmii on-line. Chiar mai mult decât atât, resturile parțiale w_{i-1} sunt stocate în forma redundantă (carry save sau cifră cu semn).

4.1.1. Împărțirea cu operanzi întregi

Cea mai simplă schemă de împărțire (împărțitor) operează cu numere fără semn şi produce câte un bit la fiecare pas de împărțire. Figura 4.1 prezintă un astfel de împărțitor binar cablat (hardware). Pentru a se calcula X/Y, se introduce deîmpărțitul X în registrul Q, împărțitorul Y în registrul M, se resetează registrul A, şi apoi se procedează în felul următor:

1 Se deplasează perechea de registre AQ spre stânga cu un bit.
2 Se scade conținutul registrului M din conținutul registrului A.

3. Dacă rezultatul pasului 2 este negativ, se poziționează bitul cel mai puțin semnificativ al registrului Q pe 0, în caz contrar se poziționează pe 1.
4. Dacă rezultatul pasului 2 este negativ, se reface valoarea anterioară a registrului A prin adăugarea conținutului registrului M la A.

După procesarea acestor pași de *n* ori, registrul Q va conține câtul iar registrul A va conține restul împărțirii. Acest algoritm este versiunea binară a clasicei metode creion-hârtie.

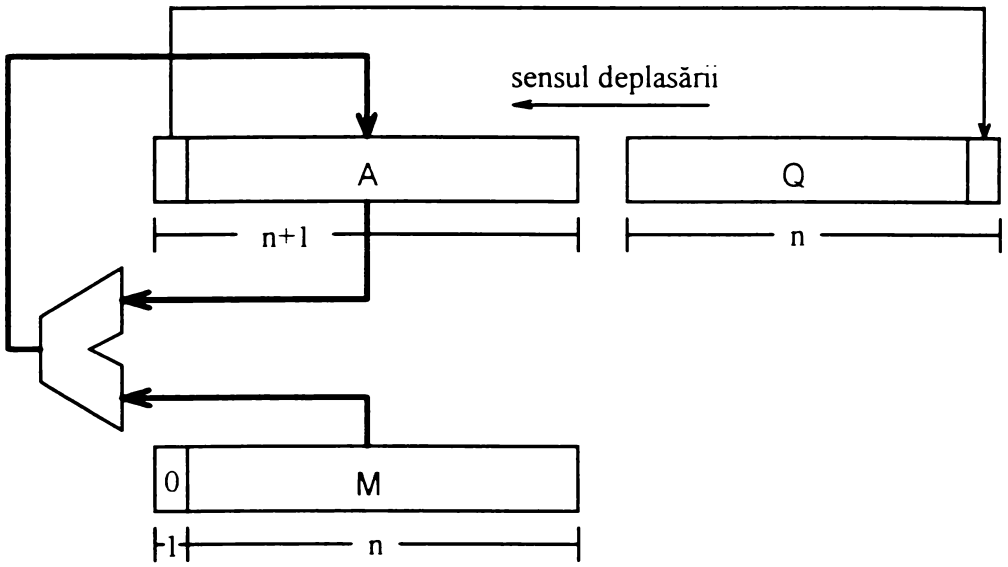


Figura 4.1 Schema bloc a unei scheme simple de împărțire pentru numere întregi fără semn reprezentate pe *n* biți. Fiecare pas de împărțire implică mai întâi deplasarea la stânga a registrelor A și Q cu câte un bit, scăderea lui M din A, și dacă diferența este pozitivă, stocarea ei în A. Dacă diferența este pozitivă, bitul cel mai puțin semnificativ a lui A se setează în 1.

De asemenea, algoritmul de împărțire descris mai sus este numit "cu refacerea restului", pentru că dacă scăderea lui Y (conținut în registrul M) din conținutul registrului A produce un rezultat negativ, registrul A se restaurează, se reface prin adăugarea lui Y. Pasul 4, de restaurare, poate fi totuși eliminat pe baza următorului considerent teoretic [PATT90]: Fie R conținutul perechii de registre AQ. Fiecare pas al algoritmului calculează valoarea 2R-Y, stocând partea mai semnificativă a acestei diferențe în registrul A, și partea mai puțin semnificativă în registrul Q. Presupunem că rezultatul unei iterații este negativ. În mod normal, se adaugă valoarea Y (obținând 2R), se deplasează spre stânga cu un bit (obținând 4R), și se continuă operația scăzând din nou pe Y din conținutul registrelor AQ (obținând 4R-Y). Să presupunem acum că nu s-a restaurat conținutul lui A, iar operația continuă. Mai întâi, se deplasează la stânga valoarea nerestaurată 2R-Y, devenind 4R-2Y, iar pe urmă se adaugă valoarea Y obținând 4R-Y, fiind exact la fel cu ce am obține dacă s-ar fi restaurat conținutul lui A. Astfel, algoritmul de împărțire fără refacerea restului este:

Dacă A este negativ,
    1a. Se deplasează perechea de registre AQ spre stânga cu un bit.
    2a. Se adaugă conținutul registrului M la conținutul registrului A.
dacă nu,
    1a. Se deplasează perechea de registre AQ spre stânga cu un bit.
    2a. Se scade conținutul registrului M din conținutul registrului A.
și în final,

3. Dacă A este negativ, se poziționează bitul cel mai puțin semnificativ al registrului Q pe 0, în caz contrar se poziționează pe 1.

După procesarea acestor paşi de n ori, câtul se va afla în registrul Q. Dacă registrul A este pozitiv atunci conține restul împărțirii, iar dacă este negativ, trebuie refăcut prin adăugarea conținutului registrului M ca să se formeze restul împărțirii. Este de notat faptul că semnul registrului A trebuie testat înainte de deplasarea lui din moment ce bitul de semn se pierde după deplasare.

Dacă X şi Y sunt numere fără semn din intervalul închis $[0, 2^n - 1]$ atunci registrul A trebuie extins la n+1 biți, pentru schema de împărțire, în vederea păstrării semnului. În consecință şi sumatorul trebuie să fie pe n+1 biți.

Notă: Înainte de a începe operația de împărțire, hardware-ul trebuie să verifice dacă cumva împărțitorul este zero.

4.1.2. Împărțire şi rest în virgulă flotantă

4.1.2.1. Împărțirea iterativă

S-a discutat mai înainte un algoritm pentru împărțirea cu numere întregi. Convertirea acestuia într-un algoritm de împărțire cu virgulă flotantă este similară cu convertirea algoritmului de înmulțire pentru numere întregi la unul cu numere în virgulă flotantă. Dacă numerele care trebuie împărțite sunt $s_1 2^{e_1}$ şi $s_2 2^{e_2}$ atunci schema de împărțire (divider) va calcula s_1 / s_2 iar răspunsul final va fi acest cât înmulțit cu $2^{e_1 - e_2}$. Referitor la figura 4.1 alinierea operanzilor este puțin diferită față de cea de la împărțirea cu numere întregi. Se încarcă s_2 în registrul M şi $s_1 / 2$ în registrul A astfel încât s_1 este deplasat la dreapta cu un bit. Pentru a simplifica notațiile renumim s_2 cu b şi s_1 cu a. Atunci algoritmul de împărțire pentru numere întregi poate fi utilizat şi rezultatul va fi de forma $q_0.q_1....$ În cazul împărțirii în virgulă flotantă, registrul Q nu este utilizat pentru a păstra operanzii. Pentru a rotunji, pur şi simplu se calculează încă doi biți adiționali ai câtului, garda şi rotunjirea (guard and round), şi se utilizează restul ca un bit de alipire (sticky bit). Bitul de gardă este necesar pentru că s-ar putea ca primul bit al câtului să fie zero. Oricum, din moment ce numărătorul şi numitorul sunt normalizați, nu este posibil ca cei doi biți semnificativi ai câtului să fie pe zero în acelaşi timp.

4.1.2.1.1. Iterația lui Newton

Există o altă metodă pentru împărțirea bazată pe iterații. În primul rând se vor descrie cei doi algoritmi iterativi principali şi apoi se va discuta despre pro şi contra iterației în comparație cu algoritmii direcți. Există o tehnică generală pentru a construi algoritmi iterativi, numită iterația lui Newton prezentată în figura 4.2.

Mai întâi se consideră problema sub forma de a găsi punctul zero al unei funcții. Apoi, pornind de la ó presupunere pentru zero se aproximează funcția prin intermediul tangentei la această presupunere şi se formează o nouă presupunere bazată pe punctul

unde tangenta are un zero. Dacă x_i este o presupunere la un zero, atunci linia tangentei are ecuația: $y - f(x_i) = f'(x_i)(x - x_i)$.

Această ecuație are un zero la: $x = x_{i+1} = x_i - \dfrac{f(x_i)}{f'(x_i)}$.

Pentru a reconsidera împărțirea ca şi căutarea de zero a unei funcții, se consideră $f(x) = 1/x - b$. Din moment ce zero-ul acestei funcții este la $1/b$, aplicând iterația lui Newton va da o metodă iterativă de calculare a lui $1/b$ din b. Utilizând $f'(x) = -1/x^2$ ecuația dinainte devine:

$$x_{i+1} = x_i - \frac{1/x_i - b}{-1/x_i^2} = x_i + x_i - x_i^2 b = x_i(2 - x_i b). \qquad (4.3)$$

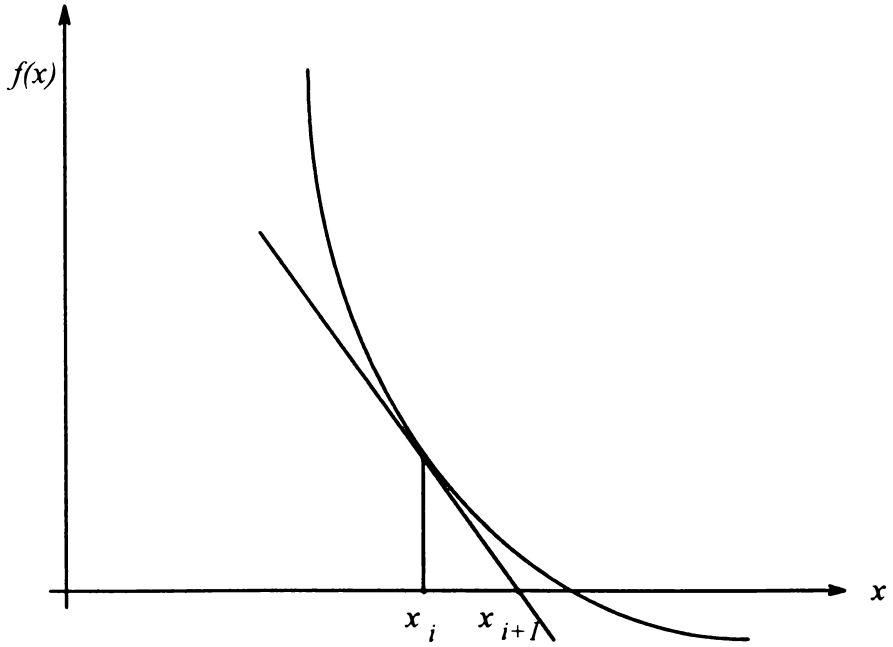


Figura 4.2 Iterația lui Newton pentru găsirea de zero. Dacă x_i este o estimare pentru zero a lui f atunci x_{i+1} este o estimare mai bună. Pentru a calcula x_{i+1} se localizează intersecția axei x cu tangenta lui f în punctul x_i.

Astfel am putea implementa calculul *a/b* utilizând următoarea metodă:

1 Scalarea lui b ca să se potrivească în intervalul $1 \le b < 2$ şi obținerea unei valori aproximative a lui $1/b$ (numit x_0) utilizând un tabel.

2. Se iterează $x_{i+1} = x_i(2 - x_i b)$ până când se ajunge la un x_n care este suficient de precis.

3 Se calculează $a\, x_n$ şi se inversează scalarea făcută în pasul 1.

Problema care se pune, este de câte ori trebuie să fie iterat pasul 2. Pentru a spune că x_i este precis la p biți înseamnă că $(x_i - 1/b)/(1/b) = 2^{-2p}$ iar simpla

manipulare algebrică arată că $(x_{i+1} - 1/b)/(1/b) = 2^{-2p}$. Astfel numărul de biți corecți se dublează la fiecare pas. Iterația lui Newton este cu autocorecție în sensul că o eroare în x_i în realitate nu contează. Aceasta înseamnă că x_i este tratat ca o estimare a lui $1/b$ şi returnează x_{i+1} ca o îmbunătățire a lui (aproape dublând cifrele). Un lucru care poate cauza ca x_i să fie în eroare este eroarea de rotunjire. Totuşi, este mai important ca în primele iterații să se profite de avantajul că nu sunt de aşteptat mulți biți corecți efectuând înmulțirea în precizie redusă, astfel câştigăm viteză fără să sacrificăm acuratețea.

4.1.2.1.2. Algoritmul lui Goldschmidt

A doua metodă de împărțire iterativă este numită algoritmul lui Goldschmidt. Ea se bazează pe ideea că pentru a calcula a/b ar trebui înmulțit numărătorul şi numitorul cu un număr r cu care $rb \approx 1$. Mai detaliat, fie $x_0 = a$ şi $y_0 = b$. La fiecare pas se calculează $x_{i+1} = r_i x_i$ şi $y_{i+1} = r_i y_i$. Astfel câtul $x_{i+1}/y_{i+1} = x_i/y_i = a/b$ este constant. Dacă luăm r_i astfel încât $y_{i+1} \to 1$, atunci $x_{i+1} \to a/b$, astfel x_{i+1} converge spre răspunsul dorit. Această idee poate fi utilizată şi pentru calcularea altor funcții. De exemplu pentru a calcula rădăcina pătrată a lui a, fie $x_0 = a$ şi $y_0 = a$, şi la fiecare pas se calculează $x_{i+1} = r_i^2 x_i$, $y_{i+1} = r_i y_i$. Astfel $x_{i+1}/y_{i+1}^2 = x_i/y_i^2 = 1/a$, astfel încât dacă r_i este ales pentru a conduce $x_{i+1} \to 1$, şi atunci $y_{i+1} \to \sqrt{a}$. Această tehnică este utilizată în calcularea rădăcinilor pătrate la TI8847 [PATT90].

Revenind la algoritmul de împărțire Goldschmidt, se presupune că $x_0 = a$ şi $y_0 = b$, şi că $b = 1 - \delta$, unde $|\delta| < 1$. Dacă se ia $r_0 = 1 + \delta$, atunci $y_1 = r_0 y_0 = (1+\delta)(1-\delta) = 1 - \delta^2$. Iar în continuare se ia $r_1 = 1 + \delta^2$, astfel încât $y_2 = r_1 y_1 = 1 - \delta^4$, ş.a.m.d. Din moment ce $|\delta| < 1$, atunci $y_{i+1} \to 1$. Cu această alegere a lui r_i, x_{i+1} va fi calculat ca fiind $x_{i+1} = r_i x_i = \left(1+\delta^{2^i}\right)x_i = \left(1+(1-b)^{2^i}\right)x_i$ sau

$$x_{i+1} = a\left[1+(1-b)\right]\left[1+(1-b)^2\right]\left[1+(1-b)^4\right]\cdots\left[1+(1-b)^{2^i}\right]. \qquad (4.4)$$

În acest algoritm se pare că sunt două probleme. Prima, convergența este lentă atunci când b nu este aproape 1 (adică, δ nu este aproape 0); a doua, formula nu este cu autocorecție (din moment ce câtul este calculat ca un produs de termeni independenți), o eroare în unul dintre ei nu va fi corectată. Pentru a trata convergența lentă, dacă se doreşte calcularea $a\,b$, se caută un invers aproximativ al lui b (numit b') şi se rulează algoritmul pentru $ab'\ bb'$. Acest lucru va converge rapid din moment ce $bb' \approx 1$.

Pentru a trata problema autocorecției, calculul ar trebui să fie executat cu câțiva biți de precizie suplimentară pentru a compensa erorile de rotunjire. Totuşi, şi algoritmul lui Goldschmidt are o tentă de autocorecție, prin faptul că nu interesează valoarea exactă a lui r_i. Astfel, în primele câteva iterații se poate alege r_i ca fiind o trunchiere a lui $1+\delta^{2^i}$ care poate face aceste iterații să meargă mai repede, fără să

afecteze viteza convergenței. Dacă r_i este trunchiat atunci y_i nu mai este exact $1-\delta^{2^i}$ astfel încât ecuația (4.4) nu mai poate fi folosită, dar se poate organiza în aşa fel calcularea încât să nu depindă de valoarea precisă a lui r_i. Cu aceste modificări algoritmul lui Goldschmidt se prezintă în felul următor (notele din paranteze drepte arată conexiunile cu formulele dinainte):

1. Se scalează a şi b astfel încât $1 \le b < 2$.
2. Se caută o aproximare a lui $1/b$ (numit b') din tabel.
3. Se poziționează $x_0 = ab'$ şi $y_0 = bb'$.
4. Se iterează până când x_i devine suficient de apropiat de a/b:

   $r \approx 2 - y$ [dacă $y_i = 1+\delta_i$ atunci $r \approx 1-\delta_i$]

   $y = y \times r$ [$y_{i+1} = y_i \times r \approx 1-\delta_i^2$]

   $x = x \times r$ [$x_{i+1} = x_i \times r$]

Cele două metode de iterație sunt corelate. Să presupunem că din metoda lui Newton scoatem iterația şi calculăm fiecare termen x_{i+1} direct în termenii lui b, în loc de recursiv în termenii lui x_i. Pentru efectuarea acestui calcul, se descoperă că

$$x_{i+1} = x_0(2 - x_0 b)\left(1+(x_0 b-1)^2\right)\left(1+(x_0 b-1)^4\right)\cdots\left(1+(x_0 b-1)^{2^i}\right). \quad (4.5)$$

Această formulă este într-o formă foarte asemănătoare cu ecuația (4.4) atunci când $a=1$. De fapt dacă iterațiile ar fi fost făcute cu precizie infinită, cele două metode ar fi produs exact aceeaşi secvență de x_i.

4.1.2.2. Pro şi contra algoritmilor iterativi față de algoritmii direcți

Avantajul iterației este că nu necesită hardware special pentru împărțire, dar poate folosi în locul lui, o schemă de înmulțire (care oricum necesită semnale de comandă suplimentare). De asemenea, la fiecare pas se furnizează de două ori mai multe cifre față de pasul anterior (în contrast cu împărțirea obişnuită, care produce un număr fix de cifre la fiecare pas). Există două dezavantaje ale iterației. Primul este că standardul IEEE pretinde la împărțire rotunjirea corectă, iar iterația livrează un rezultat care este doar aproape de răspunsul corect rotunjit. În cazul iterației lui Newton care calculează $1/b$ în loc de a/b direct, există o problemă adițională. Chiar dacă $1/b$ a fost rotunjit corect, nu este nici o garanție că şi a/b va fi. Să luăm ca exemplu 5/7: pentru două cifre de acuratețe 1/7 este 0,14, şi 5x0,14 este 0,70, dar 5/7 este 0,71. Al doilea dezavantaj este acela că iterația nu dă rest. Acesta este mai problematic în cazul în care hardware-ul de împărțire în virgulă flotantă este utilizat şi pentru împărțirea cu numere întregi, ținând cont că o operație de rest este prezentă aproape în orice limbaj de nivel înalt.

De obicei calea pentru a obține rotunjirea corectă a rezultatului prin iterație este aceea de a calcula $1/b$ la puțin mai mult de $2p$ biți, a calcula a/b la puțin mai mult de $2p$ biți, şi după aceea a rotunji la p biți. Oricum, există o metodă mai rapidă care a fost aparent mai întâi implementată la TI8847. În această metodă, a/b este calculat cu aproape 6 biți suplimentari de precizie, dând un cât preliminar q. Prin comparația lui qb cu a (tot cu numai 6 biți suplimentari, este posibil să se decidă rapid dacă q este corect

rotunjit sau dacă este nevoie să se incrementeze sau să se decrementeze cu 1 la poziția cea mai puțin semnificativă

Un factor care trebuie luat în considerare atunci când se decide asupra algoritmilor de împărțire este viteza relativă a împărțirii față de înmulțire. Din moment ce împărțirea este mult mai complexă decât înmulțirea, ea se derulează mult mai lent. Ca o regulă generală, algoritmii de împărțire ar trebui să aibă o viteză aproape de o treime față de cea a înmulțirii. Un argument în favoarea acestei reguli este că există programe reale (ca unele versiuni ale lui Spice) unde raportul de împărțire pe înmulțire este 1/3. Un alt loc unde apare factorul de trei este metoda iterativă standard pentru calcularea rădăcinii pătrate. Această metodă implică o împărțire pe fiecare iterație, dar poate fi înlocuită utilizând trei înmulțiri.

4.1.2.2.1. Restul în virgulă flotantă

Pentru numere întregi non-negative, împărțirea și restul trebuie să satisfacă următoarele relații: $a = (a \text{ DIV } b)b + a \text{ REM } b$, $0 \le a \text{ REM } b < b$.
Un rest în virgulă flotantă $x \text{ REM } y$ poate fi definit în mod similar ca $x = \text{INT}(x/y)y + x \text{ REM } y$. Deci, se pune problema convertirii lui x/y la un întreg. Funcția restului IEEE utilizează regula de rotunjire la număr par (round-to-even rule). Aceasta înseamnă că se ia $n = \text{INT}(x/y)$ astfel încât $|x/y - n| \le 1/2$. Dacă două n-uri diferite satisfac această relație se ia cel cu valoare pară. Atunci REM este definit ca $x - yn$. Spre deosebire de numerele întregi unde $0 \le a \text{ REM } b < b$, pentru numere în virgulă flotantă $|x \text{ REM } y| \le y/2$. Chiar dacă prin aceasta REM este definit în mod precis, nu este o definiție practică deoarece n poate fi imens. În simplă precizie, n poate să fie atât de mare cât $2^{127}/2^{-126} = 2^{253} \approx 10^{76}$.

Există o cale naturală pentru a calcula REM atunci când este utilizat un algoritm de împărțire direct. Se procedează ca și când s-ar calcula x/y. Dacă $x = s_1 2^{e_1}$ și $y = s_2 2^{e_2}$ iar împărțitorul (schema de împărțire) este ca în figura 4.1, atunci se încarcă s_1 în registrul A și s_2 în registrul M. După $e_1 - e_2$ pași de împărțire, registrul A va avea un număr r de forma $x - yn$ satisfăcând relația $0 \le r < y$. Restul IEEE este atunci, fie r, fie $r - y$. Este necesar deci să se păstreze urma ultimului bit al câtului, fiind necesară pentru rezolvarea cazurilor particulare. Din păcate, $e_1 - e_2$ poate însemna mulți pași, și unitățile de virgulă flotantă în mod tipic au o cantitate maximă de timp care li se acordă pentru executarea unei instrucțiuni. Astfel, de obicei nu este posibilă implementarea lui REM direct. Totuși, așa cum este în familia Intel 8087, există posibilitatea calculării restului prin intermediul unei instrucțiuni pas de rest. Astfel, un pas de rest primește ca parametri două numere x și y și efectuează pașii de împărțire până când fie registrul A conține restul, fie după n pași, unde n este un număr mic, ca numărul de pași necesari pentru efectuarea împărțirii în cea mai înaltă precizie suportată. Driver-ul REM apelează instrucțiunea pas de rest de $\lfloor (e_1 - e_2)/n \rfloor$ ori, utilizând ca deîmpărțit inițial (în primul pas) pe x, iar în următori pași, îl înlocuiește cu restul pasului anterior.

Majoritatea circuitelor integrate rapide de virgulă flotantă nu au implementată funcția REM, chiar dacă ea face parte din cerințele standardului IEEE. Din moment ce standardul permite implementărilor să fie o combinație între hardware și software,

operația REM poate fi implementată în întregime în software. Totuşi, dispunerea unei instrucțiuni pas de rest ar face calculul REM mult mai simplu. Oricum, principalul avantaj provenit din furnizarea unei instrucțiuni REM nu este creşterea performanței ci sporirea acurateței.

4.2. Împărțirea a două numere binare cu semn

În literatura de specialitate se abordează operația aritmetică de împărțire având ca operanzi numere binare, ajungând la scheme seriale sau paralele utilizând diferiți algoritmi (cu restaurarea restului sau fără restaurarea restului). Sintetizând [PATT90] şi [PRAD86], rezultă că la o operație de împărțire binară se parcurg în secvență următoarele etape:

i. se normalizează operanzii făcând în aşa fel încât deîmpărțitul să fie mai mic decât împărțitorul, ambii având semn pozitiv
ii. se efectuează operația de împărțire cu operanzi întotdeauna pozitivi
iii. dacă este cazul, se aplică o corecție asupra rezultatului
iv. se calculează semnele rezultatelor în funcție de semnele operanzilor

Din aceste patru etape doar etapa ii (eventual şi iii) sunt cablate hardware, restul tratându-se software, fapt care conduce la diminuarea performanțelor.
Ne-am propus deci să realizăm o schemă care să poată efectua operația de împărțire lucrând cu operanzi cu tot cu semn, făcându-se software numai normalizarea operanzilor.

Fără a pierde din generalitate, operanzii se presupun a avea cinci biți, un bit pentru semn şi patru pentru valoare. Excepție face deîmpărțitul care poate avea câțiva biți în plus în funcție de numărul nivelurilor pe care le are schema. În ceea ce urmează se consideră o schemă cu cinci niveluri.

Fie deci numărul $X = (x_8x_7x_6x_5x_4x_3x_2x_1x_0)$ deîmpărțitul, unde x_8 reprezintă semnul, şi $Y = (y_4y_3y_2y_1y_0)$ împărțitorul, unde y_4 reprezintă semnul. Ambele fiind subunitare.

O condiție care se pune de la bun început este că în valori absolute împărțitorul trebuie să fie mai mare decât deîmpărțitul

$$|Y| \rangle |X| \qquad (4.6)$$

Motivul este acela că împărțirea, la fel ca adunarea şi scăderea, poate produce DCR (depăşirea capacității registrului). Numai înmulțirea nu produce DCR pentru că înmulțirea a două numere subunitare dă ca rezultat tot un număr subunitar.

Dacă nu este respectată relația (4.6) câtul este supraunitar şi trebuie să se aranjeze operanzii înainte de efectuarea operației de împărțire în felul următor:

- se deplasează la stânga împărțitorul Y dacă se poate (atâta timp cât $y_4 = y_3$), câte poziții este necesar şi se poate fără să apară DCR (Left Shift Arithmetic)
- dacă nu este de ajuns deplasarea împărțitorului, atunci se deplasează la dreapta deîmpărțitul X de câte ori mai este nevoie până să se îndeplinească condiția (4.6). Deplasarea deîmpărțitului X spre dreapta trebuie să se facă perpetuându-se semnul (Right Shift Arithmetic).

Câtul final este câtul rezultat din operație, înmulțit cu $2^{(nr\ de\ deplasări\ ale\ lui\ Y\ +\ nr\ de\ deplasări\ ale\ lui\ X)}$ iar restul final este restul rezultat din operație, înmulțit cu $2^{(nr\ de\ deplasări\ ale\ lui\ X)}$.

Este ușor de observat că am putea renunța la deplasarea lui Y ajungând la îndeplinirea relației (4.6) numai prin deplasări succesive a lui X fără să se piardă nici un bit dacă numărul de niveluri ale schemei este cel puțin egal cu numărul de biți al operanzilor. În ceea ce privește restul, precizia este mai mare atunci când se ajunge la îndeplinirea condiției (4.6) fără să se deplaseze deîmpărțitul X.

4.2.1. Algoritmul propus

Bazându-ne pe împărțirea fără restaurarea (refacerea) restului (nonrestoring type division) în cod complementar, am ajuns la un algoritm a cărui schemă logică este prezentată în figura 4.3. O implementare celulară a acestuia se prezintă în figura 4.4, utilizând celulele de împărțire [PRAD86] ale căror structură detaliată este arătată în figura 4.5. De fapt fiecare celulă este un sumator controlat, realizând fie adunare fie scădere în funcție de valoarea semnalului q_{i+1}.

$$\begin{aligned} r_{i,j} &= r_{i+1,j-1} \oplus c_{i,j} \oplus q_{i+1} \oplus y_j \\ c_{i,j-1} &= r_{i+1,j-1} \cdot c_{i,j} + c_{i,j} \cdot (q_{i+1} \oplus y_j) + (q_{i+1} \oplus y_j) \cdot r_{i+1,j-1} \end{aligned} \qquad (4.7)$$

Rândurile $r_{i,j}$ pentru j=0,1,...,4, constituie resturile parțiale, unde bitul cel mai din stânga este bitul de semn, și satisface următoarea relație:

$$q_5 = \overline{x_8 \oplus y_4} \quad \text{iar} \quad q_i = \overline{r_{i,4} \oplus y_4} \quad \text{pentru} \quad i = 4,3,\ldots,0 \qquad (4.8)$$

Valoarea lui q_i pentru i = 5,4,...,1 determină dacă se va face adunare sau scădere pentru a obține următorul rest parțial $R_{i-1} = (r_{i-1,4} r_{i-1,3} r_{i-1,2} r_{i-1,1} r_{i-1,0})$. Atunci când q_i=0, se realizează suma între $R_i^* = (r_{i,3} r_{i,2} r_{i,1} r_{i,0} x_{i-1})$ și împărțitorul $Y = (y_4 y_3 y_2 y_1 y_0)$, iar atunci când q_i=1, se scade din $R_i^* = (r_{i,3} r_{i,2} r_{i,1} r_{i,0} x_{i-1})$ împărțitorul Y. Scăderea se realizează însumând $R_i^* = (r_{i,3} r_{i,2} r_{i,1} r_{i,0} x_{i-1})$ cu complementul de unu a lui $Y = (y_4 y_3 y_2 y_1 y_0)$ având transportul $c_{i-1,0} = q_i = 1$.

În figura 4.4 câțiva biți au fost denumiți special și anume pentru i,j = 0,1,...,4, $r_{i+1,-1} = x_i$ $r_{5,j-1} = x_{j+4}$. De asemenea pentru subschema CR (corecția restului) s-au folosit notațiile: $r_{-1,j+1} = r_j$ pentru j = 0,1,...,4.

Dacă schema era proiectată să funcționeze numai cu operanzi pozitivi, în urma efectuării operației câtul $q_4 q_3 q_2 q_1 q_0$ era întotdeauna corect, iar restul $r_{0,4} r_{0,3} r_{0,2} r_{0,1} r_{0,0}$ obținut este corespunzător numai dacă bitul cel mai puțin semnificativ din cât este zero ($q_0 = 0$), în caz contrar, ($q_0 = 1$), ca să se obțină restul corect trebuia să-i se adauge împărțitorul.

La schema pe care o propunem noi având posibilitatea să opereze cu numere cu semn, logica de corecție este mai complexă. Tabelul 4.1 arată toate situațiile posibile împreună cu corecțiile care trebuie aplicate. Rândurile haşurate din tabel sunt cele care corespund unor situații imposibile datorită relației:

$$q_0 = \overline{y_4 \oplus r_{0.4}} \tag{4.9}$$

Nr.	x_8	y_4	q_4	$r_{0.4}$	q_0	Q' = câtul final	R = restul final
0	0	0	0	0	0	---	---
1	0	0	0	0	1	√	√
2	0	0	0	1	0	√	$R \leftarrow R_0 + Y$
3	0	0	0	1	1	---	---
4	1	0	1	0	0	---	---
5	1	0	1	0	1	$Q' \leftarrow Q + 1$	$R \leftarrow R_0 - Y$
6	1	0	1	1	0	$Q' \leftarrow Q + 1$	√
7	1	0	1	1	1	---	---
8	1	1	0	0	0	√	$R \leftarrow R_0 + Y$
9	1	1	0	0	1	---	---
10	1	1	0	1	0	---	---
11	1	1	0	1	1	√	√
12	0	1	1	0	0	$Q' \leftarrow Q + 1$	√
13	0	1	1	0	1	---	---
14	0	1	1	1	0	---	---
15	0	1	1	1	1	$Q' \leftarrow Q + 1$	$R \leftarrow R_0 - Y$

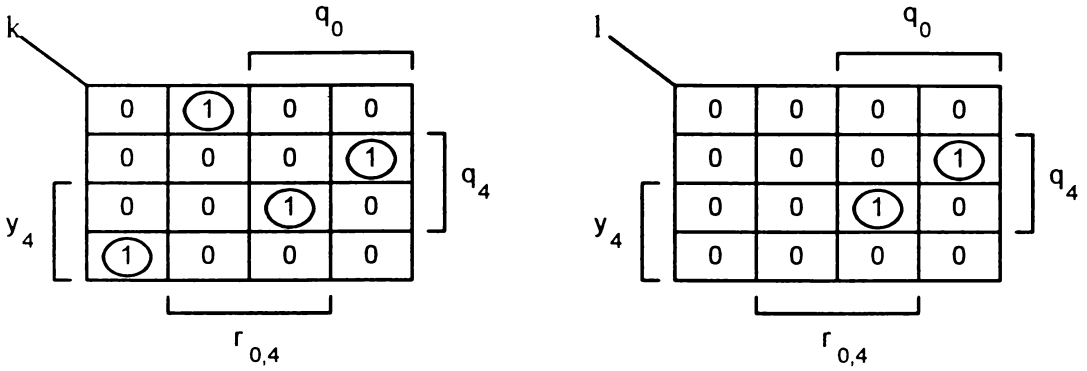
Tabelul 4.1 Situațiile posibile în cazul împărțirii şi corecțiile corespunzătoare.

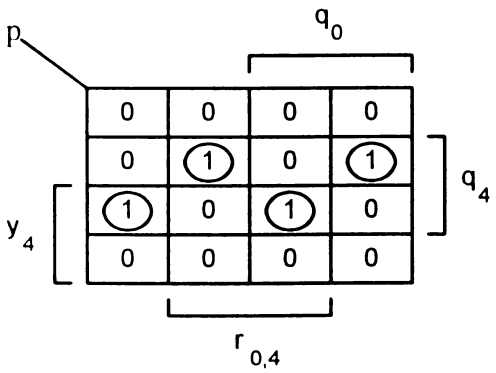
Unde $Q' = q'_4 q'_3 q'_2 q'_1 q'_0$, $R_0 = r_{0.4} r_{0.3} r_{0.2} r_{0.1} r_{0.0}$ şi $R = r_4 r_3 r_2 r_1 r_0$. Pe coloana a doua, valorile lui x_8 au fost puse în conformitate cu $x_8 = y_4 \oplus q_4$ ca să se păstreze regula semnelor. Astfel, este uşor de observat că la liniile 2, 5, 8 şi 15 semnul restului ($r_{0.4}$) nu este la fel cu semnul deîmpărțitului (x_8), lucru incorect ceea ce necesită corecție. De asemenea câtul trebuie corectat atunci când bitul lui cel mai puțin semnificativ este zero ($q_0 = 0$), iar semnul lui este negativ ($q_4 = 1$), situații care apar în liniile 6 şi 12, precum şi atunci când se corectează restul, iar deîmpărțitul şi împărțitorul posedă semne contrare, liniile 5 şi 15.

Se observă din tabelul 4.1 că există trei feluri de corecție, incrementarea lui Q, adăugarea lui Y peste R_0 şi scăderea lui Y din R_0. Vom folosi trei semnale de comandă k, l, p pentru a controla subschemele de corecție CC şi CR din figura 4.4. Tabelul 4.2 prezintă modul de codificare care a fost ales. Subschema CC, este de fapt un sumator degenerat într-un circuit care atunci când câtul $q_4 q_3 q_2 q_1 q_0$ este negativ (adică $q_4 = 1$), adaugă câtului valoarea 0.0001, iar atunci când câtul este pozitiv ($q_4 = 0$) îl lasă nemodificat (adică adaugă valoarea 0.0000).

k	l	p	Operație
1			Y'←Y
0			Y'←0
	1		se scade din rest 'Y'
	0		se însumează în rest Y'
		1	se incrementează câtul
		0	câtul rămâne neschimbat

Tabelul 4.2 Codificarea semnalelor de comandă utilizate în schema de împărțire propusă.





Diagramele Karnaugh pentru funcțiile k, l, p din tabelul 4.1. Nu se poate face nici o minimizare.

Ecuațiile care se obțin cu ajutorul tabelului 4.1 referitor la cele trei semnale de comandă sunt cele care urmează:

$$k = \bar{y}_4 \cdot \bar{q}_4 \cdot r_{0,4} \cdot \bar{q}_0 + \bar{y}_4 \cdot q_4 \cdot \bar{r}_{0,4} \cdot q_0 + y_4 \cdot \bar{q}_4 \cdot \bar{r}_{0,4} \cdot \bar{q}_0 + y_4 \cdot q_4 \cdot r_{0,4} \cdot q_0 \Rightarrow$$

$$k = \bar{q}_4 \cdot \bar{q}_0 \cdot \left(\bar{y}_4 \cdot r_{0,4} + y_4 \cdot \bar{r}_{0,4}\right) + q_4 \cdot q_0 \cdot \left(\bar{y}_4 \cdot \bar{r}_{0,4} + y_4 \cdot r_{0,4}\right) \Rightarrow$$

$$\left.\begin{array}{l} k = \bar{q}_4 \cdot \bar{q}_0 \cdot \left(y_4 \oplus r_{0,4}\right) + q_4 \cdot q_0 \cdot \overline{\left(y_4 \oplus r_{0,4}\right)} \\ \text{și cu relația:} \qquad (4.9) \end{array}\right\} \Rightarrow k = \bar{q}_4 \cdot \bar{q}_0 + q_4 \cdot q_0 = \overline{q_4 \oplus q_0} = q_4 \oplus \bar{q}_0$$

$$l = \bar{y}_4 \cdot q_4 \cdot \bar{r}_{0,4} \cdot q_0 + y_4 \cdot q_4 \cdot r_{0,4} \cdot q_0 \Rightarrow$$

$$l = q_4 q_0 \cdot \left(\bar{y}_4 \cdot \bar{r}_{0,4} + y_4 \cdot r_{0,4}\right) \Rightarrow$$

$$\left.\begin{array}{l} l = q_4 \cdot q_0 \cdot \left(\overline{y_4 \oplus r_{0,4}}\right) \\ \text{şi cu relaţia: (4.9)} \end{array}\right\} \Rightarrow l = q_4 \cdot q_0$$

$$p = \bar{y}_4 \cdot q_4 \cdot \bar{r}_{0,4} \cdot q_0 + \bar{y}_4 \cdot q_4 \cdot r_{0,4} \cdot \bar{q}_0 + y_4 \cdot q_4 \cdot \bar{r}_{0,4} \cdot \bar{q}_0 + y_4 \cdot q_4 \cdot r_{0,4} \cdot q_0 \Rightarrow$$

$$p = \bar{y}_4 \cdot q_4 \cdot \left(\bar{r}_{0,4} \cdot q_0 + r_{0,4} \cdot \bar{q}_0\right) + y_4 \cdot q_4 \cdot \left(\bar{r}_{0,4} \cdot \bar{q}_0 + r_{0,4} \cdot q_0\right) \Rightarrow$$

$$p = \bar{y}_4 \cdot q_4 \cdot \left(r_{0,4} \oplus q_0\right) + y_4 \cdot q_4 \cdot \left(\overline{r_{0,4} \oplus q_0}\right) \Rightarrow$$

$$p = q_4 \cdot \left(\bar{y}_4 \cdot \left(r_{0,4} \oplus q_0\right) + y_4 \cdot \left(\overline{r_{0,4} \oplus q_0}\right)\right) \Rightarrow \left.\begin{array}{l} p = q_4 \cdot \left(y_4 \oplus r_{0,4} \oplus q_0\right) \\ \text{şi cu relaţia: (4.9)} \end{array}\right\} \Rightarrow p = q_4$$

$$\begin{aligned} k &= \overline{q_4 \oplus q_0} = q_4 \oplus \overline{q_0} \\ l &= q_4 q_0 \\ p &= q_4 \end{aligned} \qquad (4.10)$$

După cum se observă din relaţiile (4.10), cele trei semnale k, l, şi p, folosite pentru comandarea operaţiilor de corecţie în cadrul algoritmului de împărţire propus, se obţin doar cu ajutorul a două porţi logice, cele notate cu 1 şi 2 în figura 4.4.

4.2.1.1. Descrierea formală a algoritmului de împărţire propus

În continuare se prezintă o descriere formală a acestui algoritm. Această descriere a fost experimentată cu succes sub forma unui program numit DIVISION.EXE scris în limbajul C++. În subcapitolul 5.2 se vor prezenta rezultatele obţinute din acest program alături de cele obţinute prin simulare.

```
                declare register A(4:0), M(4:0), Q(4:0), COUNT(2:0)
                declare bus INBUS(4:0), OUTBUS(4:0)
BEGIN:          COUNT←0;
                A← INBUS;
                Q← INBUS;
                M← INBUS;
                if M(4)=0 then go to IMPART_POZ;
IMPART_NEG:     if A(4)=0 then
                  begin
NEG_POZ:                A(4:0)← A(4:0)+M(4:0);
                        if A(4)=1 then go to NEG_NEG2;
NEG_POZ2:               Q(0)← 0;
                        if COUNT=4 then go to CORRECTION;
                        A.Q(4:1)← A(3:0).Q;
```

```
                    COUNT← COUNT+1;
                    go to NEG_POZ;
                end
              else
                begin
NEG_NEG:            A(4:0)← A(4:0)-M(4:0);
                    if A(4)=0 then go to NEG_POZ2;
NEG_NEG2:           Q(0)← 1;
                    if COUNT=4 then go to TEST_CORRECTION;
                    A.Q(4:1)← A(3:0).Q;
                    COUNT← COUNT+1;
                    go to NEG_NEG;
                end
IMPART_POZ:     if A(4)=0 then
                begin
POZ_POZ:            A(4:0)← A(4:0)-M(4:0);
                    if A(4)=1 then go to POZ_NEG2;
POZ_POZ2:           Q(0)← 1;
                    if COUNT=4 then go to TEST_CORRECTION;
                    A.Q(4:1)← A(3:0).Q;
                    COUNT← COUNT+1;
                    go to POZ_POZ;
                end
              else
                begin
POZ_NEG:            A(4:0)← A(4:0)+M(4:0);
                    if A(4)=0 then go to POZ_POZ2;
POZ_NEG2:           Q(0)← 0;
                    if COUNT=4 then go to  CORRECTION;
                    A.Q(4:1)← A(3:0).Q;
                    COUNT← COUNT+1;
                    go to POZ_NEG;
                end
TEST_CORRECTION:    if Q(4)=1 then
                begin
                    A(4:0)← A(4:0)-M(4:0);
                    Q(4:0)← Q(4:0)+0.0001;
                end
              go to OUTPUT;
CORRECTION:         if Q(4)=0 then A(4:0)← A(4:0)+M(4:0);
              else Q(4:0)← Q(4:0)+0.0001;
OUTPUT:       OUTBUS ←Q;
              OUTBUS ← A;
END
```

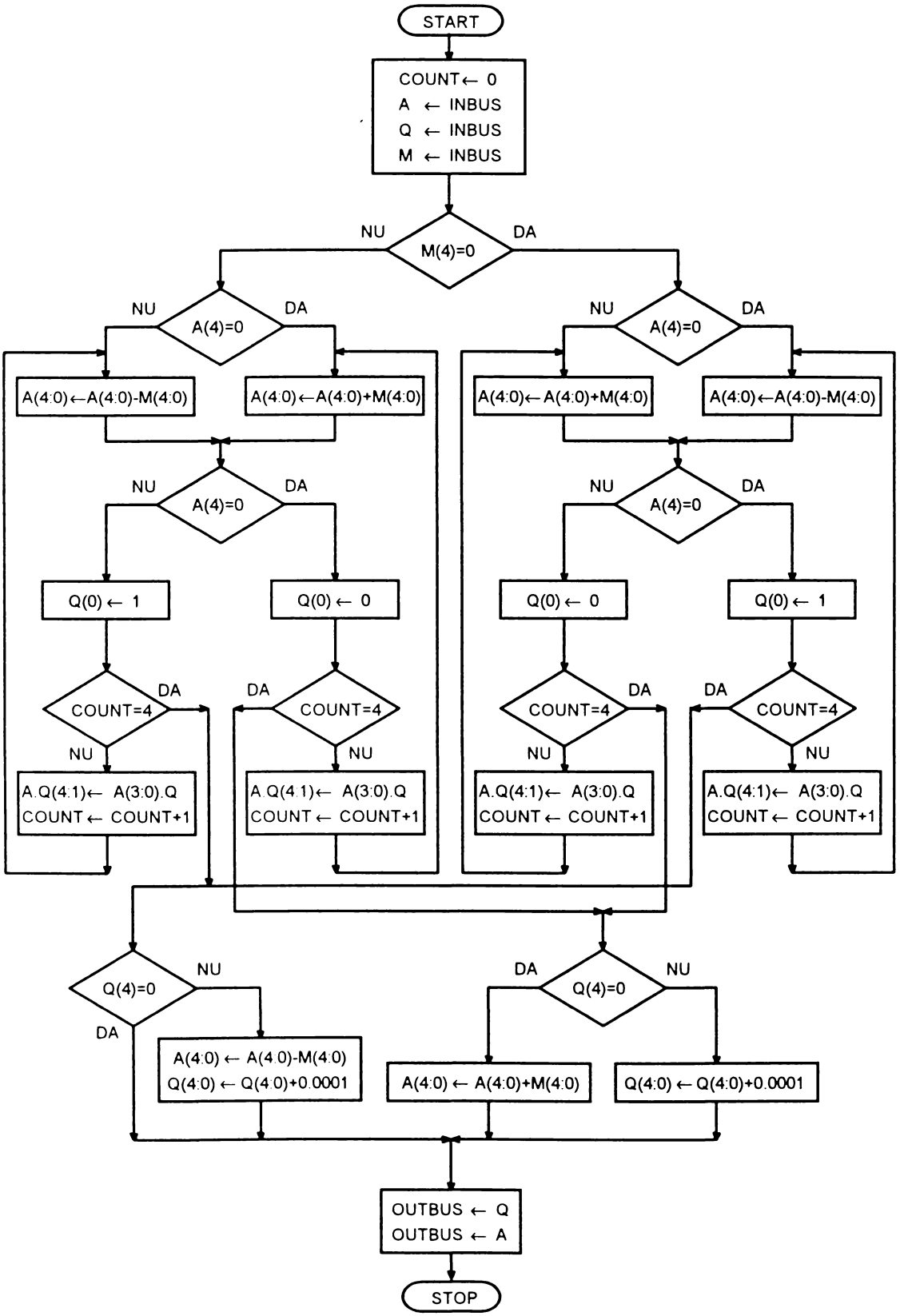
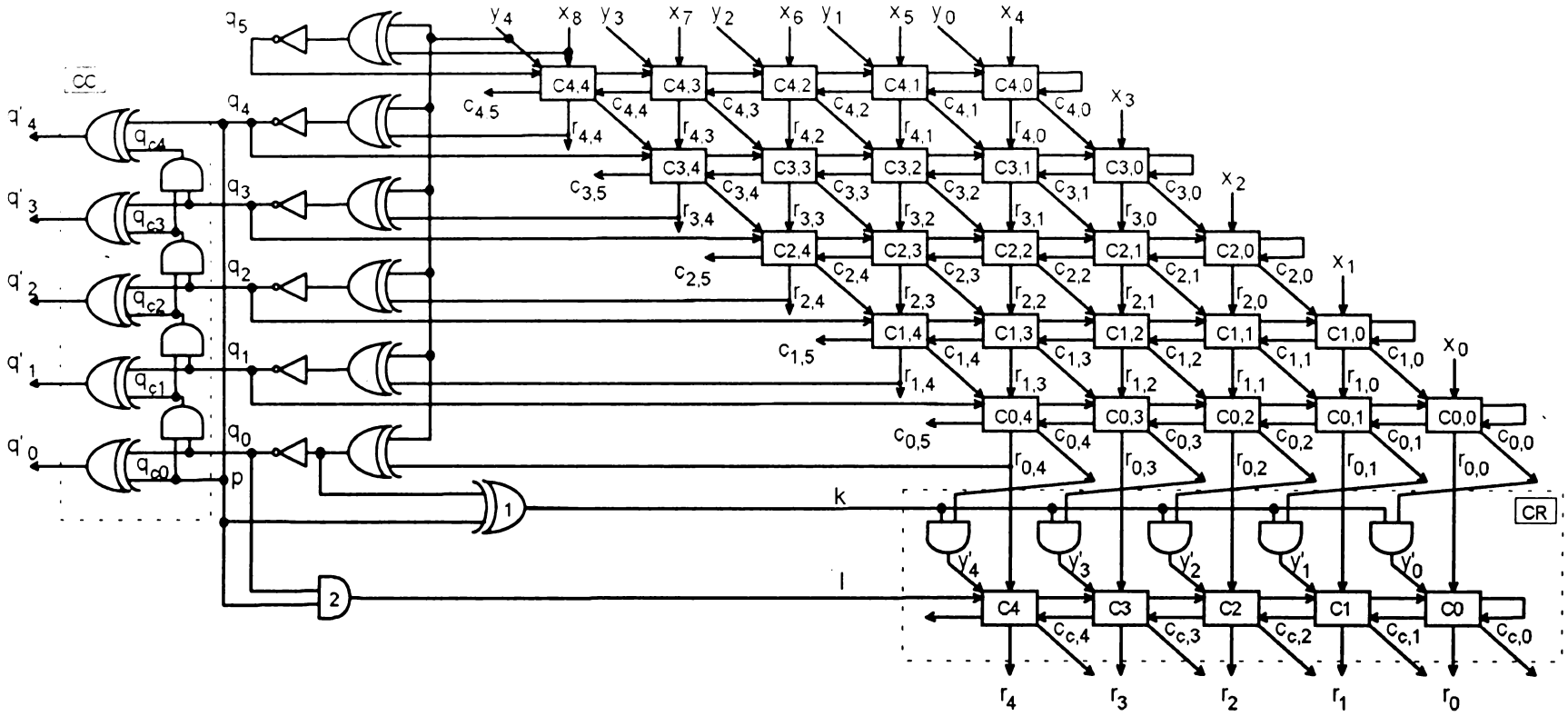


Figura 4.3 Ordinograma împărțirii numerelor binare cu semn. Inițial în registrele A și Q se află deîmpărțitul, iar în registrul M se află împărțitorul. După efectuarea operației registrul A conține restul împărțirii, registrul Q conține câtul, iar registrul M rămâne neafectat



CC - Corecția Câtului

CR - Corecția Restului

Figura 4.4 (Divizor) Împărțitor celular.

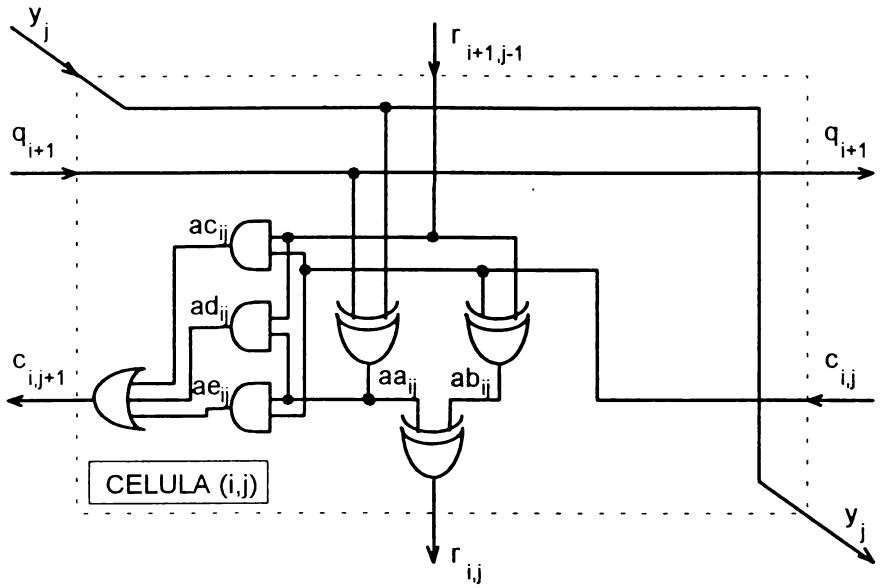


Figura 4.5 Celula de împărțire.

4.2.1.2. Evaluarea timpilor de propagare în cadrul împărțitorului

În ceea ce urmează se va realiza estimarea timpului necesar pentru apariția rezultatelor după momentul alimentării schemei de împărțire cu operanzi valizi şi stabili în cazul cel mai nefavorabil. Din acest timp se poate deduce debitul maxim de rezultate pe care schema le poate furniza. Aceasta este o trăsătură importantă a schemelor paralele de împărțire deoarece, aşa cum s-a menționat anterior, ele sunt destinate sistemelor de mare performanță.

Pentru evaluarea timpului de propagare în cazul general, considerăm o schemă ca cea din figura 4.4 având *m* niveluri şi lucrând cu operanzi pe *n* biți. Se fac următoarele notații:

t_{XOR} - timpul de propagare al unei porți SAU-EXCLUSIV;

t_{INV} - timpul de propagare al unei porți INVERTOARE;

t_{OR} - timpul de propagare al unei porți SAU;

t_{AND} - timpul de propagare al unei porți ŞI.

Prin defalcarea schemei şi pe baza notațiilor de mai sus se determină următorii timpi auxiliari în vederea evaluării timpului total de propagare:

- timpul pentru stabilirea tipului de operație (adunare sau scădere) care se va executa la următorul nivel al schemei: $t_o = t_{XOR} + t_{INV}$;
- timpul pentru stabilirea tipului de corecție a restului: $t'_o = 2 \cdot t_{XOR} + t_{AND}$;
- timpul de propagare al transportului: $t_c = t_{AND} + t_{OR}$;
- timpul calculării unui rest parțial: $t_r = 2 \cdot t_{XOR}$;
- timpul necesar pentru corecția câtului: $t_{cc} = (m-1) \cdot t_{AND} + t_{XOR}$;
- timpul necesar pentru corecția restului: $t_{cr} = t'_o + (n-1) \cdot t_c + t_r$.

Astfel, timpul de propagare pentru un nivel este: $t_o+(n-1)\cdot t_c+t_r$, iar pentru toate cele *m* niveluri este: $t'_P=m\cdot(t_o+(n-1)\cdot t_c+t_r)$.

La acest timp trebuie adăugat fie timpul pentru corecția câtului (t_{cc}), fie timpul pentru corecția restului (t_{cr}), în funcție de care timp este mai mare pentru a se acoperi cazul cel mai nefavorabil. Deci, timpul total de propagare a schemei este:

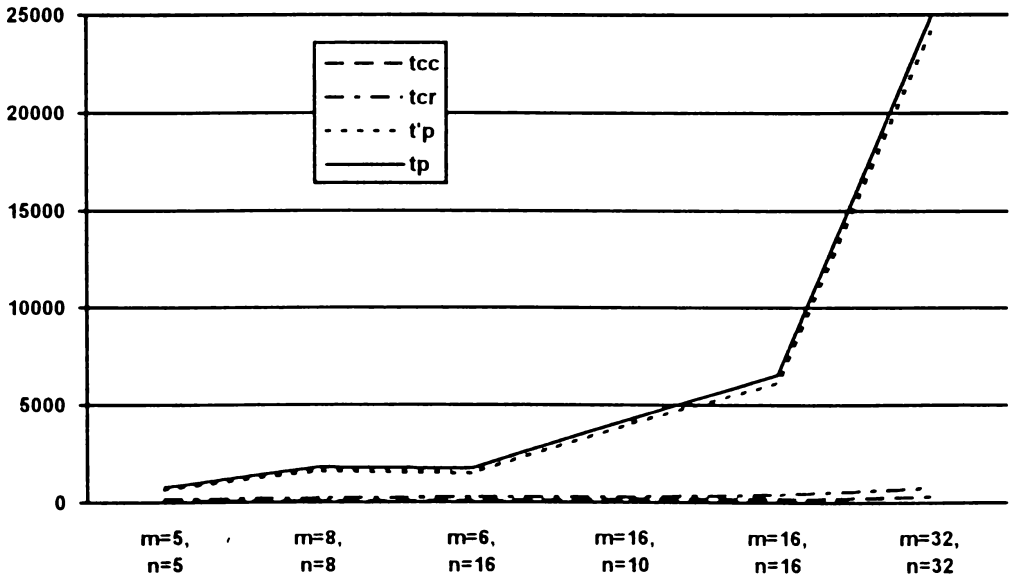
$$t_P=m\cdot(t_o+(n-1)\cdot t_c+t_r)+\max(t_{cc},t_{cr}) \qquad (4.11)$$

De obicei $t_{cr}>t_{cc}$ deoarece *n* se ia egal cu *m*.

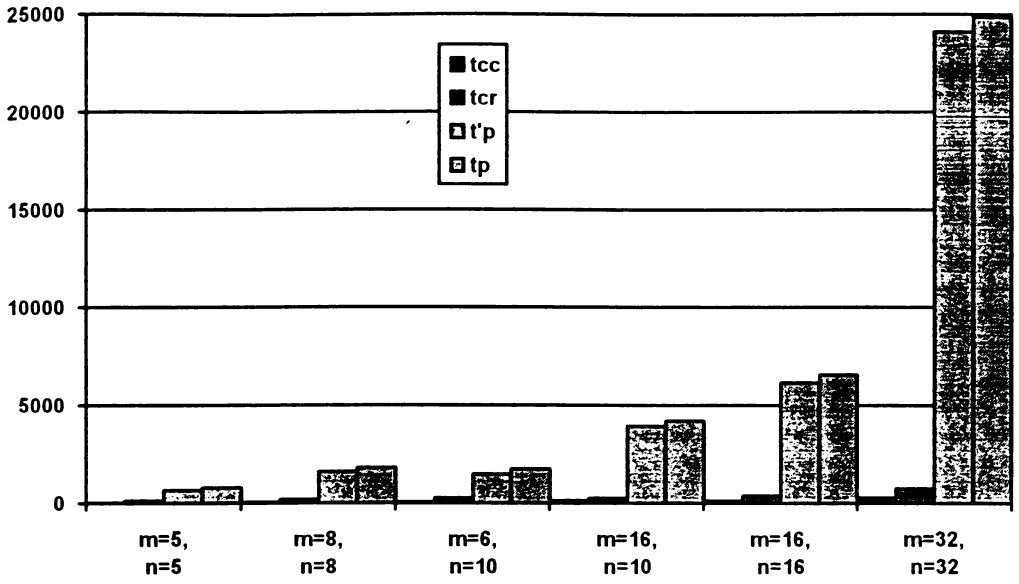
Tabelul 4.3 prezintă timpii de propagare pentru diferite mărimi ale împărțitorului binar propus, calculați cu ajutorul formulei (4.11), considerând următoarele valori pentru timpii de propagare ai porților logice: t_{XOR}=10ns, t_{INV}=10ns, t_{OR}=14ns, t_{AND}=9ns. Ținând cont că acești timpi de propagare reprezintă valori tipice pentru porți logice fundamentale de tehnologie LowSchottky, ne putem imagina că implementarea acestui împărțitor într-un singur circuit integrat într-o tehnologie modernă şi rapidă ar conduce la obținerea unor performanțe mult mai spectaculoase.

Timpi de propagare	m=5, n=5	m=8, n=8	m=6, n=10	m=16, n=10	m=16, n=16	m=32, n=32
t_{cc}	46	73	55	145	145	289
t_{cr}	141	210	256	256	394	762
t'_P	660	1608	1482	3952	6160	24096
t_P	**801**	**1818**	**1738**	**4208**	**6554**	**24858**

Tabelul 4.3 Timpii de propagare (în nsec) pentru diferite mărimi ale schemei.



(a)



(b)

Figura 4.6 Reprezentarea grafică a timpilor de propagare pentru subschemele care constituie împărțitorul binar propus în raport cu mărimea acestuia.

Se poate observa, privind cu atenție tabelul 4.3, precum și figura 4.6 care este o redare grafică a acestui tabel, că timpul total de propagare se află într-o directă proporționalitate atât cu numărul de niveluri *m* ale împărțitorului celular cât și cu numărul de biți *n* ai operanzilor. De asemenea, se poate observa că cu cât cresc valorile *m* și *n* cu atât timpii de propagare ai subschemelor de corecție a restului și câtului devin mai nesemnificativi în calculul timpului total de propagare.

O soluție posibilă pentru a îmbunătăți debitul informațional al schemei pentru valori ale lui *m* și *n* mai mari decât 16 este subdivizarea schemei în câteva partiții egale și introducerea unor stadii de pipeline.

Trebuie menționat faptul că în aceste calcule nu au fost incluse întârzierile suplimentare cauzate de eventualele circuite pentru detecția sau corecția erorilor.

4.2.2. Controlul de paritate al operației de împărțire

Ne propunem acum să grefăm controlul de paritate pe structura de împărțire din figura 4.4 pentru detecția unor erori detectabile cu ajutorul acestei tehnici de control. Trebuie menționat faptul că în [PRAD86] se deduc ecuațiile de paritate pentru un caz particular al unei structurii de împărțire binară. Însă, această schemă nu are capacitatea de a opera cu numere cu semn cum este cazul schemei noastre din figura 4.4, și nici nu conține subschemele de corecție a restului și a câtului care sunt operații inerente ale algoritmului de împărțire (vezi figura 4.3). De asemenea, în cele prezentate în [PRAD86], nu există o formă generală (nici nu poate fi dedusă într-o astfel de manieră), fiind necesară deducerea laborioasă a relațiilor de control pentru fiecare caz particular.

În ceea ce urmează, se va face deducerea într-un mod original a relațiilor de control. Pentru a se putea urmări mai ușor, se deosebesc câteva cazuri în funcție de lungimea *n* a împărțitorului Y și numărul *m* de niveluri ale structurii de împărțire, iar ulterior se ajunge la o formulă generală.

Un prim caz este atunci când n=5 (operandul Y (împărțitorul) are un număr impar de biți) și m=5 (numărul de niveluri ale structurii este impar).

Începem prin a scrie ecuațiile de paritate pentru fiecare nivel de sumator/scăzător (numit pe scurt S/S), iar pe urmă se face însumarea modulo 2 a acestor ecuații de paritate din toate nivelurile. Ținând cont că operandul Y este format dintr-un număr impar de biți, trebuie omis din calcul bitul cel mai puțin semnificativ al transportului, deoarece, atunci când se efectuează operația de adunare la un S/S avem $P_Y \oplus c_{i,0} = P_Y$ pentru că $c_{i,0} = 0$, iar când se efectuează operația de scădere avem $\overline{P}_Y \oplus c_{i,0} = P_Y$ pentru că $c_{i,0} = 1$.

$$x_8 \oplus x_7 \oplus x_6 \oplus x_5 \oplus x_4 \oplus P_Y \oplus \sum_{j=0}^{4} r_{4,j} \oplus \sum_{j=1}^{4} c_{4,j} = 0$$

$$\sum_{j=0}^{3} r_{4,j} \oplus x_3 \oplus P_Y \oplus \sum_{j=0}^{4} r_{3,j} \oplus \sum_{j=1}^{4} c_{3,j} = 0$$

$$\sum_{j=0}^{3} r_{3,j} \oplus x_2 \oplus P_Y \oplus \sum_{j=0}^{4} r_{2,j} \oplus \sum_{j=1}^{4} c_{2,j} = 0$$

$$\sum_{j=0}^{3} r_{2,j} \oplus x_1 \oplus P_Y \oplus \sum_{j=0}^{4} r_{1,j} \oplus \sum_{j=1}^{4} c_{1,j} = 0$$

$$\sum_{j=0}^{3} r_{1,j} \oplus x_0 \oplus P_Y \oplus \sum_{j=0}^{4} r_{0,j} \oplus \sum_{j=1}^{4} c_{0,j} = 0$$

$\oplus$

$$P_X \oplus P_Y \oplus \sum_{i=1}^{4} r_{i,4} \oplus \sum_{j=0}^{4} r_{0,j} \oplus \sum_{i=0}^{4} \sum_{j=1}^{4} c_{i,j} = 0$$

Din ecuația parității de mai sus care se referă la împărțitorul celular din figura 4.4 exceptând subschemele de corecție a câtului și a restului, se poate ajunge la relația (4.12) care reprezintă o formulă mai generală pentru cazul în care n și m sunt numere impare.

$$\boxed{P_X \oplus P_Y \oplus \sum_{i=1}^{m-1} r_{i,n-1} \oplus \sum_{j=0}^{n-1} r_{0,j} \oplus \sum_{i=0}^{m-1} \sum_{j=1}^{n-1} c_{i,j} = 0} \qquad (4.12)$$

Este ușor de observat că dacă numărul de niveluri ar fi fost par, termenul P_Y care apare o singură dată la fiecare nivel, ar fi fost eliminat pentru că însumarea se face modulo 2. Eliminarea termenului P_Y, care reprezintă bitul de paritate al împărțitorului Y, din ecuația de paritate a structurii împărțitorului celular ar indica insuficiența controlului de paritate privind capacitatea de detecție a erorilor. În astfel de cazuri trebuie apelate alte mijloace pentru detecția erorilor în cazul operandului Y

În ceea ce priveşte subschema de corecţie a restului (CR) din figura 4.4, aceasta poate fi văzută prin prisma controlului de paritate ca un nivel de sumator/scăzător având următoarea ecuaţie de paritate pentru n=5 (număr impar):

$$\sum_{j=0}^{4} r_{0,j} \oplus k \cdot P_Y \oplus \sum_{j=0}^{4} r_j \oplus \sum_{j=1}^{4} c_{c,j} = 0 \qquad (4.13)$$

Un alt caz este acela în care n=4 (număr par) şi m=5. Spre deosebire de cazul anterior când n era număr impar, într-o astfel de situaţie, în calculul parităţii fiecărui nivel de sumator/scăzător trebuie să se includă cel mai puţin semnificativ bit al transportului. Argumentul care stă la baza acestei afirmaţii este acela că atunci când se efectuează o operaţie de adunare la un S/S avem $P_Y \oplus c_{i,0} = P_Y$ şi $c_{i,0} = 0$, iar atunci când se efectuează o operaţie de scădere avem $P_Y \oplus c_{i,0} = \overline{P}_Y$ şi $c_{i,0} = 1$.

$$x_7 \oplus x_6 \oplus x_5 \oplus x_4 \oplus P_Y \oplus \sum_{j=0}^{3} r_{4,j} \oplus \sum_{j=0}^{3} c_{4,j} = 0$$

$$\sum_{j=0}^{2} r_{4,j} \oplus x_3 \oplus P_Y \oplus \sum_{j=0}^{3} r_{3,j} \oplus \sum_{j=0}^{3} c_{3,j} = 0$$

$$\sum_{j=0}^{2} r_{3,j} \oplus x_2 \oplus P_Y \oplus \sum_{j=0}^{3} r_{2,j} \oplus \sum_{j=0}^{3} c_{2,j} = 0$$

$$\sum_{j=0}^{2} r_{2,j} \oplus x_1 \oplus P_Y \oplus \sum_{j=0}^{3} r_{1,j} \oplus \sum_{j=0}^{3} c_{1,j} = 0$$

$$\sum_{j=0}^{2} r_{1,j} \oplus x_0 \oplus P_Y \oplus \sum_{j=0}^{3} r_{0,j} \oplus \sum_{j=0}^{3} c_{0,j} = 0$$

$\oplus$

$$P_X \oplus P_Y \oplus \sum_{i=1}^{4} r_{i,3} \oplus \sum_{j=0}^{3} r_{0,j} \oplus \sum_{i=0}^{4} \sum_{j=0}^{3} c_{i,j} = 0$$

Generalizând ecuaţia parităţii de mai sus se poate ajunge la relaţia (4.14) pentru cazul în care n este număr par iar m este număr impar.

$$\boxed{P_X \oplus P_Y \oplus \sum_{i=1}^{m-1} r_{i,n-1} \oplus \sum_{j=0}^{n-1} r_{0,j} \oplus \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} c_{i,j} = 0} \qquad (4.14)$$

În mod asemănător şi în acest caz se observă că dacă numărul m de niveluri ar fi fost par, termenul P_Y care apare o singură dată la fiecare nivel, ar fi fost eliminat.

Revenind la subschema de corecție a restului (CR), ecuația de paritate pentru n=4 (număr par) are următoarea formă:

$$\sum_{j=0}^{3} r_{0,j} \oplus k \cdot P_Y \oplus \sum_{j=0}^{3} r_j \oplus \sum_{j=0}^{3} c_{c,j} = 0 \qquad (4.15)$$

Cu ajutorul relațiilor (4.12) şi (4.14) se obține relația (4.16) care reprezintă formula generală a controlului de paritate al unui împărțitor celular neincluzând însă subschemele de corecție a restului şi a câtului. Factorul (m mod 2) condiționează includerea bitului de paritate al operandului Y în calculul parității în funcție de numărul m al nivelurilor structurii de împărțire. De asemenea, la ultima sumă modulo 2 din ecuația (4.16) variabila j ia valori din intervalul [(n mod 2), n-1], adică [0, n-1] pentru valori pare ale lui n iar [1, n-1] pentru valori impare, condiționând astfel includerea în calculul parității a bitului cel mai puțin semnificativ al transportului în funcție de numărul n de biți al operanzilor.

$$P_X \oplus (m \bmod 2)P_Y \oplus \sum_{i=1}^{m-1} r_{i,n-1} \oplus \sum_{j=0}^{n-1} r_{0,j} \oplus \sum_{i=0}^{m-1} \sum_{j=(n \bmod 2)}^{n-1} c_{i,j} = 0 \qquad (4.16)$$

Din relațiile (4.13) şi (4.15), prin generalizare, se obține relația (4.17) care reprezintă ecuația de paritate a subschemei pentru corecția restului din cadrul împărțitorului celular propus. În relația (4.17), cum era şi firesc, nu intervine decât numărul n de biți al împărțitorului Y.

$$k \cdot P_Y \oplus \sum_{j=0}^{n-1} r_{0,j} \oplus \sum_{j=0}^{n-1} r_j \oplus \sum_{j=(n \bmod 2)}^{n-1} c_{c,j} = 0 \qquad (4.17)$$

Singura parte a împărțitorului celular propus pentru care nu s-a elaborat până acum ecuația controlului de paritate este subschema de corecție a câtului (CC). Această subschemă, aşa cum s-a mai menționat anterior, atunci când semnalul de comandă p este pe "1" logic efectuează incrementarea câtului Q, iar în caz contrar lasă câtul nemodificat. Astfel, subschema CC poate fi tratată ca un sumator cu observația că unul dintre operanzi este câtul Q iar celălalt este fie operandul nul fie unitatea. Deci, ecuația pentru controlul de paritate are următoarea formă:

$$\sum_{i=0}^{m-1} q_i \oplus \sum_{i=0}^{m-1} q'_i \oplus \sum_{i=0}^{m-1} q_{ci} = 0 \qquad \text{unde } q_{m-1} = q_{c0} \text{ şi se pot elimina.}$$

Din păcate, dacă în cadrul subschemei CC operanzii (în mod special câtul Q) ajung corupți, controlul de paritate nu are posibilitatea de a semnala acest lucru deoarece paritatea operandului (câtului) se calculează local pe baza informației primite. Astfel, se pot semnala doar erori legate de incrementarea câtului. Totuşi, din fericire, acest neajuns se poate depăşi calculând paritatea câtului cu ajutorul altor informații diferite de câtul în sine.

Astfel, o soluție mai bună poate fi obținută dacă ne bazăm pe faptul că cel mai semnificativ bit de transport (carry) al unui nivel de sumator/scăzător este tot timpul

invers față de bitul cel mai semnificativ al restului parțial din același nivel, adică $c_{i,n} = \bar{r}_{i,n-1}$ pentru $\forall\, i \ni [0, m-1]$, lucru care se demonstrează prin lema 1.

Deci, știind că $q_i = \overline{y_{n-1} \oplus r_{i,n-1}}$ și că $c_{i,n} = \bar{r}_{i,n-1}$ pentru $\forall\, i \ni [0, m-1]$ rezultă că $q_i = y_{n-1} \oplus c_{i,n}$ pentru $\forall\, i \ni [0, m-1]$ și astfel formula dinainte pentru subschema CC devine:

$$\sum_{i=0}^{m-1} (y_{n-1} \oplus c_{i,n}) \oplus \sum_{i=0}^{m-1} q'_i \oplus \sum_{i=0}^{m-1} q_{c_i} = 0 \Rightarrow$$

$$\boxed{(m \bmod 2) y_{n-1} \oplus \sum_{i=0}^{m-1} c_{i,n} \oplus \sum_{i=0}^{m-1} q'_i \oplus \sum_{i=0}^{m-1} q_{c_i} = 0} \qquad (4.18)$$

Relația (4.18) reprezintă formula generală a controlului de paritate pentru subschema de corecție a câtului. Se observă că dacă numărul *m* de niveluri al împărțitorului celular este un număr par, cel mai semnificativ bit al operandului Y, și anume y_{n-1}, nu mai intervine în calculul parității subschemei CC.

Soluția prezentată mai sus, și anume calcularea parității câtului Q din alte semnale decât biții proprii, sporește capacitatea de detecție a schemei. Mai precis se pot detecta erori care au fost generate de defecțiuni ale circuitelor (SAU-EXCLUSIV sau INVERTOARE) care realizează funcțiile logice q_i pentru $\forall\, i \ni [0, m-1]$.

Însumând în modulo 2 relațiile (4.16), (4.17) și (4.18) se obține relația (4.19) care reprezintă forma generală a ecuației de paritate pentru întreaga structură a împărțitorului celular propus.

$$\boxed{P_X \oplus k \cdot P_Y \oplus (m \bmod 2)(P_Y \oplus y_{n-1}) \oplus \sum_{i=0}^{m-1} \sum_{j=(n \bmod 2)}^{n} c_{i,j} \oplus \sum_{j=(n \bmod 2)}^{n-1} c_{c,j} \oplus \sum_{i=1}^{m-1} r_{i,n-1} \oplus \sum_{j=0}^{n-1} r_j \oplus \sum_{i=0}^{m-1} q'_i \oplus \sum_{i=0}^{m-1} q_{c_i} = 0} \qquad (4.19)$$

În calitate de exemplu se prezintă, în ceea ce urmează, particularizarea relației (4.19) pentru cazul în care m=5 și n=5, adică cazul din figura 4.4. Astfel, avem:

$$P_X \oplus k \cdot P_Y \oplus P_Y \oplus y_{n-1} \oplus \sum_{i=0}^{4} \sum_{j=1}^{5} c_{i,j} \oplus \sum_{j=1}^{4} c_{c,j} \oplus \sum_{i=1}^{4} r_{i,4} \oplus \sum_{j=0}^{4} r_j \oplus \sum_{i=0}^{4} q'_i \oplus \sum_{i=0}^{4} q_{c_i} = 0$$

de unde rezultă:

$$P_X \oplus \bar{k} \cdot P_Y \oplus y_{n-1} \oplus \sum_{i=0}^{4} \sum_{j=1}^{5} c_{i,j} \oplus \sum_{j=1}^{4} c_{c,j} \oplus \sum_{i=1}^{4} r_{i,4} \oplus \sum_{j=0}^{4} r_j \oplus \sum_{i=0}^{4} q'_i \oplus \sum_{i=0}^{4} q_{c_i} = 0$$

Este foarte important de menționat că relația de mai sus precum și forma ei generalizată (4.19) este valabilă atât în cazul celulei de împărțire din figura 4.5 cât și în cazul altor celule de împărțire mai mult sau mai puțin asemănătoare. Oricum, introducând câteva defecțiuni de tipul "blocare la" (stuck-at) în diverse puncte ale structurii de împărțire constituite din celule de împărțire ca cea din figura 4.5, simularea a confirmat faptul că defecțiunile apărute în circuitele pentru determinarea transportului nu se pot detecta prin paritate. Acest neajuns se înlătură, fie prin duplicarea circuitelor

de transport, fie utilizând alte tipuri de celule care se vor discuta în subcapitolele următoare.

Lema 1: Într-o structură celulară de împărțire ca cea descrisă anterior, cel mai semnificativ bit de transport al unui nivel de celule S/S este tot timpul invers față de bitul cel mai semnificativ al restului parțial din același nivel, adică $c_{i,n} = \bar{r}_{i,n-1}$ pentru $\forall i \ni [0, m-1]$.

Demonstrație: Se ia cazul general al unei scheme de împărțire cu *m* niveluri și lungimea împărțitorului de *n* biți. Deîmpărțitul poate avea până la (*n*+*m*-1) biți $x^*_{n+m-2}x^*_{n+m-3}\cdots x^*_2x^*_1x^*_0$. Figura 4.7 prezintă o parte din structura unui astfel de împărțitor celular.

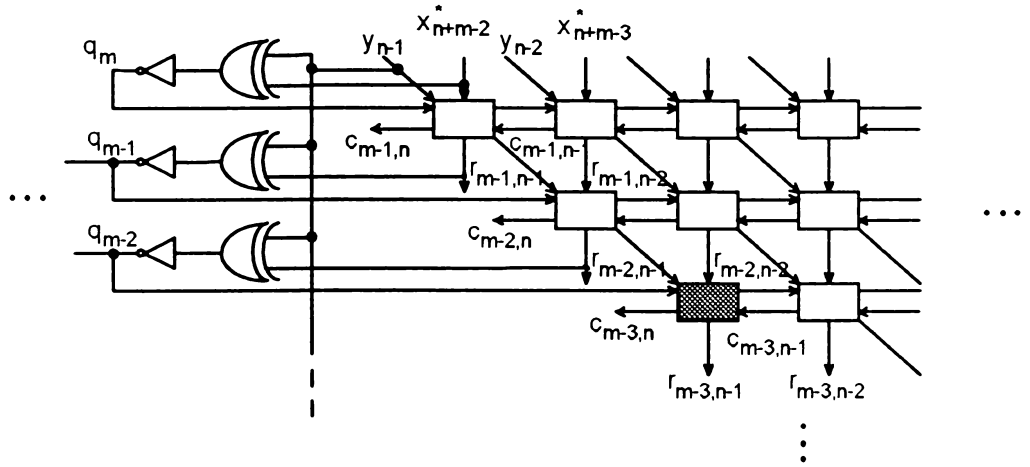


Figura 4.7 Un fragment din structura împărțitorului celular.

Nr.	y_{n-1}	$r_{m-2,n-1}$	$r_{m-2,n-2}$	$c_{m-3,n-1}$	q_{m-2}	$y_{n-1} \oplus q_{m-2}$	$c_{m-3,n}$	$r_{m-3,n-1}$
0	0	0	0	0	1	1	0	1
1	0	0	0	1	1	1	1	0
2	0	0	1	0	1	1	1	0
3	0	0	1	1	1	1	1	1
4	0	1	0	0	0	0	0	0
5	0	1	0	1	0	0	0	1
6	0	1	1	0	0	0	0	1
7	0	1	1	1	0	0	1	0
8	1	0	0	0	0	1	0	1
9	1	0	0	1	0	1	1	0
10	1	0	1	0	0	1	1	0
11	1	0	1	1	0	1	1	1
12	1	1	0	0	1	0	0	0
13	1	1	0	1	1	0	0	1
14	1	1	1	0	1	0	0	1
15	1	1	1	1	1	0	1	0

Tabelul 4.4 Cele 16 cazuri posibile de funcționare a unei celule de împărțire

Tabelul 4.4 prezintă cele 16 cazuri posibile teoretic de funcționare a unei celule de împărțire (de ex. cea înegrită din figura 4.7) care calculează bitul cel mai semnificativ al unui rest parțial. Se observă că singurele cazuri în care nu este respectată relația: $c_{i,n} = \bar{r}_{i,n-1}$ pentru $\forall i \ni [0, m-1]$ sunt cazurile 3, 4, 11 şi 12. Aceste cazuri trebuie să se demonstreze că sunt practic imposibile.

Cazul 4

Pentru simplitate, restul parțial $r_{m-2,n-1}r_{m-2,n-2}\cdots r_{m-2,0}x^*_{m-3}$ s-a notat cu $x_{n-1}x_{n-2}\cdots x_0x_d$, iar restul parțial $r_{m-3,n-1}r_{m-3,n-2}\cdots r_{m-3,0}$ s-a notat cu $s_{n-1}s_{n-2}\cdots s_0$. De asemenea, transportul $c_{m-3,n}c_{m-3,n-1}c_{m-3,n-2}\cdots c_{m-3,0}$ s-a notat $c_nc_{n-1}c_{n-2}\cdots c_0$.

Aşa cum se observă din $q_{m-2}=0$, operația care se efectuează în acest caz pe tot nivelul sumatorului/scăzătorului este operația de adunare. Astfel, avem:

x_{n-1}	x_{n-2}	x_{n-3}	x_{n-4}	x_{n-5}	$\cdots$	x_{j+1}	x_j	x_{j-1}	$\cdots$	x_1	x_0	x_d
+ y_{n-1}	y_{n-2}	y_{n-3}	y_{n-4}	$\cdots$	y_{j+2}	y_{j+1}	y_j	$\cdots$	y_2	y_1	y_0	
s_{n-1}	s_{n-2}	s_{n-3}	s_{n-4}	$\cdots$	s_{j+2}	s_{j+1}	s_j	$\cdots$	s_2	s_1	s_0	
c_n	c_{n-1}	c_{n-2}	c_{n-3}	$c_{n-4}\cdots$	c_{j+3}	c_{j+2}	c_{j+1}	c_j	$\cdots$	c_3	c_2	c_1 c_0

Deoarece $x_{n-1} = 1$, $x_{n-2} = 0$, $y_{n-1} = 0$ şi $y_{n-2} = \bar{x}_{n-2} = 1$ rezultă că:

$s_{n-1} = x_{n-2} \oplus y_{n-1} \oplus c_{n-1} = 0 \oplus 0 \oplus c_{n-1} = c_{n-1}$

$c_n = x_{n-2}y_{n-1} + x_{n-2}c_{n-1} + y_{n-1}c_{n-1} = 0 \cdot 0 + 0c_{n-1} + 0c_{n-1} = 0$

Trebuie demonstrat că c_{n-1} este diferit de 0, adică $c_{n-1}=1$ atunci când condiția $|X| \le |Y|$ este respectată, lucru care este valabil fiindcă valoarea absolută a resturilor parțiale este cel mult egală cu valoarea absolută a împărțitorului.

$c_{n-1} = x_{n-3}y_{n-2} + x_{n-3}c_{n-2} + y_{n-2}c_{n-2} = x_{n-3}1 + x_{n-3}c_{n-2} + 1c_{n-2} = x_{n-3} + x_{n-3}c_{n-2} + c_{n-2} = x_{n-3} + c_{n-2}$

$c_{n-2} = x_{n-4}y_{n-3} + x_{n-4}c_{n-3} + y_{n-3}c_{n-3}$

$c_{n-1} = x_{n-3} + c_{n-2} = x_{n-3} + x_{n-4}y_{n-3} + x_{n-4}c_{n-3} + y_{n-3}c_{n-3} = x_{n-3} + x_{n-4} + (x_{n-4} + y_{n-3})c_{n-3}$

$c_{n-3} = x_{n-5}y_{n-4} + x_{n-5}c_{n-4} + y_{n-4}c_{n-4}$

$c_{n-1} = x_{n-3} + x_{n-4} + (x_{n-4} + y_{n-3})c_{n-3} =$

$= x_{n-3} + x_{n-4} + (x_{n-4} + y_{n-3})(x_{n-5}y_{n-4} + x_{n-5}c_{n-4} + y_{n-4}c_{n-4}) =$

$= x_{n-3} + x_{n-4} + x_{n-4}x_{n-5}c_{n-4} + y_{n-3}x_{n-5}y_{n-4} + y_{n-3}x_{n-5}c_{n-4} + y_{n-3}y_{n-4}c_{n-4} =$

$= x_{n-3} + x_{n-4} + x_{n-5} + (x_{n-4}x_{n-5} + y_{n-3}x_{n-5} + y_{n-3}y_{n-4})c_{n-4}$

$c_{n-1} = x_{n-3} + x_{n-4} + x_{n-5} + x_{n-6} + \cdots + x_{j+2} + x_{j+1} + (\ldots + y_{n-3}y_{n-4}y_{n-5}y_{n-6}\cdots y_{j+2})c_{j+2}$

$$c_{j+2} = x_j y_{j+1} + x_j c_{j+1} + y_{j+1} c_{j+1} = 1 y_{j+1} + 1 c_{j+1} + y_{j+1} c_{j+1} = y_{j+1} + c_{j+1} + y_{j+1} c_{j+1} = y_{j+1} + c_{j+1}$$

$$c_{n-1} = x_{n-3} + x_{n-4} + x_{n-5} + x_{n-6} + \cdots + x_{j+2} + x_{j+1} + (\ldots + y_{n-3} y_{n-4} y_{n-5} y_{n-6} \cdots y_{j+2}) c_{j+2} =$$

$$= x_{n-3} + x_{n-4} + x_{n-5} + x_{n-6} + \cdots + x_{j+2} + x_{j+1} + (\ldots + y_{n-3} y_{n-4} y_{n-5} y_{n-6} \cdots y_{j+2})(y_{j+1} + c_{j+1}) =$$

$$= x_{n-3} + x_{n-4} + x_{n-5} + x_{n-6} + \cdots + x_{j+2} + x_{j+1} + y_{n-3} y_{n-4} y_{n-5} y_{n-6} \cdots y_{j+2} y_{j+1} + y_{n-3} y_{n-4} y_{n-5} y_{n-6} \cdots y_{j+2} c_{j+1} = 1$$

pentru că $x_i = \overline{y}_i$ pentru valori ale lui i din intervalul închis [j+1,n-2], conform observației 3.4 prezentată în anexa A, cazul $|X| \le |Y|$.

Cazul 11)

Convenția de notații făcută anterior rămâne valabilă şi pentru acest caz.

Din tabelul 4.4 se observă că $q_{m-2}=0$ fapt care înseamnă că operația care se efectuează în acest caz pe întreg nivelul celulelor S/S este operația de adunare. Deci:

x_{n-1}	x_{n-2}	x_{n-3}	x_{n-4}	x_{n-5}	$\cdots$	x_{j+1}	x_j	x_{j-1}	$\cdots$	x_1	x_0	x_d
+ y_{n-1}	y_{n-2}	y_{n-3}	y_{n-4}	$\cdots$	y_{j+2}	y_{j+1}	y_j	$\cdots$	y_2	y_1	y_0	
s_{n-1}	s_{n-2}	s_{n-3}	s_{n-4}	$\cdots$	s_{j+2}	s_{j+1}	s_j	$\cdots$	s_2	s_1	s_0	
c_n	c_{n-1}	c_{n-2}	c_{n-3}	c_{n-4}	$\cdots$	c_{j+3}	c_{j+2}	c_{j+1}	c_j	$\cdots$	c_3	c_2	c_1	c_0

Pentru că: $x_{n-1} = 0$, $x_{n-2} = 1$, $y_{n-1} = 1$ şi $y_{n-2} = \overline{x}_{n-2} = 0$ atunci avem că:

$s_{n-1} = x_{n-2} \oplus y_{n-1} \oplus c_{n-1} = 1 \oplus 1 \oplus c_{n-1} = c_{n-1}$

$c_n = x_{n-2} y_{n-1} + x_{n-2} c_{n-1} + y_{n-1} c_{n-1} = 1$

Trebuie demonstrat că c_{n-1} este diferit de 1, adică $c_{n-1}=0$ atunci când condiția $|X| \le |Y|$ este respectată, lucru care este valabil fiindcă valoarea absolută a resturilor parțiale este cel mult egală cu valoarea absolută a împărțitorului.

Observația 3.3 prezentată în anexa A, cazul $|X| \le |Y|$, are trei subpuncte şi de aceea se va demonstra că $c_{n-1}=0$ pentru fiecare subpunct în parte.

11.a)

$c_{n-1} = x_{n-3} y_{n-2} + x_{n-3} c_{n-2} + y_{n-2} c_{n-2} = x_{n-3} c_{n-2}$

$c_{n-2} = x_{n-4} y_{n-3} + x_{n-4} c_{n-3} + y_{n-3} c_{n-3}$

$c_{n-1} = x_{n-3} c_{n-2} = x_{n-3}(x_{n-4} y_{n-3} + x_{n-4} c_{n-3} + y_{n-3} c_{n-3}) = x_{n-3} x_{n-4} c_{n-3}$

$c_{n-3} = x_{n-5} y_{n-4} + x_{n-5} c_{n-4} + y_{n-4} c_{n-4}$

$c_{n-1} = x_{n-3} x_{n-4} c_{n-3} = x_{n-3} x_{n-4}(x_{n-5} y_{n-4} + x_{n-5} c_{n-4} + y_{n-4} c_{n-4}) = x_{n-3} x_{n-4} x_{n-5} c_{n-4}$

$c_{n-1} = x_{n-3} x_{n-4} x_{n-5} x_{n-6} \cdots x_2 x_1 c_2$

$c_2 = x_0 y_1 + x_0 c_1 + y_1 c_1$

$c_{n-1} = x_{n-3}x_{n-4}x_{n-5}x_{n-6}\cdots x_2x_1c_2 = x_{n-3}x_{n-4}x_{n-5}x_{n-6}\cdots x_2x_1(x_0y_1 + x_0c_1 + y_1c_1) =$

$= x_{n-3}x_{n-4}x_{n-5}x_{n-6}\cdots x_2x_1x_0c_1$

$c_1 = x_dy_0 + x_dc_0 + y_0c_0 = x_dy_0 + x_d0 + y_00 = x_dy_0$

$c_{n-1} = x_{n-3}x_{n-4}x_{n-5}x_{n-6}\cdots x_2x_1x_0c_1 = x_{n-3}x_{n-4}x_{n-5}x_{n-6}\cdots x_2x_1x_0x_dy_0 = 0$

11.b)

$c_{n-1} = x_{n-3}y_{n-2} + x_{n-3}c_{n-2} + y_{n-2}c_{n-2} = x_{n-3}c_{n-2}$

$c_{n-2} = x_{n-4}y_{n-3} + x_{n-4}c_{n-3} + y_{n-3}c_{n-3}$

$c_{n-1} = x_{n-3}c_{n-2} = x_{n-3}(x_{n-4}y_{n-3} + x_{n-4}c_{n-3} + y_{n-3}c_{n-3}) = x_{n-3}x_{n-4}c_{n-3}$

$c_{n-3} = x_{n-5}y_{n-4} + x_{n-5}c_{n-4} + y_{n-4}c_{n-4}$

$c_{n-1} = x_{n-3}x_{n-4}c_{n-3} = x_{n-3}x_{n-4}(x_{n-5}y_{n-4} + x_{n-5}c_{n-4} + y_{n-4}c_{n-4}) = x_{n-3}x_{n-4}x_{n-5}c_{n-4}$

$c_{n-1} = x_{n-3}x_{n-4}x_{n-5}x_{n-6}\cdots x_{j+2}x_{j+1}c_{j+2}$

$c_{j+2} = x_jy_{j+1} + x_jc_{j+1} + y_{j+1}c_{j+1} = 1y_{j+1} + 1c_{j+1} + y_{j+1}c_{j+1} = y_{j+1} + c_{j+1} + y_{j+1}c_{j+1} = y_{j+1} + c_{j+1}$

$c_{n-1} = x_{n-3}x_{n-4}x_{n-5}x_{n-6}\cdots x_{j+2}x_{j+1}c_{j+2} = x_{n-3}x_{n-4}x_{n-5}x_{n-6}\cdots x_{j+2}x_{j+1}(y_{j+1} + c_{j+1}) =$

$= x_{n-3}x_{n-4}x_{n-5}x_{n-6}\cdots x_{j+2}x_{j+1}c_{j+1}$

dacă $x_{j-1} \neq x_d$, adică j>0, atunci avem:

$c_{j+1} = x_{j-1}y_j + x_{j-1}c_j + y_jc_j = 0\cdot 1 + 0c_j + 1c_j = c_j$

$c_{n-1} = x_{n-3}x_{n-4}x_{n-5}x_{n-6}\cdots x_{j+2}x_{j+1}c_{j+1} = x_{n-3}x_{n-4}x_{n-5}x_{n-6}\cdots x_{j+2}x_{j+1}c_j$

$c_j = x_{j-2}y_{j-1} + x_{j-2}c_{j-1} + y_{j-1}c_{j-1} = 0\cdot 0 + 0c_{j-1} + 0c_{j-1} = 0$

deci, $c_{n-1} = x_{n-3}x_{n-4}x_{n-5}x_{n-6}\cdots x_{j+2}x_{j+1}c_j = x_{n-3}x_{n-4}x_{n-5}x_{n-6}\cdots x_{j+2}x_{j+1}0 = 0$

chiar dacă $x_{j-2} = x_d = 1$ fiindcă în acest caz $c_{j-1} = c_0 = 0$,

iar dacă $x_{j-1} = x_d$, adică j=0 şi $c_j = c_0 = 0$, avem

$c_{j+1} = x_{j-1}y_j + x_{j-1}c_j + y_jc_j = x_d\cdot 1 + x_dc_0 + 1c_0 = x_d$

şi $c_{n-1} = x_{n-3}x_{n-4}x_{n-5}x_{n-6}\cdots x_{j+2}x_{j+1}c_{j+1} = x_{n-3}x_{n-4}x_{n-5}x_{n-6}\cdots x_2x_1x_d$.

În acest caz împărţitorul trebuie să aibă valoare $(-2^{n-1} - 1)$. Din ultima descriere a lui c_{n-1} se observă că pentru a fi $c_{n-1} = 1$ trebuie ca $x_d = 1$ iar restul parţial înainte de a se deplasa la stânga să aibă valoare $(+2^{n-1} - 1)$, adică să fie de forma $x_{n-1}.x_{n-2}x_{n-3}\cdots x_2x_1x_0 = 0.11\cdots 111$. Astfel de rest parţial nu se poate obţine în condiţiile cerute, deci rezultă $c_{n-1} = 0$.

11.c)

$c_{n-1} = x_{n-3}y_{n-2} + x_{n-3}c_{n-2} + y_{n-2}c_{n-2} = x_{n-3}c_{n-2}$

$c_{n-2} = x_{n-4}y_{n-3} + x_{n-4}c_{n-3} + y_{n-3}c_{n-3}$

$$c_{n-1} = x_{n-3}c_{n-2} = x_{n-3}(x_{n-4}y_{n-3} + x_{n-4}c_{n-3} + y_{n-3}c_{n-3}) = x_{n-3}x_{n-4}c_{n-3}$$

$$c_{n-3} = x_{n-5}y_{n-4} + x_{n-5}c_{n-4} + y_{n-4}c_{n-4}$$

$$c_{n-1} = x_{n-3}x_{n-4}c_{n-3} = x_{n-3}x_{n-4}(x_{n-5}y_{n-4} + x_{n-5}c_{n-4} + y_{n-4}c_{n-4}) = x_{n-3}x_{n-4}x_{n-5}c_{n-4}$$

$$c_{n-1} = x_{n-3}x_{n-4}x_{n-5}x_{n-6}\cdots x_{j+2}x_{j+1}c_{j+2}$$

$$c_{j-2} = x_j y_{j+1} + x_j c_{j+1} + y_{j+1}c_{j+1} = 0y_{j+1} + 0c_{j+1} + y_{j+1}c_{j+1} = y_{j+1}c_{j+1}$$

$$c_{n-1} = x_{n-3}x_{n-4}x_{n-5}x_{n-6}\cdots x_{j+2}x_{j+1}c_{j+2} = x_{n-3}x_{n-4}x_{n-5}x_{n-6}\cdots x_{j+2}x_{j+1}y_{j+1}c_{j+1} = 0$$

Cazul 3)

În acest caz atât restul parțial cât şi împărțitorul sunt numere pozitive iar operația care se efectuează între ele este cea de scădere (aşa cum se observă din tabelul 4.4 în care $q_{m-2}=1$). De fapt, se realizează tot adunare, şi anume se însumează restul parțial (bineînțeles dublat prin deplasare) cu negatul împărțitorului care devine număr negativ. Din acest punct demonstrația devine foarte asemănătoare cu cea din cazul 11.

Cazul 12)

În acest caz atât restul parțial cât şi împărțitorul sunt numere negative iar operația care se efectuează între ele este cea de scădere ($q_{m-2}=1$). De fapt, tot operația de adunare se realizează, şi anume se însumează restul parțial (dublat prin deplasare) cu negatul împărțitorului care devine număr pozitiv. Din acest punct demonstrația devine foarte asemănătoare cu cea din cazul 4.

4.2.2.1. Soluția bazată pe un sumator/scăzător cu circuitele de transport duplicate

Aşa cum s-a menționat anterior, prin simularea unor modele cu structuri de împărțire paralelă constituite din celule de împărțire ca cea din figura 4.5, am observat că defecțiunile singulare introduse în schemă nu produceau tot timpul erori detectabile prin paritate. Defecțiunile inserate erau de tip "blocare la" "0" sau "1" logic. S-a confirmat faptul că doar defecțiunile apărute în circuitele pentru determinarea transportului nu se pot detecta prin paritate. Acest neajuns se înlătură apelând la metoda tradițională de duplicare a circuitelor de transport pentru fiecare bit de transport. Astfel, aşa cum se observă din figura 4.8, de data aceasta, avem două copii de la fiecare bit de transport; una care se transmite mai departe ca şi transportul original participând la operația de adunare/scădere, şi a doua, cea stelată din figura 4.8, care intervine doar în calculul parității. Astfel, relația (4.19), adică ecuația de paritate se poate rescrie în felul următor:

$$P_X \oplus k \cdot P_Y \oplus (m \bmod 2)(P_Y \oplus y_{n-1}) \oplus \sum_{i=0}^{m-1} \sum_{j=(n \bmod 2)}^{n} c^*_{i,j} \oplus \sum_{j=(n \bmod 2)}^{n-1} c^*_{c,j} \oplus \sum_{i=1}^{m-1} r_{i,n-1} \oplus \sum_{j=0}^{n-1} r_j \oplus \sum_{i=0}^{m-1} q'_i \oplus \sum_{i=0}^{m-1} q^*_{c_i} = 0$$

unde, n este lungimea operanzilor, iar m este numărul de niveluri ale structurii de împărțire celulară.

În felul acesta, controlul de paritate aplicat unei structuri de împărțire celulară cu circuitele de transport duplicate va detecta toate erorile de transport cauzate de defecțiuni singulare

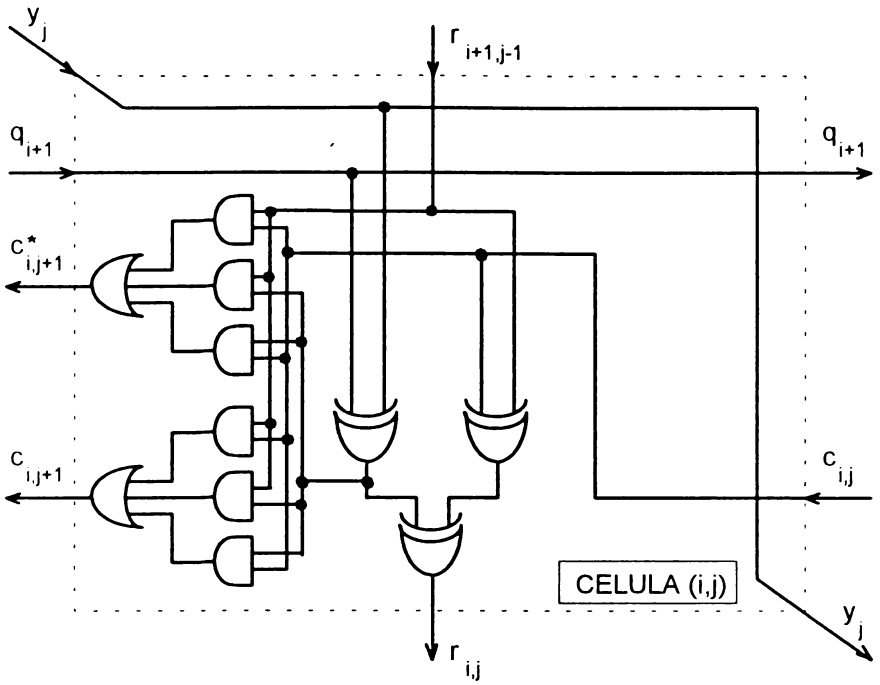


Figura 4.8 Celula de împărțire cu circuitele de transport duplicate.

Duplicarea transportului nu este necesară pentru celulele care generează cel mai semnificativ bit al transportului din fiecare nivel. Explicația este tocmai proprietatea demonstrată în lema 1 referitor la celulele din rangul cel mai semnificativ dintr-un împărțitor celular, şi anume că, bitul de transport este tot timpul complementul bitului de rezultat.

4.2.2.2. O altă soluție bazată pe un sumator/scăzător cu rezultat dependent de transport

Defecțiunile circuitelor pentru determinarea transportului (carry) la o celulă i oarecare de sumator, de obicei afectează bitul de transport c_i. Spun de obicei fiindcă există situații la care efectul defecțiunii nu se propagă până la linia de transport. Atunci când bitul de transport al celulei i este eronat cauzează tot timpul un pachet cu număr par de erori $c_i, s_{i+1}, \ldots, c_{i+t-1}, s_{i+t}$. Acest fapt împiedică detecția erorilor de transport prin paritate. Dacă fenomenul de defectare a circuitului de transport se transformă astfel încât să provoace o eroare multiplă impară (odd error burst), atunci controlul de paritate este de ajuns pentru controlarea completă a sumatorului. Sumatorul care satisface această cerință se numeşte sumator cu suma dependentă de transport (carry-dependent sum adder) [RAO89], iar în cazul nostru, deoarece este vorba de celule care realizează într-un mod controlat fie adunare fie scădere, îl putem numi mai precis sumator/scăzător cu rezultat dependent de transport. Astfel, bitul eronat c_i provoacă să fie eronat şi bitul de sumă (rezultat) s_i obținând o eroare multiplă cu număr impar de erori. Acest lucru poate fi obținut utilizând o ecuație pentru sumă (rezultat) de felul următor:

$$s_i = f_i \oplus c_i \qquad (4.20)$$

unde f_i este o funcție logică cu parametri a_i, b_i, c_{i-1}. Funcția f_i poate fi obținută din tabelul 4.5, şi anume:

$$f_i = \overline{a_i b_i c_{i-1} + \bar{a}_i \bar{b}_i \bar{c}_{i-1}} \qquad (4.21)$$

a_i	b_i	c_{i-1}	s_i	c_i	$f_i = s_i \oplus c_i$	G_i	T_i	I OK	I ER	II OK	II ER	III OK	III ER	IV OK	IV ER	V OK	V ER
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	1	1	0	1	0	0	0	0	1	0	0	0	0	0	0	0
0	1	0	1	0	1	0	1	0	0	0	0	1	0	0	0	1	0
0	1	1	0	1	1	0	1	0	0	0	1	0	0	0	0	0	1
1	0	0	1	0	1	0	1	0	0	0	0	1	0	0	0	1	0
1	0	1	0	1	1	0	1	0	0	0	1	0	0	0	0	0	1
1	1	0	0	1	1	1	1	0	0	0	0	0	1	0	0	0	0
1	1	1	1	1	0	1	1	1	0	0	0	0	0	0	0	0	0

Tabelul 4.5 Apendixul tabelului 4.5

În vederea alegerii dintre sintezele posibile ale elementului de structură pentru viitorul dispozitiv de împărțire, se parcurg în ceea ce urmează în baza diferitelor strategii cunoscute câteva alternative. Printre ele, cele mai importante sunt cea cu transport serial (ripple carry) şi cea cu transport anticipat (carry look-ahead).

4.2.2.2.1. Varianta cu transport serial

Aplicând teorema lui De Morgan la relația (4.21) obţinem o altă formă a funcției f_i la care nici o variabilă nu este negată:

$$f_i = \overline{a_i b_i c_{i-1} + \bar{a}_i \bar{b}_i \bar{c}_{i-1}} = \overline{a_i b_i c_{i-1}} \cdot \overline{\bar{a}_i \bar{b}_i \bar{c}_{i-1}} = \left(\overline{a_i b_i c_{i-1}}\right) \cdot \left(a_i + b_i + c_{i-1}\right) \qquad (4.22)$$

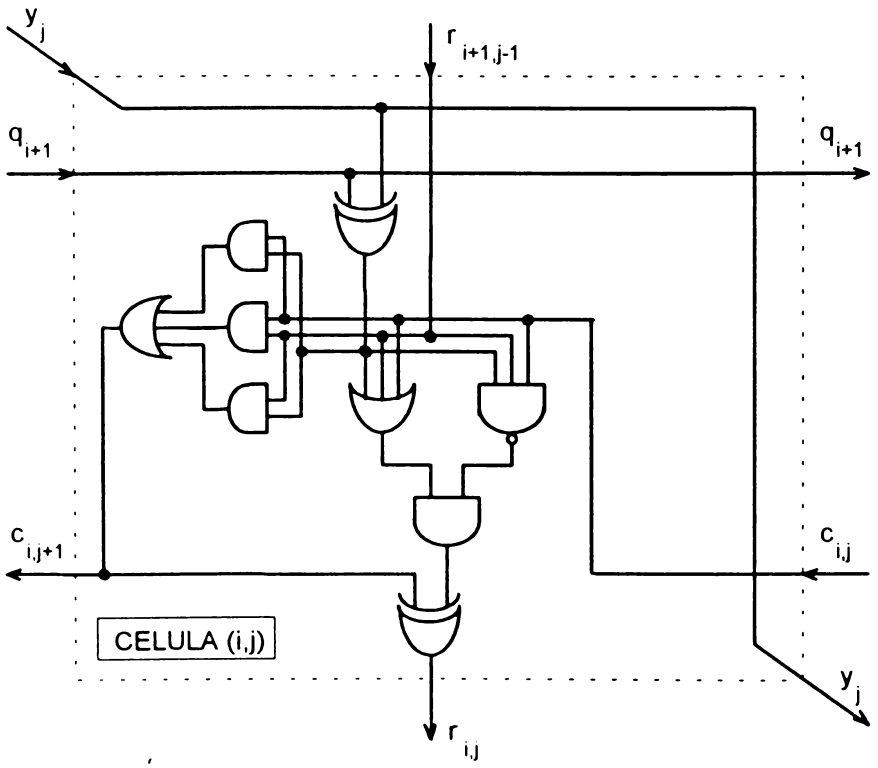


Figura 4.9 Schema logică a unei celule de sumator/scăzător cu rezultat dependent de transport

Pe baza relaţiilor (4.20) şi (4.22) am construit o celulă de împărţire, cea prezentată în figura 4.9, cu rezultat dependent de transport ţinând cont că b_i în cazul nostru este înlocuit cu $b_i = y_j \oplus q_{i+1}$ fiindcă de fapt avem nevoie de un sumator/scăzător controlat de semnalul q_{i+1}. De asemenea, corespodenţa la celelalte notaţii este următoarea:

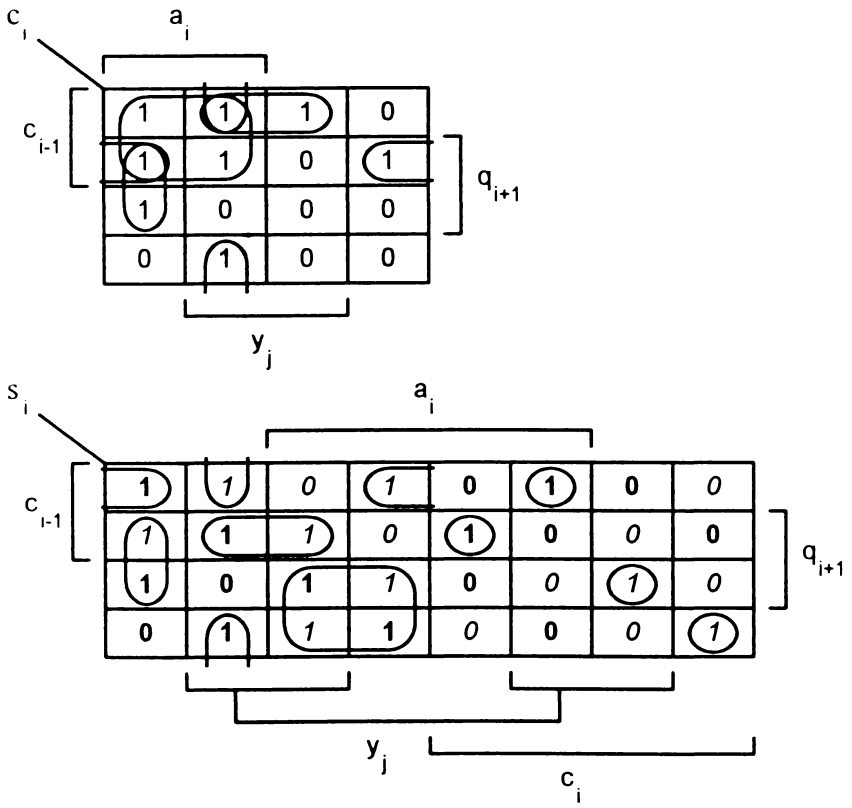
$$a_i = r_{i+1,j-1}, \quad c_i = c_{i,j+1}, \quad c_{i-1} = c_{i,j} \quad \text{şi} \quad s_i = r_{i,j} \qquad (4.23)$$

Pentru uşurinţa descrierii, vom păstra în continuare notaţiile mai simplificate şi în cazul tabelului 4.6, urmând să le înlocuim ulterior după ce obţinem forma finală a funcţiilor logice pentru transport şi sumă (rezultat).

Nr.	a_i	y_j	q_{i+1}	c_{i-1}	c_i	s_i
0	0	0	0	0	0	0
1	0	0	0	0	1	
2	0	0	0	1	0	1
3	0	0	0	1	1	
4	0	0	1	0	0	1
5	0	0	1	0	1	
6	0	0	1	1	0	
7	0	0	1	1	1	0
8	0	1	0	0	0	1
9	0	1	0	0	1	
10	0	1	0	1	0	
11	0	1	0	1	1	0
12	0	1	1	0	0	0
13	0	1	1	0	1	
14	0	1	1	1	0	1
15	0	1	1	1	1	
16	1	0	0	0	0	1
17	1	0	0	0	1	
18	1	0	0	1	0	
19	1	0	0	1	1	0
20	1	0	1	0	0	
21	1	0	1	0	1	0
22	1	0	1	1	0	
23	1	0	1	1	1	1
24	1	1	0	0	0	
25	1	1	0	0	1	0
26	1	1	0	1	0	
27	1	1	0	1	1	1
28	1	1	1	0	0	1
29	1	1	1	0	1	
30	1	1	1	1	0	
31	1	1	1	1	1	0

Tabelul 4.6 Cele 32 de combinaţii ale celor 5 semnale care intră în calculul bitului de rezultat dintr-o celulă de sumator/scăzător

Tabelul 4.6 prezintă cele 32 de combinații posibile ale celor 5 semnale (a_i, y_j, q_{i+1}, c_{i-1} și c_i) care intervin în calculul bitului de rezultat dintr-o celulă de sumator/scăzător cu rezultatul s_i dependent de transport. Jumătate din aceste combinații presupuse teoretic nu pot apărea în mod normal, adică, atunci când celula nu conține defecte, deoarece valoarea atașată transportului c_i nu este în concordanța cu valorile semnalelor a_i, y_j, q_{i+1} și c_{i-1} (vezi relația (4.7) din subcapitolul 4.2.1.). Aceste combinații sunt excluse din discuție și s-au marcat în tabelul 4.6 prin înegrirea rândurilor respective. Pe baza rândurilor rămase din tabelul 4.6 am obținut următoarele diagramele Veitch-Karnaugh pentru funcțiile de transport și rezultat, adică c_i și respectiv s_i:



Valorile scrise cu cifre îngroșate din diagrama Veitch-Karnaugh cu cinci variabile sunt cele ale rezultatului s_i atunci când transportul c_i este corect, iar valorile rezultatului s_i scrise cu cifre înclinate sunt cele la care transportul este eronat. Se observă că valorile logice ale rezultatului s_i cu cifre înclinate sunt alese să fie complementare față de cele care trebuiau să fie în cazul în care transportul nu era eronat. În acest mod se reușește deziderатul proiectării celulelor S/S cu rezultat dependent de transport, și anume acela că apariția unei erori la bitul de transport c_i determină de asemenea complementarea (intrarea în eroare) bitului de rezultat s_i, făcând astfel numărul total de biți eronați din circuit să fie impar, și bineînțeles detectabil prin mijloacele controlului de paritate.

Din diagramele Veitch-Karnaugh de mai sus, construite în maniera prezentată, după minimizare, putem extrage ecuațiile logice ale funcțiilor de transport c_i și rezultat s_i. Astfel avem:

$$c_i = a_i\overline{y}_j q_{i+1} + a_i y_j \overline{q}_{i+1} + y_j \overline{q}_{i+1} c_{i-1} + \overline{y}_j q_{i+1} c_{i-1} + a_i c_{i-1} = \overline{\overline{a_i\overline{y}_j q_{i+1} + a_i y_j \overline{q}_{i+1} + y_j \overline{q}_{i+1} c_{i-1} + y_j q_{i+1} c_{i-1} + a_i c_{i-1}}} \Rightarrow$$

$$\Rightarrow c_i = \overline{a_i\overline{y}_j q_{i+1}} \cdot \overline{a_i y_j \overline{q}_{i+1}} \cdot \overline{y_j \overline{q}_{i+1} c_{i-1}} \cdot \overline{\overline{y}_j q_{i+1} c_{i-1}} \cdot \overline{a_i c_{i-1}}$$

$$s_i = a_i\overline{c}_{i-1}\overline{c}_i + \overline{y}_j q_{i+1}\overline{c}_{i-1}\overline{c}_i + y_j q_{i+1} c_{i-1}\overline{c}_i + \overline{a}_i y_j \overline{q}_{i+1}\overline{c}_i + \overline{a}_i\overline{y}_j q_{i+1}\overline{c}_i + a_i y_j \overline{q}_{i+1} c_{i-1} c_i + a_i y_j q_{i+1} c_{i-1} c_i + \overline{a}_i y_j q_{i+1}\overline{c}_{i-1} c_i + \overline{a}_i y_j \overline{q}_{i+1} c_{i-1} c_i =$$

$$= \overline{\overline{a_i\overline{c}_{i-1}\overline{c}_i + \overline{y}_j q_{i+1}\overline{c}_{i-1}\overline{c}_i + y_j q_{i+1} c_{i-1}\overline{c}_i + \overline{a}_i y_j \overline{q}_{i+1}\overline{c}_i + \overline{a}_i\overline{y}_j q_{i+1}\overline{c}_i + a_i y_j \overline{q}_{i+1} c_{i-1} c_i + a_i y_j q_{i+1} c_{i-1} c_i + \overline{a}_i y_j q_{i+1}\overline{c}_{i-1} c_i + \overline{a}_i y_j \overline{q}_{i+1} c_{i-1} c_i}} =$$

$$\Rightarrow s_i = \overline{a_i\overline{c}_{i-1}\overline{c}_i} \cdot \overline{\overline{y}_j q_{i+1}\overline{c}_{i-1}\overline{c}_i} \cdot \overline{y_j q_{i+1} c_{i-1}\overline{c}_i} \cdot \overline{\overline{a}_i y_j \overline{q}_{i+1}\overline{c}_i} \cdot \overline{\overline{a}_i\overline{y}_j q_{i+1}\overline{c}_i} \cdot \overline{a_i y_j \overline{q}_{i+1} c_{i-1} c_i} \cdot \overline{a_i\overline{y}_j q_{i+1} c_{i-1} c_i} \cdot \overline{\overline{a}_i y_j q_{i+1}\overline{c}_{i-1} c_i} \cdot \overline{\overline{a}_i\overline{y}_j \overline{q}_{i+1} c_{i-1} c_i}$$

(4.24)

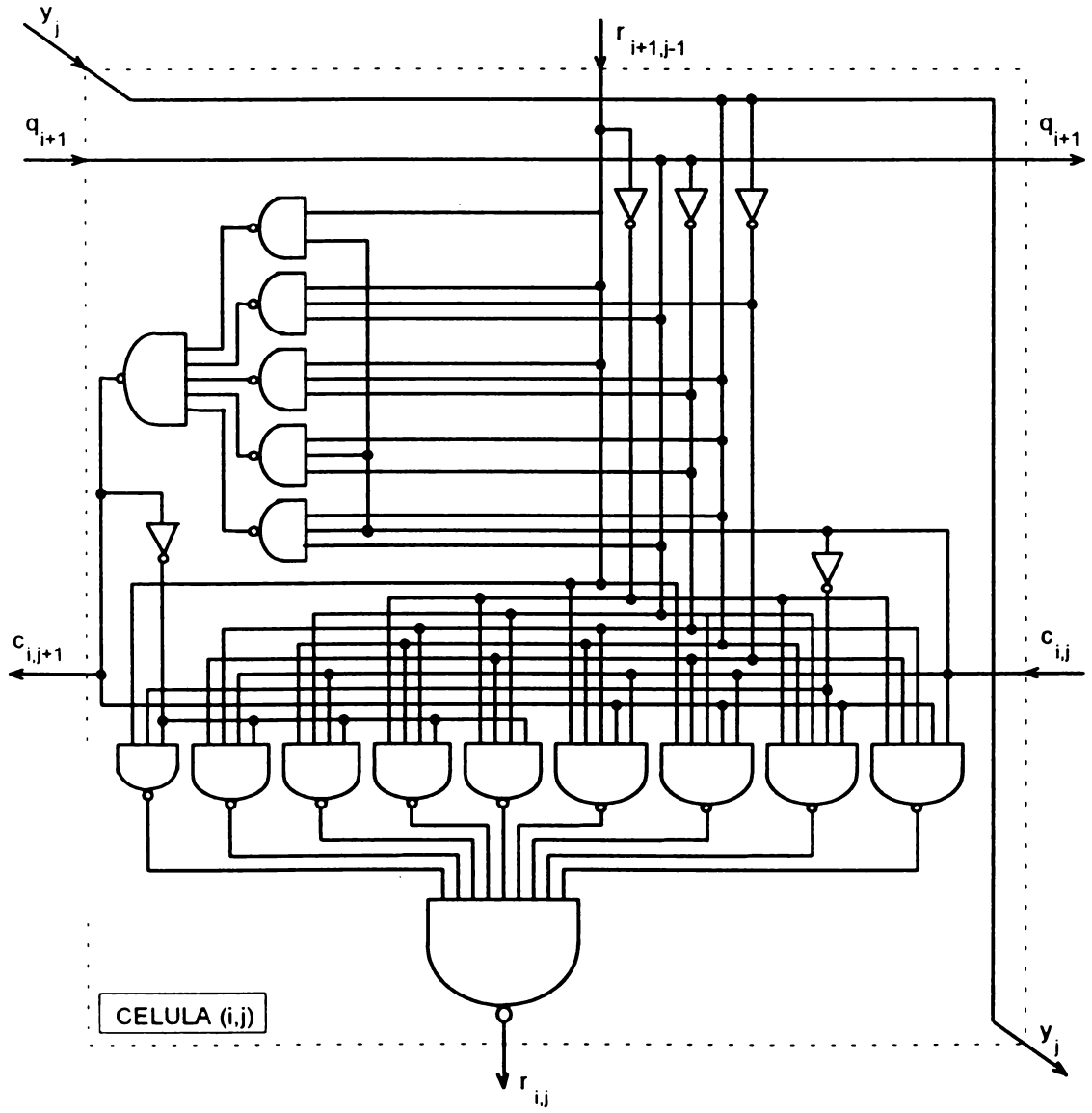


Figura 4.10 Implementarea cu circuite de tip INVERTOR şi ŞI-NU a unei celule de sumator/scăzător cu rezultat dependent de transport.

În continuare, înlocuind în relațiile (4.24) notațiile din relațiile (4.23) obținem următoarele:

$$\Rightarrow c_{i,j-1} = \overline{\overline{r_{i-1,j-1}\bar{y}_j q_{i+1}} \cdot \overline{r_{i+1,j-1} y_j \bar{q}_{i+1}} \cdot \overline{y_j \bar{q}_{i+1} c_{i,j}} \cdot \overline{\bar{y}_j q_{i+1} c_{i,j}} \cdot \overline{r_{i+1,j-1} c_{i,j}}}$$

$$\Rightarrow r_{i,j} = \overline{\overline{r_{i+1,j-1}\bar{c}_{i,j}\bar{c}_i} \cdot \overline{\bar{y}_j q_{i+1}\bar{c}_{i,j}\bar{c}_i} \cdot \overline{y_j q_{i+1} c_{i,j}\bar{c}_i} \cdot \overline{\bar{r}_{i+1,j-1} y_j \bar{q}_{i+1}\bar{c}_i} \cdot \overline{\bar{r}_{i+1,j-1}\bar{y}_j q_{i+1}\bar{c}_i} \cdot}$$
$$\overline{\cdot \bar{r}_{i+1,j-1} y_j \bar{q}_{i+1} c_{i,j} c_i} \cdot \overline{r_{i+1,j-1}\bar{y}_j q_{i+1}\bar{c}_{i,j} c_i} \cdot \overline{\bar{r}_{i+1,j-1} y_j q_{i+1}\bar{c}_{i,j} c_i} \cdot \overline{\bar{r}_{i+1,j-1}\bar{y}_j \bar{q}_{i+1} c_{i,j} c_i} \quad (4.25)$$

Astfel, am obținut funcțiile logice ale transportului şi rezultatului pentru celula i,j din dispozitivul de împărțire. O celulă sumator/scăzător (S/S) de acest gen este prezentată în figură 4.10 implementată cu logica NAND/NAND.

O astfel de celulă s-ar putea să fie mai uşor de implementat în tehnologia VLSI decât celula din figura 4.9, chiar dacă aceasta din urmă are doar 9 porți logice față de cele 16 porți ŞI-NU plus cele 5 porți INVERTOARE care o formează pe prima. Din punct de vedere al timpului de propagare nu există mare diferență în ceea ce priveşte numărul de niveluri de porți, iar în schimb depinde foarte mult de tehnologia de fabricație. În vederea accelerării vitezei de lucru a unei celule de sumator/scăzător şi prin urmare a întregii structuri de împărțire celulară vom apela, în ceea ce urmează, la metode de anticipare a transportului cunoscute pentru sumatoare.

4.2.2.2.2. Varianta cu anticiparea transportului

Revenim în tabelul 4.5 de unde se poate obține expresia lui f_i şi anume:

$$f_i = \overline{a_i b_i c_{i-1} + \bar{a}_i \bar{b}_i \bar{c}_{i-1}} = \overline{G_i c_{i-1} + \bar{T}_i \bar{c}_{i-1}} \quad (4.26)$$

unde, $G_i = a_i \cdot b_i$ este funcția de generare şi $T_i = a_i + b_i$ este funcția de propagare a transportului pentru rangul i, unde $0 \le i \le n-1$, la un sumator cu n ranguri.
Astfel, putem obține ecuația bitului de sumă pentru rangul i.

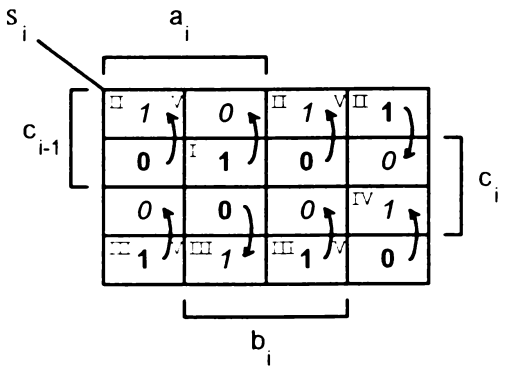
$$s_i = f_i \oplus c_i = \left(\overline{G_i c_{i-1} + \bar{T}_i \bar{c}_{i-1}}\right) \oplus c_i = \left(G_i c_{i-1} + \bar{T}_i \bar{c}_{i-1}\right) c_i + \left(\overline{G_i c_{i-1} + \bar{T}_i \bar{c}_{i-1}}\right) \bar{c}_i =$$

$$= \underset{(I)}{G_i c_{i-1} c_i} + \underset{(II)}{\bar{G}_i c_{i-1} \bar{c}_i} + \underset{(III)}{T_i \bar{c}_{i-1} \bar{c}_i} + \underset{(IV)}{\bar{T}_i \bar{c}_{i-1} c_i} + \underset{(V)}{\bar{G}_i T_i \bar{c}_i} \quad (4.27)$$

În partea dreaptă a tabelului 4.5 sunt prezentate comparativ valorile fiecărui termen din relația (4.27) în două cazuri: (cazul OK) când bitul de transport c_i are valoarea corectă şi (cazul ER) când bitul de transport c_i are valoare opusă față de cea corectă. Se poate observa că ultimul termen, termenul (V) din relația (4.27), se poate elimina fiind acoperit de termenii (II) şi (III). Termenul (IV) este tot timpul nul atunci când nu apare eroare de transport. Totuşi nu se poate elimina pentru că după cum se poate observa din prima linie a tabelului 4.5, în caz de eroare el devine 1 forțând astfel şi bitul de rezultat s_i să fie eronat.

Diagrama Karnaugh care urmează scoate în evidență observațiile de mai sus, poate într-un mod mai familiar sau mai sugestiv. Valorile cu cifre îngroşate sunt cele

ale rezultatului atunci când transportul este corect, iar valorile rezultatului cu cifre înclinate sunt cele la care transportul este eronat. Săgețile arată cum se schimbă valoarea rezultatului s_i în funcție de corectitudinea bitului de transport c_i. În fiecare element al tabelului care conține un 1 este trecut numărul termenului care asigură această valoare. Redundanța ultimului termen (V) este evidentă.



Deci, ecuația pentru rezultat este: $s_i = G_i c_{i-1} c_i + \overline{G}_i c_{i-1} \overline{c}_i + T_i \overline{c}_{i-1} \overline{c}_i + \overline{T}_i \overline{c}_{i-1} c_i$

iar ecuația pentru transport este: $c_i = a_i \cdot b_i + (a_i + b_i) \cdot c_{i-1} = G_i + T_i \cdot c_{i-1}$

Cu ajutorul relațiilor (4.23) relațiile de mai sus devin:

$$r_{i,j} = G_{i,j} c_{i,j} c_{i,j+1} + \overline{G}_{i,j} c_{i,j} \overline{c}_{i,j+1} + T_{i,j} \overline{c}_{i,j} \overline{c}_{i,j+1} + \overline{T}_{i,j} \overline{c}_{i,j} c_{i,j+1}$$

$$c_{i,j+1} = G_{i,j} + T_{i,j} \cdot c_{i,j} \qquad (4.28)$$

Pentru a accelera procesul de însumare/scădere, putem utiliza implementarea paralelă a ecuației (4.28), metodă cunoscută în literatură sub denumirea anticiparea transportului (carry look-ahead) [RAO89].

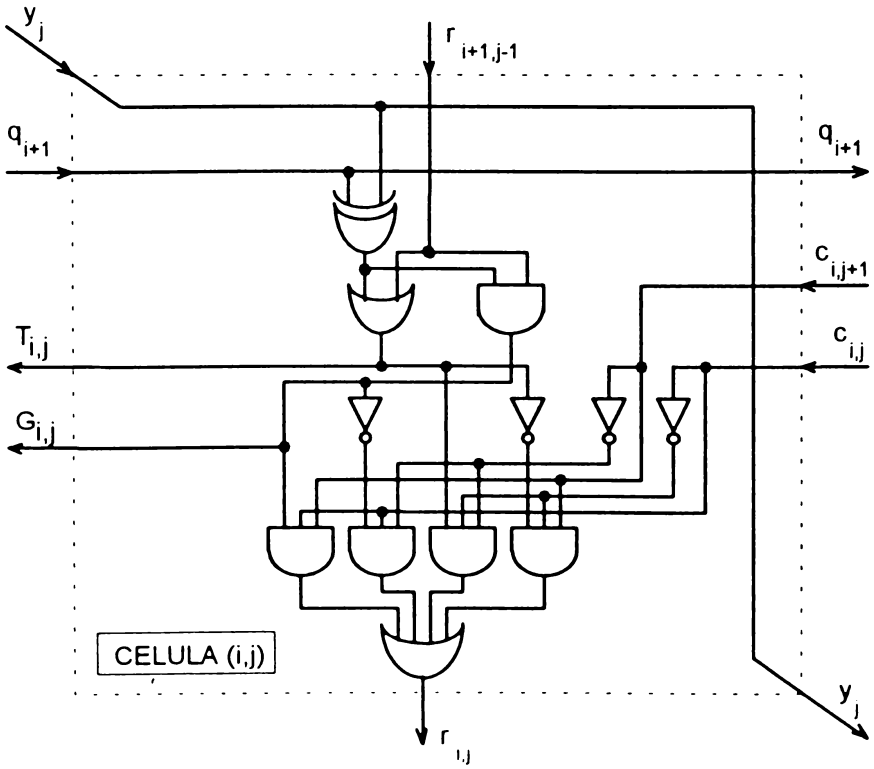


Figura 4.11 Circuitul de rezultat al unei celule sumator/scăzător.

$$c_{i,n'-1} = G_{i,n'-2} + T_{i,n'-2} \cdot c_{i,n'-2} = G_{i,n'-2} + T_{i,n'-2} \cdot \left(G_{i,n'-3} + T_{i,n'-3} \cdot c_{i,n'-3}\right) =$$

$$= G_{i,n'-2} + T_{i,n'-2} \cdot G_{i,n'-3} + T_{i,n'-2} \cdot T_{i,n'-3} \cdot c_{i,n'-3} =$$

$$= G_{i,n'-2} + T_{i,n'-2} \cdot G_{i,n'-3} + T_{i,n'-2} \cdot T_{i,n'-3} \cdot G_{i,n'-4} + \cdots + T_{i,n'-2} \cdot T_{i,n'-3} \cdot T_{i,n'-4} \cdots T_{i,0} \cdot c_{i,0} \quad (4.29)$$

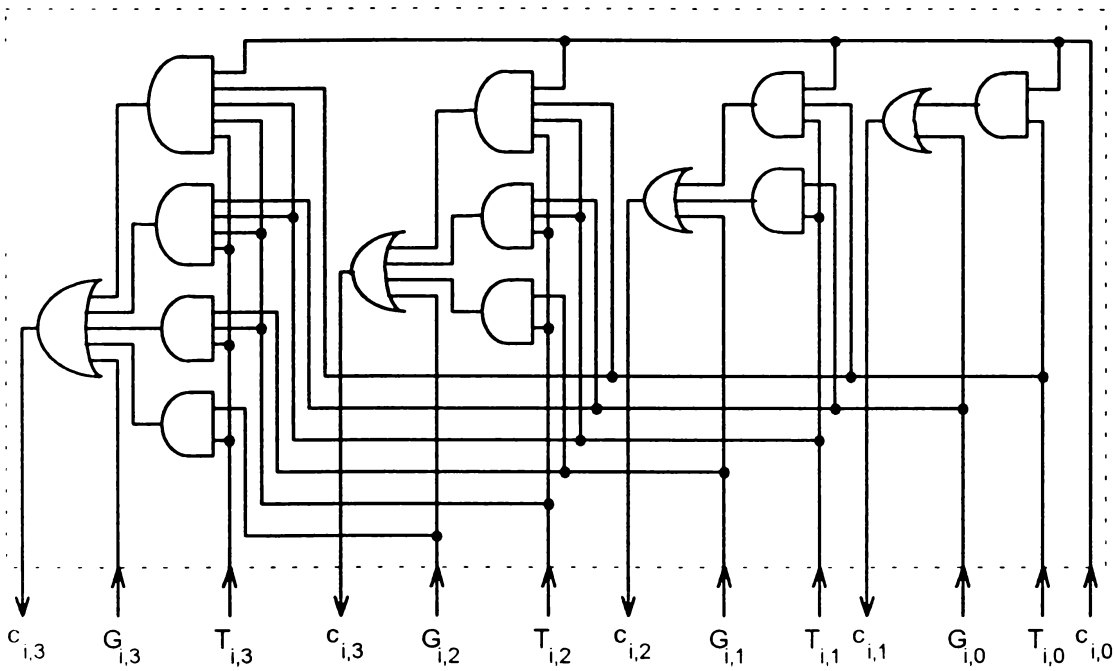


Figura 4.12 Circuitul de anticipare a transportului pentru patru celule S/S adiacente.

Din relația (4.29) se observă că bitul de transport nu depinde de bitul de transport precedent ci numai de funcțiile de generare şi propagare ale transportului. Figura 4.11 prezintă circuitul de rezultat al unei celule S/S, iar fiecare câteva celule adiacente de acest tip trebuie să împartă un circuit de anticipare a transportului asemănător celui prezentat în figura 4.12. Numărul de celule adiacente care se deservesc din acelaşi circuit trebuie ales în funcție de raportul dorit performanță/cost. Cu cât numărul acesta este mai mare cu atât schema este mai rapidă, dar creşterea costului este din ce în ce mai substanțială.

4.3. Evaluarea redundanței hardware a structurii de împărțire cu control de paritate

Ne propunem în acest subcapitol să efectuăm o evaluare a numărului de porți logice necesare structurii de împărțire celulară studiată, precum şi evaluarea cantității de hardware introdus redundant în favoarea autocontrolului materializat prin controlul de paritate.

Figura 4.13 prezintă blocul unui împărțitor celular, prevăzut cu control de paritate, scoțând în relief fluxul de informație care se vehiculează în jurul unui astfel de bloc. Astfel, se pot vedea ca intrări operanzii, deîmpărțitul X şi împărțitorul Y,

împreună cu biții lor de paritate, P_X și respectiv P_Y, care s-au format în etajele anterioare ale sistemului gazdă. Câtul final Q' și restul final R reprezintă ieșirile acestui bloc, fiind de asemenea, însoțite fiecare cu un bit de paritate, $P_{Q'}$ și respectiv P_R. Trebuie subliniat faptul că această redundanță informațională de un bit de control se păstrează indiferent de numărul *n* de biți utili ai operanzilor sau ai rezultatelor.

Pe lângă semnalele, descrise mai sus, poate apărea semnalul de eroare numit "error indicator" activ pe zero. Acest semnal este rădăcina arborelui de porți SAU-EXCLUSIV unde la frunze se află toate acele semnale care constituie ecuația de paritate (vezi relația (4.19)) și în plus semnalul "test line". Ultimul se află în starea "1" logic în timpul funcționării normale, și se poziționează pe "0" logic numai în scopul testării structurii de împărțire forțând semnalul de eroare "error indicator" pe starea activă. În acest mod am obține o structură de împărțire celulară cu checker de paritate testabil. Testarea poate fi efectuată, în funcție de criticitatea aplicației, cu o periodicitate mare în momente prestabilite de către personalul de întreținere, sau cu o periodicitate mai mică de către sistemul gazdă.

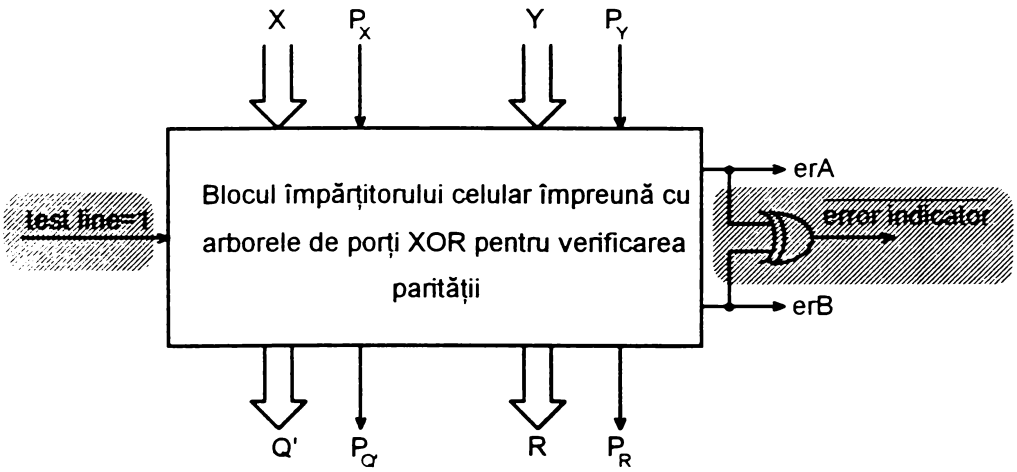


Figura 4.13 Schema generalizată a unui bloc de împărțire binară înzestrat cu autocontrol materializat prin paritate.

Se poate adopta, totuși, o altă soluție care ar spori fiabilitatea și disponibilitatea întregului sistem, pretinzând un minim de modificări, și anume, convertind circuitele de verificare din testabile în autotestabile. În cazul nostru, acest lucru înseamnă desființarea semnalului "test line" cuplând linia permanent la "1" logic, precum și eliminarea porții SAU-EXCLUSIV care conducea semnalul "error indicator" (părțile haşurate din figura 4.13). În acest mod, semnalarea erorilor se va face prin intermediul celor două semnale de eroare "erA" și "erB". Ultimele, astfel construite, reprezintă un cod dublă cale (two-rail) având valori logice complementare, fie (0,1), fie (1,0) în timpul funcționării normale fără defecțiuni. În momentul în care va apărea prima defecțiune singulară sau de multiplicitate impară acest lucru se va semnala prin afișarea aceleiași valori logice pe ambele linii, fie (0,0), fie (1,1).

Construcția arborelui de paritate trebuie realizată în așa fel încât întârzierea suplimentară adusă de operația de control să fie minimă. De asemenea, este nevoie să apară în mod explicit biții de paritate ai câtului și ai restului, $P_{Q'}$ și respectiv P_R, ca să însoțească rezultatele la ieșirea din structura de împărțire. Pentru a îndeplini primul țel, se alocă semnalelor care apar cel mai târziu pozițiile din arbore cât mai aproape de

rădăcină, iar pentru al doilea țel, se grupează biții câtului final şi ai restului final aşa cum se arată în figura 4.14 formând subarbori distincți.

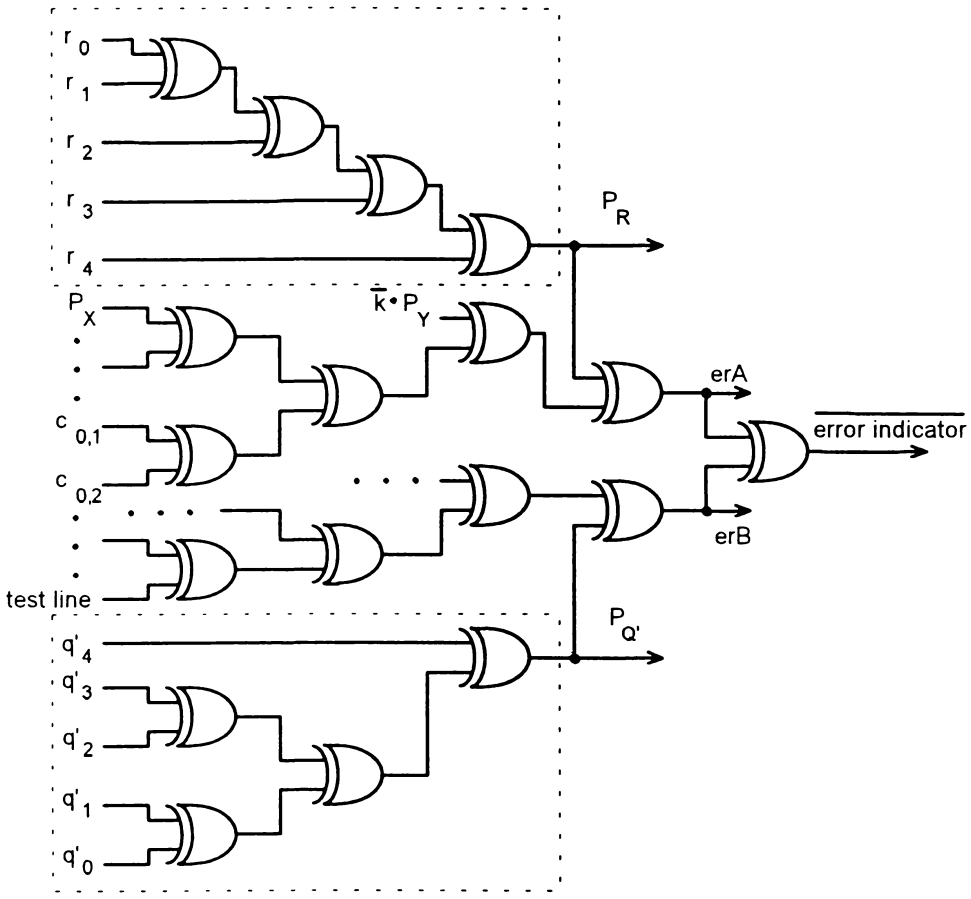


Figura 4.14 Arborele de paritate pentru o structură de împărțitor celular ca cea din figura 4.4 presupunând că operanzii vin cu biții lor de paritate.

Prin evaluarea timpilor de propagare s-a arătat, în subcapitolul 4.2.1.2, că cel mai semnificativ bit al restului final este cel care capătă ultimul valoarea validă în timpul efectuării unei operații de împărțire. Raportându-ne la acest semnal observăm din figura 4.14 că semnalele de eroare "erA" şi "erB" devin valide după scurgerea unui timp echivalent cu doar două niveluri de porți SAU-EXCLUSIV. Această diferență de două niveluri de porți logice rămâne constantă indiferent de mărimea structurii de împărțire, lucru care ne satisface pe deplin.

Arborele de paritate, construit aşa cum s-a precizat mai sus, este comun pentru toate structurile de împărțire celulară din această lucrare, indiferent de tipul celulei utilizate. În ceea ce urmează, se vor trata separat cele două variante constructive prin prisma redundanței hardware.

4.3.1. Evaluarea redundanței în cazul duplicării transportului

Numărul de porți logice dintr-o structură de împărțire fără redundanță, ca cea prezentată în figura 4.4, constituită din celule de tipul celei prezentate în figura 4.5, este notat cu N_{pfr}. Prin enumerare rezultă următoarea formulă de evaluare:

$$Npfr = Npc \cdot (m+1) \cdot n + n + 2 + 2 \cdot (m+1) + 2 \cdot m - 1 = Npc \cdot (m+1) \cdot n + n + 4 \cdot m + 3 \Rightarrow$$

$$\Rightarrow Npfr = Npc \cdot m \cdot n + 8 \cdot n + 4 \cdot m + 3 \qquad (4.30)$$

unde, *Npc* reprezintă numărul de porți dintr-o celulă (de exemplu, *Npc*=7 pentru celula din figura 4.5, iar *Npc*=9 pentru cea din figura 4.9), *n* reprezintă lungimea împărțitorului Y, iar *m* reprezintă numărul de niveluri ale structurii de împărțire.

Numărul de porți SAU-EXCLUSIV care formează arborele calculului parității în vederea controlului prin paritate a unei structuri de împărțire se notează cu *Npap*. Pe baza relației (4.19) se poate ajunge la următoarea formulă de evaluare a numărului de porți *Npap*:

$$Npap = 2 + (m \bmod 2) \cdot 1 + m \cdot (n+1) - (n \bmod 2) \cdot m + n - (n \bmod 2) \cdot 1 + m - 1 + n + 2 \cdot m \Rightarrow$$

$$\Rightarrow Npap = m \cdot (n+1) + 3 \cdot m + 2 \cdot n + 1 + (m \bmod 2) \cdot 1 - (n \bmod 2) \cdot (m+1) \qquad (4.31)$$

Numărul de porți logice necesare pentru duplicarea transportului în scopul sporirii eficacității controlului prin paritate a unei structuri de împărțire se notează cu *Npdt* şi se poate estima cu ajutorul următoarei formule:

$$Npdt = Npt \cdot (m+1) \cdot n + m - 1 + m = Npt \cdot (m+1) \cdot n + 2 \cdot m - 1 \Rightarrow$$

$$\Rightarrow Npdt = Npt \cdot m \cdot n + Npt \cdot n + 2 \cdot m - 1 \qquad (4.32)$$

unde, *Npt* reprezintă numărul de porți necesare circuitului pentru determinarea transportului (de exemplu, *Npt*=4 în cazul celulei din figura 4.5).

În tabelul 4.7 sunt prezentate valorile acestor numere de porți logice pentru structuri de împărțire de diferite mărimi. De asemenea, se arată procentajul de redundanță *Pred* al structurii finale, calculat cu formula (4.33), precum şi procentajul de creştere a hardware-ului *Pcre* raportat la structura neredundantă, calculat cu formula (4.34).

$$Pred = \frac{Npap + Npdt}{Nptot} \cdot 100\% \qquad (4.33)$$

$$Pcre = \frac{Npap + Npdt}{Npfr} \cdot 100\% \qquad (4.34)$$

Număr de porți logice	**m=5, n=5**	**m=8, n=8**	**m=6, n=10**	**m=16, n=10**	**m=16, n=16**	**m=32, n=32**	**m=64, n=64**	**m=128, n=128**
Npfr	238	547	527	1267	1987	7555	29443	116227
Npap	51	113	105	245	353	1217	4481	17153
Npdt	129	303	291	711	1119	4287	16767	66303
Nptot	**418**	**963**	**923**	**2223**	**3459**	**13059**	**50691**	**199683**
Pred (%)	43,06%	43,19%	42,90%	43,00%	42,55%	42,14%	41,91%	41,79%
Pcre (%)	75,63%	76,05%	75,14%	75,45%	74,08%	72,85%	72,16%	71,80%

Tabelul 4.7 Numărul de porți logice al subschemelor constituante ale structurii de împărțire pentru diferite mărimi ale acesteia în cazul duplicării transportului.

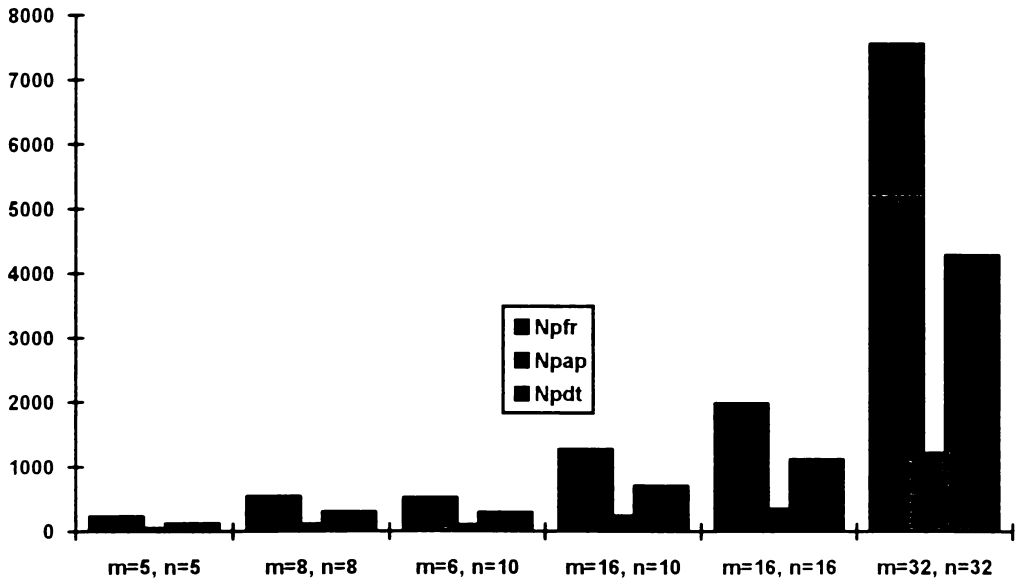


Figura 4.15 Procentajele de resurse hardware distribuite într-o structură de împărțire binară pentru valori diferite ale lui *m* și *n*.

4.3.2. Evaluarea redundanței în cazul rezultatului dependent de transport

Pentru evaluarea numărului de porți din acest caz vom considera că structura de împărțire este formată din celule de tipul celei din figura 4.9. Deci, se poate folosi, de asemenea, relația (4.30) având de această dată *Npc*=9 (numărul de porți al unei celule). Astfel, se notează cu *Nprdt* numărul de porți logice dintr-o structură de împărțire în cazul rezultatului dependent de transport obținut prin relația (4.30).

Formula pentru definirea numărului de porți SAU-EXCLUSIV (notat anterior cu *Npap*) care formează arborele calculului parității rămâne valabilă, conform relației (4.31).

Pe lângă aceste valori, pentru a obține numărul total de porți ale unei structuri mai trebuie evaluat numărul porților suplimentare, notat cu *Npsdt*, care sunt necesare în subcircuitul (CC) destinat pentru corecția câtului. Din tabelul 4.8, care prezintă cele patru stări posibile ale semnalelor de intrare ale unui circuit pentru corecția unui bit al câtului (CC), rezultă funcția f_i și anume:

$$f_i = \overline{q}_i \cdot q_{ci} + q_i \cdot \overline{q}_{ci} + q_i \cdot q_{ci} = q_i + q_{ci} \qquad (4.35)$$

Cu ajutorul acestei funcții f_i se obține bitul de cât corectat q'_i, dependent de bitul de transport q_{ci+1} pe baza relației următoare:

$$q'_i = f_i \oplus q_{ci+1} \qquad (4.36)$$

Figura 4.16(b) prezintă un astfel de circuit în comparație cu circuitul inițial, fără redundanță, prezentat în 4.16(a). Se observă că este nevoie doar de o singură poartă suplimentară de tip SAU pentru fiecare bit al câtului, adică, pentru fiecare nivel al structurii de împărțire. În felul acesta, formula de evaluare a numărului de porți *Npsdt* ar fi: $Npsdt = m$

q_i	q_{ci}	q_{ci+1}	q'_i	$f_i = q'_i \oplus q_{ci+1}$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	0	1

Tabelul 4.8 Cele patru cazuri posibile ale semnalelor de intrare ale unui circuit CC.

O soluție optimă se poate obține prin utilizarea celulelor sumator/scăzător cu rezultat dependent de transport pe toată structura împărțitorului celular cu excepția subschemei de corecție a câtului la care este mai convenabilă duplicarea transportului (pe motiv că sumatorul aflat în această subschemă este unul degenerat, circuitul lui pentru determinare a unui bit de transport fiind doar o poartă ŞI cu două intrări, vezi figura 4.16(c)). În plus, nu este nevoie de bitul cel mai semnificativ de transport din subschema de corecție a câtului, transformându-se astfel formula de evaluare a numărului *Npsdt* în:

$$Npsdt = m - 1 \tag{4.37}$$

Avantajul semnificativ al utilizării duplicării transportului (doar în cadrul subschemei de corecție a câtului), nu este, bineînțeles, această poartă în minus per total, ci faptul că poarta ŞI suplimentară se pune în paralel cu cea existentă neintroducând întârzieri suplimentare (vezi figura 4.16(c)), în contrast cu metoda anterioară care introduce o poartă SAU în serie cu cea SAU-EXCLUSIV existentă (vezi figura 4.16(b)).

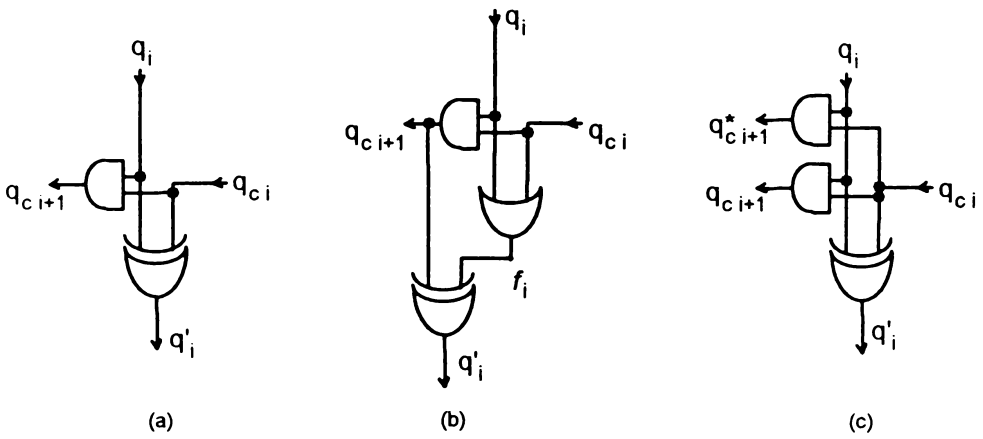


Figura 4.16 Variante de celule ale subcircuitului pentru corecția câtului: a) fără redundanță; b) cu suma dependentă de transport; c) cu duplicarea circuitului de transport.

În tabelul 4.9 se prezintă valorile acestor numere de porți logice pentru structuri de împărțire de diferite mărimi, în cazul unei implementări hibride (parțial cu rezultat dependent de transport şi parțial cu duplicarea transportului). De asemenea, se arată procentajul de redundanță *Pred* al structurii finale, calculat cu formula (4.38), precum şi procentájul de creştere a hardware-ului *Pcre* raportat la structura neredundantă, calculat cu formula (4.39).

$$Pred = \frac{Nptot - Npfr}{Nptot} \cdot 100\% \quad (4.38)$$

$$Pcre = \frac{Nptot - Npfr}{Npfr} \cdot 100\% \quad (4.39)$$

Număr de porți logice	m=5, n=5	m=8, n=8	m=6, n=10	m=16, n=10	m=16, n=16	m=32, n=32	m=64, n=64	m=128, n=128
Npfr	238	547	527	1267	1987	7555	29443	116227
Nprdt	298	691	667	1607	2531	9667	37763	149251
Npap	51	113	105	245	353	1217	4481	17153
Npsdt	4	7	5	15	15	31	63	127
Nptot	**353**	**811**	**777**	**1867**	**2899**	**10915**	**42307**	**166531**
Pred (%)	32,57%	32,55%	32,17%	32,13%	31,45%	30,78%	30,40%	30,20%
Pcre (%)	48,31%	48,26%	47,43%	47,35%	45,89%	44,47%	43,69%	43,28%

Tabelul 4.9 Numărul de porți logice al subschemelor constituante ale structurii de împărțire pentru diferite mărimi ale acesteia în cazul implementării hibride.

Cu excepția valorii *Npsdt* care trebuie incrementată cu 1, celelalte valori din tabelul 4.9 sunt aceleași și pentru cazul utilizării celulelor cu rezultatul dependent de transport.

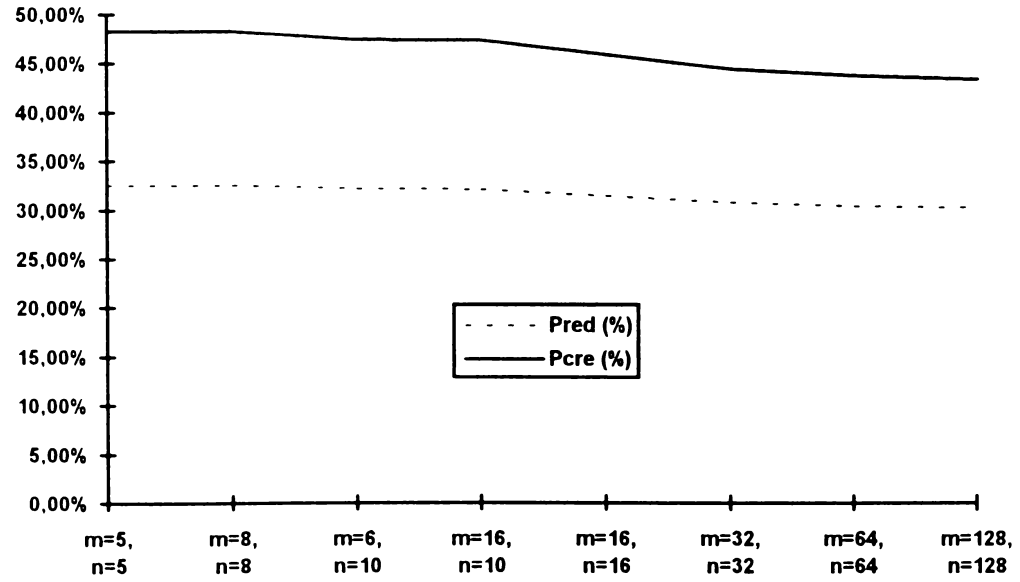


Figura 4.17 Procentajele de redundanță hardware și de creștere a hardware-ului pentru structuri de împărțire binară cu valori diferite ale lui *m* și *n*.

Precum se observă, atât din tabelul 4.9 cât și din figura 4.17, este posibilă atât construirea unor structuri de împărțire celulară cu capacitatea de a opera cu numere binare cu semn, reprezentate în complement de 2, cât și înzestrarea lor cu autocontrol materializat prin paritate în scopul detecției în timp real a erorilor de multiplicitate impară, în detrimentul doar a unei încărcări hardware suplimentare *Pcre* raportată la structura neredundantă de sub 50%. Adică, acest lucru poate fi tradus în procentajul de redundanță *Pred* al structurii finale sub 35%.

Întârzierea introdusă de circuitele verificatoare s-a precizat că este aproape nesemnificativă fiind echivalentă cu două niveluri de porți SAU-EXCLUSIV.

Din punct de vedere a fezabilității, se precizează că astfel de structuri devin din ce în ce mai tentante datorită progreselor tehnologice uriaşe din ultimii ani, precum şi din cauza necesității continue de sisteme din ce în ce mai rapide şi mai dependabile. Ca reper al tehnologiei actuale aş vrea să mă refer la CMOS pentru circuite ASIC cu densitate de împachetare între 27.000 şi 318.000 de porți logice, cu timp de propagare tipic pe o poartă NAND cu două intrări de 180 psec, şi cu până la 556 de pini pentru alimentare, masă sau semnale logice [MOTO92].

Se observă că nici numărul de porți nu reprezintă o problemă din moment ce o structură gigant, cu 128 de niveluri şi lungimea operanzilor de 128 de biți totalizând aproape 167.000 de porți, ocupă doar jumătatea unui circuit ASIC CMOS. Cu ajutorul formulelor de evaluare a timpilor de propagare prezentate în subcapitolul 4.2.1.2 şi ținând cont de timpul de propagare tipic pe poarta de 180 psec se obține un timp de propagare al întregii structuri de circa 6μsec.

5. Experimentarea prin simulare a dispozitivelor de împărțire cu autocontrol

În acest capitol se vor prezenta rezultatele experimentale obținute prin simularea schemelor concepute în capitolul anterior. Am recurs la simulare din două motive: unul, din cauza dificultăților realizării practice în cadrul facultății şi al doilea, datorită faptului că aceasta este practica modernă a proiectării de azi. Programul de simulare utilizat este conceput de autorul tezei în anii 1993-'94 în limbajul de programare C. Acest program se numeşte COMP_SIM.EXE şi este utilizat pentru scopuri didactice în cadrul orelor de laborator ale Facultății de Calculatoare din Timişoara la disciplinele "Fiabilitatea sistemelor de calcul", "Testarea sistemelor de calcul" şi "Sisteme tolerante la defecte".

Autorul consideră utilă, înaintea prezentării rezultatelor experimentale, descrierea sumară a ideilor care stau la baza acestui program de simulare precum şi prezentarea facilităților pe care le oferă.

5.1. Despre programul de simulare utilizat

5.1.1. Reprezentarea proiectării

Din cauza numărului din ce în ce mai mare de circuite cuprinse în sistemele moderne, precum şi a complexității lor ridicate se apelează la metode pentru automatizarea procesului de proiectare (design automation). În acest sens este nevoie de descrierea sistemului în conformitate cu anumite reguli creându-se astfel modelul sistemului. Acest model poate fi scris, fie într-un editor schematic (OrCAD, DIXI-CAD etc.), fie într-un editor oarecare de text, şi se foloseşte drept intrare pentru ustensilele de proiectare automată. Ustensile menționate anterior oferă, mai mult sau mai puțin interactiv, facilități ca: verificarea corectitudinii proiectului, simularea funcționării sistemului, generarea automată a vectorilor stimuli test şi calcularea acoperirii defecțiunilor, şi altele.

De asemenea, foarte importantă este clasificarea sistematică a reprezentării de proiectare, care serveşte ca şi criteriu de comparare a diferitelor tipuri de modelare. Aşa cum se arată în tabelul 5.1 [CAMP90], această clasificare implică în mod obişnuit două axe ortogonale: axa domeniilor şi cea a nivelurilor. Există trei domenii: al comportamentului, al structurii, şi cel fizic. În mod ideal, comportamentul pur este descris în termenii unei relații de intrare-ieşire, cum este un set de ecuații booleane pentru un circuit combinațional. Structura descrie topologia unui sistem şi este dată în general ca o listă de cutii conectate, cum este lista de interconexiuni de porți logice. Layout-ul este reprezentat prin figuri geometrice, care sunt poligoane adesea reduse la rectangulare.

Pentru fiecare domeniu de reprezentare, se poate defini o ierarhie de niveluri. Această ierarhie se observă cât mai bine în domeniul structural, în care obiectele modelate prin cutiile conectate determină nivelul reprezentării. De exemplu, la nivelul arhitecturii sunt procesoare, memorii şi magistrale. La nivelul transferului la registru (RTL) se află registre, unități funcționale (sumatoare şi multiplexoare), şi căi de transfer. Bistabilele şi porțile logice sunt la nivelul logic, iar în final tranzistoarele, condensatoarele şi rezistoarele sunt la nivel de dispozitiv. Atât limitele domeniilor cât şi cele ale nivelurilor se pot suprapune între ele.

Specificația de comportament urmăreşte numai descrierea funcționalității unui circuit, adică ceea ce trebuie să facă circuitul. Specificația este de obicei scrisă într-un limbaj secvențial sau procedural similar cu limbajele de programare cum sunt C şi Pascal, de exemplu VHDL-ul secvențial. Pe de altă parte, structura circuitului furnizează sugestii importante despre implementarea circuitului, adică cum este el construit. Structura este descrisă printr-o listă de conexiuni (netlist), adică o listă a componentelor şi a interconexiunilor lor.

NIVELUL	DOMENIUL		
	Comportamental	**Structural**	**Fizic**
arhitectural	Performanță, Set de instrucțiuni, Excepții	Procesoare, Memorii, Magistrale	Partiții de bază, Macrocelule
de transfer la registru	Algoritmi, Secvențe de operații	Registre, Unități funcționale, Transfere	Floorplan
logic	Tranziții de stare, Ecuații booleane, Tabele	Bistabile, Porți logice,	Celule
de dispozitive	Ecuații de rețele, Frecvența de răspuns, V(t), I(t)	Tranzistoare, Condensatoare, Rezistențe	Geometrie exactă

Tabelul 5.1 Reprezentări obişnuite de proiectare: domenii şi niveluri.

În general, reprezentarea unui proiect va avea niveluri şi domenii mixte. O descriere tipică de transfer la registru arată caracteristicile atât ale comportamentului cât şi ale structurii şi de asemenea conține componentele de la nivelul logic. Uneori chiar se poate defini un nivel de algoritm între nivelurile de arhitectură şi transfer la registru.

În cazul de față, programul de simulare COMP_SIM primeşte ca intrare modele ce se situează la nivelul logic din domeniul structural (zona haşurată din tabelul 5.1). Scrierea modelelor se poate face cu editorul de text din Norton-Commander sau cu WordStar în regim non-document.

5.1.2. Structura unui model destinat simulatorului COMP_SIM

Programul de simulare COMP_SIM, înaintea simulării propriu-zise, efectuează câteva faze preliminare, asemenea unui compilator de limbaj obişnuit. Cele mai importante faze preliminare sunt analiza lexicală şi analiza sintactică. De fapt, s-a definit un limbaj ale cărui instrucțiuni de bază sunt porțile logice şi bistabilele de tip D şi T. Astfel, descrierea unui circuit se poate face înşirând toate porțile una câte una specificând explicit semnalele de intrare şi de ieşire. Structura unui model descris în limbajul respectiv este ilustrată în figura 5.1. Deci, prezența instrucțiunii ***inputs*** urmată de două puncte este obligatorie la începutul unui model. Prin această instrucțiune se declară valorile logice ale semnalelor de intrare primare precum şi valorile logice inițiale ale bistabilelor. De asemenea, trebuie inițializate şi semnalele care provin direct din elementele de memorare formate prin legături cu reacție între porți logice.

Ceea ce este inclus între paranteze drepte este opțional şi indică doar modalitatea şi direcția de extindere. De exemplu, instrucțiunea ***clock*** are rost doar când se descriu circuite sincrone pentru a declara semnalul de tact. Valorile numerice scrise după declararea semnalul de tact determină de la al câtelea până la al câtelea tact să se afişeze rezultatele simulării. De fapt, în primele versiuni ale programului, prin această instrucțiune se specifica fereastra temporală a afişării, iar în versiunile următoare acest lucru se poate face interactiv înainte de simulare. De asemenea se poate stabili tot în mod interactiv factorul de umplere al semnalului de tact.

```
inputs: a=0[, b=1[, a1a=1[, bau=0[, c=1[, e=0]]]]];
      ⋮
[inputs: abc=1[, bcd=0[, aq1=0[, d=0]]];]
[clock: clk=30,40;]
 nor(a,b)>c;
 xor(a,c)>d;
      ⋮
[%                  comentarii                  %]
      ⋮
 and(d,a)>bcd;
 nand4(a,b,bcd,d)>abc;
 not(abc)>bau;
      ⋮
[%                  comentarii                  %]
      ⋮
 bistt(b,clk)>aq1;
 bistd(aq1,bau)>a1a;
 or8(a,b,c,d,aq1,abc,bau,a1a)>e;
      ⋮
[outputs: aq1[, a[, b[, c[, d[, e]]]]];]
      ⋮
outputs: abc[, bcd[, a1a]].
```

Figura 5.1 Structura unui model destinat simulatorului COMP_SIM.

Prin instrucțiunea ***outputs*** se specifică semnalele circuitului care se vor afişa în urma simulării. Celelalte instrucțiuni implementate în acest limbaj sunt prezentate în prima coloană a tabelului 5.3. Sensul lor este evident.

Orice comentarii pot fi incluse între două caractere "%". Ultimul caracter al unei astfel de descrieri de structură trebuie să fie ".".

Programul de simulare, aşa cum s-a menționat anterior, înainte de a trece la simularea schemei reprezentate în model, realizează fazele de analiză lexicală şi sintactică. Analizorul lexical citeşte caracterele din care se constituie programul sursă (modelul descris) şi identifică grupuri de caractere care reprezintă atomi ai programului. La solicitarea analizorului sintactic, analizorul lexical identifică următorul atom din programul sursă şi retransmite (analizorului sintactic) informații relative la acest atom. Datorită simplității gramaticii limbajului de descriere structurală adoptat, analizorul sintactic este mai simplu decât unul pentru un limbaj de programare obişnuit.

În timpul analizei lexicale şi sintactice, orice sesizare de eroare determină terminarea programului de simulare şi afişarea mesajului corespunzător de eroare, precum şi indicarea liniei din model la care s-a ajuns cu verificarea. Printre erorile

detectabile se numără: lipsa cuvintelor cheie "inputs" sau "outputs", operație necunoscută, identificator nedefinit, caracter incorect, etc. Se corectează eroarea şi se execută din nou până când se elimină toate erorile din model.

5.1.3. Principiul de funcționare al simulatorului COMP_SIM

Fişierele de intrare pentru programul COMP_SIM trebuie să aibă extensie de tipul ".DES". În timpul analizei lexicale şi sintactice se realizează ordonarea instrucțiunilor (adică a dispozitivelor logice) pornind de la cele care au numai intrări provenite de la intrările primare ale circuitului şi continuând cu cele ale căror intrări sunt semnale deduse din instrucțiunile anterioare. Dacă nu apar erori în timpul acestor procese, etapă numită precompilare, atunci programul COMP_SIM creează un fişier cu acelaşi nume ca şi fişierul sursă dar cu extensia ".PRE". Trebuie menționat faptul că pe durata acestei ordonări ieşirile elementelor de memorare se consideră intrări primare, şi că această ordine a instrucțiunilor este valabilă pentru momentul inițial al începerii simulării.

Cu ajutorul unor reguli existente în programul de simulare axate pe specificul hardware, se pot detecta eventualele greşeli de proiectare sau de scriere cum ar fi: declararea unui semnal cu valoare ilegală, apariția unui conflict la un semnal din cauza cuplării împreună a două sau a mai multor ieşiri, ieşire de bistabil neinițializată, etc. Este de notat faptul că se pot simula scheme digitale atât combinaționale cât şi secvențiale, sincrone sau asincrone.

Câmp	Înregistrări						Semnificație
ind							identificator
lg							lungimea identificatorului
valt	0	1	0	0	1	...	valoarea trecută
val	0	1	1	0	0	...	valoare actuală (folosită la afişare)
valu	0	0	0	0	1	...	valoarea următoare calculată
valp	0	1	1	0	0	...	valoarea posterioară, numai pentru bistabile
leg							legătura
actual	9	9	0	2	2	...	9-intrare primară, 2-valoare actualizată, 0-neactualizată
tprop	0	2	4	7	12	...	timp de propagare corespunzător valorii calculate

Tabelul 5.2 Tabela de simboluri (tabsim[tabmax]).

După această fază preliminară, are loc compilarea efectivă având ca fişier sursă fişierul cu extensia ".PRE". În timpul acestei faze se completează tabela de simboluri (tabelul 5.2) precum şi tabelul de execuție (tabelul 5.3), ambele constituind reprezentarea internă a modelului. Tabelul de execuție astfel format conține pe fiecare linie valoarea atomului unui dispozitiv logic, precum şi toate simbolurile semnalelor de intrare şi de ieşire ale acestuia, motiv pentru care se mai numeşte şi tabelul de intrări-ieşiri.

Fiecărui identificator definit în model îi corespunde o intrare în tabela de simboluri (tabelul 5.2). Intrările corespunzătoare identificatorilor sinonimi (din punct de vedere al algoritmului de hashing) sunt înlănțuite prin câmpul "leg". Câmpul "ind" indică în tabela identificatorilor începutul secvenței de caractere corespunzătoare identificatorului. În câmpul "lg" se memorează lungimea identificatorului.

Aşa cum se observă din tabelul 5.2, la orice moment, pentru fiecare semnal se păstrează simultan atât valoarea logică curentă "val" cât şi cea anterioară "valt" precum şi valoarea următoare "valu" dedusă în urma evaluării semnalelor. În plus, în cazul semnalelor care provin de la o ieşire a unui bistabil se păstrează şi valoarea posterioară "valp". Această valoare este intermediară valorii următoare, şi se transcrie în valoarea următoare numai în momentul apariţiei unui front pozitiv la intrarea de clock a bistabilului.

Denumire	0	1	2	3	4	5	6	7	8	9	10	11
not	6	I_1	∅	*Out*								0
amp	7	I_1	∅	*Out*								11
(and) and2	8	I_1	I_2	∅	*Out*							11
(nand) nand2	10	I_1	I_2	∅	*Out*							0
(or) or2	9	I_1	I_2	∅	*Out*							0
(nor) nor2	11	I_1	I_2	∅	*Out*							11
xor	12	I_1	I_2	∅	*Out*							
and3, nand3 or3, nor3	22, 24 23, 25	I_1	I_2	I_3	∅	*Out*						⋮
and4, nand4 or4, nor4	26, 28 27, 29	I_1	I_2	I_3	I_4	∅	*Out*					⋮
and5, nand5 or5, nor5	30, 32 31, 33	I_1	I_2	I_3	I_4	I_5	∅	*Out*				⋮
and8, nand8 or8, nor8	34, 36 35, 37	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	∅	*Out*	
bistd	38	*D*	*Clk*	∅	*Out*							0
bistt	39	*T*	*Clk*	∅	*Out*							11

Tabelul 5.3 Tabelul de intrări-ieşiri (intries[500][12]).

Reprezentarea semnalelor logice se face având la dispoziţie trei stări logice: "1" logic, "0" logic şi "X" logic, unde "X" logic reprezintă starea nedefinită (necunoscută) a unui semnal.

Pe ultima coloană din tabelul 5.3 se înscrie valoarea 0 dacă instrucţiunea nu a fost executată iar valoarea 11 în caz contrar. Execuţia unei instrucţiuni înseamnă deducerea valorii logice de la ieşirea dispozitivului logic reprezentat prin instrucţiunea respectivă, precum şi aprecierea timpului necesar pentru ca această valoare logică să apară efectiv la ieşirea dispozitivului. Aceste acţiuni se bazează pe valorile intrărilor actualizate şi pe timpii de propagare ai dispozitivului. Dacă nu există informaţii suficiente pentru a deduce valoarea logică a ieşirii unui dispozitiv, instrucţiunea se marchează ca neexecutată şi se trece la următoarea. Evaluarea valorii logice a semnalelor dintr-un model se face baleând tabelul de execuţie linie cu linie de mai multe ori până când toate instrucţiunile sunt marcate ca executate.

În primele versiuni ale programului COMP_SIM, simularea consta în efectuarea repetitivă a mecanismului descris mai sus, cu o perioadă de repetiţie suficient de mică. Dezavantajul evident al acestei soluţii este acela că pentru valori mici ale perioadei de repetiţie durata şi volumul calculelor executate în favoarea simulării este nejustificat de mare, iar pentru valori mari ale acestei perioade, scade

acuratețea simulării. Din acest motiv în versiunile următoare s-a adoptat strategia pașilor de comandă cu durata variabilă. În felul acesta se efectuează reevaluarea semnalelor modelului numai în momentul în care se modifică cel puțin un semnal dintre ele.

Momentele de tranziție ale ieșirii unui dispozitiv logic se calculează ținând cont de timpii de propagare tipici dați în catalog ai dispozitivului. Simulatorul poate opera cu timpi de propagare diferiți în funcție de tipul tranziției (front ridicător tpLH, front căzător tpHL) chiar dacă de obicei ei sunt egali. Se pot simula scheme beneficiind de baze de date cu timpi de propagare tipici ai dispozitivelor logice fundamentale descrise anterior privind tehnologiile TTL, LS, S, etc. Nu se ține cont de timpii de tranziție care de altfel sunt nesemnificativi față de timpii de propagare.

5.2. Rezultatele experimentale

Pentru verificarea corectitudinii algoritmului de împărțire propus în capitolul 4, autorul a realizat un program numit DIVISION.EXE care efectuează împărțiri cu numere binare conform acestui algoritm. Despre această verificare s-a discutat în subcapitolul 4.2.1.1.

În paralel și independent de această verificare, autorul a obținut confirmarea corectitudinii schemei originale de împărțire provenite din algoritmul propus prin simularea acesteia cu ajutorul simulatorului descris in subcapitolul 5.1.

5.2.1. Urmărirea funcționării împărțitorului cu control de paritate prin simulare

Modelul unui împărțitor cu 5 niveluri și cu operanzi pe 5 biți, redat în conformitate cu cerințele simulatorului COMP_SIM, este prezentat în anexa B. Pe baza acestui model s-au efectuat zeci de simulări demonstrând corectitudinea proiectului.

În cadrul simulării, pentru dispozitivele logice fundamentale au fost preluați din catalog timpii tipici de propagare pentru diferitele tehnologii de integrare cum sunt TTL, Schottky, LowSchottky. Când se apelează pentru prima dată la o tehnologie, programul de simulare cere utilizatorului să înregistreze setul de valori ale timpilor de propagare tipici pentru tehnologia respectivă.

Se vor folosi valorile din tehnologia LowSchottky pentru că tot ele au fost folosite în subcapitolul 4.2.1.2. pentru evaluarea timpilor de propagare în cadrul împărțitorului. Aceste valori sunt prezentate în figura 5.2, și este ecranul pe care îl prezintă programul COMP-SIM înainte de fiecare simulare.

Este adevărat că există alte tehnologii care se pot folosi la ora actuală, mult mai rapide, cum este de exemplu CMOS, dar ideea principală nu este obținerea unui timp foarte performant ca valoare absolută în urma simulării, ci compararea timpului de propagare estimat a întregii structuri de împărțire cu cel obținut prin simulare. Din moment ce proiectarea și modelarea nu este făcută la nivel de tranzistor, ci la nivel de poartă logică, trecerea de la o tehnologie la alta nu modifică decât timpii de propagare.

Figurile 5.3-5.6 ilustrează câteva cazuri de împărțire binară dintre cele posibile. Fiecare dintre figuri prezintă în partea superioară rezultatul programului DIVISION, cu confirmarea aritmetică alături, iar în partea inferioară imaginea captată în urma simulării pentru aceiași operanzi.

Figura 5.3 prezintă cazul în care în urma împărțirii nu este nevoie de corecție nici pentru cât nici pentru rest. Se menționează din nou faptul că nu este vorba de o corecție privită ca o măsură a tolerării de defecțiuni, ci, de ajustarea rezultatelor (inerentă algoritmului de împărțire fără refacerea restului) necesară în unele cazuri.

The Propagation Delay Times (LH and HL) used from TP_LS.DTA			
Gate	AMP	tpLH = 5	tpHL = 5
Gate	NOT	tpLH = 10	tpHL = 10
Gate	NAND	tpLH = 10	tpHL = 10
Gate	AND	tpLH = 9	tpHL = 9
Gate	NOR	tpLH = 10	tpHL = 10
Gate	OR	tpLH = 14	tpHL = 14
Gate	XOR	tpLH = 10	tpHL = 10
Flip-flop D		tpLH = 20	tpHL = 20
Flip-flop T		tpLH = 20	tpHL = 20
		Press any key to continue.	

Figura 5.2 Valorile timpilor de propagare ale dispozitivelor logice utilizate în simulare.

De exemplu, pentru figura 5.3 avem:
deîmpărțitul este 1.01101101 reprezentat în complement de 2, unde primul bit reprezintă semnul, astfel, reprezentarea în semn mărime este -10010011, iar valoarea sa se poate calcula prin:

$$X = -\left(1\cdot\left(1/2\right)+0\cdot\left(1/4\right)+0\cdot\left(1/8\right)+1\cdot\left(1/16\right)+0\cdot\left(1/32\right)+0\cdot\left(1/64\right)+1\cdot\left(1/128\right)+1\cdot\left(1/256\right)\right)$$

sau mai simplu prin:

$$X = -\left(1\cdot\left(128/256\right)+0\cdot\left(64/256\right)+0\cdot\left(32/256\right)+1\cdot\left(16/256\right)+0\cdot\left(8/256\right)+0\cdot\left(4/256\right)+1\cdot\left(2/256\right)+1\cdot\left(1/256\right)\right)$$

$$\Rightarrow X = -147/256;$$

de asemenea împărțitorul este 1.0011, adică -1101 şi are valoarea:

$$Y = -\left(1\cdot\left(1/2\right)+1\cdot\left(1/4\right)+0\cdot\left(1/8\right)+1\cdot\left(1/16\right)\right)$$ sau altfel

$$Y = -\left(1\cdot\left(8/16\right)+1\cdot\left(4/16\right)+0\cdot\left(2/16\right)+1\cdot\left(1/16\right)\right) \Rightarrow Y = -13/16.$$

Câtul final este ultimul conținut al registrului Q iar restul final este ultimul conținut al registrului A, iar valorile lor se calculează în mod asemănător. În ceea ce priveşte restul, există o particularitate, şi anume că trebuie ținut cont că este deplasat spre stânga de patru ori în timpul împărțirii şi deci este nevoie să se deplaseze aritmetic spre dreapta tot cu atâtea poziții, sau să se înmulțească valoarea obținută în urma efectuării operației de împărțire cu $1/2^4$ (în cazul general deplasarea este de (m-1) ori şi astfel trebuie înmulțit cu $1/2^{(m-1)}$).

```
Introduceti lungimea operanzilor n: 5
Introduceti numarul de niveluri m: 5
Introduceti deimpartitul X: 10110
Introduceti contin. reg. Q: 11010
Introduceti impartitorul Y: 10011
```

	A	Q	Count
	1 0 1 1 0	1 1 0 1 0	0
-	1 0 0 1 1		
1	0 0 0 1 1	1 1 0 1 0	
<-	0 0 1 1 1	1 0 1 0 0	1
+	1 0 0 1 1		
0	1 1 0 1 0	1 0 1 0 1	
<-	1 0 1 0 1	0 1 0 1 0	2
-	1 0 0 1 1		
1	0 0 0 1 0	0 1 0 1 0	
<-	0 0 1 0 0	1 0 1 0 0	3
+	1 0 0 1 1		
0	1 0 1 1 1	1 0 1 0 1	
<-	0 1 1 1 1	0 1 0 1 0	4
-	1 0 0 1 1		
0	1 1 1 0 0	0 1 0 1 1	

$$X = -\frac{147}{256}$$

$$Y = -\frac{13}{16}$$

$$-\frac{147}{256} = \left(-\frac{13}{16}\right) \times \frac{11}{16} + \left(-\frac{4}{256}\right) \Rightarrow$$

$$\Rightarrow -\frac{147}{256} = -\frac{143}{256} - \frac{4}{256}$$

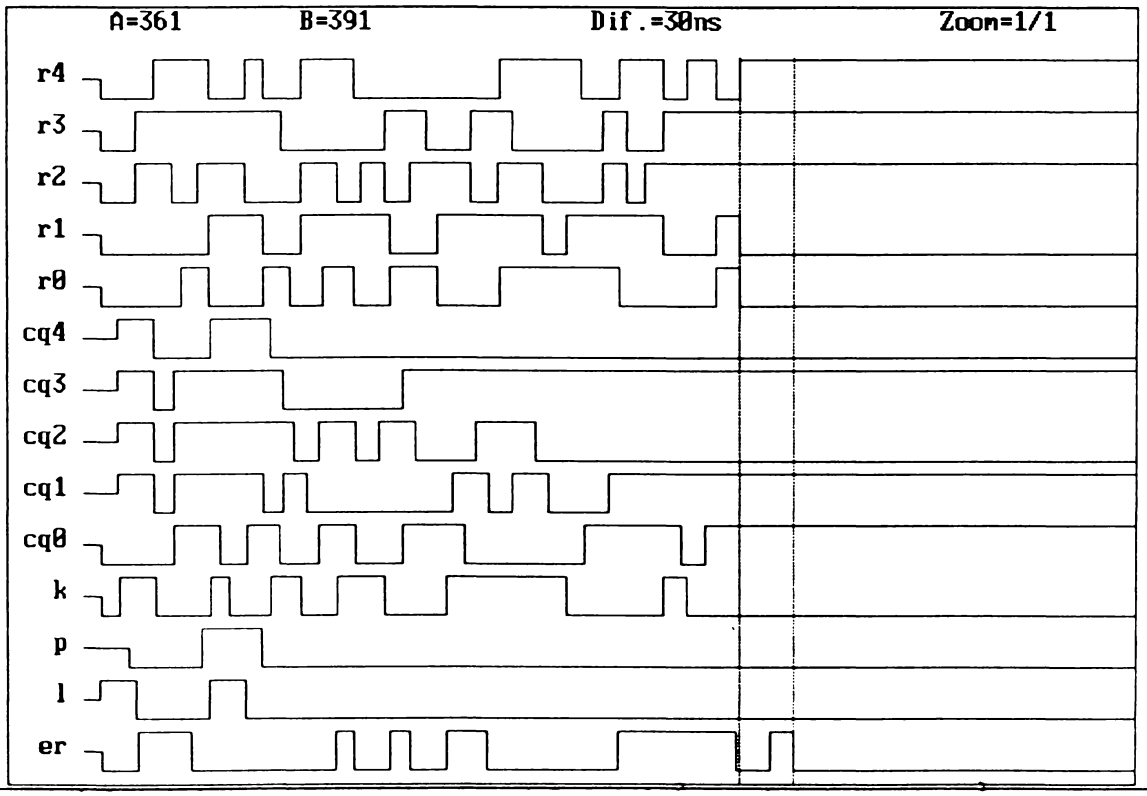


Figura 5.3 Caz de împărțire fără necesitatea vreunei corecții la cât sau la rest.

```
Introduceti lungimea operanzilor n: 5
Introduceti numarul de niveluri m: 5
Introduceti deimpartitul X: 01001
Introduceti contin. reg. Q: 10100
Introduceti impartitorul Y: 01011
```

	A	Q	Count
	0 1 0 0 1	1 0 1 0 0	0
-	0 1 0 1 1		
0	1 1 1 1 0	1 0 1 0 0	
<-	1 1 1 0 1	0 1 0 0 0	1
+	0 1 0 1 1		
1	0 1 0 0 0	0 1 0 0 1	
<-	1 0 0 0 0	1 0 0 1 0	2
-	0 1 0 1 1		
1	0 0 1 0 1	1 0 0 1 1	
<-	0 1 0 1 1	0 0 1 1 0	3
-	0 1 0 1 1		
1	0 0 0 0 0	0 0 1 1 1	
<-	0 0 0 0 0	0 1 1 1 0	4
-	0 1 0 1 1		
0	1 0 1 0 1	0 1 1 1 0	
c-R			
+	0 1 0 1 1		
1	0 0 0 0 0	0 1 1 1 0	

$$X = \frac{154}{256}$$

$$Y = \frac{11}{16}$$

$$\frac{154}{256} = \frac{11}{16} \times \frac{14}{16} + \frac{0}{256} \Rightarrow$$

$$\Rightarrow \frac{154}{256} = \frac{154}{256} + \frac{0}{256}$$

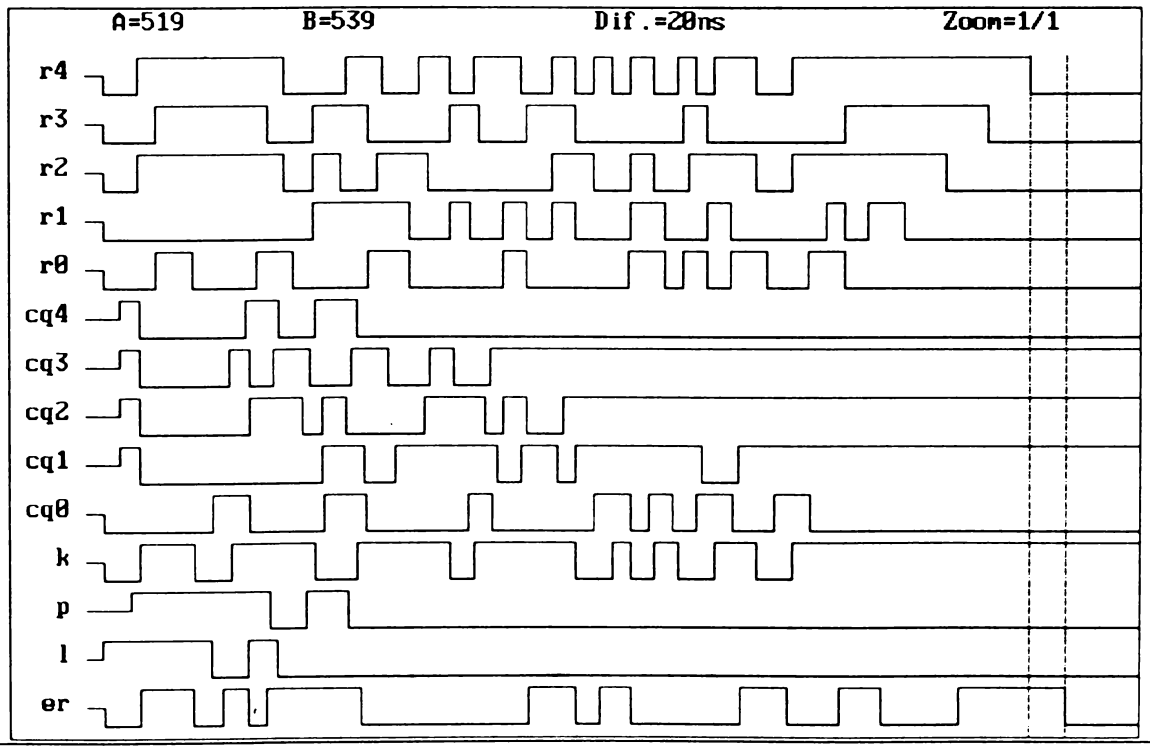


Figura 5.4 Caz de împărțire cu corecția restului.

```
Introduceti lungimea operanzilor n: 5
Introduceti numarul de niveluri m: 5
Introduceti deimpartitul X: 10010
Introduceti contin. reg. Q: 00110
Introduceti impartitorul Y: 01110
```

	A	Q	Count
	1 0 0 1 0	0 0 1 1 0	0
+	0 1 1 1 0		
1	0 0 0 0 0	0 0 1 1 1	
<-	0 0 0 0 0	0 1 1 1 0	1
-	0 1 1 1 0		
0	1 0 0 1 0	0 1 1 1 0	
<-	0 0 1 0 0	1 1 1 0 0	2
+	0 1 1 1 0		
0	1 0 0 1 0	1 1 1 0 0	
<-	0 0 1 0 1	1 1 0 0 0	3
+	0 1 1 1 0		
0	1 0 0 1 1	1 1 0 0 0	
<-	0 0 1 1 1	1 0 0 0 0	4
+	0 1 1 1 0		
0	1 0 1 0 1	1 0 0 0 0	
c- Q			
	1 0 1 0 1	1 0 0 0 1	

$$X = -\frac{221}{256}$$

$$Y = \frac{14}{16}$$

$$-\frac{221}{256} = \frac{14}{16} \times \left(-\frac{15}{16}\right) + \left(-\frac{11}{256}\right) \Rightarrow$$

$$\Rightarrow -\frac{221}{256} = -\frac{210}{256} - \frac{11}{256}$$

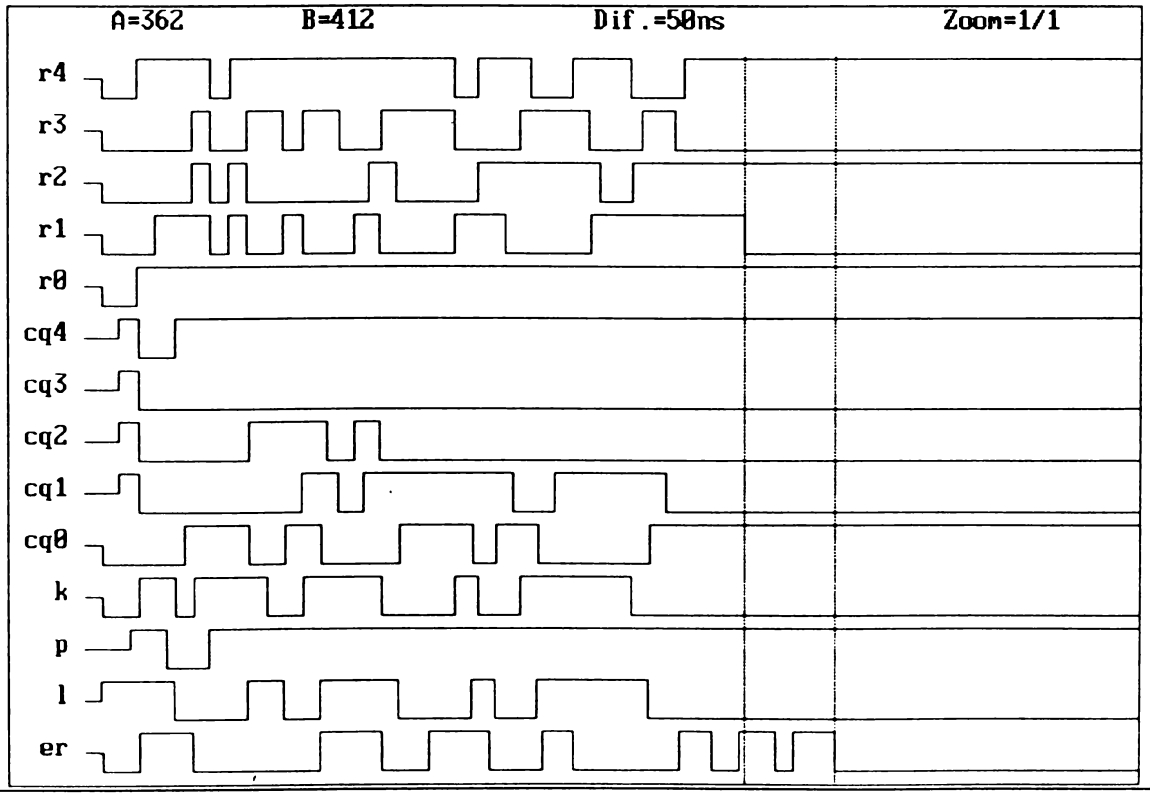


Figura 5.5 Caz de împărțire cu corecția câtului.

```
Introduceti lungimea operanzilor n: 5
Introduceti numarul de niveluri m: 5
Introduceti deimpartitul X: 01001
Introduceti contin. reg. Q: 11010
Introduceti impartitorul Y: 10011
```

	A	Q	Count
	0 1 0 0 1	1 1 0 1 0	0
+	1 0 0 1 1		
0	1 1 1 0 0	1 1 0 1 1	
<-	1 1 0 0 1	1 0 1 1 0	1
-	1 0 0 1 1		
1	0 0 1 1 0	1 0 1 1 0	
<-	0 1 1 0 1	0 1 1 0 0	2
+	1 0 0 1 1		
1	0 0 0 0 0	0 1 1 0 0	
<-	0 0 0 0 0	1 1 0 0 0	3
+	1 0 0 1 1		
0	1 0 0 1 1	1 1 0 0 1	
<-	0 0 1 1 1	1 0 0 1 0	4
-	1 0 0 1 1		
0	1 0 1 0 0	1 0 0 1 1	
c-RQ			
-	1 0 0 1 1		
1	0 0 0 0 1	1 0 1 0 0	

$$X = \frac{157}{256}$$

$$Y = -\frac{13}{16}$$

$$\frac{157}{256} = \left(-\frac{13}{16}\right) \times \left(-\frac{12}{16}\right) + \frac{1}{256} \Rightarrow$$

$$\Rightarrow \frac{157}{256} = \frac{156}{256} + \frac{1}{256}$$

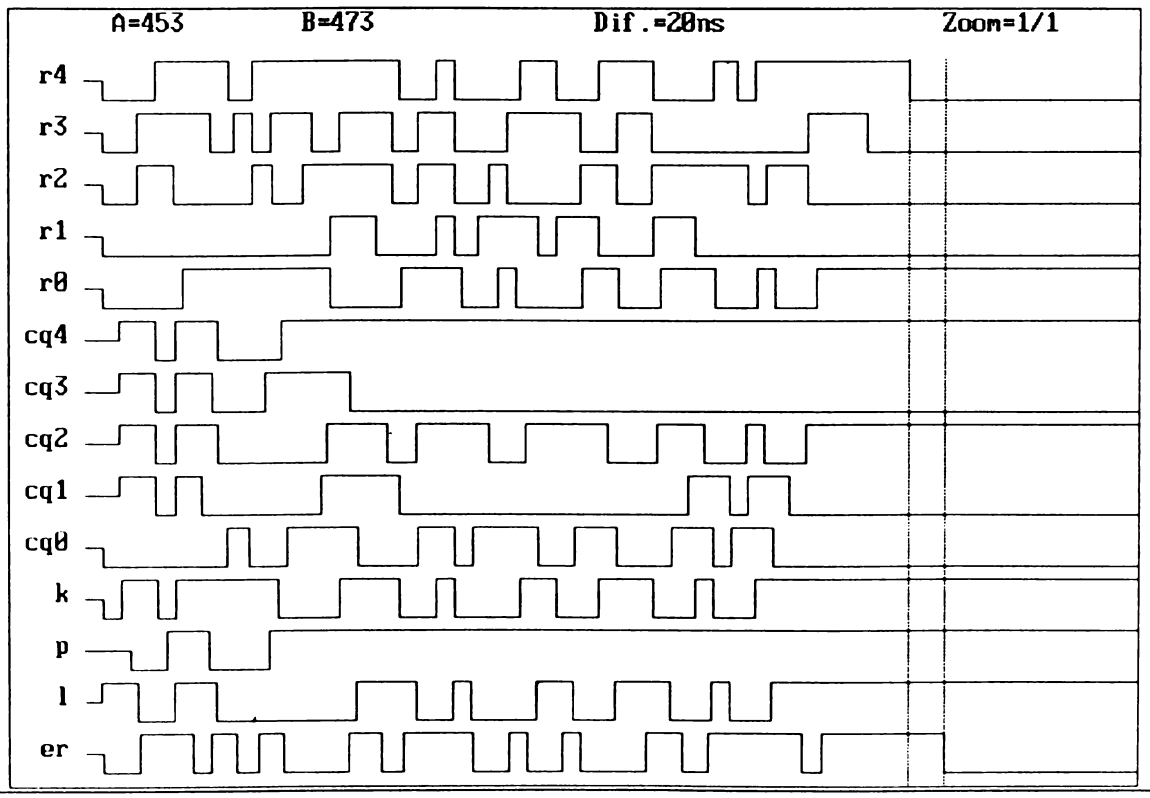


Figura 5.6 Caz de împărţire cu corecţia atât a câtului cât şi a restului.

Semnalele (**r4**, **r3**, **r2**, **r1**, **r0**) şi (**cq4**, **cq3**, **cq2**, **cq1**, **cq0**) vizualizate în imaginea captată în urma simulării sunt restul final şi respectiv câtul final al împărţirii. De asemenea, se pot vedea semnalele de comandă **k**, **p** şi **l** ale operaţiilor de corecţie, precum şi semnalul de eroare **er** care este corespondentul semnalului "error indicator" obţinut din semnalele de eroare "erA" şi "erB" descrise amănunţit în capitolul 4.

Figura 5.4 prezintă un caz la care în urma împărţirii este nevoie de corecţia restului prin adăugarea deîmpărţitului Y. În urma acestei corecţii câtul final, din întâmplare, este zero. În acest caz, se poate observa pe imaginea preluată din simulator, care este ordinea semnalelor prin prisma timpului de întârziere într-o situaţie defavorabilă din punct de vedere a timpului de propagare a întregii structuri. Astfel, semnalul de eroare **er** apare ultimul cu o întârziere suplimentară faţa de bitul cel mai semnificativ al restului **r4**, egală cu două niveluri de porţi SAU-EXCLUSIV. Înaintea semnalului **r4**, s-au stabilizat semnalele **r3**, **r2**, **r1** având cam aceeaşi diferenţă între ele.

Figura 5.5 prezintă un caz la care în urma efectuării operaţiei de împărţire este nevoie de corecţia câtului prin incrementarea sa.

În fine, figura 5.6 exemplifică un caz de împărţire la care trebuie corectate atât restul prin scăderea deîmpărţitului Y, cât şi câtul prin incrementarea sa.

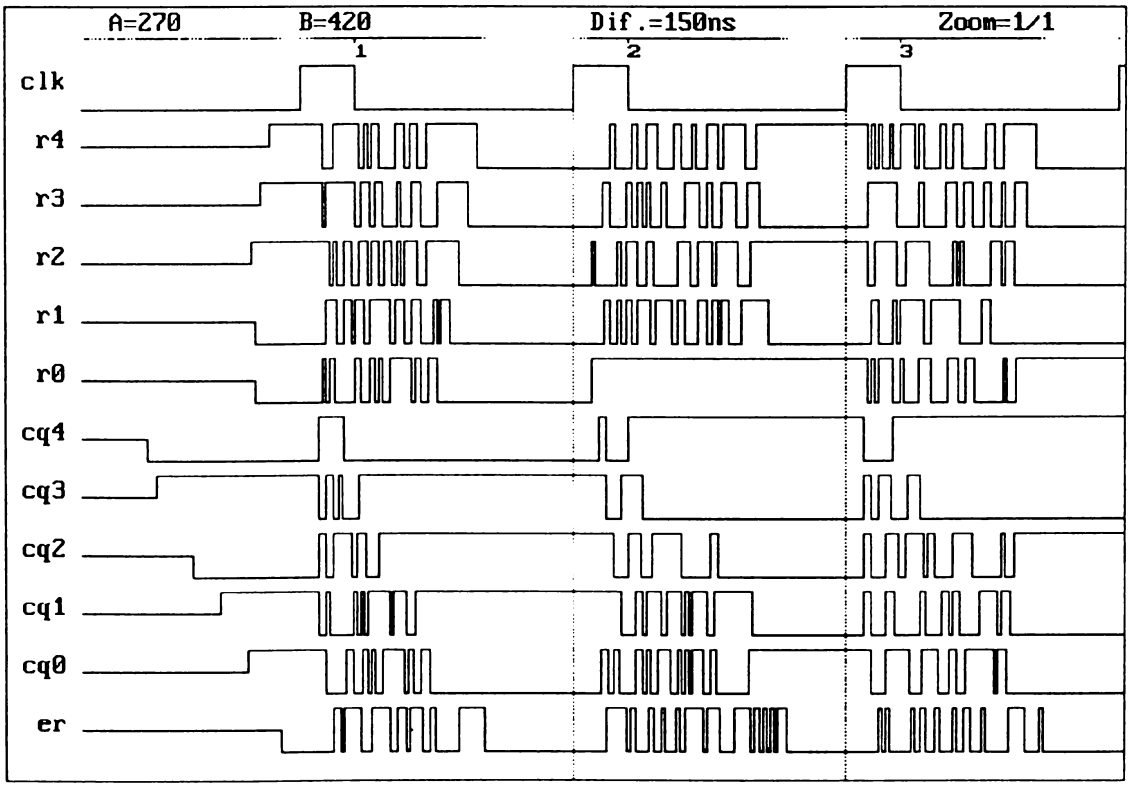


Figura 5.7 Secvenţa de patru împărţiri executate una după alta.

Pentru a reda o vedere de ansamblu legată de funcţionarea schemei de împărţire, am captat o imagine, cea din figura 5.7, în urma unei simulări la care modulul structurii de împărţire este alimentat succesiv cu patru perechi de operanzi (cei din figurile 5.3-5.6) la interval de 150nsec cât este perioada tactului **clk** de sincronizare. Deoarece utilizarea zoom-ului din simulator în mod obligatoriu diminuează exactitatea rezultatelor de simulare, pentru a afişa pe ecran un interval mai mare de timp am recurs la folosirea unor timpi de propagare ai porţilor logice de 4-5

ori mai mici faţa de cei utilizaţi anterior, şi anume: t_{XOR}=2ns, t_{INV}=2ns, t_{OR}=3ns, t_{AND}=2ns. Ca urmare, perioada tactului **clk** de sincronizare trebuie percepută ca o valoare de 5×150nsec=750nsec. Precum se observă din figura 5.7, acest interval de timp este suficient ca să se propage semnalele prin structura de împărţire.

5.2.2. Comparaţia rezultatelor experimentale cu cele teoretice

În subcapitolul 4.2.1.2. am evaluat timpii de propagare ai împărţitorului celular pentru diferite mărimi ale acestuia (variând lungimea *n* de operanzi, precum şi numărul de niveluri *m* ale structurii de împărţire). Evaluarea s-a făcut având în vedere schema neredundantă (bazată pe celule de tipul celei din figura 4.5), iar ulterior s-a ajuns la concluzia că arborele de paritate construit în conformitate cu indicaţiile date în subcapitolul 4.3 adaugă o întârziere suplimentară de doar două niveluri de porţi SAU-EXCLUSIV, indiferent de mărimea structurii de împărţire. Astfel, pentru modelul simulat mai sus, care reprezintă o structură de împărţire cu *m*=5 şi *n*=5 format din celule cu circuitele de transport duplicate (de tipul celei din figura 4.8) şi cu control de paritate, timpul maxim de propagare estimat este 801nsec (vezi tabelul 4.3) plus 2×10nsec, adică 821nsec. Timp comparabil cu cele 750 de nanosecunde constatate prin simulare.

Dacă am folosi o tehnologie mai rapidă cum ar fi ASIC CMOS [MOTO92], cu timp de propagare tipic pe poartă de circa 180 psec, evaluarea timpilor de propagare în cadrul structurii de împărţire neredundante, cu ajutorul formulelor prezentate în subcapitolul 4.2.1.2, ar duce la tabelul 5.4 care prezintă valori mult mai performante decât cele din tabelul 4.3. Pentru o structură cu autocontrol prin paritate având transportul duplicat se adaugă la timpul de propagare, prezentat în tabelul 5.4, valoarea de ≅0,5nsec echivalentă celor două niveluri de porţi SAU-EXCLUSIV în tehnologia CMOS.

Timpi de propagare	**m=5, n=5**	**m=8, n=8**	**m=6, n=10**	**m=16, n=10**	**m=16, n=16**	**m=32, n=32**	**m=64, n=64**	**m=128, n=128**
t_{cc}	0,9	1,4	1,0	2,8	2,8	5,7	11,5	23,0
t_{cr}	2,3	3,4	4,1	4,1	6,3	12,0	23,5	46,6
t'_p	10,8	25,9	23,7	63,3	97,9	380,1	1497,6	5944,3
t_p	**13,1**	**29,3**	**27,9**	**67,5**	**104,2**	**392,2**	**1521,1**	**5990,9**

Tabelul 5.4 Timpii de propagare (în nsec) pentru diferite mărimi ale schemei de împărţire celulară fără redundanţă, evaluaţi în baza unei posibile implementări în tehnologia CMOS.

Evaluarea timpilor de propagare făcută până acum s-a referit la structuri de împărţire celulară, fie fără redundanţă, fie cu celule cu circuitele de transport duplicate. Dar ce se întâmplă cu timpii de propagare în cazul utilizării celulelor cu rezultatul dependent de transport, care, aşa cum am văzut la sfârşitul capitolului 4, conferă structurii de împărţire aceeaşi capacitate de detecţie a erorilor, făcând uz de o redundanţă hardware de proporţie mai mică? Singura diferenţă, care apare în acest caz faţă de ceea ce este prezentat în subcapitolul 4.2.1.2., este valoarea timpului

necesar calculării unui bit de rest parțial notat cu t_r. Noua formă este: $t_r = t_{AND} + t_{OR} + t_{XOR}$. Ținând cont de această modificare s-au calculat valorile timpilor de propagare pentru diferite mărimi ale structurii de împărțire și sunt prezentați în tabelul 5.5. Se observă că, se obțin valori puțin mai mari față de varianta cu duplicarea transportului, dar această diferență nu justifică utilizarea ultimei variante deoarece ea provoacă o creștere a redundanței hardware cu circa 75% față de aproape 50% în cazul variantei bazate pe celule cu rezultatul dependent de transport.

Timpi de propagare	m=5, n=5	m=8, n=8	m=6, n=10	m=16, n=10	m=16, n=16	m=32, n=32	m=64, n=64	m=128, n=128
t_{cr}	2,5	3,6	4,3	4,3	6,4	12,2	23,7	46,8
t'_P	11,7	27,3	24,8	66,2	100,8	385,9	1509,1	5967,3
t_P	**14,2**	**30,9**	**29,1**	**70,5**	**107,2**	**398,1**	**1532,8**	**6014,1**

Tabelul 5.5 Timpii de propagare (în nsec) pentru diferite mărimi ale schemei de împărțire celulară propusă, realizată din celule S/S cu rezultat dependent de transport.

De asemenea, trebuie menționat că se adaugă la timpul de propagare, prezentat în tabelul 5.5, valoarea de ≅0,5nsec pentru evaluarea timpului total de propagare a schemei de împărțire propusă.

Analiza performanțelor de fiabilitate ale structurii de împărțire celulară propuse se va face în comparație cu soluția clasică a dublării întregului subansamblu. De asemenea, în figura 5.8 sunt prezentate spre comparare funcțiile de fiabilitate, atât a unei structurii neredundante, cât și a uneia cu dublarea circuitelor de transport. Pentru efectuarea calculelor s-a considerat structura de împărțire cu m=8 niveluri, și lungimea operanzilor de n=8. Calculul ratei de defectare pentru fiecare caz se va face în conformitate cu standardul MIL-HDBK-217B care estimează rata de defectare a unui circuit integrat cu ajutorul relației:

$$\lambda = \pi_L \cdot \pi_Q \cdot (C_1 \cdot \pi_T + C_2 \cdot \pi_E) \cdot \pi_P \quad \text{în defectări per un milion de ore,}$$

unde π_L este factorul de cunoaștere, π_Q este factorul de calitate, π_T este factorul de temperatură, π_E este un factor de mediu, π_P este un factor de pini, și C_1 și C_2 sunt factori de complexitate.

Factorul de cunoaștere π_L reprezintă maturitatea per total a procesului de fabricație și primește valori de la 1 la 10. Bineînțeles, cu cât procesul de fabricație este mai nou și mai puțin cunoscut cu atât factorul ia valori mai mari. În calculul nostru vom considera $\pi_L = 1$.

Factorul de calitate π_Q reprezintă cât de mult a fost testat un dispozitiv înainte de a fi vândut de către fabricant, și variază de la 1 la 300. Vom considera $\pi_Q = 16$.

Factorul de temperatură π_T este funcție de: tehnologia dispozitivului, temperatura de operare, tehnologia de împachetare, și puterea disipată. Ecuația specifică utilizată pentru acest factor în cazul circuitelor digitale bipolare este:

$$\pi_T = 0,1 \cdot e^{[-4794((1/(T_J+273))-(1/298))]}.$$

unde *Tj* este temperatura joncțiunii în grade Celsius. Dacă se consideră *Tj*=50°, atunci π_T=0,347343.

Factorul de mediu π_E este funcție de duritatea condițiilor în mediul unde operează dispozitivul, şi poate lua valori de la 0,2 până la 10. Pentru calculul nostru vom considera condiții de sală cu climatizare, astfel π_E=0,2.

Factorul de pini π_P este funcție de numărul de pini ai capsulei. În MIL-HDBK-217B acest factor variază de la 1,0 la 1,2 în cazul tehnologiei LSI odată cu creşterea numărului de pini de la 1 la peste 64. În mod normal LSI este definit pentru circuitele integrate (CI) care conțin de la 100 la 1000 de porți logice. În cazul unui astfel de CI, factorul de pini este 1,0 când CI-ul are până la 25 de pini, 1,1 când CI-ul are între 26 şi 64, şi 1,2 când CI-ul are peste 64 de pini. Deci, vom considera π_P=1,1 deoarece în cazul nostru CI-ul are 34 de pini inițial, iar după aplicarea controlului de paritate are 40.

Ultimii factori care se includ în calculul ratei de defectare sunt factorii de complexitate, C_1 şi C_2, care sunt funcție de numărul de: porți în cazul circuitelor logice, tranzistoare în cazul circuitelor liniare, şi biți în cazul memoriilor. Factorii de complexitate pentru un CI care are între 100 şi 1300 de porți sunt:

$$C_1 = 0,0187 \cdot e^{(0,00471)(Ng)} \qquad C_1 = 0,013 \cdot e^{(0,00423)(Ng)}$$

unde Ng este numărul de porți din circuitul integrat.
Astfel, pentru schema neredundantă care conține 547 de porți avem:

$$C_1 = 0,0187 \cdot e^{(0,00471)(547)} = 0,245892 \qquad C_1 = 0,013 \cdot e^{(0,00423)(547)} = 0,131467$$

şi prin urmare rata de defectare este:

$$\lambda = \pi_L \pi_Q \left(C_1 \pi_T + C_2 \pi_E\right) \cdot \pi_P =$$

$$= 1 \cdot 16 \cdot \left((0,245892) \cdot (0,347343) + (0,131467) \cdot (0,2)\right) \cdot 1,1 = 1,97$$

de defecțiuni per un milion de ore.

În acelaşi mod am obținut pentru schema cu autocontrol propusă, rata de defectare de $\lambda = 6,63$ defecte per un milion de ore. De asemenea, în cazul dublării doar a circuitelor de transport am obținut $\lambda = 13,4 \cdot 10^{-6}$ defecțiuni/oră, iar în cazul dublării întregului subansamblu am obținut $\lambda = 24,4 \cdot 10^{-6}$ defecțiuni/oră.

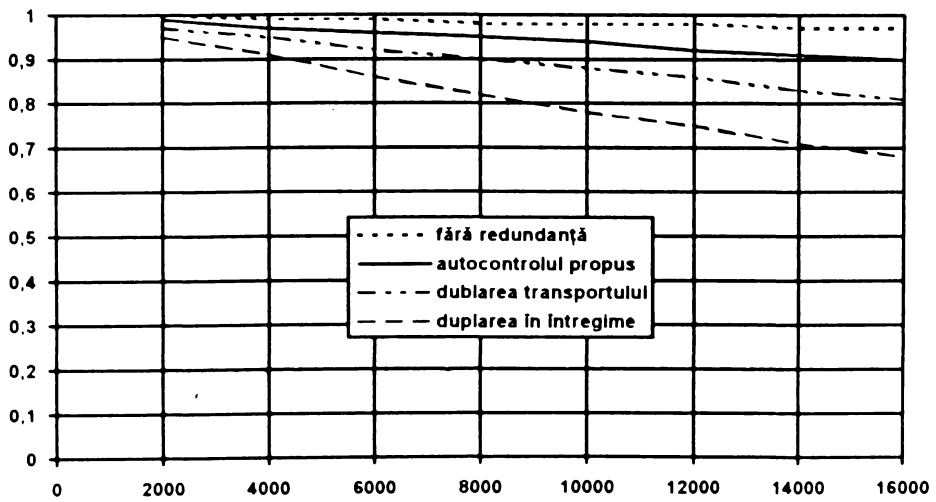


Figura 5.8 Funcțiile de fiabilitate ale structurii propuse de împărțire celulară.

Din figura 5.8 se observă performanțele superioare funcției de fiabilitate structurii de împărțire cu autocontrol propuse față de cea bazată pe dublarea întregii structurii. Diferența de datorează redundanței mai mici invocate de prima. Din același motiv, se observă superioritatea funcției de fiabilitate structurii neredundante, care în contrast cu celelalte trei soluții nu are capacitatea de detecție a erorilor, și astfel prezintă credibilitate mai redusă.

Creșterea fiabilității constă în faptul că am reușit să formăm reguli de proiectare pentru o structură de împărțire celulară cu autocontrol care îi conferă capacitate de detecție în timp real a tuturor erorilor de multiplicitate impară la un cost de redundanță hardware de sub 50%.

Rezolvarea problemei de verificare în timp real a operației de împărțire, care este cea mai stufoasă operație aritmetică dintre cele fundamentale, ne readuce la ideea de fond care este aplicarea parității ca metodă unică de autocontrol al subansamblelor hardware care efectuează operațiile aritmetice, logice și de transfer, obținând soluții optime prin prisma eficacității performanță/cost.

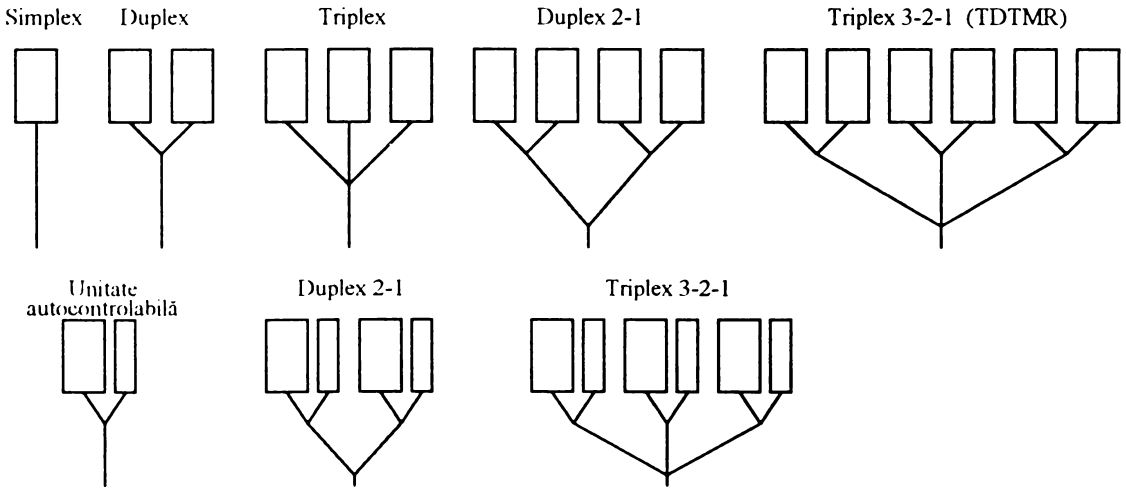


Figura 5.9 Arhitecturi redundante.

În figură 5.9 se prezintă diferite arhitecturi redundante atât în cazul replicării întregii unități neredundante, cât și în cazul replicării unității autocontrolabile, obținând sisteme duplex, triplex, triplex 3-2-1, etc. Se poate observa avantajul utilizării unităților autocontrolabile spre a obține sisteme duplex 2-1, triplex 3-2-1 în mod special atunci când redundanța din cadrul unității autocontrolabile este sub 50%.

5.3. Aplicațiile dispozitivelor de împărțire paralelă cu autocontrol

Unitățile aritmetice și logice dispuneau de obicei de mecanisme hardware pentru operațiile ca și adunarea, scăderea, înmulțirea, deplasări, rotiri, etc. Când era nevoie să se facă vreo operație mai complexă ca și împărțirea sau extracția de rădăcină pătrată se recurgea la metode software care dirijau hardware-ul existent. La ora actuală, datorită progreselor tehnologice, firmele producătoare de microprocesoare sau procesoare includ din ce în ce mai multe facilități hardware pentru a câștiga în viteză. Chiar și procesoarele RISC conțin subunități specializate în împărțirea de întregi pe 32 de biți, și există opțional coprocesoare aritmetice specializate în operații

în virgulă flotantă care de asemenea conțin subansamble specializate în operații mai complexe, care până acum câțiva ani din punct de vedere tehnologic nu erau posibile.

Cu toate aceste progrese, operația de împărțire necesită în continuare cel mai mult timp pentru a se efectua. Spre exemplu, un coprocesor Digital RISC specializat în operații cu virgulă flotantă necesită 12 cicluri pentru a executa o împărțire în simplă precizie, iar pentru precizie dublă are nevoie de 19 cicluri [DIGI91]. Spre comparație, tot același FPU necesită pentru înmulțire 4 cicluri în simplă precizie și 5 în dublă precizie, iar pentru restul operațiilor timpul de execuție este de la 1 ciclu până la maxim 3.

De asemenea, structurile de împărțire celulară ca cele propuse se pot folosi în cadrul procesoarelor fuzzy digitale în circuitul de "defazificare". Logica fuzzy este o ustensilă puternică care poate să urmărească aplicarea unor reguli exprimate într-o formă naturală. De exemplu, "dacă o mașină este rapidă, atunci ea este scumpă", unde "rapidă" și "scumpă" se numesc variabile lingvistice. Fiecare variabilă lingvistică deține un set asociat de termeni, care reprezintă setul de valori lingvistice pe care variabila le poate primi.

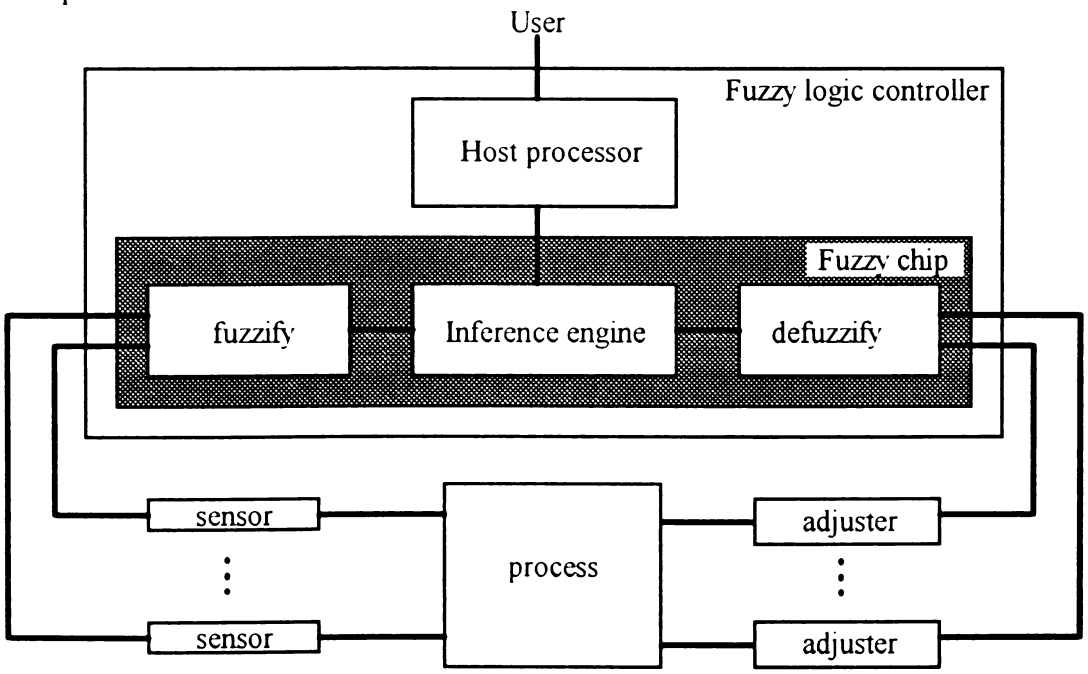


Figura 5.10 Principiul unui controlor cu logică fuzzy.

Atunci când se utilizează un set de reguli fuzzy în aplicații în timp real, procesarea regulilor prin software s-ar putea să se dovedească prea lentă. În astfel de situații este recomandată utilizarea implementării hardware a mașinilor de inferență fuzzy. De asemenea, în aplicații care pretează înaltă fiabilitate este nevoie de procesoare fuzzy cu capacități de tolerare a defectelor. Figura 5.10 prezintă un circuit integrat fuzzy dintr-un controlor fuzzy în buclă închisă al unei aplicații tipice în timp real. Ieșirile procesului sunt preluate și digitizate de către senzori și pe urmă aplicate controlorului fuzzy. Acestea sunt valori "crisp" care se fazifică în interiorul chip-ului fuzzy pentru a fi procesate de către mașina de inferențe. Rezultatul se defazifică și se aplică procesului prin intermediul unor activatoare.

Figura 5.11 ilustrează procesul de inferență pentru cazul în care sunt activete două reguli, $A_1 \rightarrow B_1$ și $A_2 \rightarrow B_2$. A' este evenimentul de intrare reprezentat ca o mulțime fuzzy

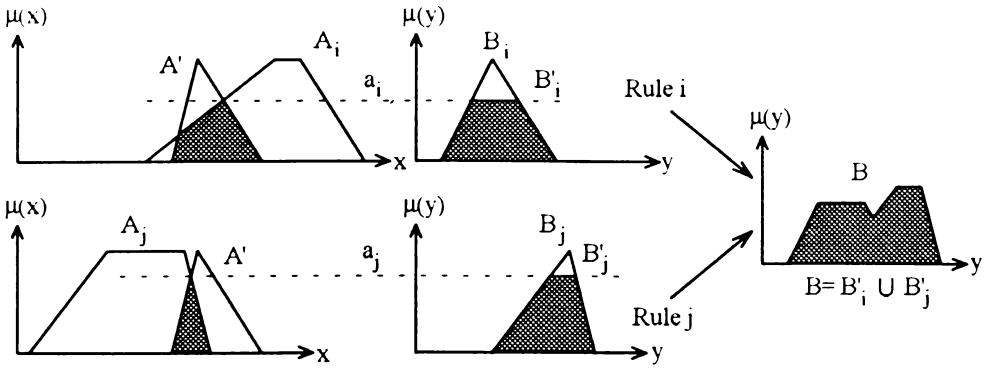


Figura 5.11 Mecanismul inferenței fuzzy.

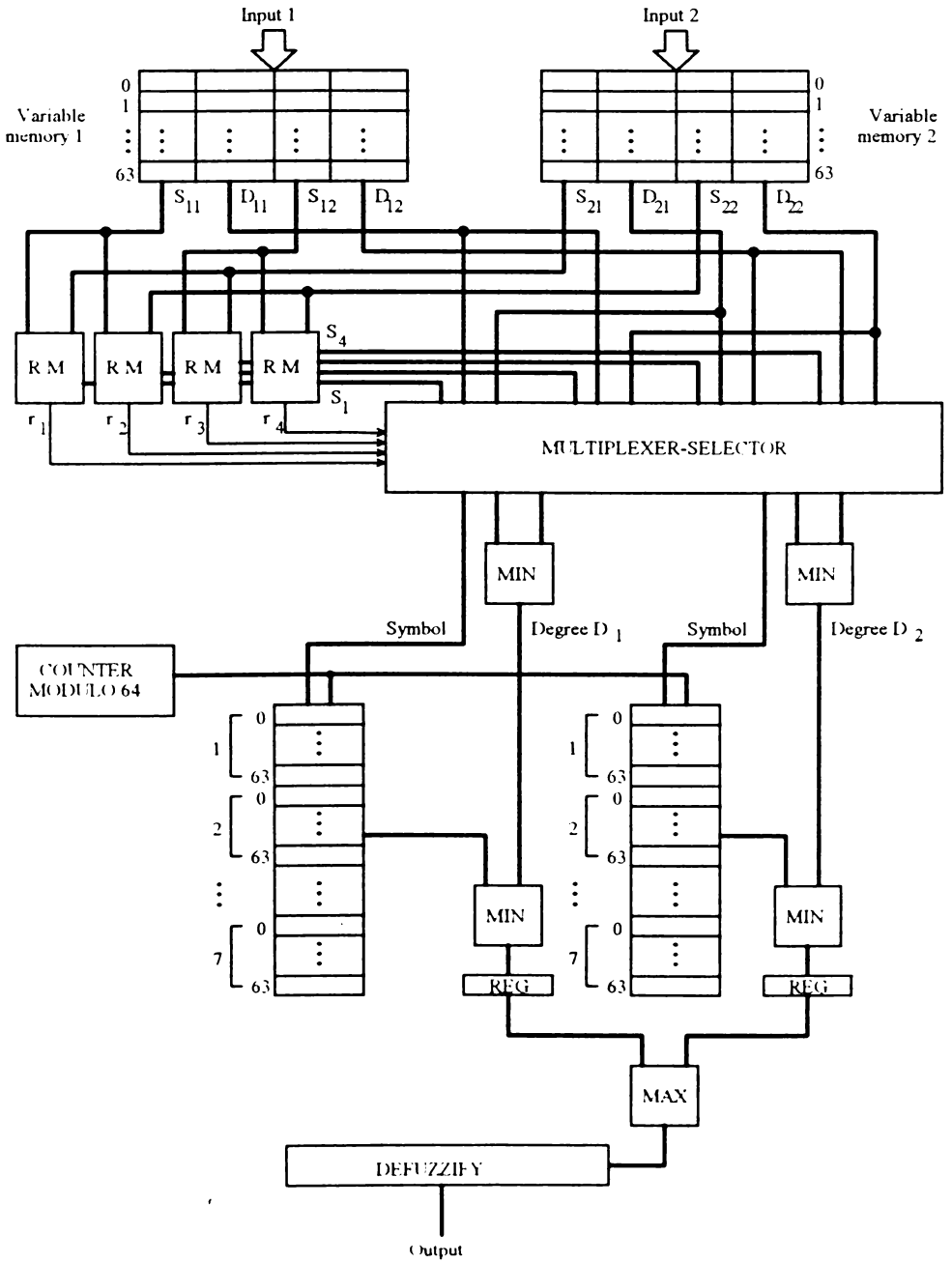


Figura 5.12 Arhitectura unei maşini de inferențe în logica fuzzy.

Figura 5.12 arată arhitectura unei maşini de inferenţe în logica fuzzy digitală care se pretează la aplicarea autocontrolului. În [VLAD95] sunt prezentate detaliile legate de grefarea controlului de paritate şi nu se va insista aici decât în ceea ce priveşte partea de defazificare. În această parte, rezultatul inferenţei se converteşte dintr-o valoare fuzzy în una "crisp", prin calcularea centrului de greutate. Formula pentru aflarea centrului de greutate este:

$$\frac{\sum_{i=1}^{64} i\cdot\mu(i)}{\sum_{i=1}^{64}\mu(i)} = \frac{1\cdot\mu(1)+2\cdot\mu(2)+3\cdot\mu(3)+\cdots+63\cdot\mu(63)+64\cdot\mu(64)}{\mu(1)+\mu(2)+\mu(3)+\cdots+\mu(63)+\mu(64)} =$$

$$= \frac{\mu(64)+(\mu(63)+\mu(64))+(\mu(62)+\mu(63)+\mu(64))+\cdots+(\mu(1)+\mu(2)+\mu(3)+\cdots+\mu(63)+\mu(64))}{\mu(1)+\mu(2)+\mu(3)+\cdots+\mu(63)+\mu(64)}$$

Relaţia de mai sus se traduce în circuitul din figura 5.13 care conţine două sumatoare ce efectuează câte o adunare la sosirea fiecărui eşantion din cele 64, şi o structură de împărţire paralelă.

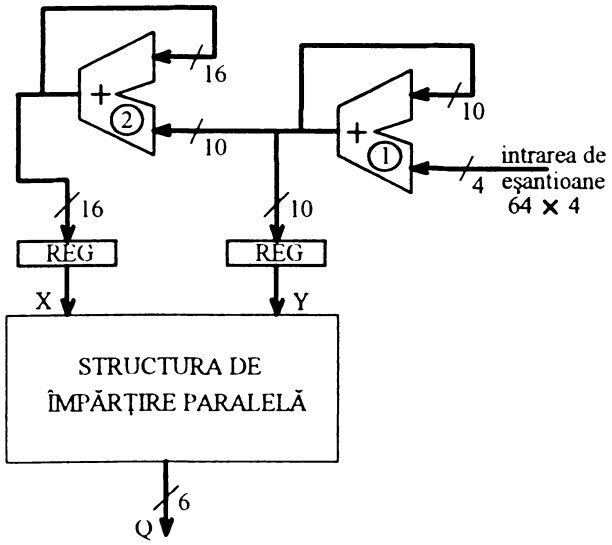


Figura 5.13 Detalierea părţii de defazificare a unei maşinii de inferenţe.

După ce au sosit toate cele 64 de eşantioane trebuie să se efectueze, în timp cât mai scurt, operaţia de împărţire având ca deîmpărţit conţinutul sumatorului 2, şi ca împărţitor conţinutul sumatorului 1. Acest lucru se poate realiza cu o structură de împărţire celulară asemănătoare celor prezentate în această lucrare cu autocontrol materializat prin paritate. În felul acesta, se asigură unicitatea controlului într-un procesor fuzzy [VLAD95] la un cost redus.

6. Concluzii

Lucrarea de față se încadrează în domeniul creşterii fiabilității şi disponibilității sistemelor de calcul pe seama exploatării metodelor de autocontrol în vederea detecției eventualelor erori. Se prezintă o soluție originală pentru realizarea unei structuri paralele de împărțire binară înzestrată cu control de paritate pentru detecția erorilor.

Ideea de fond care stă la bază este realizarea modulelor (procesoarelor, memoriilor, etc.) capabile să detecteze şi să semnaleze, cu influență minimă asupra costului, cel puțin acele erori dintre cele mai probabile să apară. În cadrul unui modul se abordează doar detecția datorită faptului că se pot folosi două, trei sau chiar mai multe astfel de module în funcție de cerințele dependabilității care se impun pentru o aplicație dată.

Justificarea alegerii operației de împărțire constă în faptul că, față de alte operații aritmetice fundamentale sau operații logice, ea este cea care necesită cel mai mult timp de execuție şi în plus este greu controlabilă. Astfel, dezvoltarea unei soluții valabile pentru verificarea în timp real a acestei operații se poate adapta uşor şi pentru alte operații mai puțin complexe, păstrându-se unicitatea controlului într-un sistem de calcul numeric.

Alegerea controlului de paritate s-a bazat pe următoarele motive:

- redundanță informațională minimă (un bit de control pentru n biți utili);
- redundanță hardware incomparabil mai mică față de duplicare;
- capacitatea de detecție a tuturor erorilor singulare, precum şi a celor de multiplicitate impară;
- reducerea minimă a performanțelor sistemului pe seama verificării on-line;
- constatarea că prin paritate se pot controla relativ uşor toate operațiile aritmetice şi logice, precum şi cele de transfer, asigurându-se unicitatea controlului într-un echipament numeric la un cost redus.

Rezultatele investigațiilor autorului au fost conform aşteptărilor, şi au confirmat buna orientare a direcției de aprofundare impusă de conducătorul său. Contribuțiile autorului se rezumă la următoarele:

1. Prezentarea sintetizată a noțiunilor şi a conceptelor fundamentale utilizate în domeniul fiabilității echipamentelor numerice de calcul.

2. Sistematizarea metodelor de bază care sporesc tolerarea defecțiunilor în sisteme de calcul, insistând asupra acelor metode care se bazează pe implementarea autocontrolului atât software, cât şi hardware.

3. Clasificarea într-o manieră originală a acelor metode de autocontrol implementate hardware, care asigură îmbunătățirea testabilității, în funcție de nivelul de structură dintr-un sistem la care se adresează aplicarea lor. Astfel, apar metode de proiectare pornind de la nivel de bistabil, la nivel de registru, la nivel de bloc funcțional ajungând până la nivel de procesor.

4 Analiza şi aprofundarea metodelor de proiectare a circuitelor de verificare cu proprietăți speciale cum ar fi testabilitatea şi autotestabilitatea, precum şi a circuitelor cu autoverificare totală. Se prezintă sintetic problematica de proiectare a unor checker-e incorporate, cu autocontrol, des utilizate în cadrul echipamentelor de calcul tolerante la defecțiuni.

5 Abordarea sintetică a problematicii operației de împărțire binară privită prin diferite prisme cum sunt performanța, acuratețea şi autocontrolul, evidențiindu-se principalele avantaje şi dezavantaje ale metodelor existente.

6. Definirea şi propunerea unui nou algoritm de împărţire binară bazat pe algoritmul fără refacerea restului, având ca elemente de noutate operarea cu numere binare cu semn şi furnizarea unor rezultate exacte (atât câtul, cât şi restul împărţirii).

7. Proiectarea unei structuri de împărţire paralelă bazată pe algoritmul propus în vederea exemplificării şi studierii posibilităţilor de autocontrol.

8. Referitor la grefarea controlului de paritate la împărţitorul paralel (celular) propus, investigaţiile autorului conduc la:

8.1. Deducerea într-un mod original a relaţiilor de control. Spre deosebire de cele prezentate în [PRAD86], unde nu există o formă generală iar deducerea este atât necesară la fiecare caz particular, cât şi laborioasă, ecuaţiile de paritate deduse de autor sunt valabile în general, fiind realizabile în mod facil particularizările pentru diferite numere de niveluri de structură, precum şi diferite lungimi ale operanzilor.

8.2. Elaborarea formalismului, bazat pe o lemă originală, care permite îmbunătăţirea, prin prisma capacităţii de detecţie a erorilor, a structurii propuse. Demonstraţia acestei leme se efectuează cu ajutorul unor reguli (observaţii), de asemenea sistematizate de autorul tezei şi prezentate în anexa A, legate de configuraţiile binare ale numerelor binare raportate la egalitatea sau inegalitatea valorilor absolute.

9. În scopul verificării atât a algoritmului de împărţire, cât şi a transpunerii acestuia într-o structură hardware împreună cu circuitele aferente de control prin paritate, autorul a întreprins următoarele acţiuni:

9.1. Elaborarea unui program numit DIVISION.EXE bazat pe descrierea formală a algoritmului propus care efectuează împărţiri afişând rezultatele într-un mod familiar şi echivalent cu metoda creion-hârtie.

9.2. Definirea unui limbaj în domeniul structural de descriere hardware la nivelul logic, precum şi elaborarea unui program de simulare numit COMP_SIM.EXE capabil să simuleze funcţionarea circuitelor numerice descrise în acest limbaj. Acest limbaj este utilizat pentru scopuri didactice în cadrul laboratoarelor Facultăţii de Calculatoare din Timişoara.

9.3. Modelarea structurii propuse în conformitate cu acest limbaj.

9.4. Verificarea corectei funcţionări a structurii propuse cu ajutorul simulării, inclusiv prin inserarea unor defecţiuni.

Concluziv, se poate afirma că soluţia propusă în lucrare, care pledează în favoarea abordării autocontrolului prin paritate a operaţiilor aritmetice şi logice, precum şi a celor de transfer, în vederea sporirii dependabilităţii sistemelor de calcul, duce la rezolvarea problemei autoverificării în timp real fără introducerea unui cost prohibitiv. Această soluţie se poate fructifica în cadrul proiectării modulare de sisteme numerice dependabile datorită unicităţii controlului.

Anexa A

OBSERVAȚII

Se consideră două numere binare X şi Y, fiecare cu n biți, reprezentate în complement de doi. Bitul cel mai semnificativ, cel cu indice n-1 reprezintă semnul.

$X = x_{n-1}\, x_{n-2}\, x_{n-3} \cdots x_1 x_0$

$Y = y_{n-1}\, y_{n-2}\, y_{n-3} \cdots y_1 y_0$

Se dau câteva reguli (observații) legate de configurațiile binare ale acestor numere raportate la egalitatea sau inegalitatea valorilor absolute.

1) Ca să fie $|X| = |Y|$ avem următoarele două cazuri:

1.1) Dacă $x_{n-1} = y_{n-1}$ atunci trebuie $x_i = y_i$ pentru $\forall\, i \in [0,\ n-2]$

1.2) Dacă $x_{n-1} \neq y_{n-1}$ atunci trebuie să existe un $j \in [0,\ n-2]$

astfel încât $x_i \neq y_i$ pentru $\forall\, i \in (j,\ n-2]$

şi $x_j = y_j = 1$

şi $x_i = y_i = 0$ pentru $\forall\, i \in [0,\ j)$

2) Ca să fie $|X| < |Y|$ avem următoarele patru cazuri:

2.1) Dacă $x_{n-1} = y_{n-1} = 0$ atunci trebuie să existe un $j \in [0,\ n-2]$

astfel încât $x_j = 0$ şi $y_j = 1$ iar $x_i = y_i$ pentru $\forall\, i \in (j,\ n-2]$

2.2) Dacă $x_{n-1} = y_{n-1} = 1$ atunci trebuie să existe un $j \in [0,\ n-2]$

astfel încât $x_j = 1$ şi $y_j = 0$ iar $x_i = y_i$ pentru $\forall\, i \in (j,\ n-2]$

2.3) Dacă $x_{n-1} = 0$, $y_{n-1} = 1$ atunci trebuie:

(a): $x_i \neq y_i$ pentru $\forall\, i \in [0,\ n-2]$

sau

(b): să existe un $j \in [0,\ n-2]$ astfel încât $x_j = y_j = 0$

şi $x_i \neq y_i$ pentru $\forall\, i \in (j,\ n-2]$

2.4) Dacă $x_{n-1} = 1$, $y_{n-1} = 0$ atunci trebuie să existe un $j \in [1,\ n-2]$

astfel încât $x_j = y_j = 1$ şi $x_i \neq y_i$ pentru $\forall\, i \in (j,\ n-2]$

şi să existe cel puțin un unu între $x_{j-1}\, x_{n-2} \cdots x_1 x_0$ sau $y_{j-1}\, y_{j-2} \cdots y_1 y_0$

3) Ca să fie $|X| \le |Y|$ avem următoarele patru cazuri:

3.1) Dacă $x_{n-1} = y_{n-1} = 0$ atunci trebuie:

(a): $x_i = y_i$ pentru $\forall\, i \in [0,\ n-2]$

sau

(b): să existe un $j \in [0,\ n-2]$

astfel încât $x_j = 0$ şi $y_j = 1$ iar $x_i = y_i$ pentru $\forall\, i \in (j,\ n-2]$

3.2) Dacă $x_{n-1} = y_{n-1} = 1$ atunci trebuie:

(a): $x_i = y_i$ pentru $\forall\, i \in [0,\ n-2]$

sau

(b): să existe un $j \in [0,\ n-2]$

astfel încât $x_j = 1$ şi $y_j = 0$ iar $x_i = y_i$ pentru $\forall\, i \in (j,\ n-2]$

3.3) Dacă $x_{n-1} = 0, \quad y_{n-1} = 1$ atunci trebuie:

(a): $x_i \ne y_i$ pentru $\forall\, i \in [0,\ n-2]$

sau

(b): să existe un $j \in [0,\ n-2]$ astfel încât $x_j = y_j = 1$

şi $x_i \ne y_i$ pentru $\forall\, i \in (j,\ n-2]$

şi $x_i = y_i = 0$ pentru $\forall\, i \in [0,\ j)$

sau

(c): să existe un $j \in [0,\ n-2]$ astfel încât $x_j = y_j = 0$

şi $x_i \ne y_i$ pentru $\forall\, i \in (j,\ n-2]$

3.4) Dacă $x_{n-1} = 1, \quad y_{n-1} = 0$ atunci trebuie să existe un $j \in [0,\ n-2]$

astfel încât $x_j = y_j = 1$ şi $x_i \ne y_i$ pentru $\forall\, i \in (j,\ n-2]$

Anexa B

MODELUL ÎMPĂRȚITORULUI BINAR UTILIZAT LA SIMULARE

```
inputs:  x8=1,x7=0,x6=1,x5=1,x4=0;
inputs:  x3=1,x2=1,x1=0,x0=1;
inputs:  y4=1,y3=0,y2=0,y1=1,y0=1;
inputs:  test=0,defect=0;
 clock :  clk=0,3;

% ********************** level 4 ********************** %
   xor(y4,x8)>q55;
   not(q55)>q5;

   xor(y4,q5)>aa44;
   xor(c44,x8)>ab44;
   xor(aa44,ab44)>r44;
   and(x8,c44)>ac44;
   and(x8,aa44)>ad44;
   and(aa44,c44)>ae44;
   or3(ac44,ad44,ae44)>c45;

   xor(y3,q5)>aa43;
   xor(c43,x7)>ab43;
   xor(aa43,ab43)>r43;
   and(x7,c43)>ac43;
   and(x7,aa43)>ad43;
   and(aa43,c43)>ae43;
   or3(ac43,ad43,ae43)>c44;

   xor(y2,q5)>aa42;
   xor(c42,x6)>ab42;
   xor(aa42,ab42)>r42;
   and(x6,c42)>ac42;
   and(x6,aa42)>ad42;
   and(aa42,c42)>ae42;
   or3(ac42,ad42,ae42)>c43;

   xor(y1,q5)>aa41;
   xor(c41,x5)>ab41;
   xor(aa41,ab41)>r41;
   and(x5,c41)>ac41;
   and(x5,aa41)>ad41;
   and(aa41,c41)>ae41;
   or3(ac41,ad41,ae41)>c42;

   xor(y0,q5)>aa40;
   xor(c40,x4)>ab40;
   xor(aa40,ab40)>r40;
   and(x4,c40)>ac40;
   and(x4,aa40)>ad40;
   and(aa40,c40)>ae40;
   or3(ac40,ad40,ae40)>c41;
   amp(q5)>c40;

% ********************** level 3 ********************** %
   xor(y4,r44)>q44;
```

```
not(q44)>q4;

xor(y4,q4)>aa34;
xor(c34,r43)>ab34;
xor(aa34,ab34)>r34;
and(r43,c34)>ac34;
and(r43,aa34)>ad34;
and(aa34,c34)>ae34;
or3(ac34,ad34,ae34)>c35;

xor(y3,q4)>aa33;
xor(c33,r42)>ab33;
xor(aa33,ab33)>r33;
and(r42,c33)>ac33;
and(r42,aa33)>ad33;
and(aa33,c33)>ae33;
or3(ac33,ad33,ae33)>c34;

xor(y2,q4)>aa32;
xor(c32,r41)>ab32;
xor(aa32,ab32)>r32;
and(r41,c32)>ac32;
and(r41,aa32)>ad32;
and(aa32,c32)>ae32;
or3(ac32,ad32,ae32)>c33;

xor(y1,q4)>aa31;
xor(c31,r40)>ab31;
xor(aa31,ab31)>r31;
and(r40,c31)>ac31;
and(r40,aa31)>ad31;
and(aa31,c31)>ae31;
or3(ac31,ad31,ae31)>c32;

xor(y0,q4)>aa30;
xor(c30,x3)>ab30;
xor(aa30,ab30)>r30;
and(x3,c30)>ac30;
and(x3,aa30)>ad30;
and(aa30,c30)>ae30;
or3(ac30,ad30,ae30)>c31;
amp(q4)>c30;

% ********************** level 2 ********************** %
xor(y4,r34)>q33;
not(q33)>q3;

xor(y4,q3)>aa24;
xor(c24,r33)>ab24;
xor(aa24,ab24)>r24;
and(r33,c24)>ac24;
and(r33,aa24)>ad24;
and(aa24,c24)>ae24,
or3(ac24,ad24,ae24)>c25;

xor(y3,q3)>aa23,
xor(c23,r32)>ab23;
xor(aa23,ab23)>r23,
```

```
and(r32,c23)>ac23;
and(r32,aa23)>ad23;
and(aa23,c23)>ae23;
or3(ac23,ad23,ae23)>c24;

xor(y2,q3)>aa22;
xor(c22,r31)>ab22;
xor(aa22,ab22)>r22;
and(r31,c22)>ac22;
and(r31,aa22)>ad22;
and(aa22,c22)>ae22;
or3(ac22,ad22,ae22)>c23;

xor(y1,q3)>aa21;
xor(c21,r30)>ab21;
xor(aa21,ab21)>r21;
and(r30,c21)>ac21;
and(r30,aa21)>ad21;
and(aa21,c21)>ae21;
or3(ac21,ad21,ae21)>c22;

xor(y0,q3)>aa20;
xor(c20,x2)>ab20;
xor(aa20,ab20)>r20;
and(x2,c20)>ac20;
and(x2,aa20)>ad20;
and(aa20,c20)>ae20;
or3(ac20,ad20,ae20)>c21;
amp(q3)>c20;

% ********************** level 1 ********************** %
xor(y4,r24)>q22;
not(q22)>q2;

xor(y4,q2)>aa14;
xor(c14,r23)>ab14;
xor(aa14,ab14)>r14;
and(r23,c14)>ac14;
and(r23,aa14)>ad14;
and(aa14,c14)>ae14;
or3(ac14,ad14,ae14)>c15;

xor(y3,q2)>aa13;
xor(c13,r22)>ab13;
xor(aa13,ab13)>r13;
and(r22,c13)>ac13;
and(r22,aa13)>ad13;
and(aa13,c13)>ae13;
or3(ac13,ad13,ae13)>c14;

xor(y2,q2)>aa12;
xor(c12,r21)>ab12;
xor(aa12,ab12)>r12;
and(r21,c12)>ac12;
and(r21,aa12)>ad12;
and(aa12,c12)>ae12;
or3(ac12,ad12,ae12)>c13;
```

```
xor(y1,q2)>aa11;
xor(c11,r20)>ab11;
xor(aa11,ab11)>r11;
and(r20,c11)>ac11;
and(r20,aa11)>ad11;
and(aa11,c11)>ae11;
or3(ac11,ad11,ae11)>c12;

xor(y0,q2)>aa10;
xor(c10,x1)>ab10;
xor(aa10,ab10)>r10;
and(x1,c10)>ac10;
and(x1,aa10)>ad10;
and(aa10,c10)>ae10;
or3(ac10,ad10,ae10)>c11;
amp(q2)>c10;

% ********************** level 0 ********************** %
xor(y4,r14)>q11;
not(q11)>q1;

xor(y4,q1)>aa04;
xor(c04,r13)>ab04;
xor(aa04,ab04)>r04;
and(r13,c04)>ac04;
and(r13,aa04)>ad04;
and(aa04,c04)>ae04;
or3(ac04,ad04,ae04)>c05;

xor(y3,q1)>aa03;
xor(c03,r12)>ab03;
xor(aa03,ab03)>r03;
and(r12,c03)>ac03;
and(r12,aa03)>ad03;
and(aa03,c03)>ae03;
or3(ac03,ad03,ae03)>c04;

xor(y2,q1)>aa02;
xor(c02,r11)>ab02;
xor(aa02,ab02)>r02;
and(r11,c02)>ac02;
and(r11,aa02)>ad02;
and(aa02,c02)>ae02;
or3(ac02,ad02,ae02)>c03;

xor(y1,q1)>aa01;
xor(c01,r10)>ab01;
xor(aa01,ab01)>r01;
and(r10,c01)>ac01;
and(r10,aa01)>ad01;
and(aa01,c01)>ae01;
or3(ac01,ad01,ae01)>c02;

xor(y0,q1)>aa00;
xor(c00,x0)>ab00;
xor(aa00,ab00)>r00;
and(x0,c00)>ac00;
and(x0,aa00)>ad00;
```

```
and(aa00,c00)>ae00;
or3(ac00,ad00,ae00)>c01;
amp(q1)>c00;

xor(y4,r04)>q00;
not(q00)>q0;

% ************************ Correction ********************** %

and(q4,q0)>l;
xor(q4,q00)>k;
amp(q4)>p;

% ********************** Quotient Correction ******************** %

and(p,q0)>qc1;
and(qc1,q1)>qc2;
and(qc2,q2)>qc3;
and(qc3,q3)>qc4;
xor(p,q0)>cq0;
xor(qc1,q1)>cq1;
xor(qc2,q2)>cq2;
xor(qc3,q3)>cq3;
xor(qc4,q4)>cq4;

% ********************** Remainder Correction ******************** %
and(k,y4)>yy4;
and(k,y3)>yy3;
and(k,y2)>yy2;
and(k,y1)>yy1;
and(k,y0)>yy0;

xor(yy4,l)>aa4;
xor(cc4,r04)>ab4;
xor(aa4,ab4)>r4;
and(r04,cc4)>ac4;
and(r04,aa4)>ad4;
and(aa4,cc4)>ae4;
or3(ac4,ad4,ae4)>cc5;

xor(yy3,l)>aa3;
xor(cc3,r03)>ab3;
xor(aa3,ab3)>r3;
and(r03,cc3)>ac3;
and(r03,aa3)>ad3;
and(aa3,cc3)>ae3;
or3(ac3,ad3,ae3)>cc4;

xor(yy2,l)>aa2;
xor(cc2,r02)>ab2;
xor(aa2,ab2)>r2;
and(r02,cc2)>ac2;
and(r02,aa2)>ad2;
and(aa2,cc2)>ae2;
or3(ac2,ad2,ae2)>cc3;

xor(yy1,l)>aa1;
xor(cc1,r01)>ab1;
```

```
xor(aa1,ab1)>r1;
and(r01,cc1)>ac1;
and(r01,aa1)>ad1;
and(aa1,cc1)>ae1;
or3(ac1,ad1,ae1)>cc2;

xor(yy0,1)>aa0;
xor(cc0,r00)>ab0;
xor(aa0,ab0)>r0;
and(r00,cc0)>ac0;
and(r00,aa0)>ad0;
and(aa0,cc0)>ae0;
or3(ac0,ad0,ae0)>cc1;
amp(1)>cc0;

% ********************** Parity Check ********************** %
xor(y0,y1)>xy01;
xor(y2,y3)>xy23;
xor(xy01,xy23)>xy03;
xor(xy03,y4)>xy04;              % Py %

xor(x0,x1)>xx01;
xor(x2,x3)>xx23;
xor(x4,x5)>xx45;
xor(x6,x7)>xx67;
xor(xx01,xx23)>xx03;
xor(xx45,xx67)>xx47;
xor(xx03,xx47)>xx07;
xor(xx07,x8)>xx08;              % Px %

xor(c41,c42)>xc412;
xor(c43,c44)>xc434;
xor(xc412,xc434)>xc414;
xor(xc414,c45)>xc415;           % C41-45 %

xor(c31,c32)>xc312;
xor(c33,c34)>xc334;
xor(xc312,xc334)>xc314;
xor(xc314,c35)>xc315;           % C31-35 %

xor(c21,c22)>xc212;
xor(c23,c24)>xc234;
xor(xc212,xc234)>xc214;
xor(xc214,c25)>xc215;           % C21-25 %

xor(c11,c12)>xc112;
xor(c13,c14)>xc134;
xor(xc112,xc134)>xc114;
xor(xc114,c15)>xc115;           % C11-15 %

xor(c01,c02)>xc012;
xor(c03,c04)>xc034;
xor(xc012,xc034)>xc014;
xor(xc014,c05)>xc015;           % C01-05 %

xor(xc415,xc315)>xc4315;
xor(xc215,xc115)>xc2115;
xor(xc4315,xc2115)>xc4115;
```

```
xor(xc4115,xc015)>xc4015;        % Sume Cij %

xor(r44,r34)>xr87;
xor(r24,r14)>xr65;
xor(xr87,xr65)>xr85;             % r8-5  ( sau altfel r44-14 ) %

not(k)>negk;
and(negk,xy04)>nkxy;
xor(test,y4)>ty4;                % test line %
xor(ty4,xx08)>ty4xx08;
xor(nkxy,ty4xx08)>xo;

xor(xr85,xc4015)>xrc;
xor(xo,xrc)>xoxrc;               % * xoxrc %

xor(qc4,qc3)>xqc43;
xor(qc2,qc1)>xqc21;
xor(xqc43,xqc21)>xqc41;
xor(xqc41,p)>xqc40;              % qc4-0   ( qc0 = p ) %

xor(cq4,cq3)>xcq43;
xor(cq2,cq1)>xcq21;
xor(xcq43,xcq21)>xcq41;
xor(xcq41,cq0)>xcq40;            % cq4-0   ( cqi este q'i )  %

xor(xqc40,xcq40)>xcqc;
xor(xoxrc,xcqc)>erA;             % erA %

xor(cc1,cc2)>xcc12;
xor(cc3,cc4)>xcc34;
xor(xcc12,xcc34)>xcc14;          % Cc1-c4 %

xor(r0,r1)>xr01;
xor(r2,r3)>xr23;
xor(xr01,xr23)>xr03;
xor(xr03,r4)>xr04;               % r0-4 %

xor(xcc14,xr04)>erB;             % erB %
xor(erA,erB)>er;                 % Error Indicator %

outputs: r4,r3,r2,r1,r0,cq4,cq3,cq2,cq1,cq0,k,p,l,er.
```

Bibliografie

[AHMA90] A.Ahmad, N.K.Nanda, K.Garg, "*Are Primitive Polynomials always best in Signature Analysis?*", IEEE Design & Test of Computers, August 1990, pag.36-38.

[AKER89] S.B.Akers, B.Kreshnamurthy, "*Test Counting: A Tool for VLSI Testing*", IEEE Design & Test of Computers, October 1989, pag.58-77.

[ATKI68] D.Atkins, "*Higher-Radix Division Using Estimates of the Divisor and Partial Remainders*", IEEE Transactions on Computers, October 1968, vol.C-17, pag. 925-934.

[AVIZ71] A.Avižienis, G.C.Gilley, F.P.Mathur, D.A.Rennels, J.A.Rohr, D.K.Rubin, "*The STAR (Self-Testing and Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design*", IEEE Transactions on Computers, November 1971, pag. 1312-1321.

[AVIZ85] A.Avižienis, "*The N-Version Approach to Fault-Tolerant Software*", IEEE Transactions on Software Engineering, December 1985, pag. 1491-1501.

[AVIZ87] A.Avižienis, "*On the Achievement of a Highly Dependable and Fault-Tolerant Air Traffic Control System*", 1987.

[BALA90] M.Balakrishnan, C.Raghavendra, "*On Reliability Modeling of Closed Fault-Tolerant Computer Systems*", IEEE Transactions. on Computers, April 1990, vol. 39, pag. 571-575.

[BALA93] M.Balakrishnan, C.Raghavendra, "*An Analysis of a Reliability Model for Repairable Fault-Tolerant Systems*", IEEE Transactions. on Computers, March 1993, vol. 42, pag. 327-338.

[BARB92] G.Barbu, "*Modele de simulare cu aplicații în fiabilitate*", Editura Tehnică, Bucureşti, 1992.

[BASS94] S.A.Bassam, B.Bose, "*Asymmetric/Unidirectional Error Correcting and Detecting Codes*", IEEE Transactions on Computers, May 1994, vol. 43, pag. 590-597.

[BHAT89] D.Bhattacharya, B.T.Murray, J.P.Hayes, "*High-Level Test Generation for VLSI*", IEEE Computer, April 1989, pag. 16-24.

[BLAK91] T.Blakeslee, "*Proiectarea cu circuite logice MSI şi LSI standard*", Editura Tehnică, Bucureşti, 1988.

[BLAU89] M.Blaum, H.V.Tilborg, "*On t-Error Correcting All Unidirectional Error Detecting Codes*", IEEE Transactions on Computers, November 1989, vol. 38, pag. 1493-1501.

[BLEE91] H.Bleeker, "*Boundary-Scan wird erschwinglich*", Electronik, November 1991, pag. 114-119.

[BOSE91] B.Bose, "*On Unordered Codes*", IEEE Transactions on Computers, February 1991, vol. 40, pag. 125-131.

[BREU76] M.A.Breuer, A.Friedman, "*Diagnosis and Reliable Design of Critical Systems*", Computer Science Press Inc, 1976.

[BRYA93] M.J.Bryan, S.Devadas, K.Keutzer, "*Analysis and Design of Regular Structures for Robust Dynamic Fault Testability*", VLSI Design, 1993, vol. 1, No.1, pag. 45-60.

[BURN94] S.W.Burns, N.K.Jha, "*A Totally Self-Checking Checker for a Parallel Unordered Coding Scheme*", IEEE Transactions. on Computers, April 1994, vol. 43, pag. 490-495.

[CAMP90] R.Camposano, "*From behavior to structure: high-level synthesis*", IEEE Design & Test of Computers, October 1990, pag.8-19.

[CAMP91] R.Camposano, L.Saunders, R.Tabet, "*VHDL as Input for High-Level Synthesis*", IEEE Design & Test of Computers, March 1991, pag.43-49.

[CART90] T.M.Carter, J.E.Robertson, "*Radix-16 Signed-Digit Division*", IEEE Transactions on Computers, December 1990, vol. 39, pag. 1424-1433.

[CATU85] V.Cătuneanu, I.Bacivarof, "*Fiabilitatea sistemelor de telecomunicații*", Editura Militară, Bucureşti, 1985.

[CATU89] V.Cătuneanu, A.Bacivarof, "*Structuri electronice de înaltă fiabilitate*", Editura Militară, Bucureşti, 1989.

[CHEN90] K.Cheng, V.D.Agrawal, E.S.Kuh, "*A Simulation-Based Method for Generating Tests for Sequential Circuits*", IEEE Transactions on Computers, December 1990, vol. 39, pag. 1456-1463.

[CHEN91] L.G.Chen, T.H.Chen, "*Fault-tolerant serial-parallel multiplier*", IEE PROCEEDINGS-E, July 1991, vol. 138, No.4, pag. 276-280.

[COEH90] D.Coehlo, "*Follow simple rules to create VHDL models*", ELECTRONIC DESIGN, June 1990, pag. 65-74.

[CORE90] I.Coren, A.Singh, "*Fault-Tolerance in VLSI Circuits*", IEEE Computer, July 1990, pag. 73-82.

[CRIŞ93] M.Crişan, "*Controlul operațiilor de procesare în sistemele numerice complexe de măsurare*", teză de doctorat, Timişoara, 1993.

[D&T89] D&T ROUNDTABLE, "*Built-in self-test: Are expectations too high?*", IEEE Design & Test of Computers, June 1989, pag. 66-74.

[DAS90] C.R.Das, J.T.Kreulen, M.J.Thazhuthaveetil, "*Dependability Modeling for Multiprocessors*", IEEE Computer, October 1990, pag. 7-19.

[DAVI86] A.David, "*Signature analysis for multiple input circuits*", IEEE Transactions on Computers, September 1986, vol. 35, pag. 830-836.

[DERV89] B.Dervisoglu, "*Scan-path architecture for Pseudorandom Testing*", IEEE Design & Test of Computers, August 1989, pag.32-48.

[DIGI91] d i g i t a l, "*Digital's RISC Architecture Technical Handbook*", Digital Equipment Corporation, 1991.

[DONN91] J.Donnel, "*Boundary-scan puts tomorrow's devices to test*", ELECTRONIC DESIGN, June 1991, pag. 75-86.

[ERCE87] M.D.Ercegovac, T.Lang, "*On-the-Fly Conversion of Redundant into Conventional Representations*", IEEE Transactions on Computers, July 1987, vol.C-36, pag. 895-897.

[GAIT83a] N.Gaitanis, C.Halatsis, "*Near-Perfect Codes for Binary-Coded Radix-r Arithmetic Units*", IEEE Transactions on Computers, May 1983, vol.C-32, pag. 494-497.

[GAIT83b] N.Gaitanis, "*Totally Self-Checking Checker for 3N Arithmetic Codes*", ELECTRONICS LETTERS, 18th August 1983, vol. 19, No.17, pag. 685-686.

[GAIT84] N.Gaitanis, "*Single Error Correcting and Multiple Unidirectional Error Detecting Cyclic AN Arithmetic Codes*", ELECTRONICS LETTERS, 19th July 1984, vol. 20, No.15, pag. 638-640.

[GAIT85] N.Gaitanis, "*A Totally Self-Checking Error Indicator*", IEEE Transactions on Computers, August 1985, vol.C-34, pag. 758-761.

[GAIT88a] N.Gaitanis, "*TSC Error CD Circuits for SEC/DED Product Codes*", IEE PROCEEDINGS, September 1988, vol. 135, pag. 253-258.

[GAIT88b] N.Gaitanis, "*The Design of TSC Error C/D Circuits for SEC/DED Codes*", IEEE Transactions on Computers, March 1988, vol. 37, pag. 258-265.

[GAIT88c] N.Gaitanis, "*Totally Self-Checking Checkers with Separate Internal Fault Indication*", IEEE Transactions on Computers, October 1988, vol. 37, pag. 1206-1213.

[GEBE84] T.Geber, E.Stăicuț, I.Tutoveanu, M.Grigorescu, M.Bălan, I.Popa, "*Fiabilitatea și mentenabilitatea sistemelor de calcul*", Editura Tehnică, București, 1984.

[GEIS90] R.Geist, K.Triveti, "*Reliability Estimation of Fault-Tolerant Systems: Tools and Techniques*", IEEE Computer, July 1990, pag. 52-61.

[GORK91] W.Görke, "*Digitale Fehlerdiagnose*", Institut für Rechnerentwurf und Fehlertoleranz: Universität Karlsruhe, Vorlesungsskript SS 1991.

[FING85] A.Finger, "*Digitale Signalstructuren in der Informationstechnik*", Berlin: Verlag, 1985.

[FOWK93] R.Fowkes, "*Hardware Efficient Algorithms for Trigonometric Functions*", IEEE Transactions on Computers, February 1993, vol. 42, pag. 235-239.

[FRIE86] A.D.Friedman, "*Fundamentals of Logic Design and Switching Theory*", Computer Science Press, 1986.

[FUJI90a] H.Fujiwara, "*Logic Testing and Design for Testability*", England: MIT Press, 1990.

[FUJI90b] E.Fujiwara, K.Pradhan, "*Error-Control Coding in Computers*", IEEE Computer, July 1990, pag. 63-71.

[FUNA89] S.Funatsu, "*Scan design at NEC*", IEEE Design & Test of Computers, June 1989, pag. 50-57.

[FURH94] B.Furht, "*Parallel Computing: Glory and Collapse*", IEEE Computer, November 1994, pag. 74-75.

[HALL89] J.J.Hallenbeck, J.R.Cyprynski, N.Kanopoulos, T.Makras, N.Vasanthavada, "*The Test Engineer's Assistant. A Support Environment for Hardware Design for Testability*", IEEE Computer, April 1989, pag. 59-68.

[HAYE85] J.P.Hayes, "*Fault Modeling*", IEEE Design & Test of Computers, April 1985, pag.88-95.

[HAYE88] J.Hayes, "*Computer Architecture and Organization*", McGraw-Hill International Editions, Computer Science Series, 1988.

[HECH79] H.Hecht, "*Fault-Tolerant Software*", IEEE Transactions on Reliability, August 1979, pag. 227-232.

[HOPK78] A.L.Hopkins, Jr., T.B.Smithm III, J.H.Lala, "*FTMP--A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft*", PROCEEDINGS OF THE IEEE, October 1978, pag. 1221-1239.

[HORT90] P.D.Hortensius, R.D.McLeod, H.C.Card, "*Cellular Automata-Bases Signature Analysis for Built-in Self-test*", IEEE Transactions on Computers, October 1990, vol. 39, pag. 1273-1283.

[HU87] M.Hu, H.T.Mouftah, "*Fault-Tolerant System Using 3-Value Logic Circuits*", IEEE Transactions on Reliability, June 1987, pag. 227-231.

[IONE81] D Ionescu, "*Codificare și coduri*", Editura Tehnică București, 1981.

[JOHN89] B.W Johnson, "*Design and Analysis of Fault Tolerant Digital Systems*", Addison-Wesley Publishing Company, 1989.

[JOHN93] B Johnson, "*Boundary Scan Eases Test of New Technologies*", TEST & MEASUREMENT EUROPE, autumn 1993

[JONE94] W.B.Jone, C.J.Wu, "*Multiple Fault Detection in Parity Checkers*", IEEE Transactions on Computers, September 1994, vol. 43, pag. 1096-1099.

[KANO89] N.Kanopoulos, N.Vasanthavada, "*A built-in test module for Fault Isolation*", IEEE Design &,Test of Computers, June 1989, pag. 58-65.

[KROL86] T.Krol, "*(N,K) Concept Fault Tolerance*", IEEE Transactions on Computers, April 1986, vol.C-35, No.4, pag. 339-349.

[KUHL86] J.G.Kuhl, M.Reddy, "*Fault-Tolerant Considerations in Large, Multiple Processor Systems*", IEEE Computer, March 1986, pag. 56-67.

[KUND90] S.Kundu, S.Reddy, "*Embedded totally self-checking checkers: a practical design*", IEEE Design & Test of Computers, August 1990, pag. 5-12.

[LALA85] P.K.Lala, "*Fault Tolerant and Fault Testable Hardware Design*", Prentice-Hall International, London, 1985.

[LALA91] J.H.Lala, R.E.Harper, L.S.Alger, "*A Design Approach for Ultrareliable Real-time Systems*", IEEE Computer, May 1991, pag. 12-22.

[LAPR90] J.C.Laprie, C.Beounes, J.Arlat, K.Kanoun, "*Definition and Analysis of Hardware and Software Fault Tolerant Architectures*", IEEE Computer, July 1990, pag. 39-51.

[LEVE90] R.Leveugle, G.Saucier, "*Optimized Synthesis of Concurrently Checked Controllers*", IEEE Transactions on Computers, April 1990 vol. 39, pag. 414-425.

[LI91] Hungwen Li, Q.F.Stout, "*Reconfigurable SIMD Massively Parallel Computers*", PROCEEDINGS OF THE IEEE, April 1991, vol. 79, pag. 429-443.

[LIPS90] R.Lipset, C.Schaefer, C.Ussery, "*VHDL: Hardware description and design*", Kluwer academic publishers, 1990.

[LO93] J.C.Lo, S.Thanawastien, T.R.N.Rao, "*Berger Check Prediction for Array Multipliers and Array Dividers*", IEEE Transactions on Computers, July 1993, vol. 42, pag. 892-896.

[LO94] J.C.Lo, "*Reliable Floating-Point Arithmetic Algorithms for Error-Coded Operands*", IEEE Transactions on Computers, April 1994, vol. 43, pag. 400-412.

[MAHM88] A. Mahmood, E.J.McCluskey, "*Concurrent Error Detection Using Watchdog Processors - A Survey*", IEEE Transactions on Computers, February 1988, vol. 37, pag. 160-174.

[MAND90] D.M.Mandelbaum, "*A Systematic Method for Division with High Average Bit Skipping*", IEEE Transactions on Computers, January 1990, vol. 39, pag. 127-130.

[MANO93] M.Mano, "*Computer System Architecture*", Prentice-Hall International Editions, 1993.

[MARC93] R.Marculescu, "*Tehnici de bază în proiectarea pentru testabilitate*", Tempus postgraduate school of computer aided electrical engineering, Editor Daniel IOAN, Bucureşti, 1993.

[MAXW88] P.C.Maxwell, "*Comparative analysis of different implementations of multiple input signature analyzers*", IEEE Transactions on Computers, November 1988, vol. 37, pag. 1411-1414.

[MCCL90] E.McCluskey, "*Design Techniques for Testable Embedded Error Checkers*", IEEE Computer, July 1990, pag. 84-88

[MOLY89] G.Molyneaux, "*Components on Ternary-Scan Design for VLSI testability*", IEEE Transactions on Computers, February 1989, vol. 38, pag. 256-268.

[MONT93] P.Montuschi, L.Ciminiera, "*Reducing Iteration Time When Result Digit is Zero for Radix 2 SRT Division and Square Root with Redundant Remainders*", IEEE Transactions on Computers, February 1993, vol. 42, pag. 239-246.

[MONT94] P.Montuschi, L.Ciminiera, "*Over-Redundant Digit Sets and the Design of Digit-by-Digit Division Units*", IEEE Transactions on Computers, vol 43, No.3 ,March 1994, pag. 269-278.

[MOTO92] MOTOROLA Inc., "*MOTOROLA Semiconductor, Master Selection Guide*", 1992.

[NAGV91] P.Nagvajara, M.G.Karpovsky, L.B.Levitin, "*Pseudorandom Testing for Boundary-Scan Design with Built-In Self-Test*", IEEE Design & Test of Computers, September 1991, pag. 58-65.

[NANY87a] T.Nanya, T.Kawamura, "*A Note on Strongly Fault-Secure Sequential Circuits*", IEEE Transactions on Computers, September 1987, vol.C-36, pag. 1121-1123.

[NANY87b] T.Nanya, T.Kawamura, "*On Error Indication for Totally Self-Checking Systems*", IEEE Transactions on Computers, November 1987, vol.C-36, pag. 1389-1392.

[NANY88] T.Nanya, T.Kawamura, "*Error Secure/Propagating Concept and its Application to the Design of Strongly Fault-Secure Processors*", IEEE Transactions on Computers, January 1988, vol. 37, pag. 14-24.

[NELS87] V.P.Nelson, B.D.Carroll "*Tutorial: Fault-Tolerant Computing*", IEEE Computer Society Press, 1987.

[NELS90] V.Nelson, "*Fault-Tolerant Computing: Fundamental Concepts*", IEEE Computer, July 1990, pag. 19-25.

[NIKO91] D.Nikolos, "Theory and Design of t-Error Correcting/d-Error Detecting (d>t) and All Unidirectional Error Detecting Codes", IEEE Transactions on Computers, February 1991, vol. 40, pag. 132-142.

[NIKO88] D.Nikolos, A.M.Paschalis, G.Philokyprou, "*Efficient Design of Totally Self-Checking Checkers for all Low-Cost Arithmetic Codes*", IEEE Transactions on Computers, July 1988, vol. 37, pag. 807-814.

[NURI91] G.Nurie, "*Attain testability with hierarchical design*", ELECTRONIC DESIGN, June 1991, pag. 89-99.

[PARH87] B.Parhami, "*On the Complexity of Table Lookup for Iterative Division*", IEEE Transactions on Computers, October 1987, vol.C-36, pag. 1233-1236.

[PARK89] K.Parker, "*The Impact of Boundary Scan on Board Test*", IEEE Design & Test of Computers, August 1989, pag.18-29.

[PASC88] A.M.Paschalis, D.Nikolos, C.Halatsis, "*Efficient Modular Design of TSC Checkers for M-out-of-2M Codes*", IEEE Transactions on Computers, March 1988, vol. 37, pag. 301-309.

[PATE83] J.H.Patel, L.Y.Fung, "*Concurrent Error Detection in Multiply and Divide Arrays*", IEEE Transactions on Computers, April 1983, vol.C-32, pag. 417-422.

[PATT90] D.A.Patterson, J.L.Hennessy, "*Computer Architecture: A Quantitative Approach*", 1990 by Morgan Kaufmann Publishers, Inc, anexa scrisă de David Goldberg, pag. A1-A66

[PETR91a] N.S.Petrakis, "*Asupra stadiului actual al implementării metodelor de autocontrol*", referat de doctorat, 1991.

[PETR91b] N.S.Petrakis, "*Facilitarea autocontrolului prin metode de proiectare structurată*", referat de doctorat, 1991.

[PETR92] N.S.Petrakis, "*Implementarea autocontrolului la sisteme tolerante la defectare*", referat de doctorat, 1992.

[PETR94] N.S.Petrakis, "*Applying Parity Check to an Adapted Combinational Array Divider*", International Conference on Technical Informatics, PROCEEDINGS Vol. 5, Timişoara 1994, pag. 37-48.

[PETR95] N.S.Petrakis, "*About Synthesis on Parity Check Adder/Subtractor with Carry-Dependent Result for an Adapted Combinational Array Divider*", Buletinul Ştiinţific U.T. Timişoara, 1994, Tom 39 (53), Seria Automatizări şi Calculatoare (în curs de publicare).

[PRAD86] D.K.Pradhan, "*Fault-Tolerant Computing: Theory and Techniques*", Volumes I &' II, Prentice-Hall, 1986.

[PURD87] C.N.Purdy, G.B.Purdy, "*Integer Division in Linear Time with Bounded Fan-In*", IEEE Transactions on Computers, May 1987, vol.C-36, pag. 640-644.

[RAO89] T.R.N.Rao, E.Fujiwara, "*Error-Control Coding for Computer Systems*", Prentice-Hall, 1989.

[RENN84] D.A.Rennels, "*Fault-Tolerant Computing-Concepts and Examples*", IEEE Transactions on Computers, December 1984, pag. 1116-1129.

[RETT90] R.Rettberg, P.Carvey, "*The Monarch Parallel Processor Hardware Design*", IEEE Computer, April 1990, pag. 18-30.

[RUSS85] G.Russel, D.Kiniment, E.Chester, M.McLauchlan, "*CAD for VLSI*", Van Nostrand Reinhold (UK) Co. Ltd., 1985.

[SAVI93] J.Savir, P.H.Bardell, "*Built-In Self-Test: Milestones and Challenges*", VLSI design, 1993, vol. 1, No.1, pag. 23-45.

[SAXE90] N.R.Saxena, E.J.McCluskey, "*Control-Flow Checking Using Watchdog Assists and Extended-Precision Checksums*", IEEE Transactions on Computers, April 1990, vol. 39, pag. 554-559.

[SCHW93] E.M.Schwarz, M.J.Flynn, "*Parallel High-Radix Nonrestoring Division*", IEEE Transactions on Computers, October 1993, vol. 42, pag. 1234-1246.

[SELL68] F.F.Sellers, M.Y.Hsiao, L.W.Bearnson, "*Error Detecting Logic for Digital Computers*", McGraw-Hill, 1968.

[SERL84] O.Serlin, "*Fault-Tolerant Systems in Commercial Applications*", IEEE Computer, August 1984, pag. 19-30.

[SIEV82] D.P.Siewiorek, R.S.Swarz, "*The Theory and Practice of Reliable System Design*", Digital Press, 1982.

[SIEV90] D.Siewiorek, "*Fault Tolerance in Commercial Computers*", IEEE Computer, July 1990, pag. 26-37.

[SIEV91] D.P.Siewiorek, "*Architecture of Fault-Tolerant Computers: An Historical Perspective*", PROCEEDINGS OF THE IEEE, December 1991, vol. 79, pag. 1710-1734.

[SING90] A.Singh, S.Murugesan, "*Fault-Tolerant Systems*", IEEE Computer, July 1990, pag. 15-17

[SINH89] B.P.Sinha, P.K.Srimani, "*Fast Parallel Algorithms for Binary Multiplication and Their Implementation on Systolic Architectures*", IEEE Transactions on Computers, March 1989, vol. 38, pag. 424-431.

[TAKA87] N.Takagi, S.Yajima, "*On-Line Error-Detectable High-Speed Multiplier Using Redundant Binary Representation and Three-Rail Logic*", IEEE Transactions on Computers, November 1987, vol.C-36, pag. 1310-1317.

[TARC89] C.Târcolea, A.Filipoiu, S.Bontaş, "*Tehnici actuale în teoria fiabilităţii*", Editura Ştiinţifică şi Enciclopedică, Bucureşti, 1989.

[TEXA92] Texas Instruments, "*Advanced Digital Logic Guide*", Texas Instruments Inc., 1992.

[THOM91] D.Thomas, P.Moorby, "*The Verilog hardware description language*", Kluwer academic publishers, 1991.

[TOY78] W.N.Toy, "*Fault-Tolerant Design of Local ESS Processors*", PROCEEDINGS OF THE IEEE, October 1978, pag. 1126-1145.

[TOY88] W.N.Toy, "*Fault-Tolerant Computing*", 1988.

[VASS89] S.Vassiliadis, E.M.Schwarz, D.J.Hanrahan, "*A General Proof for Overlapped Multiple-Bit Scanning Multiplications*", IEEE Transactions on Computers, February 1989, vol. 38, pag. 172-183.

[VLAD82a] M.Vlăduţiu, "*Tehnologie de ramură şi fiabilitate*", Curs, Litografia I.P.T.V.T. Timişoara, 1982.

[VLAD82b] M.Vlăduţiu, "*Contribuţii la creşterea coeficientului de disponibilitate al echipamentelor automate de prelucrare a informaţiei prin testare neconvenţionala*", teză de doctorat, Bucureşti, 1982.

[VLAD89] M.Vlăduţiu, M.Crişan, "*Tehnica testării echipamentelor automate de prelucrare a datelor*", Editura Facla, Timişoara, 1989.

[VLAD94a] M.Vlăduţiu, N.S.Petrakis, "*About New Signature Generation Techniques for Binary Sequences*", International Conference on Technical Informatics, PROCEEDINGS Vol. 5, Timişoara 1994, pag. 11-15.

[VLAD94b] M.Vlăduţiu, N.S.Petrakis, "*Adapted Combinational Array for Exact Binary Division with Signed Operands*", International Conference on Technical Informatics, PROCEEDINGS Vol. 5, Timişoara 1994, pag. 1-10.

[VLAD95] M.Vlăduţiu, D.Todinca, N.S.Petrakis, I.Brînzaş, "*The Principle of a Fuzzy Processor with Parity Check*", Buletinul Ştiinţific U.T. Timişoara, 1994, Tom 39 (53), Seria Automatizări şi Calculatoare (în curs de publicare).

[WAGN87] K.D.Wagner, C.K.Chin, E.J.McCluskey, "*Pseudorandom Testing*", IEEE Transactions on Computers, March 1987, vol.C-36, pag. 332-343.

[WANG94] C.L.Wang, "*Bit-Level Systolic Array for Fast Exponentiation in* GF(2^m)", IEEE Transactions on Computers, July 1994, vol. 43, pag. 838-841.

[WENS78] J.H.Wensley, L.Lamport, J.Goldberg, M.W.Green, K.N.Levitt, "*SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control*", PROCEEDINGS OF THE IEEE, October 1978, pag. 1240-1255.

[WILK92] K.Wilken, J.P.Shen, "*Concurrent error detection using signature monitoring and encryption*", 1992.

[YARM90] V.N. Yarmolik, "*Fault Diagnosis of Digital Circuits*", New York: Wiley, 1990.

[ZACC87] R.J.Zaccone, *J.L.Barlow*, "*Eliminating the Normalization Problem in Digit On-Line Aritmetic*", IEEE Transactions on Computers, January 1987, vol.C-36, pag. 36-46.

[ZURA87] J.H.P.Zurawski, *J.B.Gosling*, "*Design of a High-Speed Square Root Multiply and Devide Unit*", IEEE Transactions on Computers, January 1987, vol.C-36, pag. 13-23